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

Sol Swords, Cuong Chau

Divide and Square Root Hardware Implementations

Hardest part of a typical arithmetic datapath:

- Simple algorithms don’t scale
- Even advanced algorithms are iterative and take longer than most other operations
- Therefore, aggressive optimizations are common
- Verification is correspondingly difficult
    - Can’t be sufficiently tested
    - Can’t practically be checked by fully automatic methods.

Verification options

Full automation	Partial automation	Low automation
● Algebraic methods + Fully automatic - Only works with simple algorithms – low-radix, low performance - Broken by ad hoc optimizations	● Verify computation steps using fully automatic tools ● E.g. STE, model checkers, equivalence + Practical and relatively efficient - Difficult or impossible to prove end-to-end functional correctness	● Mathematical proof using interactive theorem prover + Results in full end-to-end proof of functional correctness - High manual effort - Design changes will almost always break proofs

Verification options

Full automation	Partial automation	Low automation
● Algebraic methods + Fully automatic - Only works with simple algorithms – low-radix, low performance - Broken by ad hoc optimizations	● Verify computation steps using fully automatic tools ● E.g. STE, model checkers, equivalence + Practical and relatively efficient - Difficult or impossible to prove end-to-end functional correctness	● Mathematical proof using interactive theorem prover + Results in full end-to-end proof of functional correctness - High manual effort - Design changes will almost always break proofs

Our Approach

- Parse RTL, process into FSM object in the ACL2 theorem prover
- Define top-level specification in ACL2
- Break computation into steps and determine corresponding design cutpoints
  - Want big steps that can be automatically verified
- Define specifications for those steps
  - Want the composition of these to correspond to the top-level specification
- Use automatic methods to prove correctness of each step according to its spec
  - Proof engines verified in ACL2
- Compose theorems about steps into end-to-end proof

Features of this approach

+ Robust to design changes due to block size of automated proofs
+ Works on highly optimized designs with advanced algorithms, high radix
+ Produces end-to-end correctness proof
+ Fast runtime, relatively* short development time

Automation for Hardware Proofs

Two main engines, both written and **verified** in ACL2:

- FGL: bit blasts ACL2 specs and FSM unrollings into and/inverter graph
  - Queries solved using off-the-shelf SAT solvers, Boolean simplification, equivalence checking
  - Extensible rewriter for adding new abstractions, normal forms
  - Generates counterexamples
- VeSCMul: Specialized rewriter for adders and multipliers
  - Rewrites spec and implementation to same normal form
  - Hardened on many industrial multiplier designs

