Efficient Formal Verification and Debugging of Arithmetic Divider Circuits

Jiteshri Dasari and Maciej Ciesielski
University of Massachusetts Amherst, MA/USA
jdasari@umass.edu, ciesiel@umass.edu

***Abstract*—This paper proposes an efficient verification and debugging method for arithmetic divider circuits. The technique involves setting select signals to some predefined constants in order to reduce the design to easily verifiable circuit components. These components are then verified using logic equivalence checking and SAT tools. An important feature of the proposed approach is that it naturally enables debugging by identifying and localizing bugs through proper selection of accessible signals. This method can verify and debug large restoring dividers within single minutes using synthesis and verification tools, such as ABC. The general debugging concept proposed here is applicable to both the restoring and non-restoring dividers. To the best of our knowledge the proposed debugging capability is not offered by any of the existing verification tools.**

I. Introduction

Considerable progress has been made in recent years in verification of arithmetic circuits, such as multipliers, multiply-accumulate, and other components of arithmetic data-path, both in the integer and finite field domain [1] [2]. However, formal verification of gate-level divider circuits has not been well developed, with only a few exceptions [3] [4] [5]. Furthermore, not much progress has been made in terms of debugging of the divider circuits.

This paper describes a novel formal verification and debugging method for gate-level divider circuits, illustrated with a restoring divider, the type commonly used in industry for computation in integer and fixed point arithmetic. The method is based on setting some accessible signals of the circuit to predefined constants to identify and isolate its essential subfunctions, namely quotient bit generation and partial remainder generation, and enable their verification. The proposed verification scheme is done by using gate-level functional verification of circuit components without any reverse engineering. The main contribution of the paper is that it replaces the current state-of-the art, but memory-intensive symbolic computer algebra (SCA) and algebraic rewriting technique, successfully applied to multipliers [1] [6] [7] with a novel functional technique targeting gate-level implementation of array dividers.

The rest of the paper is organized as follows. Section II provides the necessary background and detailed review the related work in this field. Sections III and IV address the issues of verification and debugging, respectively. Finally, Sections V and VI present the results and conclusions.

II. Background and Related Work

A. Theorem Proving

A popular technique employed in industry in arithmetic circuit verification is Theorem Proving [8] [9][10] Theorem provers are inductive, non-automated reasoning systems that use mathematical models to verify functional correctness of the design. These systems typically concentrate on proving correctness of the underlying algorithms of arithmetic designs and on the resulting architectures rather than of the low-level hardware verification. The proof relies on a carefully constructed set of rewriting rules and complex formulas to represent the circuit and requires an in-depth, domain-specific user knowledge of the design and the system The success of the proof relies on the choice of the rules and on the order in which they are applied to the system, with no guarantee of a successful conclusion.

Another class of formal verification employs model checking, which has been used to verify a table-based SRT division implemented in an Intel Pentium Processor [11][12]. Although effective and able to catch (post mortem) the infamous Pentium bug in its FDIV instruction, this approach requires generating a checker circuit, which itself needs to be proved. However, no reliable means were offered for the verification of the checker circuit itself. Other theorem proving and model checking techniques tried in divider verification include [13][14]; however it was reported that in some cases it took two and a half person-years to complete the proof.

B. Canonical Diagrams

Many of the formal verification techniques employ various canonical diagrams, such as BDDs [15] and *BMDs [16], or resort to SAT-based approach for equivalence checking. Some of those methods have been used in proving correctness of the SRT divider as well [17][18], but were considered only for a single gate-level stage of the divider.

C. Symbolic Computer Algebra

A different approach to formal verification of arithmetic circuits that emerged in recent years is based on symbolic computer algebra (SCA). In this approach, the specification of an arithmetic function and its implementation are represented in algebraic (rather than Boolean) domain as polynomial rings. The verification problem is then posed as checking if the implementation satisfies the specification using canonical

Groëbner basis, known in computer algebra lexicon as *ideal membership testing* [19].

An alternative approach to arithmetic verification of gate-level circuits has been proposed in [1], using algebraic rewriting of the specification polynomial. With this approach, the polynomial representing encoding of the primary outputs (called the *output signature*) is transformed by a series of backward rewriting steps into a polynomial expressed in terms of the primary inputs (the *input signature*). The transformation uses algebraic models of circuit elements, such as logic gates or bit-level arithmetic modules. The method effectively derives arithmetic function of the circuit from its gate-level implementation. This method has been successfully used to verify complex adders and large multipliers [6][7]. A thorough review of the state-of-the-art computer algebra methods for multiplier circuit verification can be found in [20].

The rewriting technique, however, is plagued by an excessive number of polynomials generated during rewriting, many of them converging later during rewriting and vanishing; these *vanishing monomials* should be detected and removed early during the rewriting. Several modifications have been reported in the literature to improve the efficiency of the technique by finding proper ordering of terms and removing vanishing monomials as early as possible [7][2]. However, those techniques have been largely applied to multipliers and have not been successful in verifying gate-level dividers.

