Computer Arithmetic – A Programmer's Perspective*

Richard P. Brent
Computing Laboratory
University of Oxford

rpb@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk

*Presented at ARITH14, Adelaide, 14 April 1999.

arith14t typeset using LaTeX

Abstract

Advances in computer hardware often have little impact until they become accessible to programmers using high-level languages. In this talk we discuss several areas in which computer hardware, especially arithmetic hardware, can or should significantly influence programming language design. These include:

- vector units,
- floating-point exception handling,
- floating-point rounding modes,
- high/extended precision registers and arithmetic.

Relevant application areas include interval arithmetic, high-precision integer arithmetic for computer algebra and cryptography, and testing of hardware by comparison with software simulations.

Hardware, Systems & Software

Computing involves many levels of abstraction. Some of the most important are –

- Hardware (memories, processors, arithmetic units, ...)
- System Design (instruction set, connection and communication, ...)
- System Software (operating systems, compilers, display managers, web interfaces, editors, word processors, ...)
- User-level software/applications (written by/for a particular user or group of users)

Of course, one can argue about the boundaries between these levels, but our point is that computer arithmetic is at the "bottom" level and is hidden from the typical user by several intervening levels.

Computer Arithmetic and Computer Users

This is a symposium on *Computer Arithmetic*. For example, over the next three days there are sessions on algorithms for implementing

- addition, multiplication, division, sqrt etc
- floating-point arithmetic
- various number systems, etc

Some advances in Computer Arithmetic lead to faster/cheaper arithmetic units and these advances automatically help at the other levels (unless hindered by some other bottleneck, e.g. memory bandwidth).

Other advances give new capabilities, e.g. longer arithmetic, different rounding modes, hardware square root or elementary functions.

Such advances at the hardware level have little impact on most users unless the other levels make them readily accessible.

Example - Vector Operations

Since the 1970s vector operations have been popular as a way of obtaining high throughput and hiding memory latency (CDC, Cray, ...). However, building the vector arithmetic units is not enough.

- Other system components need to be designed to avoid bottlenecks (e.g. memories and I/O systems need high bandwidth, and vector registers are desirable to reduce memory traffic)
- Instruction sets need to include vector operations
- Compilers need to be written to take advantage of the vector operations (preferably automatically, or at least in a way which is backward compatible)
- Writers of system software may need to change their algorithms and programming style

Floating Point Arithmetic

Since about 1985 we have become accustomed to the IEEE Standard for (binary) floating-point arithmetic. It is worth remembering that the standard was the subject of heated debate and its introduction was controversial.

Base 2 ?

There was debate about the base/radix (two, ten, sixteen, ...). IBM had used base sixteen in the System 360 and 370 series, and was reluctant to change to base 2 although the technical advantages were clear. Others, e.g. Hull, thought that decimal was more natural and would have advantages in I/O, variable-precision arithmetic, etc.

The radix debate was "resolved" by deciding to have *two* standards, one for binary arithmetic and one more general, but the binary standard had by far the greater impact and is usually called *the* standard.

Rounding Modes and Exceptions

Probably the most controversial aspects of the IEEE binary floating-point standard were the different rounding modes and the treatment of exceptions, including the introduction of gradual underflow, infinities, and non-numbers (NaNs). These features could all be justified, but they made the standard much lengthier and more difficult to implement than would otherwise have been the case.

Now, 15 years later, it seems that different rounding modes, infinities, NaNs etc have had minimal impact, because they are difficult (or in many cases impossible) to use from high-level languages. Certainly the rounding modes (e.g. round up, round down) have been used in specialised packages to implement interval arithmetic, but interval arithmetic is itself rarely used. The default of round to nearest is probably used more than 99% of the time.

NaNs etc

Regarding NaNs, infinities, and unnormalised numbers – most high level languages do not even have a convenient way of denoting or testing for these. They are usually recognised only by the output routines (which, of course, is better than nothing).

