Big and Little Endian

Basic Memory Concepts

In order to understand the concept of big and little endian, you need to understand memory. Fortunately, we only need a very high level abstraction for memory. You don't need to know all the little details of how memory works.

All you need to know about memory is that it's one large array. But one large array containing what? The array contains *bytes*. In computer organization, people don't use the term "index" to refer to the array locations. Instead, we use the term "address". "address" and "index" mean the same, so if you're getting confused, just think of "address" as "index".

Each address stores one element of the memory "array". Each element is typically one byte. There are some memory configurations where each address stores something besides a byte. For example, you might store a nybble or a bit. However, those are exceedingly rare, so for now, we make the broad assumption that all memory addresses store bytes.

I will sometimes say that memory is *byte-addresseable*. This is just a fancy way of saying that each address stores one byte. If I say memory is *nybble-addressable*, that means each memory address stores one nybble.

Storing Words in Memory

We've defined a word to mean 32 bits. This is the same as 4 bytes. Integers, single-precision floating point numbers, and MIPS instructions are all 32 bits long. How can we store these values into memory? After all, each memory address can store a single byte, not 4 bytes.

The answer is simple. We split the 32 bit quantity into 4 bytes. For example, suppose we have a 32 bit quantity, written as $90AB12CD_{16}$, which is hexadecimal. Since each hex digit is 4 bits, we need 8 hex digits to represent the 32 bit value.

So, the 4 bytes are: 90, AB, 12, CD where each byte requires 2 hex digits.

It turns out there are two ways to store this in memory.

Big Endian

In big endian, you store the most significant byte in the smallest address. Here's how it would look:

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Little Endian

In little endian, you store the *least* significant byte in the smallest address. Here's how it would look:

Address	Value
1000	CD
1001	12
1002	AB
1003	90

Notice that this is in the reverse order compared to big endian. To remember which is which, recall whether the least significant byte is stored first (thus, little endian) or the most significant byte is stored first (thus, big endian).

Notice I used "byte" instead of "bit" in least significant bit. I sometimes abbreciated this as LSB and MSB, with the 'B' capitalized to refer to byte and use the lowercase 'b' to represent bit. I only refer to most and least significant byte when it comes to endianness.