Verification of Polynomial Division using BDDs

Nathan Sartnurak, Kidus Yohannes, Fernando Araujo

Abstract – **For this project, we proposed the verification of polynomial division using BDDs to understand if it can be done just like it has been done for ZDDs. We can represent our polynomials as BDDs. However, reducing our polynomials is the biggest challenge when distinguishing the term ordering for variables within our polynomials to apply polynomial division. This report discusses multiple equations and concepts for attempting polynomial division. We used tools such as CUDD and Singular to define and verify the operations for a chain of OR gates and a 2-bit multiplier. Additionally, ZDD graphs will be given to visualize why polynomial division is possible for ZDDs and not BDDs.**

I. Introduction

In the world of verification and testing of digital circuits, polynomial formal verification of arithmetic circuits is used heavily in VLSI as it makes verifying highly error-prone circuits the most efficient way possible. Polynomial division using Gröbner basis (GB) reduction is used to derive canonical representations of our circuit from internal nodes to our primary inputs. This method is used heavily to verify large-scale circuits, which can be implemented in various formal verification tools. However, some of these tools prove to be infeasible as the circuits become larger and more complex. For this project, we used the CUDD 3.0.0 package to represent Boolean polynomials as characteristic sets of binary decision diagrams (BDDs) and zero-domain BDDs (ZDDs) for their ability to design a GB reduction algorithm effectively. Additionally, we would develop a couple of Singular scripts that verify two separate methods of GB reduction to verify our ZDD structures that would be written out and used for the reduction iterations for our circuits.

In the paper, “Boolean Gröbner Basis Reductions on Finite Field Datapath Circuits using the Unate Cube Set Algebra,” the authors discuss the methods of implementing GB reductions using ZDDs, which planted the idea of replicating these results but using BDDs [1]. This project aimed to develop methods of GB reduction for BDDs, which implies representing different sets of circuits using BDDs and identifying methods of transversing our input BDD nodes, such as that of a ZDD. The problem is that GB reduction hasn’t been done for BDDs before because it’s impossible to represent a BDD with distinguishable leading terms, which is crucial for the GB reduction algorithm. In ZDDs, we can distinguish our leading terms based on term order with index numbers based on the edges of our variables. However, BDDs are represented as whole functions without indication of leading terms.

II. Preliminaries

To perform a Gröbner basis reduction, we define a Boolean domain as a finite field of two elements ($B = F_2$), where operations are done for modulo 2. Under this finite field, a Gröbner basis reduction can be defined as $f \xrightarrow{g} r$, where f, g, and r are polynomials contained within the finite field of F_2. The figure below defines the GB reduction in terms of Boolean polynomials.

$$f \xrightarrow{g} r = f - \frac{lt(f)}{lt(g)} \cdot g \tag{1}$$

$$= f - \frac{lm(f)}{lm(g)} \cdot g; \ \ (lt(f) = lm(f)) \tag{2}$$

$$= f + \frac{lm(f)}{lm(g)} \cdot g; \ \ (-1 = +1 \ (mod\ 2)) \tag{3}$$

$$= f \oplus \frac{lm(f)}{lm(g)} \cdot g; \ (as \ + \ (mod\ 2) = \oplus\ ) \tag{4}$$

This derivation shows that the reduction can be simplified into finding the leading monomial of f ($lm(f)$) and the leading monomial of g ($lm(g)$). The sum operation is an XOR due to the Field of 2. The product and divide functions are also defined under the modulo 2 field. Under F_2, gate-level Boolean logic functions for an inverter, and 2 input and gate, a 2 input and gate, a 2 input or gate, and a 2-input exclusive or gate. The following equations below show the corresponding gates under the F_2 where z is the output, with a and b are inputs:

$$z = \neg a \ \rightarrow \ z + a + 1 \qquad (mod\ 2) \tag{5}$$

$$z = a \wedge b \ \rightarrow \ z + a \cdot b \qquad (mod\ 2) \tag{6}$$

$$z = a \vee b \ \rightarrow \ z + a + b + a \cdot b \quad (mod\ 2) \tag{7}$$

$$z = a \oplus b \ \rightarrow \ z + a + b \qquad (mod\ 2) \tag{8}$$

Now consider this circuit below and perform a Gröbner basis reduction upon it. Assume variable RTTO $\{z > y > x > d > c > b > a\}$ upon the circuit in Figure 1.

