Combining Formal Verification and Testing for Debugging of Arithmetic Circuits

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

***Abstract*—Formal verification has been successfully used to verify different types of digital circuits, including combinational and sequential logic, arithmetic circuits, and datapath designs. However, the verification techniques concentrate on confirming whether the circuit performs its intended function, while the issue of debugging, i.e., detection and correction of functional errors of the design, remains an open problem. Elaborate testing techniques have been developed that target certain types of manufacturing faults, but there are no general techniques that address the debugging issue for functional bugs.**

This paper addresses the issue of debugging of arithmetic circuits that due to their large size and complexity are particularly hard to verify and debug. Current debugging techniques handle only simple types of bugs: gate replacement, wrong gate polarity, or a missing gate, but cannot handle more realistic faults, such as wrong wiring or using a wrong combination of logic gates. We describe a novel method that combines formal verification and testing techniques to enable efficient identification and correction of faults. The technique involves setting select signals to some predefined constants to reduce the design to easily verifiable circuit components; these components are then verified using logic equivalence checking and SAT tools. The fault can then be identified in form or a small logic area (with a few logic gates) to be replaced by a new, functionally correct logic. The proposed technique is illustrated with debugging of different types of divider circuits up to 1024 bit-wide.

I. Introduction

Verification of arithmetic circuits poses a considerable challenge due to their size and large bit-widths of their operands. Strictly Boolean methods, such as those based on BDDs [1] and SAT [2], require "bit blasting", i.e., flattening of the design into bit-level netlists. A large size of such netlists makes it inefficient for handling complex arithmetic circuits, such as multipliers and dividers. On the other hand, higher level inductive techniques, such as Theorem Provers, do not offer sufficient insight into their structure [3][4], required for gate-level debugging. They are highly interactive, require intimate knowledge of the design and user guidance, and offer no guarantee for a successful termination.

Recent work in arithmetic circuit verification has relied mostly on computer algebra (SCA) techniques, successfully applied to the verification of datapaths and multipliers. While the goal of verification is to confirm whether the circuit performs its intended function, the issue of debugging, i.e., detection and correction of functional errors of the design, remains an open problem. Elaborate techniques have been developed by the testing community that target certain types of manufacturing faults, such as stuck-at/open, or bridging faults, but there are no general techniques that address the debugging issue for functional rather than manufacturing fault detection and *debugging*, i.e., identifying and fixing logical design errors that affect functionality of the design.

This paper addresses the issue of debugging of arithmetic circuits and illustrates the debugging technique on the divider circuits. Currently available debugging techniques target only simple types of bugs: gate replacement, wrong gate polarity, or a missing gate, but cannot handle more realistic faults, such as wrong wiring or an incorrect logic. The proposed method combines formal verification and sensitizability techniques to enable efficient identification, localization and correction of the fault. The technique is based on setting some properly chosen observable and controllable signals to predefined constants, which reduces the design to easily verifiable circuit components. These components are then verified using logic equivalence checking and Boolean satisfiability (SAT) tools. This allows one to localize a fault to a small logic area, composed of just a few logic gates, that can be replaced by a new, functionally correct and clearly defined logic. The proposed debugging technique is illustrated on different types of array divider circuits.

II. Related Work

The state of the art techniques for arithmetic circuit verification are largely based on Symbolic Computer Algebra (SCA). In this approach, an arithmetic circuit is represented in algebraic (rather than Boolean) domain, with the gate-level implementation and its specification modeled as pseudo-Boolean polynomials. There are two basic flavors of this technique: the classical methods, based on Gröbner basis polynomial reduction [5][6][7]; and others, based on a more practical implementation, called *algebraic rewriting*, originally proposed in [8]. Algebraic rewriting is the process of transforming the polynomial representing binary encoding of the output, called *output signature*, into a polynomial of the binary encoding of primary inputs, the *input signature*, using algebraic models of logic gates implementing the circuit. In a functionally correct circuit, the resulting input signature should match the specification polynomial. If the specification is not known, the resulting input signature provides the arithmetic function computed by the circuit. Hence, the method can be used as a reverse engineering tool for arithmetic function extraction.

