A fast and flexible software library for large integer arithmetic

Antoon Bosselaers, René Govaerts, and Joos Vandewalle

Katholieke Universiteit Leuven, Dept. Electrical Engineering-ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium
antoon.bosselaers@esat.kuleuven.ac.be

18 April 1994

Abstract

An ANSI C library of subroutines for multiprecision operations on unsigned integers is presented, that is both fast and flexible. Usability and applicability of such a library are shown to depend both on the basic design decisions as well as on the library's actual functionality. Basic design decisions are the choice of programming language, the representation of multiprecision integers, error handling, and memory management. The high-level functionality includes modular exponentiation, square roots modulo a prime, Jacobi symbol, and a prime generation package, providing the basis for the implementation of many well known and widely used public key algorithms.

1 Introduction

Since the advent of public-key cryptography in 1975 many secure public-key algorithms have been proposed. An important subset is formed by the algorithms based on modular arithmetic with large unsigned integers, like Diffie-Hellman, RSA, ELGamal, Guillou-Quisquater, Schnorr, DSA (cf. [Sim92]). The development of a fast, portable, and well-documented software library providing the building blocks for these algorithms is therefore an interesting research topic, with many practical applications.

Publicly available libraries for arithmetic operations on large integers usually suffer from bad performance and restricted applicability, often lack decent documentation and have very restricted error handling capabilities. We present a fast and flexible software library, in the development of which great care has been taken of speed, maintainability and portability of all available functions. It has been written in ANSI C, a standardized high-level language, providing both the ease of maintaining your program and the possibility of porting it to other platforms, as well as speed by means of advanced optimizations.

The current arithmetic functionality of the library can be roughly grouped into the following categories: conversion between types, I/O, low-level arithmetic (bitwise shift, bit manipulation, copying, comparing, addition, subtraction, multiplication, squaring, integer division, modular reduction, greatest common divisor, modular inverse), and high-level arithmetic (Chinese Remainder Theorem, modular exponentiation, square root modulo a prime, Jacobi symbol, pseudo-random number generation, generation of primes, 'strong' primes, and RSA key sets).

Sections 2 through 5 discuss the basic design decisions of the library: choice of programming language, representation of multiprecision integers, error handling, and memory management. The functionality is dealt with in Section 6. A number of non-portable extensions written in 32-bit 80x86-assembly language are discussed in Section 7. Finally, an idea of the performance is given in Section 8.

2 Programming language

For the implementation of our multiple precision library we chose the ANSI C programming language [ISO9899], one of the most widely used and best supported computer languages. It is a standardized high-level language, that is at the same time in many aspects still quite close to assembly language, reflecting the architecture of contemporary computers. Therefore, C maps well into most machine languages, allowing compilers to produce efficient code. This unique mixture provides on the one hand the ease of maintaining your program and porting it without difficulty to other platforms, and on the other hand the opportunity to optimize your code beyond what is thought possible for a high-level language.

Many good C compilers currently exist, and most modern implementations fully support ANSI C. This allows for writing portable, yet very efficient programs. Our library was developed on a PC with Borland C 3.1 and Watcom C/386 9.5, and has been been successfully tested in the same environment using Microsoft C 7.0 and in an ULTRIX environment using the `cc` and GNU C compilers.

3 Representation of multiprecision integers

The internal radix representation is introduced. It is shown that it is desirable to choose this radix as large as possible. These choices are translated into ANSI C.

3.1 Choice of internal representation

For operations on arbitrarily large, unsigned integers a representation in radix b notation is used, where b can be in principle any integer ≥ 2. That is, an arbitrary nonnegative integer x is represented as a sequence of radix b digits $\langle x_0, x_1, \ldots, x_{k-1} \rangle$, where

$$x = \sum_{i=0}^{k-1} x_i b^i , \quad 0 \leq x_i < b \quad \text{for } i = 0, 1, \ldots, k-1.$$

The number of available radix b digits for the representation of x is either fixed or a parameter coded in the representation of x as an additional digit (if necessary with respect to a different base). The former approach implies that all integers appearing as input or output parameters of multiprecision operations are of fixed, although not necessarily equal, length, i.e., they are represented with a fixed number of radix b digits. A typical example is a squaring, where the result will be represented with twice as many digits as the argument. Fixed length representation is at the same time an advantage and a disadvantage. The advantage is that constant parameters in an operation allow for additional optimizations. The disadvantage is that the operations are explicitly designed to handle integers of a given length. Smaller integers can be handled by padding them with a sufficient number of leading zeroes, at the cost of efficiency. Larger integers cannot be handled at all by these operations. Using the length as an extra parameter in the representation of a multiprecision integer results in slightly more involved code, but all operations can handle multiprecision integers of *any* length, only limited by the available computer memory. This approach is clearly much more flexible. This is especially important in the context of cryptography, where on a regular basis the size of the integers has to be increased to account for the developments in cryptanalysis. Therefore, we will represent an arbitrary nonnegative integer x as a sequence of radix b digits $\langle k, x_0, x_1, \ldots, x_{k-1}\rangle$, where the first digit is the number of digits that follow, the second is the least significant digit of x, and the last digit is the (nonzero) most significant digit of x. Hence,

