Formal Verification of a State-of-the-Art Integer Square Root

Guillaume Melquiond Raphaël Rieu-Helft

Inria, TrustInSoft, Université Paris-Saclay

June 11th, 2019

Arbitrary-Precision Integer Arithmetic

The GNU Multiple Precision arithmetic library (GMP)

- Free software, widely used.
- State-of-the-art algorithms, unmatched performances.

Arbitrary-Precision Integer Arithmetic

The GNU Multiple Precision arithmetic library (GMP)

- Free software, widely used.
- State-of-the-art algorithms, unmatched performances.

- Highly intricate algorithms written in low-level C and ASM.
- Ill-suited for random testing.

  GMP 5.0.4: "Two bugs in multiplication [...] with extremely low probability [...]. Two bugs in the gcd code [...] For uniformly distributed random operands, the likelihood is infinitesimally small."

Arbitrary-Precision Integer Arithmetic

The GNU Multiple Precision arithmetic library (GMP)

- Free software, widely used.
- State-of-the-art algorithms, unmatched performances.

- Highly intricate algorithms written in low-level C and ASM.
- Ill-suited for random testing.

  GMP 5.0.4: "Two bugs in multiplication [...] with extremely low probability [...]. Two bugs in the gcd code [...] For uniformly distributed random operands, the likelihood is infinitesimally small."

Objectives

- Produce a verified library compatible with GMP.
- Attain performances comparable to a no-assembly GMP.
- Focus on the low-level `mpn` layer.

GMP's Square Root

```
mp_size_t mpn_sqrtrem
  (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t n);
```

- takes a number $\texttt{np}[n-1] \ldots \texttt{np}[0]$ (with $\texttt{np}[n-1] \neq 0$),
- stores its square root into $\texttt{sp}[\lceil n/2 \rceil - 1] \ldots \texttt{sp}[0]$,
- stores the remainder into $\texttt{rp}[n-1] \ldots \texttt{rp}[0]$,
- returns the actual size of the remainder.

GMP's Square Root

```
mp_size_t mpn_sqrtrem
  (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t n);
```

- takes a number $\texttt{np}[n-1]\ \ldots\ \texttt{np}[0]$ (with $\texttt{np}[n-1] \neq 0$),
- stores its square root into $\texttt{sp}[\lceil n/2 \rceil - 1]\ \ldots\ \texttt{sp}[0]$,
- stores the remainder into $\texttt{rp}[n-1]\ \ldots\ \texttt{rp}[0]$,
- returns the actual size of the remainder.

Three sub-algorithms (assuming a normalized input)

- divide and conquer for $n > 2$,
- an ad-hoc specialization for $n = 2$,
- a bit-fiddling algorithm for $n = 1$.

GMP's Square Root

```
mp_size_t mpn_sqrtrem
  (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t n);
```

- takes a number $\texttt{np}[n-1] \ldots \texttt{np}[0]$ (with $\texttt{np}[n-1] \neq 0$),
- stores its square root into $\texttt{sp}[\lceil n/2 \rceil - 1] \ldots \texttt{sp}[0]$,
- stores the remainder into $\texttt{rp}[n-1] \ldots \texttt{rp}[0]$,
- returns the actual size of the remainder.

Three sub-algorithms (assuming a normalized input)

- divide and conquer for $n > 2$, (proved in Coq in 2002)
- an ad-hoc specialization for $n = 2$,
- a bit-fiddling algorithm for $n = 1$. (actually intricate)

GMP's 64-bit Square Root

```
mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
  mp_limb_t a1, x0, t2, t, x2;
  unsigned abits = a0 >> (GMP_LIMB_BITS - 1 - 8);
  x0 = 0x100 | invsqrttab[abits - 0x80];
  /* x0 is now an 8 bits approximation of 1/sqrt(a0) */
  a1 = a0 >> (GMP_LIMB_BITS - 1 - 32);
  t = (mp_limb_signed_t) (CNST_LIMB(0x2000000000000)
        - 0x30000 - a1 * x0 * x0) >> 16;
  x0 = (x0<<16) + ((mp_limb_signed_t) (x0 * t) >> (16+2));
  /* x0 is now a 16 bits approximation of 1/sqrt(a0) */
  t2 = x0 * (a0 >> (32-8));
  t = t2 >> 25;
  t = ((mp_limb_signed_t)((a0<<14) - t*t - MAGIC)>>(32-8));
  x0 = t2 + ((mp_limb_signed_t) (x0 * t) >> 15);
  x0 >>= 32;
  /* x0 is now a full limb approximation of sqrt(a0) */
  x2 = x0 * x0;
  if (x2 + 2*x0 <= a0 - 1) { x2 += 2*x0 + 1; x0++; }
  *rp = a0 - x2;
  return x0;
}
```

