WhyMP, a Formally Verified Arbitrary-Precision Integer Library

GUILLAUME MELQUIOND, Université Paris-Saclay, CNRS, Inria, LRI, France
RAPHAËL RIEU-HELFT, TrustInSoft, France and Université Paris-Saclay, CNRS, Inria, LRI, France

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.

CCS Concepts: • **Mathematics of computing → Mathematical software**; • **Theory of computation → Hoare logic**; • **Computing methodologies → Exact arithmetic algorithms**.

ACM Reference Format:
Guillaume Melquiond and Raphaël Rieu-Helft. 2020. WhyMP, a Formally Verified Arbitrary-Precision Integer Library. In *International Symposium on Symbolic and Algebraic Computation (ISSAC '20), July 20–23, 2020, Kalamata, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373207.3404029

1 INTRODUCTION

The GNU Multi-Precision library, or GMP, 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 such, 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 [7]. 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

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

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

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

WhyMP is not a full implementation of `mpn` and `mpz`. In particular, `mpn` 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.

2 VERIFICATION AND TCB

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 2.1 and 2.2 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. But as always, the devil resides in the details. So, Sections 2.3 and 2.4 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 Specifications

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. The next step is to provide a specification for the function. Let us consider the `mpn_copyi(r,x,n)` function as an example. It copies n limbs of an `mpn` number starting at the memory location pointed by x to a number starting at r.

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. For mpn_copyi, there is an obvious postcondition: when the function returns, the limbs stored in memory in the range $[r; r+n)$ have the same value as the limbs that were stored in memory in the range $[x; x+n)$ when the function started.

This postcondition is not sufficient, though, as it does not state anything about other memory locations. An mpn_copyi implementation that would mess with all the other limbs in memory would satisfy the postcondition but would be meaningless. So, the function has a second postcondition, which states that the limbs stored outside the range $[x; x+n)$ have the same values they had at start.

Preconditions state sufficient conditions for a function to behave safely and accordingly to the postconditions. In the case of mpn_copyi, the precondition simply states that accessing the memory ranges $[x; x+n)$ and $[r; r+n)$ is safe.

With other verification tools, there would usually be a second precondition that states that the ranges $[x; x+n)$ and $[r; r+n)$ do not overlap. In the case of Why3, the type system ensures that x and r point into separate memory blocks. More precisely, Why3 statically rejects any program that can possibly perform a logical data race, that is, when there are two potentially aliased pointers and one at least is used for writing. So, when introducing new pointers, the user has to explicitly tell Why3 their relation to existing ones.

That is it for the part of specification dedicated to the functional correctness of a program. For verification purpose, we also need to provide *variants* and *invariants*. A variant is an element from a well-founded ordering, *e.g.*, a positive integer, whose decrease ensures that loops and recursive calls terminate. For instance, for divide-and-conquer algorithms, the decreasing value is generally the length of the integers passed as arguments. Note that, in general, variants do not tell anything about the time complexity of a function.

Unless a loop can be unrolled, there is generally no obvious way of turning its behavior into a first-order formula. That is why it is critical to annotate loops with invariants. In fact, for Why3, the body of a loop is a black box. The only thing the tool knows about the program state after a loop is that the loop exited and that its invariant 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, and it needs to hold at loop start. So, finding proper invariants requires a deep understanding of why an algorithm works correctly.

We now have all the pieces to specify the WhyML code corresponding to mpn_copyi:

```
let wmpn_copyi (r x: ptr uint64) (n: int32): unit
  requires { valid x n /\ valid r n }
  ensures  { forall i. 0 <= i < n -> r[i] = x[i] }
  ensures  { forall i. i<0\/n<=i -> r[i]=old r[i] }
= let ref i = 0 in
  while (Int32.(<) i n) do
    variant { n - i }
    invariant {forall j. 0 <= j < i -> r[j] = x[j]}
    invariant {forall j. j<0\/i<=j -> r[j]=old r[j]}
    r[i] <- x[i];
    i <- i + 1;
  done
```

2.2 Proof effort

From the code and its specification, Why3 computes a verification condition, which is a first-order formula. If the code is safe and if its specification, including invariants, is adequate, the verification condition holds. So, in an ideal world, we would just have to submit it to an automated theorem prover and it would answer that it found a proof.

