Fixed Arithmetic for Computers & ALU Design

Chapter- 4

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

And underlying all these questions is a mystery: How does hardware really multiply or divide numbers?

3.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 2, base 2 is best for computers. To avoid confusion we subscript decimal numbers with *ten* and binary numbers with *two*.

In any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^i$$

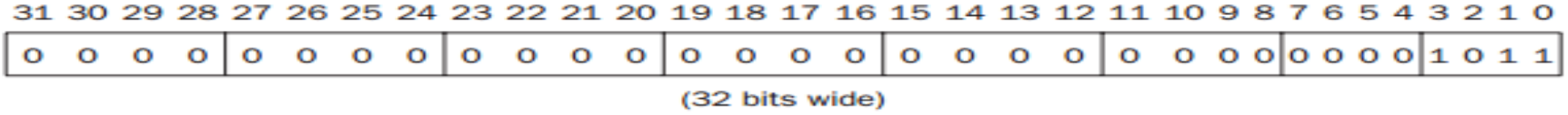
where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

$$1011_{two}$$

represents

$$\begin{aligned} &(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten} \\ &= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten} \\ &= 8 + 0 + 2 + 1_{ten} \\ &= 11_{ten} \end{aligned}$$

Hence the bits are numbered 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:



Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 31).

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude was soon abandoned.

- →To overcome this use 2's complement

If MSB=0, it is positive number

If MSB=1, then it is negative number

0000=0	1000= -8 (-1*2^3 + 0*2^2+0*2^1+0*2^0)
0001=1	1001= -7
0010=2	1010= -6
0011=3	1011= -5
0100=4	1100= -4
0101=5	1101= -3
0110=6	1110= -2
0111=7	1111= -1

- Rules to negate a two's complement binary number:

1. Simply invert 0 to 1 and 1 to 0
2. Then add 1 to this result

Example:

Negate 2_{ten}, and then check the result by negating -2_{ten}.

$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
Negating this number by inverting the bits and adding one,

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

Self : (-100) ten to (100) ten

- Example:

  Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

- Answer:

  The 16-bit binary version of the number 2 is

  $$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

  It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

  $$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

  Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

  $$0000\ 0000\ 0000\ 0010_{two}$$

  becomes

  $$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{two} \\ +\qquad\qquad\qquad\quad 1_{two} \\ \hline = 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

  Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

  $$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$

- **Addition and subtraction:**

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

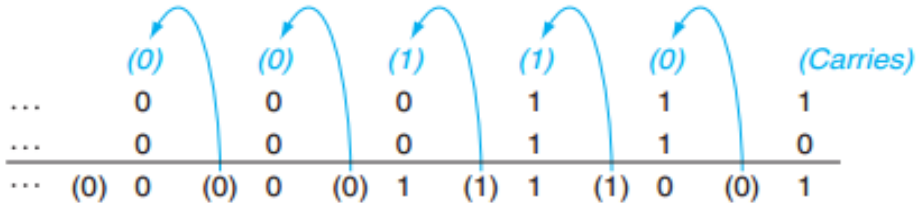


FIGURE 3.2 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

4.3 Addition and Subtraction

Overflow Conditions MIPS=3bits

Operation	Operand A	Operand B	Result
A + B	≥ 0	≥ 0	< 0
	011 (3) + 011 (3) = 110 (-2)		
A + B	< 0	< 0	≥ 0
	100 (-4) + 101 (-3) = 1 001 (1)		
A – B	≥ 0	< 0	< 0
	011 (3) – 100 (-4) = 011 - 100 = 1111 (-1)		
A – B	< 0	≥ 0	≥ 0
	100 (-4) – 001 (1) = 100 -001 = 011 (3)		

4.4 Logical Operations

- Shift left logical (sll)

  sll $t2, $s0, 8 # reg $t2 = reg $s0 « 8 bits

$t2 = 0000 0000 0000 0000 0000 0000 0000 1101

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

- Shift right logical (srl)
- AND (and)
- OR (or)

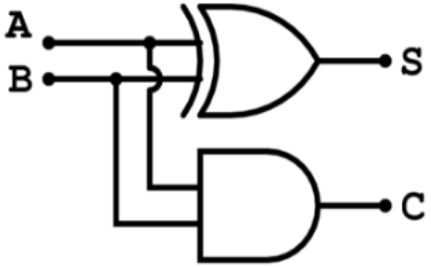
Half Adder

HA block symbol

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

HA Truth Table

Half adder



HA Logic Circuit

Sum $S = A \oplus B$

Carry $C = AB$

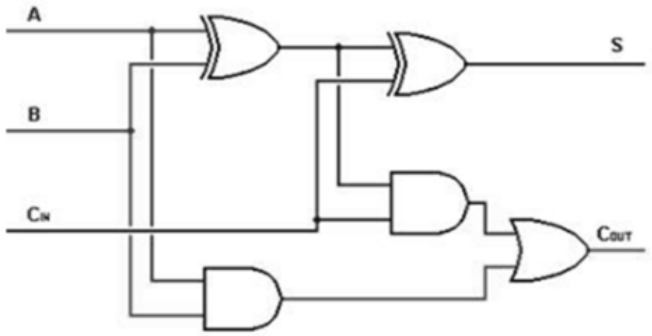
HA Formula

Full Adder

FA block symbol

Inputs			Outputs	
A	B	Carry in C_{in}	Sum S	Carry out
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

FA Truth Table

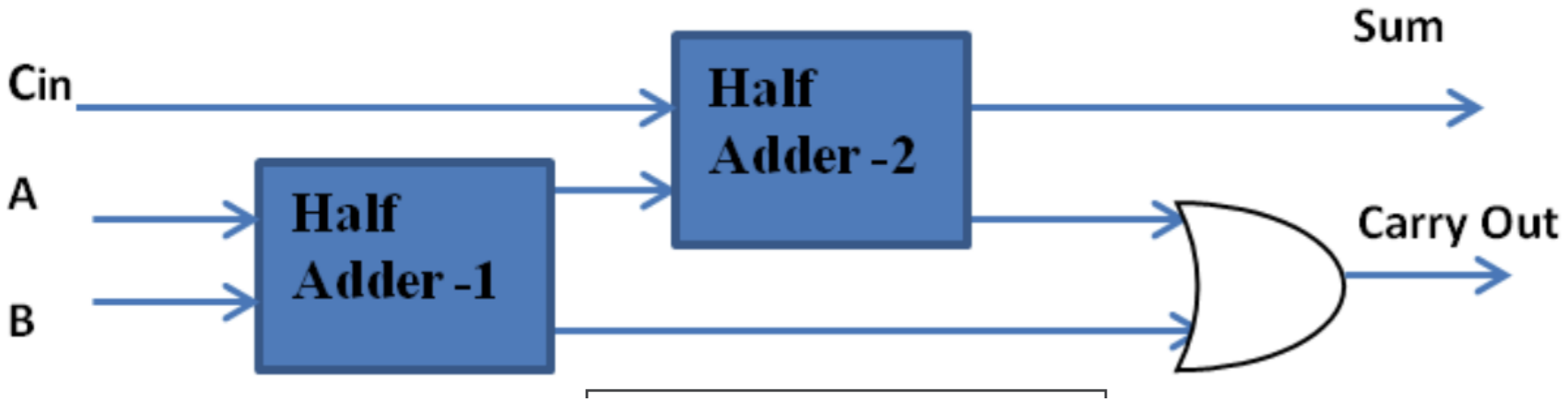


FA Logic Circuit

$$S = A \oplus B \oplus C_i$$
$$C_{out} = AB + C_i(A + B)$$

FA Formula

Full Adder

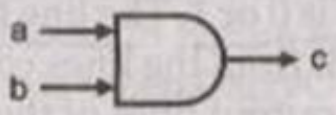


Error Detection and Status Flags

4.5 Constructing an Arithmetic Logic Unit

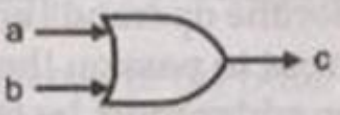
- Four hardware building blocks

1. AND gate ($c = a \cdot b$)



a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ($c = a + b$)

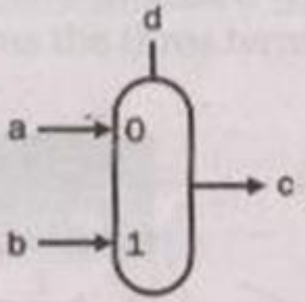


a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)

a	$c = \bar{a}$
0	1
1	0

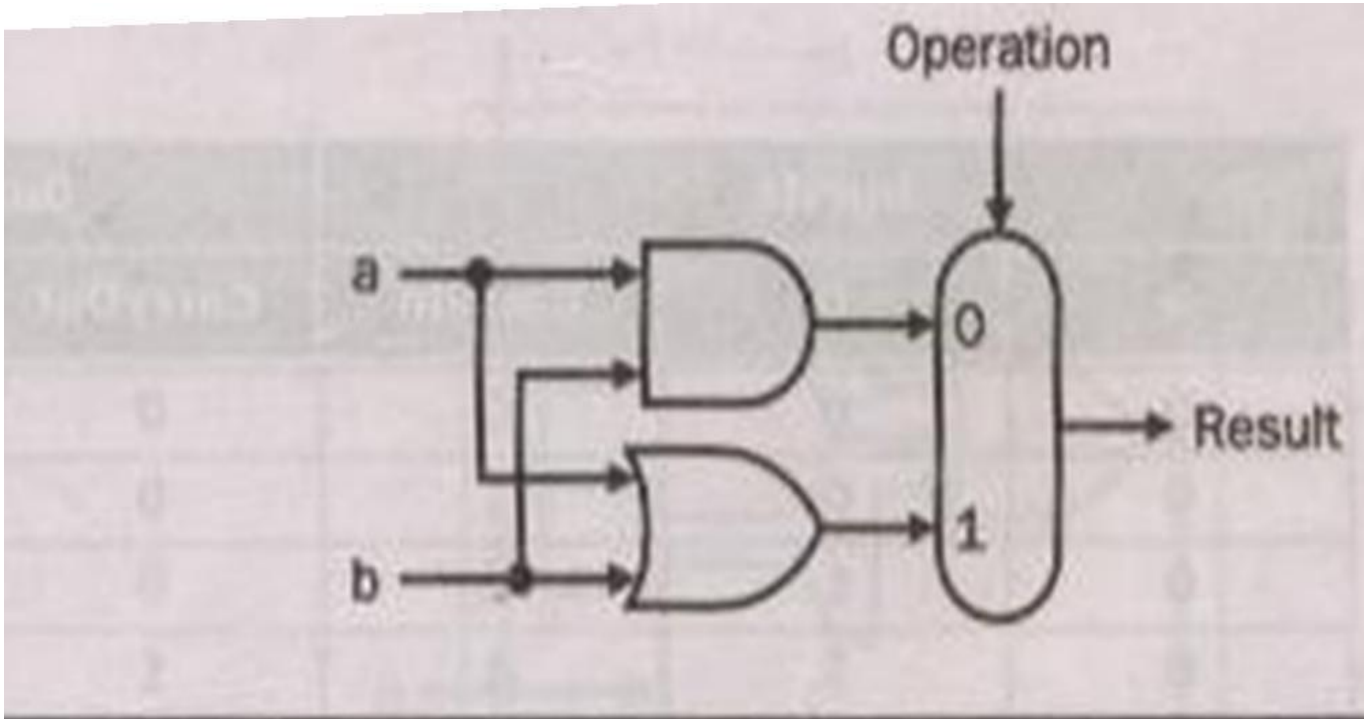
4. Multiplexor
(if d = = 0, c = a;
else c = b)



d	c
0	a
1	b

A 1-Bit ALU

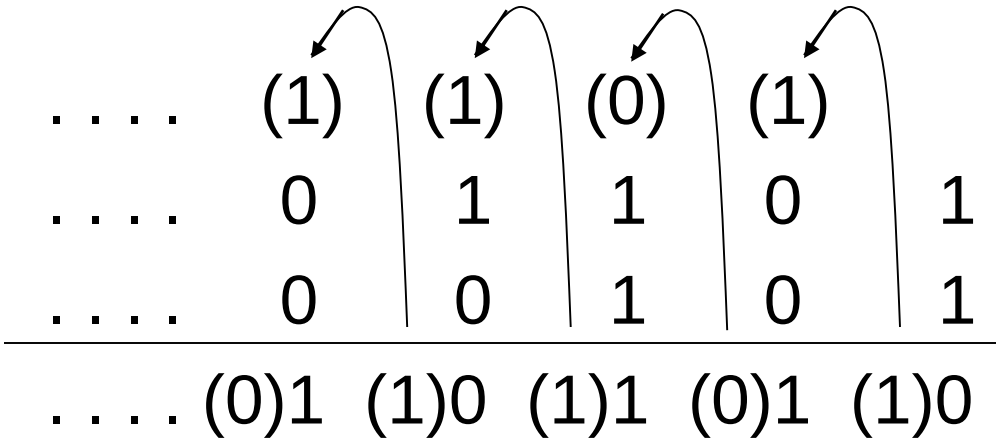
- Logical Unit

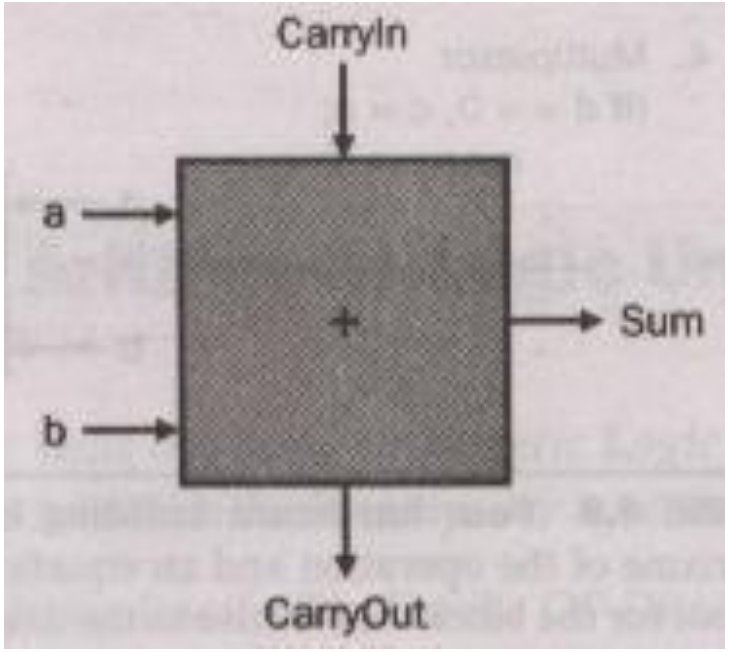


The 1-bit logical unit for AND and OR.

A 1-Bit ALU (Cont.)

