LibTomMath v0.15
A Free Multiple Precision Integer Library
http://math.libtomcrypt.org

Tom St Denis
tomstdenis@iahu.ca

March 22, 2003

1 Introduction

"LibTomMath" is a free and open source library that provides multiple-precision integer functions required to form a basis of a public key cryptosystem. LibTomMath is written entire in portable ISO C source code and designed to have an application interface much like that of MPI from Michael Fromberger.

LibTomMath was written from scratch by Tom St Denis but designed to be drop in replacement for the MPI package. The algorithms within the library are derived from descriptions as provided in the Handbook of Applied Cryptography and Knuth's "The Art of Computer Programming". The library has been extensively optimized and should provide quite comparable timings as compared to many free and commercial libraries.

LibTomMath was designed with the following goals in mind:

1. Be a drop in replacement for MPI.

2. Be much faster than MPI.

3. Be written entirely in portable C.

All three goals have been achieved to one extent or another (actual figures depend on what platform you are using).

Being compatible with MPI means that applications that already use it can be ported fairly quickly. Currently there are a few differences but there are many similarities. In fact the average MPI based application can be ported in under 15 minutes.

Thanks goes to Michael Fromberger for answering a couple questions and Colin Percival for having the patience and courtesy to help debug and suggest optimizations. They were both of great help!

2 Building Against LibTomMath

As of v0.12 LibTomMath is not a simple single source file project like MPI. LibTomMath retains the exact same API as MPI but is implemented differently. To build LibTomMath you will need a copy of GNU cc and GNU make. Both are free so if you don't have a copy don't whine to me about it.

To build the library type

```
make
```

This will build the library file libtommath.a. If you want to build the library and also install it (in /usr/bin and /usr/include) then type

```
make install
```

Now within your application include "tommath.h" and link against libtommath.a to get MPI-like functionality.

2.1 Microsoft Visual C++

A makefile is also provided for MSVC (*tested against MSVC 6.00 with SP5*) which allows the library to be used with that compiler as well. To build the library type

```
nmake -f makefile.msvc
```

Which will build "tommath.lib".

3 Programming with LibTomMath

3.1 The mp_int Structure

All multiple precision integers are stored in a structure called **mp_int**. A multiple precision integer is essentially an array of **mp_digit**. mp_digit is defined at the top of "tommath.h". The type can be changed to suit a particular platform.

For example, when **MP_8BIT** is defined a mp_digit is a unsigned char and holds seven bits. Similarly when **MP_16BIT** is defined a mp_digit is a unsigned short and holds 15 bits. By default a mp_digit is a unsigned long and holds 28 bits which is optimal for most 32 and 64 bit processors.

The choice of digit is particular to the platform at hand and what available multipliers are provided. For MP_8BIT either a $8 \times 8 \Rightarrow 16$ or $16 \times 16 \Rightarrow 16$ multiplier is optimal. When MP_16BIT is defined either a $16 \times 16 \Rightarrow 32$ or $32 \times 32 \Rightarrow 32$ multiplier is optimal. By default a $32 \times 32 \Rightarrow 64$ or $64 \times 64 \Rightarrow 64$ multiplier is optimal.

This gives the library some flexibility. For example, a i8051 has a $8 \times 8 \Rightarrow 16$ multiplier. The 16-bit x86 instruction set has a $16 \times 16 \Rightarrow 32$ multiplier. In practice this library is not particularly designed for small devices like an i8051 due to the size. It is possible to strip out functions which are not required to drop the code size. More realistically the library is well suited to 32 and 64-bit processors that have decent integer multipliers. The AMD Athlon XP and Intel Pentium 4 processors are examples of well suited processors.

Throughout the discussions there will be references to a **used** and **alloc** members of an integer. The used member refers to how many digits are actually used in the representation of the integer. The alloc member refers to how many digits have been allocated off the heap. There is also the β quantity which is equal to 2^W where W is the number of bits in a digit (default is 28).

3.2 Calling Functions

Most functions expect pointers to mp_int's as parameters. To save on memory usage it is possible to have source variables as destinations. The arguements are read left to right so to compute $x + y = z$ you would pass the arguments in the order x, y, z. For example:

```
mp_add(&x, &y, &x);            /* x = x + y */
mp_mul(&y, &x, &z);            /* z = y * x */
mp_div_2(&x, &y);             /* y = x / 2 */
```

3.3   Various Optimizations

Various routines come in several "flavours" which are optimized for particular cases of inputs. For instance the multiplicative inverse function "mp_invmod()" has a routine for odd and even moduli. Similarly the "mp_exptmod()" function has several variants depending on the modulus as well. Several lower level functions such as multiplication, squaring and reductions come in "comba" and "baseline" variants.

The design of LibTomMath is such that the end user does not have to concern themselves too much with these details. This is why the functions provided will determine *automatically* when an appropriate optimal function can be used. For example, when you call "mp_mul()" the routines will first determine if the Karatsuba multiplier should be used. If not it will determine if the "comba" method can be used and finally call the standard catch-all "baseline" method.

Throughout the rest of this manual several variants for various functions will be referenced to as the "comba", "baseline", etc... method. Keep in mind you call one function to use any of the optimal variants.

3.4   Return Values

All functions that return errors will return **MP_OKAY** if the function was succesful. It will return **MP_MEM** if it ran out of heap memory or **MP_VAL** if one of the arguements is out of range.

