

# Computer Memory: Bits and Bytes

---

This handout was written by Nick Parlante and Julie Zelenski.

To begin, we are going to take a glimpse into the inner workings of a computer. The goal is that by seeing the basics of how the computer and compiler cooperate, you will better understand how language features work.

## Basic Architecture

Almost all modern computers today are designed using the Von Neumann architecture of 1954. In the Von Neumann architecture, the computer is divided into a Central Processing Unit, CPU, and memory. The CPU contains all the computational power of the system while the memory stores the program code and data for the program. Von Neumann's innovation was that memory could be used to store both the program instructions and the program's data. The instructions that constitute a program are laid out in consecutive words in memory, ready to be executed in order. The CPU runs in a "fetch-execute" cycle where it retrieves and executes program instructions from memory. The CPU executes the current instruction, and then fetches and executes the next instruction, and so on. The sort of instructions the CPU executes are detailed later in this handout.

## Memory

The smallest unit of memory is the "bit". A bit can be in one of two states— on vs. off, or alternately, 1 vs. 0. Technically any object that can have two distinct states can remember one bit of information. This has been done with magnets, gear wheels, and tinker toys, but almost all computers use little transistor circuits called "flip-flops" to store bits. The flip-flop circuit has the property that it can be set to be in one of two states, and will stay in that state and can be read until it is reset.

Most computers don't work with bits individually, but instead group eight bits together to form a "byte". Each byte maintains one eight-bit pattern. A group of  $N$  bits can be arranged in  $2^N$  different patterns. So a byte can hold  $2^8 = 256$  different patterns. The memory system as a whole is organized as a large array of bytes. Every byte has its own "address" which is like its index in the array. Strictly speaking, a program can interpret a bit pattern any way it chooses. By far the most common interpretation is to consider the bit pattern to represent a number written in base 2. In this case, the 256 patterns a byte can hold map to the numbers 0..255.

The CPU can retrieve or set the value of any byte in memory. The CPU identifies each byte by its address. For this class, we will write memory operations like array operations, so something like the notation `Mem[20]=34` sets the value of memory at address 20 to the value 34. The byte is sometimes defined as the "smallest addressable

unit" of memory. Most computers also support reading and writing larger units of memory— 2 byte "half-words" (sometimes known as a "short" word) and 4 byte "words". Half-words and words span consecutive bytes in memory. By convention the address of any multiple-byte thing is the address of its lowest byte— its "base-address". So the 4-byte word at address 400 is composed of bytes 400, 401, 402, and 403. Most computers restrict half-word and word accesses to be "aligned"— a half-word must start at an even address and a word must start at an address that is a multiple of 4.

Thankfully, most programming languages shield the programmer from the detail of bytes and addresses. Instead, programming languages provide the abstractions of *variable* and *type* for the programmer to manipulate. In the simplest scheme, a variable is implemented in the computer as a collection of bytes of memory. The type of the variable determines the number of bytes required. Here are the basic types and their sizes:

**Character**— The ASCII code defines 128 characters and a mapping of those characters onto the numbers 0..127. For example, the letter 'A' is assigned 65 in the ASCII table. Expressed in binary, that's  $2^6 + 2^0$  ( $64 + 1$ ), and so the byte that represents 'A' is:

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

All standard ASCII characters have zero in the uppermost bit (the "most significant" bit) since they only span the range 0..127. Some computers use an extended character set which adds characters like é and ö using the previously unused numbers in the range 128..255. Some systems use the 8th bit to store parity information so a modem for example, can notice if a byte has been corrupted. Some memory hardware systems keep a 9th parity bit for every byte so the hardware can notice if memory is getting flaky— avoiding troublesome HAL 9000 type problems.

**Short Integer**— 2 bytes or 16 bits. 16 bits provide  $2^{16} = 65536$  patterns. This number is known as "64k", where 1 "k" of something is  $2^{10} = 1024$ . For non-negative numbers these patterns map to the numbers 0..65535. For example, consider the 2-byte short representing the value 65. It has the same binary bit pattern as the 'A' above in the lowermost (or "least significant") byte and zeros in the most significant byte. However, if a short occupies the 2 bytes at addresses 450 and 451, is the most significant byte at the lower or higher numbered address? Unfortunately, this is not standardized. Systems that are *big-endian* (Motorola 68K, PowerPC, Sparc, most RISC chips) store the most-significant byte at the lower address, so 65 as a short would look like this:

0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A *little-endian* (Intel x86, Pentium) system arranges the bytes in the opposite order, so it would look like this:

0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



**Records—** The size of a record is equal to at least the sum of the sizes of its component fields. The record is laid out by allocating the components sequentially in a contiguous block, working from low memory to high. Sometimes a compiler will add invisible “pad” fields in a record to comply with processor alignment restrictions. For the purposes of this class, you can ignore pad bytes unless explicitly mentioned in the problem.

**Arrays—** The size of an array is at least equal to the size of each element multiplied by the number of components. The elements in the array are laid out consecutively starting with the first element and working from low memory to high. Given the base address of the array, the compiler can generate constant-time code to figure the address of any element. As with records, there may be pad bytes added to the size of each element to comply with alignment restrictions.

**Pointer—** A pointer is an address. The size of the pointer depends on the range of addresses on the machine. Currently almost all machines use 4 bytes to store an address, creating a 4GB addressable range. There is actually very little distinction between a pointer and a 4 byte unsigned integer. They both just store integers— the difference is in whether the number is *interpreted* as a number or as an address.

**Instruction—** Machine instructions themselves are also encoded using bit patterns, most often using the same 4-byte native word size. The different bits in the instruction encoding indicate things such as what type of instruction it is (load, store, multiply, etc.) and the registers involved. We'll blow this concept off until later.