# On Construction of a Library of Formally Verified Low-level Arithmetic Functions*

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology
Central 2, 1-1-1, Umezono, Tsukuba, Ibaraki, Japan
reynald.affeldt at aist.go.jp

## ABSTRACT

Most information security infrastructures rely on cryptography, which is usually implemented with low-level arithmetic functions. The formal verification of these functions therefore becomes a prerequisite to firmly assess any security property. We propose an approach for the construction of a library of formally verified low-level arithmetic functions that can be used to implement realistic cryptographic schemes in a trustful way. For that purpose, we introduce a formalization of data structures for signed multi-precision arithmetic and we experiment it with formal verification of basic functions, using Separation logic. Because this direct style of formal verification leads to technically involved specifications, we also propose for larger functions to show a formal simulation relation between pseudo-code and assembly. This is illustrated with the binary extended gcd algorithm.

## Keywords

proof-assistant, Hoare logic, multi-precision, simulation

## 1. INTRODUCTION

Most information security infrastructures rely on cryptography, which is usually implemented with arithmetic functions, themselves implemented with low-level languages for performance reasons. The formal verification of these low-level arithmetic functions therefore becomes a prerequisite to firmly assess any security property.

We propose an approach for the construction of a library of formally verified low-level arithmetic functions that can be used to implement realistic cryptographic schemes in a trustful way. There exist experiments about formal verification of low-level unsigned multi-precision arithmetic (e.g., [7, 8, 14] for assembly using proof-assistants) and verification of high-level multi-precision arithmetic (e.g., [15] for a subset of Ada using, among other tools, the Isabelle/HOL proof-assistant). To the best of our knowledge, no comprehensive effort for a fully formalized library of low-level multi-precision arithmetic has ever been undertaken, in particular encompassing signed multi-precision arithmetic. In this paper, we aim at formally verifying in the Coq proof-assistant [11] a usable set of arithmetic functions written in assembly. We build on top of [14] that provides a framework for formal verification of SmartMIPS [4] (a MIPS variant for smartcards) programs using Separation logic [5] (an extension of Hoare logic to deal with pointers), together with several functions for unsigned multi-precision arithmetic.

Our first contribution is the formalization of data structures (Sect. 2) and the verification of basic functions (Sect. 3) for signed multi-precision arithmetic. Because experience showed that it leads to good performance, we choose to mimic the data structure used in the GNU Multi-Precision Arithmetic Library (GMP) [12]. In order to make an effective use of [14], we choose a layered approach where signed arithmetic is implemented on top of unsigned arithmetic. We experiment this formalization with several basic functions, including the formal verification of signed multi-precision subtraction. Here, verification is carried out in *direct-style*, i.e., by providing a Hoare triple (pre/post conditions) and applying Hoare rules, resorting to the *frame rule* of Separation logic for code composition.

The problem with this direct-style formal verification is that it leads to technically involved specifications. In particular, the verification of functions that call several other functions generates large intermediate subgoals that are difficult to manipulate formally, and ultimately this makes for specifications that are difficult to read. On the other hand, arithmetic functions are traditionally specified by pseudo-code (e.g., [3]). This is however at the price of some imprecision. Besides inaccuracies due to the absence of formal definitions (it is not rare to find wrong corner cases and initialization issues, see the errata of [3] for examples) that are in general eventually spotted and dealt with by programmers, there are more difficult issues that are left unspecified with pseudo-code, for example, concrete storage in the memory of the computer: this is the work of the programmer to provide concrete data structures and to make sure that they are adequate.

Our second contribution is to propose, as an alternative to direct-style verification, to show a formal simulation relation between pseudo-code and assembly (Sections 4 and 5),


*SAC'12* March 25-29, 2012, Riva del Garda, Italy.

so as to overcome the problems explained above. In this way, we end up with more readable specifications, akin to standard textbooks. We illustrate concretely this approach with the binary extended gcd algorithm (Sect. 6, experiment in progress), an important function in cryptography, that is used to compute inverses modulo, as in, say, ElGamal decryption. This approach of showing a formal simulation relation naturally allows for hand-written assembly (this issue is discussed in more depth with a pencil-and-paper formalization in [13]), which is often necessary, e.g., to use special instructions.

To avoid ambiguities, we display in this paper the Coq formalization (almost) as it is, using obvious non-ascii symbols and a few shortcuts (e.g., variables are universally quantified by default) to ease reading. The formalization is available online [16].