A Topical Application for Computer Arithmetic

Although it is not new, cryptography is rapidly becoming more important with the growth of electronic commerce and increasing dependence on the security of computer systems which are connected to the outside world via the Internet. *Public key* or *asymmetric* cryptography is becoming as important as the more traditional *symmetric* cryptography.

The public key systems which are generally believed to be secure include

- RSA (Rivest-Shamir-Adleman)
- El Gamal (possibly over elliptic curves)

Also important as part of a public-key system is the *Diffie-Hellman* key exchange protocol, which is closely related to the El Gamal signature scheme.

Implementations of RSA etc

RSA, El Gamal and Diffie-Hellman require arithmetic operations over large integers (typically 1024 bits) or large finite fields (e.g. $GF(p)$ or $GF(2^n)$). For example, RSA and Diffie-Hellman require exponentiation mod N,

$$x \leftarrow y^e \bmod N$$

and this can be performed by repeated squaring and multiplication mod N, using the binary representation of the exponent e. The modulus N is fixed but large (in RSA it might be the product of two 512-bit primes).

On a typical RISC microprocessor, the operation of exponentiation modulo a 1024-bit number is *slow* and for the obvious algorithm the time increases like the cube of the number of bits.

For example, we might split 1024-bit numbers into 32×32-bit numbers, and use 32-bit multiplication (giving a 64-bit product) etc to implement multiplication of 1024-bit numbers.

Choice of (software) base

On many 32-bit systems it is hard to get a full 64-bit unsigned product (especially when writing in a high-level language) so the base may have to be reduced to 31 (or even 15) bits.

For division (the *mod* operation) we would like to divide a 64-bit number by a 32-bit number, getting 32-bit quotient and remainder, in order to estimate each 32-bit quotient “digit”. It is tricky to do this correctly (avoiding any overflows etc) on most systems, so again the base may have to be reduced, which increases the time and space required for multiple-precision arithmetic.

Similar comments (even more so) apply to 64-bit systems and 128-bit products.

Integers via Floating Point

For an implementation on a vector processor (Fujitsu VP2200) in Fortran we found it more convenient to use *floating-point* arithmetic for the integer operations ! The base was chosen to be 2^{26} because multiplication of 26-bit integers could be performed exactly in 64-bit floating-point arithmetic.

The program was later ported to machines with IEEE floating-point arithmetic (VPP300, SGI, DEC alpha, Sun Sparc etc) and on these machines the base was reduced to 2^{24}.

Extracting Digits

When performing multiple-precision arithmetic in base β, we often compute a product

$$p \leftarrow a \times b$$

and then want to extract the high and low base-β digits

$$p \operatorname{div} \beta \ \text{ and } \ p \bmod \beta.$$

On most machines it is hard to do this without performing redundant computations (e.g. dividing by β twice). For our implementation using floating point, we scaled by a suitable power of 2 and used the (vectorised) INT operation to extract the high digit, then subtracted it off and rescaled to get the low digit.

The inner loop required 9 operations and 4 cycles on the VP/VPP vector processors. A machine optimised for multiple-precision arithmetic might perform the inner loop (using base 2^{32} or larger) in 1 or 2 cycles.

Redundant number representations

When computing

$$x \leftarrow y^e \bmod N$$

we repeatedly form products mod N. It is not necessary for these products to be in the range $[0, N-1]$. If we relax the allowable range to say $[0, N]$ or even $(-N, 2N)$ the “mod” operation can be implemented more easily because the estimated quotient can be slightly incorrect.

Similarly, when using base β for multiple-precision arithmetic, we may allow digits in the range $[-1, \beta]$, not just in the range $[0, \beta - 1]$.

Factorisation of Fermat Numbers

An application of multiple-precision arithmetic is the factorisation of large integers using the *elliptic curve method* (ECM).

Fermat numbers are examples of integers with no “obvious” (algebraic) factors. They are integers of the form $2^{2^n} + 1$.

