TomsFastMath User Manual
v0.07

Tom St Denis
tomstdenis@gmail.com

November 18, 2005

 This book has been formatted for B5 [176x250] paper using the LaTeX *book* macro package.

This project was sponsored in part by
Secure Science Corporation `http://www.securescience.net`.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is TomsFastMath?

TomsFastMath is meant to be a very fast yet still fairly portable and easy to port large integer arithmetic library written in ISO C. The goal specifically is to be able to perform very fast modular exponentiations and other related functions required for ECC, DH and RSA cryptosystems.

Most of the library is pure ISO C portable source code while a small portion (three files) contain a mixture of ISO C and assembler inline fragments. Compared to LibTomMath this new library is meant to be much faster while sacrificing flexibiltiy. This is accomplished through several means.

1. The new code is slightly messier and contains asm blocks.

2. This uses fixed not multiple precision integers.

3. It is designed only for fast modular exponentiations [e.g. less flexibility].

To mitigate some of the problems that arise from using assembler it has been carefully and appropriately used where it would make the most gain in performance. Also we use macro's for assembler code which allows new ports to be inserted easily.

The new code uses fixed precision arithmetic which means at compile time you choose a maximum precision and all numbers are limited to that. This has the benefit of not requiring any memory heap operations (which are slow) in

any of the functions. It has the downside that integers that are too large are truncated.

The goal of this library is to be able to perform modular exponentiations (with an odd modulus) very fast. This is what takes the most time in systems such as RSA and DH. This also requires fast multiplication and squaring and has the side effect of speeding up ECC operations as well.

1.2 License

TomsFastMath is public domain.

1.3 Building

To build the library simply type "make". Or to install in typical *unix like directories use "make install". Similarly a shared library can be built with "make -f makefile.shared install".

You can build the test program with "make test". To perform simple static testing (useful to test out new assembly ports) use the stest program. Type "make stest" and run it on your target. The program will perform three multiplications, squarings and montgomery reductions. Likely if your assembly code is invalid this code will exhibit the bug.

1.3.1 Intel CC

In theory you should be able to build the library with

```
CFLAGS="-O3 -ip" CC=icc make IGNORE_SPEED=1
```

However, Intels inline assembler is way less advanced than GCCs. As a result it doesn't compile. Fortunately it doesn't really matter.

1.3.2 MSVC

The library doesn't build with MSVC. Imagine that.

1.3.3 Build Limitations

TomsFastMath has the following build requirements which are non–portable but under most circumstances not problematic.

1. "CHAR_BIT" must be eight.
2. The "fp_digit" type must be a multiple of eight bits long.
3. The "fp_word" must be at least twice the length of fp_digit.

1.3.4 Optimization Configuration

By default TFM is configured for 32–bit digits using ISO C source code. This mode while portable is not very efficient. While building the library (from scratch) you can define one of several "CFLAGS" defines.

For example, to build with with SSE2 optimizations type

```
CFLAGS=-DTFM_SSE2 make clean libtfm.a
```

x86–32

The "x86–32" mode is defined by "TFM_X86" and covers all i386 and beyond processors. It requires GCC to build and only works with 32–bit digits. In this mode fp_digit is 32–bits and fp_word is 64–bits. This mode will be autodetected when building with GCC to an "i386" target. You can override this behaviour by defining TFM_NO_ASM or another optimization mode (such as SSE2).

SSE2

The "SSE2" mode is defined by "TFM_SSE2" and requires a Pentium 4, Pentium M or Athlon64 processor. It requires GCC to build. Note that you shouldn't define both TFM_X86 and TFM_SSE2 at the same time. This mode only works with 32–bit digits. In this mode fp_digit is 32–bits and fp_word is 64–bits. While this mode will work on the AMD Athlon64 series of processors it is less efficient than the native "x86–64" mode and not recommended.

