Efficient and Exhaustive Floating Point Verification Using Sequential Equivalence Checking

Travis W. Pouarz
Travis_Pouarz@mentor.com
Mentor Graphics Corp.
5000 Plaza on the Lake Suite 310, Austin TX 78746

Vaibhav Agrawal
Vaibhav.Agrawal@arm.com
ARM Inc.
5707 Southwest Pkwy #100, Austin TX 78735

***Abstract*- Sequential Equivalence Checking is the process of proving formal equivalence between two non-state matching implementations of a given design specification. Nowhere has the VLSI industry adopted this technology as much as to prove correctness of floating point designs against a given reference model. Any error in floating point operations can have severe consequences. ARM and Mentor have partnered to undertake the formal verification of some of the most difficult floating point blocks, including FMUL, FMA, FDIV, and FSQRT, in their CPU designs. We also worked outside the FPU, on a Branch Predictor from the same CPU designs.**

I. INTRODUCTION

Back in 1995, the Pentium FDIV bug cost Intel $475 million. Today, the industry does not have any appetite for floating-point (FP) bugs. Given this expectation, and the astronomical number of input operand combinations required to exhaustively validate these operations, simulation-based verification cannot suffice. For example, exhaustive simulation of operations requiring two 16-bit Half Precision input operands takes 150 days of CPU time. For operations requiring two 32-bit Single Precision (SP) input operands, exhaustive simulations are impossible, and 64-bit Double Precision (DP) are much worse [1].

Thus, all CPU/GPU design companies have explored Formal Techniques in some form for exhaustive FP verification. Intel has been the pioneer in this domain, developing and using internal tools and technologies. However, in the last decade, with the availability of Sequential Equivalence Checking (SEC) as a Formal Technology [2] from the big-three EDA vendors and other EDA players, the industry has seen a trend toward the use of SEC to prove equivalence of FP designs against C/C++/SystemC reference models for operations (like FADD, FMUL, FDIV, FSQRT, and FDIV).

The FP designs in the high-end Cortex® A-class CPUs from ARM® (e.g. Cortex A72) use interesting algorithms and design techniques to meet aggressive power, performance, and area (PPA) targets. This makes for a strong case for exhaustive verification of these designs using formal techniques. In 2014, ARM began a partnership with Calypto Design Systems (now a division at Mentor Graphics) to explore using **SLEC** (the division's Sequential Equivalence Checking tool) to formally verify these designs. The golden reference models were extracted from an architectural C/C++ model developed and owned by ARM. Over the past 2 years, Mentor and ARM have worked together to prove different FP operations across 4 generations of high end Cortex A-class ARM CPUs (starting with Cortex A72). In the latest generation, so far we have managed to prove Single Precision (SP) and Double Precision (DP) FMUL, SP/DP FDIV, SP FSQRT, and SP FMA.

This paper will introduce a formal sequential equivalence checking tool and present how it is used. It will talk about the models used for comparison. It will talk about the FPU operations, and then discuss in varying levels of detail what we did to perform the equivalence check on FPU format conversions, FADD, FMUL, FMA, FDIV, and FSQRT. We then branch out to describe the formal equivalence check we did on a Branch Predictor from the same CPU design and the results are summarized. Formal verification's place in methodology is touched on and we describe our ongoing work.

II. SLEC, A FORMAL EQUIVALENCE CHECKER

SLEC is a sequential formal equivalence checker that was recently added to the Mentor verification portfolio as part of the Calypto Design Systems acquisition [3].

SLEC is built on a hybrid formal engine that overcomes the limitations of combinational equivalence checkers that require designs to have matching flip-flops.

SLEC detects design differences, if they exist, in minutes, giving designers immediate feedback on RTL changes. Sequential equivalence checking augments system-level and full chip simulations by verifying that new RTL implementations are functionally equivalent to a previously verified reference design.

SLEC verifies micro-architectural optimizations for power, timing and area does not introduce functional side effects. Since formal methods yield high coverage, sequential equivalence checking gives designers confidence to make changes late in the design process. Instead of relying on testbenches or properties, sequential equivalence checking uses a golden RTL model or system-level reference design written in Verilog, VHDL, SystemC or C/C++.

Unlike combinational equivalence checkers, SLEC proves functional equivalence across levels of sequential and data abstraction. Sequential changes like re-timing and resource sharing require only minor changes to setup parameters to reflect new timing information. As a result, sequential equivalence checking boosts verification

productivity. SLEC gives designers the flexibility and confidence to make micro-architectural changes to achieve challenging timing, power, and performance goals.

SLEC can be used in different situations, such as High-Level Synthesis, Power Optimization, and Manual implementations. The first two lend themselves to automated flows, while the last gives the flexibility to compare designs no matter their origin. In the case described here, ARM used SLEC to compare manually created designs.

When only mappings for inputs and outputs are necessary, we call that an end-to-end comparison since only information about the block's inputs and outputs need to be conveyed to the tool.

Sometimes, the formal solvers need more information about internal design points to solve the proof. Much of this paper will describe what we did with select ARM designs to achieve proofs: designs that require significant internal-aware understanding and setup above and beyond the simple end-to-end setup.

