FARAD: Automated Formal Verification of Approximate Restoring Array Dividers

Chandan Kumar Jha[ш], Khushboo Qayyum[†], Muhammad Hassan[ш,†], and Rolf Drechsler[ш,†]
[ш] Institute of Computer Science, University of Bremen, Germany
[†] Cyber-Physical Systems, DFKI GmbH, Germany
{chajha, hassan, drechsler}@uni-bremen.de, khushboo.qayyum@dfki.de

***Abstract*—Approximate circuits have shown immense potential in the area of error-resilient applications. These circuits have tailored specifications depending on the resilience of an application towards the introduction of approximation. Formal verification is essential to guarantee the approximate circuit matches its tailored specifications. Hence, formal verification of approximate circuits has gained traction in recent years. However, most prior works focused on relaxed equivalence checking, i.e., only ensuring that the approximate circuits are within a specified error bound of the exact circuit. Recently, it was shown that formal error analysis is insufficient to ensure the approximate circuit matches its tailored functional specifications. However, this work was only limited to adders and multipliers. Hence, in this work, we propose a method called FARAD, that guarantees the approximate restoring array divider matches its functional specification. We use the functional specification of the approximate divider to generate the correctors. These correctors are then inserted in the approximate divider to generate the corrected divider. The corrected divider can then be formally verified by performing equivalence checking against a golden reference exact divider. If the corrected divider is equivalent to the golden reference divider, the approximate divider matches its functional specification. We generated more than half a million approximate dividers and verified them using FARAD.**

***Index Terms*—formal verification, combinational equivalence checking, approximate computing, restoring array dividers**

I. Introduction

Formal verification is essential to guarantee the correctness of a hardware design [1]. Given the ubiquitous use of approximate circuits, researchers have started focusing on using formal verification methodologies to ensure their correctness [2]–[4]. The correctness of an approximate circuit can be defined in two ways a) the approximate circuit's output is within a desired error bound, or b) the approximate circuit matches its functional specification. The verification technique that guarantees the error bound is also called *Relaxed Equivalence Checking (REC)* [5]–[7]. Error metrics like maximum error, mean absolute error, and mean square error are used to perform REC [8]. The second approach deals with ensuring that the functional specification of the approximate circuit is met. This approach is called the *Combinational Equivalence Checking (CEC)* for approximate circuits. This requires that the number of approximated blocks, the Boolean function of the approximated blocks, and the Boolean function of the exact blocks exactly match their specification [7].

While REC has received much attention and is suitable for designs where only error bound is a requirement, recently it was shown that the method is unsuitable, especially when the input data is skewed [7]. REC can allow multiple designs with different functional specifications to meet the criteria as it only checks for the error bound. However, for the same error bounds, multiple designs meeting the criteria can lead to larger-than-expected output errors, especially in skewed input data distributions. To cater to the skewed input data distributions the approximate circuits are tailored for some particular specification [9]–[11]. We explain this with the example of a *Full Adder (FA)*. If the input data is sparse most of the time, the input seen by the FA is 000. Thus the approximate circuit designer will not approximate for the case when the input is 000. However, for REC, the input pattern for which the data is approximated is immaterial, the error values only depend on the number of input patterns for which the FA can be approximated. So if as a result of the bug the approximation is for input pattern 111 instead of 000, the REC will not be able to detect this bug. In [7], it was shown that this can lead to a deterioration of 20 dB in the output quality.

Hence, there has been some recent investigations into developing CEC based methodologies for approximate circuits [7], [12]. In [12], *Answer Set Programming (ASP)* based formal verification of three types of approximate adders namely Ripple Carry Adder, Carry Save Adder, and Carry Look-ahead Adders was shown. However, this work has the following limitations a) this method requires implementing the specifications that are used as the golden reference, and b) the approach is limited to circuits having a constant cutwidth. Since *Restoring Array Divider (RAD)* does not have a constant cutwidth this approach cannot be used. In [7], a method was shown that relied on structural preservation, i.e., the location of the approximated blocks was known. This was used to correct these blocks using the same functional specification of the approximation. The reason behind correcting the design is that the exact design can be used as a golden reference for verification. However, they only showed their approach on Ripple Carry Adders and Dadda Tree Multipliers [13].

