WhyMP, a Formally Verified Arbitrary-Precision Integer Library

Guillaume Melquiond, Raphaël Rieu-Helft

▶ **To cite this version:**

Guillaume Melquiond, Raphaël Rieu-Helft. WhyMP, a Formally Verified Arbitrary-Precision Integer Library. 2020. hal-03233220

HAL Id: hal-03233220
https://hal.inria.fr/hal-03233220

Preprint submitted on 24 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

WhyMP, a Formally Verified Arbitrary-Precision Integer Library

Guillaume Melquiond[a,*], Raphaël Rieu-Helft[b]

[a] *Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, 91190, France*
[b] *TrustInSoft, Paris, 75014, France*

Abstract

Arbitrary-precision integer libraries such as GMP are a critical building block of computer algebra systems. GMP provides state-of-the-art algorithms that are intricate enough to justify formal verification. In this paper, we present a C library that has been formally verified using the Why3 verification platform in about four person-years. This verification deals not only with safety, but with full functional correctness. It has been performed using a mixture of mechanically checked handwritten proofs and automated theorem proving. We have implemented and verified a nontrivial subset of GMP's algorithms, including their optimizations and intricacies. Our library provides the same interface as GMP and is almost as efficient for smaller inputs. We detail our verification methodology and the algorithms we have implemented, and include some benchmarks to compare our library with GMP.

1. Introduction

The GNU Multi-Precision library, or GMP for short, is a widely used arbitrary-precision arithmetic library implemented in C and assembly. It provides state-of-the art algorithms for basic arithmetic operations and number-theoretic primitives. It is used in computer algebra software, as well as safety-critical contexts such as cryptography and security of Internet applications.

GMP is extensively tested, but some parts of the code are visited with very low probablity, such as $1/2^{64}$. This makes random testing a poor way of ensuring GMP's correctness. Moreover, most of the algorithms are quite intricate, so finding bugs through manual inspection of the code is challenging. As a result, GMP has had its share of bugs.[1] We advocate using formal verification to ensure memory safety and the absence of correctness bugs for all inputs.

GMP features several layers, each one handling different kinds of numbers. The innermost one, *mpn*, handles natural numbers. The other three layers, *mpz*, *mpq*, and *mpf*, are mostly wrappers around *mpn* and handle relative numbers, rational numbers, and floating-point numbers respectively.

We have verified a subset of algorithms from the *mpn* and *mpz* layers of GMP using the Why3 verification platform using the following approach. We first implement the GMP algorithms in WhyML, the high-level specification and programming language that Why3 provides [1]. We also give them a formal specification based on GMP's documentation and our own understanding of the algorithms. Then, Why3 computes verification conditions that, once proved, guarantee that the WhyML functions are memory-safe and satisfy the specifications we provided. Using a collection of automated theorem provers, we check these verification conditions, thereby proving the functions correct. Finally, using Why3's extraction mechanism, we obtain an efficient and correct-by-construction C library that closely mirrors the original GMP code. We give more details on the verification process, guarantees, and caveats, in Section 2. The resulting C library, named WhyMP, can be found at

*Corresponding author

Email addresses: `guillaume.melquiond@inria.fr` (Guillaume Melquiond), `raphael.rieu-helft@trust-in-soft.com` (Raphaël Rieu-Helft)

[1] Look for "division" at `https://gmplib.org/gmp5.0.html`

https://gitlab.inria.fr/why3/whymp/

WhyMP is not a full implementation of the *mpn* and *mpz* layers of GMP. In particular, the *mpn* layer contains many algorithms for each basic operation, so that the optimal one can be used, depending on the size of the inputs. We have implemented and verified at least one algorithm for each of addition, subtraction, multiplication, division, square root, modular exponentiation, and base conversion (I/O). In most cases, we have verified only the algorithm best suited to smaller numbers (typically up to 1,000 bits). The *mpz* wrapper is also a work in progress. Moreover, while our algorithms attempt to mirror GMP's implementation closely, there are a few differences. We provide a detailed list of functions and differences between WhyMP and GMP in Section 3.

While WhyMP does not fully implement GMP's API, it is compatible with GMP. Indeed, the functions have the same signatures and specifications. Therefore, in a C program that uses GMP, it is possible to substitute the calls to GMP for calls to the corresponding WhyMP functions. Moreover, WhyMP is roughly performance-competitive with versions of GMP that do not use handwritten assembly. This is easily explained by the fact that WhyMP closely mirrors GMP's code. Most of the performance difference comes from a small number of very short critical primitives. Therefore, it should be possible to check them carefully and add them to the trusted code base to recoup most of the performance loss. We present a more detailed benchmark of various configurations of GMP and WhyMP in Section 4.

This work was originally presented at the ISSAC symposium [2]. This article greatly extends the description of the verification process (Section 2), as readers of JSC might not be familiar with deductive program verification. This article also provides more details about the implementation of our library (Section 3) as well as some additional benchmarks (Section 4).

2. Verification and trusted code base

With software developed in a traditional way, users expect bugs to be plentiful. Only careful code reviews, safety analysis tools and other memory sanitizers, and lots of testing, both automated and manual, will ultimately ensure that software is bug-free with a high level of confidence.

Here, we follow a different approach: formal verification, and more specifically, deductive program verification. First, we mathematically specify what the library functions are supposed to compute. Then, using Why3, we turn both the code and its specification into a large logical formula. Finally, we look for a proof of this formula using automated theorem provers. If we succeed, it means that the code is safe and that it behaves as documented by the specification. Sections 2.2 and 2.3 give an overview of what the specification process entails.

Once formally verified, users should be able to assume that the library is bug-free, as its correctness has been proved and this proof has been mechanically checked. In particular, Section 2.4 shows what users can expect from the verified functions. But as always, the devil resides in the details. So, Sections 2.5 and 2.6 carefully review all the hypotheses the correctness of our library depends on, in order to understand how large the user's leap of faith has to be.

2.1. Translation to WhyML

Adding a function to WhyMP starts with the conversion of the C code of a GMP function to WhyML. This manual process is mostly straightforward, as most C features used by GMP are mapped to our WhyML model of C.

Consider the macro `MPN_NORMALIZE` from GMP. It takes a pointer `DST` pointing to a sequence of `NLIMBS` memory words. Possibly, the rightmost words of the sequence are zero, which GMP functions usually cannot cope with. So, this macro decreases `NLIMBS` until the rightmostword is nonzero or until `NLIMBS` reaches zero. The code of the macro looks as follows:

```
#define MPN_NORMALIZE(DST, NLIMBS)        \
  do {                                    \
    while ((NLIMBS) > 0)                  \
      {                                   \
```

```
        if ((DST)[(NLIMBS) - 1] != 0)     \
          break;                          \
        (NLIMBS)--;                       \
      }                                   \
  } while (0)
```

This C macro is manually translated into the following WhyML function:

```
let normalize (dst: ptr limb) (ref nlimbs: int32) =
  while nlimbs > 0 do
    if get_ofs dst (nlimbs - 1) <> 0 then break;
    nlimbs <- nlimbs - 1;
  done
```

Since it is a function (WhyML has no macros), we have to specify the types of the arguments. Argument `dst` is a pointer to one or more memory words (a "limb" in GMP parlance) and `nlimbs` is a 32-bit signed integer. Since modifications to `nlimbs` are meant to be seen by the caller, it is marked with the `ref` keyword. The translation of the body of the macro is then straightforward.

This process is performed for all the macros and functions of interest from GMP. At the end, the extraction mechanism from Why3 will automatically convert the library written in WhyML back into a C library [3]. For `normalize`, this gives the following C code:

```
void normalize(uint64_t * dst, int32_t * nlimbs) {
  while (*nlimbs > 0) {
    if (!(dst[*nlimbs - 1] == UINT64_C(0))) {
      break;
    }
    *nlimbs = *nlimbs - 1;
  }
}
```

The main difference with the original C code is that `nlimbs` is now a pointer. This is unavoidable, since the C language does not allow passing arguments by reference, unlike C++ for example. Other than that, the obtained C code is quite close to the original code.

If we were worried about the overhead of using a function rather than a macro, we could annotate the `normalize` function with the attribute `[@extraction:inline]`, which instructs Why3 to inline the body of the function into each of its callers at extraction time.

2.2. Specification

At this point, nothing of interest has been achieved. In fact, we might even have introduced bugs when converting the C code of GMP to WhyML. So, the next step is to specify what the WhyML functions are expected to do.

The first two components of a specification are the *preconditions* and *postconditions*. Those are first-order formulas about the state of the program and the arguments of the function, as well as its result in the case of postconditions. Let us start with postconditions. They express the mathematical relation between the inputs and the outputs of a function.

In the case of the `normalize` function, one postcondition is that, at the end of the execution `nlimbs` is either 0 or the rightmost word is now nonzero. We express this postcondition using the following directive:

```
ensures { nlimbs = 0 \/
          (pelts dst)[dst.offset + nlimbs - 1] > 0 }
```

The `pelts` and `offset` symbols come from our memory model for C, which makes the formula a bit complicated. The `pelts` function takes a pointer and returns an array containing all the memory words reachable by pointer arithmetic. The `offset` field of a pointer tells which of these words the pointer currently points at.

The array returned by `pelts` is actually unbounded. It maps mathematical integers to limb values. While one can index arbitrary cells of this array in the specification, our memory model leaves the value of cells outside the reachable zone completely unspecified.

With a bit of syntactic sugar, the right-hand side of the postcondition could be written as `dst[nlimbs - 1] > 0`. This postcondition is not sufficient, though, as the following pointless function would satisfy it just as well:

```
let bad_normalize (dst: ptr limb) (ref nlimbs: int32)
  ensures { nlimbs = 0 \/ dst[nlimbs - 1] > 0 }
= nlimbs <- 0
```

So, we have to be a lot more precise. For example, we could state that the new value of `nlimbs` is nonnegative and no larger than its old value:

```
ensures { 0 <= nlimbs <= old nlimbs }
```

We could also state that all the memory words found between the new value of `nlimbs` and its old value are zero:

```
ensures { forall i. nlimbs <= i < old nlimbs -> dst[i] = 0 }
```

Moreover, Why3 infers from the body of the function that the memory words accessible from `dst` were not modified by `normalize`. So, the three postconditions above are sufficient to fully characterize the state of the program once the function returns.

