SOFTWARE ENGINEERING

What is the advantage of little endian format?

Asked 11 years, 2 months ago Modified 2 years, 3 months ago Viewed 94k times

194

Intel processors (and maybe some others) use the little endian format for storage.

I always wonder why someone would want to store the bytes in reverse order. Does this format have any advantages over the big endian format?

architecture storage advantages endianness

69

Share Improve this question Follow

edited Jul 16, 2012 at 13:05 Radu Murzea 1,812 2 18 24

asked Jul 24, 2011 at 19:04 Cracker 3,003 5 18 20

1 The 6502 was an early (the first?) pipelined processor. I seem to remember some claim about it being little-endian for some performance-related issue due to the pipeline - but I have no idea now what that issue might have been. Any suggestions? – user8709 Jul 24, 2011 at 20:51

4 Little-endian, big-endian - you must choose one or the other. Like driving on the left or the right side of the road. – user1249 Jul 16, 2012 at 13:13

3 I suggest you to write some code in ASM, preferably for an "old-school" architecture such as 6502 or Z80. You will immediately see why these use little endian. Architectures that use big endian have certain characteristics to their instruction set that make that format preferable instead. It's not an arbitrary decision to make! – Stefan Paul Noack May 15, 2013 at 15:01

3 Each byte-order system has its advantages. Little-endian machines let you read the lowest-byte first, without reading the others. You can check whether a number is odd or even (last bit is 0) very easily, which is cool if you're into that kind of thing. Big-endian systems store data in memory the same way we humans think about data (left-to-right), which makes low-level debugging easier. – Koray Tugay Jan 21, 2016 at 8:05

2 Actually it's the Big Endian which has the bytes swapped. Why would you ever want to read 0001 as "a thousand"?. – Pablo Ariel Jul 3, 2018 at 2:02

11 Answers

Sorted by: Highest score (default)

240

There are arguments either way, but one point is that in a little-endian system, the address of a given value in memory, taken as a 32, 16, or 8 bit width, is the same.

In other words, if you have in memory a two byte value:

```
0x00f0   16
0x00f1    0
```

taking that '16' as a 16-bit value (c 'short' on most 32-bit systems) or as an 8-bit value (generally c 'char') changes only the fetch instruction you use — not the address you fetch from.

On a big-endian system, with the above layed out as:

```
0x00f0    0
0x00f1   16
```

you would need to increment the pointer and then perform the narrower fetch operation on the new value.

So, in short, 'on little endian systems, casts are a no-op.'

Share Improve this answer Follow

edited Jul 25, 2011 at 3:35
Neil G
420 2 14

answered Jul 24, 2011 at 20:04
jimwise
7,225 1 29 32

3 Assuming, of course, that the high-order bytes that you didn't read can reasonably be ignored (e.g. you know that they're zero anyway). – user8709 Jul 24, 2011 at 21:06

12 @Steve314: If I'm in C downcasting from 32 to 16 bits (e.g.) on a 2's-complement system - the vast majority of systems - the bytes don't need to be zero to be ignored. Regardless of their value I can ignore them and remain compliant with the C standard and programmer expectations. – user23679 Jul 24, 2011 at 22:08

10 @Stritzinger -- we're talking about the assembly/machine code generated by a compiler, which can't be portable. The higher level language code to compile is portable -- it just compiles to different operations on the different architectures (as all ops do). – jimwise Jul 25, 2011 at 14:51

7 I don't buy this argument, because on big-endian architectures, a pointer could point to the end, rather than the beginning, of a whatever it is that you are referring to and than you'd have exactly the same advantage. – dan_waterworth Jul 26, 2011 at 14:54

4 @dan_waterworth not quite -- keep in mind the pointer arithmetic rules in C, for instance, and what happens when you increment or decrement casts of the same pointer. You can move the complexity, but you can't eliminate it. – jimwise Jul 26, 2011 at 20:34

63

OK, here's the reason as I've had it explained to me: **Addition and subtraction**

When you add or subtract multi-byte numbers, you have to start with the least significant byte. If you're adding two 16-bit numbers for example, there may be a carry from the least significant byte to the most significant byte, so you have to start with the least significant byte to see if there is a carry. This is the same reason that you start with the rightmost digit when doing longhand addition. You can't start from the left.

