

Great Ideas in Computing

Allan Borodin

University of Toronto CSC196
Fall 2026

Class 3: September 16 (2026)

Announcements and Todays Agenda

Announcements

- I have to cancel Monday's class. I will probably do a part tutorial/part class on Friday.
- I will post by tonight Assignment A1. It is not yet set up for submission on Markus. It is due September 30 at 2:30 PM.
- I hope to list some exact dates for our guest presentations by tonight. There may or may not be slides for the guest presentations but you are responsible for what is discussed in those guest presentations. But the main reason to be sure to attend, is that these will be presentations by experts on topics of current interest.

Todays Agenda

- The von Neumann model
- Fixed point and Floating Point Representations.

Our first great idea

Our first topic/great idea for discussion: Digitilization and the von Neumann architecture.

- By the mid 40s the first computers were being built. (I am not going to talk say about Babbage and the recent implementation of Babbages 1850 machine.)
- The earliest computers had fixed programs and were not general purpose machines.
- The stored program computer. The conceptual idea is associated with von Neumann who first described the model in a paper "First Draft of a Report on the EDVAC" dated June 30, 1945.
- It is clear that the stored program idea is present in Turing's 1936 paper which we will discuss later. But the Turing model was not meant to be a model of an actual computer.

The von Neumann architecture and digitization

- The basic organization consists of 4 units: memory, I/O, ALU, control. (Some would say that a "bus" to carry signals between units is a fifth component but most say 4 units.) The memory is organized into a list or array of "words", each with its own address.
- Each word w_i (with say address i) is some fixed length string of bits ; i.e., $w = b_{n-1}b_{n-2} \dots b_0$ which as an integer represents $\sum_{j=0}^{n-1} b_j 2^j$. In order to represent negative numbers, we could reserve the high order bit as a sign bit. Thus $b_{n-1}b_{n-2} \dots b_0$ could represent $(-1)^{b_{n-1}} \sum_{j=0}^{n-2} b_j 2^j$. That is, $b_{n-1} = 1$ specifies a negative number. (The choice of 2 is not essential mathematically.). It is also possible to think of a word as a string of "bytes" in which each byte can also be addressable but we will ignore that. **Why not unary, decimal, or some other representation?**

The von Neumann architecture and digitilization

- The basic organization consists of 4 units: memory, I/O, ALU, control. (Some would say that a "bus" to carry signals between units is a fifth component but most say 4 units.) The memory is organized into a list or array of "words", each with its own address.
- Each word w_i (with say address i) is some fixed length string of bits ; i.e., $w = b_{n-1}b_{n-2}, \dots, b_0$ which as an integer represents $\sum_{j=0}^{n-1} b_j 2^j$. In order to represent negative numbers, we could reserve the high order bit as a sign bit. Thus $b_{n-1}b_{n-2} \dots b_0$ could represent $(-1)^{b_{n-1}} \sum_{j=0}^{n-2} b_j 2^j$. That is, $b_{n-1} = 1$ specifies a negative number. (The choice of 2 is not essential mathematically.). It is also possible to think of a word as a string of "bytes" in which each byte can also be addressable but we will ignore that. **Why not unary, decimal, or some other representation?**
- But before we go any further we need to step back and talk about *digitalization and encoding*. **That is, what are we storing in these words?**

The von Neumann architecture and digitilization

- The basic organization consists of 4 units: memory, I/O, ALU, control. (Some would say that a "bus" to carry signals between units is a fifth component but most say 4 units.) The memory is organized into a list or array of "words", each with its own address.
- Each word w_i (with say address i) is some fixed length string of bits ; i.e., $w = b_{n-1}b_{n-2}, \dots, b_0$ which as an integer represents $\sum_{j=0}^{n-1} b_j 2^j$. In order to represent negative numbers, we could reserve the high order bit as a sign bit. Thus $b_{n-1}b_{n-2} \dots b_0$ could represent $(-1)^{b_{n-1}} \sum_{j=0}^{n-2} b_j 2^j$. That is, $b_{n-1} = 1$ specifies a negative number. (The choice of 2 is not essential mathematically.). It is also possible to think of a word as a string of "bytes" in which each byte can also be addressable but we will ignore that. **Why not unary, decimal, or some other representation?**
- But before we go any further we need to step back and talk about *digitalization and encoding*. **That is, what are we storing in these words?** Data and instructions (as part of a program)

Digital computing vs the continuous real world

- The concept of digitalization is one of the most profound concepts that underlies science today.

Digital computing vs the continuous real world

- The concept of digitalization is one of the most profound concepts that underlies science today.
- In the "real world", space and time are continuous A typical computing application might wish to study the trajectory or movement of an object (person, ball, missile, etc). An object is moving through continuous 3 dimensional space in continuous time according to some (hopefully) known laws of motion.
- In contrast, digital computers use discrete finitely represented data and instructions.

Digital computing vs the continuous real world

- The concept of digitalization is one of the most profound concepts that underlies science today.
- In the "real world", space and time are continuous. A typical computing application might wish to study the trajectory or movement of an object (person, ball, missile, etc). An object is moving through continuous 3 dimensional space in continuous time according to some (hopefully) known laws of motion.
- In contrast, digital computers use discrete finitely represented data and instructions.

However, a real value x can be approximated by a rational number. Every integer has a finite representation and every rational can be represented by a pair of integers and hence has a finite representation.

- We will see how we can *approximate* any real number (within some reasonably large range) using *floating point representation*.

Digital computing vs the real world continued

- Furthermore there are some stochastic (i.e., probabilistic) aspects (such as wind and air pollution in the trajectory example) in real world processes.
- In contrast, digital computers are **deterministic**; that is, each “state” of the computer is precisely determined from the previous state.

