Robust, End-to-end Correctness Proofs of Industrial Divide and Square Root RTL Designs

Sol Swords
Previous Institution: Intel Corporation
Current Institution: Arm, Inc.
https://orcid.org/0000-0002-5958-9580
sol.swords@arm.com

Cuong Chau
Intel Corporation
Vietnam
cuong.chau@intel.com

***Abstract*—Hardware implementations of divide and square root operations are difficult and high-value targets for formal verification. We describe an approach using the ACL2 theorem prover that has resulted in robust, end-to-end correctness proofs for highly optimized industrial implementations of such operations. Using this approach, we developed initial proofs for divide and square root operations in less than three person-months each. We subsequently proved the correctness of all operations on a floating point divide/square root design and an integer divider implementing up to 128-by-64-bit divides, both highly optimized industrial implementations. These proofs run in minutes per operation and have been straightforward to maintain against design changes. This methodology allows lemmas to be proved about portions of the hardware model, and these lemmas seamlessly composed to complete a top-level proof that the whole operation runs correctly. This decomposition allows fully automatic proof methods to be applied to the portions of the hardware model that capacity allows, and the composition of these portions is amenable to rewriting and other traditional interactive theorem proving methods.**

***Index Terms*—decomposition proof methodology, theorem proving, automated reasoning, equivalence checking, divide and square root verification, hardware verification**

I. Introduction

Formal verification of complex datapath operations in modern hardware designs, such as division and square root, is still not amenable to fully automatic methods. Lack of support for ad-hoc proof methods in automatic tools prevents them from scaling to complex operations. In order to verify several divide and square root operations on a highly optimized industrial register-transfer level (RTL) design, we have developed a methodology based on the ACL2/SVTV hardware verification framework [1], a set of open-source libraries built on the ACL2 theorem prover. This methodology allows seamless composition of proofs about separate parts of the hardware operation, which may be done independently with different automated tools and subsequently composed into an ACL2 theorem stating the correctness of the full operation. We have used this methodology to verify several integer divide and floating-point (FP) divide and square root operations on state-of-the-art, highly optimized, high-radix designs at Intel. The operations verified include up to 128-by-64-bit integer divide and single-instruction/multiple-data (SIMD) IEEE-compliant single and double-precision floating point operations. These proofs run in minutes per operation, easily allowing for nightly regression testing and offering no impediment to the debugging loop with designers. The initial development of the proofs for the first version of single-precision divide and square root operations took less than a quarter each for a single expert user. Proofs for subsequent operations and proof modifications due to major design changes both requiring much less development time—up to 4 weeks including design bug fix iterations. All such proofs produce a final theorem stating that when the given operation runs on our formal model of the RTL design, it produces outputs equal to those produced by our ACL2 specification function for the operation. To our knowledge, these are the most comprehensive correctness proofs of divide/square root implementations that have been done within a single formal framework.

Key to our methodology is the ability to reason about isolated portions of the hardware model's behavior. For example, in an iterative division algorithm we might express the next partial quotient Q_{i+1} and remainder R_{i+1} in terms of the current partial quotient Q_i and remainder R_i, separately from how Q_i and R_i are computed from primary inputs. We need to do this in a manner that truly isolates the target logic. In this example, we do not want the automated proof tools used in this proof to consider the fanin cones for Q_i and R_i—they should be treated as primary inputs instead. We accomplish this by *overriding* these signals at the appropriate time steps, setting these signal values to free variables disconnected from the respective fanin cones. The ACL2/SVTV framework offers an automated and verified method of using theorems about circuits with overrides to prove analogous theorems without overrides. Multiple such override-free theorems can then be seamlessly composed together, resulting in a top-level theorem pertaining to an override-free run of the design.

The decomposition facility and the integration of all parts of the proof in ACL2 allows a choice of automated tools to be applied to each part of the problem. For most portions of the design, we use the bit-blasting rewriter FGL [2], [3] to reduce conjectures to Boolean formulas that are then proved with an external SAT solver or with combinational equivalence checking methods. For multipliers, adder trees, and similar arithmetic blocks, we use VeSCMul [4]–[6], a proof engine that transforms such arithmetic functions into a normal form

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00031

in terms of sum/carry functions. Both VeSCMul and FGL are *verified clause processors* [7]—reasoning engines defined and proved correct in ACL2 that are then allowed to be used in ACL2 proofs.

A particular aid to our floating-point square root verification effort is a tool for finding and proving lower and upper bounds for arithmetic terms [8], [9]. It is not always straightforward to determine and prove bound invariants in iterative approximation algorithms, especially when implemented in highly optimized industrial designs. As demonstrated in our square root verification, we instead used our bound-finding tool to establish pragmatic bounds that were sufficient for proving the convergence of the square root algorithm. We successfully applied this tool to find and verify bounds for the partial root and remainder in each iteration in our proofs.

