Formal Verification of Division and Square Root Implementations, an Oracle Report

David L. Rager, Jo Ebergen, Dmitry Nadezhin, Austin Lee, Cuong Kim Chau, Ben Selfridge

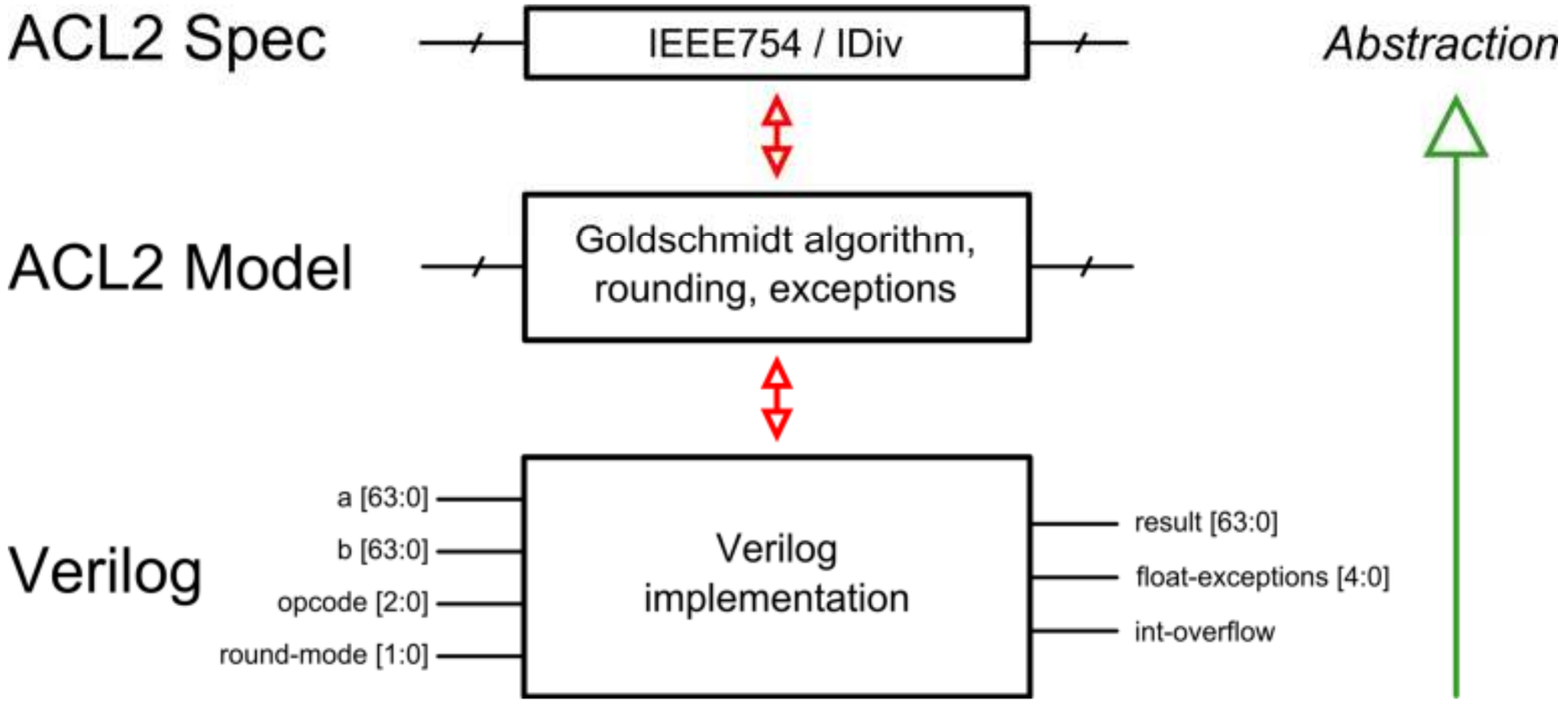
October 6, 2016

ORACLE

Goal

- Verify data-path for new implementations of:
  - 32/64-bit floating-point division and square root
    - fdivd
    - fdivs
    - fsqrtd
    - fsqrts
  - 32/64-bit integer divide
    - udivx
    - sdivx
    - udiv
    - sdiv

