Why is x86 ugly? aka Why is x86 considered inferior when compared to others? [closed]

Recently I've been reading some SO archives and encountered statements against x86 architecture.

- Why do we need different CPU architecture for server & mini/mainframe & mixed-core? says "*PC architecture is a mess, any OS developer would tell you that.*"
- Is learning Assembly Language worth the effort? says "*Realize that the x86 architecture is horrible at best*"
- http://forums.anandtech.com/showthread.php?t=976577 says "*Most colleges teach assembly on something like MIPS because it's much simpler to understand, x86 assembly is really ugly*"

and many more comments like

- *Compared to most architectures, X86 sucks pretty badly.*
- *It's definitely the conventional wisdom that X86 is inferior to MIPS, SPARC, and PowerPC x86 is ugly*

I tried searching but didn't find any reasons. I don't find x86 bad probably because this is the only architecture I'm familiar with.

Can someone kindly give me reasons for considering x86 ugly/bad/inferior compared to others.

x86 mips x86-64 computer-architecture assembly

edited May 1 '12 at 4:12

Ryan Tenney
1,556 ● 2 ● 9 ● 25

asked Apr 21 '10 at 2:14

claws
10.4k ● 23 ● 87 ● 154

closed as not constructive by casperOne ♦ Mar 15 '12 at 0:34

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

If this question can be reworded to fit the rules in the help center, please edit the question.

1 I'm going with S&A on the basis of the answers so far, but I'll note in passing that CISC isn't a problem for the m68k instruction set. x86 is what it is, and you can keep it. – dmckee Apr 21 '10 at 5:04

what is "S&A"? " CISC isn't a problem for the m68k instruction set." -- Why not? – claws Apr 21 '10 at 10:37

2 The motorala 68000 series chips have a highly CISC architecture, but they have a uniform, fairly orthogonal, and very easy instruction set. Why the difference from x86? I don't know. But take note that there is a big difference between complexity *in the chip* and complexity in the instruction set (i.e. in the interface that an assembly programmer sees). – dmckee Apr 21 '10 at 15:48

1 +1 for a very interesting question. – Turing Complete Jun 30 '10 at 8:41

1 +1: great links – user295190 Oct 21 '10 at 0:28

10 Answers

Couple of possible reasons for it:

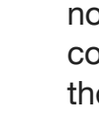
1. x86 is a relatively old architecture (its progenitors were 8086s, after all)
2. x86 has evolved significantly several times, but hardware is required to maintain backwards compatibility with old binaries. Modern x86 hardware still has to contain support hardware to run 16 bit code natively. Additionally, several memory-addressing models have to be in place to allow older code to interoperate on the same processor without modification. This can be confusing to some.
3. In some respects, the x86 isn't inferior, it's simply different than how perhaps every other

processor architecture operates. For example, input/output is handled as memory mapping on the vast majority of architectures, but not on the x86.

4. x86 is a CISC machine, which for a long time meant it was slower than RISC machines. Nowadays the x86 is translated into RISC-style instructions before it's executed anyway, so this matters little in practice.
5. The x86 has a very small number of registers compared to most other architectures. Again, most modern x86 chips are RISCs internally so this is less of a problem, but that's not transparent looking at the ISA.
6. x86 assembly code is complicated because x86 is a complicated architecture with many features. I can fit the features available on a MIPS machine on a single letter sized piece of paper. Even a simple lookup table of x86 instructions fills several pages. While that doesn't necessarily make MIPS superior, for teaching an introduction to assembler class it'd make sense to start with a simpler architecture.
7. The x86 uses variable-length opcodes, which add hardware complexity with respect to the parsing of instructions.