GMP's 64-bit Square Root

```
mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
  mp_limb_t a1, x0, t2, t, x2;
  unsigned abits = a0 >> (GMP_LIMB_BITS - 1 - 8);
  x0 = 0x100 | invsqrttab[abits - 0x80];
  /* x0 is an 8 bits approximation of 1/sqrt(a0) */
  a1 = a0 >> (GMP_LIMB_BITS - 1 - 32);
  t = (mp_limb_signed_t) (CNST_LIMB(0x2000000000000)
        - 0x30000 - a1 * x0 * x0) >> 16;
  x0 = (x0<<16) + ((mp_limb_signed_t)(x0*t)>>(16+2));
  /* x0 is a 16 bits approximation of 1/sqrt(a0) */
  ...
```

GMP's 64-bit Square Root

```
mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
  mp_limb_t a1, x0, t2, t, x2;
  unsigned abits = a0 >> (GMP_LIMB_BITS - 1 - 8);
  x0 = 0x100 | invsqrttab[abits - 0x80];
  /* x0 is an 8 bits approximation of 1/sqrt(a0) */
  a1 = a0 >> (GMP_LIMB_BITS - 1 - 32);
  t = (mp_limb_signed_t) (CNST_LIMB(0x2000000000000)
        - 0x30000 - a1 * x0 * x0) >> 16;
  x0 = (x0<<16) + ((mp_limb_signed_t)(x0*t)>>(16+2));
  /* x0 is a 16 bits approximation of 1/sqrt(a0) */
  ...
```

- Table lookup, Newton iteration toward $1/\sqrt{a}$, modified Newton iteration toward $a/\sqrt{a}$, correcting step.

GMP's 64-bit Square Root

```
mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
  mp_limb_t a1, x0, t2, t, x2;
  unsigned abits = a0 >> (GMP_LIMB_BITS - 1 - 8);
  x0 = 0x100 | invsqrttab[abits - 0x80];
  /* x0 is an 8 bits approximation of 1/sqrt(a0) */
  a1 = a0 >> (GMP_LIMB_BITS - 1 - 32);
  t = (mp_limb_signed_t) (CNST_LIMB(0x2000000000000)
        - 0x30000 - a1 * x0 * x0) >> 16;
  x0 = (x0<<16) + ((mp_limb_signed_t)(x0*t)>>(16+2));
  /* x0 is a 16 bits approximation of 1/sqrt(a0) */
  ...
```

- Table lookup, Newton iteration toward $1/\sqrt{a}$, modified Newton iteration toward $a/\sqrt{a}$, correcting step.
- Hand-coded fixed-point arithmetic.

GMP's 64-bit Square Root

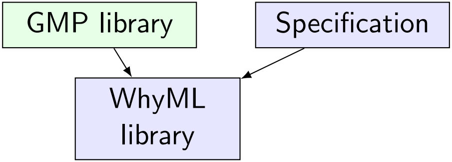
```
mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
  mp_limb_t a1, x0, t2, t, x2;
  unsigned abits = a0 >> (GMP_LIMB_BITS - 1 - 8);
  x0 = 0x100 | invsqrttab[abits - 0x80];
  /* x0 is an 8 bits approximation of 1/sqrt(a0) */
  a1 = a0 >> (GMP_LIMB_BITS - 1 - 32);
  t = (mp_limb_signed_t) (CNST_LIMB(0x2000000000000)
        - 0x30000 - a1 * x0 * x0) >> 16;
  x0 = (x0<<16) + ((mp_limb_signed_t)(x0*t)>>(16+2));
  /* x0 is a 16 bits approximation of 1/sqrt(a0) */
  ...
```

- Table lookup, Newton iteration toward $1/\sqrt{a}$, modified Newton iteration toward $a/\sqrt{a}$, correcting step.
- Hand-coded fixed-point arithmetic.
- Intentional overflow: `(a0<<14) - t*t`.

