CSE338 PS – Chapter 3 Problems

Lokman ALTIN
lokman.altin@marmara.edu.tr

Overflow

- Overflow occurs when a result is too large to be represented accurately given a finite word size.

- Overflow conditions:

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Problem 1.

Overflow occurs when a result is too large to be represented accurately given a finite word size. **Underflow** occurs when a number is too small to be represented correctly - a negative result when doing unsigned arithmetic, for example. (The case when a positive result is generated by the addition of two negative integers is also referred to as underflow by many, but in this textbook, that is considered an overflow). The following table shows pairs of decimal numbers.

	A	B
a.	69	90
b.	102	44

1) Assume A and B are unsigned 8-bit decimal integers. Calculate A – B. Is there overflow, underflow, or neither?
2) Assume A and B are signed 8-bit decimal integers stored in sign-magnitude format. Calculate A + B. Is there overflow, underflow, or neither?

Problem 1.1.a

$$
\begin{array}{rr}
 & 69 = (01000101)_2 \\
- & 90 = (01011010)_2 \\
\hline
 & (11101011)_2
\end{array}
$$

Signed (-21)

Unsigned (235)

- There is **no overflow** since the size (8 bit) is large enough to represent the result.
- There is **underflow** since the result is negative while doing unsigned arithmetic.

Problem 1.1.b

$$
\begin{array}{rr}
 & 102 = (01100110)_2 \\
- & 44 = (00101100)_2 \\
\hline
 & (00111010)_2
\end{array}
$$

$\longrightarrow$ Signed & Unsigned (58)

- There is **no overflow** since the size (8 bit) is large enough to represent the result.
- There is **no underflow** since the result is large enough to be represented in 8 bit.

Problem 1.2.a

$$69 = (01000101)_2$$

$$+ \quad 90 = (01011010)_2$$

$$(10011111)_2$$

$\boxed{\text{Signed} \quad (-97)}$

$\boxed{\text{Unsigned} \quad (159)}$

- There is **overflow** since the size (8 bit) is not large enough to represent the result. The addition of two positive numbers gives a negative result.

Problem 1.2.b

$$
\begin{array}{rr}
 & 102 = (01100110)_2 \\
+ & 44 = (00101100)_2 \\
\hline
 & (10010010)_2
\end{array}
$$

Signed (-110)

Unsigned (146)

- There is **overflow** since the size (8 bit) is not large enough to represent the result. The addition of two positive numbers gives a negative result.

Problem 2.

In this exercise we will look at a couple of other ways to improve the performance of multiplication, based on primarily on doing more shifts and fewer arithmetic operations. The following table shows pairs of hexadecimal numbers.

	A	B
a.	24	c9
b.	41	18

As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9 x 6, for example, can be written (2 x 2 x 2 + 1) x 6, we can calculate 9 x 6 by shifting 6 to the left three times and then adding 6 to that result. Show the best way to calculate A x B using shifts and adds/subtracts. Assume that A and B are 8-bit unsigned integers.

Problem 2.a

- <u>First way:</u>
  - $(24)_{16} = (36)_{10} = 32 + 4 = (2 \times 2 \times 2 \times 2 \times 2) + (2 \times 2)$
  - Addition of shift by 5 (c9 << 5) and shift by 2 (c9 << 2).
  - Total 2 shift operations and 1 add operation.

- <u>Second way:</u>
  - $(c9)_{16} = (201)_{10} = 128 + 64 + 8 + 1$
    $= (2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2) + (2 \times 2 \times 2 \times 2 \times 2 \times 2) + (2 \times 2 \times 2) + 1$
  - Addition of shift by 7 (24 << 7) and shift by 6 (24 << 6) and shift by 3 (24 << 3) and 1 (24).
  - Total 3 shift operations and 3 add operations.

- First way is the best way!

Problem 2.b

- First way:
  - $(41)_{16} = (65)_{10} = 64 + 1 = (2 \times 2 \times 2 \times 2 \times 2 \times 2) + 1$
  - Addition of shift by 6 (18 << 6) and 1 (18)
  - Total 1 shift operation and 1 add operation.

- Second way:
  - $(18)_{16} = (24)_{10} = 16 + 8 = (2 \times 2 \times 2 \times 2) + (2 \times 2 \times 2)$
  - Addition of shift by 4 (41 << 4) and shift by 3 (41 << 3) .
  - Total 2 shift operations and 1 add operation.

