5118007-02 Computer Architecture

Ch. 3 Arithmetic for Computers

9 Apr 2024

Shin Hong

Integer Arithmetic

- addition and subtraction
  - dealing with overflow
- multiplication
- division

Addition

- Digits are added bit-by-bit from right to left, considering carries passed from the right

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten}
\end{array}
$$

Subtraction

- A subtraction is basically the same as a combination of negation and addition

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

Overflow (1/2)

- An overflow occurs when the resulting value cannot be represented within given bits
  - only when adding two numbers of the same sign, and not possible when two numbers of different signs are added

- Overflow for signed numbers
  - adding two positive numbers yields a negative number
  - subtracting two negative numbers yields a positive number
  - the occurrence of overflow with `add`, `addi`, `sub` instructions causes an exception

Overflow (2/2)

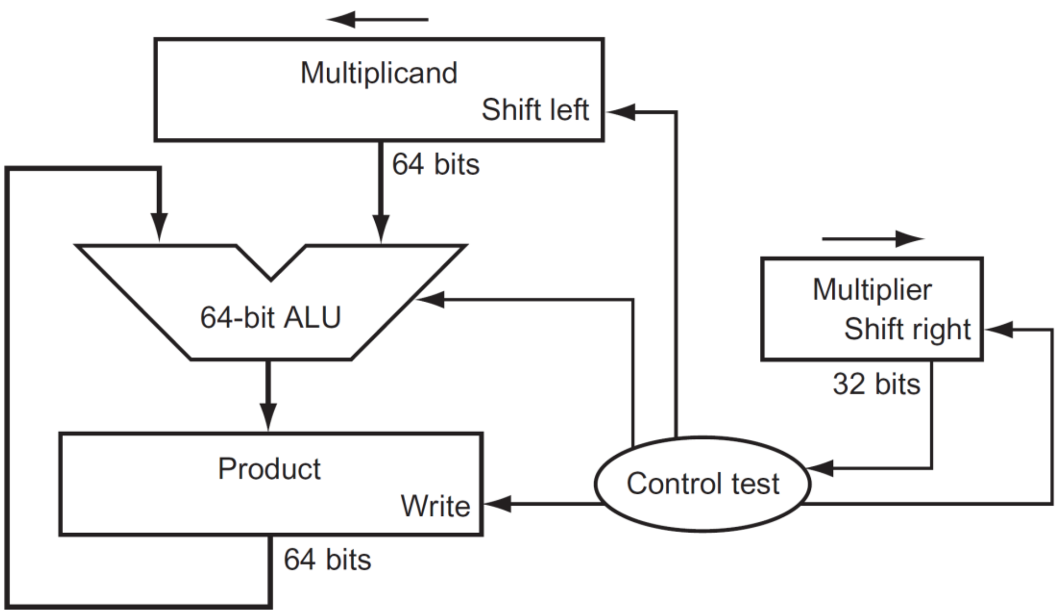
- Overflow for signed numbers
  - addition of two numbers yields a smaller number
  - subtraction of two numbers yields a greater number
  - overflow with `addu`, `addiu`, `subu` do not cause exception, thus C compiler uses these instructions for arithmetics.

Multiplication

```
Multiplicand          1000ten
Multiplier     x      1001ten
                      -------
                      1000
                     0000
                    0000
                   1000
                   -------
Product            1001000ten
```

- Basically, multiplication can be implemented as a sequential operations of right-shifts, left-shifts and additions
- Multiplication of a *n*-bits multiplicand and a *m*-bits multiplier produces a *n*+*m* product
  - overflow may occur