Digital computing vs the real world continued

- Furthermore there are some stochastic (i.e., probabilistic) aspects (such as wind and air pollution in the trajectory example) in real world processes.
- In contrast, digital computers are **deterministic**; that is, each “state” of the computer is precisely determined from the previous state.

However, we can simulate stochastic events by drawing random variables from some (*pseudo*) *random* distribution.

- Stochastic processes can then be simulated by a discrete deterministic machine. Of course, the conclusions to be made from simulations relative to a specific outcome in the real world will depend on how well we have modelled the system and how unpredictable is the system.
- The early debate: relative benefits of digital computation vs that of analog computers. (In the past, thermostats and other control devices were essentially simple analog computers.) **Why did digitalization win the debate?**

Fixed and Floating point representation

- As mentioned we have to approximate real non rational numbers by fractions. It is easy to see that say every real number can be approximated to arbitrary precision. In particular, every number in the interval $[0, 1]$ can be approximated by a fraction $.b_0b_1b_2 \dots b_n$ where $b_i \in \{0, 1\}$. The more bits better the approximation.
- Some fractions can be represented exactly in such a binary representation (e.g. $1/2$, $1/4$, $3/4$, etc.) while other numbers like $1/10$ and $1/3$ can only be approximated. (Note: One can, of course, represent these numbers exactly as a ratio of two integers.)
- We may need very small or very large numbers but the number of bits in a computer word is fixed (for example, 32 bits) so this limits how big or how small numbers can be. This is not an artifact of the binary representation. The same limitations would apply to any base.
- In a *fixed point representation*, we represent numbers by agreeing to have some fixed number of fractional bits.

Fixed and floating point numbers continued

For example, in an 8 bit fixed point representation $b_7b_6 \dots b_0$, where b_7 is the sign bit, we can agree that the two lower order bits b_1 and b_0 are the fractional parts. Then the decimal number -18.5 would be represented exactly by 11001010 and -18.25 would be represented exactly by 11001001.

Note that in a pure integer or fixed point representation, the sizes of the smallest and largest numbers are severely restricted. For a 32 bit word with a sign bit, the largest number is $2^{31} - 1 \approx (10)^9$ (i.e. approximately one billion). And every bit of precision we use for the fractional part decreases the range of numbers representable by approximately a factor of 2.

- The common solution to provide a large range as well as providing good precision is to use *floating point representation*.

Simplifying the discussion of floating point numbers

NOTE: I am going to first simplify the discussion and ignore the special meaning when all the bits are 0's or 1's (which in the IEEE standard are used for special numbers such as ∞).

Then I will simplify the representation of the exponent field.

As in practice, we will continue here to use binary (i.e., base 2) representation but again any base (e.g., base 10) could be used.

Floating point numbers continued

A floating point number uses the following representation (where I am using # just for clarity) as the bits would all be consecutive.

$$s \# e_{k-1} e_{k-2} \dots e_0 \# b_{\ell-1} \dots b_1 b_0$$

Here the bit s determines the sign (i.e. $+$ or $-$) of the number.

The e_i bits represent the unbiased exponent E with value $E \in [0, 2^k - 1]$.
The biased exponent $E' = E - (2^{k-1} - 1)$.

The b_i bits represent the significand (i.e, the significant bits)

The number being represented is

$$(-1)^s \cdot 2^{E'} \cdot (1.b_{\ell-1} \dots b_1 b_0)_2 = (-1)^s \cdot 2^{E'} \cdot (1 + \sum_{i=1}^{\ell} b_{\ell-i} 2^{-i}).$$

NOTE: There is an implicit “1” preceding the implicit binary point.

For an 8 bit word, with say a $k = 4$ bit exponent field and 3 bit significand, the integer -15 would be represented as

$$11010111 = -1 \cdot 2^3 \cdot (1 + \frac{7}{8}) \text{ since } E' = E - (2^{k-1} - 1) = 10 - 7 = 3.$$

Floating point numbers continued

- The IEEE standard for a 32 bit *single precision* number uses 8 bits for the exponent and 23 bits for the significand (and therefore 24 bits of precision counting the implicit “1” . .
- There are also double (and multiple) precision numbers where a double precision number would occupy two 32 bit words.
- History: According to Wikipedia, Leonardo Torres y Quevedo used floating point numbers in his design of Babbage’s Analytical Engine. See also the reference to Konrad Zuse who designed a computer in 1938 and later versions in 1941 who using floating point numbers.
- It is interesting to note that von Neumann argued for fixed point numbers (and not floating point) in the design for an Institute of Advanced Study machine.
- It is important to note that an algorithm designer (usually) doesn’t need to know the specifics of the fixed point and floating point representations. but just needs to know the commands for specifying the type (i.e. integer using fixed point or “real” using floating point) of the number.

An example for the simplified representation

Lets define a slightly simplified version as follows:

Consider again, a number x represented by

$$s \# e_{k-1} e_{k-2} \dots e_0 \# b_{\ell-1} \dots b_1 b_0$$

Now let e_{k-1} be the sign of the exponent so that

$$x = (-1)^s \cdot 2^{E'} \cdot (1 + \sum_{i=1}^{\ell} b_{i-1} 2^{-i}) \text{ where now } E' = (-1)^{e_{k-1}} \cdot E.$$

That is we are now letting the leading bit of the exponent be the sign of the exponent rather than using a biased exponent.

Let's again consider an 8 bit number with a 4 bit exponent field and a 3 bit significand.

Now the floating point word 11010111 is equal to

$$-1 \cdot 2^{-2} \cdot (1 + \frac{7}{8}) = -\frac{1}{4} \cdot \frac{15}{8} = -\frac{15}{32}.$$

And the floating point word 10101111 is equal to $-2^5 \cdot \frac{15}{8} = -60$ in decimal.