Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code

Magnus O. Myreen[1] and Gregorio Curello[2]

[1] Computer Laboratory, University of Cambridge, UK
[2] Autonoma University of Barcelona, Spain

Abstract. Verification of machine code can easily deteriorate into an endless clutter of low-level details. This paper presents a case study which shows that machine-code verification does not necessitate ghastly low-level proofs. The case study we describe is the construction of an x86-64 implementation of arbitrary-precision integer arithmetic. Compared with closely related work, our proofs are shorter and, more importantly, the reasoning is at a more convenient high level of abstraction, e.g. pointer reasoning is largely avoided. We achieve this improvement as a result of using an abstraction for arrays and previously developed tools, namely, a proof-producing decompiler and compiler. The work presented in this paper has been developed in the HOL4 theorem prover. The case study resulted in 800 lines of verified 64-bit x86 machine code.

1 Introduction

Hardware executes all software in the form of machine code. As a result, program verification ought to, ultimately, provide guarantees about the execution of the machine code. However, direct manual verification of machine code is to be avoided as such verification proofs easily become lengthy and unmaintainable.

Recent advances in compiler verification seem promising (e.g. [10]), making it possible to relate verification results from the source code to the compiler-generated machine code. Unfortunately, current verified compilers do not support source code with real inlined assembly (since the semantics of inlined assembly is difficult to state in terms of the semantics of the source language). Inlined assembly is a natural component of certain programs: programs that need direct access to hardware peripherals (e.g. operating systems), hand-optimised code or special-purpose machine instructions.

This paper presents a case study which shows that machine-code verification does not always require ghastly unmaintainable proofs. This paper describes how a proof-producing decompiler and compiler can together make it easy to produce verified machine code that essentially contains inlined assembly.

The case study we describe is the construction of an x86-64 implementation of arbitrary-precision arithmetic (bignum) functions. We have implemented the basic integer arithmetic operations (i.e. $+, -, \times, \mathsf{div}, \mathsf{mod}, <, =$) for arbitrary sized integers (represented as arrays in memory) and have proved that this x86-64 implementation correctly performs the desired arithmetic operations and leaves memory untouched outside the result array. The implementation makes use of special-purpose instructions for multi-word arithmetic.

This paper makes the following contributions.

- The proofs presented in this paper have produced a *reusable* verified x86-64 implementation of bignum integer operations. We envisage that this implementation will be of use in construction of larger bodies of verified code, for example, verified language runtimes that provide support for bignum arithmetic. For the purpose of reuse, we keep all interfaces clean and simple.
- Compared with closely related work (Section 7), our proofs are shorter and, more importantly, the reasoning is at a more convenient high level of abstraction, e.g. pointer reasoning is largely avoided. This improvement in the length and level of detail in the proofs is due to the use of a convenient abstraction for arrays and use of previously developed tools, namely a proof-producing decompiler and compiler, that can easily be made to operate over this *domain-specific abstraction*.
- To the best of our knowledge, we are the first to have formally verified functional correctness of machine code that implements bignum integer division.

The case study resulted in 800 lines of verified 64-bit x86 machine code. The proof development[3] presented in this paper has been carried out in the HOL4 theorem prover [20].

2 Method

The method by which we construct the verified x86 implementation consists of three steps:

1. We start by defining the algorithms involved as functions in logic. The functions operate over lists of binary words. We prove that these functions correctly implement integer arithmetic, e.g. given two lists that represent the 'digits' of two integer numbers, the function for an arithmetic operation returns a list that is the representation of the 'digits' of the resulting integer. These high-level functions summarise the operations of the algorithm separately from any architecture details, e.g. the machine-word length is kept as a (type) variable throughout. (Section 3)
2. In order to generate and reason about machine code that implements the functions from above, we instantiate a proof-producing compiler and decompiler with information about how lists of 64-bit 'digits' can be represented in memory as arrays. Concretely, we define a separation logic assertion about arrays and prove theorems about x86 machine instructions for load and store instructions that access and update arrays. The decompiler and compiler can use these theorems to make it seem as if the underlying machine has a memory that consists of arrays. (Section 4)

[3] The HOL4 scripts are available at `http://www.cl.cam.ac.uk/~mom22/cpp13/`

3. Finally, we use the decompiler to prove theorems for hand-written x86 assembly, and then use the array-aware compiler to produce x86 machine code that uses hand-written assembly and the x86 instructions that we proved to have array-like behaviour. The compiler takes as input functions that are restricted in format, but otherwise operate over the same types as the algorithm specifications from Step 1 (instantiated to 64-bit word length). The compiler produces as output a proof of a theorem which states that the input function is an accurate description of the behaviour of the generated machine code. We manually prove that the input to the compiler perform the same steps as the algorithm specifications from Step 1. (Section 5)

The result of combining all of the correctness theorems together is a single theorem (Section 6) describing the behaviour of a single chunk of x86 machine code, for which we have a top-level correctness theorem: given an operation identifier (referring to one of $+, -, \times, \mathsf{div}, \mathsf{mod}, <, =$), pointers to two immutable input arrays and a pointer to a separate mutable array, where the result is to be stored, execution of the verified x86 code terminates with the result of the arithmetic operation stored in the mutable array.