But this characterization might not be the one the callers of the function are interested in, since it is too low level. They do not care that some memory words were zero. They want to be sure that the arbitrarily-precise integer represented by the pair (`dst`, `nlimbs`) was left unchanged by the function.

To do so, we first have to specify what is the mathematical integer represented by a pointer and a size. That is the goal of the `value` function, which delegates most of its work to the more general function `value_sub`. This function takes an unbounded map x of limbs and two indices $n \leq m$ and it recursively computes the following mathematical integer:

$$\texttt{value_sub}(x, n, m) = \sum_{i=n}^{m-1} x_i \beta^{i-n}.$$

These two functions are defined as follows in WhyML (with radix β being 2^{64}).

```
let rec ghost function
  value_sub (x: map int limb) (n m: int): int
  variant {m - n}
= if n < m then
    x[n] + radix * value_sub x (n+1) m
  else 0

function value (x: ptr limb) (sz: int): int =
  value_sub (pelts x) x.offset (x.offset + sz)
```

The `function` keyword states that these functions can be used in specifications. The `ghost` keyword states that they are ignored by the extraction mechanism [4]. (Function `value` is implicitly marked as `ghost`.) Indeed, notice the type `int` used for the return value of these functions. It is the type of mathematical integers, which means that these functions are performing arbitrarily-precise integer computations. So, we do not want them to have any concrete meaning, since we are in the process of writing an arbitrary-precision library.

The `variant` directive in the specification of `value_sub` explains to Why3 that, at every recursive call, the integer $m-n$ decreases yet it stays nonnegative. (Why3 will use external solvers to verify that it is indeed the case.) Thus, `value_sub` always terminates, which is mandatory for functions used in specifications.

Now that we have defined `value`, we can use it to give much a more useful specification to the `normalize` function:

```
let normalize (dst: ptr limb) (ref nlimbs: int32)
  ensures { 0 <= nlimbs <= old nlimbs }
  ensures { value dst nlimbs = value dst (old nlimbs) }
  ensures { nlimbs = 0 \/ dst[nlimbs - 1] > 0 }
= ...
```

In fact, there is no reason to stop there. We can add any postcondition that might be useful for a caller of the function, even if they are redundant with respect to the previous ones. For example, we can state that the represented integer is at least as large as $\beta^{\texttt{nlimbs}-1}$ when `nlimbs` is nonzero:

```
ensures { nlimbs = 0 \/
    value dst nlimbs >= power radix (nlimbs - 1) }
```

At this point, we could ask Why3 to verify that the `normalize` function satisfies its specification. Why3 will compute a first-order logical formula encoding this property and send it to external solvers for verification. Unfortunately, these solvers run seemingly forever, never answering whether the formula is correct. In fact, they will not even succeed to prove the part of the formula that states that the function is safe to execute. This should not come as a surprise, since there is absolutely no guarantee that `get_ofs dst (nlimbs - 1)` is a valid memory access. The accessed limb might be outside of the memory block. Or the block might already have been freed.

So, we have to add some preconditions to `normalize`. They state sufficient conditions for the function to behave safely and accordingly to the postconditions. Here, the preconditions are straightforward. We want `nlimbs` to be nonnegative and all the words in the memory range $[\texttt{dst}; \texttt{dst} + \texttt{nlimbs})$ to be accessible. The latter is expressed using the `valid` predicate of our memory model. This gives the following preconditions:

```
let normalize (dst: ptr limb) (ref nlimbs: int32)
  requires { nlimbs >= 0 }
  requires { valid dst nlimbs }
```

2.3. Verification

Now that we have given a seemingly correct contract to the `normalize` function, it is time to tackle the verification of its functional correctness. As mentioned earlier, Why3 automates part of this work by combining the body of the function and its specification into a first-order formula. If this formula can be proved (using external solvers), then Why3 guarantees that the function satisfies its specification.

Unfortunately, unless a loop can be unrolled, there is generally no obvious way of expressing its behavior with a first-order formula. In other words, loops are black boxes that prevent any kind of verification. So, the specification work does not stop after the contract of the function is stated, we also have to annotate loops. We do so by the way of *invariants*.

The only thing Why3 knows about the program state at the end of `normalize` is that the loop exited (either `nlimbs > 0` is false, or `break` was called) and that the invariant of the loop holds. So, the invariant needs to be strong enough, so that the specification of the function can be verified, yet it needs to be *inductive* so that it is preserved by a loop iteration. Lastly, it needs to hold at loop start. For `normalize`, there is no difficulty; but in general, finding proper invariants requires a deep understanding of why an algorithm works correctly [5].

First, we need the invariant to guarantee that `get_ofs dst (nlimbs - 1)` is safe to execute. To do so, we could annotate the loop with

```
invariant { valid dst nlimbs }
```

This invariant holds at the entry of the loop, since it is a precondition of the function. It is also preserved by the body of loop, since the value of `nlimbs` at the end of an iteration is still nonnegative and it is no larger than at the start of the iteration. So, this is indeed an inductive invariant.

But there is a better invariant. Indeed, remember that we eventually want to verify that the postcondition `0 <= nlimbs <= old nlimbs` holds. So, we might just as well choose an invariant that both guarantees the safety of memory accesses and ensures this postcondition. The following invariant does.

```
invariant { 0 <= nlimbs <= nlimbs at Start }
```

`Start` is a *label* that we put at the beginning of the function, so that we can refer not only to the current value of the variables but also to their initial value. Notice how similar the invariant is to the postcondition. For a function such as `normalize` which contains a single loop, this is expected. This points us at another invariant that we need in order to prove the whole specification of the function. It states that the mathematical integer represented by the pair $(\texttt{dst}, \texttt{nlimbs})$ is left unchanged despite modifying `nlimbs`:

```
invariant { value dst nlimbs = value dst nlimbs at Start }
```

Why3 does not require programs to terminate, as long as explicitly stated. So, we could just annotate the function with the `diverges` keyword and be done with it. But for the sake of completeness, let us also prove that the function always terminates. To do so, we have to annotate the loop with a *variant*, as we already did for the `value_sub` function. (Note that, in general, variants do not tell anything about the time complexity of a function.) This time, the decreasing nonnegative integer is simply `nlimbs`:

```
variant { nlimbs }
```

At this point, if we ask Why3 to verify the correctness of `normalize`, it almost succeeds. Indeed, the external solvers are able to verify the logical formula that states that all the memory accesses are safe, and most postconditions are proved to hold when the function returns. Indeed, the first two postconditions are trivial consequences of the invariants. The third postcondition is inferred from the way the loop terminates. Unfortunately, the external solvers run seemingly forever (or consume all the available memory) when they try to tackle the last postcondition:

```
ensures { nlimbs = 0 \/ value dst nlimbs >= power radix (nlimbs - 1) }
```

That is where the verification process becomes intricate. Indeed, as far as we can tell, this failing postcondition is a corollary of the third one:

```
ensures { nlimbs = 0 \/ dst[nlimbs - 1] > 0 }
```

Unfortunately, there is no magic. Even if a logical formula holds, there is no guarantee that a solver will succeed in proving it. So, even if we have eliminated all the bugs from a function and we have correctly specified it (including invariants), the verification might still fail. At this point, we have to start some kind of debugging process, but at the conceptual and logical levels.

Remember that we have defined the `value_sub` function in WhyML using the following equation:

$$\texttt{value_sub}(x, n, m) = x_i + \beta \cdot \texttt{value_sub}(x, n+1, m) \qquad \text{if } n < m.$$

In other words, this is a Horner scheme. Thus, the behavior of the function with respect to the value of the least significant limbs is obvious. But the behavior regarding the most significant ones requires a proof by induction, which solvers are notoriously bad at.

So, we have to perform a formal proof by hand. To do so, we define a *lemma* function. This is a *ghost* function that does not compute and return anything of interest. Its only purpose is its specification: If its preconditions hold, so do its postconditions.

```
let rec lemma value_sub_tail (x: map int limb) (n m: int)
  requires { n <= m }
  variant  { m - n }
  ensures  { value_sub x n (m + 1) =
      value_sub x n m + x[m] * power radix (m - n) }
= if n < m then value_sub_tail x (n + 1) m
```

As for program functions, Why3 combines the body of the function and its specification into a first-order formula and sends it to external solvers for verification. This time, they succeed in proving the formula. Indeed, since the lemma function is recursive, the precondition and postcondition we just wrote act as an induction hypothesis. So, the solvers no longer need to perform a proof by induction; they just have to verify that the induction hypothesis is preserved.

From the above lemma about `value_sub`, we derive its corollary about `value`, again expressed as a lemma function:

```
let lemma value_tail (x: ptr limb) (sz: int32)
  requires { 0 <= sz }
  ensures  { value x (sz + 1) =
    value x sz + (pelts x)[x.offset + sz] * power radix sz }
= value_sub_tail (pelts x) x.offset (x.offset + sz)
```

```
let normalize (dst: ptr limb) (ref nlimbs: int32)
  requires { nlimbs >= 0 }
  requires { valid dst nlimbs }
  ensures  { 0 <= nlimbs <= old nlimbs }
  ensures  { value dst nlimbs = value dst (old nlimbs) }
  ensures  { nlimbs = 0 \/ (pelts dst)[offset dst + nlimbs - 1] > 0 }
  ensures  { nlimbs = 0 \/ value dst nlimbs >= power radix (nlimbs - 1) }
= label Start in
  while nlimbs > 0 do
    invariant { value dst nlimbs = value dst nlimbs at Start }
    invariant { 0 <= nlimbs <= nlimbs at Start }
    variant   { nlimbs }
    if get_ofs dst (nlimbs - 1) <> 0
    then begin
      value_tail dst (nlimbs - 1);
      break
    end;
    nlimbs <- nlimbs - 1;
  done
```

Figure 1: Code and specification of `normalize`, which discards the rightmost zero limbs of a number to ensure that the most significant limb is nonzero.

Now, we just have to tell Why3 to make use of it. To do so, we insert a call to this lemma function right before the `break` statement of `normalize`. Indeed, at this point, we know that the most significant limb is nonzero thus positive. So, external solvers have no trouble to infer the last postcondition of `normalize` from the postcondition of `value_tail`.

