Purpose of memory alignment

Asked 16 years ago Modified 1 year, 2 months ago Viewed 129k times

288

Admittedly I don't get it. Say you have a memory with a memory word of length of 1 byte. Why can't you access a 4 byte long variable in a single memory access on an unaligned address(i.e. not divisible by 4), as it's the case with aligned addresses?

memory alignment memory-alignment

Share
Improve this question
Follow

edited Nov 20, 2009 at 20:02
Dan Hook
7,057 ● 8 ● 36 ● 53

asked Dec 19, 2008 at 15:18
Daar
3,545 ● 4 ● 22 ● 18

24 After doing some **additional** Googling I found this great link, that explains the problem really well. – Daar Dec 19, 2008 at 15:31

11 @ark link broken – John Jiang Mar 22, 2020 at 5:00

11 @JohnJiang I think I found the new link here: developer.ibm.com/technologies/systems/articles/pa-dalign – ejohnso49 Apr 17, 2020 at 3:56

6 As of 2022 the link provided by @ark seems to have moved again. Therefore here's a snapshot of the original page linked back in 2008: web.archive: Data alignment - Straighten up and fly right – Frederik Hoeft Jun 8, 2022 at 12:08

7 Updated link for 2023: developer.ibm.com/articles/pa-dalign – Dennis Jan 20, 2023 at 10:03

8 Answers

Sorted by: Highest score (default)

433

The memory subsystem on a modern processor is restricted to accessing memory at the granularity and alignment of its word size; this is the case for a number of reasons.

Speed

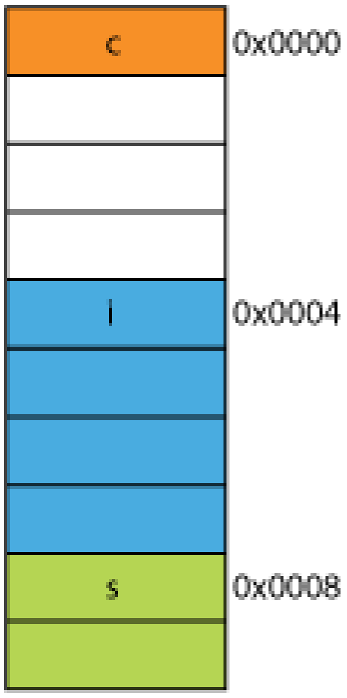
Modern processors have multiple levels of cache memory that data must be pulled through; supporting single-byte reads would make the memory subsystem throughput tightly bound to the execution unit throughput (aka cpu-bound); this is all reminiscent of how PIO mode was surpassed by DMA for many of the same reasons in hard drives.

The CPU **always** reads at its word size (4 bytes on a 32-bit processor), so when you do an unaligned address access — on a processor that supports it — the processor is going to read multiple words. The CPU will read each word of memory that your requested address straddles. This causes an amplification of up to 2X the number of memory transactions required to access the requested data.

Because of this, it can very easily be slower to read two bytes than four. For example, say you have a struct in memory that looks like this:

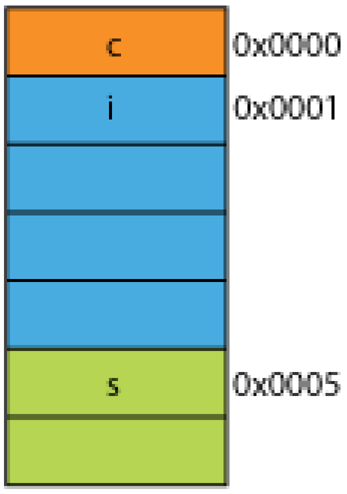
```
struct mystruct {
    char c;  // one byte
    int i;   // four bytes
    short s; // two bytes
}
```

On a 32-bit processor it would most likely be aligned like shown here:



The processor can read each of these members in one transaction.

Say you had a packed version of the struct, maybe from the network where it was packed for transmission efficiency; it might look something like this:



Reading the first byte is going to be the same.

When you ask the processor to give you 16 bits from 0x0005 it will have to read a word from 0x0004 and shift left 1 byte to place it in a 16-bit register; some extra work, but most can handle that in one cycle.

When you ask for 32 bits from 0x0001 you'll get a 2X amplification. The processor will read from 0x0000 into the result register and shift left 1 byte, then read again from 0x0004 into a temporary register, shift right 3 bytes, then `OR` it with the result register.