- First way is the best way!

Problem 3. (Past Quiz Question)

(a). Write down the binary representation of the decimal number -17.125. Assuming the IEEE 754 single precision format.

- The general representation for a single precision number:

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

- $-17.125 \times 10^0 = 10001.001 \times 2^0$
  $= 1.0001001 \times 2^4$

- Sign = negative → **1**
- Exponent = $4 + 127 = (131)_{10} = \mathbf{(10000011)_2}$
- Fraction = **00010010000000000000000**

- Final bit pattern (Single Precision → 32-bit) :

1100 0001 1000 1001 0000 0000 0000 0000

Problem 3. (Past Quiz Question)

(b). Write down the binary representation of the decimal number -17.125. Assuming the IEEE 754 double precision format.

- The general representation for a single precision number:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 1023)}$$

- $-17.125 \times 10^0 = 10001.001 \times 2^0$
  $= 1.0001001 \times 2^4$

- sign = negative → **1**
- Exponent = $4 + 1023 = (1027)_{10} = \mathbf{(10000000011)_2}$
- Fraction = **0001001000000000000000000000000000000000000000000000**

- Final bit pattern (Double Precision → 64 bit) :

1100 0000 0011 0001 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Problem 4. (Past Quiz Question)

What decimal number does the bit pattern 0x12D00000 represent if it is a floating point number? Use the IEEE 754 format.

- 0x12D00000 = 0001 0010 1101 0000 0000 0000 0000 0000
  = 0 0010 0101 1010 0000 0000 0000 0000 000

- sign = positive → **0**
- Exponent = $(0010\ 0101)_2$ = 37 → x + 127 = 37, then x = -90
- Fraction = .101

- The floating point number: $(-1)^0 \times (1 + 0.101) \times 2^{-90} = 1.101 \times 2^{-90}$

- The decimal number = ~ $(1.625) \times 10^{-27}$
  - Assume that $2^{10} =\sim 10^3$