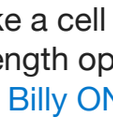
EDIT: This is not supposed to be a *bash the x86!* party. I had little choice but to do some amount of bashing given the way the question's worded. But with the exception of (1), all these things were done for good reasons (see comments). Intel designers aren't stupid -- they wanted to achieve some things with their architecture, and these are some of the taxes they had to pay to make those things a reality.

edited Feb 18 '13 at 20:05



0x499602D2
47.1k ● 14 ● 64 ● 130

answered Apr 21 '10 at 2:26



Billy ONeal
54.7k ● 18 ● 157 ● 369

13 It's a tradeoff. It's a strength in that the binary size might be smaller, but it's a weakness in that you need to have very complicated hardware to implement a parser for these instructions. The vast majority of instructions are the same size anyway -- most of the reason for variable length opcodes on x86 is for when they decided to add features and found they couldn't represent what they wanted in the number of bits they had to work with. The vast majority of people aren't concerned with binary size nearly as much as hardware complexity or power consumption. – Billy ONeal Apr 21 '10 at 2:34

6 @Joey Adams: Contrast the x86's variable length instructions with the ARM's Thumb Mode (en.wikipedia.org/wiki/ARM_architecture#Thumb). Thumb Mode results in significantly smaller object code for the ARM because the shorter instructions map directly to normal instructions. But since there is a 1:1 mapping between the larger instructions and the smaller ones, the parsing hardware is simple to implement. The x86's variable length instructions don't have these benefits because they weren't designed that way in the first place. – Billy ONeal Apr 21 '10 at 2:42

5 (6) Not every op-code needs to be used by every program, but dammit, when I need SSE3, I'm glad I have it. – Chris Kaminski Apr 21 '10 at 3:08

3 @Chris Kaminski: How does that not affect the hardware? Sure, on a modern full sized computer nobody's going to care, but if I'm making something like a cell phone, I care more about power consumption than almost anything else. The variable length opcodes don't increase execution time but the decode hardware still requires power to operate. – Billy ONeal Apr 21 '10 at 3:17

3 Which is one of the things that make the x86 instruction set so ugly, since it can't decide if it's an accumulator or a register-file based architecture (though this was mostly fixed with the 386, which made the instruction set much more orthogonal, irregardless of whatever the 68k fans tell you). – ninjalj Jul 16 '11 at 11:45

The main knock against x86 in my mind is its CISC origins - the instruction set contains a lot of implicit interdependencies. These interdependencies make it difficult to do things like instruction reordering on the chip, because the artifacts and semantics of those interdependencies must be preserved for each instruction.

For example, most x86 integer add & subtract instructions modify the flags register. After performing an add or subtract, the next operation is often to look at the flags register to check for overflow, sign bit, etc. If there's another add after that, it's very difficult to tell whether it's safe to begin execution of the 2nd add before the outcome of the 1st add is known.

On a RISC architecture, the add instruction would specify the input operands and the output register(s), and everything about the operation would take place using only those registers. This makes it much easier to decouple add operations that are near each other because there's no bloomin' flags register forcing everything to line up and execute single file.

The DEC Alpha AXP chip, a MIPS style RISC design, was painfully spartan in the instructions

dependencies. There was no hardware-defined stack register. There was no hardware-defined flags register. Even the instruction pointer was OS defined - if you wanted to return to the caller, you had to work out how the caller was going to let you know what address to return to. This was usually defined by the OS calling convention. On the x86, though, it's defined by the chip hardware.

Anyway, over 3 or 4 generations of Alpha AXP chip designs, the hardware went from being a literal implementation of the spartan instruction set with 32 int registers and 32 float registers to a massively out of order execution engine with 80 internal registers, register renaming, result forwarding (where the result of a previous instruction is forwarded to a later instruction that is dependent on the value) and all sorts of wild and crazy performance boosters. And with all of those bells and whistles, the AXP chip die was still considerably smaller than the comparable Pentium chip die of that time, and the AXP was a hell of a lot faster.

You don't see those kinds of bursts of performance boosting things in the x86 family tree largely because the x86 instruction set's complexity makes many kinds of execution optimizations prohibitively expensive if not impossible. Intel's stroke of genius was in giving up on implementing the x86 instruction set in hardware anymore - all modern x86 chips are actually RISC cores that to a certain degree interpret the x86 instructions, translating them into internal microcode which preserves all the semantics of the original x86 instruction, but allows for a little bit of that RISC out-of-order and other optimizations over the microcode.

