DSC204A: Scaleable Data Systems Spring 2025

3: Digital Representation of Data

Instructor: Umesh Bellur *Scribe: Devana Perupurayil, Lila Horwitz*

1 Introduction

In the previous lecture, we explored how integers can be represented in the binary number system. However, certain limitations arise due to finite memory. The following algorithm illustrates one such issue.

Algorithm 1 Overflow Demonstration

```
Initialize x ← 5
while x ≤ 5,000,000 do
    Print x and x^2
    x ← x × 10
end while
```

Sample Output:
$x = 5, \quad x^2 = 25$
$x = 50, \quad x^2 = 2500$
$x = 500, \quad x^2 = 250000$
$x = 5000, \quad x^2 = 25000000$
$x = 50000, \quad x^2 = -1794967296$
$x = 500000, \quad x^2 = 891896832$
$x = 5000000, \quad x^2 = -1004630016$

Starting from $x = 50000$, the values of x^2 no longer make sense due to **overflow**. This occurs because the program cannot handle numbers beyond a certain limit, as they exceed the maximum value that can be stored in an `int` data type.

2 Signed Values

Some languages offer the option of representing numbers as signed or unsigned. Although signed representations of numbers can store negative values, they can only store integers half the size of unsigned representations.

There are several representations of signed numbers including sign magnitude, one's complement, and two's complement.
Our focus: **Two's complement** representation for negative numbers

1. All positive representations for four bits would range from 0 to 15 but two's complement representation

only ranges from -8 to 7. This is interesting as there is one more negative value than positive value. This happens because 0 is neither negative nor positive.

2. It takes the two's complement number and adds it to the unsigned interpretation.

3. Computer flips all the bits from the unsigned binary representation and then adds one to the least significant bit

Process: Take the leftmost power of two, flip the sign on that value, and then add the unsigned number to it.

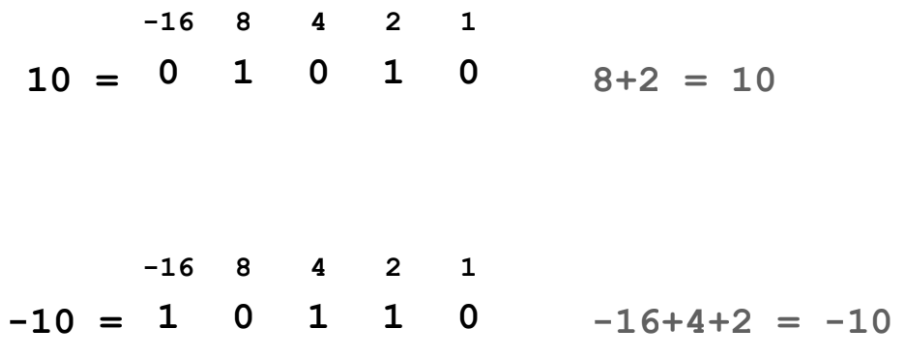


Figure 1: Simple example of two's complement

Using this algorithm, the leftmost bit determines the sign where 1 will represent a negative value and 0 will represent a positive value.

