An efficient yet formally verified arbitrary-precision integer library

Raphaël Rieu-Helft [1,2,3,4] Claude Marché [2] Guillaume Melquiond [2]

[1]École normale supérieure [2]Inria [3]TrustInSoft [4]Université Paris-Saclay

May 31, 2017

context

goal: efficient and formally verified large integer library

GMP:

- widely-used, high-performance library
- safety-critical
- tested, but hard to ensure good coverage (unlikely branches)

idea:

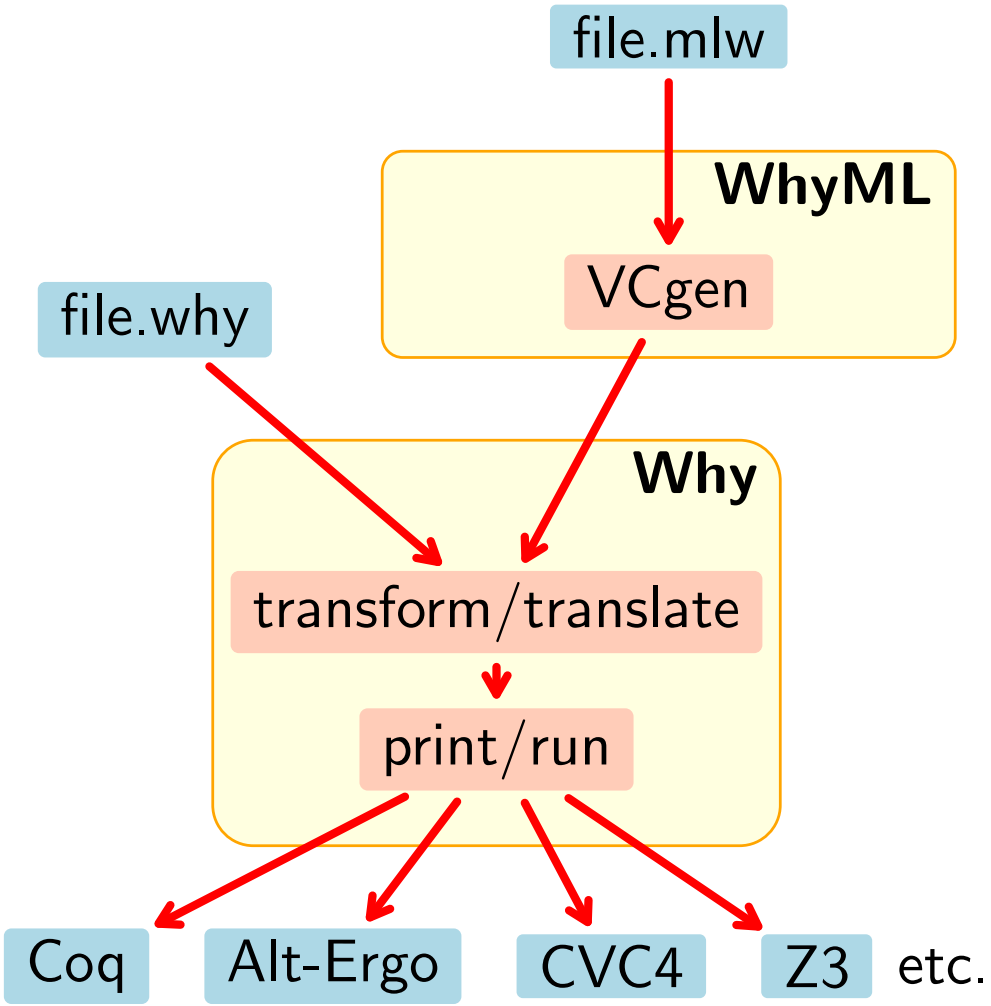
1. formally verify GMP algorithms with Why3
2. extract efficient C code

outline

1. Reimplementing GMP in Why3
2. An algorithm: long division
3. Benchmarks and conclusions

Reimplementing GMP in Why3

tool: the Why3 platform



approach:

- implement the GMP algorithms in WhyML
- verify them with Why3
- extract to C

difficulties:

