Formal Verification of Floating-Point Division

Ashish Kapoor
Synopsys Inc.
Hillsboro, OR, USA
akapoor@synopsys.com

Warren Ferguson
Synopsys Inc.
Hillsboro, OR, USA
warrenf@synopsys.com

Himanshu Jain
Synopsys Inc.
Hillsboro, OR, USA
hjain@synopsys.com

Sudipta Kundu
Synopsys Inc.
Hillsboro, OR, USA
sudiptak@synopsys.com

***Abstract*—Verification of complex datapath circuits such as floating-point dividers are known to be a challenging problem. In this paper, we present a formal verification methodology to verify floating-point (FP) dividers. In general, floating-point division unit builds around a fixed-point division implementation. Our solution performs a two-step verification.**

The first step verifies the fixed-point division implementation. We target fixed-point division algorithms that compute a fixed number of quotient bits in each iteration. This step uses a combination of equivalence checking and assertion-based property checking techniques. We used property checking to show correctness of radix-2 restoring division and equivalence checking to show equivalence between the radix-2 restoring division and a prescaled radix-4 non-restoring division.

The second step uses equivalence checking to compare the floating-point divider with a software golden reference of FP division. In this step we assume the fixed-point division is working correctly to make the proof tractable. Using the proposed steps, verification of single precision FP divider took 1 hour 30 minutes and double precision FP divider took 7 hours and 30 minutes.

Keywords—formal verification, equivalence checking, datapath verification, floating-point division.

I. INTRODUCTION

Circuits that perform division and that compute a square root are used extensively in hardware designs. There have been many cases in which a bug in a circuit implementation of a mathematical operator had a significant impact on the company's finances. For example, in one well-publicized instance, a bug in a floating-point division circuit cost the company hundreds of millions of dollars [1]. Therefore, it is very important to guarantee the correctness of these operators.

Traditional verification testing techniques such as random testing and scenario-based testing do not scale well and do not guarantee completeness. Formal verification has emerged as an alternative to ensure the correctness of hardware designs and overcoming several of the limitations found in traditional testing techniques. Moreover, advances in SAT solvers, equivalence checkers, and model checkers have allowed engineers to formally verify many properties of floating-point designs.

Verification of fixed-point division and square root algorithms has been an active area of research. One of the recent publications in this area by Melquiond et al. [6] require expertise in automated solvers such as Gappa. Another publication by Scholl et al. [2] presents a solution using Symbolic Computer Algebra (SCA). Expertise in SCA is required for extending this technique. In the paper they state that:

We were able to prove that the canonical polynomials for pseudo-boolean functions occurring at cuts between stages have exponential sizes.

They were able to avoid the exponential behavior through stronger propagation and exploitation of constraints for a very specific implementation of division. These techniques would require significant changes on other variations of the division algorithm such as saving the result in a carry save format.

Theorem provers have been widely used to prove correctness of systems. They are either fully automatic: SMT solvers, Alt-Ergo, CVC5 and Gappa; or interactive: ACL2 and Coq. For example, Gappa [6] has been used to prove that a 28-line C program computes the square root of an integer. In another work, Russinoff [7] has provided a mechanical mathematical proof of the correctness of the FSQRT using the ACL2 theorem prover. Harrison [10] has formally verified various IA-64 floating-point and integer division algorithms using the HOL Light theorem prover. These techniques require an expert user driving a theorem prover. As the RTL changes it becomes challenging to update the proofs.

Verification of floating-point divider implemented in RTL against the SoftFloat C model was successfully verified in [8]. They verified single precision division of the DesignWare [11] model which used a division operator inside FP RTL. Our work extends this to verification of double precision radix-2 restoring division and a prescaled radix-4 non-restoring division.

As stated earlier, the proposed solution performs a two-step verification. The first step verifies the fixed-point division implementation. The second step verifies the floating-point logic by comparing it with the SoftFloat [3] C model.

The fixed-point division verification has been extensively studied [2][6][7]. All the research in this area has focused on verification of the divider as a single unit. There are multiple division algorithms and optimizations which make this a very hard problem. Our approach is unique in the way that it requires the user to identify the partial quotient and remainder after each iteration. This breaks the problem down into smaller sub-problems that can be verified by current state-of-the-art solvers. Proposed approach applies to any iterative division algorithm that computes fixed number of quotient bits in each iteration.

