Formal Verification of Arithmetic RTL: Translating Verilog to C++ to ACL2

David M. Russinoff
david@russinoff.com

We present a methodology for formal verification of arithmetic RTL designs that combines sequential logic equivalence checking with interactive theorem proving. An intermediate model of a Verilog module is hand-coded in *Restricted Algorithmic C (RAC)*, a primitive subset of C augmented by the integer and fixed-point register class templates of Algorithmic C. The model is designed to be as abstract and compact as possible, but sufficiently faithful to the RTL to allow efficient equivalence checking with a commercial tool. It is then automatically translated to the logic of ACL2, enabling a mechanically checked proof of correctness with respect to a formal architectural specification. In this paper, we describe the RAC language, the translation process, and some techniques that facilitate formal analysis of the resulting ACL2 code.

1 Introduction

A prerequisite for applying interactive theorem proving to arithmetic circuit verification is a reliable means of converting an RTL design to a semantically equivalent representation in a formal logic. Within the ACL2 community, this has been achieved in at least two industrial settings by mechanical translation from Verilog directly to the ACL2 logic. At Advanced Micro Devices, our approach was to generate a large clique of mutually recursive executable ACL2 functions in correspondence with the signals of a Verilog module [9], a scheme commonly known as "shallow embedding". The Centaur Technology verification team has followed a different path [3], converting an RTL design to a netlist of S-expressions to be executed by an interpreter coded in ACL2, i.e., a "deep embedding" of the design. Each approach has its advantages [1]—in this case, the result of the first is more suitable for traditional theorem proving while the second better enables verification by "bit-blasting", but both burden the user with an unwieldy body of ACL2 code, at least comparable in size to the Verilog source.

Theorem proving in general is uncommon in the chip design industry. A more popular formal verification technique is sequential logic equivalence checking, by which a design is checked against a trusted model, either a high-level C++ program or a legacy Verilog design, with the use of a commercial tool such as SLEC [6] or Hector [10]. This method has the advantage of being largely automatic and requiring less expertise of the user, but it suffers from two main deficiencies. First, the so-called "golden model" is usually trusted solely on the basis of testing, not having been formally verified itself. The second is the complexity limitations of these tools, which are likely to render them ineffective unless either the RTL is relatively simple or its design closely matches that of the model.

Here we describe a hybrid solution, initially developed at Intel [7] and now in regular use at Arm [8], that combines equivalence checking with theorem proving in a two-step process. First, an intermediate model is derived from the RTL by hand, coded in *Restricted Algorithmic C (RAC)*, a primitive subset of C augmented by the register class templates of Algorithmic C [5], which essentially provide the bit manipulation features of Verilog. The objective is a high-level model that is more manageable than the RTL but mirrors its microarchitecture to the extent required to allow efficient equivalence checking with

To appear in EPTCS.

SLEC [6]. This program is then processed by a RAC-ACL2 translator, which itself involves two steps: a special-purpose Flex/Bison parser [4] generates a set of S-expressions, which are converted to ACL2 functions by a code generator written in ACL2. Finally, the prover is used to check a proof of correctness of the design with respect to a formal architectural specification. Such proofs are supported by an ACL2 library of lemmas pertaining to RTL and computer arithmetic, found in the directory `books/rtl` of the ACL2 repository. This directory also contains specifications of the elementary arithmetic instructions of the x86 and Arm architectures. The library and the underlying arithmetic theory are documented in [8]. The RAC translator resides in `books/projects/rac`.

The primary advantage of this approach over direct translation is that it provides an abstract and readable representation of the design that is amenable to mathematical analysis, and consequently a compact ACL2 model that is more susceptible to formal proof. The translation of Arm floating-point units has been found to result in a reduction in code size by approximately 85%. The intermediate model serves other purposes as well, including documentation, design guidance (in some cases the RAC model has been written before the RTL), and simulation in a C++ environment. One disadvantage is the number of software systems involved, each of which may be viewed as a possible source of error. Another is the significant effort required in extracting the model from the design. On the other hand, much of this effort would still be needed for the proof effort.

This methodology has been successfully applied in the verification of a wide range of arithmetic components of Intel and Arm CPUs and GPUs, including high-precision multipliers, adders, and fused multiply-add modules; 64-bit integer multipliers and dividers; a variety of SRT division and square root modules of radices 2, 4, and 8; and a radix-1024 divider with selection by rounding. Examples of these applications, including RAC models, their ACL2 translations, and proof scripts, may be found in `books/projects/arm`. RAC is also the basis of the hardware/software co-assurance work of Hardin. [2]

In Section 2, we present the basic features of the RAC language. The parser and code generator are described in Sections 3 and 4. Some established techniques for proving theorems about RAC programs are discussed in Section 5.