Consider an 8-bit system that fetches bytes sequentially from memory. If it fetches the least significant byte *first*, it can start doing the addition *while* the most significant byte is being fetched from memory. This parallelism is why performance is better in little endian on such as system. If it had to wait until both bytes were fetched from memory, or fetch them in the reverse order, it would take longer.

This is on old 8-bit systems. On a modern CPU I doubt the byte order makes any difference and we use little endian only for historical reasons.

Share Improve this answer Follow

answered Jul 25, 2011 at 20:15
Martin Vilcans
839 1 7 8

3 Ah - so it's roughly the same reason I use little-endian chunk ordering for big integers. I should have worked that out. People really need to get working on cybernetics *now* - my brain's already in desperate need of some replacement parts and some radical upgrades, I can't wait forever! – user8709 Jul 25, 2011 at 20:51

2 A thought - the 6502 didn't do much 16-bit math in hardware - it was, after all, an 8 bit processor. But it *did* do relative addressing, using 8-bit signed offsets relative to a 16-bit base address. – user8709 Jul 25, 2011 at 20:55

2 Note that this idea still matters for multiple precision integer arithmetic (as said by Steve314), but at the word level. Now, most operations are not directly affected by the endianness of the processor: one can still store the least significant word first on a big-endian system, as done by GMP. Little-endian processors still have an advantage for the few operations (e.g. some string conversions?) that could be easier done by reading one byte at a time, since only on a little-endian system, the byte ordering of such numbers is correct. – vinc17 Aug 19, 2014 at 9:29

little-endian processors have an advantage in case the memory bandwidth is limited, like in some 32-bit ARM processors with 16-bit memory bus, or the 8088 with 8-bit data bus: the processor can just load the low half and do add/sub/mul... with it while waiting for the higher half – phuclv Mar 23, 2018 at 15:00

Well, you could just read the value from the highest address to the lowest. – Simon Richter Apr 28, 2020 at 12:27

I always wonder why someone would want to store the bytes in reverse order.

54

Big-endian and little-endian are only "normal order" and "reverse order" from a human perspective, and then only if all of these are true...

1. You're reading the values on the screen or on paper.
2. You put the lower memory addresses on the left, and the higher ones on the right.
3. You're writing in hex, with the high-order nybble on the left, or binary, with the most significant bit on the left.
4. You read left-to-right.

Those are all human conventions that don't matter at all to a CPU. If you were to retain #1 and #2, and flip #3, little-endian would seem "perfectly natural" to people who read Arabic or Hebrew, which are written right-to-left.

And there are other human conventions that make big-endian that seem unnatural, like...

- The "higher" (most significant) byte should be at the "higher" memory address.

Back when I was mostly programming 68K and PowerPC, I considered big-endian to be "right" and little-endian to be "wrong". But since I've been doing more ARM and Intel work, I've gotten used to little-endian. It really doesn't matter.

Share Improve this answer Follow

answered Jul 24, 2011 at 21:14

Bob Murphy
15.9k 3 51 77

39 Numbers are in fact written from [most significant digit] left to [least significant digit] right in Arabic and Hebrew. – Random832 Jul 24, 2011 at 23:32

6 Then why are bits within a byte stored in "big endian" format? Why not be consistent? – tskuzzy Jul 25, 2011 at 12:42

12 They aren't - bit 0 is by convention the least significant, and bit 7 the most significant. Moreover, you can't generally put an order on bits within a byte, since bits aren't individually addressable. Of course, they might have a physical order in a given communication protocol or storage media, but unless you're working at the low-level protocol or hardware level, you don't need to concern yourself with this order. – Stewart Jul 25, 2011 at 13:22

