Representing and working with almost none of the integers the suckless way

Or

The design, decisions, and discoveries behind and from libzahl

Mattias Andrée ⟨maandree@kth.se⟩

Thanks to Marc Collin for proofreading.

Copyright © 2016 Mattias Andrée ⟨maandree@kth.se⟩
Permission to use, copy, and/or distribute, but not modify, this document for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

Abstract

This is a technical discussion about the design and decisions behind libzahl, a big integer library whose goal align with the suckless philosophy: simplicity, clarity, and frugality. These are not goals of numeric libraries, traditionally; they are difficult to combine with performance when the performance is dependent on algorithms. Is it possible to create a simple big number library competitive with the fastest big number libraries around?

What is libzahl?

libzahl[1] is a big integer library. It aims only to support integers and to be suckless[2]. Principal questions in its development include:

1. How much complexity is acceptable?
2. What functions do we really need to implement?

   Omitting functions is not about reducing complexity; they are independent and do not contribute to complexity. It is about not wasting time on stuff we do not need, and about keeping the API simple and easy to learn.
3. When is it worth using assembly?

   The reason for avoiding assembly is simply that it is not portable and cannot replace C code.
4. Which functions used internally should also be available via the API?
5. How can the API and use of libzahl suck less?

In a typical suckless project, question 2 is the central question, and question 1 need not be considered and complexity is always minimised. There are of course a few programs, e.g. `diff`, where question 1 is important. This is when the program needs an algorithm but needs to do optimisation for the algorithm to work on all common input without exhausting memory or taking too long, the latter is of course why this question is asked in libzahl. Question 5 is also asked for suckless libraries.

libzahl does not try to fit suckless into performance, but tries to fit performance into suckless. This is an important distinction, if I have to sacrifice too much simplicity for performance, I wouldn't do it.

The first release (version 1.0) of libzahl was tagged one week after its initial commit. The initial commit was made a day or two after the project began, I don't recall exactly. libzahl has not changed too much since.

[1] http://git.suckless.org/libzahl

[2] http://suckless.org/philosophy

Initialise the library

Unlike other bigint libraries, the user must initialise libzahl before it is used. The original reason for this is that libzahl uses global variables for temporary storage of big integers it uses internally. Hence, big integer constants, a preallocated stack which we will discuss later, and explicit zeroing of global memory have been added.

This function, which the user must call, is named `zsetup`. It could easily have been avoided by using ‘‘constructors’’.

```
static void __attribute__((__constructor__))
init_libzahl(void)
{
    zsetup();
}
```

However, as you can see, constructors are a compiler-extension, and using it would make libzahl compiler-unportable. Worse, I don't believe constructors are advantageous enough to warrant inclusion in a suckless compiler. However, we will see momentarily that letting the user call `zsetup` is the only viable option for libzahl's design — I left one thing out.

Dealing with exceptional conditions

The best way to deal with exceptions is the C way: checking the return value, and in extraordinary cases: the value of a global lvalue (named `errno` in C). For example

```
if (write(fd, buf, n) < 0)
    "handle error";
```

This method is very clean and makes the programmer perfectly aware of what is going on and what could go wrong. The author of ØMQ has a great article that discusses some aspects of this.[3]

A less common strategy in C is to use long jumps (`setjmp`). This strategy was selected for libzahl. Its advantages are:

- Less branching, thus better performance.[4]
- Fewer error-checks to clutter the code.

These advantages can be seen both in the user software and in the library. Obviously the advantages of the traditional method are lost (except it is even cleaner), but since we do not really expect errors to occur and we basically know which errors can occur, not much is actually lost. The real disadvantage, however, is that unwinding changes, such as freeing temporary allocations, on errors become much more difficult. In libzahl, all temporary allocations are available as global variables, or are pushed onto a stack, and are freed on error. Consequently, all temporary variables that are created by recursive functions, are pushed onto a stack when they are initialised, and freed in reverse order. This excerpt from libzahl's multiplication function demonstrates this principle:

```
zinit_temp(b_high);
zinit_temp(b_low);
zinit_temp(c_high);
zinit_temp(c_low);
/* Do some maths. */
zfree_temp(c_low);
zfree_temp(c_high);
zfree_temp(b_low);
zfree_temp(b_high);
```

[3] http://250bpm.com/blog:4

[4] However, branching is unbelievably cheap on modern CPU:s.