Sequential Multiplication Hardware



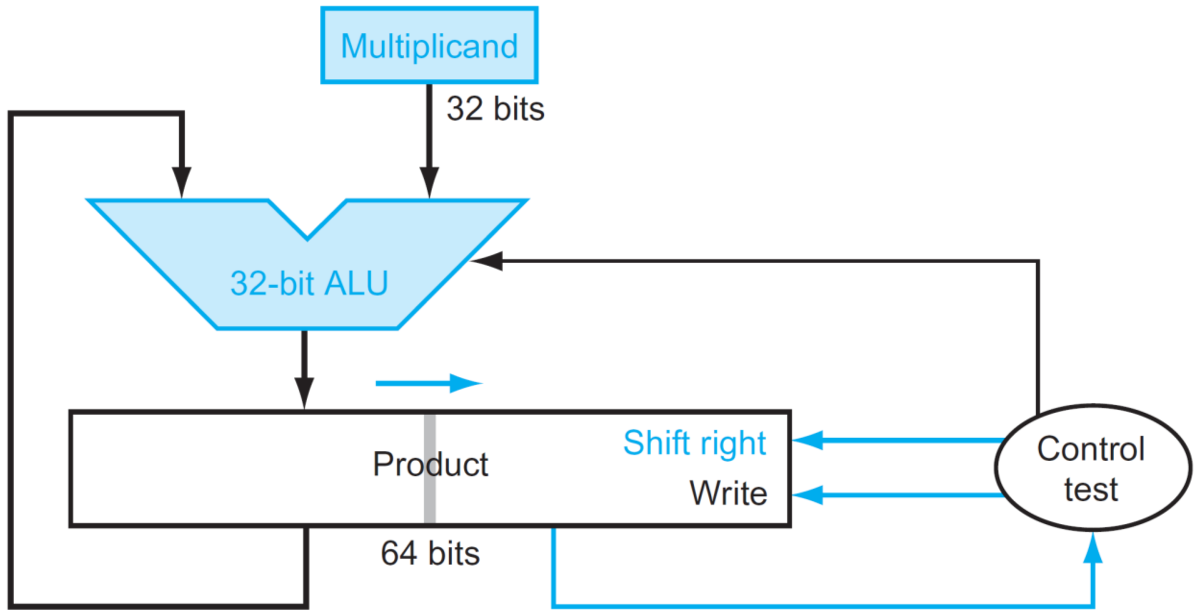

```
foreach bit of multiplier begin
  if the LSB of multiplier is 1 then
    add multiplicand and product;
  end-if
  shift-left the multiplicand ;
  shift-right a multiplier ;
end-for
```

Example

- Multiply $0010_{(2)}$ and $0011_{(2)}$

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

Refined Version



Signed Multiplication

- Naive approach
  - store the signs of multiplicand and multiplier respectively
  - convert them to positive numbers
  - perform multiplication
  - apply the proper sign to the product

- Interestingly, the aforementioned multiplication algorithm works sound for two's compelemtns
  - https://pages.cs.wisc.edu/~markhill/cs354/Fall2008/beyond354/int.mult.html

No Thinking Method

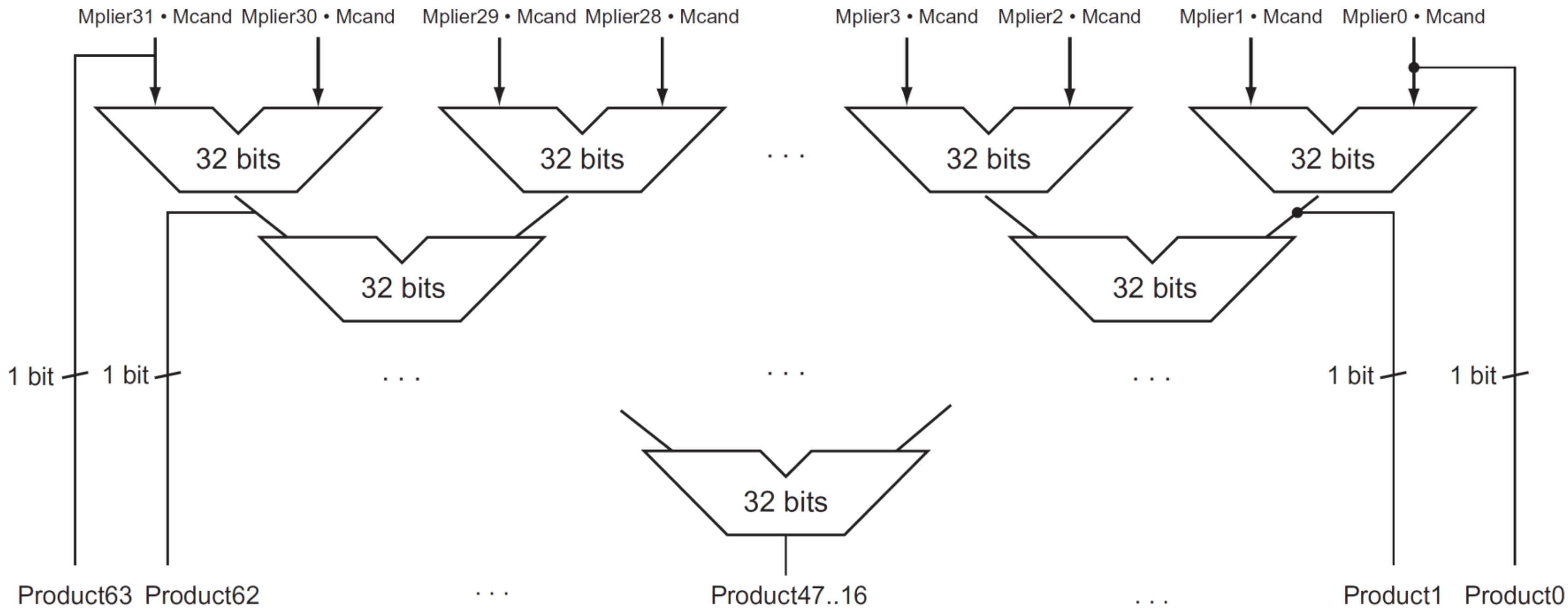
- sign-extend to both multiplicand and multiplier before multiplication

```
         1111 1111       -1
       × 1111 1001     ×  -7
  ----------------    ------
          11111111         7
         00000000
        00000000
       11111111
      11111111
     11111111
    11111111
+  11111111
  ----------------
   1  00000000111
```

```
WRONG !
  0011 (3)
× 1011 (-5)
------
  0011
 0011
0000
+ 0011
--------
 0100001
not -15 in any
 representation!
```

```
Sign extended:
  0000 0011 (3)
x 1111 1011 (-5)
-----------
   00000011
  00000011
 00000000
00000011
00000011
00000011
00000011
+ 00000011
-----------------
 1011110001
```

