Toward Formal Construction of Assembly Arithmetic Functions from Pseudo-code[*]

Reynald Affeldt

National Institute of Advanced Indutrial Science and Technology (AIST)
Research Center for Information Security (RCIS)

Abstract

Most cryptographic software relies on arithmetic functions, and these functions must be implemented correctly and efficiently. In practice, they are written by hand directly in assembly and undergo costly testing. Proof-assistants provide a way to avoid testing without sacrificing efficiency, but formal verification of low-level code is technically difficult. We propose a way to address the scalability issues raised by the formal verification of large arithmetic functions. We advocate formal construction from pseudo-code to split the effort into formal verification of an idealized implementation written in pseudo-code and a formal proof that this pseudo-code simulates the target assembly program. In this setting, properties of the assembly program can be derived from proofs at the simpler level of pseudo-code. As for the formal proof of the simulation, it can be made systematic, in particular when the target assembly program is built out of a library of more basic but already verified arithmetic functions. We illustrate this approach with preliminary but concrete examples.

1 Introduction

Much cryptography used in practice is based on number theory and cryptographic software therefore relies on arithmetic functions. They must be implemented efficiently because cryptography requires much calculation, that is often bound to happen interactively. Needless to say, arithmetic functions must also be implemented correctly; correctness is all the more important as most high-level security properties rely on it. As a practical consequence, arithmetic functions are written by hand directly in assembly and undergo costly testing. The importance of the testing phase is actually emphasized by certification of cryptographic algorithms (e.g., the North American Cryptographic Algorithm Validation Program).

In theory, we can improve the above situation by using proof-assistants because they provide a way to model arithmetic functions faithfully and to verify them formally, thus avoiding testing without sacrificing efficiency. In addition, proof-assistants, compared to other formal methods, have the advantage of providing proofs checkable by third-parties, thus paving the way for formal verification of further security properties.

However, formal verification in proof-assistants of assembly arithmetic functions is technically difficult. This is mainly because assembly programs (actually, low-level code in general, see for example [15] for illustrations other than arithmetic functions) leads to formal proofs cluttered with low-level details, such as precise information about the size or the alignment of data structures in the (finite) memory of the computer. These details are a hindrance to scalability when formal verification is carried out directly on an assembly program.

We propose as an approach to improve scalability of formal verification of assembly arithmetic functions to split the effort into a formal proof about an implementation in pseudo-code and a formal proof that this idealized implementation simulates the target assembly program. This

[*]This work was presented at the 12th JSSST Workshop on Programming and Programming Languages (`http://isw3.kankyo-u.ac.jp/ppl2010`).

facilitates formal verification because many properties (such as functional correctness) can be established at the simpler level of pseudo-code and then "transported" by generic lemmas at the level of assembly. Also, the formal proof of simulation can be built systematically, by composing assembly programs for which Hoare-logic triples, simulation proofs, termination proofs, etc. are readily available as a library. In fact, the potential benefits of this approach can already by appreciated by looking at our previous work. In [14], we formally verify in the Coq proof assistant a hand-written program of more than 200 assembly instructions by reusing the formal proof of a semantically equivalent 2-lines Coq function. Although it is not specified in terms of a simulation, the link between both proofs can actually be seen as such. This case study is all the more convincing as the property of interest here is a non-trivial security property, specified in terms of probability and number theory, already requiring a significant proof effort for the 2-lines Coq function [11].

We also expect our approach to improve reusability and readability. Formal proofs about idealized implementations are reusable because largely independent from assembly programs: the effects of local modifications of the latter should be limited to the formal proof of simulation. We also want to emphasize reusability of simulation proofs for basic arithmetic functions used as subroutines to build larger arithmetic functions. Besides reusability, our approach has also the desirable side-effect of improving readability by letting functional correctness be expressed at the level of pseudo-code.

Formal construction of assembly functions from pseudo-code is closely related to but different from formally-verified compilation. First, it targets pseudo-code, a high-level language with an unbounded memory and, in the case of arithmetic functions, manipulating data such as arbitrary-precision integers. Therefore, the end-languages that we are dealing with when constructing assembly functions from pseudo-code are much more apart than the end-languages of a compiler pass. Second, as explained above, formal construction by way of simulation allows for flexible modifications of the target assembly programs. In contrast, a compiler is deterministic and one may not always have an adequate formal translation validation scheme at hand for the intended modification. To summarize, it seems difficult to come up with a certified compiler for pseudo-code that generates efficient assembly and accommodates seamlessly (i.e., without invalidating the formal proof) local re-engineering of its output.

Contribution In this paper, we set up an experimental framework in the Coq proof-assistant [2] to formally verify hand-written arithmetic functions in assembly by reusing proofs for programs written in pseudo-code. More precisely, we aim at organizing verification as follows: (1) a formal proof about imperative pseudo-code, (2) a formal proof that the pseudo-code simulates the assembly program, and (3) a lemma to transform Hoare-logic triples about the pseudo-code into Separation-logic triples about the assembly program. For this purpose, we extend two existing formalizations:

- A formalization of Hoare logic[1] for structured imperative programs with an unbounded number of variables that we use in this paper with arbitrary-precision integers, the data structure of interest when dealing with arithmetic functions.
- A formalization of Separation logic for SmartMIPS assembly programs [8], an instruction set designed for the implementation of efficient cryptography. This formalization is biased toward structured assembly programs but it can also deal with gotos through certified compilation ([9], formalized in [14]). For the sake of simplicity, we stick to structured assembly programs in this paper.

On top of these two formalizations, we instrument a notion of simulation. As a preliminary experiment, we apply the resulting formalization to the prelude of an assembly program that

[1]This formalization is in fact a more general formalization of Separation logic to deal with mutable memory. Technically, this will cause small discrepancies between the intended lemmas and lemmas actually formalized. For example, we sometimes need to "reinforce" lemmas with conditions on the absence of dynamic allocation in pseudo-code even though we actually never intend to use it here. We prefer this situation to the burden of managing another, less-general formalization of Hoare logic.

implements the binary extended binary GCD algorithm[2], an important algorithm used in cryptography to compute for example inverses modulo. Despite the limitations of our framework in its current state (in particular, a large-scale experiment still to be completed), we hope to provide a concrete evaluation of the effort toward formal construction of number-theory based cryptographic schemes.

Outline In Sect. 2, we give an overview of the existing frameworks for pseudo-code and assembly that we use in this paper. In Sect. 3, we give as an example a program in pseudo-code and an implementation in assembly and explain informally the relation between the Hoare-logic triple for the former and the Separation-logic triple for the latter. In Sect. 4, we formalize the notion of simulation between pseudo-code and assembly. In Sect. 5, we define for the case of arithmetic functions a concrete relation between states in the pseudo-code and states in the assembly. We also provide a lemma that transforms Hoare-logic triples for pseudo-code into Separation-logic triples for the assembly, thus reducing the effort of formal proof of assembly to formal proof of pseudo-code and its simulation. In Sect. 6, we explain in detail examples of simulation proofs. To better appreciate the organization of the formal proofs, we also give a schematic overview of the whole framework in Coq. We discuss related work in Sect. 7 and conclude in Sect. 8. (For the sake of completeness, Appendix A explains some technical lemmas used in Sect. 6.)

