An Approach to Formal Verification of Arithmetic Functions in Assembly

Reynald Affeldt[1] and Nicolas Marti[2]

[1] Research Center for Information Security,
National Institute of Advanced Industrial Science and Technology
[2] Department of Computer Science, University of Tokyo

Abstract. It is customary to write performance-critical parts of arithmetic functions in assembly: this enables finely-tuned algorithms that use specialized processor instructions. However, such optimizations make formal verification of arithmetic functions technically challenging, mainly because of many bit-level manipulations of data. In this paper, we propose an approach for formal verification of arithmetic functions in assembly. It consists in the implementation in the Coq proof assistant of (1) a Hoare logic for assembly programs augmented with loops and (2) a certified translator to ready-to-run assembly with jumps. To properly handle formal verification of bit-level manipulations of data, we propose an original encoding of machine integers. For concreteness, we use the SmartMIPS assembly language, an extension of the MIPS instruction set for smartcards, and we explain the formal verification of an optimized implementation of the Montgomery multiplication, a de facto-standard for the implementation of many cryptosystems.

1 Introduction

It is customary to write performance-critical parts of arithmetic functions in assembly: this enables finely-tuned algorithms that use specialized processor instructions. However, such optimizations make formal verification of arithmetic functions technically challenging. Indeed, the best algorithms for arithmetic functions usually rely on bit-level manipulations of data, whose properties can be tricky to figure out, especially when it comes to signed integers. But also, the usage of non-standard specialized processor instructions often calls for adequate adjustments of standard handbook algorithms, that may endanger the correctness of the algorithm itself. For these reasons, it is important to provide a concrete way to formally verify such assembly code.

In this paper, we propose an approach for formal verification of arithmetic functions in assembly. Our approach is in two steps. It consists in providing in the Coq proof assistant [3] a certified implementation of (1) a Hoare logic for assembly programs augmented with loops and (2) a certified translator to ready-to-run assembly with jumps. We favor this two-steps approach because a Hoare logic for while-programs is more familiar-looking than non-standard Hoare logics for programs with jumps (such as [12, 13]).

The Hoare logic we encode enables verification of assembly programs that manipulate machine integers and bounded memory. To cope with machine integers, we implement a library where integers are represented as lists of bits interpreted as unsigned or signed in two's complement notation. This library enables faithful modeling of the hardware circuitry. To cope with bounded memory, we encode the *separation logic* [6] variant of Hoare logic, that extends traditional Hoare logic with a native notion of mutable memory; because we use machine integers to access memory, the accessible range of addresses is natively bounded.

The certified translator ensures that assembled programs behave like the verified programs. Concretely, it injects structured programs (programs with loops) into the more general set of programs with jumps; the correctness of this translation is formally proved in Coq.

For concreteness, we use the SmartMIPS assembly language, an application-specific extension of the MIPS32 4Km processor core that extends the core instruction set with instructions to enhance cryptographic computations and improve the performance of virtual machines [5].

To validate our approach, we show how to formally verify several arithmetic functions, including in particular an optimized implementation of the Montgomery multiplication [2], a de facto-standard for the implementation of many cryptosystems. This verification requires bit-level predicates to specify the usage of carries and overflow flags, for which our Hoare logic turns out to be well-suited.

This paper is organized as follows. In Sect. 2, we explain how we encode machine-integers arithmetic in Coq. In Sect. 3, we explain how we encode separation logic for a subset of SmartMIPS. In Sect. 4, we explain the formal verification of the Montgomery multiplication. In Sect. 5, we explain how to extract ready-to-run SmartMIPS programs from our verifications. In Sect. 6, we review related work. In Sect. 7, we conclude and comment on future work.

2 Machine-Integers Arithmetic

Formal verification of arithmetic functions is usually done w.r.t. high-level mathematical specifications. However, at the level of assembly code, many arithmetic properties of instructions depend on the finiteness of registers and on the physical representation of data. For example, the (signed) integer "-1" appears to be larger than any unsigned (and therefore positive) integer; some instructions trap on integer overflow while others do not, etc. Overlooking such problems often leads to security breaches, most famously integer-overflow bugs (see [11] for illustrations). It is therefore important to provide formal means to define the semantics of instructions together with lemmas that capture their properties in terms of mathematical (i.e., unbounded) integers.

Our approach to encode machine-integers arithmetic is to closely model the hardware circuitry using lists of bits (booleans) to represent the contents of registers and recursive functions to represent the operations on registers. We choose this approach because it is easy to extend with new, specialized instruc-

tions, compared to encoding machine integers with, say, sign-magnitude integers modulo.