# 2. MULTI-PRECISION INTEGERS

We first formalize the data structures for multi-precision integers in assembly. We are given a type `reg` of registers that are put together into a register file (hereafter, *store*) of type `store.t`. The value held by register `rx` in store `st` is noted ⟦`rx`⟧`_st`: it is a finite-size integer that can be interpreted either as an unsigned integer (by the function `u2Z`) or as a signed integer (by the function `s2Z`). Sticking to Separation logic parlance, we call *heap* the memory of the computer, ranged over by `h` and of type `heap.t`. The heap is finite, of size $\beta = 2^{32}$ (we assume a 32-bit architecture).

## 2.1 Unsigned Multi-precision Integer

A multi-precision integer is encoded as an array of words (its *payload*), with the least significant word first. The length of the payload is kept in a dedicated register (Fig. 1). We formalize the fact that the value `val` is implemented by

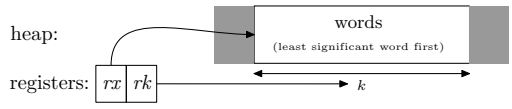


**Figure 1: An unsigned multi-precision integer**

an array of length `k` pointed to by register `rx` as follows:

```
0 Definition var_unsign k rx val st h : Prop :=
1  u2Z ⟦rx⟧_st + 4 * k < β  ∧  0 ≤ val < β^k  ∧
2  (rx ↦ Z2ints 32 k val) st h.
```

Line 1 specifies that the array does not "wrap around" the heap and that `val` can safely be encoded in base $\beta$. Line 2 is a Separation logic formula. The formulas of Separation logic (as well as the pre/post-conditions of Hoare triples hereafter) are *shallow-encoded* (as done in [14]), i.e., they have the type `Definition assert := store.t →heap.t →Prop`. Above, ↦ is the *mapsto* connective of Separation logic. Here, it specifies that the register `rx` points to the encoding of `val` (`Z2ints` converts an arbitrary-precision integer into a list of finite-size integers).

## 2.2 Signed Multi-precision Integer

We use sign-magnitude, with a special treatment for zero, for the data structure for signed multi-precision integers (Fig 2). This is similar to GMP ([12, Sect. 17.1]) except that we do not plan to do any reallocation yet (the `_mp_alloc` field

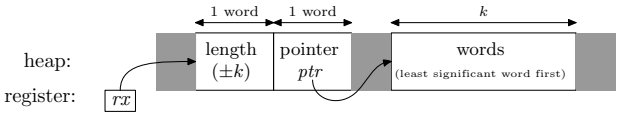


**Figure 2: A signed multi-precision integer**

in GMP). The first word of this data structure contains the size in words of the magnitude (as for the unsigned case, we call the magnitude *payload*). It is interpreted as a signed integer whose sign gives the sign of the encoded value, with the special case that the value zero is represented by having the size set to zero (in which case the payload is unused). The second word is a pointer to the payload. The fact that the value `val` is implemented by a signed multi-precision integer pointed to by register `rx` is formalized as follows:

```
0 Definition var_signed k rx val st h : Prop :=
1  u2Z ⟦rx⟧_st + 4 * 2 < β ∧
2  ∃ l, ptr, A. length A = k ∧
3   s2Z l = Zsgn(s2Z l) * k ∧
4   val = Zsgn(s2Z l) * Sum k A ∧
5   Zsgn(s2Z l) = Zsgn val ∧
6   u2Z ptr + 4 * k < β ∧
7   Zabs val < β^k ∧
8   (rx ↦ l :: ptr :: nil ⋆ ptr ↦ A) st h.
```

`Zsgn` is 0, 1, or −1, according to whether its argument is zero, positive, or negative. `Zabs` is the absolute value. `Sum k A` computes the value encoded by the first `k` finite-size integers of list `A` (it is the converse of `Z2ints`). Like the unsigned case, lines 1 and 6 avoid wrap-arounds and line 7 guarantees that the payload can safely be encoded in base $\beta$. Lines 2 and 3 formalize the relation between the length of the payload and the embedded length. Lines 4 and 5 formalize the relation between the value `val` and its encoding `A`. Observe that when the value is zero, the magnitude is unspecified, because it is supposed to be unused (same design principle as GMP). The Separation logic formula (line 8) has two conjuncts that correspond to disjoint heap areas (this is the intuitive meaning of the *separating conjunction* ⋆); the first conjunct contains a pointer to the second one. Finally, note that the length `k` of the magnitude has been made a parameter of the definition to simplify further technical developments.

