Formal Verification of Integer Multiplier Circuits

Jonathan Wang
Price College of Engineering
University of Utah
Salt Lake City, Utah
u1306458@utah.edu

Henry Silverman
Price College of Engineering
University of Utah
Salt Lake City, Utah
henry.silverman@utah.edu

Garrett Slack
Price College of Engineering
University of Utah
Salt Lake City, Utah
u1315263@utah.edu

Dmitry Panin
Price College of Engineering
University of Utah
Salt Lake City, Utah
dmitry.panin@utah.edu

***Abstract*—This paper employs advanced mathematical techniques, specifically polynomials, and ideals, to rigorously verify the correctness of an integer multiplier circuit. By leveraging algebraic methods, this approach will provide a deeper understanding of the circuit's behavior and enable a more robust verification process.**

***Index Terms*—polynomials, vanishing polynomials (J_0), ideals (J), multiplier circuit**

I. Introduction

In the field of digital circuit design, integer multiplier circuits stand as fundamental components with critical applications in various computing systems, ranging from embedded devices to high-performance computing architectures. The accurate and efficient operation of these multiplier circuits is crucial for ensuring the reliability and correctness of arithmetic computations. As technology advances and design complexities escalate, the need for rigorous verification methodologies has become increasingly pronounced to guarantee the integrity of digital circuits. Our project tests and verifies these integer multiplier circuits by employing the various methods studied in class on smaller circuits and circuits over fields.

II. Approach

Ideal membership testing is a great method to test and verify any n-bit circuit. We manually designed the 2-bit and 3-bit multiplier and generated the larger circuits utilizing the (.blif writing capabilities of the?) ABC synthesis tool. Next, we converted the blif files to sing files using our parsing python script. The reverse topological term order (RTTO) was then derived from the circuit, and used to represent the minimal Gröbner Basis. In digital circuit verification, there are several approaches to ensure a multiplier circuit implementation performs the same function as its specification. The first option could be creating a miter between the specification polynomials and the implementation, then checking if the Gröbner Basis (GB) is 1. Other approaches will be necessary at larger circuit sizes, such as weak Nullstellensatz, where the GB of the ideals and vanishing polynomials is 1 ($\mathrm{GB}(J+J_0) = 1$). Another form of verification is checking if the reduction of the spec polynomial f_{spec} mod $\mathrm{GB}(J + J_0) = 0$, i.e., the GB reduction of the spec modulo $J + J_0$ is 0.

III. Singular

A. Singular

The Singular file consists of a ring, polynomials, ideals, and other code to define the behavior of a circuit and interpret its output. A ring is defined by properties, including the type of coefficients, number and names of variables, and arithmetic operations. The ordering of the inputs and outputs in the ring R should follow RTTO, consisting of the following order: outputs, internal gate outputs, and primary inputs. A polynomial (poly) is the algebraic representation of the logic in the circuit. This can mean a poly is defined for each logic gate in the circuit or a more complex poly may be used to represent the functions of groups of gates. For example, f_{spec} represents the desired mathematical functionality of the implemented circuit. Ideals are a subset of the polynomial ring that consists of all polynomials satisfying certain conditions. Vanishing polynomials are polynomials over a ring that output 0 for all elements in the ring. Algorithm 1 shows the pseudo-code for an integer arithmetic circuit.

B. Singular results for 2-bit, 4-bit, and 16-bit multipliers

First we tested the 2-bit multiplier for simplicity. If the Gröbner Basis of ideal J is 1, the variety of ideal J is empty or the miter is infeasible, shown in Fig. 1. After introducing

```
Ideal J is:
J[1]=Z_s+A*B
J[2]=A+a_0+(X)*a_1
J[3]=B+b_0+(X)*b_1
J[4]=Z+z_0+(X)*z_1
J[5]=a_0*b_0+s_0
J[6]=a_0*b_1+s_1
J[7]=a_1*b_0+s_2
J[8]=a_1*b_1+s_3
J[9]=s_1+s_2+r_0
J[10]=s_0+s_3+z_0
J[11]=s_3+r_0+z_1
J[12]=Z_s*t+Z*t+1

Groebner basis G of ideal J:
G[1]=1
If G = 1, then variety of ideal J is empty
```

Fig. 1. Gröbner Basis = 1 without any bugs.

the bug(changing an AND gate to an OR gate), G has 18 polynomials, and the ideal J equals to 1, shown in Fig.

