ACM *Sigplan Notices* **27**,1 (Jan. 1992),95-98.

Computing A*B (mod N) Efficiently in ANSI C

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 501-4956 (818) 986-1360 FAX

Copyright © 1991 by Nimble Computer Corporation

The modular product computation A*B (mod N) is a bottleneck for some public-key encryption algorithms, as well as many exact computations implemented using the Chinese Remainder Theorem. We show how to compute A*B (mod N) efficiently, for single-precision A,B, and N, on a modern RISC architecture (Intel 80860) in ANSI C. On this architecture, our method computes A*B (mod N) faster than ANSI C computes A%N, for unsigned longs A and N.

INTRODUCTION

Many exact integer computations, rather than being performed using multiple-precision arithmetic, are performed instead over a number of single-precision modular rings or fields, with the results being then combined using the Chinese Remainder Theorem [Moses71] [Knuth81] [Gregory84] [Schroeder86]. Organizing the computation in this way avoids many multiple-precision arithmetic operations and the storage management overheads that these operations entail. Unfortunately, unless the hardware architecture and computer language offers double-precision multiplication and double-precision Euclidean division-with-remainder operations, the cost of simulating these double-precision operations can eliminate any savings from the modular approach. In order to profit from modular techniques, therefore, one must either restrict the program to use only half-precision integer operations (e.g., 15 bits), or seek another way to avoid double-precision operations.

The popularity of ANSI C [ANSI88] has led to architectures which efficiently implement its primitives, including its unsigned arithmetic primitives. While most of these primitives are unremarkable, ANSI C offers an unusual capability in its unsigned integer multiplication operation, which returns the low order bits (e.g., 32 bits) of the nonnegative double-precision product *without raising an overflow interrupt if the high order bits are non-zero*. Thus, although there is no *simple* way in ANSI C to produce exactly the high order bits of this product,[1] the low order bits (equivalent to the product (mod 2^w), where w is the word size in bits) are trivially obtained through unsigned integer multiplication. While exact higher-precision multiplication can often be performed using floating point operations, many architectures are limited to the 24-bit and 53-bit precisions of the IEEE-754 single and double formats, and hence cannot produce a full 64-bit integer product.

Given these restrictions, computing a full 64-bit double-precision integer product can be painfully expensive, while computing a double-precision quotient/remainder can be much worse. Yet because modular multiplication is so important, one is led to consider alternative computations in which double-precision operations are minimized or eliminated. We present such an alternative modular multiplication computation, which also maps efficiently onto certain modern RISC architectures. Other approaches to modular products, including those more appropriate for the case with high-precision A,B, and N, are given in [Blakeley83] [Gibson88] [Kukihara89].

---

[1]
```
void emul(d,q,r,hp,lp) unsigned long d,q,r,*hp,*lp; /* 31x31+31 bit multiplication. */
{unsigned long l; double fd,fr,fl;
 fd=d; fr=r; fl=l=(d*q+r)&0x7fffffff; *lp=l;
 *hp=((q&0x7fff0000)*fd+(((q&0xffff)*fd+fr)-fl))/2147483648.0;}
```

ACM *Sigplan Notices* **27**,1 (Jan. 1992),95-98.

COMPUTING A*B (MOD N)

Let W be the word size of the ANSI C "long integer" implementation. Then W-1 is the largest unsigned long integer, and W/2-1 is the largest signed long integer. Since we assume that N is a positive (but signed) integer, $0<N<W/2$.

Let A,B be nonnegative integers such that $0\le A,B<N$. Let P=A*B; i.e., P requires double precision, and can be as large as $W^2/4-2W+4$. Let R=mod(P,N)=mod(A*B,N). In other words, there exists an integer Q such that R=P-Q*N=A*B-Q*N, and $0\le R<N$.

If we could calculate Q, then we could calculate R using two double precision multiplies and a double precision subtract; i.e., R=A*B-Q*N. However, since $0\le R<N<W/2<W$, we can more efficiently calculate R using single-precision ANSI C unsigned (modular) arithmetic. In other words, R=mod(R,W)=mod(A*B-Q*N,W)=mod(mod(A*B,W)-mod(Q*N,W),W), which follows from the fact that x->mod(x,W) is a ring homomorphism. This calculation requires only 2 single-precision unsigned multiplies and 1 single-precision unsigned subtraction.

