Verification of efficient C arithmetic algorithms with Why3

Raphaël Rieu-Helft
joint work with Guillaume Melquiond and Claude Marché

December 7, 2020
Machine Arithmétique, Pl. I.

Fig. 2.

Cette image montre une machine arithmétique avec des disques marqués de différents chiffres, représentant les différentes décimales de l'arithmétique. Les disques sont disposés de manière à indiquer les valeurs milliers, centaines, dizaines, unités, dixièmes, centièmes, millièmes, pour illustrer les opérations arithmétiques.
Computer arithmetic, integer representation

Usual integer representation

- Machine word: string of $k$ bits, $k$ depends on the architecture
- Typically $k = 64$ or $k = 32$
- A machine word can represent any integer between 0 and $2^k - 1$
Computer arithmetic, integer representation

**Usual integer representation**
- Machine word: string of $k$ bits, $k$ depends on the architecture
- Typically $k = 64$ or $k = 32$
- A machine word can represent any integer between 0 and $2^k - 1$

**What about larger numbers?**
- Required for cryptography, computer algebra systems...
Arbitrary-precision arithmetic

Integer representation

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

value$(a, n) = \sum_{i=0}^{n-1} a_i \beta^i$  \hspace{1cm} 0 $\leq$ $a_i$ $<$ $\beta$  \hspace{1cm} $\beta = 2^{64}$
Arbitrary-precision arithmetic

**Integer representation**

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

\[
\text{value}(a, n) = \sum_{i=0}^{n-1} a_i \beta^i \\
0 \leq a_i < \beta \\
\beta = 2^{64}
\]

**The GNU Multiple Precision library (GMP)**

- Free software, widely used arbitrary-precision arithmetic library
- State-of-the-art algorithms written in C
Motivation

Decrementing a long integer by 1 (simplified from `mpn_decr_u`)

```c
#define mpn_decr_1(x) \ 
mp_ptr __x = (x); \ 
while (*((__x++))-- == 0) ;
```

- Hard-to-read single-line code from the GMP library
- Can the program crash? (safety)
- Does it compute the right value? (functional correctness)
Motivation

Decrementing a long integer by 1 (simplified from `mpn_decr_u`)

```c
#define mpn_decr_1(x) \ 
    mp_ptr __x = (x); \ 
    while (*((__x++))-- == 0) ;
```

- Hard-to-read single-line code from the GMP library
- Can the program crash? (safety)
- Does it compute the right value? (functional correctness)

How to verify this?
Functional verification in a nutshell

Verification tool

Code

Specification

OK!
Informal specification (incomplete)

// requires: x valid over some length sz
// requires: value x sz >= 1
// ensures: value x sz = old (value x sz) - 1

#define mpn_decr_1(x)\
    mp_ptr __x = (x);\
    while (*((__x ++))-- == 0) ;

Next steps:
formalize the specification
check whether the program matches the specification
How to do so efficiently?
Informal specification (incomplete)

```c
// requires: x valid over some length sz
// requires: value x sz >= 1
// ensures: value x sz = old (value x sz) - 1

#define mpn_decr_1(x) \
    mp_ptr __x = (x); \
    while (*((__x++)--) == 0) ;
```

Next steps:
- formalize the specification
- check whether the program matches the specification
Informal specification (incomplete)

```c
// requires: x valid over some length sz
// requires: value x sz >= 1
// ensures: value x sz = old (value x sz) - 1
#define mpn_decr_1(x) \
  mp_ptr __x = (x); \
  while (*((__x++)--) == 0) ;
```

Next steps:
- formalize the specification
- check whether the program matches the specification

How to do so efficiently?
The Why3 workflow

C library
The Why3 workflow

- C library
- WhyML library
- Specification, assertions, invariants

Verification of efficient C arithmetic algorithms with Why3
The Why3 workflow

- C library
- WhyML library
- Why3
- Verification conditions

- Specification, assertions, invariants
The Why3 workflow

- **C library**
- **WhyML library**
- **Why3**
- **Verification conditions**
- **SMT solvers**

- **Specification, assertions, invariants**
The Why3 workflow
The Why3 workflow

- C library
- WhyML library
- Why3
- Verification conditions
- SMT solvers (Coq, Gappa)
- Specification, assertions, invariants
The Why3 workflow

- **C library**
- **WhyML library**
- **Why3**
- **Verification conditions**
- **SMT solvers**
- **OK!**