In this work, we develop an automated methodology based on the approach of correcting the approximate RAD from functional specification and then using the exact RAD as a golden reference for formal verification. The key idea behind this is the correction is only done for the cases that have been approximated. Hence, after correction, if the design does not match the exact golden reference, there is a bug in the approximate design. One of the most popular methods of designing Approximate RAD has been using functional approximation.

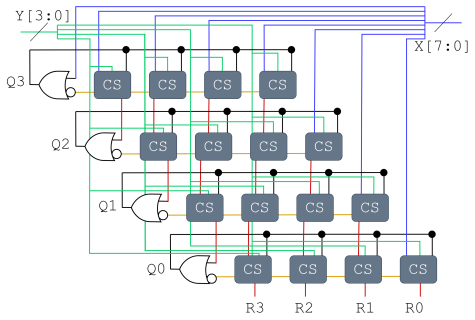

Fig. 1: Exact Restoring Array Divider

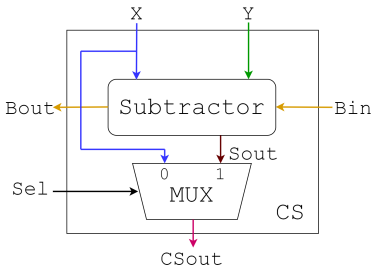

Fig. 2: Controlled Subtractor

In functional approximation, the Boolean function of the exact subtractor in the *Controlled Subtractor (CS)* is replaced with the Boolean function of the approximate subtractor to generate *Approximate Controlled Subtractor (ACS)* [14], [15]. There are different works based on the design of the approximate RAD [14]–[17]. Each of these works has three or four different Boolean functions for the approximate subtractors. In recent work, all possible Boolean functions were exhaustively generated for the approximate subtractors [15]. Since our goal in this work is to develop an automated verification methodology for the approximate RAD, we will show that our methodology works for any possible Boolean function approximation as long as the structure is preserved. Unlike prior works which focus on adders and multipliers [7], in dividers, we do not have direct access to the output of the approximated block. However, we will show how the Boolean function of the approximate subtractor is enough to generate the corrector for the entire ACS, and how the correction for the approximate subtractor is done without direct access to its output. Following are the contributions of our work:

- We propose an automated formal verification methodology for structurally preserved and functionally approximated RAD called FARAD.
- FARAD is based on automatically correcting the approximate RAD and then uses combinational equivalence checking to formally verify it using the exact RAD as the golden reference.
- We show how generating the corrector for the approximate subtractor is equivalent to generating the corrector for the ACS.
- We generated and verified more than half a million approximate RAD designs thus showing the efficacy of FARAD.

The rest of the paper is organized as follows. In Section Section II, we discuss the necessary background required for the paper. In Section III, we explain the overall FARAD methodology in detail. In Section IV, we discuss the results obtained using the FARAD methodology. We conclude the paper in Section V.

II. Preliminaries

In this section, we discuss the necessary background required for the paper. We will discuss RAD, functional approximation, approximate RAD, and formal verification.

A. Restoring Array Dividers

The block diagram of the 8 by 4 RAD is shown in Fig. 1. The dividend is the 8-bit input *X[7:0]* (*shown in blue*) and the divisor is the 4-bit input *Y[3:0]* (*shown in green*). The condition to prevent the overflow in the result of a *2N* by *N* RAD is that the *N* most significant bit of the input, i.e., *X[7:4]* must be less than *Y[3:0]* [18]. This ensures that the results, i.e., the quotient (*Q[3:0]*) and the remainder (*R[3:0]*) can be represented using 4 bits. The 8 by 4 RAD contains 16 CSs with 4 in each row as shown in Fig. 1. CS is a subtractor with the multiplexer selecting between the subtracted result and one of the inputs as shown in Fig. 2. The CS has four inputs namely X, Y, Bin, and Sel. It has two outputs namely Bout and CSout. The multiplexer allows the subtracted result to pass to the output in two cases, i.e., for these two cases the *Selector Line (Sel)* has a value of 1. i) When the *Most Significant Bits (MSBs)*, that are obtained at each stage either form the bits of *X* or the result of the previous stage controlled subtraction is a 1, or ii) the MSB of the subtraction result is a 0, i.e., the result of the sub is a positive number. In all the other cases the Sel has the value 0, and the input is copied to the output as shown in Fig. 2. The Sel at each stage gives the quotient value and after the *N* stages the remainder is obtained from the last controlled subtraction.

