Formal Verification of Arithmetic Functions in SmartMIPS Assembly*

Reynald Affeldt† Nicolas Marti‡

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

‡Department of Computer Science,
University of Tokyo

Abstract

In embedded systems, the recent trend is to manufacture processors with application-specific extensions. This makes it often necessary to write assembly programs to take advantage of the added hardware facilities. In such situations, formal verification is technically difficult because the programs in question manipulate data in a bitwise fashion, using non-standard specialized instructions, and under strict constraints for memory usage. In this paper, we propose an encoding of Hoare logic in the Coq proof assistant for formal verification of assembly programs that manipulate machine integers and bounded memory. Using this encoding, we formally verify arithmetic functions used in cryptography and written in SmartMIPS, an extension of the MIPS instruction set for smartcards.

1 Introduction

In embedded systems, the recent trend is to manufacture processors with application-specific extensions. For example, SmartMIPS is an application-specific extension of the MIPS32 4Km processor core: it extends the core instruction set with instructions to enhance cryptographic calculations and improve the performance of virtual machines [1].

In order to take advantage of the hardware facilities added by application-specific extensions, it is often necessary to write assembly programs. In such situations, formal verification is technically difficult because the programs in question manipulate data in a bitwise fashion, using non-standard specialized instructions, and under strict constraints for memory usage. In particular, such verifications require much effort to check overflow conditions and the validity of memory accesses.

In this paper, we propose an encoding of Hoare logic [2] in the Coq proof assistant [3] for formal verification of assembly programs that manipulate machine integers and bounded memory. First, we develop a library for machine integers, with lemmas for overflow conditions. Second, we use this library to encode the *separation logic* [6] variant of Hoare logic, that extends traditional Hoare logic with a native notion of mutable memory; because of our use of machine integers to access memory, the accessible range of addresses is natively bounded.

Using this encoding, we formally verify arithmetic functions used in cryptography and written in SmartMIPS. Arithmetic functions typically deal with overflow conditions and bit-level predicates to specify the usage of carries. In particular, we verify an optimized implementation of the Montgomery multiplication, that is used in most public-key cryptosystems.

This paper is organized as follows. In Sect. 2, we explain how we encode machine-integer arithmetic in Coq. In Sect. 3, we explain how we encode separation logic for SmartMIPS in Coq. In Sect. 4, we explain how we apply our encoding to the verification of SmartMIPS functions for multi-precision arithmetic. In Sect. 5, we comment on technical aspects of our experiments. In Sect. 6, we review related work. In Sect. 7, we conclude and comment on future work.

2 Machine Integers

In this section, we explain how we encode machine-integer arithmetic in Coq. This encoding is important for formal verification of assembly programs because many properties of instructions depend on the physical representation of data in computers, and this has often counter-intuitive consequences. For example, in the C programming language, the (signed) integer "-1" happens to be larger than any unsigned integer. Another example is the remainder of a signed integer: the sign of the result depends on the value

*This work has been presented at the 23rd Workshop of the Japan Society for Software Science and Technology: `http://www.ipl.t.u-tokyo.ac.jp/jssst2006/`.

of the input. Overlooking such problems often leads to bugs.

In order to encode machine integers faithfully, our approach is (1) to provide an encoding of the computer circuitry in terms of lists of bits, (2) to prove the properties of the computer circuitry, in particular w.r.t. the interpretation of lists of bits as decimal integers, and (3) to encapsulate these results in an abstract type for machine integers. There are two advantages in adopting this encoding approach: there is a close correspondance with the hardware, thus enabling the safe formalization of most properties useful for verification, and it is easy to extend the abstract type with new operations defined as recursive functions over lists of bits.

2.1 Hardware Arithmetic

The computer circuitry can be modeled by recursive functions over lists of bits, and the properties of these functions can be proved by induction. In the following, we assume that bits are represented by the inductive type: Inductive bit : Set := o : bit | i : bit.

Arithmetic Operations For illustration, let us consider the addition. It can be encoded as a recursive function that does bitwise comparisons and carry propagation:

```
(* least significant bit first *)
Fixpoint add_lst' (a b:list bit) (carry:bit)
  {struct a} : 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.

(* most significant bit first *)
Definition add_lst a b carry :=
  rev (add_lst' (rev a) (rev b) carry).
```

The hardware properties of the addition such as commutativity can be proved by induction:

```
Lemma add_lst_com : ∀ a b carry,
  add_lst a b carry = add_lst b a carry.
```

Other arithmetic operations and their properties are encoded similarly.