The Problem and Key Result



Tools

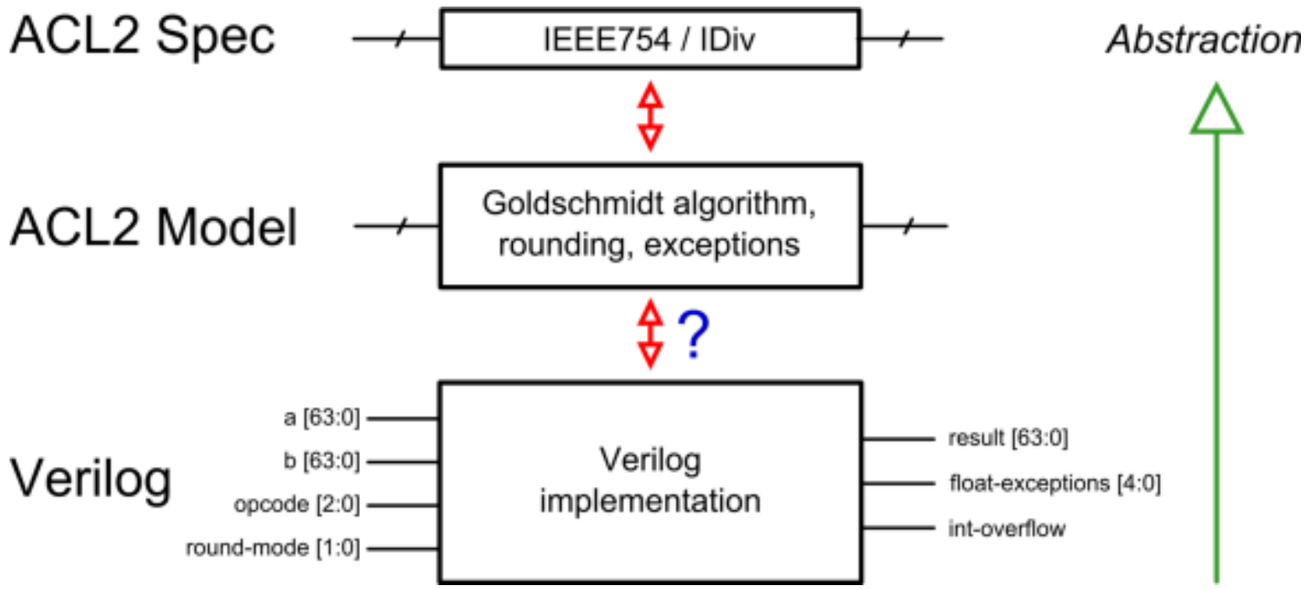
- ACL2
  - Programming language written in subset of Lisp
  - Theorem prover written in ACL2
    - Proof engine used at AMD, IBM, Centaur, Motorola, Intel
    - 2005 ACM Software System Award
  - Maintained at Univ. of Texas with help from community
- ACL2 Books (~5500)
  - A “book” is a library of functions and lemmas
    - Arithmetic, RTL, security, proof and definition utilities
  - Includes a Verilog parser and hardware symbolic simulator
- Support Tools: SAT solvers, waveform viewer

Related Work

- Symbolic trajectory evaluation (Intel)
  - C.-J. H. Seger and R. E. Bryant, “Formal verification by symbolic evaluation of partially-ordered trajectories,” *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.
- Floating-point verification
  - D. Russinoff, “A mechanically checked proof of IEEE compliance of the floating-point multiplication, division, and square root algorithms of the AMD-K7™ processor,” *London Mathematics Society Journal of Computation and Mathematics*, no. 1, pp. 148–200, 1998.
  - J. O’Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, “Formally verifying IEEE compliance of floating-point hardware,” *Intel Technology Journal*, vol. 3, no. 1, pp. 1–14, 1999.
- Hardware verification and tools
  - A. Slobodova, J. Davis, S. Swords, and W. A. Hunt, “A flexible formal verification framework for industrial scale validation,” in *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, July 2011, pp. 89–97.

Outline

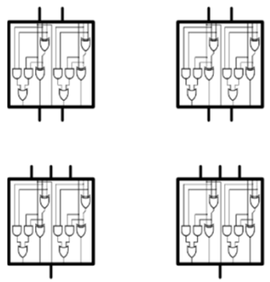
- Intro
- *Algorithm extraction*
- Algorithm verification
- Reflections and challenges



- Goal: raise level of abstraction from low-level bit operations to higher-level operations like *, +, and ~ of *m*-bit operands

