# A Methodology for Large-Scale Hardware Verification

*Mark D. Aagaard, Robert B. Jones, Thomas F. Melham*, John W. O'Leary, Carl-Johan H. Seger*

**Strategic CAD Labs, Intel Architecture Group**
**Intel Corporation, Hillsboro, OR**

***Dept. of Computing Science, Univ. of Glasgow**
**Glasgow, Scotland**

## Why Methodology?

- **Theory by itself is not usable (in practice)**
- **Theory with a system is not usable (in practice)**
- **A *methodology* is required for theory and a system to be usable in practice**

## Outline

- Motivating Example: Floating-Point Adder
  - Proof Overview
- Methodology Requirements
- Methodology Description: datapath verification with STE
  - Wiggling
  - Targeted Scalar Verification
  - Symbolic Verification
  - Theorem Proving
- Conclusion

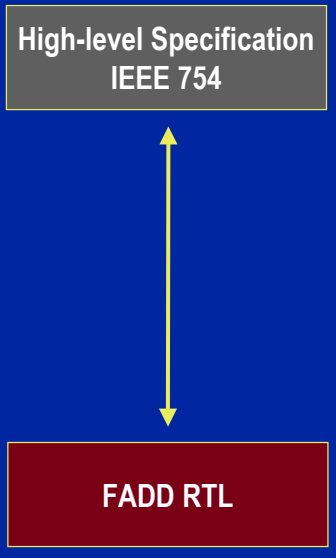
## Motivating Example: FP Adder

- IEEE-compliant floating-point addition and subtraction
- Multiple precisions
  - single, double, extended
- Multiple rounding modes
  - toward 0, +∞, -∞, to nearest
- FPU from Intel Pentium® Pro processor
  - Large block of RTL code written by architects and logic designers
  - No modification from design database

# Proof Overview

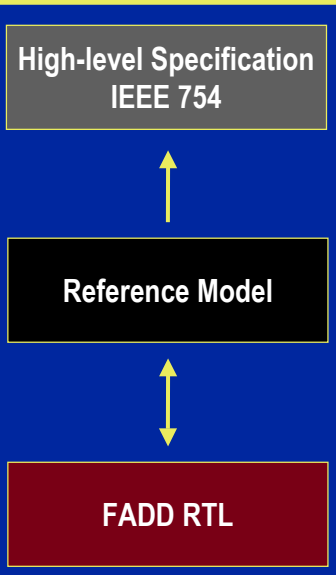

- Proof target: RTL model
  - (register transfer level)
- Design activity centers on RTL
- It is the "golden" functional reference model for the implementation
- RTL is the primary focus of validation efforts

# Proof Overview



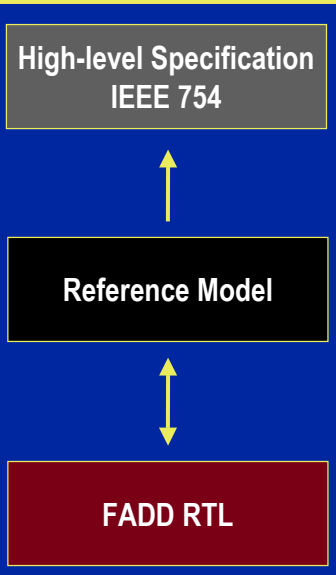

- RTL should implement IEEE 754 floating-point specification
- IEEE 754 not formal, many pages of English descriptions
- *Huge* abstraction gap between specification and implementation
- How can we bridge the gap?

## Proof Overview




- We create a *reference model*
- It must be formal
- It must be executable
- It should be *functionally equivalent* to the RTL
- It should satisfy IEEE specification

## Technical Issues




- IEEE 754 is not expressed formally
- What abstraction level for the reference model?
- Where do we get a reference model?
- How do we connect the reference model to the RTL?
- Production RTL is *huge*
- RTL saddled with implementation considerations
- How do we manage complexity of the proof itself?

## Suppose we find solutions...

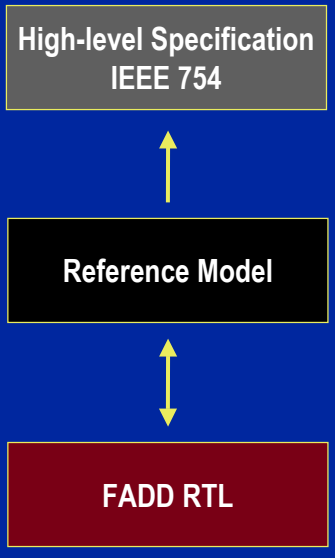
## "Top 10" reasons we can still fail...