2 Formalization of Pseudo-code and Assembly

Generic Notations Prop is the canonical type in Coq for propositions, e.g., $\mathsf{True} : \mathsf{Prop}$. Logical symbols ($\Rightarrow$, $\wedge$, $\vee$, $\forall$, $\exists$, etc.) have to be understood as the corresponding Coq constructs. Anonymous functions are written with the usual λ notation. Relations are understood as functions with codomain Prop. When the names of the parameters of a function or a relation are not relevant, we use for conciseness "$\cdot$" as a place-holder. We note $\mathbb{B}$ the type of Coq booleans and $\mathbb{Z}$ the type of (arbitrary-precision) integers.

2^{32} is abbreviated as β. The remainder of x modulo n is noted $x \mathbin{\mathsf{mod}} n$. The length of a list l is noted $|l|$.

$\mathsf{int}\ n$ is the type of machine integers encoded as bitstrings of n bits. Constants of this type are annotated with the length of the underlying bitstring, e.g., 0_{32} for the 32-bit constant zero, 1_{16} for the 16-bit constant one, etc. This type comes with various functions that mimic computer instructions, for example, right-shift of x by k is noted $x \gg k$. The function $(\cdot)_{\mathsf{int}\ n \to \mathbb{Z}}$ interprets a n-bit bitstring as an unsigned integer.

Multi-precision integers are encoded as lists of machine integers. Given a list l of machine words of type $\mathsf{int}\ 32$, $\sum_k l$ is the value, as an integer in $\mathbb{Z}$, of the multi-precision integer corresponding to the first k machine integers of l.

We use finite maps in several places, in particular to encode mutable memory and stores of variables. For a map m, if $x \in \mathsf{dom}\,(m)$, then the image y of x is such that $m(x) = \mathsf{Some}(y)$; if $x \notin \mathsf{dom}\,(m)$, we have $m(x) = \mathsf{None}$. The codomain of m is noted $\mathsf{codom}\,(m)$. The fact that two maps m_1, m_2 have disjoint domains is noted $m_1 \perp m_2$. $m|_d$ is the projection of the map m on the domain d. $m - d$ is the map m from which entries from domain d have been removed.

2.1 Pseudo-code

The pseudo-code is an imperative language with (arbitrary-precision) integers, assignments, and structured control-flow.

Variables are ranged over by $x, y, \ldots$ and carry integers as values. States of pseudo-code programs are ranged over by s and are finite maps from variables to values, or *stores* for short. For assignments, there is a language of arithmetic expressions ranged over by e; for branching structures, there is a language of boolean expressions ranged over by b. The value of the expression e (resp. b) in state s is noted $[\![e]\!]_s$ (resp. $[\![b]\!]_s$) and belongs to $\mathbb{Z}$ (resp. $\mathbb{B}$).

[2]At the time of this writing, this experiment is still in progress but we are confident that the main issues have been addressed. See Sect. 6 for details about what has been achieved and what remains to do.

Pseudo-code programs are ranged over by p. They are made of the two one-step commands `skip` and $x{:=}e$ and of the following control-flow commands: $p_1\ ;\ p_2$, $\texttt{while}\,(b)\,\{p\}$, and $\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2$. The (big-step) operational semantics is given by the relation $s \xrightarrow{p} s'$ that represents the execution of program p starting from state s and resulting in state s'.

Assertions in Hoare-logic triples are shallow-encoded, i.e., they are functions from states to Prop. All usual assertions can be defined this way; they connect to programs by sharing the same language of expressions.

For any assertions P, Q and program p, the Hoare-logic triples are noted $\{P\}\,p\,\{Q\}$. The formalization of Hoare-logic axioms is standard (details can be found in [7]).

2.2 Assembly

The assembly we are dealing with is a formalization of the SmartMIPS instruction set [4].

There are 32 general-purpose registers ranged over by $rx, ry, rk, rg, \ldots$ whose contents have type $\mathsf{int}\ 32$. Among them, there is in particular a special register `reg_zero` constantly holding 0_{32}. Since this is assembly, the syntax does not require any language of expressions; yet, to simplify notations, we introduce a small language ranged other by b for binary tests that are used in control-flow instructions (for example, the test of inequality between the contents of two registers is noted $rx \neq ry$).

There are about 30 one-step instructions taken from the standard documentation [4]. These instructions can be put together using the following control-flow commands[3] (assembly code is ranged over by c): $c_1\ \underline{;}\ c_2$, $\underline{\texttt{while}}(b)\{c\}$, and $\underline{\texttt{if}}\ b\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2$.

A *state* of an assembly program is a pair of a store and a *heap*. Here, stores (ranged over by st) are finite maps from registers to 32-bit integers (this includes not only general-purpose registers but also the MIPS *multiplier*, an additional set of three registers dedicated to cryptographic computations). Heaps (ranged over by h) are finite maps from natural numbers to 32-bit integers: we actually restrict ourselves to assembly programs that address memory by words, as customary with MIPS. States can actually be either *safe states*, prefixed with Some, or *error states*, noted None.

The (big-step) operational semantics for assembly is noted $s \succ c \twoheadrightarrow s'$ (details can be found in [8, 14]). This semantics is deterministic:

Lemma 2.1 (Determinism)**.** For any s, c, s', s'' such that $s \succ c \twoheadrightarrow s'$ and $s \succ c \twoheadrightarrow s''$ then $s' = s''$.

Assertions in Separation-logic triples are functions from (safe) states to Prop. Even though assembly has no arithmetic expressions, we introduce a small language of arithmetic expressions ranged over by e to ease the writing of assertions. Given e (a register, an arithmetic expression, or a boolean test), $[\![e]\!]_{st}$ (of type $\mathsf{int}\ 32$) represents the value of e in the store st. As an example of assertion $\lambda st, h.\, (e \mapsto l)\ st\, h$ holds in a store st and a heap h when h starts at the $[\![e]\!]_{st}$th word and contains consecutively the $|l|$ words of l.

Separation-logic triples are noted $[P]\,c\,[Q]$ and their definitions follow [5], modulo restrictions to accommodate finite memory (only β bytes).

3 Simulation of Arithmetic Functions: Illustrating Example

This section illustrates with an example the relation between a Hoare-logic triple for pseudo-code and a Separation-logic triple for assembly. The program in question is division by 2. In pseudo-code, it amounts to a single instruction: $x{:=}x\,/\,2$. The assembly implementation below performs in-place multi-precision division by 2. It uses repeated logical right shifts and implements its own

[3]Even though we restrict ourselves to structured assembly in this paper, our framework is more general in the sense that it provides also a formally-verified compiler to compile structured control-flow to labeled instructions and gotos [14]. This facility will let us accommodate gotos, but, for the sake of clarity, this paper does not deal with this issue.

overflow (more precisely, underflow) detection/propagation:

```
multi_div2 $k\ a\ i\ tmp\ prev\ next \overset{def}{=}$
 addiu $i\ k\ 0_{16}$ ;                        start at the most-significant word
 addiu $prev$ reg_zero $0_{16}$ ;              initialize overflow flag to 0
 while($i \neq$ reg_zero){                     loop until the least-significant word
   addiu $i\ i\ -1_{16}$ ;                     }
   lwxs $tmp\ i\ a$ ;                          } load $i$th word
   andi $next\ tmp\ 1_{16}$ ;                  detect potential overflow
   srl $tmp\ tmp\ 1_5$ ;                       shift the loaded word
   or $tmp\ tmp\ prev$ ;                       insert the overflow detected at the previous iteration
   addiu $prev\ next\ 0_{16}$ ;                }
   sll $prev\ prev\ 31_5$ ;                    } set detected overflow to most-significant bit position
   sll $next\ i\ 2_5$ ;                        }
   addu $next\ a\ next$ ;                      } set index for storing back the shifted word
   sw $tmp\ 0_{16}\ next$}                     store back the shifted word
```

The Hoare-logic triple for the pseudo-code of division by 2 is straightforward. It can be stated as follows: for any variable x and for any integer vx (a ghost variable, intuitively the value of x), we have $\{\lambda s.\ [\![x]\!]_s = vx\}\ x{:=}x\,/\,2\ \{\lambda s.\ [\![x]\!]_s = vx/2\}$.