The tenth Fermat number $F_{10} = 2^{2^{10}} + 1$ was the “most wanted” number in various lists of composite numbers published after the factorization of $F_9 = 2^{2^9}$ in 1990 (by the *Special number field sieve*, or SNFS method).

Factorisation of F_{10}

Using ECM we found a 40-digit factor p_{40} on October 20, 1995. The 252-digit quotient p_{252} was proved to be prime. Thus, the complete factorization of F_{10} is

$$45592577 \cdot 6487031809 \cdot$$
$$4659775785220018543264560743076778192897 \cdot$$
$$p_{252} \cdot$$

The two “small” factors were known since 1953.

The decimal representation of p_{252} is

```
1304398744054881897274847687965099039466085308416118921868952957
7683241625147186357414022797757310489589878392884292384483114903
2913798729088601617946094119449010595906710130531906171018354491
6096191939124885381160807122996723228062178207531270144245773
```

The Amount of Computation

Overall, our factorization of F_{10} took 1.4×10^{11} multiplications (mod N), where $N = p_{40} \times p_{252} = c_{291}$. Numbers mod c_{291} were represented with 38 digits and base 2^{26} (on the VP100/VP2200) or with 41 digits and base 2^{24} (on the Sparc), so each multiplication (mod N) required more than 10^4 floating-point operations.

Special-purpose hardware capable of performing long integer operations efficiently would have saved quite a few KWhours !

Twin Primes and Pentium Bugs

A *twin prime* is a prime p such that $p-2$ or $p+2$ is also prime. In 1919 Brun showed that the sum of reciprocals of twin primes converges to a finite limit B, now called *Brun's constant*. In contrast, the sum of reciprocals of all primes diverges (Euler).

An amusing (and useful !) application of multiple-precision arithmetic is the computation of $B(x)$ (the sum over twin $p \le x$) and by extrapolation the estimation of B.

In 1976 I estimated B using the twin primes to 8×10^{10}. In the '90s, Thomas Nicely started to improve my estimate by going much further, using Intel 80486 and later Pentium processors. He has now gone past 10^{15}.

The leading bits of primes are not at all special (any pattern $1xxx\cdots$ can occur). Thus, forming sums of reciprocals of primes to high accuracy is an excellent test of reciprocal/division hardware (and software).

Discovery of the Pentium Bug

Nicely found a discrepancy between the sums of reciprocals computed on 80486 and certain Pentium processors, and eventually (after eliminating many other possibilities such as logical errors, compiler/library bugs, disk and memory problems) narrowed it down to a bug in the Pentium hardware. This was the (in-)famous *Pentium bug*. It was caused by incomplete entries in a table of reciprocals used for first approximations (although Nicely did not know this).

On December 20, 1994, Intel offered to replace faulty Pentium processors free of charge.

Nicely's computations up to 10^{14} have been confirmed (to at least 16 decimal places) by an independent computation by Kutrib and Richstein. Repeating part of them is a good test of reciprocal/division hardware/software.

Changes in Technology

As new technologies are developed, the tradeoffs between different algorithms and systems implementation decisions change. What used to be a good idea may no longer be one.

The best-known example is the idea of representing years with only two decimal digits – this was convenient and efficient in the days of 8-bit microprocessors and small, expensive memories, and when the year 2000 seemed a very long way off. Now it does not seem such a good idea!

Some other examples –

- Carry lookahead for addition – Manchester carry chains versus binary "parallel prefix" trees versus n-way trees $\cdots$
- Highly parallel computers (e.g. CM1)
- Virtual memory, memory hierarchies, ...
- Microcode

Integer Arithmetic

We are accustomed to thinking of computer arithmetic performed on fixed-length operands, e.g. 16/32/64 bits. In the past there were good reasons for this, and software was developed to handle "long" integers (though at considerable cost in speed and convenience).