3 @Stewart: Yes they are - bit 0 is by convention the right-most bit, with bit-7 on the left. Also, all C-programmers should concern themselves with this ordering; if they don't, it's very easy to write code that is not portable across platforms *(the original version of `sed` - which we had to fix for an "Advanced C in Unix" class - was badly broken on all platforms that weren't 32-bit little-endian)* – BlueRaja - Danny Pflughoeft Jul 25, 2011 at 16:08

5 BlueRaja: only by convention of writing on paper. This has nothing in common with CPU architecture. You can write the byte as 0-7 LSB-MSB instead of 7-0 MSB-LSB and nothing changes from algorithm point of view. – SF. Jul 26, 2011 at 7:14

18

With 8 bit processors it was certainly more efficienct, you could perform an 8 or 16bit operation without needing different code and without needing to buffer extra values.

It's still better for some addition operations if you are dealing a byte at a time.

But there is no reason that big-endian is more natural - in English you use thirteen (little endian) and twenty three (big endian)

Share Improve this answer Follow

answered Jul 24, 2011 at 19:15

Martin Beckett
15.7k 3 42 69

2 Big-endian is indeed easier for humans because it does not require rearranging the bytes. For example, on a PC, `0x12345678` is stored as `78 56 34 12` whereas on a BE system it's `12 34 56 78` (byte 0 is on the left, byte 3 is on the right). Note how the larger the number is (in terms of bits), the more swapping it requires; a WORD would require one swap; a DWORD, two passes (three total swaps); a QWORD three passes (7 total), and so on. That is, `(bits/8)-1` swaps. Another option is reading them both forwards **and** backwards (reading each byte forwards, but scanning the whole # backwards). – Synetech Jul 24, 2011 at 20:12

One-hundred-and-thirteen is either middle-endian, or else big-endian with "thirteen" being essentially one non-decimal digit. When we spell out numbers, there are some minor deviations from the constant-base conventions that we use for digits, but once you strip out those special cases the rest is big-endian - millions before thousands, thousands before hundreds etc. – user8709 Jul 24, 2011 at 20:58

2 @Steve314, the spelled-out words of numbers don't matter, it's the numerical readout that is what we use when we program. Martin, no computers don't have to care how humans read numbers, but if it's easy for humans to read them, then programming (or other related work) becomes easier and some flaws and bugs can be reduced or avoided. – Synetech Jul 24, 2011 at 23:39

2 @steve314 And in Danish, "95" is pronounced "fem halvfems" (five, plus four-and-a-half twenties). – Vatine Nov 12, 2012 at 16:53

1 @Synetech: `0x12345678` is stored as `87 65 43 21`, depending on how one *chooses* to write it. How one writes something rather arbitrary (read as: context dependent). – Thomas Eding Jan 26, 2014 at 10:37

11

The Japanese date convention is "big endian" - yyyy/mm/dd. This is handy for sorting algorithms, which can use a simple string-compare with the usual first-character-is-most-significant rule.

Something similar applies for big-endian numbers stored in a most-significant-field-first record. The significance order of the bytes within the fields matches the significance of the fields within the record, so you can use a `memcmp` to compare records, not caring much whether you're comparing two longwords, four words, or eight separate bytes.

Flip the order of significance of the fields and you get the same advantage, but for little-endian numbers rather than big-endian.

This has very little practical significance, of course. Whether your platform is big-endian or little-endian, you can order a records fields to exploit this trick if you really need to. It's just a pain if you need to write *portable* code.

I may as well include a link to the classic appeal...

http://tools.ietf.org/rfcmarkup?url=ftp://ftp.rfc-editor.org/in-notes/ien/ien137.txt

EDIT

An extra thought. I once wrote a big integer library (to see if I could), and for that, the 32-bit-wide chunks are stored in little-endian order, irrespective of how the platform orders the bits in those chunks. The reasons were...

1. A lot of algorithms just naturally start working at the least significant end, and want those ends to be matched. For example in addition, the carries propogate to more and more significant digits, so it makes sense to start at the least significant end.
2. Growing or shrinking a value just means adding/removing chunks at the end - no need to shift chunks up/down. Copying may still be needed due to memory reallocation, but not often.