Once we have verified the FP radix-2 restoring division we then use it as the specification for the verification of the FP prescaled radix-4 non-restoring division implementation that is used inside the FP divider. We make the equivalence checking problem tractable by breaking down the problem into a series of internal equivalence problems, one for each partial quotient and remainder after each iteration.

Since the introduction of SRT dividers in 1958 [4], significant enhancements have been made in the design of fixed-point dividers. See [5] for more details. In this paper redundant quotient values and prescaling are used to improve performance. Significant mathematical analysis is done to identify the next quotient bit by using part of the divisor and part of remainder in carry save format. An error in this computation will result in design bugs that will be hard to catch using random or targeted simulation.

Further, even if the fixed-point division is formally verified, there are a lot of special cases associated with floating-point arithmetic. The IEEE standard defines handling of special cases that include positive and negative versions of NaN, Infinity, and Zero. Also, there are five exception flags defined in the IEEE standard: Invalid operation, Division by Zero, Overflow, Underflow, Inexact calculation.

This paper is organized as follows. In the next section we give a short introduction to the SoftFloat library and briefly explain the reference golden model. This is followed by a description of our implementation of the radix-2 restoring division and the prescaled radix-4 non-restoring division. Next, we present our verification flow for both these dividers. Finally, the experimental results of this work are presented.

II. REFERENCE MODEL (SOFTFLOAT LIBRARY)

SoftFloat is a software implementation of binary floating-point that conforms to the IEEE Standard for Floating-Point Arithmetic. It supports the floating point formats that our proposed flow verifies: 32-bit single precision, and 64-bit double precision. It supports the five rounding modes specified in the IEEE standard, and can be configured to emulate Intel x86 and ARM architectures, among others. See [3] for more details.

The SoftFloat library was chosen as a reference model because it has been around for over two decades. During this time, it has been extensively used by the community and all the identified issues have been addressed. SoftFloat library function f32_div implements the division algorithm. This function was instantiated in a top-level C function as follows:

```
// inputs
float32_t op1; float32_t op2; uint8_t rndmode;
// outputs
float32_t res; uint8_t   flags;

softfloat_detectTininess =
softfloat_tininess_afterRounding; //IEEE TINY signal
softfloat_roundingMode = rndmode; // rounding mode
softfloat_exceptionFlags = 0;     // initialize to 0
res = f32_div(op1, op2);          // do the division
flags = softfloat_exceptionFlags; // extract flags
```

III. IMPLEMENTATION OF DIVISION

Details of the two divider implementations are as follows. Figure 1 presents a typical floating-point division datapath. The PREOP block unpacks the inputs, normalized denorm mantissa values, and detects if the operands are exceptional.

For exceptional cases the results (both value and exception flags) are predefined, and the flow passes directly from the PREOP block to the POSTOP block.

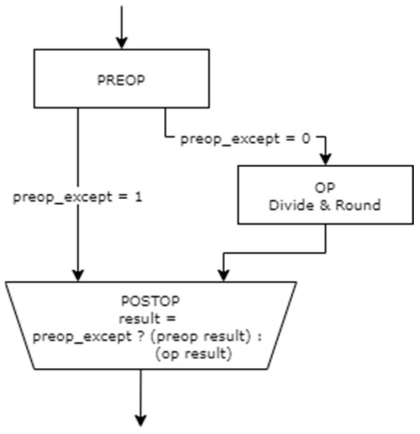

Figure 1: Floating-point division block diagram.

A. *Implementation of radix-2 restoring division*

The fixed-point radix-2 division is define in the OP block.

Consider the computation of the quotient X/Y of two floating-point operands of precision p, e.g., $p=53$ for double precision. The normalized representation of X and Y are:

$$X = (-1)^s 2^e N \quad \text{and} \quad Y = (-1)^t 2^f D$$

And $\qquad N \in [1, 2) \quad \text{and} \quad D \in [1, 2)$

So $\qquad X/Y = (-1)^{s-t} 2^{e-f} N/D \quad \text{where} \quad ½ < N/D < 2$

Given that the p-bit significands N and D have $(p-1)$-bit fractions, their binary points can be removed without changing N/D; this allows us to treat N and D as p-bit unsigned integers.