The idea of algebraic rewriting is conceptually simple but computationally challenging, as it is plagued by the exponential growth in the number and size of polynomials generated during rewriting. A number of techniques have been invented to limit this growth and successfully applied in verification of multipliers [9] [10] [11]. Some of the recent works attempted to apply algebraic rewriting to divider circuit verification. In [12], a special type of don't cares, representing disallowed inputs values at the internal arithmetic blocks, are extracted; they are then used in the ILP-based optimization procedure to reduce the number of intermediate polynomials. This method, however, relies on reverse engineering to extract basic arithmetic blocks (half- and full-adders) and on a computationally expensive ILP optimization.

In another approach, termed *Hardware Rewriting*, [13][14] the circuit is appended with an inverse circuit and the modified circuit is re-synthesized; if the resulting circuit is reduced to identity (bare wires) it proves that the original circuit is correct, i.e., performs the required arithmetic function. This method however is limited by the power of logic synthesis tool, which may not be able to reduce large circuits to such a redundant state. In a novel method proposed in [15] verification is accomplished by setting some signals to predefined constants in order to isolate its essential subfunctions to facilitate their verification. However, none of these approaches solves the actual *debugging* problem, i.e., identifying and fixing logical design errors that affect the functionality of the design. Most of the debugging and fault detection methods rely on traditional testing techniques designed to uncover manufacturing faults; they do not apply to arithmetic circuits, where the goal is to find logical errors and the design space is too large.

In one of the first works on arithmetic circuit debugging [16], potential bugs are identified by performing forward and backward rewriting and analyzing the difference between the resulting polynomials, pointing to an incorrect gate. The method requires guidance as to which area of the circuit to target for bugs and is not scalable because of large size of the generated "difference" polynomials. In the works of [17] and [18] an automated method has been proposed to generate tests to detect potential logical faults in arithmetic circuits. However, these methods handle only gate replacement and signal inversion as the adopted fault model.

An attempt to provide automatic debugging of complex multipliers has been described in [19]. It uses a combination of symbolic computer algebra (SCA) and Boolean satisfiability (SAT) but still suffers from a large number of intermediate polynomials generated during rewriting that may overload the memory. The method proposed in [20] also computes a set of corrector polynomials that are added to the original circuit to compensate for the error. All these methods require significant computational resources to compute Gröbner basis or to perform memory intensive rewriting and do not scale to larger circuits. Furthermore, they are limited to correcting errors in adders and multipliers and does not work well with 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 [21], which applies reverse engineering and structural matching of the extracted circuit against a known structure. However, the reliance on the knowledge of the circuit structure limits its applicability to general architectures.

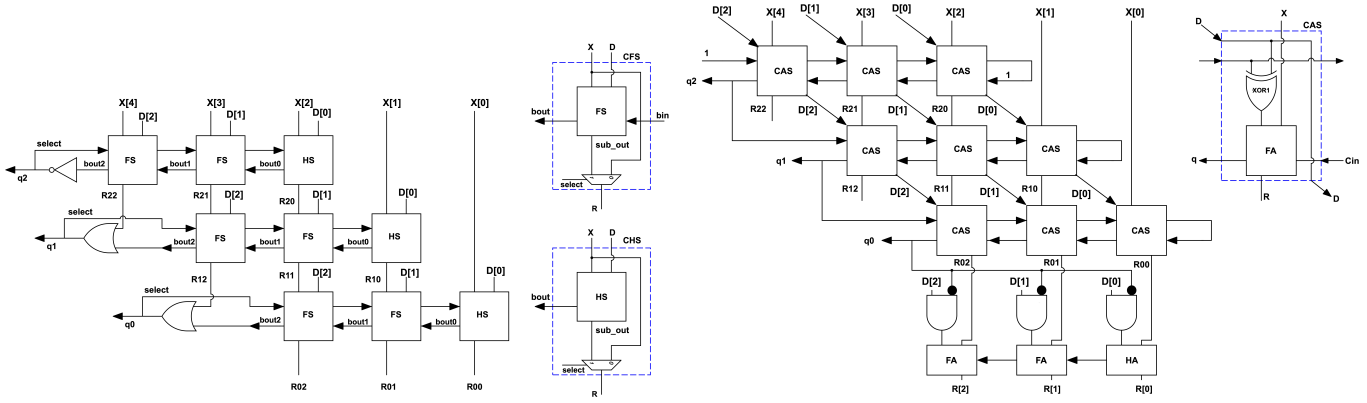
The debugging method described in this paper applies the techniques of controllability and observability, commonly used in testing in the context of manufacturing faults. The method was originally proposed in [15] for logic verification of circuits such as dividers. It has been already applied to the debugging of *restoring* dividers in [22] and is extended here to the debugging of *nonrestoring* dividers (the terms to be defined in the next Section). Unlike in testing, which typically concentrates on well defined types of manufacturing faults, this approach covers a wide range of faults, including gate replacement, wrong gate polarity, missing or miss-wiring connections, and wrong logic blocks.

