A Note on Carry and Overflow

Paul Ossientis

April 13, 2017

1 Introduction

In this note we assume given a natural number $q = 2^n$ where $n \in \{8, 16, 32, 64\}$. We denote $\mathbf{Z}_q$ the ring of integers modulo q. The purpose of this note is to provide a formal presentation of the mathematics underlying some operations of a computer's CPU, and explain the notions of *carry* and *overflow* within that formal framework. It all starts with the realization that a computer's hardware is designed to perfectly carry out the operation of addition $+$ on the ring $\mathbf{Z}_q$. There is no approximation, there is no error, no overflow, the result is always perfect and exact: a computer's hardware naturally operates on $\mathbf{Z}_q$. It is also very good at inverting bits so given $x \in \mathbf{Z}_q$, it can easily compute $\neg x$. Furthermore, since $x + \neg x = q - 1$ we have the equality $-x = \neg x + 1$. It follows that a CPU can easily (and exactly) compute the opposit $-x$ of any $x \in \mathbf{Z}_q$, simply by incrementing $\neg x$. Hence we see that not only the addition $+$, but also the subtraction $-$ can be viewed as natural primitives of a computer's hardware, where $x - y$ is defined as $x - y = x + (-y)$ for all $x, y \in \mathbf{Z}_q$.

Definition 1 *For all $x \in \mathbf{Z}_q$, we define x^* the unique integer with the property:*

$$x^* = x \ \mathit{mod}\ q \quad \mathit{and} \quad x^* \in [0, q[$$

2 Carry for Unsigned Addition

Users of computer hardware are not interested in $\mathbf{Z}_q$. One of their first interests is to carry out the operation of addition $+$ on the set of natural numbers $\mathbf{N}$, an operation commonly referred to as *unsigned addition* by computer scientists. While it is possible and indeed a built-in feature of many modern computer languages (Python, Haskell) to handle every possible values of $\mathbf{N}$, for historical reasons and the purpose of this note, it is important to restrict our attention to natural numbers which can be represented as *unsigned integers* within an n-bits register. These *representable* natural numbers are exactly those contained in the interval $[0, q[$ and the representation is exactly the associated integer modulo q. In other words, the *representable* natural numbers are the range of the mapping $x \to x^*$ from $\mathbf{Z}_q$ to $\mathbf{N}$, and for all $x \in \mathbf{Z}_q$, x is the representation (in computer hardware) of the natural number x^*. Now, given two representable natural numbers x^* and y^*, the users of computer hardware are interested in computing the sum x^*+y^*, while their machine only knows about $x+y$. Luckily, the following proposition shows that computing $x+y$ allows us to infer the value of x^*+y^*, provided the latter is *representable*:

Proposition 1 *For all $x, y \in \mathbf{Z}_q$ the following are equivalent:*

$$\begin{array}{ll} (i) & x^* + y^* \in [0, q[\\ (ii) & (x+y)^* = x^* + y^* \end{array}$$

Proof
For all $x \in \mathbf{Z}_q$, x^* is an element of $[0, q[$ by definition. It follows that (i) is an immediate consequence of (ii). Conversely, if we assume that (i) is true, then since it is clear that $x^* + y^* = x + y$ modulo q, we conclude from definition (1) that $(x+y)^* = x^* + y^*$, which completes our proof. .

Hence we see that as long as $x^* + y^*$ is a *representable* natural number, it does not matter that our CPU should only know about addition in $\mathbf{Z}_q$: adding the two representations x and y gives us a representation of $x^* + y^*$, and all is well. However, there are cases when $x^* + y^*$ is not a representable natural number, in which case equality (ii) of proposition (1) does not hold. One way to think about this situation is saying that *the result of $x+y$ is wrong.* As it turns, there is nothing wrong with $x + y$ which is a perfectly correct answer to the question of adding two numbers in $\mathbf{Z}_q$. However, $x + y$ is not a representation of $x^* + y^*$, and in that sense, it is clearly wrong. This is where the notion of *carry for unsigned addition* naturally comes in:

Definition 2 *We call* carry for unsigned addition *the map $c : \mathbf{Z}_q \times \mathbf{Z}_q \to 2$:*

$$c(x, y) = 1 \quad \Leftrightarrow \quad x^* + y^* \notin [0, q[$$

Note that $2 = \{0, 1\}$ is our choice for denoting a boolean type, and the *carry for unsigned addition* is therefore a boolean function defined on the cartesian product $\mathbf{Z}_q \times \mathbf{Z}_q$, the purpose of which is to flag any situation when the result of $x + y$ is *wrong*, or to phrase it more accurately, any situation when $x + y$ is not a representation of the natural number $x^* + y^*$.