Example: Hardware Arithmetic Operations We model the hardware addition as a recursive function that does bitwise comparisons and carry propagation:

```
Inductive bit : Set := o : bit | i : bit.
(* addition with LSB first *)
Fixpoint add_lst' (a b:list bit) (carry:bit) : list bit :=
  match (a, b) with
      (o :: a', o :: b') => carry :: add_lst' a' b' o
    | (i :: a', i :: b') => carry :: add_lst' a' b' i
    | (_ :: a', _ :: b') => match carry with
      o => i :: add_lst' a' b' o | i => o :: add_lst' a' b' i end
    | _ => nil
  end.
(* addition with MSB first *)
Definition add_lst a b carry := rev (add_lst' (rev a) (rev b) carry).
```

Most computers distinguish between unsigned integers and signed integers in two's complement notation. The negation of a signed integer is defined using ones' complement and addition:

```
(* bit complement *)
Definition cplt b := match b with i => o | o => i end.
(* ones' complement *)
Fixpoint cplt1 (lst:list bit) : list bit :=
  match lst with nil => nil | hd :: tl => cplt hd :: cplt1 tl end.
(* two's complement *)
Definition cplt2 lst :=
  add_lst (cplt1 lst) (zero_extend_lst (length lst - 1) (i::nil)) o.
```

Using the addition, we further modeled the unsigned multiplication; using two's complement, we further modeled the signed multiplication, and so on.

Physical constraints and implementation choices make hardware arithmetic operations peculiar. Because of the finiteness of registers, they actually implement arithmetic modulo. A list of bits `(an::...::a0)` is interpreted as $(a_n \dots a_0)_2$, the encoding in base 2 of a mathematical integer; but depending on the context, this integer is unsigned, in which case its decimal value is $a_n 2^n + \ldots + a_0$, or signed in two's complement notation, in which case its decimal value is $-a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_0$. It is customary for assembly code to rely on properties of arithmetic modulo (e.g., to detect overflows) and to freely mix unsigned and signed integers (e.g., to access memory). Precise characterization of the properties of the hardware arithmetic operations w.r.t. their mathematical counterpart is therefore a must-have for formal verification of assembly code.

Example: Overflow Properties of Addition Let us note `[[lst]]u` (resp. `[[lst]]s`) the decimal value of the list of bits `lst` seen as an unsigned (resp. signed) integer. In Coq, these notations are implemented as recursive functions from lists of bits to mathematical integers. The hardware addition behaves like the mathematical addition only when non-overflow conditions are met:

```
Lemma add_lst_nat : ∀ n a b, length a = n → length b = n →
 0 ≤ [[a]]u + [[b]]u < 2^^n → [[add_lst a b o]]u = [[a]]u + [[b]]u.

Lemma add_lst_Z : forall n a b, length a = S n → length b = S n →
 -2^^n ≤ [[a]]s + [[b]]s < 2^^n → [[add_lst a b o]]s = [[a]]s + [[b]]s.
```

We proved further lemmas that capture the overflow properties of the hardware addition when overflow conditions are *not* met, the correctness of subtraction and multiplications, the relations between unsigned and signed integers, etc.

Because a processor usually manipulates machine integers of different sizes (e.g., to represent constants or contents of special registers such as accumulators), it is cumbersome to use directly lists of bits: the conditions about their lengths clutter formal verification. To simplify our development, we encapsulate all the functions modeling the hardware circuitry and the lemmas capturing their properties in a Coq module that provides an abstract type for machine integers. This abstract type is parameterized by the length of the underlying list of bits: `Parameter int : nat → Set.` This makes the relation between the lengths of the input and the output of operations explicit in the type of hardware operations.

Technically, this abstract type is implemented using dependent pairs: a machine integer of length `n` is a dependent pair whose first projection is a list of bits `lst` and whose second projection is the proof that its length is equal to `n`: `Inductive int (n:nat) : Set := mk_int : ∀ (lst:list bit), length lst = n → int n.` An excerpt of the interface of the resulting module is given below:

```
Parameter add : ∀ n, int n → int n → int n.
Notation "a ⊕ b" := (add a b).
Parameter u2Z : ∀ n, int n → Z.  (* lists of bits as unsigned *)
Parameter s2Z : ∀ n, int n → Z.  (* lists of bits as signed *)
Parameter add_u2Z : ∀ n (a b:int n), u2Z a + u2Z b < 2^^n →
  u2Z (a ⊕ b) = u2Z a + u2Z b.
Parameter add_s2Z : ∀ n (a b:int (S n)), -2^^n ≤ s2Z a + s2Z b < 2^^n →
  s2Z (a ⊕ b) = s2Z a + s2Z b.
Parameter Z2u : ∀ n, Z → int n.
Parameter Z2s : ∀ n, Z → int n.
```

`Z2u n z` (resp. `Z2s n z`) builds an unsigned (resp. a signed) machine integer of decimal value `z` and length `n` (if possible). These two constructors are used to defined constants, such as: `Definition four32 := Z2u 32 4.`

3 A Hoare Logic for SmartMIPS

In this section, we encode a Hoare logic for a subset of SmartMIPS [5]. For this purpose, the module for machine-integer arithmetic introduced in the previous section is important: it enables faithful encoding of the semantics of arithmetic instructions that trap on overflow and the semantics of memory accesses, that are restricted to finite memory. The subset of SmartMIPS we consider consists of structured programs, i.e., programs whose syntax only allows for sequences and while-loops. Hereafter, we call WhileSMIPS this subset. In Sect. 5, we will

certify a translator that injects WhileSMIPS programs into the set of SmartMIPS programs with jumps, that can be directly assembled and run. We favor this two-steps approach is that a Hoare logic for while-programs is more familiar-looking than non-standard Hoare logics for programs with jumps (such as [12, 13]).