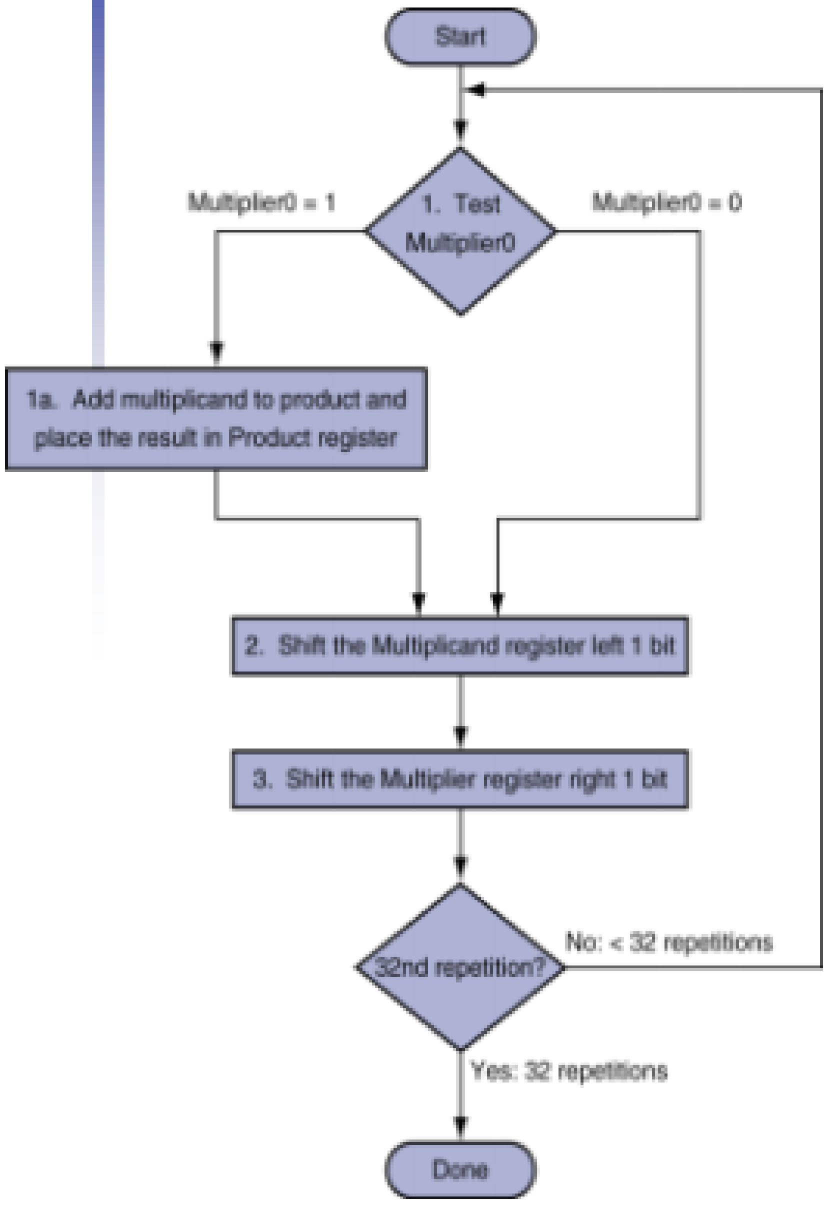
Exercise 3.5

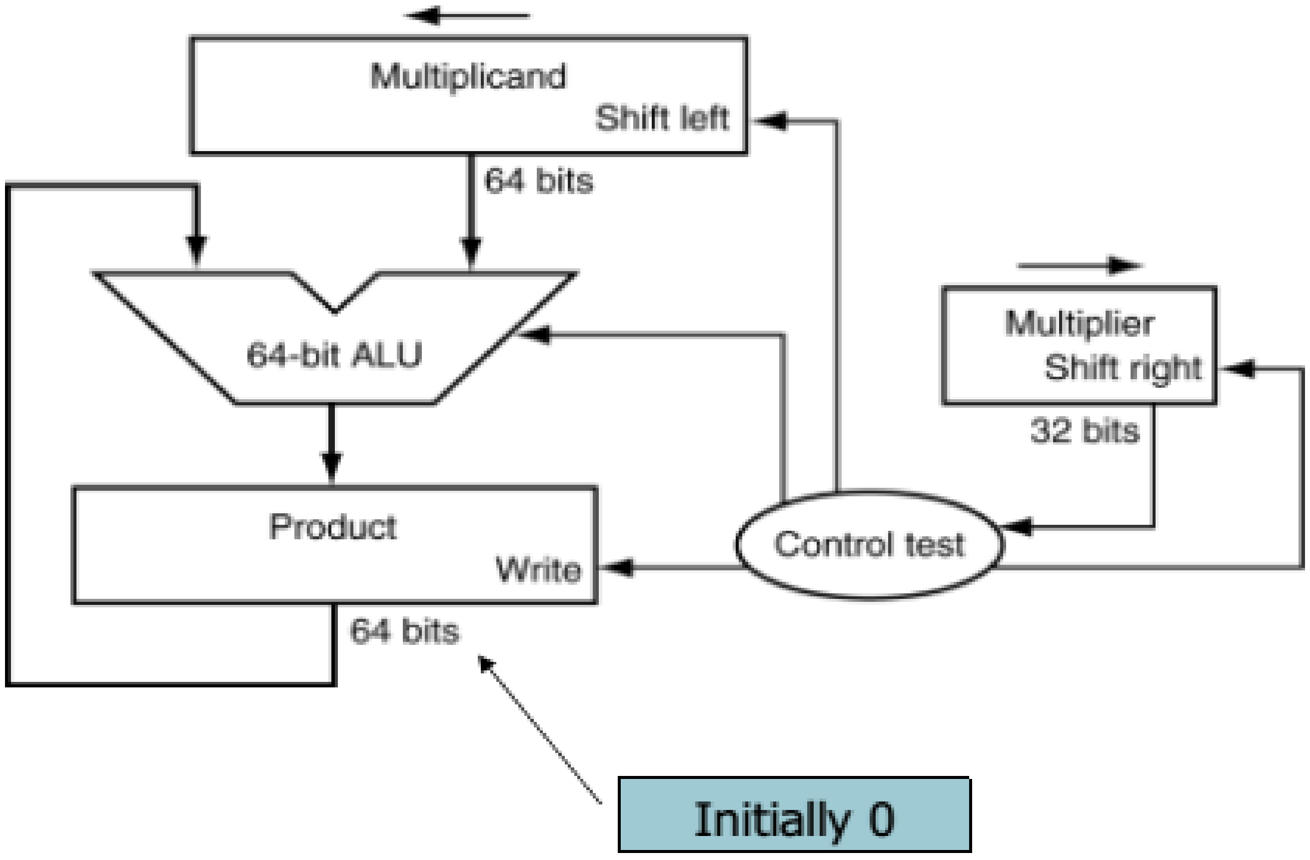
For many reasons, we would like to design multipliers that require less time. Many different approaches have been taken to accomplish this goal. In the following table, A represents the bit width of an integer, and B represents the number of time units (tu) taken to perform a step of an operation.