We compare our framework to other related work in Section II. Section III describes our hardware verification environment and the steps to build a formal hardware model. Section IV goes through an example proof to show how our decomposition methodology works. Section V describes the series of steps in proving the correctness of a divide/square root hardware operation. Finally, Section VI discusses the results achieved by this methodology.

II. Related Work

The present work is based on an evolution of the datapath verification framework developed at Centaur Technology, Inc. [2], [10], [11], eventually encompassing the VL SystemVerilog toolsuite, the GL and FGL bit-blasting engines, and the SVEX hardware modeling framework. Rager et al. applied a previous version of that framework to their formal verification of divide and square root implemented in the SPARC microprocessor [12]. Decomposition supported by that framework was done with equivalence checking techniques that are less scalable and reliable than our method: the override transparency technique described in Section IV allows us to check once and for all that overrides will behave as expected, rather than relying on multiple ad-hoc equivalence checks.

Another approach to arithmetic datapath verification also uses ACL2 to verify RTL designs through intermediate models written in a higher-level language called RAC (Restricted Algorithmic C) [13], [14]. The correctness of an RTL design is established by a two-step verification process that combines sequential logic equivalence checking with interactive theorem proving. First, an intermediate RAC model is hand-coded and equivalence checked against the RTL implementation using a commercial C-to-RTL equivalence checking tool. The RAC model is then automatically translated to the logic of ACL2 using a shallow embedding scheme [15]. This translated model can then be verified in ACL2 against a specification using ACL2's rewriter and arithmetic libraries. In contrast, our methodology models the RTL behavior within ACL2 directly. Therefore, we do not require a hand-coded intermediate model written in a third language. While we also rely on equivalence checking to prove the RTL model's correspondence with an ACL2 specification (either the full operation specification or a lower-level algorithmic specification that must itself be proven correct), this equivalence check occurs within ACL2 via the FGL proof procedure and does not require a commercial C-to-RTL equivalence checking tool.

The reFLect/Goaled framework [16], [17] (internal to Intel) uses relational symbolic trajectory evaluation (rSTE) as its proof engine for conjectures amenable to fully automatic proof. A successful run of rSTE allows Goaled to use an axiom similar to the override-free versions of theorems we generate (such as our `boothpipe-sum-correct` in Section IV). Such axioms are the method for obtaining facts about the design in Goaled; there is no mathematical model of the hardware design in Goaled independent of rSTE. We describe the process of modeling the RTL under verification in Section III-A.

III. Verification Environment

The hardware verification environment we use within ACL2 is based around an expression language called SVEX (for Symbolic Vector Expressions). An SVEX may be either a constant, a variable, or a function applied to subexpressions. There are currently 48 such functions, including, e.g., bitwise AND, concatenate, plus, multiply, etc., designed mainly to replicate SystemVerilog operators. The values of SVEX expressions are called 4vecs, and are vectors of 4-valued "bits" of 1, 0, X, Z, following SystemVerilog semantics. Such a vector has finitely many such bits, but the most-significant of these is considered a sign bit as in two's-complement arithmetic and imagined to extend out to infinity. An SVEX expression may be evaluated to a 4vec value given an environment, which is a mapping from variables to 4vec values.

The formal model of a hardware design is ultimately expressed as a finite state machine (FSM) with next-state functions and output values given as SVEX expressions. The FSM representation is a pair of two elements *values* and *nextstate*, each of which is a mapping from variable names to SVEX expressions. The variables used in all such expressions are primary input and previous state signals. The keys bound in the nextstate mapping are the previous state signals, where the expression bound to each such signal gives the update function for that state signal. The keys bound in the values mapping are the driven signals of the design (that is, outputs and internal signals but not primary inputs or previous-states), and the expression bound to each one gives its current value.

With this formal model, we can run concrete simulations of the hardware design and perform various design exploration and debugging tasks. We have a tool to trace root causes of unexpected values and can also write out waveforms from simulations. We can also prove functional properties of the design. We primarily use two engines for these proofs, both of which are verified in ACL2. First, FGL [2], [3] is a bit-blasting rewriter which translates the expressions and the properties to be proved to a Boolean formula (and/inverter graph) and proves it using SAT or combinational equivalence checking methods. Second, for multipliers and nests of adders, we usually use VeSCMul [4], [6], a specialized rewriter implementing

the *s-c-rewriting* technique, which efficiently converts such arithmetic formulas into a normal form of nests of sum and carry operators.

As with most fully automatic proof procedures, it is crucial to limit the scope of a proof. Decomposition proofs rely on the efficiency of proving lemmas about portions of the design in isolation, and therefore it is important not to involve the fanin cones of the input signals of that portion. That is, for such proofs we want to ignore the values to which these internal signals are driven by surrounding logic, and instead set (override) these with values of our choice. To allow this, before creating the FSM we augment the design's netlist with override muxes, described below, that let us decide on a signal by signal basis whether to let the signal operate as normal or to override its value.