There is an additional "TFM_PRESCOTT" flag that you can define for P4 Prescott processors. This causes the mul/sqr functions to use x86_32 and the montgomery reduction to use SSE2 which is (so far) the fastest combination. If you are using an older (e.g. Northwood) generation P4 don't define this.

x86–64

The "x86–64" mode is defined by "TFM_X86_64" and requires a "x86–64" capable processor (Athlon64 and future Pentium processors). It requires GCC to build and only works with 64–bit digits. Note that by enabling this mode it will automatically enable 64–bit digits. In this mode fp_digit is 64–bits and fp_word is 128–bits. This mode will be autodetected when building with GCC to an "x86–64" target. You can override this behaviour by defining TFM_NO_ASM.

ARM

The "ARM" mode is defined by "TFM_ARM" and requires a ARMv4 with the M instructions (enhanced multipliers) or higher processor. It requires GCC and works with 32–bit digits. In this mode fp_digit is 32–bits and fp_word is 64–bits.

PPC32

The "PPC32" mode is defined by "TFM_PPC32" and requires a standard PPC processor. It doesn't use altivec or other extensions so it should work on all compliant implementations of PPC. It requires GCC and works with 32–bit digits. In this mode fp_digit is 32–bits and fp_word is 64–bits.

Future Releases

Future releases will support additional platform optimizations. Developers of MIPS and SPARC platforms are encouraged to submit GCC asm inline patches (see chapter 4 for more information).

Processor	**Recommended Mode**
All 32–bit x86 platforms	TFM_X86
Pentium 4	TFM_SSE2
Pentium 4 Prescott	TFM_SSE2 + TFM_PRESCOTT
Athlon64	TFM_X86_64
ARMv4 or higher with M	TFM_ARM
G3/G4 (32-bit PPC)	TFM_PPC32
x86–32 or x86–64 (with GCC)	Leave blank and let autodetect work

Figure 1.1: Recommended Build Modes

1.3.5 Precision Configuration

The precision of all integers in this library are fixed to a limited precision. Essentially the rule of setting the precision is if you plan on doing modular exponentiation with k–bit numbers than the precision must be fixed to $2k$–bits plus four digits.

This is changed by altering the value of "FP_MAX_SIZE" in tfm.h to your desired size. By default, the library is configured to handle upto 2048–bit inputs to the modular exponentiator.

Chapter 2

Getting Started

2.1 Data Types

TomsFastMath is a large fixed precision integer library. It provides the functionality to manipulate large signed integers through a relatively trivial api and a single data type.

The "fp_int" or fixed precision integer is the data type that the functions operate with.

```
typedef struct {
    fp_digit dp[FP_SIZE];
    int      used,
             sign;
} fp_int;
```

The **dp** member is the array of digits that forms the number. It must always be zero padded. The **used** member is the count of digits used in the array. Although the precision is fixed the algorithms are still tuned to not process the entire array if it does not have to. The **sign** indicates the sign of the integer. It is **FP_ZPOS** (0) if the integer is zero or positive and **FP_NEG** (1) otherwise.

2.2 Initialization

2.2.1 Simple Initialization

To initialize an integer to the default state of zero use the fp_init() function.

```
void fp_init(fp_int *a);
```

This will initialize the fp_int a to zero. Note that the function fp_zero() is an alias for fp_init().

2.2.2 Initialize Small Constants

To initialize an integer with a small single digit value use the fp_set() function.

```
void fp_set(fp_int *a, fp_digit b);
```

This will initialize a and set it equal to the digit b.

2.2.3 Initialize Copy

To initialize an integer with a copy of another integer use the fp_init_copy() function.

```
void fp_init_copy(fp_int *a, fp_int *b)
```

This will initialize a as a copy of b. Note that for compatibility with LibTomMath the function fp_copy() is also provided.

Chapter 3

Arithmetic Operations

3.1 Odds and Evens

To quickly and easily tell if an integer is zero, odd or even use the following functions.

```
int fp_iszero(fp_int *a);
int fp_iseven(fp_int *a);
int fp_isodd(fp_int *a);
```

These will return **FP_YES** if the answer to their respective questions is yes. Otherwise they return **FP_NO**. Note that these are implemented as macros and as such you should avoid using ++ or − − operators on the input operand.