While this is true for a function as simple as `mpn_copyi`, automated theorem provers either give up or time out, in practice. In particular, whenever a verification condition contains non-linear terms, provers tend to get lost in their search for a proof. In that case, the user needs to annotate the WhyML code with assertions. This makes for a larger verification condition, but it can now 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 these kinds of arithmetic facts [10].

Once all the verification conditions are proved, we use Why3 to extract a C library from the WhyML code [15].

2.3 Guarantees and assumptions

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.

First, let us consider postconditions. The second postcondition of `wmpn_copyi` states that the function only touches some specific parts of memory. It might seem easy to forget this 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.

As for the first postcondition of `wmpn_copyi`, it states what the function actually performs. Unfortunately, it is a bit too simple and not quite representative of most functions of a GMP-like library. So, let us have a look at the WhyML signature of `mpz_add` as well as its first postcondition. This function takes three `mpz` numbers and stores the sum of the last two into the first one.

```
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 `mpz` 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 [6]. The expression `(value_of` x `mpz)` designates the mathematical integer represented in memory by some `mpz` number x. The plus operator is the mathematical integer addition and it has no computational content. 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.

For preconditions, the situation is a bit more subtle. They have to be as weak as possible, as the callers of a function have to make sure that the program state satisfies its preconditions before calling it. That is true of all 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. As the functions are not programmed in a defensive fashion, they might then fail in unpredictable ways.

For `wmpn_copyi`, the precondition states that the input pointers are valid for accessing a large enough zone. Similarly, for `wmpz_add`, there is a validity precondition on the `mpz` numbers used as inputs and outputs. There is nothing surprising about these preconditions; GMP functions have the same requirements.

As mentioned earlier, pointer-manipulating functions also have some implicit preconditions about the aliasing of pointers. In particular, the `wmpn_copyi` function cannot be called if the ranges $[r; r+n)$ and $[x; x+n)$ overlap. Yet, GMP documentation of `mpn_copyi` implicitly states that overlap is allowed, as long as $r < x$.[2] That does not mean that `wmpn_copyi` will behave badly when passed aliased pointers; it just means that the formal verification does not cover this case. So, the function, as presented above, is not a perfect replacement for `mpn_copyi`.[3]

The header file `wmp.h` states all the preconditions that are not present in GMP's documentation. They are all related to aliased pointers, so only functions from the `mpn` layer are impacted. Since `mpz` numbers abstract the notion of pointer away, the WhyMP functions from the `mpz` layer do not suffer from these formal deficiencies. They have no hidden preconditions; they all behave in accordance with GMP's documentation.

The last point regarding correctness is termination. 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 function terminates and the outputs satisfy the postconditions. In the case of our library, all the functions have thus been formally proved to return, but under the assumptions 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.

2.4 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 [9]. 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 [4]. 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

[2]The "i" in `mpn_copyi` actually means "increasing".

[3]Contrarily to the simplified version presented in this paper, the full specification of `wmpn_copyi` in WhyMP permits aliased pointers, just as GMP.

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 [15].

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 [15] was not expressive enough to support aliasing, which is critical for some functions.

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. This is implemented as a plain array of integer literals, but due to a technical limitation of Why3, we cannot express this in WhyML. So, we have performed the verification outside Why3 [11].

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.