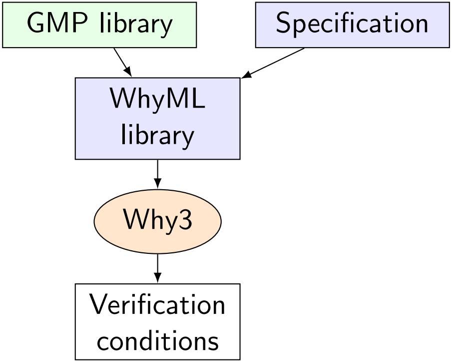
The Why3 Workflow

GMP library

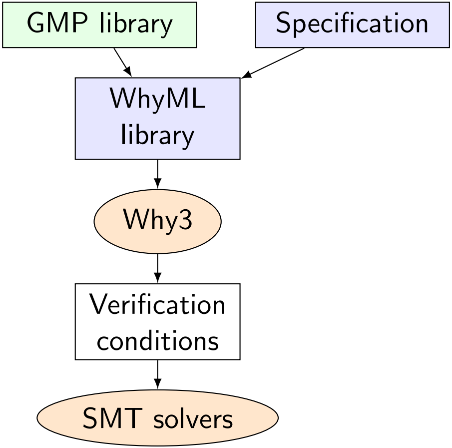
The Why3 Workflow



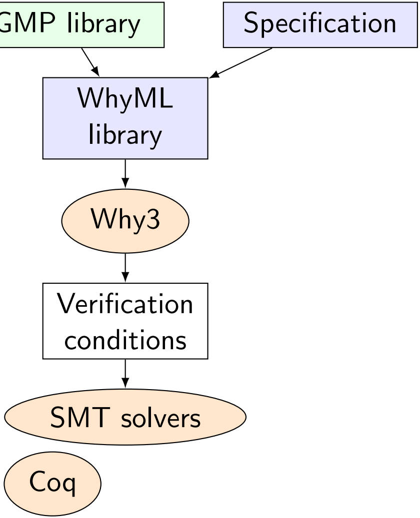
The Why3 Workflow



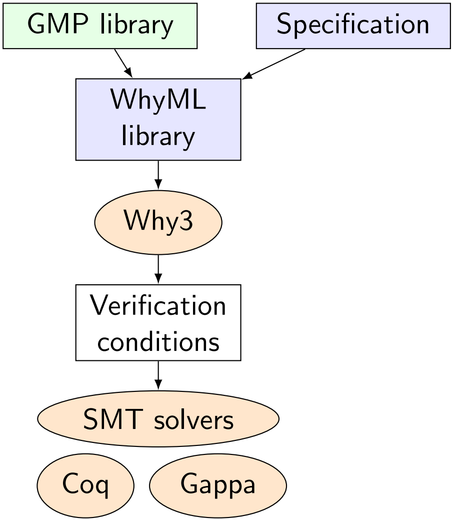
The Why3 Workflow



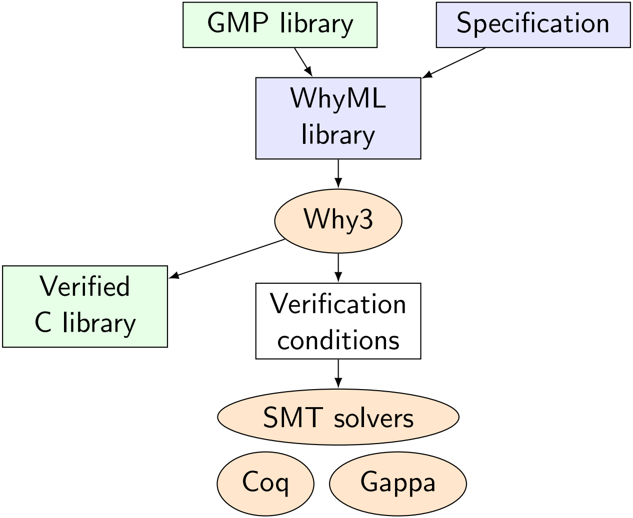
The Why3 Workflow



The Why3 Workflow



The Why3 Workflow



Outline

1. Introduction
2. Fixed-point arithmetic
   - A Why3 theory
   - The case of shifts
   - Translating from C to WhyML
3. Doing the verification
4. Conclusion