I've written a lot of x86 assembler and can fully appreciate the convenience of its CISC roots. But I didn't fully appreciate just how complicated x86 was until I spent some time writing Alpha AXP assembler. I was gobsmacked by AXP's simplicity and uniformity. The differences are enormous, and profound.

answered Apr 21 '10 at 3:04

dthorpe

24.2k ● 1 ● 37 ● 90

4 I'll listen to no bashing of CISC *per se* unless and until you can explain m68k. – dmckee Apr 21 '10 at 5:04

@dmckee : I'm the OP. I don't know anything about m68k But can you explain why doesn't these things hold for m68k? – claws Apr 21 '10 at 10:33

1 I'm not familiar with the m68k, so I can't critique it. – dthorpe Apr 22 '10 at 18:05

2 I don't think this answer is bad enough to downvote, but I do think the whole "RISC is smaller and faster than CISC" argument isn't really relevant in the modern era. Sure, the AXP might have been a hell of a lot faster for it's time, but the fact of the matter is that modern RISCs and modern CISCs are about the same when it comes to performance. As I said in my answer, the slight power penalty for x86 decode is a reason not to use x86 for something like a mobile phone, but that's little argument for a full sized desktop or notebook. – Billy ONeal Apr 23 '10 at 3:31

2 @Billy: size is more than just code size or instruction size. Intel pays quite a penalty in chip surface area to implement the hardware logic for all those special instructions, RISC microcode core under the hood or not. Size of the die directly impacts cost to manufacture, so it's still a valid concern with modern system designs. – dthorpe Apr 23 '10 at 16:56

The x86 architecture dates from the design of the 8008 microprocessor and relatives. These CPUs were designed in a time when memory was slow and if you could do it on the CPU die, it was often a *lot* faster. However, CPU die-space was also expensive. These two reasons are why there are only a small number of registers that tend to have special purposes, and a

complicated instruction set with all sorts of gotchas and limitations.

Other processors from the same era (e.g. the 6502 family) also have similar limitations and quirks. Interestingly, both the 8008 series and the 6502 series were intended as embedded controllers. Even back then, embedded controllers were expected to be programmed in assembler and in many ways catered to the assembly programmer rather than the compiler writer. (Look at the VAX chip for what happens when you cater to the compiler write.) The designers didn't expect them to become general purpose computing platforms; that's what things like the predecessors of the POWER archicture were for. The Home Computer revolution changed that, of course.

answered Apr 21 '10 at 3:03

staticsan

16k ● 3 ● 34 ● 61

3 +1 for the only answer here from someone who actually seems to have historical background on the issue. – Billy ONeal Apr 23 '10 at 3:34

1 there are other cisc processors, coming out from the 8 bit era (m68k can be considered the descending of 6800; z8000 or alike from z80...) that evolved into "better" cisc, so it's not a good excuse. Extinction is the only path to real evolution, and trying to be backward compatible is a defect and limitation, not a feature. The Home Computer status is late if you think about Home Computer revolution promises. And I believe part of the guiltiness is for the "backward compatibility" issue, which is about *marketing*, not technology. –

ShinTakezou Jun 20 '10 at 19:13

2 Memory has always been slow. It is possibly (relatively speaking) slower today than it was when I began with Z80s and CP/M in 1982. Extinction is not the only path of evolution because with extinction that particular evolutionary direction stops. I would say the x86 has adapted well in its 28 year (so far existence). – Olof Forshell Dec 7 '10 at 23:18

3 Memory speeds briefly hit near parity with CPUs around the time of the 8086. The 9900 from Texas Instruments has a design that only works because this happened. But then the CPU raced ahead again and has stayed there. Only now, there are caches to help manage this. – staticsan Dec 9 '10 at 6:13

2 @Olof Forshell: It was assembler compatible in that 8080 assembly code could translate into 8086 code. From that point of view, it was 8080 plus extensions, much like you could view 8080 as 8008 plus extensions. – David Thornley Jan 31 '11 at 21:48

I have a few additional aspects here:

Consider the operation "a=b/c" x86 would implement this as

```
mov eax,b
xor edx,edx
div dword ptr c
mov a,eax
```

As an additional bonus of the div instruction edx will contain the remainder.