Figure 1 shows what the complete code and specification of `normalize` look like. The failure of the solvers to fully verify its functional correctness is not specific to `normalize`. Even a non-linear term in a verification condition can cause solvers to get lost in their search for a proof. In that case, we need to define lemma functions or to annotate the WhyML code with *assertions*. Assertions make for a larger verification condition, but it can be split into lots of smaller ones, which hopefully are more agreeable to automated provers. For example, a property as simple as $\forall x, y, z \in \mathbb{Z},\ y > 0 \Rightarrow (x \cdot y + z) \div y = x + z \div y$ might require about 10 user assertions before automated theorem provers succeed.

Unfortunately, verification conditions related to the correctness of GMP-like libraries are full of non-linearity. In fact, even the representation of an integer from its limbs is already awfully non-linear: $a = \sum_i a_i \beta^i$. As a consequence, a lot of time is spent annotating the code with extra assertions. To alleviate this issue, we even implemented and formally verified our own decision procedure dedicated to proving this kind of arithmetic facts [6].

2.4. Postconditions and library guarantees

The functional correctness of the library states the adequacy between the code of the library and its specification. Thus, if the pre- and postconditions are incorrect, even if the code has been formally verified, it might still be unfit for any meaningful usage. So, let us see how bad this can get. The second postcondition of `normalize`, *i.e.*,

```
ensures { value dst nlimbs = value dst (old nlimbs) }
```

states that the number represented by (`dst`, `nlimbs`) stays the same, despite modifying its length `nlimbs`. This postcondition is a bit too simple to be representative of most functions of a GMP-like library. So, let us have a look at the `mpz_add` function of GMP.

Unlike `normalize`, which was a low-level function to which the location of a number was given by a pointer and a length, functions from the *mpz* layer of GMP represent numbers using opaque handles of type `mpz_ptr` (also known as `mpz_t`). The `mpz_add` function takes three handles and stores the sum of the numbers represented by the last two handles into the first handle. Here is the WhyML signature of this function, as well as its first postcondition:

```
let wmpz_add (w u v: mpz_ptr): unit
  ensures { value_of w mpz =
            old (value_of u mpz + value_of v mpz) }
```

In the postcondition above, `mpz` is a global variable that keeps track of all the handles that currently represent numbers in memory. This variable is ghost, *i.e.*, only visible from the specification; it has no existence in the code of the function and is erased from the generated code [4]. A term (`value_of` x `mpz`) designates the mathematical integer represented by the handle x. As with the other specification formulas, the plus operator denotes the mathematical integer addition. Thus, the postcondition states that, when the function returns, the integer represented by `w` is the sum of the two integers that were represented by `u` and `v` at the entry of the function. So, there is no difficulty for the user to trust that the function actually performs an addition. The situation is similar for all the functions of the library, as they can always be described by a simple relation between mathematical integers represented by inputs and outputs.

This postcondition of `wmpz_add` is not sufficient, though. Indeed, it states what number is represented by `w` at the end of the execution, but it does not state anything about all the other handles. Perhaps it has modified the numbers represented by `u` and `v` (which it does, if they happen to be aliased to `w`). Or perhaps it has modified some other number not even passed as argument. Thus, the specification of `wmpz_add` has another postcondition, which states that no number except for `w` has been modified by the function:

```
ensures { forall x. x <> w -> mpz_unchanged x mpz (old mpz) }
```

It might seem easy to forget such a postcondition, but in practice, it hardly happens. Indeed, if it was missing or too weak, it would be impossible to verify any nontrivial code that calls this function, so this kind of mistake gets detected early.

2.5. Preconditions and library assumptions

For preconditions, the situation is a bit more subtle. They have to be as weak as possible, as the program state has to satisfy the preconditions of a function before it can be executed. This has been verified for all the function calls inside WhyMP. But once the library has been turned into C functions, nothing prevents the user from calling them with bad arguments or in an inconsistent state. Since the functions are not programmed in a defensive fashion, they might then fail in unpredictable ways. The same is true of GMP's functions, unless the debugging profile is enabled at compile time.

For `normalize`, the precondition states that the pointer `dst` is valid for accessing a zone large enough for `nlimbs` limbs. Similarly, for `wmpz_add`, there is a validity precondition on the handles used as inputs and outputs. There is nothing surprising about these preconditions; GMP functions have the same requirements, and they are mentioned in GMP's documentation. For example, it is undefined behavior to call GMP's `mpz_add` on a handle that was not yet initialized using `mpz_init` or already freed using `mpz_clear`. The same is true of its translation to WhyMP, *i.e.*, `wmpz_add`.

The header file `wmp.h` of WhyMP states all the additional preconditions of our library. They are all related to aliased pointers, so only functions from the *mpn* layer are impacted. Here is an example of such a function:

```
// not verified when rp and np are aliased.
wmp_size_t wmpn_sqrtrem
  (wmp_ptr sp, wmp_ptr rp, wmp_srcptr np, wmp_size_t);
```

This function computes the square root of a number, as well as the remainder. GMP's documentation states that this function is expected to work fine, even if the remainder is output at the exact same location as the input number. This is not the case for its counterpart from WhyMP, as indicated by the comment above. That does not mean that a program that passes aliased pointers to `wmpn_sqrtrem` will crash. (That would be extremely unlikely, since our code mirrors GMP's.) It just means that we have not yet verified that it does not.

Since GMP's `mpz_ptr` type abstracts the notion of pointer away, the WhyMP functions from the *mpz* layer do not suffer from these formal deficiencies. They have no extraneous preconditions; they all behave in accordance with GMP's documentation. In particular, they support aliased arguments.

There is a last assumption regarding our library. While the specifications do not tell anything about the time or space complexity of the functions, they have something to say about their termination. Indeed, by default, Why3 implicitly enforces *total* correctness, that is, assuming that the preconditions hold, the functions terminate and their outputs satisfy the postconditions. In the case of our library, all the functions have thus been formally proved to return, but under the assumption that the program does not run out of resources. More precisely, given valid inputs, functions from our library either return correct results, or they abruptly terminate the program because of a heap overflow, or they signal a stack overflow, *e.g.*, by a segmentation fault, as would GMP. Theoretically, nothing prevents WhyMP from exhausting resources much faster than GMP in some situations. But the code of both libraries is similar enough to deem that issue unlikely.

2.6. Trusted code base

As we have seen, the user has to understand the mathematical specification of the library (which is not much different from understanding its documentation), but at no point does the user need to understand the code in any way. Yet, to have confidence in the library, the user still needs to trust several other components.

First, given the WhyML code of the library as well as its specification, Why3 generates a set of verification conditions by a calculus of weakest precondition [7]. So, the user has to trust that Why3 has performed this computation correctly, that is, if all the verification conditions hold, then the specification adequately describes the behavior of the library. Why3 is a generic verification platform that has been used in numerous occasions, and the calculus of weakest precondition is a well-known approach to program verification, so this is not the component the user should worry about.

The verification conditions produced by Why3 are first-order formulas, which are often too complicated to be proved by a human. So, Why3 dispatches them to automated theorem provers, either SMT solvers or superposition-based provers [8]. The user needs to trust that the verification conditions were properly converted to the input languages of the provers and that the provers did not succeed in proving an incorrect theorem. For our library, we use the SMT solvers Alt-Ergo, CVC3, CVC4, and Z3, and the superposition-based E prover. All these theorem provers are off-the-shelf tools that are widely used. Unfortunately, that does not make them bug-free, so the usual approach to increase the confidence in a WhyML development is to consider a verification condition to be proved only if several provers agree on it.

At this point, the user should be confident that the library is correct. But this is still WhyML code; it has to be converted to C code. Why3 is responsible for the extraction to C and the user needs to trust that none of the meaningful properties of the WhyML code were lost in the translation to C. There are three parts to it. The first one is the translation itself: WhyML constructs should be translated to C constructs. To increase the confidence, we have kept this translation as simple as possible. In particular, we did not try to convert any high-level feature of WhyML, such as automatic memory management or higher-order functions. As a consequence, the translation from WhyML to C is mostly syntactic [3].

The second part is the model of the C language we have implemented in WhyML. For example, we have defined an abstract type `ptr` to represent C pointers, as well as some abstract functions to read and write the pointed location. These functions have specifications that require the pointers to be valid and ensure that the memory is consistent, *e.g.*, reading a valid memory location after writing to it gives back the written value. All these WhyML functions are then mapped to C functions or operators. So, the user needs to trust that our specifications of these functions properly model the semantics of the C language. We had to improve the memory model, as its original version was not expressive enough to support aliasing, which is critical for some functions [9].

The third part is composed of the arithmetic primitives used by our library. Indeed, WhyMP heavily relies on the availability of a multiplication of one limb by one limb returning two limbs, and conversely, of a division of two limbs by one limb. As with the memory model, these primitives are defined as abstract functions in WhyML and are mapped to handwritten C functions. Fortunately, most of those functions are trivial, as we rely on 128-bit support from C compilers. For example, here is the primitive for division:

```
uint64_t div64_2by1
    (uint64_t ul, uint64_t uh, uint64_t d)
{ return (((uint128_t)uh << 64) | ul) / d; }
```

The most complicated primitive is the one used to compute an 8-bit approximation of the square root for any integer between 128 and 511. GMP implements it as a plain array of integer literals, but due to a technical limitation of Why3, we cannot express this array in WhyML yet. So, we have performed the verification of the primitive outside Why3 [10].

At this point, we have a C library that satisfies a meaningful specification. The last step is to compile and link it. So, the user also needs to trust that the compiler will not perform an incorrect optimization that would mess with the C code. This is quite a leap of faith, but no larger than the one needed when compiling any C library out there. A potential solution to alleviate this issue would be to use a formally verified C compiler such as CompCert, which guarantees that any behavior of the generated assembly code is an allowed behavior of the original C code [11].

3. Verified algorithms

Each WhyMP function comes at a significant cost in terms of time and proof effort, so only a subset of GMP's functions have been implemented. Moreover, while we strive to mirror GMP's algorithms as closely as possible, some differences remain. Some of these differences are due to time constraints, others come from technical limitations of the Why3 platform. Finally, some GMP functions are specialized according to whether the hardware provides some non-standard primitives natively. In these cases, we only considered the most generic version of the algorithm, that is, the one for which no specific primitive is provided by the hardware. Let us review WhyMP's algorithms and the differences between GMP and WhyMP. More details on the algorithms themselves can be found in previous work [5, 9].