Range

For any given address space, if the architecture can assume that the 2 LSBs are always 0 (e.g., 32-bit machines) then it can access 4 times more memory (the 2 saved bits can represent 4 distinct states), or the same amount of memory with 2 bits for something like flags. Taking the 2 LSBs off of an address would give you a 4-byte alignment; also referred to as a stride of 4 bytes. Each time an address is incremented it is effectively incrementing bit 2, not bit 0, i.e., the last 2 bits will always continue to be `00`.

This can even affect the physical design of the system. If the address bus needs 2 fewer bits, there can be 2 fewer pins on the CPU, and 2 fewer traces on the circuit board.

Atomicity

The CPU can operate on an aligned word of memory atomically, meaning that no other instruction can interrupt that operation. This is critical to the correct operation of many lock-free data structures and other concurrency paradigms.

Conclusion

The memory system of a processor is quite a bit more complex and involved than described here; a discussion on how an x86 processor actually addresses memory can help (many processors work similarly).

There are many more benefits to adhering to memory alignment that you can read at this IBM article.

A computer's primary use is to transform data. Modern memory architectures and technologies have been optimized over decades to facilitate getting more data, in, out, and between more and faster execution units–in a highly reliable way.

Bonus: Caches

Another alignment-for-performance that I alluded to previously is alignment on cache lines which are (for example, on some CPUs) 64B.

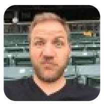
For more info on how much performance can be gained by leveraging caches, take a look at Gallery of Processor Cache Effects; from this question on cache-line sizes

> Understanding of cache lines can be important for certain types of program optimizations. For example, the alignment of data may determine whether an operation touches one or two cache lines. As we saw in the example above, this can easily mean that in the misaligned case, the operation will be twice slower.

Share Improve this answer Follow

edited Jun 8, 2023 at 1:06

answered Dec 19, 2008 at 15:51



joshperry
42.2k ● 16 ● 92 ● 104

1 If I understand correctly, the reason WHY a computer cannot read an unaligned word in one step is because the addesses use 30 bits and not 32 bits?? – GetFree Jun 16, 2014 at 22:50

1 @chux Yes it's true, absolutes never hold. The 8088 is an interesting study of the tradeoffs between speed and cost, it was basically a 16-bit 8086 (which had a full 16-bit external bus) but with only half the bus-lines to save production costs. Because of this the 8088 needed twice the clock cycles to access memory than the 8086 since it had to do two reads to get the full 16-bit word. The interesting part, the 8086 can do a *word aligned* 16-bit read in a single cycle, unaligned reads take 2. The fact that the 8088 had a half-word bus masked this slowdown. – joshperry Jun 22, 2014 at 17:40

2 @joshperry: Slight correction: the 8086 can do a word-aligned 16-bit read in *four* cycles, while unaligned reads take *eight*. Because of the slow memory interface, execution time on 8088-based machines is usually dominated by instruction fetches. An instruction like "MOV AX,BX" is nominally one cycle faster than "XCHG AX,BX", but unless it is preceded or followed by an instruction whose execution takes more than four cycles per code byte, it will take four cycles longer to execute. On the 8086, code fetching can sometimes keep up with execution, but on the 8088 unless one uses... – supercat Mar 1, 2015 at 3:19

4 I think the alignment of `mystruct` is wrong. C structs are always aligned to the alignment of its largest member, thus there should be two additional padding bytes after `s`.
– Martin Dec 16, 2015 at 19:51

2 Very true, @martin. I elided those padding bytes to focus the discussion intra-struct, but perhaps it would be better to include them. – joshperry Dec 16, 2015 at 20:04

83

It's a limitation of many underlying processors. It can usually be worked around by doing 4 inefficient single byte fetches rather than one efficient word fetch, but many language specifiers decided it would be easier just to outlaw them and force everything to be aligned.

There is much more information in this link that the OP discovered.

Share Improve this answer Follow

edited Sep 25, 2023 at 2:37

answered Dec 19, 2008 at 15:20

Paul Tomblin
183k ● 59 ● 323 ● 410

Thanks, @AveMilia, I've updated the answer. – Paul Tomblin Mar 9, 2021 at 21:56

"this link" not present – Dilip Kumar Sep 24, 2023 at 1:18

@DilipKumar link fixed. Never thought when I started on StackOverflow that I'd be maintaining links in 15 year old answers. – Paul Tomblin Sep 25, 2023 at 2:38

you can with some processors (the nehalem can do this), but previously all memory access was aligned on a 64-bit

32

(or 32-bit) line, because the bus is 64 bits wide, you had to fetch 64 bit at a time, and it was significantly easier to fetch these in aligned 'chunks' of 64 bits.