3.5   Basic Functionality

Before an mp_int can be used it must be initialized with

```
int mp_init(mp_int *a);
```

For example, consider the following.

```
#include "tommath.h"
int main(void)
{
   mp_int num;
   if (mp_init(&num) != MP_OKAY) {
      printf("Error initializing a mp_int.\n");
   }
   return 0;
}
```

A mp_int can be freed from memory with

```
void mp_clear(mp_int *a);
```

This will zero the memory and free the allocated data. There are a set of trivial functions to manipulate the value of an mp_int.

```
/* set to zero */
void mp_zero(mp_int *a);

/* set to a digit */
void mp_set(mp_int *a, mp_digit b);

/* set a 32-bit const */
int mp_set_int(mp_int *a, unsigned long b);

/* init to a given number of digits */
int mp_init_size(mp_int *a, int size);

/* copy, b = a */
int mp_copy(mp_int *a, mp_int *b);

/* inits and copies, a = b */
int mp_init_copy(mp_int *a, mp_int *b);
```

The **mp_zero** function will clear the contents of a mp_int and set it to positive. The **mp_set** function will zero the integer and set the first digit to a value specified. The **mp_set_int** function will zero the integer and set the first 32-bits to a given value. It is important to note that using mp_set can have unintended side effects when either the MP_8BIT or MP_16BIT defines are enabled. By default the library will accept the ranges of values MPI will (and more).

The **mp_init_size** function will initialize the integer and set the allocated size to a given value. The allocated digits are zero'ed by default but not marked as used. The **mp_copy** function will copy the digits (and sign) of the first parameter into the integer specified by the second parameter. The **mp_init_copy** will initialize the first integer specified and copy the second one into it. Note that the order is reversed from that of mp_copy. This odd "bug" was kept to maintain compatibility with MPI.

3.6 Digit Manipulations

There are a class of functions that provide simple digit manipulations such as shifting and modulo reduction of powers of two.

```
/* right shift by "b" digits */
void mp_rshd(mp_int *a, int b);

/* left shift by "b" digits */
```

```
int mp_lshd(mp_int *a, int b);

/* c = a / 2^b */
int mp_div_2d(mp_int *a, int b, mp_int *c);

/* b = a/2 */
int mp_div_2(mp_int *a, mp_int *b);

/* c = a * 2^b */
int mp_mul_2d(mp_int *a, int b, mp_int *c);

/* b = a*2 */
int mp_mul_2(mp_int *a, mp_int *b);

/* c = a mod 2^d */
int mp_mod_2d(mp_int *a, int b, mp_int *c);

/* computes a = 2^b */
int mp_2expt(mp_int *a, int b);

/* makes a pseudo-random int of a given size */
int mp_rand(mp_int *a, int digits);
```

3.7   Binary Operations

```
/* c = a XOR b  */
int mp_xor(mp_int *a, mp_int *b, mp_int *c);

/* c = a OR b */
int mp_or(mp_int *a, mp_int *b, mp_int *c);

/* c = a AND b */
int mp_and(mp_int *a, mp_int *b, mp_int *c);
```

3.8   Basic Arithmetic

Next are the class of functions which provide basic arithmetic.

```
/* b = -a */
int mp_neg(mp_int *a, mp_int *b);

/* b = |a| */
int mp_abs(mp_int *a, mp_int *b);
```

```
/* compare a to b */
int mp_cmp(mp_int *a, mp_int *b);

/* compare |a| to |b| */
int mp_cmp_mag(mp_int *a, mp_int *b);

/* c = a + b */
int mp_add(mp_int *a, mp_int *b, mp_int *c);

/* c = a - b */
int mp_sub(mp_int *a, mp_int *b, mp_int *c);

/* c = a * b */
int mp_mul(mp_int *a, mp_int *b, mp_int *c);

/* b = a^2 */
int mp_sqr(mp_int *a, mp_int *b);

/* a/b => cb + d == a */
int mp_div(mp_int *a, mp_int *b, mp_int *c, mp_int *d);

/* c = a mod b, 0 <= c < b  */
int mp_mod(mp_int *a, mp_int *b, mp_int *c);
```

3.9   Single Digit Functions

```
/* compare against a single digit */
int mp_cmp_d(mp_int *a, mp_digit b);

/* c = a + b */
int mp_add_d(mp_int *a, mp_digit b, mp_int *c);

/* c = a - b */
int mp_sub_d(mp_int *a, mp_digit b, mp_int *c);

/* c = a * b */
int mp_mul_d(mp_int *a, mp_digit b, mp_int *c);

/* a/b => cb + d == a */
int mp_div_d(mp_int *a, mp_digit b, mp_int *c, mp_digit *d);

/* c = a mod b, 0 <= c < b  */
int mp_mod_d(mp_int *a, mp_digit b, mp_digit *c);
```

Note that care should be taken for the value of the digit passed. By default, any 28-bit integer is a valid digit that can be passed into the function. However, if MP_8BIT or MP_16BIT is defined only 7 or 15-bit (respectively) integers can be passed into it.

3.10   Modular Arithmetic

There are some trivial modular arithmetic functions.

```
/* d = a + b (mod c) */
int mp_addmod(mp_int *a, mp_int *b, mp_int *c, mp_int *d);

/* d = a - b (mod c) */
int mp_submod(mp_int *a, mp_int *b, mp_int *c, mp_int *d);

/* d = a * b (mod c) */
int mp_mulmod(mp_int *a, mp_int *b, mp_int *c, mp_int *d);

/* c = a * a (mod b) */
int mp_sqrmod(mp_int *a, mp_int *b, mp_int *c);

/* c = 1/a (mod b) */
int mp_invmod(mp_int *a, mp_int *b, mp_int *c);

/* c = (a, b) */
int mp_gcd(mp_int *a, mp_int *b, mp_int *c);

/* c = [a, b] or (a*b)/(a, b) */
int mp_lcm(mp_int *a, mp_int *b, mp_int *c);

/* find the b'th root of a  */
int mp_n_root(mp_int *a, mp_digit b, mp_int *c);

/* computes the jacobi c = (a | n) (or Legendre if b is prime)  */
int mp_jacobi(mp_int *a, mp_int *n, int *c);

/* used to setup the Barrett reduction for a given modulus b */
int mp_reduce_setup(mp_int *a, mp_int *b);

/* Barrett Reduction, computes a (mod b) with a precomputed value c
 *
 * Assumes that 0 < a <= b^2, note if 0 > a > -(b^2) then you can merely
 * compute the reduction as -1 * mp_reduce(mp_abs(a)) [pseudo code].
 */
int mp_reduce(mp_int *a, mp_int *b, mp_int *c);
```

```
/* setups the montgomery reduction */
int mp_montgomery_setup(mp_int *a, mp_digit *mp);

/* computes xR^-1 == x (mod N) via Montgomery Reduction */
int mp_montgomery_reduce(mp_int *a, mp_int *m, mp_digit mp);

/* returns 1 if a is a valid DR modulus */
int mp_dr_is_modulus(mp_int *a);

/* sets the value of "d" required for mp_dr_reduce */
void mp_dr_setup(mp_int *a, mp_digit *d);

/* reduces a modulo b using the Diminished Radix method */
int mp_dr_reduce(mp_int *a, mp_int *b, mp_digit mp);

/* d = a^b (mod c) */
int mp_exptmod(mp_int *a, mp_int *b, mp_int *c, mp_int *d);
```