3 Algorithm specification and verification

As mentioned above, the first step is to specify the bignum algorithms as functions in logic and verify that they correctly compute integer arithmetic. This section provides details on this first step.

3.1 Abstract representation of bignums

The algorithms operate over lists of machine words. In order to make sure these algorithm specifications do not get tied to any particular architecture, we use a variable as the length of the machine word. In HOL, machine words are most conveniently modelled as finite cartesian products of booleans, a neat idea by Harrison [7], which allows (the cardinality of) a type to define the size of the word. We will write $\texttt{bool}^\alpha$ for the type of words of width α and $\texttt{bool}^{64}$ for the type of words with 64 bits. In this section, all words will have a variable width, i.e. have type $\texttt{bool}^\alpha$. In subsequent sections, all words will be specialised to be 64 bits wide, i.e. have type $\texttt{bool}^{64}$.

For this representation, we have the usual word operations and mappings for turning a natural into a word (`n2w`) and back (`w2n`):

$$\texttt{n2w} : \mathbb{N} \rightarrow \texttt{bool}^\alpha$$

$$\texttt{w2n} : \texttt{bool}^\alpha \rightarrow \mathbb{N}$$

Note that `n2w` and `w2n` have polymorphic types and their definition depends on this type. The following theorems describe their relationship:

$$\forall \texttt{n}.\ \ \texttt{w2n (n2w n) = n MOD } 2^\alpha \qquad\qquad \forall \texttt{w}.\ \ \texttt{n2w (w2n w) = w}$$

The algorithms operate over lists of such words, i.e. lists of type `bool`$^{\alpha}$ `list`. We have functions that map natural numbers to lists of multiple words (`n2mw`) and back (`n2mw`). Here and throughout `::` is list cons.

```
n2mw n = if n = 0 then [] else n2w (n MOD 2^α) :: n2mw (n DIV 2^α)

mw2n [] = 0
mw2n (w::ws) = w2n w + 2^α × mw2n ws
```

We also define functions which translate between integers and a representation of integers as a pair consisting of a sign and a list of machine words.

```
i2mw i = (i < 0, n2mw (abs i))

mw2i (sign, ws) = if sign then 0 - mw2n ws else mw2n ws
```

Thus, the algorithm functions operate over bignum integers as represented by terms of type `bool` $\times$ (`bool`$^{\alpha}$ `list`).

3.2 Algorithm specifications

The algorithm specification for each arithmetic function is a function of the following type. The comparison operations, of course, return `bool`.

$$(\texttt{bool} \times (\texttt{bool}^{\alpha}\ \texttt{list})) \to (\texttt{bool} \times (\texttt{bool}^{\alpha}\ \texttt{list})) \to (\texttt{bool} \times (\texttt{bool}^{\alpha}\ \texttt{list}))$$

The following presents our specification of the long-multiplication algorithm (`mwi_mul`). Multiplication will be our running example, since it is neat and simple compared with the tedium of dealing with alternating signs and variable length arguments for bignum integer addition or subtraction.

Our specification of multiplication describes the operations of the standard school-book long-multiplication.

```
    6 2 3 5 1
        2 4 6
-------------
  3 7 4 1 0 6
 2 4 9 4 0 4
1 2 4 7 0 2
-------------
1 5 3 3 8 3 4 6
```

There are, of course, a number of more sophisticated and better algorithms [9], e.g. the Karatsuba and Tom-Cook algorithms are significantly faster for large inputs; and Montgomery multiplication is better suited for multiplications that are to be performed modulo a prime number.