- Addition (A 1 bit adder)





A 1-Bit ALU (Cont.)

- Addition

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

Input and output specification for a 1-bit adder.

A 1-Bit ALU (Cont.)

Inputs			Outp
a	b	CarryIn	CarryOut
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

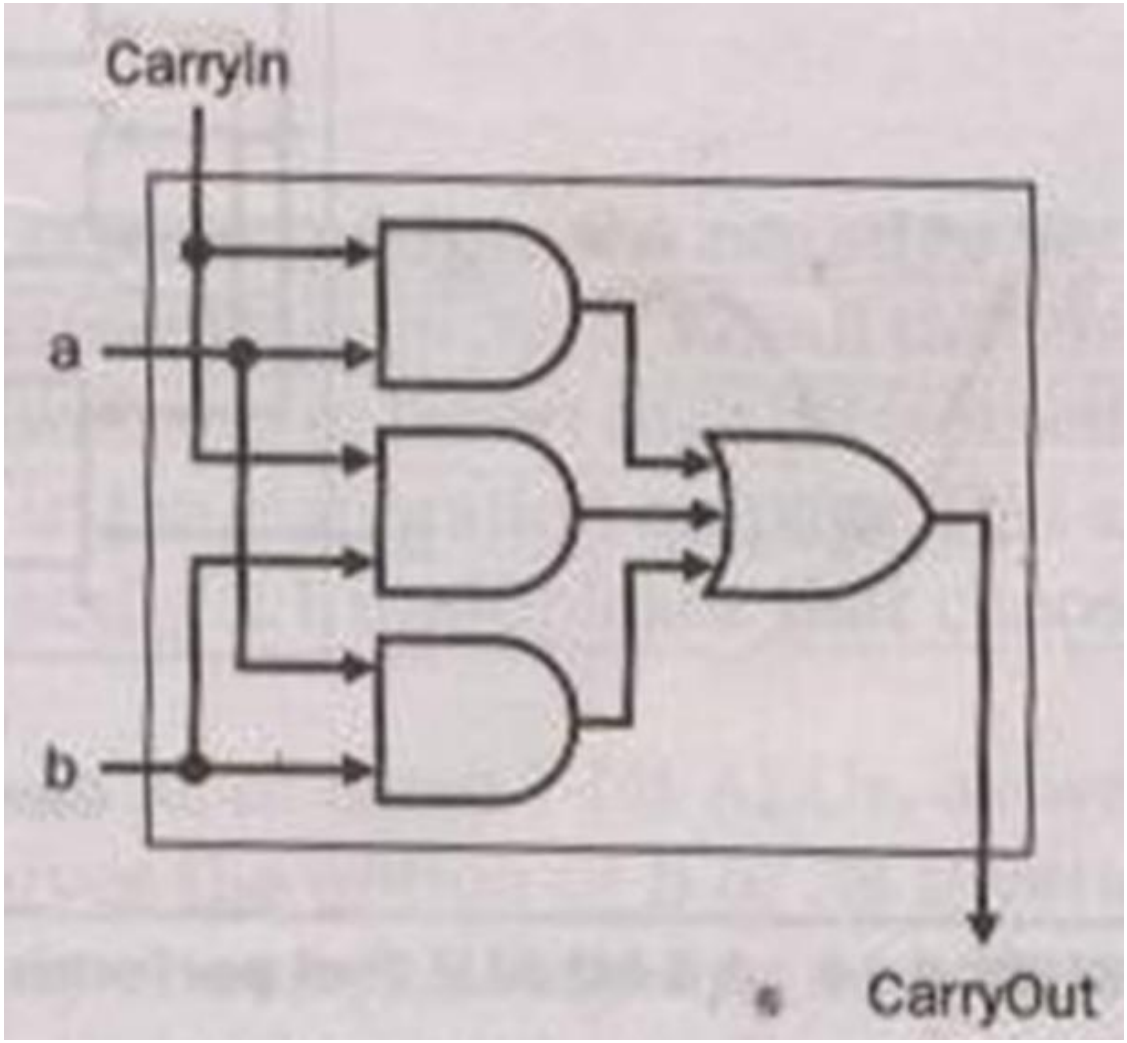
CarryOut

	a'b'	a'b	ab	ab'
c'	0	0	1	0
c	0	1	1	1

CarryOut = (a · CarryIn) + (b · CarryIn) + (a · b)

A 1-Bit ALU (Cont.)

CarryOut = (a · CarryIn) + (b · CarryIn) + (a · b)



A 1-Bit ALU (Cont.)

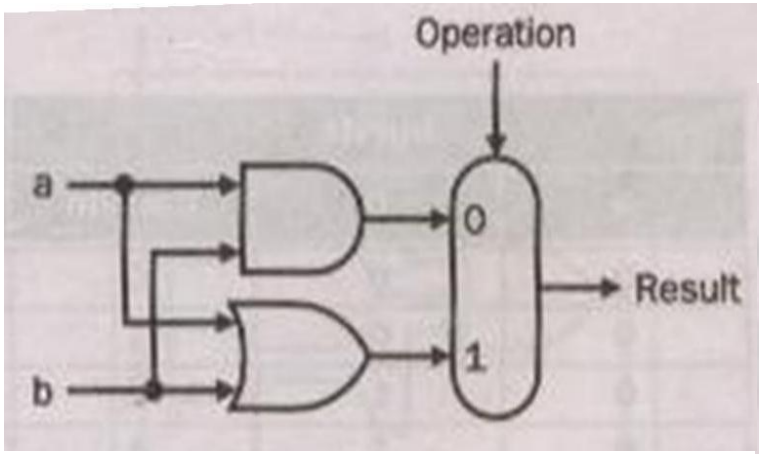
- Addition

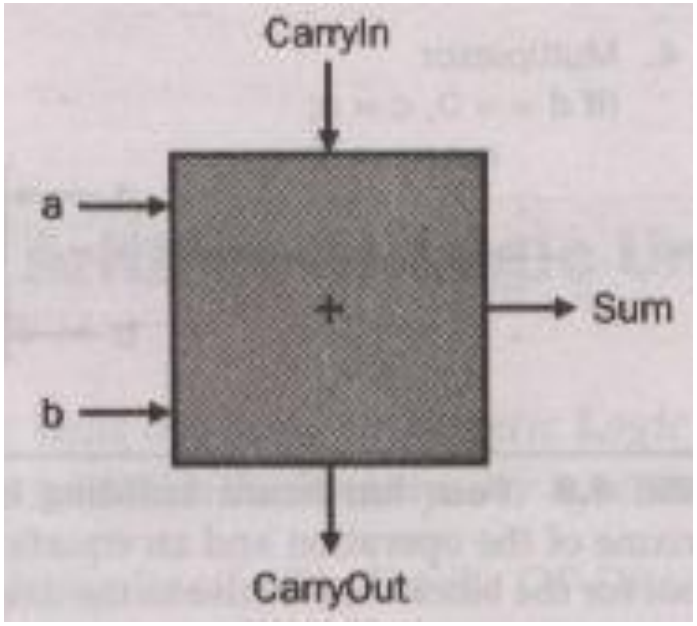
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

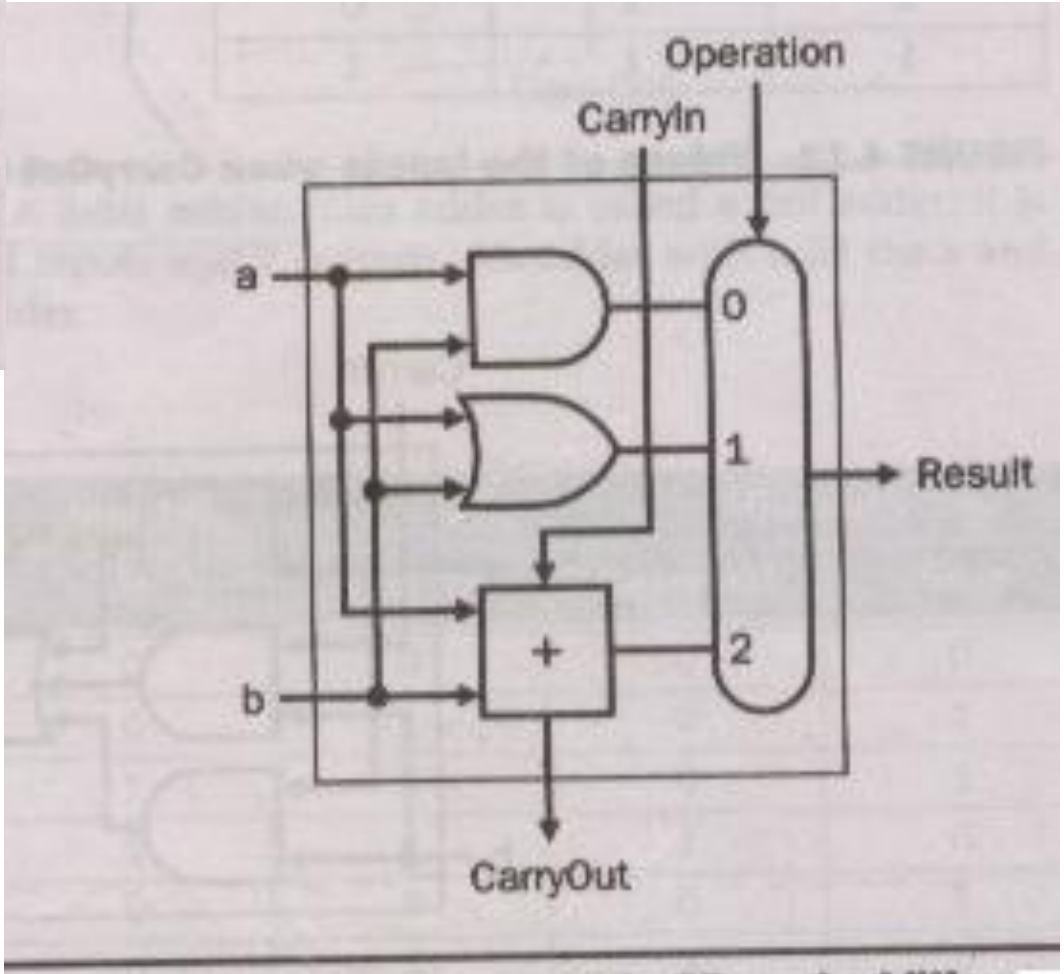
Input and output specification for a 1-bit adder.

Sum = (a · b' · C') + (a' · b · C') + (a' · b' · C) + (a · b · C)

A 1-Bit ALU (Cont.)



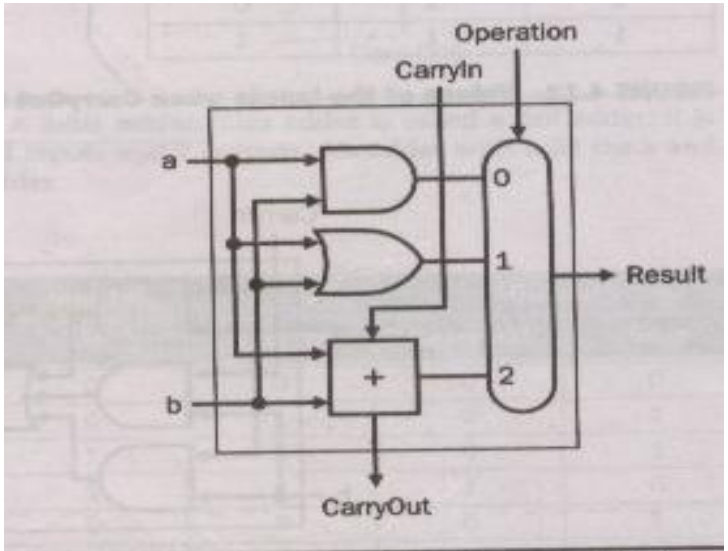




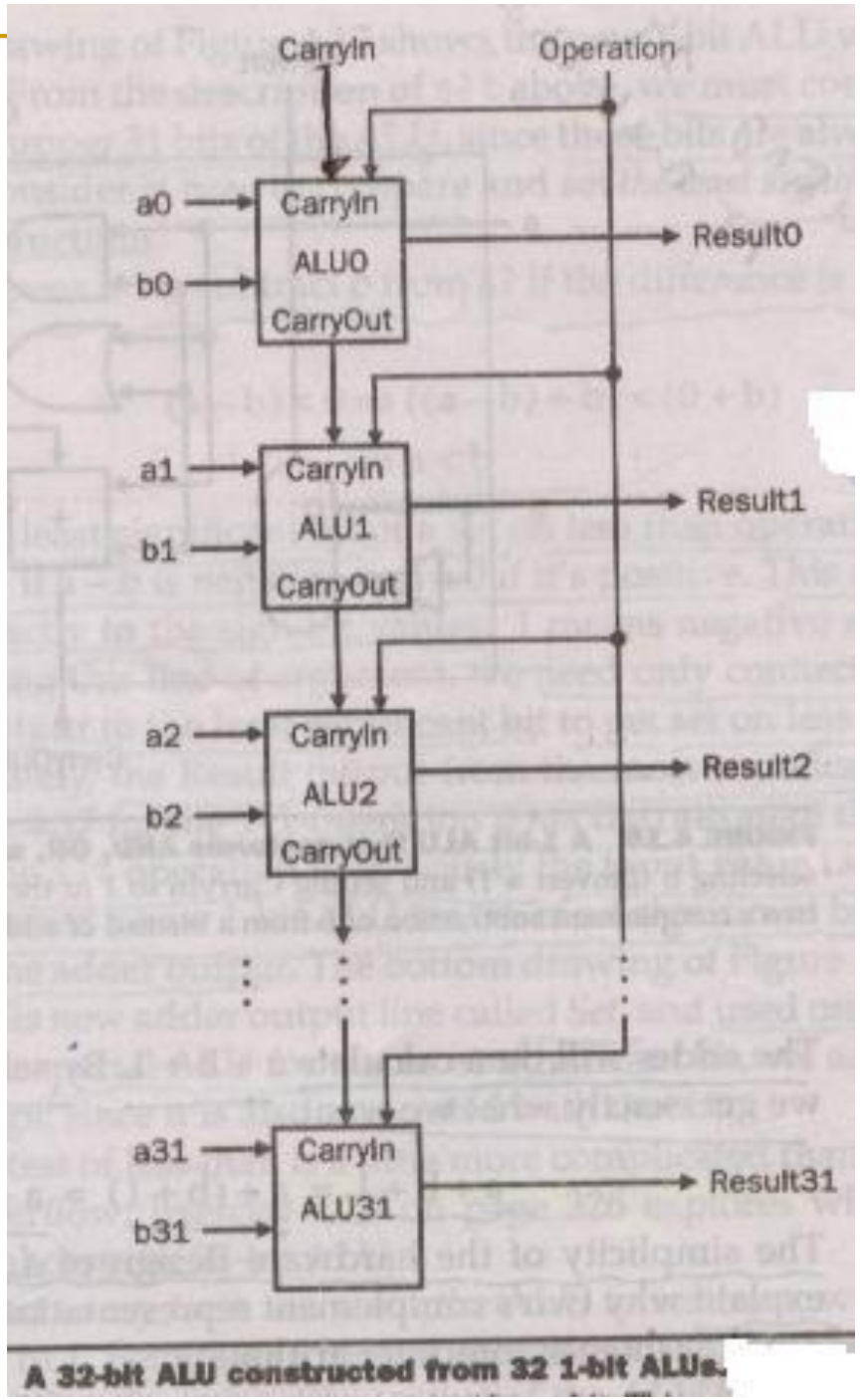
A 1-bit ALU that performs AND, OR, and addition