III. Verification of Array Dividers

This work addresses debugging of an array divider circuit X/D. 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. The vectors D, Q and R are n-bit wide, and the bit-width of a dividend is $2n$-1. While in principle the divider can be constructed as an array of n rows and $(2n-1)$ columns, additional constraint is typically imposed on the divider to ensure the same bit-width of D and Q and to guarantee that the resulting quotient Q will not overflow. For the integer version of the divider this constraint takes the form $X < 2^{n-1}D$ [23]. With this, the optimized version of the divider takes the form shown in Figure 1.

There are two types of array dividers, *restoring* and *nonrestoring*, depending of the type of the applied "long division" algorithm. Both types perform division by a series of controlled subtract/add operations organized in n rows of add/subtract circuits, each row producing a partial remainder. The top "remainder" is the dividend X and the bottom one is the final remainder R. Each step of the division 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 partial remainder R_{i+1} and divisor D, and the outputs are R_i and q_i.

An example of a 5-bit divider is shown in Figure 1a) for a restoring divider, and Figure 1b) for a nonrestoring divider. In a *restoring* divider the division is performed by a repeated subtraction of the partial remainder by the divisor D, shifted one bit to the right after each subtraction. If the result of the subtraction is positive, the subtraction takes place; otherwise the subtraction is not performed and the unchanged ("restored") partial remainder is propagated to the next level. In contrast, in the *nonrestoring* divider the current remainder is subjected to either subtraction or addition, depending on the result of the value of previous remainder. If it is positive, the subtraction takes place, otherwise a correction is made in the subsequent layer by adding D. If the final remainder has a



a) Restoring divider b) Nonrestoring divider

Fig. 1: Array integer divider: Restoring and Nonrestoring types.

different sign than the dividend X, a correction is made by adding D; this is accomplished by an additional adder layer.

Each of those dividers can perform either integer or fractional division, and the only difference between the two is the ordering of bits, with the position of MSB and LSB interchanged. In the following we consider an integer version of the divider. Note the difference between the two divider types in terms of how the quotient bits q_i interact with the circuit and affect the computation: in a *restoring* divider the quotient bit q_i controls the flow of data of the same layer i (to choose between the subtraction and passing the input remainder to the output); in a *nonrestoring* divider the quotient bit q_i controls the operation of the next layer that produces quotient bit q_{i-1}.

The layers are either provided explicitly in the structured HDL/Verilog input and preserved during synthesis (using "don't touch" directives of the Synopsys DC compiler); or they need to be extracted using reverse engineering with structural matching, such as in the works of [21] and [12]. In our approach the internal layered structure of the divider doesn't need to be provided; instead, the method will automatically identify the signals on the layer boundaries, enabling its verification and debugging, regardless of the type of the divider. This is achieved by selectively setting some internal signals to predefined constants and synthesizing the circuit, which will expose the signals at the layer boundaries.

A. Layer Extraction

The layer extraction procedure is based on the observation that each quotient bit q_i produced by layer i is used to control the operation of the same layer (in case of the restoring divider), or of the next layer (in case of the nonrestoring divider), as shown in Figure 1.

- In the restoring divider circuit, signal q_i determines whether the input vector R_{i+1} or the difference $R_{i+1}-D$ is produced as the output remainder, R_i. That is, $R_i = R_{i+1} - q_i D$.
- In the nonrestoring divider the operation of layer i is controlled by the bit q_{i+1} of the previous layer, to choose between subtracting D (for $q_{i+1} = 1$) or adding D (for $q_{i+1} = 0$). In this case, $R_i = R_{i+1} + (1 - 2q_{i+1})D$.

