Optimising pointer subtraction with 2-adic integers

Dan Piponi

February 2024

1 Introduction

Here is a simple C type and a function definition:

```
struct A
{
    char x[7];
};

int diff(struct A *a, struct A *b)
{
    return a-b;
}
```

It doesn't seem like there could be much to say about that. The `-A-` structure is 7 bytes long so the subtraction implicitly divides by 7. That's about it. But take a look at the assembly language generated when it's compiled with gcc:

```
movl  4(%esp), %eax
subl  8(%esp), %eax
imull $-1227133513, %eax, %eax
ret
```

Where is the division by 7? Instead we see multiplication by -1227133513.

A good first guess is that maybe this strange constant is an approximate fixed point representation of $1/7$. But this is a single multiplication with no shifting or bit field selection tricks. So how does this work? And what is -1227133513? Answering that question will lead us on a trip through some suprising and abstract mathematics. Among other things, we'll see how not only can you represent negative numbers as positive numbers in binary using twos complements, but that we can also represent fractions similarly in binary too.

But first, some history.

2 Some n-bit CPUs

That's an Intel 4004 microprocessor, the first microprocessor completely contained on a single chip. It was a 4 bit processor equipped with 4 bit registers. With 4 bits we can represent unsigned integers from 0 to 15 in a single register. But what happens if we want to represent larger integers?

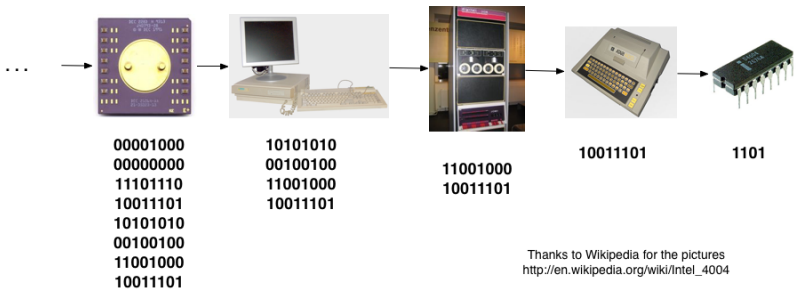
Let's restrict ourselves to arithmetic operations using only addition, subtraction and multiplication and using one register per number. Then a curious thing happens. Take some numbers outside of the range 0 to 15 and store only the last 4 bits of each number in our registers. Now perform a sequence of additions, subtractions and multiplications. Obviously we usually expect to get the wrong result because if the final result is outside of our range we can't represent it in a single register. But the result we do get will have the last 4 bits of the correct result. This happens because in the three operations I listed, the value of a bit in a result doesn't depend on higher bits in the inputs. Information only propagates from low bit to high bit. We can think of a 4004 as allowing us to correctly resolve the last 4 bits of a result. From the perspective of a 4004, 1 and 17 and 33 all look like the same number. It doesn't have the power to

distinguish them. But if we had a more powerful 8 bit processor like the 6502, we could distinguish them.

This is analogous to the situation we have with distances in physical world. With our eyes we can resolve details maybe down to 0.5mm. If we want to distinguish anything smaller we need more powerful equipment, like a magnifying class. When that fails we can get a microscope, or an electron microscope, or these days even an atomic force microscope. The more we pay, the smaller we can resolve. We can think of the cost of the equipment required to resolve two points as being a kind of measure of how close they are.

We have the same with computers. To resolve 1 and 17 we need an 8-bit machine. To resolve 1 and 65537 we need a 32-bit machine. And so on. So if we adopt a measure based on cost like in the previous paragraph, there is a sense in which 1 is close to 17, but 1 is even closer to 257, and it's closer still to 65537. We have this inverted notion of closeness where numbers separated by large (in the usual sense) powers of two are close in this new sense.

We have an interesting relationship between computing machines with different 'resolving' power. If we take an arithmetical computation on an N-bit machine, and then take the last M bits of the inputs and result, we get exactly what the M-bit machine would have computed. So an M-bit machine can be thought of as a kind of window onto the last M-bits onto an N-bit machine. Here's a sequence of machines:



Each machine provides a window onto the low bits of the previous machine in the sequence. But what happens at the "..." on the left? That suggests the bizarre idea that maybe all of these finite machines could be thought of as window to some infinite bit machine. Does that idea make any kind of sense?

I'll try to convince you that's a sensible idea by pointing out that it's something familiar to anyone who's taken a rigorous analysis course. (And I'll mention in passing that the above diagram illustrates a colimit in an appropriate category!)