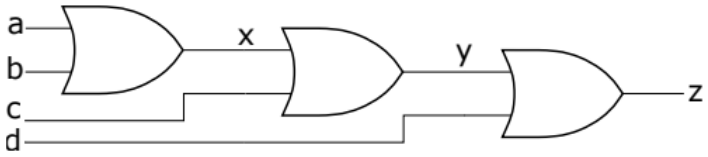


Figure 1: Chain of OR gates implementation. [1]

Therefore, the following Boolean polynomials can be defined as the following using the gate level Boolean functions under F_2 where the leading monomials of $f_1, f_2, and\ f_3$ are z, y, and x respectively.

$$f_1 = z + yd + y + d; \tag{9}$$

$$f_2 = y + xc + x + c; \tag{10}$$

$$f_3 = x + ba + b + a; \tag{11}$$

To create the canonical representations of the circuit above, where the outputs are represented as their inputs alone, a GB reduction can be performed through each corresponding polynomial until the leading terms are no longer present in the computed resulting reduction formula.

1) $z \xrightarrow{f_1} yd + y + d$
2) $yd + y + d \xrightarrow{f_2} y + xdc + xd + dc + d \xrightarrow{f_2} xdc + xd + xc + x + dc + d + c$
3) $xdc + xd + xc + x + dc + d + c \xrightarrow{f_3} xd + xc + x + dcba + dcb + dca + dc + d + c \xrightarrow{f_3} xc + x + dcba + dcb + dca + dc + dba + db + da + d + c \xrightarrow{f_3} x + dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c \xrightarrow{f_3} dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c + ba + b + a = r$

ZDDs are an efficient data structure for finding the leading monomials. This can be seen in Fig 2 while traversing through the then edges (edges that are solid) on the ZDDs using the polynomials of equations 9, 10, and 11. This is a special feature of the ZDDs as they differentiate themselves from the standard binary decision diagrams (BDDS) by eliminating vertices that then edge point to the zero terminal node and merge isomorphic graphs. By using ZDDs, basic functions like taking the product and performing division are clearly defined using the ZDD package. However, taking the sum between two different ZDDs is not defined in the CUDD standard library but was implemented in our design, which will be reviewed later in this paper.

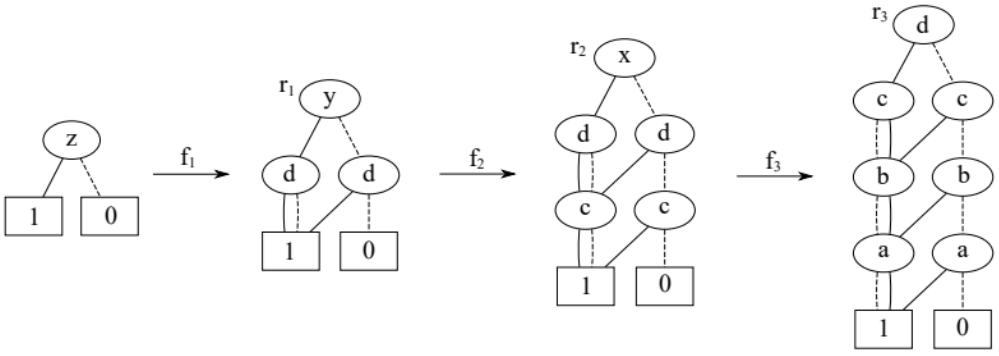


Figure 2: ZDD transversal using GB reduction of the chain of OR gates implementation. [1]

ZDDs also allowed us to perform reduction in a way that can cancel multiple monomials in a single step. This optimization can be seen only through the ZDDs. First, define $r_i \xrightarrow{f_j} r_j = r_i + q \cdot f_j$. The following resulting remainder, r_j, can be written in terms of the then and else edges as the following:

$$r_j = else(r_i) + then(r_i) \cdot else(f_j) \tag{12}$$

Therefore, to calculate the remainder through ZDDs, first find the else and then for the input (r_i). Then calculate the else edge of the polynomial f_j and take the ZDD product between the then edge of r_i and the else edge of f_j, and take the modulo 2 sums using the else edge of r_i. This

algorithm is more optimized because it does not require us to traverse until a terminal node of 1 is reached or when the resulting quotient is completely reduced. Instead, the algorithm only needs to calculate the top nodes, traverse once to the true and else edges, and calculate the reduction after getting those nodes. We developed a singular implementation of Gröbner basis reduction, developing it on the CUDD package.