3.1 States

The state of a SmartMIPS processor is modeled as a tuple of a store of general-purpose registers, a store of control registers, an integer multiplier, and a heap (the mutable memory):

```
Definition state := gpr.store * cp0.store * multiplier.m * heap.h.
```

The module `gpr` is a map from the type `gp_reg` of general-purpose registers to (32-bit) words, the module `cp0` is a map from the type `cp0_reg` of control registers, and `heap` is a map from natural numbers to words. We restrict ourselves to a word-addressable heap because it is all we need for arithmetic functions (see Sect. 4). The module for heap is implemented using a module for finite maps developed in previous work [16]. Let us comment more in detail on the implementation of the `multiplier` module, that makes an extensive use of our module for machine integers.

The SmartMIPS multiplier is a set of registers called ACX, HI, and LO that has been designed to enhance cryptographic computations. HI and LO are 32 bits long; ACX is only known to be at least 8 bits long. We implement the multiplier as an abstract data type `m` with three lookup functions `acx`, `hi`, and `lo` that return respectively a machine integer of length at least 8 bits and machine integers of length 32. Here follows the corresponding excerpt of the module interface:

```
Parameter acx_size : nat.
Parameter acx_size_min : 8 ≤ acx_size.
Parameter m : Set.
Parameter acx : m → int acx_size.
Parameter lo : m → int 32.
Parameter hi : m → int 32.
Parameter utoZ : m → Z. (* multiplier as an unsigned *)
```

The SmartMIPS instruction set features special instructions to take advantage of the SmartMIPS multiplier. For illustration, let us explain the encoding of the `mflhxu` instruction, that is often used in arithmetic functions: it performs a division of the multiplier by $\beta = 2^{32}$, whose remainder is put in a general-purpose register and whose quotient is left in the multiplier. The corresponding hardware circuitry is essentially a shift: it puts the contents of LO into some general-purpose register, puts the contents of HI into LO, and zeroes ACX. Here is how we model this operation:

```
Definition mflhxu_op m := let (acx', hi') := (acx m, hi m) in
  (Z2u acx_size 0, (zero_extend 24 acx', hi')).
```

What is important for verification is the properties of `mflhxu` w.r.t. the decimal value of the multiplier. Such properties can be derived as lemmas from the definition of `mflhxu_op`. For example, the decimal values of the multiplier before

and after `mflhxu` are related as follows (`Zbeta n` stands for $\beta^n = 2^{32n}$):

```
Lemma mflhxu_utoZ : ∀ m, utoZ m = utoZ (mflhxu_op m) * Zbeta 1 + u2Z (lo m).
```

3.2 Axiomatic Semantics

The syntax of WhileSMIPS programs is encoded as the inductive type `cmd`:

```
Definition immediate := int 16.
Inductive cmd : Set :=
  | add : gp_reg → gp_reg → gp_reg → cmd
  | addi : gp_reg → gp_reg → immediate → cmd
  | addiu : gp_reg → gp_reg → immediate → cmd
  | addu : gp_reg → gp_reg → gp_reg → cmd
  | lw : gp_reg → immediate → gp_reg → cmd
  | lwxs : gp_reg → gp_reg → gp_reg → cmd
  | maddu : gp_reg → gp_reg → cmd
  | mflhxu : gp_reg → cmd
  | sw : gp_reg → immediate → gp_reg → cmd
  | seq : cmd → cmd → cmd
  | ifte_beq : gp_reg → gp_reg → cmd → cmd → cmd
  | while_bne : gp_reg → gp_reg → cmd → cmd
  ...
```

Except for control-flow commands (`seq`, `ifte_beq`, `while_bne`, etc.), the type constructors have the same names as their SmartMIPS counterparts. As usual with MIPS, instructions with suffix "`u`" do not trap on overflow and instructions with prefix "`m`" use the multiplier. In the excerpt above, SmartMIPS-specific instructions include `lwxs`, that loads words using scaled indexed addressing, and `maddu` and `mflhxu`, that use the ACX register.

The assertion language is an instance of separation logic [6], an extension of Hoare logic with a native notion of heap. We choose separation logic because it is a general solution to represent mutable data structures such as arrays, that are used to represent multi-precision integers in arithmetic functions.