D. Algebraic Rewriting and SAT

Recently, a promising approach has been proposed to use a combination of symbolic computer algebra (SCA) and SAT in a method called *SAT-based Information Forwarding* (SBIF) [5], applied specifically to dividers. This work recognizes the failure of a straightforward application of algebraic rewriting caused by the excessive number of vanishing monomials, which can be summarized as follows: 1) There is a large number of equivalent and antivalent signals present in the circuit that converge causing many monomials to vanish. To identify those monomials, the circuit is simulated with input vectors satisfying the input range constraint, $0 \leq X \leq D \cdot 2^{n-1}$, where n is the size of the divider D. The signals that fall in the same equivalence class are then checked using SAT to be classified as equivalent or antivalent. This is supported by signal propagation from the primary inputs, hence termed as "SAT-based Information Forwarding" (SBIF). The affected monomials are removed before they get propagated further to avoid potential memory blowout. 2) Another important source of the problem is that practical integer divider circuits are optimized based on the input constraint, $0 \leq X \leq D \cdot 2^{n-1}$. In the resulting architecture the MSB cells in the upper levels of the divider have unused outputs, as shown later in Figures 1 and 2 making them "unclean" for backward rewriting. This deprives the rewriting process of the monomials that are essential to annihilate other monomials. These redundant polynomials are analogous to don't cares (DC) in logic circuits and are modeled as *satisfiability don't cares*. Basically, the input constraint implies that certain value combinations at the input to atomic blocks cannot occur. Instead of computing all don't cares, the paper proposes a method to choose a subset of the DCs that will minimize the polynomial size at that level. To support this method, they extract atomic blocks (half and full adders) and use BDDs and standard Boolean logic techniques to compute satisfiability DCs at the inputs to these blocks. When the size of the polynomial increases by some predefined rate, they stop the procedure, backtrack to the previous step, apply the ILP optimization to reduce the vanishing monomials, and continue with polynomial rewriting.

This approach proved rather efficient in restricting the size of the intermediate polynomials. It can handle non-restoring 256-bit and 512-bit dividers in 16.5 min and 162 minutes, respectively. One must note, however, that this method depends on the extraction of the HA/FA blocks to gain access to the their inputs on which to apply don't cares. Furthermore, this technique does not help in proving an important constraint on the value of the final R remainder relative to the divisor D, namely that $R < D$. The authors resort to BDDs to solve this problem; this however is the most expensive part of the method as it requires variable ordering and time-intensive construction of the BDDs.

E. Hardware Rewriting

Yet another attempt to divider verification applies a somewhat controversial technique of *hardware rewriting* [4][21]. It accomplishes verification by appending the circuit with a block that implements an inverse function, followed by logic resynthesis. If the circuit under verification is correct, the resulting logic becomes trivially redundant. The method works well for dividers only when applied to individual layers (each producing a quotient bit), but not for the entire circuit. However this approach does not address debugging.

F. Debugging

Despite all these advances in arithmetic circuit verification, very little work has been done on actual *debugging*, i.e., identifying and fixing logical design errors that change the functionality of the design. Most of the work in debugging concentrates on detection of manufacturing faults in logic circuits using fault (mostly stuck-at fault) modeling techniques. These methods rely on simulation or manufacturing tests to identify potential faults and do not apply to arithmetic circuits, where the goal is to find logical errors, and where the space for such faults is too large. In [22] an automated method is proposed to generate tests to detect potential logical faults in arithmetic circuits. However, the method handles only gate replacement or signal inversion as the adopted fault model. In contrast, the work proposed here can handle arbitrary type of logic errors without simulation or fault modeling.

An attempt to provide automatic debugging of complex multipliers is described in [23]. However it uses a combination of symbolic computer algebra (SCA) and Boolean satisfiability (SAT) and suffers from a large number of intermediate polynomials generated during rewriting that may overload the memory. The method proposed in [24] also computes a set of

corrector polynomials that are added to the original circuit to compensate for the error. All of these methods require significant computational resources to compute Groebner basis or to perform memory intensive rewriting and do not scale to larger circuits. Furthermore, these approaches are limited to correcting errors in *multipliers* and do not apply to other types of arithmetic circuits, such as dividers.

The work on *debugging* of divider circuits is almost non-existent. A notable exception in this domain is the work of [3]. In this work, a reverse-engineering techniques are used to extract components of arithmetic operators from the gate-level circuit and compared with an architectural model of the divider using structural matching.

In contrast, the debugging method proposed in this paper applies the controllability and observability techniques commonly used in testing, but without assuming any particular type of fault. While the SCA methods consider only one type of fault, the gate replacement or signal inversion, our approach can handle an arbitrary fault type: wrong or missing gate, missing or crossed wires, incorrect logic composed of several gates, etc. The method can significantly reduce the debugging times for large divider circuits up to 1024 bit-widths in a matter of single minutes.

III. Verification Approach

A. Restoring Divider

Arithmetic operation of the divider can be described as $X = Q \cdot D + R$, where X, D, Q, R are the dividend, divisor, quotient and remainder, respectively. In this work we consider an array divider X/D of the *restoring type*, shown in Figure 1. In this architecture, the vectors D, Q and R are n-bit wide, and the bit-width of a dividend is $2n$-1. To ensure the same bit-width of D and Q and to guarantee that the resulting quotient Q will not overflow, condition $X < 2^{n-1}D$ is imposed on the inputs, X, D [25]. With this condition the divider can be implemented with minimum area as shown in the figure, rather than having layers $2n$-1 bit wide.

