Why I Prefer Little Endian

by Thomas Walker Lynch

2022-02-18

Abstract

Positive place value represented integers are used by computers for instruction counters, array indexes, and memory addresses, among other things. This makes them fundamental to computing. Place value representation and the accompanying arithmetic came to Europe from the Middle East during the middle ages; however, it arrived with a wrinkle. Arabic writing for text is right-to-left, while European writing is left-to-right. Translators chose to transcribe numbers 'as is' without reversing their digit order, thus in effect they reversed the order of digits in numbers with respect to the character scanning order used by the reader. As a direct consequence of this writing convention, when contemporary computer architects of European cultural heritage read computer dumps of memory, they find it more natural for numbers to print most significant digit first. However, it is more consistent to have the weights of digits go from low to high at the same time addresses go from low to high, so other computer architects prefer for the least significant digit to come first. The debate goes on from there. This has lead to some subtle but profound compatibility issues among computers, standards, and even documentation.

Digital Computing

The first computing machines which performed discrete state computation made use of ten symbols for digits, and thus had ten state logic. Examples include Pascal's calculator, Babbage's machine, the many mechanical calculators that came after, the Mark computers, and ENIAC. To implement ten state logic these machines used actual gears with ten index marks enscribed on them, or as In the case of ENIAC, a circular shift register of ten vacuum tubes. Hence the term *digital* initially carried with it the sense of both being discrete, and of making use of ten state logic.

The Atanasoff-Berry machine that was developed in the late 1930s used Shannon's two state switch logic and binary computation instead of ten index mark gear logic or circular shift registers made from vacuum tubes. Because there are two states in switch logic it is maximually efficient to use a base two number system for arithmetic. In such a system each binary digit will have either the value zero, or the value one. Of course, it has become commonplace to shorten the term *binary digit* to just*bit*.

By the 1950s almost all discrete state computers used two state switch logic, yet the term *digital* continued to be used to describe them, which has lead to some curious naming conventions. For example, the company *Digital Equipment Corporation* which was founded in the 1960s only built two state switch logic computers. As another example a person who studies *digital electronics* will probably never see any wheels or circular shift registers used for computing.

Though the term *digital* continues to refer to computing with discrete states, the term has lost the connotation that there must be *ten* of those states. Today computing based on ten state digits, such as we see on hand held calculators, is known as *decimal* computing. If we had a machine that made use of 256 state digits it would be neither decimal nor binary. Because we do not have a special name for the number 256, we would say that such a computer uses *base 256* digits and that it is a base 256 computer. Decimal computing, binary computing, and even base 256 computing are all examples of digital computing.

As computer implementations moved from ten state logic to two state switch logic it was hard to move away from base ten number systems. This was particularly true in business applications where users want to see dollars and cent results that match computations done by hand, even when fractions and rounding were involved. Hence a IBM came up with a scheme whereby groups of 4 bits could be used to represent the decimal digits, this code is called *binary coded decimal* or just *BCD.* Even today IBM machines may be found where the machine instructions themselves expect their operands to be encoded according to BCD, and thus by definition, at least in one aspect, such machines have a decimal architecture.

Most computer users will never see a memory dump. Instead they will see the output of print functions. Print functions will print in decimal by default Independent of the internal representation used for numbers. In contemporary computing, the time it takes to convert a binary number to a decimal number is typically negligible compared to the time it takes to do the rest of the computation, so in the overall picture, binary is by far the more efficient encoding for computing on machines with two state switch logic.

Place Value Numbers

Lets consider a special set of vectors. For each vector in this special set, each component of said vector will be a single decimal digit. An example of such a vector written using a conventional vector notation is <7, 8, 9>. This vector has as its index zero component the digit 7, has as its index one component the digit 8, and of course has as its index two component the digit 9.

We may interpret such a vector as a place value number by using this function:

$$\sum_{i=0}^{n} \boldsymbol{a}_i \cdot ten^i \quad \text{; \textit{gives meaning to a decimal place value represented number}}$$

Here $\boldsymbol{a}$ is a symbol standing for the digit vector, while $\boldsymbol{a}_i$ is *i* th component of said vector. $\boldsymbol{a}_i$ is also known as the *i* th digit. Here the *base* is the number *ten*. It is fortunate that *ten* is so well known that it has a name, because otherwise we might have been tempted to do what many other authors have done, and to write the base while using the very same representation that we are trying to define.