Assertions are encoded as truth-functions from states to `Prop`, the type of predicates in Coq (this technique of encoding is called *shallow embedding*):

```
Definition assert := gpr.store → cp0.store → multiplier.m → heap.h → Prop.
```

Using this type, one can easily encode any first-order predicate. For example, the predicate that is true when variables `x` and `y` have the same contents is encoded as follows (`lookup` is a function provided by the interface of stores):

```
Definition x_EQ_y (x y : gp_reg) : assert :=
  fun s _ _ _ => gpr.lookup x s = gpr.lookup y s.
```

To encode separating connectives, we use a module for heaps. In the following, we omit the definitions of heap-related functions: their names and notations are self-explanatory and details can be found in [16]. First, we introduce a language for expressions used in separating connectives:

```
Inductive expr : Set :=
 var_e : gp_reg → expr | int_e : int 32 → expr | add_e : expr → expr → expr | ...
```

In this language, variables are registers and constants are (32-bit) words. Given an expression `e` and a store `s`, the function `eval` returns the value of the expression `e` in store `s`.

The *mapsto* connective `e1 ↦ e2` holds in a state with a store `s` and a singleton heap with address `eval e1 s` and contents `eval e2 s`:

```
Definition mapsto (e e':expr) : assert := fun s _ _ h =>
  ∃ p, u2Z (eval e s) = 4 * p ∧ h = heap.singleton p (eval e' s).
Notation "e1 ↦ e2" := (mapsto e1 e2).
```

The *separating conjunction* `P⋆Q` holds in a state whose heap can be divided into two disjoint heaps such that `P` and `Q` hold:

```
Definition sep_con (P Q:assert) : assert := fun s s' m h =>
  ∃ h1, ∃ h2, h1 ⊥ h2 ∧ h = h1 ∪ h2 ∧ P s s' m h1 ∧ Q s s' m h2.
Notation "P ⋆ Q" := (sep_con P Q).
```

In practice, the separating conjunction provides a concise way to express that two data structures reside in disjoint parts of the heap.

Using the separating conjunction, the mapsto connective can be generalized to arrays of words: `(e ⇻ a::b::...)` holds in a state whose heap contains a list of contiguous words `a`, `b`, ... starting at address `eval e s`:

```
Fixpoint mapstos (e:expr) (lst:list (int 32)) : assert :=
  match lst with
    | nil => empty_heap
    | hd::tl => (e ↦ int_e hd) ⋆ (mapstos (add_e e (int_e four32)) tl)
  end.
Notation "e ⇻ lst" := (mapstos e lst).
```

Hoare triples are encoded as an inductive relation `{{P}} c {{Q}}` (notation for `semax P c Q`) between commands and pre/post-conditions encoded as assertions. For illustration, here follows the implementation of the Hoare triples for the commands `add` and `lw`:

```
Inductive semax : assert → cmd → assert → Prop :=
| semax_add: ∀ Q rd rs rt,
  {{ update_store_add rd rs rt Q }} add rd rs rt {{ Q }}
| semax_lw : ∀ P rt offset base,
  {{ lookup_heap_lw rt offset base P }} lw rt offset base {{ P }}
| ...
```

In the preconditions, `update_store_add` and `lookup_heap_lw` are predicate transformers. The effect of executing `add rd rs rt` is to update the contents of the register `rd` with the result of the operation `(vrs ⊕ vrt)` (where `vrs` and `vrt` are the contents of the registers `rs` and `rt`), provided the addition in two's complement does not overflow:

```
Definition update_store_add rd rs rt P : assert := fun s s' m h =>
  - 2^^31 ≤  s2Z (gpr.lookup rs s) + s2Z (gpr.lookup rt s) < 2^^31 →
  P (gpr.update rd (gpr.lookup rs s ⊕ gpr.lookup rt s) s) s' m h.
```

The function `lookup_heap_lw` checks that the access is word-aligned and the cell-contents specified:

```
Definition lookup_heap_lw rt offset base P : assert := fun s s' m h =>
  ∃ p, u2Z (gpr.lookup base s ⊕ sign_extend offset) = 4 * p ∧
    ∃ z, heap.lookup p h = Some z ∧ P (gpr.update rt z s) s' m h.
```

4 Application to Multi-precision Arithmetic

Using our encoding of Hoare logic for SmartMIPS, we have written, specified, and verified several SmartMIPS implementations of multi-precision arithmetic functions in Coq. In this section, we give an overview of our experiments with a detailed account of the formal verification of the Montgomery multiplication.

4.1 Specification of Multi-precision Arithmetic Functions

Multi-precision Integers We encode multi-precision integers as lists of machine integers stored in memory (using the $\Mapsto$ connective defined in Sect. 3.2). In the following, `Nth i A` represents the `i`th element of the list `A` of machine integers. The interpretation of a multi-precision integer as a mathematical integer is provided by a recursive function: `Sum k A` represents the decimal value of the `k` first words of the list `A` of machine integers (least significant word first) interpreted as unsigned. The fact that the length of multi-precision integers is explicit is important to write loop invariants, that often talk about "partial" multi-precision integers, to represent the decimal values of partial products for example.