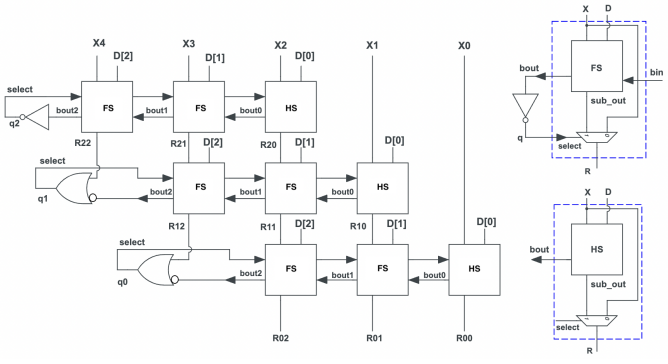

Fig. 1: Restoring integer divider - general architecture

The division is performed by a repeated subtraction of the dividend by a divisor D shifted to the right by one bit after each subtraction, basically mimicking the standard long division algorithm. If the result of a subtraction is non-negative, the subtraction takes place; otherwise the subtraction is not performed and the partial remainder is propagated to the next level. The last subtraction step produces the final remainder R. To the best of our knowledge all commercial restoring array dividers use this algorithm. In contrast, in a *non-restoring* divider each layer performs subtraction and a correction is made in the subsequent layer if the result of the subtraction is negative.

Each step of the division process corresponds to a physical layer i which performs a controlled subtraction and produces quotient bit q_i. The layers are labeled from $n-1$ on top to 0 at the bottom. The inputs to layer i are the product R_{i+1} and D, and outputs are R_i and q_i.

All known works in divider verification rely on the knowledge of layered structure of gate-level implementation, where each layer implements one step of division producing a quotient bit q_i. The layers are either provided explicitly in the structured HDL/Verilog input and preserved during synthesis (e.g., using "don't touch" directives of the Synopsys DC compiler); or can be extracted using reverse engineering with structural matching, such as in the works of [3][5]. In contrast, in our approach, the internal layered structure of the divider does not need to be provided. Instead, our method will *identify the names of signals* on the layer boundaries, enabling its verification and debugging. This is achieved by selectively setting some internal signals to predefined constants and synthesizing the circuit, which will expose the boundary signal names. This section describes the details of this procedure.

B. Layer Identification

Our approach to layer identification is based on the premise that, while the subtractor logic can be optimized "horizontally", i.e., from the LSB at the right of the subtractor of layer i to its MSB and q_i on the left, the "vertical" logic between the layers (from R_{i+1} at the top of the layer to R_i at its bottom) is never simplified. This hypothesis has been stated in [3] as Theorem 2 as follows: "In an array divider there is no way to optimize logic of adjacent rows". This feature, crucial to our approach, has been verified and confirmed in our experiments by synthesizing a divider from a generic Verilog code (shown later in Figure **??**) with Synopsys DC, as well as with Yosys/ABC synthesis tools. In all cases we were able to gain access to the intermediate signals on the layer boundaries, without knowing their internal signal names.

The procedure is based on the following observation: the quotient bit q_i produced as the MSB of each subtraction step serves as an internal *select* signal, which determines whether the input vector R_{i+1} or the difference $R_{i+1} - D$ is produced at the output, R_i. As long as such a select signal derived from q_i is available in the synthesized logic, it can be used to control the behaviour of the conditional subtractor, to then reason about its correctness. Obviously, one cannot count on finding the name sel_i in that logic, since the signal names can be changed during synthesis. Instead, we look for the appearance of q_i, which as a primary output will be preserved