Mathematicians (often) build the real numbers from the rational numbers by a process known as completion. Consider a sequence like

$$1, 14/10, 141/100, 1414/1000, \ldots.$$

The nth term is the largest fraction, with 10^n in the denominator, such that its square is less than 2. It's well known that there is no rational number whose square is 2. And yet it feels like this sequence ought to be converging to something. It feels this way because successive terms in the sequence get as close to each other as you like. If you pick any ϵ there will be a term in the series, say x, with the property that later terms never deviate from x by more than ϵ. Such a sequence is called a Cauchy sequence. But these sequences don't all converge to rational numbers. A number like $\sqrt{2}$ is a gap. What are we to do?

Mathematicians fill the gap by defining a new type of number, the *real* number. These are by definition Cauchy sequences. Now every Cauchy sequence converges to a real number because, by definition, the real number it converges to is the sequence. For this to be anything more than sleight of hand we need to prove that we can do arithmetic with these

sequences. But that's just a technical detail that can be found in any analysis book. So, for example, we can think of the sequence I gave above as actually being the square root of two. In fact, the decimal notation we use to write $\sqrt{2} = 1.414213, \ldots$ can be thought of as shorthand for the sequence $(1, 14/10, 141/100, \ldots)$.

The notion of completeness depends on an an idea of closeness. I've described an alternative to the usual notion of closeness and so we can define an alternative notion of Cauchy sequence. We'll say that the sequence $x_1, x_2, ...$ is a Cauchy sequence in the new sense if all the numbers from x_n onwards agree on their last n bits. (This isn't quite the usual definition but it'll do for here.) For example, $1, 3, 7, 15, 31, \ldots$ define a Cauchy sequence. We consider a Cauchy sequence equal to zero if x_n always has zero for its n lowest bits. So $2, 4, 8, 16, 32, \ldots$ is a representation of zero. We can add, subtract and multiply Cauchy sequences pointwise, so, for example, the product and sum of x_n and y_n has terms $x_n y_n$. Two Cauchy sequences are considered equal if their difference is zero. These numbers are called 2-adic integers.

Exercise: prove that if x is a 2-adic integer then $x + 0 = x$ and that $0x = 0$.

There's another way of looking at 2-adic integers. They are infinite strings of binary digits, extending to the left. The last n digits are simply given by the last n digits of x_n. For example we can write $1, 3, 7, 31, ...$ as $\ldots 1111111$. Amazingly we can add subtract and multiply these numbers using the obvious extensions of the usaul algorithms. Let's add .....1111111 to 1:

```
...11111111
...00000001
-----------
...00000000
```

We get a carry of 1 that ripples off to infinity and gives us zeroes all the way.

We can try doing long multiplication of $\ldots 111111$ with itself. We get:

```
...11111111
...1111111
```

```
...111111
...11111
...
-----------
...00000001
```

It's important to notice that even though there are an infinite number of rows and columns in that multiplication you only need to multiply and add a finite number of numbers to get any digit of the result. If you don't like that infinite arrangement you can instead compute the last n digits of the product by multiplying $11\ldots n$ digits $\ldots 111$ by itself and taking the last n digits. The infinite long multiplication is really the same as doing this for all n and organising it in one big table.

So $\ldots 1111111$ has many of the properties we expect of -1. Added to 1 we get zero and squaring it gives 1. It is -1 in the 2-adic integers. This gives us a new insight into twos complement arithmetic. The negative twos-complements are the truncated last n digits of the 2-adic representations of the negative integers. We should properly be thinking of twos-complement numbers as extending out to infinity on the left.

The field of analysis makes essential use of the notion of closeness with its δ and ϵ proofs. Many theorems from analysis carry over to the 2-adic integers. We find ourselves in a strange alternative number universe which is a sort of mix of analysis and number theory. In fact, people have even tried studying physics in p-adic universes. (p-adics are what you get when you repeat the above for base p numbers, but I don't want to talk about that now.) One consequence of analysis carrying over is that some of our intuitions about real numbers carry over to the 2-adics, even though some of our intuitive geometric pictures seem like they don't really apply. I'm going to concentrate on one example.

3 The Newton-Raphson Method

I hope everyone is familiar with the Newton-Raphson method for solving equations. If we wish to solve $f(x) = 0$ we start with an estimate x_n. We find the tangent to $y = f(x)$ at $x = x_n$. The tangent line is an approxima-

tion to the curve $y = f(x)$ so we solve the easy problem of finding where the tangent line crosses the x-axis to get a new estimate x_{n+1}. This gives the formula

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

.