	A (bit width)	B (time units)
a.	4	3 tu
b.	32	7 tu

3.5.1 [10] <3.3> Calculate the time necessary to perform a multiply using the approach given in Figures 3.4 and 3.5 if an integer is A bits wide and each step of the operation takes B time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a 0 will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

Multiplication Hardware





Exercise 3.5.1

- Hardware case
  - 1 add operation + 1 shift operation + 1 test operation = 3 operations

- Software case
  - 1 add operation + 1 shift multiplicand operation + 1 shift multiplier operation + 1 test operation = 4 operations

Exercise 3.5.1.a

- Hardware case: 3 operations x 3 time units x 4 repetitions = 36 time units
- Software case: 4 operations x 3 time units x 4 repetitions = 48 time units

Exercise 3.5.1.b

- Hardware case: 3 operations x 7 time units x 32 repetitions = 672 time units
- Software case: 4 operations x 7 time units x 32 repetitions = 896 time units

The following table shows further pairs of hexadecimal numbers.

	A	B
a.	42	36
b.	9F	8E

3.6.5 [30] <3.3> Show the step-by-step result of multiplying A and B, using Booth's algorithm. Assume A and B are 8-bit two's-complement integers, stored in hexadecimal format.

Optimized Multiplier

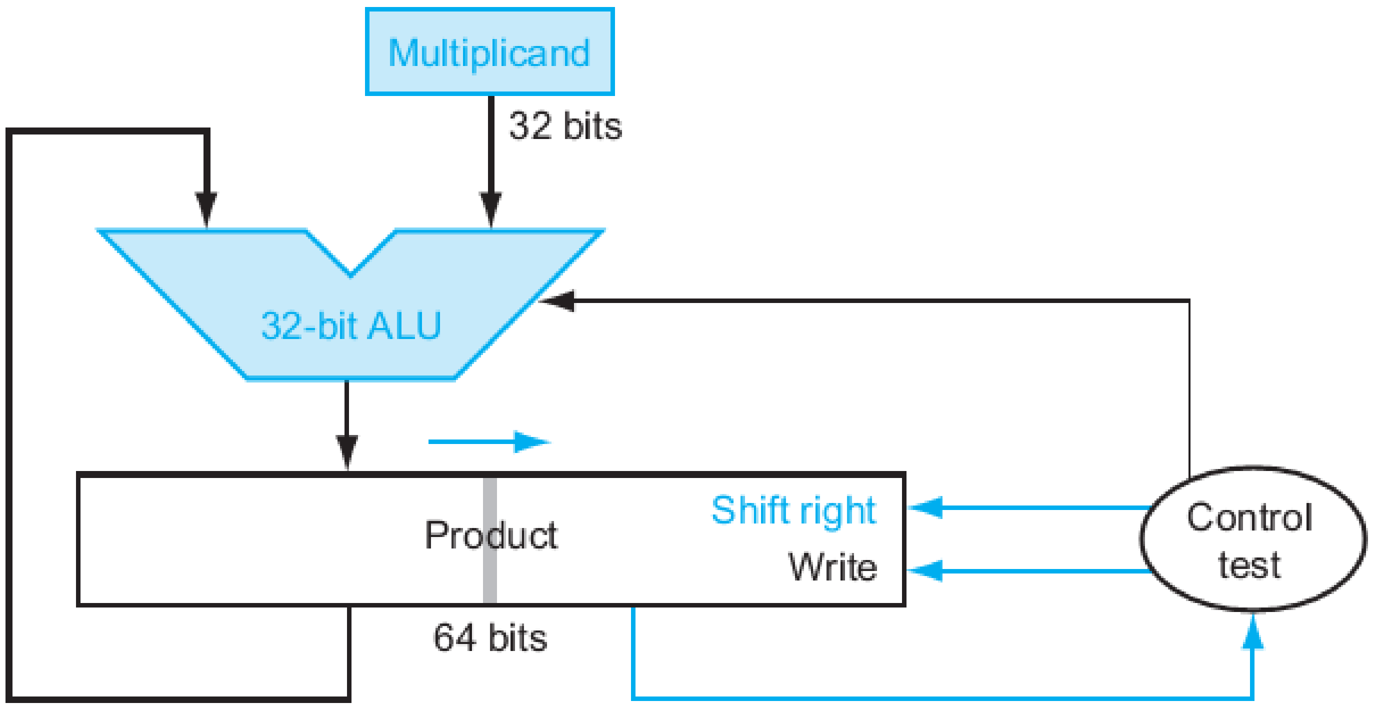


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Exercise 3.6.5.a

$(42)_{16} = 0100\ 0010$ $(36)_{16} = 0011\ 0110$

Iteration	Step	Multiplicand	Product / Multiplier
0	Initial values	0100 0010	0000 0000 0011 0110 0
1	00 , no operation shift right product	0100 0010	0000 0000 0011 0110 0 0000 0000 0001 1011 0
2	10, P = P - Multiplicand shift right product	0100 0010	1011 1110 0001 1011 0 1101 1111 0000 1101 1
3	11, no operation shift right product	0100 0010	1101 1111 0000 1101 1 1110 1111 1000 0110 1
4	01, P = P + Multiplicand shift right product	0100 0010	0011 0001 1000 0110 1 0001 1000 1100 0011 0
5	10, P = P - Multiplicand shift right product	0100 0010	1101 0110 1100 0011 0 1110 1011 0110 0001 1
6	11, no operation shift right product	0100 0010	1110 1011 0110 0001 1 1111 0101 1011 0000 1