2 Restricted Algorithmic C

The RAC language consists of a small set of constructs that have been found to be suitable for modeling RTL designs, especially floating-point units:

- Basic numerical data types: `bool`, `uint`, `int`, and `enums` (but no pointers);
- Composite types: arrays and `structs`;
- Simple statements: variable and constant declarations, assignments, type declarations, and assertions;
- Control statements (with restrictions): `if`, `for`, `switch`, and `return`;
- Functions (with value parameters only);
- Standard C++ library templates: `array` and `tuple`;
- Algorithmic C class templates: arbitrary width signed and unsigned integer and fixed-point registers, bit manipulation primitives, and logical operations.

An object of a register type is a bit vector of a width specified by a template parameter. The interpretation of the vector upon evaluation depends on its type. An unsigned integer register of width n is evaluated

```
ui32 add8(ui32 a, ui32 b) {
  ui32 result; ui8 sum;
  for (uint i=0; i<4; i++) {
    si8 aSgnd = a.slc<8>(8 * i);
    si8 bSgnd = b.slc<8>(8 * i);
    si9 sumSgnd = aSgnd + bSgnd;
    if (sumSgnd < -128)
      sum = -128;
    else if (sum >= 128)
      sum = 127;
    else
      sum = sumSgnd;
    result.set_slc(8 * i, sum);}
  return result;}
```

Figure 1: A signed integer adder

simply as an integer in the range $[0, 2^n)$. For a signed integer of the same width, bit $n-1$ is interpreted as a sign bit and the value is in the range $[-2^{n-1}, 2^{n-1})$. Fixed-point registers are similarly interpreted, except that an additional template parameter indicates the location of an implicit binary point. For example, an unsigned fixed-point register of width n with m integer bits holds an n-bit vector x with interpreted value $2^{m-n}x$.

The use of integer registers is illustrated in the first example of Figure 1. By convention, we denote an unsigned (resp., signed) integer register class of width n as `ui`n (resp., `si`n). In this case, the definition of `add8` has been preceded by the type declarations

```
typedef ac_int<32, false> ui32;
typedef ac_int<8, true> si8;
typedef ac_int<9, true> si9;
```

(This allows us subsequently to avoid some of the cumbersome syntax of C++.) Thus, the function takes 2 32-bit arguments, from which it extracts corresponding 8-bit slices, adds them as signed integers, "saturates" the sums to lie in the range $[-128, 128)$, and stores them in a 32-bit result vector. The bit slice extraction method `slc` has a template parameter indicating the width of the slice and an argument representing the base index. The `set_slc` method, which writes a value to a slice, takes two arguments, which determine the base index of the slice and the value to be written, the type of which determines the width of the slice. In the application of these methods, no disctinction between signed and unsigned or integer and fixed-point types is recognized. The same is true of logical operations on registers. But when a register is evaluated, the type is relevant. Thus, in the declaration

```
si9 sumSgnd = aSgnd + bSgnd;
```

the signed values of `aSgnd` and `bSgnd` are added using unbounded arithmetic (matching the semantics of ACL2) and the result is written to the signed register `sumSgnd`, truncating if necessary (although here the width 9 is chosen to avoid any loss of data). Similarly, in the next line, it is the signed value of that register that is compared to -128.

The absence of pointers and reference parameters from our language implies that a function simply returns a value without side-effects and that the C parameter-passing mechanism for arrays is disallowed. The purpose of including the two standard library templates is to compensate for these restrictions: the `array` template allows arrays to be passed by value and `tuple` provides the effect of multiple-valued functions (and in fact its use is restricted to this context).

A variety of control restrictions are imposed to facilitate translation to ACL2. Note that `do`, `while`, `continue`, and `break` (except in a limited way within a `switch` statement) are excluded. A `for` loop, in order to ensure admissibility of the generated recursive function (see Section 4), is required to have the form

`for (` *init*`;` *test*`;` *update*`) { ... }`

where

- *init* is either a declaration of, or an assignment to, the loop variable *var*, which must be of type `uint` or `int`.
- *test* is either a comparison between the loop variable and a numerical expression of the form *var op limit*, where *op* is `<`, `<=`, `>`, or `>=`, or a conjunction of the form $test_1$ `&&` $test_2$, where $test_1$ is such a comparison and $test_2$ is any boolean-valued term.
- *update* is an assignment to the loop variable.

