INSTRUCTION SET ARCHITECTURE SPECIFICATION, VERIFICATION, AND VALIDATION USING ALGORITHMIC C AND ACL2

David Hardin
Collins Aerospace

<first>.<last>@collins.com

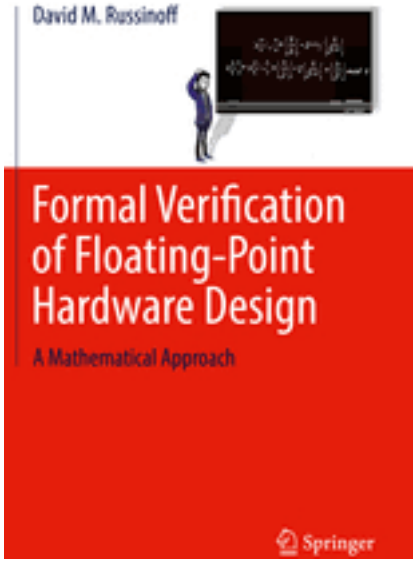
Collins Aerospace

INTRODUCTION

- Floating-point hardware verification is a signature success of formal methods
- Automated theorem proving has been used in the verification of many floating-point hardware designs, including those from:
  - AMD
  - ARM
  - Centaur (x86-compatible)
  - Intel
  - Oracle (SPARC)
- In this talk, we will describe an experiment in the use of a particular approach to floating-point verification to create a performant Instruction Set Architecture (ISA) simulator, of the sort commonly written during ISA development
  - Implementable in either software or (FPGA-based) hardware
  - Supports proofs of correctness for programs targeting that ISA

THE RUSSINOFF-O'LEARY APPROACH TO FLOATING POINT HARDWARE VERIFICATION

- The floating-point hardware verification approach we employ was developed by David Russinoff and John O'Leary, while both were at Intel (ACL2 Workshop 2014)
  - The approach was initially based on SystemC, and was called MASC
  - Russinoff changed the source language from SystemC to Algorithmic C after he moved to ARM, made several enhancements, and renamed the system RAC (Restricted Algoritmic C)
- RAC is extensively documented in Russinoff's 2018 book, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, wherein RAC is applied to the verification of realistic ARM floating-point designs
  - RAC, and the verifications described in the book, are all available as part of the standard ACL2 distribution