- Spend all your time finding bugs in the spec
- Can't keep up with daily design evolution
- System crashes while printing final BDD order
- At the end, you have no idea what you've actually proved
- And neither does your manager
- Design changes break structural decomposition
- "120,000" line proof little (or no) use for next project
- Spell checker removed those funny ∀ & ∃ characters
- You're the only one who knows how to do this--do you want to verify adders until retirement?
- After three years of effort, project canned because project was "all-or-nothing", and you have ...

## We Must Also Have ...

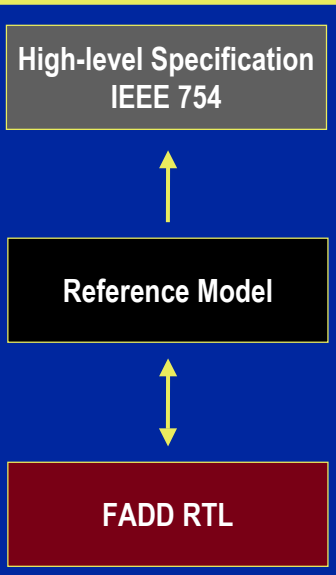
- *A systematic, pragmatic approach to organizing large-scale hardware verification efforts*
- Methodology: a systematic approach to problem solving
  - Scope can be very broad (the scientific method)
  - Or quite narrow (BDD variable ordering)
- Methodologies are familiar in CAD
  - Sometimes called *flows*
  - Dictate the order and scope of design activities
  - Often involve multiple tools and data representations

## Key Messages



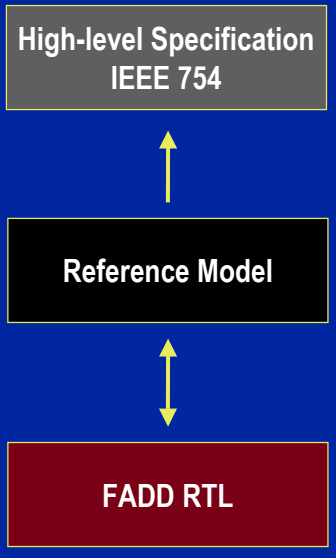

- Realistic
  - Complete specifications are usually not available
  - Access to design engineers is always limited
- Incremental
  - Must be able to measure progress
  - Effort should develop "debugging value" early

# Key Messages



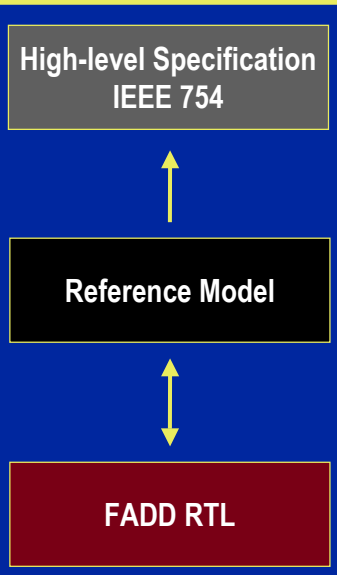

- Transparent
  - Should indicate what has (not) been proved
  - Must be sound (no false positives)
- Structured
  - Helps new users learn
  - Increases productivity of experienced users

# Key Messages




- Works top-down and bottom-up
  - Top-down for problem reduction
  - Bottom-up for design understanding and tool capacity limits
- Optimized for common case
  - Bulk of verification effort is debugging
  - Optimize for proof *failure*, not success

## Key Messages




- Supports regression efficiently
  - Verification *artifacts* should be maintainable
  - And reusable after design or specification changes
- Allows effort reuse
  - Verification is human-intensive (expensive)
  - Amortize cost over changes or multiple designs
  - Proof effort should be relatively circuit independent