A RISC processor would require first loading the addresses of b and c, loading b and c from memory to registers, doing the division and loading the address of a and then storing the result. Dst,src syntax:

```
mov r5,addr b

mov r5,[r5]
mov r6,addr c
mov r6,[r6]
div r7,r5,r6
mov r5,addr a
mov [r5],r7
```

Here there typically won't be a remainder.

If any variables are to be loaded through pointers both sequences may become longer though this is less of a possibility for the RISC because it may have one or more pointers already loaded in another register. x86 has fewer register so the likelihood of the pointer being in one of them is smaller.

Pros and cons:

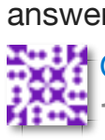
The RISC instructionss may be mixed with surrounding code to improve instruction scheduling, this is less of a possibility with x86 which instead does this work (more or less well depending on the sequence) inside the CPU itself. The RISC sequence above will typically be 28 bytes long (7 instructions of 32-bit/4 byte width each) on a 32-bit architecture. This will cause the off-chip memory to work more when fetching the instructions (seven fetches). The denser x86 sequence contains fewer instructions and though their widths vary you're probably looking at an average of 4 bytes/instruction there too. Even if you have instruction caches to speed this up seven fetches means that you will have a deficit of three elsewhere to make up for compared to the x86.

The x86 architecture with fewer registers to save/restore means that it will probably do thread switches and handle interrupts faster than RISC. More registers to save and restore requires more temporary RAM stack space to do interrupts and more permanent stack space to store thread states. These aspects should make x86 a better candidate for running pure RTOS.

On a more personal note I find it more difficult to write RISC assembly than x86. I solve this by writing the RISC routine in C, compiling and modifying the generated code. This is more efficient from a code production standpoint and probably less efficient from an execution standpoint. All those 32 registers to keep track of. With x86 it is the other way around: 6-8 registers with "real" names makes the problem more manageable and instills more confidence that the code produced will work as expected.

Ugly? That's in the eye of the beholder. I prefer "different."

answered Dec 7 '10 at 10:00



Olof Forshell

1,980 ● 9 ● 18

1 Great counter-point! – user295190 Dec 7 '10 at 22:18

a, b and c in my examples should be viewed as memory-based variables and not to immediate values. – Olof Forshell Dec 20 '10 at 11:30

... "dword ptr" is used to specifiy the size of a variable whose size is not known if, for instance, it is simply

declared as external or if you've been lazy. – Olof Forshell Dec 20 '10 at 13:14

1 That isn't the first time I heard the suggestion to write it in C first, and then distill it into assembler. That definitely helps – Joe Plante Sep 27 '14 at 2:19

In the early days all processors were RISC. CISC came about as a mitigation strategy for ferric core memory systems that were VERY slow, thus CISC, with fewer, more powerful instructions, put less stress on the memory subsystem, and made better use of bandwidth. Likewise, registers were originally thought of as on-chip, in-CPU memory locations for doing accumulations. The last time I seriously benchmarked a RISC machine was 1993 - SPARC and HP Prisim. SPARC was horrible across the board. Prisim was up to 20x as fast as a 486 on add/sub/mul but sucked on transcendentals. CISC is better. – RocketRoy Oct 22 '14 at 6:52

I think this question has a false assumption. It's mainly just RISC-obsessed academics who call x86 ugly. In reality, the x86 ISA can do in a single instruction operations which would take 5-6 instructions on RISC ISAs. RISC fans may counter that modern x86 CPUs break these "complex" instructions down into microops; however:

1. In many cases that's only partially true or not true at all. The most useful "complex" instructions in x86 are things like `mov %eax, 0x1c(%esp,%edi,4)` i.e. addressing modes, and these are not broken down.
2. What's often more important on modern machines is not the number of cycles spent (because most tasks are not cpu-bound) but the instruction cache impact of code. 5-6 fixed-size (usually 32bit) instructions will impact the cache a lot more than one complex instruction that's rarely more than 5 bytes.

x86 really absorbed all the good aspects of RISC about 10-15 years ago, and the remaining qualities of RISC (actually the *defining* one - the minimal instruction set) are harmful and undesirable.

Aside from the cost and complexity of manufacturing CPUs and their energy requirements, x86 is **the best ISA**. Anyone who tells you otherwise is letting ideology or agenda get in the way of their reasoning.