Unfortunately, calculating Q precisely requires that we calculate Q=floor(P/N) precisely, and since P=A*B is double precision, a precise Q needs 1 double-precision multiply and 1 double-precision quotient. In this case, we would do better by simply using a brute force double-precision Euclidean division which would provide us with the correct remainder that we seek.

On the other hand, we can use an approximation Q' to Q, and perform instead the calculation R'=P-Q'*N=A*B-Q'*N. So long as $-N\le R'<N$, we can quickly recover R=(R'<0)?(R'+N):R', and this expression can be computed using ANSI C *signed* single-precision integer arithmetic. Furthermore, we can compute the bit pattern for R' using ANSI C *unsigned* single-precision integer arithmetic, since $|R'|<W$.

An approximation to Q is obtained by the computation Q''=round(float(A)*float(B)/float(N)). So long as the precision of the floating point operations is somewhat greater than the precision of integers, then Q'' will be close enough—i.e., Q'=Q''. If the floating point calculation were perfectly accurate, then the integer Q'' would never deviate more than ±.5 from the rational number A*B/N, which would imply that $|R'|\le N/2$. If the floating point calculation involved some round-off error, then this error would be relative to the size of Q''. This error is greatest for the largest Q'', which is about the size of N, but the error is still dominated by the round-to-integer operation. For example, if we utilize IEEE 64-bit precision having 53 bits of accuracy, then it is certainly true that $|R'|<N/2(1+2^{-k})$, where k<<22 (=53-31) bits.[2]

To summarize, we can perform the modular product calculations as follows:

```
{     double dn = n;                       /* 31 bits of n>0 */
      double da = a;                       /* 31 bits of a≥0 */
      double db = b;                       /* 31 bits of b≥0 */
      double dp = da*db;                   /* ~53 bits of a*b */
      double dq = dp/dn;                   /* ~51 bits of a*b/n */
      unsigned long qpp = dq+0.5;          /* q''=round(q) */
      unsigned long rp = a*b-qpp*n;        /* r' */
      long r = (rp&0x80000000)?(rp+n):rp;}
```

This calculation involves 1 double-precision floating point multiplication, 1 double-precision floating-point division, 2 double-precision floating-point additions (the implicit truncation to an

---

[2]Indeed, on a test of a few hundred million triples A,B,N whose size flirted with 2^{31}, $|R'|>N/2$ fewer than 10 times, and the difference never exceeded $201\approx 2^8<2^{31-22}=2^9$.

integer is equivalent to a floating-point addition by a constant which right-justifies the mantissa),[3] 2 single-precision unsigned integer multiplications, 1 single-precision unsigned integer subtraction, 1 integer sign test,[4] and 50% of a single-precision unsigned integer addition.

REPLACING DIVISION WITH MULTIPLICATION

Some hardware architectures do not provide a fast floating-point divide instruction, and hence a double-precision floating-point division can require a very large number of cycles.[5] In such a case, it is often advantageous to keep the inverse of `n` (`1.0/fn`) in a "cache", since the value of `n` changes much less often than does the value of `a` and `b`. If this caching of n^{-1} is done, then the double-precision division can be replaced by a double-precision multiply, as in the following sequence:

```
if (n!=saved_n)                             /* Cache hit? */
 {saved_n=n; dn=n; din=1.0/dn;}             /* ~53 bits of 1/n */
{    double da = a;                         /* 31 bits of a≥0 */
     double db = b;                         /* 31 bits of b≥0 */
     double dp = da*db;                     /* ~53 bits of a*b */
     double dq = dp*din;                    /* ~51 bits of a*b/n */
     unsigned long qpp = dq+0.5;            /* q''=round(q) */
     unsigned long rp = a*b-qpp*n;          /* r' */
     long r = (rp&0x80000000)?(rp+n):rp;}
```

The 64-bit Intel 80860 architecture [Intel89] has very fast floating-point add, subtract and multiply operations, as well as the usual complement of integer operations. However, it does not have any double-precision integer operations, nor does it have a hardware floating-point division operation. As a result, the code sequences demonstrated here work very well for this architecture. In particular, the "extra" integer multiplication "`a*b`" can be completely overlapped with the other operations, and hence is free.