## The Methodology

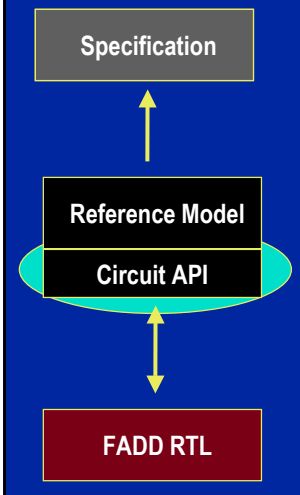
- Intended for datapath verification based on symbolic trajectory evaluation (STE)
- Assumes a comprehensive verification framework
- Our (internal) framework is called Forte
  - Formerly known as Voss [Seger UBC]
  - Programmable interface (FL)
  - Multiple verification engines
  - Debugging support

# The Methodology

- Four primary stages:
  - Wiggling
  - Targeted Scalar Verification
  - Symbolic Verification
  - Theorem Proving



# Circuit API




- The "glue" between our datapath reference models and the RTL
- Captures signal names, interface, timing, protocol, etc.
- Programmed in FL
- Challenges:
  - Complex interface
  - Complex internal behavior
  - Protocols are poorly documented, and will change

# Challenge: Complex Interface


- This abstracted interface looks quite simple
  - The real interface has several hundred signals!



# Interface Abstraction

- Define a mapping from abstract transaction packet to concrete signal names and timing

```
let Inputs(vld,opcode,pc,rc,src1,src2)=
    // valid bit
      ("op_vld" is vld in_phase (clk, t)) and
    // opcode
      ("op_code" isv opcode in_phase (clk, t)) and
    // precision and rounding control
      ("pre_ctl" isv pc in_phase (clk, t+1)) and
      ("rnd_ctl" isv rc in_phase (clk, t+1)) and
    // sources
      ("in_data1" isv src1 in_phase (clk, t+1)) and
      ("in_data2" isv src2 in_phase (clk, t+1)) ;
```

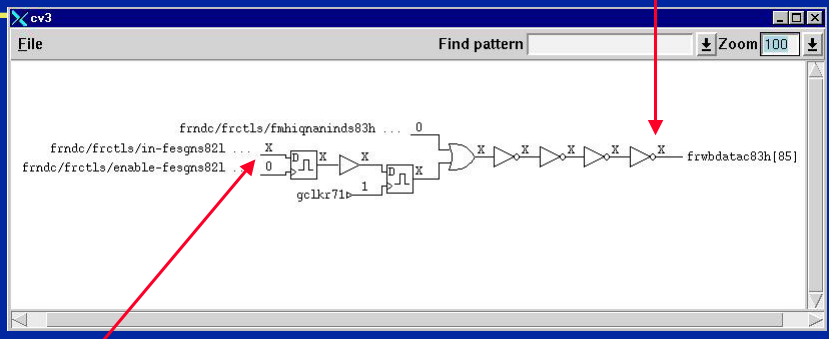
# Developing Circuit APIs

- Use STE as a scalar simulator
- Begin with trivial inputs
  - We started by adding 0E0 + 0E0
- Observe the simplest outputs
  - We started by looking at the sign bit of the result
  - We expect the sign bit of 0E0 + 0E0 to be 0
- Appearance of X on an output is symptomatic of incomplete/missing/wrong information in the API
  - Fix API
  - Lather, rinse, and repeat
- For example ...

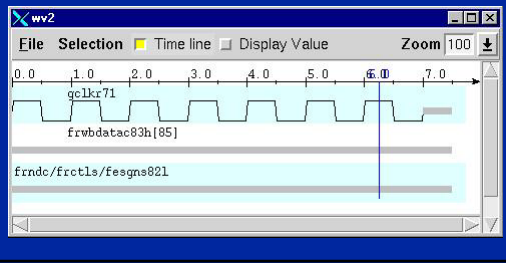


# Example

Want 0 at this output in cycle 6 (sign bit of result)




X at output is caused by X here




Theorem Proving
Symbolic Verif.
Targeted Scalar Verif.
Wiggling

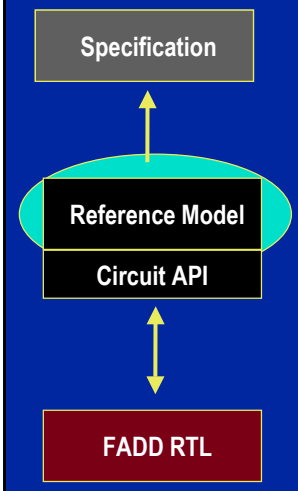
## Summary: Wiggling

- *Artifact:* **Circuit API that defines circuit interface:**
  - **Signals**
  - **Timing**
- *Artifact:* **Primitive specification**
- *Time to move on when:* **circuit outputs can be driven non-X**