To enable long jumps, a return point must be selected. When designing libzahl, there were two choices for how this could be done:

```
/* Alternative 1: */
int
zsetup(void)
{
    if (setjmp(jmppoint))
        return 1;
    /* Initialise. */
    return 0;
}

int
main(void)
{
    if (zsetup())
        "handle error";
    /* Do stuff. */
}
```

```
/* Alternative 2: */
void
zsetup(jmp_buf jmp)
{
    *jmppoint = *jmp;
    /* Initialise. */
}

int
main(void)
{
    jmp_buf jmp
    if (setjmp(jmp))
        "handle error";
    zsetup(jmp);
    /* Do stuff. */
}
```

The second alternative was chosen for libzahl because it is slightly more flexible, and the complexity is the same if you consider both the user program and the library. Another advantage with this option is that the compiler is aware that `setjmp` can return multiple times in the same process — and is aware of `setjmp`'s return-value semantics, — and can warn the developer of its consequences: non-`volatile` variables with automatic store duration may contain unspecified values.[5]

The return point for the long jump can be updated by calling `zsetup` with a new return point. Reinitialisation is only performed if `zunsetup` has been called.

To be clear, error handling by long jumps is usually a bad idea.

[5] longjmp(3p). (No, it is not documented in setjmp(3p).)

Function names

An important part of making the API simple is to use intuitive, short, and familiar function names, in contrast to technical and mathematical names. Below is a list of libzahl function names side-by-side with corresponding GNU MP[6] function names, where the names differ in more than namespace.

`zfree`	`mpz_clear`
`zsave`	`mpz_out_raw`
`zload`	`mpz_inp_raw`
`zstr`	`mpz_get_str(, 10,)`
`zsets`	`mpz_set_str(,, 10)`
`zsetu`	`mpz_set_ui`
`zseti`	`mpz_set_si`
`zptest`	`mpz_probab_prime_p`
`zbtest`	`mpz_tstbit`
`zbset(,,, 1)`	`mpz_setbit`
`zbset(,,, 0)`	`mpz_clrbit`
`zbset(,,, -1)`	`mpz_combit`
`zcmpmag`	`mpz_cmpabs`
`zeven`	`mpz_even_p`
`zodd`	`mpz_odd_p`
`zsignum`	`mpz_sgn`
`zor`	`mpz_ior`
`zbits`	`mpz_sizeinbase(, 2)`
`zlsb`	`mpz_scan1(, 0)`
`zlsh`	`mpz_mul_2exp`
`zrsh`	`mpz_tdiv_q_2exp`
`ztrunc`	`mpz_tdiv_r_2exp`
`zdiv`	`mpz_tdiv_q`
`zmod`	`mpz_tdiv_r`
`zdivmod`	`mpz_tdiv_qr`
`zpowu`	`mpz_pow_ui`
`zmodpow`	`mpz_powm`
`zmodpowu`	`mpz_powm_ui`
`zrand(,, UNIFORM,)`	`mpz_urandomm`

In `zsets`, `zsetu`, `zseti`, `zpowu`, and `zmodpowu`, the last letter lets us know the type of one of the parameters. These letters are '`s`', '`u`', and '`i`' which are derived from the `printf`/`scanf`-format string codes '`%s`', '`%u`', and '`%i`', respectively. In GNU MP, this is also known, but the suffixes are '`_str`', '`_ui`', and '`_si`', which are probably derived from "string", '`unsigned int`',

[6] https://gmplib.org/

and '`signed int`', respectively.