3.11   Primality Routines

```
/* ---> Primes <--- */
/* table of first 256 primes */
extern const mp_digit __prime_tab[];

/* result=1 if a is divisible by one of the first 256 primes */
int mp_prime_is_divisible(mp_int *a, int *result);

/* performs one Fermat test of "a" using base "b".
 * Sets result to 0 if composite or 1 if probable prime
 */
int mp_prime_fermat(mp_int *a, mp_int *b, int *result);

/* performs one Miller-Rabin test of "a" using base "b".
 * Sets result to 0 if composite or 1 if probable prime
 */
int mp_prime_miller_rabin(mp_int *a, mp_int *b, int *result);

/* performs t rounds of Miller-Rabin on "a" using the first
 * t prime bases.  Also performs an initial sieve of trial
 * division.  Determines if "a" is prime with probability
 * of error no more than (1/4)^t.
 *
 * Sets result to 1 if probably prime, 0 otherwise
 */
int mp_prime_is_prime(mp_int *a, int t, int *result);
```

```
/* finds the next prime after the number "a" using "t" trials
 * of Miller-Rabin.
 */
int mp_prime_next_prime(mp_int *a, int t);
```

3.12 Radix Conversions

To read or store integers in other formats there are the following functions.

```
int mp_unsigned_bin_size(mp_int *a);
int mp_read_unsigned_bin(mp_int *a, unsigned char *b, int c);
int mp_to_unsigned_bin(mp_int *a, unsigned char *b);

int mp_signed_bin_size(mp_int *a);
int mp_read_signed_bin(mp_int *a, unsigned char *b, int c);
int mp_to_signed_bin(mp_int *a, unsigned char *b);

int mp_read_radix(mp_int *a, unsigned char *str, int radix);
int mp_toradix(mp_int *a, unsigned char *str, int radix);
int mp_radix_size(mp_int *a, int radix);
```

The integers are stored in big endian format as most libraries (and MPI) expect. The **mp_read_radix** and **mp_toradix** functions read and write (respectively) null terminated ASCII strings in a given radix. Valid values for the radix are between 2 and 64 (inclusively).

4 Function Analysis

Throughout the function analysis the variable N will denote the average size of an input to a function as measured by the number of digits it has. The variable W will denote the number of bits per word and c will denote a small constant amount of work. The big-oh notation will be abused slightly to consider numbers that do not grow to infinity. That is we shall consider $O(N/2) \neq O(N)$ which is an abuse of the notation.

4.1 Digit Manipulation Functions

The class of digit manipulation functions such as **mp_rshd**, **mp_lshd** and **mp_mul_2** are all very simple functions to analyze.

4.1.1 mp_rshd(mp_int *a, int b)

Shifts a by given number of digits to the right and is equivalent to dividing by β^b. The work is performed in-place which means the input and output are the same. If the shift count b is less than or equal to zero the function returns

without doing any work. If the the shift count is larger than the number of digits in a then a is simply zeroed without shifting digits.

This function requires no additional memory and $O(N)$ time.

4.1.2 mp_lshd(mp_int *a, int b)

Shifts a by a given number of digits to the left and is equivalent to multiplying by β^b. The work is performed in-place which means the input and output are the same. If the shift count b is less than or equal to zero the function returns success without doing any work.

This function requires $O(b)$ additional digits of memory and $O(N)$ time.

4.1.3 mp_div_2d(mp_int *a, int b, mp_int *c, mp_int *d)

Shifts a by a given number of **bits** to the right and is equivalent to dividing by 2^b. The shifted number is stored in the c parameter. The remainder of $a/2^b$ is optionally stored in d (if it is not passed as NULL). If the shift count b is less than or equal to zero the function places a in c and returns success.

This function requires $O(2 \cdot N)$ additional digits of memory and $O(2 \cdot N)$ time.

4.1.4 mp_mul_2d(mp_int *a, int b, mp_int *c)

Shifts a by a given number of bits to the left and is equivalent to multiplying by 2^b. The shifted number is placed in the c parameter. If the shift count b is less than or equal to zero the function places a in c and returns success.

This function requires $O(N)$ additional digits of memory and $O(2 \cdot N)$ time.

4.1.5 mp_mul_2(mp_int *a, mp_int *b)

Multiplies a by two and stores in b. This function is hard coded todo a shift by one place so it is faster than calling mp_mul_2d with a count of one.

This function requires $O(N)$ additional digits of memory and $O(N)$ time.

4.1.6 mp_div_2(mp_int *a, mp_int *b)

Divides a by two and stores in b. This function is hard coded todo a shift by one place so it is faster than calling mp_div_2d with a count of one.

This function requires $O(N)$ additional digits of memory and $O(N)$ time.