## The Methodology

- **Wiggling**
- **Targeted Scalar Verification**
- **Symbolic Verification**
- **Theorem Proving**

# Developing the Reference Model




- Incrementally craft a specification
  - Start with simple behaviors
  - Captures numerical function of the datapath
  - Purely algorithmic
  - Written in FL
- Adapted a textbook algorithm as a first approximation
- Debug with scalar values:
  - inspection
  - comparison with circuit
  - pencil-and-paper computations



# Developing the Reference Model

- Result in the textbook algorithm was rounded toward 0 (truncated)
- Rounding in the completed reference model:

```
let RND pc rc s sgf =
     // Extract lsb, guard, round sticky bits.
        let L = Lsb pc sgf in
        let G = Guard pc sgf in
        let RS = RoundS pc sgf in
     // Conditionally add one to LSB
        let rbit =
            (rc '=' TO_ZERO)     => F
          | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
          | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
          | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
          | F in
      // Result truncates mantissa to precision specified
      // by pc, adds rbit and pads result with zeros.
         Result rbit pc sgf;
```

## Summary: Scalar Verification

- *Artifact*: Functional specification
- *Artifact*: Improved circuit API
- *Artifact*: Scalar test vectors (useful for regression)

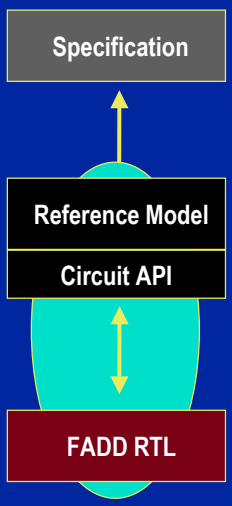
- *Time to move on when:* difficult to find discrepancies between circuit and specification



## The Methodology

- Wiggling
- Targeted Scalar Verification
- Symbolic Verification
- Theorem Proving

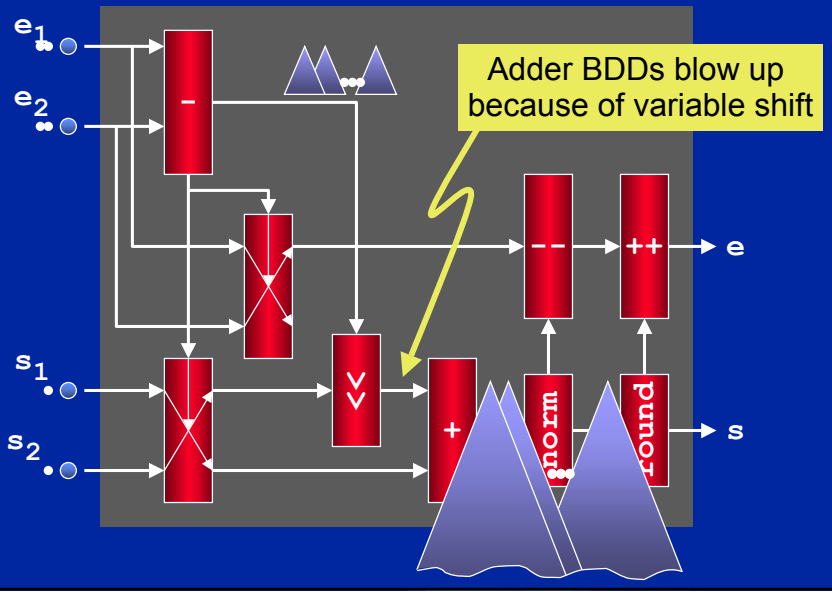
# Verifying RTL Against Ref. Model



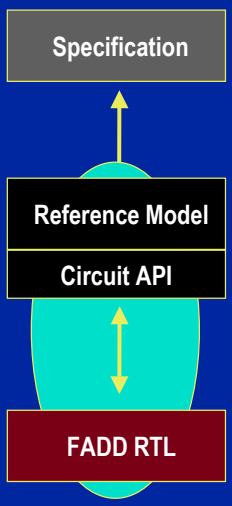

- Used symbolic trajectory evaluation (STE)
- Complexity issues:
  - data-dependent shifts in reference algorithm and RTL
  - BDD blowup in RTL's LZA circuitry