On the other hand, if you are targetting embedded devices where the cost of the CPU counts, or embedded/mobile devices where energy consumption is a top concern, ARM or MIPS probably makes more sense. Keep in mind though you'll still have to deal with the extra ram and binary size needed to handle code that's easily 3-4 times larger, and you won't be able to get near the performance. Whether this matters depends a lot on what you'll be running on it.

answered Jul 16 '11 at 13:36

R..

106k ● 11 ● 141 ● 330

where energy consumption is a top concern, ARM or MIPS probably makes more sense... so, if there is at least one aspect where ARM or MIPS make more sense, doesn't it make x86 **not necessarily** the best ISA? – Shahbaz Sep 24 '13 at 16:59

That's why I qualified "the best" with "aside from the cost...and their energy requirements". – R.. Sep 24 '13 at 17:41

1 I think Intel's throttling down the CPU speed, and smaller die sizes have largely eliminated the power differential. The new Celeron dual 64-bit CPU with 64k L1 and 1MB L2 caches is a 7.5 watt chip. It's my "Starbucks" hangout machine, and the battery life is ridiculously long and will run rings around a P6 machine. As a guy doing mostly floating point computations I gave up on RISC a long time ago. It just crawls. SPARC in particular was atrociously glacial. The perfect example of why RISC sucks was the Intel i860 CPU. Intel never went THERE again. – RocketRoy Oct 22 '14 at 7:04

@RocketRoy: 7.5 watt isn't really acceptable for a device that's powered 24/7 (and not performing useful computations the whole time) or running off a 3.7v/2000mAh battery. – R.. Oct 22 '14 at 13:18

You size the CPU to the task at hand and resources available. That doesn't rule out a CISC or x86 architecture, it just means it may be overkill for some apps. – RocketRoy Oct 22 '14 at 22:45

x86 assembler language isn't so bad. It's when you get to the machine code that it starts to get really ugly. Instruction codes, addressing modes, etc are much more complicated than the ones for most RISC CPUs. (There's actually an addressing mode for [BX+SI], iirc, but not one for [AX+BX]. Inconsistencies like that complicate register usage, since you need to ensure your value's in a register that you can use as you need to.

There's also the leftovers from the olden days, when Intel was trying to make x86 the ultimate processor. Instructions a couple of bytes long that performed tasks that no one actually does any more, cause they were frankly too freaking slow or complicated. The ENTER and LOOP instructions, for two examples -- note the C stack frame code is like "push ebp; mov ebp, esp" and not "enter" for most compilers.

answered Apr 21 '10 at 2:47

answered Apr 21 '10 at 2:47

cHao

46k • 9 • 61 • 109

1 I believe the "enter" versus "push/mov" issue arose because on some processors, "push/mov" is faster. On some processors, "enter" is faster. C'est la vie. – Dietrich Epp Apr 21 '10 at 3:23

2 When I was forced to a x86 based machine and started to take a look at it (having m68k background),I started to feel asm programming frustrating, ... like if I've learned programming with a language like C, and then be forced to get in touch with asm... you "feel" you lose power of expression, ease, clarity, "coherence", "intuitionability".I am sure that if I would have started asm programming with x86,I would have thought it is not so bad...maybe... I did also MMIX and MIPS, and their "asm lang" is far better than x86 (if this is the right PoV for the Q, but maybe it is not) – ShinTakezou Jun 20 '10 at 19:25

I'm not an expert, but it seems that many of the features why people don't like it can be the reasons it performs well. Several years ago, having registers (instead of a stack), register frames, etc. were seen as nice solutions for making the architecture seem simpler to humans. However, nowadays, what matters is cache performance, and x86's variable-length words allow it to store more instructions in cache. The "instruction decode", which I believe opponents pointed out once took up half the chip, is not nearly so much that way anymore.

I think parallelism is one of the most important factors nowadays -- at least for algorithms that already run fast enough to be usable. Expressing high parallelism in software allows the hardware to amortize (or often completely hide) memory latencies. Of course, the farther reaching architecture future is probably in something like quantum computing.