# 3. VERIFICATION OF BASIC FUNCTIONS

We explain with an example how we formally prove the functional correctness of basic functions for signed arithmetic. Implementations are written in SmartMIPS assembly. We do not explain in detail the semantics here: used instructions are standard ones, their semantics can be found in [4], and we add comments in the assembly code for understanding. To ease reading, assembly constructs are underlined (instructions: `lw` (load word), etc.; structured control-flow[1]: `ifte`, etc.). The null register is noted `r0`, while general-purpose registers are parameters of definitions. We refer to the whole assembly language as the type `cmd`.

We display below the signed-unsigned addition, that adds in-place a signed multi-precision integer and an unsigned multi-precision integer (of the same length). It is implemented on top of unsigned functions (`multi_add` and `multi_sub`). The algorithm first sorts out the situations in which one of the argument is zero: when the second argument is zero,

[1] [14] provides certified compilation to labeled jumps.

nothing needs to be done besides clearing the overflow register (line 3); when the first argument is zero, it is enough to copy (function `copy`) the contents of the second argument (line 11). Otherwise, when the first argument is positive, it boils down to an unsigned multi-precision addition (line 15). Because of the special handling of zero, the situation where the first argument is strictly negative requires comparison between the two arguments. Upon equality, it is enough to set the length to zero (line 21); the (now unused) payload is left untouched but we still have a pointer to it. Otherwise, in-place unsigned subtraction, possibly coupled with negation (function `negate`), finishes the algorithm (lines 22 and 24).

```
0  Definition multi_add_s_u rk rx ry a0 a1 a2 a3 a4 a5 X :=
1   multi_is_zero rk ry a0 a1 a2 ;
2   ifte (bne a2 r0) (* y = 0 ? *)
3   (addiu a3 r0 0_16) (* y = 0 *)
4   (multi_add_s_u' rk rx ry a0 a1 a2 a3 a4 a5 X).
5
6  Definition multi_add_s_u' rk rx ry a0 a1 a2 a3 a4 a5 X :=
7   lw X 4_16 rx ; (* pointer to payload *)
8   pick_sign rx a0 a1 ;
9   ifte (bgez a1) (* 0 ≤ x ? *)
10  (ifte (beq a1 r0) (* x = 0 ? *)
11   (copy rk X ry a2 a3 a4 ; (* x = 0 *)
12    addiu a3 r0 0_16 ;
13    sw rk 0_16 rx)
14   (addiu a3 r0 1_16 ; (* x ≠ 0 *)
15    multi_add rk a3 ry X X a0 a1 a2 ;
16    mflo a3))
17  (multi_lt rk ry X a0 a1 a5 a2 a3 a4 ;
18   ifte (beq a5 r0) (* x ≤ y ? *)
19   (ifte (beq a2 r0) (* x = y ? *)
20    (addiu a3 r0 0_16 ; (* x = y *)
21     sw r0 0_16 rx)
22    (multi_sub rk ry X X a0 a1 a2 a3 a4 a5 ; (* x < y *)
23     negate rx a0) )
24   (multi_sub rk X ry X a0 a1 a5 a3 a2 a4)). (* x > y *)
```

For such basic functions, it is technically manageable to perform a proof in Hoare logic, using the frame rule of Separation logic to compose the called functions. Below follows the Hoare triple (notation: $\{\cdot\}\cdot\{\cdot\}$ ) for the functional correctness of the addition above. The `nodup` predicate specifies a list of pairwise distinct elements (here, registers). The precondition essentially specifies that the heap contains a signed integer and an unsigned integer (payloads `A` and `B`) (line 5). The most informative part of the postcondition is its last conjunct (lines 13 and 14) that specifies that the resulting length `l'` and payload `A'` are indeed the result of the addition. Note that because of potential overflow (stored in $a_3$), the result may not necessarily be a signed multi-precision integer as defined in Sect. 2.

```
0  Lemma multi_add_s_u_triple :
1   nodup(rk, rx, ry, a0, a1, a2, a3, a4, a5, X, r0) →
2   0 < k < 2^31 → (* not the weird number *)
3  { fun s h ⇒ ⟦rx⟧_s = vx ∧ ⟦ry⟧_s = vy ∧
4    u2Z ⟦rk⟧_s = k ∧
5    (var_signed k rx A ⋆ var_unsign k ry B) s h }
6   multi_add_s_u rk rx ry a0 a1 a2 a3 a4 a5 X
7  { fun s h ⇒ ∃ A', l', ptr, length A' = k ∧
8    ⟦rx⟧_s = vx ∧ ⟦ry⟧_s = vy ∧
9    s2Z l' = Zsgn (s2Z l') * k ∧
10   Zsgn (s2Z l') = Zsgn (A + B) ∧
11   (rx ↦ l' :: ptr :: nil ⋆ ptr ↦ A' ⋆
12    var_unsign nk ry B) s h ∧ u2Z (⟦a3⟧_ s) ≤ 1 ∧
13   Zsgn (s2Z l') * (Sum k A' + (u2Z ⟦a3⟧_ s) * β^k)
14    = A + B  }.
```

