URC97054

# How difficult is it to add 1? A pedagogical example of how theory of computing may be useful

Michael Hampton

Center for Theoretical Research and its
Applications in Computer Science (TRACS)
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968
email mhampton@cs.utep.edu

Abstract

We show that while adding two generic b-bit integers requires $\approx b$ bit operations, adding 1 to an integer requires, on average, only 2 bit operations. This fact explains why the operation of adding 1 is often separately hardware supported, and why this operation is often separately described in high-level programming languages like C++.

This results shows that t heretical analysis can help in deciding which operations must be hardware supported, and thus. hopefully, will help in designing faster computers.

## 1 General Introduction

**For NASA-oriented computer applications, it is very important to make computers faster.** From the computer science viewpoint, one of the major problems of NASA research is that so much data is coming from every space mission that the existing computer systems can process, by some estimates, only about 10% of this data.

***How can** we **increase the*** throughput? Since the main problem is that computers cannot process all the data, a natural solution is to make computers *faster*.

**At first glance, making computers faster is a computer engineering problem, but computer science can also help.** In principle, there are two ways to make computers faster:

- First, we can try to further miniaturize the electronics by making the chip's elements even smaller. This is desirable but extremely difficult.
- Second, even within the same technological level, we can change the computation speed by selecting different sets of computer operations to be hardware supported.
  - When an operation (e.g., addition, multiplication, etc. ) is implemented in ***hardware,*** it is extremely *fast*;
  - otherwise, if this operation is *not* directly ***hardware*** supported, we have to implement it on the microprogramming level, as a sequence of two or more hardware supported operations, which make computations slower-.

[t is therefore desirable to be very careful in choosing which operations we want to hardware support: ideally, we should support, the operation that are used most frequently, so that the resulting slow down of not supported (less frequent) operations will have the smallest impact on the resulting computation speed.

At first glance, the problems related to both ways of increasing computer speed lie in ***computer engineering.*** However, as we will show in this paper, at least for the second approach, ***computer science*** (and. especially, ***theoretical computer science)*** can be very useful.

Namely, we will show, on a simple example, that some existing ***semi-heuristic*** choices of the hardware supported operations can be explained and formally justified in t heretical computer science. This example shows that in more complicated situations, in which an optimal choice of hardware supported operation is not yet **known,** methods of theoretical computer science can he of help.

## 2 Formulation of the Case Problem: Adding 1 and Why It Is Import ant

**Adding 1 is an important particular case of addition. One** of the simplest arithmetic operations, that is directly hardware supported on all the computers, is addition of two integers.

Not all additions are born equal. If we try to trace, on the level of computer instructions, what exactly additions are performed in a typical data processing algorithm, then we will see that these additions fall into the following two categories:

- First, there are additions of data values, that come directly from the additions in the original program (written in Fortran, C, C++, or any other high-level programming language).
- Second, additions that were "hidden" in the FOR loop or in other construction of the original program. The most typical of such hidden addition is a FOR loop, in which a loop index i starts with its lower bound, and then gets increased by 1 on each iteration until it reaches its upper bound.

In other words, a reasonably frequent addition is adding 1: i:=i+1. This particular case of addition is so frequent that in C++, there is a special denotation for adding 1, as opposed to any other addition: the famous notation i++ that has lead to the very name of this language.

**Adding 1 is implemented in hardware: why?** In several processing languages, adding 1 is described as a separate operation; in several processors, it is hardware supported as a separate operation, ***in addition*** to the general addition.

This support is, usually, done on a ***semi-heuristic*** basis, in the sense that it is not justified by any precise arguments.

## 3 How Can We Decide Which Operations to Support: an Idea

When does it make sense to design a special hardware support for a particular case of an already hardware supported operation?

If we try to support too many operations in hardware, then the very choice of an operation will take more computation time. Thus, every additional operation that we hardware support makes the other computations slower. Since our objective is to speed up the computations, the only reason why we can tolerate a little slow-down of all other operations is when this particular operation becomes ***much faster*** than the one that we have used before.

Therefore, the following idea helps to decide which operations to support:

- If the direct hardware support makes an operation much faster, then it is probably desirable to directly support it in hardware.
- If, on the other hand, the direct hardware support, would not drastically speed u p this operation, then this operation is, probably, not the best candidate for hardware support.