A basic property that we prove about this modified design is *override transparency*: *If we override a signal to the value that it is already driven to, this override does not result in any change to signal values.* This property lets us take theorems proved with overrides, each saying something like "if we override signal s with value v at time t, then…", and derive from them theorems about a run without overrides, of the form "if the sampled value of signal s at time t is v, then…". These latter theorems can be combined with other theorems about the design, since they do not conflict in their assumptions about what signals are overridden.

We also rely on the *X-monotonicity* of SVEX expressions, relating to the information ordering in which a value of X signifies that a bit's value is unknown whereas 1, 0, or Z are known values. If an expression is monotonic, then if you replace an X bit with 1, 0, or Z in the environment under which that expression is evaluated, each bit of its evaluation either does not change or changes from X to 1, 0, or Z. This allows us to generalize results from symbolic simulations with X values: if we prove that the result of a symbolic simulation is non-X under an environment where some inputs are bound to X, then this result is identical if we change some of these input bindings from X to other values. We generally do proofs by symbolic simulation with any don't-care inputs set to X values; the results can then be generalized to theorems in which we make no assumptions about these don't-care inputs.

A. Model Build

We now give an overview of how an FSM is obtained from a hardware design, briefly describing each step in the process.

1) Parse: SystemVerilog files are parsed into an abstract syntax tree format called a VL design.

2) Elaborate: The parsed VL design is processed to resolve parameters, function definitions, types/vector widths, etc.

3) Translate: The elaborated VL design is translated to a simpler hierarchical format, called an SVEX design, with many complex SystemVerilog features removed and expressions, procedural blocks, etc. all translated to SVEX expressions. The SVEX design hierarchy consists of modules which only contain signal declarations, submodule instances, assignments, and aliases between signals (which mainly occur due to port connections). Signals in expressions may be referenced with delay values to express sequential logic; here we use s' to denote the previous time step's value of s.

4) Flatten: The SVEX design hierarchy is expanded into a flat list of assignments and aliases. E.g., an assignment `a = b` within a module with two instances `i` and `j` would become two assignments, `i.a = i.b` and `j.a = j.b`. A port connection between a signal `c` in the top-level module and a port signal `d` in an instance `i` within that module would become an alias, `c` $\leftrightarrow$ `i.d`.

5) Normalize Netlist: Canonical names for all signals are chosen from among their aliases, and the mapping of each signal to its canonical name is applied to the flattened assignments. E.g., with an alias `c` $\leftrightarrow$ `i.d` we would replace all occurrences of `i.d` with `c`. Additionally, assignments are combined and split apart so that every driven signal has exactly one assignment. For example, assignments `{a[3:0], b[2:0]} = c[6:0], a[5:4] = d[1:0]` would become `a[5:0] = {d[1:0], c[6:3]}, b[2:0] = c[2:0]`.

6) X-monotonify: The normalized netlist may contain non-X-monotonic operations (such as SystemVerilog's === comparison operator). We replace these constructs with monotonic ones and prove that the transformation is conservative. That is, the new design may sometimes produce X values where the old design did not, but otherwise they are equivalent.

7) Insert Override Muxes: For some subset of the driven signals of the design that may be involved in a cutpoint for decomposition, we insert *override muxes*. That is, for each such signal s, we replace references to s with expressions $s_t \mathbin{\vec{?}} s_v \ : \ s$. Here s_t and s_v are new primary input variables, respectively the override test and override value of s, and the ternary $\vec{?}$: operator is if-then-else on corresponding bits of the test and branches. Later when we have our FSM model, this will allow us to override certain bits of s in an FSM run by setting s_t to the bitmask of the bits we want to override and s_v to the values with which we want to override those bits. The result of this is our final netlist. This step is verified to produce no change in the logic when no signals are overridden (all override tests s_t are false), and the override transparency property relates the behavior when there are overridden signals to the case where there are not.

8) Zero-delay compose: The final netlist is composed with itself to obtain new expressions that no longer reference driven signals (any appearing as a key in the key-value pairs of the netlist), only primary inputs and delayed signals. For example if our netlist contains assignments `a = f(b, c), b = g(c, d)` where `c` and `d` are non-driven signals, self-composition would yield `a = f(g(c, d), c), b = g(c, d)`. Driven signals that cannot be eliminated from these expressions by self-composition are replaced with X; this conservatively deals with signals that are overconstrained (e.g. `a = ~a`) or unconstrained (e.g. `a = a`). This step results in formulas that are provably conservative relative to any fixpoint composition of the netlist.

9) Create FSM: The result of the zero-delay compose step is the values field of the FSM. The nextstate field is derived by

mapping all referenced delay variables s' to the value bound to key s, or to s itself if s is not a driven signal.

This FSM is the main formal model of a hardware design. However, in the datapath domain, there is usually a particular stimulus and sampling pattern of interest. For example, supplying the valid, opcode, and data input signals at the first cycle and sampling the data output at the second cycle might be the stimulus/sampling pattern of interest on a simple latency-1 arithmetic unit. Usually instead of doing symbolic simulation proofs directly on the FSM, we create a set of SVEX expressions giving the result of applying the symbolic stimulus pattern to the FSM and sampling the signals of interest at the appropriate times, and do proofs with reference to this object. We call such an object an *SVTV* (for SVEX Test Vector). The evaluations of the expressions making up this SVTV are provably the same as the corresponding sampled values from the FSM run on the stimulus pattern.