7	01, P = P + Multiplicand shift right product	0100 0010	0011 0111 1011 0000 1 0001 1011 1101 1000 0
8	00 , no operation shift right product	0100 0010	0001 1011 1101 1000 0 **0000 1101 1110 1100** **0**

Exercise 3.6.5.b

$(9F)_{16}$ = 1001 1111 $(8E)_{16}$ = 1000 1110

Iteration	Step	Multiplicand	Product / Multiplier
0	Initial values	1001 1111	0000 0000 1000 1110 0
1	00 , no operation shift right product	1001 1111	0000 0000 1000 1110 0 0000 0000 0100 0111 0
2	10, P = P - Multiplicand shift right product	1001 1111	0110 0001 0100 0111 0 0011 0000 1010 0011 1
3	11, no operation shift right product	1001 1111	0011 0000 1010 0011 1 0001 1000 0101 0001 1
4	11, no operation shift right product	1001 1111	0001 1000 0101 0001 1 0000 1100 0010 1000 1
5	01, P = P + Multiplicand shift right product	1001 1111	1010 1011 0010 1000 1 1101 0101 1001 0100 0
6	00 , no operation shift right product	1001 1111	1101 0101 1001 0100 0 1110 1010 1100 1010 0
7	00 , no operation shift right product	1001 1111	1110 1010 1100 1010 0 1111 0101 0110 0101 0
8	10, P = P - Multiplicand shift right product	1001 1111	0101 0110 0110 0101 0 **0010 1011 0011 0010** 1

Binary Divison EX:1

```
          000111
        ________
110    | 101010
-------
       −   0            1<110
       ---------
          10
       −   0            10<110
       ---------
          101
       −    0           101<110
       ---------
          1010
       −  110           1010>110
       ---------
           100
           1001
        −  110
        ---------
             110
          −  110
          ---------
              000
```

Binary Divison EX:2

```
            0001001
          ---------
1000  |   1001010
------
          1
          10
          100
          1001
        _ 1000
        ----------
              10
              101
              1010
          _   1000
          ----------
                 10
```

Exercise 3.7

Let's look in more detail at division. We will use the octal numbers in the following table.

	A	B
a.	50	23
b.	25	44

3.7.1 [20] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using the hardware described in Figure 3.9. You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers.