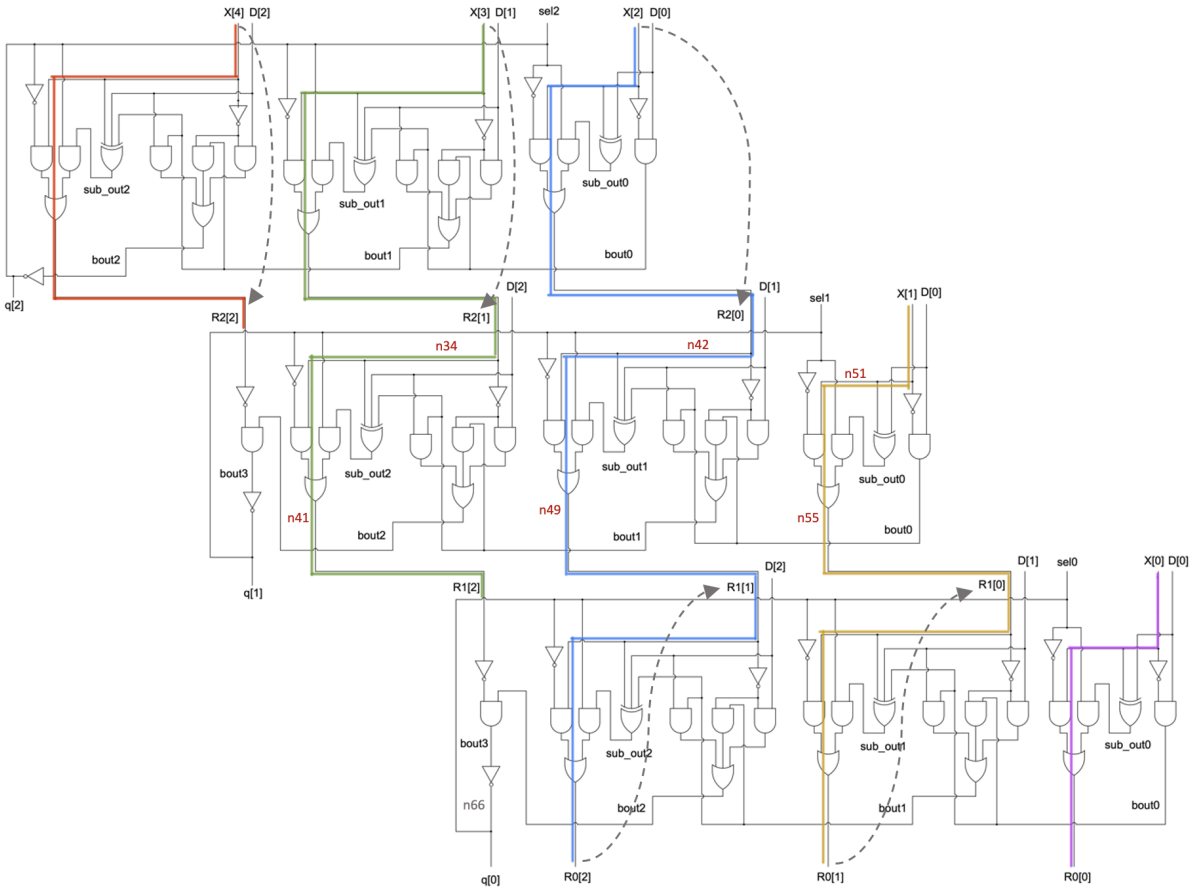

Fig. 2: Restoring divider, schematic. Arrows illustrate identification of boundary signals for Layer 1 by setting $sel_0, sel_2 = 0$.

in the synthesized code, to appear as *input* to some logic. This is shown in the fragment of the Verilog code of the synthesized 5-bit divider in Figure 3. Here the signal q_1 that appears on the input side of the logic (lines 2 and 8) will be replaced by a new name sel_1 to provide the desired control.

```
.names X[4] new_n32_ q[1]
00 0                  sel1
.names X[3] q[1] new_n34_
10 1
.names X[3] D[2] new_n35_
10 1
.names new_n34_ new_n40_ new_n41_
00 1                  sel1
.names X[2] q[1] new_n42_
10 1
.names X[2] D[1] new_n43_
10 1
.names new_n42_ new_n48_ R0[2]
00 0                  sel1
.names X[1] q[1] new_n51_
10 1
```

Fig. 3: Fragment of the $blif$ code (ABC) showing quotient bit q_1 appearing as input signal to nodes 59 and 65, changed to sel_1 and then set to a desired constant.

The correctness of this essential hypothesis (Theorem 2) has been confirmed by running several experiments: with designs synthesized from structured (layered) and unstructured (generic) Verilog code, each synthesized using different synthesis engines (Yosys/ABC and Synopsys DC). In all cases the quotient signal q_i appearing as input to some other logic (in addition to providing the required output quotient bit) has always been retained. By gaining access to that signal (the fanout of q_i) one can control the operation of the layer by either performing subtraction (with $sel_i = 1$) or a straight transfer of the vector R_{i+1} to R_i (with $sel_i = 0$), which will be used to reason about the functional correctness of the layer, without explicitly extracting it.

The result of boundary identification for Layer 1 of the bug-free divider from Figure 2 is shown in Figure 4. It is an AIG (And-Invert Graph) logic diagram generated by ABC [26] after synthesizing the logic with $sel_0 = sel_2 = 0$. The nodes of the AIG diagram are AND gates and the dotted lines represent inverters. **Interpretation:** in the following, examine the set of nodes encircled by a blue oval, representing the internal logic of layer 1 in bit position 2 (middle column) and compare the node names with signals names in Figure 2

- Note the dotted line connecting output $R_0[2]$ with node $n49$, labeled in Figure 2 as $R_1[1]$. This determines the lower boundary signal $R_1[1]$ as the logic node $n49$.

- Similarly, the solid line connecting input $X[2]$ with node $n42$, labeled in Figure 2 as $R_2[0]$, determines the upper boundary signal $R_2[0]$ as node $n42$.

C. Layer Verification

Once the signal names for the layer boundaries have been found, we can perform verification of the layer. Recall that the operation of each layer implements a controlled subtraction, governed by the equation : $R_i = R_{i+1} - sel_i D$, where sel_i is derived from q_i as described above. To verify if a given layer performs its intended function we consider two cases, determined by the value of sel_i.

- *Vertical flow verification* (checking if the signals at the top boundary R_{i+1} of layer i are connected to the signals at its bottom boundary R_i): when $sel_i = 0$, the partial remainder vector at the input of the layer is passed directly to its output, i.e., $R_{i+1} = R_i$.
- *Horizontal flow verification* (checking if the data flow along the subtractor in layer i, from its LSB to MSB, indeed performs the subtraction): when $sel_i = 1$ the layer performs subtraction, $R_i = R_{i+1} - D$.

These steps are implemented by setting the sel signals to the corresponding values, resynthesizing the circuit under those values, and observing the result of the simplified circuit on logic level. This is explained below with the example of vertical verification for Layer 1 shown in the AIG graph, Figure 5 for $sel_1 = 0$. Direct wires connecting the layer's inputs X_1, $R2[0]$ and $R2[1]$ to its outputs $R1[0]$, $R1[1]$ and $R1[2]$ indicate the correctness of the computation for $sel_i = 0$. Logic synthesis tools such as ABC can readily accomplish this reduction, while also providing the needed signal names of the output vector R_{i-1}. This process is then repeated for each layer, by selectively setting sel_1 to 0 for that layer. Advantage of using ABC as synthesis tool is that it automatically exposes *signal names* of partial remainders (layer boundary), which otherwise (except for the very top and bottom layers) are hidden and not recognizable in the input Verilog or a $blif$ file. Now, that the boundary signal names are known, the horizontal flow verification is achieved by simply running a combinational equivalence checking (CEC) of ABC.