The stimulus pattern used in creating an SVTV may include overrides, either conditional or fixed. A proof with cutpoints may be done with a single SVTV where the cutpoint signals have both conditional overrides and sampled outputs at a particular time step.

IV. Decomposition Methodology Example

We illustrate our decomposition methodology using a simple example from the public ACL2 community books repository [18]. This proof shows the correctness of a 16x16-bit multiplier with radix-4 Booth encoding, using a single cutpoint at the partial product vector. We use a multiplier for this example because a divide or square root would require many more steps to be described, adding complexity to the presentation that is not relevant to the illustration of the decomposition method. However, an example showing the verification of a radix-4 SRT 32-bit unsigned integer divider is also publicly available [19].

The multiplier is proved correct using a single cutpoint at the partial product vector. Thus, the proof is split into two parts: correctness of the Booth encoding resulting in the partial products, and the summation of the partial products to compute the final result. The specifications for these two parts are proved to compose to a multiply (Figure 1); we do not cover the details of this proof in this paper as it is a standard ACL2 proof using conventional interactive theorem proving methods.

```
(defthm sum-pps-of-pps-spec
  (equal (sum-pps 8 16 0 (pps-spec 8 16 0 a b))
         (* (logext 16 b) (logext 16 a))))
```

Fig. 1. Specification-level composition theorem

Here a and b are the data inputs to be multiplied, `pps-spec` is the specification for the Booth encoder, `sum-pps` is the specification for the summation tree, and `logext` is the sign extension operator. The eight partial products (produced by `pps-spec` and consumed by `sum-pps`) are packed into a single vector. This says that the Booth encoding followed by partial product summation produces the product of the low 16 bits (sign-extended) of the two inputs.

```
(defsvtv$ boothpipe-run
  :design *boothpipe*
  :cycle-phases (list (sv::make-svtv-cyclephase
                       :constants '(("clk" . 0))
                       :inputs-free t
                       :outputs-captured t)
                      (sv::make-svtv-cyclephase
                       :constants '(("clk" . 1))))
  :stages ((:label c0
            :inputs    (("en"           en :hold t)
                        ("a"            a)
                        ("b"            b)))
           (:label c1)
           (:label c2
            :overrides (("pps"          pps
                         :cond          pps-ovr
                         :output        pps)))
           (:label c3
            :outputs   (("o"            o)))))
```

Fig. 2. Boothpipe SVTV declaration

It then remains to show that the hardware design implements `pps-spec` and `sum-pps`. To begin, we read, parse, and translate the design to the hierarchical SVEX form, and following the rest of the steps of Subsection III-A, creating an FSM and an SVTV. The form to introduce the SVTV is listed in Figure 2. This describes the clock cycle, specifying that inputs are to be supplied and outputs sampled on the clock-0 phase of each cycle, then describes the stimulus and sampling pattern: we supply the inputs and clock enable signal at cycle 0 (labeled as `c0` in Figure 2), then at cycle 2 both sample as output and conditionally override the partial products, then at cycle 3 sample the output. The `:hold` keyword, set to `t` in Figure 2, indicates that the associated assigned value, which is `en`, will be held until end (or until the signal ``'en'' is set again, which is not the case in this example).

The SVTV resulting from this is an object containing SVEX expressions for the two outputs, `pps` and `o`, in terms of the inputs `en`, `a`, `b`, `pps`, and `pps-ovr`. (Note that the input and output namespaces are distinct, allowing both an input and an output to be named `pps`.) These expressions can be evaluated on concrete values, e.g., if we set `en` to 1, `a` to 3, `b` to 4, and `pps-ovr` to 0 — meaning the partial product vector is not overridden, so the setting of `pps` is irrelevant — then the expression associated with output `o` evaluates to 12. The intermediate value `pps` represents the eight partial products packed into a single vector.

To verify this design, we prove two theorems automatically using FGL, one about the Booth encoding step and one about the partial product summation. These are listed in Figure 3. The SVTV representing the hardware model behavior is evaluated using the `svtv-run` function, which evaluates the expressions in the SVTV under the given assignment (environment) mapping input variables to their values. The `svtv-run` call returns an output environment, and the signal of interest is looked up from this environment using `svex-env-lookup`. Each of these theorems is fairly limited due to the stringent assumptions about the environments under which the SVTV is evaluated: they are exactly a list of `cons`

