Prof. Dr. rer. nat. habil. Martin O. Steinhauser
National Defense Academy, Yokosuka, Japan

Frankfurt University, Germany
October 2024

Computational Materials Science with Atomistic and Coarse-Grained Methods

Handout 3

Bits and Bytes in Memory

1 Introduction

In this handout, we are going to take a glimpse into the inner workings of a computer. The goal of this exercise is that by seeing the basics of how the computer and compiler cooperate, you will better understand how language features (not only in pure C!) actually work. And this, in turn, will turn you into a much better scientific programmer.

2 Basic Architecture of Computers

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. Before that time, "programming" meant, that cables were put together in the hardware. The "program" was not part of memory. 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.

3 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 switch between 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. The byte is sometimes defined as the "smallest addressable unit" of memory. Most computers also support reading and writing larger units of memory, for example 2 byte "half-word" (in computer languages often called a "*short*" word) and 4 byte "words", often known as "*int*" or "*float*" 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 600 is composed of bytes 600, 601, 602, and 603. Most computers restrict half-word and word accesses to be "aligned", i.e. a half-word must start at an even address and a word must start at an address that is a multiple of 4.

Don't worry! 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.

4 Basic Data Types in C

Here are the basic types and their typical sizes on most systems:

4.1 Character (*char* in C – one byte)

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: All standard ASCII characters have zero in the

0	1	0	0	0	0	0	1

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.

4.2 Short Integer (*short* in C – 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 nonnegative numbers these patterns map to the numbers 0...65535. For example, consider the 2-bye 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[1]. However, if a short occupies the 2 bytes at addresses 650 and 651, 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

[1] The most significant bit (MSB) is the bit in a multiple-bit binary number with the largest value. This is usually the bit farthest to the left, or the bit transmitted first in a sequence. For example, in the binary number 1000, the MSB is 1, and in the binary number 0111, the MSB is 0. The most significant byte is the byte in a multiple-byte word with the largest value. As with bits, the most significant byte is normally the byte farthest to the left, or the byte transmitted first in a sequence.

A so-called little-endian[2] (Intel x86, Pentium) system arranges the bytes in the opposite order, so it would look like this: This means when exchanging data through files or over a network between

0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0

different endian machines, there is often a substantial amount of “byteswapping“ required to rearrange the data, because of the lack of a standard.

To get negative numbers, there‘s a slightly different system which interprets the patterns as the numbers $-32768...32767$, with one bit reserved for storing sign information. The “sign bit“ is usually the most significant bit of the most significant byte.

4.3 Long Integer (*long* in C – 4 bytes or 32 bits)

32 bits provide $2^{32} = 4294967296$ patterns. Most programmers just remember this numbers as “about 4 billion“. The signed representation can deal with numbers in the approximate range ± 2 billion. 4 bytes is the contemporary default size for an integer, also known as a “word“. The representation of a *long* is just like that of a *short*. On a big-endian machine, the four bytes are arranged in order of most significant to least and vice versa for a little-endian machine.

4.4 Floating Point (*float* in C – $4, 8, 10,$ or 12 bytes)

Almost all computers use the standard IEEE[3] representation for floating point numbers that is a system much more complex than the scheme for integers. The important thing to note is that the bit pattern for the floating point number 1.0 is not the same as the pattern for the integer 1. For example, 65 expressed as a floating point value has this bit pattern:

0	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Interpreted as a big-endian long, this pattern would be 1079001088, which is not at all the integer 65. IEEE floats are in a form of scientific notation. A 4-byte float uses 23 bits for the mantissa, 8 bits for the exponent, and 1 bit for the sign. It can represent values as large as 3×10^{38} and as small 1×10^{-38} (both positive and negative). Clearly there are many more floating point numbers in that range than the number of distinct patterns that can be represented with a 4-byte float (which is 4 billion), so floats are necessarily approximate. A floating point value is usually only accurate up to about 6 decimal digits of precision, and any digits after that are suspect. 8 byte doubles range up to around 10^{308} and have 15 digits of reliable precision. Floating point operations have usually been considerably slower than the corresponding integer operations. Some processors have a special hardware Floating Point Unit, FPU, that substantially speeds up floating point operations. With separate integer and floating point processing units, it is often possible that an integer and a floating point computation can proceed in parallel to an extent. This has clouded the old ïnteger is fasterrule greatly; on some computers a mix of 2 integer and 2 floating point operations may be faster than 4 integer operations. So, if you are really concerned about optimizing a bit of code, then you'll need to run some tests. Most of the time you should just write the code however you like best and let the compiler deal with the optimization issues.

[2] Both the big and little endian names originate from *Gulliver‘s Travels*. In the novel, there were two political factions, the “Big Endians“ and the “Little Endians“. The difference between the two was how they broke their eggs. The big endians choose to do so on the large end of the egg, while the little endians on the small end.

[3] IEEE: Institute of Electrical and Electronic Engineers.

4.5 Arrays (e.g. *int* arrayExample[5]; in C)

The size of an array is at least equal to the size of each element (here: int, i.e. 4 bytes) multiplied by the number of components (here: 5). 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.

4.6 Pointer(e.g. *int* *pointerExample; in C)

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.

4.7 Instructions (Assembly, machine code)

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 are not dealing with this most elementary level of programming.