Once all the layers have been verified for the vertical and horizontal flows, the proof is concluded by showing that the composition of such detected layers indeed implements a divider, that is that: $X = Q \cdot D + R$ and $0 \leq R < D$. This can be done by recalling that each layer i implements the computation $R_i = R_{i+1} - sel_i D$. The proof requires that we consider the computation of each division step across the entire length of the dividend X, rather than just over n bits. With this, the arithmetic function of each division step can be rewritten as: $PR_i = PR_{i+1} - sel_i 2^{i-1} D$. Here PR_i is a $(2n-1)$-bit wide (rather then just n-bit) partial remainder: in the top layer it includes the entire X, and in each consecutive layer the upper-significant bits of X are replaced by intermediate partial products coming from higher layers, to be finally replaced by the final remainder R in the lower n bits. That is, PR_i decreases at each iteration by half and $PR_i < 2^i D$. With this, it can be shown by induction that $X = Q \cdot D + R$ and (for $i = 0$) $R < D$. A complete proof basically follows the formal proof of the array division given in [25].

IV. Debugging Approach

Important feature of our approach that distinguishes it from other works on the subject is that it naturally facilitates debugging. The verification technique described in Section III exploits the effect of setting the select signal sel_i to a constant, assuming that the circuit is *bug-free*. If the assignment of sel_i to 0 does not simplify the logic by connecting the corresponding inputs and outputs of that layer, there must be a bug on a vertical logic path from R_{i+1} to R_i at the corresponding bit position. Similarly, if setting sel_i to 1 does not produce a subtractor (verified with a CEC tool), the horizontal logic path propagating the $borrow_{out}$ signal along the subtractor is faulty.

Assume that the divider has a logic error in cell k of layer l. This will cause the horizontal verification to fail during combinational equivalence checking (CEC) for that layer. Having access to all partial remainder signals (boundaries) of each layer, we can examine the internal logic of the cell (a half- or full-subtractor) by scanning the layer from its LSB to its MSB to determine in which cell (column) the fault occurred. To do that we need to access a given cell, which requires gaining access to its $borrow_{out}$ signal. This is done by setting $R_{l+1}[k]$ and $D[k]$ to predetermined controlling values that will make the cell observable. Similar to verification, the debugging process is composed of the vertical and horizontal flow debugging, as described next.

A. Vertical Flow Debugging

As described earlier, in a functionally correct circuit, setting sel_i to 0 in a given layer and synthesizing the circuit reduces the logic between signals R_{i+1} and R_i to bare wires. Figure 5, presented earlier, shows the AIG logic diagram generated by ABC after such a reduction in case when there are no bugs. In case of a bug, the direct connection between the respective signals will contain some logic, as shown in Figure 7. The erroneous logic on the path $R2[0]$ to $R1[1]$, outlined by the blue oval, indicates the bug in this layer at this bit position. Since the subtractor cells are very shallow, it will contain only a few, easy to examine logic gates.

B. Horizontal Flow Debugging

To find potential bugs in the "horizontal" logic of layer i, we set sel_i = 1, which results in the layer performing subtraction, R_i = R_{i+1} - D. Standard combinational equivalence check (CEC) of ABC is performed using any of the trusted "textbook" subtractor circuits as reference.

We then need to to identify the faulty cell and take care of the propagation of the $bout$ signal, culminating at the generation of quotient bit q_i. This is done by scanning the subtractor layer, starting at the LSB and setting $R_{l+1}[0] = 0$, as shown in Figure 8. As indicated in the figure, this will connect $D[0]$

