Noname manuscript No.
(will be inserted by the editor)

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

Reynald Affeldt

the date of receipt and acceptance should be inserted later

Abstract Arithmetic functions are used in many important computer programs such as computer algebra systems and cryptographic software. The latter are critical applications whose correct implementation deserves to be formally guaranteed. They are also computation-intensive applications, so that programmers often resort to low-level assembly code to implement arithmetic functions. We propose an approach for the construction of a library of formally verified low-level arithmetic functions. To build our library, we first introduce a formalization of data structures for signed multi-precision arithmetic in low-level programs. We use this formalization to verify the implementation of several primitive arithmetic functions using Separation logic, an extension of Hoare logic to deal with pointers. Since 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 style of verification is illustrated with a concrete implementation of the binary extended gcd algorithm.

1 Introduction

Arithmetic functions are used in many important computer programs such as computer algebra systems and cryptographic software. The latter are critical applications whose correct implementation deserves to be formally guaranteed. For example, formal verification of arithmetic functions is a prerequisite to firmly assess the security properties of cryptographic software. Programs using arithmetic functions also turn out to be computation-intensive, so that in practice programmers often resort to low-level assembly code to implement arithmetic functions. For example, the good performance of the GNU Multi-Precision Arithmetic Library [31] comes mostly from its low-level part being optimized with assembly code [9, Sect. 5.1.2].

We propose an approach for the construction of a library of formally verified low-level arithmetic functions. There already exist experiments about formal verification of low-level unsigned multi-precision arithmetic (for assembly using proof-assistants [1,2,23]) and verification of high-level multi-precision arithmetic (for a subset of Ada using the Isabelle/HOL proof-assistant [7]), see Sect. 7 for details. Yet, to the best of our knowledge, no 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 work, we aim at providing means for formal verification in the Coq proof-assistant [29] of arithmetic functions written in assembly. Concretely, we experiment with Hoare logic-based verification (see "first contribution" below) and relational verification between pseudo-code and actual implementations ("second contribution" below). We build on top of an existing framework [1, 2] for formal verification of SmartMIPS (a MIPS variant for smartcards) programs [21] using Separation logic [26] (an extension of Hoare logic that allows for local reasoning in the presence of pointers), a framework that already comes together with several functions for unsigned multi-precision arithmetic.

Our first contribution is the formalization of data structures for signed multi-precision arithmetic and the

To appear in 2013 in Innovations in Systems and Software Engineering, Nasa/Springer. A preliminary version of this work appeared in the proceedings of the 27th ACM SIGAPP Symposium On Applied Computing (SAC 2012), Software Verification and Testing Track [3].

National Institute of Advanced Industrial Science and Technology, Central 2, 1-1-1, Umezono, Tsukuba, Ibaraki, Japan

verification of primitive arithmetic functions. We choose to mimic the data structure for signed integer arithmetic used in the GNU Multi-Precision Arithmetic Library [31] (hereafter, GMP). GMP is the main reference for arbitrary-precision arithmetic and it is known to deliver good performance (indeed, using GMP routines is recognized as a way to improve the performance of other implementations of multi-precision arithmetic—see [9, Sect. 5.1.1] or "Tips for Getting the Best Performance out of NTL" [27]). We experiment this formalization with the verification of several primitive multi-precision functions, such as signed subtraction, signed halving, etc. In order to simplify our development, we choose a layered approach (functions for signed multi-precision arithmetic are implemented using functions for unsigned multi-precision arithmetic) and restrict ourselves to functions parametrized by the size of the integer size (the verification of arithmetic functions that dynamically determine or extend the integer size is deferred to future work). Verification of these primitive functions is carried out in *direct-style*, i.e., by providing a Hoare triple (pre/post-conditions) and applying Hoare rules, using the *frame rule* of Separation logic to compose code.

The problem with direct-style formal verification is that it leads to technically involved specifications. In particular, the verification of functions that call several other functions leads to large intermediate subgoals that are difficult to manipulate formally, and ultimately this leads to specifications that are difficult to read. This comes in contrast to handbooks where arithmetic functions are traditionally specified using pseudo-code (e.g., [20]). This is however at the price of 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 [20] 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, the delicate task of providing concrete data structures and to make sure that they are adequate is left to the programmer alone, though it is well-known that "in many cases the intellectual heart of a program lies in the ingenious choice of data representation rather than in the abstract algorithm" [25, p. 298].

Our second contribution is to propose, as an alternative to direct-style verification, to show a formal simulation relation between pseudo-code and assembly, so as to overcome the issues raised by direct-style verification as explained above. In this way, we end up with more readable specifications in pseudo-code, akin to standard handbooks. Exhibiting a formal simulation relation shows that the assembly is "as sound as" the pseudo-code, the correctness of the latter being also formally provable. We illustrate concretely this approach with the binary extended gcd algorithm. An important application of this algorithm is the computation of modular multiplicative inverses, that are pervasive in cryptography, e.g., in ElGamal decryption [12]. This approach of showing a formal simulation relation has the additional advantage of naturally allowing for hand-written assembly (this issue is discussed in more depth with a pencil-and-paper formalization in [14]). Hand-written assembly is often necessary, for example, to use special assembly instructions (see Sect. 3.1 for examples of such instructions in the case of SmartMIPS), or more generally to improve performance.

About notations in this paper To avoid ambiguities, we display the Coq formalization as it is, using obvious non-ascii symbols and a few shortcuts (in particular, variables are universally quantified by default) to ease reading. To clear up any ambiguity, the complete formalization is available online [5].

Outline In Sect. 2, we formalize the data structures for signed multi-precision integers using Separation logic. In Sect. 3, we explain how to verify the correctness of assembly implementations of signed multi-precision arithmetic. In Sect. 4, we formalize the notion of forward simulation and explain how to carry out simulation proofs between pseudo-code and assembly programs. In Sect. 5, we explain how to prove formally simulation for primitive arithmetic instructions. In Sect. 6, we show how to prove simulation for a larger example, namely the binary extended gcd algorithm. We review related work in Sect. 7 and conclude in Sect. 8.

2 Multi-precision integers

In this section, we formalize the data structures for signed multi-precision integers in assembly using Separation logic. For this purpose, we first introduce our formal model of execution states of assembly programs (Sect. 2.1). Second, we give a brief overview of Separation logic and its formal encoding in Coq (Sect. 2.2). We then explain in turn the formalization of unsigned multi-precision integers (Sect. 2.3) and signed multi-precision integers (Sect. 2.4).

2.1 Execution states of assembly programs

This section explains how we formalize the execution states of assembly programs. (This presentation of assembly programs will be completed in Sect. 3.1 where

we further explain the syntax and the semantics of assembly programs.)

The assembly programming language we are dealing with is SmartMIPS, a superset of the MIPS assembly programming language [21]. Informally, an execution state of a SmartMIPS program comprises the contents of registers and of the memory. They both consist of *finite-size integers* (of type `int n` when the underlying bit representation is `n`-bit long). A finite-size integer can be interpreted either as an unsigned integer (by the function `u2Z`) or as a signed integer (by the function `s2Z`), according to the two's complement notation.

The register file is formalized as a *store* (of type `store.t`): it is a finite map from registers to finite-size integers. There are 32 general-purpose registers (hereafter ranged over by `rx`, $\mathtt{a}_i$, etc.) that hold 32-bit integers. Among them, there is in particular a special register `r0` constantly holding 0_{32}. There is also the MIPS *multiplier*, an additional set of three registers (LO, HI, ACX) dedicated to arithmetic computations. The value held by the general-purpose register `rx` in the store `s` is noted $[\![\mathtt{rx}]\!]_{\mathcal{R}}\ \mathtt{s}$.

The flat memory of the computer is formalized as a *heap* (of type `heap.t`): a finite map from natural numbers to 32-bit integers. (We restrict ourselves to assembly programs that address memory by words, as customary with MIPS.) The heap is finite, of size $\beta = 2^{32}$.

Formally, a *state* of execution of an assembly program is a pair of a store and a heap, as defined above.

2.2 Overview of our formalization of Separation logic

Separation logic [26] is an extension of Hoare logic that deals elegantly with pointers while supporting local reasoning (see the end of this section).

The assertions that appear in the pre- and post-conditions of Separation logic triples are *shallow encoded* in our development, i.e., they are formalized as functions from states to Prop, the sort of propositions in Coq:

```
Definition assert := store.t → heap.t → Prop.
```

The simplest assertion is `emp`, that holds when the heap is empty, regardless of the store:

```
Definition emp : assert :=
  fun s h ⇒ h = heap.emp.
```

The assertion that is the most characteristic of Separation logic is the *separating conjunction*. Given two assertions `P` and `Q`, `P ⋆ Q` holds when the heap of the state can be split into disjoint heaps such that `P` and `Q` hold respectively. This is formalized by the following definition:

```
Definition con P Q : assert := fun s h ⇒
  ∃ h₁ h₂, h₁ ⊥ h₂ ∧ h = h₁ ⊎ h₂ ∧ P s h₁ ∧ Q s h₂.
```

The most primitive assertion of Separation logic is the *mapsto* formula that specifies individual memory cells. Let us assume that we are given a language of expressions ranged over by `e` and an evaluation function from stores to finite-size integers (notation: $[\![\ \mathtt{e}\ ]\!]_{\mathcal{E}}\ \mathtt{s}$). The mapsto formula $\mathtt{e} \overset{1}{\mapsto} \mathtt{e'}$ holds when the heap consists exactly of the memory cell that contains the word `e'` and whose address is `e` (aligned on a word-boundary). This is formalized by the following definition:

```
Definition mapsto e e' : assert := fun s h ⇒
  ∃ p, u2Z ([[ e ]]ℰ s) = 4 * p ∧
  h = heap.sing p ([[ e' ]]ℰ s).
```

We extend the mapsto formula to deal with contiguous memory cells. The formula $\mathtt{e} \mapsto \mathtt{l}$ holds when the heap consists exactly of the contiguous memory cells that contain the words of the list `l` whose head is pointed to by `e`. This is formalized by the following definition:

```
Fixpoint mapstos e l : assert :=
match l with
  | nil ⇒ fun s h ⇒
    u2Z ([[ e ]]ℰ s) mod 4 = 0 ∧ emp s h
  | hd :: tl ⇒ (e ↦¹ int_e hd) ⋆
                (mapstos (e + int_e 4₃₂) tl)
end.
```

Above, `int_e` is the constructor for constant expressions and the semantics of + is the addition as implemented by the hardware.

