Instruction Set Architecture Specification, Verification, and Validation using Algorithmic C and ACL2

David S. Hardin
Collins Aerospace, USA
david.hardin@collins.com

Abstract

It is common practice during Instruction Set Architecture (ISA) development to create an ISA simulator, usually in C/C++. We describe an experiment in which we implement such an ISA simulator for a derivative of a popular ISA written in a subset of Algorithmic C, to allow for the verification of binary programs targeting that ISA, as well as to aid in the validation of the ISA model via simulated execution of test programs on the model. Algorithmic C defines C++ header files that enable compilation to both hardware and software platforms, providing support for the peculiar bit widths employed, for example, in floating-point hardware design. We utilize a toolchain, due to Russinoff and O'Leary, that provides a translation from a restricted subset of Algorithmic C to the Common Lisp subset supported by the ACL2 theorem prover. This toolchain, called RAC, is documented in Russinoff's recent book on floating-point hardware verification. We create an ISA simulator in this C++ subset, use RAC to translate this simulator code to ACL2, produce small binary programs for the ISA that we use to validate the simulator, and utilize the ACL2 Codewalker decompilation-into-logic facility to prove those programs correct.

2012 ACM Subject Classification Hardware → Hardware validation

Keywords and phrases Formal verification, theorem proving, instruction set architecture, decompilation into logic

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

in his *tour de force* book on floating-point hardware verification [4], David Russinoff details a 20-year quest to provide mathematical proofs for complex arithmetic hardware utilizing an automated theorem prover, namely ACL2. Russinoff begins by presenting a formalization of modular arithmetic in standard mathematical notation, but backed by collections of ACL2 lemmas, called *books* in ACL2 parlance. The *RTL* books are named after the "Register Transfer Logic" artifacts that hardware designers produce using Hardware Description Languages (HDLs), and are designed for reasoning at the RTL level. After developing progressively more complex arithmetic circuits, Russinoff concludes his text with the complete verification of representative ARM floating-point RTL for addition, multiplication, fused multiply/add, division and square root. All of the sources and tools described in Russinoff's book are available as part of the ACL2 distribution, so the curious reader can reproduce his verification results.

The HDL that Russinoff employs in his book is a subset of Algorithmic C. Algorithmic C[1] entails a set of freely available C++ header files providing support for hardware development, including the peculiar bit widths utilized in floating-point design, and enables compilation to both hardware and software platforms. The Restricted Algorithmic C toolchain, or *RAC*, evolved from a similar toolchain for SystemC called MASC [3]. RAC accepts Algorithmic C

[1] Available at `https://www.mentor.com/hls-lp/downloads/ac-datatypes`

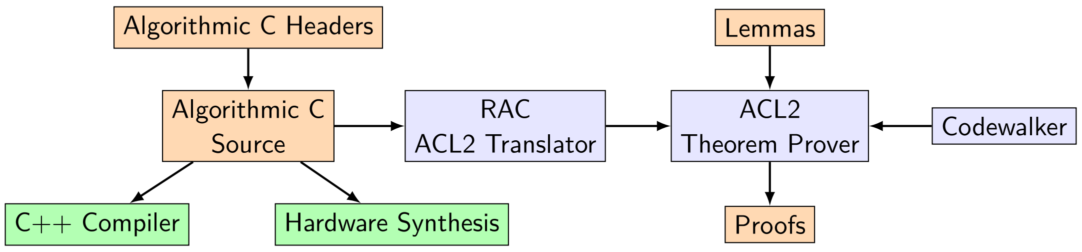

Figure 1 Restricted Algorithmic C (RAC) toolchain and Codewalker.

```
struct leg64St {
  ui10 pc;
  array<ui64, REG_SZ>regs;
  array<ui32, CMEM_SZ>cmem;
  array<ui64, DMEM_SZ>dmem;
  ui8 opcode;
  ui8 op1; [...] }
```

```
leg64St leg64step(leg64St s) {
  return do_Inst(nextInst(s)); }

leg64St leg64steps(leg64St s, uint n) {
  for (uint i=n; i>0; i--) {
    s = leg64step(s); }
  return s; }
```

Figure 2 Fragment of the LEG64 ISA state and step functions in Restricted Algorithmic C.

source, then translates it to S-expression forms acceptable to ACL2. A simplified view of the RAC toolchain is shown in Fig. 1; for a complete depiction, see Fig. V.1 of [4].

RAC imposes several restrictions on the Algorithmic C developer. The most significant of these is that all RAC function arguments must be pass-by-value, and all functions must be side-effect-free. Please refer to Chapter 15 of [4] for details.

2 ISA Specification in Restricted Algorithmic C

Our present challenge is to determine whether the RAC toolchain can be used to develop an Instruction Set Architecture (ISA) simulator, of the sort commonly crafted (usually in C/C++) during ISA design. The simulator should support proof-based verification, as well as validation by simulation. For this experiment, we have created a simple 64-bit ISA, quite similar to a commercially popular architecture, called LEG64. Figure 2 presents a fragment of the LEG64 ISA state vector (left side), and instruction-stepping functions (right side).[2]

RAC translates the state struct, including its arrays, to ACL2 records, with `ag` get function, and `as` set function. The fixed-width data types are enforced in ACL2 using the RTL `(bits y i j)` function, which returns a bit-slice of `y` between bit indices `i` and `j`. Loop bodies are translated into ACL2 recursive functions, with generated ACL2 `:measure` forms to aid in termination analysis. The resulting ACL2 functions are readable, if not pretty.