With luck the new estimate will be closer than the old one. We can do some analysis to get some sufficient conditions for convergence.

The surprise is this: the Newton-Raphson method often works very well for the 2-adic integers even though the geometric picture of lines crossing axes doesn't quite make sense. In fact, it often works much better than with real numbers allowing us to state very precise and easy to satisfy conditions for convergence.

Now let's consider the computation of reciprocals of real numbers. To find $1/a$ we wish to solve $f(x) = 0$ where $f(x) = 1/x - a$. Newton's method gives the iteration $x_{n+1} = x_n(2 - ax_n)$. This is a well know iteration that is used internally by CPUs to compute reciprocals. But for it to work we need to start with a good estimate. The famous Pentium divide bug was a result of it using an incorrect lookup table to provide the first estimate. So let's say we want to find $1/7$. We might start with an estimate like 0.1 and quickly get estimates $0.13, 0.1417, 0.142848, \ldots$. It's converging to the familiar 0.142857...

But what happens if we start with a bad estimate like 1. We get the sequence:

```
1
-5
-185
-239945
-403015701065
-1136951587135200126341705
```

It's diverging badly. But now let's look at the binary:

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111011
1111111111111111111111111111111111111111111111111111111111111111111111111111111101000111
```

```
11111111111111111111111111111111111111111111111111111111111111000101011010110111
11111111111111111111111111111111111111111111101000100010101001100100011011011011 0111
11100001110011110110111011001000000100000000110110101101101101101101101101101101 0111
```

Our series may be diverging rapidly in the usual sense, but amazingly it's converging rapidly in our new 2-adic sense!

If it's really converging to a meaningful reciprocal we'd expect that if we multiplied the last n digits of these numbers by 7 then we'd get something that agreed with the number 1 in the last 7 digits. Let's take the last 32 digits:

```
10110110110110110110110110110111
```

and multiply by 7:

```
10100000000000000000000000000001
```

The last 32 bits are

```
00000000000000000000000000000001.
```

So if we're using 32-bit registers, and we're performing multiplication, addition and subtraction, then this number is, to all intents and purposes, a representation of $1/7$. If we interpret as a twos complements number, then in decimal it's -1227133513. And that's the mysterious number gcc generated.

4 Epilogue

There are many things to follow up with. I'll try to be brief.

Try compiling C code with a struct of size 14. You'll notice some extra bit shifting going on. So far I've only defined the 2-adic integers. But to get he reciprocal of every non-zero number we need numbers whose digits don't just extend leftwards to infinity but also extend a finite number of steps to the right of the "binary point". These are the full 2-adic numbers as opposed to merely the 2-adic integers. That's how the extra shifts can be interpreted. Or more simply, if you need to divide by 14 you can divide by 2 first and then use the above method to divide by 7.

I don't know how gcc generates its approximate 2-adic reciprocals. Possibly it uses something based on the Euclidean GCD algorithm. I wasn't able to find the precise line of source in a reasonable time.

An example of a precise version of the Newton-Raphson method for the p-adics is the Hensel's lemma.

The last thing I want to say is that all of the above is intended purely to whet your appetite and point out that a curious abstraction from number theory has an application to compiler writing. It's all non-rigourous and hand-wavey. Recommend reading further at Wikipedia. I learnt most of what I know on the subject from the first few chapters of Koblitz's book p-adic Numbers, p-adic Analysis, and Zeta functions. The proof of the Optimising pointer subtraction with 2-adic integers in that book is mind-blowing. It reveals that the real numbers and the p-adic numbers are equally valid ways to approximately get a handle on rational numbers and that there are whole alternative p-adic universes out there inhabited by weird versions of familiar things like the Riemann zeta function.

(Oh, and please don't take the talk of CPUs too literally. I'm fully aware that you can represent big numbers even on a 4 bit CPU. But what what I say about a model of computation restricted to multiplication, addition and subtraction in single n-bit registers holds true.)

5 Some Challenges

1. Prove from first principles that the iteration for 1/7 converges. Can you prove how many digits it generates at a time? 2. Can you find a 32 bit square root of 7? Using the Newton-Raphson method? Any other number? Any problems?

6 Acknowledgements

The pictures are lifted from Wikipedia. I can't find the precise wording of the magic incantation that makes it legal to do this. But if I tell you

Wikipedia is cool, recommend all their products, and give you a link to the GFDL maybe it's alright.