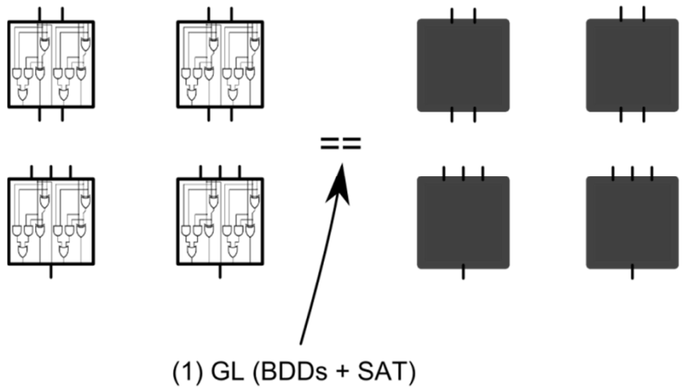
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- Choose modules of interest
  - For example:
    - Tree of carry-save adders (CSAs)
    - Nest of Booth encoders



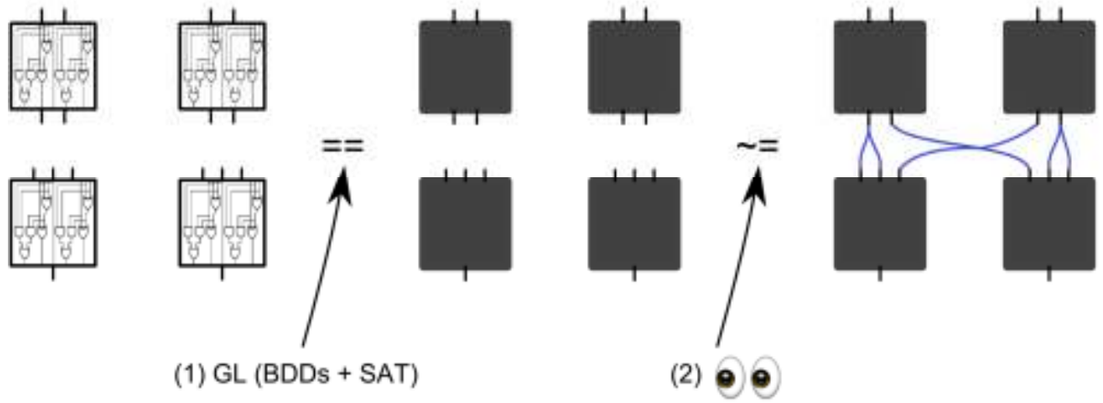
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- (1) Black-box chosen modules
  - Write specification for those modules in ACL2
  - Automatically verify the validity of those specifications using GL
    - GL uses BDDs and SAT solvers “under the hood”



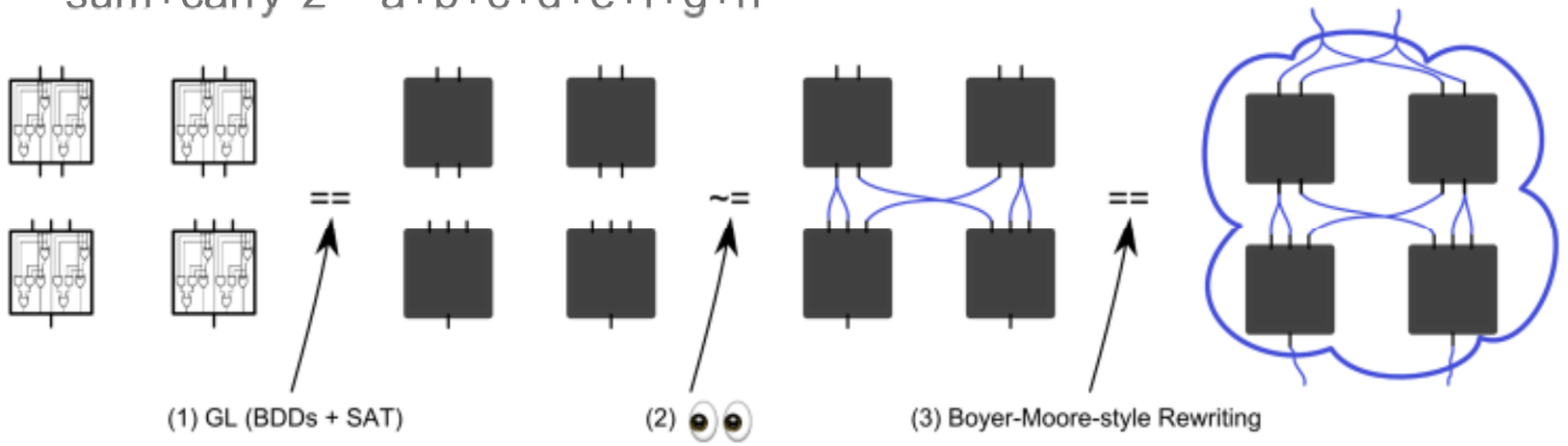
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- (2) Create ACL2 version of the interconnect
  - For example:
    - The wires that connect the CSAs are connected in a particular way
  - ACL2 version of interconnect is unverified at this point