Like many EDA tools, SLEC is controlled via scripts written in Tool Command Language (Tcl). A script usually has the basic form shown in Fig. 1– a setup for an adder. This example is an end-to-end demonstration design setup. First, the specification (spec) and implementation (impl) models are compiled and synthesized into an internal model ("build_model"). Then, clock and reset handling are defined. Specific clock period length is unimportant, so the period value supplied is only used for creating waveforms. For SLEC formal verification, time is measured in clock cycles. If the inputs have different names, input mappings tell the tool how these correlate. Finally, the output mapping is used to describe the signal waveforms to compare. In this example, the spec model generates results in the cycle after the inputs are supplied, while the impl model generates results two cycles after the inputs are supplied. By creating waveforms with different latencies derived from each model's output signals, the outputs can then be compared despite the different output latencies.

```
# BUILD MODEL
build_design -spec ectop_add120.h
build_design -impl ectop_add120.v

# DEFINE CLOCK
create_clock -name CLK -period 100
create_map -input spec.clk impl.clk

# RESET HANDLING
create_map -input spec.rst impl.rst
create_constraint -impl -waveform
    RESET_OFF impl.rst
set_reset_value -all -impl -value x
set_reset_value -all -spec -value x

# INPUT MAPPING
create_map -input spec.sc_opA impl.opa
create_map -input spec.sc_opB impl.opb

# OUTPUT MAPPING
create_map -output \
  [create_waveform -sample_start 1 spec.sc_result] \
  [create_waveform -sample_start 2 impl.result]

# VERIFY
verify -mode full_proof
```

Fig. 1. Example SLEC setup.

III. Setup of Models for Comparison

ARM has an architectural C/C++ model ("C model") that it develops and uses for multiple purposes, that has been validated over the years from use on multiple projects. Importantly, the FPU implementation is bit-accurate. This model serves as a reference or specification model ("spec").

SLEC compiles the C model and RTL models and then combines them into a unified transaction model [4].

The C model must be synthesizable – fully elaborated at compile time. We made some minor adjustments for synthesizability, but the bulk of the model was usable without change. The largest change was removing a large behavioral, information-gathering monitor that was not directly involved in actual FPU function.

For each block compared, we created a top-level SystemC wrapper for the C model that provides clock and reset pins so that SLEC can synchronize the C model and RTL design. This created a C spec model where inputs supplied on any given cycle were available on the following cycle.

The RTL was written in standard synthesizable Verilog or SystemVerilog. The hierarchical arrangements of modules are conducive to this verification: each block that we wanted to test was available as a module. While wrappers are optional, they were easy to create and we chose to create them to match the top-level SystemC wrappers per-tested-block. If we had not created RTL wrappers, we would have achieved the same results by using a few more commands in the Tcl SLEC setup.

IV. FPU Operations

IEEE 754 has various classes of operations:

1. Conversions
2. Arithmetic (add, multiply, divide, square root, fused multiply-add, remainder)
3. Scaling and quantizing
4. Comparisons and total ordering
5. Miscellaneous operations that are simple from a formal equivalence checking perspective [5]

Most operations can be tested with a simple end-to-end sequential equivalence check. ARM has a multifaceted verification strategy. SLEC was chosen to complement the existing strategy and so initial applications have been targeted to certain Conversions and Arithmetic operations.

In addition to operands, the blocks have mode control bits that control normal IEEE 754 variations, such as these:

1. Selecting a rounding mode (4 modes are supported for most operations)
2. Turning on "flush-to-zero" mode (ARM architecture uses one Boolean to select the mode for both inputs and outputs)
3. Turning on Default-NaN mode, which controls what NaN value may be returned from an operation

The IEEE 754 floating point number format used by the operations we are most interested in this paper are:

• Single-Precision, also called binary32 in the 2008 standard. It is composed of 32-bits: 1 sign bit, 8 exponent bits, and 23 mantissa bits.

• Double-Precision, also called binary32 in the 2008 standard. It is composed of 64-bits: 1 sign bit, 11 exponent bits, and 52 mantissa bits.

This paper won't delve into the details of the formats. The most significant aspect here is the total width, since the number of bits of input describes the size of the full input state space, and the number of bits of mantissa (also called significand), since that has some effect on how we setup our equivalent checks.

Floating point operations also need to set exception bits, several of them dictated by the IEEE 754 standard, such as invalid operation, division by zero, overflow, underflow, and inexact. In our equivalence checks, we compared both the results and the associated exception bits that the operation generated.

V. Conversions

Conversions are a good example of something straightforward to check. They are good value for formal verification. They are easy to setup. They run quickly. And they provide full coverage. ARM has 8 new converts for a next generation CPU. Running the formal equivalence proofs takes about an hour of wall time altogether. The proofs are done end-to-end. Four of them have 32-bit Single-Precision inputs and four of them have 64-bit Double-Precision inputs. Exhaustive simulation would be prohibitive. At this point, only the new converts were run using SLEC, as the others were already being verified using a mix of simulation and other formal techniques.

VI. FADD

FADD Single-Precision was as easy to setup as a conversion. It ran end-to-end in less than an hour. We also verified FADD Double-Precision end-to-end. The proof for Double-Precision poses additional complexity because of the size. We found that case-splitting on the input space addresses the complexity.

An IEEE 754 floating point number has five non-overlapping ranges (ignoring the sign bit): zero, NaN, infinity, normals, and subnormals. All five ranges taken together comprise all possible combinations.

Zero, NaN, and infinity all tend to energize special-case control paths in FP implementations, so we take them as a group. That leaves us with three ranges: {zero, NaN, infinity}, normals, subnormals.

If each of the two operands can have each of the three ranges, then there are nine total combinations. So, we split the job into nine jobs. The equivalence check (of result and exception bits) succeeded, taking about 5 hours. This type of case-splitting is used again when we get to FMA.

VII. FMUL

Formal Verification of multipliers is well known to be a challenging problem. This is particularly true when the RTL uses a Booth Encoding in a Wallace Tree implementation, as many advanced modern designs do. For Single-Precision and Double-Precision, an end-to-end approach is not successful at finding a full proof in a reasonable amount of time ("does not converge"). To formally verify this block, it needed to be decomposed into a number of sub-problems that, taken together, provide the full check. The technique employed was developed by Mike Case at Calypto [6].