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 of GMP functions are specialized according to whether the hardware provides some non-standard primitives natively. In these cases, we only considered the "generic" version of the algorithm, that is, the one where no particular primitive is expected to be 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 [14].

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 they are optimal depending on the sizes of the operands. 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 500,000 bits on x86_64). 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`. 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 for multiplication is changed slightly to call `toom_32` instead. As will be seen in the benchmarks, this has little impact on performances.

Finally, GMP features a specialized function for squaring integers faster than with the general multiplication. It is not yet implemented in WhyMP.

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 computed using a 3-limb by 2-limb division algorithm [12], using a precomputed pseudo-inverse of the top two 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. However, GMP's schoolbook algorithm uses an entirely different algorithm when the length of the denominator is more than half that of the numerator. The goal is for the complexity to depend only on the size of the quotient. WhyMP does not implement this second algorithm. The performance disparity becomes significant when the denominator is very close to the numerator in length. Moreover, WhyMP does not implement divide-and-conquer division.

3.4 Square root

GMP implements a divide-and-conquer square root, with a very intricate base case that uses precomputed 8-bit approximations and performs only two Newton iterations and a fast adjustment to compute the square root of a 64-bit number. WhyMP implements the exact same square-root algorithms as GMP [11]. However, the complexity of the square root is dominated by that of the long division. Therefore, WhyMP's square root is quadratic (like the division) whereas GMP's is subquadratic thanks to the divide-and-conquer division. The absence of a dedicated squaring function is also felt somewhat.

3.5 Modular exponentiation

GMP features a modular exponentiation algorithm 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.

GMP also features a variant of the modular exponentiation algorithm 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, however 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 algorithms that translate a large number into a string that represents it in an arbitrary base (between 2 and 62) and vice versa. This algorithm is surprisingly intricate, in particular when converting from/to base 10. Due to time constraints, we have chosen to instead verify the 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.

3.7 The `mpz` layer

The `mpz` layer is a wrapper around `mpn` 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 `mpz` 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.

However, 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.

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 an `mpn` number. Thus, one can easily pass a number from one library to a function of the other library.

There are two potential sources of incompatibility, as our library lacks a bit of genericity. First, it only works with 64-bit limbs, so it cannot be interfaced with a 32-bit GMP. If the user code only uses `mpz` numbers and does not need to mix both libraries, this incompatibility does not matter. Second, our library does not support the custom memory handler of GMP. In particular, it performs its allocations using `malloc`. Thus, `mpz` numbers from one library will wreak havoc when passed to the other library and freed, unless GMP's default memory handler is used.

4 BENCHMARKS

The next sections show how GMP and WhyMP compare on three benchmarks: multiplication, square root, and a primality test. More precisely, three variants of GMP and three variants of WhyMP are tested. 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 performances.

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 $64 \times 64 \longrightarrow 128$ 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 "*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.*" It is distributed along GMP. It uses the same kind of implementation as GMP without assembly for the two primitives above, that is, it uses only 64-bit operations.

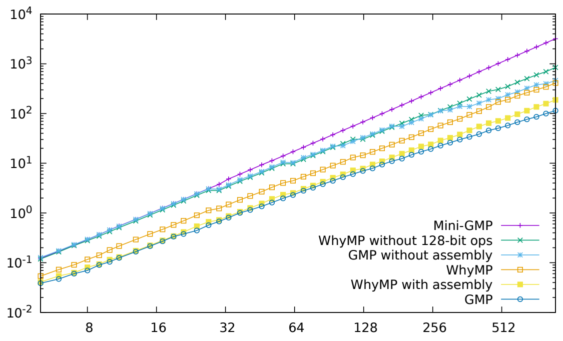
Finally, some low-level `mpn` functions of WhyMP are replaced by their respective GMP counterparts, as these functions are typically written in assembly. Those functions are `add_n` (resp. `sub_n`), which computes the sum (resp. difference) of equally-sized `mpn` numbers; `add` and `sub`, for `mpn` numbers 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. All the libraries are compiled using GCC 8.3.0 using the options selected by GMP, *i.e.*, "`-O2 -march=sandybridge -mtune=sand... -fomit-frame-pointer`".