- preserve all GMP implementation tricks
- prove them correct
- extract to efficient C code

an example: comparison

large integer $\equiv$ pointer to array of unsigned integers $a_0 \ldots a_{n-1}$ called limbs

$$\mathtt{value}(a, n) = \sum_{i=0}^{n-1} a_i \beta^i$$

```
let wmpn_cmp (x y:ptr uint64) (sz:int32): int32
= let i = ref sz in
  try
    while !i ≥ 1 do
      i := !i - 1;
      let lx = x[!i] in
      let ly =  y[!i] in
      if lx ≠ ly then
        if lx > ly
        then raise (Return32 1)
        else raise (Return32 (-1))
      end
    done;
    0
  with Return32 r → r
  end
```

comparison: extracted C code

```
int32_t wmpn_cmp(uint64_t * x, uint64_t * y, int32_t sz) {
  int32_t i, o;
  uint64_t lx, ly;
  i = (sz);
  while (i >= 1) {
    o = (i - 1); i = o;
    lx = (*(x+(i)));
    ly = (*(y+(i)));
    if (lx != ly) {
      if (lx > ly) return (1);
      else return (-(1));
    }
  }
  return (0);
}
```

uses machine types directly: no closures, no indirections

tradeoff: only a small fragment of WhyML is handled by our extraction

memory model

```
type ptr 'a = { mutable data: array 'a ; offset: int }

function plength (p:ptr 'a): int = p.data.length
function pelts (p:ptr 'a): (int → 'a) = p.data.elts

predicate valid (p:ptr 'a) (sz:int) =
  0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p
```

memory model

```
type ptr 'a = { mutable data: array 'a ; offset: int }

function plength (p:ptr 'a): int = p.data.length
function pelts (p:ptr 'a): (int → 'a) = p.data.elts

predicate valid (p:ptr 'a) (sz:int) =
  0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p


val incr (p:ptr 'a) (ofs:int32): ptr 'a          (*   p+ofs   *)
  requires { p.offset + ofs ≤ plength p }
  alias    { p.data ~ result.data }
  ensures  { result.offset = p.offset + ofs }
  ensures  { result.data = p.data }
```

example specification: long addition

specifications are defined in terms of `value`

```
(** [wmpn_add r x sx y sy] adds [(x, sx)] to [(y,sy)] and writes the
result in [(r, sx)].  [sx] must be greater than or equal to
[sy]. Returns carry, either 0 or 1. Corresponds to [mpn_add]. *)

let wmpn_add (r x:ptr uint64) (sx:int32) (y:ptr uint64) (sy:int32): uint64
  requires { 0 ≤ sy ≤ sx }
  requires { valid x sx ∧ valid y sy ∧ valid r sx }
  writes   { r.data.elts }
  ensures  { 0 ≤ result ≤ 1 }
  ensures  { value r sx + (power radix sx) * result = value x sx + value y sy }
```

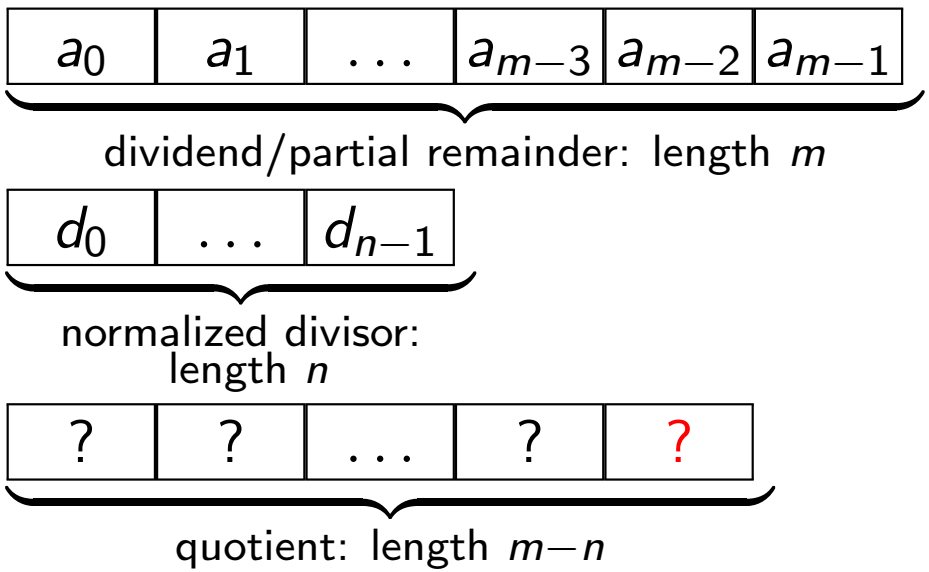
Why3 typing constraint: `r` cannot be aliased to `x` or `y`