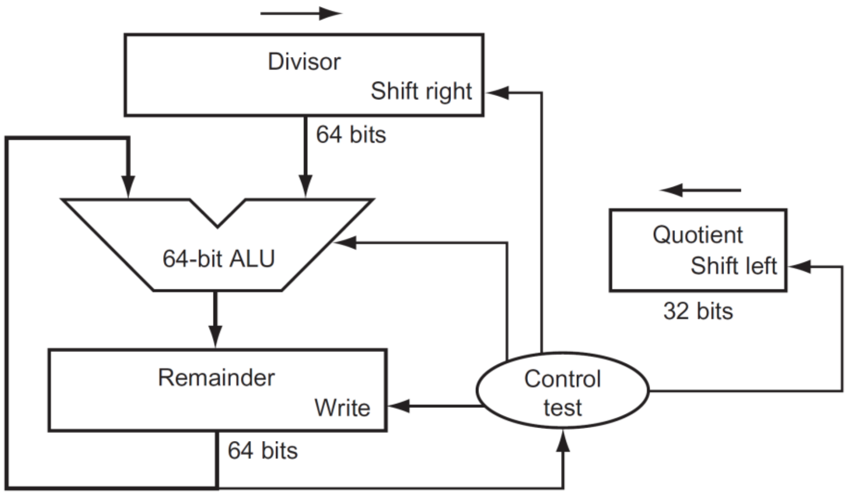
Fast Multiplication with Large Circuits



Division

- Division can be implemented as a series of shift-right, shift-left and subtraction