Although this function gives numeric meaning to our digit vectors, actually performing the suggested computation is pointless, because the result would be a number, and we would have to represent that number, and that representation would be the very same vector that was input into the function in the first place.

In many contexts we may drop the decoration we have been using to signify a vector without causing ambiguity. It is conventional that when dropping the decoration that we also reverse the order of the components. So the example vector from the prior paragraph, <7, 8, 9>, becomes 987. We call this writing convention *most-significant-digit-first*. This is because the first digit we write, the 9, is the one that is given the largest weight in the representation sum. Just to make sure we are on the same page, please note that the number 987 is only 13 away from a thousand.

We usually continue on from this definition to talk about place value numbers in *normal form*. For normal form we consider that zero has a special symbol of its own. In manual writing we represent zero in normal form with a single digit written as a ‘0’. If we see something like two or more zeros in a row, e.g. '00', it looks peculiar, because it is not in normal form.

For non-zero numbers in normal form, we require that the index n be the largest index for which $\boldsymbol{a}_n$ is not zero. The digit $\boldsymbol{a}_n$ is called the *most-significant-digit* because it is the digit with greatest weight in the representation function. Then $\boldsymbol{a}_0$ is called the *least-*

significant-digit, because it is the digit with the smallest weight. The scaling weight for the least significant digit is always one, so $\boldsymbol{a}_0$ is also said to be in the *one's place.* Although for such a non-zero number $\boldsymbol{a}_n$ can not be zero, $\boldsymbol{a}_0$ might be zero. It is possible that the index n turns out to be zero, in which case the most-significant-digit will also be the least-significant-digit . Such a number would have only one digit when written in normal form, that digit would be in the one's place, and because we said that the number has a most significant digit, it can not be zero.

Typically in computing we do not use normal form for integers. Instead we use a fixed number of digits without the possibility of there being a *blank*. In this fixed length representation the digits that appear with more significance than the most significant digit will be zero. These zeros are are said to be *leading zeros,* although this term has build into it an assumption that we are using most-significant-digit-first order. In least-significant-digit-first order these zeros that follow the most-significant digit will be trailing zeros in that they come after the other other digits of the number relative to the scanning order. Yet, by convention they are still called *leading zeros*.

Scanning-Order and Digit-Order

Had I written this paper in Arabic, the text would have been written right-to-left. When writing from right-to-left our initial vector with its decorations will be reversed from the prior example, but as per our more than thousand year old convention, the digits in the number would not be reversed, so we would get something like this:

.987 ot seifilpmis <9 ,8 ,7> rotcev eht ,tfel ot thgir morf gnitirw nehW

Now imagine, we start with this, as the early translators did, but instead of making an exception for numbers as they did, we literally reverse the entire string. Then our vector becomes <7, 8, 9> and without decoration it becomes 789. In both the scan order of the digits is the same, and that is a nice property. We call this convention for writing numbers *least-significant-digit-first*. Independent of our writing conventions we are still talking about the same number. Hence, while using the least-significant-digit-first writing convention, we notice that the number 789 is still only 31 away from a thousand.

Least-significant-digit-first representation of numbers is what the inventors of the place value number system intended. Arabs today are still right-to-left writers and thus *they are still writing numbers least-significant-digit-first*. When an Arabic reader scans across a line on a page and sees the first digit of a number, the reader immediately knows this first digit is in the one's place.

In contrast, when a European reader scans across a line of text the first digit of a number he or she sees will be of unknown weight. The European reader then must continue scanning to the right to find the last digit scanned, and then he or she will know that this digit at the far right will be in the one's place. The European reader may then read the number right-to-left just as an Arab reader would, while knowing the weights of the other digits. Essentially, out of practical necessity European readers acknowledge that numbers are written backwards for them, *and thus the one true digit order for reading place value integers is least-significant-digit-first.*

The same happens when European readers perform the most common arithmetic operations such as addition or multiplication. Fact is, starting with the least-significant-digit first is *an artifact of the place value number system*, so it was never an option for the translators to change it, even though they probably did so by accident.