The high-level structure is shown in Fig. 2. The operands are inputs. The multiplication logic is used to multiply the mantissas. There is additional logic that computes the result exponent and sign bits. The verification complexity lies in the multiplier portion of the FMUL and to a lesser extent in the rounding logic. Our approach addresses the full verification of the FMUL, and our focus here will center on the mantissa multiplier and subsequent rounding.

Various tasks may be performed in parallel with the mantissa multiplication path. These include normalization handling, leading zero counting, detection of special cases, and exponent handling. The multiplication logic creates a mantissa product. That product, together with data such as exponent information and leading zero counts (labeled "other stuff" in Fig. 2: data used in a parallel path to the mantissa multiply), is used to create a properly rounded, normalized result. The output labeled "ex" in Fig. 2 is a bit level encoding of the IEEE 754 exceptions that can be raised by the FMUL operation.

The decomposition we use first divides the full problem into two parts: the mantissa multiplier and everything else.

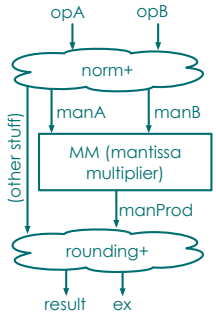

Fig. 2. High-level architecture of the FMUL.

A. Verify Mantissa Multiplication

The ARM mantissa multiplier in this design was a Radix-4 Booth multiplier. This section describes an iterative verification approach that's well suited to Booth multipliers. The iterative approach takes advantage of the incremental nature of the hardware. Each bit of the multiplicand can be tested separately, taking advantage of the equivalence relation established for the previous iteration. For each iteration, only the delta logic that varies when 1 bit of the multiplicand is symbolic needs to be tested.

The description here is intended to give the user enough understanding of how to implement the check and intuition as to why it is correct. The full description can be found in [6].

Fig. 3 shows the approach, using the Double-Precision mantissa size. Consider the block has two operands, *a* and *b*.(corresponding to *manA* and *manB* in Fig. 2). Operand *b* will be left entirely symbolic. Operand *a* will have *n* symbolic bits in the least-significant position, while the most-significant (52 – *n*) bits are forced to be constant 0.

The first case, where *n* = 2, is the base case. The base case is setup as if testing the multiplier in one C-vs-RTL comparison, except that *a* is limited to 2 bits. What was intractable when *a=51* is easily proven when *a=2*.

From that point on, each iterative case that follows can take advantage of the earlier iterations that already were proven correct.

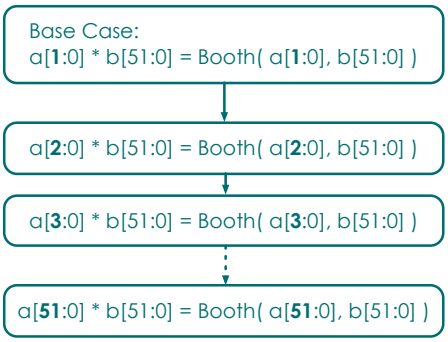

Fig. 3. Iterative approach to FMUL proof.

To take advantage of the iterative cases, consider the breakdown in Fig. 4. After the base case, each iteration on *n=N* will assume that the previous iteration (*n=N* – 1) has been proven. That is, for our Goal (referring to Fig. 4), we will assume that Assume is true. Lemmas 1, 2, and 3 are a splitting of Goal into three equations: the left-hand-side (LHS) of Goal is the same as the LHS of Lemma 1. The right-hand-side (RHS) of Goal is the same as RHS of Lemma 3. Using substitution, Lemma 1, 2, and 3 can be combined to re-create Goal.

```
Goal (for a given N):
a[N:0] * b[51:0] = Booth( a[N:0], b[51:0] )

Assume:
a[N-1:0] * b[51:0] = Booth( a[N-1:0], b[51:0] )

Lemma 1:                    easy word-level problem
a[N:0] * b[51:0] =
      (a[N] * b[51:0]) << N + (a[N-1:0] * b[51:0])

Lemma 2:                    trivial — vacuously true
(a[N] * b[51:0]) << N + (a[N-1:0] * b[51:0]) =
     (a[N] * b[51:0]) << N + Booth( a[N-1:0], b[51:0] )

Lemma 3:                    manageable bit-level problem
(a[N] * b[51:0]) << N + Booth( a[N-1:0], b[51:0] ) =
     Booth( a[N:0], b[51:0] )
```

Fig. 4. Decompose iterations into checkable lemmas.

When the three Lemmas are true, we know that the Goal is true. This breakdown of Goal is useful since the Lemma equations use N-1 and allow us to use our Assume equation.

To prove an iteration of case *n=N*, the three Lemmas must be checked.

Consider first Lemma 2. When substituting the values from the assumption, both sides of the equation are identical. Thus, given the assumption, Lemma 2 is vacuously true. There is no further check required.

For Lemma 1, only the "*" operator appears: the multiplication in the C model. "Booth" does not appear. To check this Lemma, the C model is checked against itself. The spec model is the C model where the low *N* bits of operand a_{spec} are symbolic: the most significant of those is symbolic bit S_m (a_{spec}[N:0] = {S_m, S[N-1:0]}). The impl model is the C model where the low *N* – 1 bits of operand a_{impl} are symbolic, and a_{impl}[N] is zero (a_{impl}[N:0] = {0, S[N-1:0]}). If the result were compared as-is, then it would falsify if S_m takes the value 1. However, if the model is implementing a multiplication, it will embody Lemma 1 and we can add a transactor that allows us to create comparable results (as shown in Fig. 5):

$final\_result_{spec} = manProd_{spec} = a_{spec} * b$

$final\_result_{impl} = (S_m * b) << N + manProd_{impl} = (S_m * b) << N + a_{impl} * b$

