

# Is < faster than <=?

Asked 12 years, 3 months ago    Modified 2 years, 3 months ago

Viewed 153k times



1777



Is `if (a < 901)` faster than `if (a <= 900)` ?

Not exactly as in this simple example, but there are slight performance changes on loop complex code. I suppose this has to do something with generated machine code in case it's even true.

c++

c

performance

assembly

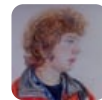
relational-operators

Share

Improve this question

Follow

edited Nov 26, 2021 at 10:29



ivaigult

6,657 ● 7 ● 43 ● 73


asked Aug 27, 2012 at 2:10



Vinícius

15.7k ● 3 ● 33 ● 54

197 I see no reason why this question should be closed (and especially not deleted, as the votes are currently showing) given its historical significance, the quality of the answer, and the fact that the other top questions in [performance](#) remain open. At most it should be locked. Also, even if the question itself is misinformed/naive, the fact that it appeared in a book means that the original misinformation exists out there in "credible" sources somewhere, and this

question is therefore constructive in that it helps to clear that up. – [Jason C](#) Mar 22, 2014 at 23:49 

---

10 It was true on the 8086. – [Joshua](#) Nov 15, 2016 at 16:28

---

2 All of the answers seem to be about C, whereas the question is tagged C++ -- where there is simply no telling, what `<` and `<=` might mean for any given `a` ...  
– [Mikhail T.](#) Jan 26, 2021 at 23:01

---

1 @Joshua: 8086 has `jg` and `jge` (and the equivalent unsigned branch conditions). So no, it's not faster on x86 and never has been, if we're talking about normal compare-and-branch. [Assembly - JG/JNLE/JL/JNGE after CMP](#) - those instructions have all existed since 8086. I'd suggest removing your earlier comment that has incorrect information, or clarify what special case you meant. In asm, `<` allows smaller immediate constants [that can save a byte, so GCC transforms for you](#). – [Peter Cordes](#) Nov 26, 2021 at 10:47

---

@PeterCordes: For some reason the compiler I learned on would always generate `jle +2 jmp ...` instead of `jge` when compiling for x86, but had no problem generating `jg ...` directly. As you have pointed out, this is pretty dumb and I'm reasonably certain modern compilers don't do this. But you put up with the compilers you had.  
– [Joshua](#) Nov 26, 2021 at 16:16

---

15 Answers

Sorted by:

Highest score (default)



1845

No, it will not be faster on most architectures. You didn't specify, but on x86, all of the integral comparisons will be typically implemented in two machine instructions:

- A `test` or `cmp` instruction, which sets `EFLAGS`



- And a `Jcc` [\(jump\) instruction](#), depending on the comparison type (and code layout):
- `jne` - Jump if not equal --> `ZF = 0`
- `jz` - Jump if zero (equal) --> `ZF = 1`
- `jg` - Jump if greater --> `ZF = 0` and `SF = 0F`
- (etc...)

---

**Example** (Edited for brevity) Compiled with `$ gcc -m32 -S -masm=intel test.c`

```
if (a < b) {  
    // Do something 1  
}
```

Compiles to:

```
mov     eax, DWORD PTR [esp+24]    ; a  
cmp     eax, DWORD PTR [esp+28]    ; b  
jge     .L2                        ; jump if a is greater or equal to b  
; Do something 1  
.L2:
```

And

```
if (a <= b) {  
    // Do something 2  
}
```

Compiles to:

```

mov     eax, DWORD PTR [esp+24]    ; a
cmp     eax, DWORD PTR [esp+28]    ; b
jg      .L5                        ; jump if a i
; Do something 2
.L5:

```

So the only difference between the two is a `jg` versus a `jge` instruction. The two will take the same amount of time.

---

I'd like to address the comment that nothing indicates that the different jump instructions take the same amount of time. This one is a little tricky to answer, but here's what I can give: In the [Intel Instruction Set Reference](#), they are all grouped together under one common instruction, `Jcc` (Jump if condition is met). The same grouping is made together under the [Optimization Reference Manual](#), in Appendix C. Latency and Throughput.

**Latency** — The number of clock cycles that are required for the execution core to complete the execution of all of the  $\mu$ ops that form an instruction.

**Throughput** — The number of clock cycles required to wait before the issue ports are free to accept the same instruction again. For many instructions, the throughput of an instruction can be significantly less than its latency

The values for `Jcc` are:

	Latency	Throughput
<code>Jcc</code>	N/A	0.5

with the following footnote on `Jcc`:

7. Selection of conditional jump instructions should be based on the recommendation of section Section 3.4.1, “Branch Prediction Optimization,” to improve the predictability of branches. When branches are predicted successfully, the latency of `jcc` is effectively zero.