Division Hardware

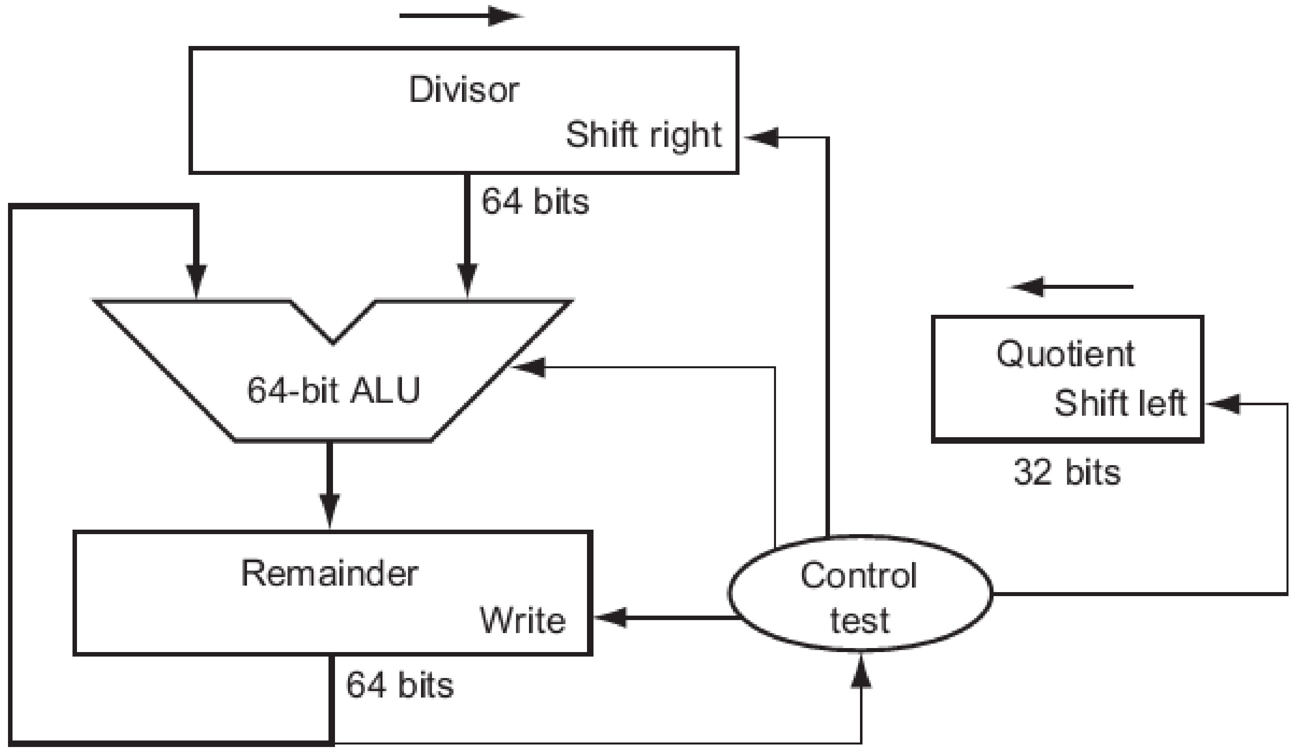
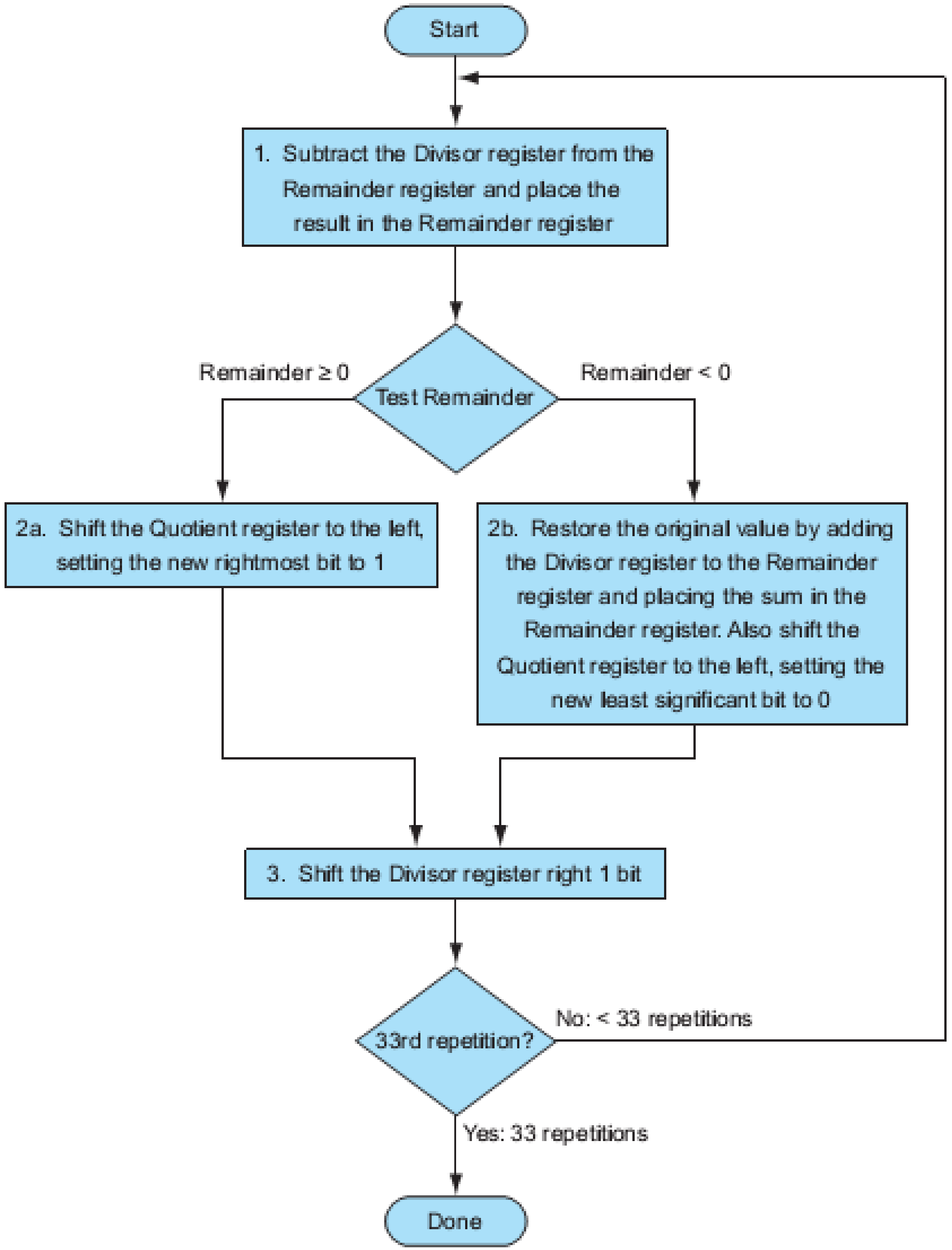