A 32-Bit ALU

Just a single stone can cause ripple to the shores of a quite lake, a single carry out of the least significant bit can ripple all the way through the adder, causing a carry out of the most significant bit. Hence, the adder created by directly linking the carries of 1-bit adders is called a **ripple carry adder.**



A 1-bit ALU that performs AND, OR, and addition

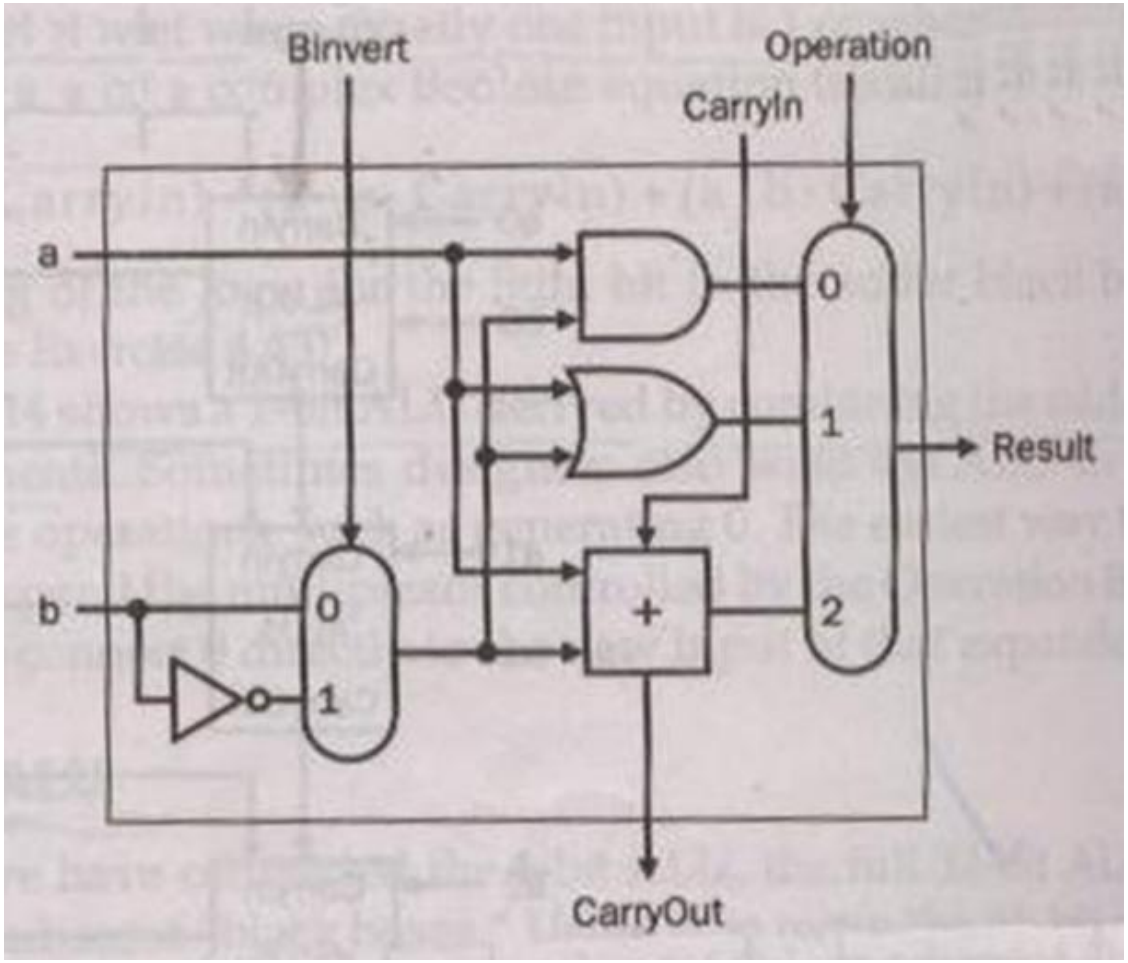


A 32-bit ALU constructed from 32 1-bit ALUs.

A 32-Bit ALU (Cont.)

- Subtraction

  a + b' + 1 = a + (b' + 1) = a + (-b) = a – b



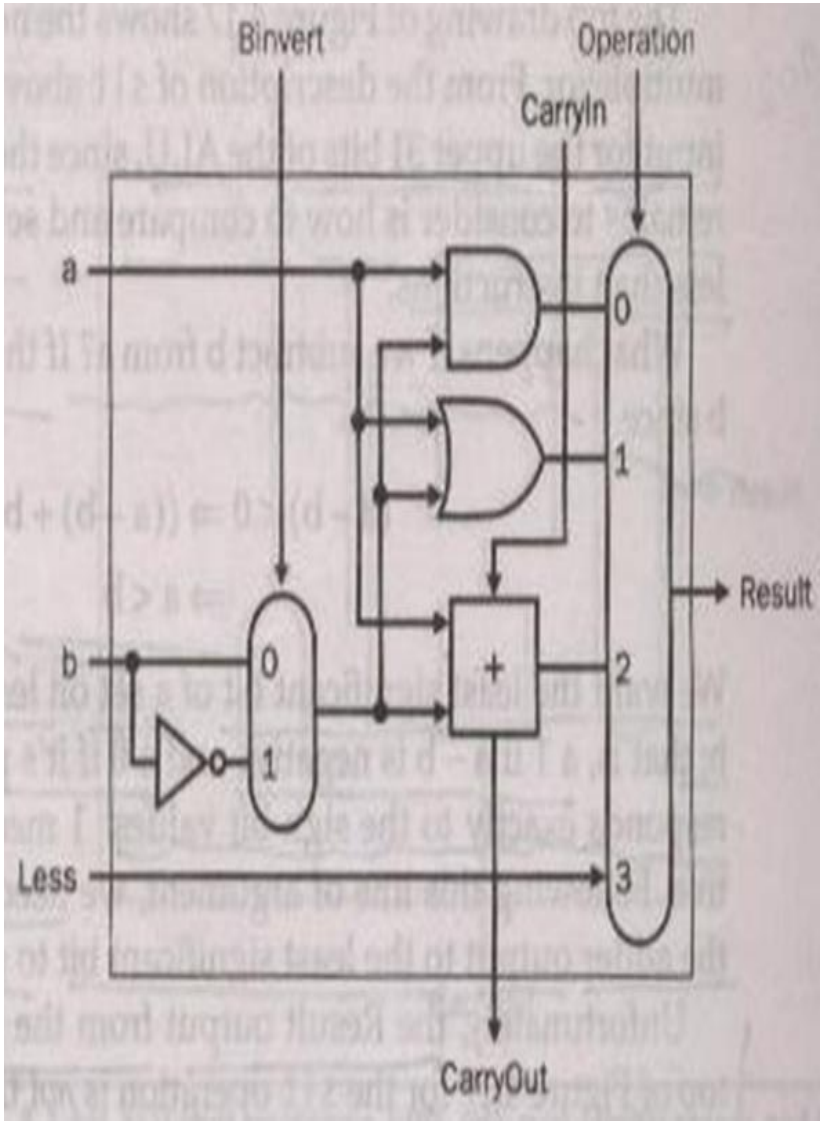
Tailoring the 32-Bit ALU to MIPS

- Set on less than (slt)

  (a – b) < 0

  a < b

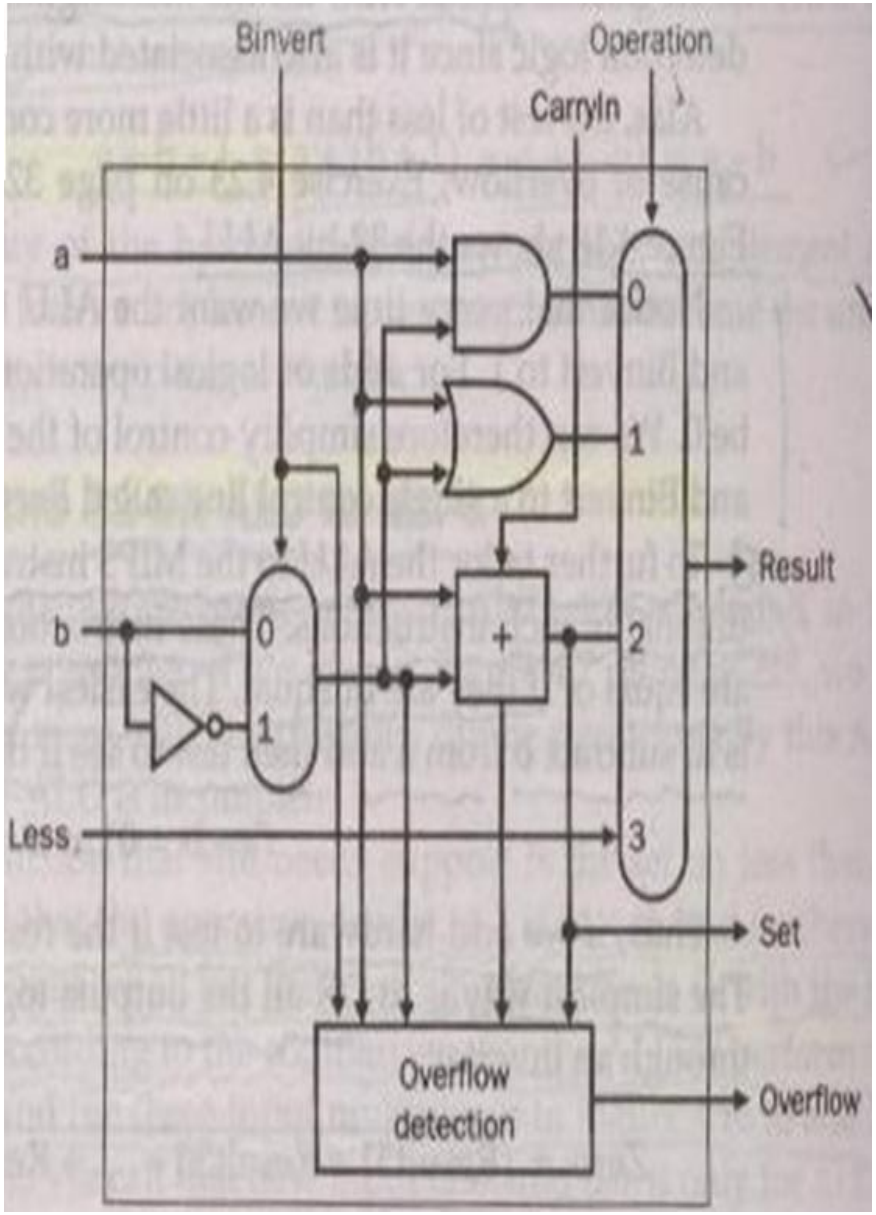
We want the least significant bit of a set on less than operation to be a 1 if a < b; that is, a 1 if a – b is negative and a 0 if it's positive. This desired result corresponds exactly to the sign bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.



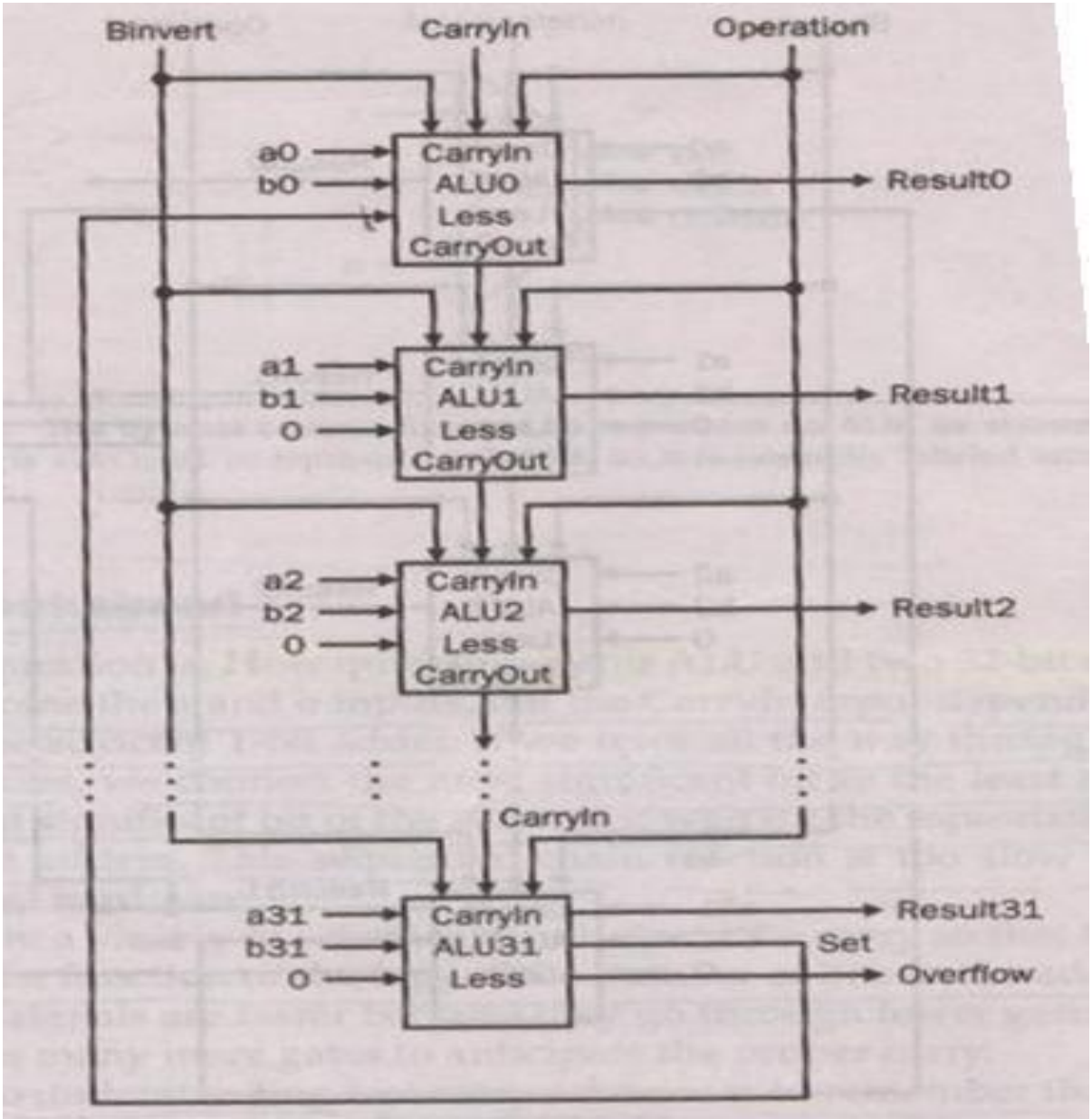
Tailoring the 32-Bit ALU to MIPS (Cont.)

