Technische Hochschule Rosenheim

Computer Science Fundamentals

Number Systems – Binary Multiplication & Division, Floating-Point Numbers

Technische Hochschule Rosenheim
Winter 2021/22
Prof. Dr. Jochen Schmidt

Overview

- Binary multiplication, division
- Floating-point numbers

Binary Multiplication

Technische Hochschule Rosenheim

- Rules for the multiplication of two binary digits
  - $0 \cdot 0 = 0$
  - $0 \cdot 1 = 0$
  - $1 \cdot 0 = 0$
  - $1 \cdot 1 = 1$
- Identical to the rules of logical AND!
- Multiplication of multi-digit numbers
  - Multiplication of the multiplicand by the individual digits of the multiplier
  - Proper addition (at the correct position) of the interim results

Binary Multiplication – Example

Example: 10 · 13

$$\begin{array}{r} 1010 \cdot 1101 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 10000010 \end{array}$$

Result: 130_{10}

Binary Multiplication – Example

Typically

- with fixed number of bits
- multiplier digits from right to left (instead of left to right as in the previous slide)
- shift operations

Example: 8 Bits, 10 · 13 = 0000 1010 · 0000 1101

0000 1010	1
0000 0000	0, shift multiplicand left by 1 bit
0010 1000	1, shift left by 2 bits
0101 0000	1, shift left by 3 bits
0000 0000	0, shift left by 4 bits
…	
<u>0000 0000</u>	0, shift left by 7 bits
1000 0010	= 130

Note:
We do not have to store store all these and add at the end.
We can directly add each shifted multiplicand and get an intermediate result

Binary Multiplication – Example

This also works in **complement** representation with negative numbers (example: 8 Bits, two's complement)

(–5) · 3 = 1111 1011 · 0000 0011

1111 1011	1
1111 0110	1, shift left by 1 bit
0000 0000	0, shift left by 2 bits
…	
0000 0000	0, shift left by 7 bits
10 1111 0001	= –15

overflow discarded

3 · (–5) = 0000 0011 · 1111 1011

0000 0011	1
0000 0110	1, shift left by 1 bit
0000 0000	0, shift left by 2 bits
0001 1000	1, shift left by 3 bits
0011 0000	1, shift left by 4 bits
0110 0000	1, shift left by 5 bits
1100 0000	1, shift left by 6 bits
1 1000 0000	1, shift left by 7 bits
10 1111 0001	= –15

overflows discarded

(–3) · (–5) = 1111 1101 · 1111 1011

1111 1101	1
1111 1010	1, shift left by 1 bit
0000 0000	0, shift left by 2 bits
1110 1000	1, shift left by 3 bits
1101 0000	1, shift left by 4 bits
1010 0000	1, shift left by 5 bits
0100 0000	1, shift left by 6 bits
1000 0000	1, shift left by 7 bits
0000 1111	= +15

overflows discarded

note: there are also other ways to do this multiplication

Binary Multiplication – Fixed-Point Example

Example: 17.375 · 9.75

```
1 0 0 0 1.0 1 1 · 1 0 0 1.1 1
-------------------------------
      1 0 0 0 1 0 1 1
            1 0 0 0 1 0 1 1
              1 0 0 0 1 0 1 1
                1 0 0 0 1 0 1 1
-------------------------------
      1 0 1 0 1 0 0 1.0 1 1 0 1
```

Intermediate result

Insert point at correct position

Result: $\mathbf{169.40625_{10}}$

Binary Division

Basically the same as decimal

Example 20 : 6

```
1 0 1 0 0   :   1 1 0   =   1 1.0 1 0 1 …
 -1 1 0
 ------
   1 0 0 0
    -1 1 0
   -------
       1 0 0 0
        -1 1 0
       -------
           1 0 0 0
            -1 1 0
           -------
                 …
```

Result: 3.333…$_{10}$

Special Case

If multiplier or divisor is a power of two 2^k

- Multiplication or division can be done easier and faster
- By shifting a corresponding number of bits (k) to the left or right
- For 2^1 by 1 Bit, for 2^2 by 2 bits, for 2^3 by 3 bits etc.

Special Case – Shift – Examples

				1	1	0	1
			1	1	0	1	0
		1	1	0	1	0	0

2 bits to the left

= 52

- $13 \cdot 4$

  1101 · 100 = 110100

- $20 \cdot 8$

  10100 · 1000 = 10100000

  3 bits to the left

- $20 : 4$

  10100 : 100 = 101

  2 bits to the right

- $26 : 4$

  11010 : 100 = 110 (Remainder 2) (⟶ possible loss of information!)