3.1. Addition, subtraction

The algorithms for addition and subtraction in GMP are the schoolbook ones, and they are reproduced identically in WhyMP. However, almost all *mpn* addition and subtraction functions allow parameter overlap, which results in an unfortunately large amount of almost-identical variants of the addition and subtraction algorithms in WhyMP. Since these functions are very commonly used, we have tweaked the memory model in order to be able to prove generic versions of these functions that allow overlapping parameters. In the end, the exported versions of addition and subtraction can be called by external users in the same way as their GMP counterparts. However, they cannot always be called internally, due to limitations in Why3's type system, so the library still has a large amount of addition and subtraction variants for internal use.

3.2. Multiplication

GMP features more than ten different multiplication algorithms, which are each called when the size of the operands makes them optimal. These algorithms can be split into three categories: the schoolbook algorithm, suited for smaller numbers, Toom-Cook variants, and finally Schönhage-Strassen multiplication, which is only used on very large numbers (about 300,000 bits on the computer we used for benchmarks[2]).

GMP's Toom-Cook variants have names of the form `toom_`xy, with $x \geq y \geq 2$. They start by splitting their larger operand into x parts and the smaller into y parts. All these parts need to have roughly equal length, so the ratio between the lengths of the operands should be roughly x to y. In addition, the asymptotic complexity decreases when y grows, so for larger numbers, variants that split operands into more parts should be called.

For a specific range of number sizes, GMP performs multiplication by calling `toom_22` and `toom_32` depending on the relative sizes of the operands. We have implemented and verified these two functions. Thus, WhyMP's multiplication is comparable with GMP's one until the inputs reach the threshold where GMP starts using `toom_33` (about 4,200 bits on our benchmark computer). For very unbalanced operands, GMP provides a wrapper that first splits the larger one into many smaller segments, and then calls `toom_42` on each one. As we have not verified `toom_42`, the wrapper we have implemented calls `toom_32` instead. As will be seen in the benchmarks, this has little impact on performance.

[2]The thresholds are determined upon configuration of GMP and depend on the architecture.

Finally, GMP features an optimized function for squaring integers (`mpn_sqr`), as well as a specialized function that computes only the lower half of a product (`mpn_mullo`). None of them are implemented in WhyMP yet, so it always uses the general multiplication (`mpn_mul`) instead.

3.3. Division

There are two main division algorithms in GMP: a so-called schoolbook algorithm, and a subquadratic divide-and-conquer algorithm. The schoolbook algorithm is far from trivial. For example, each candidate quotient digit is obtained by dividing 3 limbs by 2 limbs [12], using a precomputed pseudo-inverse of the two most significant limbs of the divisor. This 3-by-2 division is more costly than the usual 2-by-1 division but greatly reduces the number of subsequent adjustment steps. WhyMP implements this algorithm faithfully.

GMP's schoolbook algorithm, however, uses an entirely different algorithm when the size of the divisor is more than half that of the numerator. The goal is for the complexity to depend only on the size of the quotient, rather than that of the divisor. WhyMP does not implement this variant, so it always uses the first one instead, irrespective of the size of the inputs. The performance disparity between both schoolbook variants is hardly noticeable when the quotient exceeds three limbs. But when the quotient is only one limb long, the variant used by GMP can be twice as fast.

Finally, WhyMP does not implement the divide-and-conquer algorithm either, which is used by GMP for dividing inputs larger than about 3,200 bits on our benchmark computer.

3.4. Square root

WhyMP implements the same divide-and-conquer square root algorithm as GMP. The general case (three limbs or more) has been formally verified by Bertot *et al* using the Coq proof assistant [13]. Although our own verification of this algorithm was performed using a different tool, it is largely inspired by theirs.

The most intricate part of GMP's divide-and-conquer square root algorithm is by far the base case, which computes the square root of a 64-bit number. It is best understood as a fixed-point arithmetic algorithm that implements Newton's method, although it only manipulates integers. Starting from a precomputed 8-bit approximation of the square root, it performs only two Newton iterations and a fast adjustment to compute the square root. The fact that only two iterations are needed in all cases is due to the use of carefully chosen magical constants. We were able to verify the base case algorithm by using a custom-made WhyML model of fixed-point integers [10] and sending the resulting Why3 verification conditions to the Gappa tool.

3.5. Modular exponentiation

GMP features a fast modular exponentiation algorithm (`mpn_powm`) that implements the sliding-window method and uses Montgomery reduction so that only one division is needed in the whole computation. We have implemented and verified the same algorithm in WhyMP. Once again, the main performance difference comes from the algorithm's dependencies. Indeed, modular exponentiation involves a lot of squaring, so the lack of a dedicated squaring function hurts WhyMP's performance.

Moreover, GMP switches from the Montgomery reduction algorithm that is implemented in WhyMP (`redc_1`) to another algorithm (`redc_n`) when computing modulo numbers larger than 3,500 bits on our benchmark computer. The `redc_n` algorithm makes use of `mpn_mullo` to compute the lower half of a product. Comments in GMP's source code indicate that this is about 20% faster than computing a regular product and ignoring the upper half.

GMP also features a variant `mpn_sec_powm` of the modular exponentiation that is designed to be side-channel secure. More precisely, its control flow and memory accesses do not depend on the values of the operands. We have also verified this function, but it relies on a side-channel secure division, whose verification is still a work in progress. As a result, WhyMP's side-channel resistant modular exponentiation is not usable yet. Moreover, the formal verification of this function only offers guarantees on its correctness. We currently do not have any good way to prove that it is indeed side-channel resistant.

3.6. Base conversions and I/O

GMP's I/O functions include divide-and-conquer algorithms that translate a large number into a string that represents it in an arbitrary base and vice versa. These algorithms are surprisingly intricate, especially the optimized ones that convert from/to base 10. Starting from about 750 bits on our benchmark computer, divide-and-conquer algorithms are used by GMP. Due to time constraints, we have chosen to instead verify the schoolbook base conversion algorithms from Mini-GMP, the standalone version of GMP that is distributed alongside it. These algorithms are simpler, and we expect that I/O is usually not the bottleneck in computations on large numbers.

Much like in GMP, bases from 2 to 62 are supported. Negative bases (down to -36) are also supported. They are used to encode whether the letters that represent digits greater than 9 in the target base should be uppercase or lowercase. For negative bases, digits and uppercase letters are used, while for bases between 2 and 36, digits and lowercase letters are used. (For bases 37 to 62, digits, uppercase letters, and lowercase letters are all used). Our verified algorithms do not support whitespace in the input or base detection based on the prefix.

3.7. The mpz *layer*

The *mpz* layer is a wrapper around the *mpn* functions that takes care of number signs and storage. Most users of GMP interact only with this layer, so that they do not have to manually manage memory allocations. Functions of the *mpz* layer typically do not perform any computation themselves. Instead, they call the corresponding *mpn* function and handle the various cases required depending on the signs and lengths of the operands.

For each arithmetic operation, there can be several *mpz* functions that each handle various cases, even though they all rely on the same *mpn* function. For example, there are about twenty division functions in GMP, so that the best one can be used depending on whether the quotient, the remainder or both are needed, whether the divisor is a machine integer or a large integer, and the rounding mode. While this is the most extreme example, the *mpz* layer does represent a large amount of work. In the end, WhyMP's *mpz* layer is still largely a work in progress. It exports about 25 functions, which cover memory management, the four basic operations, comparisons, logical shifts, and base conversions. A full list of supported functions is shown in Fig. 2.

```
mpz_init        mpz_abs
mpz_clear       mpz_add
mpz_realloc2    mpz_add_ui
mpz_set_ui      mpz_sub
mpz_set_si      mpz_sub_ui
mpz_get_ui      mpz_ui_sub
mpz_set_str     mpz_neg
mpz_get_str     mpz_mul
mpz_cmp         mpz_mul_si
mpz_cmp_ui      mpz_mul_ui
mpz_cmp_si      mpz_mul_2exp
mpz_cmpabs      mpz_tdiv_q_2exp
mpz_cmpabs_ui   mpz_tdiv_qr
mpz_sgn         mpz_tdiv_qr_ui
```

Figure 2: Full list of supported *mpz* functions

As stated in Section 2.5, WhyMP's *mpz* functions do not have any extra aliasing-related preconditions compared to their GMP counterparts. They follow GMP's documentation with only one exception: the `mpz_get_str` function, which converts an *mpz* number to a string in an arbitrary base. Indeed, GMP's function accepts a null pointer as destination, in which case it automatically allocates a suitably sized string first. WhyMP's implementation does not support this use case; the pointer has to be valid.

3.8. Compatibility concerns

The signatures of the WhyML functions of our library are such that the generated C functions have the exact same signature as GMP functions. Numbers are also represented the same way in memory, that is, *mpn* numbers are pointers to an array of limbs stored from least significant to most significant, while *mpz* numbers are a record whose third field is a pointer to an array of limbs. In total, about fifty functions from the GMP API are exported by WhyMP.

All the functions and types exported by WhyMP's `wmp.h` header are prefixed by the letter `w`, so that they do not conflict with GMP's. The library also offers a header `wmp-gmp.h`, which imports GMP's header `gmp.h` but substitutes WhyMP's functions to GMP's ones whenever available. As a consequence, the minimal change needed to use WhyMP is to modify the preprocessor directive `#include <gmp.h>` and to pass `-lwmp` at link time.

There are two potential sources of incompatibility when using WhyMP in place of GMP, as our library lacks a bit of genericity. First, WhyMP numbers are made of 64-bit limbs only, while GMP can be compiled to use 32-bit limbs instead. In particular, functions from the *mpn* layer make strong assumptions on the alignment of pointers and the size of numbers. Thus, user code meant to be used only with a 32-bit GMP cannot be used with WhyMP (nor can it be used with a 64-bit GMP). However, if the user code only calls high-level functions from the *mpz* layer, then this incompatibility does not matter, as the size of the numbers is abstracted away by the opaque type `mpz_t`.