B. Functional Approximation

Approximation can be introduced in a design in several ways like truncation, voltage overscaling, overclocking, etc [2]. In this work, we have used the designs that have been functionally approximated [4], [19]. In functional approximation, the Boolean function of the design is replaced with another Boolean function. A typical example is used where the XOR gate is replaced with an OR gate as occupies a lesser area, has a lesser delay, and consumes less power [19]. Since the application that uses the approximate hardware is known, the functional approximation is tailored to obtain the best benefits while producing the least deterioration in the output quality. It has been shown that for the same number of ACS in an approximate RAD, the different approximate Boolean functions produce very different deterioration in the output quality. The functional approximation technique has been widely used as the approximation strategy for the design of approximate RAD.

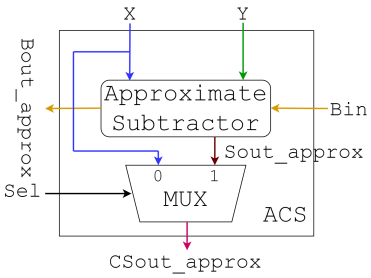

Fig. 3: Approximate Controlled Subtractor

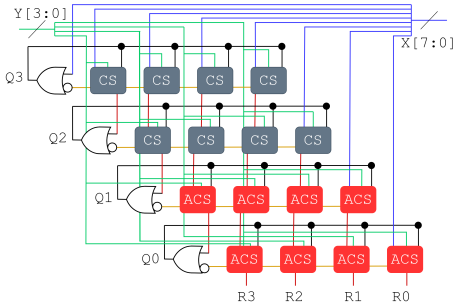

Fig. 4: Row Approximate Restoring Array Divider

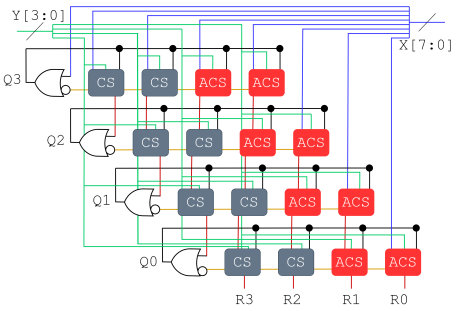

Fig. 5: Column Approximate Restoring Array Divider

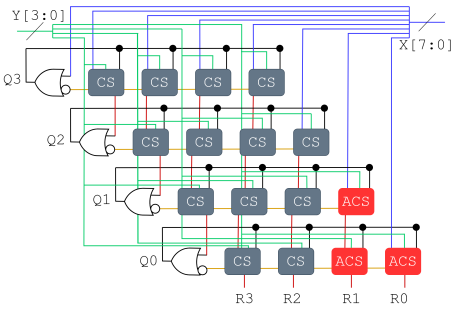

Fig. 6: Triangular Approximate Array Divider

C. Approximate Restoring Array Dividers

Approximate RADs are mostly designed using functional approximation [14]–[17]. The Boolean function of the subtractor in CS is replaced with the approximate subtractor to generate the ACS as shown in Fig. 3. The ACS has four inputs namely X, Y, Bin, and Sel. It has two outputs namely Bout_approx and CSout_approx. In addition, which of the CS needs to be approximated is decided based on the application. There are three major approximation strategies a) Row Approximation: All the CS in a row are replaced with ACS (See Fig. 4), b) Column Approximation: All the CS in a column are replaced with ACS (See Fig. 5), and c) Triangular Approximation: All the CS are replaced with ACS in a triangular fashion (See Fig. 6). As discussed in Section I, we can see that the Sout_approx is not directly accessible as the primary outputs of the ACS are Bout_approx and CSout_approx.

D. Formal Verification

Formal verification is used to guarantee that the *Design Under Verification (DUV)* is equivalent to the *Golden Reference Design (GRD)* or some specification. In this work, we use CEC to perform formal verification using a miter circuit [20]. To build the miter circuit the inputs of the DUV and GRD that have the same name are combined. The outputs having the same name are fed as an input to the *Exclusive OR (XOR)* gate and all the outputs of the XOR gate are then fed as input to an OR gate. If the designs are equivalent for all the possible combinations of the input the output of the miter is 0 [21].

III. FARAD Methodology

In this section, we discuss the steps involved in the FARAD methodology. FARAD methodology consists of three major steps. a) Generation, b) Correction, c) Verification as shown in Fig. 7. We now discuss each of these steps in detail.