# Challenge: Tool Capacity (BDDs)



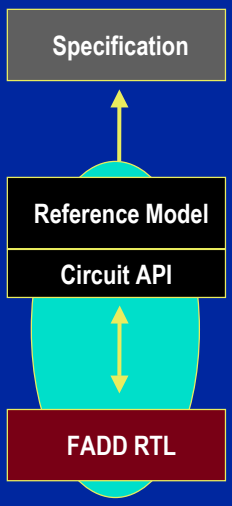
# Verifying RTL Against Ref. Model




- Used symbolic trajectory evaluation (STE)
- Complexity issues:
  - data-dependent shifts in reference algorithm and RTL
  - BDD blowup in RTL's LZA circuitry



# Verifying RTL Against Ref. Model




- Used symbolic trajectory evaluation (STE)
- Complexity issues:
  - data-dependent shifts in reference algorithm and RTL
  - BDD blowup in RTL's LZA circuitry
- Key insights:
  - split input data space to avoid data-dependent shifts (exploits *parametric representation*)
  - employ two STE verifications w/different BDD orderings (one for LZA circuit, one for result)
  - *dynamic weakening* of symbolic values
- 342 cases verified in parallel on workstation network

Theorem Proving
Symbolic Verif.
Targeted Scalar Verif.
Wiggling

# Case Splitting

- How do we generate the cases?
  - Split according to the difference between the input exponents of `A` and `B`
  - For exponent difference of 0 or 1, further split on number of leading zeros in mantissa

```
let true_add_case n =
    TrueAdd AND (ExpDiff n);
```

- Did we forget a case?
  - Easily checked in FL by a BDD computation

```
TrueAdd ==>
  (OR_list true_add_cases);
```



# Summary: Symbolic Verification

- *Artifact*: improved specification
- *Artifact*: input-space partitioning
  - Feasible model checking
- *Artifact*: BDD variable orderings

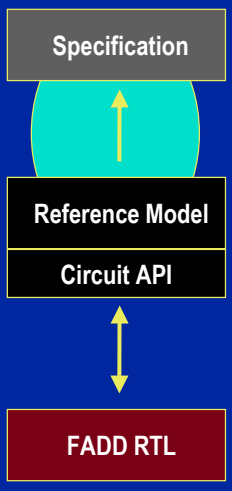
- *Time to move on when:* complexity management techniques provide diminishing returns

# The Methodology

- Wiggling
- Targeted Scalar Verification
- Symbolic Verification
- Theorem Proving



# Theorem Proving




- Verify reference model against an external benchmark (in this case, a formalization of IEEE compliance)
- Algorithm verification, done in a higher order logic theorem prover
- Finds bugs:
  - Proof bugs: incorrect reasoning, incomplete case splits
  - Model bugs: missing behavior, corner cases
  - Fixing bugs may require model-checking modification--which may reveal circuit bugs

## Summary: Theorem Proving

- *Artifact*: Final version of functional specification
- *Artifact*: Top-level correctness statement
- *Artifact*: Collection of model-checking runs
- *Artifact*: Mechanized proof
  - Connects correctness statement to STE runs
- *Time to move on when:* the top-level specification has been formally linked with the STE runs



## The Bigger Picture

- In practice of course, methodology stages are not strictly sequential
  - Fair amount of overlap
  - And backtracking
  - And more backtracking ...

## Regression

- Finishing theorem proving doesn't mean verification is finished
- A "live" design is still changing
  - Architecture changes for performance enhancements
  - ECOs for circuit implementation issues
- Regression is a significant overhead in an on-going verification effort
  - We were surprised by this finding
  - Optimize both platform and methodology for failing cases



## Regression

- An effective methodology localizes changes textually
- Greatly assisted by programmable interface to verification engines
- Closely related to regression is proof *re-use*
- *Time to move on when:* Your manager tells you you're "done"

# Conclusions

- An effective methodology is key to successful verification "in the large"
- An effective methodology must fit skill set of verification team
- We have presented a four-stage methodology for STE datapath verification
  - Proven through many large industrial verifications
  - Applicable in other verification approaches
- Still a work in progress
  - Technology advances require (and enable) advances in methodology
  - Tension between tool capability and usability