Second, the user can call `mp_set_memory_functions` to instruct GMP to use some custom handler instead of `malloc` for allocating memory for numbers. Our library does not support this feature. It is thus incompatible with user applications that do not want to use `malloc`.

4. Benchmarks

The next sections show how GMP and WhyMP compare on several benchmarks: multiplication, square root, a primality test, base conversion, and division. These benchmarks exercise three variants of GMP and three variants of WhyMP. Indeed, the direct comparison between GMP and WhyMP is not that meaningful, as GMP relies on native assembly routines. So, in addition to the timings of WhyMP and GMP, four other timings are measured to give a better view of the performance.

First, GMP is also compiled without support for assembly, which means that only the generic C code is compiled. GMP without assembly and WhyMP are not exactly in the same ballpark though, since they do not use the same primitive operations for doing a $64 \times 64 \to 128$ multiplication and a 128-by-64 division. Indeed, in assembly-free GMP, these are implemented in C using only 64-bit operations, while WhyMP delegates these operations to the 128-bit support of the C compiler.

Second, to measure the impact of these two primitives, WhyMP is also compiled in a way such that their 128-bit implementation is replaced by the 64-bit one from GMP without assembly.

Third, the timings of Mini-GMP are measured. Mini-GMP is a C library distributed along GMP. It is "*intended for applications which need arithmetic on numbers larger than a machine word, but which don't need to handle very large numbers very efficiently*." For the two primitives above, Mini-GMP uses the same kind of implementation as the assembly-free version of GMP, that is, it uses only 64-bit operations.

Finally, for the third variant of WhyMP, some low-level *mpn* functions are replaced by their GMP counterparts, as these functions are typically written in assembly. Those functions are `add_n` (resp. `sub_n`), which computes the sum (resp. difference) of equally-sized *mpn* numbers; `add` and `sub`, for *mpn* numbers with different sizes; `mul_1`, which multiplies an *mpn* number by a single limb; `addmul_1` (resp. `addmul_2`), which multiplies an *mpn* number by a single limb (resp. a two-limb number), and then accumulates the product into the destination; and `submul_1`, which accumulates the opposite of the product. Note that we could have replaced a lot more functions of WhyMP by their assembly counterparts from GMP, including rather complicated ones, *e.g.*, division by two-limb numbers. Instead, we chose to focus on a few simple functions, so as to not blow the trusted code base out of proportions, which would defeat the point of formally verifying an arithmetic library.

The version of GMP is 6.1.2. The benchmarks are executed on an Intel Xeon E5-2450 at 2.50 GHz with 64GB of memory. All the libraries are compiled by GCC 10.2.0 using the options selected by GMP at configuration time, *i.e.*, "`-O2 -march=sandybridge -mtune=sandybridge -fomit-frame-pointer`".

Figures 3, 4, 5, 6, and 7 show the timings obtained on the various benchmarks. On every figure, abscissas are the number of 64-bit limbs, while ordinates are the time in microseconds. All the figures except Figure 7 are in log-log scale, so that the asymptotic complexity is apparent. Performance-wise, the general ordering of the plots is the same on every benchmark: GMP is the fastest, then comes WhyMP with some assembly primitives from GMP, then WhyMP, then GMP without assembly primitives, then WhyMP without 128-bit primitives, and finally Mini-GMP is the slowest.

4.1. Multiplication

The first benchmark (Figure 3) simply tests multiplication (`mpn_mul`) for various sizes of *mpn* numbers, so as to exercise both the base-case multiplication as well as Toom-Cook algorithms. Two cases are tested: equal-sized inputs, and $n \times 24n$ unbalanced inputs.

The unbalanced case tests the algorithmic differences between WhyMP and GMP. Indeed, WhyMP performs 16 calls to `toom_32`, which results in 64 $\frac{n}{2} \times \frac{n}{2}$ multiplications, while GMP performs 12 calls to `toom_42`, which results in 60 $\frac{n}{2} \times \frac{n}{2}$ multiplications. Due to the extra cost of interpolation for `toom_42`, WhyMP hardly suffers from not having `toom_42` at this level of unbalance.

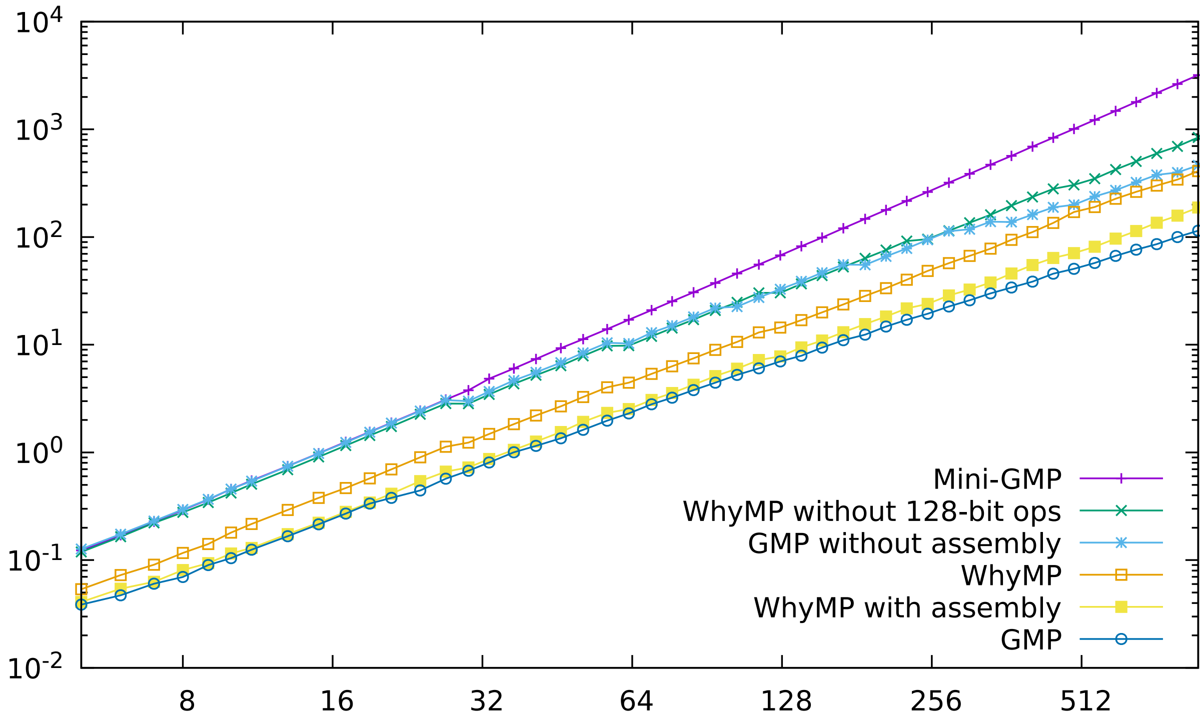
The plots of Mini-GMP, WhyMP without 128-bit support, and GMP without assembly, make it apparent when the libraries switch to asymptotically faster algorithms. Mini-GMP sticks with the quadratic schoolbook algorithm, while WhyMP and GMP switch to `toom_22` around $n = 30$, and then GMP switches to `toom_33` around $n = 60$. Starting around $n = 170$ (`toom_44` for GMP), the lack of higher variants of Toom-Cook in WhyMP becomes noticeable, as the library becomes progressively slower with respect to GMP. For $n \leq 170$, WhyMP is at most twice as slow as GMP, and when replacing the primitive operations with the assembly ones from GMP, the slowdown does not exceed 20%. The smaller n is, the smaller the slowdown, down to about 5% for $n \leq 20$.

4.2. Square root

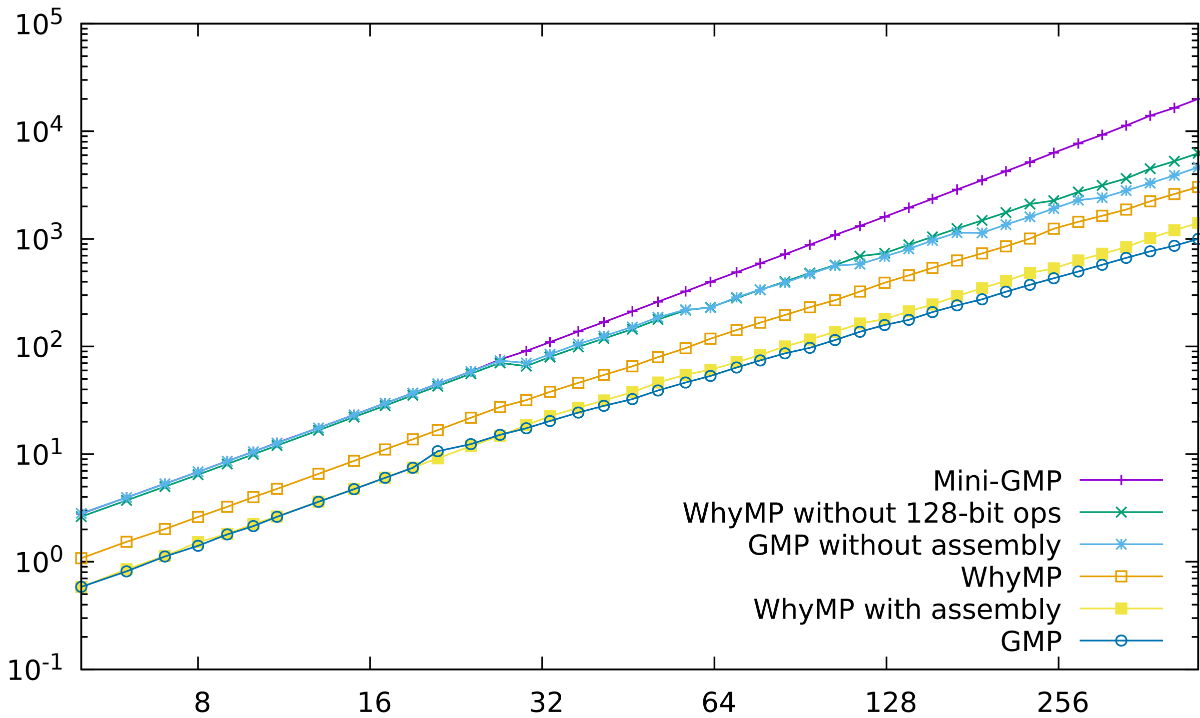
The second benchmark (Figure 4) tests `mpn_sqrtrem` for various sizes of *mpn* numbers. Each iteration of GMP's divide-and-conquer square root algorithm performs a long division, so WhyMP greatly suffers from featuring only the schoolbook division, despite using the same square root algorithm as GMP. Indeed, around $n = 50$, GMP switches to the divide-and-conquer division, which WhyMP does not implement. This makes WhyMP with assembly about 50% slower than GMP for $n \leq 600$. Without assembly, WhyMP is twice as slow for $n \leq 90$, and thrice as slow for $n \leq 600$. As for Mini-GMP, its poor performance (up to $\times 150$ times slower for $n \leq 600$) can be explained by the use of a converging sequence $y_{n+1} = (x/y_n + y_n)/2$, rather than a dedicated algorithm.