The combination of *test* and *update* must guarantee termination of the loop. The translator derives a `:measure` declaration from *test*, which is used to establish the admissibility of the resulting function. In some cases, the *test* may be used to achieve the functionality of `break`. For example, instead of

`for (uint i=0; i<N; i++) {if (` *expr*`) break; ... }`

we may write

`for (uint i=0; i<N && !` *expr*`; i++) { ... }`

This feature may also be used in cases where the equivalence checker is unable to establish an absolute upper bound on the number of iterations executed by the loop, which is required for the "unrolling" that is performed by the tool. Thus, while ACL2 has no trouble establishing termination of either of the above loops, SLEC may require something like the following, which may be used when N is known never to exceed 128:

```
for (uint i=0; i<N && i<128; i++) { ... }
```

Further restrictions are imposed on the placement of `return` statements. We require every function body to be a statement block that recursively satisfies the following conditions:

(1) The statement block consists of a non-empty sequence of statements;

(2) None of these statements except the final one contains a `return` statement;

(3) The final statement of the block is either a `return` statement or an `if...else` statement of which each branch is either a `return` statement, an `if...else` statement that satisfies this condition, or a statement block that satisfies all three of these conditions.

The design of a program in this language is generally a compromise between two opposing objectives. On the one hand, a higher-level model is more susceptible to mathematical analysis and allows a simpler correctness proof. On the other hand, successful equivalence checking of a complex design generally requires a significant amount of proof decomposition, using techniques that depend on structural similarities between the model and the design.

As a rule of thumb, the model should be as abstract as possible while performing the same essential computations as the design. For example, in the case of a Booth multiplier, it is advisable to replicate the partial products and each level of the compression tree in order to allow the necessary decomposition.

```
ui6 CLZ64(ui64 x) {
  assert(x != 0);
  bool z[64];
  ui6 c[64];
  for (uint i=0; i<64; i++) {
    z[i] = !x[i];
    c[i] = 0;}
  uint n = 64;
  for (uint k=0; k<6; k++) {
    n = n/2; // n = 2^(5-k)
    for (uint i=0; i<n; i++) {
      c[i] = z[2*i+1] ? c[2*i] : c[2*i+1];
      c[i][k] = z[2*i+1];
      z[i] = z[2*i+1] && z[2*i];}}
  return c[0];}
```

Figure 2: A leading zero counter

For a high-precision SRT divider, we find that successful equivalence checking requires a bitwise match between the partial remainders and quotients on each iteration.

Abstraction and simplicity are achieved by eliminating implementation details and ignoring timing and cycle structure, all of which may be done without adversely affecting the equivalence check. Settling on the proper level of abstraction often requires some experimentation. Consider the leading zero counter of Figure 2, which executes in logarithmic time with respect to the argument width. After k iterations of the loop, where $0 \leq k \leq 6$, the value of the variable n is 2^{6-k} and the argument is conceptually partitioned into n slices of width 2^k. Each of the low order n entries of the boolean array z is set if and only if the corresponding slice is 0, and when this is not the case, the corresponding entry of the array c holds the number of leading zeroes of the slice. This function, which was designed to match the behavior of a component of a floating-point adder, consists of considerably less code than the Verilog version but allows a fast equivalence check. It is natural to ask whether the check would succeed if the function were to be based on a more transparent linear-time algorithm:

```
ui6 CLZ64(ui64 x) {
  int i;
  for (i=63; i>=0 && !x[i]; i--) {}
  return i;}
```

It does indeed, but the overall SLEC run-time for the adder that includes the function increases from 2 minutes to 22 minutes.

3 The RAC Parser

A by-product of the RAC parser is a more readable pseudocode version of a function, with C++ templates, methods, etc., replaced with a syntax that is more familiar to Verilog programmers. For example, the `slc` and `set_slc` calls in `add8` are replaced by

```
si8 aSgnd = a[8*i+7:8*i];
si8 bSgnd = b[8*i+7:8*i];
```

```
(FUNCDEF ADD8 (A B)
  (BLOCK (DECLARE RESULT 0)
         (DECLARE SUM 0)
         (FOR ((DECLARE I 0) (LOG< I 4) (+ I 1))
           (BLOCK (DECLARE ASGND (BITS A (+ (* 8 I) 7) (* 8 I)))
                  (DECLARE BSGND (BITS B (+ (* 8 I) 7) (* 8 I)))
                  (DECLARE SUMSGND (BITS (+ (SI ASGND 8) (SI BSGND 8)) 8 0))
                  (IF (LOG< (SI SUMSGND 9) -128)
                      (ASSIGN SUM (BITS -128 7 0))
                      (IF (LOG>= SUM 128)
                          (ASSIGN SUM (BITS 127 7 0))
                          (ASSIGN SUM (BITS (SI SUMSGND 9) 7 0))))
                  (ASSIGN RESULT (SETBITS RESULT 32 (+ (* 8 I) 7) (* 8 I) SUM))))
         (RETURN RESULT)))
```

Figure 3: RAC parser output

and

```
result[8*i+7:8*i] = sum;
```

Its primary purpose, however, is the conversion of a function definition to an S-expression that preserves its structure. The parser output for the function `add8` is displayed in Figure 3. Note that some RAC primitives correspond to built-in ACL2 functions, while others correspond to functions defined in the RTL book `lib/rac`: `BITS` and `SETBITS` extract and assign slices of a bit vector; `LOG<`, `LOG>=`, etc., are comparators that return 1 or 0 (emulating C); `SI` computes the signed integer value of a register of a given width. Other symbols that appear in the output—`FUNCDEF`, `BLOCK`, `FOR`, etc.—are not defined as ACL2 functions but are processed later by the code generator.