Similarly to the above function, we have formally verified the signed-unsigned subtraction and used these two functions to formally verify in-place signed subtraction (see [16]).

# 4. VERIFICATION USING SIMULATION

As seen in Sect. 3, Hoare triples for arithmetic functions can become technically involved, and it is questionable whether verification scales to larger functions. This is why we experiment formal verification of larger functions by providing a pseudo-code version of the verification target together with an assembly version and by showing a formal simulation relation between both. For that purpose, we introduce a generic definition of simulation (Sect. 4.1) that we instantiate with a relation between arbitrary-precision and multi-precision integers (Sect. 4.2).

## 4.1 Forward Simulation

We are given a type `pstore` for stores with variables holding arbitrary-precision integers. The type of the relations between `pstores` and the states of the `cmd` assembly language is defined as follows:

```
Definition relat := pstore →store.t →heap.t →Prop.
```

We are also given a pseudo-code programming language `pcmd`. Let us note `Some(s, h)` ≻ `x` ⇀ `Some(s', h')` the (big-step) operational semantics of both the `cmd` assembly and the `pcmd` pseudo-code languages: starting from state `(s, h)`, execution of the program `x` leads to the state `(s', h')` (`None` models error states). Knowing which operational semantics we are referring to will be obvious from the context; in particular, the heap is always $\epsilon$ (empty) in the case of pseudo-code.

Given the pseudo-code `p` and the assembly program `c`, the simulation that preserves the relation `R` under initial conditions $P_0$ is defined as follows:

```
Definition fwd_sim (R P0 : relat) p c :=
 ∀ s st h, R s st h →P0 s st h →
 ∀ s', Some (s,ϵ) ≻ p ⇀ Some (s',ϵ) →
 ∃ st', h'. Some (st,h) ≻ c ⇀ Some (st',h') ∧
  R s' st' h'.
```

This is a forward simulation in the sense of [10] (it is biased towards imperative programs and departs from the definitions in the theory of automata [1]). A definition similar to `fwd_sim` called "correspondence" can also be found in [9]. Once forward simulation is proved, it becomes possible to transport formally correctness from the pseudo-code to assembly, thus effectively reducing the proof of correctness of the assembly to a simulation proof and the proof of correctness of the pseudo-code (see [16]).

Reasoning with forward simulation calls for several lemmas, such as lemmas akin to Hoare logic weakening and strengthening, and, more importantly *composition lemmas*. For example, the following composition lemma shows that simulation of a sequence of instructions can be broken down to simulations of each part:

```
Lemma fwd_sim_seq : ∀ R p p' c c' P Q,
 rela_hoare P Q p c →
 fwd_sim R P p c → fwd_sim R Q p' c' →
 fwd_sim R P (p ; p') (c ; c').

Definition rela_hoare (P Q : relat) p c :=
 ∀ s st h, P s st h →
 ∀ s', Some (s,ϵ) ≻ p ⇀ Some (s',ϵ) →
 ∀ st' h', Some (st,h) ≻ c ⇀ Some (st',h') →
   Q s' st' h'.
```

Composition lemmas typically have side-conditions. For example, in the case of the lemma `fwd_sim_seq`, one needs to prove the propagation of initial conditions; the latter is here expressed by the side-condition `rela_hoare P Q p c`, using relational Hoare logic [6].

We have proved similar composition lemmas for while-loops and structured branching. They are a bit more involved because requiring a definition of simulation between boolean expressions of the pseudo-code and assembly. Also, as can be expected, simulation of while-loops requires, as a side-condition, an invariant about propagation of initial conditions (see [16]).