The Separation-logic triple for assembly is more involved because one needs to take into account various restrictions about the registers and the values they carry. Examples of such restrictions are the fact that some instructions require data to be aligned at word-boundaries (i.e., are multiple of 4 (bytes)), or the fact that addressable memory is finite (β bytes). In the pre-condition of the following Separation-logic triple, the multi-precision integer A stored in memory is pointed to by register a and its length is stored in register k. In the post-condition, the multi-precision integer at the same position is noted A' and the fact that it is the quotient of A divided by 2 is captured by an equality relation between A and A'.

Lemma 3.1 (Separation-logic Triple for Multi-precision Division by 2)**.** For any pairwise distinct non-zero registers $k, a, i, tmp, prev, next$, for any strictly positive nk, for any na, va such that $4 \times na + 4 \times nk < \beta$ and $(va)_{\mathsf{int}\ 32\to\mathbb{Z}} = 4 \times na$, for any list A of length nk, we have:

$$\left[\lambda s, h.\ \underline{[\![a]\!]}_s = va \wedge \left(\underline{[\![k]\!]}_s\right)_{\mathsf{int}\ 32\to\mathbb{Z}} = nk \wedge (a \Mapsto A)\ \ s\ h\right]$$

$$\texttt{multi\_div2}\ k\ a\ i\ tmp\ prev\ next$$

$$\left[\begin{array}{c}\lambda s, h.\ \exists A'.\ |A'| = nk \wedge \underline{[\![a]\!]}_s = va \wedge \left(\underline{[\![k]\!]}_s\right)_{\mathsf{int}\ 32\to\mathbb{Z}} = nk \wedge (a \Mapsto A')\ \ s\ h\ \wedge \\ 2 \times \sum_{nk} A' + \left(\underline{[\![prev]\!]}_s \gg 31\right)_{\mathsf{int}\ 32\to\mathbb{Z}} = \sum_{nk} A\end{array}\right]$$

In this example, the relation between both triples is that the effect of the pseudo-code on the contents of the variable x is the same as the effect of the assembly program on the contents of the memory pointed to by register a. In other words, if x and a initially "point to" the same value, they still "point to" the same value after execution of both programs. We specify this relation formally in Sect. 5 and this is simulation w.r.t. this relation that we aim at formalizing.

4 Formalization of Simulation

To bridge Hoare-logic triples for pseudo-code and Separation-logic triples for assembly, we introduce a notion of simulation. Simulation techniques have a long history and a wide field of application that encompasses concurrent systems. There exist generic definitions of simulations [1], but since we do not intend to use concurrency, we refer instead to definitions biased toward imperative programs [10, 12, 13].

The notions of simulations below are parameterized by a relation $\mathcal{R}$ between, on the one hand, a store of variables for the pseudo-code, and, on the other hand, a register-file and a heap for the assembly.

Forward simulation states that the executions of two programs p and c preserve a relation $\mathcal{R}$ under some initial condition $\mathcal{I}$. The initial condition is important to accommodate assembly programs that may fail to preserve the relation $\mathcal{R}$ under some exceptional circumstances (e.g., overflows).

Definition 4.1 (Forward Simulation)**.** $p \ \leq^{\mathcal{I}}_{\mathcal{R}} \ c \overset{def}{=} \forall s, st, h.\ \mathcal{R}\ s\ st\ h \ \Rightarrow\ \mathcal{I}\ s\ st\ h \ \Rightarrow\ \forall s'.\ s \xrightarrow{p} s' \ \Rightarrow\ \exists st', h'.\ \mathsf{Some}(st, h) \succ c \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h') \wedge \mathcal{R}\ s'\ st'\ h'$

We introduce a notion of relational Hoare logic [6] that is used in composition lemmas to establish propagation of initial conditions (see Lemmas 4.1 and 4.4):

Definition 4.2 (Relational Hoare Logic)**.** $p \sim c : \mathcal{P} \rightsquigarrow \mathcal{Q} \overset{def}{=} \forall s, st, h.\ \mathcal{P}\ s\ st\ h \ \Rightarrow\ \forall s'.\ s \xrightarrow{p} s' \ \Rightarrow\ \forall st', h'.\ \mathsf{Some}(st, h) \succ c \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h') \ \Rightarrow\ \mathcal{Q}\ s'\ st'\ h'$

Forward simulation is preserved by sequential composition:

Lemma 4.1 (Simulation of Sequences)**.** $\forall \mathcal{R}, p, c, p', c', \mathcal{P}, \mathcal{Q}.\ p \sim c : \mathcal{P} \rightsquigarrow \mathcal{Q} \ \Rightarrow\ p \ \leq^{\mathcal{P}}_{\mathcal{R}} \ c \ \Rightarrow\ p' \ \leq^{\mathcal{Q}}_{\mathcal{R}} \ c' \ \Rightarrow\ p\ ;\ p' \ \leq^{\mathcal{P}}_{\mathcal{R}} \ c \mathbin{\underline{;}} c'$

Hereafter, we assume that we work with terminating programs. This is a reasonable assumption for our purpose since arithmetic functions are in general terminating programs. In this context, forward simulation is actually equivalent to *partial forward simulation*, a definition that is easier to work with formally:

Definition 4.3 (Partial Forward Simulation)**.** $p \ \lesssim^{\mathcal{I}}_{\mathcal{R}} \ c \overset{def}{=} p \sim c : \lambda s, st, h.\ \mathcal{R}\ s\ st\ h \wedge \mathcal{I}\ s\ st\ h \rightsquigarrow \mathcal{R}$

In general, this definition has the defect that any pseudo-code program simulates a non-terminating assembly program, but under the assumption that the assembly program terminates it implies forward simulation:

Definition 4.4. $\mathsf{safely\_terminating}\ \mathcal{R}\ c \overset{def}{=} \forall s, st, h.\ \mathcal{R}\ s\ st\ h \ \Rightarrow\ \exists st', h'.\ \mathsf{Some}(st, h) \succ c \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h')$

Lemma 4.2. $\forall \mathcal{R}, \mathcal{I}, p, c.\ p \ \lesssim^{\mathcal{I}}_{\mathcal{R}} \ c \ \Rightarrow\ \mathsf{safely\_terminating}\ \mathcal{R}\ c \ \Rightarrow\ p \ \leq^{\mathcal{I}}_{\mathcal{R}} \ c$

Moreover, since the execution of assembly is deterministic (Lemma 2.1), we also have that forward simulation implies backward simulation[4] (under the assumption that the pseudo-code terminates):

Definition 4.5 (Backward Simulation)**.** $p \ \geq^{\mathcal{I}}_{\mathcal{R}} \ c \overset{def}{=} \forall s, st, h.\ \mathcal{R}\ s\ st\ h \ \Rightarrow\ \mathcal{I}\ s\ st\ h \ \Rightarrow\ \forall st', h'.\ \mathsf{Some}(st, h) \succ c \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h') \ \Rightarrow\ \exists s'.\ s \xrightarrow{p} s' \wedge \mathcal{R}\ s'\ st'\ h'$