If per chance we desired to fix this error and scan numbers least-significant-digit-first when reading, as originally envisioned and practiced, we European culture writers would either have to start writing from right-to-left like the Arabs do, or we would have to reverse the order of digits in our numbers.

Given the burden of over a thousand years of convention you might wonder why anyone would go through the trouble of fixing this now? Why even talk about this at all? Well as it turns out, internally many *computers* follow the Arabic style least-significant-digit-first convention of writing numbers to memory. These computers scan text characters and numbers in their internal memory in the same address order while going from small addresses to larger addresses. In this way the one's digit is consistently found at the lower address, independent of the precision of the number. I.e. computers do this for the same reason that the inventors of place value representation originally put least significant digits first.

However not all computers do this. Some computers do as European left-to-right authors do, and index digit vectors in the opposite direction from how they index other objects in memory. These machines do this for a couple of reasons, among them, however, is that printouts of byte streams are more natural for European readers to make sense of. For example, when doing a hex dump of an IP packet starting with the first byte transmitted on the left, then the next byte, etc. working right, the source and destination IP numbers may be read directly from the hex dump without having to reverse the four quads.

It is not just hardware that is a mixed up, so are standards specifications and explanatory documents.

In summary, in this section we have defined two concepts. One being that of *scanning-order* the other being *digit-order*. Humans tend to scan a page either left-to-right or right-to-left. Scanning is inherently a one dimensional activity while pages are two dimensional, so we might need to also specify top-to-bottom or right-to-left, and for a book, we also have front-to-back, and back-to-front conventions. There are other possible scanning-orders.

In computing we have two primary scanning orders, low-to-high address incrementing, and high-to-low address decrementing. All conventional computers have used low-to-high address scanning for writing and reading text. Humans find it natural to count objects starting from one, or measure distance starting from zero, and as memory addresses are numbers, incrementing an address is a counting activity. This human inclination causes low-to-high address scanning to feel natural. Hence, it is only when the backwards high-to-low scanning order is used that anyone bothers to mention the scanning-order.

The other concept we discussed, *digit-order,* refers to the order that we scan the digits of a number relative to the scanning-order. The two digit-orders that we discussed are least-significant-digit-first, and most-significant-digit-first. There have been other digit orders used in computing. Only God and the designers of the PDP-11 know why. Designers typically specify digit-order, and leave the scanning-order assumed to be low-to-high.

We still have a loose end. That is the mapping of scanning order of computer memory to print order to paper (or the computer screen) when doing hex dumps of memory. If we limit ourselves to the two possible human writing conventions of left-to-right, and right-to-left, we have four possible hex dump orders for <paper scanning-order, address traversal order>:

1. <left-to-right, low-to-high>,
2. <left-to-right, high-to-low>
3. <right-to-left, low-to-high>
4. <right-to-left, high-to-low>

There are pairs here that are similar, for example <left-to-right, low-to-high> and <right-to-left ,high-to-low>. Usually there are differences in formatting on the page because printing always starts from the margin.

Suppose we have a machine that stores numbers least-signficant-digit-first (with low-to-high address order) on a stack. Further suppose that in our hex dump we want computer engineers to see numbers in a manner they find to be natural, and further yet that we want stacks to grow upwards, as stacking things is want to do. Then we would use bottom-to-top and right-to-left writing order on the printed page, with addresses going from low-to-high starting at the bottom right. On such a diagram our numbers would look natural to a computer engineer, but text would be reversed for Europeans, but just find for Arabs.

Binary, Octal, Hexadecimal, BCD

We may interpret a vector of *n* bits as a number by using this function:

$$\sum_{i=0}^{n} \boldsymbol{a}_i \cdot two^i \quad ; \textit{gives meaning to a binary place value represented number}$$

Just as for vectors of decimal digits we may drop the vector decorations on bit vectors to give us strings of bits. Just as for decimal digit strings, we have two options for writing the string, either most-significant-digit-first, or least-significant-digit-first.   These are also known as most-significant-*bit*-first and least-significant-*bit*-first.