I have heard from nVidia that one of Intel's mistakes was that they kept the binary formats close to the hardware. CUDA's PTX does some fast register use calculations (graph coloring), so nVidia can use a register machine instead of a stack machine, but still have an upgrade path that doesn't break all old software.

answered Apr 21 '10 at 2:43

gatoatigrado

6,748 • 3 • 34 • 86

6 RISC was not designed with human developers in mind. One of the ideas behind RISC was to offload some of the complexity of the chip onto whoever wrote the assembly, ideally the compiler. More registers meant less memory usage and fewer dependencies between instructions, allowing deeper pipelines and higher performance. Note that x86-64 has twice as many general registers as x86, and this alone is responsible for significant performance gains. And instructions on most x86 chips are decoded before they are cached, not after (so size doesn't matter here). – Dietrich Epp Apr 21 '10 at 3:15

3 @Dietrich Epp: That's not entirely true. The x86-64 does have more registers visible in the ISA, but modern x86 implementations usually have a RISC style register file which is mapped to the ISA's registers on demand to speed up execution. – Billy ONeal Apr 21 '10 at 3:22

"I have heard from nVidia that one of Intel's mistakes was that they kept the binary formats close to the hardware." -- I didn't get this and the CUDA's PTX part. – claws Apr 21 '10 at 3:57

1 @Dietrech Epp: "And instructions on most x86 chips are decoded before they are cached, not after" That's not true. They are cached before they are decoded. I believe the Pentium 4 had an additional trace cache that cached after decode, but that's been discontinued. – Nathan Fellman Apr 23 '10 at 8:02

that is not true, the newest "sandy bridge" processors use a kind of a trace cache (like that for the pentium 4, oh that old boy :D), so technologies go away and come back... – Quonux Apr 18 '11 at 0:15

Besides the reasons people have already mentioned:

- x86-16 had a rather strange memory addressing scheme which allowed a single memory location to be addressed in up to 4096 different ways, limited RAM to 1 MB, and forced programmers to deal with two different sizes of pointers. Fortunately, the move to 32-bit made this feature unnecessary, but x86 chips still carry the cruft of segment registers.
- While not a fault of x86 *per se*, x86 calling conventions weren't standardized like MIPS was (mostly because MS-DOS didn't come with any compilers), leaving us with the mess of `__cdecl`, `__stdcall`, `__fastcall`, etc.

answered Apr 21 '10 at 3:32

dan04

37.4k • 10 • 93 • 131

Hmm.. when I think of x86 competitors, I don't think of MIPS. ARM or PowerPC maybe.... – Billy ONeal Apr 23 '10 at 3:35

@Billy: x86 has been around near forever. At one time MIPS was an x86 competitor. As I remember x86 had its work cut out to get to a level where it was competitive with MIPS. (Back when MIPS and SPARC were fighting it out in the workstation arena.) – Shannon Severance Jun 17 '10 at 23:01

@Shannon Severance: Just because something once was does not mean something that is. – Billy ONeal Jun 17 '10 at 23:09

2 @supercat: what people in the era of the flat x86-32 memory model tend to forget is that 16 bits means 64k of memory (anyone who bothers doing the math will understand that magic isn't possible, that the 8086 wasn't a nasty punishment for unsuspecting programmers). There are few ways to get around 64k but the 8086 solution was a good compromise. – Olof Forshell Jun 12 '13 at 5:59

1 @OlofForshell: I think many people bemoaned the fact that the 8086 wasn't as nice as the 68000 (which had a 16MB linear addressing space and a clear path to 4 gigs). Certainly going to a 32-bit processor will make it easier to access more than 64K, but the 8086 is a 16-bit architecture which was designed to be a step up from the 8-bit 8080. I see no reason Intel should have leapt directly from an 8-bit to a 32-bit one. – supercat Jun 12 '13 at 16:30

I think you'll get to part of the answer if you ever try to write a compiler that targets x86, or if you write an x86 machine emulator, or even if you try to implement the ISA in a hardware design.

Although I understand the "x86 is ugly!" arguments, I still think it's more *fun* writing x86 assembly than MIPS (for example) - the latter is just plain tedious. It was always meant to be nice to compilers rather than to humans. I'm not sure a chip could be more hostile to compiler writers if it tried...