III. Singular Implementation

The Singular tool was used to help verify our circuit's functionality after every reduction to gain the best understanding of what the GB reduction algorithm should be doing for our BDD and ZDD plots. Constructing the original GB reduction scheme, as seen in the recently mentioned paper [1], involved back-to-back calculations of the circuit's reduced parameters, which may have proved to be a fundamental challenge if implemented using ZDDs. Fortunately, there is an improved GB algorithm that focuses on the 'then' and 'else' factors of our ZDD functions, making it simpler to derive our reduction results with ease.

Two implementations have been done for the benefit of calculating the GB reduction for ZDDs. Algorithms 1 and 2 explain them further.

Algorithm 1: Multivariate Reduction [1]

1: **procedure** multi_variate_reduction($f, \{f_1, \dots, f_s\}, f_i \neq 0$)
2: $u_i \leftarrow 0;\ r \leftarrow 0; h \leftarrow f$
3: **while** $h \neq 0$ **do**
4: **if** $\exists i\ s.t.\ lm(f_i) \mid lm(h)$ **then**
5: choose $i\ least\ s.t.\ lm(f_i) \mid lm(h)$
6: $u_i = u_i + \frac{lt(h)}{lt(f_i)}$
7: $h = h - \frac{lt(h)}{lt(f_i)} \cdot f_i$
8: **else**
9: $r = r + lt(h)$
10: $h = h - lt(h)$
11: **return** $(\{u_1, \dots, u_s\}, r)$

Algorithm 2: Reduction under RTTO [1]

1: **procedure** multi_mon_red($z_i, poly\_list$)
2: **for** each $g \in poly\_list$ **do**
3: **if** $index(g) == index(z_i)$ **then**
4: $z_i = else(z_i) + then(z_i) \cdot else(g)$
5: **return** z_i

Algorithm 1 is the original reduction method that calculates the remainder between 2 functions in various steps based on the quotient of leading terms for each function. If the quotient exists, continue to the calculation as the h continues to reduce to 0 and the remainder gathers the formal specification of our function using only primary inputs without implementing the interior nodes. Algorithm 2 performs the same functionality as Algorithm 1; however, it only focuses on the indexes of the functions when implemented in the CUDD package. Additionally, implementing our designs in ZDD makes it convenient to implement the THEN and ELSE commands for transversing the nodes after every reduction until complete. Both algorithms are the same and were implemented in Singular to ensure that the CUDD code is correct based on reordering and overall execution.

The first algorithm was to help our team understand the math and how we could optimize the calculations with our built-in don't-care functions that would have affected the entire circuit by almost resembling it as a BDD, which would not have been as effective after the 2nd algorithm implementation. Now that our algorithms were written in our singular scripts, we

would test the chain of OR gates and 2-bit multiplier designs and then compare them with the CUDD implementation.

IV. CUDD Implementation

A. CUDD Package

We use the CUDD package to implement and work with ZDDs and BDDs. We used online resources such as David Kebo's site [2] for examples and to understand the functions. We also make use of package documentation [3].

B. Use of GIT

We used GitHub for our project. It allowed us to all work on different sections at the same time by working on separate branches. It allowed us to keep track of our changes in cases we needed to revert files. It allowed us to save our project to the cloud and provide easy use for anyone interested in running it. All one must do to run our code is to download the repo, run the start-up script, and then run nanotrav [4].

C. Start-up Script

Inside the GIT repo for our project, we include the CUDD package (as a tar file) and a start-up script. This script automatically extracts the tar file, restores the main.c file that is saved on git, and then restores the makefile. The modified makefile is set up so debugging works in VS Code. This allows the user easy setup for working with CUDD and our project. The source file is in the nanotrav directory. After modifying main.c, you must run 'make check' to recompile the code. Finally, all that's left is to run the nanotrav command to run the code.

D. Main Functions

i. Main

Inside of main is where we define our nodes and create our ZDD functions. We began our project with much testing and experimentation, ensuring we understood how to build ZDD functions and verifying them against examples in the reference paper. We developed on multiple branches, running different examples, such as verifying the function for a four-input XOR gate. We also developed incrementally, starting with simple functions and slowly incorporating the steps required for polynomial division, ensuring our outputs looked as expected and debugging throughout. We faced several challenges that we were able to overcome, which provided us with some insight into working with ZDDs and the polynomial division algorithm.