Unfortunately, the Result output from the most significant ALU bit in the top of Figure B.5.10 for the `slt` operation is *not* the output of the adder; the ALU output for the `slt` operation is obviously the input value Less.

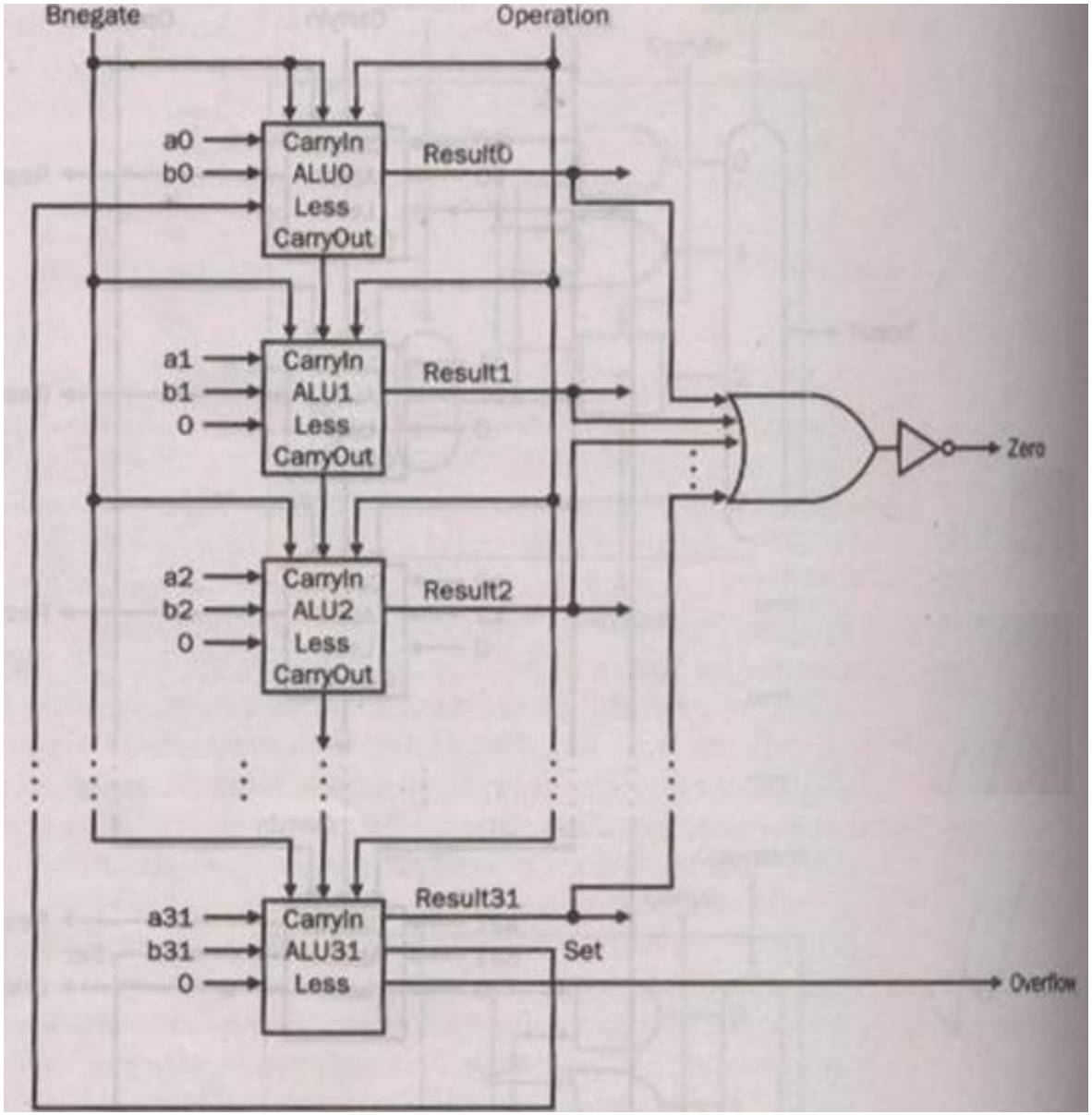
Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure B.5.10 shows the design, with this new adder output line called *Set*, and used only for `slt`. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.



Tailoring the 32-Bit ALU to MIPS (Cont.)



Tailoring the 32-Bit ALU to MIPS (Cont.)



Bnegate:

Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1. For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called *Bnegate*.

Zero detector:

To further tailor the ALU to the MIPS instruction set, we must support conditional branch instructions. These instructions branch either if two registers are equal or if they are unequal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0 since

$$(a - b = 0) \Rightarrow a = b$$

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

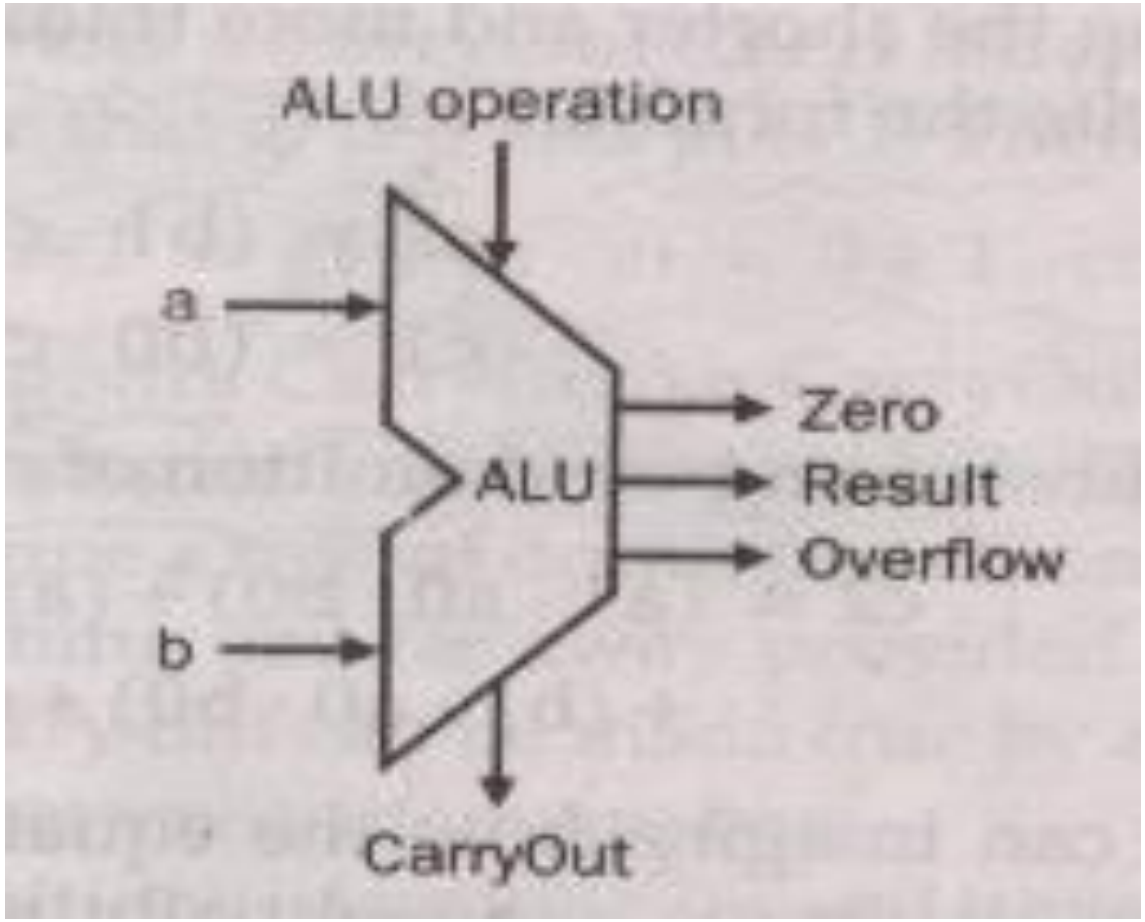
Tailoring the 32-Bit ALU to MIPS (Cont.)

ALU control lines	Function
000	and
001	or
010	add
110	subtract
111	set on less than

FIGURE 4.20 The values of the three ALU control lines Bnegate and Operation and the corresponding ALU operations.

Tailoring the 32-Bit ALU to MIPS (Cont.)

Universal symbol for a complete ALU



4.6 Multiplication

Multiplicand	1000
Multiplier	× 1001
	1000
	0000
	0000
	1000
Product	1001000

Length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long.

First Version of the Multiplication Algorithm and Hardware

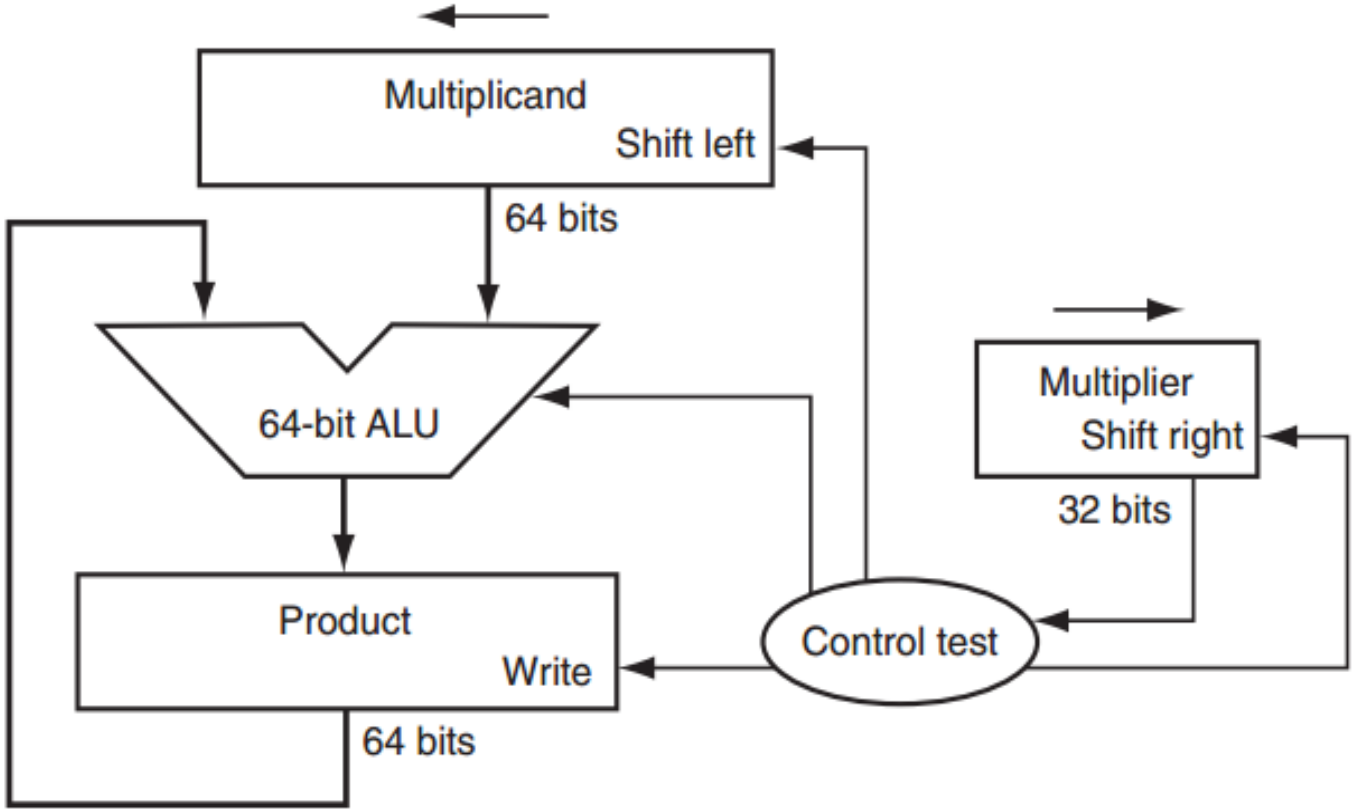
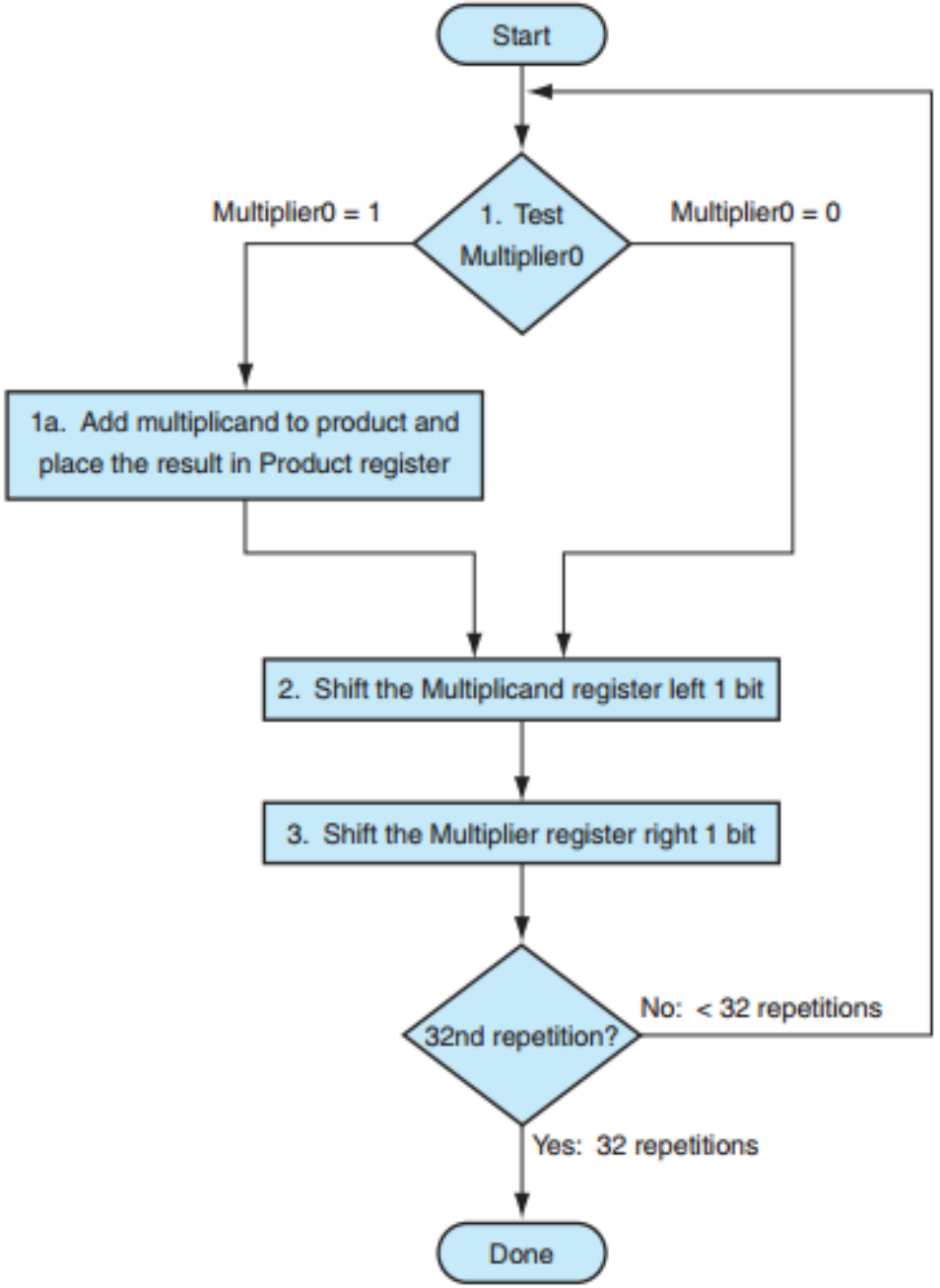