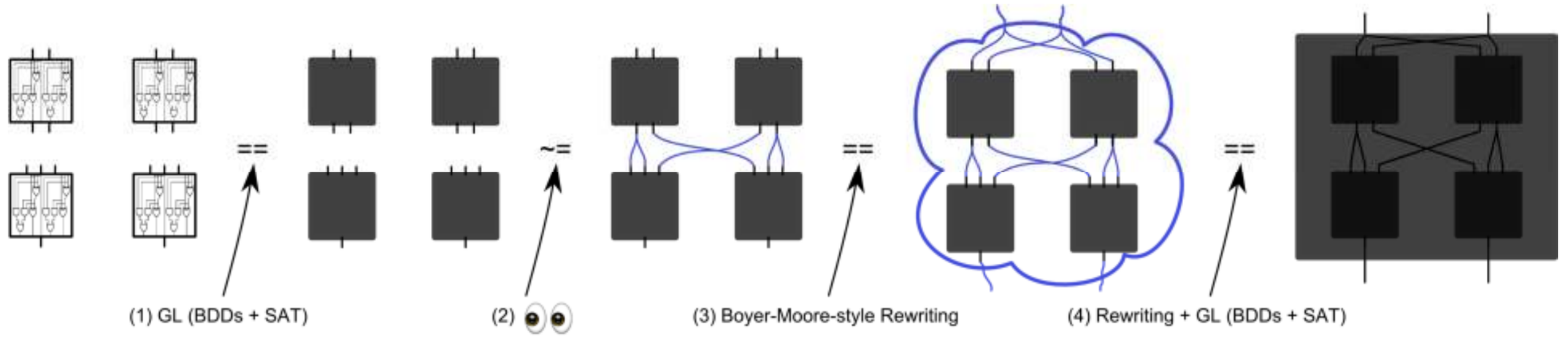
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- (3) Prove a higher-level specification
  - Define a higher-level specification for the connected modules
  - Prove specification's validity using Boyer-Moore rewriting
  - For example:
    - sum+carry*2 = a+b+c+d+e+f+g+h



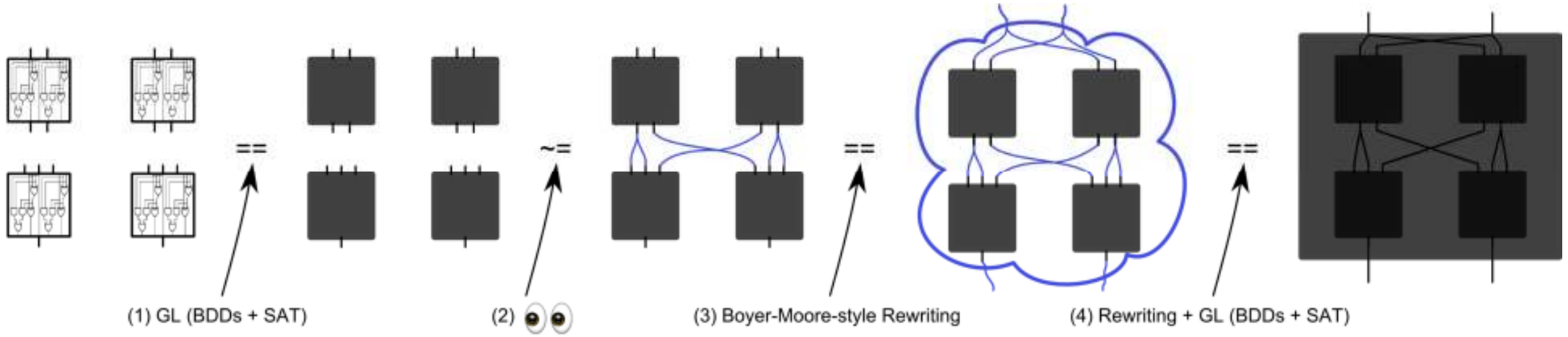
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- (4) Black-box your larger piece of circuitry
  - Prove that the ACL2 interconnect is the same as the Verilog interconnect
    - I.E., that the Verilog wires really do connect the CSA's that way!