We have actually formalized Separation logic in previous work following Reynolds [26]; we refer the reader to [1, 2, 18] for the complete detail of such an encoding in Coq. Let us just state the *frame rule* that allows for local reasoning. We will use the frame rule primarily to compose the code of functions (see Sect. 3.3 for illustration).

Assume that we are given a syntax and a semantics for some programming language, and let `c` be some program. We write `{ P } c { Q }`, where `P` and `Q` are Separation logic assertions, for Hoare triples. Let us assume that we are given a function `modified_regs`, that extracts the variables modified by the execution of a program `c`, as well as a predicate `independent l R`, indicating whether the validity of the assertion `R` depends on the variables occurring in `l`. The frame rule is formally stated as follows:

```
Lemma frame_rule P c Q : { P } c { Q } →
  ∀ R, independent (modified_regs c) R →
    { P ⋆ R } c { Q ⋆ R }.
```

Intuitively, it means that any Hoare triple that has been established locally (i.e., relatively to the memory footprint captured by the assertions `P` and `Q`) can be safely

extended beyond its memory footprint (the extra memory being captured by the assertion R) as long as the execution of the program does not interfere with it.

2.3 Unsigned multi-precision integer

As depicted in Fig. 1, an unsigned multi-precision integer consists of an array of words in memory (its *payload*) that is pointed to by a register, the length of the payload being kept in another register. The payload is of course interpreted as the encoding of a positive integer (it is understood that the least significant word comes first in the payload). Since we are dealing with a 32-bit architecture, the base of the encoding is 2^{32}.

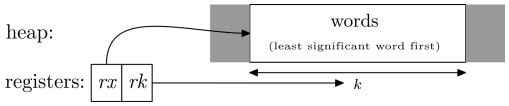

Fig. 1 An unsigned multi-precision integer

We define a Separation logic assertion to state that a (positive) integer is implemented as an unsigned multi-precision integer. var_unsign k rx val holds when the positive integer val is implemented by a payload of size k pointed to by register rx. This is formalized in Coq as follows:

```
0 Inductive var_unsign k rx val s h : Prop :=
1 | mkVarUnsign : u2Z ⟦ rx ⟧ℛ s + 4 * k < β →
2   0 ≤ val < β^k →
3   (rx ↦ Z2ints 32 k val) s h →
4   var_unsign k rx val s h.
```

Line 1 specifies that the array does not "wrap around" the heap (recall from Sect. 2.1 that the heap is finite of size β). This is an expected property that is for example guaranteed by standard dynamic memory allocation routines and that prevents overflows when accessing the memory. Line 2 specifies that val can safely be encoded in the base $2^{32} = \beta$ as a payload of size k, in other words that the multi-precision integer indeed fits in memory. As seen in Sect. 2.2, line 3 is a Separation logic formula; it specifies that the register rx points to the encoding of val (Z2ints 32 converts an arbitrary-precision integer into the corresponding list of words).

2.4 Signed multi-precision integer

Compared to unsigned multi-precision integers, the encoding (and therefore the formalization) of signed multi-precision integers is of course more involved. Signed multi-precision integers are usually encoded using *sign-magnitude*. This means that they are represented by a payload to be interpreted as unsigned, the sign information being kept separately. Performance being crucial, the zero integer is treated as a special case; this is for example what is done in GMP [31, Sect. 17.1].

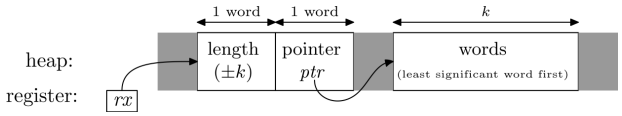

Fig. 2 A signed multi-precision integer

Fig. 2 pictures a typical encoding of signed multi-precision integers. The first word of this data structure contains the size in words of the magnitude (as for the unsigned case, we call it payload). More precisely, this size is a signed integer whose absolute value is the length of the payload and whose sign is the sign of the multi-precision integer being represented. There is a special case: the zero multi-precision integer is represented by having the size set to zero, in which case the contents of the payload are ignored.

The second word of the data structure depicted in Fig. 2 is a pointer to the payload. Actually, we are here mimicking GMP (except for the _mp_alloc field in GMP, that is used to do reallocation, which we defer to future work.)

Before the formalization in terms of Separation logic, we first formalize the relation between an integer value val and the elements of its sign-magnitude encoding: the size sz (of type int 32) and the list X (of type list (int 32) and of length k):

```
0 Inductive SignMagn sz k X val : Prop :=
1 | mkSignMagn : length X = k →
2   s2Z sz = Zsgn (s2Z sz) * k →
3   Zsgn (s2Z sz) = Zsgn val →
4   val = Zsgn (s2Z sz) * Sum k X →
5   SignMagn sz k X val.
```

Line 1 just fixes the length of the payload. Line 2 formalizes the relation between the actual length of the payload (k) and the size that is encoded (sz). (Zsgn is 0, 1, or −1, according to whether its argument is zero, positive, or negative, respectively.) When sz is not 0, this ensures that the absolute values of sz and k are the same; but, importantly, the fact that sz is 0 does not imply that k is 0. Line 3 forces the size that is encoded (sz) and the value that is encoded (val) to be of the same sign (in particular, to be 0 simultaneously). Finally, line 4 formalizes the relation between the actual contents of the payload (X) and the value that is encoded (val). (Sum k X computes the value encoded by the first k finite-size integers of list X; it is the converse of Z2ints 32). In particular, the fact that val is 0, does

not imply that the payload X is zeroed. Also, X cannot be 0 if val and l are not 0.

In summary, the peculiarity of the SignMagn predicate is to let the contents of the payload unspecified when the encoded value val is zero, thus achieving the same design principle as in GMP.

We define a Separation logic assertion to state that a (possibly negative) integer is implemented as a signed multi-precision integer. var_signed k rx val holds when the relative integer val is implemented by a signed multi-precision integer pointed to by register rx with a payload of length k:

```
0 Inductive var_signed k rx val s h : Prop :=
1   mkVarSigned : ∀ sz p X,
2   u2Z ⟦ rx ⟧ℛ s + 4 * 2 < β →
3   u2Z p + 4 * k < β →
4   SignMagn sz k X val →
5   (rx ↦ sz :: p :: nil ⋆ p ↦ X) s h →
6   var_signed k rx val s h.
```

Like the unsigned case, lines 2 and 3 avoid wrap-around's. In the Separation logic formula line 5, the first conjunct contains a pointer p to the second conjunct. Here, the length k of the payload has been made a parameter of the definition because we will be dealing later with a mix of signed and unsigned multi-precision integers, the latter requiring explicit mention of the payload length. Formalization of dynamic allocation that we plan for future work should alleviate this restriction.

3 Verification of primitive arithmetic functions

The purpose of this section is to explain how we formally prove the functional correctness of primitive operations for signed arithmetic. We consider a version of signed multi-precision addition as a running example.

First, we explain the syntax and semantics of SmartMIPS assembly programs (Sect. 3.1); this presentation completes Sect. 2.1 where we explained the formalization of the states of execution. Then, we comment on the concrete example of an implementation of the signed addition (Sect. 3.2). (See Table 1 for references to more examples.) Finally, we explain the corresponding correctness statement and comment on its formal verification (Sect. 3.3).

3.1 Syntax and semantics of assembly programs

Fig. 3 summarizes the syntax of the SmartMIPS programs that we are dealing with. We have formalized about thirty basic, one-step instructions (entry cmd0, instructions are named after the official documentation [21]). Some of them are specific to SmartMIPS

```
b    ::=  beq r1 r2    | bne r1 r2
       |  bltz r       | bgtz r
       |  bgez r       | blez r
cmd0 ::=  nop          | add r1 r2 r3
       |  addi r1 r2 i  | addiu r1 r2 i
       |  addu r1 r2 r3 | and r1 r2 r3
       |  andi r1 r2 i  | lw r1 i r2
       |  lwxs r1 r2 r3 | maddu r1 r2
       |  mfhi r1       | mflhxu r1
       |  mflo r1       | movn r1 r2 r3
       |  movz r1 r2 r3 | msubu r1 r2
       |  mthi r1       | mtlo r1
       |  multu r1 r2   | nor r1 r2 r3
       |  or r1 r2 r3   | sll r1 r2 a
       |  sllv r1 r2 r3 | sltu r1 r2 r3
       |  sra r1 r2 a   | srl r1 r2 a
       |  srlv r1 r2 r3 | subu r1 r2 r3
       |  sw r1 i r2    | xor r1 r2 r3
       |  xori r1 r2 i with 0 ≤ i < 2^16, 0 ≤ a < 2^5
cmd  ::=  cmd0
       |  c1 ; c2
       |  If b Then c1 Else c2
       |  While b { c }
```

Fig. 3 Syntax of SmartMIPS assembly programs

and not part of MIPS32. For example, lwxs, that loads a word from memory using scaled indexed addressing, has been introduced to improve performance of bytecode interpreters. maddu, multu, and mflhxu rely on the new ACX register that receives the carry out from the HI register. (Fig. 19 illustrates the usage of special instructions with the code for unsigned multi-precision addition.) In Fig. 3, i corresponds to 16-bit integers, a (in shifts) corresponds to 5-bit integers. These basic instructions can be composed together using sequence, structured branching, and while-loops (entry cmd in Fig. 3). Programs written with this syntax have therefore a structured control-flow. Standard SmartMIPS programs with labeled jumps are obtained via certified compilation [2]. Conditional control-flow commands make use of tests between registers (entry b in Fig. 3). To ease reading, we write If_beq r1 r2 instead of If (beq r1 r2), and so on. General-purpose registers are not hard-wired in programs but will be made parameters of programs (see for example Fig. 4).