Although both strings of binary digits and strings of decimal digits grow in length logarithmic against a count - binary strings will grow more than twice as fast. The expansion to a length two string will occur immediately at the count of two. Then length expands to three at a count of four. Then to string length four at a count of eight. Hence, while incrementing to eight, the binary digit string has already expanded to length four, yet the decimal digit string hasn't yet budged from being one digit long.

So as to make working with binary strings shorter and more convenient, we typically treat bits in groups. When we group bits in threes, we will then actually be working in base eight. This is called octal. We use one of the symbols 0, 1, 2, ... 7 for each octal digit.

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

In the 1960s it was common to find computer panels with switches and lights organized in threes, and for coding forms to be filled out in octal. However, today almost all documents that must show bit strings will use groups of fours. This is the hexadecimal system. It has sixteen base symbols. After reaching 9 we just use letter:

octal	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
a	1010
b	1011
c	1100
d	1101
e	1110
f	1111

It is not a coincidence that the table of hexidecimal digits is twice as long as the table of octal digits. Each time we add another bit to the grouping, the table will double in size.

The memory of a computer can be thought of as a very large array of bits. All contemporary computers will present an architectural model to the programmer where groups of bits called *bytes* are addressable. Byte is of course a pun off of the term *bit.*

Today a byte is universally a group of eight bits; however, computers of the past have used other values ranging from six to twelve bits. Vestiges of that past remain, such as found in the C language standard. The designers of UTF-8 wanted to make clear that they considered groups of eight, so they named such a group an *octet* rather than a *byte*. When a group of eight bits, i.e. an octet, is considered as a digit of a number, we will be working in base 256 (here the number 256 is given in decimal representation. In hexadecimal representation it would be written as 100.). Note *octet* with *octal* are different concepts. An octet is a group of 8 bits, where as octal is a number system based on digits that have values running from 0 to 7.

Groups of bits get larger than this. The organization of early RISC microprocessors was such that memory was always moved in groups of 32 bits, called *words*. Today it is common that address variables will be 64 bit words while integer variables will be either 32 or 64 bits. Groups of bits found on internal buses can be larger yet.

We can also group bits to create numbers that have bases that are not powers of two. For example, in the BCD code we group bits in fours to create decimal digits.

BCD	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

In BCD some possible bit value combinations are simply not used. This will always be the case for emulating a number base in binary switch logic when that base is not a power of 2.

Octal and hexadecimal are really just writing aids used by computer engineers. Internally the computer will be computing in binary, i.e. base 2. In contrast, BCD computation can not just be a writing aid because of the ignored encodings. When two BCD digits are added, the carry must occur when the sum of two digits reaches ten or more, which is not on a binary number boundary.

Some processors only support BCD to binary conversion operations. So numbers that appear as BCD in memory are converted to binary before computation. However, other computers, most notably many IBM machines, will implement BCD architectures, i.e. have instructions for directly operating on BCD encoded numbers.

Although BCD encoding is less efficient, BCD encoded binary place values numbers still grow in length logarithmically while counting. Because numbers will be represented in base ten no number conversation has to be done when printing. They also divide by ten without creating infinite fractions. It was not long ago that programmers who wrote business programs preferred this encoding scheme, and some still do.

Allocation

In most text documents a number is written down once and that is it. In contrast, while computing we often come back and change a given value many times. This would be equivalent to going to a paper document and repeatedly erasing the old value, and then writing a new value in the same space.

Consider the case of recording a count in real time, where at any moment we might have to erase the current count value, and then to write a new one. Suppose we begin at zero. Then when something arrives to be counted we increment to 1. We still have a single digit. Our count will grow to two digits in length upon reaching the count of *ten*. As we continue the count will grow by another digit in length when we reach a *hundred*. It grows again when we reach a *thousand*, etc. It is probably not a coincidence that we have names for the numbers at these boundaries. Thus a digit vector that represents the count grows in number of components, i.e. in length, against the log of the count value. The log function grows without bound, but it does so quite slowly, so the net effect is that relatively short digit vectors can be used to represent rather large numbers. This observation turns out to be key to understanding contemporary computer architecture.

If we had a paper document and only left space for a single digit count, we would run into a problem when the count grows to ten, and an even a bigger problem when it eventually grows to a thousand. If we knew in advance that we would want work our way up to counts in the thousands, we could simply *allocate* enough space for four digits to start with. Such an allocation by itself is not much of a waste of space because the digit vectors for very large numbers are not much longer than those for small numbers.

