CS 3410 Lab 2

Fall 2026

Cornell Bowers CIS
Computer Science

Agenda

1. Float Recap
2. Minifloat: Conversions, Addition, and Practice
3. Assignment 2 Tips

Float Recap

Recap on floating-point numbers

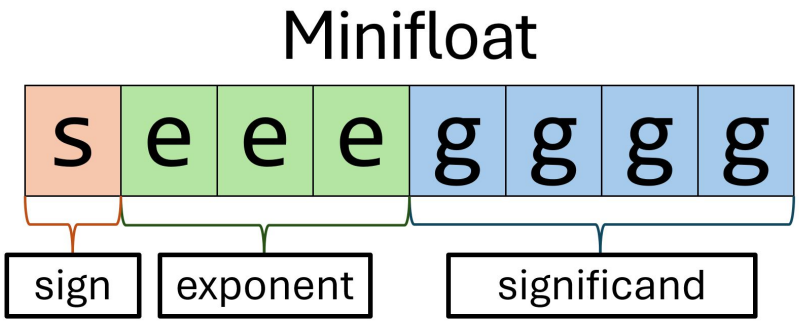
Floating-point numbers have three components:

- **`Sign s:`** leftmost bit—indicates a positive (0) or negative (1) number
- **`Exponent e:`** bits after the sign bit—adjusts the magnitude of the number
- **`Significand/mantissa g:`** rightmost bits—represents the significant digits

IEEE 754 Standard for 32-bit: 1 bit for sign, 8 bits for exponent, 23 bits for significand

In this section we'll use **8-bit minifloats**: 1 bit for sign, 3 bits for exponent (bias of 3), and 4 bits for significand

Minifloat representation



1 bit for sign, **3 bits** for exponent, **4 bits** for significand (mantissa)

Bias of 3, special cases for +/-0.0

$$(-1)^s \times 1.g_3g_2g_1g_0 \times 2^{e-3}$$

Decimal-to-float conversion

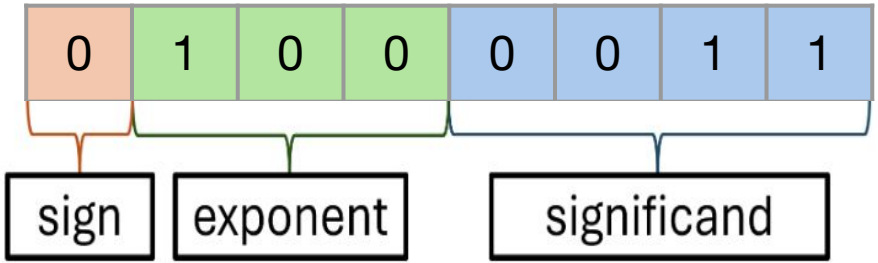
Steps to convert decimal to floating point:

1. **Convert** integer and fractional parts to binary
2. **Normalize** to the form $1.g_3g_2g_1g_0 \times 2^e$
3. **Adjust exponent** with bias (add 3)
4. **Set** the sign bit

Example: Converting **2.25** into an 8-bit minifloat

Example: Converting 2.4 into an 8-bit minifloat

1. **Convert** integer and fractional parts to binary
   - There isn't enough precision in 8 bits to represent 2.4, so we approximate with the closest possible value
   - 2.4 ~= 2.375 = 10.011
2. **Normalize** to the form $1.g_3g_2g_1g_0 \times 2^e$ *Add 0s to RHS to fit significand if needed*
   - $10.011 = 1.0011 \times 2^1$
3. **Adjust exponent** with bias (add 3)
   - $2^1 \rightarrow \text{bias} \rightarrow 2^{1+3} = 2^4 = 2^{100}$
4. **Set** the sign bit
   - 0 for positive



Float-to-decimal conversion

Steps to convert float to decimal:

1. **Extract** the sign, exponent, and significand
2. **Normalize significand**: restore implied '1.' and remove trailing zeros
3. **De-normalize** to make exponent 0
4. **Convert** the integer and fractional parts to decimals
5. **Set the sign** according to sign bit

Example: Converting **11011100** into a decimal

Example: Converting **11000100** into a decimal

1. **Extract** the sign, exponent, and significand
   - 1 100 0100
2. **Normalize significand**: restore implied ‘1.’ and remove trailing zeros
   - 1.01×2^{4}
3. **De-normalize** to make exponent 0 (Subtract 3 and shift)
   - $1.01 \times 2^{4-3} = 1.01 \times 2^{1} = \mathbf{10.10}$
4. **Convert** the integer and fractional parts to decimals
   - $10.1 = \mathbf{2.5}$
5. **Set the sign** according to sign bit
   - -2.5

Practice on floats!

Steps to convert decimal to floating point:

1. **Convert** integer and fractional parts to binary
2. **Normalize** to the form $1.g_3g_2g_1g_0 \times 2^e$
3. **Adjust exponent** with bias (add 3)
4. **Set** the sign bit

Steps to convert floating point to decimal:

1. **Extract** the sign, exponent, and significand
2. **Normalize significand:** restore implied '1.' and remove trailing zeros
3. **De-normalize** to make exponent 0
4. **Convert** the integer and fractional parts to decimals
5. **Set the sign** according to sign bit

A2: Minifloat

Addition, Multiplication, and Practice

Rounding minifloats

After addition or multiplication, the exact result may not fit in 4 significand bits. When this happens we must round the result:

1. Choose the **nearest representable minifloat**
2. If exactly halfway between two values, **round away from 0**
3. Rounding happens **after** operation

