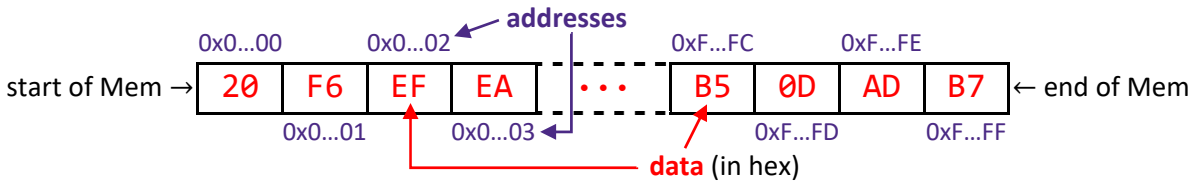
CSE 351 Lecture 2 – Memory & Data I

Memory and Addresses

The central processing unit (CPU) is the piece of hardware that executes computer instructions. We define the ***word size*** to be the fixed width of data (*i.e.* number of bits) used by the CPU in the execution of instructions. ***Memory*** is a separate piece of hardware that acts as a local pool of data storage for the CPU during the execution of instructions.

From the perspective of a programmer, memory is ***byte-oriented*** in that we treat memory as a large array of bytes. This means that a byte (8 bits = 2 hex digits) is the smallest unit of data transfer possible to and from memory, though data often span multiple bytes.

Each individual byte in memory is given a unique, numerical ***address***, which we start numbering from 0 and mostly commonly represent in hex. Memory is a fixed size (*e.g.* 8 gigabytes) based on your installed hardware so we treat addresses as a *fixed-length* quantity (*i.e.* all addresses are represented using the same number of bits), which we set to be the *word size* by convention. The set of all addresses in memory are defined as the ***address space***.



Data in Memory

In a computer, data are moved and manipulated in *fixed-length chunks*, meaning that leading zeros may be required to represent the full extent of the data. Within each chunk, we refer to the left-most bit as the ***most-significant bit (MSB)*** and the right-most bit as the ***least-significant bit (LSB)***. For example, if we store the number 5 in two bytes of memory, we actually store the following bits:

$$MSB \rightarrow \ 00000000\ 00000101_2 \ \leftarrow LSB$$

Multibyte data in memory span multiple addresses. Instead of having to refer to a range of addresses each time, by convention *the address of any chunk of memory is given by the address of the first byte*. This means that we can uniquely specify any piece of data in memory via (1) its address and (2) its size.

For multibyte data, consecutive bytes should be stored in consecutive addresses, but there are two different orderings that preserve this rule; we refer to the choice of ordering as *endianness*. In a ***big-endian*** machine, the least significant byte of data is stored at the *highest* address, while in a ***little-endian*** machine, the least significant byte of data is stored at the *lowest* address. For example, if we store the 4-byte data 0xA1B2C3D4 at address 0x100:

Pointers

Pointers are special variables that store addresses. Since the length of an address is the *word size*, the size of a pointer is also the *word size* (*e.g.* 8 bytes in a 64-bit machine). We say that the 8 bytes stored in a pointer "points" to the data stored at that address, but the pointer data type must also encode size information, otherwise we haven't completely specified the piece of data in memory that we want. It is important to try to keep the distinction clear between (1) the address of a chunk of data, (2) the size of the chunk of data, and (3) the chunk of data itself.

We will learn how to use pointers in C in the next lecture.