- **Specification, assertions, invariants**
- **Coq**
- **Gappa**

Why3 Division WhyMP Conclusion
The Why3 workflow

- C library
- Specification, assertions, invariants
- WhyML library
- Why3
- Verification conditions
- SMT solvers
- OK!
- Verified C library
- Coq
- Gappa

R. Rieu-Helft
Verification of efficient C arithmetic algorithms with Why3
Verifying `mpn_decr_1`

### Original macro (simplified from 18-line `mpn_decr_u`)

```c
#define mpn_decr_1(x)  
  mp_ptr __x = (x);  
  while (((*(__x++))--) == 0) ;
```

### Translation to WhyML

```whyml
let wmpn_decr_1 (x: ptr uint64) (ghost sz: int32): unit
  requires { valid x sz }
  requires { 1 <= value x sz }
  ensures { value x sz = value (old x) sz - 1 }
  =
  let ref lx = 0 in
  let ref xp = incr x 0 in
  while lx = 0 do
    lx <- get xp;
    set xp (sub_mod lx 1);
    xp <- incr xp 1;
  done
```

R. Rieu-Helft
Verification of efficient C arithmetic algorithms with Why3
Verifying `mpn_decr_1`

**Original macro (simplified from 18-line `mpn_decr_u`)**

```c
#define mpn_decr_1(x)  
    mp_ptr __x = (x);  
    while (*((__x++))-- == 0) ;
```

**Extraction to C**

```c
void wmpn_decr_1(uint64_t *x) {
    uint64_t lx, *xp, res;
    lx = 0;
    xp = x + 0;
    while (lx == 0) {
        lx = *xp;
        res = lx - 1;
        *xp = res;
        xp = xp + 1;
    }
}
```
Overview

State-of-the-art C library: GMP

Specification, assertions, invariants

WhyML library

Model of C

Why3

Verification conditions

Verified C library: WhyMP

SMT solvers

Coq

Gappa
Plan

1. Introduction
2. Memory model and extraction
3. An algorithm: long division
4. WhyMP
5. Conclusion, perspectives
Memory model: goals and challenges

**Goals**
- Accurate transcription of C programs in WhyML
- Tractable proofs

**Challenges**
- No native notion of pointers in WhyML
- Alias handling:
  - Aliased pointers
  - Function arguments that may or may not be aliased
**Memory model**

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

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

val malloc (sz:uint32) : ptr 'a (* malloc(sz * sizeof('a)) *)
    ensures { is_not_null result → valid result sz }
    ...

val free (p:ptr 'a) : unit (* free(p) *)
    ...
```
Alias control

aliased C pointers ⇔ point to the same memory object
aliased WhyML pointers ⇔ shared value in the data field

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

val incr (p:ptr 'a) (ofs:int32): ptr 'a (* p + ofs *)
  alias { result.data with p.data }
  ensures { result.offset = p.offset + ofs }

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

Goals

- Straightforward extraction (trusted)
- Performance: no added complexity, no closures or indirections
- Predictable output
- Tradeoff: handle only a small, C-like fragment of WhyML

- loops, references
- records
- machine integers
- manual memory management
- polymorphism, abstract types
- higher order
- mathematical integers
- garbage collection
Exceptions and break/return

- Recognize break- and return-like patterns
- Reject other exceptions

```plaintext
exception B

try
  while ... do
    ...
    if (...) then raise B;
    ...
  done
with B → ()
end

exception R of t

let f (...) : t =
  ...
try
  ...
  raise (R e)
  ...
with R v → v
end
```
### Tuple return values

```ocaml
let f (x:int32) : (int32, int32) = x + 1, x + 2

let g (x:int32) =
    let (y, z) = f x in
    y + z
```

```c
struct __f_result {
    int32_t __field_0;
    int32_t __field_1;
};

struct __f_result f(int32_t x) {
    struct __f_result result;
    result.__field_0 = x + 1;
    result.__field_1 = x + 2;
    return result;
}

int32_t g(int32_t x) {
    int32_t y, z;
    struct __f_result struct_res;
    struct_res = f(x);
    y = struct_res.__field_0;
    z = struct_res.__field_1;
    return y + z;
}
```
Extracted code

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

