

# Bignum Arithmetic

Felix von Leitner

CCC Berlin

`felix-bignum@fefe.de`

December 2006

## Abstract

Everyone has used encryption software before; PGP, SSL, IPsec, ... most of these use public key cryptography, which involves working with large numbers (*bignums*, 1024 bits and up). But how do you actually work with numbers like that?

Implementing Bignum Arithmetic

# Agenda

1. Representing big numbers
2. Adding big numbers
3. Multiplying big numbers
4. Modulo operations on big numbers

## Representing bignums

1. Two's complement?  
Does not work well with bignums.
2. COBOL style decimal (0-9 in each nibble)?  
Works. Does not scale, though.
3. Absolute number plus sign bit!

## Representing bignums: OpenSSL

```
struct bignum_st
{
    BN_ULONG *d;    /* ptr to array of 'BN_BITS2' bit chunks. */
    int top;        /* Index of last used d +1. */
    /* The next are internal book keeping for bn_expand. */
    int dmax;       /* Size of the d array. */
    int neg;        /* one if the number is negative */
    int flags;
};
```

BN\_ULONG is basically a `size_t`, i.e. as wide as a register.

## Representing bignums: gcrypt (gnutls)

```
struct gcry_mpi {
    int allocated;    /* array size (# of allocated limbs) */
    int nlimbs;      /* number of valid limbs */
    int sign;        /* indicates a negative number and is used for
                     * opaque MPIs to store the length */
    unsigned flags; /* bit 0: array in secure memory space */
                   /* bit 2: pointer to some m_allocated data */
    mpi_limb_t *d;  /* array with the limbs */
};
```

Again, `mpi_limb_t == size_t`.

## Representing bignums: gmp

```
typedef struct
{
    int _mp_alloc;          /* Number of *limbs* allocated and pointed
                           to by the _mp_d field.  */
    int _mp_size;          /* abs(_mp_size) is the number of limbs the
                           last field points to.  If _mp_size is
                           negative this is a negative number.  */
    mp_limb_t *_mp_d;      /* Pointer to the limbs.  */
} __mpz_struct;
```

Again, `mp_limb_t == size_t`.

## Adding big numbers

- $\text{sizeof}(\text{result}) = \max(\text{sizeof}(\text{src1}), \text{sizeof}(\text{src2})) + 1$
- CPUs have add with carry
- C exposes that through casting to the next bigger integer type
- However, there might not be one. gcc on amd64 has `__int128`.

```
for (l=0, i=0; i<m; ++i) {  
    l += (unsigned long long)src1[i] + src2[i];  
    dest[i] = l;  
    l >>= 32;  
}
```

## Adding big numbers - issues

- Linear (not parallelizable)
- Could do addition, save carries, do second pass for carry adjustment
- Turns out: not worth it
- Bottleneck is memory access, not addition
- Pentium 4: 1/2 cycle per add, but 8 cycles per adc



## Subtracting big numbers

- Same as addition, basically
- First settle sign bits, might end up as addition
- If result changes sign, swap operands and xor sign bit

# Multiplication

- $\text{sizeof}(\text{result}) = \text{sizeof}(a) + \text{sizeof}(b)$
- "School method"

```
a  b  c  *  d  e  f
-----
                cd ce cf
            bd be bf
        ad ae af
-----
                [...] cf
```

## Multiplication - issues

- Results of limb multiplications have double width
- Cannot use double width number to add two and store carry!
- Costly:  $n^2$  muls,  $3 * n^2$  adds
- Interestingly, the adds are free (hidden by mul latency).  
Welcome to 21st century multiscalar CPUs!  
This also means that we can't speed this up with SIMD.

## Multiplication - issues

```
unsigned long long temp[n*n];  
for (i=0; i<n; ++i)  
    for (j=0; j<n; ++j)  
        temp[j*n+i]=(acc)a[i]*b[j];
```

This code takes over 7000 cycles for two random 1024 bit numbers, and it does neither addition, nor carry calculation, nor does it write the actual result.

The whole OpenSSL multiplication routine takes less than 5000 cycles.

So how do they do it?

## Modern Computer Architecture

- Keep stuff in registers, not memory  
Potential loss: **200+ cycles**
- Mispredicted conditional jumps are expensive  
Potential loss: 20 cycles on P4, less for other CPUs
- Multiplication is expensive, use shift if you can  
Potential loss: 8 cycles

Only important optimization: minimize memory accesses.  
No temp arrays or variables! Read each bignum once, write result once.

## Multiplication, next attempt

- Do multiply and carry adjustment for each row
- For first row, write directly to result. For other rows, add to result
- Still unnecessary memory accesses, but at least no temp array
- This is my current version: 1414 amd64 cycles vs 1509 for openssl

## Other Ideas

- Need  $\lg(n)$  bits to hold carry
- Reduce word length so that accumulator can hold product plus carry
- Pro: Needs less additions in inner loop
- Con: Loop needs to run more often
- Might still be worth it to get better parallelism
- Haven't actually tried this

## Comba Multiplication

- Like school method, but go by column
- Do rightmost column first, write directly to result
- Should be even faster, but isn't on my Athlon 64
- Speed record: tomfastmath with comba: 948 amd64 cycles
- But that is massively unrolled, and special cased; 300k code just for mul
- Can (with tricks) be done just within x86 registers when unrolled