3.2 Sign Manipulation

To negate or compute the absolute of an integer use the following functions.

```
void fp_neg(fp_int *a, fp_int *b);
void fp_abs(fp_int *a, fp_int *b);
```

This will compute the negation (or absolute) of a and store the result in b. Note that these are implemented as macros and as such you should avoid using ++ or − − operators on the input operand.

3.3 Comparisons

To perform signed or unsigned comparisons use following functions.

```
int fp_cmp(fp_int *a, fp_int *b);
int fp_cmp_mag(fp_int *a, fp_int *b);
```

These will compare a to b. They will return **FP_GT** if a is larger than b, **FP_EQ** if they are equal and **FP_LT** if a is less than b.

The function fp_cmp performs signed comparisons while the other performs unsigned comparisons.

3.4 Shifting

To shift the digits of an fp_int left or right use the following functions.

```
void fp_lshd(fp_int *a, int x);
void fp_rshd(fp_int *a, int x);
```

These will shift the digits of a left (or right respectively) x digits.

To shift individual bits of an fp_int use the following functions.

```
void fp_div_2d(fp_int *a, int b, fp_int *c, fp_int *d);
void fp_mod_2d(fp_int *a, int b, fp_int *c);
void fp_mul_2d(fp_int *a, int b, fp_int *c);
void fp_mul_2(fp_int *a, fp_int *c);
void fp_div_2(fp_int *a, fp_int *c);
void fp_2expt(fp_int *a, int b);
```

fp_div_2d() will divide a by 2^b and store the quotient in c and remainder in d. Either of c or d can be **NULL** if their value is not required. fp_mod_2d() is a shortcut to compute the remainder directly. fp_mul_2d() will multiply a by 2^b and store the result in c.

The fp_mul_2() and fp_div_2() functions are optimized multiplication and divisions by two. The function fp_2expt() will compute $a = 2^b$ quickly.

To quickly count the number of least significant bits that are zero use the following function.

```
int fp_cnt_lsb(fp_int *a);
```

This will return the number of adjacent least significant bits that are zero. This is equivalent to the number of times two evenly divides a.

3.5 Basic Algebra

The following functions round out the basic algebraic functionality of the library.

```
void fp_add(fp_int *a, fp_int *b, fp_int *c);
void fp_sub(fp_int *a, fp_int *b, fp_int *c);
void fp_mul(fp_int *a, fp_int *b, fp_int *c);
void fp_sqr(fp_int *a, fp_int *b);
int fp_div(fp_int *a, fp_int *b, fp_int *c, fp_int *d);
int fp_mod(fp_int *a, fp_int *b, fp_int *c);
```

The functions fp_add(), fp_sub() and fp_mul() perform their respective operations on a and b and store the result in c. The function fp_sqr() computes $b = a^2$ and is faster than using fp_mul() to perform the same operation.

The function fp_div() divides a by b and stores the quotient in c and remainder in d. Either of c and d can be **NULL** if the result is not required. The function fp_mod() is a simple shortcut to find the remainder.

3.6 Modular Exponentiation

To compute a modular exponentiation use the following function.

```
int fp_exptmod(fp_int *a, fp_int *b, fp_int *c, fp_int *d);
```

This computes $d \equiv a^b \pmod{c}$ for any odd c and b. b may be negative so long as $a^{-1} \pmod{c}$ exists. The initial value of a may be larger than c. The size of c must be half of the maximum precision used during the build of the library. For example, by default c must be less than 2^{2048}.

3.7 Number Theoretic

To perform modular inverses, greatest common divisor or least common multiples use the following functions.

```
int fp_invmod(fp_int *a, fp_int *b, fp_int *c);
void fp_gcd(fp_int *a, fp_int *b, fp_int *c);
void fp_lcm(fp_int *a, fp_int *b, fp_int *c);
```

The fp_invmod() function will find the modular inverse of a modulo an odd modulus b and store it in c (provided it exists). The function fp_gcd() will compute the greatest common divisor of a and b and store it in c. Similarly the fp_lcm() function will compute the least common multiple of a and b and store it in c.