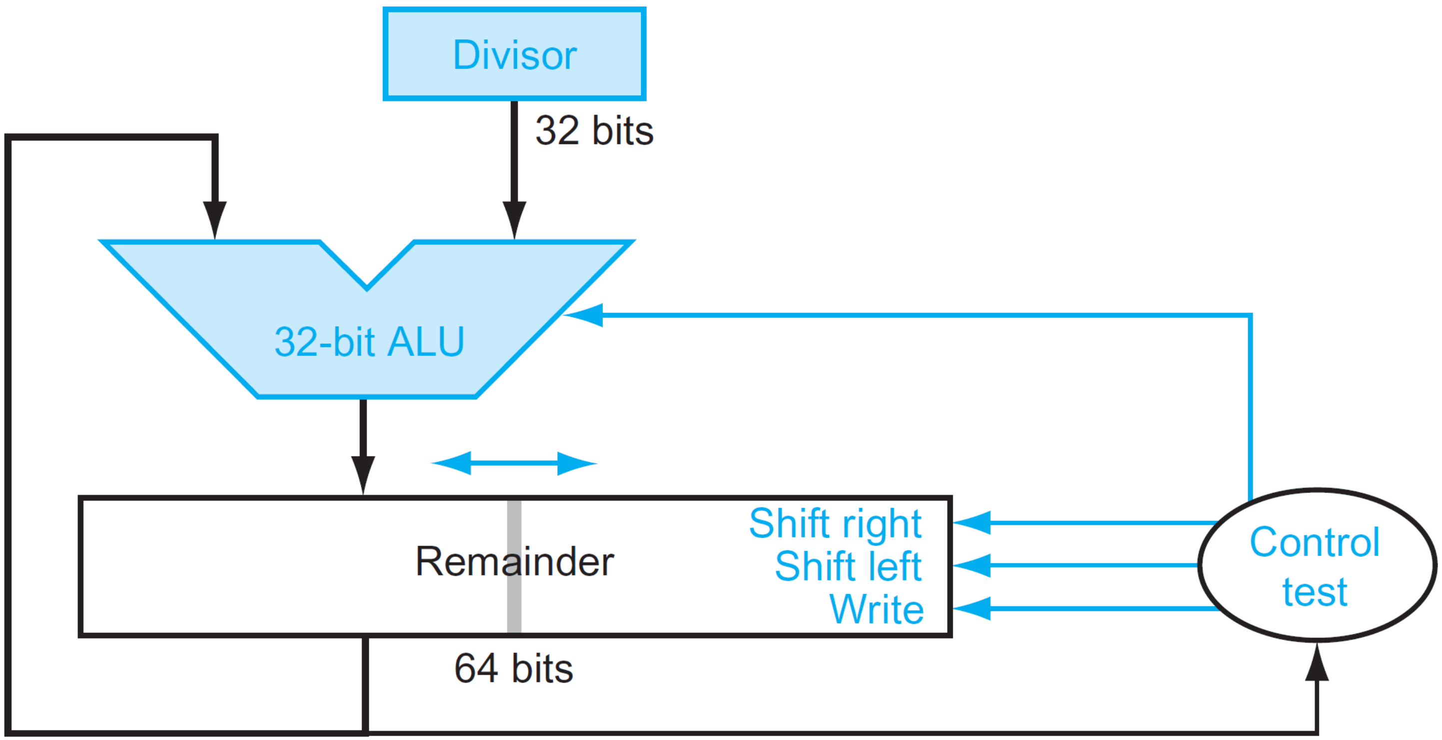
```
                          1001ten      Quotient
Divisor 1000ten | 1001010ten      Dividend
                  -1000
                     10
                     101
                     1010
                    -1000
                       10ten       Remainder
```



Example: $0111_{(2)}$ / $0010_{(2)}$

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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ 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 $\Longrightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Faster Implementation



Floating Point (1/2)

- A real number can be represented as a combination of an integer and a fraction
- Scientific notation for decimal numbers
  - a single digit to the left of the decimal point and a power of ten
  - ex.
    - 1.02×10^{-9}
    - 3.15576×10^{9}

Floating Point (2/2)

- Scientific notation of binary number
  - $1.abcd$ X 2^n
    - 1 X 2^n + $a \times 2^{n-1}$+ $b \times 2^{n-2}$ + $c \times 2^{n-3}$ + $d \times 2^{n-4}$

- Convert a decimal real number to binary
  - Examples
    - 4.5
    - 25.25
    - 32.45

Floating Number Representation (1/2)

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
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

- $(-1)^S$ X F X 2^E
  - S: sign
  - F: fraction
  - E: exponent

- Single-precision: 8 bits for E and 23 bits for F
  - the smallest possible non-zero positive: 2.0 X 10^{-38}
  - the greatest possible positive: 2.0 X 10^{38}

Floating Number Representation (2/2)

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
s	exponent												fraction																		
1 bit	11 bits												20 bits																		
fraction (continued)																															
32 bits																															

- Double precision: 11 bits for E and 52 bits for F
  - the smallest possible non-zero positive: 2.0×10^{-308}
  - the greatest possible positive: 2.0×10^{308}

IEEE 754 Standard of Floating Number Encoding

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- Special symbols
  - Infinity
  - Not a Number (NaN)

Biased Notation

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
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

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
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

- Exponent does not use the two's complement notation, but a biased notation to make real number comparison similar to that of integer
- If an actual exponent is X with bias B, then represent it as X + B
  - Ex. for single precision with bias 127, if an exponent is -1,
    it is represented as $0111\ 1110_{(two)} = 126 = (-1 + 127)$

Example

- Represent -0.75$_{(ten)}$ in IEEE 754 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

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

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

Example

- Represent $-0.75_{(ten)}$ in IEEE 754 single and double precision

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

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

- What decimal number it is?

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	.	.	.

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

- Example

$$9.999_{\text{ten}} \times 10^{1} + 1.610_{\text{ten}} \times 10^{-1}.$$

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:

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

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:

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

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

This sum is then

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

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