Composability of automatic proofs

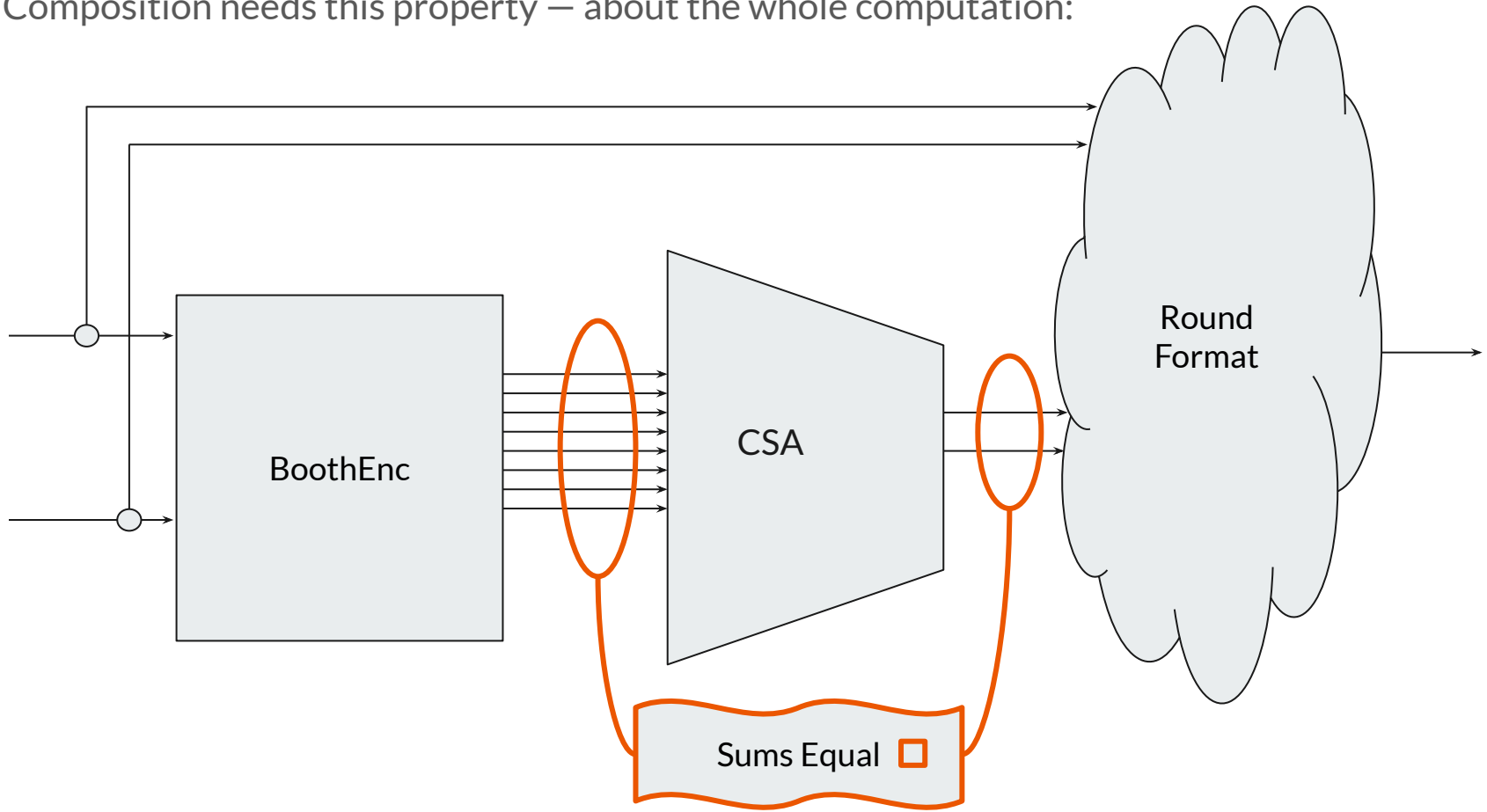
Composable proof of a computation step: relates values of internal signals on a run of the whole circuit:

> “The sum of the values of signals s, c at time n+2 equals the sum of the values of signals p0, …, p7 at time n+1”
>
> “If the absolute value of signal rem at time n+3 is less than 3/2 the value of divis at time n, then the absolute value of rem at time n+4 is also”