Arithmetic Relations Thanks to shallow encoding and our lemmas that relate machine integers to their decimal values, we can reuse predicates and functions from the standard Coq library. In the following, `a == b [[n]]` is the Coq version of $a \equiv b[n]$.

4.2 Formal Verification of the Montgomery Multiplication

The Montgomery multiplication [2] is a modular multiplication. Given three `k`-word integers `X`, `Y`, and `M` such that

```
Sum k X < Sum k M ∧ Sum k Y < Sum k M                    (1)
```

the Montgomery multiplication computes a `k+1`-word integer `Z` such that

```
Zbeta k * Sum (k+1) Z == Sum k X * Sum k Y [[ Sum k M ]]  (2)
```

The advantage of the Montgomery multiplication is that it does not require a multi-precision division, but uses less-expensive shifts instead. The price to pay is the parasite factor `Zbeta k` whose elimination requires a second pass.

The implementation of the Montgomery multiplication we deal with (Fig. 1) is the so-called "Finely Integrated Operand Scanning" (FIOS) variant [4]. Intuitively, it resembles the classical algorithm for multi-precision multiplication: it has two nested loops, the inner-loop incrementally computes partial products (modulo) that are successively added by the outer-loop. These partial products

modulo are computed in such a way that the least significant word is always zero, thus guaranteeing that the final result will fit in `k+1` words of storage. For this to be possible, the Montgomery multiplication requires the pre-computation of the modular inverse `alpha` of the least significant word of the modulus:

```
u2Z (Nth 0 M) * u2Z alpha == -1 [[ Zbeta 1 ]]                    (3)
```

```
Definition montgomery
  k alpha x y z m j i
  X Y M Z one zero quot C t s :=

  addiu one zero one16;
  addiu C zero zero16;
  addiu i zero zero16;
  while_bne i k (
    lwxs X i x;
    lw Y zero16 y;
    lw Z zero16 z;
    multu X Y;
    lw M zero16 m;
    maddu Z one;
    mflo t;
    mfhi s;
    multu t alpha;
    addiu j zero one16;
    mflo quot;
    mthi s;
    mtlo t;
    maddu quot M;
    mflhxu Z;
    addiu t z zero16;
    while_bne j k (
      lwxs Y j y;
      lwxs Z j z;
      maddu X Y;
      lwxs M j m;
      maddu Z one;
      maddu quot M;
      addiu j j one16;
      mflhxu Z;
      addiu t t four16;
      sw Z mfour16 t
    );
    maddu C one;
    mflhxu Z;
    addiu i i one16;
    sw Z zero16 t;
    mflhxu C
  ).
```

Fig. 1. The Montgomery Multiplication in WhileSMIPS

The SmartMIPS architecture is well-suited to the implementation of the FIOS variant of the Montgomery multiplication because the addition performed in the inner-loop (that adds two products of 32-bits integers) fits in the integer multiplier (of size greater than 72 bits).

The formal Hoare triple that specifies the Montgomery multiplication is displayed in Fig. 2. The hypotheses `HX`, `HY`, and `Halpha` correspond to the input-conditions (1) and (3), as explained above. Other hypotheses are technical: they prevent overflows and enforce alignments (similar-looking conditions are abbreviated by "..."). The output-condition (2) appears in the post-condition; observe that the `k+1`$^{\text{th}}$ word of storage is provided by the register `C`. The existence in memory of input-words and output-words is specified by the separation logic formula.

Formal verification of the Montgomery multiplication is done by forward reasoning. Let us comment on two key aspects of this verification.

Let `A` be the value of the multiplier before entering the inner-loop and `A%n` the remainder of the division of `A` by `2^^n`. The Montgomery multiplication computes `quot = ((A%32) ⊙ alpha)%32` and adds `quot ⊙ M0` to the multiplier (⊙ is the unsigned multiplication, definition not displayed in this paper). The lemma be-

```
Lemma montgomery_specif : ∀ nk (Hk: 0 < nk) nx ny nm nz
 (Hnx: 4 * nx + 4 * nk < Zbeta 1) (Hny: ...) (Hnm: ...) (Hnz: ...)
 X Y M (Hx: length X = nk) (Hy: ...) (Hm: ...)
 (HX: Sum nk X < Sum nk M) (HY: Sum nk Y < Sum nk M)
 vx vy vm vz (Hvx: u2Z vx = 4 * nx) (Hvy: ...) (Hvm: ...) (Hvz: ...)
 valpha (Halpha: u2Z (Nth 0 M) * u2Z valpha == -1 [[ Zbeta 1 ]]),

 {{ fun s s' m_ h => ∃ Z,
    length Z = nk ∧ list_of_zeros Z ∧
    gpr.lookup x s = vx ∧ gpr.lookup y s = vy ∧
    gpr.lookup z s = vz ∧ gpr.lookup m s = vm ∧
    u2Z (gpr.lookup k s) = nk ∧ gpr.lookup alpha s = valpha ∧
    (var_e x ⤇ X) ⋆ (var_e y ⤇ Y) ⋆ (var_e z ⤇ Z) ⋆ (var_e m ⤇ M) s s' m_ h ∧
    multiplier.is_null m_ }}

 montgomery k alpha x y z m j i X_ Y_ M_ Z_ one gpr_zero quot C t s

 {{ fun s s' m_ h => ∃ Z, length Z = nk ∧
    (var_e x ⤇ X) ⋆ (var_e y ⤇ Y) ⋆ (var_e z ⤇ Z) ⋆ (var_e m ⤇ M) s s' m_ h ∧
    Zbeta nk * Sum (nk+1) (Z ++ gpr.lookup C s :: nil) ==
      Sum nk X * Sum nk Y [[ Sum nk M ]] ∧
    Sum (nk+1) (Z ++ gpr.lookup C s :: nil) < 2 * Sum nk M }}.
```