For the final code structure inside of main, we implemented the polynomial reduction for the chain of OR gates and 2-bit multiplier. We initially define our node manager, all our node variables in RTTO, and their respective names. Then we create the single variable ZDDs for all the nodes to build our functions. Then we define our polynomial gate functions using the CUDD and helper functions. Finally, we call our multi-monomial reduction function, which computes the reduction with all the given polynomial functions. We dump out the dot files and visually verify them against the expected results.

ii. ZDD & BDD Plots

One of the first functions we implemented was dumping ZDDs and BDDs into dot files. That way, we could quickly visually verify whether the functions we created were correct. This allowed us to debug our code when something went wrong, and it allowed us to examine the results from our ZDD operations. We used CUDD's built-in functions DumpDot and zddDumpDot and made use of the available arguments. For example, in addition to passing in the manager and ZDD node to dump, we specified the dot filename automatically, appending the extension ".dot". We also keep track and pass in an array of node variable names, so the plot shows the variable name for each node in the ZDD. Finally, we pass in the output name to label the plot and the function it represents.

iii. ZDD Don't-Care

When working with ZDDs and BDDs, we realized that a major difference arose when we attempted to represent a single variable. We discovered that to represent a single variable ZDD correctly, we need to specify that the other variables are NOT don't-cares explicitly. Otherwise, when we convert from a BDD to a ZDD, the variables not involved in the function are implicitly don't-cares and included in the plot representation. The set represented by the ZDD becomes much larger, like the example provided below. We determined that while building our function as a BDD (before converting it to a ZDD), we would AND our function with the NOT of every other variable not involved to represent our ZDDs accurately.

$$(d, c, b, a) \rightarrow f_{BDD}(ab + c) \rightarrow g_{ZDD}\{(ab, abc, abd, abcd), (c, ac, bc, cd, abc, bcd, acd, abcd)\}$$

However, we saw this would be inconvenient to do every time we wanted to create a function, especially for circuits involving many variables. We determined a better approach would be to initially create a single variable ZDD for every variable in our circuit and build our functions using those. That way, we would only have to deny the don't-cares once in the beginning explicitly. And because the operation was simple enough, all we had to do was loop through all the existing variables and AND their NOT version with the desired ZDD variable. We abstracted this into a helper function that we could call for every variable. At this point, we had developed our ZDD helper functions for creating polynomials, so working with the single variable ZDDs was straightforward.

iv. Modulo 2 Sum

As described previously, the operators used in polynomial reduction are applied for modulo 2. This means the sum operation acts like an XOR function, where the output is true when either input is true but not at the same time. A built-in function that implements this operation with ZDDs does not exist in CUDD, so we created this function ourselves. We used CUDD's built-in functions zddUnion, zddIntersect, and zddDiff to implement the following equation:

$$f \oplus g = (f \cup g) - (f \cap g) \tag{12}$$

v. Modulo 2 Product

To implement the product operation for modulo 2 we used CUDD's built-in function zddUnateProduct. This function operates similarly to the sum function, where the intermediate

partial product terms are added and canceled modulo 2. Initially, we used zddProduct, which gave us issues and was inconsistent until we realized we wanted the unate version.

vi. ZDD AND, OR, XOR Gates

While implementing the polynomial gates as ZDDs, we thought it would be useful to create helper functions that implement each gate type. For example, given the gate: $z = a \oplus b$, the polynomial representation is: $f = z + ab + a + b$. In terms of ZDD operations, first we compute the zddUnateProduct of a and b Then we use module 2 sum to combine with z, a, and b. We abstract all these ZDD operations into a single function where all we must do is pass in the associated parameters and it returns the desired ZDD. We made these functions for AND & OR gates as well.

vii. Multi-Monomial Reduction

One of our final function implementations was creating the multi-monomial reduction algorithm. We referenced the pseudocode provided in the paper to concisely implement the algorithm using a single for loop and single-step reduction. And because we incrementally developed and tested our code, we were confident and ensured that all our function calls and one-step reduction process were correct and without bugs. As parameters, our function takes the node manager, the starting node to reduce, and the array of polynomial ZDDs to reduce with. The for-loop iterates through each polynomial and applies division using the one-step reduction. We use Cudd_T and Cudd_E to get the 'then' and 'else' parts of the ZDD for the formula. We use zddUnateProduct and zddModSum to compute the product and sum operations. Throughout this we also ensure correct reference and dereferencing of the ZDD nodes. We iteratively update the initial node to reduce so that the result is the complete reduction made up of only primary inputs.

viii. Add Node Names

Initially, our dot file plots displayed each node's address in the middle and the variable name on the side. This was not a big deal, but we wanted to make our plots look like the examples we saw in the paper and online. It also made keeping track of the variable name easier by having it in the middle instead of to the side and some arbitrary address in the middle. So we created a short Python script to modify our dot files to replace the addresses with the respective variable names. The script uses regex to match and find all the variable names and associated addresses. Once it has those, it just iterates through each one and replaces any occurrence of an address with the respective variable name. One thing to note is that it keeps track of would-be identical node addresses/names by appending subscripts to differentiate the nodes. Finally, it applies this update, rewrites the dot file, and does it for every dot file in the current directory.