Note also that variable types do not explicitly appear in the output. The problem of translating a typed language to an untyped language is addressed by the parser, mainly by converting implicit register evaluations and type conversions to explicit computations. Thus, in the expression that is derived from the declaration

```
si9 sumSgnd = aSgnd + bSgnd;
```

the registers `ASGND` and `BSGND` are evaluated according to their type (computing their signed integer values), and when their sum is assigned to the 9-bit register `SUMSGND`, the low order 9 bits are extracted. Evaluation and assignment of fixed-point registers are more complicated, involving division and multiplication by appropriate powers of 2.

4 The ACL2 Code Generator

The primary objective considered in the design of the code generator was simplicity, of both the program and its output. Since the translation is yet another component of the verification process that must be trusted, along with SLEC and ACL2, it should be as transparent as possible. A secondary consideration is executability: efficiency is not an overriding concern, but simulation is important for the purpose of testing. A third objective is susceptibility of the output to formal analysis, but this was found to be in conflict with the first two and therefore ignored in the design. Instead, as discussed in Section 5, it is addressed by converting the object code to a more manageable and provably equivalent form.

Obviously, there are various idiomatic differences between C and ACL2 to be managed in the translation. Large constant arrays, which occur in RTL designs to represent blocks of read-only memory, are

```
tuple<ui53, ui53, si13> normalize(ui11 expa, ui11 expb, ui52 mana, ui52 manb) {
  ui53 siga = mana, sigb = manb;
  const uint bias = 0x3FF;
  si13 expaShft, expbShft;
  if (expa == 0) {
    ui6 clz = CLZ64(siga);
    siga <<= clz;
    expaShft = 1 - clz;}
  else {
    siga[52] = 1;
    expaShft = expa;}
  if (expb == 0) {
    ui6 clz = CLZ64(sigb);
    sigb <<= clz;
    expbShft = 1 - clz;}
  else {
    sigb[52] = 1;
    expbShft = expb;}
  si13 expQ = expaShft - expbShft + bias;
  return tuple<ui53, ui53, si13>(siga, sigb, expQ);}
```

```
(DEFUND NORMALIZE (EXPA EXPB MANA MANB)
  (LET ((SIGA MANA) (SIGB MANB) (BIAS 1023))
    (MV-LET (SIGA EXPASHFT)
            (IF1 (LOG= EXPA 0)
                 (LET ((CLZ (CLZ64 SIGA)))
                   (MV (BITS (ASH SIGA CLZ) 52 0)
                       (BITS (- 1 CLZ) 12 0)))
                 (MV (SETBITN SIGA 53 52 1) EXPA))
      (MV-LET (SIGB EXPBSHFT)
              (IF1 (LOG= EXPB 0)
                   (LET ((CLZ (CLZ64 SIGB)))
                     (MV (BITS (ASH SIGB CLZ) 52 0)
                         (BITS (- 1 CLZ) 12 0)))
                   (MV (SETBITN SIGB 53 52 1) EXPB))
        (MV SIGA SIGB
           (BITS (+ (- (SI EXPASHFT 13) (SI EXPBSHFT 13)) BIAS) 12 0))))))
```

Figure 4: Normalization of the operands of a floating-point divider

converted to lists of values; variable arrays and `structs` are represented as alists. Among the operators defined in `lib/rac` are the array access and assignment functions `AG` and `AS`.

The difference between the boolean values native to C and ACL2 (1 and 0 vs. `T` and `NIL`) requires attention. Along with the comparators `LOG<`, etc., that were mentioned in Section 3, `lib/rac` includes a macro `IF1`, which is similar to `IF` but tests its first argument against 0.

The primary difficulty faced in code generation, however, is the translation from an imperative to a functional paradigm. Our overall strategy is based on the conversion of sequences of assignments to nested bindings, using `LET`, `LET*`, and `MV-LET`. Each statement in the function body except the last corresponds to one or more variables that are bound in this nest. For each of these statements, the translator generates a triple consisting of the following:

(1) a list of the variables whose previous values are read by the statement;

(2) a list of the variables that are written by the statement;

(3) a term that evaluates to a multiple value consisting of the updated values of the variables of (2), or a single value if (2) is a singleton.