$$x = \sum_{i=0}^{k-1} x_i b^i\,, \quad 0 < x_{k-1} < b \text{ and } 0 \leq x_i < b\,, \text{for } i = 0, 1, \ldots, k-2.$$

The option to use the same base for the length digit k as for the other digits x_i is no real limitation, since in practise b will be at least equal to 2^{16}, allowing for integers up to 1 Mbit long.

3.2 Choice of radix

The best choice for b will be determined by the computer, the programming language, and the compiler. In particular, b should be chosen such that whenever a power of b acts as multiplier, divisor or modulus, the result is easily obtained. The most obvious choice for b will therefore be one of the available integer types, in which case multiplication with b^k is reduced to a shift over k digits to the left, a division by b^k to a shift over k digits to the right (i.e., discarding the least significant k digits), and a reduction modulo b^k to discarding all but the least significant k digits.

All multiprecision operations are performed using a number of primitive single precision operations. The larger b is, the smaller the number of single precision radix b operations to perform the same multiprecision operation, and hence the faster the latter will be. But one of these primitive operations is the single precision multiplication, returning a two-digit product. This means that besides a basic integer type that can represent the values 0 through $b-1$, we also need an integer type that is able to represent the values 0 through $(b-1)^2$. In addition, we normally want the ability to add and multiply concurrently [Knu81, Algorithm 4.3.1M], and hence we need an integer type that is able to represent the values 0 through b^2-1, i.e., a type which is at least twice as long as the basic type.

3.3 Translation of our choices to C

These choices are in our library translated in the following definitions of ANSI C macro's and integer types:

`BITSINWORD` is the number of bits in one (machine) word. An arbitrary nonnegative integer x is represented in radix $b \stackrel{\text{def}}{=} 2^{\texttt{BITSINWORD}}$ notation.

`word` is an unsigned integral type that contains *precisely* `BITSINWORD` bits. This choice allows to use casts for the conversion from an integer type containing more than `BITSINWORD` bits to the integer type `word`. A `word` has the minimum value of zero, and a maximum value of `WORD_MAX`.

`WORD_MAX` is the largest value that can be held by an object of type `word`, being $b-1$. The smallest value is, of course, zero.

`dword` is an unsigned integral type that can contain at least $2 \cdot$ `BITSINWORD` bits. A `dword` has the minimum value of zero, and a maximum value of at least $(\texttt{WORD\_MAX}+1)^2 - 1$.

Possible choices for `BITSINWORD` are 8, 16, and 32. The corresponding types of `word` and `dword`, as well as the possible values of `WORD_MAX` are given in Table 1.

BITSINWORD	8	16	32
WORD_MAX	0xFFU	0xFFFFU	0xFFFFFFFFU
word	unsigned char	unsigned short	unsigned int
dword	unsigned short	unsigned long	unsigned long

Table 1: Possible choices for `BITSINWORD`, and derived types and constant. The constant suffix `U` specifies an *unsigned* integer constant.

The first option is so far only used for testing purposes, since an ANSI C compiler provides at least 16-bit (`short`) `int`'s and 32-bit `long int`'s (the second option). On 8-bit computers it might nevertheless be faster to use the first option, although in that case only integers up to 2048 bit long can be represented (see Section 3.1). The second option should work on all currently available ANSI C compilers, and is our current choice. The third option will work on ANSI C compilers having 32-bit integers and 64-bit long integers, but has not been tested yet.

4 Error Handling

A very important aspect of any program is the way errors are handled. This is certainly true of libraries: good error handling makes or breaks a library. High level library routines are built on lower level routines, which in turn are built on basic level routines. An error introduced on the highest level may only result in program failure on the lowest level. A message explaining place and reason of program failure will therefore in most cases not be enough to discover what exactly caused the failure. This explains the need for extensive error tracing capabilities in any good library, but certainly in a multiprecision library, where there exist a myriad of dependencies, and in which the complexity of certain operations is quite impressive. In our library the occurrence of an error therefore results in following actions:

1. a message describing place and kind of error is pushed on an internal stack of trace messages. No two messages are the same, i.e., an error message uniquely defines the place where the error occurred;
2. all local memory that was dynamically allocated is freed again;
3. an error code indicating the kind of error is returned to the calling function.