V. Results

A. Chain of OR gates

The first circuit we verified our polynomial division implementation with was a chain of OR gates, as seen in Figure 1. We chose this circuit because it was the example given in the paper and allowed us to follow along and verify that each intermediary step was accurate. So given the variables with RTTO, we implemented the ZDD gate polynomials f1, f2, f3 and verified that the plots matched the ones in the paper.

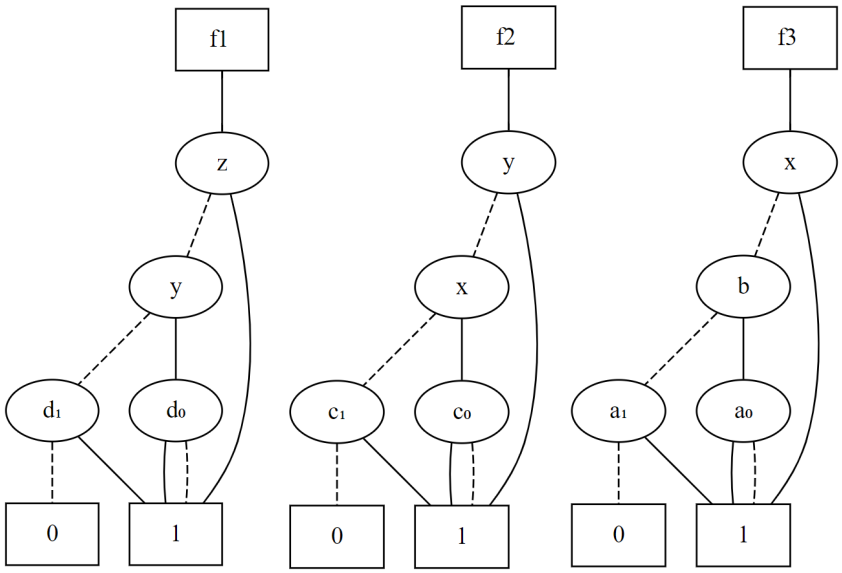


Figure 3: ZDD gate polynomials for a chain of OR gates.

We can visually confirm a given ZDD's representation by following all the 'TRUE' paths. So for example, f1 has a valid path through z, then through yd and y, and lastly d. This makes up the set:

$$f_1 = z + yd + y + d$$
$$f_2 = y + xc + x + c$$
$$f_3 = x + ba + b + a$$

Once we confirmed our ZDD polynomials, we applied polynomial division on the output net 'z'. Before implementing the multi-monomial reduction, we individually computed the one-step reduction for each polynomial. This means we computed r1, r2, r3 and verified that they matched the plots in the paper.

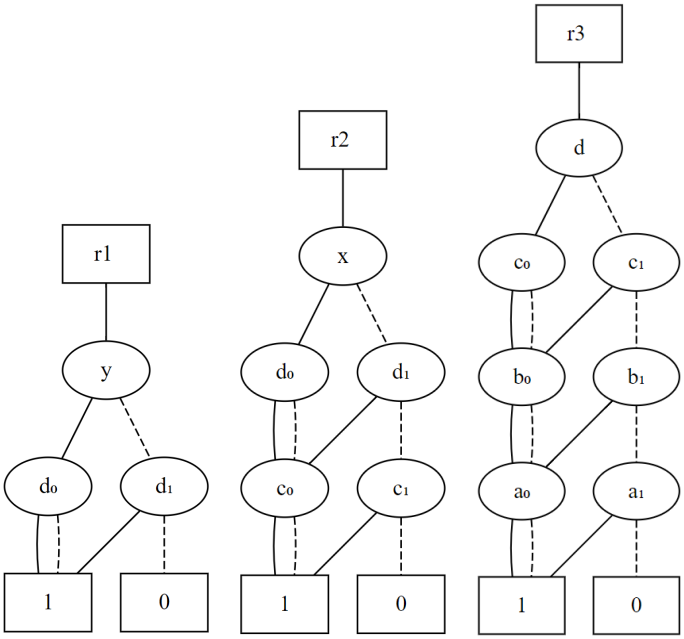


Figure 4: ZDD reduced polynomials for a chain of OR gates.