FIGURE 3.5 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

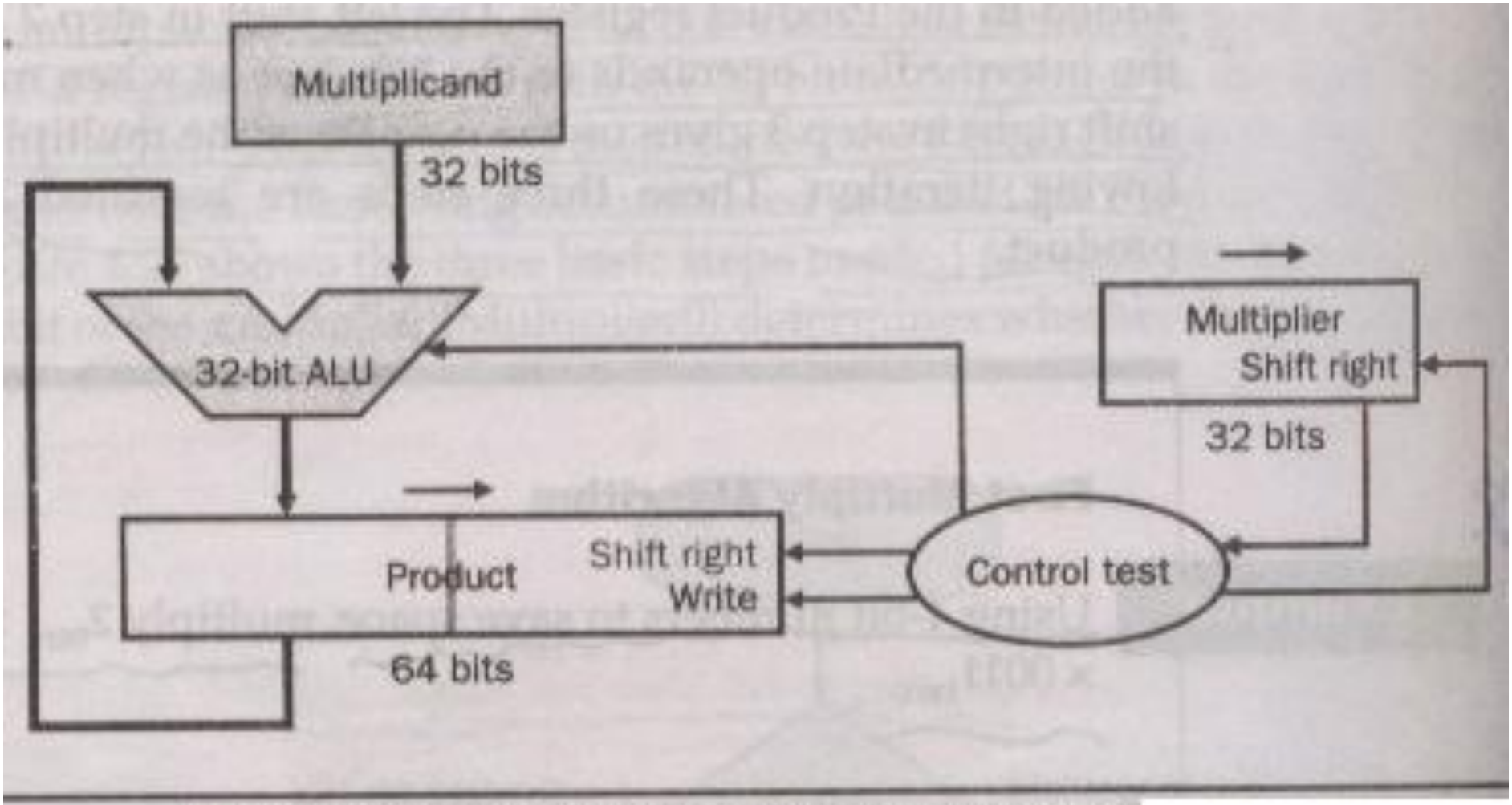
First Version of the Multiplication Algorithm and Hardware



Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$, or $0010_{two} \times 0011_{two}$.

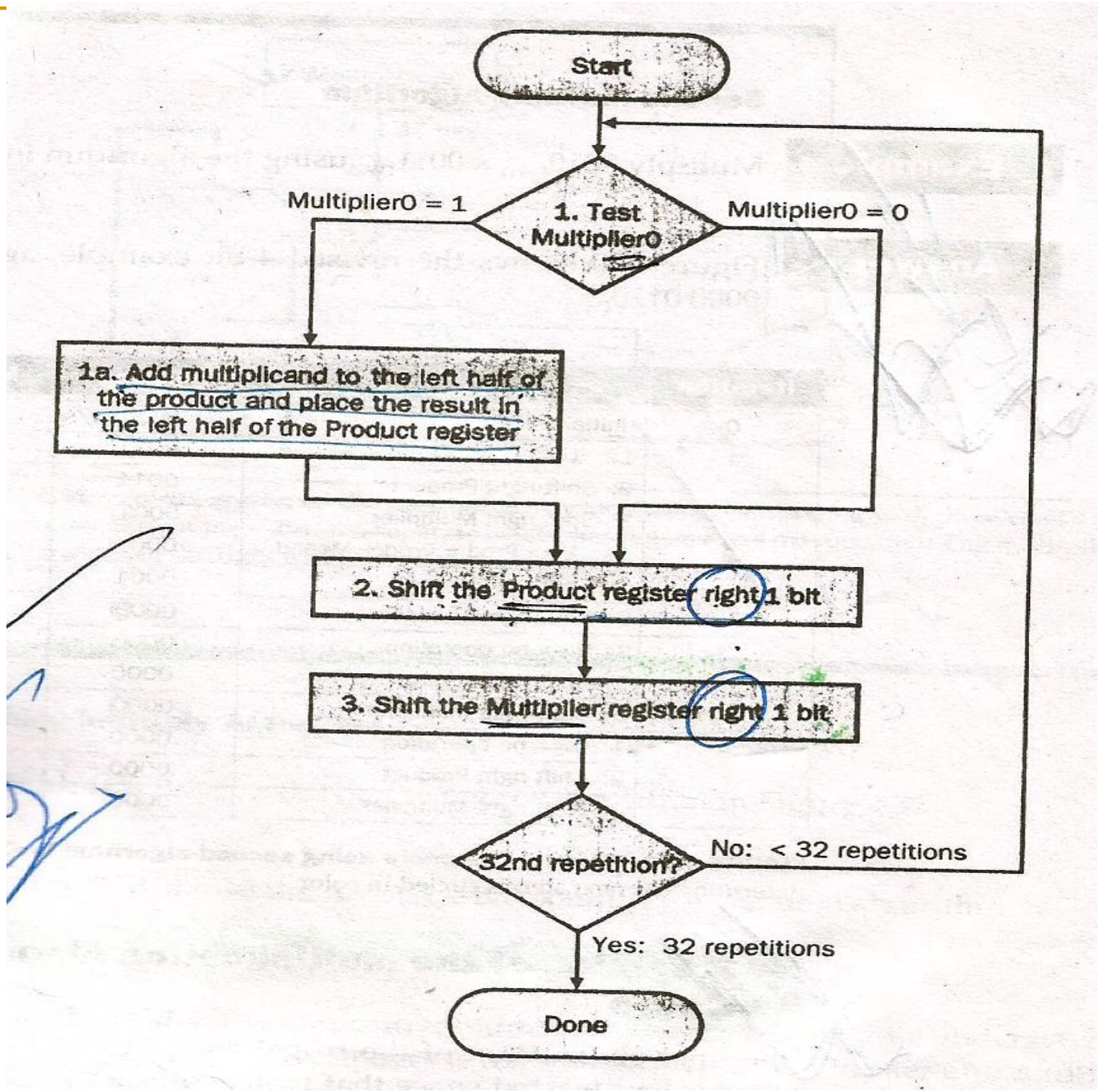
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	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

Second Version of the Multiplication Algorithm and Hardware



Second version of the multiplication hardware.

Second Version of the Multiplication Algorithm and Hardware



Second Multiply Algorithm

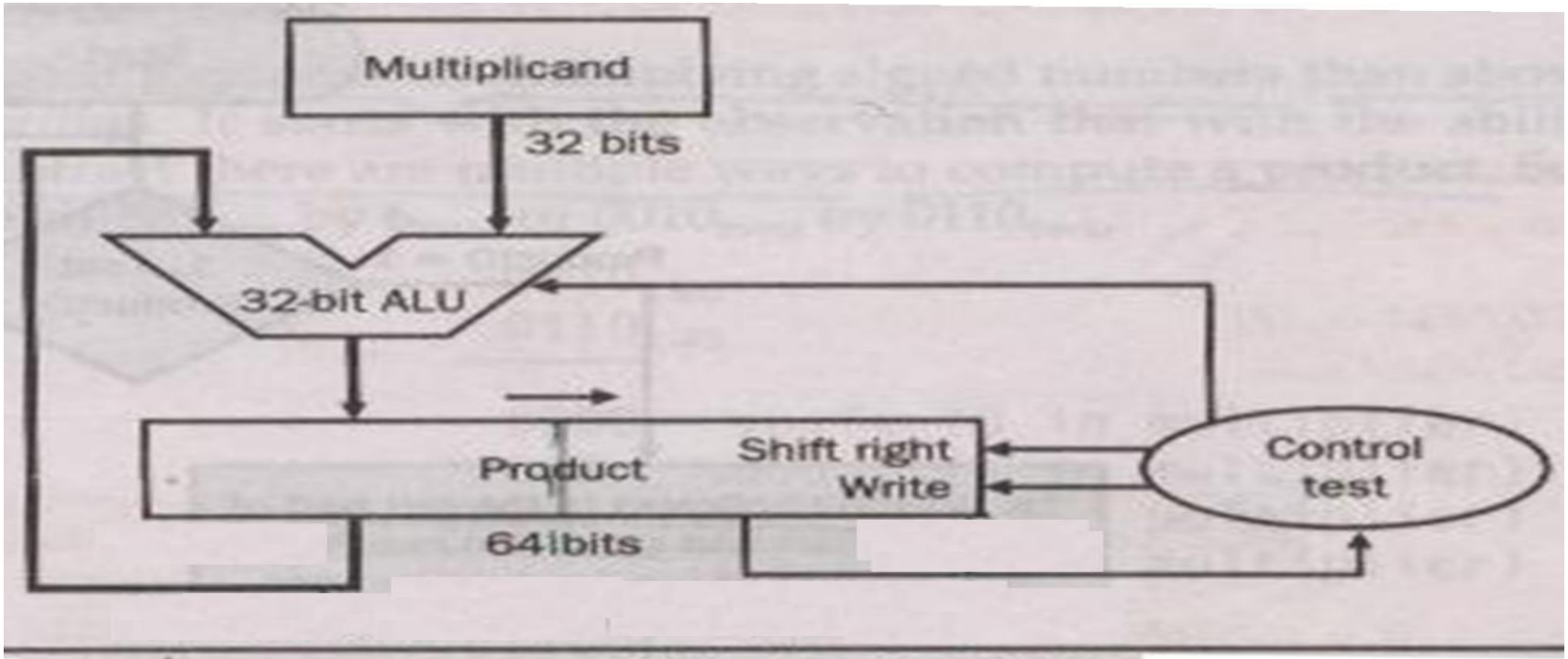
Multiply $0010_{two} \times 0011_{two}$ using the algorithm in Figure 4.29.

Figure 4.30 shows the revised 4-bit example, again giving a product of $0000\ 0110_{two}$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	0001	0010	0001 0000
2	1a: 1 => Prod = Prod + Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	0000	0010	0001 1000
3	1: 0 => no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	0000	0010	0000 1100
4	1: 0 => no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

FIGURE 4.30 Multiply example using second algorithm in Figure 4.29. The bit examined to determine the next step is circled in color.