This allows the calling function to take appropriate measures in case of an error, i.e., it can either decide to undertake precisely the same actions and return an error code to its calling function (which is the case for all functions in our library), or it can decide to communicate the error to the user (which will be the case at some point in any program). The advantage of this approach is that the failure of a particular function to execute properly can be traced back to the point and function where the error was introduced. This makes it possible to determine (very quickly) what exactly went wrong, which immensely facilitates the debugging operation.

5 Memory Management

As explained in Section 3.1 the length of a multiprecision integer is an additional parameter in its internal representation, and hence the necessary amount of memory for storing the multiprecision integer is in most cases not known on beforehand, i.e., at compile time. We therefore need a mechanism of dynamic size memory allocation, that allows both for easy allocation and freeing of dynamic size memory. Many solutions exist to this problem, but we decided once again to use a stack based solution [Ses89, Chapter 10]. An item on the memory stack can either be an address of dynamically allocated memory or a mark. Three functions are available for manipulating the memory stack. The function `mark_stack` pushes a mark on the stack. The function `alloc_stack` first calls the standard function `calloc` to allocate the necessary amount of memory for a multiprecision integer. It then pushes the address onto the stack and returns the address. Finally, the function `free_stack` pops items off the stack until a mark is found, or until the stack is empty, if no mark is found. If the item popped off is a memory address, the corresponding memory is freed by a call to the standard function `free`.

Memory management within a function becomes now very easy. First, we declare as many pointers to `word` as we need multiprecision integers in that particular function. Then, before any dynamic size memory allocation is done, we mark the memory stack by a call to `mark_stack`. Next, we allocate memory for the different multiprecision integers by consecutive calls to `alloc_stack`. Before leaving the function all memory is freed by a single call to `free_stack`.

6 Functionality

6.1 Conversion between types and I/O

The library comprises routines for conversion in both directions between on the one hand multiprecision integers and on the other hand the integral types `word` and `dword`, the floating type `double`, and character strings. In case the destination type has fewer significant bits available than the source type, only the appropriate number of significant bits are returned. The conversion from or to character strings is performed without loss of significant bits, and is especially useful whenever multiprecision integers are part of messages (e.g., hash function arguments), or whenever multiprecision operations have to be performed on messages (e.g., hash function results in digital signature schemes).

Routines for reading and writing multiprecision variables in a standard format and in decimal format from standard I/O or from a specified file are provided for.

6.2 Low level arithmetic

Addition, subtraction, multiplication, and integer division are (with small modifications) implemented according to the classical algorithms [Knu81, Algorithms 4.3.1A, 4.3.1S, 4.3.1M, and 4.3.1D, respectively]. Alternatives for e.g., multiplication have been looked at, but turn out to be slower in software for our `word` lengths of 16 or 32 bits. For squaring we use the fact that most partial products occur twice.

Modular reduction has been implemented according to three different algorithms: the classical algorithm in the formalization by Knuth [Knu81, Algorithm 4.3.1D], Barrett's algorithm [Bar86], and Montgomery's algorithm [Mon85]. The three algorithms are quite close to each other in performance [Bos93]. The classical algorithm is the best choice for single modular reductions. Modular exponentiation based on Barrett's algorithm is superior to the others for small arguments. For general modular exponentiations the exponentiation based on Montgomery's algorithm has the best performance.

The calculation of the greatest common divisor and the modular inverse are implemented with a version of Euclid's gcd algorithm, optimized for multiprecision integers. It is based on the observation by Lehmer [Leh38] that almost all divisions in Euclid's algorithm have a small quotient. For more details, see [Knu81, Algorithm 4.5.2L].

6.3 High level arithmetic

The Chinese Remainder Theorem is implemented for the case of two equations with relatively prime moduli, which is the case we are mostly interested in to speed up the RSA secret key operation [QC82].

The calculation of $a^e \bmod m$ in our library uses a generalization of the standard binary square and multiply method, in which a table of small powers of a is used. For $p = 16$ the average number of modular multiplications is reduced to about 1/5 the number of bits in e (compared to 1/2 for binary square and multiply). The number of squarings in both methods is the same and equal to the number of bits in e. Each of the three reduction algorithms can be used, resulting in three modular exponentiation functions. The speed differences between the reduction functions will be reflected in speed differences between the exponentiation functions [Bos93]. For a full length exponentiation the Montgomery based exponentiation is superior to the other two. Cascade exponentiations of the form $a^{e_1}b^{e_2} \bmod m$ and $a^{e_1}b^{e_2}c^{e_3} \bmod m$ have been implemented according to the method described in [YL92], which is an extension of Shamir's method [ElG85].