We can see in Figure 4 the reduced polynomials for r1, r2, r3. These match the ones shown in Figure 2 and highlight how, despite the exponential growth in terms after each reduction, the ZDD representation only grows linearly. Only one to two nodes are needed to represent several terms. Once we verified that these individual reductions were correct, we applied the multi-monomial reduction algorithm, so that the for-loop handles all the reductions for us. We verified that the result matches the final ZDD, which comprises all the primary inputs and represents the chain of OR gates, where the output is true if any variable is true.

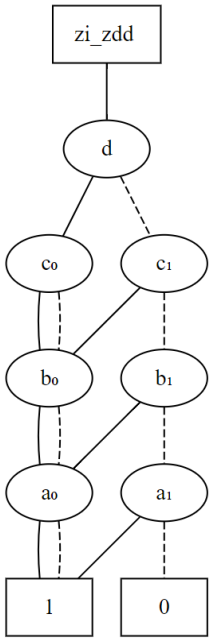
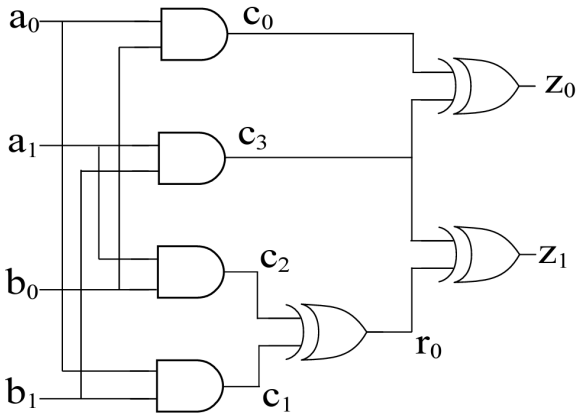


Figure 5: Final ZDD reduction for a chain of OR gates.

B. 2-bit Multiplier

To confirm that our algorithm is working, we used a 2-bit modulo 2 multiplier circuit and the corresponding polynomials to represent each gate under variable RTTO ($\{z_0 > z_1 > r_0 > c_0 > c_3 > c_1 > c_2 > a_0 > a_1 > b_0 > b_1\}$) as an input into our algorithm and checked to see if the output ZDDs represent the function in terms of the inputs only. This circuit and the polynomials can be seen below:



$$f_1 = c_0 + a_0 \cdot b_0;\ lm = c_0;\quad f_2 = c_1 + a_0 \cdot b_1;\ lm = c_1;\quad f_3 = c_2 + a_1 \cdot b_0;\ lm = c_2;$$
$$f_3 = c_2 + a_1 \cdot b_0;\ lm = c_2;\quad f_4 = c_3 + a_1 \cdot b_1;\ lm = c_3;\quad f_5 = r_0 + c_1 + c_2;\ lm = r_0;$$
$$f_6 = z_0 + c_0 + c_3;\ lm = z_0;\quad f_7 = z_1 + r_0 + c_3;\ lm = z_1;$$

Figure 6: A 2-bit modulo Multiplier circuit and their polynomials using RTTO. [1]

When computing the Groebner basis, the outputs z_0 and z_1 should be represented in terms of the inputs. The following equations are the results after computing the Groebner basis, where f is a set of polynomials that represent each gate ($f_1, f_2, f_3, f_4, f_5, f_6, f_7$):

$$z_1 \xrightarrow{f} r_1 = a_0 \cdot b_0 + a_1 \cdot b_1 \tag{13}$$

$$z_0 \xrightarrow{f} r_0 = a_0 \cdot b_1 + a_1 \cdot b_0 + a_1 \cdot b_1 \tag{14}$$

To test if our algorithm computed the correct results, we compared our ZDDs against the results of the ZDDs that were manually drawn using the CUDD and modulo 2 sum functions using the equations above. The figure below shows our results after passing the modulo 2 multiplier circuit as an input. The nodes z0_out and z1_out represent the output from the algorithm, while z0_manual and z1_manual are the manually drawn ZDD diagrams when using the equations above.