Signed Integers MIPS distinguishes between unsigned integers and signed integers in two's complement notation. The negation of a signed integer is defined using ones' complement and addition:

```
Definition cplt b :=
 match b with i => o | o => i end.

Fixpoint cplt1 (lst:list bit) {struct lst} :
  list bit := match lst with
     nil => nil
   | hd :: tl => cplt hd :: cplt1 tl
 end.

Definition cplt2 lst := add_lst (cplt1 lst)
 (zero_extend_lst (length lst - 1) (i::nil)) o.
```

The properties of complement notations are essential to prove the correctness of arithmetic operations. In practice, the most important property turns out to be the relation between the two's complement of a list and its tail:

```
Lemma cplt2_prop : ∀tl, ~(∃k, tl = zeros k) →
  ∀hd, ~(∃k, hd::tl = i::zeros k) →
    cplt2 (hd::tl) = cplt hd :: cplt2 tl.
```

The conditions state that the list (hd::tl) is neither zero, nor the "weird number" (i followed by os).

2.2 Programmer's View

In general, the programmer sees a list of bits an::...::a0 as the encoding of the integer $(a_n \ldots a_0)_2$ (in base 2). Depending on the context, this integer is unsigned, in which case its decimal value is $a_n 2^n + \cdots + a_0$, or signed in two's complement notation, in which case its decimal value is $-a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_0$. Let us note in Coq [[lst]]u (resp. [[lst]]s) the decimal value of the list of bits lst seen as an unsigned (resp. signed) integer.

Integers Modulo Because of the finiteness of registers, list of bits actually implement arithmetic modulo. As a consequence, the hardware addition behaves as the decimal addition only when non-overflow conditions are met, otherwise the result is only equal modulo (2^^n stands for the power function 2^n):

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

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

There are similar properties for other arithmetic operations, and signed integers.

Relation Between Signed and Unsigned Integers Positive signed integers coincide with unsigned integers (equivalently, unsigned integers smaller than half of the modulus coincide with signed integers):

```
Lemma slst2Z_ulst2Z_pos : ∀ n lst,
  length lst = n → 0 ≤ [[lst]]s →
  [[lst]]s = [[lst]]u.
```

Negative signed integers are equal to unsigned integers modulo:

```
Lemma slst2Z_ulst2Z_neg : ∀ n lst,
  length lst = n → [[lst]]s < 0 →
  [[lst]]s = [[lst]]u - 2^^n.
```

2.3 Abstract Type

We have encapsulated all the functions and properties about lists of bits in a module that provides an abstract type for machine integers. This abstract type appears as a type constructor where the length of the underlying list of bits is explicit:

```
Parameter int : nat → Set.
```

Technically, it is implemented using dependent pairs. A machine integer of size `n` is a pair of a list of bits with the proof that its length is equal to `n`:

```
Inductive int (n:nat) : Set := mk_int :
  ∀ (lst:list bit), length lst = n → int n.
```

Here follows an excerpt of the module for machine integers:

```
Parameter add : ∀ n, int n → int n → int n.
  Notation "a ’(+)’ b" := (add a b).
Parameter u2Z : ∀ n, int n → Z.
Parameter add_u2Z : forall n (a b:int n),
  u2Z a + u2Z b < 2^^n →
  u2Z (a (+) b) = u2Z a + u2Z b.
Parameter add_u2Z_overflow : ∀ n (a b:int n),
  2^^n ≤ u2Z a + u2Z b →
  u2Z (a (+) b) = u2Z a + u2Z b - 2^^n.
```

Equipped with this module, one can derive properties needed for formal verification of assembly programs. Let us illustrate this point with an example. Arithmetic operations may use a mix of unsigned and signed integers. Depending on the specification, it may be important to check for overflows. The lemma below captures for example the conditions under which one can safely add an unsigned and a signed integer:

```
Lemma add_u2Z_s2Z : ∀ n (a b:int n),
  0 ≤ u2Z a + s2Z b < 2^^n →
  u2Z (a (+) b) = u2Z a + s2Z b.
```

Concretely, this lemma says that it is safe to add “$2^{32}-8$” with “-4” to find “$2^{32}-12$”, despite the fact that both values are encoded as $(1\cdots1000)_2$ and $(1\cdots100)_2$, whose addition would overflow if both considered unsigned.

3 Hoare Logic for SmartMIPS

One difficulty of encoding in Coq a Hoare logic for assembly is the faithful representation of bitwise instructions and low-level data such as machine integers. In this section, we explain how we use machine-integer arithmetic defined in the previous section to encode separation logic, a variant of Hoare logic, for a subset of the SmartMIPS instruction set.