```ocaml
let wmpn_cmp (x y: ptr uint64) (sz: int32): int32
= let ref i = sz in
  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 return 1
        else return (-1)
    done;
  0
```

R. Rieu-Helft
Verification of efficient C arithmetic algorithms with Why3

18/34
1. Introduction

2. Memory model and extraction

3. An algorithm: long division

4. WhyMP

5. Conclusion, perspectives
Long division: naïve algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

\[ a_0 \ a_1 \ \ldots \ a_{m-3} \ a_{m-2} \ a_{m-1} \]

dividend/partial remainder: length \( m \)

\[ d_0 \ \ldots \ d_{n-1} \]

normalized divisor: length \( n \)

\[ ? \ ? \ \ldots \ ? \ ? \ ? \]

quotient: length \( m-n \)
Long division: naïve algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

1. Estimate the most significant quotient limb (with a short division)

\[
\hat{q} = \frac{a_{m-2}a_{m-1}}{d_{n-1}}
\]

\[
\begin{array}{cccccc}
a_0 & a_1 & \ldots & a_{m-3} & a_{m-2} & a_{m-1} \\
\hline
d_0 & \ldots & d_{n-1} \\
\hline
? & ? & \ldots & ? & ?
\end{array}
\]

dividend/partial remainder: length \( m \)

normalized divisor: length \( n \)

quotient: length \( m-n \)
Long division: naïve algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

1. Estimate the most significant quotient limb (with a short division)
2. Multiply by the divisor, subtract the product from the dividend

\[
\begin{align*}
\bar{a}_0 \bar{a}_1 \ldots \bar{a}_{m-3} \bar{a}_{m-2} \bar{a}_{m-1} \\
\text{dividend/partial remainder: length } m
\end{align*}
\]

\[
\begin{align*}
d_0 \ldots d_{n-1} \\
\text{normalized divisor: length } n
\end{align*}
\]

\[
\begin{align*}
\hat{q} \\
\text{quotient: length } m-n
\end{align*}
\]

\[
\begin{align*}
\bar{a}'_0 \bar{a}'_1 \ldots \bar{a}'_{m-2} = \bar{a} - \hat{q} \times \bar{d}
\end{align*}
\]
Long division: na"ive algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

1. Estimate the most significant quotient limb (with a short division)
2. Multiply by the divisor, subtract the product from the dividend

\[
a'_0 a'_1 \ldots a'_{m-3} a'_{m-2} a_{m-1} = a - \hat{q} \times d
\]

if \( \hat{q} \) is right \( a'_{m-1} = 0 \)
Long division: naïve algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

1. Estimate the most significant quotient limb (with a short division)
2. Multiply by the divisor, subtract the product from the dividend
3. If the quotient is too large, adjust it

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

until it works...
Long division: naïve algorithm

One iteration of the main loop

Goal: compute most significant limb of the quotient

1. Estimate the most significant quotient limb (with a short division)
2. Multiply by the divisor, subtract the product from the dividend
3. If the quotient is too large, adjust it

\[
\begin{array}{ccccccc}
  a_0' & a_1' & \ldots & a_{m-3}' & a_{m-2}' & a_{m-1}
\end{array}
\]

\begin{align*}
\text{dividend/partial remainder: length } m \\
\begin{array}{cccc}