Definition 4.6. $\mathsf{terminating}\ p \overset{def}{=} \forall s.\ \exists s'.\ s \xrightarrow{p} s'$

Lemma 4.3. $\forall \mathcal{R}, \mathcal{I}, p, c.\ \mathsf{terminating}\ p \ \Rightarrow\ p \ \leq^{\mathcal{I}}_{\mathcal{R}} \ c \ \Rightarrow\ p \ \geq^{\mathcal{I}}_{\mathcal{R}} \ c$

We define a notion of simulation between a boolean expression b for the pseudo-code and a pair of an assembly snippet *pre_b* and of boolean test *post_b* about registers' contents:

Definition 4.7 (Simulation of Boolean Expressions)**.**

$$
\begin{aligned}
b \ \leq_{\mathcal{R}} \ \langle pre\_b, post\_b \rangle \overset{def}{=} & \forall s, st, h.\ \mathcal{R}\ s\ st\ h \ \Rightarrow \\
& \Big([\![b]\!]_s \ \Rightarrow\ \exists st'.\ \mathsf{Some}(st, h) \succ pre\_b \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h) \wedge \underline{[\![post\_b]\!]}_{st'} \Big) \wedge \\
& \Big(\neg [\![b]\!]_s \ \Rightarrow\ \exists st'.\ \mathsf{Some}(st, h) \succ pre\_b \mathrel{-\!\!\triangleright} \mathsf{Some}(st', h) \wedge \neg \underline{[\![post\_b]\!]}_{st'} \Big)
\end{aligned}
$$

[4]This is the notion of backward simulation from [13], that departs from the definition of backward simulation in the theory of automata [1].

To prove a simulation between a while-loop in pseudo-code and a while-loop in assembly, it essentially suffices to prove simulation between the boolean tests of the while-loops and the code in the body of the while-loops. In addition, the initial condition must be set to an invariant preserved by the body of the while-loops.

Definition 4.8 (Invariant Relation)**.** $\mathsf{inv}_c\,\mathcal{R} \overset{def}{=} \forall s, st, h.\ \mathcal{R}\ s\ st\ h\ \Rightarrow\ \forall st', h'.\ \mathsf{Some}(st, h) \succ c \twoheadrightarrow \mathsf{Some}(st', h')\ \Rightarrow\ \mathcal{R}\ s\ st'\ h'$

Lemma 4.4 (Simulation of While-loops)**.**

$$\begin{gathered}\forall b, pre\_b, post\_b, p, c, \mathcal{R}, \mathcal{I}.\quad \mathsf{inv}_{pre\_b}\,\lambda s, st, h.\ \mathcal{R}\ s\ st\ h \wedge \mathcal{I}\ s\ st\ h\ \Rightarrow \\ p \sim c : \lambda s, st, h.\ \mathcal{I}\ s\ st\ h \wedge \underline{[\![post\_b]\!]}_{st} \wedge [\![b]\!]_s \rightsquigarrow \mathcal{I}\ \Rightarrow \\ b\ \leq_{\mathcal{R}}\ \langle pre\_b, post\_b\rangle\ \Rightarrow\ p\ \leq_{\mathcal{R}}^{\lambda s, st, h.\ \mathcal{I}\ s\ st\ h \wedge \underline{[\![post\_b]\!]}_{st} \wedge [\![b]\!]_s}\ c\ \Rightarrow \\ \texttt{while}\,(b)\,\{p\}\ \leq_{\mathcal{R}}^{\mathcal{I}}\ pre\_b\ \underline{;}\ \underline{\texttt{while}}(post\_b)\{c\ \underline{;}\ pre\_b\}\end{gathered}$$

5 Formalization of Simulation for Arithmetic Functions

We provide a concrete example of relation for arithmetic functions. This is actually the formalization of the relation informally explained in Sect. 3. We instantiate the generic definitions of Sect. 4 with this relation and provide an additional lemma to derive properties of assembly programs from Hoare-logic triple for pseudo-code that simulates them.

5.1 Formal Manipulation of Heaps

The formalization of simulation for arithmetic functions calls for precise statements about heaps. The formalization of those statements and their formal manipulation cause technical difficulties. In order to facilitate the formalization, we introduce (and instrument) the following definitions for heaps.

Projection $h|_{a,k}$ is the projection of the heap h to the interval-domain starting at a and of length k. The following abbreviation will be useful:

$$h|_{[rx,rk)_{st}} \overset{def}{=} h|_{\left(\underline{[\![rx]\!]}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}} \div 4, \left(\underline{[\![rk]\!]}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}}}$$

Here, division by 4 comes from the fact that memory in our formalization of assembly is word-addressable by default.

Difference The following abbreviation will be useful: $h - [rx, rk)_{st} \overset{def}{=} h - \mathsf{dom}\left(h|_{[rx,rk)_{st}}\right)$

Constructions of Heaps from Lists $(v)_{\mathbb{Z} \xrightarrow{k} \mathsf{int}\ n}$ is a partial function that takes an integer v and returns a multi-precision integer of length k with integers of type $\mathsf{int}\ n$ encoding the value v (if space allows). The following abbreviation will be useful:

$$(v)_{\mathbb{Z} \xrightarrow{rk, st} \mathsf{int}\ 32} \overset{def}{=} (v)_{\mathbb{Z} \xrightarrow{\left(\underline{[\![rk]\!]}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}}} \mathsf{int}\ 32}$$

$\mathsf{list\_to\_heap}\ x\ l$ is the heap that starts at x and in which the contents of l are stored consecutively (it therefore has $|l|$ entries by construction). The following abbreviation will be useful:

$$\mathsf{list\_to\_heap}\ rx, rk, st\ v \overset{def}{=} \mathsf{list\_to\_heap}\ \left(\left(\underline{[\![rx]\!]}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}} \div 4\right)\ (v)_{\mathbb{Z} \xrightarrow{rk, st} \mathsf{int}\ 32}$$

5.2 Implementation Relations

First, we define a relation $(x, s) \sim ((rk, rx), (st, h))$ between a variable x in a store s in pseudo-code and a pair of registers (rk, rx) in a state (st, h) in assembly. Intuitively, on the one hand, x contains some non-negative integer v; on the other hand, rx points to a position in the heap h where are stored rk 32-bit integers that encode v as an unsigned multi-precision integer.

Definition 5.1 (Implementation of a Variable)**.**

$$\begin{array}{l}(x, s) \sim ((rk, rx), (st, h)) \overset{def}{=} \\ \mathsf{let}\ nk\ :=\ \left(\llbracket rk \rrbracket_{st}\right)_{\mathsf{int\ 32} \to \mathbb{Z}}\ \mathsf{in\ let}\ vx\ :=\ \left(\llbracket rx \rrbracket_{st}\right)_{\mathsf{int\ 32} \to \mathbb{Z}}\ \mathsf{in\ let}\ nx\ :=\ vx \div 4\ \mathsf{in} \\ h = \mathsf{list\_to\_heap}\ nx\ \left(\llbracket x \rrbracket_s\right)_{\mathbb{Z} \xrightarrow{nk} \mathsf{int\ 32}} \wedge 0 < nk \wedge vx + 4 \times nk < \beta \wedge vx\ \mathsf{mod}\ 4 = 0 \wedge 0 \leq \llbracket x \rrbracket_s < \beta^{nk}\end{array}$$

Now, we can define a relation between a store s for the pseudo-code and a state (st, h) for the assembly. Intuitively, it means that each variable in s is implemented in the state (st, h), in the sense of the above relation (Definition 5.1). This new relation noted $s \sim_{rk,d} (st, h)$ below is indexed by a register rk that holds the size of multi-precision integers in assembly and a finite map d that associates variables with registers.

