Integer binary representations

Posted on April 23, 2018 in

Integer binary reprensentations

I had never had to look closely at integer binary reprensentations in computers. The mental model I had for them was not wrong, but it turns out it is sub-optimal and there are better ways to do things. If you use high level abstractions and do not mainly work with fundamental types, or if you do not convert between integer types, you do not have to be mindful of the binrary representation of integers all the time as you program. Thus, before the last few weeks, I never had to look more closely at that, but I have started a project for which binary representation had a direct effect and I finally looked into them. I thought I would write down some notes and observations.

I am pretty sure that this is probably covered in all computer science degrees and so might seem trivial and basic knowledge to many programmers, but since I don't have a CS degree and never had to think much about binary representation, this was informative to me! I should also point out that although I have used the C and C++ standards as references, the concepts here are not exclusive to these languages.

Unsigned integers

The C standard is explicit in its definition of unsigned integers. It can be found in section 6.2.6.2, paragraph 1:

> For unsigned integer types other than unsigned char, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary

> representation; this shall be known as the value representation. The values of any padding bits are unspecified.

which is sometimes refered to as the pure binary representation. Other than the fact that the wording confused me at first[1], this basically describes a usual binary positional number notation. This is rather intuitive if you are familiar with positional number systems. I haven't found a place where the standard specifies whether the most significant bit position (*i.e.* the largest exponant bit) is the left most or the right most. The range of such a representation is the following:

$$0 \quad \text{to} \quad 2^{n} - 1$$

where `n` is the number of bits used in the representation. That is how you get to the range 0 to 255 (= 2^8 - 1) for an 8 bit number.

On the other hand, the C++ standard is more vague on the subject (at least up to C++17, the latest standard at the time of writing). As far as I can tell, it does not impose an explicit representation for its unsigned type. Section 6.9 of the standard deals with type representations and the closest I have found to having an explicit representation specified for unsigned types is footnote 45 which says:

> The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.

which would suggest, I think, that the type representations have to be compatible. But that is not exactly explicit. Then, in paragraph 3, section 6.9.1, the standard says:

> The range of non-negative values of a signed integer type is a subrange of the corresponding unsigned integer type, the representation of the same value in each of the two types is the same, and the value representation of each corresponding signed/unsigned type shall be the same. […] The signed and unsigned integer types shall satisfy the constraints given in the C standard, section 5.2.4.2.1.

This does not imply a pure binary representation. That said, most if not all C++ implementations in the field will actually have a pure binary representation for unsigned integers.

Signed integers

There are a few signed integer representations and for now at least[2], none of them is explicitely specified (or fobidden) by the C or C++ standards. I have looked at three different representations, the last one being the most common if I understand correctly.

Signed magnitude

Signed magnitude is the obvious solution to the problem: take the first bit and make it a sign bit, *i.e.* model the + or – sign as a 0 or a 1. This is actually the mental model I had for signed integers. The range of this solution is:

$$-(2^{(n - 1)} - 1) \quad \text{to} \quad 2^{(n - 1)} - 1$$

where `n` is the number of bits in the representation. This gives only one less number then the unsigned solution, because there are now two bit patterns that represent 0. For instance, for 8 bits, both:

`00000000` and `10000000`

reprensent the number zero (albeit, positive 0 and negative 0). This is not really a problem although comparison with zero now has to check for two cases.

Although the signed magnitude approach seems very natural, but it makes the hardware to do simple arithmetics operations (+, –) more complex to write. From what I read (I am no expert), this is mostly because the sign bit has to be dealt with before the operation and the circuitry becomes more complex. Mainly for this reason, other approaches have been developped.

One's complement

In the one's complement signed number representation, a negative number is obtained by taking the complement of it's unsigned representation, *i.e.* inverting every bit. The range of this binrary representation is the same as that of the signed magnitude representation, for the exact same reason: there are two ways of representing the number 0. So, again, the range is:

$$-(2^{(n-1)} - 1) \quad \text{to} \quad 2^{(n-1)} - 1$$

where `n` is the number of bits in the representation and again, there are two representations of zero, albeit not the same as for signed magnitude (e.g. for 8 bits):

`00000000` and `11111111`

This binary representation makes algorithms for the addition and subtraction of integers much simpler than the signed magnitude representation. With one's complement encoding, the usual algorithm that we do by hand for addition works and yeilds the correct value so long as the left most carry bit is added back to the result (if its 0, that's fairly easy ;-)). There is a way to remove the need to add back the carry bit and that is one characteristic of the next representation discussed.

Two's complement

Two's complement is the last method discussed (although not the last one there is, see Wikipedia's article for at least two more). This binary representation scheme is, today at least, the most prevalent signed integer representation in hardware. This encoding is almost the same as one's complement, except that once you have calculated the inverted bits of the number, you add one to it. Two's complement range is:

$$-(2^{(n-1)}) \quad \text{to} \quad 2^{(n-1)} - 1$$

where `n` is the number of bits in the representation. It should be noted that the range is not exactly the same as the one's complement: it is larger by one. This is explained by the fact that in this enconding scheme, there is only one representation of 0, and it is the same as the unsigned 0, *i.e.* all bits set to 0. Opposed to the one's complement representation, in this scheme, when all the bits are set to 1, the value encoded is not 0, but rather the smallest negative number (*i.e.* -1). For 8 bits, the first row of the following table illustrates this:

bits	**two's**	**unsigned**

	complement	
11111111	-1	255
01111111	127	127
10000000	-128	128
10000001	-127	129
11010111	-41	215
11111110	-2	254

Practically, when you get to the largest signed (positive) integer you can represent with the number of bits available, if you increment by one, the bit pattern becomes that of the lowest signed (negative) integer you can represent (which is illustrated in the second and third rows of the table above). After that, increasing the bit pattern by 1 will increase the value by one (fourth row of the table). Citing Wikipedia's entry on two's complement:

> Fundamentally, the system represents negative integers by counting backward and wrapping around.

This representation then as the interesting property that when going from unsigned to signed or vice-versa by only reinterpreting the bit pattern as if it were the destination type, the behavior is that of modulo 2^{n-1} wrapping (which is the same as the wrapping behavior mandated by the C standard for unsigned integers: wrapping to the value of the highest value plus one).

Another property of this encoding scheme, and probably the most important compared to what was discussed above, is that the carry bit for the usual algorithm of arithmetic operations (additions, subtractions) must simply be ignored to give the correct result. This is opposed, to the one's complement encoding scheme where it has to be added back. Thus, the arithmetic operations are even simpler to implement. This is probably a big reason why two's complement is the dominating binary representation right now.

Final thoughts

I never had to think much about the binary representation of the integers I used. I guess that can be attributed to me never working on the kind of applications where it matters or always working with a single architecture. Not sure I will use this knowledge very often, but in any case, it is good to know.

Notes

[1]The standard talks about the values represented and not the exponant, so that it talks about the series:

> 1, 2, 4, 8, 16…

of successive evaluations of the exponants of 2 rather than the successive exponants themselves, which actually start at zero. This had confused me at first, but Patrice Roy and Aaron Ballman helped me see that I had misinterpreted the standard. Thank you to both of them.

[2]In the 2018 Jacksonville meeting of the ISO C++ Committee, a paper has been presented to officially make signed integers two's complement. There is no certainty on the future of this paper, but the idea was also presented to the C standard committee and the discussions in both committee will take place to see if this is something they will pursue.

Hi, I'm Ghyslain

I am a medical physicist by day and a hobby programmer by night. You can follow me on Twitter and on GitHub. You can reach me via codeattempts [at] gmail [dot] com.