Yet, for any choice we make for amount of space to allocate, the possibility remains that a number will come along that is so large that its representation doesn't fit in the allocation, i.e. that the number will *overflow* the allocation. Basically to make reasonable allocations we have to know how much numbers grow with the operations applied to them, and how many of those operations will be applied. This latter question could well be related to how long we plan to keep working on a given problem. As any child knows who has tried to count to a million, at some point this issue becomes moot, and contemporary computer architectures rely upon this phenomena.

Just as numbers require space to be written on a page of print, they also require space to be written into computer memory. The conventional solution is to assume that all numbers will fit into fixed length allocations called *words* . Then a map is made of the location of all the words in memory even before the program begins. Because the locations of the words are known in advance, these addresses of these words may be placed directly into the instructions of the program by the compiler, the linker, or loader; with an analogous situation for interpreters.

If we assume that a program starts with initialized state, then when said program begins running, all numbers will fit in the words they have been allocated into. However, as the program runs, it is possible that larger and larger values will be computed then stored. This might not happen, but depending on the details of the program, it can happen.

In general it is not possible to write a program that can analyze another program and determine how long it would run before stopping, or if it would ever stop at all. Even in specific cases where it is possible to succeed at such an analysis, the analysis itself might be so difficult and time consuming that the programmer does not see it as a worth while exercise. In point of fact, nobody does such an analysis unless they plan to put the code into a life critical application, and often times, not even then. Most programmers would not even know how to go about doing it anyway. Because data might grow in length as a computation runs longer, computer users live under the Sword of Damocles of not knowing if an allocation will overflow and cause an error. If the sword does fall and the program crashes, reports an incomprehensible error, or returns the wrong answer - the user simply reports it as a bug. In a sense, programmers let the users do the analysis for them, and users become accustom to this.

Each fixed length allocation, such as a word, will contain multiple bytes of data, where each byte has an address. Hence there will be a byte with the smallest address, and a byte with the largest address. All memory managers use the smallest address as the address of the allocation itself.

Let us say that ‘a number starts’ with its least-significant-digit. This nomenclature is justified based on the observation that independent of the writing scheme, a reader will have to read a number starting with its least-significant-digit so as to the know the weights of the digits. Accordingly, the allocation scheme where the number starts at the smallest byte address in the word is called *little-endian*, while the allocations scheme where the number starts at the biggest byte address within the word is called *big-endian*.

With the little-endian justification of the digits of a number within an allocation, said number will be least-significant-digit first relative to low-to-high scanning of memory. With big-endian justification said number will be most-significant-digit-first relative to low-to-high scanning of memory. Because of this, some computer engineers will conflate the terms little-endian and big-endian with the terms least-significant-digit first and most-significant-digit first, though this is sloppy. The two concepts are different.

The terms *little-endian* and *big-endian* were coined by Danny Cohen as a sort of joke in a paper he published on April 1st, 1980, "ON HOLY WARS AND A PLEA FOR PEACE". Like many engineers Danny was frustrated by the failure of industry to adopt one convention of as a standard. He alludes to "Gulliver's Travels" where Swift described the Lilliputians being at war with the Blefuscians over the question of which end of an egg should be facing up in an egg cup.

In Figure 1 we see a word that has bytes addressed from *c0* to *c3,* where those addresses are given in hexadecimal. In decimal those addresses would be written as 192, 193, 194, 195; however, we almost always use hexadecimal notation for addresses. The address of the byte before c*0* is the address *bf.* The address after *c3* is *c4,* as shown. The address for the word itself is taken to be *c0*, because the address for an allocation is always that of the smallest address appearing in the allocation. If we consider that a digit of the number is a byte, and bytes are octets, the binary encoding for the least significant digit of this number is 0001 1000. The most significant digit is 1010 1110.