So, nothing in the Intel docs ever treats one `Jcc` instruction any differently from the others.

If one thinks about the actual circuitry used to implement the instructions, one can assume that there would be simple AND/OR gates on the different bits in `EFLAGS`, to determine whether the conditions are met. There is then, no reason that an instruction testing two bits should take any more or less time than one testing only one (Ignoring gate propagation delay, which is much less than the clock period.)

---

**Edit: Floating Point**

This holds true for x87 floating point as well: (Pretty much same code as above, but with `double` instead of `int`.)

```
        fld      QWORD PTR [esp+32]
        fld      QWORD PTR [esp+40]
        fucomip  st, st(1)                ; Compare ST(0)
PF, ZF in EFLAGS
        fstp     st(0)
        seta     al                        ; Set al if abc
        test     al, al
        je       .L2
        ; Do something 1
.L2:

        fld      QWORD PTR [esp+32]
        fld      QWORD PTR [esp+40]
        fucomip  st, st(1)                ; (same thing a
        fstp     st(0)
        setae    al                        ; Set al if abc
        test     al, al
        je       .L5
        ; Do something 2
.L5:

        leave
        ret
```

Share Improve this answer

Follow

edited Nov 15, 2021 at 14:48



tolik518

130 ● 2 ● 15

answered Aug 27, 2012 at 2:13



Jonathon Reinhart

137k ● 37 ● 263 ● 339

---

248 @Dyppl actually `jg` and `jnl` are the same instruction, `7F` :-)  
– Jonathon Reinhart Aug 27, 2012 at 5:30

---

- 24 Not to mention that the optimizer can modify the code if indeed one option is faster than the other.  
– [Elazar Leibovich](#) Aug 28, 2012 at 6:33
- 
- 4 just because something results in the same amount of instructions doesn't necessarily mean that the sum total time of executing all those instructions will be the same. Actually more instructions could be executed faster. Instructions per cycle is not a fixed number, it varies depending on the instructions. – [jontejj](#) May 31, 2013 at 16:51
- 
- 25 @jontejj I'm very much aware of that. Did you even *read* my answer? I didn't state anything about the same *number* of instructions, I stated that they are compiled to essentially the exact same *instrutcions*, except one jump instruction is looking at one flag, and the other jump instruction is looking at two flags. I believe I've given more than adequate evidence to show that they are semantically identical.  
– [Jonathon Reinhart](#) Jun 1, 2013 at 5:22
- 
- 5 @jontejj You make a very good point. For as much visibility as this answer gets, I should probably give it a little bit of a cleanup. Thanks for the feedback. – [Jonathon Reinhart](#) Jun 3, 2013 at 14:20
- 



623



Historically (we're talking the 1980s and early 1990s), there were *some* architectures in which this was true. The root issue is that integer comparison is inherently implemented via integer subtractions. This gives rise to the following cases.



Comparison		Subtraction
-----		-----
$A < B$	-->	$A - B < 0$

$A = B \quad \rightarrow A - B = 0$   
 $A > B \quad \rightarrow A - B > 0$

Now, when  $A < B$  the subtraction has to borrow a high-bit for the subtraction to be correct, just like you carry and borrow when adding and subtracting by hand. This "borrowed" bit was usually referred to as the *carry bit* and would be testable by a branch instruction. A second bit called the *zero bit* would be set if the subtraction were identically zero which implied equality.

There were usually at least two conditional branch instructions, one to branch on the carry bit and one on the zero bit.

Now, to get at the heart of the matter, let's expand the previous table to include the carry and zero bit results.

Comparison	Subtraction	Carry Bit	Zero Bit
$A < B$	$A - B < 0$	0	0
$A = B$	$A - B = 0$	1	1
$A > B$	$A - B > 0$	1	0

So, implementing a branch for  $A < B$  can be done in one instruction, because the carry bit is clear *only* in this case, that is,

```
;; Implementation of "if (A < B) goto address;"  
cmp  A, B          ;; compare A to B  
bcz  address       ;; Branch if Carry is Zero to the n
```



But, if we want to do a less-than-or-equal comparison, we need to do an additional check of the zero flag to catch the case of equality.

```
;; Implementation of "if (A <= B) goto address;"  
cmp A, B          ;; compare A to B  
bcz address       ;; branch if A < B  
bzs address       ;; also, Branch if the Zero bit is
```

So, on some machines, using a "less than" comparison *might save one machine instruction*. This was relevant in the era of sub-megahertz processor speed and 1:1 CPU-to-memory speed ratios, but it is almost totally irrelevant today.