- simplifies proofs : aliases are known statically
- we need another function for in-place addition

an algorithm: long division

long division: naïve algorithm

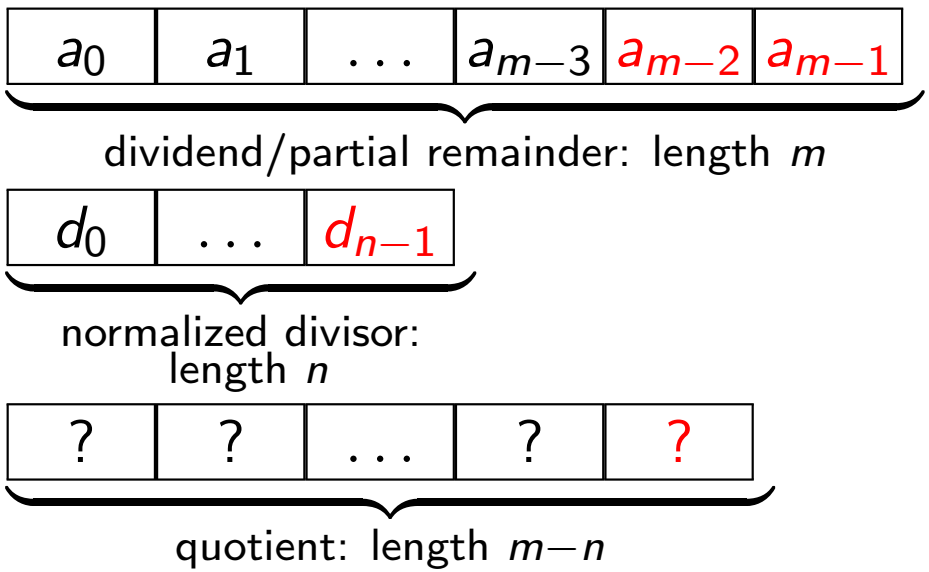
one loop iteration:



long division: naïve algorithm

one loop iteration:

1. estimate the highest limb of the quotient (with a short division)

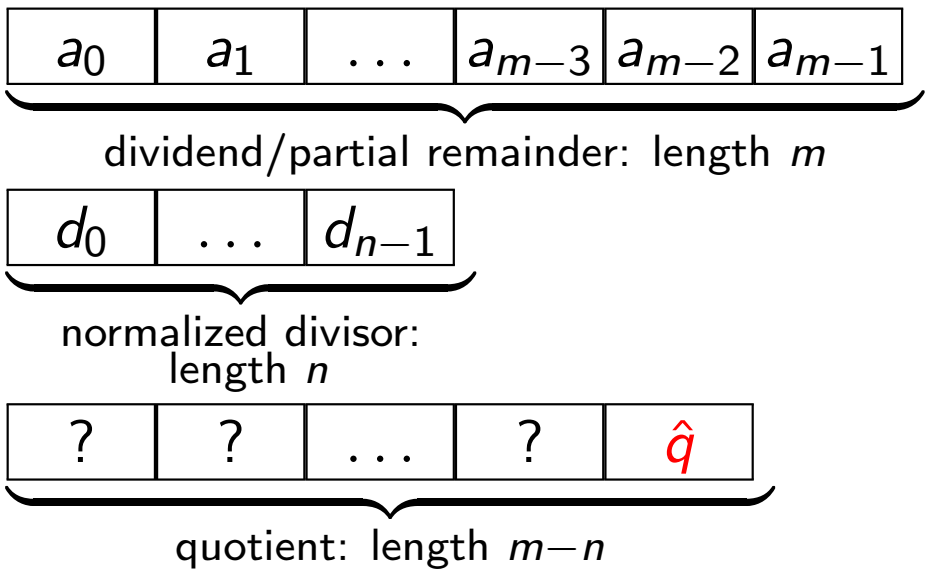


$$\hat{q} = \overline{a_{m-2}a_{m-1}}/d_{n-1}$$

long division: naïve algorithm

one loop iteration:

1. estimate the highest limb of the quotient (with a short division)
2. substract the product of that and the divisor from the dividend

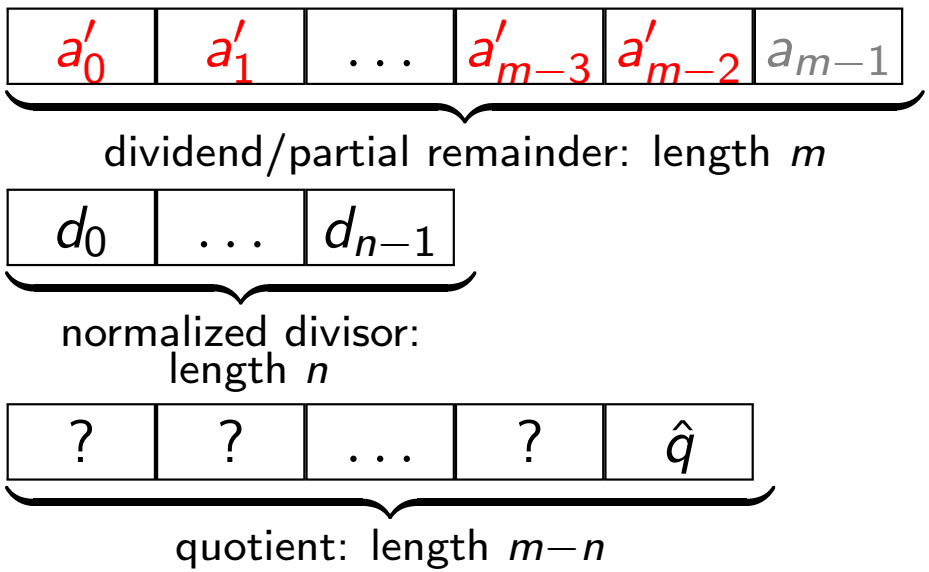


$$\overline{a'_0 a'_1 \dots a'_{m-2}} = \overline{a} - \hat{q} \times \overline{d}$$

long division: naïve algorithm

one loop iteration:

1. estimate the highest limb of the quotient (with a short division)
2. substract the product of that and the divisor from the dividend



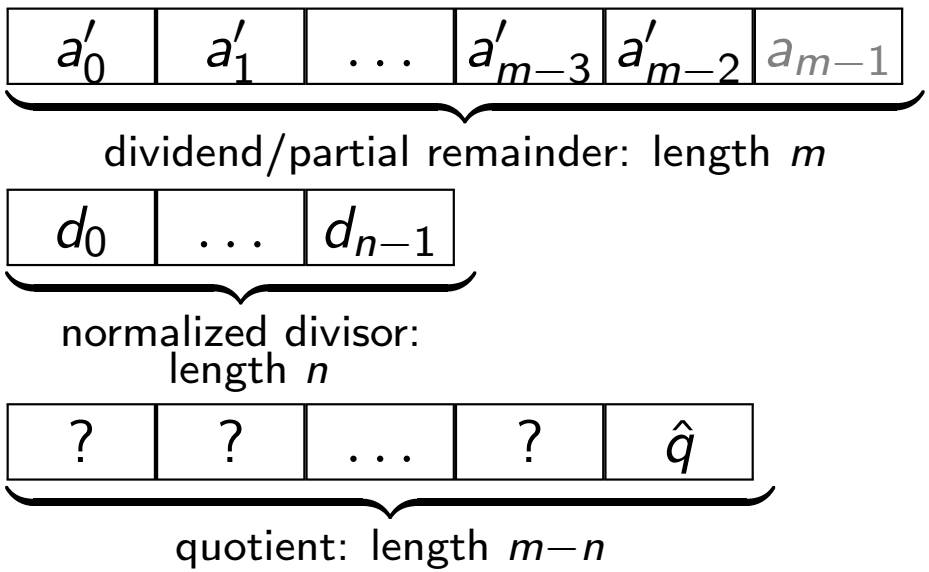
$$\overline{a'_0 a'_1 \dots a'_{m-2}} = \overline{a} - \hat{q} \times \overline{d}$$

if $\hat{q}$ is right $a'_{m-1} = 0$

long division: naïve algorithm

one loop iteration:

1. estimate the highest limb of the quotient (with a short division)
2. substract the product of that and the divisor from the dividend
3. if quotient is too large, adjust it



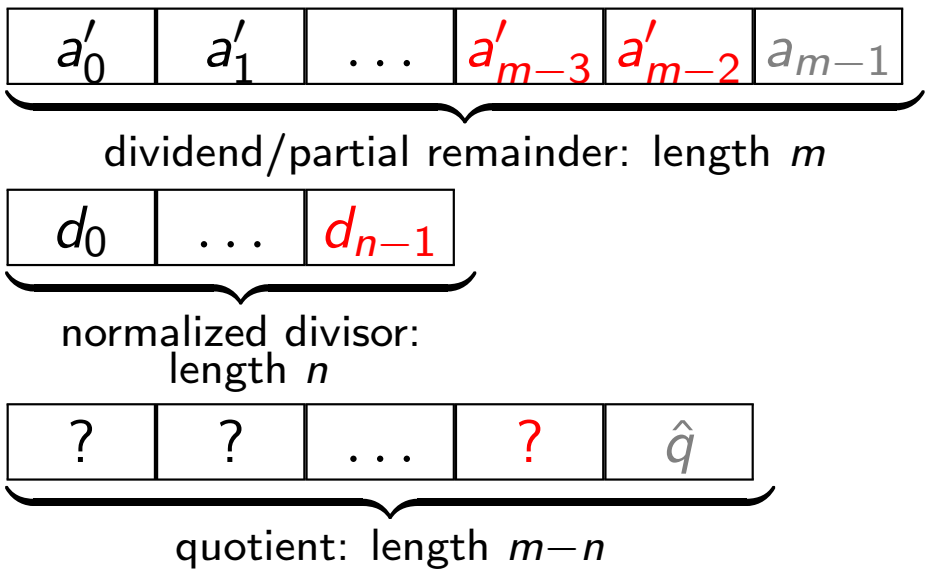
$\hat{q} \leftarrow \hat{q} - 1$ $\quad$ $\overline{a'} \leftarrow \overline{a'} + \overline{d}$

until it works...

long division: naïve algorithm

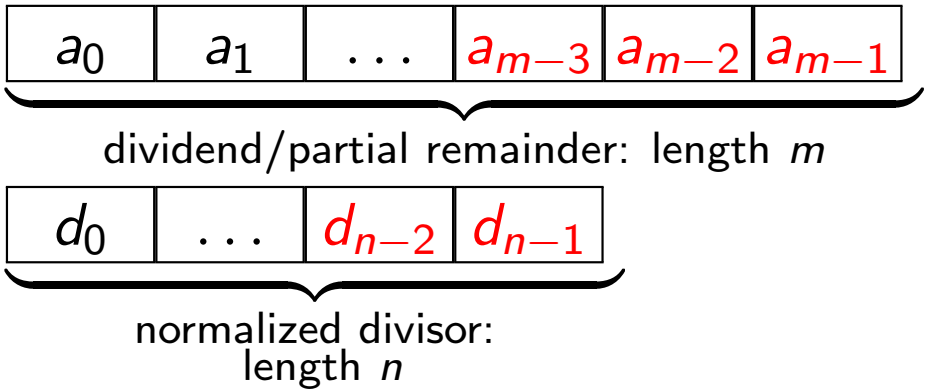
one loop iteration:

1. estimate the highest limb of the quotient (with a short division)
2. substract the product of that and the divisor from the dividend
3. if quotient is too large, adjust it