Of course, this is simply a raw idea, and in the real applications, we have to actually compare the speed-ups and slow-downs to make a decision, but this idea gives a picture of how such a decision can be made.

## 4 At First Glance, From the Theoretical Viewpoint, Adding 1 Is Not That Different from General Addition, So There Seems To Be No Reason to Implement This Operation Separately

**What we must do.** From the viewpoint of this general idea, in order to decide whether adding 1 must be separately hardware supported or not, we must check whether the hardware support of adding 1 will really speed this operation up as compared to its implementation as particular case of the general integer addition.

In other words, we needs to compare the computation time of the following two operations:

- (general) addition of two integers
- adding 1 to an integer.

How can we estimate the computation time of each operation?

**Operation time reformulated in computer terms.** On the hardware level, each elementary hardware operation is an operation with bits. Therefore, the computation time of each operation can be estimated, crudely speaking, as proportional to the total number of ***bit operations*** that we have to perform.

**The first seemingly natural choice: worst-case complexity.** The number of bit operations depends on what exactly numbers wc add. So, in order to compare two different algorithms that can be applied to different numbers, we must come up with some natural characteristics of these algorithms that will characterize their overall behavior.

In theoretical computer science (see, e.g., [1]), a typical measure of time complexity of an algorithm is its ***worst-case*** number of computational steps, i.e., the maximal number of steps that this algorithm needs to process the input of a given length $n$.

In the computer, usually, integers require a fixed number $b$ of bits (e.g., in PC's, usually, 2 bytes = 16 bits arc reserved for each normal size integer). So, we are interested in the worst-case bit complexity of

- adding two numbers of size $h$ (i.e., integers with $b$ bits in each of them), and
- adding 1 to a $b$-bit number.

**Worst-case complexity of adding two integers.** To add two b-bit integers, we must add their last bits, then add the previous bits (and maybe a carry), etc. If we count adding a ***carry,*** then we need 2 bit additions per each bit. Thus, out of $b$ bits, in the worst case, we need 2 hit operations per bit except for the very last bit that requires only one operation. So, the worst-case bit complexity (= number of bit operations) of adding two numbers is $2(b-1)+1=2b-1$.

**Worst-case complexity of adding 1 to an integer.** The worst case of adding 1 is when we add 1 to a number 11...1 that consists of all 1's. In this case, adding 1 will change all the bits in the number (to 100...0). Changing each bit requires at least one bit operation, so, in the worst case, adding 1 requires $b-1$ bit operations.

**From the viewpoint of the worst-case complexity, the gain is minimal.** Comparing these two results, we conclude that from the viewpoint, of the worst-case complexity, adding 1 takes, at best, half a time of the general addition.

This is faster, but, since adding each hardware operation increases the running time of other operations, this increase does not seem to necessarily justify the separate implementation of adding 1.

## 5 If We Consider a (More Meaningful) Average Time Instead of the Worst-Case Time, Then We Get the Drastic Difference Between Adding 1 and General Addition

**Average-case complexity is more meaningful for our problem than the worst-case complexity.** Our main objective is to make the computers faster. From the viewpoint of this objective, bad worst-case time is tolerable is the corresponding situations are rare. What we are really interested in is the ***average*** bit complexity, averaged over all possible numbers.

We will now show that, in contrast to the worst-case complexity, the average complexity does explain why adding 1 is sometimes implemented as a separate operation.

**Average-case complexity of adding two integers.** It is easy to show that an average-case bit complexity of adding two integers is still a linear function of $b$.

**Average-case complexity of adding 1 to an integer: deducing a formula.** The natural algorithm of adding 1 is as follows: We start at the lower digits. Whenever we encounter 1, we change it to 0. When we encounter 0, we change it to 1 and stop. For this algorithm, the total number of bit operations is equal to the number of bits that are changed.

For example, if the number ends in 0 (e.g., 11 10), we replace this 0 by 1 and stop. which take exactly 1 bit operation. If the number end in 01 (e.g., 11 01), we replace two last bits and stop, making it two bit operations.

Let us calculate the average number of bit operations:

- For numbers ending in a single 0, we need 1 bit operation. Such numbers constitute $1/2 = 2^{-1}$ of all numbers of size $b$.
- For numbers ending with a 0 and one 1 (i.e.. with 01), we need 2 bit operations. Such numbers constitute $1/4 = 2^{-2}$ of all numbers of size $b$.
- For numbers ending with a 0 and two 1's (i.e., with 011), we need 3 bit operations. Such numbers constitute $1/8 = 2^{-3}$ of all numbers of size $b$.

