Arbitrary Sized Integers

Sam Moore, David Gow

July 24, 2014

Abstract

We have implemented arbitrary sized integers which sometimes don't segfault, using a combination of the C++ standard library and stand alone x86-64 assembly. Performance tests reveal that we would have been better off just using the GNU Multiprecision Library (GMP).

1 Integer Representation

A positive integer (natural number) can be written as the sum of smaller integers "digits" multiplied by powers of a base.

$$z = \sum_{i=0}^{\infty} d_i \beta^i \tag{1}$$

Where each digit $d_i < \beta$ the base. A set of β unique symbols are used to represent values of d_i. A fixed size representation truncates the sum at some $i = N$, which can represent all values $0 \leq z \leq \beta^{n+1} - 1$.

A seperate sign symbol (eg: '-') can be used to represent negative integers using the same digit sum.

Example in base 10 (decimal):

$$5682_{10} = 5 \times 10^3 + 6 \times 10^2 + 8 \times 10^1 + 2 \times 10^0 \tag{2}$$

In base 2 (binary) the same integer is:

$$1011000110010_2 = 1 \times 2^{12} + 0 \times 2^{11} + \; ... + 0 \times 2^0 \tag{3}$$

1.1 Representation on computer hardware

Computer hardware implements operations for fixed size integers. The base is $\beta = 2$ and the digits are $\{0, 1\}$. The most significant bit can be reserved for the sign instead of a digit.

We can construct larger size integers by considering some sequence of fixed size integers to be individual digits. In practice we will still be limited by the memory and processing time required for “big” integers.

For example, we can represent 5682_{10} as a single 16 bit digit or as the sum of two 8 bit digits. Each digit is being written in base 2 or 10 because there is not a universal base with $\geq 2^8$ unique symbols.

$$\begin{aligned} 5682_{10} &= 0001011000110010_2 = 10110_2 \times 2^8 + 110010_2 \times 2^0 \quad (4) \\ &= 22_{10} \times 2^8 + 50_{10} \times 2^0 \quad (5) \end{aligned}$$

2 Addition Algorithms

Addition $s = a + b$ is done by adding digits from least to most significant.

$$s = \sum_{i=0}^{\infty} (a_i + b_i)\beta^i$$

Considering the contributions to the sum of the i^{th} and $(i+1)^{\text{th}}$ digits:

$$\begin{aligned} s_i\beta^i + s_{i+1}\beta^{i+1} &= (a_i + b_i)\beta^i + (a_i + b_i)\beta^{i+1} \quad (6) \\ \implies s_i + s_{i+1}\beta &= (a_i + b_i) + (a_{i+1} + b_{i+1})\beta \quad (7) \\ & \quad (8) \end{aligned}$$

If the sum $a_i + b_i \geq \beta$, ie: It cannot be represented in base β, then we can rewrite this as:

$$\begin{aligned} s_i + s_{i+1}\beta &= \beta + (a_i + b_i - \beta) + (a_{i+1} + b_{i+1})\beta \quad (9) \\ &= (a_i + b_i - \beta) + (a_{i+1} + b_{i+1} + 1)\beta \quad (10) \end{aligned}$$

So we can use the digits $s_i = (a_i + b_i - \beta) < \beta$ and $s_{i+1} = (a_{i+1} + b_{i+1} + 1)$. This operation is the *carry*[1].

The x64 instruction set includes an *add with carry* instruction `adc` which will add fixed sized digits and set a flag to indicate a carry. This allows for easy adding of an array of digits representing an arbitrary sized integer.

3 Subtraction Algorithms

Similarly, subtraction $s = a - b$ is done from least to most significant digit. If the result of $a_i - b_i < 0$ then we *borrow* from a higher digit.

$$s_i + s_{i+1}\beta = \beta + (a_i - b_i + \beta) + (a_{i+1} - b_{i+1} - 1)\beta$$

[1] I'm pretty sure that is not a rigorous definition but close enough

The x64 instruction set also includes a *subtract with borrow* instruction `sbb` which will set a borrow flag.

4 Multiplication Algorithms

In general, the result of multiplying two n digit numbers may require up to $2n$ digits.

5 Division Algorithms

5.1 Naive Algorithm

5.2 Shifting Algorithm

6 Base conversion

Since humans are not very good at understanding binary, it is convenient to convert integer representations from one base to another.

7 Performance Comparison of IPDF::Arbint and GMP Integers

We repeated 1000 trials of the four basic operations on arbitrary integers initialised from `rand(3)`

Here are the average IR costs per operation collected using the *callgrind* tool with the memory analysis program *valgrind*.

Operation	IR Cost Arbint	IR Cost Gmpint	Arbint/Gmpint
*=	3957	255	15.6
/=	395008	388	1018.1
+=	252	98	2.5
-=	458	102	4.5

Figure 1: GMP wins

Clearly we are not as good at implementing arbitrary integer arithmetic as the GMP project. We are particularly bad at division. This is probably because we used the second algorithm on wikipedia.

Examining the GMP source code shows that the library is mostly implemented using highly optimised assembly which is selected based on the build target. We've used C++ classes with all their overhead. We also used a shittier division algorithm although our addition and subtraction are pretty similar.

8 Conclusion

Just use GMP.