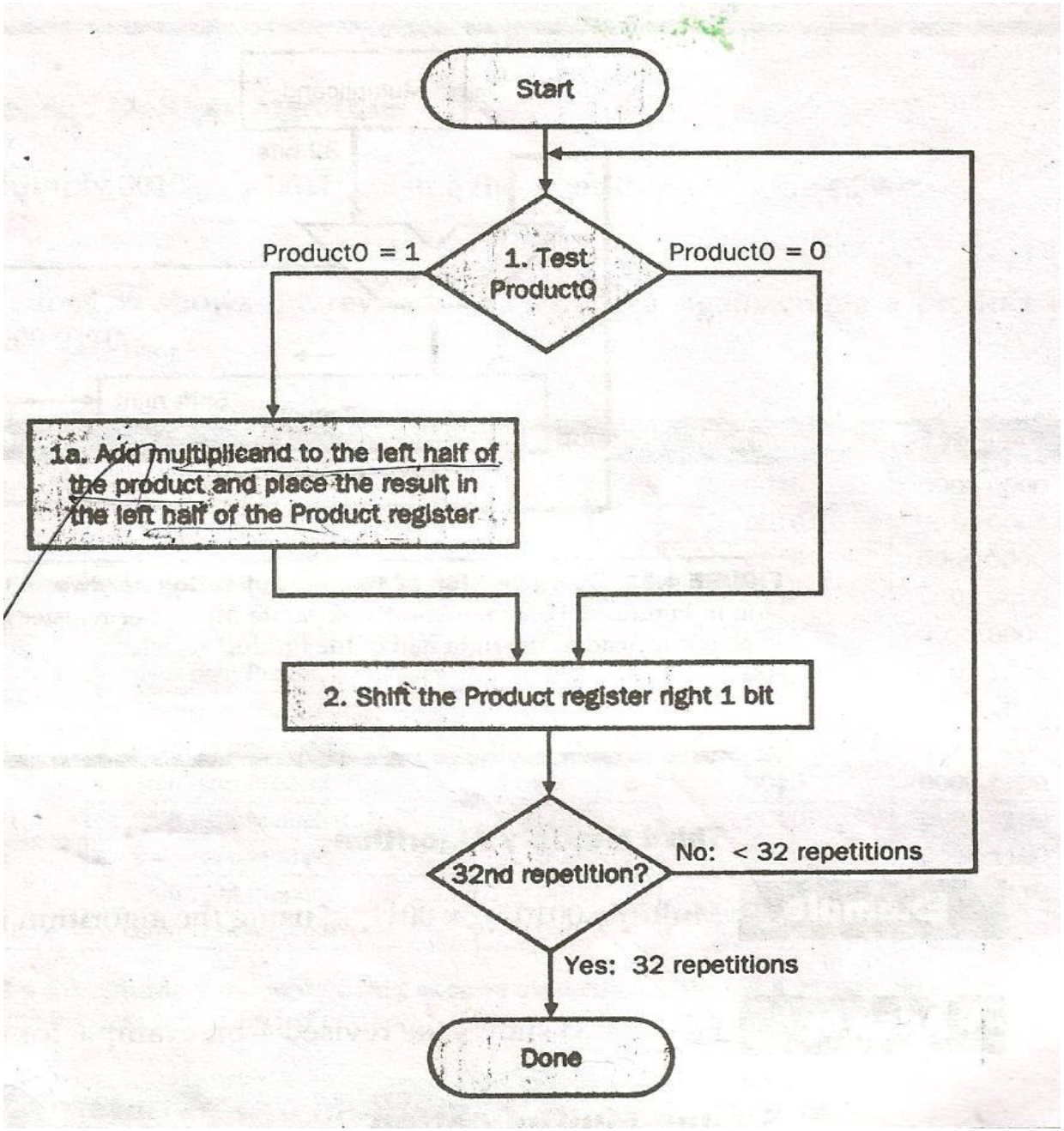
Final Version of the Multiplication Algorithm and Hardware



Third version of the multiplication hardware.

The multiplier is placed in the right half of the product register to remove wasted space

Final Version of the Multiplication Algorithm and Hardware

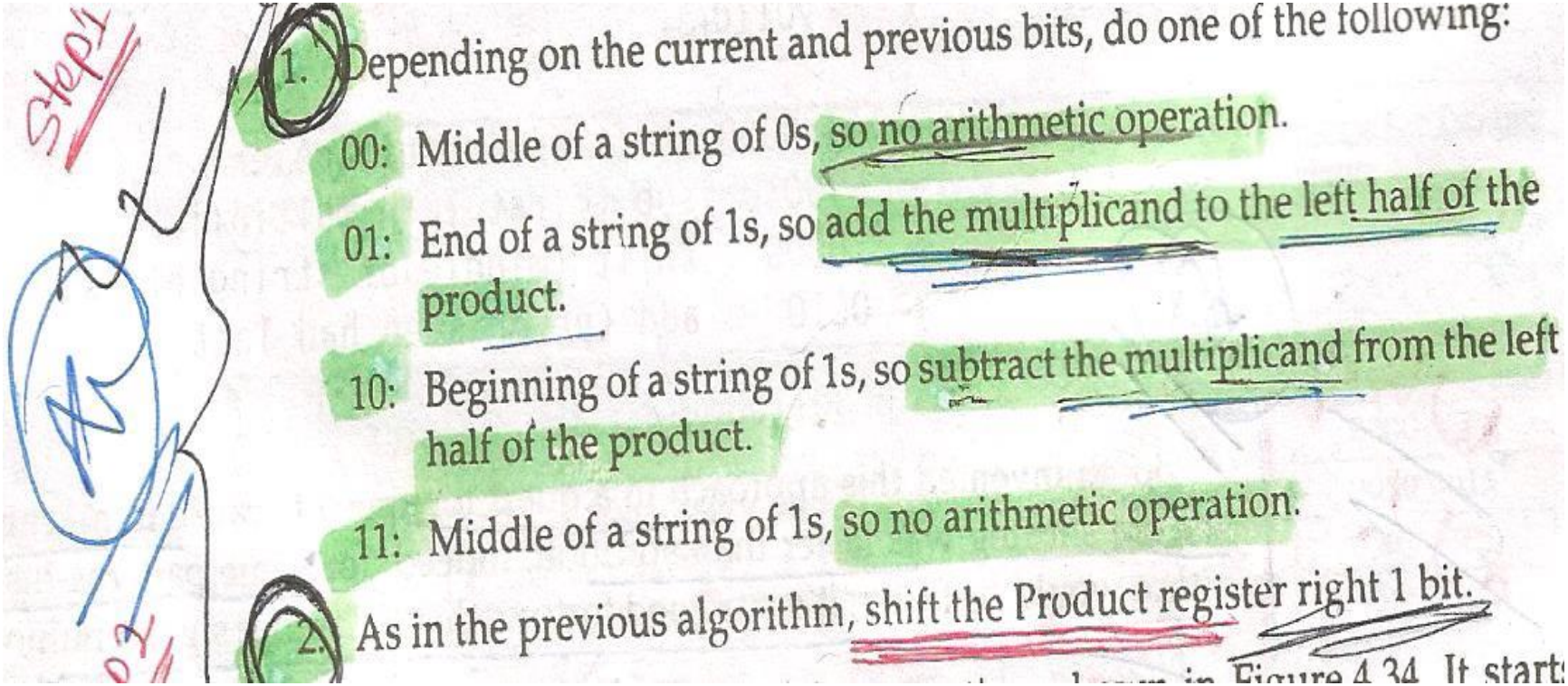


Final Version of the Multiplication Algorithm and Hardware

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

Multiply example using third algorithm in Figure 4.32.

Booth's algorithm:



1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

Figure 4.34 It starts

Itera-tion	Multi-plicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod – Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

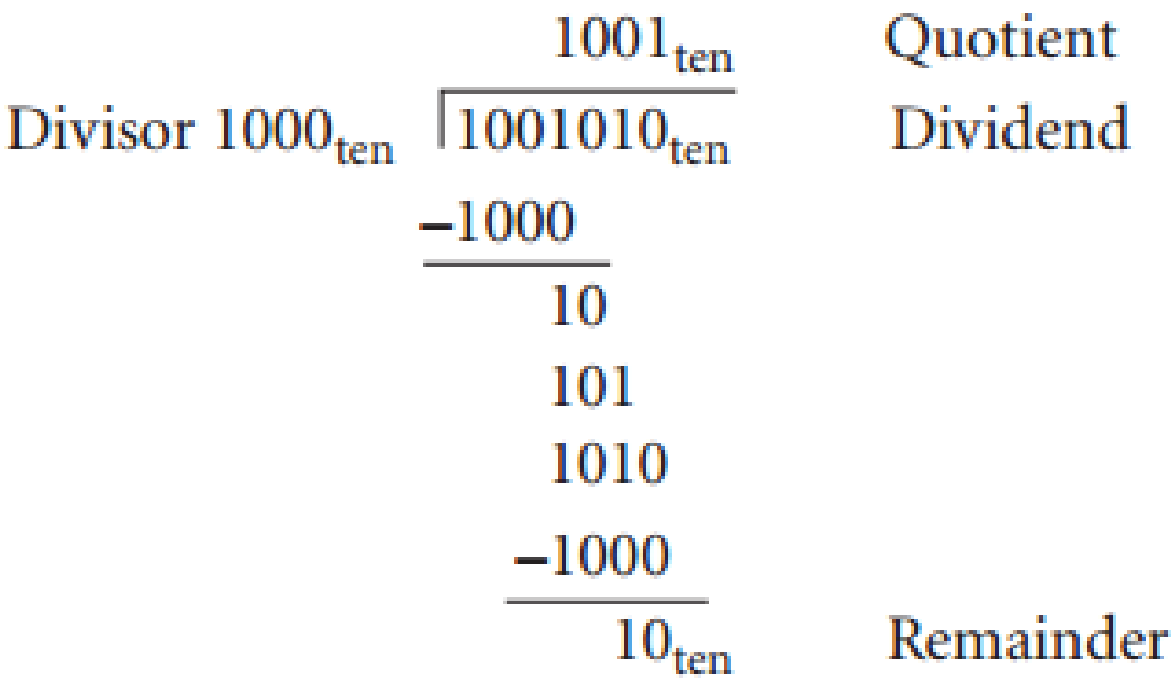
FIGURE 4.34 Comparing algorithm in Figure 4.32 and Booth's algorithm for positive numbers. The bit(s) examined to determine the next step is circled in color.

2*-3=-6

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

FIGURE 4.35 Booth's algorithm with negative multiplier example. The bits examined to determine the next step are circled in color.

Division


1001ten
Quotient
Divisor 1000ten
1001010ten
Dividend
–1000
10
101
1010
–1000
10ten
Remainder

First Version of the Division Algorithm and Hardware

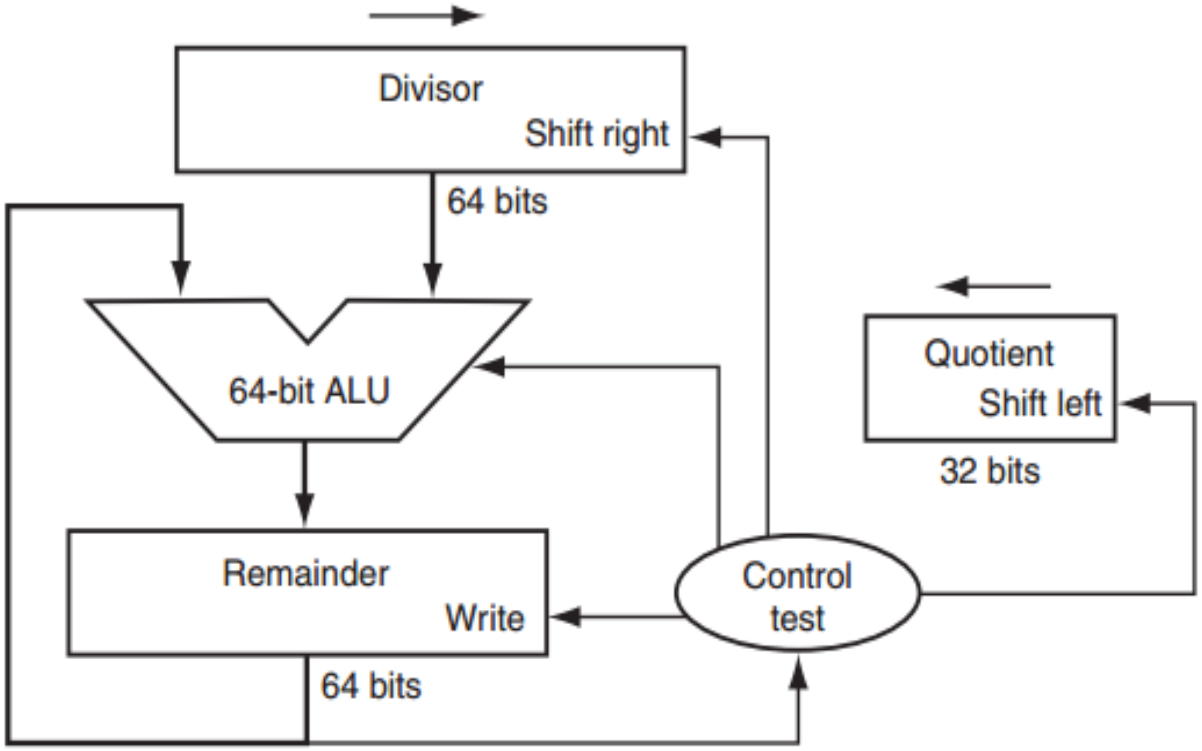
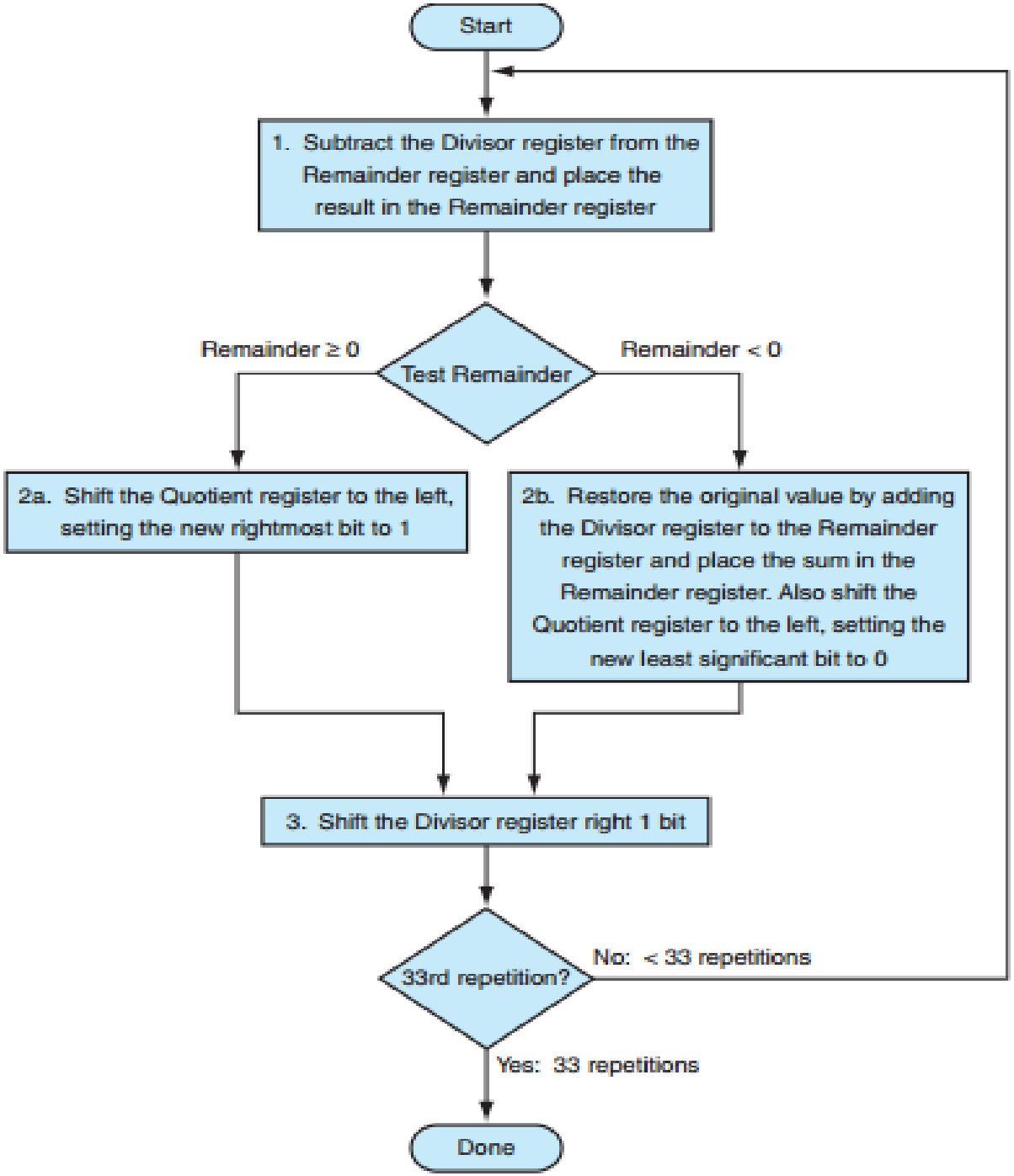