pairs of a certain length, with keys in a certain order, etc. Unfortunately, the assumptions made by the two theorems are not compatible with each other, so the theorems cannot be composed directly. In particular, in the second theorem, the evaluation environment sets `pps-ovr` to -1 (the vector containing all 1-bits), which means that the signal ``pps'' is overridden to the value of the variable `pps`; this signal is not overridden in the first theorem.

```
(fgl::def-fgl-thm boothpipe-pp-correct-lemma
 (implies
  (and (unsigned-byte-p 16 a)
       (unsigned-byte-p 16 b))
  (let* ((env (list (cons 'en 1)
                    (cons 'a a)
                    (cons 'b b)))
         (run (svtv-run (boothpipe-run) env
                        :include '(pps)))
         (pps (svex-env-lookup 'pps run)))
    (equal pps (pps-spec 8 16 0 a b)))))

(fgl::def-fgl-thm boothpipe-sum-correct-lemma
 (implies
  (unsigned-byte-p 144 pps)
  (let* ((env (list (cons 'en 1)
                    (cons 'pps-ovr -1)
                    (cons 'pps pps)))
         (run (svtv-run (boothpipe-run) env
                        :include '(o)))
         (o (svex-env-lookup 'o run)))
    (equal o (loghead 32 (sum-pps 8 16 0 pps))))))
```

Fig. 3. Partial product and summation tree lemmas

However, due to the override transparency and X-monotonicity properties of the design, we can derive from these theorems more general ones, shown in Figure 4, that can be composed together. These theorems have two main differences. First, the environments are not fixed in shape, but instead are free variables with assumptions about the bindings of certain keys. Second, the ``pps'' signal that was overridden in the second lemma is no longer assumed to be overridden. In the lemma, the override value for that signal was also passed to the `sum-pps` specification function; in the generalized theorem, this is replaced by the sampled value of the ``pps'' signal from the outputs of the SVTV run. In fact, the final hypotheses of both theorems (`svtv-override-triplemaplist-envs-match`) say that no signals are overridden in the environment (that is, none of the override test variables of the SVTV are set with any 1-bits). This generalized theorem can be derived from the lemma because we have the override transparency property, namely that we get the same results if we do not override ``pps'' as we do if we override it with its own sampled value.

These theorems can be composed to produce the top-level theorem of Figure 5. The hypotheses of the final theorem are the same as those of the partial product correctness theorem. This then says that the partial products sampled from the run equal `pps-spec`. This (by another theorem not shown) is a 144-bit unsigned value, which suffices to allow the summation correctness theorem to apply. This shows that the result equals the zero-extension (`loghead`) at 32 bits

```
(defthm boothpipe-pp-correct
 (let* ((a   (svex-env-lookup 'a env))
        (b   (svex-env-lookup 'b env))
        (en  (svex-env-lookup 'en env))
        (run (svtv-run (boothpipe-run) env))
        (pps (svex-env-lookup 'pps run)))
   (implies
    (and (equal en 1)
         (unsigned-byte-p 16 a)
         (unsigned-byte-p 16 b)
         (svtv-override-triplemaplist-envs-match
          (boothpipe-run-triplemaplist) env 'nil))
    (equal pps (pps-spec 8 16 0 a b)))))

(defthm boothpipe-sum-correct
 (let* ((en (svex-env-lookup 'en env))
        (run (svtv-run (boothpipe-run) env))
        (pps (svex-env-lookup 'pps run))
        (o   (svex-env-lookup 'o run)))
   (implies
     (and (equal en 1)
          (unsigned-byte-p 144 pps)
          (svtv-override-triplemaplist-envs-match
           (boothpipe-run-triplemaplist) env 'nil))
     (equal o (loghead 32 (sum-pps 8 16 0 pps))))))
```

Fig. 4. Generalized partial product and summation tree theorems

of the `sum-pps` of the `pps-spec`, which as previously shown in `sum-pps-of-pps-spec` (Figure 1) equals the multiplication of the sign-extensions of the inputs.

```
(defthm boothpipe-correct
 (let* ((a   (svex-env-lookup 'a env))
        (b   (svex-env-lookup 'b env))
        (en  (svex-env-lookup 'en env))
        (run (svtv-run (boothpipe-run) env))
        (o   (svex-env-lookup 'o run)))
   (implies
     (and (equal en 1)
          (unsigned-byte-p 16 a)
          (unsigned-byte-p 16 b)
          (svtv-override-triplemaplist-envs-match
           (boothpipe-run-triplemaplist)
           env 'nil))
     (equal o (loghead 32 (* (logext 16 a)
                             (logext 16 b)))))))
```

Fig. 5. Top-level correctness theorem for the 16-bit multiplier

Manual derivation of the generalized theorems from the automatically-proved lemmas would be tedious and repetitive, but this is instead automated by a macro called `def-svtv-generalized-thm`, which first proves the lemma and then generalizes it. This macro also supports setting defaults for its arguments, so the theorem forms tend to be more concise than writing the theorem directly. The forms for the two decomposition theorems and the final theorem are shown in Figure 6, after the settings for default options. The final theorem uses the `:no-lemmas` option to prove the generalized theorem directly via rewriting rather than by proving a lemma with FGL first.