Share Improve this answer

edited Nov 22, 2012 at 5:54

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 27, 2012 at 17:53



Lucas

8,083 ● 2 ● 33 ● 45

---

11 Additionally, architectures like x86 implement instructions like `jge`, which test both the zero and sign/carry flags.



– [greyfade](#) Aug 27, 2012 at 18:23

---

5 Even if it is true for a given architecture. What are the odds that none of the compiler writers ever noticed, and added an optimisation to replace the slower with the faster?

– [Jon Hanna](#) Aug 27, 2012 at 21:50

---

- 9 This is true on the 8080. It has instructions to jump on zero and jump on minus, but none that can test both simultaneously. – user597225 Aug 27, 2012 at 22:43 
- 
- 5 This is also the case on the 6502 and 65816 processor family, which extends to the Motorola 68HC11/12, too.  
– Lucas Aug 27, 2012 at 22:56
- 
- 39 Even on the 8080 a `<=` test can be implemented in *one* instruction with swapping the operands and testing for `not <` (equivalent to `>=` ) This is the desired `<=` with swapped operands: `cmp B,A; bcs addr` . That's the reasoning this test was omitted by Intel, they considered it redundant and you couldn't afford redundant instructions at those times :-)  
– Gunther Piez Aug 29, 2012 at 11:10 
- 



100



Assuming we're talking about internal integer types, there's no possible way one could be faster than the other. They're obviously semantically identical. They both ask the compiler to do precisely the same thing. Only a horribly broken compiler would generate inferior code for one of these.

If there was some platform where `<` was faster than `<=` for simple integer types, the compiler should *always* convert `<=` to `<` for constants. Any compiler that didn't would just be a bad compiler (for that platform).

Share Improve this answer

edited Aug 27, 2012 at 2:58

Follow

answered Aug 27, 2012 at 2:31



David Schwartz

183k ● 18 ● 224 ● 287


---


8 +1 I agree. Neither `<` nor `<=` have speed until the compiler decides which speed they'll have. This is a very simple optimisation for compilers when you consider that they generally already perform dead code optimisation, tail call optimisation, loop hoisting (and unrolling, on occasions), automatic parallelisation of various loops, etc... *Why waste time pondering premature optimisations?* Get a prototype running, profile it to determine where the most significant optimisations lie, perform those optimisations in order of significance and profile again along the way to measure progress... – [autistic](#) Jun 10, 2013 at 2:52

---

There are still some edge cases where a comparison having one constant value could be slower under `<=`, e.g., when the transformation from `(a < C)` to `(a <= C-1)` (for some constant `C`) causes `C` to be more difficult to encode in the instruction set. For example, an instruction set may be able to represent signed constants from -127 to 128 in a compact form in comparisons, but constants outside that range have to be loaded using either a longer, slower encoding, or another instruction entirely. So a comparison like `(a < -127)` may not have a straightforward transformation. – [BeeOnRope](#) Jun 16, 2016 at 2:18 ✎

---

@BeeOnRope The issue was not whether performing operations that differed due to having different constants in them could affect performance but whether *expressing* the *same* operation using different constants could affect performance. So we're not comparing `a > 127` to `a > 128` because you have no choice there, you use the one you need. We're comparing `a > 127` to `a >= 128`, which can't require different encoding or different instructions because they have the same truth table. Any encoding of one is equally an encoding of the other. – [David Schwartz](#) Jun 16, 2016 at 4:36 

I was responding in a general way to your statement that "If there was some platform where [`<=` was slower] the compiler should always convert `<=` to `<` for constants". As far as I know, that transformation involves changing the constant. E.g., `a <= 42` is compiled as `a < 43` because `<` is faster. In some edge cases, such a transformation wouldn't be fruitful because the new constant may require more or slower instructions. Of course `a > 127` and `a >= 128` are equivalent and a compiler should encode both forms in the (same) fastest way, but that's not inconsistent with what I said. – [BeeOnRope](#) Jun 16, 2016 at 20:36 



68

I see that neither is faster. The compiler generates the same machine code in each condition with a different value.



```
if(a < 901)
  cmpl $900, -4(%rbp)
  jg .L2
```

```
if(a <=901)
  cmpl $901, -4(%rbp)
  jg .L3
```

My example `if` is from GCC on x86\_64 platform on Linux.

Compiler writers are pretty smart people, and they think of these things and many others most of us take for granted.