When modelling the multiplication algorithm, we start by defining a few primitive operations that we can expect to implement in custom assembly. For example, we define a function for word addition with a carry-in and carry-out.

```
single_add (x:bool^α) (y:bool^α) (c:bool) =
  (n2w (w2n x + w2n y + if c then 1 else 0),
   2^α ≤ w2n x + w2n y + if c then 1 else 0)
```

And a similar function for multiplication, which given three words, `x`, `y`, `z`, computes `w2n x` $\times$ `w2n y + w2n z` and returns two words describing this result. We expect either to find such a machine instruction in each architecture or implement this operation using a few instructions.

```
single_mul (x:bool^α) (y:bool^α) (z:bool^α) =
  (n2w (w2n x × w2n y + w2n z),
   n2w ((w2n x × w2n y + w2n z) DIV 2^α))
```

Equipped with the functions from above, we can define a function for the body of the inner loop of multiplication. We follow the standard school-book long-multiplication algorithm almost exactly. The only minor optimisation is that the additions that are done on paper last are done by this algorithm in conjunction with the rest of the computation. The function describing the body of the inner loop takes word, `p` and `q`, from each input and a word `k` from the accumulated result. The body performs a multiplication and two additions:

```
single_mul_add p q k s =
  let (x1,x2) = single_mul p q k in
  let (y1,c1) = single_add x1 s false in
  let (y2,c2) = single_add x2 0 c1 in
    (y1,y2)
```

The function describing the inner loop traverses one of the inputs `ys` and the accumulated result `zs` for one word from the other input `x`.

```
mw_mul_pass x [] zs k = [k]
mw_mul_pass x (y::ys) (z::zs) k =
  let (y1,k1) = single_mul_add x y k z in
    y1 :: mw_mul_pass x ys zs k1
```

The outer loop calls the inner loop for each word in the first input.

```
mw_mul [] ys zs = zs
mw_mul (x::xs) ys zs =
  let zs2 = mw_mul_pass x ys zs 0 in
    HD zs2 :: mw_mul xs ys (TL zs2)
```

The entire multiplication algorithm comes together in `mwi_mul`, which computes the resulting sign and initialises the accumulated result to all zeros before starting the loop. Below, `mw_rm_zero` deletes the possible leading zero from the result.

```
mwi_mul (s,xs) (t,ys) =
  if (xs = []) ∨ (ys = []) then (false,[]) else
    (s ≠ t, mw_rm_zero (mw_mul xs ys (MAP (λx. 0) ys)))

mw_rm_zero [] = []
mw_rm_zero (xs ++ [x]) = if x = 0 then xs else xs ++ [x]
```

In the eventual machine code, `mw_rm_zero` shortens the length of the array where the result is stored so that that array has no leading zero.

3.3 Algorithm verification

The top-level correctness theorem for each arithmetic operation is easy to state using the function `i2mw` for converting an integer into a signed list of words. For multiplication, the correctness statement relates `mwi_mul` to multiplication over the integers (×).

```
∀i j.  mwi_mul (i2mw i) (i2mw j) = i2mw (i × j)
```

Such statements guarantee that zero will never have the negative sign set and that `mwi_mul` never returns a list of words with redundant leading zeros.

Although the correctness theorem is stated in terms of `i2mw`, it seems easiest to arrive at the correctness theorem via proofs about `mw2n`. Each component in the algorithm has a neat description in terms of `mw2n` and `w2n`.

```
∀p q k1 k2 x1 x2.
  single_mul_add p q k1 k2 = (x1,x2)  ⟹
  w2n x1 + 2^α × w2n x2 = w2n p × w2n q + w2n k1 + w2n k2

∀ys zs x k.
  LENGTH ys = LENGTH zs  ⟹
  mw2n (mw_mul_pass x ys zs k) = w2n x × mw2n ys + mw2n zs + w2n k

∀xs ys zs.
  LENGTH ys = LENGTH zs  ⟹
  mw2n (mw_mul xs ys zs) = mw2n xs × mw2n ys + mw2n zs
```

4 Instantiation of proof tools for arrays

With the bignum arithmetic algorithms specified and verified in the previous section, this section describes the Hoare logic and proof tools that are used in the next section for construction of the verified machine-code implementation.

4.1 Hoare logic for machine code

We will skip a detailed description of the operational semantics for x86 used in this paper, since that semantics has been described previously [15]. Instead, a few examples will be used to explain features of a machine-code Hoare logic [14] that sits on top of the bare operational semantics.

All our reasoning about x86 machine code is performed through a machine-code Hoare logic, which can be instantiated to different instruction set architectures. Here we consider only an instantiation to 64-bit x86.