  ...
- For numbers ending with a 0 and $(k-1)$ 1's (i.e., with 01 . . . 1), we need 2 bit operation. Such numbers constitute $2^{-k}$ of all numbers of size $b$.

  ...
- For numbers ending in a single 0 and $(b-1)$ 1's (i.e., for a number 01 . . . 1), we need $b$ bit operation. There is only one such number, and it therefore constitutes $2^{-b}$ of all numbers of size $b$.
- The only remaining number is 1 ...1 (all 1 's), for which we also need $b$ bit operations. This number also constitutes $2^{-b}$ of all numbers of size $b$.

The resulting average-time complexity $c(b)$ of adding 1 to a b-bit number is equal to

$$c(b) = 1 \cdot 2^{-1} + 2 \cdot 2^{-2} + 3 \cdot 2^{-3} + \ldots + b \cdot 2^{-b} + b \cdot 2^{-b}.$$

**Average-case complexity of adding 1 to an integer: simplifying a formula.** For large $b$, it is natural to estimate this sum by considering the infinite sum

$$c(\infty) = 1 \cdot 2^{-1} + 2 \cdot 2^{-2} + 3 \cdot 2^{-3} + \ldots + b \cdot 2^{-b} + (b+1) \cdot 2^{-(b+1)} + \ldots$$

Since

$$2^{-b} = 2^{-(b+1)} + 2^{-(b+2)} + \ldots$$

and $b \le b+1$, $b \le b+2, \ldots$ we can conclude that

$$b \cdot 2^{-b} = b \cdot (2^{-(b+1)} + 2^{-(b+2)} + \ldots) = b \cdot 2^{-(b+1)} + b \cdot 2^{-(b+2)} + \ldots \le (b+1) \cdot 2^{-(b+1)} + (b+2) \cdot 2^{-(b+2)} + \ldots.$$

and that, therefore, $c(b) \le c(\infty)$.

The value

$$c(\infty) = \sum_{k=1}^{\infty} k \cdot 2^{-k} \qquad (1)$$

can be easily computed if we multiply both sides of this equality by 2 and take into consideration that $2 \cdot 2^{-k} = 2^{-(k-1)}$:

$$2 \cdot c(\infty) = \sum_{k=1}^{\infty} k \cdot 2 \cdot 2^{-k} = \sum_{k=1}^{\infty} k \cdot 2^{-(k-1)}.$$

Introducing a new variable $j = k-1$, we conclude that

$$2c(\infty) = \sum_{j=0}^{\infty} (j+1) \cdot 2^{-j} = \sum_{j=0}^{\infty} j \cdot 2^{-j} + \sum_{j=0}^{\infty} 2^{-j} \qquad (2)$$

The first sum in the right-hand side of the equation (2) differs from the expression (1) only by the term corresponding to $j = 0$, which is equal to 0; thus, the first sum is equal to $c(\infty)$. The second sum $1 + 2^{-1} + 2^{-2} + \ldots$ is a geometric progression, its sum is 2. Hence, from (2), we can conclude that $2c(\infty) = c(\infty) + 2$ and $c(\infty) = 2$.

Thus, $c(\infty) = 2$, and $c(b) \leq 2$.

**Average-case complexity of adding 1 to an integer: the result.** As a result, wc conclude that on average, adding 1 takes at most 2 bit operations, while adding two generic integers takes at least $b$ bit operations. For 16-bit integers, it means a 8 times speed up. For double-size integers (with 32 bits) it means 16 times speed up, etc.

**Conclusion. On average, *adding 1*** is ***much faster*** than ***the general*** addition ***of two integers. This*** drastic speed- up explains why ***the operation of*** adding ***1*** is often separately hardware ***supported, and why this*** operation is often separately described in high-level ***programming languages like*** C++.

**Acknowledgments.** This work was supported by the NASA Pan American Center for Environmental and Earth Studies (PACES). The author is thankful to Ann Gates, Vladik Kreinovich, Luc Longpré, and Scott Starks for their help.

## References

[1] C. H. Papadimitriou, ***Computational Complexity,*** Addison Wesley, San Diego, 1994.