The semantics of SmartMIPS assembly programs is formalized following the official documentation [21]; in particular, it takes into account the various error situations such as overflow conditions or alignment restrictions that lead to undefined behaviors. We distinguish error states with an option type: ⌊ s, h ⌋ represents a valid state, ⊥ represents error states. We note ⌊ s, h ⌋ ≻ c ↠ s' the (big-step) operational semantics of the cmd assembly language; it reads: starting from state ⌊ s, h ⌋, execution of the program p leads to the state s'; s' can be a valid state or an error state. Concretely, this operational semantics is formalized in Coq

as an inductive predicate. It is provably deterministic. By way of example, here follows the semantics of the instruction `lw` ("load word"). The execution of `lw r1 i r2` amounts to loading in register `r1` the contents of the address obtained by (hardware) addition of the base `r2` with the offset `i` (with `i` appropriately sign-extended). This is captured by the constructor `exec0_lw` of the operational semantics:

```
exec0_lw : ∀ s h r1 i r2 p z,
 u2Z (⟦ r2 ⟧ℛ s +ₕ signext 16 i) = 4 * p →
 heap.get p h = ⌊ z ⌋ →
 ⌊ s, h ⌋ ≻ lw r1 i r2 ↠
                  ⌊ store.upd r1 z s, h ⌋
```

In contrast, when the memory is not properly initialized or when the address is not on a word-boundary, the execution of `lw r1 i r2` fails. This is captured by this other constructor of the operational semantics:

```
exec0_lw_error : ∀ s h r1 i r2,
 ¬ (∃ p, u2Z (⟦ r2 ⟧ℛ s +ₕ signext 16 i) =
    4 * p ∧ ∃ z, heap.get p h = ⌊ z ⌋) →
 ⌊ s, h ⌋ ≻ lw r1 i r2 ↠ ⊥
```

See the online documentation [5] for more details, or [1, 2]. The semantics of further instructions will be illustrated concretely via examples in the course of this paper.

3.2 Example: In-place signed multi-precision addition

As an example of assembly code making use of signed integers, we display in Fig. 4 a variant of the signed-unsigned addition, that adds in-place a signed multi-precision integer and an unsigned multi-precision integer (with payloads of the same length). This example illustrates in particular the technical difficulties caused by treating the zero multi-precision integer as a special case. To simplify our development, we adopt a layered approach: functions for signed arithmetic are implemented using functions for unsigned arithmetic; this is possible because (the absolute value of) a signed integer can always been seen as an unsigned integer by just looking at its payload. This lets us implement functions in an incremental way and factorizes the formalization. Below, we assume that we are given two functions (`multi_add_u_u` and `multi_sub_u_u`) that respectively add and subtract unsigned multi-precision integers (these functions originally come from [1]; they can be found online [5]; Fig. 19 reproduces the code of `multi_add_u_u`).

The algorithm for multi-precision addition in Fig. 4 first sorts out the situations in which one of the argument is zero: when the second argument is zero, nothing needs to be done besides clearing the overflow register (line 3); when the first argument is zero, it is enough

```
0  Definition multi_add_s_u :=
1   multi_is_zero_u rk ry a0 a1 a2 ;
2   If_bne a2 , r0 Then (* y = 0 ? *)
3    addiu a3 r0 0₁₆ (* no overflow *)
4   Else (* y ≠ 0 *)
5    multi_add_s_u0 rk rx ry a0 a1 a2 a3 a4 a5 rX.
6
7  Definition multi_add_s_u0 :=
8   lw rX 4₁₆ rx ; (* payload of X *)
9   pick_sign rx a0 a1 ;
10  If_bgez a1 Then (* 0 ≤ x ? *)
11   If_beq a1 , r0 Then (* x = 0 ? *)
12    copy_u_u rk rX ry a2 a3 a4 ;
13    addiu a3 r0 0₁₆ ; (* no overflow *)
14    sw rk 0₁₆ rx (* fix size *)
15   Else (* 0 < x *)
16    addiu a3 r0 1₁₆ ;
17    multi_add_u_u rk a3 ry rX rX a0 a1 a2 ;
18    mflo a3 (* overflow *)
19  Else (* x < 0 *)
20   multi_lt rk ry rX a0 a1 a5 a2 a3 a4 ;
21   If_beq a5 , r0 Then (* x ≤ y ? *)
22    If_beq a2 , r0 Then (* x = y ? *)
23     addiu a3 r0 0₁₆ ; (* no overflow *)
24     sw r0 0₁₆ rx  (* fix size *)
25    Else (* x < y *)
26     multi_sub_u_u rk ry rX rX a0 a1 a2 a3 a4 a5 ;
27     multi_negate rx a0
28   Else (* x > y *)
29    multi_sub_u_u rk rX ry rX a0 a1 a5 a3 a2 a4.
```

Fig. 4 In-place signed-unsigned addition (see Fig. 19 for `multi_add_u_u`)

```
Definition multi_sub_s_s :=
  lw rY 4₁₆ ry ;
  pick_sign ry a0 a1 ;
  If_bgez a1 Then (* 0 ≤ y ? *)
    If_beq a1 , r0 Then (* y = 0 ? *)
       addiu a3 r0 0₁₆ (* no overflow *)
    Else (* 0 < y *)
       multi_sub_s_u rk rx rY a0 a1 a2 a3 a4 a5 rX
  Else (* y < 0 *)
    multi_add_s_u rk rx rY a0 a1 a2 a3 a4 a5 rX.
```

Fig. 5 In-place signed subtraction (see Fig. 4 for `multi_add_s_u` and Fig. 18 for `multi_sub_s_u`)

to copy (function `copy_u_u`) the contents of the second argument (line 12) and to update the size. Otherwise, when the first argument is strictly positive, it boils down to an unsigned multi-precision addition (line 17). Because of the special handling of zero, the situation where the first argument is strictly negative requires comparison between the two arguments. Upon equality, no computation needs to be performed and it is enough to set the size to zero (line 24); the (now unused) payload is left untouched, though we still have a pointer to it. Otherwise, in-place unsigned subtrac-

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

Fig. 6 Formal specification of the in-place signed-unsigned multi-precision addition of Fig. 4

tion (lines 26 and 29), possibly coupled with negation (function `multi_negate`), finishes the algorithm.

The algorithm we have just commented on for the in-place signed-unsigned addition as an easy counterpart for subtraction; let us call `multi_sub_s_u` its implementation (for the sake of completeness, we provide its code in Fig. 18). Given these two functions, it is easy to finally implement the in-place signed multi-precision subtraction (see Fig. 5).

3.3 Example: Verification of multi-precision addition

For primitive arithmetic functions (functions that correspond to primitive arithmetic operations such as addition), it is technically manageable to perform a direct proof in Hoare logic. Here, Separation logic (Sect 2.2) comes in handy: it is easy to deal with pointers that navigate inside the multi-precision integers; the use of the separating conjunction in specifications makes it clear whether the function operates in-place or not; when the target functions uses other functions, their proofs can be composed using the frame rule.

For example, the result of the formal verification of the multi-precision addition of Fig. 4 takes the form of the Hoare triple of Fig. 6.

Precondition of Fig. 6 The most important part of the precondition is the specification that the heap contains a signed integer (`var_signed k rx X`, encoding of `X`) and an unsigned integer (`var_unsign k ry Y`, encoding of `Y`) (see line 4). The `nodup` predicate of line 1 specifies a list of pairwise distinct registers, a necessary condition for correct execution. Line 2 seeks to avoid the undesirable case of the size being the so-called "weird number": in two's complement notation -2^{31} has indeed no positive inverse.

Postcondition of Fig. 6 The most informative part of the postcondition is its last conjunct (see line 10). If we forget about the overflow (the overflow bit is stored in a_3), it says that the resulting size `sz'` and the resulting payload `X'` are indeed the result of the addition `X` and `Y`:

```
Zsgn (s2Z sz') * Sum k X' = X + Y
```

Yet, potential overflow makes the most generic postcondition a little bit more involved. Potential overflows can indeed break the encoding of the first argument as a signed multi-precision integer, that is why the memory after execution of the program exhibits the points-to structure of the once-signed integer (line 8). Line 9 nevertheless keeps tract of the relations between the results of the computation.

Similarly to the example of in-place signed-unsigned addition, we have also formally verified the in-place signed subtraction whose code appeared in Fig. 5, as well as other assembly functions that are summarized in Table 1.

4 Verification using simulation

As seen in Sect. 3, Hoare triples for arithmetic functions can become technically involved. Our experience led us to seek for alternative approaches to verify larger functions, such as the binary extended gcd algorithm we will see in Sect. 6. This is why we experiment with 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 between both.

In this section, we first introduce a pseudo-code programming language (Sect. 4.1). Second, we introduce a generic definition of simulation between pseudo-code and assembly (Sect. 4.2). Then, we provide lemmas to prove simulation compositionally (Sect. 4.3). Last, we instantiate the generic definition of simulation with a relation between arbitrary-precision and multi-precision integers (Sect. 4.4).