The following is a Hoare triple describing an x86 instruction `add r8,r9`, encoded as `4D01C8`, that adds the content of 64-bit register 8 with register 9 and stores the result in register 8. The following Hoare triple can be read informally as follows: *for any state where* the program counter (`PC`) is `p`, register 8 and 9 are `r8` and `r9`, respectively, the flags have some value (`S _`), and `4D01C8` is at location p in memory, *execution will reach a state where* the program counter is set to p + 3, register 8 contains the value r8 + r9 and the flags again have some value (S _). Here * is a form of separating conjunction [18, 14]. Details of this separating conjunction are unimportant for this paper. However, it is worth noting that these Hoare triples are part of a separation logic and, in particular, that all other resources not mentioned in the precondition of such a Hoare triple must have been kept unchanged (e.g. the theorem below implicitly states that the value of register 10 was unaffected by the add r8,r9 instruction).

```
{ PC p * R8 r8 * R9 r9 * S _ }
  p : 4D01C8
{ PC (p + 3) * R8 (r8 + r9) * R9 r9 * S _ }
```

An unusual feature of these Hoare triples is that the pre- and postconditions include the value of the program counter. Its inclusion makes it easy to specify branch instructions. Example: a jump-if-equal instruction, je -40 encoded as 48EBD5, is described by the following Hoare-triple theorem. The jump is conditional on the x86 z flag, which is set by most arithmetic operations.

```
{ PC p * S (a,c,o,p,z) }
  p : 48EBD5
{ PC (if z then p - 40 else p + 3) * S (a,c,o,p,z) }
```

Memory accesses are specified using a memory assertion memory m, which states that a part of memory (the set of addresses in domain m) are described by the partial function m. The following is a Hoare triple for a store instruction, mov [r8],r9 encoded as 4D8908, which stores the content of register 9 at an address given in register 8. This instruction is independent of the flags (S).

```
r8 ∈ domain m ∧ word_aligned r8  ⟹
{ PC p * R8 r8 * R9 r9 * memory m }
  p : 4D8908
{ PC (p + 3) * R8 r8 * R9 r9 * memory (m[r8 ↦ r9]) }
```

Note that the underlying model of x86 treats code as part of memory, but here the Hoare triples separate 'data' memory from the code. This is achieved by internally separating the precondition from the code segment using the separating conjunction *, for details see Myreen [15].

This Hoare logic supports the usual inference rules. As a result, one can perform proofs directly using these Hoare triples, as was done in previous work [13]. However, it is significantly easier if tools are used which automate much of the routine reasoning.

4.2 Proof-producing decompiler and compiler

Tool support, developed and carefully explained in previous work [14], is able to automate much of the routine Hoare logic reasoning. An example will illustrate what our decompiler can do. The HOL4 syntax below calls our decompiler for assembly code that computes, in r9, Knuth's D constant ahead of his bignum division algorithm [9].

```
val (x64_calcd_cert,x64_calcd_def) = x64_decompile "x64_calcd"
  ` LOOP: cmp r8,0
          js EXIT
          add r8,r8
          add r9,r9
          jmp LOOP
    EXIT:          '
```

This call to `x64_decompile` first runs an assembler to turn the assembly into concrete machine code, it then derives Hoare-triple theorems for each of the instructions and finally composes the Hoare triples together. The result of this composition is a theorem that describes one pass through the loop:

```
{ PC p * R8 r8 * R9 r9 * S _ }
  p : 4983F80 48789 4D1C0 4D1C9 48EBF0
{ let (p,r8,r9) =
    (if word_sign_bit r8 then (p + 14, r8, r9) else (p, r8+r8, r9+r9))
  in
    PC p * R8 r8 * R9 r9 * S _ }
```

By applying a special loop rule [14], the decompiler turns this theorem into a theorem describing a full terminating execution of the loop. The decompiler returns two functions describing the behaviour of the machine code: the first function describes the data update performed by the code, the second function encodes a side condition on termination. Loops always produce tail-recursive functions, which can be defined HOL without termination proofs.

```
x64_calcd (r8,r9) =
  if word_sign_bit r8 then (r8,r9)
  else let r8 = r8 + r8 in let r9 = r9 + r9 in x64_calcd (r8,r9)

x64_calcd_pre (r8,r9) =
  if word_sign_bit r8 then true
  else let r8 = r8 + r8 in let r9 = r9 + r9 in x64_calcd_pre (r8,r9)
```

The result of running the decompiler is the extraction of functions that describe the behaviour of the given machine code and a separate Hoare-triple theorem, which states that the function `x64_calcd` is an accurate description of the effect of executing the x86-64 machine code, if the side-condition `x64_calcd_pre (r8,r9)` is provable (which, in this case, is true if `r8` is initially non-zero). We call such Hoare-triple theorems *certificate theorems*.

```
x64_calcd_pre (r8,r9) ⟹
{ PC p * R8 r8 * R9 r9 * S _ }
  p : 4983F80 48789 4D1C0 4D1C9 48EBF0
{ let (r8,r9) = x64_calcd (r8,r9) in
     PC (p + 14) * R8 r8 * R9 r9 * S _ }
```

The beauty of using the decompiler is that all subsequent reasoning can be done on the extracted function `x64_calcd`, since any result proved for this function is related back to the machine code through the certificate theorem.