The ugliest part for me is the way (real-mode) segmentation works - that any physical address has 4096 segment:offset aliases. When last did you *need* that? Things would have been so much simpler if the segment part were strictly higher-order bits of a 32-bit address.

answered May 14 '10 at 16:41

Bernd Jendrissek
782 ● 3 ● 11

m68k is a lot funnier, and nice to humans far more than x86 (which can't seem so "human" to many m68k programmers), if the right PoV is the way human can write code in those assembly. – ShinTakezou Jun 20 '10 at 19:34

The segment:offset addressing was an attempt to stay compatible to some extent with the CP/M - world. One of the worst decisions ever. – Turing Complete Jun 30 '10 at 8:47

@Turing Complete: segment:offset was NOT primarily an attempt to stay compatible with the CP/M world. What it was was a very successful attempt to allow a 16 bit processor to address more than 64 KBytes by placing code, data, stack and other memory areas in different segments. – Olof Forshell Jan 14 '11 at 11:30

1 In reality placing data and stack in different segments was utterly useless for C; it was only usable for asm. In C, a pointer can point to data with static, automatic, or dynamically allocated storage duration, so there's no way to elide the segment. Maybe it was useful for Pascal or Fortran or something, but not for C, which was already the dominant language at the time... – R.. Jul 20 '11 at 20:20

1 @Bernd: The reason fs/gs were chosen for thread-local storage is not that segment registers are good for this. It's just that x86 is seriously starved for registers, and the segment registers were unused. A general-purpose register pointing to the thread structure would have worked just as well, and in fact many RISC systems with more registers use one as a thread pointer. – R.. Aug 6 '11 at 12:17

---

1. x86 has a very, very limited set of general purpose registers
2. it promotes a very inefficient style of development on the lowest level (CISC hell) instead of an efficient load / store methodology
3. Intel made the horrifying decision to introduce the plainly stupid segment / offset - memory adressing model to stay compatible with (at this time already!) outdated technology
4. At a time when everyone was going 32 bit, the x86 held back the mainstream PC world by being a meager 16 bit (most of them - the 8088 - even only with 8 bit external data paths, which is even scarier!) CPU

For me (and I'm a DOS veteran that has seen each and every generation of PCs from a developers perspective!) point 3. was the worst.

Imagine the following situation we had in the early 90s (mainstream!):

a) An operating system that had insane limitations for legacy reasons (640kB of easily accessible RAM) - DOS

b) An operating system extension (Windows) that could do more in terms of RAM, but was limited when it came to stuff like games, etc... and was not the most stable thing on Earth (luckily this changed later, but I'm talking about the early 90s here)

c) Most software was still DOS and we had to create boot disks often for special software,

because there was this EMM386.exe that some programs liked, others hated (especially gamers - and I was an AVID gamer at this time - know what I'm talking about here)

d) We were limited to MCGA 320x200x8 bits (ok, there was a bit more with special tricks, 360x480x8 was possible, but only without runtime library support), everything else was messy and horrible ("VESA" - lol)

e) But in terms of hardware we had 32 bit machines with quite a few megabytes of RAM and VGA cards with support of up to 1024x768

Reason for this bad situation?

A simple design decision by Intel. Machine instruction level (NOT binary level!) compatibility to something that was already dying, I think it was the 8085. The other, seemingly unrelated problems (graphic modes, etc...) were related for technical reasons and because of the very narrow minded architecture the x86 platform brought with itself.

Today, the situation is different, but ask any assembler developer or people who build compiler backends for the x86. The insanely low number of general purpose registers is nothing but a horrible performance killer.

answered Jun 30 '10 at 8:39

Turing Complete

548 ● 1 ● 8 ● 19

The only major problems with the 8086 segmented architecture was that there was only one non-dedicated segment register (ES), and that programming languages were not designed to work with it effectively. The style of scaled addressing it uses would work very well in an object-oriented language which does not expect objects to be able to start at arbitrary addresses (if one aligns objects on paragraph boundaries, object references will only need to be two bytes rather than four). If one compares early Macintosh code to PC code, the 8086 actually looks pretty good compared to 68000. – supercat Feb 9 '14 at 1:05