We can also observe how modulus is indicated by the function names. In libzahl, division and modulus is named `zdivmod`, which is derived from, "**Z**ahlen, **div**ision and **mod**ulus". Modular multiplication (not listed above) and modular exponentiation are called, respectively, `zmodmul` and `zmodpow`, derived from "**Z**ahlen, **mod**ular **mul**tiplication" and "**Z**ahlen, **mod**ular **pow**er". '`mod`' comes before the base operation name if its a modular arithmetic function, but if it calculates the modulus of the input, '`mod`' is at the end. In GNU MP, '`m`' if appended to the base operation's name if it is a modular variant, and for division/modulus, the affixes '`_q`' and '`_r`' (where '`_`' is shy) are used to tell whether the quotient or the remainder is returned.

In GNU MP, we can see that the suffix '`_p`' is used indicate that it is a predicate (returns true or false[7]).[8] In libzahl we skip this, you know it is a predicate. Similarly, you know `zor` is inclusive, so there is no 'i' in there to tell you so. You would probably forget the 'i' if it was there, or not think to try it if you haven't read the documentation.

Another interesting difference is the naming of `zsignum` and `mpz_sgn`. 'sgn' is the mathmatical abbreviation for 'signum', but it could also be confused with 'sign'. The signum function is defined as

$$\operatorname{sgn} x = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases},$$

but there is some ambiguity to what 'sign' means: what is the sign of 0, and how are negative and positive — and potentially zero — encoded?

[7] Of course GNU MP is a little lax about this in the case of `mpz_probab_prime_p` which can return "not only probably, it is definitely a prime".

[8] http://www.catb.org/~esr/jargon/html/p-convention.html

Parameter order

A second important part of making the API simple is to order parameters in an intuitive order. Results cannot be returned as the function's return value for technical reasons, lest there will be a drastic performance penalty and using the library will become messy. Traditionally, in C, the output parameters are at the end parameter list, or in case the function is variadic, at the beginning of the parameter list. There are of course a few odd-cases like `read` — were the output is the middle, — but these usually make sense. We call this [output is last] BSD MP-style. GNU MP put the output parameters at the beginning.

BSD MP-style	GNU MP-style
`add(augend, addend, sum);`	`add(sum, augend, addend);`

This can be compared with how it would be expressed in pseudocode and mathematics:

BSD MP-style	GNU MP-style
$augend + addend \rightarrow sum$	$sum \leftarrow augend + addend$
$augend + addend =: sum$	$sum := augend + addend$
	$sum \triangleq augend + addend$

BSD MP-style is used in LibTomMath[9] and TomsFastMath[10], whilst GNU MP-style is used in libzahl and Hebimath[11].

Since you are doing mathematics with libzahl, it is natural to order the parameters in the same order as in mathematics, and in a way that makes it easier to translate from pseudocode.

[9] `https://github.com/libtom/libtommath`

[10] `https://github.com/libtom/tomsfastmath`; not a true bignum library.

[11] `https://github.com/suiginsoft/hebimath`

Reduced function set

libzahl avoids compound functions, such as initialise and assign in one single function, or initialising multiple big integers. The only compound functions available in libzahl are:

`zdivmod` Calculate the quotient and the remainder. This is available because when you calculate the quotient you get the remainder for free. `zdiv` and `zmod` simply call `zdivmod` with the dummy variable as one of the output variables.

`zsplit` Combines `zrsh` and `ztrunc`. It is useful for divide-and-conquer algorithms. It orders the calls to `zrsh` and `ztrunc` so that it is safe to use input variables as output variables too.

Compound functions pollute the API and seldom make the user code cleaner. If the user needs compound functions, they can easily be implemented as simple macros. Variadic functions like multi-integer initialisation and multi-integer deinitialisation are not only seldom especially useful, they are also slower and the overhead can be avoided with macros.[12]

libzahl also does not have variants of functions that take an intrinsic integer instead of a big integer as one of the input parameters, with the exception of exponentiation functions and comparison. The former exception is simply because raising a number to more than the $(2^{64} - 1)^{\text{th}}$ power is probably uncommon, and would — provided that the base is not -1, 0, or 1 — result in a number too large to be stored. Similarly, the latter exception is because it turns out it is very common than we want to compare against small constants. Exponentiation is also significantly faster when working on an intrinsic type as the exponent. Currently, `zpow` and `zmodpow`, which do not take an intrinsic integer as the exponent, do not convert the exponent to an intrinsic integer.