The bindings of the nest are derived from these triples. Each statement generates either a `LET` or an `MV-LET` depending on whether (2) is a singleton. Whenever possible, a nested sequence of `LET`s is combined into a single `LET` or `LET*`. The body of the nest is generated from the final statement of the function body.

Statements that require `MV-LET` include multiple-valued function calls and some conditional branching statements. In the latter case, the translation may be especially complicated, as in Figure 4, which displays the translation of a function that normalizes the operands of a floating-point divider and computes the biased exponent of the quotient.

Naturally, iteration is translated to recursion, with an auxiliary recursive function generated for every `for` loop. The arguments of this function consist of (a) the loop variable, (b) any previously assigned variables that are accessed inside the loop, including those that occur in the loop initialization or test, and (c) the variables that are assigned within the loop and are not local to the loop. A multiple value is returned comprising the updated values of the variables of (c). For example, for the recursive function `ADD8-LOOP-0` displayed in Figure 5, we have (a) `I`, (b) `A` and `B`, and (c) `SUM` and `RESULT`. Note that only one of the two values returned by the non-recursive call to this function is subsequently used. For this reason, along with various other possibilities for optimization that are ignored in the interest of simplifying the process, every translated RAC file begins with these two lines to pacify ACL2:

```
(SET-IGNORE-OK T)
(SET-IRRELEVANT-FORMALS-OK T)
```

The construction of this recursive function is similar to that of the top-level function, but the final statement of the body is not treated specially. Instead, the body of the nest of bindings is a recursive call in which the loop variable is replaced by its updated value. The resulting term becomes the left branch of an `IF` expression, of which the right branch is simply the returned variable (if there is only one) or a multiple value consisting of the returned variables (if there are more than one). The test of the `IF` is

`(AND (INTEGERP` *var*`) (INTEGERP` *limit*`)` *term*`)`

where the test of the loop is *var op limit* and *term* is the result of converting the test to an expression that evaluates to `T` or `NIL`. (The second conjunct of this term is omitted when *limit* is a constant.)

As shown in Figure 6, the translation of the leading zero counter includes three auxiliary functions, corresponding to the loop that initializes variables and the subsequent nested pair. Note that this program

```
(DEFUN ADD8-LOOP-0 (I A B SUM RESULT)
  (DECLARE (XARGS :MEASURE (NFIX (- 4 I))))
  (IF (AND (INTEGERP I) (< I 4))
      (LET* ((ASGND (BITS A (+ (* 8 I) 7) (* 8 I)))
             (BSGND (BITS B (+ (* 8 I) 7) (* 8 I)))
             (SUMSGND (BITS (+ (SI ASGND 8) (SI BSGND 8)) 8 0))
             (SUM (IF1 (LOG< (SI SUMSGND 9) -128)
                       (BITS -128 7 0)
                       (IF1 (LOG>= SUM 128)
                            (BITS 127 7 0)
                            (BITS (SI SUMSGND 9) 7 0))))
             (RESULT (SETBITS RESULT 32
                              (+ (* 8 I) 7) (* 8 I)
                              SUM)))
        (ADD8-LOOP-0 (+ I 1) A B SUM RESULT))
      (MV SUM RESULT)))

(DEFUN ADD8 (A B)
  (LET ((RESULT 0) (SUM 0))
    (MV-LET (SUM RESULT) (ADD8-LOOP-0 0 A B SUM RESULT)
      RESULT)))
```

Figure 5: Translation of the signed integer adder

```
(DEFUN CLZ64-LOOP-0 (I N K C Z) ... )

(DEFUN CLZ64-LOOP-1 (K N C Z)
  (DECLARE (XARGS :MEASURE (NFIX (- 6 K))))
  (IF (AND (INTEGERP K) (< K 6))
      (LET ((N (FLOOR N 2)))
        (MV-LET (C Z) (CLZ64-LOOP-0 0 N K C Z)
          (CLZ64-LOOP-1 (+ K 1) N C Z)))
      (MV N C Z)))

(DEFUN CLZ64-LOOP-2 (I X Z C)
  (DECLARE (XARGS :MEASURE (NFIX (- 64 I))))
  (IF (AND (INTEGERP I) (< I 64))
      (LET ((Z (AS I (LOGNOT1 (BITN X I)) Z))
            (C (AS I (BITS 0 5 0) C)))
        (CLZ64-LOOP-2 (+ I 1) X Z C))
      (MV Z C)))

(DEFUN CLZ64 (X)
  (LET ((ASSERT (IN-FUNCTION CLZ64 (LOG<> X 0)))
        (Z NIL)
        (C NIL))
    (MV-LET (Z C) (CLZ64-LOOP-2 0 X Z C)
      (LET ((N 64))
        (MV-LET (N C Z) (CLZ64-LOOP-1 0 N C Z)
          (AG 0 C))))))
```