How to Relate Integers with Real Numbers?

To express the quadratic convergence of Newton's iteration,
the computed integers have to be seen as real numbers.

How to Relate Integers with Real Numbers?

To express the quadratic convergence of Newton's iteration, the computed integers have to be seen as real numbers.

1. Stored as integers

```
type fxp = uint64

let fxp_add (x y: fxp): fxp
= x + y
```

How to Relate Integers with Real Numbers?

To express the quadratic convergence of Newton's iteration,
the computed integers have to be seen as real numbers.

2. Representing real numbers

```
type fxp = { ival: uint64; ghost iexp: int }
let ghost rval (x: fxp): real = ival x *. pow2 iexp

let fxp_add (x y: fxp): fxp
  requires { iexp x = iexp y }
  requires { ival x + ival y <= max_uint64 }
  ensures  { iexp result = iexp x }
  ensures  { rval result = rval x +. rval y }
= { ival = ival x + ival y; iexp = iexp x }
```

How to Relate Integers with Real Numbers?

To express the quadratic convergence of Newton's iteration, the computed integers have to be seen as real numbers.

3. Accounting for overflows

```
type fxp = { ival: uint64;
             ghost rval: real; ghost iexp: int }
  invariant { rval = floor_at rval iexp }
  invariant { ival = mod (floor (rval *. pow2(-iexp)))
                         (uint64'maxInt + 1) }

val fxp_add (x y: fxp): fxp
  requires { iexp x = iexp y }
  ensures  { iexp result = iexp x }
  ensures  { rval result = rval x +. rval y }
```

The Case of Shifts

```
(mp_limb_signed_t) (x0 * t) >> (16+2)
```

A shift might

- cause rounding (here, loss of the 18 least significant bits),
- align the point for subsequent operations (here, by 17 bits),
- perform a multiplication on real numbers (here, by 2^{-1}).

The Case of Shifts

(mp_limb_signed_t) (x0 * t) >> (16+2)

A shift might

- cause rounding (here, loss of the 18 least significant bits),
- align the point for subsequent operations (here, by 17 bits),
- perform a multiplication on real numbers (here, by 2^{-1}).

```
val fxp_asr' (x: fxp) (k: uint64) (ghost m: int): fxp
  requires { int64'minInt *. pow2 (iexp x) <=.
    rval x <=. int64'maxInt *. pow2 (iexp x) }
  ensures { iexp result = iexp x + k - m }
  ensures { rval result =
    floor_at (rval x *. pow2 (-m)) (iexp x + k - m) }
```

Translating from C to WhyML

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64 =
  let a = fxp_init a0 (-64) in
  let x0 = rsa_estimate a in
  let a1 = fxp_lsr a 31 in
  let m1 = fxp_sub (fxp_init 0x2000000000000 (-49))
                   (fxp_init 0x30000 (-49)) in
  let t1' = fxp_sub m1 (fxp_mul (fxp_mul x0 x0) a1) in
  let t1 = fxp_asr t1' 16 in
  let x1 = fxp_add (fxp_lsl x0 16)
                   (fxp_asr' (fxp_mul x0 t1) 18 1) in
  let a2 = fxp_lsr a 24 in let u1 = fxp_mul x1 a2 in
  let u2 = fxp_lsr u1 25 in
  let m2 = fxp_init 0x24000000000 (-78) in
  let t2' = fxp_sub (fxp_sub (fxp_lsl a 14) (fxp_mul u2 u2)) m2 in
  let t2 = fxp_asr t2' 24 in
  let x2 = fxp_add u1 (fxp_asr' (fxp_mul x1 t2) 15 1) in
  let x = fxp_lsr x2 32 in
  let ref c = ival x in let ref s = c * c in
  if (s + 2 * c <= a0 - 1) then begin
    s <- s + 2 * c + 1;
    c <- c + 1;
  end;
  set rp (a0 - s); c
```

Outline

Verification Conditions

Specification

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 <= a0 }
  ensures { result*result <= a0
            < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
  ensures { get rp <= 2 * result }
```