3.8 Prime Numbers

To quickly test a number for primality call this function.

```
int fp_isprime(fp_int *a);
```

This will return **FP_YES** if a is probably prime. It uses 256 trial divisions and eight rounds of Rabin-Miller testing. Note that this routine performs modular exponentiations which means that a must be in a valid range of precision.

Chapter 4

Porting TomsFastMath

4.1 Getting Started

Porting TomsFastMath to a given processor target is usually a simple procedure. For the most part assembly is used to get around the lack of a "add with carry" operation in the C language. To make matters simpler the use of assembler is through macro blocks.

Each "port" is defined by a block of code that re-defines the portable ISO C macros with assembler inline blocks. To add a new port you must designate a TFM_XXX define that will enable your port when built.

4.2 Multiply with Comba

The file "fp_mul_comba.c" is responsible for providing the fast multiplication within the library. This comba multiplication is fairly simple. It uses a sliding three digit carry system with the variables $c0$, $c1$, $c2$. For every digit of output $c0$ is the what will be that digit, $c1$ will carry into the next digit and $c2$ will be the "c1" carry for the next digit. For every "next" digit effectively $c0$ is stored as output, $c1$ moves into $c0$, $c2$ into $c1$ and zero into $c2$.

The following macros define the assmebler interface to the code.

```
#define COMBA_START
```

This is issued at the beginning of the multiplication function. This is in place to allow you to initialize any registers or machine words required. You

can leave it blank if you do not need it.

```
#define COMBA_CLEAR \
   c0 = c1 = c2 = 0;
```

This clears the three comba carries. If you are going to place carries in registers then zero the appropriate registers. Note that the functions do not use $c0$, $c1$ or $c2$ directly so you are free to ignore these varibles and use registers directly.

```
#define COMBA_FORWARD \
   c0 = c1; c1 = c2; c2 = 0;
```

This propagates the carries after a digit has been produced.

```
#define COMBA_STORE(x) \
   x = c0;
```

This stores the $c0$ digit in the memory location specified by x. Note that if you manually aliased $c0$ with a register than just store that register in x.

```
#define COMBA_STORE2(x) \
   x = c1;
```

This stores the $c1$ digit in the memory location specified by x. Note that if you manually aliased $c1$ with a register than just store that register in x.

```
#define COMBA_FINI
```

If at the end of the function you need to perform some action fill this macro in.

```
#define MULADD(i, j)                                          \
   t  = ((fp_word)i) * ((fp_word)j);                          \
   c0 = (c0 + t);              if (c0 < ((fp_digit)t))  ++c1; \
   c1 = (c1 + (t>>DIGIT_BIT)); if (c1 < (t>>DIGIT_BIT)) ++c2;
```

This macro performs the "multiply and add" step that is central to the comba multiplier. It multiplies the fp_digits i and j to produce a fp_word result. Effectively the double–digit value is added to the three-digit carry formed by $c0$, $c1$, $c2$ where $c0$ is the least significant digit.

4.3 Squaring with Comba

Squaring is similar to multiplication except that it uses a special "multiply and add twice" macro that replaces multiplications that are not required.

```
#define COMBA_START
```

This allows for any initialization code you might have.

```
#define CLEAR_CARRY \
   c0 = c1 = c2 = 0;
```

This will clear the carries. Like multiplication you can safely alias the three carry variables to registers if you can/want to.

```
#define COMBA_STORE(x) \
   x = c0;
```

Store the $c0$ carry to a given memory location.

```
#define COMBA_STORE2(x) \
   x = c1;
```

Store the $c1$ carry to a given memory location.

```
#define CARRY_FORWARD \
   c0 = c1; c1 = c2; c2 = 0;
```

Forward propagate all three carry variables.

```
#define COMBA_FINI
```

If you need to clean up at the end of the function.

```
/* multiplies point i and j, updates carry "c1" and digit c2 */
#define SQRADD(i, j)                          \
   t  = ((fp_word)i) * ((fp_word)j);          \
   c0 = (c0 + t);              if (c0 < ((fp_digit)t))  ++c1; \
   c1 = (c1 + (t>>DIGIT_BIT)); if (c1 < (t>>DIGIT_BIT)) ++c2;
```