4.1 Formalization of pseudo-code

Fig. 7 displays the syntax of pseudo-code programs. Variables are ranged over by x, y, etc. We are given a type for primitive arithmetic expressions (addition, subtraction, multiplication, division, remainder, and negation) ranged over by e. We are also given a type for boolean expressions (ranged over by b). The syntax for commands correspond to a type pcmd and they are ranged over by p.

```
e   ::=  x | i ∈ ℤ | e1 + e2 | e1 − e2 | e1 × e2
         | e1 / e2 | e1 % e2 | − e
b   ::=  true | b1 && b2 | b1 || b2 | ¬ b
         | e1 = e2 | e1 ≠ e2 | e1 ≥ e2 | e1 > e2
p   ::=  x ← e | skip | p1 ; p2 | If b Then p1 Else p2
         | While b { p }
```

Fig. 7 Syntax of pseudo-code programs

The semantics for pseudo-code commands is unsurprising. We are given a type pstore of stores with variables holding arbitrary-precision integers. Given a store st, we note ⟦ b ⟧$_{\mathcal{B}}$ st the truth value of the boolean expression b and ⟦ e ⟧$_{\mathcal{E}}$ st the integer value of the arithmetic expression e. We note ⌊ st ⌋ ≻ p ⇾ ⌊ st' ⌋ the (big-step) operational semantics of the pseudo-code language: starting from store ⌊ st ⌋, the execution of the program p leads to the store ⌊ st' ⌋ (⊥ models error states). For illustration, the operational semantics of the assignment x ← e is captured by the following constructor:

```
exec0_assign : ∀ st x e,
⌊ st ⌋ ≻ x ← e ⇾
              ⌊ store.upd x (⟦ e ⟧ℰ st) st ⌋
```

See the online documentation [5] for more details, or [18] for the original presentation of this pseudo-code language.

4.2 Forward simulation

We want to reason about the equivalence between two programs: (abstract) pseudo-code on the one hand, and (concrete) assembly code on the other hand. For that purpose, we introduce the type of relations between a state of the pseudo-code (type pstore) and a state of the assembly language (a pair of a store store.t and a heap heap.t):

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

Given the pseudo-code p and the assembly program c, the simulation that preserves the relation R under initial conditions P$_0$ is noted "p ≲(R, P$_0$) c" and is defined as follows:

```
Definition fwd_sim (R P₀ : Rel) p c :=
  ∀ st s h, R st s h → P₀ st s h →
    ∀ st', ⌊ st ⌋ ≻ p ⇾ ⌊ st' ⌋ →
    ∃ s' h', ⌊ s, h ⌋ ≻ c ⇾ ⌊ s', h' ⌋ ∧
      R st' s' h'.
```

In other words, when p and c are started at states related by R, they still end up in states related by R.

fwd_sim is a forward simulation in the sense of Leroy [16]. It is biased towards imperative programs and is therefore a simplification of the homonymous definition for concurrent systems [17]. fwd_sim can also be seen as a simplification of the "correspondence" predicate of Winwood et al. [32], in particular because we are in a deterministic setting.

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 easier pseudo-code (see [5]).

We also define a notion of simulation that relates boolean expressions from the pseudo-code (as defined in Fig. 7) and their equivalent in assembly code. Strictly speaking, there is no boolean expression in assembly but they can be simulated with a piece of code and a test between registers (second entry in Fig. 3). We note "b ≲$_b$(R) pre ; post" the fact that a boolean test b is simulated by a piece of code pre and a test between registers post under the relation R. This is formalized as follows:

```
Definition fwd_sim_b R b pre post :=
 ∀ st s h, R st s h →
  ∃ s', ⌊ s, h ⌋ ≻ pre ⇾ ⌊ s', h ⌋ ∧
    ⟦ b ⟧ℬ st ↔ ⟦ post ⟧ℬ s'.
```

4.3 Composition lemmas

This section presents the main lemmas to reason compositionally about forward simulation. As a preliminary step, we introduce intermediate definitions to express the side-conditions of these composition lemmas. The first definition is similar to the semantics of relational Hoare logic, a proof system to relate the execution of pairs of programs, typically a program and its optimized version [6]. Concretely, relational Hoare logic expresses the fact that a relation is preserved by the execution of two programs p and c in the following way: if the initial states of p and c satisfy the relation P, then the final states of p and c satisfy the relation Q, which we note p ∼ c : P ⇒ Q. Our definition actually slightly

generalizes relational Hoare logic because we are dealing with two programs p and c in different languages:

```
Definition rela_hoare (P Q : Rel) p c :=
 ∀ st s h, P st s h →
  ∀ st', ⌊ st ⌋ ≻ p ↠ ⌊ st' ⌋ →
  ∀ s' h', ⌊ s, h ⌋ ≻ c ↠ ⌊ s', h' ⌋ →
    Q st' s' h'.
```

Simulation of sequence The composition lemma for sequences shows that simulation of a sequence of instructions can be broken down to the simulations of each part:

```
Lemma fwd_sim_seq : ∀ R p p' c c' P Q,
 p ∼ c : P ⇒ Q →
 p ≲(R, P) c → p' ≲(R, Q) c' →
 (p ; p') ≲(R, P) (c ; c').
```

The side-condition p ∼c : P ⇒ Q formalizes the correct propagation of initial conditions.

We now come to the composition lemmas for structured branching and while-loops. They are a bit more involved because simulation between boolean expressions (as formalized by fwd_sim_b) is required. Again to express side-conditions, we introduce another intermediate definition that states that the execution of an assembly program c does not change the validity of a relation, ignoring the execution of pseudo-code:

```
Definition invariant (R : Rel) c :=
 ∀ st s h, R st s h →
  ∀ s' h', ⌊ s, h ⌋ ≻ c ↠ ⌊ s', h' ⌋ →
   R st s' h'.
```

Simulation of structured branching There is a simulation between structured branching in pseudo-code and in assembly code, if there is a simulation between the taken branches and if there is a simulation between boolean tests on both sides:

```
Lemma fwd_sim_ifte
    pre post p1 p2 c1 c2 R P₀ :
  invariant (R ∧ P₀) pre →
  b ≲ᵦ(R ∧ P₀) pre ; post →
  p1 ≲(R, fun st s h ⇒ P₀ st s h ∧
          ⟦ post ⟧ℬ s ∧ ⟦ b ⟧ℬ st) c1 →
  p2 ≲(R, fun st s h ⇒ P₀ st s h ∧
          ¬ ⟦ post ⟧ℛ s ∧ ¬ ⟦ b ⟧ℬ st) c2 →
  (If b Then p1 Else p2) ≲(R, P₀)
    (pre ; If post Then c1 Else c2).
```

Simulation of while-loops As can be expected, simulation of while-loops requires, as a side-condition, an invariant about the propagation of the initial condition (P_0 below), the latter being captured by relational Hoare logic:

```
Lemma fwd_sim_while b pre post p c R P₀ :
  invariant (R ∧ P₀) pre →
  p ∼ c : (fun st s h ⇒ P₀ st s h ∧
    ⟦ post ⟧ℬ s ∧ ⟦ b ⟧ℬ st) ⇒ P₀ →
  b ≲ᵦ(R ∧ P₀) pre ; post →
  p ≲(R, fun st s h ⇒ P₀ st s h ∧
        ⟦ post ⟧ℬ s ∧ ⟦ b ⟧ℬ st) c →
  (While b { p }) ≲(R, P₀)
    (pre ; While post { c ; pre }).
```

We will see a concrete application of this composition lemma in Sect. 6.4.

4.4 Instantiation for signed arithmetic

We now define concretely the relation for arithmetic between the execution states of pseudo-code and assembly programs. More precisely, the relation we define relates (1) a store for pseudo-code where variables contain arbitrary-precision integers, and (2) a state of execution of an assembly program where registers point to multi-precision integers (as explained in Sect. 2) containing the same values as the pseudo-code variables.

We first introduce a type for multi-precision integers. A multi-precision integer is either an unsigned multi-precision integer, implemented by two registers, one for the pointer to the payload and one for the size, or a signed multi-precision integer, implemented by the length of the payload and a pointer to the header of the data structure:

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

The point of this abstract definition of a multi-precision integer is to define the relation between, on the one hand, one pseudo-code variable, and, on the other hand, one multi-precision integer. A pseudo-code variable x is related to a multi-precision integer mx (of type mint) when the variable x contains the same value as the one encoded by the multi-precision integer mx. Put formally:

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

We now come to the relation between a pseudo-code store and an assembly state. 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 their image according to the map d:

```
0 Definition state_mint d : Rel :=
1  fun st s h ⇒
2   (∀ x mx, get x d = ⌊ mx ⌋ →
3    var_mint x mx st s (heap_mint mx s h)) ∧
4   (∀ x y, x ≠ y → ∀ mx my,
5    get x d = ⌊ mx ⌋ → get y d = ⌊ my ⌋ →
6    heap_mint mx s h ⊥ heap_mint my s h).
```

In this definition, `heap_mint mx s h` cuts out exactly that part of the heap that encodes the multi-precision integer `mx`. The first conjunct (starting at line 2) states that pseudo-code variables are related to the corresponding multi-precision integer in the association list. The second conjunct (starting at line 4) ensures that multi-precision integers are disjoint in the heap. The relation `state_mint` between pseudo-code variables and multi-precision integers is technically involved. This is essentially because there is a significant gap between, on one side, arbitrary-precision integers and, on the other side, their concrete implementations. This comes in contrast to relations used in proving, say, correctness of a compiler pass, where the gap between data on both sides is typically smaller than here.