## 4.2 Instantiation for Signed Arithmetic

We now define concretely the relation between, on the one hand, a store for pseudo-code (where variables contain arbitrary-precision integers) and a state of assembly (where registers point to multi-precision integers encoding the same values as the variables). For that purpose, we first introduce a type for multi-precision integer. A multi-precision integer is either `unsigned`, implemented by two registers, one for the length and one for the pointer to the payload, or `signed`, implemented by the length of the payload and a pointer to the (header of the) data structure (we record explicitly the size of signed integers to simplify the formalization):

```
Inductive mint : Type :=
  unsign of reg & reg | signed of nat & reg.
```

A pseudo-code store and an assembly state are related by `var_mint x mx` when the variable `x` contains the same value as the one encoded by the multi-precision integer `mx`:

```
Definition var_mint x mx : relat := fun s st h ⇒
 match mx with
  | unsign rk rx ⇒
      var_unsign (u2Z ⟦rk⟧_st) rx ⟦x⟧_s st h
  | signed k rx ⇒ var_signed k rx ⟦x⟧_s st h
 end.
```

A pseudo-code store and an assembly state are related by `state_mint d`, where `d` is an association list, when all the arbitrary-precision integer variables in the domain of `d` are implemented by the corresponding multi-precision integers in the codomain of `d`:

```
Definition state_mint d : relat := fun s st h ⇒
 (∀ x mx, get x d = Some mx →
   var_mint x mx s st (heap_mint mx st h)) ∧
 (∀ x y, x ≠ y → ∀ mx my,
   get x d = Some mx → get y d = Some my →
   heap_mint mx st h ⊥ heap_mint my st h).
```

The second conjunct of this definition ensures that multi-precision integers are disjoint in the heap: `heap_mint mx st h` cuts out exactly that part of the heap that encodes the multi-precision integer `mx`. This relation between pseudo-code variables and multi-precision integers is technically involved compared to relations used in proving correctness of compiler phases, where the gap between data on both sides is typically smaller than here.

# 5. SIMULATION FOR BASIC FUNCTIONS

In order to verify larger functions using simulation, we first need to equip basic functions, such as multi-precision signed addition, with simulation proofs.

## 5.1 Approach for Simulation Proofs

We found it easier in practice to split the proof of forward simulation into a proof of termination and a proof of *partial forward simulation* as defined below:

```
Definition pfwd_sim (R P₀ : relat) p c :=
 ∀ s st h, R s st h →P₀ s st h →
 ∀ s', Some (s,ϵ) ≻ p ↠ Some (s',ϵ) →
 ∀ st' h', Some (st,h) ≻ c ↠ Some (st',h') →
  R s' st' h'.

Definition safe_termination (R : relat) (c : cmd)
 := ∀ s st h, R s st h →
  ∃ s'. Some (st, h) ≻ c ↠ Some s'.
```

Put together, `pfwd_sim` and `safe_termination` imply forward simulation. These two proofs make use of the Hoare triple of the assembly program at hand. Concretely, proofs of `safe_termination` are traditional termination proofs (typically by exhibiting a variant) after which one shows that the final state is not an error state by using the Hoare triple. Proofs of `pfwd_sim` are illustrated below.

## 5.2 Simulation for Multi-precision Addition

Intuitively, the proof of partial forward simulation for the addition defined in Sect. 3 makes explicit the condition under which `multi_add_s_u` behaves as the addition for arbitrary-precision integers (see Fig. 3). More precisely, the formal