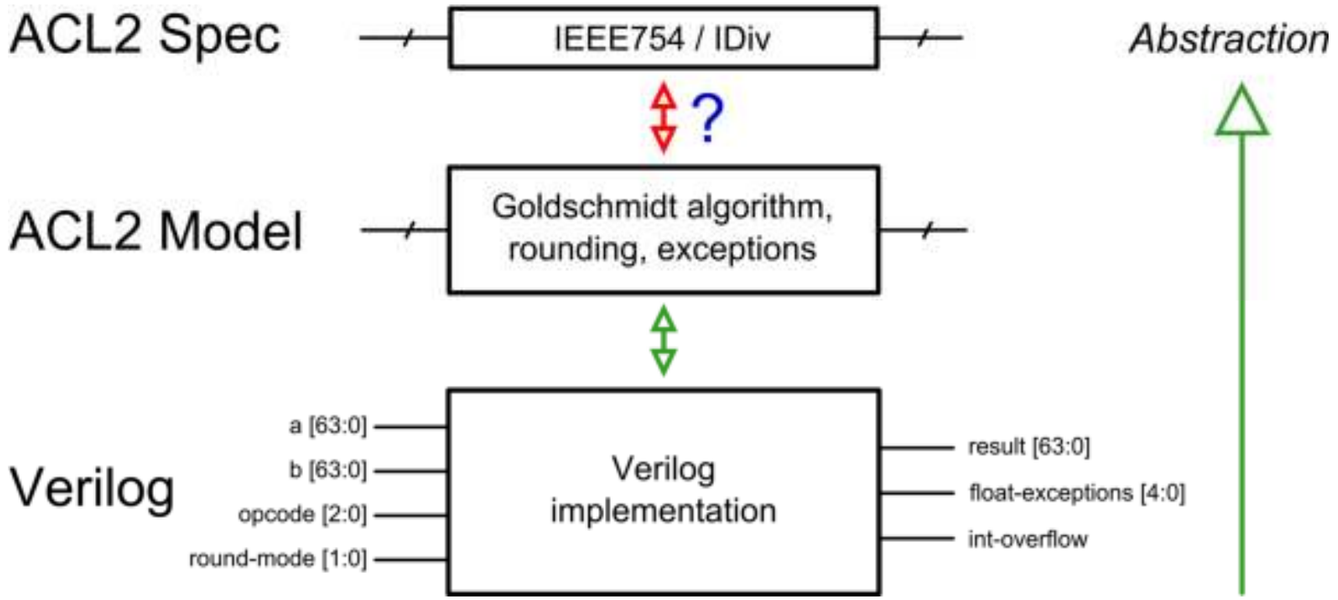
Breaking Up Is Hard To Do

- Decompose circuit into appropriately-sized blocks
- (4) Black-box your larger piece of circuitry
  - Black-boxing doesn't scale using Esim and GL
  - Use SV (successor to Esim) in our latest work
    - Scales better but we still have problems too large



Outline

- Intro
- Algorithm extraction
- *Algorithm verification*
- Reflections and challenges



- Goal: show that the Goldschmidt algorithm (consisting of operations like *, +, and ~ of *m*-bit operands), rounding, and exceptions implement IEEE 754

IEEE754 Specification in ACL2

- IEEE754 Standard on Floating-Point Arithmetic
  - 80-page document written in English
- Our IEEE 754 specification in ACL2 includes
  - Div, sqrt, add, mul, and fused mul-add
  - All special values (+/- 0, +/-Infinity, NaNs)
  - All exception flags
  - Denormals
  - Four rounding modes
  - Customization for NaN values
- Validated our spec against millions of test vectors from Oracle's test suite