5 Simulation for primitive arithmetic functions

In order to verify large arithmetic functions using simulation, we first need to equip primitive arithmetic 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*. Given the pseudo-code `p` and the assembly program `c`, the partial simulation that preserves the relation `R` under initial conditions P_0 is noted "$p \lesssim_p (R, P_0)\ c$" and is defined as follows:

```
Definition pfwd_sim (R P₀ : Rel) p c :=
  p ∼ c : (R ∧ P₀) ⇒ R.

Definition safe_termination (R : Rel) c :=
  ∀ s st h, R s st h →
    ∃ s', ⌊ st, h ⌋ ≻ c ↠ ⌊ s' ⌋.
```

Put together, `pfwd_sim` and `safe_termination` imply forward simulation `fwd_sim`:

```
Lemma pfwd_sim_fwd_sim (R P₀ : Rel) p c :
  p ≲ₚ(R, P₀) c →
    safe_termination (R ∧ P₀) c →
      p ≲(R, P₀) c.
```

Formal proof of `fwd_sim` for a pair of a pseudo-code program and an assembly program is conceptually easy when we are given the Hoare triple that establishes the functional correctness 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 corresponding Hoare triple. Proofs of `pfwd_sim` are illustrated by a concrete example below.

5.2 Example: Simulation for multi-precision addition

Intuitively, the proof of partial forward simulation for the addition `multi_add_s_u` defined in Sect. 3 makes explicit the conditions under which it behaves as the addition for arbitrary-precision integers. This setting is pictured in Fig. 8.

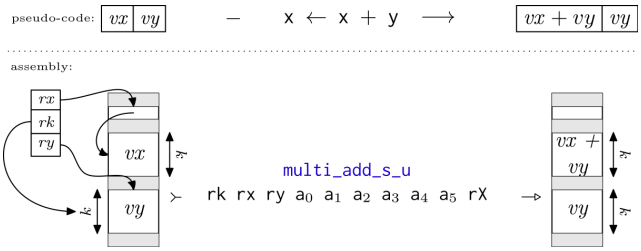

Fig. 8 Simulation for signed-unsigned addition

Fig. 9 is the formal version of Fig. 8. It states that the executions of, on the one hand, the pseudo-code `x ←x + y` (with $0 \leq$ `y`) (see line 4) and of the assembly code `multi_add_s_u rk rx ry...` (see line 9) preserve the relation

```
state_mint
  (x ↦ signed k rx ⊎ y ↦ unsign rk ry ⊎ d)
```

that appears at line 5. The relation `state_mint` was defined in Sect. 4.4 and `d` is an association list with no register in common with the code (as specified by lines 2–3). Because of potential overflows, this relation does not hold in general, hence the restrictions enforced at line 8 in the initial conditions (`Zabs` is the absolute value in the Coq standard library).

6 Application: Binary extended gcd algorithm

This section illustrates verification using simulation as introduced in Sect. 4 on a non-trivial program. We show that there is a simulation between the pseudo-code version of the binary extended gcd algorithm and an assembly implementation. The proof of simulation is performed compositionally (using the lemmas from Sect. 4.3) using the individual simulation proofs for primitive arithmetic functions (as the one we proved in Sect. 5.2 for addition).

```
Lemma pfwd_sim_multi_add_s_u :
  nodup(x, y) → nodup(rk, rx, ry, a₀, a₁, a₂, a₃, a₄, a₅, rX, r0) →
  disj (mints_regs (cdom d)) (a₀ :: a₁ :: a₂ :: a₃ :: a₄ :: a₅ :: rX :: nil) →
  x ∉ dom d → signed k rx ∉ cdom d → y ∉ dom d → unsign rk ry ∉ cdom d →
  x ← x + y
    ≲ₚ(state_mint (x ↦ signed k rx ⊎ y ↦ unsign rk ry ⊎ d),
       fun st s _ ⇒ 0 < u2Z ([[ rk ]]ᵣ s) < 2^31 ∧
       k = u2Z ([[ rk ]]ᵣ s) ∧
       Zabs ([[ x ]]ₑ st) < β^k ∧ 0 ≤ [[ y ]]ₑ st < β^k ∧ Zabs ([[ x ]]ₑ st + [[ y ]]ₑ st) < β^k)
  multi_add_s_u rk rx ry a₀ a₁ a₂ a₃ a₄ a₅ rX.
```

Fig. 9 Simulation for multi-precision addition

6.1 Pseudo-code for the binary extended gcd algorithm

The binary extended gcd algorithm is an extension of the celebrated 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 borrow the pseudo-code for the binary extended gcd algorithm from authoritative literature [15, p. 646]. The algorithm `begcd` of Fig. 10 computes the gcd of `u` and `v` and stores the result in u_3 and `g`.

```
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 }.
```

Fig. 10 The binary extended gcd algorithm (see Fig. 20 for the implementation in assembly, and Fig. 12, 14, 15, 16, and 17 for auxiliary functions)

The auxiliary function `prelude` halves the inputs as much as possible, recording the number of iterations in `g`. It is an interesting step because it reduces the size of data by means of mere shifts. Indeed, the gcd is preserved (modulo shifts) because when a and b are even, $\gcd(a, b) = 2 \times \gcd(a/2, b/2)$. We detail this step further as an illustration in Sect. 6.4.

There are several other variables (u_i, v_i, t_i) in the binary extended gcd algorithm that are manipulated in such a way that the relations $u * u_1 + v * u_2 = u_3$ (that will give the Bézout identity), $u * v_1 + v * v_2 = v_3$, and $u * t_1 + v * t_2 = t_3$ hold after each call to one of the auxiliary functions. In short, the auxiliary function `init` initializes the variables u_i, v_i, and t_i using `u`, `v`, 0, or 1 (see the code in Fig. 14). The auxiliary function `halve` tries to halve the temporary variables t_i (see the code in Fig. 15). The auxiliary function `reset` updates the variables u_i or v_i using the variables t_i (see the code in Fig. 16). The auxiliary function `subtract` updates the temporary variables t_i using $u_i - v_i$ (see the code in Fig. 17).

6.2 Correctness of the binary extended gcd algorithm

The first task is of course to make sure that the pseudo-code behaves as expected. This correctness statement is captured by the Hoare triple displayed in Fig. 11, where `Zgcd` is the gcd function of the standard Coq library.

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

Fig. 11 Correctness of the binary extended gcd algorithm

Precondition of Fig. 11 Line 2 declares two strictly positive inputs as ghost variables; the predicate `uv_init` at line 3 specifies that the input variables `u` and `v` are initialized correctly with the ghost variables:

```
Definition uv_init vu vv u v st :=
  [[ u ]]ₑ st = vu ∧ [[ v ]]ₑ st = vv.
```

Postcondition of Fig. 11 Line 6 states that the variables u_3 and `g` indeed contains the gcd of the inputs; lines 7–8 show that the variables u_1 and u_2 realize the Bézout identity. Besides functional correctness, the postcondition also states that all the variables that are manipulated by the algorithm remain bounded by the inputs.

Ensuring this fact is the role of the uivi_bounds and ti_bounds predicates:

```
Definition uivi_bounds u v u1 v1 u2 v2 u3 v3 st :=
  0 ≤ ⟦ u1 ⟧ε st ≤ ⟦ v ⟧ε st ∧
  0 ≤ ⟦ v1 ⟧ε st ≤ ⟦ v ⟧ε st ∧
  − ⟦ u ⟧ε st ≤ ⟦ u2 ⟧ε st ≤ 0 ∧
  − ⟦ u ⟧ε st ≤ ⟦ v2 ⟧ε st ≤ 0 ∧
  0 < ⟦ u3 ⟧ε st ≤ ⟦ u ⟧ε st ∧
  0 < ⟦ v3 ⟧ε st ≤ ⟦ v ⟧ε st.

Definition ti_bounds u v t1 t2 t3 st :=
  0 ≤ ⟦ t1 ⟧ε st ≤ ⟦ v ⟧ε st ∧
  − ⟦ u ⟧ε st ≤ ⟦ t2 ⟧ε st ≤ 0 ∧
  − ⟦ v ⟧ε st ≤ ⟦ t3 ⟧ε st ≤ ⟦ u ⟧ε st.
```

6.3 From pseudo-code to assembly

Starting from the pseudo-code we explained in the previous section, we produce assembly programs using our library of verified assembly functions for signed multi-precision arithmetic. In practice, we map to each pseudo-code instruction an assembly function such that there is an adequate simulation between both. For example, the result of this process can be observed in the case of the auxiliary function halve in Fig. 15. There, the pseudo-code addition is mapped to the assembly function multi_add_s_u, the corresponding simulation between both being the one we saw in Sect. 5.

The correspondence between the pseudo-code and the assembly program is further highlighted by Figures 12, 14, 16, and 17. The assembly programs are naturally given the same control-flow structure as the pseudo-code; this is convenient for applying the composition lemmas discussed in Sect. 4.2. Even though our library of assembly functions does not handle dynamic allocation, this correspondence is perfectly meaningful because we know that all the variables in the binary extended gcd algorithm lie between −vv and vu (as specified in Fig. 11); as a consequence, we can assume that the payload of all the multi-precision integers is stored into the same amount of words.

6.4 Example: Simulation of the prelude of the binary extended gcd algorithm

As explained in the previous section, the formal proof of simulation between the binary extended gcd in pseudo-code and its assembly counterpart decomposes naturally according to the auxiliary functions. Let us illustrate one of these subgoals using the simulation proof for the auxiliary function prelude. This function is displayed with its assembly counterpart prelude_mips in Fig. 12.