I noticed that if it is not a constant, then the same machine code is generated in either case.

```
int b;  
if(a < b)  
    cmpl    -4(%rbp), %eax  
    jge     .L2  
  
if(a <=b)  
    cmpl    -4(%rbp), %eax  
    jg      .L3
```

Share Improve this answer

Follow

edited Nov 22, 2012 at 5:48



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 27, 2012 at 2:16



Adrian Cornish

23.8k ● 13 ● 63 ● 80

---

10 Note that this is specific to x86. – [Michael Petrotta](#) Aug 27, 2012 at 2:17

---

11 I think you should use that `if(a <=900)` to demonstrate that it generates exactly the same asm :) – [Lipis](#) Aug 27, 2012 at 2:22 ✎

---

- 2 @AdrianCornish Sorry.. I edited it.. it's more or less the same.. but if you change the second if to  $\leq 900$  then the asm code will be exactly the same :) It's pretty much the same now.. but you know.. for the OCD :) – [Lipis](#) Aug 27, 2012 at 2:25
- 
- 3 @Boann That might get reduced to if (true) and eliminated completely. – [Qsario](#) Aug 27, 2012 at 2:32
- 
- 5 No one has pointed out that this optimization *only applies to constant comparisons*. I can guarantee it will *not* be done like this for comparing two variables. – [Jonathon Reinhart](#) Aug 27, 2012 at 3:05
- 



54

For floating point code, the  $\leq$  comparison may indeed be slower (by one instruction) even on modern architectures. Here's the first function:



```
int compare_strict(double a, double b) { return a < b;
```



On PowerPC, first this performs a floating point comparison (which updates `cr`, the condition register), then moves the condition register to a GPR, shifts the "compared less than" bit into place, and then returns. It takes four instructions.

Now consider this function instead:

```
int compare_loose(double a, double b) { return a <= b;
```

This requires the same work as `compare_strict` above, but now there's two bits of interest: "was less than" and

"was equal to." This requires an extra instruction (`cror` - condition register bitwise OR) to combine these two bits into one. So `compare_loose` requires five instructions, while `compare_strict` requires four.

You might think that the compiler could optimize the second function like so:

```
int compare_loose(double a, double b) { return ! (a >
```

However this will incorrectly handle NaNs. `NaN1 <= NaN2` and `NaN1 > NaN2` need to both evaluate to false.

Share Improve this answer

edited Aug 27, 2012 at 18:38

Follow

answered Aug 27, 2012 at 18:32



[ridiculous\\_fish](#)

18.4k ● 1 ● 58 ● 67

---

Luckily it doesn't work like this on x86 (x87). `fucomip` sets ZF and CF. – [Jonathon Reinhart](#) Aug 27, 2012 at 20:30

---

- 5 @JonathonReinhart: I think you're misunderstanding what the PowerPC is doing -- the condition register `cr` is the equivalent to flags like `ZF` and `CF` on the x86. (Although the CR is more flexible.) What the poster is talking about is moving the result to a GPR: which takes two instructions on PowerPC, but x86 has a conditional move instruction.
- [Dietrich Epp](#) Aug 28, 2012 at 6:19
- 

@DietrichEpp What I meant to add after my statement was: Which you can then immediately jump based upon the value

of EFLAGS. Sorry for not being clear. – [Jonathon Reinhart](#)  
Aug 28, 2012 at 7:16

---

- 1 @JonathonReinhart: Yes, and you can also immediately jump based on the value of the CR. The answer is not talking about jumping, which is where the extra instructions come from. – [Dietrich Epp](#) Aug 28, 2012 at 7:38
-





34

Maybe the author of that unnamed book has read that `a > 0` runs faster than `a >= 1` and thinks that is true universally.



But it is because a `0` is involved (because `CMP` can, depending on the architecture, be replaced e.g. with `OR`) and not because of the `<`.



Share Improve this answer

answered Aug 27, 2012 at 13:05

Follow



glglgl

90.9k ● 13 ● 154 ● 228

1 Sure, in a "debug" build, but it would take a bad compiler for `(a >= 1)` to run slower than `(a > 0)`, since the former can be trivially transformed to the latter by the optimizer..

– BeeOnRope Jun 16, 2016 at 2:22

2 @BeeOnRope Sometimes I am surprised what complicated things an optimizer can optimize and on what easy things it fails to do so. – glglgl Jun 16, 2016 at 7:31

1 Indeed, and it's always worth checking the asm output for the very few functions where it would matter. That said, the above transformation is very basic and has been performed even in simple compilers for decades. – BeeOnRope Jun 16, 2016 at 20:27



30



At the very least, if this were true a compiler could trivially optimise  $a \leq b$  to  $!(a > b)$ , and so even if the comparison itself were actually slower, with all but the most naive compiler you would not notice a difference.