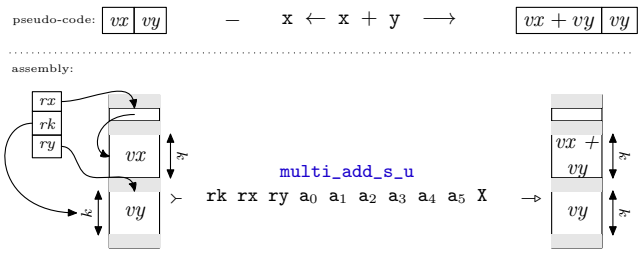


**Figure 3: Simulation for signed-unsigned addition**

statement below specifies that the executions of `x ←x + y` ($0 \leq$ `y`) and of `multi_add_s_u rk rx ry ...` preserve the relation `state_mint (x ↦signed L rx ⊎y ↦unsign rk ry ⊎d)` (lines 7–8), for any association list `d` with no register in common with the code (lines 3–6), provided the initial conditions from line 9 hold. In particular, these initial conditions specify that register `rk` contains the length `L` (line 10) and rule out potential overflows (line 12).

```
0  Lemma pfwd_sim_multi_add_s_u :
1   nodup(x, y) →
2   nodup(rk, rx, ry, a₀, a₁, a₂, a₃, a₄, a₅, X, r0) →
3   disj (mints_regs (cdom d))
4     (a₀ :: a₁ :: a₂ :: a₃ :: a₄ :: a₅ :: X :: nil) →
5   x ∉ dom d → signed L rx ∉ cdom d →
6   y ∉ dom d → unsign rk ry ∉ cdom d →
7   pfwd_sim (state_mint
8     (x ↦ signed L rx ⊎ y ↦ unsign rk ry ⊎ d))
9   (fun s st _ ⇒ ⟦rk⟧_st ≠ 0₃₂ ∧
10    u2Z ⟦rk⟧_st < 2^31 ∧  L = u2Z ⟦rk⟧_st ∧
11    Zabs ⟦x⟧_s < β^L ∧ 0 ≤ ⟦y⟧_s < β^L ∧
12    Zabs (⟦x⟧_s + ⟦y⟧_s) < β^L)
13  (x ← x + y)
14  (multi_add_s_u rk rx ry a₀ a₁ a₂ a₃ a₄ a₅ X).
```

# 6. BINARY EXTENDED GCD ALGORITHM

We apply the approach introduced in Sect. 4 to the binary extended gcd algorithm. This simulation proof makes use of simulation proofs for basic functions such as the one explained in Sect. 5.

## 6.1 Pseudo-code and Hoare Triple

The binary extended gcd algorithm is an extension of the Euclid algorithm: it combines the extended gcd algorithm, that computes the gcd of two integers together with the integers that satisfy the corresponding Bézout identity, with the binary gcd algorithm, that computes the gcd efficiently by replacing multi-precision divisions with shifts. We take the pseudo-code algorithm from authoritative literature ([2, p. 646]):

```
Definition begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  g ← 1 ;
  prelude u v g ;
  init u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
  while (t3 ≠ 0) (
    while (t3 % 2 = 0)
      halve u v t1 t2 t3 ;
    reset u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
    subtract u v u1 u2 u3 v1 v2 v3 t1 t2 t3).
```

To give an idea of the role of each function, here follows a rough explanation (see [2] for the precise pseudo-code and accurate explanations). `prelude` halves the inputs as much as possible, recording the number of iterations in `g`. `init` initializes the variables $u_i$, $v_i$, and $t_i$. `halve` tries to halve the variables $t_i$. (By the way, Sect. 6.2 provides the code of `halve` for illustrative purposes.) `reset` updates the variables $u_i$ or $v_i$ using the variables $t_i$. `subtract` updates the variables $t_i$ using $u_i - v_i$. Here follows the correctness statement in the form of a Hoare triple:

```
0 Lemma begcd_triple :
1  nodup(g, u, v, u1, u2, u3, v1, v2, v3, t1, t2, t3) →
2  0 < vu → 0 < vv →
3  { fun s h ⇒ uv_init vu vv u v s }
4    begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3
5  { fun s h ⇒ Zgcd vu vv = ⟦g⟧_s * ⟦u3⟧_s ∧
6    vu * ⟦u1⟧_s + vv * ⟦u2⟧_s = ⟦g⟧_s * ⟦u3⟧_s ∧
7    uivi_bounds u v u1 v1 u2 v2 u3 v3 s ∧
8    ti_bounds u v t1 t2 t3 s }.
```

In the precondition, the `uv_init` predicate specifies that the variables `u` and `v` are initialized with the ghost variables `vu` and `vv`. Regarding the postcondition, it is important to observe that this lemma is not only about functional correctness (i.e., the fact that this program does implement the extended gcd—lines 5–6) but also proves that the integers remain bounded by the program inputs. Ensuring this fact is the role of the `uivi_bounds` and `ti_bounds` predicates. (To be more precise, they specify that: $u_1$,$v_1$,$u_3$,$v_3$,$t_1$ are positive and bounded by `vv`; $u_2$,$v_2$,$t_2$ are negative and bounded by $-$`vv`; and $t_3$ lies between $-$`vv` and `vu`.)