Writing assembly code manually is tiresome. To help with this, a proof-producing *compiler* has been constructed using the decompiler. This compiler essentially takes as input tail-recursive functions of the form x64_calcd, it then: (1) generates (without proof) assembly code based on the input function, (2) decompiles the assembly as above, and (3) proves that the function decompilation produced is identical to the function that was to be compiled, i.e. the compiler can return the certificate theorem produced by the underlying decompiler.

4.3 Array support in the compiler

As explained above, the decompiler and compiler produce their proofs by simply composing machine-code Hoare triples together. By default these tools use only automatically derived Hoare triples that provide a cumbersome flat functional view of the memory of the underlying x86 machine semantics.

The technique by which we instantiate the tools to the problem domain of bignum-array programs is to supply the tools with custom Hoare-triple theorems that are stated in terms of a domain-specific bignum-memory assertion. With such an assertion the decompiler and compiler can make the machine seem as if it has a memory containing arrays (in which we will store bignums).

We define the domain-specific assertion bignums based on the default memory assertion memory as explained below. The definition of bignums uses a few basic concepts of separation logic defined next. The separating conjunction ⋆ is defined as usual, taking the disjoint union (⊎) of two memory segments. The emp assertion is true only for the empty memory segment. The unusual part is our definition of the maps-to assertion: a ↦ x is true for a memory segment if the bytes of 64-bit word x are stored from 64-bit address a onwards. Here [7--0]x is notation for selecting bits 7 to 0 from x.

```
(p ⋆ q) m = ∃m1 m2. p m1 ∧ q m2 ∧ (m = m1 ⊎ m2)

emp m = (domain m = ∅)

(a ↦ x) m = (domain m = {a,a+1,a+2,a+3,...,a+7}) ∧
            (m a = [7--0]x) ∧ (m (a+1) = [15--8]x) ∧ ...
```

These basic concepts of separation logic are enough to define an array assertion for memory segments: array a xs is true for a memory segment m if the 64-bit words in list xs are stored in order from address a onwards.

```
array a [] = emp
array a (x::xs) = a ↦ x ⋆ array (a + 8) xs
```

The bignum code that we produce uses three arrays: we call the content of these arrays xs, ys and zs, and have pointers, xa, ya, za, point to the (word-aligned) base of these arrays. We allow pointers xa and ya to alias. If they do alias, then the content of xs and ys must be identical. The intention is that xs and ys hold bignum inputs and zs is the mutable result array.

```
bignum_memory m xa xs ya ys za zs =
  word_aligned xa ∧ word_aligned ya ∧ word_aligned za ∧
  if xa = ya then
    (xs = ys) ∧ (array xa xs ⋆ array za zs) m
  else
    (array xa xs ⋆ array ya ys ⋆ array za zs) m
```

The definition of the `bignums` assertion constrains the default `memory` assertion with the `bignum_memory` condition and states that pointers `xa`, `ya`, `za` are kept in registers 13, 14 and 15, respectively.

```
bignums (xa,xs,ya,ys,za,zs) =
  ∃m. memory m * R13 xa * R14 ya * R15 za *
      ⟨bignum_memory m xa xs ya ys za zs⟩
```

Using this `bignums` assertion, we can now manually verify a number of Hoare-triple theorems which make certain machine instructions seem as if they operate over arrays directly. For example the load instruction `mov r0,[8*r10+r13]`, encoded as `4B8B44D500`, loads the list element (`EL`) at index `w2n r10` from list `xs`, if `w2n r10` is within the size of `xs`.

```
w2n r10 < LENGTH xs  ⟹
{ PC p * R0 r0 * R10 r10 * bignums (xa,xs,ya,ys,za,zs) }
  p : 4B8B44D500
{ let r0 = EL (w2n r10) xs in
    PC (p + 5) * R0 r0 * R10 r10 * bignums (xa,xs,ya,ys,za,zs) }
```

Similarly, `mov [8*r10+r15],r0`, encoded as `4B8904D7`, updates (`LUPDATE`) list index `w2n r10` of list `zs`, if `w2n r10` is within the size of `zs`.

```
w2n r10 < LENGTH zs  ⟹
{ PC p * R0 r0 * R10 r10 * bignums (xa,xs,ya,ys,za,zs) }
  p : 4B8904D7
{ let zs = LUPDATE r0 (w2n r10) zs in
    PC (p + 4) * R0 r0 * R10 r10 * bignums (xa,xs,ya,ys,za,zs) }
```

Supplied with such Hoare-triple theorems, the compiler can compile functions which contain the following lines:

```
let r0 = EL (w2n r10) xs in

let zs = LUPDATE r0 (w2n r10) zs in
```

By supplying enough such Hoare-triple theorems, we can exclusively use only statements about recognised list/array operations and thus never, in manual proofs, require pointer reasoning beyond this point. Examples of compiled array accessing functions are given in Section 5.2.