Share Improve this answer

edited Nov 30, 2012 at 0:22

Follow

answered Aug 27, 2012 at 9:23



Eliot Ball

728 ● 5 ● 11

Why  $!(a > b)$  is optimised version of  $a \leq b$  . Isn't  $!(a > b)$  2 operation in one ? – [Abhishek Singh](#) Apr 7, 2015 at 11:33

6 @AbhishekSingh NOT is just made by other instruction ( `jne` vs. `jne` ) – [Pavel Gatnar](#) Apr 14, 2015 at 16:03

In IEEE floating point, `a <= b` does not mean the same thing as `!(a > b)` . – [Don Hatch](#) Sep 11, 2020 at 15:25



18



## [TL;DR](#) answer

For most combinations of architecture, compiler and language, `<` will not be faster than `<=` .

## Full answer

Other answers have concentrated on [x86](#) architecture, and I don't know the [ARM](#) architecture (which your example assembler seems to be) well enough to

comment specifically on the code generated, but this is an example of a [micro-optimisation](#) which *is* very architecture specific, and is **as likely to be an anti-optimisation as it is to be an optimisation**.

As such, I would suggest that this sort of [micro-optimisation](#) is an example of [cargo cult](#) programming rather than best software engineering practice.

## Counterexample

There are probably *some* architectures where this is an optimisation, but I know of at least one architecture where the opposite may be true. The venerable [Transputer](#) architecture only had machine code instructions for *equal to* and *greater than or equal to*, so all comparisons had to be built from these primitives.

Even then, in almost all cases, the compiler could order the evaluation instructions in such a way that in practice, no comparison had any advantage over any other. Worst case though, it might need to add a reverse instruction (REV) to swap the top two items on the [operand stack](#). This was a single byte instruction which took a single cycle to run, so had the smallest overhead possible.

## Summary

Whether or not a micro-optimisation like this is an *optimisation* or an *anti-optimisation* depends on the specific architecture you are using, so it is usually a bad

idea to get into the habit of using architecture specific micro-optimisations, otherwise you might instinctively use one when it is inappropriate to do so, and it looks like this is exactly what the book you are reading is advocating.

Share Improve this answer

edited Nov 4, 2020 at 11:54

Follow

answered Aug 31, 2012 at 18:33



Mark Booth

7,894 ● 2 ● 76 ● 97



15



They have the same speed. Maybe in some special architecture what he/she said is right, but in the x86 family at least I know they are the same. Because for doing this the CPU will do a subtraction ( $a - b$ ) and then check the flags of the flag register. Two bits of that register are called ZF (zero Flag) and SF (sign flag), and it is done in one cycle, because it will do it with one mask operation.

Share Improve this answer

edited Nov 22, 2012 at 5:52

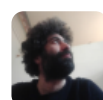
Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 27, 2012 at 8:25



Masoud

1,405 ● 6 ● 20 ● 32



14



This would be highly dependent on the underlying architecture that the C is compiled to. Some processors and architectures might have explicit instructions for equal to, or less than and equal to, which execute in different numbers of cycles.

That would be pretty unusual though, as the compiler could work around it, making it irrelevant.

Share Improve this answer

answered Aug 27, 2012 at 2:15

Follow



Telgin

1,604 ● 10 ● 10

---

2 IF there was a difference in the cycles. 1) it would not be detectable. 2) Any compiler worth its salt would already be making the transformation from the slow form to the faster form without changing the meaning of the code. So the resulting instruction planted would be identical. – [Loki Astari](#) Aug 27, 2012 at 7:00

---

Agreed completely, it would be a pretty trivial and silly difference in any case. Certainly nothing to mention in a book that should be platform agnostic. – [Telgin](#) Aug 28, 2012 at 3:46

---

@Ittlrck: I get it. Took me a while (silly me). No they are not detectable because there are so many other things happening that make their measurement impossible. Processor stalls/ cache misses/ signals/ process swapping. Thus in a normal OS situation things on the single cycle level can not be physically measurable. If you can eliminate all that interference from your measurements (run it on a chip with on-board memory and no OS) then you still have granularity of your timers to worry about but theoretically if you run it

long enough you could see something. – [Loki Astari](#) Aug 29, 2012 at 6:57

---



6



You should not be able to notice the difference even if there is any. Besides, in practice, you'll have to do an additional `a + 1` or `a - 1` to make the condition stand unless you're going to use some magic constants, which is a very bad practice by all means.