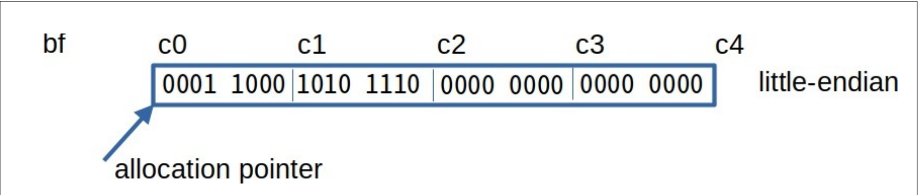

Figure 1: Number in word using little-endian digit order

Now in Figure 2 for big-endian digit order the number is placed into the word going in the opposite direction. For all but very large numbers, the digit being pointed at by the allocation pointer will be zero. We will continue to scan zeros until either reaching the end of the allocation or upon reaching the most significant digit. If we reach the end of the allocation, the contained number is taken to be zero. Because this is the same number as shown in the prior figure, it also has the same least significant digit and the same most significant digit.

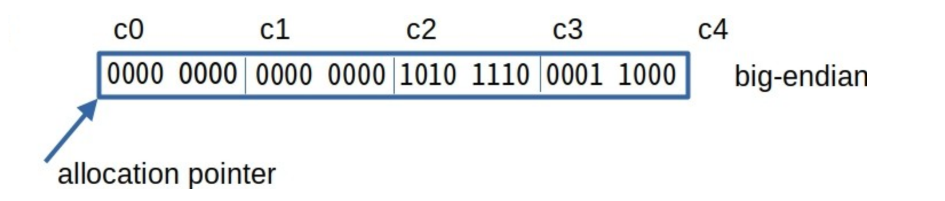

Figure 2: Same number in word using big-endian digit order

Now suppose our word is holding a count. This is the same as saying that the word is a *counter*. When counting with the little-endian convention, a number will grow into larger addresses as the count carries into new digits. In contrast, with big-endian, counting will carry into ever smaller memory addresses.

Now consider what happens when a number grows and overflows the allocation. In the little-endian case fixing an overflow when incrementing would conceptually require expanding the allocation the number is held in. Something becoming too long for its allocation is a common enough problem that some heap libraries include a function that attempts to expand an allocation. In contrast, in the big-endian case an overflow will run over the allocation pointer, so we can not fix this problem by just calling a library routine to expand the allocation (when that option happens to be available). One trick we might try would be to expand the allocation and then slide the number over into higher addresses to make space for another digit. If a programmer uses such a scheme, then the program that loads the number would have to take into account the size of the adjusted allocation to know the weight of each digit. That is a complex task.

Hence, the little-endian allocation convention fits our thinking about data and allocation better than does the big-endian convention. As we noted earlier little-endian words will print least-significant-digit first in a hex dump, i.e. backwards. If perchance a clever dump program prints data from large address to small address going left-to-right, digits will appear in a natural order, but European text will be backwards. If instead we use the big-endian allocation scheme, a dump of memory will show everything in the expected order for the European reader. Though, we really have to question, is 'looking natural' in memory dumps that important? After all, the end user will be looking at printouts, not memory dumps, and any competent computer engineer would be capable of the mental gymnastics required for reversing digit order.

This issue with memory allocation and overflow is not generally discussed today because when a number gets too large for its allocation, no matter the digit order convention, one of the following will happen: the number will be truncated to fit and the overflow will be ignored - thus leading to run time errors, the overflow will result in an exception being signaled, or for some specialized processors a bit pattern representing infinity will be substituted. Either way, the programmer will not be aware of the direction of growth in the number that lead up to the allocation overflow event.

Conventionally processor registers are also one word in length, and a processor will load the entire contents of a word of memory into a register as a single operation. Consequently the contemporary processors do not care how many bytes/digits are in a number held within said word. However, the ALU will impose an order on the digits. Take for example, when a register is incremented, the increment must be inserted at the least-significant-digit end.

Suppose we had an unconventional processor, or perhaps a future processor, that loaded numbers as digit streams. The load instruction would have to have a means of detecting the end of a number being loaded, or it would somehow have to know a-priori how long the number is. Such a problem is nearly identical to the already existing problem of loading strings of characters. Both length counts and end terminators have been used for solving the problem of knowing how many characters to load.

Bit Order Within Bytes

Data is transported between different points within a processor or computer system over bundles of wires called buses. Buses have specifications that include the order of bits in bytes, and for all contemporary machines bytes are octets. All compute processors, channel processors, and other devices plugged into a bus must conform to the bus's specifications.