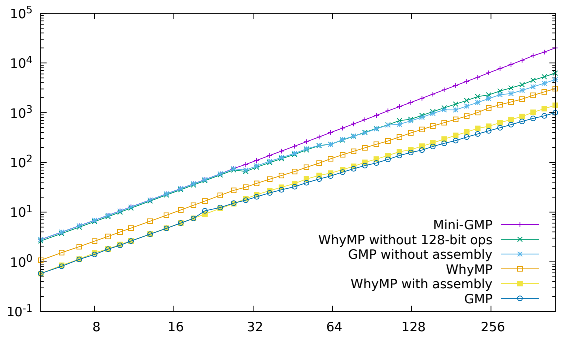
Figure 1 shows the timings obtained on the various benchmarks. On every figure, abscissas are the number of 64-bit limbs, while ordinates are the time in microseconds. All the figures are in log-log scale, so that the asymptotic complexity is apparent. Performance-wise, the general ordering of the plots is the same on every figure: GMP is the fastest, then comes WhyMP with GMP's assembly primitives, then WhyMP, then GMP without assembly support, then WhyMP without 128-bit support, and Mini-GMP is the slowest.

4.1 Multiplication

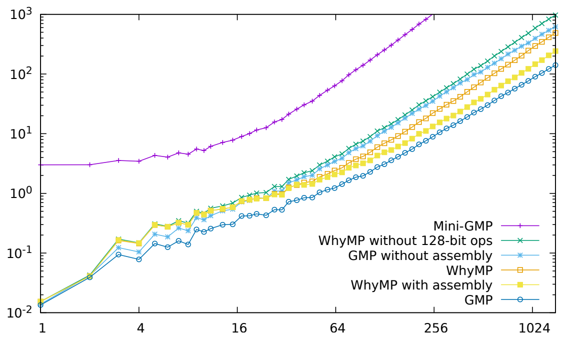
The first benchmark simply tests multiplication for various sizes of `mpn` numbers, so as to exercise both the base-case multiplication as well as Toom-Cook algorithms. Two cases are tested: equal-sized inputs, and $n \times 24n$ unbalanced inputs.



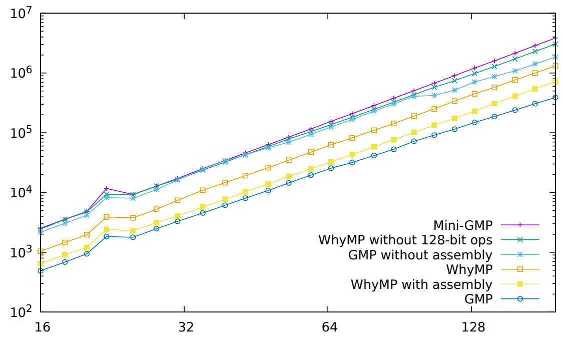
(a) Multiplication $n \times n$



(b) Multiplication $n \times 24n$



(c) Square root



(d) Miller-Rabin

Fig. 1. Timings for multiplication, square root, and Miller-Rabin.

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.

Comparing the plots of Mini-GMP, WhyMP without 128-bit support, and GMP without assembly, makes it apparent when the libraries switch to different 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 tests the square root for various sizes of `mpn` numbers. GMP's algorithm performs a long division, so WhyMP greatly suffers from featuring only the schoolbook division, despite using the same divide-and-conquer square-root algorithm as GMP. 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 implements Miller-Rabin's primality test for number sizes commonly encountered in cryptography applications. This is a simple implementation inspired from GMP's one. It exercises the `mpz` layer as well as the modular exponentiation. Note that the modular exponentiation used in WhyMP is just a wrapper over `mpn_powm`, so it supports neither even modulos nor negative exponents, contrarily to `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 $n \leq 60$.

4.4 Evaluation

Overall, two factors have a large impact on performance: the complexity of the algorithms, and the quality of the underlying arithmetic primitives. On large numbers, WhyMP's multiplication and division falls behind even that of the assembly-free version of GMP when the latter switches to a more efficient algorithm. In all other cases, the algorithms are similar enough that the primitives seem to be the deciding factor. What we conclude from this is that WhyMP's algorithms are close enough to the original that most of the performance difference comes from the primitives written in handwritten assembly, at least for smaller inputs.

5 RELATED WORK

We have used the Why3 tool [4, 5, 7] to develop a formally verified arbitrary-precision integer arithmetic library that closely mirrors GMP. We obtain a verified and efficient C library. Previous work generally does not deal with a large number of highly optimized algorithms. As far as we know, this work is the first formally verified arbitrary-precision integer library that has comparable performance to the state of the art. Let us discuss a few examples of existing verifications of arithmetic libraries.

Bertot *et al.* verified GMP's square root general case algorithm [3] using Coq. Our Why3 proof of that algorithm is directly inspired from 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 [13] 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 manage 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 [1], as well as the basic arithmetic functions the algorithm depends on. The work uses GMP's number representation and a memory model based on separation logic. The author verifies an implementation of the algorithm in a pseudo-code language and proves the fact that the pseudocode correctly simulates the MIPS assembly code.

Fischer verified a modular exponentiation library [8] using Isabelle/HOL and a framework for verifying imperative programs developed by Schirmer [16]. 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 reports 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 (see the discussion on `mpn_copyi` in Section 2.3).

Berghofer used Isabelle/HOL to develop a verified bignum library programmed in the Spark fragment of the Ada programming language [2]. 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-optimised Karatsuba multiplication assembly routines for the AVR architecture [17]. The algorithms are not 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 and the proofs are much easier for SMT solvers.