Share Improve this answer

answered Aug 27, 2012 at 2:17



Follow



[shinkou](#)

5,154 ● 1 ● 23 ● 34

---

1 What's the bad practice? Incrementing or decrementing a counter? How do you store index notation then? – [jcolebrand](#) Aug 27, 2012 at 14:22

6 He means if you're doing comparison of 2 variable types. Of course it's trivial if you're setting the value for a loop or something. But if you have  $x \leq y$ , and  $y$  is unknown, it would be slower to 'optimize' it to  $x < y + 1$  – [JustinDanielson](#) Aug 27, 2012 at 21:48

---

@JustinDanielson agreed. Not to mention ugly, confusing, etc. – [Jonathon Reinhart](#) Aug 27, 2012 at 23:49

---



6



When I wrote the first version of this answer, I was only looking at the title question about  $<$  vs.  $\leq$  in general, not the specific example of a constant `a < 901` vs. `a <= 900`. Many compilers always shrink the magnitude of constants by converting between  $<$  and  $\leq$ , e.g.



because x86 immediate operand have a shorter 1-byte encoding for -128..127.

For ARM, being able to encode as an immediate depends on being able to rotate a narrow field into any position in a word. So `cmp r0, #0x00f000` would be encodeable, while `cmp r0, #0x00efff` would not be. So the make-it-smaller rule for comparison vs. a compile-time constant doesn't always apply for ARM. AArch64 is either shift-by-12 or not, instead of an arbitrary rotation, for instructions like `cmp` and `cmn`, unlike 32-bit ARM and Thumb modes.

---

## < vs. <= in general, including for runtime-variable conditions

In assembly language on most machines, a comparison for `<=` has the same cost as a comparison for `<`. This applies whether you're branching on it, booleanizing it to create a 0/1 integer, or using it as a predicate for a branchless select operation (like x86 CMOV). The other answers have only addressed this part of the question.

**But this question is about the C++ operators, the *input to the optimizer*.** Normally they're both equally efficient; the advice from the book sounds totally bogus because compilers can always transform the comparison that they implement in asm. But there is at least one exception where using `<=` can accidentally create something the compiler can't optimize.

As a loop condition, there are cases where `<=` is *qualitatively* different from `<`, when it stops the compiler from proving that a loop is not infinite. This can make a big difference, disabling auto-vectorization.

Unsigned overflow is well-defined as base-2 wrap around, unlike signed overflow (UB). Signed loop counters are generally safe from this with compilers that optimize based on signed-overflow UB not happening:

`++i <= size` will always eventually become false. ([What Every C Programmer Should Know About Undefined Behavior](#))

```
void foo(unsigned size) {  
    unsigned upper_bound = size - 1; // or any calcul  
    UINT_MAX  
    for(unsigned i=0 ; i <= upper_bound ; i++)  
        ...  
}
```

Compilers can only optimize in ways that preserve the (defined and legally observable) behaviour of the C++ source for *all* possible input values, except ones that lead to undefined behaviour.

(A simple `i <= size` would create the problem too, but I thought calculating an upper bound was a more realistic example of accidentally introducing the possibility of an infinite loop for an input you don't care about but which the compiler must consider.)

In this case, `size=0` leads to `upper_bound=UINT_MAX`, and `i <= UINT_MAX` is always true. So this loop is



**infinite for `size=0` , and the compiler has to respect that** even though you as the programmer probably never intend to pass `size=0`. If the compiler can inline this function into a caller where it can prove that `size=0` is impossible, then great, it can optimize like it could for `i < size` .

Asm like `if(!size) skip the loop; do{...}while(--size);` is one normally-efficient way to optimize a `for(i<size )` loop, if the actual value of `i` isn't needed inside the loop ([Why are loops always compiled into "do...while" style \(tail jump\)?](#)).

But that `do{}while` can't be infinite: if entered with `size==0` , we get  $2^n$  iterations. ([Iterating over all unsigned integers in a for loop](#) C makes it possible to express a loop over all unsigned integers including zero, but it's not easy without a carry flag the way it is in asm.)

**With wraparound of the loop counter being a possibility, modern compilers often just "give up", and don't optimize nearly as aggressively.**

## Example: sum of integers from 1 to n

Using unsigned `i <= n` defeats clang's idiom-recognition that optimizes `sum(1 .. n)` loops with a closed form based on Gauss's  `$n * (n+1) / 2$`  formula.

```

unsigned sum_1_to_n_finite(unsigned n) {
    unsigned total = 0;
    for (unsigned i = 0 ; i < n+1 ; ++i)
        total += i;
    return total;
}

```

[x86-64 asm from clang7.0 and gcc8.2 on the Godbolt compiler explorer](#)

```

# clang7.0 -O3 closed-form
    cmp     edi, -1          # n passed in EDI: x86-64 Sy
    je      .LBB1_1          # if (n == UINT_MAX) return
times
                                # else fall through into the closed-form cal
    mov     ecx, edi          # zero-extend n into RCX
    lea     eax, [rdi - 1]    # n-1
    imul    rax, rcx          # n * (n-1)
    shr     rax               # n * (n-1) / 2
    add     eax, edi          # n + (stuff / 2) = n * (
32-bit
    ret                                # computed without possible overflow
right shifting
.LBB1_1:
    xor     eax, eax
    ret