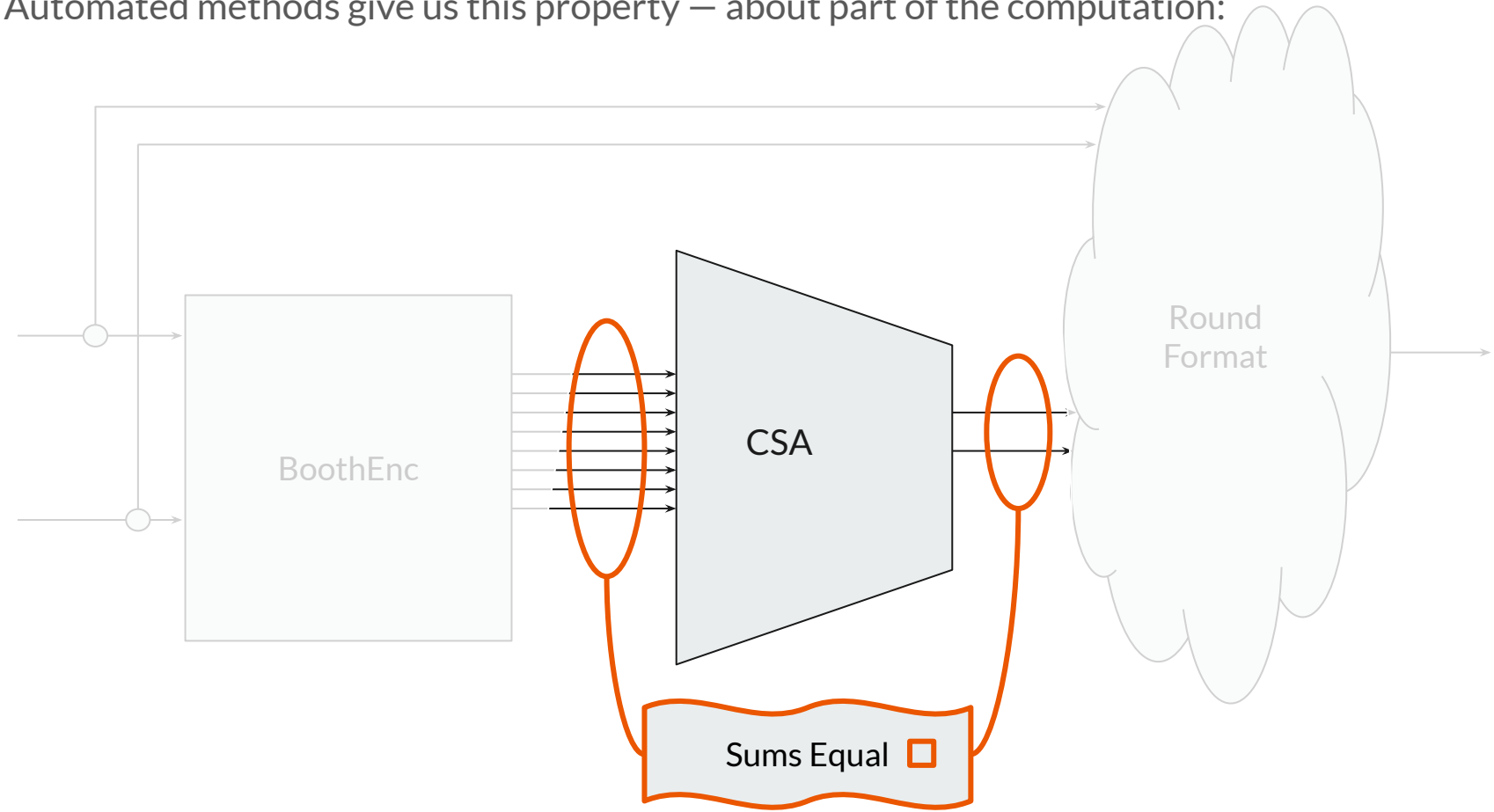
Automatic methods need a strict reduction of the circuit without the fanin cones of the inputs:

> “If we *override* the values of signals p0, …, p7 at time n+1, then the sum of the values of signals s, c at time n+2 equals the sum of those values”
>
> “If rem is overridden at time n+3 and divis overridden at time n, and the absolute overridden value of rem is less than 3/2 that of divis, then the absolute value of rem at time n+4 is also”

Composition needs this property — about the whole computation:



Automated methods give us this property — about part of the computation:



Composable Theorems from Override Proofs

Automatic conversion of override theorem to composable theorem uses global properties of the circuit artifact:

- Override transparency: overriding a signal to its driven value doesn't change any signal values
- X-monotonicity: setting an input signal to the X (indeterminate) value is as general as setting it to a free variable.

These are proved once for the whole circuit and then used for every cutpoint proof.

Easy* Parts

- Lookup table approximations — error bounds usually provable with SAT
- Rounding/formatting — usually a single automated proof
  - Most bugs found here
- Iterative parts: prove it once, repeat it N times
  - Or equivalence check
- SIMD: prove once (or sometimes twice), equivalence check with other lanes

*moreso than might be expected

Hard parts

- Sometimes the algorithms themselves
  - Good idea to prove the algorithm correct independent of the RTL
  - Reverse engineering can be hard if documentation is lacking
- Bounding arithmetic expressions
  - Quantities have to fit in their bitwidths
  - Error bounds need to converge
  - Def-bounds bound-finding tool

Results

- Initial proofs for divide and square root (while developing methodology) ~3 person-months each
  - Short by industrial standards
  - Subsequent proofs (different sizes/variants, other designs based on same algorithms) were developed faster — weeks rather than months
- Composition of steps is now similar to other interactive theorem proving pursuits
- Scalable — verified up to double extended precision FP, 128 / 64 bit integer operations
- Proofs run in minutes — allow fast feedback and bug iterations
- Verifier needs to understand algorithm but not low-level details of design
- Changes in low-level details of design don't usually break proofs
- Debugging is aided by counterexamples from cutpoint proofs.

Qs?