# Carry Bit: How does it work?

Veröffentlicht am **2013-12-22**

*Deutsch*

Most of us know from elementary school how to add multi-digit numbers on paper. Usage of the carry bit is the same concept, but not for base 10, not even for base 2, but for base 256 (in the old 8-bit-days), base 65536 (in the almost as old 16-bit-days), base 4294967296 (32 bit) or base 18446744073709551616 (64 bit), whatever is the word width of the CPU. Always using powers of two is common today, but it is quite possible that this will change from bits to trits (having three possible values, -1, 0 and 1) in the far future.

I do not think that application development should be dealing with low level stuff like bits and bytes, but currently common programming languages like Java, C, C++, C# and more would not let you get away with that, you have to be aware of the bits underlying their numeric types to some extent. So it is a good idea to spend some effort on understanding this. Unfortunately all of these languages are lacking the carry bit, but it is anyway useful to understand the concept.

I have been writing software since the beginning of the 80es. The computers available to me at that time where 8-bit with a 6502- or 6510-CPU and 1 MHz clock speed. Yes, it was 1 MHz, not 1 GHz. It was possible to program them in some BASIC-dialect, but that was quite useless for many purposes because it was simply too slow. Compiled languages existed, but were too clumsy and too big to be handled properly on those computers, at least the ones that I have seen. So assembly language was the way to go. In later years I have also learned to use the 680×0 assembly language and the 80×86 assembly language, but after the mid 90es that has not happened any more. An 8-bit CPU can add two 8-bit numbers and yield an 8-bit result. For this two variants need to be distinguished, namely signed and unsigned integers. For signed numbers it is common to use 2's complement. That means that the highest bit encodes the sign. So all numbers from 0 to 127 are positive integers as expected. 127 has the 8-bit representation 01111111. Now it would be tempting to assume that 10000000 stands for the next number, which would be +128, but it does not. Having the highest bit 1 makes this a negative number, so this is -128. Those who are familiar with modular arithmetic should find this easily understandable, it is just a matter of choosing the representatives for the residue classes. But this should not

disturb you, if you have no recent experience with modular arithmetic, just accept the fact that 10000000 stands for -128. Further increments of this number make it less negative, so 10000001 stands for -127 and 11111111 for -1. For unsigned numbers, the 8 bits are used to express any number from 0 to 255.

For introducing the carry bit let us start with unsigned integral numbers. The possible values of a word are $0$ to $2^n - 1$ where $n$ is the word width in bits, which would be 8 in our example. Current CPUs have off course 64-bit word width, but that does not change the principle, so we stick with 8-bit to make it more readable. Just use your imagination for getting this to 32, 64, 96 or 128 bits.

So now the bit sequence 11111111 stands for 255. Using an assembly language command that is often called ADD or something similar, it is possible to add two such numbers. This addition can typically be performed by the CPU within one or two clock cycles. The sum of two 8-bit numbers is in the range from 0 through 510 (111111110 in binary), which is a little bit too much for one byte. One bit more would be sufficient to express this result. The workaround is to accept the lower 8 bits as the result, but to retain the upper ninth bit, which can be 0 or 1, in the so called carry bit or carry flag. It is possible to query it and use a different program flow depending on it, for example for handling overflows, in case successive operation cannot handle more than 8 bit. But there is also an elegant solution for adding numbers that are several bytes (or several machine words) long. From the second addition onwards a so called ADC (“add with carry”) is used. The carry bit is included as third summand. This can create results from 0 to 511 (111111111 in binary). Again we are getting a carry bit. This can be continued until all bytes from both summands have been processed, just using 0 if one summand is shorter than the other one. If the carry bit is not 0, one more addition with both summand 0 and the carry bit has to be performed, yielding a result that is longer than the longer summand. This can off course also be achieved by just assuming 1, but this is really an implementation detail.

So it is possible to write a simple long integer addition in assembly language. One of the most painful design mistakes of current programming languages, especially of C is not providing convenient facilities to access the carry bit, so a lot of weird coding is needed to work around this when writing a long integer arithmetic.

Subtraction of long integers can be done in a quite similar way, using something like SBC (“subtract with carry”) or SBB (“subtract with borrow”), depending on how the carry bit is interpreted when subtracting.

For signed integer special care has to be taken for the highest bit of the highest word of each summand, which is the sign. Often a so called overflow but comes in handy, which allows to recognize if an additional machine word is needed for the result.

Within the CPU of current 64 bit hardware it could theoretically be possible to do the 64-bit addition internally bit-wise or byte-wise one step after the other. I do not really know the implementation details of ARM, Intel and AMD, but I assume that much more parallelism is used for performing such operation within one CPU cycle for all 64 bits. It is possible to use algorithms for long integer addition that make use of parallel computations and that can run much faster than

what has been described here. They work for the bits and bytes within the CPU, but they can also be used for very long numbers when having a large number of CPUs, most typically in a SIMD fashion that is available on graphics devices misused for doing calculations. I might be willing to write about this, if interest is indicated by readers.

It is quite interesting to look how multiplication, division, square roots, cube roots and more are calculated (or approximated). I have a lot of experience with that so it would be possible to write about hat. In short these operations can be done quite easily on modern CPUs, because they have already quite sophisticated multiplication and division functions in the assembly language level, but I have off course been able to write such operations even for 8-bit CPUs lacking multiplication and division commands. Even that I could cover, but that would be more for nostalgic reasons. Again there are much better algorithms than the naïve ones for multiplication of very long integers.

Share

Dieser Eintrag wurde veröffentlicht in **Arithmetic**, **English** von **bk1**. **Permanenter Link des Eintrags [http://brodowsky.it-sky.net/2013/12/22/carry-bit-how-does-it-work/]** .

5 GEDANKEN ZU “CARRY BIT: HOW DOES IT WORK?”

Pingback: Carry-Bit: Wie funktioniert das? | Karl Brodowskys IT-Blog

Manohar
sagte am **2014-03-26 um 12:54:09** :

explain ROR and ROL instrucions with examples

> **bk1**
> sagte am **2014-03-28 um 01:10:50** :
>
> Yes, I can do that in one of the future blogs or in a comment:
>
> Shift and Rotation Functions

Pingback: Shift- und Rotationsfunktionen | Karl Brodowskys IT-Blog

Pingback: How to recover the Carry Bit | Karl Brodowsky's IT-Blog