A. Generation of Approximate RAD

In the first step, we generate the approximate RAD. We generated the approximate RAD by using as input a) the exact RAD, b) the functional specification of the appropriate subtractor, i.e., the Boolean expression of the approximate subtractor, and c) the type and the count of the approximation. The generated approximate RAD have their structure preserved, i.e., which of the RADs have been approximated are known.

In this work, we have generated the approximate divider for all three types of approximation strategies namely a) row, b) column, and c) triangular. We generated all possible Boolean functions for the ACS. In the ACS the subtractor module is approximated. Since the approximate subtractor has three inputs (X, Y, and Bin) and two outputs (Bout_approx and Sout_approx) as shown in Fig. 3, for each output the number of possible Boolean functions is 2^{2^3}, i.e., 256. Since there are 256 combinations for each of the two outputs the total possible Boolean functions are 256×256, i.e., 65536. We chose the count of approximation to be 2, 4, and 6. For each combination of the type and the count of approximation, we generated 65536 ACS. These ACS were used to design 65536 approximate RAD for each combination.

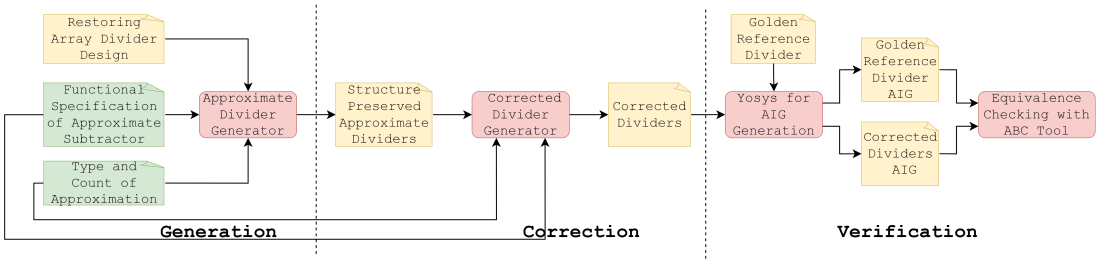

Fig. 7: FARAD Verification Methodology

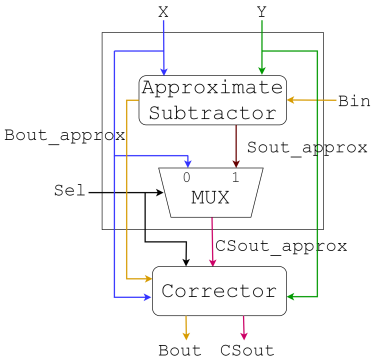

Fig. 8: Corrected ACS

```
module ApproxControlledSubtractor(X, Y, Bin, Sel,
     CSout_approx, Bout_approx);
input X, Y, Bin, Sel;
output CSout_approx, Bout_approx;
wire Sout_approx;
assign Bout_approx = (~X & ~Y & ~Bin) | (~X & ~Y & Bin) |
     (~X & Y & ~Bin) | (~X & Y & Bin) | (X & Y & ~Bin) | (X
     & Y & Bin) ;
assign Sout_approx = (~X & ~Y & ~Bin) | (~X & ~Y & Bin) |
     (~X & Y & ~Bin) | (X & Y & Bin) ;
assign CSout_approx = Sel?Sout_approx:X ;
endmodule
```

Listing 1: Verilog Code of 1-bit ACS Generated From Functional Specification in the form of a Truth Table

TABLE I: Truth Table of the Approximate Subtractor

X	Y	Z	Sout	Bout	Sout_ approx	Bout_ approx
0	0	0	0	0	**1**	**1**
0	0	1	1	1	1	1
0	1	0	1	1	1	1
0	1	1	0	1	0	1
1	0	0	1	0	**0**	0
1	0	1	0	0	0	0
1	1	0	0	0	0	**1**
1	1	1	1	1	1	1

B. Correction of the Approximate Divider

The approximate divider can be corrected by correcting the Boolean function of all the ACS in the approximate RAD. Since the approximate RADs structure is preserved we know which of the CS has been changed to ACS. Hence, we can correct each of the ACS. By correcting each of the ACS, the overall approximate RAD can be corrected.