ALGORITHMIC C

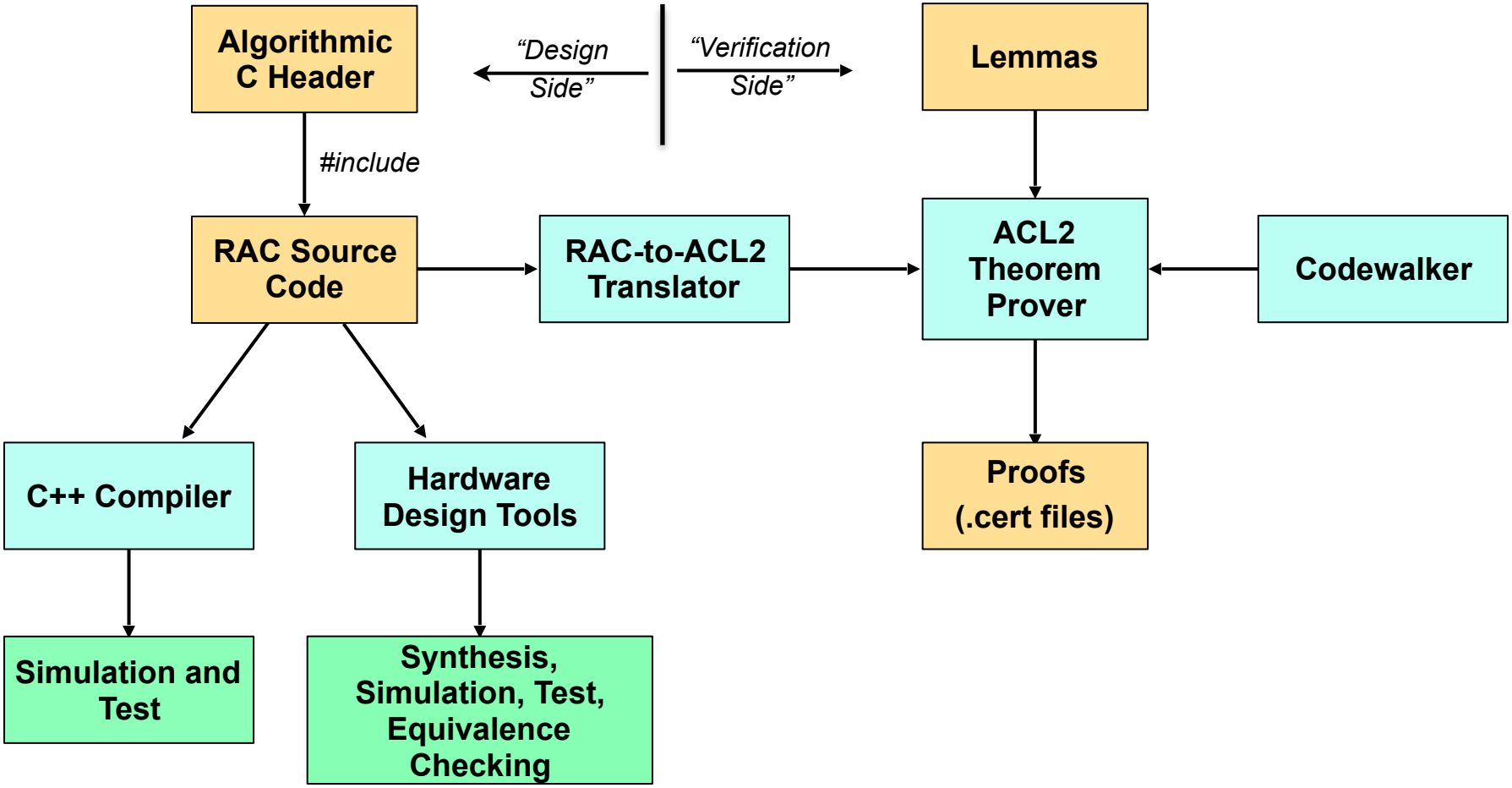
- The Algorithmic C datatypes “provide a basis for writing bit-accurate algorithms to be synthesized into hardware”
- The Algorithmic C datatypes are defined via an open source C++ header file that users can #include in their designs
  - No runtime library required
- Example use:
  - `typedef ac_int<112,false> ui112;`

  declares an unsigned 112-bit type used in floating-point hardware datapaths
- Supported by Mentor hardware synthesis tools
- Further information is available at **`https://hlslibs.org`**

RESTRICTED ALGORITHMIC C (RAC)

- Restricted Algortihmic C defines a C subset that promotes proof, hardware synthesis, and simulation
- Use case: A hardware developer expresses hardware functionality in RAC, which is then translated into a theorem prover language used by the verification expert
- RAC encompasses many of the restrictions common in “high-assurance” C, such as no function pointers, disallowing recursion, etc.
  - RAC also disallows all pointers, as well as function side-effects
  - Certain control constructs (e.g., breaking out of a `for` loop) are disallowed
- Provides support for bit slices and multiple-value return
- For more information on RAC, please consult Chapter 15 of Russinoff’s book

TOOLCHAIN FOR THE CURRENT WORK



ACL2

- ACL2 is “A Computational Logic for Applicative Common Lisp”, developed by Matt Kaufmann and J Moore
  - ACL2 is a winner of the ACM Software Systems Award
- ACL2 developers model their system as Common Lisp functions, then state and prove theorems about their model using ACL2’s highly automated proof heuristics
  - These functions and theorems are gathered into libraries, called books, which are proved once, then utilized many times
- ACL2 has been used in many large academic and industrial verification efforts:
  - Floating-point unit verification (AMD, ARM, Centaur, Intel, Oracle)
  - AAMP7 separation kernel microcode and Green Hills INTEGRITY-178B kernel information flow verification (Collins Aerospace)
  - Used to certify the correctness of the “world’s largest math proofs” (Heule)
    - Proofs are discovered by massively parallel SAT solving