In both cases the signal that controls the operation of the layer can be readily derived from the corresponding quotient bit, q_i or q_{i+1}. By gaining access to that signal one can control the operation of the layer, reducing it to a simpler logic. This, in turn, will make it possible to reason about the functional correctness of the layer without explicitly extracting it by structural matching.

Layer identification has been described in our earlier work in [22] in the context of a restoring divider. The following describes the procedure applicable to nonrestoring dividers. Recall that each layer of the nonrestoring divider implements a controlled Add/Subtract (CAS) operation, indicated in Figure 1b) as CAS nodes.

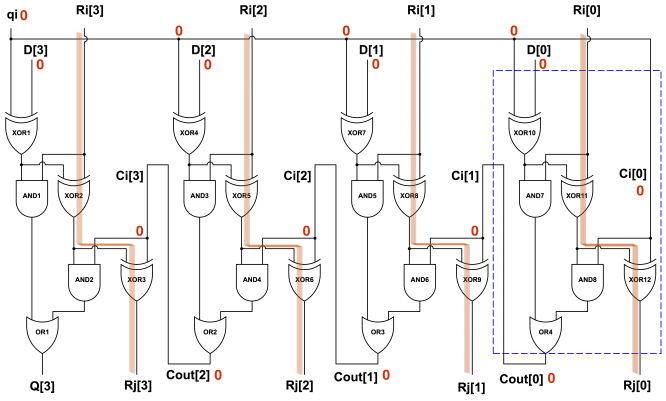


Fig. 2: Setting $D_k = q_i = 0$ to identify layer boundaries: make $Rj[k]$ accessible from $R_i[k]$.

Consider the LSB of a single layer of the nonrestoring divider, shown in blue rectangle in Figure 2, with carry-in coming from q_i. We need to find sensitization conditions that will connect the R_i^k signal at the top of the layer with the R_j^k signal at the bottom of the layer. This will allow us to identify the signal name of the lower boundary of the

layer for that bit k; and then apply it to other bits. Standard sensitization technique shows that this can be accomplished by setting $D_k = q_i = 0$, as shown in the Figure. With this, the XOR_{10} gate and the two AND gates and are set to 0, while the XOR_{11} and XOR_{12} gates pass the partial remainder signal $R_i[k]$ to $R_j[k]$ at the lower boundary of the layer. This also causes $C_{out}[k] = 0$, which provides a carry-in to the next cell. Note that this is independent of the actual gate-level implementation of the cell, and is determined only based on the expected logic relationship between $R_j[k]$ and $R_i[k]$. The same procedure is repeated for the consecutive cells, each generating $C_{out}[k] = 0$ and exposing the partial remainder signals at the layer boundaries. Even if the add/subtract layer is implemented with a carry-look-ahead (CLA) circuit, the individual outputs (partial remainder signals) can be detected since they have the same logic function.

B. Layer Verification

Once all the signals at the layer boundaries have been identified, one can perform verification of each layer, one by one; it is referred to as *horizontal verification*. To verify if a given layer performs its intended function, we consider two cases, determined by the value of the quotient bit q_i. In a *restoring* divider, setting $q_i = 0$ results in connecting the partial remainder R_{i+1} at the top of layer i to R_i at the bottom. In a *nonrestoring* divider, it configures the layer to an adder. Setting $q_i = 1$ turns both, the restoring and nonrestoring divider, into a subtractor. These functions can be readily verified by synthesizing the layer under the respective values of q_i, and checking if the resulting circuit performs the required ADD or SUB function. This is readily done using combinational equivalence checking (CEC) or SAT tool of ABC [24].

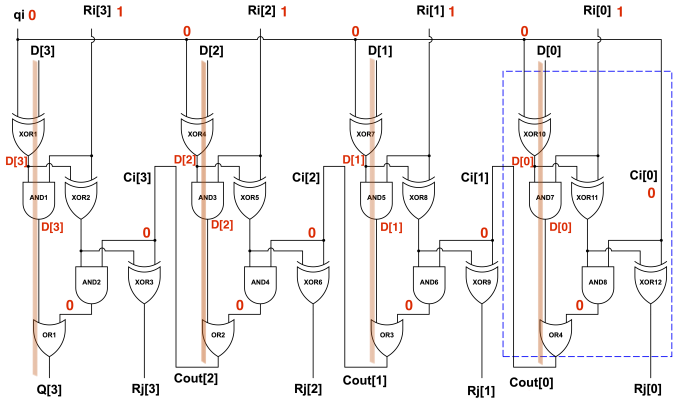


Fig. 3: Horizontal verification and debugging. Global: set $q_i = 0$ for ADD and $q_i = 1$ for SUB. Local cell scanning: set $q_i = 0, R_i[k] = 1$ to identify $cout[k]$ from $d[k]$. Then apply CEC to the adder cell with inputs $R_i[k], D[k], c_i[k]$ and outputs $R_j[k], cout[k]$

.