In most cases, taking intrinsic integers as an input does not provide a significant performance improvement. If the user wants to use an intrinsic integer, she can either have preassigned constants or use a preinitialised variable for temporary storage, like so:

[12] See libzahl reference manual, section ‘‘Variadic initialisation’’.

```
static z_t addu_temp;
static z_t one;

static inline void
addu(z_t r, z_t a, uint64_t b)
{
    zsetu(addu_temp, b);
    zadd(r, a, addu_temp);
{

static inline void
inc(z_t r, z_t a)
{
    zadd(r, a, one);
{

int
main(void)
{
    zinit(addu_temp);
    zinit(one);
    zseti(one, 1);
    /* Do stuff. */
}
```

Not implemented here

When implementing a function, there is a responsibility to make sure it is correct, robust, cannot leak resources even on failure, and is optimised. There is also a responsibility for libzahl to keep the implementation as simple as possible whilst achieving this. Therefore, only functions that are absolutely necessary: really common functions, and functions that are unrealistic for users to implement reasonably optimised on their own outside libzahl.

The set of functions implemented in libzahl is fairly small. Functions that are not implemented in libzahl often have a prototypical implementation, or mathematical expression, in the manual, so that it will be easier for users that do need the functions. This list is fairly large.[13]

Notable functions that have been left out from libzahl include:

- Extended greatest common divisor
- Least common multiple
- Modular multiplicative inverse
- Random prime number generation
- Legendre/Jacobi/Kronecker symbol
- Roots
- Modular roots
- Factorial
- Fibonacci/Lucas numbers

libzahl only provides truncated division. That is, rounded towards zero. This is the simplest division to implement and is also the division you will find in C99,[14] and newer revisions of ISO C, and is the most intuitive. In this division, modulo has the same sign as the dividend, and the absolute value of the remainder is less than the absolute value of the divisor.

There are several other ways to round values.[15] The libzahl manual include prototypical implementations of variants of `zdivmod` that implement the tie-breaking rules.[16]

[13] See libzahl reference manual, section "Not implemented".

[14] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, page 82 (94)

[15] https://en.wikipedia.org/wiki/Rounding#Rounding_to_integer

[16] See libzahl reference manual, section "Division".

Memory management

One problem in designing a bignum library is how to handle memory. Should the user allocate memory manually? This is probably the best for performance. The user can use dynamic memory allocation only when it is necessary. On the other hand, this makes the library less friendly, and there would be a lot of memory allocation code in the user program. Additionally, recursive functions, such as multiplication, may require memory to be allocated at each recursion. This memory could be allocated with either a `malloc`-family function, or with `alloca`. The problem with `alloca` however is that if it cannot allocate memory the program crashes, and it is too difficult to predict whether it will be successful or not without crashing if it would fail.[17]

Since libzahl requires optimised dynamic memory allocation internally, and letting the user do memory allocation would make the user program cluttered, libzahl does all memory allocation. It is however possible for the user to preallocate memory if she knows what she is doing.

To optimise dynamic memory allocation, libzahl uses a memory pool which consists of allocation buckets where all allocations have a size that is a power of two, plus a few fluff-bytes.[18] The buckets have a growth factor of 1.5.

[17] `https://github.com/maandree/slibc/blob/master/src/alloca/needstack.c#L28-L79`

[18] See libzahl reference manual, section “Integer structure”.

Future directions

In the future, modular multiplication and modular exponentiation may be removed and archived in the manual. It does not appear that modular bignum arithmetic is particularly useful outside cryptography — a field that libzahl does not intend to cover. Supporting cryptography means that side-channel attack resilient versions of all functions must be written (sometimes it entails adding alternative less performant implementations), this requires cryptographic expertise that I do not possess. I do welcome cryptographers to start a cryptographic branch or cryptographic bignum library that uses libzahl.