## 6.2 From Pseudo-code to Assembly

The assembly code that we verify has the same control-flow structure as the pseudo-code, thus allowing the use of the composition rules discussed in Sect. 4.1. Since all the variables of the binary extended gcd algorithm lie between $-$`vv` and `vu`, we assume that the payload of all multi-precision integers is stored into the same amount of words, and that this value is stored in register `rk`. We illustrate the assembly implementation with the function `halve`:

```
Definition halve u v t1 t2 t3 :=
ifte (t1 % 2 = 0 && t2 % 2 = 0) thendo
 t1 ← t1 / 2 ; t2 ← t2 / 2 ; t3 ← t3 / 2
elsedo
 t1 ← (t1 + v) / 2 ; t2 ← (t2 − u) / 2 ; t3 ← t3 / 2.
```

Assembly is produced by mapping each pseudo-code instruction with a function such that there is an adequate simulation between both. For example, the pseudo-code addition is mapped to `multi_add_s_u`, the corresponding simulation being the one provided in Sect. 5:

```
Definition multi_is_even_s_and rx ry a0 a1 a2 :=
 multi_is_even_signed rx a0 a1 ;
 multi_is_even_signed ry a0 a2 ;
 and a0 a1 a2. (* NB: bitwise logical and *)

Definition halve_mips
  rk ru rv rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 :=
 multi_is_even_s_and rt1 rt2 a0 a1 a2 ;
 ifte (bne a0 r0)
 (multi_div2_s rt1 a0 a1 a2 a3 a4 ;
  multi_div2_s rt2 a0 a1 a2 a3 a4 ;
  multi_div2_s rt3 a0 a1 a2 a3 a4)
 (multi_add_s_u rk rt1 rv a0 a1 a2 a3 a4 a5 a6 ;
  multi_div2_s rt1 a0 a1 a2 a3 a4 ;
  multi_sub_s_u rk rt2 ru a0 a1 a2 a3 a4 a5 a6 ;
  multi_div2_s rt2 a0 a1 a2 a3 a4 ;
  multi_div2_s rt3 a0 a1 a2 a3 a4).
```

## 6.3 Verification of the BEGCD

We are now in a position to verify the binary extended gcd algorithm by showing a formal simulation relation. The verification goal is displayed in Fig. 4. Regarding the initial condition, `uv_init` has already been explained in Sect. 6.1; the predicate `uv_bound` establishes the link between the pseudo-code inputs and the length of the payload of multi-precision integers:

```
Lemma begcd_simu :
nodup(g, u, v, u1, u2, u3, v1, v2, v3, t1, t2, t3) →
nodup(rk, rg, ru, rv, ru1, ru2, ru3, rv1, rv2, rv3,
 rt1, rt2, rt3, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, r0)
 → 0 < vu → 0 < vv →
fwd_sim (state_mint (g ↦ unsign rk rg ⊎
  u ↦ unsign rk ru ⊎ v ↦ unsign rk rv ⊎
  u1 ↦ signed L ru1 ⊎ u2 ↦ signed L ru2 ⊎
  u3 ↦ signed L ru3 ⊎ v1 ↦ signed L rv1 ⊎
  v2 ↦ signed L rv2 ⊎ v3 ↦ signed L rv3 ⊎
  t1 ↦ signed L rt1 ⊎ t2 ↦ signed L rt2 ⊎
  t3 ↦ signed L rt3))
 (fun s st h ⇒ uv_init vu vv u v s ∧
               uv_bound rk st u v s L)
 (begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3)
 (begcd_mips rk rg ru rv ru1 ru2 ru3 rv1 rv2 rv3
             rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9).
```

**Figure 4: Simulation for the binary extended gcd algorithm**

```
0 Definition uv_bound rk st u v s L :=
1  0 < u2Z ⟦rk⟧_st < 2^31 ∧ L = u2Z ⟦rk⟧_st ∧
2  0 < ⟦u⟧_s < β^(L−1) ∧ 0 < ⟦v⟧_s < β^(L−1).
```

Line 1 specifies that the length of the payload of the multi-precision integers is stored into `L` words. Line 2 specifies that the input values are strictly smaller that $\beta^{L-1}$. The latter condition makes it possible to guarantee that there is

no overflow during execution. Indeed, as we have explained in Sect. 6.1, the values of variables in the pseudo-code are all bounded by the inputs during execution (to be precise, after each invocation of `init`, `halve`, `reset` and `subtract`), and this fact transports to the assembly code through the `state_mint` relation. Since this is a non-trivial part of the proof of correctness of `begcd` (consider for example potential overflow in the `subtract` function), it is very satisfactory to be able to avoid dealing with this issue directly at the assembly level thanks to the simulation relation.