For any nonnegative index j, integer long division tells us that there is a pair of integers Q (quotient) and R (remainder) which satisfy the invariant:

$$2^j N/D = Q + R/D \qquad \text{[Equation 1]}$$

While there are an infinite number of integer pairs $\{Q, R\}$ that satisfy this invariant, long division produces the unique integer pair $\{Q_j, R_j\}$, called the standard pair for stage j, for which $0 \le R_j < D$. It follows from $½ \le N/D < 2$ and $0 \le R_j < D$ that Q_j is a $(j+1)$-bit unsigned integer, so $Q_j / 2^j$ is a significand with a j-bit fraction whose integer bit is 0 when $N < D$ and 1 otherwise, and R_j is a p-bit unsigned integer. When rounding the quotient X/Y to precision p, it suffices to obtain $\{Q_{p+1}, R_{p+1}\}$ because Q_{p+1} always includes the round-bit while R_{p+1} provides the additional information needed to form the sticky-bit.

From the invariant for stages j and $j+1$ we find that $\{Q_j, R_j\}$ satisfy the radix-2 digit-recurrence

$$Q_{j+1} = 2Q_j + d_{j+1} \qquad \text{[Equation 2]}$$

$$R_{j+1} = 2R_j - d_{j+1}D \qquad \text{[Equation 3]}$$

where digit $d_{j+1} \in \{0, 1\}$. Equations 2 and 3 are implemented by the OP block.

B. Implementation of prescaled radix-4 non-restoring division

The division block diagram is similar to Figure 1. The main difference is in block OP. This was changed from radix-2 restoring fixed-point division to prescaled radix-4 non-restoring fixed-point division. The division implementation follows the architecture proposed in [5].

Any integer pair $\{q_j, r_j\}$ that satisfies Equation 1 is related to the standard pair $\{Q_j, R_j\}$ as follows:

$$\{Q_j, R_j\} = \{q_j - c, r_j + cD\} \qquad \text{[Equation 4]}$$

where c is the unique integer for which $0 \le r_j + cD < D$. Typically, $-D \le r_j < D$ and so $c = (r_j < 0)\ ?\ 1 : 0$.

Consider a higher-radix division with radix $\beta = 2^m$ and the digit set Φ_b composed of all integers whose magnitude is at most b. As there are at least β digits required, $(\beta\text{-}1)/2 \le b \le \beta\text{-}1$. The radix-$\beta$ digit-recurrence generates the integer pairs $\{q_j,r_j\}$:

$$q_{j+m} = \beta q_j + d_{j+m} \qquad \text{[Equation 5]}$$

$$r_{j+m} = \beta r_j - d_{j+m}D \qquad \text{[Equation 6]}$$

where the digits $d_{j+m} \in \Phi_b$. The computation of $\beta r_j - d_{j+m}D$ takes the form of an integer multiply-add, so implementations often use redundant representations of the remainder to shorten this datapath. Figure 2 below presents a procedure that implements one stage of radix-β digit-recurrence division.

```
Procedure SingleStage
  Inputs: $q_j$, $r_j$, $D_j$
  Outputs: $q_{j+m}$, $r_{j+m}$, $D_{j+m}$
  Digit selection: Choose $d_{j+m} \in \Phi_b$
  $q_{j+m} = \beta q_j + d_{j+m}$
  $r_{j+m} = \beta r_j - d_{j+m}D_j$
  $D_{j+m} = D_j$
End Procedure
```

Figure 2: One step of radix-β digit-recurrence division.

This digit-recurrence algorithm assumes that:

$$| r_j / D | \le \rho \qquad \text{[Equation 7]}$$

where $\rho = b / (\beta - 1)$ is the so-called redundancy factor; we assume that $½ < \rho \le 1$. So, for a specific range of values of D, procedure *SingleStage* must have the following property:

If: $| r_j | \le \rho D_j$ and $D_j = D$

Then: $| r_{j+m} | \le \rho D_{j+m}$ and $D_{j+m} = D$ [Property 1]

The proposed implementation in [5] uses $\beta = 2^2 = 4$ and $b = 2$ with carry-save remainder, hence the redundancy factor $\rho = 2/3$. It also relies on operand scaling. Without operand scaling, the delay associated with a radix-4 step would be twice compared to a radix-2 step and hence would not result in a performance advantage. With the proposed operand scaling, quotient digit selection is independent of the divisor and only requires 6 bits of the assimilated partial remainder. According to [5], this results in a speedup of 1.20 to 1.45.

