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

David M. Russinoff
Arm

May 28, 2020

Verification of RTL Design with ACL2

Requirements for RTL verification by theorem proving:

- Semantics-preserving translation from a HDL to a logical language
- Formal architectural specification
- Mechanically checked proof of correctness

This talk will focus on translation: Verilog $\rightarrow$ ACL2.

VERILOG-ACL2: DIRECT TRANSLATION

- AMD: RTL is converted to a mutually recursive clique of executable ACL2 functions corresponding to RTL signals ("shallow embedding")
- Centaur: RTL is converted to a set of S-expressions to be executed by an interpreter coded in ACL2 ("deep embedding")

Each approach has its advantages, but both produce unwieldy amounts of ACL2 code.

Sequential Logic Equivalence Checking

An Alternative to ITP, more widely used in industry.

RTL is checked against a trusted high-level C++ or established Verilog model with a commercial tool.

Advantages:

- Less expertise required of user
- Automatic, fast results

Disadvantages:

- "Golden model" has usually not been formally verified
- Complexity limitations

Verilog-ACL2 Translation at Arm

A two-step process combining SLEC with ITP, based on Restricted Algorithmic C (RAC), a primitive C++ subset

- Verilog $\rightarrow$ RAC
  - Intermediate abstract model derived from RTL by hand, coded in RAC
  - Functional equivalence established with a commercial tool (Hector, SLEC)
- RAC $\rightarrow$ ACL2
  - Special-purpose Flex/Bison parser derives an S-expression representation of the model
  - ACL2 code generator derives an executable ACL2 model

Pros and Cons of this Approach

Advantages:

- More manageable ACL2 model:
  - 85% code reduction
  - Abstract, readable, amenable to formal analysis
- Intermediate RAC model has a variety of uses:
  - Documentation
  - Simulation
  - Design guidance

Disadvantages:

- Additional tool to be trusted
- Development of the model is a significant effort, but most of this is required for the proof.

RAC Features

A primitive subset of C augmented by several C++ class templates

- Numerical data types: `bool`, `uint`, `int` (but no pointers)
- Composite types: arrays, `structs`, `enums`
- Control constructs: `if`, `for`, `switch`, `return` (with restrictions)
- Functions (value parameters only)
- Standard library class templates: `array`, `tuple` (to facilitate parameter passing)
- Arbitrary width integer and fixed point register class templates of Algorithmic C

EXAMPLE: A SIGNED INTEGER ADDER

```
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;}
```

Verilog-RAC Translation: Designing Models

Successful equivalence checking requires that the model replicate the essential computations of the design, e.g.,

- Booth multiplication: partial products, compression tree
- Iterative division: partial quotients and remainders

However, the design may be simplified by eliminating implementation details:

- Removing optimizations
- Ignoring timing, parallelism, and cycle structure
- Exploiting RAC features

Example: A Leading Zero Counter

```
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];}

ui6 CLZ64Simple(ui64 x) {
  int i;
  for (i=63; i>=0 && !x[i]; i--) {}
  return i;}
```

Translating RAC to ACL2

Design Goals:

- Intelligibility
- Executability
- Susceptibility to formal proof

The design of the translator focuses on the first two; the third is addressed during the proof process by converting the code to a more convenient and provably equivalent form.

RAC Parser

The first phase of translation is performed by a Flex/Bison parser:

- Each RAC function is converted to an S-expression
- C++ expressions are converted to ACL2 terms
- Implicit evaluations and type conversions are replaced by explicit computations
- By-product: a more readable pseudocode version of the program

Parser Output for the Adder

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

Pseudocode Version of the Adder

```
ui32 result; ui8 sum;
for (uint i=0; i<4; i++) {
  si8 aSgnd = a[8*i+7:8*i], bSgnd = b[8*i+7:8*i];
  si9 sumSgnd = aSgnd + bSgnd;
  if (sumSgnd < -128)
    sum = -128;
  else if (sum >= 128)
    sum = 127;
  else
    sum = sumSgnd;
  result[8*i+7:8*i] = sum;}
return result;}
```

ACL2 Code Generator

Parser output is converted to ACL2 functions by an ACL2 function that addresses the different programming paradigms:

- Variable declarations and assignments are converted to bindings (`LET`, `LET*`, `MV-LET`)
- Iteration is converted to recursion
- Constant arrays (ROM) are represented as lists of values (`NTH`)
- Structs and variable arrays are represented as alists (`AG`, `AS`)
- Difference between native C and ACL2 booleans is addressed (`LOG<`,..., `IF1`)

Primitives are defined in the RTL library (`"lib/rac"`)

Translation of the Adder

```
(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)))
```

TRANSLATION OF THE LEADING ZERO COUNTER

```
(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))))))
```

ASSERTIONS

Have no logical import but are useful as documentation and in simulation.

```
(DEFUN CLZ64 (X)
  (LET ((ASSERT (IN-FUNCTION CLZ64 (LOG<> X 0)))
        (Z NIL)
        (C NIL))
    ...))
```

The binding of `ASSERT` is a macro call:

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

A failed assertion throws a run-time error:

```
RTL !>(clz64 0)

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

Example: A Simple Comparator

```
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];}
```

CORRECTNESS

Lemma *Let A and B be the signed integers represented by 64-bit vectors a and b. Let $r = compare64(a, b)$. Then $r = 1 \Leftrightarrow |B| > |A|$.*

PROOF: We examine the case $A < 0$ and $B \geq 0$; the other cases are simpler:

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

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

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

and

$$add2 = carry = 2(\sim a[63:0] \ \& \ \sim b[63:0]) + 1.$$

By Lemma 8.2,

$$add1 + add2 = \sim a[63:0] + \sim b[63:0] + 1 = 2^{64} + |A| - |B| - 1,$$

and hence

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

Thus, $r = 1 \Leftrightarrow diff[64] = 0 \Leftrightarrow |B| > |A|$. ■

Formalization of Correctness

```
(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))))
```

How can we formalize our hand-written proof?

ACL2 Translation of the Comparator

```
(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))))))
```

Formalization of the Proof

(1) Introduce constrained functions corresponding to the function arguments:

```
(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 constants corresponding to the local variables from their bindings and prove that the function maps the input constants to the output constants.

(2) IS AUTOMATIC:

```
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)))))
```

(3) Derive the required properties of the outputs:

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

(4) Lift this result by functional instantiation:

```
(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))))

(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))))))))
```

```
... 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.
```

RAC Directories in `books/projects`

Subdirectories of `books/projects/rac/`:

- `src`: `parser.*` (C++ source)
- `lisp`: `translate.lisp` (code generator),
  `internal-fns-gen.lisp` (Cuong's tool)
- `bin`: `parse` (compiled parser), `rac` (script)
- `examples`: `hello.cpp`, `imul.cpp`

Subdirectories of `books/projects/arm/`:

- `fmul`
- `fadd`
- `fdiv`
- `fsqrt`