IV. Debugging

The verification approach described in the previous section provides a mechanism to perform robust gate-level debugging. The goal of debugging is to pinpoint the exact location of the fault, specified as a small logic area at the intersection of the vertical and horizontal flow of data. The vertical flow is associated with column (bit) k of the partial remainder R_i across all rows (layers). Horizontal flow is associated with data propagating along layer i through all bit positions $\{k\}$ of R_i. The segment of logic at the intersection of the two flows is annotated as R_i^k. In the circuits shown in Figure 1 such an area corresponds to an individual full/half-adder (FA, HA), full/half-subtractor (FS, HS) or a CAS cell, but in other implementations (such as CLA) it may correspond to some other logic blocks.

A. Restoring Divider

The debugging of a restoring divider has been described in detail in [22], briefly reviewed here to provide the necessary background. It starts with the *vertical debugging*, setting $q_i = 0$ and checking if the top signals of the R_{i+1} directly connect to the bottom one, R_i, for all bits k. If this fails for some bit k, there must be a bug at bit k of the layer. In this case the faulty logic is located on a path from R_{i+1} to R_i, typically containing just two or three logic gates. Figure 4 shows an AIG (and-inverter-graph) diagram, used by ABC, with a fault on the logic path between a pair of partial remainder signals (R_i, R_{i+1}) that should be connected together.

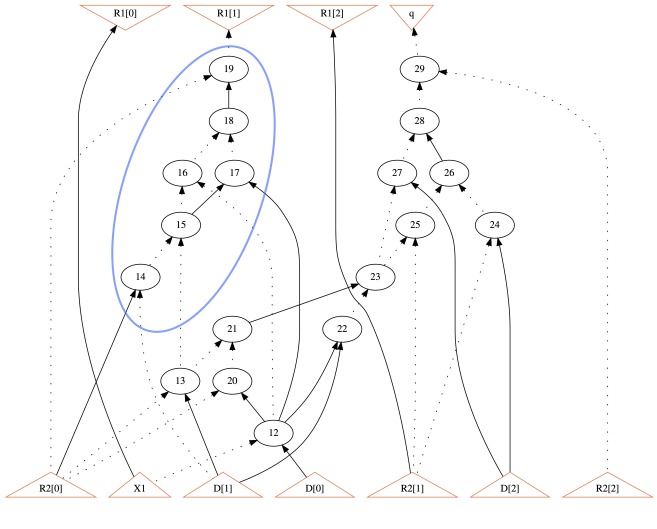


Fig. 4: Faulty area identified during vertical debugging (restoring divider) [22]

The next step, *horizontal debugging*, is performed by setting $q_i = 1$, which should result in computing $R_i = R_{i+1} - D$ word-wide. The check if the resulting logic represents a subtractor is accomplished by performing combinational equivalence checking of the layer circuit, synthesized with $q_i = 1$, against a reference subtractor. If the equivalence check fails, one needs to scan the layer from its least significant bit (LSB) to the most significant bit (MSB) and check if each cell is a single-bit subtractor (HS/FS). This is done by selectively setting individual bits k of R_i to 0, which will connect the

corresponding bit of D to the borrow-out signal, allowing it to identify the *bout* output signal of the subtractor cell. At this point all inputs and outputs of the cell are available and the cell can be checked against a reference cell (HS/FS) to identify the faulty logic. The individual cell of such extracted full/half subtractor is small enough so it can be replaced entirely.