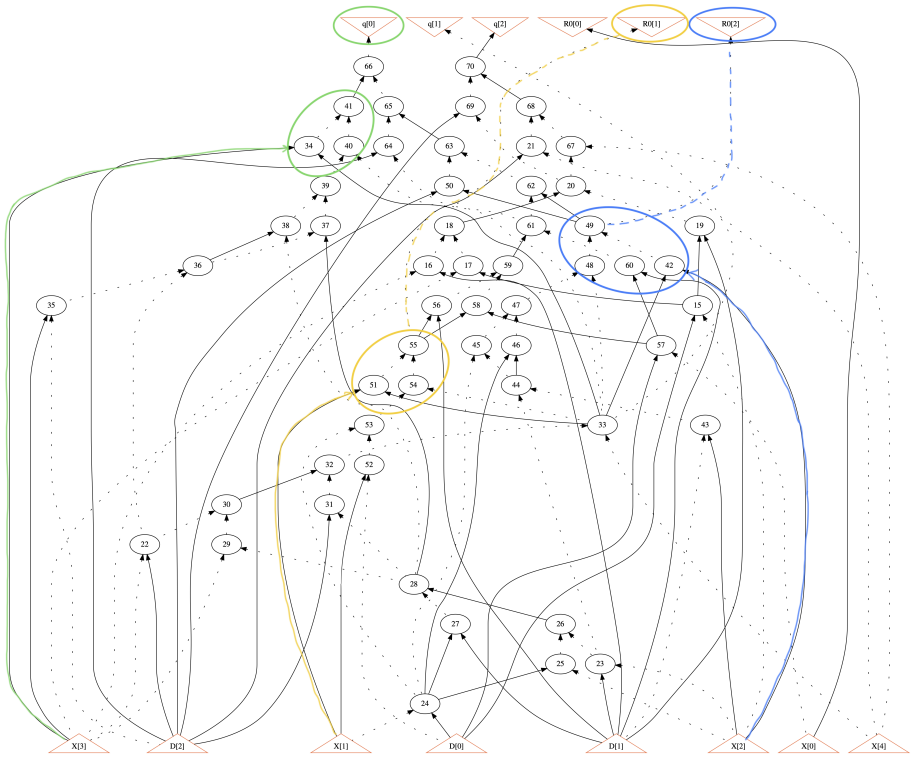

Fig. 4: Detecting layer 1 boundaries for $sel_2 = 0$ and $sel_0 = 0$

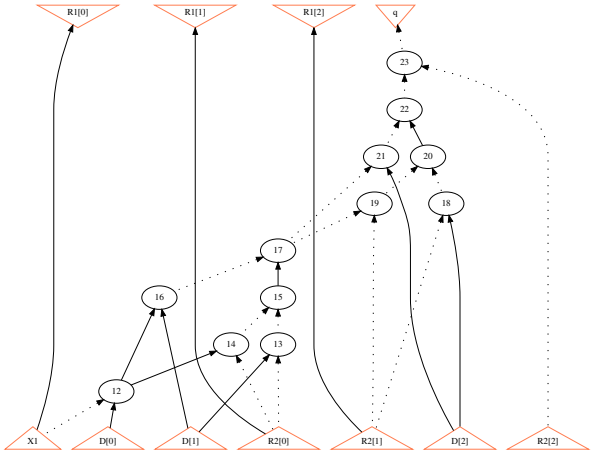

Fig. 5: Vertical flow verification, $sel_1 = 0$: layer 1 in a bug-free circuit. Note the direct connections: $R2[1] \longrightarrow R1[2]$, $X1 \longrightarrow R1[0]$, and $R2[0] \longrightarrow R1[1]$, demonstrating that the layer is bug-free.

to $bout_0$, providing access to signal $bout_0$ (hidden in the code under an unknown name). Moving to the next significant bit, we now set the $bout_0 = 0$ and $R_{l+1}[1] = 0$, to gain access to $bout_1$, then setting it to 0 as input to the next cell, etc. This procedure is similar to what has been done for layering in Section III-B, but now used for *extracting* the names of signals $bout_k$, instead of the layer boundaries. This process is

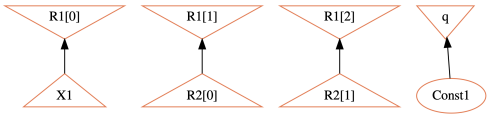

Fig. 6: Horizontal verification: Layer 1 without a bug. Note the direct connections between the signals of the input and output partial remainders.

repeated for the remaining cells, each time running the CEC on the reduced cell against a reference subtractor circuit.

Since setting $bout_k$ to 0 is essential, we need to verify if the signal identified as $bout_k$ indeed performs its expected $borrow$ function. To do this, we extract a logic cone and compare it using CEC with the reference $borrow$ logic function. This can be done efficiently using command $cone$ in ABC and is instantaneous since the $borrow$ logic is very shallow. The cell that does not pass the CEC test is faulty, and its logic should be analyzed, repaired, or replaced by a correct HS/FS logic. Figure 8 demonstrates this process for the cells in the top layer (Layer2) of our divider circuit.

The horizontal and vertical debugging process is automated; it identifies of a bug in form of a small circuit (half- or full-subtractor) containing only a handful of logic gates.

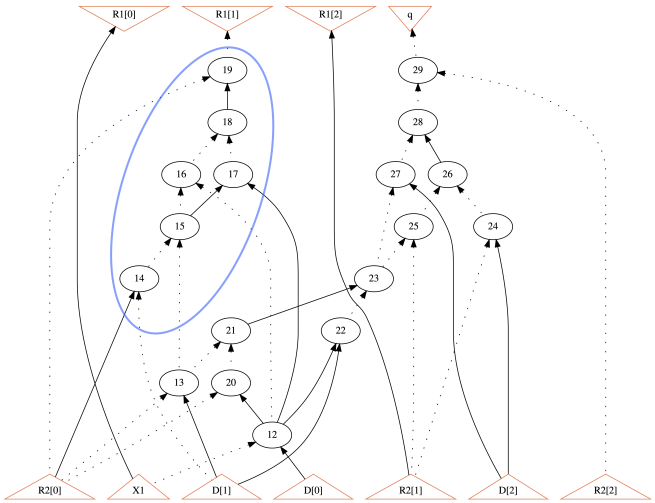

Fig. 7: Vertical debugging of Layer 1 with a bug; notice the logic block on the path $R2[0] \longrightarrow R1[1]$, indicating a fault.

V. Results

The verification and debugging method described in this paper was implemented in Python communicating with ABC for logic reduction and CEC via file transfers. All the major components of verification and debugging have been automated, including: identifying quotient q_i signals and replacing them by sel_i as needed; setting sel_i to the required constants for layer identification; identifying the layer boundaries (partial product terms R_i); applying vertical verification with $sel_i = 0$ and horizontal verification with $sel_i = 1$; determining the position of the layer i and column k in case of an error; and identifying erroneous logic in case of a bug. The entire flow is not yet automated, with files passed manually between the different procedures.