Fig. 2. Formal Specification of the Montgomery Multiplication

low captures the fact that the resulting multiplier is a multiple of β, and thus the least significant word of the partial product is always zero:

```
Import multiplier.
Lemma montgomery_lemma : ∀ alpha M0 (A:int 64) m,
 u2Z M0 * u2Z alpha == -1 [[Zbeta 1]] → utoZ m < Zbeta 2 → utoZ m = u2Z A →
  lo (maddu_op (((A % 32 ⊙ alpha) % 32) ⊙ M0) m) = zero32.
```

As usual, the heart of the verification is to produce the right invariant for the inner-loop. In the case of the FIOS variant of the Montgomery multiplication, the difficulty comes from the fact that the zeroed word of storage is used to "shift-in" the second least-significant word (LSW) of the partial product. More precisely, at the j^{th} iteration of the inner-loop, the algorithm uses the j^{th} LSW of the current partial product to compute the $j-1^{\text{th}}$ LSW of the new partial product. To write this invariant, we use a function for "multi-precision integers with a hole". Multi-precision integers with a hole are like multi-precision integers except that there is one word that we ignore in computing the represented value:

```
Definition Sum_hole l len hole (lst:list (int l)) :=
    Sum (len - 1) (del_nth hole lst).
```

Using this function, the relation between the multiplier and the multi-precision integers in memory is written:

```
Zbeta (ni + 1) * Sum_hole (nk + 1) (nj - 1) (Z ++ gpr.lookup C s :: nil) +
  multiplier.utoZ m * Zbeta (ni + nj) ==
Sum ni X * Sum nk Y + Sum nj Y * u2Z (nth ni X zero32) * Zbeta ni +
  Sum nj M * u2Z (gpr.lookup quot s) * Zbeta ni  [[ Sum nk M ]]
```

where `ni`, `nj`, and `nk` are the contents of the `i`, `j`, and `k` registers, and `s` is the current store of general-purpose registers.

4.3 Experimental Results

Besides the Montgomery multiplication, we have verified SmartMIPS implementations of several classical algorithms for multi-precision arithmetic. The code is available online [17]; the table below summarizes the sizes of programs and proof scripts. For proof scripts, we distinguish between the number of lines used to write assertions (all pre/post-conditions, including intermediate forward-reasoning steps) and the number of lines used for proof construction (calls to Coq tactics, including custom tactics, and application of lemmas).

Multi-precision function	Number of asm instructions	Size of proof scripts (lines)		
		total	assertions (ratio)	individual steps (average)
addition	11	835	203 (24%)	632 (57)
subtraction	22	1473	340 (23%)	1133 (52)
multiplication	20	1634	413 (25%)	1221 (61)
Montgomery	37	3881	955 (25%)	2926 (79)

Although assertions occupy around 24% of the proof scripts, this is not a nuisance because they only change a little from one reasoning step to the other, and it anyway helps to understand the verification. Appropriate tactics for forward reasoning could get rid of this overhead.

In average, each atomic Hoare triple is proved with 62 Coq commands. Some parts are inherently difficult because of low-level manipulations of multi-precision integers, that require many syntactic manipulations of goals and hypotheses and are difficult to automate satisfactorily. Yet, many parts of proof scripts are repetitive (`trivial` goals, obvious rewriting, etc.) and we already have a good deal of small-scale custom tactics. As a mid-term goal, we think it should be possible and desirable to use no more than 20 Coq commands per reasoning step.

5 Program Extraction

In this section, we explain how to safely extract ready-to-run SmartMIPS programs from our Coq verifications. In Sect. 3.2, we have defined an axiomatic semantics for WhileSMIPS, a subset of structured SmartMIPS programs. Though it is sufficient to specify and verify arithmetic functions and many other programs, we cannot directly assemble and run verified programs: we first need to translate them into the set of SmartMIPS programs with jumps, and ensure that this translation is correct. For this purpose, we equip both WhileSMIPS and SmartMIPS programs with an operational semantics.