# 7. CURRENT LIBRARY STATUS

Except from parts of the simulation of the binary extended gcd algorithm, all the proofs discussed in this paper have been formalized in the Coq proof-assistant. More precisely, we have extended [14] with assembly implementations and correctness proofs for (1) unsigned arithmetic (zero initialization, halving, doubling, array copy, parity test, equality test against zero), and (2) signed arithmetic (sign testing, negation, in-place signed-unsigned addition, in-place signed-unsigned subtraction, in-place signed subtraction). From [14], we inherit unsigned addition (with its in-place variant), unsigned subtraction (with in-place variants), multiplication, Montgomery multiplication/squaring/exponentation, and generic unsigned comparison ("equal, less or greater than"). We have equipped some of the operations above with simulation proofs (precisely, unsigned zero initialization, unsigned halving and doubling, in-place signed-unsigned addition and subtraction, generic unsigned comparison, as well as other, more ad-hoc, testing functions). Other functions have been implemented in assembly but their correctness and simulation proofs are just axiomatized. We claim that this is just a problem of manpower because these functions are essentially variants of operations already verified or simulated (signed variants, variants that are not in-place, etc.). Regarding the application to the binary extended gcd algorithm, we have completely formalized the simulation proofs for `begcd` (including `prelude`, `init`, `halve`, and `reset`), modulo the axiomatizations explained above.

# 8. CONCLUSION

We proposed an approach for the construction of a library of formally verified low-level arithmetic functions. First, we introduced a formalization of data-structures for signed multi-precision arithmetic that we illustrated with a direct-style Separation logic proof for a basic function using signed integers. Second, we proposed an approach based on simulation to deal with larger functions. It consists in showing a formal simulation relation between the pseudo-code and the assembly. This is illustrated with an assembly implementation of the binary extended gcd algorithm. As a consequence of this approach, the pseudo-code can serve as a specification of the implementation, as this is usually done in standard textbooks, and as it is expected from programmers.

Formal verification of the axiomatized part of the library is current work. Given the current scale of the whole library (around 30,000 lines of scripts), we plan to work towards completion in a steady way, by providing more lemmas and improving the quality of automation. In the simulation proofs, we made the hypothesis that multi-precision integers share the same length, but since we use pointers in the data structure for signed integers, we can extend this work to deal with varying-size integers. We plan to do so by connection with a formal model for the C programming language that we have been developing in the context of another project, so that dynamic allocation can be provided by C's malloc.

# 9. REFERENCES


[1] N. A. Lynch and F. W. Vaandrager. Forward and Backward Simulations Part I: Untimed Systems. *Inform. Comput.*, 121(2):214–233, 1995.

[2] D. E. Knuth. *The Art of Computer Programming.* Vol. 2, 3rd edition. Addison-Wesley, 1997.

[3] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* 5th printing. CRC Press, 2001.

[4] MIPS Technologies. *MIPS32 4KS Processor Core Family Software User's Manual.* 2001.

[5] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In 17th IEEE Symp. on Logic in Computer Science Proceedings, pages 55–74. IEEE Computer Society, 2002.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages Proceedings, pages 14–25. ACM Press, 2004.

[7] R. Affeldt and N. Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In 11th Annual Asian Computing Science Conf., vol. 4435 of LNCS, pages 346–360. Springer, Jan. 2008.

[8] M. Myreen and M. Gordon. Verification of Machine Code Implementations of Arithmetic Functions for Cryptography. In TPHOLs Emerging Trends Proceedings. Technical report 364/07. Department of Computer Science, University of Kaiserslautern, 2007.

[9] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the Gap: A Verification Framework for Low-level C. In 22nd Intl. Conf. on Theorem Proving in Higher Order Logics, vol. 5674 of LNCS, pages 500–515. Springer, 2009.

[10] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, Dec. 2009.

[11] *The Coq Proof Assistant: Reference Manual.* Ver. 8.3. `http://coq.inria.fr`. INRIA, 2010.

[12] *The GNU Multi Precision Arithmetic Library.* Edition 5.0.2. `http://gmplib.org/`. 2011.

[13] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages Proceedings, pages 133–146. ACM Press, 2011.

[14] R. Affeldt, D. Nowak, and K. Yamada. Certifying Assembly with Formal Security Proofs: the Case of BBS. *Sci. Comput. Program.* In press. `doi:10.1016/j.scico.2011.07.003`.

[15] S. Berghofer. Verification of Dependable Software using SPARK and Isabelle. In 6th Intl. Wksp. on Systems Software Verification Proceedings, pages 48–65. 2011.

[16] R. Affeldt, A Library for Formal Verification of Low-level Programs, `http://staff.aist.go.jp/reynald.affeldt/coqdev`.