We first introduce some notation to simplify the presentation. Let us note d the association list that associates all the variables from the pseudo-code in the binary extended gcd to multi-precision integers:

```
Definition d :=
  g ↦ unsign rk rg ⊎ u ↦ unsign rk ru ⊎
  v ↦ unsign rk rv ⊎ u1 ↦ signed k ru1 ⊎
  u2 ↦ signed k ru2 ⊎ u3 ↦ signed k ru3 ⊎
  v1 ↦ signed k rv1 ⊎ v2 ↦ signed k rv2 ⊎
  v3 ↦ signed k rv3 ⊎ t1 ↦ signed k rt1 ⊎
  t2 ↦ signed k rt2 ⊎ t3 ↦ signed k rt3
```

Then, the formal statement of simulation between the pseudo-code prelude and the assembly prelude_mips is written as follows (we justify the initial condition in the next paragraph):

```
Lemma fwd_sim_begcd_prelude :
  (* predicates nodup elided *)
  prelude u v g
    ≲(state_mint d,
      fun st s h ⇒
      ⟦ u ⟧ε st * ⟦ g ⟧ε st < β^u2Z⟦rk⟧R s ∧
      0 ≤ ⟦ g ⟧ε st  ∧  0 < ⟦ u ⟧ε st ∧
      0 < u2Z ⟦ rk ⟧R s)
  prelude_mips rk rg ru rv a0 a1 a2 a3.
```

The formal proof of the simulation statement above essentially amounts to applying the composition lemmas we saw in Sect. 4.3: first for the while-loop (lemma fwd_sim_while), and then for the sequences in the body of the while-loop (lemma fwd_sim_seq). This decomposes the proof into simulation proofs between primitive operations (e.g., between the pseudo-code x ←x / 2 and the assembly function multi_halve_u) that are handled directly by simulation proofs we provide as a library (on the model of what we did in Sect. 5.2 for the addition).

Observe that in the case of the prelude of the binary extended gcd, the initial condition is necessary to establish the invariant required to simulate the while-loop. In essence, this invariant limits the size of g relatively to u so as to guarantee the absence of overflows (otherwise simulation could not be established because of the multiplication by 2 occurring in the body of the loop). Yet, this invariant does not deal with any gcd property. This is an illustration of the fact that the proof of simulation deals essentially with implementation-related aspects of the verification, functional correctness being the matter of the correctness statement for the pseudo-code (as summarized in the Hoare triple of Fig. 11 in our case).

6.5 Simulation for the binary extended gcd

Equipped with the simulation proofs for each auxiliary function as illustrated in the previous section, we are now in a position to prove the simulation between the

```
Definition prelude x y g :=
  While x % 2 = 0 && y % 2 = 0 {
    x ← x / 2 ;
    y ← y / 2 ;
    g ← g × 2 }.
```

```
Definition prelude_mips :=
  multi_is_even_u_and rk rx ry a0 a1 ;
  While (bne a0 r0) {
    multi_halve_u rk rx a0 a1 a2 a3 ;
    multi_halve_u rk ry a0 a1 a2 a3 ;
    multi_double_u rk rg a0 a1 a2 a3 ;
    multi_is_even_u_and rk rx ry a0 a1 }.
```

Fig. 12 Pseudo/assembly code for the `prelude` function of the binary extended gcd algorithm (see [5] for details)

binary extended gcd algorithm and its assembly implementation, simply by appealing to the composition lemmas. The formal statement is displayed in Fig. 13.

```
Lemma fwd_sim_begcd :
  (* predicates nodup elided *)
  0 < vu < β^k → 0 < vv < β^k →
  begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3
    ≲(state_mint d,
      fun st s h ⇒ uv_init vu vv u v st ∧
                   uv_bound rk s u v st k)
  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.
```

Fig. 13 Simulation for the binary extended gcd algorithm

The association list `d` was defined in the previous section. The initial condition deserves some explanation. The predicate `uv_init` has already been explained in Sect. 6.2. The predicate `uv_bound` establishes an important link between the pseudo-code inputs and the length of the payload of multi-precision integers in order to guarantee the absence of overflows:

```
0 Definition uv_bound rk s u v st k :=
1   k = u2Z (⟦ rk ⟧ℛ s) ∧
2   0 < k < 2^31 ∧
3   0 < ⟦ u ⟧ℰ st < β^(k-1) ∧ 0 < ⟦ v ⟧ℰ st < β^(k-1).
```

Line 2 specifies that the length of the payload of the multi-precision integers is smaller than 2^{31}, thus avoiding any conversion issue due to the two's complement notation (this issue was already evoked in Sect. 3.3). Line 3 specifies that the input values are strictly smaller than β^{k-1} (not β^k). This latter condition makes it possible to guarantee that there is no overflow during execution. Indeed, as we have explained in Sect. 6.2, 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`). So, at intermediate steps, all the multi-precision integers only require `k` words of payload. Yet, we cannot rule out the possibility for overflows *inside* auxiliary functions (consider for example the `subtract` function). We solve this problem by reserving the `k`th word of the payload for extra storage. This fact about the variables being bounded transports from the pseudo-code to the assembly code through the `state_mint` relation. Since this is a non-trivial part of the proof of correctness of `begcd`, it is very satisfactory to be able to avoid dealing with this issue directly at the assembly level thanks to the simulation relation.

6.6 Technical aspects of the formalization

We use the Coq proof-assistant extended with SSReflect [13]. This is essentially to benefit from SSReflect's concise tactics; our development does not rely crucially on SSReflect's library.

The framework for pseudo-code (Sect. 4.1) and the framework for assembly code (Sect. 2.2 and 3.1) essentially come from previous work [1, 2, 18, 19]. Since our development is rather long, their availability was important to reduce the formalization burden. These frameworks use two safe axioms: proof irrelevance and the extensionality of predicates [30]. Proof irrelevance is used to provide finite maps with Leibniz equality. The extensionality of predicates is used to equate equivalent Separation logic assertions (type `assert` in Sect. 2.2) so that rewriting can be performed using Coq's native `rewrite` tactic. The `Setoid` library could have been used as an alternative. The operational semantics and Hoare logics of the pseudo-code and the assembly code are actually two instances of a module that factorizes proofs such as the soundness of Hoare logic. This module can be used to ease instantiation to another assembly language.

The introduction of simulation called for a few extensions to the above frameworks. We extended them with several lemmas to reason about operational semantics (e.g., lemmas to prove termination, see Sect. 5.1). But the main improvement lies maybe in the underlying library for finite maps. We needed to extend this library with lemmas to deal with deletions and projections; this was made necessary by our definition of the relation between arbitrary-precision integers and multi-precision integers (see `heap_mint` in Sect. 4.4). We also needed to provide tactics to deal with association lists implemented as finite maps (the `d` in `state_mint d`), e.g., proving automatically that two association lists

```
Definition init
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  u1 ← 1 ;
  u2 ← 0 ;
  u3 ← u ;
  v1 ← v ;
  v2 ← 1 − u ;
  v3 ← v ;
  If u % 2 = 1 Then
    t1 ← 0 ;
    t2 ← −1 ;
    t3 ← − v
  Else
    t1 ← 1 ;
    t2 ← 0 ;
    t3 ← u.
```

```
Definition init_mips :=
  multi_one_s ru1 rk a0 a1 a2 a3 ;
  multi_zero_s ru2 ;
  copy_s_u rk ru3 ru a0 a1 a2 a3 ;
  copy_s_u rk rv1 rv a0 a1 a2 a3 ;
  multi_one_s rv2 rk a0 a1 a2 a3 ;
  multi_sub_s_u rk rv2 ru a0 a1 a2 a3 a4 a5 a6 ;
  copy_s_u rk rv3 rv a0 a1 a2 a3 ;
  multi_is_even_u rk ru a0 ;
  If_BEQ a0 , r0 Then
    multi_zero_s rt1 ;
    multi_one_s rt2 rk a0 a1 a2 a3 ;
    multi_negate rt2 a0 ;
    copy_s_u rk rt3 rv a0 a1 a2 a3 ;
    multi_negate rt3 a0
  Else
    multi_one_s rt1 rk a0 a1 a2 a3 ;
    multi_zero_s rt2 ;
    copy_s_u rk rt3 ru a0 a1 a2 a3.
```

Fig. 14 Pseudo/assembly code for the `init` function of the binary extended gcd algorithm (see [5] for details)

```
Definition halve u v t1 t2 t3 :=
  If t1 % 2 = 0 && t2 % 2 = 0 Then
    t1 ← t1 / 2 ;
    t2 ← t2 / 2 ;
    t3 ← t3 / 2
  Else
    t1 ← (t1 + v) / 2 ;
    t2 ← (t2 − u) / 2 ;
    t3 ← t3 / 2.
```

```
Definition halve_mips :=
  multi_is_even_s_and rt1 rt2 a0 a1 a2 a3 ;
  If_BNE a0 , r0 Then
    multi_halve_s rt1 a0 a1 a2 a3 a4 a5 ;
    multi_halve_s rt2 a0 a1 a2 a3 a4 a5 ;
    multi_halve_s rt3 a0 a1 a2 a3 a4 a5
  Else
    multi_add_s_u rk rt1 rv a0 a1 a2 a3 a4 a5 a6 ;
    multi_halve_s rt1 a0 a1 a2 a3 a4 a5 ;
    multi_sub_s_u rk rt2 ru a0 a1 a2 a3 a4 a5 a6 ;
    multi_halve_s rt2 a0 a1 a2 a3 a4 a5 ;
    multi_halve_s rt3 a0 a1 a2 a3 a4 a5.
```

Fig. 15 Pseudo/assembly code for the `halve` function of the binary extended gcd algorithm (see [5] for details)

```
Definition reset
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  If t3 ≥ 0 Then
    u1 ← t1 ;
    u2 ← t2 ;
    u3 ← t3
  Else
    v1 ← v − t1 ;
    v2 ← − (u + t2) ;
    v3 ← − t3.
```

```
Definition reset_mips :=
  pick_sign rt3 a0 a1;
  If_BGEZ a1 Then
    copy_s_s ru1 rt1 a0 a1 a2 a3 a4 a5 ;
    copy_s_s ru2 rt2 a0 a1 a2 a3 a4 a5 ;
    copy_s_s ru3 rt3 a0 a1 a2 a3 a4 a5
  Else
    multi_sub_s_s_u rk rv1 rt1 rv a0 a1 a2 a3 a4 a5 a6 a7 ;
    multi_negate rv1 a0 ;
    multi_add_s_s_u rk rv2 rt2 ru a0 a1 a2 a3 a4 a5 a6 a7 ;
    multi_negate rv2 a0 ;
    copy_s_s rv3 rt3 a0 a1 a2 a3 a4 a5 ;
    multi_negate rv3 a0.
```

Fig. 16 Pseudo/assembly code for the `reset` function of the binary extended gcd algorithm (see [5] for details)

are disjoint. This part of the development is reusable because it is not tied to the target programming languages. The file that defines simulations and proves related properties (Sect. 4.2–4.4) amounts to 2480 lines of Coq script (`coqwc` figures).