So we now understand what the carry is in the context of unsigned addition. As we shall see, there will be a notion of carry for unsigned subtraction, and similar notions for signed addition and subtraction (called *overflow* rather than *carry* in the case of signed operations). What all these notions have in common is their purpose: to warn users of computer hardware that the result of a primitive operation perfomed by the CPU on elements of $\mathbf{Z}_q$ does not yield a representation of the result to the corresponding operation in $\mathbf{N}$ or $\mathbf{Z}$.

For those writing computer software, the ability to compute the carry flag $c(x, y)$ is crucial, as we need to know when $x + y$ ceases to be an accurate representation of the result we care about. For this reason, designers of computer hardware have made the computation of the carry flag for addition, one of the fundamental primitives of a CPU. In practice, a user may execute the assembly instruction 'add rax, rbx' and the carry flag will be set equal to $c(x, y)$ where x and y are the values of the 64-bits registers rax and rbx respectively. Now what if we wanted to validate the value of $c(x, y)$ in software? The problem with definition (2) is that $c(x, y)$ is defined in terms of $x^* + y^*$ and we have no way to effectively compute that sum with the primitives introduced so far. One thing we can do however is compare two elements of $\mathbf{Z}_q$:

Definition 3 *Given $x, y \in \mathbf{Z}_q$ we say that $x \leq y$ if and only if $x^* \leq y^*$.*

Mathematically speaking, the order $\leq$ thus defined on $\mathbf{Z}_q$ may not be very interesting. However, it is interesting to us as it is a relation which can be tested by a CPU. Of course as we shall see, it is likely that this new hardware primitive is implemented in terms of *carry for unsigned subtraction*, and if we intend to use it in order to validate our *carry for unsigned addtion*, there is only so much validation we achieve. However, the point is not for us to check absolute correctness, but rather to gain a better understanding of how things are, and a simple consistency check is still welcome for that. Note that having defined the relation $\leq$ on $\mathbf{Z}_q$, we have implicitely defined $<$, $>$ and $>=$, and assuming we can test these conditions, we are able to compute the corresponding binary min and max functions. Hence, all of the following conditions can be tested:

Proposition 2 *For all $x, y \in \mathbf{Z}_q$, the following are equivalent:*

$$\begin{array}{rl}
(i) & c(x, y) = 1 \\
(ii) & x + y < \min(x, y) \\
(iii) & x + y < \max(x, y) \\
(iv) & x + y < x \\
(v) & x + y < y
\end{array}$$

where $c : \mathbf{Z}_q \times \mathbf{Z}_q \to 2$ is the carry for unsigned addition of definition (2).

Proof
Since for all $x, y \in \mathbf{Z}_q$, $\min(x, y) \leq x, y \leq \max(x, y)$ we immediately have the implications $(ii) \Rightarrow (iv)$, $(ii) \Rightarrow (v)$, $(iv) \Rightarrow (iii)$ and $(v) \Rightarrow (iii)$. In order to complete the proof, it remains to show that $(i) \Rightarrow (ii)$ and $(iii) \Rightarrow (i)$. We first show that $(i) \Rightarrow (ii)$. So we assume that $c(x, y) = 1$, and need to show that $x + y < \min(x, y)$. In other words we need to show that both inequalities $x + y < x$ and $x + y < y$ hold. However from definition (2), our assumption is equivalent to $x^* + y^* \notin [0, q[$, and we know from definition (1) that both x^* and y^* are elements of $[0, q[$. It follows that $x^* + y^*$ must be an element of $[0, 2q[$ without being an element of $[0, q[$. Hence it must be an element of $[q, 2q[$, from which we see that $x^* + y^* - q$ is an element of $[0, q[$, while being equal to $x + y$ modulo q. From definition (1) it follows that $(x + y)^* = x^* + y^* - q$, and since both $y^* - q < 0$ and $x^* - q < 0$ we obtain $(x + y)^* < x^*$ and $(x + y)^* < y^*$ which by virtue of definition (3) is equivalent to $(x + y) < x$ and $(x + y) < y$ as

requested. We now prove that $(iii) \Rightarrow (i)$. So we assume that $x+y < \max(x,y)$ and we need to show that $c(x,y) = 1$, or equivalently that $x^* + y^* \notin [0,q[$. However, our assumption implies that $x+y < x$ or $x+y < y$. So we shall distinguish two cases: first we assume that $x+y < x$. From definiton (3), this means that $(x+y)^* < x^*$. Hence, it is impossible that $x^* + y^* \in [0,q[$ as this would imply that $(x+y)^* = x^* + y^*$ (being equal to $x+y$ modulo q) and consequently $x^* + y^* < x^*$, yielding the conradiction $y^* < 0$. Likewise, if we assume that $x+y < y$ then $x^* + y^* \in [0,q[$ implies the contradiction $x^* < 0$. .