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Division Hardware



Exercise 3.7.1.a

$(50)_8 = 101000$ $(23)_8 = 010011$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	000000	010 011 000 000	000 000 101 000
1	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	010 011 000 000 001 001 100 000	101 101 101 000 000 000 101 000
2	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 100 110 000	110 111 001 000 000 000 101 000
3	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 010 011 000	111 011 111 000 000 000 101 000
4	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 001 001 100	111 110 010 000 000 000 101 000
5	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 000 100 110	111 110 111 100 000 000 101 000
6	Rem = Rem – Div Rem >= 0, Q << 1 Shift right Divisor	000 001	000 000 010 011	000 000 000 010
7	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 010	000 000 001 101	111 111 101 111 000 000 000 010

Exercise 3.7.1.a

$(25)_8 = 010101$ $(44)_8 = 100100$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	000000	100 100 000 000	000 000 010 101
1	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	010 010 000 000	100 011 101 011 000 000 010 101
2	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	001 001 000 000	101 110 010 101 000 000 010 101
3	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 100 100 000	110 111 010 101 000 000 010 101
4	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 010 010 000	111 011 110 101 000 000 010 101
5	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 001 001 00	111 110 000 101 000 000 010 101
6	Rem = Rem – Div Rem < 0, R+D, Q << Shift right Divisor	000 000	000 000 100 100	111 111 001 101 000 000 010 101
7	Rem = Rem – Div Rem < 0, R + D, Q << Shift right Divisor	000 000	000 000 010 010	111 111 110 001 000 000 010 101