Efficient and Exhaustive Floating Point Verification Using Sequential Equivalence Checking

Travis Pouarz, Mentor Graphics Corp.
Vaibhav Agrawal, ARM, Inc.
Outline

• Motivation
• Formal Equivalence Checking
  – SLEC
• FPU
  – Conversions, FADD, FMUL, FMA, FDIV, FSQRT
• Bonus: Global History Buffer
• Moving Forward
Motivation and Action

• Floating Point (FP) bugs unacceptable:
  – Cost: $ and Good Will

• Search space huge; Control requirements contained
  – Sequential Equivalence Checking emerging as good fit

• ARM’s high-end Cortex® A-class CPUs
• Mentor’s SLEC (Sequential Equivalence Checker)
• Partnered to tackle FP block validation:
  – C++ vs RTL
Formal Equivalence Checking

- Mathematical state space search (no test vectors)
- Full proof is complete state space comparison
- Constrain the space if you want
- Two types:
  - Combinatorial — internal flops must map
  - Sequential — no internal flop mapping required
• **Sequential Logic Equivalence Checker**
• Provide designs and setup
• Setup: Control script written in TCL
• Setup allows comparison despite differences
  – In timing
  – In interfaces
  – In levels of abstraction
• Falsifications
  – Provide *shortest* error trace waveform
Why choose an FPU?

• Formal setup is relatively easy
  – Limited control signals
  – Vast state space
• Sophisticated high-performance designs
  – More room for bugs. More special-conditions and corners.
• Setup is portable to newer designs, architectures
FPU Operations

• Operations (IEEE 754):
  – Conversions (SP <=> DP, fixed <=> floating)
  – Scaling and quantizing
  – Comparisons and total ordering
  – Others that are even easier
    • Setup is easy
    • Full proof in minutes
  – Arithmetic (add, multiply, divide, square root, fused multiply-add, remainder)
    • We’ll go into more detail here
• Setup is easy, like conversions
• For Double-precision, case split operand inputs
  – One operand → Three ranges
    (1) \{zero, NaN, infinity\}
    (2) normals
    (3) subnormals
  – $3 \times 3 = 9$ cases total
• Full proof in a few hours for Double-precision
Decomposition Structure of FMUL

- Structure is different enough that end-to-end verification is not successful (this is expected)
- Decompose the proof
- At the core of FMUL, in both models: a multiplier
  - the Mantissa Multiplier (MM)
  - RTL: Booth Multiplier implementation
- Divide our proof into two parts
  - Verify the multiplier
  - Verify the logic outside the multiplier
FMUL Multiplier: by multipart iteration

- Check for a small N=2
  - that’s the base case
- Iterate for N=3..51
  - Assume is proven
  - Lemma 1 and Lemma 3 prove the Goal of this iteration
  - Which is Assume of next iteration

Goal (for a given N):
\[ a[N:0] \times b[51:0] = \text{Booth}( a[N:0], b[51:0] ) \]

Assume:
\[ a[N-1:0] \times b[51:0] = \text{Booth}( a[N-1:0], b[51:0] ) \]

Lemma 1: easy word-level problem
\[ a[N:0] \times b[51:0] = (a[N] \times b[51:0]) \ll N + (a[N-1:0] \times b[51:0]) \]

Lemma 2: trivial — vacuously true
\[ (a[N] \times b[51:0]) \ll N + (a[N-1:0] \times b[51:0]) = (a[N] \times b[51:0]) \ll N + \text{Booth}( a[N-1:0], b[51:0] ) \]

Lemma 3: manageable bit-level problem
\[ (a[N] \times b[51:0]) \ll N + \text{Booth}( a[N-1:0], b[51:0] ) = \text{Booth}( a[N:0], b[51:0] ) \]
FMUL: Check Everything Else

C Model

opA

opB

norm+

manA

manB

manProd

rounding+

result

ex

RTL Model

opA

opB

norm+

manA

manB

manProd

rounding+

result

ex

Equal?

MM (mantissa multiplication)

Drive C from RTL

Equal?

Equal?

Equal?

