CSE 2421

Endianness and byte order

Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*

- Chapter 2, Sections 2.1.3

Order in which bytes are written

- For humans, it is intuitive to write the most significant byte of a multi-byte number on the left, and then write the remaining bytes to the right.
- For example, suppose we represent $(550)_{10}$ as a 32 bit binary value:

0000 0000 0000 0000 0000 0010 0010 0110

(spaces provided for clarity)

- In hex, this value can be written as:

00000226

Order of Bytes in Hardware Storage

•In some machines (e.g., SPARC, Power, PowerPC, MIPS), the bytes are stored in memory in the order to which we are accustomed, that is, with the most significant byte at the first address in memory, and with the remaining bytes stored in order till the least significant byte, which is stored at the highest address.

•Such machines are called “Big-Endian”, because the most significant byte is stored first in memory.

• Internet Protocol – IP – gives us “Network ordering” which is big-endian

•For example, if the 32 bit integer on the previous slide is stored in RAM starting at address 1000 hex, the bytes, and the addresses at which they are stored, are as shown on the next slide.

Big-Endian storage of hex 00000226

Address	**Byte**
1000	00
1001	00
1002	02
1003	26

Little-Endian storage

•In other machines (Intel, including x86, and ARM processors, e.g.), however, the bytes are stored in *the opposite order* in memory; that is, **the least significant byte** is stored at the first address in memory, and the next most significant byte at the next address, and so on, in order, till **the most significant byte**, which is stored at the highest address.

•Such machines are called “Little-Endian”, because the least significant byte is stored first in memory.

•If the 32 bit integer considered above, hex 00000226, is stored in RAM starting at address 1000, the bytes, and the addresses at which they are stored in a Little-Endian machine, are as shown on the next slide.

Little-Endian storage of hex 00000226

Address	Byte
1000	26
1001	02
1002	00
1003	00

Conclusion – Big-Endian and Little-Endian

•Both Big-Endian and Little-Endian architectures work, but when we do assembly language programming, for example, we may at times have to be aware of which type of architecture the machine we are writing code for uses.

•Also, be careful not to get confused about what is changing here. Endian-ness relates to the order in which ***the bytes*** are stored in hardware, and not the order in which ***the bits*** are stored. **Bit order** for a given byte (most significant, least significant, or some byte in between) is ***the same*** for both types of machines.

•Many processors, especially RISC processors, are now configurable in terms of Endian-ness.

•Because both types of architecture are used in the real world, you have to be familiar with the difference.

Counting in Hex

- When examining hex numbers, as we will often do with addresses later, we will need to be able to do basic arithmetic operations, such as counting up (addition) or counting down (subtraction).
- The principles, of course, are exactly the same as in decimal, but the values of the digits are different (0-F in hex instead of 0-9 in decimal).
- Let’s look at a few examples.

Counting in Binary and Hex - Example

•Suppose the address of the top of the stack at the beginning of some function is 8000 hex.

•Suppose that a 8 byte value is pushed onto the stack after the processor starts executing the function’s code.

•When the value is pushed onto the stack in that function, the address stored in the stack pointer will change by 8 bytes, to get the address where the value to be pushed will be stored.

•Recall that the stack grows downward in RAM (the addresses decrease as we push more and more values onto the stack).

•Therefore, to get the address where the first value will be pushed onto the stack, we need 8000 hex – 8.

Counting in Binary and Hex – Example cont.

•How do we subtract 8 from 8000 hex?

•Reduce the least significant digit by 8; there must be a "borrow" from the next most significant digit, so we must reduce the value of the remaining digits by 1.

•Thus, the result is 7FF8, and this is the address at which the first value pushed onto the stack will be stored (more on how the stack pointer changes later).

•Compare this with subtracting 8 from decimal 8000, and it's not difficult to see how it works.