Verification Conditions

Specification

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 <= a0 }
  ensures { result*result <= a0
             < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
  ensures { get rp <= 2 * result }
```

Kinds of verification conditions

1. Memory accesses are valid; fixed-point numbers are aligned.
(automatic)

Verification Conditions

Specification

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 <= a0 }
  ensures { result*result <= a0
            < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
  ensures { get rp <= 2 * result }
```

Kinds of verification conditions

1. Memory accesses are valid; fixed-point numbers are aligned. (automatic)
2. Result is correctly reconstructed from the fixed-point value x_2. (verbose, but straightforward)

Verification Conditions

Specification

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 <= a0 }
  ensures { result*result <= a0
              < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
  ensures { get rp <= 2 * result }
```

Kinds of verification conditions

1. Memory accesses are valid; fixed-point numbers are aligned. (automatic)
2. Result is correctly reconstructed from the fixed-point value x_2. (verbose, but straightforward)
3. x_1 and x_2 are accurate, e.g., $x_2 - \sqrt{a} \in [-2^{-32}; 0]$. (???)

Error Analysis

Newton iteration toward $1/\sqrt{a}$

- Recurrence: $x_{i+1} = x_i + x_i \cdot (1 - a \cdot x_i^2)/2$.
- Relative error: $x_i = a^{-1/2} \cdot (1 + \varepsilon_i)$.
- Quadratic convergence: $|\varepsilon_{i+1}| \simeq \frac{3}{2}|\varepsilon_i|^2$.

Error Analysis

Newton iteration toward $1/\sqrt{a}$

- Recurrence: $x_{i+1} = x_i + x_i \cdot (1 - a \cdot x_i^2)/2$.
- Relative error: $x_i = a^{-1/2} \cdot (1 + \varepsilon_i)$.
- Quadratic convergence: $|\varepsilon_{i+1}| \simeq \frac{3}{2}|\varepsilon_i|^2$.

But what the code actually computes is

$\tilde{x}_1 = \tilde{x}_0 + \triangledown_{-24} \left(\tilde{x}_0 \cdot \triangledown_{-33} \left(1 - 3 \cdot 2^{-33} - \triangledown_{-33}(a) \cdot \tilde{x}_0^2\right) /2\right)$,
with $\triangledown_k(r) = \lfloor r \cdot 2^{-k} \rfloor \cdot 2^k$.

Error Analysis

Newton iteration toward $1/\sqrt{a}$

- Recurrence: $x_{i+1} = x_i + x_i \cdot (1 - a \cdot x_i^2)/2$.
- Relative error: $x_i = a^{-1/2} \cdot (1 + \varepsilon_i)$.
- Quadratic convergence: $|\varepsilon_{i+1}| \simeq \frac{3}{2}|\varepsilon_i|^2$.

But what the code actually computes is

$$\tilde{x}_1 = \tilde{x}_0 + \triangledown_{-24}\left(\tilde{x}_0 \cdot \triangledown_{-33}\left(1 - 3 \cdot 2^{-33} - \triangledown_{-33}(a) \cdot \tilde{x}_0^2\right)/2\right),$$

with $\triangledown_k(r) = \lfloor r \cdot 2^{-k} \rfloor \cdot 2^k$.

What to do about...

- Rounding errors? That is what Gappa is designed to handle.

Error Analysis

Newton iteration toward $1/\sqrt{a}$

- Recurrence: $x_{i+1} = x_i + x_i \cdot (1 - a \cdot x_i^2)/2$.
- Relative error: $x_i = a^{-1/2} \cdot (1 + \varepsilon_i)$.
- Quadratic convergence: $|\varepsilon_{i+1}| \simeq \frac{3}{2}|\varepsilon_i|^2$.

But what the code actually computes is

$\tilde{x}_1 = \tilde{x}_0 + \triangledown_{-24}\left(\tilde{x}_0 \cdot \triangledown_{-33}\left(1 - 3 \cdot 2^{-33} - \triangledown_{-33}(a) \cdot \tilde{x}_0^2\right)/2\right)$,
with $\triangledown_k(r) = \lfloor r \cdot 2^{-k} \rfloor \cdot 2^k$.

What to do about...

- Rounding errors? That is what Gappa is designed to handle.
- Magic constants? Critical for soundness; hinted to Gappa.

Prover interoperability

Help Gappa by providing equalities

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
= ...
  let ghost rsa = pure { 1. /. sqrt a } in
  let ghost e0  = pure { (x0 -. rsa) /. rsa } in
  let ghost ea1 = pure { (a1 -. a) /. a } in
  let ghost mx1 = pure { x0 +. x0 *. t1' *. 0.5 } in
  assert { (mx1 -. rsa) /. rsa =
    -0.5 *. (e0*.e0 *. (3.+.e0) +. (1.+.e0) *.
      (1. -. m1 +. (1.+.e0)*.(1.+.e0) *. ea1)) };
  ...
```