4.3. Miller-Rabin's primality test

The third benchmark (Figure 5) implements Miller-Rabin's primality test for number sizes commonly encountered in cryptography applications. This is a simple implementation inspired by GMP's one. It exercises the *mpz* layer as well as the modular exponentiation. When the candidate prime is not immediately rejected, a large majority of the time is spent computing modular exponentiations. Note that the modular exponentiation used in WhyMP's benchmarks is just a wrapper over `mpn_powm`, so it supports neither even modulos nor negative exponents, unlike `mpz_powm`. WhyMP is 110% slower than GMP for $n \leq 28$, and 140% slower for $n \leq 60$. With assembly primitives, the slowdown is less than 30% for $n \leq 60$. At $n = 65$, GMP switches to the Toom-3 multiplication algorithm. As WhyMP does not feature this algorithm, it starts lagging further behind at this point.



(a) Multiplication $n \times n$



(b) Multiplication $n \times 24n$

Figure 3: Timings for balanced and unbalanced multiplication.

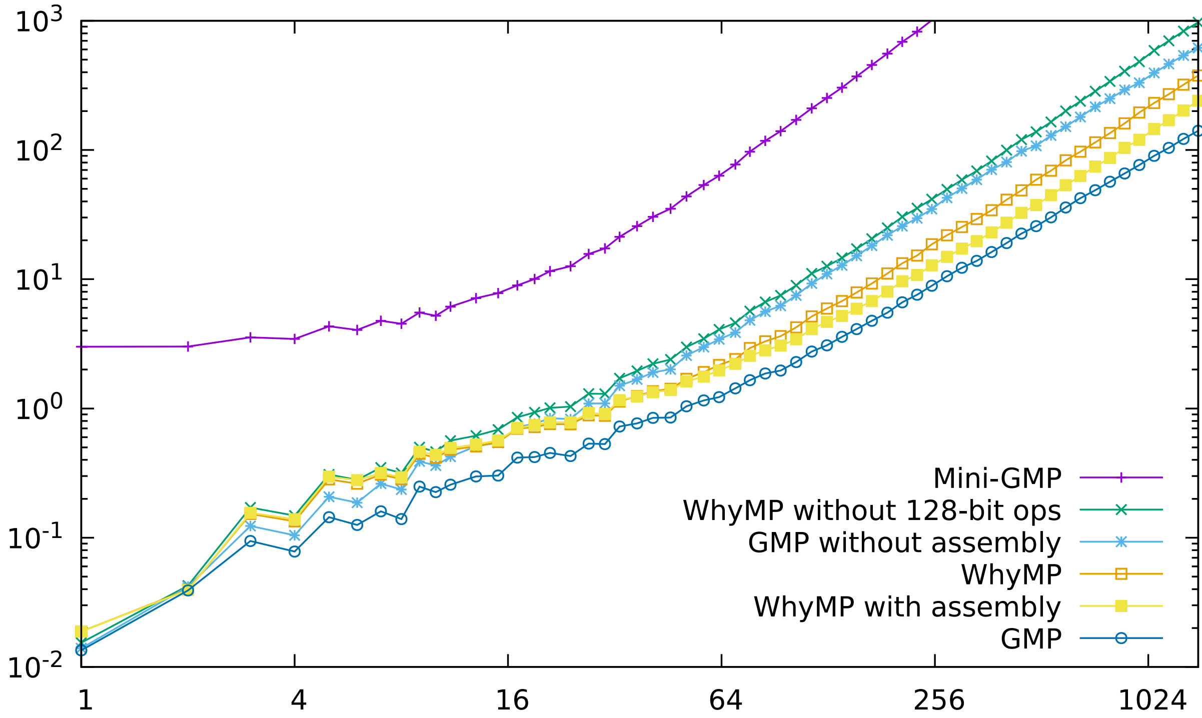


Figure 4: Timing for square root.

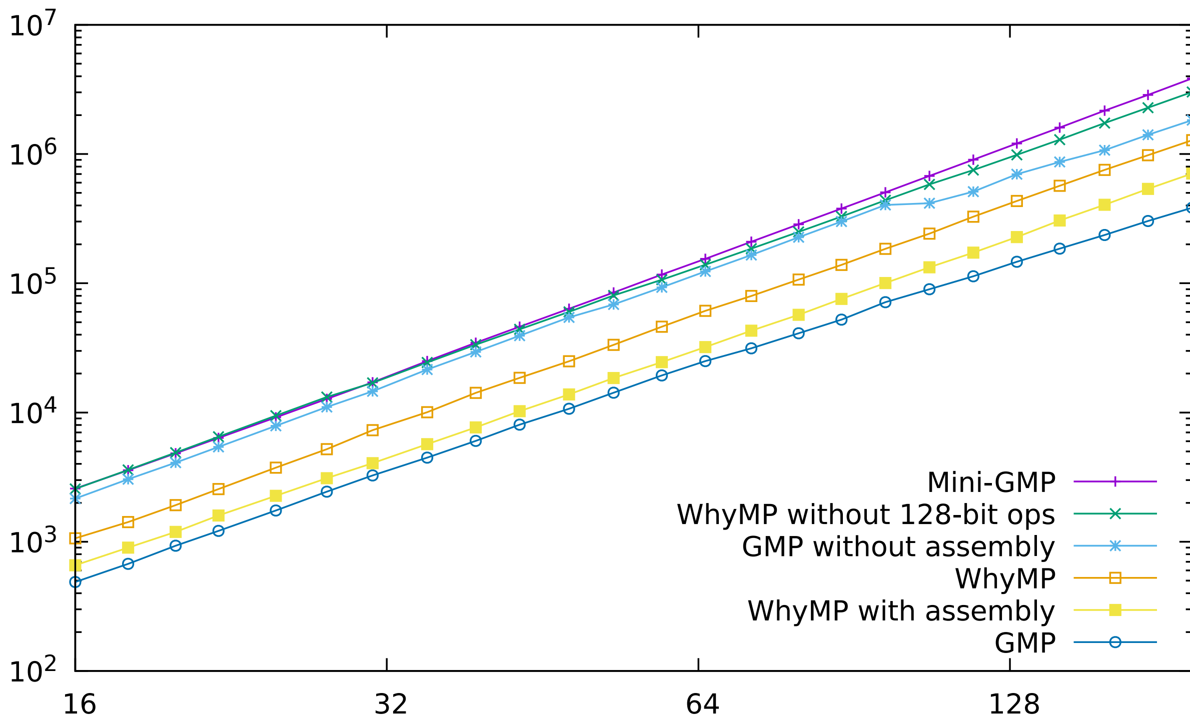


Figure 5: Timing for Miller-Rabin's primality test.

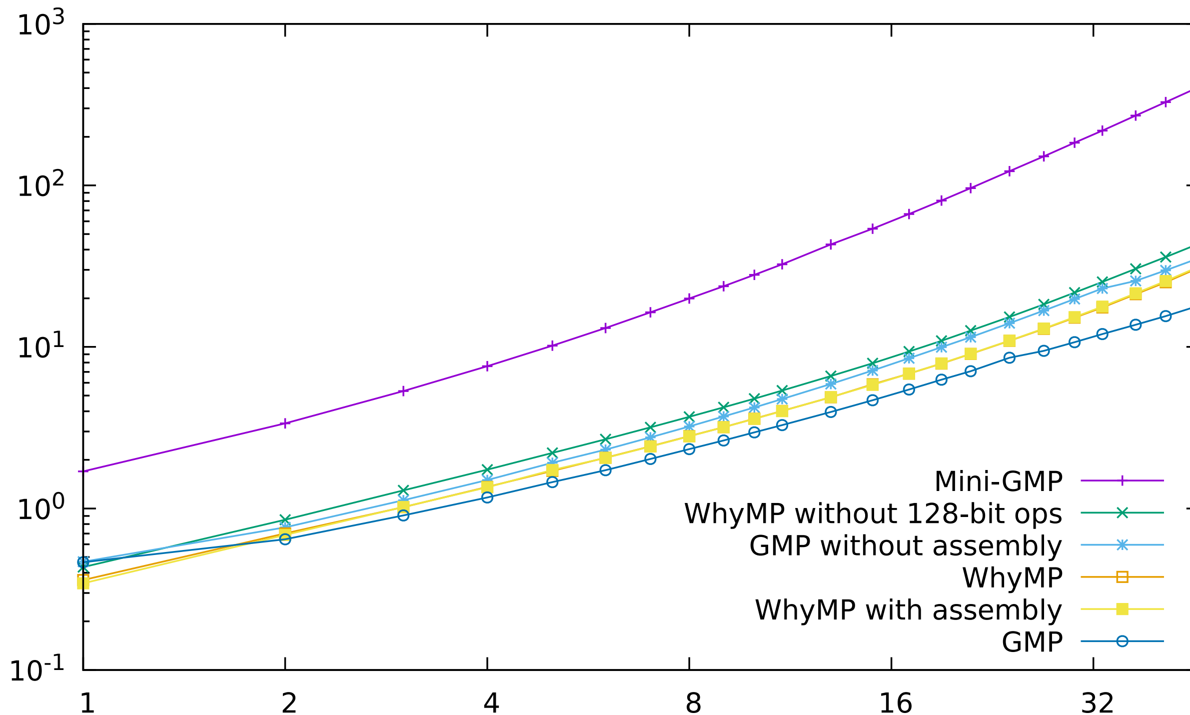


Figure 6: Timing for base conversion.