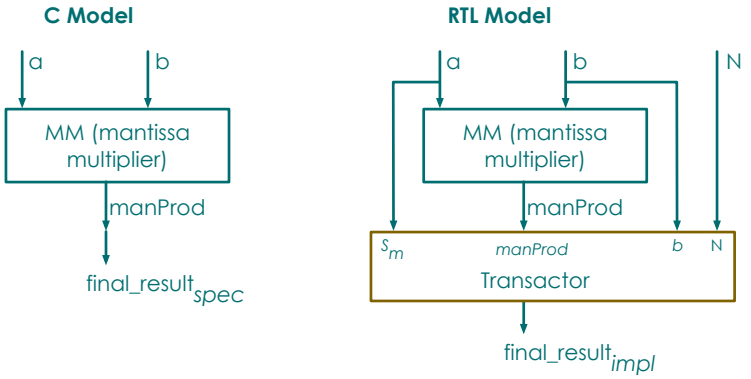

Fig. 5. Iterative approach to FMUL proof.

Transactors are a SLEC feature that allows the user to add a module in Verilog, SystemVerilog, VHDL, or SystemC into the spec or impl design model to add logic to the design for verification purposes without modifying the original design code. Transactors are built in SLEC like any other design data and then they can be instantiated with a Tcl command in either model. When instantiated in a design model, the I/O ports can be attached to internal signals of that model, and unattached ports become additional primary (top-level) I/O for the design. This feature provides enormous flexibility and has a multitude of uses.

In this case, $final\_result_{spec}$ is already available as the spec signal $manProd_{spec}$. A transactor is defined and an instantiation of it is created and bound to the impl model. The inputs of the transactors are attached to impl design signals: S_m, a_{impl}, and b (N is set as a Verilog parameter). The output is left open and thus becomes a primary output. That output has the value $final\_result_{impl}$ and it is what we map to $manProd_{spec}$.

Lemma 3 is tackled in the same way as Lemma 1 except that we are using the RTL model rather than the C model.

The base case is checked with $n = 2$. Lemma 1 and Lemma 3 are checked for each $n = 3..51$ value. This means that 99 separate SLEC jobs are required to prove the equivalence of the mantissa multiply double-precision block. Fortunately, most of these jobs are very quick and SLEC allows parallelism of these checks on an LSF or other grid system as well. So the total time is very manageable: around 4 hours.

B. Verify Mantissa Multiplication: about that Partial Product Reduction

What is described so far worked well for single-precision FMUL. For double-precision, SLEC was taking too long to produce a result. The Booth multiplier has two parts: booth encoding and then a series of shifts and adds of the partial products, similar to a Wallace tree. The hardware implementation foiled the structural similarity that was expected to make Lemma 3 so easy [6]. To handle this, the partial product reduction RTL circuit was compared with an alternate RTL implementation made up of simple "+"-operator sums, an RTL-vs-RTL equivalence check. Once that was proven equivalent with SLEC, Lemma 3 was run using the alternate implementation. This substitution reduced the total time from many hours to a few minutes for the longest Lemma 3 checks.

C. Verify Everything Else

To check everything else, we had to create a setup that would abstract away the complexity of the mantissa multiplier. We could not simply blackbox it, because there were pieces of data (labeled "other stuff" in Fig. 6) derived from the original inputs that went into the result calculation that had to be consistent with the multiplier manProd output. Fig. 6 shows the setup:

1. Compare the inputs to the mantissa multiplier (*manA* and *manB*).
2. Since the multiplier was proven equivalent under equivalent inputs, tell SLEC to drive *manProd* in one model from *manProd* in the other model. This is the key step in reducing the complexity to something solvable.
3. Compare the result outputs and the exception-bits outputs.

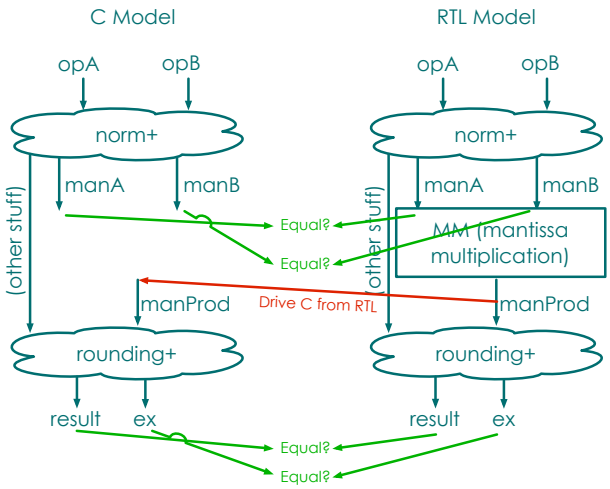

Fig. 6. Check everything else.

D. Transactors to handle design differences

Up until now, we skipped discussion about how the models were not as closely aligned in implementation as shown above. In particular, the C model normalized subnormal inputs prior to multiplication while the RTL multiplied the mantissas directly without that step. This means that, given any subnormal inputs, the inputs to the mantissa multiply are different in each model. And, given different inputs, the outputs would also be different.

To handle this situation, rather than directly map the mantissa multiply inputs and outputs, we needed to create transactors to create equivalent values that could then be mapped.

In this case, the transactors are instantiated into the RTL impl design. The inputs of the transactors are attached to impl design signals. The outputs are left open and thus become primary outputs. Those outputs are then what are mapped to signals in the C models. The mappings are shown by small capital letters A, B, C in Fig. 7.

The multiplication operand inputs are shifted by their leading zero count (LZC) in the C model during normalization (subnormal numbers are transformed into normal numbers since additional mantissa bits are available internally to the calculation). So, to compare them, we shift the RTL by each input's respective LZC, a value that's already calculated in the RTL anyway, so we could just tie the internal signal holding that value to a transactor input.