Further functionality includes the calculation of square roots modulo a prime [Per86], the evaluation of the Jacobi symbol (see e.g., [Sim92, Appendix J of Chapter 4]), and the generation of pseudo-random multiprecision integers in a given interval, based on an extension of the Blum, Blum, and Shub generator [BBS86].

Finally, the library provides a set of functions for the generation of primes and so-called strong primes, and for the generation of RSA key sets (for both plain RSA and the Rabin variant [Wil80]). The methods used are based on the work of Brandt, Damgård, and Landrock [BD92].

7 Non-portable extensions

For even greater performance very fast 32-bit 80x86-assembly code has been developed for all time consuming operations in public key algorithms (multiplication, squaring, modular reduction, and exponentiation). These functions have the same interface as their ANSI C counterparts, so that they can easily replace them on 32-bit 80x86 based computers. They primarily derive their speed from the fact that in assembly language the `word` length can be doubled to 32 bits without any overhead.

8 Performance

An idea of the library's performance is given in Table 2.

Bit length	256		512		768		1024	
Code	C	80x86	C	80x86	C	80x86	C	80x86
$a^e \bmod m$	0.062	0.015	0.43	0.11	1.41	0.30	3.27	0.76
$a^{e_1}b^{e_2} \bmod m$	0.077	–	0.54	–	1.75	–	4.06	–
$a^{e_1}b^{e_2}c^{e_3} \bmod m$	0.090	–	0.60	–	1.89	–	4.34	–
RSA Key generation	–	–	3.6	2.2	11.4	5.9	28	9.3

Table 2: Execution times in seconds on a 60 MHz Pentium based PC for three types of modular exponentiation and for RSA key generation both without and with the 32-bit 80x86 extensions of our ANSI C library.

References

[Bar86] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," *Advances in Cryptology, Proc. Crypto'86, LNCS 263*, A.M. Odlyzko, Ed., Springer-Verlag, 1987, pp. 311–323.

[BBS86] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator," *SIAM J. Comput.*, Vol. 15, No. 2, 1986, pp. 364-383.

[Bos93] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," *Advances in Cryptology, Proc. Crypto'93, LNCS 773*, D.R. Stinson, Ed., Springer-Verlag, 1994, pp. 175–186.

[BD92] J. Brandt and I.B. Damgård, "On generation of probable primes by incremental search," *Advances in Cryptology, Proc. Crypto'92, LNCS 740*, E.F. Brickell, Ed., Springer-Verlag, 1993, pp. 358–369.

[ElG85] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Trans. on Information Theory*, Vol. IT–31, No. 4, 1985, pp. 469–472.

[ISO9899] ISO/IEC Standard 9899, *"Programming Languages - C,"* International Standards Organization, Geneva, 1990.

[Knu81] D.E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, Reading, Mass., 1981.

[Leh38] D.H. Lehmer, "Euclid's Algorithm for Large Numbers," *American Mathematical Monthly*, Vol. 45, 1938, pp. 227–233.

[Mon85] P.L. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, Vol. 44, 1985, pp. 519–521.

[Per86] R.C. Peralta, "A Simple and Fast Probabilistic Algorithm for Computing Square Roots Modulo a Prime Number," *IEEE Trans. on Information Theory*, Vol. IT–32, No. 6, 1986, pp. 846–847.

[QC82] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystems", *Electronic Letters*, Vol. 18, 1982, pp. 905–907.

[Ses89] R. Sessions, *Reusable Data Structures for C*, Prentice Hall, Englewood Cliffs, N.J. , 1989.

[Sim92] G.J. Simmons, Ed., *Contemporary Cryptology: The Science of Information Integrity*, IEEE Press, Piscataway, N.J., 1992.

[YL92] S.-M. Yen and C.-S. Laih, "The Fast Cascade Exponentiation Algorithm and its Applications on Cryptography," *Advances in Cryptology, Proc. Auscrypt'92, LNCS 718*, J. Seberry and Y. Zheng, Eds., Springer-Verlag, 1993.

[Wil80] H.C. Williams, "A Modification of the RSA Public-Key Encryption Procedure," *IEEE Trans. on Information Theory*, Vol. IT–26, No. 6, 1980, pp. 726–729.