B. Nonrestoring Divider

A similar procedure has been applied to a nonrestoring divider, with logic function of layer i described as $R_i = R_{i+1} + (1 - 2q_{i+1})D$. If the quotient bit q_{i+1}, coming form the layer above, is set to 0, layer i is configured as an adder, computing $R_i = R_{i+1} + D$; otherwise, for $q_{i+1} = 1$ the layer implements a subtractor, $R_i = R_{i+1} - D$. Verifying if the resulting logic indeed represents an adder or a subtractor can be readily done using standard combinational equivalence checking tool, such as CEC of ABC [24]. Any of the trusted "textbook" adder/subtractor circuits can be used as a reference circuit. Logic error in any of the cells of the layer will cause such horizontal verification of the layer to fail during combinational equivalence checking.

In order to determine in which cell the fault has actually occurred, one must examine the internal logic of each cell, scanning the layer from its LSB to MSB and performing combinational equivalence checking on the cell. To do that, one needs to gain access to all inputs and outputs of the cell, which can be done in a similar fashion as for the restoring divider. Specifically, setting $q_i = 0, R_j[k] = 1$ will identify $cout[k]$ from $d[k]$, as shown in Figure 3. Then CEC is applied to the FA cell with inputs $R_i[k], D[k], c_i[k]$ and outputs $R_j[k], cout[k]$. The cell that does not pass the CEC test is considered faulty, and its logic should be analyzed, repaired, or replaced by a correct half- or full-adder logic.

There is one more, essential check that can be performed on a nonrestoring divider, which does not occur in a restoring divider. Recall that step i of the division is determined by the sign of the partial remainder of the previous step $i + 1$: it performs subtraction if the partial remainder R_{i+1} is positive (its $MSB_{i+1} = 0$); and it performs addition (to correct the result R_{i+1} back to its previous value) if it was negative (i.e., $MSB_{i+1} = 1$). This condition translates into the following Boolean invariant: the quotient bit q_j generated in layer j and the most significant bit (MSB) of the layer must be antivalent (of opposite polarity); that is, $q_j = MSB'_j$, for all layers j. This invariant is an essential and distinct feature of the nonrestoring division algorithm. Furthermore, this invariant is only true for the range condition $X < 2^{n-1}D$ for which the optimized nonrestoring divider show in Figure 1 was constructed. In our work this invariant is verified by checking if $q_j = MSB'_j$ is always satisfied for the layer upended with the range condition $X < 2^{n-1}D$.

V. Results

The verification and debugging method described in this paper was implemented in Python and tested on both types of array dividers. The entire verification/debugging process includes three main phases: layer extraction, verification, and debugging; each phase has been automated, with the results reported in Table I. **Layer extraction** starts with the synthesized *blif* files generated by Yosys [25] followed by ABC [24] from the original Verilog description (this time is not included in the table). It sets signals D, q_i to respective constants to determine layer boundaries and extract the individual layers. The **verification** phase includes: configuring each layer i for two cases, $q_i = 0$ and $q_i = 1$ and synthesizing each with ABC; performing combinational equivalence checking using ABC/CEC of each synthesized layer against a reference circuit; and performing the invariant check $q_i = MSB'_i$ for each layer under the range condition $X < 2^{n-1}D$. The **debugging** phase, once triggered by CEC failure, involves: determining and scanning the faulty layer; applying CEC against the corresponding single-bit reference FA/FS or HA/HS circuit; detecting the cell which fails the CEC test and declaring it as faulty. As a result, the system is able to identify a bug in form of a small circuit, containing only a few logic gates.

The entire program was tested on a set of restoring and nonrestoring divider circuits with dividend bit-widths ranging from 5 to 1023. The experiments were run on a M1 MacBook Pro computer using a single core and 8 GB of RAM. The restoring dividers were generated from a structural Verilog, parsed using Yosys synthesis tool and synthesized with ABC. Nonrestoring designs were obtained from a benchmark set provided by [12].

The results of our experiments are shown in Table I for both types of dividers. The Table shows the CPU processing time for all three phases for the correct and *faulty* divider circuits, obtained by introducing some randomly placed, up to five, faults. The faults range from an incorrect gate (e.g., XOR instead of NAND), wrong polarity (NOR instead of OR, etc), miss-wired or swapped inputs, to a completely faulty internal cell. Note that the debugging time depends on the location of the fault.