4.1.7 mp_mod_2d(mp_int *a, int b, mp_int *c)

Performs the action of reducing a modulo 2^b and stores the result in c. If the shift count b is less than or equal to zero the function places a in c and returns success.

This function requires $O(N)$ additional digits of memory and $O(2 \cdot N)$ time.

4.1.8 mp_2expt(mp_int *a, int b)

Computes $a = 2^b$ by first setting a to zero then OR'ing the correct bit to get the right value.

4.1.9 mp_rand(mp_int *a, int digits)

Computes a pseudo-random (*via rand()*) integer that is always "*digits*" digits in length. Not for cryptographic use.

4.2 Binary Arithmetic

4.2.1 mp_xor(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a \oplus b$, pseudo-extends with zeroes whichever of a or b is shorter such that the length of c is the maximum length of the two inputs.

4.2.2 mp_or(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a \lor b$, pseudo-extends with zeroes whichever of a or b is shorter such that the length of c is the maximum length of the two inputs.

4.2.3 mp_and(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a \land b$, pseudo-extends with zeroes whichever of a or b is shorter such that the length of c is the maximum length of the two inputs.

4.3 Basic Arithmetic

4.3.1 mp_cmp(mp_int *a, mp_int *b)

Performs a **signed** comparison between a and b returning **MP_GT** if a is larger than b.

This function requires no additional memory and $O(N)$ time.

4.3.2 mp_cmp_mag(mp_int *a, mp_int *b)

Performs a **unsigned** comparison between a and b returning **MP_GT** is a is larger than b. Note that this comparison is unsigned which means it will report, for example, $-5 > 3$. By comparison mp_cmp will report $-5 < 3$.

This function requires no additional memory and $O(N)$ time.

4.3.3 mp_add(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a + b$ using signed arithmetic. Handles the sign of the numbers which means it will subtract as required, e.g. $a + -b$ turns into $a - b$.

This function requires no additional memory and $O(N)$ time.

4.3.4   mp_sub(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a - b$ using signed arithmetic. Handles the sign of the numbers which means it will add as required, e.g. $a - -b$ turns into $a + b$.

   This function requires no additional memory and $O(N)$ time.

4.3.5   mp_mul(mp_int *a, mp_int *b, mp_int *c)

Computes $c = a \cdot b$ using signed arithmetic. Handles the sign of the numbers correctly which means it will correct the sign of the product as required, e.g. $a \cdot -b$ turns into $-ab$.

   This function requires $O(N^2)$ time for small inputs and $O(N^{1.584})$ time for relatively large inputs (*above the* KARATSUBA_MUL_CUTOFF *value defined in bncore.c.*). There is considerable overhead in the Karatsuba method which only pays off when the digit count is fairly high (*typically around 80*). For small inputs the function requires $O(2N)$ memory, otherwise it requires $O(6 \cdot \lg(N) \cdot N)$ memory.

4.3.6   mp_sqr(mp_int *a, mp_int *b)

Computes $b = a^2$ and fixes the sign of b to be positive.

   This function has a running time and memory requirement profile very similar to that of the mp_mul function. It is always faster and uses less memory for the larger inputs.

4.3.7   mp_div(mp_int *a, mp_int *b, mp_int *c, mp_int *d)

Computes $c = \lfloor a/b \rfloor$ and $d \equiv a \pmod{b}$. The division is signed which means the sign of the output is not always positive. The sign of the remainder equals the sign of a while the sign of the quotient equals the product of the ratios $(a/|a|) \cdot (b/|b|)$. Both c and d can be optionally passed as NULL if the value is not desired. For example, if you want only the quotient of x/y then mp_div(&x, &y, &z, NULL) is acceptable.

   This function requires $O(4 \cdot N)$ memory and $O(3 \cdot N^2)$ time.

4.3.8   mp_mod(mp_int *a, mp_int *b, mp_int *c)

Computes $c \equiv a \pmod{b}$ but with the added condition that $0 \leq c < b$. That is a normal division is performed and if the remainder is negative b is added to it. Since adding b modulo b is equivalent to adding zero ($0 \equiv b \pmod{b}$) the result is accurate. The results are undefined when $b \leq 0$, in theory the routine will still give a properly congruent answer but it will not always be positive.

   This function requires $O(4 \cdot N)$ memory and $O(3 \cdot N^2)$ time.

4.4 Number Theoretic Functions

4.4.1 mp_addmod, mp_submod, mp_mulmod, mp_sqrmod

These functions take the time of their host function plus the time it takes to perform a division. For example, mp_addmod takes $O(N + 3 \cdot N^2)$ time. Note that if you are performing many modular operations in a row with the same modulus you should consider Barrett reductions.

Also note that these functions use mp_mod which means the result are guaranteed to be positive.

4.4.2 mp_invmod(mp_int *a, mp_int *b, mp_int *c)

This function will find $c = 1/a \pmod{b}$ for any value of a such that $(a, b) = 1$ and $b > 0$. When b is odd a "fast" variant is used which finds the inverse twice as fast. If no inverse is found (e.g. $(a, b) \neq 1$) then the function returns **MP_VAL** and the result in c is undefined.

4.4.3 mp_gcd(mp_int *a, mp_int *b, mp_int *c)

Finds the greatest common divisor of both a and b and places the result in c. Will work with either positive or negative inputs.

Functions requires no additional memory and approximately $O(N \cdot log(N))$ time.

4.4.4 mp_lcm(mp_int *a, mp_int *b, mp_int *c)

Finds the least common multiple of both a and b and places the result in c. Will work with either positive or negative inputs. This is calculated by dividing the product of a and b by the greatest common divisor of both.

Functions requires no additional memory and approximately $O(4 \cdot N^2)$ time.

4.4.5 mp_n_root(mp_int *a, mp_digit b, mp_int *c)

Finds the b'th root of a and stores it in b. The roots are found such that $|c|^b \leq |a|$. Uses the Newton approximation approach which means it converges in $O(log\beta^N)$ time to a final result. Each iteration requires b multiplications and one division for a total work of $O(6N^2 \cdot log\beta^N) = O(6N^3 \cdot log\beta)$.