The decomposition methodology illustrated in this example makes it easy to prove lemmas about isolated parts of the design and then put these together into a top-level proof in a logically sound manner. While the multiplier in this example

```
(local
 (progn
  (table svtv-generalized-thm-defaults
         :svtv 'boothpipe-run)
  (table svtv-generalized-thm-defaults
         :unsigned-byte-hyps t)
  (table svtv-generalized-thm-defaults
         :input-var-bindings '((en 1)))))

(def-svtv-generalized-thm boothpipe-pp-correct
    :input-vars (a b)
    :output-vars (pps)
    :concl (equal pps (pps-spec 8 16 0 a b)))

(def-svtv-generalized-thm boothpipe-sum-correct
  :override-vars (pps)
  :output-vars (o)
  :concl (equal o (loghead 32 (sum-pps 8 16 0 pps))))

(def-svtv-generalized-thm boothpipe-correct
    :input-vars (a b)
    :output-vars (o)
    :concl (equal o (loghead 32 (* (logext 16 a)
                                   (logext 16 b))))
    :no-lemmas t)
```

Fig. 6. `def-svtv-generalized-thm` forms for main theorems

could be proved without decomposition, the rest of this paper focuses on divide and square root implementations, which at the current state of the art cannot be proved correct by top-level, fully automated methods. This newly developed decomposition methodology allows us to approach such designs by proving the functionality of parts of the design that can be approached automatically, then stitching together these proofs into a top-level correctness theorem.

A further demonstration of this methodology used to prove correctness of a radix-4 SRT 32-bit unsigned integer divider is available in the ACL2 community books repository [19]. The divider in this example takes 18 clock cycles to finish the computation, and the needed lemmas about each iteration are generated by a macro since the same basic SRT divide step is repeated 17 times. This example shows the decomposition methodology working on a larger scale, with one cutpoint for each iteration of the partial remainder plus a few more for initialization and completion.

V. Divide and Square Root Verification Approach

Our general approach for divide and square root verification is composed of the following steps:

1) Define top-level specification functions: Specification functions for each operation are written in ACL2. For integer divides, the specifications only apply the built-in ACL2 integer divide function and then check for overflows. For floating point operations, the specification functions are necessarily more complicated; we ensure their accuracy using the following principles.

- Test comprehensively. Each specification function can be efficiently executed. For operations that have existing implementations in hardware, we run millions of tests of our specifications against the hardware implementations. We test all combinations of input floating-point types on each operation to ensure that special-case behavior is covered, and additionally craft randomized testbenches to cover a broad range of normal arithmetic operands.
- Share code between operations as much as possible. In particular, the most complicated part of floating point arithmetic specifications is the post-compute rounding and exceptions, which is the same for all arithmetic operations and which we therefore handle using the same function. This routine is tested extensively, especially on convert from double precision to single precision, an operation on which it is easy to hit all the major post-compute corner cases.
- Define core arithmetic operations as simply as possible. For divide, we left-shift the dividend mantissa by a certain amount, then divide it by the divisor mantissa using the built-in ACL2 integer divide operation. Inexactness is determined by checking whether the remainder of that divide is 0. The shift amount is computed so that the result of the divide will contain enough significant bits to correctly round the result. The square root operation is similar, shifting the mantissa left then applying a truncating integer square root function.

2) Mathematically model hardware design components: We decompose the design into pieces that we can model mathematically and that can be proved to satisfy their specifications automatically. As the strength of the available automatic proof methods improves, we can choose larger chunks of the design to make the specifications simpler and the high-level proof easier to compose. For example, we used to decompose multiplier components of the divide unit into Booth encoding and summation tree parts, but now we use VeSCMul to prove the multipliers correct without decomposition. The specification for each chunk may be either functional (the outputs are proved equivalent to some function of the inputs) or relational (some property holds of the outputs and inputs, but the exact behavior is not specified). When relational specifications are sufficient, they can be less susceptible to design changes: as long as the properties still hold, changes in the behavior of the corresponding parts do not affect the proofs. For example, often divide and square root implementations use lookup table driven approximations for certain functions; we do not need to specify the exact values of these approximations, only that they have a certain relative error. Thus, any design change in lookup table constructions would not affect such proofs as long as their bounds still hold.

3) Prove algorithmic bounds: Signals in hardware are defined with fixed bit widths. In iterative arithmetic algorithms, the bounds on a value such as the partial remainder at one step in the algorithm depend on the bounds of the previous step; if the known bounds for that previous step are not tight enough, then we cannot prove that the next step does not overflow. However, it is not always obvious what bound is needed for a particular value in order to complete the full proof. For square root, in particular, we found it important to bound not only the partial remainders at each step but additionally derived terms such as the product of a partial remainder and its

corresponding root digit. Reasoning about bounds is supported by the `def-bounds` tool [8], [9].