Our results are compared with the verification times for nonrestoring dividers presented in [12], which do not address the debugging. The only meaningful comparison in terms of debugging can be made with the work of [21], which accomplishes debugging by reverse engineering and structural matching; those experiments, however, are limited to 64 bit-width dividers. As can be seen from the table, our verification method for bug-free dividers is competitive with other methods, but offers a debugging capability that is not available elsewhere. We believe that the results can be further improved if the current Python program was rewritten in C/C++.

VI. Summary and Conclusions

The paper describes an original approach to debugging of integer divider circuits, initiated in our earlier works [15][22]. In this approach the debugging is an integral part of verification; it is accomplished by setting select signals to predefined constants to make portions of the circuit controllable and observable, enabling a gate-level debugging. Contrary to the earlier methods that rely on reverse engineering, our

TABLE I: Divider Debugging times in CPU seconds

Divid bits	Auto-debug time (s) [21]	DC verif time (s) [12]	Layer extract Nonrestoring (**this work**)	Total Verif. time - bug-free (**this work**)		Verification + Debug time - Restoring (**this work**)			Verification + Debug time - Nonrestoring (**this work**)		
				Restoring	Nonrestoring	**1 bug**	**3 bugs**	**5 bugs**	**1 bug**	**3 bugs**	**5 bugs**
7	-	0.16	0.64	0.49	0.52	1.43	1.43	3.31	1.16	2.38	2.38
17	10.5/11	0.48	1.28	1.06	1.04	3.06	5.08	7.07	2.23	2.23	6.99
33	22.3/21.6	1.91	2.56	2.07	2.08	5.69	9.31	16.54	4.43	6.78	4.43
65	39.2/42.9	6.79	5.44	4.24	4.80	4.24	11.22	18.18	9.49	14.18	18.87
127	-	28.86	14.08	9.35	10.24	22.25	35.16	48.05	19.62	38.38	29.00
255	-	148.18	46.08	26.16	29.44	52.29	104.57	155.76	48.62	67.80	86.98
511	-	989.91	266.24	174.16	197.12	174.16	282.40	336.79	235.96	235.96	313.64
1023	-	9,668.70	1,976.32	974.23	921.6	974.23	1,205.69	1,436.50	1,003.50	1,085.40	1,249.20

method achieves the debugging by analyzing the logic of individual layers. The method works for both, the restoring and nonrestoring dividers, and differs only in the choice of the sensitizing signals.

The method has some limitations. It may not be able to identify bugs if they are placed in adjacent layers of the same column; or for architectures where the identification of the layer boundaries is not possible. The identification of a layer may fail if there is an error in a vertical logic path for some cell between the layer boundaries; but this in itself contributes to the identification of a bug. Currently, the method assumes a "ripple-carry" implementation of the carry/borrow chain of the internal adder/subtractor circuits that culminate at the quotient bits. In principle it is also applicable to implementations with different types of adders/subtractors, such as carry-look-ahead (CLA), as long as the partial remainder signals R_i are accessible via the procedures described here. Whether this method can be extended to other types of dividers, such as Goldsmith or SRT that are based on a different division algorithm, requires more insight.

ACKNOWLEDGEMENT

This work has been supported by a grant from the National Science Foundation, Award No. CCF-2006465.

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[2] M. Ganai and A. Gupta, *SAT-based scalable formal verification solutions*. Springer, 2007.

[3] M. Temel and W. A. Hunt, "Sound and automated verification of real-world RTL multipliers," in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 53–62.

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

[5] 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.

[6] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

[7] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1048–1053.

[8] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[9] 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.

[10] 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.

[11] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, p. 1183–1188.

[12] 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.

[13] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "SPEAR: hardware-based implicit rewriting for square-root circuit verification," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 532–537, 2020.

[14] ——, "Formal verification of divider circuits by hardware reduction," in *2023 19th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2023, pp. 1–4.

[15] J. Dasari and M. Ciesielski, "Formal verification of restoring dividers made fast and simple," in *60th Design Automation Conference (DAC)*. IEEE, 2023.

[16] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 113–118.

[17] 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.

[18] ——, "Automated test generation for debugging multiple bugs in arithmetic circuits," *IEEE Transactions on Computers*, 2018.

[19] 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.

[20] 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.

[21] 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.

[22] J. Dasari and M. Ciesielski, "Efficient formal verification and debugging of arithmetic divider circuits," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[23] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, second edition, 2002.

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

[25] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.