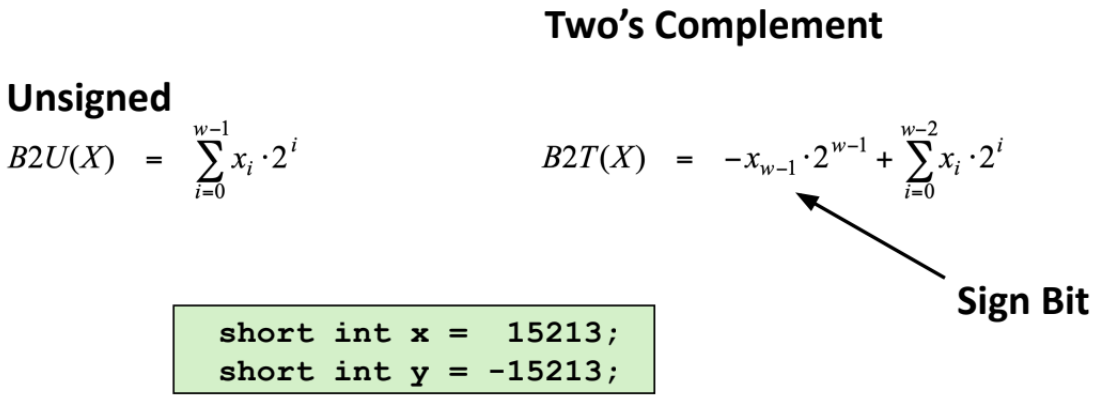


Figure 2: Unsigned vs Signed value algorithms

3 Floating Point Numbers

It is crucial to note that most weights in machine learning algorithms are represented as floating-point numbers. There are several ways to represent floating-point numbers, but the IEEE standard, which is a 32-bit representation (single precision format), is most commonly used. Consequently, double precision is 64 bits, and there is no rounding involved.
Java and C *float* is single while Python *float* is double. Float16 is now common for deep learning parameters.

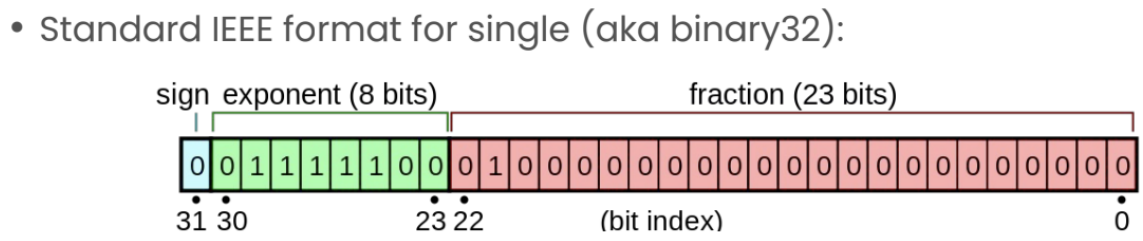


Figure 3: Standard IEEE format for single

A floating-point number consists of a sign bit, an exponent, and a mantissa (fraction as shown in he above figure).
The exponent indicates the power of 10 to which the mantissa is multiplied, and the mantissa represents the fractional part of the number.
The formula to convert binary floating point number to decimal is as follows:

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

For the example given in the image:

$$\begin{aligned}
&\text{sign} = 0 \\
&\text{exponent} = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
&\text{exponent} = 124
\end{aligned}$$

Substituting the above in given formula we get decimal number

$$\begin{aligned}
&= (-1)^0 \times 2^{124-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \\
&= (-1)^0 \times 2^{124-127} \times \left(1 + 1 \times 2^{-2}\right) \\
&= \left(\frac{1}{8}\right) \times \left(1 + \left(\frac{1}{4}\right)\right) \\
&= 0.15625
\end{aligned}$$

Notice that we are subtracting 127 from the exponent in the formula as a bias factor. The choice of 127 as the bias factor is based on the need for easy comparison when exponents are positive. Eight bits have a range of -128 to 127, and a bias factor of 127 is added to ensure that this range remains valid.

- Consider FP numbers $x_1 = s_1E_1M_1$ and $x_2 = s_2E_2M_2$.
- Let $\circ$ denote concatenation of bit fields. We interpret the bit strings $s_1 \circ E_1 \circ M_1$ and $s_2 \circ E_2 \circ M_2$ as sign-magnitude numbers.
- Then $x_1 > x_2$ if and only if $s_1 \circ E_1 \circ M_1 > s_2 \circ E_2 \circ M_2$.
- This allows us to compare floating point numbers using integer hardware.
- The normalized mantissa has one significant digit to the left of the radix point:
  - In binary, this digit must be 1.
  - Mantissa form: $1.m_{22}m_{21}\ldots m_1m_0$
- As a result:
  - Floating point registers and memory do not store the leading 1.
  - The ALU inserts this implicit 1 bit into each incoming operand.
  - It uses it internally during operations, and strips it before writing results back to memory.
  - This provides one extra bit of precision "for free".
  - This is known as the **hidden bit** or **implicit bit**.

3.1 Binary to Decimal Conversion Example

Given the binary number 101.011, we can convert it to decimal by summing powers of 2 corresponding to the positions of the 1s:

$$101.011_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$
$$= 4 + 0 + 1 + 0 + 0.25 + 0.125 = 5.375$$

So, the decimal equivalent of 101.011_2 is 5.375.