5.1 SmartMIPS Operational Semantics

The MIPS documentation [5] gives the semantics of SmartMIPS in terms of a virtual machine that represents the processor. It has an explicit program counter and its execution is described by a small-step semantics. We encode this small-step semantics in Coq. For the sake of simplicity, we restrict ourselves to word-aligned memory accesses, we ignore exceptions and pipelining optimizations[3].

Syntax We split SmartMIPS instructions into the set of instructions that modify the state and just increment the program counter (type `cmd0`), and the set of instructions that only modify the control-flow (type `branch`). We do not display in this paper the syntax of state-modifying instructions because it is similar to `cmd` (Sect. 3.2), without the control-flow commands (we just suffix type constructors with "`0`" to distinguish them). The syntax of branching instructions is encoded as the inductive type `branch`:

```
Definition label := nat.
Inductive branch : Set :=
 jmp : label → branch | beq : gp_reg → gp_reg → label → branch | ...
```

`jmp` inconditionnally jumps to a given label, and `beq`, etc. conditionally jump to some label. A SmartMIPS instruction is either a `cmd0` or a `branch` instruction (the dummy instruction `no_insn` below is just a technical convenience) and a SmartMIPS program is a list of instructions (without explicit structure, the positions of instructions in this list serving as labels):

```
Inductive insn : Set :=
 cmd_insn : cmd0 → insn | branch_insn : branch → insn | no_insn : insn.
Definition prog := list insn.
```

Operational Semantics The operational semantics of `cmd0` instructions is encoded as an inductive type `st -- c --> st'` that represents the execution of the instruction `c` from state `st` to state `st'`. For illustration, here follows the semantics of `add`:

```
Inductive exec0 : option state → cmd0 → option state → Prop :=
| exec0_add : ∀ s s' vrt vrs m h rd rs rt,
  gpr.lookup rs s = vrs → gpr.lookup rt s = vrt →
  -2^^31 ≤ s2Z vrt + s2Z vrs < 2^^31 →
  Some (s,s',m,h) -- add0 rd rs rt --> Some (gpr.update rd (vrs ⊕ vrt) s,s',m,h)
| ...
```

The operational semantics of `branch` instructions is encoded as an inductive type `n |> (pc,st) >> c >> (pc',st')` that represents the execution of the branch `c` from program counter `pc` and state `st` to program counter `pc'` and state `st'` (under the constraint that the destination label is smaller than `n`). Note that `st` and `st'` can be different when the jump destination is not valid (leading to an error state). For illustration, here follows the semantics of `jmp`:

[3] In MIPS, the first instruction following a conditional branching is unconditionally executed. In other words, the first instruction that is syntactically after a conditional branching is executed before. In this paper, we ignore this issue.

```
Inductive exec_branch (max:label)
  : branch → label * option state → label * option state → Prop :=
| exec_jmp : ∀ pc st j, max ≥ j ->
  max |> (pc, Some st) >> jmp j >> (j, Some st)
| ...
```

The operational semantics of SmartMIPS programs is encoded as an inductive type `prg ||- (pc,st) --> (pc',st')` that represents the execution of program `prg` from program counter `pc` and state `st` to program counter `pc'` and state `st'`:

```
Inductive exec_asm (prg:prog)
  : label * option state → label * option state → Prop :=
| exec_asm_cmd0 : ∀ pc c st st', Nth pc prg = cmd_insn c →
  Some st -- c --> Some st' →
  prg ||-- (pc, Some st) --> (pc+1, Some st')
| exec_asm_branch : ∀ pc j st pc' st', Nth pc prg = branch_insn j →
  length prg |> (pc, Some st) >> j >> (pc', st') →
  prg ||-- (pc, Some st) --> (pc', st')
| ...
```

For the composition of instructions to be possible, this inductive type also has type constructors (not displayed here for lack of space) that express the reflexivity and transitivity of the operational semantics.

5.2 Translation from WhileSMIPS to SmartMIPS

The role of the translator is simply to translate the control-flow commands of WhileSMIPS into SmartMIPS:

```
Fixpoint translate (lbl:label) (c:cmd) : prog :=
  match c with
    | while_bne r1 r2 c => let prg := translate (lbl + 1) c in
        branch_insn (beq r1 r2 (lbl + length prg + 2)) :: prg ++
        branch_insn (jmp lbl) :: nil
    ...
  end.
```