Example: Add **`1.0625 + 1.125`**

Exact result: `1.0625 + 1.125 = 2.1875`

Nearest representable minifloat:
`2.125 and 2.25`

Result:
`2.1875 is exactly halfway ->`
`Round away from 0 ->`
`2.25`

Result:
`2.25`

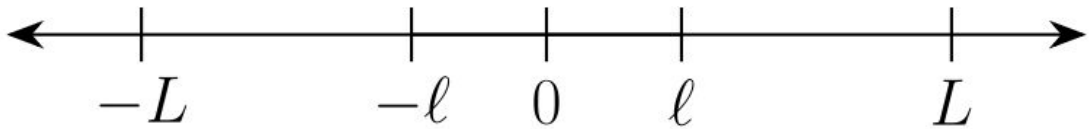
Because the exact result can't be represented by 4 significand bits, we round to the nearest minifloat. Since this value is exactly halfway, we round away from 0.

Zero and clamping

Any minifloat with an exponent of zero represents the number zero:

- `0 000 1010 => +0.0`
- `1 000 0110 => -0.0`
- `0 000 0000 => +0.0`

If an operation's **exact** (not rounded!) result is outside the representable range [– L, L], then we clamp it down (*overflow*); if it's too small to be represented, i.e. in the range (– ℓ, ℓ), we approximate it as zero (*underflow*).



Adding minifloats

Steps to add minifloats:

1. **Adjust the significand** of the number with the larger exponent by shifting it left until both exponents match
2. **Add the significands** together
3. **Recombine and round** the result if necessary
    - Rounding should be to the closest representable number, with ties being broken by choosing the furthest value from 0

Example: Add **1.5** and **0.5**

Beware of accuracy loss due to limited bit size!

Example: Add `1.5` and `0.5`

1. **Adjust the significand** of the number with the larger exponent by shifting it left until both exponents match
   - 1.5 ($1.1 \times 2^0 = \mathbf{11.0 \times 2^{-1}}$) and 0.5 ($\mathbf{1.0 \times 2^{-1}}$)
2. **Add the significands** together
   - 11.0 + 1.0 = 100.0
3. **Recombine and round** the result if necessary
   - $100.0 \times 2^{-1} = 1.0 \times 2^1$ = 0 100 0000 = 2.0_{10}

Multiplying minifloats

Steps to multiply minifloats:

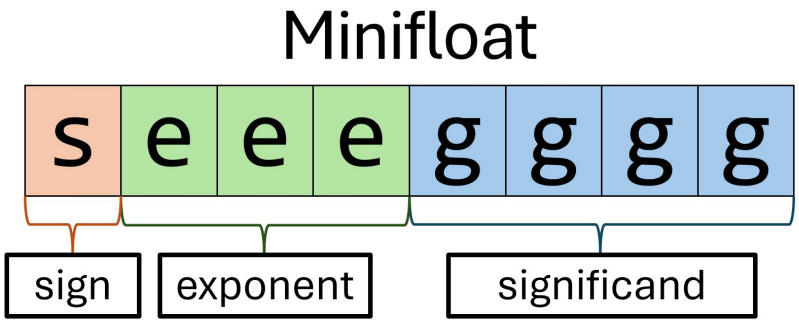
1. **Find the sign** of the product, determined by the sign bits of both minifloats
2. **Add the bit-representation (biased) exponents**, then subtract the bias once to encode the result
   - Reminder: the resulting exponent will be biased, as we essentially subtracted the combined bias of 2 minifloats and then added the bias back to get the exponent for the product's minifloat representation
3. **Multiply the significands** together (restoring the implicit leading 1 to each first)
4. **Recombine and round** the result if necessary
   - Rounding should be to the closest representable number, with ties being broken by choosing the furthest value from 0

Example: Multiply **4.0** and **0.625**

Example: Multiply `4.0` and `0.625`

1. **Find the sign** of the product, determined by the sign bits of both minifloats
   - 0 101 0000 (4) and 0 010 0100 (0.625) both have 0 in the sign bit, so the product also is positive
2. **Add the (biased) exponents**, then subtract the bias once to encode the result
   - 101 + 010 - 011 = 100
3. **Multiply the significands** together (restoring the implicit leading 1 to each first)
   - 10000 * 10100 = 101000000
4. **Recombine and round** if necessary
   - 101000000 falls in the range $[2^8, 2^9)$, so no exponent adjustment is needed here
   - Shift right by 4 to reduce back to a 5-bit significand: 101000000 >> 4 = 10100, remainder 0000 (no rounding needed)
   - Significand 10100 (significand 0100) with exponent 100 gives: 0 100 0100 = $1.25 * 2^1 = 2.5$

Minifloat practice!



1 bit for sign, **3 bits** for exponent, **4 bits** for significand (mantissa)

Bias of 3, special cases for +/-0.0

$$(-1)^s \times 1.g_3g_2g_1g_0 \times 2^{e-3}$$

Assignment 2 Tips

Assignment overview

Part 1: `print_mini`

- Display minifloat in base-10, printing **sign**, **whole number**, and **decimal**

Part 2: `mini_eq` and either `mini_add` or `mini_mul`, with testing

- Implement **equality** and either **addition** or **multiplication** of minifloats, then write out test cases to verify correctness

Assignment tips & guidelines

- If an arithmetic operation would return `0`, you must return exactly **`0000 0000`**
- Round results to the **nearest minifloat value**, and if exactly halfway, **round away from zero**
- Remember to **clamp at the very end!**
- Use utility function ONLY for testing
  - `mini_to_double`
- You cannot use any constants/variables of type **float**, **double**, or **long double**
- **Bitwise operations are CRUCIAL!**

Good luck!