3.2 Interpretation of Floating Point

Suppose we want to interpret the following binary number

1100 0001 1111 0000 0000 0000 0000 0000

We will first rearrange it to sEM format, which is the standard IEEE format for 32 bits.

1 10000011 11100000000000000000000

- The sign bit is 1, which indicates a negative number.
- The exponent is 10000011, which is 131 in decimal. As the bias for single precision is 127; the exponent is 131 - 127 = 4.
- The mantissa is 1.111 with the leading 1 implicitly included in IEEE format.

Therefore the value is represented as -1.111×2^4

We can convert into decimal scientific by normalizing the obtained value to get exponent = 0, as it makes the radix change trivial.

$$-1.111 \times 2^4 = -11110 \times 2^0 = -30 \times 10^0 = -3.0 \times 10^1$$

3.3 Decimal Scientific to Floating Point

So we have learned how to represent floating point in binary to decimal. Conversely, we can reverse the process and obtain the original binary representation.

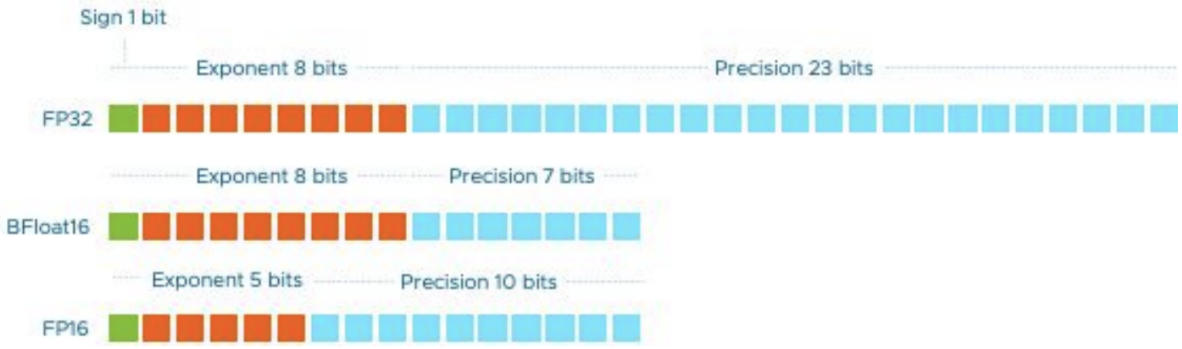
Let's take -91.75 as an example

- Since the number is negative, the sign bit is 1.
- To convert 91.75 to binary:
  - Converting the integer part to binary: $91_{10} = 1011011_2$
  - Converting the fractional part to binary: $0.75 \times 2 = 1.5$(we take the 1); $0.5 \times 2 = 1.0$(we again take the 1). Therefore $0.75_{10} = 0.11_2$
  - Therefore $91.75_{10} = 1011011.11_2$
- Normalizing the binary number we get: 1.01101111×2^6
- Next we want to compute the exponent:
  - From the normalization, the exponent e is 6.
  - However, since the bias for sEM is 127, the actual exponent is $6 + 127 = 133$.
  - Converting it to binary: $133_{10} = 10000101_2$.
- The mantissa is the fractional part 01101111000000000000000, with the leading 1 implied and not stored.

Hence, the final sEM representation of -91.75 is 1 10000101 01101111000000000000000.

3.4 New Magical Float Standards

Every model weight is a floating-point number, and its representation can have significant consequences. For instance, if instead of 8 bits and 23 mantissa bits, the scales are shifted to the right or left, the range of representable numbers changes. The purpose of this discussion is to understand the implications of different weight representations.



With an 8-bit exponent, the largest positive and negative numbers that can be represented are determined by the exponent. For 8 bits, the range is -128 to 127. If we increase the exponent to 12 bits, the range expands to -2048 to 2047.

Moving the sliding scale to the right allows for a larger range of quantities. For example, in astronomy, they work with billions and trillions, which necessitates a wider range. However, if precision is more critical and a smaller range is required, moving the scale to the left is more appropriate. Nevertheless, 32 bits is still a significant amount of space, and companies like Google and NVIDIA are developing new representations, such as BFloat16 and FP16, to address this issue.

Google's motivation for adopting BFloat16 is that precision is not as critical during training machine learning models as the exponents.

4 Contributions

Devana Perupurayil: Sections 1 and 3
Lila Horwitz: Section 2