Channel processors are often used to bridge between a computer's system bus and a standard storage bus that connects to the one or more storage devices. In which case the storage device designers only have to know about the standard storage bus specification. The compatibility problem is then limited to the channel processor, which is careful designed to satisfy both the processor bus standard and the storage bus standard.

Hence bit order issues occur only at a layer below the standards and specifications for major components of the computer. Unless a programmer is involved in the design of such things as channel processors, he or she will not even be aware of the order of bits within bytes.

This is not to say there can't be differences. In fact some processors do store bits into memory bytes in different order than others. However, the values that are read and written into the respective memories must be placed on a bus, and at that point the bit order is described by the relevant specification.

On all contemporary computers it is possible to perform arithmetic on bytes. Carries always travel from the lower significant bits to more significant bits, consequently ALUs also impose a bit order within bytes. However, that order will conform to the order documented for the processor's bus. In every document I have ever seen for a processor and buses, the ALU treats documented bit 0 as the least significant bit.

I challenge any programmer to try and write a C program that detects the physical order of bits in bytes of memory, or to identify a difference in bit order within bytes between computers. Because this is impossible in all but special cases, we typically do not say that a pointer to a byte of memory is pointing at either the most significant bit, nor the least significant bit. Rather, an address points at the whole byte.

The people who have named the shift instructions on processors have conflated print direction with bit order. Thus a *right shift* always moves bits towards the least significant end of the byte, and a *left shift* always shifts bits to the more significant end of a byte. Due to the translators in the middle ages not reversing sequences of digit characters this sense of direction works for both European convention writing and for Arabic writing. Note that this is not an endian issue, because, as explained above, the allocation pointer locates the whole byte.

Perhaps we would be better off to get directly to the point and name shift instructions based on how the significance of the bits change. I would suggest a *diminishing* shift and a *magnifying* shift. In the case that shifts are used with non-numeric data such as bit masks, diminishing and magnifying is still an accurate description of what happens to the indexes on the bits.

Currently all conventional processors have a byte addressing model. So Danny Cohen has gotten the peace that he desires when it comes to bits within bytes. However processors in the past have had bit addressing, or instructions for creating the appearance of bit addressing. For those processors we could ask the questions: when we increment bit addresses and scan over a number, do the weights of the bits go up, or do they go down, as the address increases? I.e. are addresses changes correlated or anti-correlated to bit weights? Also, we would be able to ask the endian question about bits: does a byte have leading zeros bits with increasing bit addresses, or with decreasing bit addresses?

Byte Order Within Words

A given processor architecture might simultaneously have native support for byte data and various lengths of words. The most common word lengths being 16, 32, and 64 bits. Stated equivalently as 4, 8, and 16 bytes.

Most all communication channels and storage devices have payloads organized as octets, *but they have no built in support for words*. Hence when we desire to store a word or to communicate a word, we will be forced to first serialize the word into a series of bytes, then transfer or store the data, then to read the data back while deserializing the byte stream back into words.

Place value numbers consist of vectors of digits. Words of allocation consist of consecutively addressed bytes. Bytes are handled atomically by all modern machines. Hence, any bit encoding for the digits of a number must pack cleanly into bytes, or the digits themselves might be split apart. Such a clean packing might require padding with zeros to make the length of the number to be even by 8 bits. For example, a hexadecimal string of digits might be 12 bits long. Another 4 zeros must be appended to make it a multiple of 8 bits. When this criteria is met, we suffer no loss of generalization by considering that a number is a series of bytes as digits. It is because of this fact that little-endian and big-endian are often referred to as byte orders. When this is done we are considering the bytes within a number, i.e. thinking in terms of byte digits.

The following figure shows a stream of bytes arriving as data, and then that data being copied into a word. In this case the digits of the word, the bytes, arrive in little-endian order and they are received on a little-endian machine, so they are just copied in the order they are scanned off the channel.