The mapping of the output may not be immediately obvious though. The RTL model multiplies:

$$z_r = a \times b. \tag{1}$$

The C model multiplies the inputs after shifting to generate the output:

$$z_c = (a \ll g) \times (b \ll h). \tag{2}$$

A base-2 shift can be transformed into a multiplication by 2 to the power of the shift amount:

$$z_c = \left(a \times 2^g\right) \times \left(b \times 2^h\right) \tag{3}$$

Use associative property of multiplication:

$$z_c = \left(a \times b\right) \times \left(2^g \times 2^h\right) \tag{4}$$

Use (1) as a substitution and also transform using product of powers rule:

$$z_c = (z_r) \times (2^{g+h}) \tag{5}$$

Multiplication by a power of 2 can be converted into a shift:

$$z_c = z_r \ll (g + h) \tag{6}$$

It turns out, to create comparable mantissa multiply output values, we need only shift the RTL mantissa multiply output by the sum of the LZC from each operand.

Fig. 7 shows what our models look like with the transactors in-place. The points labeled with capital "A" and "B" are the mantissa multiplier inputs that can be mapped, because the transactors create equivalent values. The capital "C" is the mantissa multiplier outputs that can be mapped.

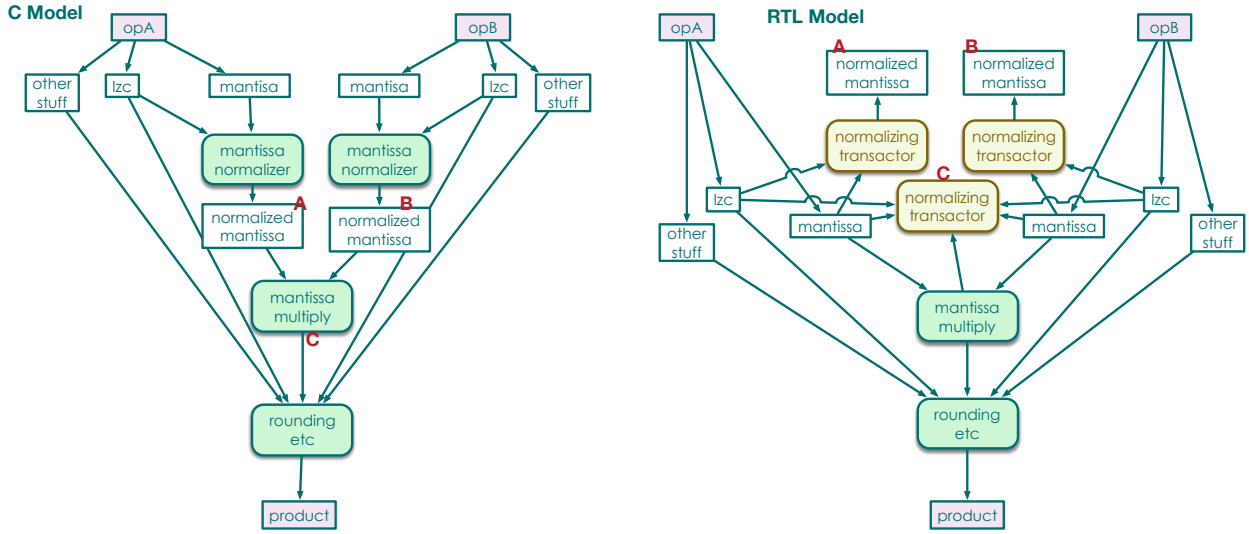

Fig. 7. FMUL with transactors for comparison.

VIII. FMA

For this design, the FMA was able to leverage a lot of work that had gone into verifying FMUL. The designs are implemented by re-using the FMUL mantissa-multiplier, in both models. The FMUL verification checked that same block. That means that the FMA can be viewed as the same check as FMUL, except that the "verify everything else" job is more complicated for the SLEC solvers. Fig. 8 shows the similarity between the two operations.

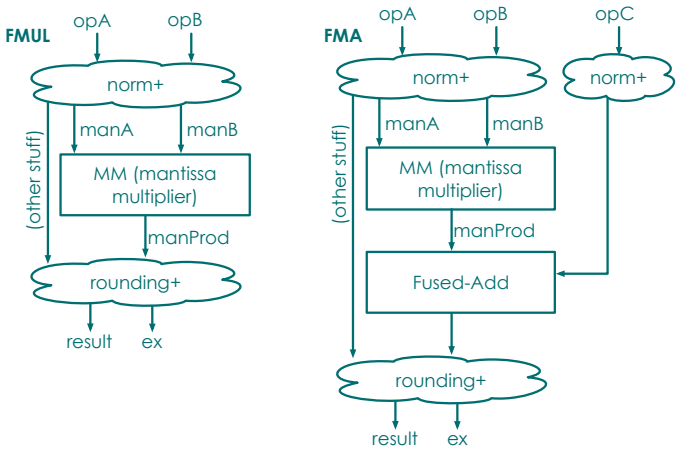

Fig. 8. FMUL and FMA have many similarities.

In Fig. 9, the "verify everything else" setup for FMA is shown. All the comparison points are the same as FMUL. The big difference is that the 3rd operand fused-add takes place after *manProd* and before rounding. This was not quite sufficient for FMA single-precision. To get a good turn-around, we case-split this using the same 3-

class split on the inputs as we used in FADD. This time, there were three inputs rather than two, so the total number of splits this added was 27. Most combinations ran in minutes, a few took up to an hour, and one took 5.5 hours.

That was success for single-precision. Double-precision still needs to be tackled and we expect that some additional technique will be necessary.