For the correction process, the Boolean function of the approximate subtractor can be obtained from its functional specification. The corrector takes in five inputs, three of which come from the inputs of the ACS (X, Y, and Bin) and two from the outputs of the ACS (Bout_approx and CSout_approx). The block diagram of the corrector is shown in Fig. 8. The corrector uses the functional specification and the inputs to generate the corrected CSout and Bout. The corrector only corrects the output for which the CS has been approximated. The cases for which the approximation needs to be done are identified from the difference in the functional specification of the exact subtractor and the approximate subtractor, i.e., for which they produce different outputs. The correction is done by inverting the output for the cases that have been approximated. Since in approximation 1 is changed to 0 and vice versa, inverting the output is the same as correcting it. Once all the ACS are corrected the overall approximate RAD is also corrected.

To further explain this, we take an example where the approximate subtractor truth table as shown in Table I. While we have taken the example of a truth table, the functional specification can be given in any other format. In the truth table, we can see that the Borrow output (Bout) has been approximated for input values of 000 and 110 to generate the Bout_approx. Similarly, we see that the difference output (Sout) has been approximated for the input values of 000 and 100 to generate the Sout_approx. The Verilog code of the ACS, generated from the truth table of the approximate subtractor is shown in Listing 1. One crucial thing to observe here is the approximation is introduced in the approximate subtractor inside the ACS [14]–[17]. Thus, for correction of the ACS, we only need to correct the approximate subtractor. However, we do not have access to the Sout_approx output and can only access the CSout_approx as can be seen from Listing 1. We will now show how we can use the CSout_approx to correct the ACS even when we cannot access Sout_approx directly.

We can see that the Sel input is only dependent on the Bout_approx and the input X as can be seen from Fig. 1. Hence Bout_approx can be separately corrected, as Bout_approx is not dependent on Sout_approx as also shown in Listing 1. However, when the Bout_approx is corrected the Sel input is also corrected. Since Bout_approx has been corrected to Bout, this ensures that Sel is also correct. We will now show how this helps in the correction of the approximate subtractor inside

```
module CorrectedApproximateControlledSubtractor (X, Y, Bin,
      Sel, Sout_approx, Bout_approx, CSout, Bout);
input X, Y, Bin, Sel, CSout_approx, Bout_approx;
output CSout, Bout;
wire Sout;
assign Bout = (~X & ~Y & ~Bin & ~Bout_approx) | (~X & ~Y &
     Bin & Bout_approx) | (~X & Y & ~Bin & Bout_approx) | (~
     X & Y & Bin & Bout_approx) | (X & ~Y & ~Bin &
     Bout_approx) | (X & ~Y & Bin & Bout_approx) | (X & Y &
     ~Bin & ~Bout_approx) | (X & Y & Bin & Bout_approx) ;
assign Sout =  (~X & ~Y & ~Bin & ~CSout_approx) | (~X & ~Y
     & Bin & CSout_approx) | (~X & Y & ~Bin & CSout_approx)
     | (~X & Y & Bin & CSout_approx) | (X & ~Y & ~Bin & ~
     CSout_approx) | (X & ~Y & Bin & CSout_approx) | (X & Y
     & ~Bin & CSout_approx) | (X & Y & Bin & CSout_approx) ;
assign CSout = Sel?Sout:X;
endmodule
```

Listing 2: Verilog Code of 1-bit Corrected ACS Generated From Functional Specification in the form of a Truth Table

```
...
ApproxControlledSubtractor ACS1(X, Y, Bin, Sel,
     CSout_approx, Bout_approx);
CorrectedApproximateControlledSubtractor CACS1(X, Y, Bin,
     Sel, Sout_approx, Bout_approx, CSout, Bout);
...
```

Listing 3: Sinppet of the Insertion of the ACS Corrector in the Verilog Code

the ACS using CSout_approx.

The CSout_approx can either be X or Sout_approx depending upon the value of the Sel as shown in Fig. 8 and Listing 1. We will explain each of the two cases separately.

Case 1: Sel = 0

The corrector uses the same Sel input as the ACS. When CSout_approx is X, the corrector also needs to give the output as X, as no approximation has been done for this case. However, we saw that correcting Bout_approx corrects the Sel output. Hence, this also ensures that the corrected Sout has the value X.

Case 2: Sel = 1:

The second case is when the CSout_approx becomes the same as Sout_approx because Sel has a value of 1. For this case correction is required. However, we see that for this case correcting Sout_approx becomes the same as correcting CSout_approx. Hence, we can use the functional specification of the approximate subtractor, specifically of Sout_approx, to correct CSout_approx.