Prover interoperability

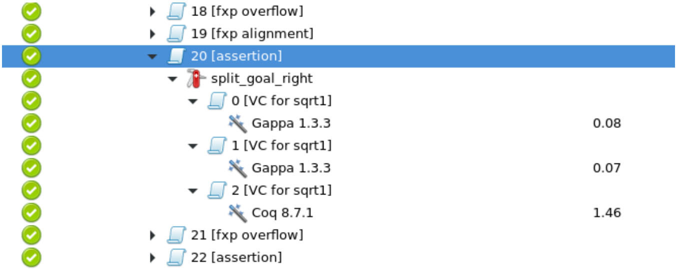
Help Gappa by providing equalities

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
= ...
  let ghost rsa = pure { 1. /. sqrt a } in
  let ghost e0  = pure { (x0 -. rsa) /. rsa } in
  let ghost ea1 = pure { (a1 -. a) /. a } in
  let ghost mx1 = pure { x0 +. x0 *. t1' *. 0.5 } in
  assert { (mx1 -. rsa) /. rsa =
    -0.5 *. (e0*.e0 *. (3.+.e0) +. (1.+.e0) *.
      (1. -. m1 +. (1.+.e0)*.(1.+.e0) *. ea1)) };
  ...
```

Four equalities are needed by Gappa:

- they hardly mention rounding errors;
- all of them are proved with straightforward Coq scripts.

Prover interoperability



- Assertions are proof cuts, not axioms.
- Why3 makes sure everything was proved.

Outline

A Verified 64-bit Integer Square Root

Fixed-point square root in WhyML

- Extracted C code is equivalent to GMP's code.
- Ghost values also serve as documentation.
- Proof replay takes less than 30 seconds.
- Verification work took only a few days.

A Verified 64-bit Integer Square Root

Fixed-point square root in WhyML

- Extracted C code is equivalent to GMP's code.
- Ghost values also serve as documentation.
- Proof replay takes less than 30 seconds.
- Verification work took only a few days.

Shortcomings

- The "magic" constant is not the same as GMP's.
- Why3 does not yet support literal arrays,
  so the lookup table is verified outside.

A Verified Library Compatible with GMP

Supported operations

- Addition, subtraction, comparison, shifts.
- Multiplication: quadratic, and Toom-Cook 2 and 2.5.
- Division: "schoolbook".
- Square root: divide-and-conquer.

A Verified Library Compatible with GMP

Supported operations

- Addition, subtraction, comparison, shifts.
- Multiplication: quadratic, and Toom-Cook 2 and 2.5.
- Division: "schoolbook".
- Square root: divide-and-conquer.

Performances

- About 10-20% slower than pure-C GMP (for "small" inputs).
- About 2x slower than GMP with hand-coded assembly.
- Faster than Mini-GMP.

`https://www.lri.fr/~rieu/wmp.html`