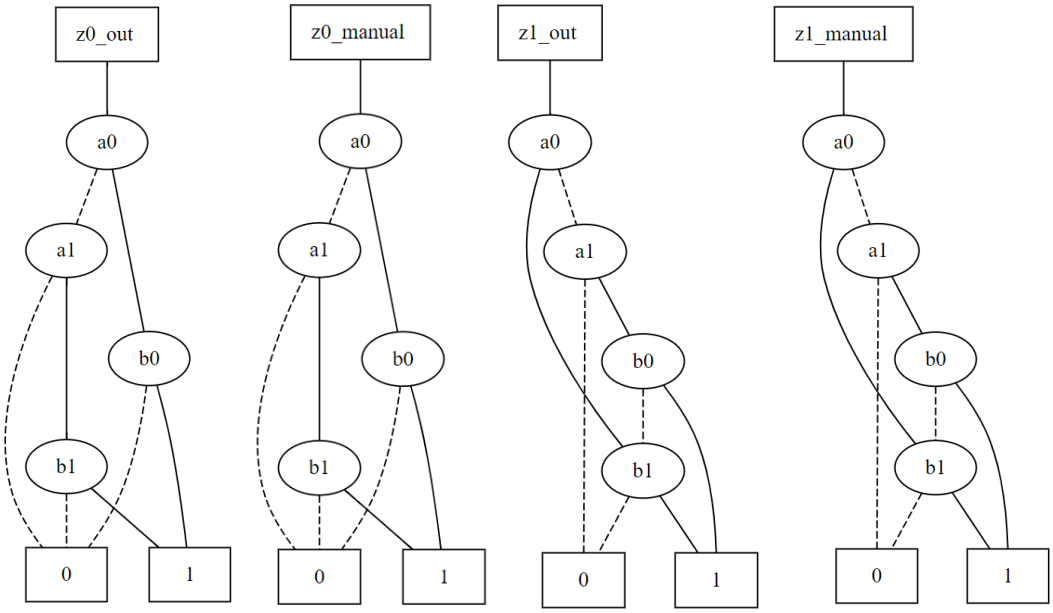


Figure 7: ZDD output from the 2-bit modulo Multiplier circuit and the result when creating the ZDDs manually using equations (14) and (15). [1]

When comparing the z0_out to z0_manual, they are the same. We can also see that the 'THEN' edges from the top node traverse to 1 and represent the leading term $a_0 \cdot b_0$, which matches the leading term of equation (13). This can also be seen when comparing the z1_out to z1_manual where the corresponding ZDDs match, as well as the leading terms when taking the 'THEN' edges, resulting in a leading term $a_0 \cdot b_1$, which matches the leading term of equation (14). By doing this, we can verify that our algorithm can successfully calculate the Groebner basis.

VI. Challenges

One main challenge was creating the functions accurately to match what was written on [1]. To overcome this issue, we used the C breakpoint debugger on VS Code to step through the code to ensure the correctness of our functions. Another challenge was to remove the don't-care on the CUDD package. We learned that every node created in the CUDD package includes don't-care conditions, which are just other node variables. As stated above, we created a function to remove all the don't-cares, but realizing that this was an issue in our design was a challenge. We also had issues figuring out how to use the ZDD divide and product functions correctly, but we eventually figured it out. We had some trouble understanding the module sum function using ZDDs. It took us some time for each ZDD from [1] to be displayed using a modulo sum function that was not part of the CUDD package library. Another challenge we had was using the correct ZDD product function. We used zddProduct during our first implementation, but we kept getting a NULL result after taking the product between two ZDD nodes. We switched to using zddProductUnate because we wanted to take the product of two unate covers, which solved our issue. The final challenge was putting the polynomials in the correct order for the 2-bit multiplier circuit. Since our implementation uses RTTO, this means that the polynomials representing the circuit must also follow the same ordering when calculating the reduction. After creating the correct order, we created outputs that match the [1].

VII. Conclusion

A. Concepts Learned

At the beginning of this project, we came up with a goal to perform polynomial division for GB reduction using BDDs. We concluded that it was impossible because performing the multi-variate reduction requires the identification of leading terms and monomials that would be put into algorithm 1 and carries out the calculation through unate products and modulo addition. Still, BDDs do not have a clear indication of transversing between all the variables without disregarding all the don't-cares that reduce our nodes drastically. By eliminating these nodes, the transversal is unlikely to be carried out correctly. Therefore, the reduction would never be completed as intended. In Figure 8, you will see the visual difference between BDDs and ZDDs.