Definition 5.2 (Implementation of States)**.**

$$\begin{array}{r}s \sim_{rk,d} (st, h) \overset{def}{=} \left(\forall x, rx.\ d(x) = \mathsf{Some}(rx)\ \Rightarrow\ (x, s) \sim \left((rk, rx), \left(st, h|_{[rx,rk)_{st}}\right)\right)\right) \wedge \\ \left(\forall x, y.\ x \neq y\ \Rightarrow\ \forall rx, ry.\ d(x) = \mathsf{Some}(rx)\ \Rightarrow\ d(y) = \mathsf{Some}(ry)\ \Rightarrow\ h|_{[rx,rk)_{st}} \perp h|_{[ry,rk)_{st}}\right)\end{array}$$

Instantiation of Simulation Relations We can instantiate the generic definitions of Sect. 4 with the concrete relation for arithmetic functions:

$$\begin{array}{lll}p\ \lessapprox^{\mathcal{I}}_{rk,d}\ c & \overset{def}{=} & p\ \lesssim^{\mathcal{I}}_{\cdot \sim_{rk,d} (\cdot,\cdot)}\ c \\ p\ \leqq^{\mathcal{I}}_{rk,d}\ c & \overset{def}{=} & p\ \leq^{\mathcal{I}}_{\cdot \sim_{rk,d} (\cdot,\cdot)}\ c \\ b\ \leqq_{rk,d}\ \langle pre\_b, post\_b \rangle & \overset{def}{=} & b\ \leq_{\cdot \sim_{rk,d} (\cdot,\cdot)}\ \langle pre\_b, post\_b \rangle\end{array}$$

5.3 From Pseudo-code Hoare-logic Triple to Assembly

As explained in the introduction, we want to formally prove properties of assembly implementations by reusing formal proofs about pseudo-code. This is the approach that we use in [14] to prove unpredictability for an assembly implementation of a pseudo-random number generator. The lemma below generalizes this approach. It shows that one can formally prove a property for an assembly program given (1) a formal proof for a pseudo-code program, (2) a proof of forward simulation, and (3) two state-transformation functions (encode and decode below). We think that the proof of forward simulation can be reasonably easy to perform provided an adequate library of basic arithmetic functions (Sect. 6.2 illustrates this point). The two state-transformation functions build an assembly state from a pseudo-code state (respectively, a pseudo-code state from an assembly state) in a way compatible with the implementation relation for arithmetic functions, as expressed below by the conditions (enc) and (dec). ($\mathsf{cmd\_vars}(p)$ are the variables that appear in the pseudo-code p.)

Lemma 5.1.

$$\begin{array}{lr}\forall rk, d, p, c.\ \mathsf{safely\_terminating}\ (\cdot \sim_{rk,d} (\cdot,\cdot))\ c\ \Rightarrow\ \mathsf{terminating}\ p\ \Rightarrow\ \mathsf{cmd\_vars}(p) \subseteq \mathsf{dom}\,(d)\ \Rightarrow & \\ \forall \mathsf{encode}, \mathsf{decode}. & \\ \quad (\forall s.\ s \sim_{rk,d} \mathsf{encode}\ d\ s)\ \Rightarrow & \text{(enc)} \\ \quad (\forall s.\ \mathsf{decode}\ d\ s \sim_{rk,d} s \wedge \mathsf{dom}\,(\mathsf{decode}\ d\ s) = \mathsf{dom}\,(d))\ \Rightarrow & \text{(dec)} \\ p\ \lessapprox^{\mathcal{I}}_{rk,d}\ c\ \Rightarrow & \\ \forall P_0, P_f.\ \{P_0\}\, p\, \{P_f\}\ \Rightarrow & \\ \forall s. & \\ [\lambda st, h.\ \mathsf{encode}\ d\ s = (st, h) \wedge P_0\ s \wedge \mathcal{I}\ s\ st\ h]\, c\, [\lambda st, h.\ P_f(\mathsf{decode}\ d\ (st, h) \cup (s - \mathsf{dom}\,(d)))] & \end{array}$$

It is not difficult to prove this lemma using Lemma 4.3 but it remains to validate it by ensuring that the hypotheses do not preclude applicability in general. We defer this to future work.

6 Examples of Formal Proofs of Simulation

6.1 Simulation of Division by 2

We show how to prove simulation between the two programs of Sect. 3 (division by 2). Such a simulation proof is already interesting in itself because it relates an arguably hard-to-read assembly program to a single instruction of pseudo-code, and therefore constitutes a form of high-level specification. But the real end of such a simulation proof is to integrate a library to be reused to prove by composition simulation of larger programs, as done in the next section.

Lemma 6.1 (Partial Forward Simulation for Division by 2)**.**

$$\begin{array}{l}\forall x, rk, rx, d, a_0, a_1, a_2, a_3.\\ \quad \mathsf{list\_is\_set}(rk :: rx :: a_0 :: a_1 :: a_2 :: a_3 :: \mathtt{reg\_zero} :: nil) \Rightarrow\\ \quad \mathsf{codom}\,(d) \cap (a_0 :: a_1 :: a_2 :: a_3 :: nil) = \emptyset \;\Rightarrow\; x \notin \mathsf{dom}\,(d) \;\Rightarrow\; rx \notin \mathsf{codom}\,(d) \;\Rightarrow\\ \quad x{:=}x/2 \;\lesssim^{\lambda\_.\mathsf{True}}_{rk,(x\mapsto rx)\uplus d}\; \mathtt{multi\_div2}\; rk\; rx\; a_0\; a_1\; a_2\; a_3\end{array}$$

Formal Proof Overview Under the hypotheses

$$\mathsf{Some}(st, h) \succ \mathtt{multi\_div2}\; rk\; rx\; a_0\; a_1\; a_2\; a_3 \twoheadrightarrow \mathsf{Some}(st', h') \tag{1}$$

$$s \xrightarrow{x:=x\,/\,2} s' \tag{2}$$

$$s \sim_{rk,(x\mapsto rx)\uplus d} (st, h) \tag{3}$$

we want to show that $s' \sim_{rk,(x\mapsto rx)\uplus d} (st', h')$, what decomposes into two subgoals:

1. We show that for any pair (x', rx') of the map $(x \mapsto rx) \uplus d$, we have

   $$(x', s') \sim \left((rk, rx'), \left(st', h'|_{[rx',rk)_{st'}}\right)\right)$$

   - If $x' \in \mathsf{dom}\,(d)$ (i.e., $x \neq x'$), it boils down to show that $h|_{[rx',rk)_{st}} = h'|_{[rx',rk)_{st'}}$. This is a consequence of $h - [rx, rk)_{st} = h' - [rx, rk)_{st'}$ that we prove as follows. We first show that there exist st'', h'' such that

     $$\mathsf{Some}(st, h|_{[rx,rk)_{st}}) \succ \mathtt{multi\_div2}\; rk\; rx\; a_0\; a_1\; a_2\; a_3 \twoheadrightarrow \mathsf{Some}(st'', h'')$$

     using the Separation-logic triple (Lemma 3.1) and Lemma A.2. From there, the conclusion follows from Lemma A.3 and hypothesis (1).
   - If $x' = x$, it boils down to show that $h'|_{[rx,rk)_{st'}} = \mathsf{list\_to\_heap}\; rx, rk, st\; [\![x]\!]_{s'}$. We first instantiate the Separation-logic triple (Lemma 3.1) with the list $([\![x]\!]_s)_{\mathbb{Z}\xrightarrow{rk,st}\mathsf{int\ 32}}$. Then by the frame rule and the soundness of Separation-logic, we find two heaps h_1, h_2 such that $h' = h_1 \uplus h_2$ and $(rx \Mapsto A')\;\; st'\; h_1$ holds with $\sum_{[\![rk]\!]_{st'}} A' = [\![x]\!]_s \div 2$. By hypothesis (2), we know that $[\![x]\!]_{s'} = [\![x]\!]_s \div 2$. From the assertion $(rx \Mapsto A')\;\; st'\; h_1$, we finally prove that $h'|_{[rx,rk)_{st'}} = \mathsf{list\_to\_heap}\; \left(\left([\![rx]\!]_{st'}\right)_{\mathsf{int\ 32}\to\mathbb{Z}} \div 4\right)\; A'$.