If the input a is negative and b is even the function returns **MP_VAL**. Otherwise the function will return a root that has a sign that agrees with the sign of a.

4.4.6 mp_jacobi(mp_int *a, mp_int *n, int *c)

Computes $c = \left(\frac{a}{n}\right)$ or the Jacobi function of (a, n) and stores the result in an integer addressed by c. Since the result of the Jacobi function $\left(\frac{a}{n}\right) \in \{-1, 0, 1\}$ it seemed natural to store the result in a simple C style **int**. If n is prime then

the Jacobi function produces the same results as the Legendre function[1]. This means if n is prime then $\left(\frac{a}{n}\right)$ is equal to 1 if a is a quadratic residue modulo n or -1 if it is not.

4.4.7   mp_exptmod(mp_int *a, mp_int *b, mp_int *c, mp_int *d)

Computes $d \equiv a^b \pmod{c}$ using a sliding window k-ary exponentiation algorithm. For an α-bit exponent it performs α squarings and at most $\lfloor \alpha/k \rfloor + 2^{k-1}$ multiplications. The value of k is chosen to minimize the number of multiplications required for a given value of α. Barrett, Montgomery or Dimminished-Radix reductions are used to reduce the squared or multiplied temporary results modulo c.

4.5   Fast Modular Reductions

A modular reduction of $a \pmod{b}$ means to divide a by b and obtain the remainder. Typically modular reductions are popular in public key cryptography algorithms such as RSA, Diffie-Hellman and Elliptic Curve. Modular reductions are also a large portion of modular exponentiation (e.g. $a^b \pmod{c}$).

In a simplistic sense a normal integer division could be used to compute reduction. Division is by far the most complicated of routines in terms of the work required. As a result it is desirable to avoid division as much as possible. This is evident in quite a few fields in computing. For example, often in signal analysis uses multiplication by the reciprocal to approximate divisions. Number theory is no different.

In most cases for the reduction of a modulo b the integer a will be limited to the range $0 \leq a \leq b^2$ which led to the invention of specialized algorithms to do the work.

The first algorithm is the most generic and is called the Barrett reduction. When the input is of the limited form (e.g. $0 \leq a \leq b^2$) Barrett reduction is numerically equivalent to a full integer division with remainder. For a n-digit value b the Barrett reduction requires approximately $2n^2$ multiplications.

The second algorithm is the Montgomery reduction. It is slightly different since the result is not numerically equivalent to a standard integer division with remainder. Also this algorithm only works for odd moduli. The final result can be converted easily back to the desired for which makes the reduction technique useful for algorithms where only the final output is desired. For a n-digit value b the Montgomery reduction requires approximately $n^2 + n$ multiplications, about half as many as the Barrett algorithm.

The third algorithm is the Diminished Radix "DR" reduction. It is a highly optimized reduction algorithm suitable only for a limited set of moduli. For the specific moduli it is numerically equivalent to integer division with remainder. For a n-digit value b the DR reduction rquires exactly n multiplications which is considerably faster than either of the two previous algorithms.

---

[1]Source: Handbook of Applied Cryptography, pp. 73

All three algorithms are automatically used in the modular exponentiation function mp_exptmod() when appropriate moduli are detected.

Algorithm	Multiplications	Limitations
Barrett Reduction	$2n^2$	Any modulus.
Montgomery Reduction	$n^2 + n$	Any odd modulus.
DR Reduction	n	Moduli of the form $p = \beta^k - p'$.

Figure 1: Summary of reduction techniques.

4.5.1 mp_reduce(mp_int *a, mp_int *b, mp_int *c)

Computes a Barrett reduction in-place of a modulo b with respect to c. In essence it computes a (mod b) provided $0 \leq a \leq b^2$. The value of c is precomputed with the function mp_reduce_setup(). The modulus b must be larger than zero.

This reduction function is much faster than simply calling mp_mod() (*Which simply uses mp_div() anyways*) and is desirable where ever an appropriate reduction is desired.

The Barrett reduction function has been optimized to use partial multipliers which means compared to MPI it performs have the number of single precision multipliers (*provided they have the same size digits*). The partial multipliers (*one of which is shared with mp_mul*) both have baseline and comba variants. Barrett reduction can reduce a number modulo a $n-$digit modulus with approximately $2n^2$ single precision multiplications.

Consider the following snippet (from a BBS generator) using the more traditional approach:

```
mp_int modulus, n;
unsigned char buf[128];
int ix, err;

/* ... init code ..., e.g. init modulus and n */
/* now output 128 bytes */
for (ix = 0; ix < 128; ix++) {
    if ((err = mp_sqrmod(&n, &modulus, &n)) != MP_OKAY) {
        printf("Err: %d\n", err);
        exit(EXIT_FAILURE);
    }
    buf[ix] = n->dp[0] & 255;
}
```

And now consider the same function using Barrett reductions:

```
mp_int modulus, n, mp;
unsigned char buf[128];
int ix, err;
```

```
/* ... init code ... e.g. modulus and n */

/* now setup mp which is the Barrett param */
if ((err = mp_reduce_setup(&mp, &modulus)) != MP_OKAY) {
   printf("Err: %d\n", err);
   exit(EXIT_FAILURE);
}
/* now output 128 bytes */
for (ix = 0; ix < 128; ix++) {
   /* square n */
   if ((err = mp_sqr(&n, &n)) != MP_OKAY) {
     printf("Err: %d\n", err);
     exit(EXIT_FAILURE);
   }
   /* now reduce the square modulo modulus */
   if ((err = mp_reduce(&n, &modulus, &mp)) != MP_OKAY) {
     printf("Err: %d\n", err);
     exit(EXIT_FAILURE);
   }
   buf[ix] = n->dp[0] & 255;
}
```

Both routines will produce the same output provided the same initial values of *modulus* and n. The Barrett method seems like more work but the optimization stems from the use of the Barrett reduction instead of the normal integer division.