BRIDGING THE DESIGN/VERIFICATION GULF

- A key issue in the formal verification of engineering artifacts is the gulf between the sorts of programs that can be readily specified and verified, and the sorts of programs that "real-world" developers actually write:

Formal Verification "Comfort Zone"	*Real World*
Functional Programming	**Imperative Programming**
Total, terminating functions	**Partial, potentially non-terminating functions**
Non-tail-recursive functions	**Loops**
Okasaki-style pure functional algebraic data types	**Structs and Arrays**
Infinite-precision Integers	**Modular Integers**
Linear Arithmetic	**Linear and non-linear arithmetic**

- The Russinoff-O'Leary toolchain, in combination with the ACL2 theorem prover, does an admirable job of bridging these two worlds

RAC-TO-ACL2 TRANSLATOR

- Translates loops into tail-recursive functions
- Generates ACL2 “measures” to aid in function termination proofs
  - All functions to be admitted into ACL2 must be proved to terminate
  - Termination proofs are conducted mostly automatically by ACL2, with hints provided by the measure annotations (if needed)
- Translates fixed-width integer operations into functions defined in Russinoff’s “RTL” (Register Transfer Language) ACL2 books
  - Ensures that translated operations are “wrapped” with an appropriate RTL bit-width coercion operator so as to accurately translate modular integer arithmetic
  - RTL is described in detail in Part I of Russinoff’s book

RAC-TO-ACL2 TRANSLATOR (CONT'D.)

- Converts assignments to Lisp let-bindings
- Converts struct/array reads/writes to ACL2 record gets/sets, for which get-over-set, set-over-get, etc. theorems are available
- In addition, ACL2's powerful arithmetic capability allows it to reason about non-linear arithmetic expressions
- ACL2 also features a very capable induction scheme generator
  - ACL2 automatically finds suitable induction schemes for the vast majority of inductive proof attempts, including hybrid schemes
- …allowing us to reason about real-world designs expressed in RAC

CODEWALKER

- A new facility as of ACL2 7.0 (January 2015), due to J Moore
- Performs “decompilation into logic” of a machine-code program to a series of “semantic functions” that summarize the program’s effect on machine state
- Works with an instruction set description written in the usual ACL2 “machine interpreter” style, as earlier described
- Produces proofs that the generated semantic functions are correct
- Inspired by Magnus Myreen’s Ph.D. thesis (2008)
- Myreen’s decompiler utilizes the HOL4 theorem prover
- Three main Codewalker API’s utilized in our work:
  - **`def-model-api`**
  - **`def-semantics`**
  - **`def-projection`**

CODEWALKER (CONT'D.)

- **`def-model-api`** instructs Codewalker on the basics of the machine model:
  - Name of the machine interpreter, machine types, how to access/update elements of the machine state
- **`def-semantics`** is the main workhorse of Codewalker. It creates "semantic functions" summarizing the actions of machine code segments on machine state
  - Semantic functions are generated by symbolic simulation of the machine previously described to **`def-model-api`**.
  - The machine interpreter function is not part of the generated semantic functions
  - **`def-semantics`** also generates a 'clock' function providing the number of instruction steps the interpreter must execute to cover a given program counter range
  - Finally, **`def-semantics`** generates a theorem that the semantic function correctly summarizes the changes to the machine state produced by executing the machine interpreter for the number of steps indicated by the generated clock function.
- **`def-projection`** takes a semantic function generated by **`def-semantics`**, and 'projects out' a function that computes the final value of some machine state component

THE EXPERIMENT

- Develop an Instruction Set Architecture (ISA) simulator for a representative ISA in RAC
  - An ISA simulator is routinely developed (mostly in C/C++) during development
  - Provides a vehicle for ISA experimentation, compilation toolchain development, etc.
  - Much faster than RTL-level simulators
- Translate the ISA simulator to ACL2 using the RAC toolchain, and determine whether the translated ISA interpreter can serve as input to Codewalker
- Perform machine code correctness proofs for programs written in our ISA using the decompilation-into-logic facilities of Codewalker

THE ISA: LEG64

- LEG64 is a 64-bit ISA similar to a popular ISA named for an appendage
- RISC-style, three-address, 32-bit fixed-format instructions
- 64-bit datapaths
- Harvard architecture
- Simple, but useful instruction set that allows us to compile C code for the LEG64 target