2. We show that for any two distinct variables x, y, $h'|_{[rx,rk)_{st'}} \perp h'|_{[ry,rk)_{st'}}$ with $(x \mapsto rx) \uplus d(x) = \mathsf{Some}(rx)$ (resp. for y, ry, and k, rk). By hypothesis (3), heaps were disjoint before execution. We can show that the registers in $\mathsf{codom}\,((x \mapsto rx) \uplus d)$ are unchanged by using Lemma A.1, and therefore heaps are still disjoint after the execution.

6.2 Simulation of the Prelude of Binary Extended GCD Algorithm

In this section, the simulation is essentially proved using the composition Lemmas 4.1 and 4.4 and the simulation proofs for division and multiplication by 2.

The pseudo-code is the prelude of the the binary extended GCD algorithm [3]. It is an interesting step because is reduces the size of data by means of mere shifts; the GCD is preserved (modulo shifts) because when a and b are even, $\mathsf{gcd}(a, b) = 2 \times \mathsf{gcd}(a/2, b/2)$. The corresponding assembly program and the simulation are given below. Note that the invariant mainly limits the size of g relatively to x and does not deal with any GCD property.

Lemma 6.2.

$\forall x, y, g, k, nx, ny, ng, nk, rx, ry, rg, rk, a_0, a_1, a_2, a_3.\ \mathsf{list\_is\_set}(x :: y :: g :: k :: nil) \ \Rightarrow$
$\quad \mathsf{list\_is\_set}(rk :: rx :: ry :: rg :: a_0 :: a_1 :: a_2 :: a_3 :: \texttt{reg\_zero} :: nil) \ \Rightarrow$

$$\begin{array}{l}\texttt{while}\left(\begin{array}{l} x \ \texttt{mod}\ 2 == 0\ \&\& \\ y\ \texttt{mod}\ 2 == 0\end{array}\right) \\ \quad \{x{:=}x\,/\,2\ ; \\ \quad\ \ y{:=}y\,/\,2\ ; \\ \quad\ \ g{:=}2 * g\}\end{array} \quad \underset{\substack{rk,\ (x \mapsto rx) \uplus \\ (y \mapsto ry) \uplus \\ (g \mapsto rg) \uplus \\ (k \mapsto rk)}}{\overset{\substack{\lambda s, st, h. 0 \le \llbracket g \rrbracket_s \wedge 0 < \llbracket x \rrbracket_s \wedge \\ 0 < \left(\underline{\llbracket rk \rrbracket}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}} \wedge \\ \llbracket x \rrbracket_s \times \llbracket g \rrbracket_s < \beta^{\left(\underline{\llbracket rk \rrbracket}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}}}}}{\leqq}} \quad \begin{array}{l} \texttt{multi\_is\_even}\ rx\ a_0\ \underline{;} \\ \texttt{multi\_is\_even}\ ry\ a_1\ \underline{;} \\ \texttt{and}\ a_0\ a_0\ a_1\ \underline{;} \\ \underline{\texttt{while}}(a_0 \neq \texttt{reg\_zero})\{ \\ \quad \texttt{multi\_div2}\ rk\ rx\ a_0\ a_1\ a_2\ a_3\ \underline{;} \\ \quad \texttt{multi\_div2}\ rk\ ry\ a_0\ a_1\ a_2\ a_3\ \underline{;} \\ \quad \texttt{multi\_mul2}\ rk\ rg\ a_0\ a_1\ a_2\ a_3\ \underline{;} \\ \quad \texttt{multi\_is\_even}\ rx\ a_0\ \underline{;} \\ \quad \texttt{multi\_is\_even}\ ry\ a_1\ \underline{;} \\ \quad \texttt{and}\ a_0\ a_0\ a_1\} \end{array}$$

Formal Proof Overview We apply Lemma 4.4 and get two simulation subgoals and two relational Hoare logic subgoals. The relational Hoare logic subgoals are easily disposed of using lemmas such as Lemma A.1. Let us focus on the two simulation subgoals.

1. We have to prove that

$$x\ \texttt{mod}\ 2 == 0\ \&\&\ y\ \texttt{mod}\ 2 == 0 \ \underset{\substack{rk,\ (x \mapsto rx) \uplus \\ (y \mapsto ry) \uplus \\ (g \mapsto rg) \uplus \\ (k \mapsto rk)}}{\leqq} \left\langle \begin{array}{l} \texttt{multi\_is\_even}\ rx\ a_0\ \underline{;} \\ \texttt{multi\_is\_even}\ ry\ a_1\ \underline{;} \\ \texttt{and}\ a_0\ a_0\ a_1 \end{array}, a_0 \neq \texttt{reg\_zero} \right\rangle$$

   There are two cases according to whether the boolean test is true or not. Let us treat the case where the boolean test is true. The goal boils down to show under the hypotheses

$$\mathsf{Some}(st, h) \succ \begin{array}{l} \texttt{multi\_is\_even}\ rx\ a_0\ \underline{;} \\ \texttt{multi\_is\_even}\ ry\ a_1\ \underline{;} \\ \texttt{and}\ a_0\ a_0\ a_1 \end{array} \twoheadrightarrow \mathsf{Some}(st', h') \tag{1}$$

$$s \sim_{rk,(x \mapsto rx) \uplus (y \mapsto ry) \uplus (g \mapsto rg) \uplus (k \mapsto rk)} (st, h) \qquad \llbracket x \rrbracket_s \ \mathsf{mod}\ 2 = 0 \qquad \llbracket y \rrbracket_s \ \mathsf{mod}\ 2 = 0$$

   that $\underline{\llbracket a_0 \rrbracket}_{st'} \neq 0_{32}$. By the Separation-logic triple of `multi_is_even` (Lemma 6.3 below) and the soundness of Separation-logic applied to the two occurrences of `multi_is_even` in hypothesis (1), we get that $\underline{\llbracket a_0 \rrbracket}_{st'} = 1_{32}$ and $\underline{\llbracket a_1 \rrbracket}_{st'} = 1_{32}$. By inversion of $\texttt{and}\ a_0\ a_0\ a_1$, we deduce that $\underline{\llbracket a_0 \rrbracket}_{st'} = 1_{32}$.