FIGURE 3.10 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

First Version of the Division Algorithm and Hardware

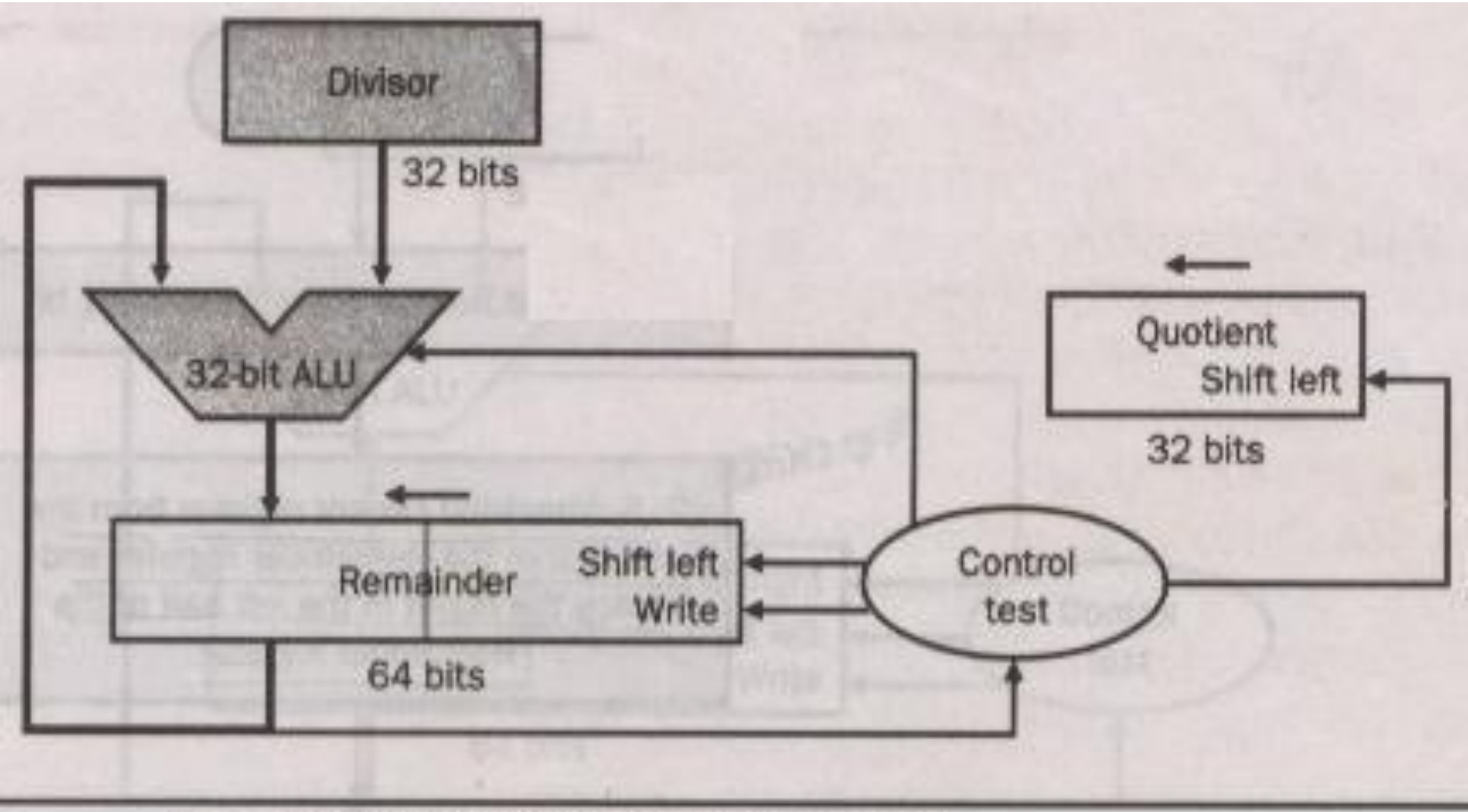


Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten}, or $0000\ 0111_{two}$ by 0010_{two}.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 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	(1)111 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	(1)111 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	(0)000 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	(0)000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.12 Division example using the algorithm in Figure 3.11. The bit examined to determine the next step is circled in color.

Second Version of the Division Algorithm and Hardware



Second version of the division hardware.

Final Version of the Division Algorithm and Hardware

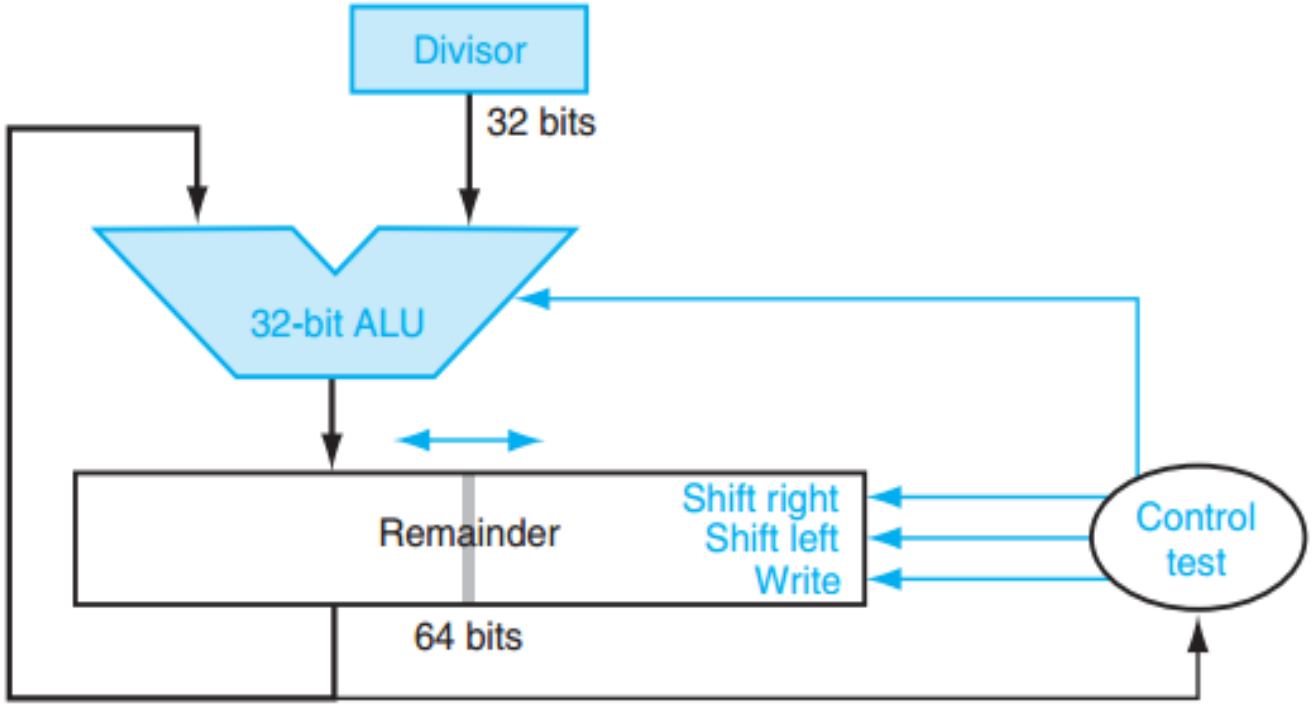


FIGURE 3.13 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.10, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register.

Final Version of the Division Algorithm and Hardware

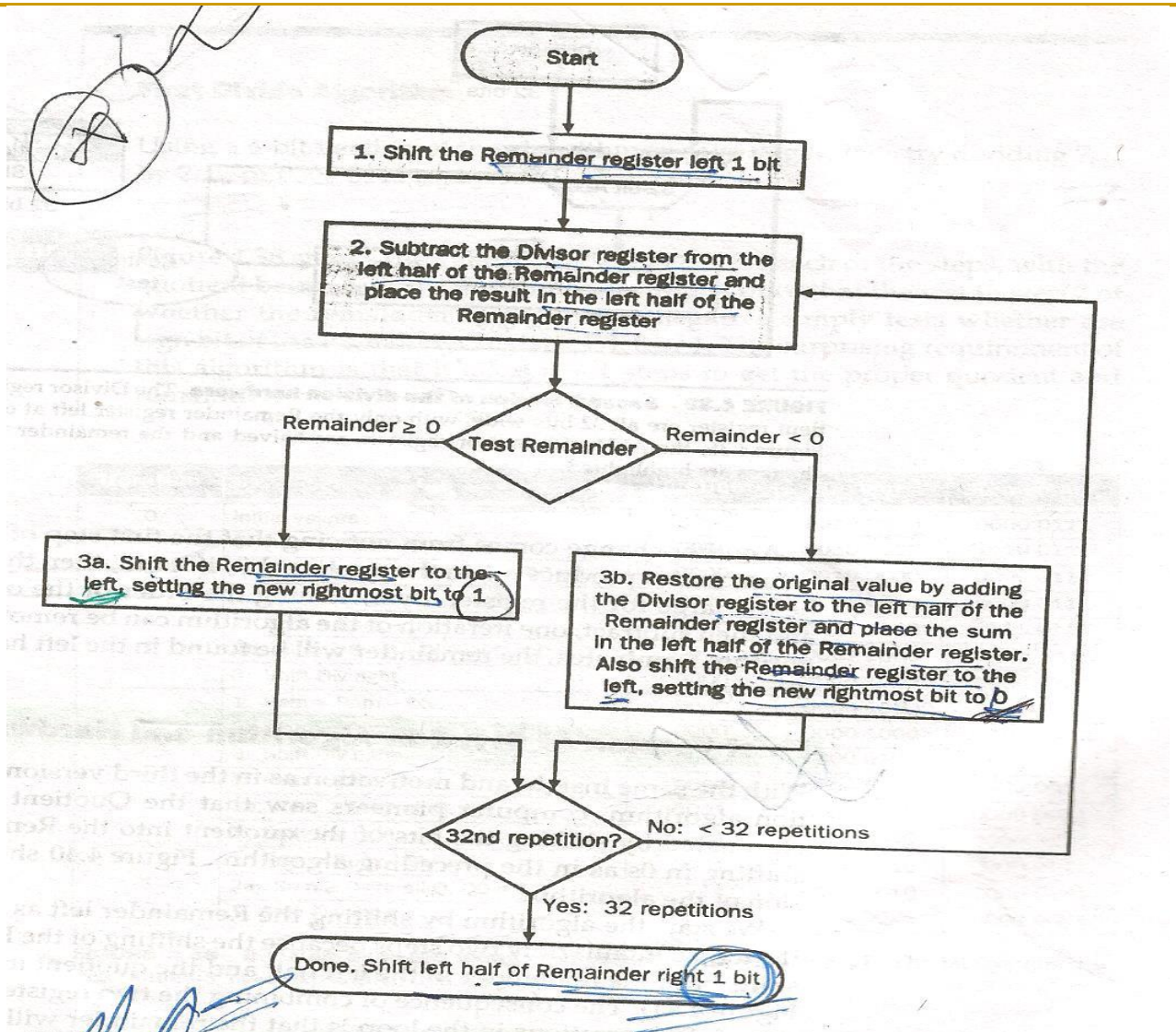


FIGURE 4.40 The third division algorithm has just two steps. The Remainder register shifts left.

Third Divide Algorithm

Example Use the third version of the algorithm to divide $0000\ 0111_{two}$ by 0010_{two}.

Answer Figure 4.42 shows how the quotient is created in the bottom of the Remainder register and how both are shifted left in a single operation.

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

FIGURE 4.42 Division example using third algorithm in Figure 4.40. The bit examined to determine the next step is circled in color.

Signed Division

So far we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

Dividend = Quotient × Divisor + Remainder

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

+7 ÷ +2: Quotient = +3, Remainder = +1

Checking the results:

7 = 3 × 2 + (+1) = 6 + 1

If we change the sign of the dividend, the quotient must change as well:

–7 ÷ +2: Quotient = –3

Rewriting our basic formula to calculate the remainder:

Remainder = (Dividend – Quotient × Divisor) = – 7 – (–3 × +2) = –7–(–6) = –1

So,

–7 ÷ +2: Quotient = –3, Remainder = –1

4.8 Floating Point

- Real numbers:
- 3.14159265…. (π)
- 2.71828… (e)
- 0.000000001 or 1.0×10^{-9} (seconds in a nanosecond)
- 3,155,760,000 or 3.15576×10^{9} (seconds in a century)

- Scientific Notation

4.8 Floating Point

- Normalized numbers:
- A number in scientific notation that has no leading zeros.
- 1.0×10^{-9}
- 0.1×10^{-8}
- 10.0×10^{-10}

4.8 Floating Point

- Binary numbers in scientific notation:
- 1.0×2^{-1}

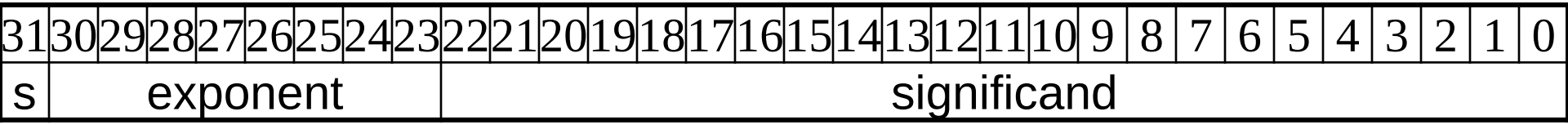
- 1. *xxxxxxxx* × 2^{yyyy}
- Computer arithmetic that support such numbers is called floating point because it represents numbers in which the binary point is not fixed.