Figure 6: Translation of the leading zero counter

```
bool compare64(ui64 a, ui64 b) {
  bool sgnA = a[63], sgnB = b[63];
  bool cin = sgnA || !sgnB;
  ui64 sum = ~a ^ ~b;
  ui64 carry = ((~a & ~b) << 1) | 1;
  ui64 add1, add2;
  if (sgnA && !sgnB) {
    add1 = sum;
    add2 = carry;}
  else {
    add1 = sgnA ? ui64(~a) : a;
    add2 = sgnB ? b : ui64(~b);}
  ui65 diff = add1 + add2 + cin;
  return !diff[64];}
```

Figure 7: Signed integer comparison

contains an assertion, a type of statement that we have not discussed. In RAC, as in C, an assertion has no semantic import but is useful in testing and as documentation. In this case, it indicates that `CLZ64` is not intended to be applied to the argument 0. In the translation, it is represented by the binding of the dummy variable `ASSERT` to a call to the macro `IN-FUNCTION`, defined as follows:

```
(defmacro in-function (fn term)
  `(if1 ,term () (er hard ',fn "Assertion ~x0 failed" ',term)))
```

Thus, attempted evaluation of `(CLZ64 0)` results in a run-time error:

```
HARD ACL2 ERROR in CLZ64:  Assertion (LOG<> X 0) failed
```

5 Proving Theorems about RAC Functions

In this section, we address certain difficulties that arise in the course of proving ACL2 theorems about translated RAC functions. The techniques presented here have been employed in every Intel or Arm RAC-based verification effort.

Proving a theorem of interest pertaining to an RTL module of any complexity is best begun, according to our experience, by analyzing the design and writing out an informal but detailed and rigorous proof. Some properties of bit vectors are simple enough to be derived automatically with `gl` [3], but even in these cases such a proof and the insight gained through its development are valuable. More often, the written proof must be checked by formalizing it step by step in a long sequence of ACL2 lemmas.

As an illustration, consider the function `compare64` of Figure 7, which compares the absolute values of two 64-bit signed integers. The computation that it performs is more complicated than necessary, using a fast carry-save adder followed by a slower carry-propagate adder, but it is also more efficient than the more obvious solutions. This becomes clear upon examination of the following informal (and uncharacteristically chatty) proof of correctness:

Lemma *Let A and B be the signed integers represented by 64-bit vectors a and b. Then*

$$compare64(a,b) = 1 \Leftrightarrow |B| > |A|.$$

PROOF: We shall examine the case $A < 0, B \geq 0$; the other cases are simpler. In this case, the function computes the complements of the operands, the values of which are

$$\tilde{}a[63:0] = 2^{64} - a - 1 = 2^{64} - (2^{64} - |A|) - 1 = |A| - 1$$

and

$$\tilde{}b[63:0] = 2^{64} - b - 1 = 2^{64} - |B| - 1.$$

If we were to compute the 65-bit sum

$$diff = \tilde{}a[63:0] + \tilde{}b[63:0] + 2 = (|A| - 1) + (2^{64} - |B| - 1) + 2 = 2^{64} + |A| - |B|,$$

then we could simply look at the most significant bit $diff[64]$ to see whether $|A| \geq |B|$. But while introducing a single carry-in to a carry-propagate adder does not affect the complexity of the operation, adding 2 instead of 1 requires a separate incrementer, consuming nearly as much time as a second addition. Therefore, the function proceeds by computing the carry-save vectors

$$add1 = sum = \tilde{}a[63:0] \hat{} \tilde{}b[63:0]$$

and

$$add2 = carry = 2(\tilde{}a[63:0] \,\&\, \tilde{}b[63:0]) + 1,$$

the sum of which, according to Lemma 8.2 of [8], is

$$\tilde{}a[63:0] + \tilde{}b[63:0] + 1 = 2^{64} + |A| - |B| - 1,$$

and then executes a single full addition to compute

$$diff = add1 + add2 + 1 = 2^{64} + |A| - |B|.$$

Thus, $compare64(a,b) = 1 \Leftrightarrow diff[64] = 0 \Leftrightarrow |B| > |A|$. ■

The purpose of this example is to illustrate the nature of proofs of this sort: we derive a sequence of intermediate results pertaining to various local variables, each assertion depending on previous assertions, until we arrive at the desired final result. Next, we would like to formalize this argument in a proof of the following, which refers to the translation displayed in Figure 8:

```
(defthm correctness-of-compare64
  (implies (and (bvecp a 64) (bvecp b 64))
           (equal (compare64 a b)
                  (if (> (abs (si b 64))
                         (abs (si a 64)))
                      1 0))))
```