3 Validation by Simulation

To validate our RAC specification for the ISA, we can use the C++ executable for the ISA simulator, a hardware simulator, or the RAC-translated ACL2 simulator, as depicted in

[2] Complete sources available at `https://github.com/david-s-hardin/algoc-isa`

```
ui64 fact(ui64 n, ui64, acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n-1, acc * n);
  }
}
```

```
.L3:  cmp r0, #0        ; r0 == 0?
      beq .L2           ; if so, done
      mul r1, r1, r0    ; r1 <- r1*r0
      sub r0, r0, #1    ; r0 <- r0-1
      b   .L3           ; goto top
.L2:  mov r0, r1        ; r0 <- r1
      ret               ; return
```

Figure 3 Example test program for the LEG64 ISA simulator.

Fig. 1. In all cases, we validate the ISA simulator by loading the simulated code memory with binary programs compiled for our target ISA, load the data memory and registers appropriately for a given test, then step the simulator for the required number of steps, then inspecting the relevant elements of the resulting state vector. The results of running tests on the C++ executable of the ISA simulator can be checked against the translated ACL2 model as a validation check for the translation; ACL2's speed aids in this effort.

4 Verification by ACL2 Proof using Codewalker

Once the LEG64 simulator has been translated to ACL2, we can reason about the translated functions, using the RTL books, as well as other ACL2 books. We can also reason about LEG64 binary code, using J Moore's decompilation-into-logic tool, Codewalker[3]. We have previously used Codewalker to prove properties of LLVM code [1].

As an example, consider a simple tail-recursive unsigned 64-bit factorial function (left of Figure 3), compiled to yield the LEG64 code on the right of Figure 3. The input parameter, `n`, is passed in using register r0, and the tail-recursive accumulator, `acc`, is passed in r1. The function actually computes `acc * n!`, which is equal to `n!` when `acc = 1` (modulo 64 bits). This result is then copied into register r0 for return to the caller.

To begin the verification, we describe the LEG64 machine, its registers, memories, and ISA simulator functions, to Codewalker, and provide it the binary of the code of Figure 3. Codewalker then "walks" the binary code, guided by the ACL2 ISA simulator, and decompiles each basic block into a *semantic function* summarizing the effect of that block on the ISA state `s`, in the style of Magnus Myreen's Ph.D. work [2]. Codewalker also generates a per-block clock function, giving the number of instruction steps needed for a given input condition. We can then instruct Codewalker to "project" the final value for a given register out of a loop, producing a function that abstracts away the LEG64 state details. That function can then be reasoned about, and/or executed in ACL2. For our factorial example, Codewalker produces `(fn1-loop pc n acc)` for the final accumulator value, a function of the ISA simulator program counter `pc`, input value `n`, and initial accumulator value `acc`. After creating a "wrapper" function, `fn1`, we prove the correctness theorems of Figure 4, where `!` is a non-tail-recursive infinite-precision factorial function, and `expt` is exponentiation.

Finally, we prove that the execution of the loop of the factorial subroutine, loaded into code memory of the simulated LEG64, using the stepping function `leg64steps`, written in Algorithmic C and then translated by RAC to ACL2, for the number of steps given by the

[3] Codewalker is part of the ACL2 standard distribution, in `books/projects/codewalker`

```
(defthm fn1-loop-is-acc-*-n!
  (implies
   (and (natp n)(natp acc)
        (< n (expt 2 64))
        (< acc (expt 2 64)))
   (equal (fn1-loop 0 n acc)
          (bits (* acc (! n)) 63 0))))
```

```
(defun fn1 (n) (fn1-loop 0 n 1))

(defthm fn1-is-n!
  (implies
   (and (natp n) (< n (expt 2 64)))
  (equal (fn1 n)
         (bits (! n) 63 0))))
```

Figure 4 Correctness theorems for the projected loop result function from Codewalker.

```
(defthm reg-1-of-fact-loop-is-acc-*-n!
  (implies
   (and (fact-routine-loadedp s) (integerp (ag 0 (ag 'regs s)))
        (<= 1 (ag 0 (ag 'regs s))) (< (ag 0 (ag 'regs s)) (expt 2 64))
        (integerp (ag 1 (ag 'regs s))) (< (ag 1 (ag 'regs s)) (expt 2 64))
        (= (ag 'pc s) 0))
   (equal (ag 1 (ag 'regs (leg64steps s (acl2::clk-0 s))))
          (bits (* (ag 1 (ag 'regs s)) (! (ag 0 (ag 'regs s)))) 63 0))))
```

Figure 5 Correctness of the factorial loop when executed on the LEG64 ISA simulator.

Codewalker-generated clock function `clk-0`, yields the expected result, as shown in Figure 5.

5 Conclusion and Future Work

We have used Restricted Algorithmic C to create an ISA-level simulator for a representative CPU, and successfully employed the RAC toolchain, initially developed to verify floating point hardware, to translate this simulator to ACL2. We produced small binary programs for the ISA that we used to validate the simulator, then utilized the ACL2 Codewalker decompilation-into-logic facility to prove those programs correct.

Future work includes refining the RAC toolchain to allow tail recursion, support for ACL2 typed records and/or stobjs, as well as investigating the verification of hardware/software codesigns. Codewalker is an experimental tool, and while its ability to comprehend new instruction sets and automatically create semantic functions, clock functions, etc., is impressive, it is difficult to configure. Additionally Codewalker is only useful for single subroutines; future work is needed to improve usability and scalability.

References

1 David S. Hardin. Reasoning about LLVM code using Codewalker. In *Proceedings of the 13th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 192, pages 79–92. EPTCS, 2015. `doi:10.4204/EPTCS.192.7`.

2 Magnus O. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, 2009.

3 John W. O'Leary and David M. Russinoff. Modeling algorithms in SystemC and ACL2. In *Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 152, pages 145–162. EPTCS, 2014. `doi:10.4204/EPTCS.152.12`.

4 David M. Russinoff. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach.* Springer, 2018.