optimization: 3-by-2 division

goal: better estimate of the quotient



$$\hat{q} = \overline{a_{m-3}a_{m-2}a_{m-1}} / \overline{d_{n-2}d_{n-1}}$$

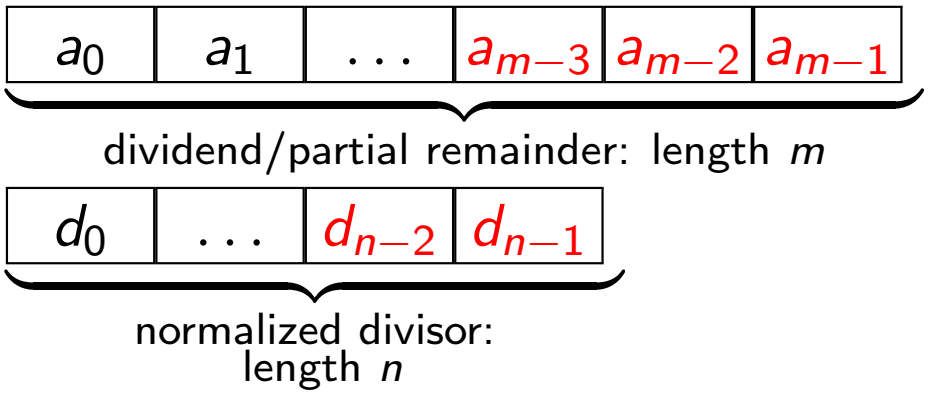
$$\overline{r_0 r_1} = \overline{a_{m-3}a_{m-2}a_{m-1}} - \hat{q} \times \overline{d_{n-2}d_{n-1}}$$

adjustment: at most once, and only if $r_1 = 0 \Rightarrow$ very unlikely

same divisor at each iteration: we can precompute a pseudo-inverse
3-by-2 division uses no division primitive (Möller & Granlund 2011)

optimizing long subtraction

only large integer operation in the critical loop ⇒ most of the cost is there



$$\hat{q} = \overline{a_{m-3}a_{m-2}a_{m-1}} / \overline{d_{n-2}d_{n-1}}$$

$$\overline{r_0 r_1} = \overline{a_{m-3}a_{m-2}a_{m-1}} - \hat{q} \times \overline{d_{n-2}d_{n-1}}$$

$$\overline{a'_0 a'_1 \dots a'_{m-2}} = \overline{a_0 \dots a_{m-3}a_{m-2}a_{m-1}} - \beta^{m-n-1}\hat{q} \times \overline{d_0 \dots d_{n-2}d_{n-1}}$$

but we already have $\overline{a_{m-3}a_{m-2}a_{m-1}} - \hat{q} \times \overline{d_{n-2}d_{n-1}} = \overline{r_0 r_1}$

⇒ subtraction over length $n-2$ instead of n, then propagate borrow

final algorithm

1: **function** div_gen(q, a, d, m, n, inv)
2: ...
3: $i = m - n$
4: **while** $i > 0$ **do**
5: $i \leftarrow i - 1$
6: **if** $a_{n+i} = d_{n-1}$ **and** $a_{n+i-1} = d_{n-2}$ **then** ▷ unlikely
7: $\hat{q} \leftarrow \beta - 1$
8: submul_limb($a + i, d, n, \hat{q}$) ▷ we know the result is d_{n-1}
9: **else**
10: $(\hat{q}, h, l) \leftarrow$ div_3by2($a_{n+i}, a_{n+i-1}, a_{n+i-2}, d_{n-1}, d_{n-2}, inv$)
11: $b \leftarrow$ submul_limb($a + i, d, n - 2, \hat{q}$)
12: $b_1 \leftarrow (l < b)$
13: $a_{n+i-2} \leftarrow (l - b \bmod \beta)$
14: $b_2 \leftarrow (h < b_1)$
15: $a_{n+i-1} \leftarrow (h - b_1 \bmod \beta)$
16: **if** $b_2 \neq 0$ **then** ▷ unlikely, and then $b_2 = 1$
17: $\hat{q} \leftarrow \hat{q} - 1$ ▷ only one adjustment step
18: $c \leftarrow$ add_in_place($a + i, d, n$)
19: $q_i \leftarrow \hat{q}$