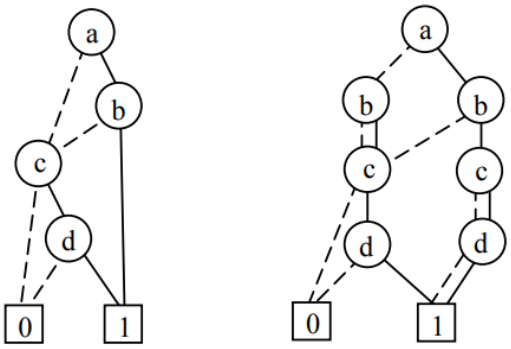


Figure 8: BDD (left) and ZDD (right) for F = ab + cd. [5]

From the figure above, the don't-cares node b on the ELSE edge and nodes c and d on the THEN edge are removed in the BDD. If transversed beginning at the 0-terminal, we would need to be able to see all the nodes there, or else this will cause our output to remainder to another function, not the same as F as compared to the ZDD. Therefore, we ruled out the possibility of polynomial division using BDDs to focus on why ZDDs worked the best.

B. Contributions

Throughout this project, we worked closely together in understanding the polynomial division algorithm and in code development. For example, we all spent time reading the paper [1], reading online resources for Singular/CUDD, developing code, putting together our report, and providing documentation. However, here are some ways we individually focused our work and especially added value to the team.

i. Nathan

Nathan helped create the setup script that unzips the CUDD package. He set up the GitHub repo and created a git-ignore file that ignored all executable files in the CUDD package and kept only the source code files (like c files, header files, makefiles, etc.) in the GitHub repo. He contributed by also planning and deriving ways to solve for GB reduction with the help of Fernando. He also outlined what functions needed to be created, making writing the code more straightforward and accessible. He also worked with Kidus and Fernando to create those specific functions. He also contributed to debugging by setting up the VS Code c debugger to perform breakpoint debugger upon our code. This allowed the group to better understand the data structure in the CUDD package.

ii. Fernando

Fernando helped develop the Singular code to verify the reduction process of the ZDD data structures using two separate polynomial reduction techniques. During every team meet-up, Fernando would oversee hand-writing the reduction algorithms, then explain to the group what the end goal of each reduction should look like and determine if the graphs had taken a different form. Fernando would help Kidus code the GB reduction algorithm in the CUDD code and help debug any problem regarding this project's analytical side. Occasionally, he would also find a bug or two in the code when it came to reordering the terms and for loop implementation to gain the correct ZDD resultant for our circuits.

iii. Kidus

Kidus helped develop the CUDD code and documentation, like the inline comments and readme file on GitHub. When the group was all working together in person, he helped drive while Fernando and Nathan gave suggestions and guidance on what to implement or different ways to debug issues that arose. He helped implement some of the helper functions in the code and helped overcome some challenges like understanding the BDD to ZDD conversion and considering the don’t-cares. And using resources to discover the zddUnion and zddUnateProduct functions for accurately creating the ZDD polynomials. He also created the ‘add_node_names.py’ Python script to improve the ZDD dot files.

C. Final Recap

Polynomial division is confirmed impossible using BDDs as we cannot identify a transversal method for reduction as BDDs are represented as whole functions as opposed to ZDD representing functions with leading monomials, which is crucial for reduction. It’s a representation of the whole function instead of representing the leading monomials. However, our implementation of polynomial division using ZDDs was successful. We approached our development incrementally and verified our individual functions and building blocks. Using variable RTTO, we created the variable nodes and applied our reduction algorithm for a 4-input OR gate and a 2-bit multiplier. We verified that the output ZDD plots comprised only the primary inputs and correctly represented the respective circuits.

References

[1] U. Gupta, P. Kalla and V. Rao, "Boolean Gröbner Basis Reductions on Finite Field Datapath Circuits Using the Unate Cube Set Algebra," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 3, pp. 576-588, March 2019, doi: 10.1109/TCAD.2018.2818726.

[2] D. Kebo, “The CUDD package, BDD and ADD Tutorial,” https://davidkebo.com/cudd/ (accessed Dec. 15, 2023).

[3] “The Cudd Package (Internal),” web.mit.edu, May 17, 2005. https://web.mit.edu/sage/export/tmp/y/usr/share/doc/polybori/cudd/cuddAllDet.html#prototypes (accessed Dec. 15, 2023).

[4] “ECE6745 Final Project,” Github, Dec. 14, 2023. https://github.com/NathanNator/ECE6745_Final_Project/tree/main

[5] A. Mishchenko, “An Introduction to Zero-Suppressed Binary Decision Diagrams,” people.eecs.berkeley.edu, https://people.eecs.berkeley.edu/~alanmi/publications/2001/tech01_zdd_.pdf (accessed Dec. 15, 2023)