5 Construction of verified machine code

With a proof-producing compiler that understands basic operations over a few arrays, we are ready to describe how one can construct verified implementations for the algorithms from Section 3. This section continues with the running example of multiplication.

5.1 Verification of hand-written assembly

Certain parts of the algorithms in Section 3 are best implemented in custom hand-written assembly. The following call to `x64_decompile` decompiles an assembly implementation of `single_mul_add` from Section 3.2. The assembler, that we use, aliases `r0` with `rax`, `r1` with `rcx`, `r2` with `rdx` and `r3` with `rbx`.

```
val (_, x64_single_mul_add_def) = x64_decompile "x64_single_mul_add"
  ‘ mul r2
    add r0,r1
    adc r2,0
    add r0,r3
    adc r2,0  ’
```

This call results in a function `x64_single_mul_add (r0,r1,r2,r3)`, which is easily proved to be an implementation of `single_mul_add`:

```
∀p k q s.
  x64_single_mul_add_pre (p,k,q,s) = true ∧
  x64_single_mul_add (p,k,q,s) =
    let (x1,x2) = single_mul_add p q k s in (x1,k,x2,s)
```

5.2 Using inlined assembly in compilations

Each run of the decompiler produces a certificate theorem. The certificate theorem produced for the decompilation above can be used in subsequent decompilations and compilations. Concretely, this means that the compiler can produce code for functions involving the line:

```
let (r0,r1,r2,r3) = x64_single_mul_add (r0,r1,r2,r3) in
```

Such lines result in code where the implementation of `x64_single_mul_add` is inlined in the generated machine code. The decompiler uses the certificate theorem for `x64_single_mul_add` at the point where it encounters the inlining.

This inlining feature allows writing an implementation of the inner loop, `mw_mul_pass`, of the multiplication algorithm. The function that we compile in order to generate machine code for `mw_mul_pass` is called `x64_mul_pass`. Its definition is shown in Figure 1. The compiler-generated machine code, shown in Figure 2, uses the custom assembly code and the list/array operations `EL` and `LUPDATE` from Section 4.3. A disassembly of the generated machine code is listed in Figure 2. The entire bignum library implementation is produced via such compilations that inline the result of previous compilations and decompilations.

```
val (_,x64_mul_pass_def,x64_mul_pass_pre_def) = x64_compile `
  x64_mul_pass (r1,r8,r9,r10,r11,ys,zs) =
    if r9 = r11 then
      let zs = LUPDATE r1 (w2n r10) zs in
      let r10 = r10 + 1w in
        (r1,r9,r10,ys,zs)
    else
      let r3 = EL (w2n r10) zs in
      let r2 = EL (w2n r11) ys in
      let r0 = r8 in
      let (r0,r1,r2,r3) = x64_single_mul_add (r0,r1,r2,r3) in
      let zs = LUPDATE r0 (w2n r10) zs in
      let r1 = r2 in
      let r10 = r10 + 1w in
      let r11 = r11 + 1w in
        x64_mul_pass (r1,r8,r9,r10,r11,ys,zs) '
```

Fig. 1. HOL4 syntax for a call to the compiler for `x64_mul_pass`

```
00: 4D39D9      L1:  cmp r9, r11
03: 48742C           je L2
06: 4B8B1CD7         mov r3,[8*r10+r15]    // EL (w2n r10) zs
0A: 4B8B14DE         mov r2,[8*r11+r14]    // EL (w2n r11) ys
0E: 498BC0           mov r0, r8
11: 48F7E2           mul r2                // inlined part
14: 4801C8           add r0,r1             // inlined part
17: 4883D20          adc r2,0              // inlined part
1B: 4801D8           add r0,r3             // inlined part
1E: 4883D20          adc r2,0              // inlined part
22: 4B8904D7         mov [8*r10+r15],r0    // LUPDATE r0 (w2n r10) zs
26: 488BCA           mov r1, r2
29: 49FFC2           inc r10
2C: 49FFC3           inc r11
2F: 48EBCE           jmp L1
32: 4B890CD7    L2:  mov [8*r10+r15],r1    // LUPDATE r1 (w2n r10) zs
36: 49FFC2           inc r10
```