  d_0 & \ldots & d_{n-1}
\end{array}
\end{align*}

\begin{align*}
\text{normalized divisor: length } n \\
\begin{array}{cccc}
  ? & ? & \ldots & ? & \hat{q}
\end{array}
\end{align*}

\begin{align*}
\text{quotient: length } m-n
\end{align*}
Optimization: 3-by-2 division (Möller & Granlund 2011)

Goal: better estimate of the quotient, simplify adjustment step

\[
\hat{q} = \frac{a_{m-3}a_{m-2}a_{m-1}}{d_{n-2}d_{n-1}}
\]

\[
r_0r_1 = a_{m-3}a_{m-2}a_{m-1} - \hat{q} \cdot d_{n-2}d_{n-1}
\]

Adjustment: at most one step, and only if \( r_1 = 0 \) \( \Rightarrow \) very unlikely

same divisor at each iteration \( \Rightarrow \) 3-by-2 division uses a precomputed pseudo-inverse and no division primitive
Implementation trick: long subtraction

\[
dividend/partial \ remainder: \ length \ m
\]
\[
\hat{q} = \frac{a_{m-3}a_{m-2}a_{m-1}}{d_{n-2}d_{n-1}}
\]
\[
r_0r_1 = a_{m-3}a_{m-2}a_{m-1} - \hat{q} \cdot d_{n-2}d_{n-1}
\]

\[
\begin{array}{cccc}
a_0 & a_1 & \ldots & a_{m-3} & a_{m-2} & a_{m-1} \\
\hline
d_0 & \ldots & d_{n-2} & d_{n-1} \\
\end{array}
\]

\[
a'_{0}a'_{1} \ldots a'_{m-2} = a_{0} \ldots a_{m-3}a_{m-2}a_{m-1} - \beta^{m-n-1} \hat{q} \times d_{0} \ldots d_{n-2}d_{n-1}
\]

but we already have \( a_{m-3}a_{m-2}a_{m-1} - \hat{q} \times d_{n-2}d_{n-1} = r_0r_1 \)

\( \Rightarrow \) subtraction over length \( n - 2 \) instead of \( n \), then propagate borrow
Final algorithm

while (i > 0) do
    i ← i - 1;
    xp ← C.incr xp (-1);
    let xd = C.incr xp mdn in
    let xp1 = xp[1] in
    if [@extraction:unlikely] (x1 = dh && xp1 = dl) then ...
    else begin
        let xp0 = xp[0] in
        (q1, x1, x0) ← div3by2_inv x1 xp1 xp0 dh dl v;
        let cy = wmpn_submul_1 xd y (sy - 2) q1 in
        let cy1 = if (x0 < cy) then 1 else 0 in
        x0 ← sub_mod x0 cy;
        let cy2 = if (x1 < cy1) then 1 else 0 in
        x1 ← sub_mod x1 cy1;
        xp[0] ← x0;
        if [@extraction:unlikely] (cy2 \neq 0) then begin (* cy2 = 1 *)
            let c = wmpn_add_n_in_place xd y (sy - 1) in
            x1 ← add_mod x1 (add_mod dh c);
            q1 ← q1 - 1;
        end;
        qp ← C.incr qp (-1);
        qp[0] ← q1;
    end;
done;
Final algorithm

while (i > 0) do
    i ← i - 1;
    xp ← C.incr xp (-1);
    let xd = C.incr xp mdn in
    let xp1 = xp[1] in
    if [@extraction:unlikely] (x1 = dh && xp1 = dl) then ...
    else begin
        let xp0 = xp[0] in
        (q1, x1, x0) ← div3by2_inv x1 xp1 xp0 dh dl v;
        let cy = wmpn_submul_1 xd y (sy - 2) q1 in
        let cy1 = if (x0 < cy) then 1 else 0 in
        x0 ← sub_mod x0 cy;
        let cy2 = if (x1 < cy1) then 1 else 0 in
        x1 ← sub_mod x1 cy1;
        xp[0] ← x0;
        if [@extraction:unlikely] (cy2 ≠ 0) then begin (* cy2 = 1 *)
            let c = wmpn_add_n_in_place xd y (sy - 1) in
            x1 ← add_mod x1 (add_mod dh c);
            q1 ← q1 - 1;
        end;
        qp ← C.incr qp (-1);
        qp[0] ← q1;
    end;
done;

3-by-2 division
Final algorithm