2. We have to prove that

$$\begin{array}{l} x{:=}x\,/\,2; \\ y{:=}y\,/\,2; \\ g{:=}2 * g \end{array} \quad \underset{\substack{rk,\ (x \mapsto rx) \uplus \\ (y \mapsto ry) \uplus \\ (g \mapsto rg) \uplus \\ (k \mapsto rk)}}{\overset{\substack{\lambda s, st, h. 0 \le \llbracket g \rrbracket_s \wedge 0 < \llbracket x \rrbracket_s \wedge \\ 0 < \left(\underline{\llbracket rk \rrbracket}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}} \wedge \\ \llbracket x \rrbracket_s \times \llbracket g \rrbracket_s < \beta^{\left(\underline{\llbracket rk \rrbracket}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}}} \wedge \\ \underline{\llbracket a_0 \neq \texttt{reg\_zero} \rrbracket}_{st} \wedge \\ \llbracket x\ \texttt{mod}\ 2 == 0\ \&\&\ y\ \texttt{mod}\ 2 == 0 \rrbracket_s}}{\leqq}} \quad \begin{array}{l} \texttt{multi\_div2}\ rk\ rx\ a_0\ a_1\ a_2\ a_3\ \underline{;} \\ \texttt{multi\_div2}\ rk\ ry\ a_0\ a_1\ a_2\ a_3\ \underline{;} \\ \texttt{multi\_mul2}\ rk\ rg\ a_0\ a_1\ a_2\ a_3 \end{array}$$

   We decompose this goal by Lemma 4.1 that requires us to provide a new initial condition for the continuations of the programs. Since these continuations use multiplication by 2 whose simulation holds under restriction (informally, the multiplied multi-precision integer should not grow out of bounds), we refine the initial condition to take advantage of the fact that division decreases the size of x:

$$\lambda s, st, h.\ \ 0 \le \llbracket g \rrbracket_s \ \wedge\ 0 < \llbracket x \rrbracket_s \ \wedge\ \llbracket x \rrbracket_s \times \llbracket g \rrbracket_s < \beta^{\left(\underline{\llbracket rk \rrbracket}_{st}\right)_{\mathsf{int}\ 32 \to \mathbb{Z}} - 1}$$

We now have to prove the simulation of division by 2

$$x{:=}x\,/\,2 \;\; \preceq^{\lambda\_.\mathsf{True}}_{rk,(x\mapsto rx)\uplus(y\mapsto ry)\uplus(g\mapsto rg)\uplus(k\mapsto rk)} \;\; \texttt{multi\_div2}\ rk\ rx\ a_0\ a_1\ a_2\ a_3$$

which is given by Lemmas 6.1 and 4.2, and also the proof that

$$\mathsf{safely\_terminating}\ (\cdot \sim_{\cdot,\cdot} (\cdot,\cdot))\ \ (\texttt{multi\_div2}\ rk\ rx\ a_0\ a_1\ a_2\ a_3)$$

which is a traditional termination proof.

The proof then goes on similarly for the continuation of the programs, using once time again the simulation of division by 2 and finally the simulation for multiplication by 2 (not displayed here for lack of space, see [16] for details).

Lemma 6.3 (Separation-logic Triple for Multi-precision Parity Check)**.** For all registers a, ret, for all $0 < nk$, for all lists A of length nk, we have:

$$\begin{gathered}[\lambda s, h.\ (a \mapsto\!\!> A)\ \ s\ \ h] \\ \texttt{multi\_is\_even}\ a\ ret \\ \left[\lambda s, h.\ (\mathsf{Zeven}\ \ (\textstyle\sum_{nk} A)\ \Rightarrow\ [\![ret]\!]_s = 1_{32}) \wedge (\mathsf{Zodd}\ \ (\textstyle\sum_{nk} A)\ \Rightarrow\ [\![ret]\!]_s = 0_{32})\right]\end{gathered}$$

6.3 Overview of Coq Scripts

All the definitions, lemmas and examples explained in this paper are formalized in the Coq proof-assistant. Figure 1 gives an overview of the whole formalization in terms of script files; details are available online [16]. A large part of this formalization is adapted from previous work. New scripts are marked as $\star$, scripts that required substantial improvements are marked as *.

`lib/stdlib_ext/`	*Small extensions to the standard libraries*	
...		
`lib/contrib/`	*Contributed libraries*	
↳ * `finmap.v`		Finite maps
↳ * `machine_int.v`		Machine integers (type $\mathsf{int}\ n$)
`seplog/`	*Formalization of pseudo-code*	
↳ `bipl.v`, `seplog.v`,...		Operational semantics and separation logic
↳ $\star$ `syntax.v`		Lemmas about the operational semantics
`cryptoasm/`	*Formalization of assembly*	
↳ `mips_bipl.v`		Expressions and formulas
↳ `mips_cmd.v`,...		Operational semantics
↳ `mips_seplog.v`, `mips_frame.v`,...		Separation logic
↳ $\star$ `mips_syntax.v`		Lemmas about the operational semantics
↳ $\star$ `multi_div2_{prg,triple}.v`		Multi-precision division by 2
↳ $\star$ `multi_mul2_{prg,triple}.v`		Multi-precision multiplication by 2
↳ $\star$ `multi_is_even_{prg,triple}.v`		Parity check
...		Other arithmetic functions
`begcd/`	*Formal verification of the binary extended GCD algorithm*	
↳ $\star$ `simu.v`		Formalization of simulations
↳ $\star$ `multi_div2_simu.v`		Simulation for division by 2 and termination proof
↳ $\star$ `multi_mul2_simu.v`		Simulation for multiplication by 2 and termination proof
↳ $\star$ `begcd.v`		Simulation for the binary extended GCD algorithm (in progress)

Figure 1: Overview of the Formalization

The formalization relies on several contributed libraries including in particular a library for finite maps (`finmap.v`, originating from [7], newly extended to deal with projections and transformations from/to lists, as seen in Sect. 5.1) and for finite-size integers (`machine_int.v`, originating from [8], newly extended with lemmas to formally verify multi-precision division and multiplication by 2).

The formalization of pseudo-code comes from [7] and the formalization of assembly comes from [8, 14]. For the purpose of this paper, we actually spent much time to clean up both formalizations using for example Coq modules and notations (while preserving our previous results). As explained in Appendix A, formal proofs of simulation require new lemmas to reason about the operational semantics; these lemmas appear in files `syntax`, `mips_syntax.v`. This is on top of these two formalizations of pseudo-code and assembly that we define our notions of simulation as well as their properties (the basic properties from Sect. 4, composition Lemmas 4.1 and 4.4, the Lemma 5.1), see file `simu.v`.

From previous work [8, 14], we inherit several arithmetic functions in assembly with their formal proofs of functional correctness. For the purpose of this paper, we additionally formally verified assembly implementation of multi-precision division and multiplication by 2 (files `multi_{div2,mul2}_{prg,triple}.v`). The formal proofs of simulation for division by 2 explained in Sect. 6.1 can be found in file `multi_div2_simu.v` (resp. in file `multi_mul2_simu.v` for multiplication by 2). We regard these formalizations of arithmetic functions in assembly as the starting point of a basic library out of which larger arithmetic functions and cryptographic schemes can be formally verified.

The formal proof (in progress) of simulation for the binary extended GCD algorithm explained in Sect. 6.2 can be found in file `begcd.v` and relies on the library of basic arithmetic functions (see the previous paragraph), the composition lemmas about simulation (see file `simu.v`), and various lemmas about the operational semantics of pseudo-code and assembly.

7 Related Work

Simulation proofs are at the heart of the formal verification of the compiler in [13]. Given the scale of the latter experiment, it would not be surprising that many issues on reasoning about operational semantics that we are currently facing have been addressed to some extent in [13]. This being said, we can nevertheless point out that there are differences between establishing a relation for a compiler-pass whose end-programs are closely related and establishing a relation between pseudo-code and assembly that are more apart. Moreover, 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 proof of correctness properties about a family of hand-written programs.