IV. Proof Methodology

For verification, we use the Synopsys VC Formal DPV [9] tool. DPV is primarily a C++ to RTL transactional equivalence checker. It also has proprietary specialized commands to aid a user to verify an implementation of fixed-point division. In our work we have used both the above features.

A. Verification of radix-2 restoring division

Our solution uses a two-step approach to verify the division algorithm. These steps are:

1) Verify the fixed-point division

The fixed-point division in the OP block in Figure 1 is implemented using the radix-2 restoring division. This is verified by using command solveNB_division in the DPV tool. The user interface of this command is:

```
solveNB_division k <getDividend> <getDivisor>
  <getQuotient> <getRemainder>
  <inputAssumptions> <identifier>
```

The functions getDividend/getDivisor are used to specify dividend and divisor signals respectively. The functions getQuotient and getRemainder identify the quotient and remainder after each iteration. The tool generates a series of properties (lemmas) checking that quotient/remainder is correct at each iteration.

2) Verifying the floating-point division

Once the fixed-point division was verified, it was replaced by the division and modulo primitives using the command solveNB_division_assumptions in the DPV tool. Next, the rest of the floating-point division functionality was verified using equivalence checking. This included the exception flags as described in the motivation section. The SoftFloat division algorithm was used as a reference model for this verification.

The C implementation of the division algorithm in SoftFloat was making the equivalence checking problem hard as there was no internal equivalence point with respect to the RTL. To address this issue, we made a change to the SoftFloat library. The remainder and the sticky bit were computed in the original code as follows:

```
sigZ = sig64A / sigB;
if (! (sigZ & 0x3F))
  sigZ |= ((uint_fast64_t) sigB * sigZ != sig64A);
```

The lower 6 bits are the rounded remainder that are finally used to compute the sticky bit. This was modified as follows:

```
sigZ = sig64A / sigB;
sigR = sig64A % sigB;
sigZ = sigZ << 6 | (sigR != 0);
```

Proof sketch for the above change: If N is the numerator, D is the denominator, Q is the quotient, and R is the remainder. The remainder can be computed as:

```
R = N - D * Q = N % D
```

The original code was computing the remainder using the first method while the new code is using the second method. Both methods will produce the same result. A detailed proof has been omitted because of lack of space.

B. *Verification of prescaled radix-4 non-restoring division*

In the previous step, the FP radix-2 restoring division was proven to be correct. Hence it was used as a reference model for this verification.

The proposed solution compared results after each iteration of the division algorithm. This created a problem because in the FP prescaled radix-4 non-restoring division:

1. The operands were scaled.
2. The results were stored in a carry-save form.
3. The quotient digits could be from set $\{-2, -1, 0, 1, 2\}$.

Following is an outline of the proof. The reference model uses radix-2 digit-recurrence division using digit set $\{0,1\}$ and non-redundant remainder to generate the standard pairs $\{Q_j,R_j\}$ for stages $j=0,1,2,\ldots,p+1$. The implementation model is a radix-β digit-recurrence division algorithm using digit set Φ_b and redundant remainder that generates the integer pairs $\{q_j,r_j\}$ starting with $j=a$ ($a = (N < D)\ ?\ 1 : 0$). Modules bound to the implementation model convert $\{q_j,r_j\}$ into $\{\mathbb{Q}_j, \mathbb{R}_j\}$, the standard pair, for which $0 \leqslant \mathbb{R}_j < D$. This involves converting the redundant representation of the implementation model remainder to nonredundant form and then adjusting the quotient and remainder to form $\{\mathbb{Q}_j, \mathbb{R}_j\}$.

Figure 3 presents the outline of the DPV proof:

```
Proof_begin: $\{\mathbb{Q}_a, \mathbb{R}_a\}=\{Q_a, R_a\}$
for j=a,a+m,a+2m,…,a+(K-1)m
  Proof_j: if $\{\mathbb{Q}_j, \mathbb{R}_j\}=\{Q_j, R_j\}$ \
         then $\{\mathbb{Q}_{j+m}, \mathbb{R}_{j+m}\}=\{Q_{j+m}, R_{j+m}\}$
Proof_end: final result matches SPEC
```

Figure 3: Outline of assume-guarantee DPV proof.