For applications such as cryptography and symbolic computation, the algorithms are naturally expressed in terms of operations on arbitrary length (and actually rather large) integers.

Is it time to reconsider what *integer* and *integer arithmetic* should mean ? In other words, is it time to bring our definitions in line with those of the mathematicians, and allow arbitrary-length integers ? Surely cycle times are now fast enough and memory cheap enough to make this practical (if it will ever be practical).

Ideal Integer Arithmetic

In 1981 I suggested how integer arithmetic might be perfected in a paper entitled *An idealist's view of semantics for integer and real types*, but clearly this was an idea before its time.

Most of the 1981 paper was concerned with programming language semantics and software, but I did consider the possibility of a hardware (or mixed hardware/microcode/software) implementation.

I suggested that each word could have a "tag" bit to indicate if the remaining bits were to be regarded (in integer operations) as single-precision integers or as pointers to multiple-precision integers. Then the facility for performing multiple-precision arithmetic would not significantly increase the cost of operations on small integers. Multiple-precision arithmetic, when necessary, could be performed by software with possible assistance from hardware.

Conclusion

It's fun to work on Computer Arithmetic, but if you want your work to have wide impact you need to

- Think about its implications for computer systems design and systems/user-level software;
- Talk to systems designers and try to get your ideas incorporated in complete systems and made accessible to users.
- Keep up with developments in technology, and consider how new developments change the tradeoffs.
- Don't be afraid to use old ideas if new technologies make them viable.

References

[1] R. P. Brent, On the precision attainable with various floating-point number systems, *IEEE Trans. on Computers* C-22 (1973), 601–607.

[2] R. P. Brent, Irregularities in the distribution of primes and twin primes, *Math. Comp.* 29 (1975), 43–56.

[3] R. P. Brent, Algorithm 524: MP, a Fortran multiple-precision arithmetic package, *ACM Trans. Math. Software* 4 (1978), 71–81.

[4] R. P. Brent, An idealist's view of semantics for integer and real types, *Australian Computer Science Communications* 4 (1982), 130–140.

[5] R. P. Brent, Factorization of the tenth Fermat number, *Math. Comp.* 68 (1999), 429–451.

[6] M. S. Cohen, V. C. Hamacher and T. E. Hull, CADAC, an arithmetic unit for clean decimal arithmetic and controlled precision, in *ARITH-5, Proc. Fifth Symposium on Computer Arithmetic* IEEE/CS Press, 1981, 105–112.

[7] W. Diffie and M. E. Hellman, New directions in cryptography, *IEEE Trans. on Information Theory* IT-22 (1976), 644–654.

[8] T. El Gamal, A public-key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. on Information Theory* IT-31 (1985), 469–472.

[9] T. E. Hull and M. S. Cohen, Towards an ideal computer arithmetic, in *ARITH-8, Proc. Eighth Symposium on Computer Arithmetic* (M. Irwin and R. Stefanelli, eds.), IEEE/CS Press, 1987, 131–138.

[10] T. E. Hull, M. S. Cohen, J. T. M. Sawchuk and D. B. Wortman, Exception handling in scientific computing, *ACM Trans. Math. Software* 14 (1988).

[11] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754–1985.

[12] W. Kahan and J. Palmer, On a proposed floating-point standard, *SIGNUM Newsletter*, Oct. 1979, 13–21.

[13] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third edition). Addison-Wesley, Menlo Park, 1997.

[14] D. J. Kuck, *High Performance Computing*, Oxford University Press, Oxford, 1996.

[15] U. W. Kulisch and W. L. Miranker (editors), *A New Approach to Scientific Computation*, Academic Press, New York, 1983.

[16] T. R. Nicely, Enumeration to 10^{14} of the twin primes and Brun's constant, *Virginia Journal of Science* 46 (1995), 195–204. Reviewed in *Math. Comp.* 66 (1997), 924–925.

[17] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM* 21 (1978), 120–126.

[18] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition, John Wiley and Sons, 1996.