4.5.2   mp_montgomery_reduce(mp_int *a, mp_int *m, mp_digit mp)

Computes a Montgomery reduction in-place of a modulo b with respect to mp. If b is some $n-$digit modulus then $R = \beta^{n+1}$. The result of this function is $aR^{-1} \pmod{b}$ provided that $0 \le a \le b^2$. The value of mp is precomputed with the function mp_montgomery_setup(). The modulus b must be odd and larger than zero.

The Montgomery reduction comes in two variants. A standard baseline and a fast comba method. The baseline routine is in fact slower than the Barrett reductions, however, the comba routine is much faster. Montomgery reduction can reduce a number modulo a $n-$digit modulus with approximately $n^2 + n$ single precision multiplications. Compared to Barrett reductions the montgomery reduction requires half as many multiplications as $n \to \infty$.

Note that the final result of a Montgomery reduction is not just the value reduced modulo b. You have to multiply by R modulo b to get the real result. At first that may not seem like such a worthwhile routine, however, the exptmod function can be made to take advantage of this such that only one normalization at the end is required.

This stems from the fact that if $a \to aR^{-1}$ through Montgomery reduction and if $a = vR$ and $b = uR$ then $a^2 \to v^2R^2R^{-1} \equiv v^2R$ and $ab \to uvRRR^{-1} \equiv$

uvR. The next useful observation is that through the reduction $a \rightarrow vRR^{-1} \equiv v$ which means given a final result it can be normalized with a single reduction. Now a series of complicated modular operations can be optimized if all the variables are initially multiplied by R then the final result normalized by performing an extra reduction.

If many variables are to be normalized the simplest method to setup the variables is to first compute $\hat{x} \equiv R^2 \bmod m$. Now all the variables in the system can be multiplied by $\hat{x}$ and reduced with Montgomery reduction. This means that two long divisions would be required to setup $\hat{x}$ and a multiplication followed by reduction for each variable.

A very useful observation is that multiplying by $R = \beta^n$ amounts to performing a left shift by n positions which requires no single precision multiplications.

4.5.3 mp_dr_reduce(mp_int *a, mp_int *b, mp_digit mp)

Computes the Diminished-Radix reduction of a in place modulo b with respect to mp. a must be in the range $0 \leq a \leq b^2$ and mp must be precomputed with the function mp_dr_setup().

This reduction technique performs the reduction with n multiplications and is much faster than either of the previous reduction methods. Essentially it is very much like the Montgomery reduction except it is particularly optimized for specific types of moduli. The moduli must be of the form $p = \beta^k - p'$ where $0 \leq p' < \beta$ for $k \geq 2$. This algorithm is suitable for several applications such as Diffie-Hellman public key cryptsystems where the prime p is of this form.

In appendix A several "safe" primes of various sizes are provided. These primes are DR moduli and of the form $p = 2q + 1$ where both p and q are prime. A trivial observation is that $g = 4$ will be a generator for all of them since the order of the multiplicative sub-group is at most $2q$. Since $2^2 \neq 1$ that means $4^q \equiv 2^{2q} \equiv 1$ and that $g = 4$ is a generator of order q.

These moduli can be used to construct a Diffie-Hellman public key cryptosystem. Since the moduli are of the DR form the modular exponentiation steps will be efficient.

4.6 Primality Testing and Generation

4.6.1 mp_prime_is_divisible(mp_int *a, int *result)

Determines if a is divisible by any of the first 256 primes. Sets *result* to 1 if true or 0 otherwise. Also will set *result* to 1 if a is equal to one of the first 256 primes.

4.6.2 mp_prime_fermat(mp_int *a, mp_int *b, int *result)

Determines if b is a witness to the compositeness of a using the Fermat test. Essentially this computes $b^a \pmod{a}$ and compares it to b. If they match *result* is set to 1 otherwise it is set to 0. If a is prime and $1 < b < a$ then this function

will set *result* to 1 with a probability of one. If a is composite then this function will set *result* to 1 with a probability of no more than $\frac{1}{2}$.

If this function is repeated t times with different bases b then the probability of a false positive is no more than 2^{-t}.

4.6.3 mp_prime_miller_rabin(mp_int *a, mp_int *b, int *result)

Determines if b is a witness to the compositeness of a using the Miller-Rabin test. This test works much (*on an abstract level*) the same as the Fermat test except is more robust. The set of pseudo-primes to any given base for the Miller-Rabin test is a proper subset of the pseudo-primes for the Fermat test.

If a is prime and $1 < b < a$ then this function will always set *result* to 1. If a is composite the trivial bound of error is $\frac{1}{4}$. However, according to HAC[2] the following bounds are known. For a test of t trials on a k-bit number the probability $P_{k,t}$ of error is given as follows.

1. $P_{k,1} < k^2 4^{2-\sqrt{k}}$ for $k \geq 2$

2. $P_{k,t} < k^{3/2} 2^t t^{-1/2} 4^{2-\sqrt{tk}}$ for $(t = 2, k \geq 88)$ or $(3 \leq t \leq k/9, k \geq 21)$.

3. $P_{k,t} < \frac{7}{20} k 2^{-5t} + \frac{1}{7} k^{15/4} 2^{-k/2-2t} + 12k 2^{-k/4-3t}$ for $k/9 \leq t \leq k/4, k \geq 21$.

4. $P_{k,t} < \frac{1}{7} k^{15/4} 2^{-k/2-2t}$ for $t \geq k/4, k \geq 21$.