Here $K=\lfloor(p+1-a)/m\rfloor$ is the largest integer k for which $a+KM \leqslant p+1$. If j^* is the implementation's final stage and $j^*>p+1$, then the specification does not form $\{Q^{j^*}, R^{j^*}\}$. In this case *Proof_end* above combines this final implementation's stage with the rounding logic that forms the final result.

To the above DPV proof we add properties that are first verified and then used as assumptions. Equations 5, 6, and 7 are examples of such properties. One goal of doing so is to help DPV avoid exploring unreachable portions of the dataspace; portions not considered in the design of *SingleStage's* digit selection logic. Additionally, these properties also helped in isolating bugs during development.

If the implementation uses a prescaled digit-recurrence, then the above proof must be modified. Prescaled division replaces the computation of N/D by that of $(SN)/(SD)$. The prescaling factor is usually determined by the leading fraction bits of D, so adding a case-split to the above DPV proof, one case for each possible value of S helps with convergence. Because $2^jN/D=Q_j+R_j/D$, it follows that $2^j(SN)/(SD)=Q_j+(SR_j)/(SD)$ and so the implementations's standard pair is matched to $\{Q_j, SR_j\}$ formed from the specifications's $\{Q_j, R_j\}$. There are several choices for computing SR_j such as direct multiplication.

To compare the result, extra logic was added to the prescaled radix-4 divider. This computed the final quotient bits and the partial remainder after each iteration in the binary format. Next assertions were added in the DPV verification to ensure the results matched after each iteration.

V. Experimental Results

There were three verifications done to verify the completeness of the results:

A. *Fixed point radix-2 restoring division*

This was verified using solveNB_division. Verification was run on a single core machine. Single precision verification took 16 seconds and 1.6 GB of memory. Double precision verification took 42 seconds and 1.6 GB of memory.

B. *Floating point radix-2 restoring division*

Once the fixed point radix-2 division was verified, the OP block was replaced by primitive division and modulo operations. Verification of remaining floating-point logic converged in under a second on a single core machine for both single and double precision radix-2 division.

C. *Floating point prescaled radix-4 non-restoring division*

The FP radix-2 restoring division was used as specification. Verification on a SGE grid with 200 cores and 8 GB of memory per core completed in 1 hour and 30 minutes for single precision division and 7 hours and 30 minutes for double precision division. As mentioned in Section IV.B., the radix-4 implementation is significantly more complex and different from the radix-2 implementation with no internal equivalence points. This resulted in the runtime increase.

References

[1] J. Harrison, "Floating-Point Verification Using Theorem Proving," Formal Methods for Hardware Verification, SFM, Lecture Notes in Computer Science, vol 3965, 2006.

[2] C. Scholl, A. Konrad, A. Mahzoon, D. Große and R. Drechsler, "Verifying Dividers Using Symbolic Computer Algebra and Don't Care Optimization," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021, pp. 1110-1115.

[3] J. Hauser, http://www.jhauser.us/arithmetic/SoftFloat.html, Release 3e, 2018.

[4] J. E. Robertson, "A New Class of Digital Division Methods," IRE Transactions on Electronic Computers, vol. 7, no. 3, pp. 218–222, 1958.

[5] M. D. Ercegovac and T. Lang, "Simple Radix-4 Division with Operands Scaling," IEEE Transactions on Computers, vol. 39, no. 9, pp. 1204-1208, 1990.

[6] G. Melquiond and R. Rieu-Helft, "Formal Verification of a State-of-the-Art Integer Square Root," IEEE 26th Symposium on Computer Arithmetic (ARITH), Kyoto, Japan, 2019, pp. 183-186.

[7] D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode", Formal Methods in System Design, 1999.

[8] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver Technology for System-level to RTL Equivalence Checking," 2009 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2009, pp. 196 – 201.

[9] Synopsys VC Formal, https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html.

[10] J. Harrison, "Formal Verification of IA-64 Division Algorithms," Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000.

[11] Synopsys DesignWare Library, https://www.synopsys.com/designware-ip/soc-infrastructure-ip/designware-library.html