4.8 Floating Point

- A standard scientific notation for reals in normalized form offers three advantages:
  - It simplifies exchange of data that includes floating-point numbers
  - It simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form
  - It increases the accuracy of the numbers that can be stored in a word

Floating Point Representation

- Single precision
- Sign and magnitude representation

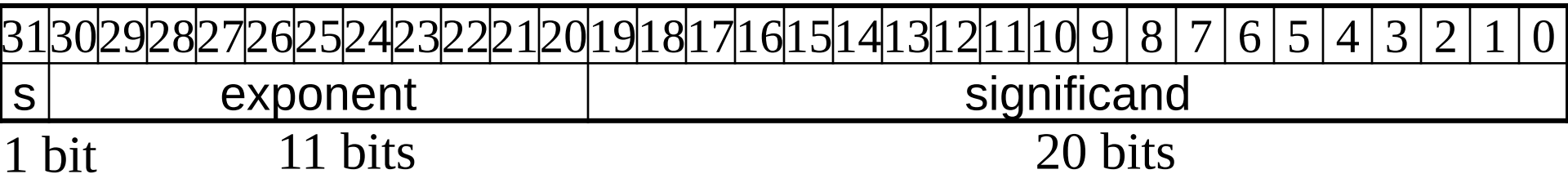


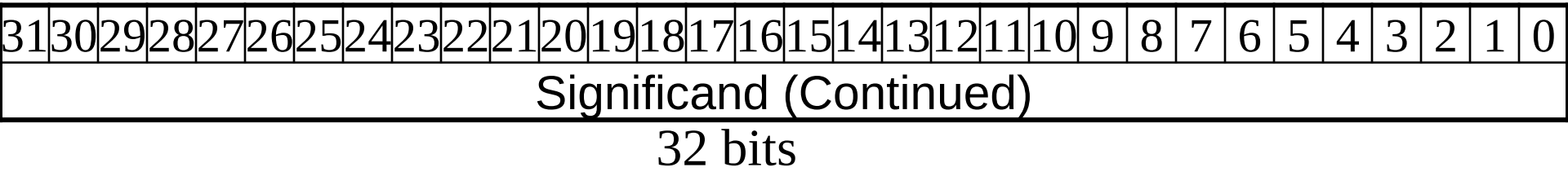
- $(-1)^S \times (1+F) \times 2^{(E-Bias)}$
- S – sign
- F – significand
- E – exponent

Floating Point Representation

- Double precision

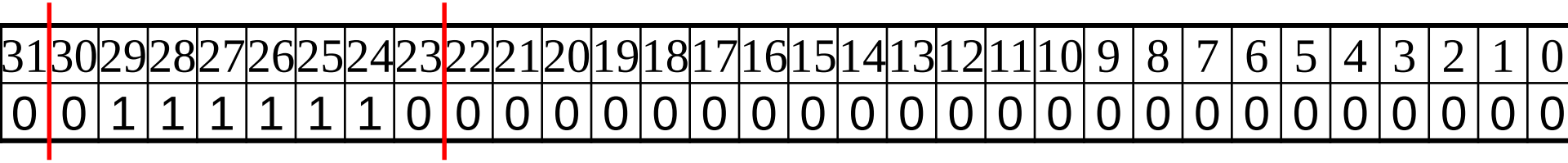
- $(-1)^S \times (1 + \text{Significand}) \times 2^E$
- $(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + \ldots\ \ldots\ ) \times 2^E$
- IEEE 754



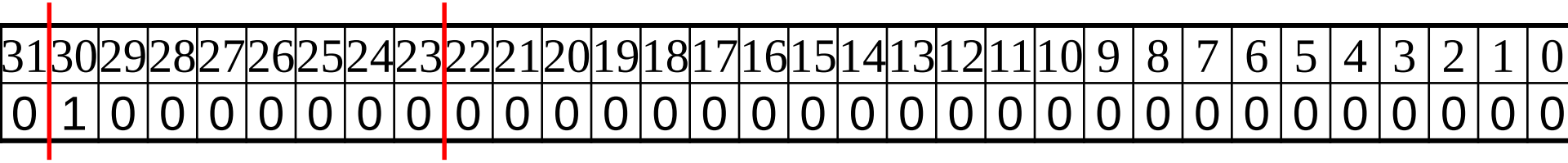


Floating Point Representation

- Biased Notation
- 1.0×2^{-1}



- $1.0 \times 2^{+1}$



- $-1 \rightarrow -1 + 127 = 126$
- $+1 \rightarrow +1 + 127 = 128$
- $(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$
- For Double Precision bias is 1023

Floating-Point Representation

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

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

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

and so when we add the bias 127 to the exponent of $-1.1_{two} \times 2^{-1}$, the result is

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

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

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

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

Converting Binary to Decimal Floating Point

Example What decimal number is represented by this word?

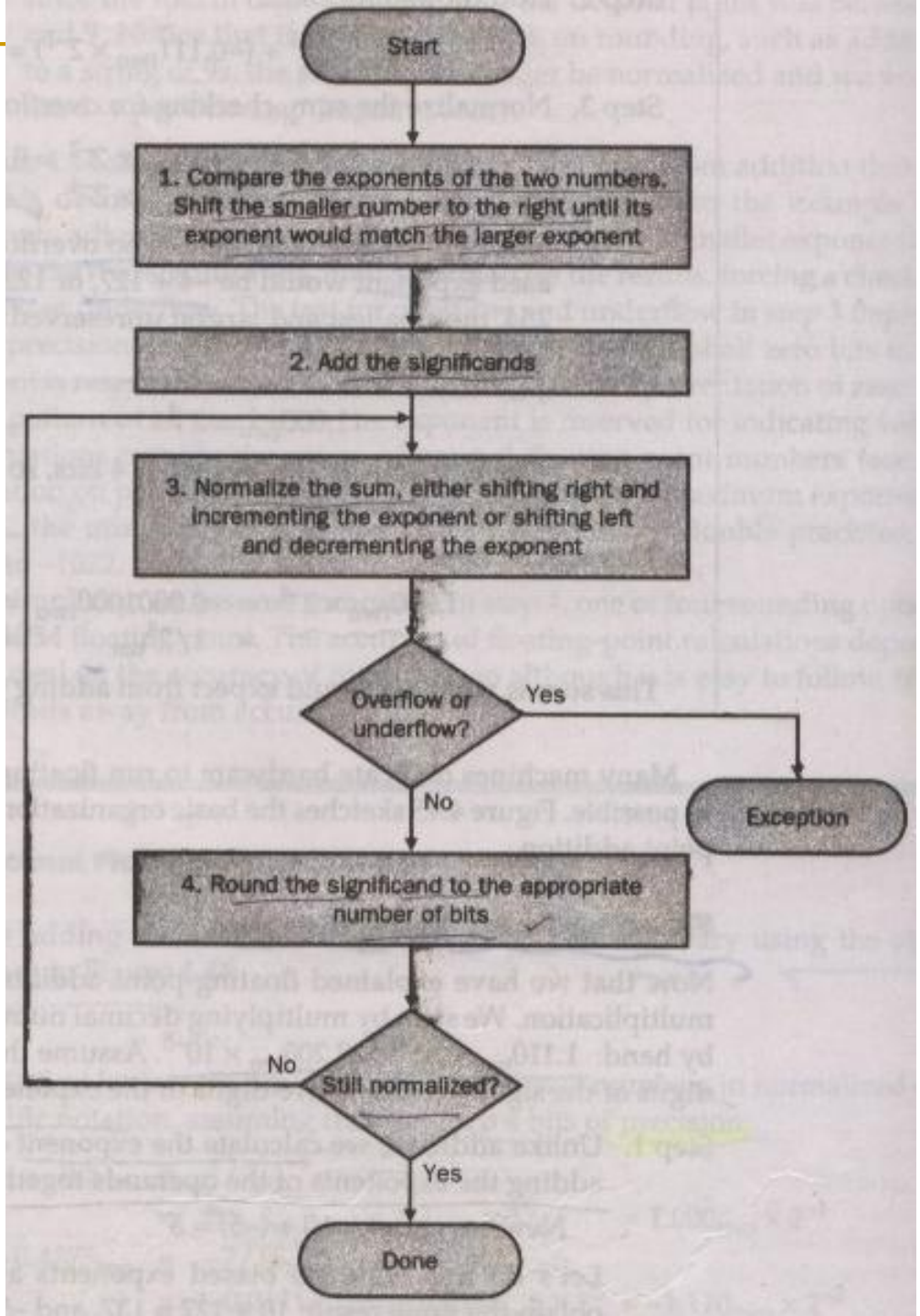
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.	.

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

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})} = (-1)^1 \times (1 + 0.25) \times 2^{(129-127)}$$
$$= -1 \times 1.25 \times 2^2$$
$$= -1.25 \times 4$$
$$= -5.0$$

In the next sections we will give the algorithms for floating-point addition

Floating Point Addition



Floating Point Addition

$9.999 \times 10^{1} + 1.610 \times 10^{-1}$

- Step 1: (Change the smaller number)

$1.610 \times 10^{-1} = 0.1610 \times 10^{0} = 0.01610 \times 10^{1}$

- Step 2: (Addition of the significands)

$$\begin{array}{r} 9.999 \\ +\ 0.016 \\ \hline 10.015 \end{array}$$

10.015×10^{1}

Floating Point Addition

- 10.015×10^1
- Step 3: (Find the normalized form)

  $10.015 \times 10^1 = 1.0015 \times 10^2$

  Check for overflow and underflow
- Step 4: (Round the number)

  $1.0015 \times 10^2 = 1.002 \times 10^2$

  Normalize again if required

Decimal Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.16.

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

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

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:

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

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:

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

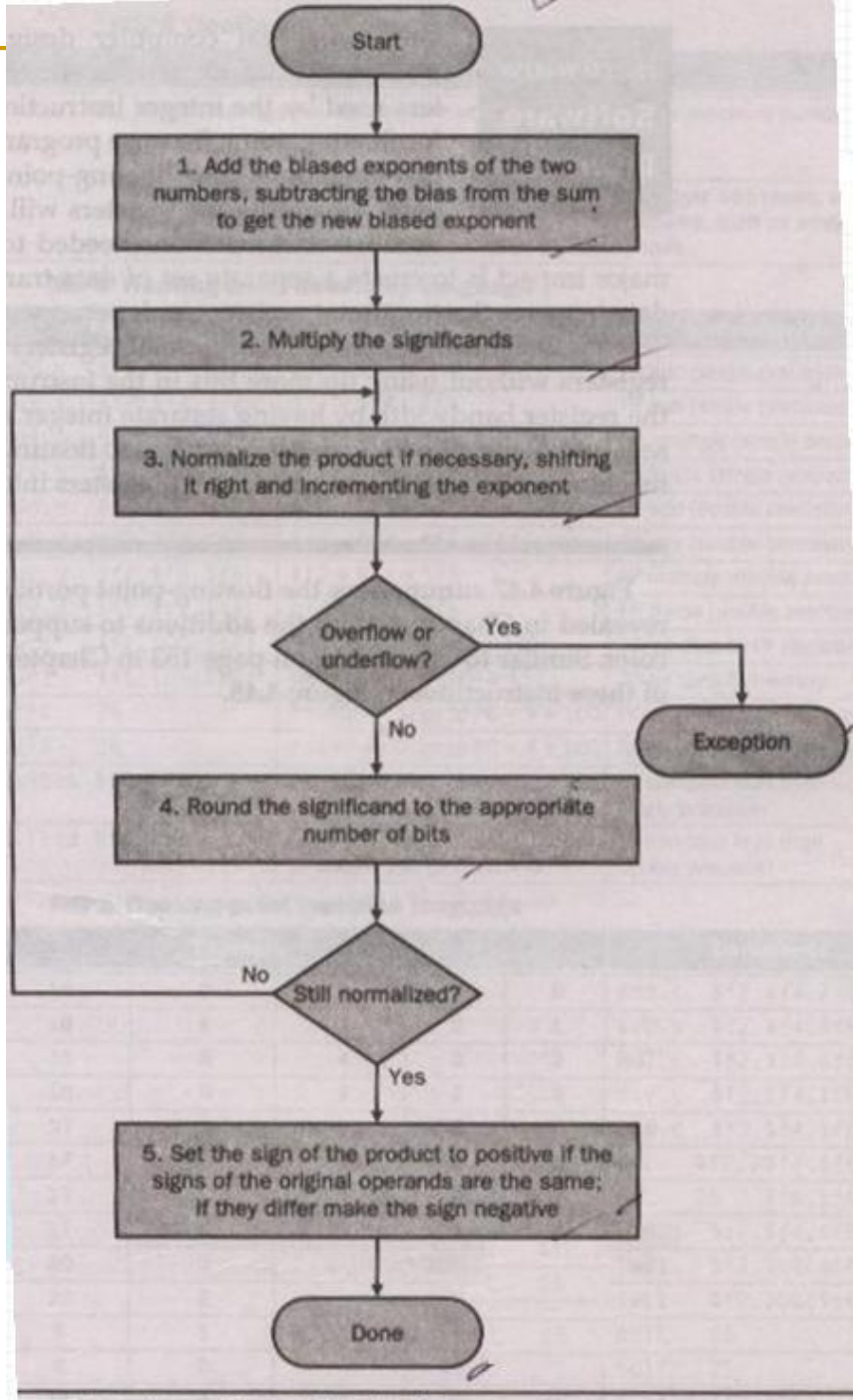
The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

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

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

Floating Point Multiplication



Floating-point multiplication.

Floating Point Multiplication

$(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$

- Step 1: (Find the exponent of the product)

  New exponent = 10 + (-5) = 5

- Considering bias

  New exponent = 137 + 122 -127

  = 132 √

Floating Point Multiplication

$(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$

New exponent = 5 or 132

- Step 2: (Multiplication of the significands)

$$
\begin{array}{r}
1.110 \\
\times\ 9.200 \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
10212000
\end{array}
$$

- $10.212000 \times 10^5 = 10.212 \times 10^5$

Floating Point Multiplication

$(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$

New exponent = 5 or 132

New product = 10.212×10^{5}

- Step 3: (Normalize the product)

  $10.212 \times 10^{5} = 1.0212 \times 10^{6}$

  Check for overflow and underflow

- Step 4: (Round the number)

  $1.0212 \times 10^{6} = 1.021 \times 10^{6}$

Floating Point Multiplication

$(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$

New product = 1.021×10^{6}

- Step 5: (Put the sign of the product)

  $+1.0212 \times 10^{6}$

Decimal Floating-Point Multiplication

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.18.

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{two} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{two} \times 2^{-3} &= -0.001110_{two} = -0.00111_{two} \\ &= -7/2^{5}{}_{ten} = -7/32_{ten} = -0.21875_{ten} \end{aligned}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

Thank You