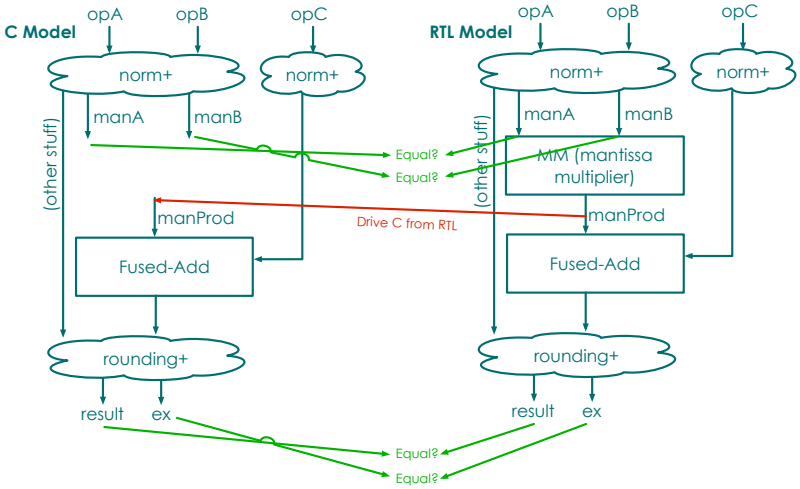

Fig. 9. FMA "verify everything else" setup.

IX. FDIV

Division presented a very different type of design. Both C and RTL models execute an iterative digit-recurrence algorithm. The C model does a restoring division. For performance, the RTL does a non-restoring radix-4 digit-recurrence using signed-digit representations.

SLEC needs intermediate mapping points. Fig. 10 shows the maps that are set up for this verification. The maps are used in an assume/guarantee manner, so they are tested for equivalence, and then assumed equivalent for downstream checks. In Fig. 10, the green arrows show all the maps in the design (with only a few iterations shown for legibility). The bold green maps show those being compared for equivalence in the first iteration.

Since the algorithms were different, the values at these intermediate points are different. The designer provided us with important input about how we could convert intermediate partial remainders and quotients from the RTL to match the corresponding values in the C model. We created Verilog transactors to do the conversion.

Fig. 11 shows an example of how a single iteration is checked. The maps of the upstream partial product and remainder are assumed proven and SLEC uses that to simplify the model. Then the maps of the next partial product and the maps of the next remainder are checked for equivalence. The checks for this iteration can be done in parallel with the checks for all other iterations. The checks of partial product and of remainder can be done in parallel as well.

Each iteration calculates 2 bits of quotient. For single-precision, there are 24 mappable intermediate values plus 2 outputs (result and exception bits), for 26 total mappings that can be checked in parallel. For double-precision, there are 56 total mappings.

As with FADD-DP and FMA-SP, we also case-split based on different inputs. In addition to the performance improvement, this was also needed because the latency of the FDIV varied by case. So, there were cases that involved early-termination (such as when at least one operand is zero, NaN, or infinity, etc.), normal operands, or subnormal operands.

For FDIV-DP, we were looking for more splits to improve verification performance. There was a variable-shifter and a larger adder related to scaling. The algorithm would scale the inputs by a scaling factor S:

$$Result = A / B = (A / S) / (B / S). \quad (7)$$

The scaling factor division is not actually performed by doing a general-purpose division as shown in (1) but by having a select set of scaling factors that can applied via shift-and-add operations [7].

Since there are a small, finite number of scaling factors, we could case-split on each of the factors. That meant that each of those cases would no longer need the variable shift-and-add since it would be a constant for any given case. This worked well for Double-Precision normal numbers. We are currently applying it to subnormal numbers.