Fig. 2. Annotated disassembly of machine code generated for `x64_mul_pass`

5.3 Verification of the generated machine code

Since the compiler produces a certificate theorem relating the given input function to the generated machine code, it suffices to prove properties of the input functions (and generated precondition functions) in order to prove the correctness of the machine code. For `x64_mul_pass`, this means that we need to prove that `x64_mul_pass` implements `mw_mul_pass`. The statement we prove, below, might seem hard to comprehend, but look closer and it becomes clear that this is a reasonably straight forward property. The length of the proof of this goal is less than twice the length of the goal statement.

```
∀ys x zs k zs1 zs2 z2.
  length zs = length ys ∧ length (zs1 ++ zs) < 2^64  ⟹
  ∃r1.
    x64_mul_pass_pre
      (k,x,n2w (length ys),n2w (length zs1),n2w 0,ys,
       zs1 ++ zs ++ z2::zs2) = true ∧
    x64_mul_pass
      (k,x,n2w (length ys),n2w (length zs1),n2w 0,ys,
       zs1 ++ zs ++ z2::zs2) =
    (r1,n2w (length ys),n2w (length (zs1 ++ zs) + 1),ys,
     zs1 ++ mw_mul_pass x ys zs k ++ zs2)
```

6 Results

The result of this verification effort is a verified library of bignum integer arithmetic functions implemented in 64-bit x86 machine code. The intention was to make this case study as reusable as possible so that future verified language implementations, e.g. future version of our verified Lisp implementation [16], can make use of arbitrary-precision integer arithmetic.

6.1 Top-level theorem

The verified library of integer arithmetic operations has a top-level entry point which implements a clean and simple interface: as inputs, it expects three pointers, pointers to two input arrays and one array for the result, it expects the length and sign of the input numbers to be provided in specific registers and it reads the operation identifier from a register. If the output array is long enough and disjoint from the input arrays, then the verified machine-code implementation will terminate with the result of the arithmetic operation of choice produced in the result array and the sign and length of the result return in a register. The input arrays are left unchanged.

6.2 In numbers

In order to give some measure of the effort involved, the table below lists how many lines of proof scripts were produced for each part of this project. The three middle columns list the length of our HOL4 proof scripts and the last column lists the number of instructions in the verified machine code that was produced.

part / operation	alg.	impl.	total	x86
prelude & tool setup	398	357	755	0
comparison	138	118	256	58
addition & subtraction	307	655	962	122
multiplication	149	266	415	105
division & modulus	2149	1482	3631	447
conversion to decimal	113	95	208	57
all parts together	3254	2973	6227	779

alg. — lines for specification and verification of algorithms (Section 3)
impl. — lines for construction and verification of machine code (Sections 4, 5)
total — sum of alg. and impl. columns
x86 — number of instructions in the verified 64-bit x86 machine code

One can (correctly) read from this table that the algorithm proofs were roughly as time consuming as the construction and verification of the machine code.

The verified algorithms are the obvious single-pass algorithms for comparison, addition and subtraction; the algorithm for multiplication was described in Section 3; the algorithm for division and modulus was taken from Knuth [9]; and the conversion into decimal form performs repeated division by 10.

7 Related work

The most closely related work on verified implementation of arithmetic functions is that of Affeldt [2], Fischer [6], Berghofer [4] and Moore [11]. We will also compare with the first author's early poster on this topic [13], and reflect on recent trends in programming logics for assembly verification.

Affeldt has constructed and verified SmartMIPS assembly code that implements the basic arithmetic functions: $+, -, \times, <, =$, notably excluding div and mod, but including Montgomery multiplication. Affeldt uses separation logic [18] and explicit reasoning about pointers in his verification proofs, which appear to be more low-level and labour intensive than the proofs reported on in this paper. Affeldt uses the GMP [1] library's bignum integer representation (which includes indirection) and, as a result, can not use the convenient array abstraction that was used in this paper. Affeldt proposes the use of a simulation relation to lift reasoning of compound operations to a more manageable level of detail.

Fischer and Berghofer both use the Isabelle/HOL theorem prover and both verify implementations written in a higher-level language. Fischer verified a C-like implementation of arbitrary-precision integer arithmetic, including division and modulus, using manual application of a separation-logic instantiation of Schirmer's Hoare logic framework [19]. Fischer reports that her proofs required significant manual effort to deal with selection of frames for the separation-logic reasoning. Her bignums were represented as linked lists. Berghofer verified a bignum library, which includes Montgomery multiplication but not division, written in Spark/Ada using a combination of the Spark/Ada tool suite and the Isabelle/HOL prover.