The correctness proof of this translator consists in showing that, for any state, the final state of the execution of a WhileSMIPS progam and the final state of the execution of its translated SmartMIPS version are the same. To do this proof, we still need to equip WhileSMIPS programs with an operational semantics. The latter is encoded as a inductive type `exec st c st'` that represents the execution of the command `c` from state `st` to state `st'` (big-step operational semantics, similar to [16]). To ensure that this operational semantics agrees with the axiomatic semantics of Sect 3.2, we show that the latter is sound and complete w.r.t. the former (formal proofs are similar to [10]). Finally, using the big-step semantics of WhileSMIPS and the small-step semantics of SmartMIPS programs, the correctness of the translator is proved by induction:

```
Lemma translate_correct: ∀ c st p c' st',
  translate (length p) c = c' →
  exec (Some st) c (Some st') →
  p ++ c' ||-- (length p, Some st) --> (length (p ++ c'), Some st').
```

6 Related Work

Much work about formal encoding of assembly languages in proof assistants has been done with application to proof-carrying code (PCC) in mind [7–9]. Although the encoded semantics often allows for programs with arbitrary jumps, details such as machine integers are usually not treated. This makes it difficult to reuse existing implementations of PCC frameworks to formally verify arithmetic functions, whose algorithms require bit-level specifications.

There exist other encodings of machine integers in Coq. Leroy has encoded such a library for integers modulo 2^{32} as part of the development of a certified compiler [14]. His encoding uses the relative integers of Coq (the `Z` type) instead of lists ot bits. We found it difficult to reuse directly his implementation because the length of integers (32) is hard-wired and we needed a similar library for several lengths. Chlipala has encoded a library similar to ours but based on dependent vectors [15]. We think that our implementation based on an abstract type is more flexible than dependent vectors because it separates the issues of formal proofs and dependent types.

In this paper, we use a combination of a Hoare logic for structured SmartMIPS with a certified translator to SmartMIPS programs with jumps. Another approach would have been to encode a (more intricate) Hoare logic for low-level programs with jumps (such as [12, 13]) and to specialize it to structured programs to carry out verifications. We chose the former approach because our primary concern was the formal verification of concrete examples of arithmetic functions, whose implementations turn out to fit well in structured SmartMIPS.

7 Conclusion

In this paper, we proposed an approach to formal verification of arithmetic functions in assembly based on the combined use of a certified implementation of a Hoare logic for assembly programs with loops and a certified translator to assembly programs with jumps. This approach enables formal verification of ready-to-run assembly programs with a familiar-looking Hoare logic. At the heart of our implementation is a module for machine integers that makes it possible to prove formally the lemmas, such as overflow conditions, needed for verification of assembly programs. Using this approach, we have formally verified several arithmetic functions written in SmartMIPS assembly, including an optimized implementation of the Montgomery multiplication, a de facto-standard for the implementation of many cryptosystems.

Future Work In order to verify more arithmetic functions, we are extending our library with a semantics for function calls and returns, and with predicates to deal with signed multi-precision integers. We also plan to encode a semantics for exceptions to enable verification of embedded systems.

Acknowledgments This work is partially supported by the Grant in Aid of Special Coordination Funds for Promoting Science and Technology, Ministry of Education, Culture, Sports, Science and Technology, Japan. The authors are grateful

to Pascal Paillier at Gemalto who provided the code of the Montgomery multiplication with detailed explanations.

References

1. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(10):576–585, 1969.
2. P. L. Montgomery. Modular multiplication without trial division. Mathematics of Computation, 44(170):519–521, 1985.
3. Various contributors. The Coq Proof assistant. `http://coq.inria.fr`.
4. Cetin Kaya Koc, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro 16(3):26–33, 1996.
5. MIPS Technologies. MIPS32 4KS Processor Core Family Software User's Manual MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353.
6. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, p. 55–74.
7. Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A Syntactic Approach to Foundational Proof-Carrying Code. In *7th IEEE Symposium on Logic In Computer Science (LICS 2002)*, p. 89–100.
8. Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. In *12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, p. 363–379. Springer.
9. Nadeem Abdul Hamid and Zhong Shao. Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code. In *17th Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, p. 118–135. Springer.
10. Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In *13th Conference on Computer Science Logic (CSL 2004)*, volume 3210 of *LNCS*, p. 250–264. Springer.
11. Domagoj Babić and Madanlal Musuvathi. Modular Arithmetic Decision Procedure. Microsoft Research Technical Report. MSR-TR-2005-114.
12. Ando Saabas and Tarmo Uustalu. A Compositional Natural Semantics and Hoare Logic for Low-Level Languages. Electronic Notes in Theoretical Computer Science, 156:151–168, 2006.
13. Gang Tan and Andrew W. Appel. A Compositional Logic for Control Flow. In *7th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, volume 3855 of *LNCS*, p. 80–94. Springer.
14. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, p. 42–65.
15. Adam J. Chlipala. Modular development of certified program verifiers with a proof assistant. In *11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, p 160–171.
16. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, p. 400–419. Springer.
17. Reynald Affeldt and Nicolas Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly—Proof Scripts. `http://staff.aist.go.jp/reynald.affeldt/seplog/asian2006`.