So, if you wanted to get a single byte, you fetched the 64-bit chunk and then masked off the bits you didn't want. Easy and fast if your byte was at the right end, but if it was in the middle of that 64-bit chunk, you'd have to mask off the unwanted bits and then shift the data over to the right place. Worse, if you wanted a 2 byte variable, but that was split across 2 chunks, then that required double the required memory accesses.

So, as everyone thinks memory is cheap, they just made the compiler align the data on the processor's chunk sizes so your code runs faster and more efficiently at the cost of wasted memory.

Share Improve this answer Follow

answered Dec 19, 2008 at 15:31

gbjbaanb
52.6k ● 12 ● 110 ● 154

"Easy and fast if your byte was at the right end, but if it was in the middle of that 64-bit chunk,..." Why can't you fetch a 64 bit starting from the location where you byte is located? – Mehdi Charife Nov 27, 2022 at 2:15

@MehdiCharife because its way easier (and thus cheaper, possibly faster too) to implement the hardware memory controller in bus-sized blocks than in bytes. Access in blocks is better for caching, aligning by byte would complexify and miss the cache a lot more often. – gbjbaanb Nov 27, 2022 at 19:40

"Access in blocks is better for caching" Why can't you access the block starting from the byte you want to read?
– Mehdi Charife Nov 27, 2022 at 19:49

one clarification is fine, but now you're asking questions in the comment section. – gbjbaanb Nov 27, 2022 at 19:52

9

Fundamentally, the reason is because the memory bus has some specific length that is much, much smaller than the memory size.

So, the CPU reads out of the on-chip L1 cache, which is often 32KB these days. But the memory bus that connects the L1 cache to the CPU will have the vastly smaller width of the cache line size. This will be on the order of 128 *bits*.

So:

```
262,144 bits - size of memory
    128 bits - size of bus
```

Misaligned accesses will occasionally overlap two cache lines, and this will require an entirely new cache read in order to obtain the data. It might even miss all the way out to the DRAM.

Furthermore, some part of the CPU will have to stand on its head to put together a single object out of these two different cache lines which each have a piece of the data.

On one line, it will be in the very high order bits, in the other, the very low order bits.

There will be dedicated hardware fully integrated into the pipeline that handles moving aligned objects onto the necessary bits of the CPU data bus, but such hardware may be lacking for misaligned objects, because it probably makes more sense to use those transistors for speeding up correctly optimized programs.

In any case, the second memory read that is sometimes necessary would slow down the pipeline no matter how much special-purpose hardware was (hypothetically and foolishly) dedicated to patching up misaligned memory operations.

Share Improve this answer Follow

answered Mar 1, 2011 at 18:38

DigitalRoss
146k ● 25 ● 252 ● 331

---

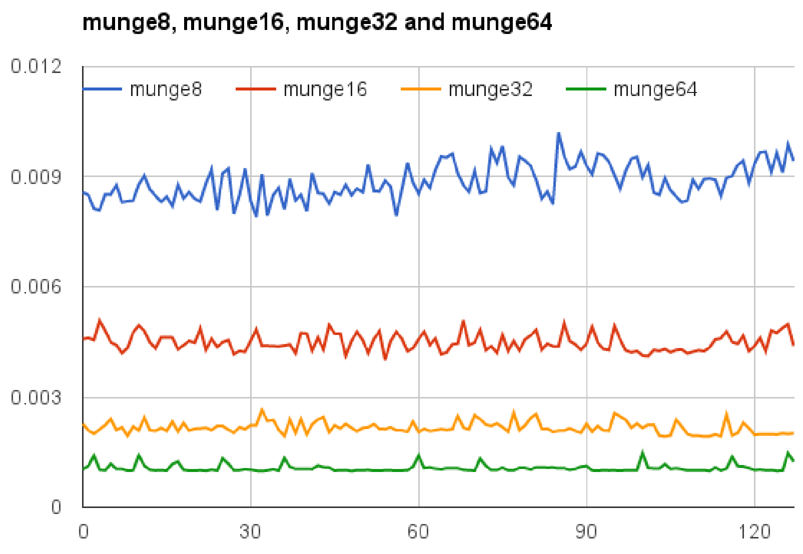
1 *no matter how much special-purpose hardware was (hypothetically and foolishly) dedicated to patching up misaligned memory operations* - Modern Intel CPUs, please stand up and /wave. :P Fully efficient handling of misaligned 256-bit AVX loads (as long as they don't cross a cache-line boundary) is convenient for software. Even split loads aren't too bad, with Skylake greatly improving the penalty for page-split loads/stores, from ~100 cycles down to ~10. (Which will happen if vectorizing over an unaligned buffer, with a loop that doesn't spend extra startup / cleanup code aligning pointers) – Peter Cordes Aug 19, 2020 at 8:53