Finally, Zinzindohoué *et al.* developed a formally verified cryptography library written in F* and extracted to C [18]. 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.

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 GMP's code. This makes it competitive with GMP in some specific cases. For instance, WhyMP is much faster than the pure C variant of GMP. Yet, GMP implements numerous finely tuned assembly routines, which makes WhyMP twice as slow as the standard GMP. By selecting a few multiply-and-accumulate primitives, WhyMP can be brought closer, making it only 5% to 20% slower than GMP, depending on the operation.

In the process of verifying WhyMP, we found one bug in the comparison function of GMP[4] 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, such 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.

During this work, the main obstacle was due to automatic solvers. While the resulting verification process can be said to be automatic, it is only so because the WhyML code was heavily annotated, to the point where it can be seen as a pen-a-paper proof of algorithms. Nonetheless, this is a machine-checked proof. 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.

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] Reynald Affeldt. 2013. On Construction of a Library of Formally Verified Low-level Arithmetic Functions. *Innovations in Systems and Software Engineering* 9, 2 (2013), 59–77. https://doi.org/10.1007/s11334-013-0195-x

[2] Stefan Berghofer. 2012. Verification of Dependable Software using SPARK and Isabelle. In *6th International Workshop on Systems Software Verification (OpenAccess Series in Informatics (OASIcs), Vol. 24)*. Dagstuhl, Germany, 15–31. https://doi.org/10.4230/OASIcs.SSV.2011.15

[3] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. 2002. A Proof of GMP Square Root. *Journal of Automated Reasoning* 29, 3-4 (2002), 225–252. https://doi.org/10.1023/A:1021987403425

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64. https://hal.inria.fr/hal-00790310.

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

[5] Jean-Christophe Filliâtre. 2013. One Logic To Use Them All. In *24th International Conference on Automated Deduction (Lecture Notes in Artificial Intelligence, Vol. 7898)*. Lake Placid, USA, 1–20.

[6] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The Spirit of Ghost Code. *Formal Methods in System Design* 48, 3 (2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x

[7] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *22nd European Symposium on Programming (Lecture Notes in Computer Science, Vol. 7792)*. Heidelberg, Germany, 125–128.

[8] Sabine Fischer. 2008. Formal Verification of a Big Integer Library. In *DATE Workshop on Dependable Software Systems*. http://www-wjp.cs.uni-sb.de/publikationen/Fi08DATE.pdf

[9] Robert W. Floyd. 1993. Assigning Meanings to Programs. In *Program Verification*. Springer, 65–81.

[10] Guillaume Melquiond and Raphaël Rieu-Helft. 2018. A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms. In *9th International Joint Conference on Automated Reasoning (Lecture Notes in Computer Science, Vol. 10900)*. Oxford, United Kingdom, 178–193. https://doi.org/10.1007/978-3-319-94205-6_13

[11] Guillaume Melquiond and Raphaël Rieu-Helft. 2019. Formal Verification of a State-of-the-Art Integer Square Root. In *IEEE 26th Symposium on Computer Arithmetic*. Kyoto, Japan. https://hal.inria.fr/hal-02092970

[12] Niels Moller and Torbjörn Granlund. 2011. Improved Division by Invariant Integers. *IEEE Trans. Comput.* 60, 2 (2011), 165–175. https://doi.org/10.1109/TC.2010.143

[13] Magnus O. Myreen and Gregorio Curello. 2013. Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code. In *3rd International Conference on Certified Programs and Proofs (Lecture Notes in Computer Science, Vol. 8307)*. Melbourne, Australia, 66–81. https://doi.org/10.1007/978-3-319-03545-1_5

[14] Raphaël Rieu-Helft. 2019. A Why3 Proof of GMP Algorithms. *Journal of Formalized Reasoning* 12, 1 (2019), 53–97. https://doi.org/10.6092/issn.1972-5787/9730

[15] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. 2017. How to Get an Efficient yet Verified Arbitrary-Precision Integer Library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments (Lecture Notes in Computer Science, Vol. 10712)*. Heidelberg, Germany, 84–101. https://doi.org/10.1007/978-3-319-72308-2_6

[16] Norbert Schirmer. 2005. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. 398–414.

[17] Marc Schoolderman. 2017. Verifying Branch-Free Assembly Code in Why3. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. 66–83.

[18] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. Cryptology ePrint Archive, Report 2017/536. https://eprint.iacr.org/2017/536.