```plaintext
while (i > 0) do
    i ← i - 1;
    xp ← C.incr xp (-1);
    let xd = C.incr xp mdn in
    let xp1 = xp[1] in
    if [@extraction:unlikely] (x1 = dh && xp1 = dl) then ...
    else begin
        let xp0 = xp[0] in
        (ql, x1, x0) ← div3by2_inv x1 xp1 xp0 dh dl v;
        let cy = wmpn_submul_1 xd y (sy - 2) ql in
        let cy1 = if (x0 < cy) then 1 else 0 in
        x0 ← sub_mod x0 cy;
        let cy2 = if (x1 < cy1) then 1 else 0 in
        x1 ← sub_mod x1 cy1;
        xp[0] ← x0;
        if [@extraction:unlikely] (cy2 ≠ 0) then begin (* cy2 = 1 *)
            let c = wmpn_add_n_in_place xd y (sy - 1) in
            x1 ← add_mod x1 (add_mod dh c);
            ql ← ql - 1;
        end;
        qp ← C.incr qp (-1);
        qp[0] ← ql;
    end;
done;
```

3-by-2 division

Shortened long subtraction
**Final algorithm**

```plaintext
while (i > 0) do
  i ← i - 1;
  xp ← C.incr xp (-1);
  let xd = C.incr xp mdn in
  let xp1 = xp[1] in
  if [@extraction:unlikely] (x1 = dh && xp1 = dl) then ...
  else begin
    let xp0 = xp[0] in
    (ql, x1, x0) ← div3by2_inv x1 xp1 xp0 dh dl v;
    let cy = wmpn_submul_1 xd y (sy - 2) ql in
    let cy1 = if (x0 < cy) then 1 else 0 in
    x0 ← sub_mod x0 cy;
    let cy2 = if (x1 < cy1) then 1 else 0 in
    x1 ← sub_mod x1 cy1;
    xp[0] ← x0;
    if [@extraction:unlikely] (cy2 ≠ 0) then begin (* cy2 = 1 *)
      let c = wmpn_add_n_in_place xd y (sy - 1) in
      x1 ← add_mod x1 (add_mod dh c);
      ql ← ql - 1;
    end;
    qp ← C.incr qp (-1);
    qp[0] ← ql;
  end;
done;
```

3-by-2 division

Shortened long subtraction

One-step adjustment

---

R. Rieu-Helft

Verification of efficient C arithmetic algorithms with Why3
Final algorithm

while (i > 0) do
    i ← i - 1;
    xp ← C.incr xp (-1);
    let xd = C.incr xp mdn in
    let xp1 = xp[1] in
    if [unlikely] (x1 = dh && xp1 = dl) then ...
    else begin
        let xp0 = xp[0] in
        (ql, x1, x0) ← div3by2_inv x1 xp1 xp0 dh dl v;
        let cy = wmpn_submul_1 xd y (sy - 2) ql in
        let cy1 = if (x0 < cy) then 1 else 0 in
        x0 ← sub_mod x0 cy;
        let cy2 = if (x1 < cy1) then 1 else 0 in
        x1 ← sub_mod x1 cy1;
        xp[0] ← x0;
        if [unlikely] (cy2 ≠ 0) then begin (* cy2 = 1 *)
            let c = wmpn_add_n_in_place xd y (sy - 1) in
            x1 ← add_mod x1 (add_mod dh c);
            ql ← ql - 1;
        end;
        qp ← C.incr qp (-1);
        qp[0] ← ql;
    end;
done;

3-by-2 division
Shortened long subtraction
One-step adjustment
Proof effort
div3by2_inv: 1kloc
long division: 2kloc
1 Introduction
2 Memory model and extraction
3 An algorithm: long division
4 WhyMP
5 Conclusion, perspectives
WhyMP

**Objectives**

- Verified C library
- Compatible with GMP
- Performances comparable to GMP, for numbers up to a certain size
- Contains a large subset of algorithms from the `mpn` and `mpz` layers
WhyMP

Objectives

- Verified C library
- Compatible with GMP
- Performances comparable to GMP, for numbers up to a certain size
- Contains a large subset of algorithms from the `mpn` and `mpz` layers

Challenges

- Understand the algorithms
- Preserve GMP’s implementation tricks
Noteworthy algorithms

Toom-Cook multiplication

- Divide-and-conquer multiplication algorithm in $O(n^k)$, $k \approx 1.58$
- Two mutually recursive variants:
  - Toom-2: split each operand in 2 parts ($\sim$ Karatsuba)
  - Toom-2.5: split large operand in 3 parts and small in 2
- Main challenge: aliasing
Noteworthy algorithms

**Toom-Cook multiplication**
- Divide-and-conquer multiplication algorithm in $O(n^k)$, $k \approx 1.58$
- Two mutually recursive variants:
  - Toom-2: split each operand in 2 parts (≈ Karatsuba)
  - Toom-2.5: split large operand in 3 parts and small in 2
- Main challenge: aliasing

**Modular exponentiation**
- Square-and-multiply exponentiation algorithm
- Montgomery reduction optimization: no division in the main loop
- Main challenge: formalization of mathematical concepts
Noteworthy algorithms

Square root of a 64-bit integer

- Hand-coded fixed-point arithmetic
- Newton iteration
- Converges in two steps for all inputs