Our technique was tested on restoring divider circuits with dividend bit-widths ranging from 5 to 1023. The designs used in the experiments were generated in two different ways:

1) Written in structural Verilog with layers provided explicitly. a) They were parsed using Yosys and synthesized with ABC; b) They were also synthesized by Synopsys DC compiler without *don't touch* directives, i.e. without preserving the layer boundaries using the NanogateOpenCell library.
2) Written in generic, unstructured Verilog, shown below and: a) synthesized using Yosys/ABC; and b) synthesized by Synopsys DC compiler.

```
module div_rest(X,D,Q,R);
parameter n= ...;
    input [2n-2:0] X;
    input [n-1:0] D;
    output [n-1:0] Q;
    output [n-1:0] R;
      assign Q = X/D;
      assign R = X%D;
endmodule
```

In all cases we were able to detect the quotient output bits q_i appearing as input to some internal logic, allowing us to replace them by the corresponding select sel_i signals, needed to perform layer identification and debugging.

The results of our experiments are shown in Table I. It gives the CPU times in seconds for the individual steps in comparison with other divider verification methods reviewed in Section II. Specifically, the Table columns describe the following:

1) Size of the divider (dividend bit-width)
2) Layered Hardware Rewriting of [4] [21]
3) SCA SBIF/DC method of [5]
4) Auto-debug method of [3]
5) This work: Layer detection
6) This work: Vertical debugging
7) This work: Horizontal debugging

The only meaningful comparison in terms of debugging can be made with the work of [3], other methods do not provide debugging facility. However, the largest divider included in their experiments has only bit-width of 64. As one can see from the table, our method can verify bug-free dividers orders of magnitude faster than other methods, including a 1023-bit divider verified in 16.2 minutes, something that no other method could handle.

The debugging results are presented in an estimated fashion, by adding the execution time of individual components performed separately. Specifically, for a divider with n layers, each being $n-bit$ wide, detecting a fault in layer l at bit/cell k takes the following time: $T_f(l,k) = T_L + T_v + (l+1) \cdot t_h + (k+1) \cdot t_c$, where

- T_L = CPU time for detecting layer boundaries; (column 4 of the Table);
- T_v = time of vertical verification of all layers (Col. 5);
- t_h = time of horizontal verification (CEC) of one layer (Col. 6/n); and
- t_c = CEC verification time of one cell.

The Table shows the debugging time as an average time between the LSB cell in the top layer and the MSB cell in the bottom layer. We should point out that the time $T_f(l,k)$ is the debugging time for debugging all cells in the faulty layer l up to the cell k.

VI. Summary and Conclusions

The proposed verification of integer restoring divider circuits is based on structural analysis of the overall design and the functional verification of circuit's components (layers). The verification is accomplished in two phases: 1) extracting boundaries of computational layers that perform conditional subtraction; and 2) verifying arithmetic functions they perform. The main advantage of this approach is that it naturally integrates verification with debugging. The faulty logic detected during verification is automatically localized to a shallow logic which can be corrected with minimum user intervention. The proposed approach does not require

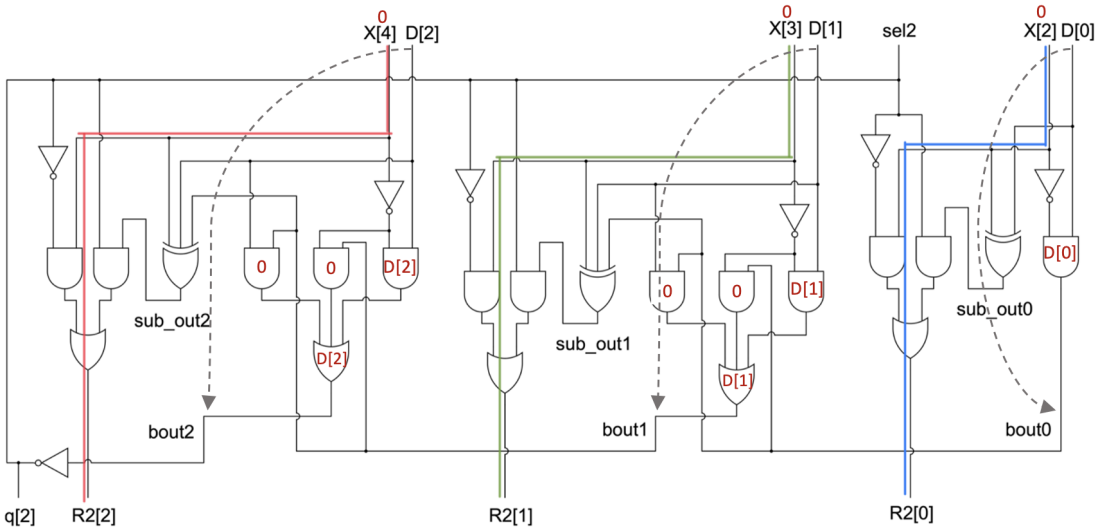

Fig. 8: Horizontal debugging; cell separation: setting $X[k] = 0$ to gain access to $bout[k]$ signals from $D[k]$.