4) Compose sub-proofs to prove correctness of the integer/mantissa operation: We typically proceed from the inputs toward the outputs, showing that each cutpoint signal is some function of or satisfies some property relative to the operation's inputs. For example, suppose we already formalize the relationship of the next partial quotient Q_{i+1} and remainder R_{i+1} in terms of the current partial quotient Q_i and remainder R_i, and suppose we also establish the relationship of Q_i and R_i in terms of primary inputs N (dividend) and D (divisor), we then compose these facts to form a new relationship of Q_{i+1} and R_{i+1} in terms of N and D as shown below, and so on for subsequent steps.

$$\begin{cases} Q_i = f_i(N,D) \\ R_i = g_i(N,D) \end{cases} \quad \wedge \quad \begin{cases} Q_{i+1} = f(Q_i, R_i) \\ R_{i+1} = g(Q_i, R_i, D) \end{cases}$$
$$\Rightarrow \quad \begin{cases} Q_{i+1} = f_{i+1}(N,D) \\ R_{i+1} = g_{i+1}(N,D) \end{cases}$$

However, we can also take composition steps in a different order if it makes more sense. For example, if an intermediate cutpoint c_i is needed between R_{i+1} and R_i, we might first eliminate c_i to resolve the function of R_{i+1} in terms of R_i without ever resolving c_i in terms of the operation's inputs.

5) Verify floating-point rounding, exponents, exceptions, and special cases: For this proof we arrange that our specification function has extra inputs that allow us to replace the computed pre-round mantissa result and inexact-mantissa flag with arbitrary values, effectively overriding them. Similar to the override transparency property for our hardware models, we also prove here that overriding these signals with their correct values does not affect the function's value. We override these values in both the specification and hardware implementation and show that the final result and flags in terms of the primary inputs and these overridden values agree with our specification. This is done automatically with FGL. This theorem can then be composed with the result of the previous step to obtain the top-level correctness theorem for a floating point operation, without overrides in the specification or hardware model.

6) Equivalence check SIMD lanes: For single-instruction/multiple-data (SIMD) floating point operations, we use Boolean equivalence checking (via FGL) to verify the top-level SIMD operation given the correctess of the scalar (single-lane) operation. A simple way to do this in a single step would be to equivalence check the SIMD operation against a composition of n invocations of the scalar operation (where n is the number of SIMD lanes for the operation). This works for some designs, but lacks flexibility and is often not efficient. Instead we typically break down the problem by first checking the equivalence of each of the n SIMD lanes with the scalar operation, then checking that the full SIMD operation is equivalent to the composition of all n separate SIMD lanes. To run an operation on SIMD lane i (as in both of these equivalence checks), we shift the appropriate inputs to input lane i and populate the inputs for the other lanes with inputs that yield no FP exceptions (e.g., 0.0/1.0 for divide, 0.0 for square root). This multi-step methodology is typically faster than the single-step method and allows for optimizations of the lane-to-scalar equivalence checks, such as case splitting and further decomposition.

TABLE I
VERIFICATION TIMES OF VARIOUS DIVIDE AND SQUARE ROOT INSTRUCTIONS

Int instruction	Verification time	FP instruction	Verification time
DIV8	4m 27s	DIVSS	3m 30s
DIV16	7m 7s	DIVPS	4m 11s
DIV32	5m 37s	DIVSD	14m 13s
DIV64	5m 53s	DIVPD	16m 4s
IDIV8	4m 6s	SQRTSS	7m 29s
IDIV16	4m 13s	SQRTPS	8m 23s
IDIV32	4m 15s	SQRTSD	15m 4s
IDIV64	7m 49s	SQRTPD	16m

VI. RESULTS

Table I shows the times consumed by executing our proof scripts of various divide and square root instructions on a machine with an 8-core Intel Xeon E-2378G processor and 128 GB memory. All of these instructions were implemented in an Intel project. For integer operations, DIVn and IDIVn perform n-bit unsigned and signed integer divides, respectively. These operations divide a $2n$-bit dividend by an n-bit divisor producing an n-bit quotient. The proof time went from about 4 minutes for IDIV8 to 8 minutes for IDIV64. For floating-point operations, we verified four DIVXY and four SQRTXY variants, where X is either S or P denoting scalar or packed (SIMD), and Y is either S or D denoting single or double precision [20]. SIMD operations consisted of either four single-precision lanes or two double-precision lanes. Decomposition was applied only to the scalar operations; the packed operations were then verified using equivalence checking. As shown in Table I, the run-time gap between the scalar and packed versions of the same precision format is fairly small, indicating the effectiveness of the equivalence checking. The numbers also illustrate the scalability of our decomposition proof method: all operations were verified in minutes. Additionally, all these proofs may be done in parallel.

The initial development of the first divide proof took about three person-months, and the first square root proof three more. This compares very favorably to other from-scratch verifications of similarly complex industrial implementations. Additionally, this development was concurrent with improvements to the ACL2/SVTV decomposition framework that enabled these proofs, so a similar from-scratch proof effort could now likely be completed much more quickly. Major changes to proofs, such as verifying a double-precision operation once the single-precision version is already verified or updating to accommodate an extensive change in the design, generally take one to four weeks. Routine design changes often do not require modifications to the proof scripts and otherwise usually only need minor changes taking less than a day.