The first author's early poster, Myreen and Gordon [13], on the topic of machine-code verification showed that it is possible to use a Hoare logic directly to manually verify, in the HOL4 theorem prover, the correctness of ARM machine code implementing an optimised version of Montgomery multiplication.

Moore seems to have been the first to have formally verified the correctness of a bignum assembly routine, using the Nqthm prover. In his paper on the verified implementation of the Piton language, Moore explains that it is possible to verify an assembly routine for addition for bignums stored as arrays.

In terms of future direction, there seems to be a trend of making high-level language reasoning seamlessly available in the context of assembly verification. Significant recent work in this area include the programming logic by Jensen et al. [8], which has a powerful 'macro feature'. This macro feature makes it possible to define functions in the logic that operate over the assembly syntax and thus introduce, say, a while-loop macro and derive neat and familiar-looking proof rules for such, even though the reasoning is still about assembly code. Another noteworthy recent result in this area is Chlipala's Bedrock framework [5]. The Bedrock framework neatly fits into the Coq prover and provides proof tools which automate most routine separation-logic reasoning for assembly programs. The current paper has shown that our previously developed tools [14] are capable of providing convenient verification environment for the HOL4 theorem prover and, for this case study, explicit proofs about pointers can be avoided.

The work of this paper has focused on proof of full functional correctness. However, great strides have also been made in proofs of safety properties. Necula's work on proof-carrying code [17] spurred a lot of interest in low-level code [3, 21]. An exciting recent result in this area is a new method for software-fault isolation for real machine code [12].

8 Summary

This paper has demonstrated how a proof-producing decompiler and compiler can be used in the construction of verified machine-code implementations of bignum arithmetic. By careful instantiation of the previously developed tools, the entire verification effort is kept at a manageable complexity with proofs involving pointer reasoning nearly completely avoided (only present in Section 4.3). The resulting 64-bit x86 machine code was produced from both inlined custom assembly and functions written at a higher level of abstraction.

Acknowledgements. The first author was funded by the Royal Society, UK. The second author was a summer intern supported by the University of Cambridge Computer Laboratory, UK.

References

1. GMP, the GNU multiple precision arithmetic library, `http://gmplib.org/`
2. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innovations in Systems and Software Engineering 9(2) (2013)
3. Appel, A.W.: Foundational proof-carrying code. In: Logic in Computer Science (LICS). IEEE Computer Society (2001)
4. Berghofer, S.: Verification of dependable software using spark and isabelle. In: Brauer, J., Roveri, M., Tews, H. (eds.) Systems Software Verification (SSV). OASICS, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
5. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Hall, M.W., Padua, D.A. (eds.) Programming Language Design and Implementation (PLDI). ACM (2011)
6. Fischer, S.: Formal verification of a big integer library. In: DATE'08: Workshop on Dependable Software Systems (2008), `http://busserver.cs.uni-sb.de/publikationen/Fi08DATE.pdf`
7. Harrison, J.: A hol theory of euclidean space. In: Hurd, J., Melham, T.F. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). LNCS, Springer (2005)
8. Jensen, J.B., Benton, N., Kennedy, A.: High-level separation logic for low-level code. In: Principles of Programming Languages (POPL). ACM (2013)
9. Knuth, D.E.: The art of computer programming, volume 2: (2nd ed.) Seminumerical Algorithms. Addison Wesley Longman Publishing (1981)
10. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Principles of Programming Languages (POPL). ACM (2006)
11. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5 (1989)
12. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: Vitek, J., Lin, H., Tip, F. (eds.) Programming Language Design and Implementation (PLDI). ACM (2012)
13. Myreen, M., Gordon, M.J.C.: Verication of machine code implementations of arithmetic functions for cryptography. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, Emerging Trends Proceedings (TPHOLs, Poster Session). University of Kaiserslautern (2007), Internal Report 364/07
14. Myreen, M.O.: Formal verification of machine-code programs. Ph.D. thesis, University of Cambridge (2009)
15. Myreen, M.O.: Verified just-in-time compiler on x86. In: Hermenegildo, M.V., Palsberg, J. (eds.) Principles of Programming Languages (POPL). ACM (2010)
16. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: Interactive Theorem Proving (ITP). LNCS, Springer (2011)
17. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages (POPL). ACM (1997)
18. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). IEEE Computer Society (2002)
19. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technical University of Munich (2006)
20. Slind, K., Norrish, M.: A brief overview of HOL4. In: Theorem Proving in Higher Order Logics (TPHOLs). Springer (2008)
21. Tan, G., Appel, A.W.: A compositional logic for control flow. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking and Abstract Interpretation (VMCAI). Springer (2006)