3.1 States

The state of a SmartMIPS processor is defined 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 finite map from the type `gp_reg` of general-purpose registers to (32-bit) words, the module `cp0` is a finite map from the type `cp0_reg` of control registers, and `heap` is a map from natural numbers to words. The restriction to a word-addressable heap is just a convenience for the subset of SmartMIPS we target. These modules are implemented using a module for finite maps developed in [8]. Let us comment in detail on the implementation of the `multiplier` module.

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 resp. a machine integer of length at least 8 bits and machine integers of length 32. At any time, the contents of the multiplier can be interpreted as an unsigned integer by the function `utoZ`. Here follows the corresponding excerpt of the module interface:

```
Module Type MULTIPLIER.
  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.
```

The SmartMIPS instruction set features special instructions to take advantage of the SmartMIPS multiplier. For illustration, let us explain the encoding of a typical instruction. The `mflhxu` instruction is extensively 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 implement the corresponding operation:

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

(`Z2u` builds a machine integer from a relative integer.) 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 implementation 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 The Programming Language

We have encoded the syntax and semantics of a subset of the SmartMIPS instruction set. In short, this subset is a restriction to structured programs, which are sufficient to model directly most arithmetic functions. This section is not detailed because most encoding techniques are standard (see [8] for example).

The syntax of instructions is encoded as an inductive type called `cmd` whose constructors encode instructions. For example, the excerpt below shows the encoding of the syntax of `add` (the addition that traps on overflow):

```
Inductive cmd : Set :=
 add : gp_reg → gp_reg → gp_reg → cmd | ...
```

Similarly, we have defined other instructions for arithmetic (`addu` that does not trap on overflow, `addi` that adds a 16-bit constant with a general-purpose register, `and` for bitwise conjunction, etc.), instructions for memory accesses (`lw` and `sw` for loading and storing words, `lwxs` for loads using scaled indexed addressing), instructions specific to the multiplier (`maddu`, `msubu`, `multu`, `mflhxu`, etc.), and instructions for tests (such as `sltu`, which is important to simulate carry flags). The only control-flow operations are while-loops, if-then-else branchings, and sequences. There is a discrepancy w.r.t. MIPS assembly: in MIPS, the first instruction that is syntactically after a conditional branching is actually executed before; in our encoding, the syntactic order reflects the execution order.

3.3 Axiomatic Semantics

The axiomatic semantics (the triples) is a relation between pre/post-conditions and instructions.

Pre/post-conditions are written using *assertions*, defined as truth-functions from states to `Prop`, the type of predicates in Coq (this technique of encoding is known as *shallow endoding*):

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

For example, the assertion that is always true is encoded as follows: `Definition TT : assert := fun s s' m h => True`. One can similarly encode any first-order predicate.

The axiomatic semantics in itself is encoded as an inductive type called `semax`. For example, let us consider the encoding of the triple for the instruction `add`:

```
Inductive semax : assert→cmd→assert→Prop :=
 semax_add: ∀ Q rs rt rd,
  semax (upd_store_add rd rs rt Q)
    (add rd rs rt) Q | ...
```

In the precondition, `upd_store_add` is a predicate transformer that does a substitution and enforces the overflow check:

```
Definition upd_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.
```

In order to deal with heap-allocated data structures, we extend the assertion language with connectives from separation logic. For example, the *mapsto* connective (`var_e x ↦ a::b::...`) holds for a heap that contains an array of contiguous words `a`, `b`, ... starting at the address contained in register `x`. Another example of a separating connective is the *separating conjunction*: `P ⋆ Q` holds for a state whose heap can be divided into two sub-heaps such that `P` and `Q` respectively hold for the "sub-states". In practice, the separating conjunction provides a concise way to express that two data structures reside in disjoint parts of the heap.

4 Multi-precision Arithmetic

Using our encoding in Coq of separation logic for SmartMIPS, we have written, specified, and verified the implementations of several multi-precision arithmetic functions. In this section, we explain the verification of the Montgomery multiplication.

4.1 A Library for Specifications

For specification of arithmetic functions, we need to introduce new predicates and functions. In particular, the writing of loop invariants requires predicates to talk about "partial" multi-precision integers, to represent the decimal values of partial products for example. For this purpose, we use the `Sum` function: `(Sum k A)` represents the decimal value of the `k` first words of the list of machine integers `A`. Another useful definition is equality modulo; in the following, `a==b[[n]]` is a Coq definition for $a \equiv b[n]$.

For the rest, we can simply reuse predicates and functions from the standard Coq library. For example, in the following, `(Nth 0 M)` represents the first element of the list of words `M`. More importantly, we can reuse the standard predicates for relative integers to specify overflow conditions; this is a benefit of our use of shallow encoding and our lemmas that relate machine integers to their decimal values.