```

**But for the naive version, we just get a dumb loop from clang.**

```

unsigned sum_1_to_n_naive(unsigned n) {
    unsigned total = 0;
    for (unsigned i = 0 ; i<=n ; ++i)
        total += i;
    return total;
}

```

```

# clang7.0 -O3
sum_1_to_n(unsigned int):
    xor     ecx, ecx           # i = 0
    xor     eax, eax           # retval = 0
.LBB0_1:
    add     eax, ecx           # retval += i
    add     ecx, 1             # ++i
    cmp     ecx, edi
    jbe     .LBB0_1            # } while( i<n );
    ret

```

**GCC doesn't use a closed-form either way, so the choice of loop condition doesn't really hurt it;** it auto-vectorizes with SIMD integer addition, running 4 `i` values in parallel in the elements of an XMM register.

```

# "naive" inner loop
.L3:
    add     eax, 1             # do {
    paddd   xmm0, xmm1         # vect_total_4.6, vect_vec_i
    paddd   xmm1, xmm2         # vect_vec_iv_.5, tmp114
    cmp     edx, eax           # bnd.1, ivtmp.14      # bound
    jmp     tmp, I think.
    ja      .L3 #,             # }while( n > i )

"finite" inner loop
# before the loop:
# xmm0 = 0 = totals
# xmm1 = {0,1,2,3} = i
# xmm2 = set1_epi32(4)
.L13:
    add     eax, 1             # do {
    add     eax, 1             # i++
    paddd   xmm0, xmm1         # total[0..3] += i[0..3]
    paddd   xmm1, xmm2         # i[0..3] += 4
    cmp     eax, edx
    jne     .L13               # }while( i != upper_limit );

    then horizontal sum xmm0

```

and peeled cleanup for the last  $n\%3$  iterations, o

It also has a plain scalar loop which I think it uses for very small  $n$ , and/or for the infinite loop case.

BTW, both of these loops waste an instruction (and a uop on Sandybridge-family CPUs) on loop overhead. `sub eax, 1 / jnz` instead of `add eax, 1 / cmp/jcc` would be more efficient. 1 uop instead of 2 (after macro-fusion of `sub/jcc` or `cmp/jcc`). The code after both loops writes EAX unconditionally, so it's not using the final value of the loop counter.

Share Improve this answer

edited Nov 2, 2020 at 7:03

Follow

answered Jan 20, 2019 at 11:30



Peter Cordes

361k ● 49 ● 699 ● 958

Nice contrived example. What about your other comment about a potential effect on out of order execution due to EFLAGS use? Is it purely theoretical or can it actually happen that a JB leads to a better pipeline than a JBE? – [rustyx](#) Jan 20, 2019 at 12:05

@rustyx: did I comment that somewhere under another answer? Compilers aren't going to emit code that causes partial-flag stalls, and certainly not for a `C <` or `<=`. But sure, `test ecx, ecx / bt eax, 3 / jbe` will jump if ZF is set (`ecx==0`), or if CF is set (bit 3 of EAX==1), causing a partial flag stall on most CPUs because the flags it reads don't all come from the last instruction to write any flags. On

Sandybridge-family, it doesn't actually stall, just needs to insert a merging uop. `cmp` / `test` write all flags, but `bt` leaves ZF unmodified. [felixcloutier.com/x86/bt](http://felixcloutier.com/x86/bt) – Peter Cordes Jan 20, 2019 at 12:23

---

As I understand it, the available immediates for `cmp` on AArch64 are simpler than your answer makes it sound: it takes a 12-bit immediate optionally shifted by 12 bits, so you can have `0xYYY` or `0xYYY000`, and you can also effectively negate the immediate by using `cmn` instead. This still supports your point, as `cmp w0, #0xf000` is encodeable and `cmp w0, #0xffff` is not. But the "rotate into any position" phrasing sounds more like a description of the "bitmask" immediates, which AFAIK are only available for bitwise logical instructions: `and`, `or`, `eor`, etc.