  2 bits to the right

“Real” Numbers

- we cannot represent real numbers in a computer
  - we’ll always have to use a finite number of bits
  - so, we actually have rational numbers only, represented as (binary) fractions

- Two main types
  - fixed-point numbers (*Festkommazahlen*)
  - floating-point numbers (*Gleitkommazahlen*)

Fixed-point Numbers

- Point separating integer from fractional part always at the same position
  - therefore, the point itself does not have to be stored
- Z_2 has length $n+m$ bit
  n digits left and **m** digits right of the point

$$Z_2 = z_{n-1}z_{n-2}\cdots z_1 z_0\,.\,z_{-1}z_{-2}\cdots z_{-m}$$

$$Z_{10} = \sum_{i=-m}^{n-1} z_i \cdot 2^i$$

- Disadvantages of fixed-point arithmetic
  - Using a fixed number of bits, only a limited range of values can be covered
  - The location of the point is always the same

    (Where to put it, if sometimes you have to operate with very small, highly accurate values, and at other times with very large values?)
- Fixed-point arithmetic is only used in special purpose computers
  - otherwise: floating-point arithmetic

Floating-point Numbers

Technische Hochschule Rosenheim

- Example: Any decimal fraction can be written in the following form

  $$2.3756 \cdot 10^3$$

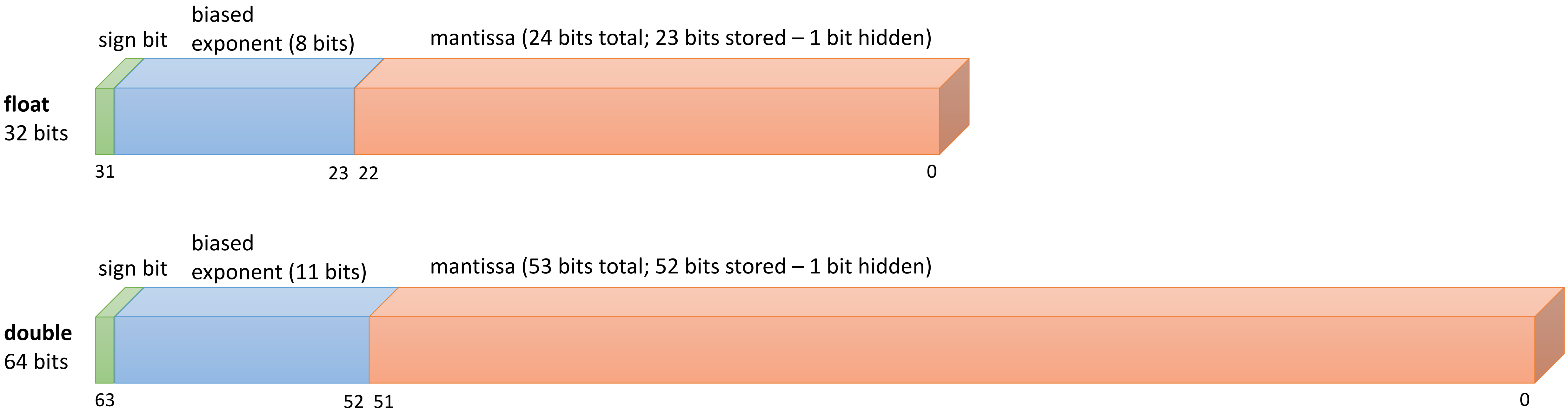
- Two components
  - **Mantissa (or Signifand)** (2.3756) and
  - **Exponent** (3), which is integer

- Used in most computers
  - with base 2 instead of base 10

- We have to specify, how many bits to use for the representation; most common:
  - single precision (32 bits, data type `float`) or
  - double precision (64 bits, data type `double`)

Binary IEEE Floating-point Format

- standard IEEE 754-2019
- C/C++ and Java data types use 4 bytes for float and 8 bytes for double
- the standard also defined types for half (16 Bit) and quadruple (128 Bit) precision



IEEE Floating-point Format

Technische Hochschule Rosenheim

- we use normalized floating-point numbers
  - the exponent is changed
  - such that the (not stored) point is always directly to the right of the first non-zero digit
  - which, in binary is always 1 ⟶ no need to store it (the hidden bit of the mantissa)

- Example: 17.625_{10}

  $= 16 + 1 + \frac{1}{2} + \frac{1}{8}$
  $= 10001.101_2$
  $= 10001.101 \cdot 2^0$ this notation is a bit of a mixture binary – decimal

- Normalized form
  - move the point directly to the right of the first significant digit
  - change the exponent accordingly

  $= 1.0001101 \cdot 2^4$

IEEE Floating-point Format

- In the mantissa the most significant bit to the left of the (not stored) point is always 1
  - ⟶ no need to store it (the hidden bit of the mantissa)
  - except for 0.0 and some other special cases

- Exponent is an integer, which (after adding a bias) can be represented without a sign
  - the value of the bias depends on precision
    - float (4 bytes, 8 bits for exponent): Bias = 127
    - double (8 bytes, 11 bits for exponent): Bias = 1023
  - using bias addition, no special sign treatment is required for exponent arithmetic (always positive)
  - it would have been possible to use complement representation instead
    - but bias representation makes comparison of floating-point numbers easier