Currently libzahl is not thread-safe, but is designed to work well on clusters and in multiprocess[19] applications. This may or may not be amended in the future. Thread-safety is a problem because libzahl uses global variables.

The stochastic superoptimiser STOKE[20] will most probably be used in the future. Traditional superoptimisers exhaustively try possible optimisations based on known rewrite rules. Before doing this, STOKE tries to synthesise equivalent programs which cannot be reached with these optimisation rules. My primary hope with STOKE is not to get faster binaries, but to get simpler source code — faster binaries is a bonus. STOKE has been tested on OpenSSL's modular multiplication algorithm, so the outlook is good. Hopefully, we can even rid ourselves of assembly code. Currently there is trace amounts of extended inline assembly in libzahl, which is a compiler-extension.

libzahl will probably *not* use CPU dispatching[21]. If it is added, it will be on an opt-in basis. (This would be useful for distributions that ship binaries.)

libzahl is an integer-only library. In the future we need at least also rationals and floating-points. This will not be implemented in libzahl. They should be separate libraries from both libzahl and each other, but should depend on libzahl.

[19] Be sure you read “process”, not “processor”.

[20] `https://cs.stanford.edu/people/sharmar/pubs/asplos291-schkufza.pdf`

[21] Calling different versions of a function based on the CPUID.

Can GNU MP be beaten?

Outperforming GNU MP, with all its decades of development[22] from computer scientists and mathematicians[23], may seem like a futile task. GNU MP has been optimised to work for all ranges of sizes, from the tiny to the very huge, and is explicit about performance being more important than simplicity or elegance.[24] How could libzahl, written by one person, be able to compete with GNU MP in the ranges of integers with tens of milliards of bits[25]?

Optimising for integers so large most computers can only fit one of them in memory is not important. I'm focusing on 100 to 4096 bits, a range I believe covers almost all uses. And the performance is promising in this range; outperforming most competitors for most functions already.[26] However, only acceptable performance is necessary for this to be a successful project in my opinion. When this range has been optimised fully, I will move on to 8K bits, and perhaps larger later still.

And we must not forget that progress made in one bignum library can be reused and refined in other bignum libraries.

Another question that has come up is whether it is fair to compare high-level functions, should you not compare low-level functions? I hope that the difference in performance is not more than an $\mathcal{O}(1)$ overhead, if this is not the case, there is obviously something wrong. Even if not, I do not believe that comparing against low-level functions is the right thing to do. Low-level functions are only recommended if all nanoseconds per second of performance is important. For most users it is not, and low-level functions in GNU MP are quite a chore and should not be bothered with unless it is necessary.

But there are advantages with libzahl over GNU MP that does not have to do with performance:

- it is better suited for small computers,
- it only uses POSIX-standardised libc functions, so it can for example be compiled with glibc but linked with musl,
- it is suckless, so it is inherently easier to fix bugs, but bugs are also less probable,

[22] Let us assume that it does not indicate decades of legacy.

[23] Four active developers.

[24] https://gmplib.org/manual/Introduction-to-GMP.html

[25] Also known as gigabytes.

[26] http://git.suckless.org/libzahl/tree/STATUS

- it is suckless, so it is better for teaching programming and bignum algorithms (in my opinion, it beats LibTomMath whose goal is education).

Benchmarking

Nobody does benchmarks right. Hence the inclusion of this here. Please suggest improvements!

To assess libzahl's performance, libzahl, GNU MP, LibTomMath, TomsFastMath, and previously Hebimath[27], are benchmarked and the results compared. The libraries are benchmarked under equal conditions; translation layers, in form of macros, and `static inline` functions when necessary, are used to translate libzahl-calls to calls for other libraries without penalty. When there are slight differences in the functions, such as whether a boundary for random number generation is inclusive or exclusive, this discrepancy is not adjusted. That is, the translations are intentionally not perfect, but there is not performance penalty. Additionally, when a function is missing, it is given an unoptimised implementation. These implementations are meant to mimic how the user would implement the function, not how the library's developer would.