1 AVX512 CPUs with 512-bit paths between L1d cache and load/store execution units do suffer significantly more from

misaligned pointers because *every* load is misaligned, instead of every other. – Peter Cordes Aug 19, 2020 at 8:53

4

@joshperry has given an excellent answer to this question. In addition to his answer, I have some numbers that show graphically the effects which were described, especially the 2X amplification. Here's a link to a Google spreadsheet showing what the effect of different word alignments look like. In addition here's a link to a Github gist with the code for the test. The test code is adapted from the article written by Jonathan Rentzsch which @joshperry referenced. The tests were run on a Macbook Pro with a quad-core 2.8 GHz Intel Core i7 64-bit processor and 16GB of RAM.



Share Improve this answer Follow edited Apr 6, 2016 at 14:38

answered Apr 6, 2016 at 14:26

adino
1,310 ● 13 ● 17

5 What do `x` and `y` co-ordinates mean ? – KRoy Oct 2, 2018 at 20:07

2 What generation core i7? (Thanks for posting links to code!) – Nick Desaulniers Jan 7, 2019 at 7:11

OMG! memcpy function is specially optimized to work with unaligned data! Such tests has no sense! – Kirill Frolov Dec 29, 2021 at 15:28

3

If you have a 32bit data bus, the address bus address lines connected to the memory will start from A_2, so only 32bit aligned addresses can be accessed in a single bus cycle.

So if a word spans an address alignment boundary - i.e. A_0 for 16/32 bit data or A_1 for 32 bit data are not zero, two bus cycles are required to obtain the data.

Some architectures/instruction sets do not support unaligned access and will generate an exception on such attempts, so compiler generated unaligned access code requires not just additional bus cycles, but additional instructions, making it even less efficient.

Share Improve this answer Follow

answered Jun 19, 2014 at 20:10

Clifford
93.3k ● 14 ● 91 ● 170

2

If a system with byte-addressable memory has a 32-bit-wide memory bus, that means there are effectively four byte-wide memory systems which are all wired to read or write the same address. An aligned 32-bit read will require information stored in the same address in all four memory systems, so all systems can supply data simultaneously. An unaligned 32-bit read would require some memory systems to return data from one address, and some to return data from the next higher address. Although there are some memory systems that are optimized to be able to fulfill such requests (in addition to their address, they effectively have a "plus one" signal which causes them to use an address one higher than specified) such a feature adds considerable cost and complexity to a memory system; most commodity memory systems simply cannot return portions of different 32-bit words at the same time.

Share Improve this answer Follow

answered Jun 15, 2011 at 20:21
supercat
80.8k ● 9 ● 174 ● 220

0

On PowerPC you can load an integer from an odd address with no problems.

Sparc and I86 and (I think) Itatnium raise hardware exceptions when you try this.

One 32 bit load vs four 8 bit loads isnt going to make a lot of difference on most modern processors. Whether the data is already in cache or not will have a far greater effect.

Share Improve this answer Follow

edited Jun 19, 2014 at 19:53
Clifford
93.3k ● 14 ● 91 ● 170

answered Dec 19, 2008 at 15:28
James Anderson
27.4k ● 7 ● 54 ● 80

1 On Sparc, this was a "Bus error", hence the chapter "Bus error, Take the train" in Peter Van der Linden's "Expert C

Programming: Deep C Secrets" – jjg Apr 1, 2020 at 19:35

It says here that the PowerPC can handle 32-bit unaligned data raises a hardware exception for 64-bit data. – Harsh Aug 21, 2020 at 10:44