Although this result is certainly simple enough for `gl`, we shall use it as an illustration of the process of formalizing a mathematical proof pertaining to a RAC program. Even in this case, in which the sequence of assignments is considerably shorter than one derived from a typical RTL module, we do not get far in this exercise before realizing that there is no convenient way to formalize the argument given above. How can we prove a lemma characterizing the bindings of the local variables `ADD1` and `ADD2` (or even state such a lemma) and then use it to prove another lemma about the value of `DIFF`? We have developed a four-step procedure that solves this problem, as illustrated below:

(1) *Introduce encapsulated constant functions corresponding to the arguments of the function of interest that are constrained to satisfy the hypotheses of the conjectured statement of correctness:*

```
(DEFUN COMPARE64 (A B)
  (LET* ((SGNA (BITN A 63))
         (SGNB (BITN B 63))
         (CIN (LOGIOR1 SGNA (LOGNOT1 SGNB)))
         (SUM (LOGXOR (BITS (LOGNOT A) 63 0)
                      (BITS (LOGNOT B) 63 0)))
         (CARRY (BITS (LOGIOR (ASH (LOGAND (BITS (LOGNOT A) 63 0)
                                           (BITS (LOGNOT B) 63 0))
                                   1)
                              1)
                      63 0)))
    (MV-LET (ADD1 ADD2)
            (IF1 (LOGAND1 SGNA (LOGNOT1 SGNB))
                 (MV SUM CARRY)
                 (MV (BITS (IF1 SGNA (LOGNOT A) A) 63 0)
                     (BITS (IF1 SGNB B (LOGNOT B)) 63 0)))
            (LET ((DIFF (BITS (+ (+ ADD1 ADD2) CIN) 64 0)))
              (LOGNOT1 (BITN DIFF 64))))))
```

Figure 8: Translation of `compare64`

```
(defund inputsp (a b)
  (and (bvecp a 64) (bvecp b 64)))

(encapsulate (((a) => *) ((b) => *))
  (local (defun a () 0))
  (local (defun b () 0))
  (defthm inputs-ok (inputsp (a) (b))
    :hints (("Goal" :in-theory (enable inputsp)))))
```

(2) *Derive definitions of constant functions corresponding to the local variables directly from the variables' bindings, and prove that the function maps the input constants to the output constants.* This is a straightforward process, performed automatically by a tool developed by Cuong Chau, which resides in `books/projects/rac/lisp/`. (It was previously done by hand, which was tedious, time-consuming, error-prone.) The output of the tool for the function `compare64` is displayed in Figure 9. Note that the arguments of Chau's function `const-fns-gen` include the function to be transformed and names to be associated with the outputs. The generated functions allow us to reason about the variables and derive a sequence of lemmas that mirrors the informal proof outlined above. Note that the theorem, generated with appropriate hints, is established by the prover simply by expanding all relevant definitions. This can result in unmanageable code explosion if a function definition is too long. With this in mind, large RAC models must be designed with sufficient modularity to avoid this result.

(3) *Derive the required properties of the outputs:*

```
(defthmd compare64-main
  (equal (r) (if (> (abs (si (b) 64)) (abs (si (a) 64))) 1 0)))
```

Virtually all of the work lies in developing a proof script culminating in this theorem, i.e., a formalization of the informal argument presented earlier.

```
RTL !>(include-book "~/acl2/books/projects/rac/lisp/internal-fns-gen")
RTL !>(const-fns-gen 'compare64 'r state)

(DEFUNDD SGNA NIL (BITN (A) 63))

(DEFUNDD SGNB NIL (BITN (B) 63))

(DEFUNDD CIN NIL (LOGIOR1 (SGNA) (LOGNOT1 (SGNB))))

(DEFUNDD SUM NIL
  (LOGXOR (BITS (LOGNOT (A)) 63 0)
          (BITS (LOGNOT (B)) 63 0)))

(DEFUNDD CARRY NIL
  (BITS (LOGIOR (ASH (LOGAND (BITS (LOGNOT (A)) 63 0)
                             (BITS (LOGNOT (B)) 63 0))
                     1)
                1)
        63 0))

(DEFUNDD ADD1 NIL
  (IF1 (LOGAND1 (SGNA) (LOGNOT1 (SGNB)))
       (SUM)
       (BITS (IF1 (SGNA) (LOGNOT (A)) (A)) 63 0)))

(DEFUNDD ADD2 NIL
  (IF1 (LOGAND1 (SGNA) (LOGNOT1 (SGNB)))
       (CARRY)
       (BITS (IF1 (SGNB) (B) (LOGNOT (B))) 63 0)))

(DEFUNDD DIFF NIL (BITS (+ (+ (ADD1) (ADD2)) (CIN)) 64 0))

(DEFUNDD R NIL (LOGNOT1 (BITN (DIFF) 64)))

(DEFTHMD COMPARE64-LEMMA
  (EQUAL (R) (COMPARE64 (A) (B)))
  :HINTS (("Goal" :DO-NOT '(PREPROCESS) :EXPAND :LAMBDAS
                  :IN-THEORY '(C SGNA SGNB CIN SUM CARRY ADD1 ADD2 DIFF COMPARE64)))))
```