All things being relative, our approach shares similarities with the one used to formally verify the seL4 micro-kernel [10, 12]; there, the notions of forward simulation and Hoare logic also play a central role. Yet, there is an important difference in terms of approach: we aim at a library of arithmetic functions equipped with formal proofs in order to formally verify larger arithmetic functions by composition; in contrast, in [10, 12], the proof effort targets only one software application whose large scale induces biases such as an intermediate simulation language or the extensive use of a VCG. In addition to our library-centric approach, we also think that our focus on assembly (instead of C in [10, 12]) can also bring new technical insights.

8 Conclusion and Future Work

In this paper, we aimed at setting up an experimental framework in the Coq proof-assistant to formally verify hand-written arithmetic functions in assembly by reusing proofs for programs written in pseudo-code. For this purpose, we explained how we instrument a notion of simulation between pseudo-code and assembly: thanks to composition lemmas, simulations can be proved incrementally and systematically. We showed how we instantiate this notion of simulation with a concrete relation for arithmetic functions and applied it to a small-scale but concrete example: the prelude of an assembly program that implements the binary extended GCD algorithm. Given a formal proof of simulation, correctness proofs about the pseudo-code can be transported at the level of assembly by generic lemmas. We think that this approach, if supported by an existing library of certified basic arithmetic functions as the one we built in our previous work, improves the scalability of formal verification of hand-written, large arithmetic functions.

We are now pursuing the experiment started in Sect. 6.2 (formal construction of an assembly implementation of the binary extended GCD algorithm): our goal is to get of formal proof of functional correctness for assembly from a proof of functional correctness of the pseudo-code. The formal verification is slowed down by many low-level details on reasoning about the semantics of pseudo-code and assembly that we find difficult to automate satisfactorily. In order to consider more elaborate arithmetic functions, it will become important to improve the generality of our results, in particular the relation for simulation of arithmetic functions of Sect. 5 and the Lemma 5.1. This issue is raised by the introduction of signed multi-precision integers (necessary for the binary extended GCD algorithm) and also multi-precision integers with various lengths (as used in the BBS pseudo-random bit generator). Equipped with the framework introduced in this paper, we also plan to rework the result of [14] to enable more generic encode/decode functions; indeed, the encode/decode functions in [14] have the defect of imposing a fixed memory layout.

Acknowledgments The author thanks the reviewers of the 12th JSSST Workshop on Programming and Programming Languages. This work is partially supported by KAKENHI 21700048.

References

[1] Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. Information and Computation 121(2): 214–233. 1995.

[2] The Coq Development Team. Reference Manual. Version 8.2. `http://coq.inria.fr`. 2009.

[3] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. Chapter 14. CRC Press, 2001.

[4] MIPS Technologies, Inc. MIPS32 4KS Processor Core Family Software User's Manual. Revision 01.03. June 12, 2001.

[5] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symp. on Logic in Computer Science (LICS 2002)*, p. 55–74.

[6] Nick Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, p. 14–25.

[7] Nicolas Marti, Reynald Affeldt and Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th Int. Conf. on Formal Engineering Methods (ICFEM 2006)*, vol. 4260 of LNCS, p. 400–419. Springer, Oct. 2006.

[8] Reynald Affeldt and Nicolas Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In *11th Annual Asian Computing Science Conf. (ASIAN 2006), Focusing on Secure Software and Related Issues*, vol. 4435 of LNCS, p. 346–360. Springer, Jan. 2008.

[9] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science 373(3), 273–302. Elsevier, 2007.

[10] David Cock, Gerwin Klein, and Thomas Sewell. Secure Microkernels, State Monads and Scalable Refinement. In *21st Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, vol. 5170 of LNCS, p. 167–182 . Springer, Aug. 2008.

[11] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In *Int. Conf. on Information Security and Cryptology*, volume 5461 of LNCS, p. 368–382. Springer, 2009.

[12] Simon Wood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the Gap—A Verification Framework for Low-Level C. In *22nd Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, vol. 5674 of LNCS, p. 500–515. Springer, Aug. 2009.

[13] Xavier Leroy. A formally verified compiler back-end. Journal of Automated Reasoning 43(4):363–446. Dec. 2009.

[14] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS. Electronic Communications of the EASST 23. Proceeding of the *9th Int. Workshop on Automated Verification of Critical Systems (AVoCS 2009)*. Jan. 2010.

[15] Holger Gast and Julia Trieflinger. High-level proofs about low-level programs. Electronic Communications of the EASST 23. Proceeding of the *9th Int. Workshop on Automated Verification of Critical Systems (AVoCS 2009)*. Jan. 2010.

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

A Some Lemmas About Operational Semantics

In the course of proving simulation, we need to prove various technical facts about states (stores of variables for the pseudo-code, register files and heaps for assembly programs). Those technical facts intervene in establishing preservation of the relation of Definition 5.2. At this stage of our work we are not close to a satisfactorily complete set of lemmas to prove the technical facts about states that arise from simulation proofs. For the sake of illustration, let us nevertheless explain what kind of lemmas turned out to be useful in Sect. 6.

About Stores Among other things, we are led to verify how the execution modifies the contents of variables. It is often not difficult to establish that a variable is left untouched by the execution of a program ($\mathsf{modified\_regs}(c)$ is the set of registers potentially modified by execution of c):

Lemma A.1 (Variable Unchanged)**.** For all st, h, c, st', h', x such that $\mathsf{Some}(st, h) \succ c \twoheadrightarrow \mathsf{Some}(st', h')$ and that $x \notin \mathsf{modified\_regs}(c)$, we have $\underline{[\![x]\!]}_{st} = \underline{[\![x]\!]}_{st'}$.

Establishing the value of variables that are modified by execution is a different matter. Since we aim at formally verifying programs by composition of assembly programs, a systematic and rational approach from the engineering point of view is to rely on readily available Hoare/Separation-logic triples. Indeed, such triples are a standard form of specification and from them we can infer by the soundness of the underlying Hoare/Separation-logic equivalent information about operational semantics.

About Heaps The situation regarding the heap is slightly more involved than for the stores because of pointers. The relation of Definition 5.2 is defined out of ("smaller") relations about single variables and sub-heaps. The consequence is that verifying that the relation holds breaks down to verifying properties about sub-heaps. This leaves us with a plethora of verification goals for which it is difficult to figure out beforehand a generic approach.

For example, the following lemma states that, given an execution in some heap, we can restrict this execution to only that portion of the heap that is required to guarantee safe execution, as specified by some Separation-logic triple:

Lemma A.2 (Execution Projection w.r.t. Triple)**.** For all P, c, Q, $[P]\, c\, [Q]$, for all d, st, h, st', h' such that $\mathsf{Some}(st, h) \succ c \twoheadrightarrow \mathsf{Some}(st', h')$ and $P\ st\ h|_d$ holds, then $\mathsf{Some}(st, h|_d) \succ c \twoheadrightarrow \mathsf{Some}(st', h'|_d)$.

As a kind of converse of the previous lemma, given two executions, one against some heap and another against a sub-heap, we can figure out the result of the latter from the result of the former (this is little more that determinism since starting states are different):

Lemma A.3 (Projection Determinism)**.** For all st, h, c, st', h' such that $\mathsf{Some}(st, h) \succ c \twoheadrightarrow \mathsf{Some}(st', h')$, for all d, st'', h'' such that $\mathsf{Some}(st, h|_d) \succ c \twoheadrightarrow \mathsf{Some}(st'', h'')$, then $st'' = st'$, $h'' = h'|_d$, and $h - d = h' - d$.