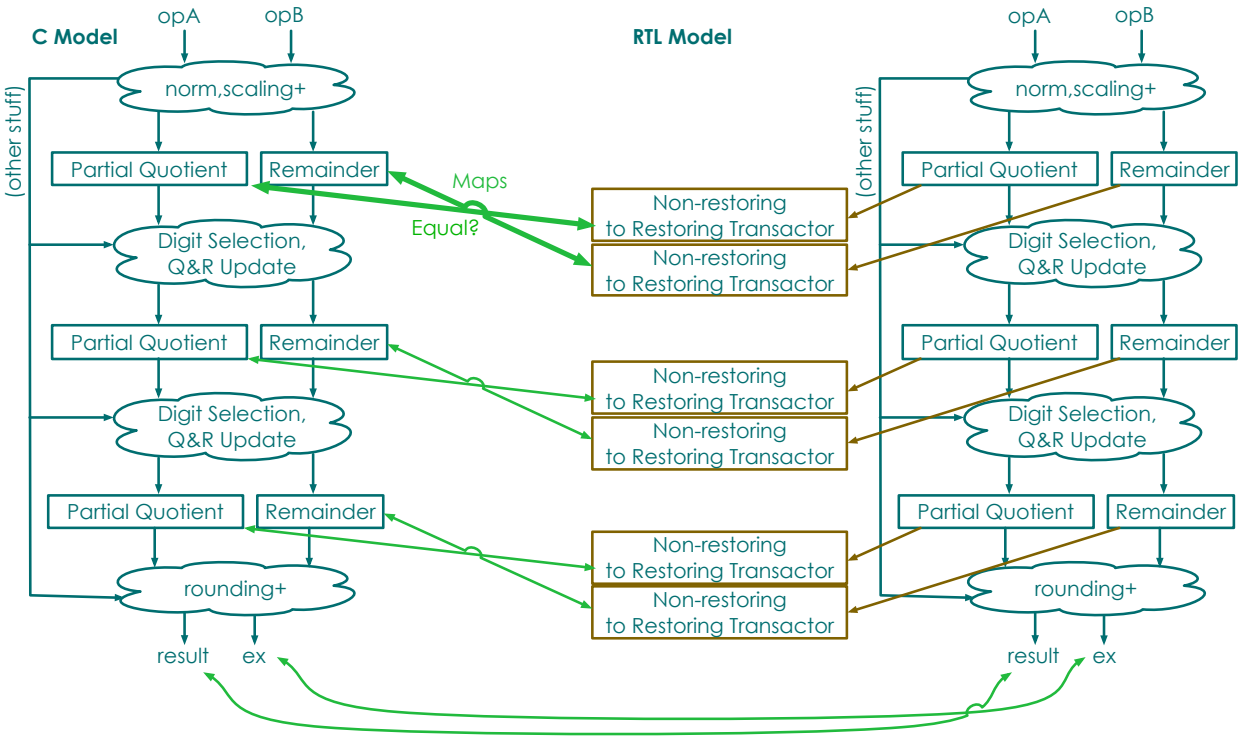

Fig. 10. FDIV complete setup.

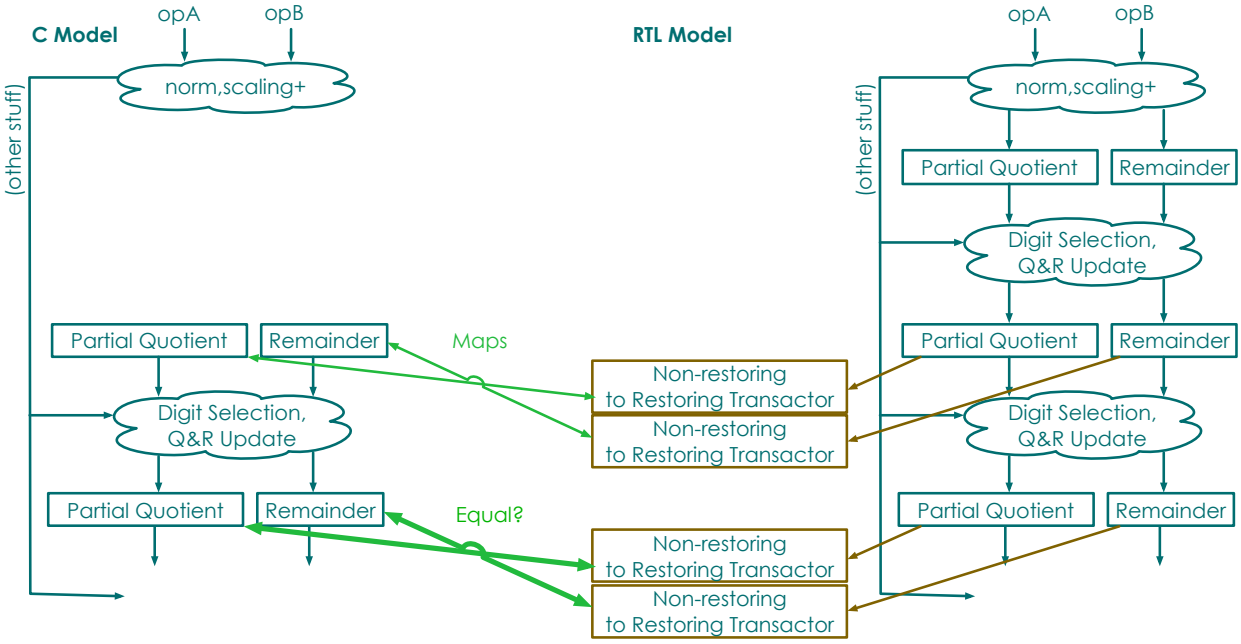

Fig. 11. FDIV single-iteration mapping.

X. FSQRT

FSQRT is similar to FDIV. Both C and RTL models execute an iterative digit-recurrence algorithm. The C model's algorithm is akin to "restoring." For performance, the RTL does a method similar to non-restoring radix-4

digit-recurrence using signed-digit representations. The algorithm calculates partial root and remainder intermediate values for each iteration. The main difference from FDIV is in the partial-root calculation per iteration, which was more complex for FSQRT. Again, we had to work closely with the designer to be able to discover the functional mapping between the C model intermediate values and the RTL model intermediate values.

XI. Branch Predictor Verification

In addition to the arithmetic, SLEC can also be used for control dominated designs. While doing the previously discussed floating-point work, ARM and Mentor also worked together to verify a new branch predictor (or Global History Buffer – GHB) design.

The GHB algorithm was developed and tuned by measuring the performance results in the C++ CPU performance model.

A. Our Branch Prediction verification strategy

Branch Predictor (GHB) bugs do not lead to inconsistent state or functional bugs in the CPU, but rather affect the performance. To verify a Branch predictor, typically, performance correlation is done between performance model and RTL by running simulations. However, perfect correlation between RTL and the CPU model is very difficult because a C++ model typically does not implement all the micro-architectural level details (e.g. pipeline stages). Also, other design components (e.g. cache replacement behavior) can change the detailed timing. However, for this type of correlation to be reasonably accurate, it has to be done relatively late in the design cycle.

For this design, an aggressive goal was set to see if the Prediction algorithm could be tested using SLEC early in the design cycle. For various reasons, the original C model of the Branch Predictor used in the performance model was not suitable for this purpose. Therefore, an alternate C model was written for the GHB, and it was verified against the RTL using SLEC (with RTL being the golden reference in this case). This alternate C model was then used to replace the original GHB C model in the overall performance model, and performance was measured. If the performance deviation was found to be statistically significant, then the alternate C model was tuned until the deviation was within acceptable limits. Finally, the RTL was modified to match this modified alternate C model. A few issues in the RTL were found and fixed using this approach.

GHBs have huge arrays to store branch history. To reduce the verification complexity for SLEC, we modified our RTL and alternate C model to parameterize the SRAMs, to enable a smaller SRAM size for SLEC-based verification. Reducing the SRAMS to 16 sufficed.