Figure 9: Automatically generated definitions

(4) *Lift this result by functional instantiation to establish the original statement of correctness.* The lemma to be instantiated is a simple combination of the results of (2) amd (3):

```
(defthmd lemma-to-be-lifted
  (equal (compare64 (a) (b))
         (if (> (abs (si (b) 64)) (abs (si (a) 64))) 1 0))
  :hints (("Goal" :use (compare64-lemma compare64-main))))
```

A lemma is functionally instantiated by replacing some subset of the functions that appear in it with another set of functions. The prover is then obligated to prove the corresponding instances of all axioms that pertain to the first set, which are usually just their definitions. In this case the only such axiom is the formula that is exported from the encapsulation that introduced the constants `(a)` and `(b)`. The `:use` hint is formulated as follows:

```
(defthm correctness-of-compare64
  (implies (inputsp a b)
           (equal (compare64 a b)
                  (if (> (abs (si b 64)) (abs (si a 64)))
                      1 0)))
  :hints (("Goal" :use (:functional-instance lemma-to-be-lifted
                         (a (lambda () (if (inputsp a b) a (a))))
                         (b (lambda () (if (inputsp a b) b (b))))))))
```

The choice of instantiation is perhaps best understood by examining the subgoals that it generates. `Subgoal 2` is just the top-level goal with the indicated functional instance inserted as a hypothesis; `Subgoal 1` is the incurred proof obligation mentioned above:

```
... We are left with the following two subgoals.

Subgoal 2
(IMPLIES (EQUAL (COMPARE64 (IF (INPUTSP A B) A (A))
                           (IF (INPUTSP A B) B (B)))
                (IF (< (ABS (SI (IF (INPUTSP A B) A (A)) 64))
                       (ABS (SI (IF (INPUTSP A B) B (B)) 64)))
                    1 0))
         (IMPLIES (INPUTSP A B)
                  (EQUAL (COMPARE64 A B)
                         (IF (< (ABS (SI A 64)) (ABS (SI B 64)))
                             1 0)))).

But we reduce the conjecture to T, by case analysis.

Subgoal 1
(INPUTSP (IF (INPUTSP A B) A (A))
         (IF (INPUTSP A B) B (B))).

This simplifies, using the simple :rewrite rule INPUTS-OK, to T.

Q.E.D.
```

Finally, we note that the book containing `const-fns-gen` includes a second program, `loop-fns-gen`, which is critical in dealing with iterative designs, dividers in particular. Such a module is typically modeled by a large `for` loop with a loop variable that is incremented on each iteration. In this situation, it is necessary to reason about the value of a variable that is computed in a given iteration and to relate it to values computed in preceding interations. Thus, instead of constant functions, `loop-fns-gen` generates a function of a single argument, representing the loop variable, for each variable that is updated iteratively. Space does not allow an illustration of this technique here, but examples may be found in the `fdiv` and `fsqrt` subdirectories of `books/projects/arm`.

References

[1] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert & J. Van Tassel (1992): *Experience with Embedding Hardware Description Languages*. In: International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, North-Holland/Elsevier, pp. 129–156.

[2] D. Hardin (2020): *Put Me on the RAC*. In: ACL2 2020: 16th International Workshop on the ACL2 Theorem Prover and its Applications, Virtual Conference.

[3] W. Hunt, S. Swords, J. Davis & A. Slobadova (2010): *Use of Formal Verification at Centaur Technology*. In David S. Hardin, editor: Design and Verification of Microprocessor Systems for High-Assurance Applications, Springer, pp. 65–88.

[4] J. Levine (2009): *Flex and Bison*. O'Reilly Media.

[5] Mentor Graphics Corp.: *Algorithmic C Datatypes*. Available at `https://www.mentor.com/hls-lp/downloads/ac-datatypes`.

[6] Mentor Graphics Corp.: *Sequential Logic Equivalence Checker*. `https://www.mentor.com/products/fv/questa-slec`.

[7] J. O'Leary & D. Russinoff (2014): *Modeling Algorithms in SystemC and ACL2*. In: ACL2 2014: 12th International Workshop on the ACL2 Theorem Prover and its Applications, Vienna. Available at `http://www.russinoff.com/papers/masc.html`.

[8] D. Russinoff (2018): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer.

[9] D. Russinoff, M. Kaufmann, E. Smith & R. Sumners (2005): *Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover*. In: 17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation, Paris.

[10] Synopsys, Inc.: *Hector*. `https://research.ibm.com/haifa/conferences/hvc2008/present/CarlPixleyHVC08.pdf`.