Table 1 provides an overview of the arithmetic functions discussed in this paper. It complements previous work [1,2] on unsigned modular arithmetic (see Sect. 7). The library that makes possible the verification of the binary extended gcd algorithm consists of a total of 313 lines of assembly code, spread over 25 functions. Almost all functions come with correctness proofs in the form of Hoare triples (for a total of 7746 lines of Coq script). Most functions are equipped with simula-

```
Definition subtract
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  t1 ← u1 − v1 ;
  t2 ← u2 − v2 ;
  t3 ← u3 − v3 ;
  If 0 ≥ t1 Then
    t1 ← t1 + v ;
    t2 ← t2 − u
  Else
    skip.
```

```
Definition subtract_mips :=
  multi_sub_s_s_s rk rt1 ru1 rv1 a0 a1 a2 a3 a4 a5 a6 a7 a8 ;
  multi_sub_s_s_s rk rt2 ru2 rv2 a0 a1 a2 a3 a4 a5 a6 a7 a8 ;
  multi_sub_s_s_s rk rt3 ru3 rv3 a0 a1 a2 a3 a4 a5 a6 a7 a8 ;
  pick_sign rt1 a0 a1;
  If_blez a1 Then
    multi_add_s_u rk rt1 rv a0 a1 a2 a3 a4 a5 a6 ;
    multi_sub_s_u rk rt2 ru a0 a1 a2 a3 a4 a5 a6
  Else
    nop.
```

Fig. 17 Pseudo/assembly code for the subtract function of the binary extended gcd algorithm (see [5] for details)

tion proofs (for a total of 4753 lines of Coq script). The binary extended gcd algorithm is written with 49 lines of pseudo-code and 68 lines of assembly code (using the library of functions above). The simulation proof for the binary extended gcd algorithm consists of 1466 lines of a systematic Coq script.

We did not seek for brevity in Coq scripts so that above figures have to be understood as improvable upper bounds.

7 Related Work

Verification of assembly programs using proof-assistants is not a new topic. The complexity of dealing with low-level aspects has already triggered much research. Tan and Appel tackled the problem of reasoning about unstructured control-flow [28]. They proposed a program logic that handles multiple-entry and multiple-exit program fragments in a way that makes possible modular reasoning. It has been instantiated to SPARC machine code, proved sound in Twelf, and applied to establish safety properties for proof-carrying code. Myreen and Gordon addressed the issue of Hoare logic reasoning for realistically modeled machine code [22]. They model finite data and finite memory, and their Hoare logic still accounts for multiple-entry and multiple-exit program fragments using position-dependent specifications. It has been in particular instantiated to ARM machine code and applied to the verification of arithmetic operations [23]. In subsequent work, Myreen also provided a solution to the problem of verification of self-modifying code by treating code as data in an even more precise model that includes an instruction cache [24]. The corresponding Hoare logic has been instantiated to x86 machine code and applied to the verification of just-in-time compilers. In our work, we do not support advanced reasoning features about unstructured control-flow, position-dependent or self-modifying code because they are not required to implement arithmetic functions. Still, our model of SmartMIPS is realistic enough (data and memory are finite [1]) so that verified programs can be turned to standard SmartMIPS programs (with labeled jumps) via certified compilation [2].

As already explained in Sect. 1, there exist other experiments about formal verification of low-level unsigned multi-precision arithmetic. In previous work, we formally verified in the Coq proof-assistant several other assembly functions for unsigned multi-precision arithmetic [1, 2]. In fact, modular multiplication, modular squaring, and modular exponentiation (all based on the Montgomery multiplication) as well as pseudo-random number generation (the Blum-Blum-Shub algorithm) come as a complement to the library summarized in Table 1. Myreen and Gordon also verified some modular arithmetic written in machine code in the HOL proof-assistant [23]. Their experiment illustrates an original approach that splits verification of machine code between verification of a functional version of the algorithm and a proof that the machine code implements the functional version. Berghofer also carried out verification of multi-precision arithmetic [7] but with a higher-level language: the Spark subset of Ada. The standard tool suite for Spark comprises a generator of verification conditions that are passed to an automatic verifier; when automation fails, it reverts to an interactive verification tool to which reasoning rules can be added without rigorous consistency checks. This is this interactive verification tool that Berghofer proposes to replace with the Isabelle/HOL proof-assistant. The resulting environment has been applied to the verification of a library for multi-precision integers. It features functions similar to Table 1 and our previous work [1, 2] but does not seem to address signed multi-precision arithmetic. Our work therefore appears as an original effort to build a fully-formalized library of low-level multi-precision arithmetic encompassing signed multi-precision arithmetic.

All things being relative, our approach shares similarities with the one used to formally verify the seL4 microkernel [10,32] in Isabelle/HOL. There also, refinement is established by forward simulation, itself proved

Description		*Proof scripts*	*Assembly (l.o.c.)*
Arithmetic computations			
x←0	x unsigned	multi_zero_u_{prg,triple,simu}.v	6
	x signed	multi_zero_s_{prg,triple,simu}.v	1
x←1	x unsigned	multi_one_u_{prg,triple,simu}.v	3
	x signed	multi_one_s_{prg,triple,simu}.v	12
x←x / 2	x unsigned	multi_halve_u_{prg,triple,simu}.v	13
	x signed	multi_halve_s_{prg,triple,simu}.v	22
x←x * 2	x unsigned	multi_double_u_{prg,triple,simu}.v	12
x←y	x signed, y unsigned	copy_s_u_{prg,triple,simu}.v	7
	x, y signed	copy_s_s_{prg,triple,simu}.v	8
x←-x	x signed	multi_negate_{prg,triple,simu}.v	3
x++	x unsigned	multi_incr_u_{prg,triple}.v	15
x←x + y	x, y unsigned	multi_add_u_u_{prg,triple}.v	12
	x signed, y unsigned	multi_add_s_u_{prg,triple,simu}.v	27
z←x + y	z, x, y unsigned	multi_add_u_u_u_triple.v	
	z, x signed, y unsigned	multi_add_s_s_u_{prg,triple,simu}.v	24
x←x - y	x, y unsigned	multi_sub_u_u_{L,R}_{prg,triple}.v	19
	x signed, y unsigned	multi_sub_s_u_{prg,triple,simu}.v	28
	x, y signed	multi_sub_s_s_{prg,triple,simu}.v	9
z←x - y	z, x, y unsigned	multi_sub_u_u_u_triple.v	
	z, x signed, y unsigned	multi_sub_s_s_u_{prg,triple,simu}.v	34
	z, x, y signed	multi_sub_s_s_s_{prg,triple,simu}.v	10
Arithmetic tests			
$x \overset{?}{=} y$, $x \overset{?}{<} y$, $x \overset{?}{>} y$	x, y unsigned	multi_lt_{prg,triple,simu}.v	15
sign of x	x signed	pick_sign_{prg,triple,simu}.v	9
parity	x unsigned	multi_is_even_u_{prg,triple,simu}.v	6
	x signed	multi_is_even_s_{prg,triple,simu}.v	8
	x, y unsigned	multi_is_even_u_and_{prg,simu}.v	3
$x \overset{?}{=} 0$	x unsigned	multi_is_zero_u_{prg,triple}.v	7
Binary extended gcd algorithm			
Correctness for the pseudo-code (Fig. 10) and variants		begcd.v	
Simulation for the prelude auxiliary function (Fig. 12)		begcd_mips_prelude.v	6
Simulation for the init auxiliary function (Fig. 14)		begcd_mips_init.v	18
Simulation for the halve auxiliary function (Fig. 15)		begcd_mips_halve.v	11
Simulation for the reset auxiliary function (Fig. 16)		begcd_mips_reset.v	12
Simulation for the subtract auxiliary function (Fig. 17)		begcd_mips_subtract.v	9
Simulation for the begcd main function (Fig. 20)		begcd_mips.v	12

Table 1 Overview of our library of formally verified low-level arithmetic functions (see [1,2] for modular arithmetic)

using Hoare logic. A first refinement step relates an abstract specification and an executable model; a second refinement step relates the executable model and a C implementation. It is tempting but difficult to compare the pseudo-code in our work with the abstract specification above because the latter is more detailed: it features nondeterminism, abstract functions are used to model pointers, some parts of the model are very concrete (e.g., decoding of machine registers to feed system calls [10]). Similarly, the executable model of seL4 is (essentially) a Haskell program that describes the low-level design with a high degree of detail (it uses machine words, doubly linked lists, etc.); as a consequence, the C source code may make small optimizations but is structurally similar [32]. In comparison, a pseudo-code instruction and a related assembly function in our work seem more apart than the executable model and the C implementation in [32]. Our work may have informative value because we are dealing with simulation and assembly whereas assembly is out of scope in [10,32].

Simulation proofs are also at the heart of the formal verification of the compiler of Leroy [16]. In this work, relations are established between the languages of each compiler pass. In our work, the relation is established between pseudo-code and assembly, that are more apart than two consecutive intermediate compilation languages. Indeed, our goal departs from the one of certifying a set of program transformations (streamlined as a compiler) in that we aim at providing a *library* to ease formal proof of correctness about hand-written assembly programs.