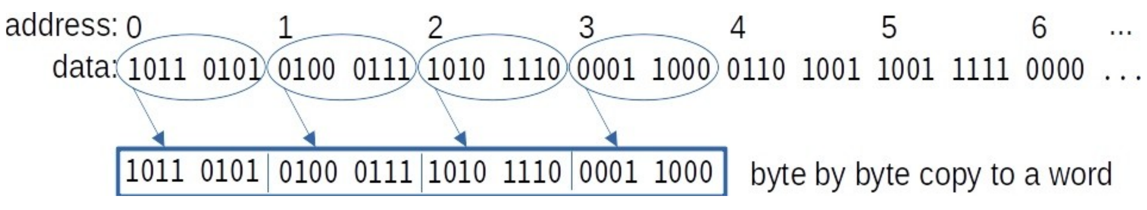
address: 0 1 2 3 4 5 6 ...
data: 1011 0101 0100 0111 1010 1110 0001 1000 0110 1001 1001 1111 0000 . . .
1011 0101 0100 0111 1010 1110 0001 1000
byte by byte copy to a word

Figure 3: In address order byte by byte copy

In this second case the same stream of data arrives with words serialized as bytes in little-endian order, but this time the receiving machine is big-endian. The bytes will have to be reversed on a word by word basis.

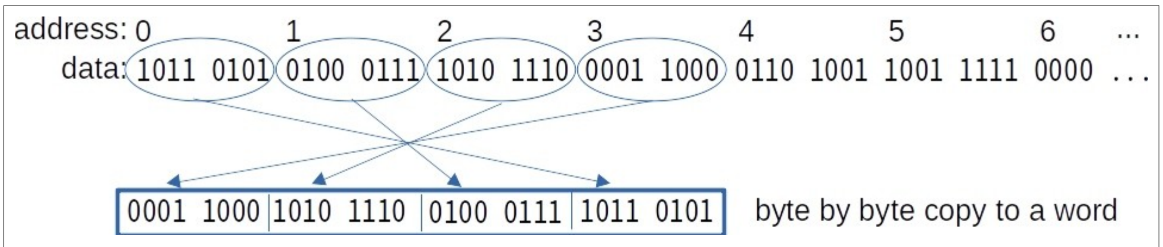
address: 0 1 2 3 4 5 6 ...
data: 1011 0101 0100 0111 1010 1110 0001 1000 0110 1001 1001 1111 0000 . . .
0001 1000 1010 1110 0100 0111 1011 0101
byte by byte copy to a word

Figure 4: Reverse order byte copy for words

Here is the problem, when the data arrives there is no way to know where the words are. That information was lost when we serialized. If we do not know where the words are we can not know when to do the byte order reversal. Hence this problem can not be solved using the same approach that was used for solving the problem of bit order in bytes.

At some level of abstraction all applications must be designed to be able to make sense of their data. It is just too bad that byte order within words has to be an application level design consideration, because it has nothing to do with applications.

In the case of something like the Internet Protocol, IP, the specification dictates the offset as to where data is to be placed into the packet header. Coding is done in a stable and efficient manner. However IP does not know where the words are located in the data payload it is carrying, so IP just passes the payload, byte order and all, up the abstraction stack.

JSON is a standard for expressing tagged structured data built from primitive types while using character only data. It comes with a specification saying how numeric character strings are to be interpreted. This makes it so that both little-endian and big-endian machines can share JSON character encoded numeric data. JSON is not a very stable means of encoding data and it is inefficient, but it has one great thing going for it. Namely, it works.

There are many file formats and data communications standards that serve various classes of applications by making it clear where words that need to be reversed are located when data is transferred between machines.

Survey

The Wikipedia article, "Endianness" has an extensive survey.

https://en.wikipedia.org/wiki/Endianness#References

Conclusion

Cohen called for peace. Let me propose a way to get that peace. Let us stop reproducing an error made by scribes in the 9th Century AD, and simply ditch big-endian and go back to the original vision for place value represented integers. That will move the problem of making numbers look pretty for European readers to the print function, but that function already has this purpose, and it will be possible to do it in the print function because that function will know what order the bytes are in. Should we do this, numbers that grow under the application of the most common operations such as incrementing, adding, and multiplying, will bump into the ends of their allocation blocks, not the beginnings of them. Characters and numbers will both scan going from little addresses to big addresses. And most importantly there will be no need to bind 'where the words are' information back into the low level reading of data from storage or communication channels. The net result will be that byte order in words will become a non-issue just like bit order in bytes.