This has no obvious relevance to processors, of course - until CPUs are made with hardware big-integer support, it's purely a library thing.

Share Improve this answer Follow edited Jul 25, 2011 at 12:20 answered Jul 24, 2011 at 20:42 user8709

10

Nobody else has answered WHY this might be done, lots of stuff about consequences.

Consider an 8 bit processor which can load a single byte from memory in a given clock cycle.

Now, if you want to load a 16 bit value, into (say) the one and only 16 bit register you have - ie the program counter, then a simple way to do it is:

- Load a byte from the fetch location
- shift that byte to the left 8 places
- increment memory fetch location by 1
- load the next byte (into the low order part of the register)

the outcome: you only ever increment the fetch location, you only ever load into the low order part of you wider register, and you only need to be able to shift left. (Of course, shifting right is helpful for other operations so this one is a bit of a side show.)

A consequence of this is that the 16 bit (double byte) stuff is stored in order Most..Least. I.e., the smaller address has the most significant byte - so big endian.

If you instead tried to load using little endian, you would need to load a byte into the lower part of your wide register, then load the next byte into a staging area, shift it, and then pop it into the top of your wider register. Or use a more complex arrangement of gating to be able to selectively load into the top or bottom byte.

The result of trying to go little endian is you either need more silicon (switches and gates), or more operations.

In other words, in terms of getting bang for buck back in the old days, you got more bang for most performance and smallest silicon area.

These days, these considerations and pretty much irrelevant, but things like pipeline fill *may* still be a bit of a big deal.

When it comes to writing s/w, life is frequently easier when using little endian addressing.

(And the big endian processors tend to be big endian in terms of byte ordering and little endian in terms of bits-in-bytes. But some processors are strange and will use big endian bit ordering as well as byte ordering. This makes life *very* interesting for the h/w designer adding memory-mapped peripherals but is of no other consequence to the programmer.)

Share Improve this answer Follow

edited Jul 4, 2018 at 6:40
The Elemental of Destruction
3 3

answered Jul 25, 2011 at 4:44
quickly_now
14.7k 1 34 48

3

jimwise made a good point. There is another issue, in little endian you can do the following:

```
byte data[4];
int num=0;
for(i=0;i<4;i++)
    num += data[i]<<i*8;

OR

num = *(int*)&data; //is interpreted as

mov dword data, num ;or something similar it has been some time
```

More straight forward for programmers which are not affected by the obvious disadvantage of swapped locations in the memory. I personally find big endian to be inverse of what is natural

:). 12 should be stored and written as 21 :)

Share Improve this answer Follow

answered Jul 24, 2011 at 21:06
Cem Kalyoncu
139 3

2 This just proves that it's faster/easier to work in whatever format that is native to the CPU. It doesn't say anything about whether it's better. The same thing goes for big endian: `for(i=0; i<4; i++) { num += data[i] << (24 - i * 8); }` corresponds to `move.l data, num` on a big endian CPU. – Martin Vilcans Jul 25, 2011 at 22:34

@martin: one less subtraction is better in my book – Cem Kalyoncu Jul 25, 2011 at 23:40

It doesn't really matter as the compiler will unroll the loop anyway. In any case, many CPUs have byte swapping instructions to handle this problem. – Martin Vilcans Jul 26, 2011 at 9:04

i dont agree bcoz on big endian, i would do { num <<= 8; num |= data[i]; } at least this does not have to calculate left shift count using mul – Hayri Uğur Koltuk Jul 26, 2011 at 13:20

@ali: your code will do the exact operation I wrote and will not work on big endian. – Cem Kalyoncu Jul 26, 2011 at 19:14

2

1) The main reason is that a cast operation itself becomes invisible to hardware; recasting a pointer does not require an arithmetic operation to change the address of a pointer and does not rely on the compiler knowing the previous type to know how much to offset the pointer by. To overcome this, the pointer could always point to the end of the data in memory but this is more unintuitive than using little endian to solve the the problem

2) Lets say someone stored a dword 00 00 00 01 in big endian format at 0x100. You'd need to know the format of the data at the address to be able to read the value you want. With little endian you'd just be able to read a byte from 0x100 and it would be 1, with little endian, you need to know that a dword was stored there and read from 0x103. This abstracts the details of the memory layout less than little endian does because the programmer needs to be aware of the representation of the data.