4.4. Base conversion

The fourth benchmark (Figure 6) tests base conversions by doing a roundtrip through `mpz_get_str` and `mpz_set_str`. It converts a n-limb number into a character string and then converts it back to a number. Bases 10, 16, and 27, are tested. Indeed, different algorithms are available depending on whether the base is 10, a power of two, or some generic value. Only GMP provides dedicated algorithms for base 10; WhyMP and Mini-GMP fall back to the generic ones. Moreover, they both use schoolbook algorithms for base conversions, while GMP relies on a divide-and-conquer algorithm starting from about 750 bits, that is, $n = 12$.

Despite using the same algorithms as Mini-GMP, WhyMP is about ten times faster. Part of this difference can be explained by the usage of `malloc` and `free` by Mini-GMP, while GMP and WhyMP perform their temporary allocations directly on the stack. This only explains half of the overhead though; a careful comparison of the implementations of WhyMP and Mini-GMP should be performed to understand what causes the remaining overhead. Another noticeable result is that WhyMP happens to be faster than GMP for one-limb numbers. Finally, unlike the other benchmarks, replacing the core routines of WhyMP with those of GMP does not bring any speedup, which explains why the "WhyMP" and "WhyMP with assembly" plots cannot be distinguished.

4.5. Division

The last benchmark (Figure 7) exercises the division on small numbers. This benchmark is slightly different from the previous ones. Indeed, instead of showing just the average runtime of the various libraries, it shows the zone covered by most of the measurements, *i.e.*, the width of every curve is twice the standard deviation. Moreover, the scale is no longer log-log. For readability, only the default versions of every library are shown: GMP with assembly primitives, WhyMP with 128-bit primitives, Mini-GMP.

The benchmark divides 20-limb numbers by n-limb numbers. On such small numbers, the three libraries use the schoolbook algorithm for division. They all use the same algorithm for divisors between 1 and 10

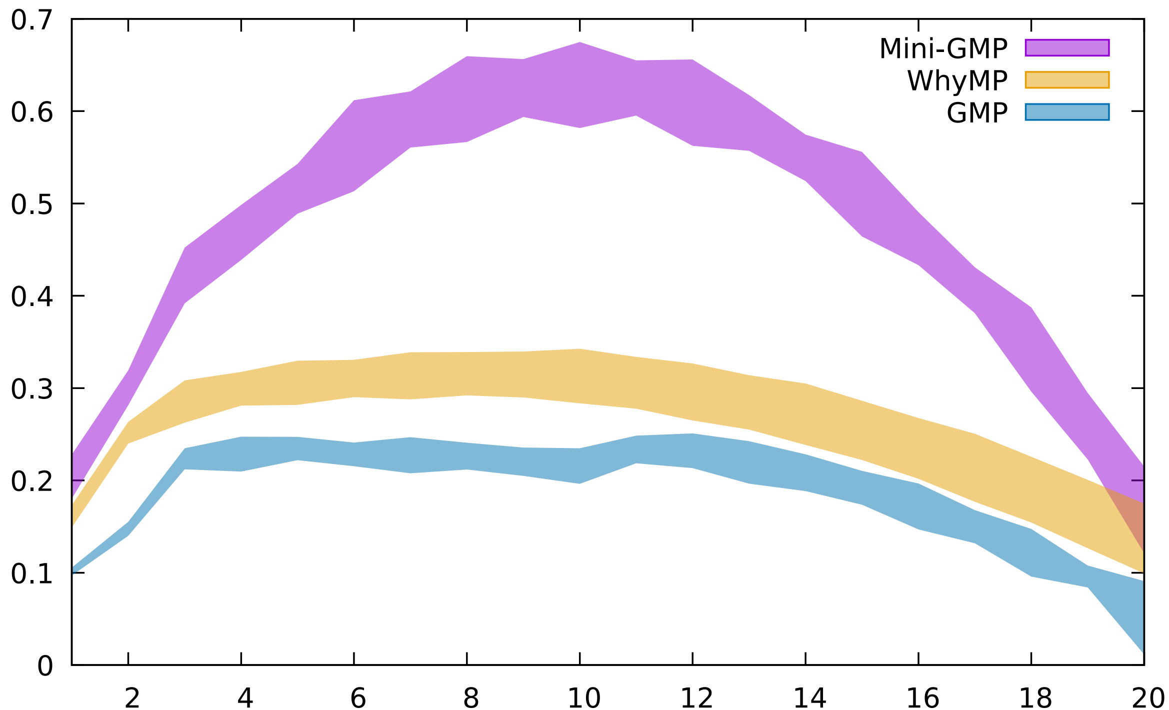


Figure 7: Timing for division of 20 limbs by n limbs.

limbs, and GMP uses a different one when the size of the divisor is more than half that of the numerator. The parabolic shape of the curves can be explained by the complexity of the algorithm, which varies like $D \cdot (N - D)$ with N and D the sizes of the numerator and divisor. The high variance partly comes from the normalization at the start of the algorithm, which shifts the inputs such that the most significant bit of the divisor is set. This makes the division algorithm much faster on inputs that are already normalized than on those that are not. Note that most other algorithms (*e.g.*, square root, base conversion) make sure that divisors are always normalized, so they do not experience this variance.

4.6. Evaluation

Overall, three factors have a large impact on performance: the asymptotic complexity of the algorithms, the specialization of algorithms for specific inputs, and the quality of the underlying arithmetic primitives. Asymptotically, the most important factor is of course the complexity of the algorithms, as can be seen on the multiplication benchmarks. On large numbers, WhyMP's performance falls behind even that of the assembly-free version of GMP when the latter switches to more efficient algorithms.

For smaller numbers, the multiplication algorithms are similar enough, thus making the primitives the deciding factor. This is especially noticeable, as the plots for "GMP without assembly" and "WhyMP without 128-bit ops" are indistinguishable till 100 limbs, despite WhyMP not implementing `toom_42` and `toom_33`. To a lesser extent, this is also true for the Miller-Rabin benchmark, for numbers between 30 and 50 limbs, despite the lack of `mpn_sqr`.

For the square root benchmark, the lack of a squaring function and of a division algorithm specialized for small quotients is, however, especially noticeable. This is shown by the constant relative overhead for intermediate sizes, between 30 and 300 limbs.

As for the plots "GMP" and "WhyMP with assembly", they are indistinguishable on the multiplication benchmarks for smaller numbers, which again shows that the algorithms are similar enough. For the Miller-Rabin benchmark, the constant relative distance between both plots is mostly due to the fact that we

used only a very small subset of GMP's assembly primitives. For instance, we did not use GMP's assembly implementation of Montgomery's reduction (`mpn_redc_1`). We did not use GMP's assembly implementation of division either (`mpn_divrem_1`), as noticeable on the base conversion benchmark. This again shows the impact of the underlying arithmetic primitives.

Overall, what we can conclude from these benchmarks is that on inputs small enough to trigger those of GMP algorithms that are also provided by WhyMP, our implementation is close enough to the original that most of the perfomance difference comes from the primitives in handwritten assembly.

5. Related work

Thanks to the Why3 tool [8, 1, 14], we have developed a formally verified arbitrary-precision integer arithmetic library that closely mirrors GMP. As far as we know, this work is the first to have comparable performance to the state of the art. Indeed, previous work generally does not deal with a large number of highly optimized algorithms. Let us discuss a few examples of existing verifications of arithmetic libraries.

Bertot *et al* verified GMP's square root general case algorithm [13] using the Coq proof assistant. Our Why3 proof of that algorithm is directly inspired by their article. Their formalization is rather similar to ours, but their proof effort is even larger, as Why3 proofs are partially automated in a way Coq proofs are not.

Myreen and Curello verified an arbitrary-precision integer arithmetic library [15] using the HOL4 theorem prover. Their work covers the four basic arithmetic operations, but not the square root or modular exponentiation. They did not attempt to produce highly efficient code. However, their verification goes all the way down to x86 machine code, using formally verified compilers and decompilers. They also managed to automate most of the proofs involving pointer reasoning, despite using an interactive tool.

Affeldt used Coq to verify a binary extended GCD algorithm implemented in a variant of MIPS assembly [16], as well as the basic arithmetic functions the algorithm depends on. While the implemented algorithm is simpler than GMP's, it nonetheless uses GMP's number representation. The author verified an implementation of the algorithm in a pseudo-code language and proved that the pseudocode correctly simulates the MIPS assembly code, using a memory model based on separation logic.

Fischer verified a modular exponentiation library [17] using the Isabelle/HOL proof assistant and a framework for verifying imperative programs developed by Schirmer [18]. The library is not meant to be efficient. For example, it represents arbitrary-precision integers as garbage-collected doubly-linked lists of machine integers. The author reported running into issues inside the tool due to the large number of invariants and conditions needed to keep track of aliasing. This is exactly the kind of issue that we avoid by forcing function parameters separation, at the cost of some expressivity.

Berghofer used Isabelle/HOL to develop a verified library programmed in the SPARK fragment of the Ada programming language [19]. The library provides modular exponentiation as well as the primitives required to implement it. The modular exponentiation algorithm is a simple square-and-multiply one, without the sliding-window optimization or the Montgomery reduction that are featured in GMP and WhyMP. However, the proof effort (2,000 lines of Isabelle written over three weeks) is surprisingly low.

Schoolderman used Why3 to verify hand-optimized Karatsuba multiplication assembly routines for the AVR architecture [20]. The verified algorithms do not support an arbitrary precision. Instead there are many routines, each specialized for a particular operand size up to 96×96 bits. This allows the loops to be unrolled, so the algorithms are branch-free, which makes the proofs much easier for SMT solvers.

Zinzindohoué *et al* developed a formally verified cryptography library written in F* and extracted to C [21]. It implements the full NaCl API, and includes a bignum library. The extracted code is as fast as state-of-the-art C implementations, and part of it is now deployed in the Mozilla Firefox web browser. Their approach is very similar to ours in that it consists in verifying the algorithms in a high-level language suited for verification, and then compiling them to C. The integers have a small, fixed size that depends on the choice of elliptic curve. Again, the fact that the number sizes are known makes the problem much easier for automated solvers. As a result, their proof enjoys a higher degree of automation than ours. Thus, while their specifications are similar or larger in length, their code requires much fewer annotations than ours.