Goldschmidt Algorithm for Division

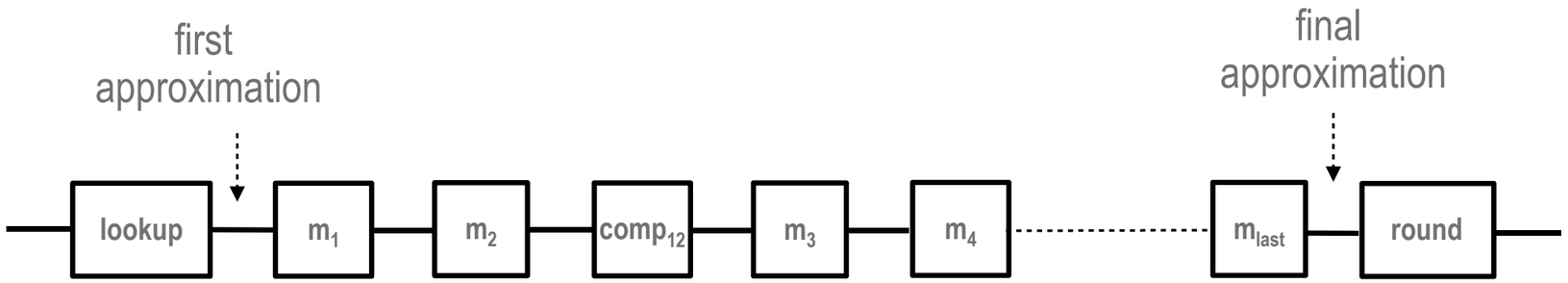
- Idea: choose T, r_i such that

$$\frac{A}{B} * \frac{T}{T} * \frac{r_0}{r_0} * \frac{r_1}{r_1} * \frac{r_2}{r_2} * \frac{r_3}{r_3} \ldots \begin{array}{l} \longrightarrow Q \\ \longrightarrow 1 \end{array}$$

- Precision doubles with each iteration
- Algorithm:

```
T = table_lookup(B);
d0 = B*T; n0 = A*T;
r0 = 2 - d0;
for (i=0; i < MAX; ++i) {
   di+1 = di*ri; ni+1 = ni*ri;
   ri+1 = 2 - di+1;
}
final_approx = nMAX + inc
```

Main Proof Obligation



$$-\text{max\_error} < \text{final\_approx} - A/B < \text{max\_error} \quad ?$$

- Each step introduces an error
  - Lookup: $T \sim 1/B$. Define relative error u by $T = 1/B - u/B$
  - Each multiplication, except last, is truncated from $2M$ to M bits. Error eps_i is in $[0, 2^{-M})$
  - $2 - d_{i+1}$ is implemented by taking one's complement of d_{i+1}. This introduces fixed error 2^{-M}
- Golden question: *Is error in final approximation small enough to yield an IEEE754 answer after rounding is applied?*

Error Analysis

- Express (final_approx - A/B) as a multivariate polynomial in u (lookup error) and eps_i *(truncation error)*
- This polynomial can be generated symbolically from the algorithm
- Given the interval for each variable, compute interval for (final_approx - A/B) using methods from interval arithmetic
- Example: If lookup error u was only error, then final error for, e.g., final_approx = n_2 can be expressed as

$$\text{final\_approx} - A/B = A*T*(-u^4 - u^5 - u^6 \ldots) + \text{inc}$$

with u in $[-2^{-k}, 2^{-k}]$ and $A*T < 2$.

Results of Error Analysis

- Proved main obligation using interval arithmetic

$$-\text{max\_error} < \text{final\_approx} - A/B < \text{max\_error}$$

- We first implemented interval arithmetic in Java™ and later verified computations in ACL2
- We then experimented with reduced lookup tables to see if main obligation still holds.
- This approach reduced the lookup table
  - for division by 50%
  - for square root by 75%

Reflections and Challenges

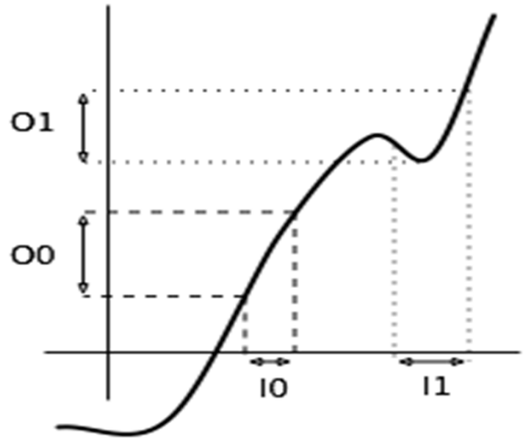
- Approach is very similar to Symbolic Trajectory Evaluation (STE)
  - Works very well for data-path verification
  - Technical challenges involving Step 4 of Extraction (recomposition)
- Invariant-based methods
  - More thorough but more time-consuming
  - Necessary for verifying control logic
  - Can community make invariant-based frameworks and methodologies more efficient for users?
    - Currently too time-consuming for industry to use on major products with deadlines
- A dream: automatically infer higher-level specifications for Verilog implementations


ORACLE®

Backup Slides

Interval Arithmetic Intermezzo

- Function of single input variable
  - For each input interval, compute output interval



- Computing the output interval for multivariate polynomials is similar to computing the output interval for univariate polynomials