Thus, we see that even though we do not have access to Sout_approx, we can use the specification of the approximate subtractor to correct the ACS. The overall Verilog code of the corrected ACS is shown in Listing 2. We also show the snippet of the Verilog code where the corrector is inserted in Listing 3.

C. Formal Verification

In the previous stage, we corrected the outputs of all the ACS in the approximate RAD to generate the corrected RAD. Hence, the corrected RAD can now be verified by comparing it against a golden reference divider. The golden reference divider can be verified through exhaustive simulation or by using tailored verification techniques for RAD [22]. We use the golden reference RAD that has been formally verified and convert it to an *And-Inverter Graph (AIG)* using the Yosys synthesis tool [23]. AIG is a representation that consists of AND and NOT gates [24]. This representation is required by the ABC tool for the CEC [25]. Similarly, we convert the corrected RAD to AIG using the Yosys Synthesis tool. We give the AIG of the corrected RAD to be checked for equivalence against the golden reference RAD to the ABC tool. The CEC is done using the &cec command [20]. The ABC tool gives as output as equivalent if the corrected RAD is the same as that of the golden reference design. Since the corrected RAD matches the golden reference divider, this ensures that the approximate RAD also matches its functional specification.

IV. Results and Discussions

In this work, we have done the formal verification of the 16 by 8 approximate RAD as it is the most widely approximated RAD. The results obtained using the FARAD methodology are shown in Table II. The first column shows the approximate RAD bitwidth. The second column gives the type of approximation, i.e., row, column, and triangular. The third column gives the count of the approximation, i.e., how many rows, columns, or the size of the triangle as discussed in Section II. We have used the count of 2, 4, and 6 in this work. The primary inputs and primary outputs denote the sum of the input and output bit widths. Since the input dividend has a bit width of 16 and the divisor has a bit width of 8 the number of primary inputs is 24. The output quotient and remainder are each of bit width 8, hence the number of primary outputs is 16. The next column gives the number of 2-input AND gates in the corrected divider design. We want to highlight that there were no optimizations performed while converting the corrected RAD into the AIG. This is because we do not require an optimized circuit for the formal verification using CEC. We have added this value to give an idea of the size of the corrected RAD. The next column gives the number of levels in the corrected RAD. The last two columns show whether the designs were found to be equivalent and the time required for the verification.

We now select row number 5 as an example to explain Table II. The RAD dividend and divisor are 16 bits and 8 bits respectively, hence it is a 16 by 8 RAD. The type of approximation is column approximation as shown in Fig. 5. The count of approximation is 4, i.e., four columns have been approximated. There are 65536 different approximate RAD designs. The number of primary inputs is 24 (16-bit input for dividend and 8-bit for divisor). The number of primary outputs is 16 (8-bit for quotient and 8-bit for remainder). We have shown the average number of 2 input AND gates and the number of levels in the 65536 designs. The average count of 2-input AND gate is 1822 and the average level is 418. Please note that we did not perform any optimizations to reduce the size of the designs while converting them. Hence, the numbers

TABLE II: Experimental Results of the Formal Verification of 16 by 8 Approximate RAD

	Bitwidth	Type	Count	Number of Designs	Primary Inputs	Primary Outputs	AND2	Levels	Equivalence Check Result	Verification Time (sec)
1	16 by 8	Row	2	65536	24	16	1325	293	Equivalent	0.06
2			4	65536	24	16	1915	431	Equivalent	0.16
3			6	65536	24	16	2505	568	Equivalent	0.24
4		Column	2	65536	24	16	1185	272	Equivalent	0.08
5			4	65536	24	16	1822	418	Equivalent	0.15
6			6	65536	24	16	2458	564	Equivalent	0.24
7		Triangular	2	65536	24	16	808	176	Equivalent	0.04
8			4	65536	24	16	1040	231	Equivalent	0.06
9			6	65536	24	16	1431	323	Equivalent	0.11

shown are of un-optimized corrected RAD. However, this has little impact on the verification time as our verification times are very fast. All the designs were found to be equivalent. The worst-case verification time was around 0.15 seconds. The reason behind the fast verification time is after correction the corrected RAD has a similar structure as the golden reference RAD. The ABC tool &cec command exploits this property very effectively, thus the time taken for verification is low.