TABLE I: Restoring Divider Debugging times in CPU seconds

Dividend bits	Layered HR-SAT time (s) [21]	Auto-debug T2/(T0+T1) (s) [3]	Verification times in seconds (**this work**)					Don't care opt time (s) [5]
			Layer detection T_L	Vertical verif sel=0 T_v	Horizontal verif sel=1 $t_h * n$	CEC of all cells $t_c * n$	**Debugging time**	
5	0.09	4.3/4.9	0.11	0.12	0.13	0.03	0.26	0.16
7	0.12	-	0.15	0.16	0.18	0.04	0.33	0.48
13	0.21	4.4/5.8	0.25	0.26	0.30	0.07	0.51	-
17	0.27	10.5/11	0.32	0.34	0.40	0.09	0.65	1.91
23	0.48	-	0.44	0.47	0.56	0.12	0.88	3.82
33	0.68	22.3/21.6	0.62	0.65	0.80	0.17	1.2	6.79
65	4.48	39.2/42.9	1.29	1.32	1.63	0.33	2.37	28.86
95	12.96	-	2.02	1.98	1.97	0.48	3.27	70.22
127	18.56	-	2.93	2.66	3.76	0.64	4.94	148.18
255	123.46	-	9.75	5.89	10.52	1.28	11.91	989.91
511	-	-	51.43	13.76	108.97	2.56	70.45	9,668.70
1023	-	-	426.94	35.01	512.28	5.12	295.05	TO (>24 CPU hours)

structural reverse engineering of the circuit. The only assumption it currently makes for debugging is that the ripple-carry implementation of the subtractor circuits needed to find each cell of the subtractor is faulty (but can verify it for any implementation of the subtractor); but even this can be handled by analysing the structure of the subtractor, if implemented e.g., as a carry-look-ahead circuit.

Our method differs from the traditional debugging approaches, which classify the bug as a wrong gate type or gate with wrong polarity. Our approach is more general since it can cover a larger class of bugs, including: incorrect gate, incorrect signal polarity, missing gate, wrong or missing wiring, or a set of gates implementing a faulty behaviour. For example, if the fault is caused by incorrectly implementing an XOR logic function as $f = \overline{a \cdot b + a' \cdot b}$, instead of $f = a' \cdot b + a \cdot b'$, our approach will indicate the problem with this logic (not just the gate), instead of trying to point out that one AND gate has wrong input polarity and the NOR should be replaced by OR. The logic at this level is so shallow that it is better to leave it to the designer to quickly see the problem and fix the bug. We believe that this is more practical and useful.

The described approach to verification and debugging is directly applicable to other array-type dividers (a predominant type used in industry, available e.g., from Design Ware of Synopsys), such as *non-restoring division*. The only difference is in setting other signals to the respective constants and drawing a conclusion if the reduced circuit implements the required function. For example, in case of a non-restoring division, each layer performs either subtraction or addition, depending on the value of the quotient q_i. Setting $q_i = 1$ should configure the layer to perform subtraction, while setting $q_i = 0$ should configure it for addition. Vertical verification is not needed in this case. Application to other, drastically different division schemes, such as table-based SRT or Goldschmidt-based division remains to be examined.

REFERENCES

[1] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model," *IEEE TCAD*, vol. 39, no. 6, pp. 1346–1357, 2019.

[2] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases," in *DAC'14*, 2014, pp. 1–6.

[3] M. H. Haghbayan and B. Alizadeh, "A dynamic specification to automatically debug and correct various divider circuits," *INTEGRATION, the VLSI journal*, vol. 53, pp. 100–114, 2016.

[4] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "Functional verification of hardware dividers using algebraic model," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2019, pp. 257–262.

[5] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1110–1115.

[6] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining sat and computer algebra," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019, pp. 28–36.

[7] A. Mahzoon, D. Große, and R. Drechsler, "REVSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[8] J. Harrison, "Floating-point verification using theorem proving," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2006, pp. 211–242.

[9] P. Manolios, "Refinement and theorem proving," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2006, pp. 176–210.

[10] D. M. Russinoff, *Formal Verification of floating-point hardware design: a mathematical approach*. Springer, 2018.

[11] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking," in *(ICFMCAD)*. Springer, 1996, pp. 19–33.

[12] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: Formal verification of pentium®4 floating-point divider," in *Proc. 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, 2011*. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 196–211.

[13] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," in *(CAV)*. Springer, 1996, pp. 111–122.

[14] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation." in *CAV*, 06 2009, pp. 414–429.

[15] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Tran. on Comp.*, vol. 100, no. 8, pp. 677–691, 1986.

[16] R. E. Bryant and Y. Chen, "Verification of arithmetic functions with binary moment diagrams," Pittsburgh, PA, USA, Tech. Rep., 1994.

[17] Yimg-An Chen and Bryant, "*phdd: an efficient graph representation for floating point circuit verification," in *1997 Proceedings of IEEE ICCAD*, Nov 1997, pp. 2–7.

[18] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *33rd (DAC)*. ACM, 1996, pp. 661–665.

[19] E. Pavlenko, M. Wedler, D. Stoffel, and W. Kunz, "STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

[20] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD'17*, 2017.

[21] A. Yasin, "Formal verification of divider and square-root arithmetic circuits using computer algebra methods," *PhD dissertation; University of Massachusetts, Amherst*, 2020.

[22] F. Farahmandi and P. Mishra, "Automated test generation for debugging arithmetic circuits," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1351–1356.

[23] A. Mahzoon, D. Große, and R. Drechsler, "Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 351–356.

[24] N. A. Sabbagh and B. Alizadeh, "Arithmetic circuit correction by adding optimized correctors based on groebner basis computation," in *2021 IEEE European Test Symposium (ETS)*, 2021, pp. 1–6.

[25] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, 2002.

[26] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," *http://www.eecs.berkeley.edu/~alanmi/abc*, 2007.