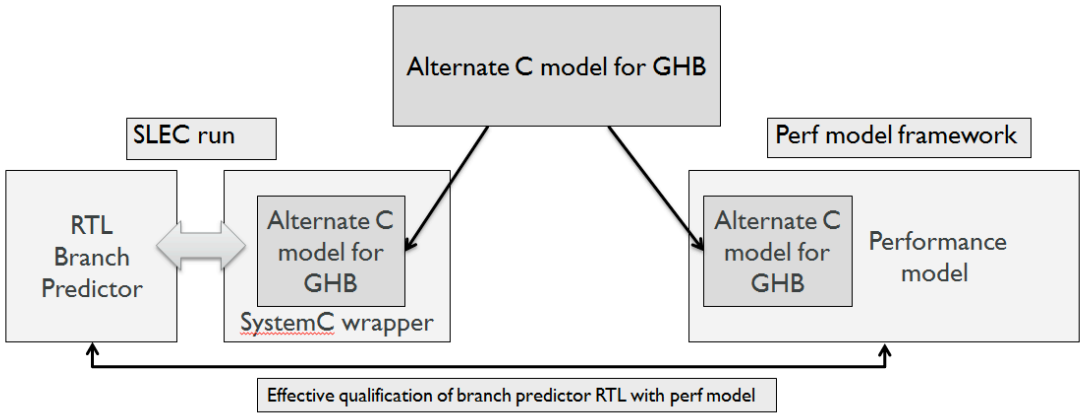

Fig. 12. Branch Prediction Verification Strategy.

XII. Results

Here, we present some information that might be of interest to a potential SLEC user, trying to use the tool for proving designs like those mentioned in this paper. It involved a verification engineer who was not familiar with the specifics of the C or RTL design or the micro-architecture. The proof development time refers to all the effort that it takes to compile C model, doing a deep dive into RTL to understand the design, experiment and develop a

proof decomposition strategy, and writing any alternate C models or transactors. So, much of this knowledge was gained in-flight during the verification exercise. Subsequent attempts at verifying similar logic in a derivative would be much faster.

The elapsed time is the wall clock time (achieved after different intermediate maps or output maps are submitted over the compute farm). The sequential CPU time is the cumulative CPU time of all the jobs running in parallel.

RTL function/operation	Proof development time	CPU-time (parallel / sequential)	
		SP	DP
FADD	2 days	40 minutes (sequential)	4.5 hours / 8.5 hours
FCVT	1 day	10 minutes (sequential)	
FMUL	2 months	0.5 hour / 2 hours	2 hours / 4.2 hours
FDIV	2 months	2 hours / 12 hours	20 hours (normals)
FMA	1 month	5.5 hours / 26 hours	Under development
FSQRT	2 months	16 hours	Under development
Branch Predictor	1 month	8 hours (sequential)	

XIII. Formal Verification in the Methodology

As a result of our work, we have been able to eliminate exhaustive simulations to prove Half Precision floating point operations, as most of these can now be proved using simple end-to-end setups, whereas some require some minor proof decompositions. This results in significant compute farm savings, which is monitored closely at ARM.

Also, we are in the process of setting up automated runs for all the proofs so that any relevant changes can trigger these runs, removing manual intervention. This has resulted in a lot of efficiency.

Finally, we plan to build end-to-end setups for all the operations and provide these to the designers so that they can run them for bug hunting while doing design, even if the full proofs do not converge. This will be helpful for design bring up. This will also complement the parallel work of proof development, which takes longer to complete, but is the goal for the floating-point operations.

IX. Future Work

We are continuing to expand the number of FP blocks that are formally verified, and refining and enhancing the complex setups for FMA and FDIV. We are considering applying this methodology to verify ARM's instruction decode unit (or parts of it) in the next project.

X. Acknowledgements

We would like to thank Mike Case (Mentor Graphics) and Naren Narsimhan (Mentor Graphics) for their help and consultation throughput this project. We would also like to thank Yasuo Ishii (ARM) for his work on Branch Predictor Verification using SLEC, and also David Lutz, Kelvin Goveas and Javier Bruguera for providing insights into the Floating Point microarchitecture.

References

[1] J. Harrison, "Formal verification of floating-point arithmetic at Intel," 02 06 2006. [Online]. Available: http://www.cl.cam.ac.uk/~jrh13/slides/jnao-02jun06/slides.pdf. [Accessed 08 11 2016].

[2] A. Koelbl, R. Jacoby, H. Jain and C. Pixley, "Solver technology for system-level to RTL equivalence checking," *DATE '09 Proceedings of the Conference on Design, Automation and Test in Europe,* pp. 196-201, 20 04 2009.

[3] Mentor Graphics Corp, "Mentor Graphics Acquires Calypto Design Systems," Mentor Graphics Corp, 16 09 2015. [Online]. Available: https://www.mentor.com/hls-lp/news/mentor-acquires-calypto-design-systems. [Accessed 04 11 2015].

[4] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur and N. Sharma, " Non-cycle-accurate sequential equivalence checking," *DAC '09 Proceedings of the 46th Annual Design Automation Conference,* pp. 460-465, 26 07 2009.

[5] Wikipedia, "IEEE floating point," Wikipedia, 21 10 2016. [Online]. Available: https://en.wikipedia.org/wiki/IEEE_floating_point.

[Accessed 07 11 2016].

[6] M. Case, "Formal verification of booth multipliers". USA Patent US20140067897 A1, 06 03 2014.

[7] M. D. Ercegovac, T. Lang and P. Montuschi, "Very-high radix division with prescaling and selection by rounding," vol. 43, no. 8, pp. 909-918, 06 08 2002.