V. Conclusion and Future Work

In this work, we proposed a methodology for the automated formal verification of the approximate RAD called FARAD. FARAD guarantees that the approximate RAD matches its functional specifications. FARAD is based on the idea of correcting the ACS using the functional specification and then verifying it against a golden reference exact RAD. We verified more than half a million approximate RAD designs with different approximation strategies namely row approximation, column approximation, and triangular approximation. In the future, we plan to extend FARAD to structural independent approximate RAD.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) within the project VerA (DR 297/37-1).

References

[1] R. Drechsler, *Advanced formal verification*. Springer, 2004, vol. 122.

[2] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[3] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of aigs for error tolerant applications," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[4] C. K. Jha, A. Nandi, and J. Mekie, "Single exact single approximate adders and single exact dual approximate adders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 7, pp. 907–916, 2023.

[5] Z. Vasicek, "Formal methods for exact analysis of approximate circuits," *IEEE Access*, vol. 7, pp. 177 309–177 331, 2019.

[6] M. Schnieber, S. Froehlich, and R. Drechsler, "Polynomial formal verification of approximate adders," in *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2022, pp. 761–768.

[7] C. K. Jha, M. Hassan, and R. Drechsler, "cecapprox: Enabling automated combinational equivalence checking for approximate circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[8] S. Froehlich, D. Große, and R. Drechsler, "One method-all error-metrics: a three-stage approach for error-metric evaluation in approximate computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 284–287.

[9] C. K. Jha, S. Ahmadi-Pour, and R. Drechsler, "Input distribution aware library of approximate adders based on memristor-aided logic," in *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*. IEEE, 2024, pp. 577–582.

[10] Z. Li, S. Zheng, J. Zhang, Y. Lu, J. Gao, J. Tao, and L. Wang, "Adaptable approximate multiplier design based on input distribution and polarity," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 12, pp. 1813–1826, 2022.

[11] C. K. Jha, I. Doshi, and J. Mekie, "Analysis of worst-case data dependent temporal approximation in floating point units," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 2, pp. 767–771, 2020.

[12] M. Nadeem, C. K. Jha, and R. Drechsler, "Polynomial formal verification of approximate adders with constant cutwidth," in *2024 IEEE European Test Symposium (ETS)*. IEEE, 2024, pp. 1–6.

[13] D. Goldberg, "Computer arithmetic," *Computer Arch itecture: A Quantitative Approach, David Patterson and John L. Hennessy, Eds. Morgan Kaufmann, Los Altos, Calif., Appendix A*, 1990.

[14] C. Jha and J. Mekie, "Design of novel cmos based inexact subtractors and dividers for approximate computing: an in-depth comparison with ptl based designs," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 174–181.

[15] C. K. Jha, S. Ahmadi-Pour, and R. Drechsler, "Maradiv: Library of magic-based approximate restoring array divider benchmark circuits for in-memory computing using memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 7, pp. 2635–2639, 2023.

[16] K. M. Reddy, M. Vasantha, Y. N. Kumar, and D. Dwivedi, "Design of approximate dividers for error tolerant applications," in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2018, pp. 496–499.

[17] E. Adams, S. Venkatachalam, and S.-B. Ko, "Approximate restoring dividers using inexact cells and estimation from partial remainders," *IEEE Transactions on Computers*, vol. 69, no. 4, pp. 468–474, 2019.

[18] A. Gardiner and J. Hont, "Comparison of restoring and nonrestoring cellular-array dividers," *Electronics Letters*, vol. 7, no. 8, pp. 172–173, 1971.

[19] A. Dalloo, A. Najafi, and A. Garcia-Ortiz, "Systematic design of an approximate adder: The optimized lower part constant-or adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 8, pp. 1595–1599, 2018.

[20] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 836–843.

[21] D. Brand, "Verification of large synthesized designs," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE, 1993, pp. 534–537.

[22] J. Dasari and M. Ciesielski, "Formal verification of restoring dividers made fast and simple," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[23] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[24] R. Brummayer, A. Cimatti, K. Claessen, N. Een, M. Herbstritt, H. Kim, T. Jussila, K. McMillan, A. Mishchenko, F. Somenzi *et al.*, "The aiger and-inverter graph (aig) format version 20070427," 2007.

[25] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.