Yang has already combined simulation and Separation logic in the form of *relational Hoare logic* [33], that Crespo and Kunz mechanized [11]. In relational Separation logic, the programs that are related are written in the same abstract, high-level language (storage is limited to integers and there are native list values).

In contrast, we relate very different programming languages (one of them being a realistic assembly language) and are seeking to establish data refinement rather than program equivalence. Also, the validating experiment by Yang [33] (the Schorr-Waite algorithm) is much smaller and therefore technically addresses different issues. Yet, Crespo and Kunz [11] hint at improvements to go beyond structurally equivalent programs, which, in our case, is an extension that highly-optimized assembly code is bound to call for.

Our work is about the verification of implementations of multi-precision arithmetic. It naturally raises the question of the verification of implementations of multiple-precision floating-point computations. This issue is certainly more challenging, in particular in terms of specification because floating-point arithmetic is used to implement an *approximation* of real numbers. To tackle this problem, Boldo and Melquiond have recently been developing a library for proving floating-point algorithms in Coq [8].

8 Conclusion

We proposed an approach for the construction of a library of formally verified low-level arithmetic functions. We introduced a formalization of data structures for signed multi-precision arithmetic. Using this formalization, it becomes possible to formally verify basic assembly functions about signed multi-precision arithmetic. This is done directly, by appealing in particular to the frame rule of Separation logic to handle composition of code. In order to deal with larger functions (beyond primitive operations such as addition), we proposed an approach based on simulation. It consists in showing formally a simulation relation between the pseudo-code and the assembly, so that the pseudo-code can serve as a specification of the implementation, as this is usually done in standard handbooks, and as it is expected by programmers. Simulation are proved directly for the most primitive arithmetic functions so as to produce a library of simulation proofs. Table 1 gives an overview of the library we have constructed in this way. Using such a library, one can then work compositionally to establish simulation for larger functions. This latter point was illustrated thoroughly with an assembly implementation of the binary extended gcd algorithm.

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 our work to deal with variable size multi-precision 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 [4], so that dynamic allocation can be provided by C's malloc. As an application of the resulting library, we plan to investigate the trustful implementation of realistic cryptographic schemes.

References

1. Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. In: Proceedings of the 11th Annual Asian Computing Science Conference. LNCS, vol. 4435, pp. 346–360. Springer, Heidelberg (2008)
2. Affeldt, R., Nowak, D., Yamada, K.: Certifying Assembly with Formal Security Proofs: the Case of BBS. Sci. Comput. Program, 77(10–11), 1058–1074 (2012)
3. Affeldt, R.: On Construction of a Library of Formally Verified Low-level Arithmetic Functions. In: Proceedings of the 27th ACM SIGAPP Symposium On Applied Computing (SAC 2012), Software Verification and Testing Track, vol. 2, pp. 1326-1331. ACM (2012)
4. Affeldt, R., Marti, N.: Towards Formal Verification of TLS Network Packet Processing Written in C. In: Proceedings of the 7th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV 2013), pp. 35–46. ACM (2013)
5. Affeldt, R.: A Library for Formal Verification of Low-level Programs. Coq documentation. `http://staff.aist.go.jp/reynald.affeldt/coqdev` (last access: 2013/02/25).
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), pp. 14–25. ACM (2004)
7. Berghofer, S.: Verification of Dependable Software using SPARK and Isabelle. In: Proceedings of the 6th International Workshop on Systems Software Verification Proceedings, pp. 48–65 (2011)
8. Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH 2011), pp. 243–252. IEEE Computer Society (2011)
9. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic. Version 0.5.9 (7 October 2010). Available at `http://www.loria.fr/~zimmerma/mca/mca-cup-0.5.9.pdf` (last access: 2012/12/03). Final version published by Cambridge University Press (2010)
10. Cock, D., Klein, G., Sewell, T.: Secure Microkernels, State Monads and Scalable Refinement. In: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008). LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
11. Crespo, J.M., Kunz, C.: A Machine-Checked Framework for Relational Separation Logic. In: Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM 2011). LNCS, vol. 7041, pp. 122–137. Springer, Heidelberg (2011)
12. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
13. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq System. Technical Report 6455. Version 11. INRIA (2012)
14. Hur, C.-K., Dreyer, D.: A Kripke logical relation between ML and assembly. In: Proceedings of the 38th ACM

SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 133–146. ACM (2011)
15. Knuth, D.E.: The Art of Computer Programming. Vol. 2, 3rd edition. Addison-Wesley (1997)
16. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning. 43(4), 363–446 (2009)
17. Lynch, N.A., Vaandrager, F.W.: Forward and Backward Simulations Part I: Untimed Systems. Inform. Comput. 121(2), 214–233 (1995)
18. Marti, N., Affeldt, R., Yonezawa, A.: Formal Verification of the Heap Manager of an Operating System using Separation Logic. In: Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006). LNCS, vol. 4260, pp. 400-419. Springer, Heidelberg (2006).
19. Marti, N., Affeldt, R.: A Certified Verifier for a Fragment of Separation Logic. Computer Software 25(3), 135–147 (2008)
20. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. 5th printing. CRC Press (2001)
21. MIPS Technologies: MIPS32 4KS Processor Core Family Software User's Manual (2001)
22. Myreen, M.O., Gordon, M.J.C.: Hoare Logic for Realistically Modelled Machine Code. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). LNCS, vol. 4424, pp. 568–582. Springer, Heidelberg (2007)
23. Myreen, M., Gordon, M.: 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)
24. Myreen, M.O.: Verified just-in-time compiler on x86. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010), pp. 107–118. ACM (2010)
25. Reynolds, J.C.: The Craft of Programming. Prentice-Hall International (1981).
26. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society (2002)
27. Shoup, V.: NTL: A Library for doing Number Theory. Version 5.5.2. Available at `http://www.shoup.net/ntl` (last access: 2012/12/03) (2009)
28. Tan, G., Appel, A.W.: A Compositional Logic for Control Flow. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006). LNCS, vol. 3855, pp. 80–94. Springer, Heidelberg (2006)
29. The Coq Proof Assistant: Reference Manual. Ver. 8.4. Available at `http://coq.inria.fr`. INRIA (2012)
30. The Coq Proof Assistant: Frequently Asked Questions. Available at `http://coq.inria.fr/faq`. INRIA (2012)
31. The GNU Multi Precision Arithmetic Library. Edition 5.0.2. `http://gmplib.org/` (2011)
32. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the Gap: A Verification Framework for Low-level C. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009). LNCS, vol. 5674, pp. 500–515. Springer, Heidelberg (2009)
33. Yang, H.: Relational separation logic. Theor. Comput. Sci. 375(1–3), 308–334 (2007)

A Additional assembly code

This section provides for the sake of completeness assembly code that is explicitly referred to in the body of this paper. See [5] for other assembly code or formal proofs.

```
Definition multi_sub_s_u :=
 multi_is_zero_u rk ry a0 a1 a2 ;
 If_bne a2 , r0 Then
  addiu a3 r0 0x10
 Else
  multi_sub_s_u0 rk rx ry a0 a1 a2 a3 a4 a5 rX.

Definition multi_sub_s_u0 :=
 lw rX 0x4 rx ; (* payload of X *)
 pick_sign rx a0 a1 ;
 If_bgez a1 Then (* 0 ≤ x ? *)
  If_beq a1 , r0 Then (* x = 0 ? *)
   copy_u_u rk rX ry a2 a3 a4 ;
   addiu a3 r0 0x10 ; (* no overflow *)
   sw rk 0x10 rx ; (* fix size *)
   multi_negate rx a0
  Else
   multi_lt rk rX ry a0 a1 a5 a2 a3 a4 ;
   If_beq a5 , r0 Then (* y ≤ x ? *)
    If_beq a2 , r0 Then (* x = y ? *)
     addiu a3 r0 0x10 ; (* no overflow *)
     sw r0 0x10 rx (* fix size *)
    Else (* y < x *)
     multi_sub_u_u rk rX ry rX a0 a1 a2 a3 a4 a5
   Else (* y > x *)
    multi_sub_u_u rk ry rX rX a0 a1 a2 a3 a4 a5 ;
    multi_negate rx a0
 Else (* x < 0 *)
  addiu a3 r0 0x1 ;
  multi_add_u_u rk a3 ry rX rX a0 a1 a2 ;
  mflo a3.
```

Fig. 18 In-place signed-unsigned subtraction (appears in Fig. 5)

```
Definition multi_add_u_u :=
  addiu a0 r0 0x10 ;
  addiu rZ rz 0x10 ;
  multu r0 r0 ;
  While (bne a0 rk) {
    lwxs rX a0 rx ;
    maddu rX r1 ;
    lwxs rX a0 ry ;
    maddu rX r1 ;
    mflhxu rX ;
    sw rX 0x10 rZ ;
    addiu rZ rZ 0x4 ;
    addiu a0 a0 0x1 }.
```

Fig. 19 Unsigned-unsigned addition (appears in Fig. 4 and 18)

```
Definition 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 :=
 multi_one_u rk rg a0 a1 ;
 prelude_mips rk rg ru rv a0 a1 a2 a3 ;
 init_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
           rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 ;
 pick_sign rt3 a0 a1 ;
 While (bne a1 r0) {
  multi_is_even_s rt3 a0 a1 a2 ;
  While (bne a2 r0) {
   halve_mips rk ru rv rt1 rt2 rt3
              a0 a1 a2 a3 a4 a5 a6;
   multi_is_even_s rt3 a0 a1 a2 } ;
  reset_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
             rt1 rt2 rt3 a0 a1 a2 a3 a4 a7 a8 a9 ;
  subtract_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
                rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 a7 a8 ;
  pick_sign rt3 a0 a1 }.
```

Fig. 20 Assembly code for the main function of the binary extended gcd algorithm (see Fig. 10 for the corresponding pseudo-code)