- Main challenge: modeling fixed-point arithmetic
Noteworthy algorithms

**Square root of a 64-bit integer**
- Hand-coded fixed-point arithmetic
- Newton iteration
- Converges in two steps for all inputs
- Main challenge: modeling fixed-point arithmetic

**mpz layer**
- Wrapper around the *mpn* layer, keeps track of number signs and sizes
- User-facing layer of GMP
- Not much arithmetic, but challenging aliasing combinatorics
- Main challenge: custom memory model required
Proof effort

- 22000 lines of WhyML code
  - 8000 of programs
  - 14000 of spec and assertions
- almost only automated provers
- total proof replay time: \(\sim 1\) hr
- extracted C code: \(\sim 5000\) lines
- \(\sim 100\) functions, 50 of which are part of GMP’s API

<table>
<thead>
<tr>
<th>Function</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>1000</td>
</tr>
<tr>
<td>subtraction</td>
<td>1000</td>
</tr>
<tr>
<td>mul (naïve)</td>
<td>700</td>
</tr>
<tr>
<td>mul (Toom)</td>
<td>2400</td>
</tr>
<tr>
<td>division</td>
<td>4500</td>
</tr>
<tr>
<td>helper lemmas</td>
<td>300</td>
</tr>
<tr>
<td>reflection</td>
<td>1700</td>
</tr>
<tr>
<td>shifts</td>
<td>1000</td>
</tr>
<tr>
<td>square root</td>
<td>1600</td>
</tr>
<tr>
<td>modular exponentiation</td>
<td>1700</td>
</tr>
<tr>
<td>util</td>
<td>200</td>
</tr>
<tr>
<td>base conversions</td>
<td>2000</td>
</tr>
<tr>
<td>mpz</td>
<td>3600</td>
</tr>
</tbody>
</table>
Trusted code base

- Axioms, WhyML model of C
- Why3 verification condition computation
- Automated theorem provers
- Compilation from WhyML to C
- Handwritten arithmetic primitives
Arithmetic primitives

GMP uses hand-coded assembly primitives:

- for basic operations:
  - $64 \times 64 \rightarrow 128$ bit multiplication
  - 128 by 64 bit division

- for critical large integer routines:
  - same-size addition
  - $n$-by-1 multiplication

R. Rieu-Helft
Verification of efficient C arithmetic algorithms with Why3
Arithmetic primitives

GMP uses handcodded assembly primitives:
- for basic operations:
  - $64 \times 64 \rightarrow 128$ bit multiplication
  - $128$ by $64$ bit division
- for critical large integer routines:
  - same-size addition
  - $n$-by-1 multiplication

Options for WhyMP
- Trust the assembly primitives $\Rightarrow$ should we? which ones?
- Verified 64-bit C primitives $\Rightarrow$ much slower
- Compromise: handcodded C basic ops using 128-bit compiler support
Comparison with GMP ($n \times n$ multiplication)
Comparison with GMP ($n \times n$ multiplication)

The graph compares the performance of different algorithms for $n \times n$ multiplication, showing the runtime in microseconds ($\mu$s) on the y-axis and the number of limbs ($n$) on the x-axis. The algorithms compared include:

- Mini-GMP
- WhyMP without 128-bit ops
- GMP without assembly
- WhyMP
- GMP

The graph highlights the performance differences between these algorithms, with specific annotations for Toom_22 and Toom_33 methods.
Comparison with GMP ($n \times n$ multiplication)

- **Mini-GMP**
- **WhyMP without 128-bit ops**
- **GMP without assembly**
- **WhyMP**
- **WhyMP with assembly**
- **GMP**

**Toom_22**

**Toom_33**

**Competitive**

**X-axis:** n (limbs)

**Y-axis:** time (µs)
Introduction
Memory model and extraction
An algorithm: long division
WhyMP
Conclusion, perspectives
Verification of C programs with Why3

Contributions
- Memory model of the C language
- Straightforward extraction to C
- Works on more than GMP! (Contiki’s ring buffer, cursors…)
⇒ Idiomatic, correct-by-construction C programs verified with Why3

Perspectives
- Memory model improvements:
  - support for C stack allocation
  - better alias handling
- Formalization of the correctness of the extraction mechanism
WhyMP

- Compatible with GMP (50 exported functions)
- Reasonable performance
- Formally verified! ⇒ minor bug found in GMP
- Preserves most of GMP’s implementation tricks

What remains to be done

- Exhaustivity: implement missing operations
- Cryptography functions, number theory functions
- Assembly code verification

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