TIMINGS

Using the Metaware ANSI C compiler with maximum optimization on a 33MHz 80860XR, a single "mod" computation A%N for unsigned longs takes 2.94µsec. Our ANSI C method using a cache for `1/n` takes 2.32µsec. If we replace `trunc(dq+0.5)` with `round(dq)`, by editing the assembler code to utilize the hardware round-to-integer capability, the time drops to 2.24µsec. Handcoded assembly takes 1.78µsec; the savings is due to reducing the movement of information between the integer and the floating point registers, which motion is very expensive on this machine. Finally, we reduced this time to 1.72µsec by utilizing the pipelined capabilities to float `a` and `b` simultaneously. Thus, we have gained about 25% by hand coding over the C version, and at least 40% over an unsigned "mod" operation.

A C compiler that did a better job of minimizing traffic between the integer and floating point registers would eliminate most of the incentive for hand-coding this calculation. Faster speed is unlikely for a single computation due to the length of the critical path; *vectorizing*, however, could reduce the amortized cost below 1µsec by keeping the multiplier busier.

---

[3]In Ada [DoD], the conversion of floating-point to integer involves rounding, not truncation, and thus avoids the addition of 0.5. Some modern architectures—e.g., Intel 80860 [Intel89]—have a single hardware "round-to-integer" instruction, which can be used by an assembly language programmer to avoid this addition.

[4]We would like to write "rp&0x800000" as "((long) rp)<0", but the conversion could trap before the sign is checked! We must therefore rely on the compiler to optimize the mask test into a sign test.

[5]On the Intel 80860, a double-precision divide takes 38 clocks; a double-precision multiply takes 4 clocks.

Copyright © 1991 by Nimble Computer Corporation

ACM *Sigplan Notices* **27**,1 (Jan. 1992),95-98.

CONCLUSIONS

We have shown how to efficiently compute the modular product A*B (mod N) in ANSI C for single-precision A,B, and N, without splitting the integers into smaller pieces. We have implemented this algorithm in Kyoto Common Lisp [Yuasa90], which compiles Lisp into ANSI C. Modular multiplication is heavily used in the Lisp-based DOE Macsyma symbolic algebra system [Wang75]. The efficiency of our method relative to other approaches depends critically on the details of the hardware architecture, but we have found it effective for the 64-bit Intel 80860 architecture. In particular, we can compute the modulo-reduced product faster than standard C can modulo reduce a single number.

REFERENCES

Alia, G., and Martinelli, E. "A VLSI Modulo *m* Multiplier". *IEEE Trans. Computers 40*,7 (July 1991),873-878.

ANSI C. *Draft Proposed American National Standard Programming Language C*. ANSI, NY, 1988.

Blakeley, G.R. "A Computer Algorithm for Calculating the Product AB modulo M". *IEEE Trans. Comps. C-32*,5 (May 1983),497-500.

DoD. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, USGPO, 1983.

Gibson, J.K. "A Generalization of Brickell's Algorithm for Fast Modular Multiplication". *BIT 28* (1988),755-764.

Gregory, R.T., and Krishnamurthy, E.V. *Methods and Applications of Error-Free Computation*. Springer, 1984.

Intel Corp. *i860™ 64-bit Microprocessor Programmer's Reference Manual*. #240329-002, 1989.

Knuth, D.E. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*. Addison-Wesley, MA, 1981.

Kukihara, K. "Euclidean Type Algorithm for Multiplication Modulo P, II". *J. Info. Proc. 12*,2 (1989),147-153.

Moses, Joel. "Algebraic Simplification: A Guide for the Perplexed". *CACM 14*,8 (Aug. 1971),527-537.

Rivest, R.L., Shamir, A., and Adleman, L.A. "A Method for Obtaining Digital Signatures and Public Key Cryptosystems". *CACM 21* (1978),120-126.

Schroeder, M.R. *Number Theory in Science and Communication*. Springer-Verlag, New York, 1986.

Wang, P. "MACSYMA—A Symbolic Manipulation System". *Proc. Int'l. Comp. Symp*. (1975), Vol. I, 103-109.

Yuasa, T. "Design and Implementation of Kyoto Common Lisp". *J. Info. Proc. 13*,3 (1990),284-295.

Copyright © 1991 by Nimble Computer Corporation