final algorithm

1: **function** div_gen(q, a, d, m, n, inv)
2: $x \leftarrow a_{m-1}$
3: $i = m - n$
4: **while** $i > 0$ **do**
5: $i \leftarrow i - 1$
6: **if** $x = d_{n-1}$ **and** $a_{n+i-1} = d_{n-2}$ **then** ▷ unlikely
7: $\hat{q} \leftarrow \beta - 1$
8: submul_limb($a + i, d, n, \hat{q}$) ▷ we know the result is d_{n-1}
9: $x \leftarrow a_{n+i-1}$
10: **else**
11: $(\hat{q}, x, l) \leftarrow$ div_3by2($x, a_{n+i-1}, a_{n+i-2}, d_{n-1}, d_{n-2}, inv$)
12: $b \leftarrow$ submul_limb($a + i, d, n - 2, \hat{q}$)
13: $b_1 \leftarrow (l < b)$
14: $a_{n+i-2} \leftarrow (l - b \bmod \beta)$
15: $b_2 \leftarrow (x < b_1)$
16: $x \leftarrow (x - b_1 \bmod \beta)$
17: **if** $b_2 \neq 0$ **then** ▷ unlikely, and then $b_2 = 1$
18: $\hat{q} \leftarrow \hat{q} - 1$ ▷ only one adjustment step
19: $c \leftarrow$ add_in_place($a + i, d, n - 1$)
20: $x \leftarrow x + d_{n-1} + c$
21: $q_i \leftarrow \hat{q}$
22: $a_{n-1} \leftarrow x$

Benchmarks and conclusions

comparison with GMP

we compare with GMP without assembly (option `--disable-assembly`)
$\Rightarrow$ we want to compare the algorithms rather than the compilers

we only consider inputs of 20 words or less ($\sim$ 1300 bits)
$\Rightarrow$ above that, GMP uses different algorithms

multiplication: less than 5% slower than GMP, satisfactory

division: $\sim$ 20% slower than GMP

- except for n very close to m
  $\Rightarrow$ GMP uses a different algorithm for $n > m/2$, to do
- normalization and operand size choice steps can be improved

performances are very dependent on the compiled code of the primitives
ongoing : link to GMP to use the exact same primitives

contribution, proof effort

GMP `mpn` functions implemented: add, sub, mul, div, compare, shifts

new Why3 features:

- extraction and memory model for C
- `alias` of return value and parameter

The proof:

- 6000 lines of Why3 code
  - 1350 of programs
  - 4650 of specifications and (mostly) assertions
- used solvers: Alt-Ergo, CVC3, CVC4, E, Z3.
- some subgoals discharged with Coq ($<$ 50 lines)
- total proof replay time: $\sim$ 18 minutes

conclusions

verified C library, bit-compatible with GMP

GMP implementation tricks preserved
$\Rightarrow$ satisfactory performances in the handled cases

proof effort too heavy to carry on without changing our methods
$\Rightarrow$ non-linear arithmetic: SMT solvers need a lot of help

coming soon:

- decision procedures for nonlinear arithmetic
  $\Rightarrow$ undecidable, but the fragment we need looks specific enough
- divide-and-conquer algorithms for multiplication and division
- parameter aliasing
- GMP `mpz` functions

Thank you for your attention

arithmetic primitives: the uint64 type

```
type uint64
val function to_int (n:uint64): int    (* ℤ *)
meta coercion function to_int

let constant max = 0xffff_ffff_ffff_ffff
predicate in_bounds (n:int) = 0 ≤ n ≤ max
axiom to_int_in_bounds: forall n:uint64. in_bounds n

val mul (x y:uint64): uint64                          (* defensive, no overflow *)
  requires { in_bounds (x * y)  }
  ensures  { result = x * y }

val mul_mod (x y:uint64): uint64                      (* with overflow *)
  ensures { result = mod (x * y) (max+1) }

val mul_double (x y:uint64): (uint64,uint64)          (* returns two words*)
  returns { (l,h) → l + (max+1) * h = x * y }
```

memory model

```
type ptr 'a = { mutable data: array 'a ; offset: int }

function plength (p:ptr 'a): int = p.data.length
function pelts (p:ptr 'a): (int → 'a) = p.data.elts

predicate valid (p:ptr 'a) (sz:int) =
  0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p

val malloc (sz:uint32) : ptr 'a                    (*   malloc(sz * sizeof('a))   *)
  requires { sz > 0 }
  ensures  { plength result = sz ∨ plength result = 0 }
  ensures  { result.offset = 0 }

val free (p:ptr 'a) : unit                         (*   free(p)   *)
  requires { p.offset = 0 }
  writes   { p.data }
  ensures  { plength p = 0 }

val incr (p:ptr 'a) (ofs:int32): ptr 'a          (*   p+ofs   *)
  requires { p.offset + ofs ≤ plength p }
  alias    { p.data ~ result.data }
  ensures  { result.offset = p.offset + ofs }
  ensures  { result.data = p.data }
```