Byte Ordering

Memory is byte-addressible, but the CPU manipulates multi-byte values, such as 4-byte integer or 8-byte double-precision floating point values.

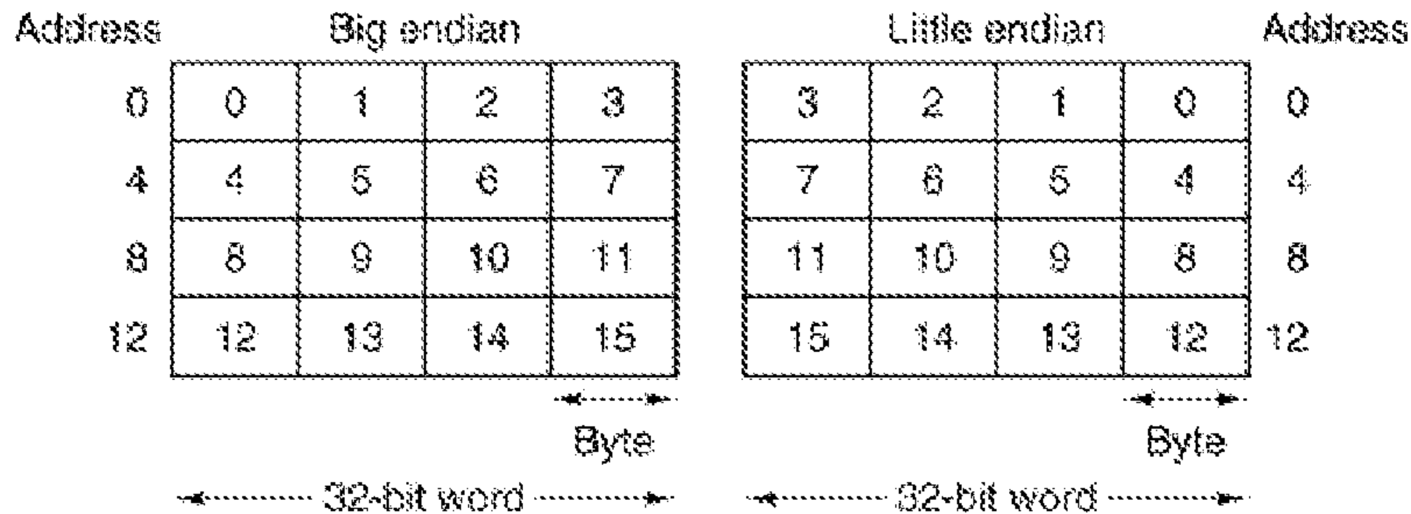
Question: When a 4-byte integer is stored in memory, which of the four bytes is stored in the lowest-numbered address? the highest-numbered address?

Answer: The choice is made by the computer designers, and varies among architectures. Two standard choices:

- *Little-endian:* The least-significant byte goes at the lowest address.
- *Big-endian:* The most-significant byte goes at the lowest address.

(Endianness)

Big-Endian and Little-Endian Byte Orderings



(a) Big endian memory
(SPARC or IBM mainframe)

(b) Little endian memory
(Intel)

Endianness: Why Do You Care?

If you write a multi-byte quantity into memory and then read it back one byte at a time, you will get different results on different architectures.

Example:

- You write the integer value 260 (a 4-byte quantity) at address 100.

  00000000 00000000 00000001 00000100

- You read back bytes from locations, 100, 101, 102, 103:
  - *little-endian:* You get 4, 1, 0, 0.
  - *big-endian:* You get 0, 0, 1, 4.

The time this makes a difference is when you are reading or writing multi-byte quantities from memory to external storage or a network.

Alignment

Another issue related to multi-byte quantities is that the CPU will only read or write them to memory in an *aligned* fashion.

- 1-byte values can be read or written to any address.
- 2-byte values can only be read or written to *even-numbered* addresses.
- 4-byte values can only be read or written to addresses that are *multiples of* 4.
- *etc.*

If you try to read or write a multi-byte quantity to a non-aligned address, an exception will occur.

Encoding Characters

- The **ASCII** character set or various **Extended ASCII** character sets use one byte per character, for a maximum of 256 distinct characters. Many *legacy files* are encoded in this fashion.
- The **Unicode** standard has over 1 million *code points* (currently `U+0000` to `U+10FFFF`), of which 109,449 are currently used (Unicode 6.0).

Encoding Text Strings

Text is typically encoded in memory by placing *character codes* in successive locations.

- The Java `char` data type consists of 16-bit unsigned 16-bit integers which can directly represent the first 65,536 Unicode code points.
- Strings in Java use `UTF-16` encoding, which uses 16-bit codes placed consecutively in memory. Each Unicode code point is represented by either one or two 16-bit codes.
  - *Byte order marks* are used at the beginning of UTF-16 text files, to detect endianness.
- `UTF-8` is a scheme for encoding Unicode as a a sequence of bytes, which is backward compatible with ASCII. It is now commonly used in communication.