Finally, Erbsen *et al* used Coq to produce Fiat-Crypto [22], a verified C elliptic curve cryptography library that is much faster than GMP's generic C functions. In order to facilitate the verification, the algorithms are first implemented as high-level, parameter-agnostic templates in continuation passing style. A certified compilation scheme, written in Coq's functional programming language Gallina, specializes these templates for each of the many supported prime fields and produces constant-time straight-line C code that is similar to what an expert implementer may write. Several of these implementations are used in BoringSSL, an SSL library currently used in the Chrome web browser.

6. Conclusion

WhyMP is an arbitrary-precision integer library. It has been developed, specified, and annotated, using the WhyML language. Its correctness has been formally verified using Why3 and several external theorem provers, mostly SMT ones. The formal verification includes the functional correctness, *i.e.*, the relations between function inputs and outputs match the definition of the corresponding mathematical operators. The memory representation of integers is identical to GMP's, and the functions have the same signatures, which makes WhyMP a potential substitute to GMP.

The compatibility with GMP is not limited to the interface. The library also implements the vast majority of the state-of-the-art tricks found in GMP, as it was implemented after a detailed analysis of its code. This makes it competitive with GMP in some specific cases. For instance, WhyMP is much faster than the pure C variant of GMP. GMP however implements numerous finely tuned assembly routines, which makes WhyMP twice as slow as the standard GMP. By importing a few multiply-and-accumulate primitives from GMP, WhyMP can be brought closer, making it only 5% to 20% slower, depending on the operation.

In the process of verifying WhyMP, we found one bug in the comparison function of GMP[3] that occurs for very large inputs (several gigabytes). This is exactly the sort of bug that is easy to find using formal methods, but hard to test against effectively. Our work also influenced the development of GMP in another way. Our correctness proof for the divide-and-conquer multiplication ended up being so intricate (much more than what GMP's developers thought) that they preferred to modify the code, so that its correctness became more obvious.

This does not mean that GMP is now formally verified, although our work increases further the (already high) confidence in its correctness. To the best of our knowledge, GMP's macro-heavy C code mixed with assembly is completely out of reach of any existing verification framework, due to the combinatorial explosion that arises from all the possible architectures and compilation options. If one really wanted to tackle a formal verification at the level of the C code, Mini-GMP would make a much more sensible target. Still, it would require a large proof development on the mathematical side, though smaller than ours, as Mini-GMP's algorithms are much simpler than GMP's and ours.

The development of WhyMP was a large undertaking. The proof effort totals about 22,000 lines of WhyML code over four person-years. A breakdown of the proof effort per operation can be found in Figure 8. The WhyML code is part of Why3's gallery of examples of verified programs:

`http://toccata.lri.fr/gallery/multiprecision.en.html`

Among the 22,000 lines of code in the WhyMP sources, about 8,000 are program code, and the remaining 14,000 are made up of specifications and assertions. This ratio is rather inefficient as Why3 proofs go. Indeed, many proofs of complex arithmetic facts had to be performed using many explicit proof hints in the form of very long assertions that were sometimes over a hundred lines long. While the resulting verification process can be said to be automatic, the WhyML code was so heavily annotated that it can almost be seen as a machine-checked pen-and-paper proof of GMP's algorithms. Some related works were much more successful in actually performing an automatic verification. But it was only for functions on fixed-size integers, as their loops are fully unrollable during verification. This is certainly not the case of GMP.

[3] `https://gmplib.org/list-archives/gmp-bugs/2020-February/004733.html`

comparison	100
addition	1000
subtraction	1000
mul (naïve)	700
mul (Toom)	2400
division	4500
helper lemmas	300
reflection	1700
logical shifts	1000
square root	1600
exponentiation	1700
base conversions	2000
mpz layer	3600
utilities	200
misc. lemmas	200

Figure 8: Proof effort in lines of code per WhyMP operation.

In an effort to increase proof automation, we have developed a framework for proofs by reflection inside Why3 and used it to automate the proof of a certain class of assertions [6]. Indeed, many of our handwritten proof annotations could be replaced by a call to a decision procedure for solving systems of linear equalities with a well-chosen coefficient set. However, further scaling up the use of this reflection framework proved difficult, as there was no longer an obvious class of similar-enough assertions that presented a good target for automation. A longer-term prospect could be to turn this reflection framework into a tactic system for Why3, which could then be used to automate proofs with more granularity than full-fledged decision procedures.

As a consequence of the constant fight to get the external solvers to automatically prove WhyMP's correctness, formally verifying functions currently consumes too much time. This explains why our library provides few variants of the functions yet, despite a proof effort of four person-years. Still, we intend to add at least a divide-and-conquer division, so that WhyMP can tackle slightly larger numbers, not only during division, but also square root and modular exponentiation. WhyMP also needs some side-channel resistant functions, so that modular exponentiation can be used in security-sensitive cryptography applications. Finally, we should investigate how to verify assembly code for some mainstream instruction sets, so as to close the performance gap with GMP.

References

[1] J.-C. Filliâtre, A. Paskevich, Why3 — where programs meet provers, in: 22nd European Symposium on Programming, Vol. 7792 of Lecture Notes in Computer Science, Heidelberg, Germany, 2013, pp. 125–128.

[2] G. Melquiond, R. Rieu-Helft, WhyMP, a formally verified arbitrary-precision integer library, in: 45th International Symposium on Symbolic and Algebraic Computation, Kalamata, Greece, 2020, pp. 352–359. `doi:10.1145/3373207.3404029`.

[3] R. Rieu-Helft, C. Marché, G. Melquiond, How to get an efficient yet verified arbitrary-precision integer library, in: 9th Working Conference on Verified Software: Theories, Tools, and Experiments, Vol. 10712 of Lecture Notes in Computer Science, Heidelberg, Germany, 2017, pp. 84–101. `doi:10.1007/978-3-319-72308-2_6`.

[4] J.-C. Filliâtre, L. Gondelman, A. Paskevich, The spirit of ghost code, Formal Methods in System Design 48 (3) (2016) 152–174. `doi:10.1007/s10703-016-0243-x`.

[5] R. Rieu-Helft, A Why3 proof of GMP algorithms, Journal of Formalized Reasoning 12 (1) (2019) 53–97. `doi:10.6092/issn.1972-5787/9730`.

[6] G. Melquiond, R. Rieu-Helft, A Why3 framework for reflection proofs and its application to GMP's algorithms, in: 9th International Joint Conference on Automated Reasoning, Vol. 10900 of Lecture Notes in Computer Science, Oxford, United Kingdom, 2018, pp. 178–193. `doi:10.1007/978-3-319-94205-6_13`.

[7] R. W. Floyd, Assigning meanings to programs, in: Program Verification, Springer, 1993, pp. 65–81.

[8] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, Why3: Shepherd your herd of provers, in: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, 2011, pp. 53–64.
URL `https://hal.inria.fr/hal-00790310`

[9] R. Rieu-Helft, Development and verification of arbitrary-precision integer arithmetic libraries, Ph.d. thesis, Université Paris-Saclay (Nov. 2020).
URL https://tel.archives-ouvertes.fr/tel-03032942

[10] G. Melquiond, R. Rieu-Helft, Formal verification of a state-of-the-art integer square root, in: IEEE 26th Symposium on Computer Arithmetic, Kyoto, Japan, 2019.
URL https://hal.inria.fr/hal-02092970

[11] X. Leroy, Formal verification of a realistic compiler, Communications of the ACM 52 (7) (2009) 107–115. doi:10.1145/1538788.1538814.

[12] N. Moller, T. Granlund, Improved division by invariant integers, IEEE Transactions on Computers 60 (2) (2011) 165–175. doi:10.1109/TC.2010.143.

[13] Y. Bertot, N. Magaud, P. Zimmermann, A proof of GMP square root, Journal of Automated Reasoning 29 (3-4) (2002) 225–252. doi:10.1023/A:1021987403425.

[14] J.-C. Filliâtre, One logic to use them all, in: 24th International Conference on Automated Deduction, Vol. 7898 of Lecture Notes in Artificial Intelligence, Lake Placid, USA, 2013, pp. 1–20. doi:10.1007/978-3-642-38574-2_1.

[15] M. O. Myreen, G. Curello, Proof pearl: A verified bignum implementation in x86-64 machine code, in: 3rd International Conference on Certified Programs and Proofs, Vol. 8307 of Lecture Notes in Computer Science, Melbourne, Australia, 2013, pp. 66–81. doi:10.1007/978-3-319-03545-1_5.

[16] R. Affeldt, On construction of a library of formally verified low-level arithmetic functions, Innovations in Systems and Software Engineering 9 (2) (2013) 59–77. doi:10.1007/s11334-013-0195-x.

[17] S. Fischer, Formal verification of a big integer library, in: DATE Workshop on Dependable Software Systems, 2008.
URL http://www-wjp.cs.uni-sb.de/publikationen/Fi08DATE.pdf

[18] N. Schirmer, A verification environment for sequential imperative programs in Isabelle/HOL, in: International Conference on Logic for Programming Artificial Intelligence and Reasoning, 2005, pp. 398–414. doi:10.1007/978-3-540-32275-7_26.

[19] S. Berghofer, Verification of dependable software using SPARK and Isabelle, in: 6th International Workshop on Systems Software Verification, Vol. 24 of OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany, 2012, pp. 15–31. doi:10.4230/OASIcs.SSV.2011.15.

[20] M. Schoolderman, Verifying branch-free assembly code in Why3, in: 9th Working Conference on Verified Software: Theories, Tools, and Experiments, Vol. 10712 of Lecture Notes in Computer Science, 2017, pp. 66–83. doi:10.1007/978-3-319-72308-2_5.

[21] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, B. Beurdouche, HACL*: A verified modern cryptographic library, in: ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 2017, pp. 1789–1806. doi:10.1145/3133956.3134043.

[22] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, A. Chlipala, Simple high-level code for cryptographic arithmetic - with proofs, without compromises, in: IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2019, pp. 1202–1219. doi:10.1109/SP.2019.00005.