Equal?
Transactors

• Modules in Verilog, SystemVerilog, VHDL, or SystemC
• Added to your spec or impl DUT in SLEC
• Ports can connect to inputs, outputs, or internal signals of the DUT
• Unconnected ports become primary inputs/outputs
• Add a lot of flexibility
• Example on next slide
FMUL: More fun, adapt with transactors

- Previous slides glossed over some model differences
- Transactors allow us to do that
  - C: concat for long vector
  - RTL: do final summation
- Transactor: similar to
  - Verilog bind feature
FMUL: More fun, adapt with transactors (2)

- The previous slide glossed over another piece of the puzzle
- Transactors again provide solution
- C normalized up front
- RTL does not
- Affects intermediate cuts in manageable way
FMA Proof Decomposition

- FMUL and FMA proof decomposition is the same:
  - Verify the MM
  - Verify the logic outside the multiplier
- This FMA reused same mantissa multipliers already proven equivalent.
FMA thus turned into a variation of FMUL decomposition proof where “the logic outside the multiplier” is different

- Case split on input types, like FADD-DP:
  - $3 \times 3 \times 3 = 27$ cases
  - Reduction exists, but not worthwhile since what is coalesced are fast cases. But they are already fast.
FMA Proof Decomposition

- FMUL and FMA proof decomposition is the same:
  - Verify the MM
  - Verify the logic outside the multiplier
• C model
  – Restoring division (elementary school algorithm)
  – Per iteration: Single bit of quotient + a new remainder

• RTL model
  – Radix 4 SRT (Non restoring division)
    • Per iteration: 2 bits of quotient + a new remainder
    • Redundant representation (signed digit representation)
  – Multiple iterations in a cycle
• Assume-guarantee reasoning based proof decomposition
  – Intermediate maps: all RTL iterations
  – Assume RTL iteration N, to prove iteration N+1
  – Alternate C model iterations skipped for comparison
    • Due to quotient bit generation throughput difference
  – Non restoring to restoring transactor required
    • Required close interaction with designer
• All compare points can be run in parallel
SP FDIV - 2

C Model

opA

opB

norm, scaling+

Partial Quotient

Remainder

Digit Selection, Q&R Update

Partial Quotient

Remainder

Digit Selection, Q&R Update

Maps

Equal?

RTL Model

opA

opB

norm, scaling+

Partial Quotient

Remainder

Digit Selection, Q&R Update

Partial Quotient

Remainder

Digit Selection, Q&R Update

Non-restoring to Restoring Transactor

Non-restoring to Restoring Transactor

Non-restoring to Restoring Transactor

Non-restoring to Restoring Transactor

Non-restoring to Restoring Transactor
Branch Predictor Verification

• Designer driven effort
• Verified using Perf correlation
  – Bugs not functional, but perf related
  – Typically, correlation done late in design cycle
• Aggressive goal to validate Branch Predictor earlier
  – Code a new C model of BP
  – Ensure its equivalence to RTL using SLEC
  – Replace original BP in perf model with new C model
  – Do perf correlation with original perf model
  – Adjust RTL, new C model till correlation satisfactory
Verification complexity

- Structural/algorithmic/abstraction difference between C/C++ model and RTL

- C/C++ model coding style, and specifying cut points for assume/guarantee reasoning
  - SLEC supports a large subset of C/C++
  - But using complicated template C++ functions creates RTL $\Leftrightarrow$ C model mapping complexity
  - If possible, re-write C models in simple form
    - Prove correct once; reuse for each project
## Results

<table>
<thead>
<tr>
<th>RTL operation</th>
<th>CPU-time (single machine / parallel / number of jobs)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>DP</td>
</tr>
<tr>
<td>FCVT</td>
<td>10 mins/n.a./1 per op</td>
<td>DP</td>
</tr>
<tr>
<td>FADD</td>
<td>40 mins/n.a./1</td>
<td>4.5 hrs/10 hrs/9</td>
</tr>
<tr>
<td>FMUL</td>
<td>0.5 hr/2 hrs/45</td>
<td>2 hrs/4.2 hrs/100</td>
</tr>
<tr>
<td>FDIV</td>
<td>2 hrs/12 hrs/9</td>
<td>Under development</td>
</tr>
<tr>
<td>FMA</td>
<td>5.5 hrs/26 hrs/27</td>
<td>Under development</td>
</tr>
<tr>
<td>FSQRT</td>
<td>16 hrs/65 hrs/6</td>
<td>Under development</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>8 hours (sequential)</td>
<td></td>
</tr>
</tbody>
</table>

*After the paper submission, 2 bugs were found on HP FMA*
Methodology benefits

• Eliminate exhaustive simulations for half precision operations
  – 100+ days of CPU time
  – Compute farm saving
• Run automatically at a regular cadence
• Provide bug hunting formal TBs to designers early in design cycle
• And of course, find bugs early.