## Karatsuba Multiplication

- Split numbers in high and low part (reachable via shifting)
- Normal "long" multiplication takes 4 muls for the four parts
- Clever arithmetic trades one multiplication for several adds and subs
- Great in academic paper, slower in practice (until 2048+ bits)

$$(a + b10^n)(c + d10^n) = ac + ((a + b)(c + d) - ac - bd) 10^n + bd10^{2n}$$

## Toom-Cook Multiplication

- Generalizes Karatsuba to  $n$ -part split
- Even harder to implement
- Looks even better in academic papers
- Is even slower in practice (4096+ bits)

## FFT Multiplication (Schönhage-Strassen)

- Choose smaller limb size
- Squint eyes, be under the influence of certain drugs
- Suddenly multiplication looks like convolution :-)
- Compute FFT for  $a$  and  $b$ , multiply pairwise, compute inverse FFT
- Looks even greater in academic paper
- Is even less worth it in practice (used for calculating  $\pi$  to 1e6 digits)

## Some Benchmarks: 64-bit mode

These are cycle counts on Athlon 64 and Core 2 CPUs in 64-bit mode:

	add (a64)	mul / comba	add (c2)	mul
My code	83	1414 / 1759	336	2100 / 1704
gmp	111	1512	324	1752
OpenSSL	153	1509	216	1680
tomsfastmath	127	948	204	1344
gcrypt	188	6375	240	5892

Note: 16 limbs! The add step in the mul alone takes 16 times the add cycles, so we can also view this as "the muls are free".

## Some Benchmarks: 32-bit mode

	add (a64)	mul / comba	add (c2)	mul
My code	156	4838 / 6716	360	6708 / 6900
gmp	123	3845	204	4776
OpenSSL	156	3996	276	5184
tomsfastmath	519	3067	540	4584
gcrypt	146	7850	228	9444

## Some Benchmarks: ASM vs various C Compilers

	gcc 64	ASM 64	gcc 32	icc 32	sun c 32	ASM 32
add (Athlon 64)	132	84	509	240	660	140
mul (Athlon 64)	2512	1414	11132	7313	13147	4780
comba (Athlon 64)	2425	1465	16489	11539	11812	6181
add (Core 2)	204	336	540	288	372	360
mul (Core 2)	2496	2112	10128	7896	9180	6744
comba (Core 2)	2604	1752	17316	10812	10104	6852
add (P4)			956	400	544	436
mul (P4)			33088	18892	26076	26112
comba (P4)			38316	24484	22492	21552

## Exponentiation

- temp = number to be squared
- result = 1
- For each bit in exponent: temp \*= temp
- If i'th bit in exponent is set, result \*= temp

Problem: HUGE result. In crypto, we don't actually want the result; we want the result modulo some prime.

Note: At least half of the multiplications are squaring.

## Squaring

$$\begin{array}{r}
 a \quad b \quad c \quad * \quad a \quad b \quad c \\
 \hline
 \phantom{a \quad b \quad c} \phantom{*} \phantom{a \quad b \quad c} ca \quad cb \quad cc \\
 \phantom{a \quad b \quad c} \phantom{*} \phantom{a \quad b \quad c} \phantom{ca} ba \quad bb \quad bc \\
 \phantom{a \quad b \quad c} \phantom{*} \phantom{a \quad b \quad c} \phantom{ca} \phantom{ba} aa \quad ab \quad ac \\
 \hline
 \phantom{a \quad b \quad c} \phantom{*} \phantom{a \quad b \quad c} \phantom{ca} \phantom{ba} \phantom{aa} [\dots] \quad cc
 \end{array}$$

Note that  $ca = ac$ ; we can save a third of the multiplications. Note however that the additional bookkeeping may make the benefit disappear. Also, we still need to do the adds that were previously hidden in the latency of the muls we just saved.



## Modulo Arithmetic

- Public key crypto works on residue classes
- We have to calculate remainder after divided by some large prime number
- mul latency: 12 cycles on Intel 64-bit, 5 cycles on AMD 64-bit
- div latency: 161 cycles on Intel 64-bit, 71 cycles on AMD 64-bit
- mul is pipelined (on Intel only on 32-bit), div is not
- **Division is horribly slow and must be avoided if at all possible**

## Modulo Arithmetic

- First: reduce operands modulo prime
- For addition, if result  $>$  prime, subtract prime
- For multiplication, some trickery is involved

## Multiplication modulo $r$

- Reduce all intermediate values
- Problem: shifting by a column was free previously, now isn't
- Still very inefficient

## Montgomery Multiplication

- $dest=0$
- For each bit in  $a$  (starting at lowest bit):
  1. if bit is set,  $dest += b$
  2. if lowest bit in  $dest$  is set,  $dest += r$
  3.  $dest \gg= 1$
- if  $dest > r$  then  $dest -= r$
- Result:  $ab2^n \bmod r$  ( $n$  is the number of bits in  $a$ )

## Montgomery Multiplication

- Only works if  $2^n$  and  $r$  are coprime (which is true for crypto)
- Get actual result by multiplying with  $2^n \bmod r$
- Alternative, multiply by  $2^n \bmod r$  (precomputed value) before and by  $2^{-n} \bmod r$  (other precomputed value) after the multiplication
- For exponentiation, you only do the pre- and post-multiply once