Data Representation

Microprocessors are the central processing units (CPUs) of computers and other digital devices. They perform operations on data, such as arithmetic, logic, and control operations. To perform these operations, microprocessors must represent data in a specific format. Data representation in microprocessors is typically done using binary number systems, where data is represented as sequences of bits (binary digits), either 0 or 1.

Here are some common data representation formats in microprocessors:

- **Binary numbers**: These are used to represent integers, where each bit position corresponds to a power of 2. For example, the binary number 1011 represents the decimal number 11 $(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)$.
- **Two's complement**: This is a method for representing **signed integers** (positive and negative numbers) in binary form. In this system, the most significant bit (MSB) represents the sign of the number (0 for positive, 1 for negative). To negate a number, you invert all the bits and add 1 to the result.
- **Fixed-point numbers**: Fixed-point representation is used to **represent real numbers** (numbers with a fractional part) by reserving a fixed number of bits for the integer part and a fixed number of bits for the fractional part. For example, a **16-bit fixed-point number** with **8 bits for the integer** part and **8 bits for the fractional part** could represent the number 12.34 as 00001100.01010011.
- **Floating-point numbers**: This is another method for representing real numbers. It uses a **scientific notation-like format**, where a number is represented by a **sign bit**, an **exponent part**, and a **mantissa part**. The **IEEE 754** standard is the most widely used floating-point representation in microprocessors.
- **ASCII and Unicode**: These are character encoding standards used to represent text in microprocessors. ASCII is an older, 7-bit standard, whereas **Unicode is a more modern**, variable-width standard that can represent a much larger range of characters, including those from non-Latin scripts.
- **Bit fields and packed data**: Microprocessors can also represent multiple smaller pieces of data in a single word or register by using bit fields. For example, a 32-bit word could store four 8-bit data items, with each 8-bit item occupying a specific position within the 32-bit word.

These data representation formats are used in various data types and instructions within microprocessors, allowing them to perform complex operations on different types of data.

Floating-point numbers

Floating-point numbers are a way to represent real numbers in computers, allowing for a wide range of values and decimal points. The IEEE 754 standard is the most widely used floating-point representation in microprocessors. It has two common formats: **single-precision** (32-bit) and **double-precision** (64-bit). We will use the **single-precision format in this example**.

A single-precision floating-point number is represented using 32 bits, divided into three parts:

- **Sign bit** (1 bit): Indicates the sign of the number (0 for positive, 1 for negative).
- **Exponent** (8 bits): Represents the exponent in a **biased form**, with a bias value of 127.
- **Mantissa** (23 bits): Represents the fractional part of the number's significand in a normalized form.

Let's represent the number **-12.5 as a single-precision** floating-point number:

1. Convert the integer part (12) to binary: 1100.
2. Convert the fractional part (0.5) to binary: 0.1 (1/2 = 0.5).
3. Combine integer and fractional parts: 1100.1.
4. Normalize the binary number: Move the binary point to the left until there is only one non-zero digit to the left of the binary point. In our case, it becomes 1.1001×2^3.
5. Determine the sign, exponent, and mantissa:
   Sign bit: The number is negative, so the sign bit is 1.
   Exponent: Add the bias (127) to the actual exponent (3). The result is 130, which is 10000010 in binary.
   Mantissa: Take the fractional part of the normalized number (1001) and fill the remaining bits with zeros to make it 23 bits long: 10010000000000000000000.
6. Assemble the floating-point representation: Combine the sign bit, exponent, and mantissa:

1 | 10000010 | 10010000000000000000000

So, -12.5 in single-precision floating-point representation is:

11000001010010000000000000000000

Example 02

Let's represent the number 45.625 as a single-precision floating-point number:

1. Convert the integer part (45) to binary: 101101.
2. Convert the fractional part (0.625) to binary: 0.101 (1/2 + 1/8 = 0.625).
3. Combine integer and fractional parts: 101101.101.
4. Normalize the binary number: Move the binary point to the left until there is only one non-zero digit to the left of the binary point. In our case, it becomes 1.01101101×2^5.
5. Determine the sign, exponent, and mantissa:
   Sign bit: The number is positive, so the sign bit is 0.
   Exponent: Add the bias (127) to the actual exponent (5). The result is 132, which is 10000100 in binary.
   Mantissa: Take the fractional part of the normalized number (01101101) and fill the remaining bits with zeros to make it 23 bits long: 01101101000000000000000.
6. Assemble the floating-point representation: Combine the sign bit, exponent, and mantissa:

0 | 10000100 | 01101101000000000000000

So, 45.625 in single-precision floating-point representation is:

01000010001101101000000000000000

Example 03:

Now let's convert a binary single-precision floating-point number back to a real number. Let's use the following binary representation as an example:

010000100**01101101000000000000000**

1. Disassemble the floating-point representation into the sign bit, exponent, and mantissa:
   Sign bit: 0 (positive)
   Exponent: 10000100 (binary) → 132 (decimal)
   Mantissa: 01101101000000000000000 (binary)
2. Calculate the actual exponent by subtracting the bias (127) from the stored exponent (132):
   Actual exponent: 132 - 127 = 5
3. Determine the significand (mantissa) by adding a leading "1." to the binary mantissa:
   Significand: 1.01101101
4. Multiply the significand by 2 raised to the power of the actual exponent:
   Number: 1.01101101×2^5
   = 1.01101101 x 32
   = 101101.101 (binary)
5. Convert the binary number back to decimal:
   Integer part: 101101 (binary) → 45 (decimal)
   Fractional part: 0.101 (binary) → 0.625 (decimal)
   Real number: 45.625

So, the binary single-precision floating-point representation 01000010001101101000000000000000 corresponds to the real number 45.625.

Question: Consider the single-precision floating-point representation of the decimal number -9.25. Answer the following multiple-choice questions about its various components.

What is the sign bit of the floating-point representation?

a) 0

b) 1

c) 01

d) 10

What is the normalized binary representation of -9.25?

a) -1001.01

b) -1.00101 x 2^3

c) 1.00101 x 2^3

d) 1.00101 x 2^-3

What is the biased exponent of the floating-point representation?

a) 110

b) 111

c) 128

d) 130

What is the binary representation of the biased exponent?

a) 10000010

b) 10000011

c) 10000001

d) 10000000

What is the 23-bit mantissa of the floating-point representation?

a) 00101000000000000000000

b) 00101010000000000000000

c) 00101011000000000000000

d) 00101101000000000000000

Question: Consider the single-precision floating-point representation of the decimal number 18.5. Answer the following multiple-choice questions about its various components.

What is the sign bit of the floating-point representation?

a) 0

b) 1

c) 01

d) 10

What is the normalized binary representation of 18.5?

a) 10010.1

b) 1.00101 x 2^4

c) 1.00101 x 2^5

d) 1.00101 x 2^-4

What is the biased exponent of the floating-point representation?

a) 128

b) 129

c) 130

d) 131

What is the binary representation of the biased exponent?

a) 10000001

b) 10000010

c) 10000011

d) 10000100

What is the 23-bit mantissa of the floating-point representation?

a) 00101000000000000000000

b) 00101010000000000000000

c) 00101011000000000000000

d) 00101101000000000000000

Question: Consider the single-precision floating-point binary representation: 11000001100110000000000000000000. Answer the following multiple-choice questions about its conversion to a real number.

What is the sign of the real number?

a) Positive

b) Negative

c) Zero

d) Undefined

What is the biased exponent in decimal?

a) 128

b) 129

c) 130

d) 131

What is the actual exponent after subtracting the bias (127)?

a) 1

b) 2

c) 3

d) 4

What is the significand (including the leading "1.") based on the 23-bit mantissa?

a) 1.1111

b) 1.1011

c) 1.1001

d) 1.0011

What is the real number represented by the given single-precision floating-point binary representation?

a) -4.5

b) -9

c) -19

d) -6.25

What is the advantage of single point representation?

Single-precision floating-point representation (often referred to as "float" or "single") has several advantages over other number representation systems, such as fixed-point or double-precision floating-point representation. Some of the advantages are:

- **Storage and memory efficiency**: Single-precision floating-point numbers require only 32 bits of storage, making them more memory-efficient than double-precision floating-point numbers, which require 64 bits. This can be advantageous when dealing with large datasets, where using single-precision numbers can save significant memory.
- **Faster computation**: Due to their smaller size, single-precision floating-point numbers can be processed more quickly by some hardware, leading to faster arithmetic operations, such as addition, subtraction, multiplication, and division. This can be particularly beneficial in real-time applications, where quick calculations are essential.
- **Wide dynamic range**: Single-precision floating-point representation allows for a wide range of numbers, both very large and very small, to be represented with reasonable accuracy. This is useful in applications where values can span multiple orders of magnitude.

However, single-precision floating-point representation also has some disadvantages, such as lower precision compared to double-precision floating-point representation. In applications where high precision is required, single-precision may not be suitable, and double-precision or other higher-precision representations might be more appropriate.

Overflow

Overflow, in the context of **signed integer arithmetic**, occurs when an operation produces a result that is too large or too small to be represented within the range of the specific data type. It typically happens when you **add or subtract two numbers of the same sign**, and the result cannot be represented within the valid range. Overflow can lead to unexpected results because **the sign bit is affected**, which changes the number's sign.

Here's an example of overflow with 4-bit signed integers (two's complement representation) where the range is from -8 to 7:

0111 (+7)

0001 (+1)

1000 (-8)

The correct result should be +8, but the answer is -8 due to overflow.

In contrast, a **carry occurs** when an addition operation results in a value that exceeds the maximum value that can be represented by the given number of bits, causing the most significant bit to "carry over" into the next position. Carry, as I described in my previous response, is more applicable to unsigned integer arithmetic.