3) At the hardware level, it may simplify store to load forwarding, because if you write a dword to 0x100 and then read a byte, it would be reading the same address on little endian rather than 0x103. This means store to load forwarding in the store buffer for a byte would require 0 checks. When you load a byte at 0x103, you'd have to check other addresses in the store buffer and their data lengths.

The answers talking about being able to immediately perform an arithmetic operation on the lowest byte while fetching the highest byte is wrong because on big endian, it could just fetch the highest byte first.

Share Follow

edited Jun 11, 2020 at 15:43

answered Jun 11, 2020 at 15:34

Lewis Kelsey
131 6

1

Big-endian is useful for some operations (comparisons of "bignums" of equal octet-length springs to mind). Little-endian for others (adding two "bignums", possibly). In the end, it

depends on what the CPU hardware has been set up for, it's usually one or the other (some MIPS chips were, IIRC, switchable on boot to be LE or BE).

Share Improve this answer Follow

answered Jul 25, 2011 at 15:34
Vatine
4,239 20 20

1

> I always wonder why someone would want to store the bytes in reverse order

Decimal number are written big endian. It also how you write it in English You start with the most significant digit and the next most significant to the least most significant. e.g.

```
1234
```

is one thousand, two hundred and thirty four.

This is way big endian is sometimes called the natural order.

In little endian, this number would be one, twenty, three hundred and four thousand.

However, when you perform arithmetic like addition or subtraction, you start with the end.

```
  1234
+ 0567
  ====
```

You start with 4 and 7, write the lowest digit and remember the carry. Then you add 3 and 6 etc. For add, subtract or comparison, it is simpler to implement, if you already have logic to read the memory in order, if the numbers are reversed.

To support big endian this way, you need logic to read memory in reverse, or you have RISC process which only operates on registers. ;)

A lot of the Intel x86/Amd x64 design is historical.

Share Improve this answer Follow

answered Jul 26, 2011 at 6:49
Peter Lawrey

936 5 7

1

When only storage and transfer with variable lengths are involved, but no arithmetics with multiple values, then LE is usually easier to write, while BE is easier to read.

Let's take an int-to-string conversion (and back) as a specific example.

```
int val_int = 841;
char val_str[] = "841";
```

When the int is converted to the string, then the least significant digit is easier to extract than the most significant digit. It can all be done in a simple loop with a simple end condition.

```
val_int = 841;
// Make sure that val_str is large enough.

i = 0;
do // Write at least one digit to care for val_int == 0
{
    // Constants, can be optimized by compiler.
    val_str[i] = '0' + val_int % 10;
    val_int /= 10;
    i++;
}
while (val_int != 0);

val_str[i] = '\0';
// val_str is now in LE "148"
// i is the length of the result without termination, can be used to reverse it
```

Now try the same in BE order. Usually you need another divisor that holds the largest power of 10 for the specific number (here 100). You first need to find this, of course. Much more stuff to do.

The string to int conversion is easier to do in BE, when it is done as the reverse write operation. Write stores the most significant digit last, so it should be read first.

```
val_int = 0;
length = strlen(val_str);

for (i = 0; i < length; i++)
{
    // Again a simple constant that can be optimized.
    val_int = 10*val_int + (val_str[i] - '0');
}
```

Now do the same in LE order. Again, you'd need an additional factor starting with 1 and being multiplied by 10 for each digit.

Thus I usually prefer to use BE for storage, because a value is written exactly once, but read at least once and maybe many times. For its simpler structure, I usually also go the route to least once and maybe many times. For its simpler structure, I usually also go the route to convert to LE and then reverse the result, even if it writes the value a second time.

Another example for BE storage would be UTF-8 encoding, and many more.

Share Improve this answer Follow

answered Nov 12, 2012 at 11:30

Secure

1,918 12 10

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.