Nightly regressions run all collected proofs so as to catch design bugs quickly. When bugs are found, fixes can be iterated on very quickly because the time from parsing a new model to finishing the proof or finding a counterexample is between 5 minutes and an hour, depending on the operation.

VII. Conclusion

We described a methodology for formal verification of divide and square root operations based on the ACL2/SVTV hardware verification framework. We use automatic proof tools such as FGL and VeSCMul on portions of the design that play to their strengths. The ACL2/SVTV framework automatically generalizes the resulting theorems, which initially pertain only to a run of the design in which some internal signals have been overridden, to obtain composable theorems that pertain to a run without such overridden signals. The resulting generalized theorems can be composed together in a manner that is idiomatic and natural in ACL2.

This methodology has been used to successfully verify state-of-the-art, highly optimized implementations of these operations in an industrial setting. Development of new proofs is contingent on understanding the algorithm implemented by the design, but not the low-level details of its implementation; in practice, new proofs can be developed in one to several person-weeks (depending on the complexity of the design) and updating proofs for major design changes can usually be completed within a person-month.

The proofs resulting from this methodology take minutes to run, allowing nightly regression tests to quickly catch any new bugs or needed proof updates. Each such proof culminates in a top-level theorem verified in ACL2 stating that the whole operation in the hardware design satisfies its specification. The proofs have been updated for several revisions of the RTL designs in question without encountering scalability issues or major impediments to the success of the methodology.

References

[1] S. Swords, "ACL2/SVTV Source Distribution," Accessed: 2024. [Online]. [Online]. Available: https://github.com/acl2/acl2/tree/master/books/centaur/sv

[2] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Balancing Automation and Control for Formal Verification of Microprocessors," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 26–45.

[3] S. Swords, "New Rewriter Features in FGL," in *Proc of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2020)*, 2020, pp. 32–46.

[4] M. Temel, "VeSCMul: Verified Implementation of S-C-Rewriting for Multiplier Verification," in *Proc of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024)*, 2024, pp. 340–349.

[5] ——, "Verified Implementation of an Efficient Term-Rewriting Algorithm for Multiplier Verification on ACL2," in *Proc of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2022)*, 2022, pp. 116–133.

[6] M. Temel, A. Slobodova, and W. A. H. Jr., "Automated and Scalable Verification of Integer Multipliers," in *Computer Aided Verification 2020 (CAV 2020)*, 2020, pp. 485–507.

[7] M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber, "Integrating external deduction tools with ACL2," *Journal of Applied Logic*, vol. 7, no. 1, pp. 3–25, 2009, special Issue: Empirically Successful Computerized Reasoning.

[8] S. Swords, "Extended Abstract: A Bound-Finding Tool for Arithmetic Terms," in *Proc of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2023)*, 2023, pp. 11–15.

[9] ——, "Def-bounds," Accessed: 2024. [Online]. [Online]. Available: https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____DEF-BOUNDS

[10] A. Slobodova, J. Davis, S. Swords, and W. Hunt, "A Flexible Formal Verification Framework for Industrial Scale Validation," in *Proc of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE2011)*, 2011, pp. 89–97.

[11] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Verifying X86 Instruction Implementations," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 47–60.

[12] D. Rager, J. Ebergen, D. Nadezhin, A. Lee, C. Chau, and B. Selfridge, "Formal Verification of Division and Square Root Implementations, an Oracle Report," in *Proc of the 2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 149–152.

[13] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, 2nd ed. Springer, 2022.

[14] D. Russinoff, J. Bruguera, C. Chau, M. Manjrekar, N. Pfister, and H. Valsaraju, "Formal Verification of a Chained Multiply-Add Design: Combining Theorem Proving and Equivalence Checking," in *Proc of the 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 120–126.

[15] D. M. Russinoff, "RAC Source Distribution," Accessed: 2024. [Online]. [Online]. Available: https://github.com/acl2/acl2/tree/master/books/projects/rac

[16] J. O'Leary, R. Kaivola, and T. Melham, "Relational STE and theorem proving for formal verification of industrial circuit designs," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 97–104.

[17] R. Kaivola and J. O'Leary, "Verification of Arithmetic and Datapath Circuits with Symbolic Simulation," in *Handbook of Computer Architecture*, A. Chattopadhyay, Ed. Singapore: Springer Nature Singapore, 2022, pp. 1–52.

[18] S. Swords, "Boothpipe-mini tutorial example," Accessed: 2024. [Online]. [Online]. Available: https://github.com/acl2/acl2/tree/master/books/centaur/sv/tutorial/boothpipe-mini.lisp

[19] ——, "SRT-4 divider tutorial example," Accessed: 2024. [Online]. [Online]. Available: https://github.com/acl2/acl2/tree/master/books/centaur/sv/tutorial/srt-div

[20] Intel Corporation, "Software Developer Manuals for Intel® 64 and IA-32 Architectures," Accessed: 2024. [Online]. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000006715/processors.html