For instance, $P_{1024,1}$ which indicates the probability of failure of one test with a 1024-bit candidate is no more than 2^{-40}. However, ideally at least a couple of trials should be used. In LibTomCrypt for instance eight tests are used. In this case $P_{1024,8}$ falls under the second rule which leads to a probability of failure of no more than $2^{-155.52}$.

Size (k)	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$
512	-58	-70	-79	-88	-96	-104
768	-75	-89	-101	-112	-122	-131
1024	-89	-106	-120	-133	-144	-155
1280	-102	-120	-136	-151	-164	-176
1536	-113	-133	-151	-167	-181	-195
1792	-124	-146	-165	-182	-198	-212
2048	-134	-157	-178	-196	-213	-228

Figure 2: Probability of error for a given random candidate of k bits with t trials. Denoted as $\log_2(P_{k,t})$.

---

[2]Handbook of Applied Cryptography, Chapter 4, Section 4, pp. 147, Fact 4.48.

4.6.4 mp_prime_is_prime(mp_int *a, int t, int *result)

This function determines if a is probably prime by first performing trial division by the first 256 primes and then t rounds of Miller-Rabin using the first t primes as bases. If a is prime this function will always set *result* to 1. If a is composite then it will almost always set *result* to 0. The probability of error is given in figure two.

4.6.5 mp_prime_next_prime(mp_int *a, int t)

This function will find the next prime **after** a by using trial division and t trials of Miller-Rabin.

5 Timing Analysis

5.1 Digit Size

The first major constribution to the time savings is the fact that 28 bits are stored per digit instead of the MPI defualt of 16. This means in many of the algorithms the savings can be considerable. Consider a baseline multiplier with a 1024-bit input. With MPI the input would be 64 16-bit digits whereas in LibTomMath it would be 37 28-bit digits. A savings of $64^2 - 37^2 = 2727$ single precision multiplications.

5.2 Multiplication Algorithms

For most inputs a typical baseline $O(n^2)$ multiplier is used which is similar to that of MPI. There are two variants of the baseline multiplier. The normal and the fast comba variant. The normal baseline multiplier is the exact same as the algorithm from MPI. The fast comba baseline multiplier is optimized for cases where the number of input digits N is less than or equal to $2^w/\beta^2$. Where w is the number of bits in a **mp_word** or simply $lg(\beta)$. By default a mp_word is 64-bits which means $N \leq 256$ is allowed which represents numbers upto $7,168$ bits. However, since the Karatsuba multiplier (discussed below) will kick in before that size the slower baseline algorithm (that MPI uses) should never really be used in a default configuration.

The fast comba baseline multiplier is optimized by removing the carry operations from the inner loop. This is often referred to as the "comba" method since it computes the products a columns first then figures out the carries. To accomodate this the result of the inner multiplications must be stored in words large enough not to lose the carry bits. This is why there is a limit of $2^w/\beta^2$ digits in the input. This optimization has the effect of making a very simple and efficient inner loop.

5.2.1 Karatsuba Multiplier

For large inputs, typically 80 digits[3] or more the Karatsuba multiplication method is used. This method has significant overhead but an asymptotic running time of $O(n^{1.584})$ which means for fairly large inputs this method is faster than the baseline (or comba) algorithm. The Karatsuba implementation is recursive which means for extremely large inputs they will benefit from the algorithm.

The algorithm is based on the observation that if

$$x = x_0 + x_1\beta$$
$$y = y_0 + y_1\beta \tag{1}$$

Where x_0, x_1, y_0, y_1 are half the size of their respective summand than

$$x \cdot y = x_1 y_1 \beta^2 + ((x_1 - y_1)(x_0 - y_0) + x_0 y_0 + x_1 y_1)\beta + x_0 y_0 \tag{2}$$

It is trivial that from this only three products have to be produced: $x_0 y_0, x_1 y_1, (x_1 - y_1)(x_0 - y_0)$ which are all of half size numbers. A multiplication of two half size numbers requires only $\frac{1}{4}$ of the original work which means with no recursion the Karatsuba algorithm achieves a running time of $\frac{3n^2}{4}$. The routine provided does recursion which is where the $O(n^{1.584})$ work factor comes from.

The multiplication by β and β^2 amount to digit shift operations. The extra overhead in the Karatsuba method comes from extracting the half size numbers x_0, x_1, y_0, y_1 and performing the various smaller calculations.

The library has been fairly optimized to extract the digits using hard-coded routines instead of the hire level functions however there is still significant overhead to optimize away.

MPI only implements the slower baseline multiplier where carries are dealt with in the inner loop. As a result even at smaller numbers (below the Karatsuba cutoff) the LibTomMath multipliers are faster.

5.3 Squaring Algorithms

Similar to the multiplication algorithms there are two baseline squaring algorithms. Both have an asymptotic running time of $O((t^2 + t)/2)$. The normal baseline squaring is the same from MPI and the fast method is a "comba" squaring algorithm. The comba method is used if the number of digits N is less than $2^{w-1}/\beta^2$ which by default covers numbers upto $3,584$ bits.

There is also a Karatsuba squaring method which achieves a running time of $O(n^{1.584})$ after considerably large inputs.

MPI only implements the slower baseline squaring algorithm. As a result LibTomMath is considerably faster at squaring than MPI is.

---

[3]By default that is 2240-bits or more.

5.4 Exponentiation Algorithms

LibTomMath implements a sliding window k-ary left to right exponentiation algorithm. For a given exponent size L an appropriate window size k is chosen. There are always at most L modular squarings and $\lfloor L/k \rfloor$ modular multiplications. The k-ary method works by precomputing values $g(x) = b^x$ for $2^{k-1} \leq x < 2^k$ and a given base b. Then the multiplications are grouped in windows of k bits. The sliding window technique has the benefit that it can skip multiplications if there are zero bits following or preceding a window. Consider the exponent $e = 11110001_2$ if $k = 2$ then there will be a two squarings, a multiplication of $g(3)$, two squarings, a multiplication of $g(3)$, four squarings and and a multiplication by $g(1)$. In total there are 8 squarings and 3 multiplications.