- Sign bit indicates the sign of the mantissa
  - mantissa stored as absolute value
  - sign bit = 0 ⟶ positive
  - sign bit = 1 ⟶ negative

IEEE Floating-point Format – Example

- 17.625 ($1.0001101 \cdot 2^4$)

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	0	0	0	0	0	1	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Biased exponent: 10000011 = 131
- Bias: 01111111 = 127
- Real exponent: 00000100 = 4

IEEE Floating-point Format

Technische Hochschule Rosenheim

Precision	Single	Double
Sign bits	1	1
exponent bits	8	11
mantissa bits	23	52
bits total	32	64
Bias	127	1023
exponenten range	[-126, 127]	[-1022, 1023]

IEEE Floating-point Format – Zero

- positive (+0.0) and negative (–0.0) zero
  - these compare to „equal“!
  - (biased) exponent = 0
  - mantissa = 0
  - sign = 0/1

- Usage
  - Representation of zero
  - Rounding to +/– 0.0 for underflows (there’s a gap around zero – *underflow gap*)

IEEE Floating-point Format – Infinity

- plus (+∞) and minus (–∞) infinity
  - (biased) exponent = 111….1
  - mantissa = 0
  - sign = 0/1

- Usage
  - Numbers with too large absolute values to be represented (overflow)
  - Computations that by definition result in infinity
    (e.g., division of a number $z \neq 0$ by zero: $z / 0.0 = \infty$)

IEEE Floating-point Format – NaN

- NaN: Not a Number
  - (biased) exponent = 111….1
  - mantissa > 0
  - sign = 0/1

- Usage
  - Representation of invalid values
  - Computations that provide undefined results, e.g.,
    - 0.0 / 0.0 = NaN
    - ∞ / ∞ = NaN
    - $\sqrt{-3}$ = NaN

- comparisons with NaNs always result in *false*
  - even when testing NaN == NaN (⟶ *false*)

Example

Convert the decimal number 125.875 to `float` (IEEE format)

1. Convert to binary as usual, ignore sign

   Integer part: $125_{10} = \mathtt{1111101}_2$

   Fractional part: $0.875_{10} = \mathtt{0.111}_2$

   $$0.875 \cdot 2 = 1.75 \longrightarrow \mathbf{1}$$
   $$0.75 \cdot 2 = 1.5 \longrightarrow \mathbf{1}$$
   $$0.5 \cdot 2 = 1.0 \longrightarrow \mathbf{1}$$

2. Normalize

   $\mathtt{1111101.111} \cdot 2^0 = \mathtt{1.111101111} \cdot 2^6$

3. Determine exponent in binary

   bias for float: 127_{10}

   $2^6 \longrightarrow 6_{10} + \text{bias} = 133_{10} = \mathtt{10000101}_2$

4. Determine sign bit: positive $\longrightarrow$ **0**

5. Combine results

s	exponent	mantissa
0	10000101	11110111100000000000000

Floating-point Inaccuracies

- Floating-point numbers that can be accurately represented in the decimal system cannot always be accurately represented in the dual system
  - Note: Never compare `float` or `double` values for equality!
- Examples on the following slides: Print numbers from 0.1 to 1.0 using step 0.1

Floating-point Inaccuracies

Loop using floats as counters: infinite loop, termination condition is never reached

```c
#include <stdio.h>

int  main(void)
{
   float  i = 0.1;

   while (i != 1.0)
   {
      printf("%.10f\n", i);
      i = i + 0.1;
   }
   return 0;
}
```

Floating-point Inaccuracies

This gives the desired result:

```
#include <stdio.h>

const float EPSILON = 1e-6;

int  main(void)
{
   float  i = 0.1;

   while (i <= 1.0+EPSILON)
    {
      printf("%.10f\n", i);
      i = i + 0.1;
    }
   return 0;
}
```

Floating-point Inaccuracies

Technische Hochschule Rosenheim

Better: Avoid inaccuracies by using integers as loop counters

```
#include <stdio.h>

int  main(void) {
   int  i = 1;

   while (i <= 10)
    {
      printf("%.10f\n", (float)i/10);
      i = i + 1;
    }
   return 0;
}
```

Floating-point Inaccuracies

Technische Hochschule Rosenheim

- Floating-point arithmetic is not associative!
  - $(u + v) + w \neq u + (v + w)$
    $(u \cdot v) \cdot w \neq u \cdot (v \cdot w)$
- ...and not distributive
  - $u \cdot (v + w) \neq (u \cdot v) + (u \cdot w)$
- Example (decimal, accuracy of 8 digits)
  - (11111113. + (-11111111.)) + 7.5111111 =
    2.0000000 + 7.5111111 = 9.5111111
  - 11111113. + (-11111111. + 7.5111111) =
    11111113. + (-11111103.) = 10.000000