4.2 Montgomery Multiplication

The Montgomery multiplication is a modular multiplication that is used in many implementations of public-key crytposystems. Given three `k`-word integers `X`, `Y`, and `M` such that `Sum k X * Sum k Y < Sum k M`, 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]]
```

The advantage of the Montgomery multiplication is that it does not require a multi-precision division, but uses shifts instead. For this to be possible, it is necessary to pre-compute the modular inverse `alpha` of the least significant word of the modulus:

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

The implementation of the Montgomery multiplication we dealt with is the "Finely Integrated Operand Scanning" (FIOS) variant [4]. Its main characteristic is to have only one inner-loop, in which it adds two products of 32-bit words. In general, this addition is problematic because it does not fit in the integer multiplier, but in SmartMIPS, the integer multiplier is large enough.

The complete triple specifying the Montgomery multiplication is displayed in Fig. 4.2.

5 Experiments

The table below summarize the sizes of verified functions and of the corresponding proof scripts:

Arithmetic function (insns)	Size of proof scripts (lines)		
	total	assertions (ratio)	tactics (average)
addition (11)	853	199 (23%)	654 (59)
subtraction (22)	1546	359 (23%)	1187 (54)
multiplication (20)	1698	436 (26%)	1262 (63)
Montgomery (36)	3758	946 (25%)	2812 (78)

Proof scripts tend to be long because of the size of assertions: pre/post-conditions occupy around 25% of proof scripts. Since assertions only change a little from step to step, appropriate tactics for forward reasoning should get rid of this overhead. The verification of each step requires in average 70 calls to tactics. Some steps are inherently difficult because low-level manipulations of multi-precision integers require many syntactic maniputations of goals and hypotheses. Yet, many parts of proof scripts are repetitive (`trivial` goals, obvious rewriting, etc.) and we already have a good deal of small-scale tactics. As a mid-term goal, it should be possible to use in average no more than 20 calls to tactics per step.

6 Related Work

Much work about encoding of assembly in proof assistants has been done for proof-carrying code (e.g., [7]). In this work, the encoded semantics allows for unstructured control-flow but details such as machine integers are not treated. For this reason, these encodings cannot be directly reused for formal verification of arithmetic functions, whose algorithms require precise specifications regarding overflow conditions and carry propagation.

The limitation to structured programs concretely means that arbitrary jumps cannot be represented directly. Ideally, we should encode a more general assembly language with arbitrary jumps in which to embed the subset presented in this paper. Such logics already exist (e.g., [9]).

Other encodings of machine integers in Coq have recently been proposed. Leroy has implemented a library for integers modulo 2^{32} using the relative integers of Coq instead of lists ot bits [10]. Chlipala has implemented a library similar to ours based on dependent vectors [11].

7 Conclusion

We have proposed an encoding in Coq of separation logic for a subset of SmartMIPS. Using this encoding, we have formally verified the implementation of several arithmetic functions, including the Montgomery multiplication, a function used in the implementation of many cryptosystems. At the heart of our encoding 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.

In order to verify programs involving several functions, we are currently working on an encoding of function calls and returns. We are also planning to encode the semantics of MIPS exceptions, so as to enable verification of embedded systems.

Acknowledgments The authors thank Pascal Paillier at Gemplus/Gemalto for providing explanations about the Montgomery multiplication.

References

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

[2] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(10):576–585, 1969.

[3] The Coq Proof assistant. `http://coq.inria.fr`.

[4] Ç. Kaya Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro 16(3):26–23, 1996.

[5] IEEE Computer Society. 17th IEEE Symposium on Logic in Computer Science (LICS 2002).

[6] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In [5].

[7] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof-Carrying Code. In [5].

[8] N. Marti, R. Affeldt, and A. Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th International Conference on Formal Engineering Methods (ICFEM 2006).*

[9] G. Tan and A. W. Appel. A Compositional Logic for Control Flow. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006).*

[10] X. Leroy. Formal certification of a compiler backend. In *33rd ACM Symposium on Principles of Programming Languages (POPL 2006).*

[11] A. Chlipala. Modular Development of Certified Program Verifiers with a Proof Assistant. In *11th ACM International Conference on Functional Programming (ICFP 2006).*

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

  {{ fun s s' m_ h => ∃ Z, length Z = nk ∧
    list_of_zeros Z ∧ multiplier.is_null m_ ∧
    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 }}

  montgomery k alpha x y z m int_ ext 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 }}.
```

Figure 1: Formal Specification of the Montgomery Multiplication