MPI uses a binary square-multiply method for exponentiation. For the same exponent $e = 11110001_2$ it would have had to perform 8 squarings and 5 multiplications. There is a precomputation phase for the method LibTomMath uses but it generally cuts down considerably on the number of multiplications. Consider a 512-bit exponent. The worst case for the LibTomMath method results in 512 squarings and 124 multiplications. The MPI method would have 512 squarings and 512 multiplications.

Randomly the most probable event is that every $2k^2$ bits another multiplication is saved via the sliding-window technique on top of the savings the k-ary method provides. This stems from the fact that each window has a probability of 2^{-1} of being delayed by one bit. In reality the savings can be much more when the exponent has an abundance of zero bits.

Both LibTomMath and MPI use Barrett reduction instead of division to reduce the numbers modulo the modulus given. However, LibTomMath can take advantage of the fact that the multiplications required within the Barrett reduction do not have to give full precision. As a result the reduction step is much faster and just as accurate. The LibTomMath code will automatically determine at run-time (e.g. when its called) whether the faster multiplier can be used. The faster multipliers have also been optimized into the two variants (baseline and comba baseline).

LibTomMath also has a variant of the exptmod function that uses Montgomery or Diminished-Radix reductions instead of Barrett reductions which are faster. The code will automatically detect when the Montgomery version can be used (*Requires the modulus to be odd and below the MONTGOMERY_EXPT_CUTOFF size*). The Montgomery routine is essentially a copy of the Barrett exponentiation routine except it uses Montgomery reduction.

As a result of all these changes exponentiation in LibTomMath is much faster than compared to MPI. On most ALU-strong processors (AMD Athlon for instance) exponentiation in LibTomMath is often more then ten times faster than MPI.

Appendix A – DR Safe Prime Moduli

These are safe primes suitable for the DR reduction techniques.

```
224-bit prime:
p == 26959946667150639794667015087019630673637144422540572481103341844143

532-bit prime:
p == 14059105607947488696282932836518693308967803494693489478439861164411
       99243959839959474700214407465892859350284572975279726002583142341968
       6528151609940203368691747

784-bit prime:
p == 10174582569701926077392351975587856746131528201775982910760891436407
       52752352543956225804474009941755789631639189671820136396606697711084
       75957692810857098847138903161308502419410142185759152435680068435915
       15940249605851361141168890243039

1036-bit prime:
p == 73633510803960459580592340614718453088992337057476877219196961242207
       30400993319449915739231125812675425079864519532271929704028930638504
       85730703075899286013451337291468249027691733891486704001513279827771
       74018362916106519487472796251714810077522836342108369176406547759082
       39193640129179846056195261408217984837127

1540-bit prime:
p == 38564998830736521417281865696453025806593491967131023221754800625044
       11826546885121070536038571753679461518026049420807660579867166071933
       31995138078062523944232834134301060035963325132466829039948295286901
       98205120921557533726473585751382193953592127439965050261476810842071
       57368450587885458870662348457392592590350574754547108886771218500413
       52012892734056144158994382765356263460989042410208779740029161680999
       51885406379295536200413493190419727789712076165162175783

2072-bit prime:
p == 54218939133169617266167044061918053674999416641599333415160174539219
       34845902966009796023786766248081296137779934662422030250545736925626
       89251250471628358318743978285860720148446448885701001277560572526947
       61939255157449083928645845499448866574499182283776991809511712954641
       41244487770339412235658314203908468644295047744779491537946899487476
       80362212954278693335653935890352619041936727463717926744868338358149
       56836864340303776864961677852601361049369618605589931826833943267154
       13281957242613296066998310166663594408748431030206661065682224010477
       20269951530296879490444224546654729111504346660859907296364097126834
       834235287147
```

```
3080-bit prime:
p == 14872591348147092640920326485259710388958656451489011805853404549855
     24155135260217788758027400478312256339496385275012465661575576202252
     06314569873207988029466422057976484876770407676185319721656326266004
     66027039730507982182461708359620055985616697068444694474354610925422
     65792444947706769156952522561309012718703410057689129744336845214362
     11126335809752272646208391793909176002665892575707673348417320292714
     14414925737999142402226287954056239531091315945236233530448983394814
     94120112723445689647986475279424446083151413667587008191682564376412
     34796414611389856588668313940700594138366932599747507691048808666325
     63356891811579575714450674901879395531659037735542902605310091218790
     44170766615232300936675369451260747671432073394867530820527479172464
     10644245072764022650374658634027981631882139521072626829153564850619
     07146160831634031899433344310568760382865303657571873671474460048559
     12033137386225053275419626102417236133948503

4116-bit prime:
p == 10951211157166778028568112903923951285881685924091094949001780089679
     55253005183831872715423151551999734857184538199864469605657805519106
     71752965504405483319768745978263629725521974299473675154181526972794
     07518606702687749033402960400061140139713092570283328496790968248002
     50742691718610670812374272414086863715763724622797509437062518082383
     05605014462496277630214789052124947706021514827516368830127584715531
     60422794055576326393660668474428614221648326558746558242215778499288
     63023018366835675399949740429332468186340518172487073360822220449055
     34058256846156864525995487330361695377639385317484513208112197632746
     27403549307444874296172025850155107442985301015477068215901887335158
     80733527449780963163909830077616357506845523215289297624086914545378
     51108253422962011656326016849452390656670941816601111275452976618355
     45793212249409511773940884655967126200762400673705890369240247283750
     76210477267488679008016579588696191194060127319035195370137160936882
     40224439969917201783514453748848639690614421772002899286394128821718
     53539149915834004216827510006035966557909908155251261543943446413363
     97793791497068253936771017031980867706707490224041075826337383538651
     82549367950377193483609465580277633166426163174014828176348776585274
     6577808019633679
```

24