Input for functions are balanced, but I intend to benchmark with unbalanced input later on where it is necessary. Currently, all functions are only tested for one input, but I hope to benchmark for worst case, average case, and best case in the future.

For each input, a function is run a number of times, 1000 times for fast functions. The results are sorted and the middle seventh is selected, and an arithmetic average is calculated for these. I have arrived to this method by testing a number of simple methods. This method seems to produce graphs with low entropy. When I run the benchmarks I run them between 50 and 500 times depending on how long I can wait for the results. For each input, the fastest result is selected. The results are very stable, if this is redone the new results do not differ significantly, if at all.

Before the benchmark starts, the process is fixed to one CPU, I never run two benchmarks concurrently, as interference could be tangible, specially if you are not careful about to which CPU:s you assign the process.

I am planning to add a warm-up loop to force the CPU to throttle itself (to prevent catching fire). This loop would run until the frequency is set the minimum. Of course, the scaling governor would be set before this loop begins, and reset when the process exits. This warm-up is intended to prevent the CPU from being throttled mid-benchmarking.

[27] Hebimath is no longer benchmarked because it was not stable enough.

On x86-64, the function below is used to measure the time:

```
static inline void
rdtsc(unsigned int *lo, unsigned int *hi)
{
    __asm__ __volatile__ ("rdtsc" : "=a"(*lo), "=d"(*hi));
}
```

Only after the benchmark loop exits are lo and hi composed to a single integer. hi is left-shifted 32 bits, and the values are than OR:ed.

RDTSC is the most precise time measurement facility available, however it requires that the process is constrained to one CPU, therefore, the process is set to only run on a preselected CPU.[28]

[28] https://www.strchr.com/performance_measurements_with_rdtsc

Optimisations

One of the most important generic optimisations is loop-unrolling.[29] One can easily be fooled into thinking that the more you unroll a loop, the faster it will be. From my experience with libzahl, I have found that either unrolling by 4 or not at all is optimal.[30] Because of this, libzahl always allocates 4 extra characters[31] to its integers, we call this ‘fluff’. This way, unrolled code does not need to care about the precise size of an integer, it can assume that it is a multiple of 4 characters and disregard that it may not start at the multiple of 4 characters into the character-array. This optimisation can almost always be done, it can be a problem if it stops before the end of the character-array. Of course, there is no reason to think 4 is always optimal. Start with 2 and unroll by 1 more until you find the optimal; don’t just unroll as much as possible.

GCC does not unroll loops by default with any of the -O flags. Of course if you want to do this optimisation of pretending that it is a multiple of 4 bytes, optimising compilers will not help you. But it is worth remembering that you cannot expect loop-unrolling to be done if your function is compiled in the user application.

Whilst it is understandable that the compiler does not know what you are doing, so you have to optimise non-trivial code yourself, it is notable that the compiler does not always know what it is doing. A simple case of this can be found in libzahl: the function `zswap` that swaps the members of two structure instances. It can be trivially implemented as

```
void zswap(z_t a, z_t b)
{ z_t t; *t = *a, *a = *b, *b = *t; }
```

This implementation is about a third as fast as the implementation found in libzahl. The problem is that the compiler does not compile `*x = *y` efficiently for `struct`:s. By swapping each member of the structure individually manually, the performance is improved by a factor of between 2.8 and 2.9. Further performance can be gained by converting the `struct` to a `long*` and swapping each element, effectively swapping padding space too.

In developing libzahl, I have also noticed that modern CPU:s are incredibly good at branching. If you have to choose between performing a simple

[29] http://www.csc.kth.se/utbildning/kth/kurser/DD2440/avalg11/dokument/gmp_slides.pdf

[30] The STOKE developers have probably noticed this too, they unroll by 4 in their whitepaper.

[31] Characters being defined as a 64-bit word.

operation, such as addition, or a conditional move, you should seriously consider a conditional move, and you should absolutely consider implementing the conditional move with a condition jump and a move instruction.