LEG64 MACHINE STATE IN RAC

```
struct leg64St {
  ui10 pc;
  ui12 sp;
  array<ui64, REG_SZ>regs;
  array<ui64, DMEM_SZ>dmem;
  array<ui32, CMEM_SZ>cmem;
  ui8 opcode;                  // Current decoded instruction
  ui8 op1;
  ui8 op2;
  ui8 op3;
  ui1 C;                       // Carry Status Flag
  ui1 N;                       // Negative Status Flag
  ui1 Z;                       // Zero Status Flag
  ui1 V;                       // oVerflow Status Flag
};
```

LEG64 ISA SIMULATOR CODE IN RAC

~800 LINES OF C CODE

```
leg64St do_ADD(leg64St s) {
  s.regs[s.op1] = s.regs[s.op2] + s.regs[s.op3];
  return s;  }

// Instruction selector
leg64St do_Inst(leg64St s) {
  ui8 opc = s.opcode;

  if (opc == NOP) {
    return do_NOP(s);
  …} else if (opc == ADD) {
     return do_ADD(s);…}

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

LEG64 ISA SIMULATOR TRANSLATED TO ACL2

~800 LINES OF ACL2

```
(DEFUN DO_ADD (S)
  (AS 'REGS
      (AS (AG 'OP1 S)
          (BITS (+ (AG (AG 'OP2 S) (AG 'REGS S))
                   (AG (AG 'OP3 S) (AG 'REGS S)))
                63 0)
          (AG 'REGS S)) S))

;; Instruction selector
(DEFUN DO_INST (S)
 (LET ((OPC (AG 'OPCODE S)))
  … (IF1 (LOG= OPC 3)
      (DO_ADD S)…)

;; Instruction stepper
(DEFUN LEG64STEP (S)
  (DO_INST (NEXTINST S)))
```

Note: '(BITS N X Y)' returns the bit slice of N from bit X to bit Y, inclusive

Note: 'AG' and 'AS' are ACL2 untyped record get and set, respectively

EXAMPLE LEG64 PROGRAM: FACTORIAL

```
unsigned long fact(unsigned long num, unsigned long acc) {
    if (num < 2) {
        return acc;
    } else {
        return fact(num - 1, acc * num);
    }
}
```

```
;; LEG64 Assembly Code

.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
```

LEG64 FACTORIAL PROGRAM CODEWALKER CORRECTNESS PROOF

```
;; Mathematical factorial
(defun ! (n)
  (if (zp n) 1 (* n (! (- n 1)))))

;; Final correctness theorem: fact(num, acc) == acc * num!
(defthm reg-1-of-program1-is-acc-*-n!
  (implies
    (and
      (fact-routine-loadedp s)
      (integerp (ag 0 (ag `regs s))) (< 0 (ag 0 (ag 'regs s)))
      (< (ag 0 (ag 'regs s)) (expt 2 64))
      (integerp (ag 1 (ag 'regs s))) (<= 0 (ag 1 (ag 'regs s)))
      (< (ag 1 (ag 'regs s)) (expt 2 64))
      (= (ag 'pc s) 0))
    (= (ag 1 (ag 'regs (leg64stepn s (acl2::clk-0 s))))
       (bits (* (ag 1 (ag 'regs s)) (! (ag 0 (ag 'regs s)))) 63 0))))
```

CONCLUDING REMARKS

- Floating-point hardware verification tools and techniques can be employed for more general hardware/software coassurance tasks
  - The Russinoff-O'Leary RAC toolchain is a nice choice for writing ISA simulators
  - Also exploring other uses, e.g. very high-assurance data structures
- We can use the ISA simulator written in RAC to prove properties of ISA programs using a decompilation-into-logic technique
  - Subject to scalability issues
- The RAC-to-ACL2 translator is untrusted code; it would be a worthwhile project to give it a formal foundation
- The use of ACL2 typed, rather than untyped, records may improve proof automation
  - Use of ACL2 stobjs would increase simulation performance, but complicate proofs
- Future work includes generation of verified software components that interface to verified hardware components, both generated using this technique