This is essentially the MULADD macro from the multiplication code.

```
/* for squaring some of the terms are doubled... */
#define SQRADD2(i, j)                          \
   t  = ((fp_word)i) * ((fp_word)j);       \
   c0 = (c0 + t);              if (c0 < ((fp_digit)t))  ++c1; \
   c1 = (c1 + (t>>DIGIT_BIT)); if (c1 < (t>>DIGIT_BIT)) ++c2; \
   c0 = (c0 + t);              if (c0 < ((fp_digit)t))  ++c1; \
   c1 = (c1 + (t>>DIGIT_BIT)); if (c1 < (t>>DIGIT_BIT)) ++c2;
```

This is like SQRADD except it adds the produce twice. It's similar to computing SQRADD(i, j*2).

To further make things interesting the squaring code also has "doubles" (see my LTM book chapter five...) which are handled with these macros.

```
#define SQRADDSC(i, j)                                                    \
   do { fp_word t;                                                        \
      t =  ((fp_word)i) * ((fp_word)j);                                   \
      sc0 = (fp_digit)t; sc1 = (t >> DIGIT_BIT); sc2 = 0;                 \
   } while (0);
```

This computes a product and stores it in the "secondary" carry registers $\langle sc0, sc1, sc2 \rangle$.

```
#define SQRADDAC(i, j)                                                    \
   do { fp_word t;                                                        \
   t = sc0 + ((fp_word)i) * ((fp_word)j);  sc0 = t;                       \
   t = sc1 + (t >> DIGIT_BIT);             sc1 = t; sc2 += t >> DIGIT_BIT;  \
   } while (0);
```

This computes a product and adds it to the "secondary" carry registers.

```
#define SQRADDDB                                                          \
   do { fp_word t;                                                        \
   t = ((fp_word)sc0) + ((fp_word)sc0) + c0; c0 = t;
   t = ((fp_word)sc1) + ((fp_word)sc1) + c1 + (t >> DIGIT_BIT); c1 = t;
   c2 = c2 + ((fp_word)sc2) + ((fp_word)sc2) + (t >> DIGIT_BIT);
   } while (0);
```

This doubles the "secondary" carry registers and adds the sum to the main carry registers. Really complicated.

4.4 Montgomery with Comba

Montgomery reduction is used in modular exponentiation and is most called function during that operation. It's important to make sure this routine is very fast or all is lost.

Unlike the two other comba routines this one does not use a single three–digit carry system. It does have three–digit carries except that the routine steps through them in the inner loop. This means you cannot alias them to registers (at all).

To make matters simple though the three arrays of carries are stored in one array. The "c0" array resides in $c[0 \ldots OFF1-1]$, "c1" in $c[OFF1 \ldots OFF2-1]$ and "c2" in $c[OFF2 \ldots OFF2 + FP\_SIZE - 1]$.

```
#define MONT_START
```

This allows you to insert anything at the start that you need.

```
#define MONT_FINI
```

This allows you to insert anything at the end that you need.

```
#define LOOP_START \
   mu = c[x] * mp;
```

This computes the μ value for the inner loop. You can safely alias mu and mp to a register if you want.

```
#define INNERMUL                                          \
   do { fp_word t;                                        \
   _c[0] = t  = ((fp_word)_c[0] + (fp_word)cy) +          \
                (((fp_word)mu) * ((fp_word)*tmpm++)); \
   cy = (t >> DIGIT_BIT);                                 \
   } while (0)
```

This computes the inner product and adds it to the destination and carry variable cy. This uses the mu value computed above (can be in a register already) and the cy which is a chaining carry. Inside the INNERMUL loop the cy value can be kept inside a register (hint: it always starts as $cy = 0$ in the first iteration).

Upon completion of the inner loop the macro LOOP_END is called which is used to fetch cy into the variable the C program can see. This is where, if you cached cy in a register you would copy it to the locally accessible C variable.

```
#define PROPCARRY \
   do { fp_digit t = _c[0] += cy; cy = (t < cy); } while (0)
```

This propagates the carry upwards by one digit.

Index