– Nate Eldredge Nov 2, 2020 at 6:10

---

@NateEldredge: I think my description fits perfectly for ARM mode, where it's an 8-bit field rotated by a multiple of 2. (so `0x1fe` isn't encodeable but `0xff0` is.) When I wrote this I hadn't understood the differences between AArch64 and ARM immediates, or that only bitwise boolean insns could use the bit-range / repeated bit-pattern encoding. (And `mov`; `or` with the zero reg is one way to take advantage of those encodings.) – Peter Cordes Nov 2, 2020 at 7:07

---



4



You could say that line is correct in most scripting languages, since the extra character results in slightly slower code processing. However, as the top answer pointed out, it should have no effect in C++, and anything being done with a scripting language probably isn't that concerned about optimization.



Share Improve this answer

answered Aug 29, 2012 at 2:47



---

I somewhat disagree. In competitive programming, scripting languages often offer the quickest solution to a problem, but correct techniques (read: optimization) must be applied to get a correct solution. – [Tyler Crompton](#) Sep 5, 2012 at 0:59

---



Only if the people who created the computers are bad with boolean logic. Which they shouldn't be.

2



Every comparison (`>=` `<=` `>` `<`) can be done in the same speed.



What every comparison is, is just a subtraction (the difference) and seeing if it's positive/negative.



(If the `msb` is set, the number is negative)

How to check `a >= b` ? Sub `a-b >= 0` Check if `a-b` is positive.

How to check `a <= b` ? Sub `0 <= b-a` Check if `b-a` is positive.

How to check `a < b` ? Sub `a-b < 0` Check if `a-b` is negative.

How to check `a > b` ? Sub `0 > b-a` Check if `b-a` is negative.

Simply put, the computer can just do this underneath the hood for the given op:

`a >= b == msb(a-b)==0`

`a <= b == msb(b-a)==0`

`a > b == msb(b-a)==1`

`a < b == msb(a-b)==1`

and of course the computer wouldn't actually need to do the `==0` or `==1` either.

for the `==0` it could just invert the `msb` from the circuit.

Anyway, they most certainly wouldn't have made `a >= b` be calculated as `a>b || a==b` lol

Share Improve this answer

answered Jul 8, 2019 at 19:50

Follow



Puddle

3,317 ● 1 ● 23 ● 34

---

It's not that simple, though. For instance, if `a` is in a register and `b` is a compile-time constant, then x86 can compute `a-b` in one instruction ( `sub rax, 12345` or `cmp` ) but not `b-a`. There is an instruction for `reg - imm` but not the other way around. Many other machines have a similar situation. – [Nate Eldredge](#) Nov 2, 2020 at 16:55 ✎

---

This whole answer is wrong. For example, `INT_MAX > INT_MIN`, but `INT_MIN - INT_MAX` is 1 (at the machine-code level, not in C), whose `msb` is 0. Doing these comparisons correctly is harder than you think. – [benrg](#) Apr 17, 2022 at 21:10

---

@benrg Obviously it wouldn't be as simple to implement in code as the actual circuit with access to the carry-out/overflow. This answer was to explain the logic of the different comparisons. (it's all about the difference. it's sign) It'd obviously be handled in the ALU. (for different types like floats also) See [this](#). So the answer isn't wrong. What you're bringing up is the limitations of integers. If we put `INT_MIN` in a long it'd store `ffffffff80000000`. And doing the subtraction



2



In C and C++, an important rule for the compiler is the “as-if” rule: If doing X has the exact same behavior as if you did Y, then the compiler is free to choose which one it uses.

In your case, “`a < 901`” and “`a <= 900`” always have the same result, so the compiler is free to compile either version. If one version was faster, for whatever reason, then any quality compiler would produce code for the version that is faster. So unless your compiler produced exceptionally bad code, both versions would run at equal speed.

Now if you had a situation where two bits of code will always produce the same result, but it is hard to prove for the compiler, and/or it is hard for the compiler to prove which if any version is faster, then you might get different code running at different speeds.

PS The original example might run at different speeds if the processor supports single byte constants (faster) and multi byte constants (slower), so comparing against 255 (1 byte) might be faster than comparing against 256 (two bytes). I’d expect the compiler to do whatever is faster.



Share Improve this answer

edited Sep 2, 2022 at 10:41

Follow

answered Sep 2, 2022 at 9:44



[gnasher729](#)

52.5k ● 5 ● 79 ● 102

Typically machine code uses sign-extended immediates, so +128 vs. +127 is the cutoff for `cmp ecx, imm8` in 3 bytes vs. 6-byte `cmp ecx, imm32` on x86 for example. That's why GCC always tries to reduce the magnitude of a constant when compiling compares: [Understanding gcc output for if \(a>=3\)](#) (This isn't always a win on ARM, e.g. `0x1000` can fit in a rotated immediate while `0x0fff` can't.) Oh, I see I [already posted an answer on this question](#) pointing out those things. – [Peter Cordes](#) Sep 2, 2022 at 12:27



**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.