Algorithm 1 Example Singular code for arithmetic circuit

// Declare ring
ring R = 0, (outputs, internal outputs, inputs), lp;

// Declare f_{spec} and polys of internal gates
poly f_{spec} = specification equation of the circuit;
//f_4 is an example of a polynomial describing gate behavior
//z is an output, s and e are internal signals
//z = s_0 - e_0 + 2*s_0*e_0
poly f_4 = z_0 - s_0 - e_0 + 2*s_0*e_0;
//Continue to model logic gates over Q

// Declare ideals
ideal J = f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16;
ideal J0 = vanishing polynomials;

//Finding Gröbner Basis of the ideals plus the vanishing polynomials
groebner(J + J_0);

//Membership test using a condition checking statement to explicitly state membership
NF(f_{spec}, J + J_0) = 0;

2. Corresponding to the funcdamental concept of the weak Nullstellensatz:

$$G = 1 \quad \longleftrightarrow \quad Variety(G) = \emptyset$$

Later, we introduced minimal and reduced Gröbner Basis

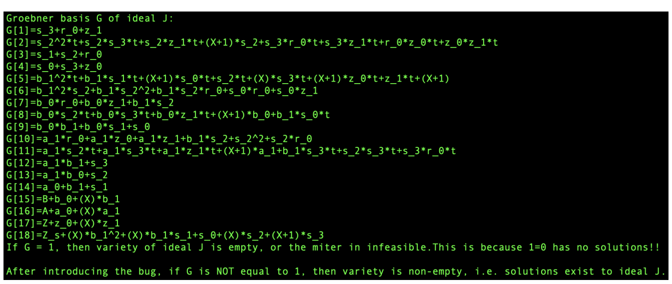

Fig. 2. Gröbner Basis = lots of polynomials = solutions exist.

for the larger 4-bit and 16-bit multiplier circuits. A minimal Gröbner basis is a special case of a Gröbner Basis that has the fewest number of elements possible while generating the same ideal. A reduced Gröbner Basis is a further refinement of the minimal Gröbner basis, which means the leading coefficient of each polynomial is one, and no monomial of another polynomial is divisible by the leading monomial of another polynomial in the basis. We could not present proper descriptions of verified 4-bit and 16-bit circuits due to the inputs and outputs not being in RTTO order, causing bugs that prevented verification.

IV. Algorithms and Techniques

Techniques for ideal membership testing:

1) Setup the verification formulation over the polynomial ring R.
2) Declare a specification polynomial from circuit f_{spec}.
3) Derive the polynomials from the gates of the circuit $\{f_1$, ..., $f_s\}$.
4) Set ideal J = f_1, ...,f_s.
5) Create the ideal of vanishing polynomials for each variable J_0
6) Take the Gröbner Basis: G = GB(J + J_0).
7) The circuit implements $f_{\text{spec}} \iff f_{\text{spec}} \in (J + J_0)$ if and only if f_{spec} is divisible by G

The steps to run Singular, ABC, and the Python script are provided in the README.

V. Software implementations

We implemented a Python function with assistance from ChatGPT to convert blif files to sing files. The code defines a gate mapping dictionary that maps gate types from the blif file to their corresponding names in Singular. It iterates through and searches for lines starting with '.gate' and extracts the gate type and its inputs, parsing and separating them accordingly into the Singular file. Additionally, we added code for error handling because the Singular file was empty when experimenting. Below is the pseudo-code for the Python file.

```
Function parse_blif_file(file_path):
Initialize gates, inputs, and outputs
  Open file_path
    For each line in file:
     If line starts with '.gate':
     Parse gate type, inputs, and output
     Replace 'new_n' with 'n' in input
     Add gate information to gates
     Update outputs and inputs set
     If line starts with '.inputs':
      Update inputs set
     If line starts with '.outputs':
      Update outputs set
 Return gates, inputs-outputs, outputs

Function topological_sort:
    Create a graph from gates
    Initialize order and visited sets
    Define topological sorting function
    For each output in graph:
     Visit(output)
    Return sorted order

Function generate_ring_declaration:
    Return ring declaration string

Function write_singular_file:
    Open output_path file
    Write ring declaration
```

```
    Write polynomial equations
    Generate vanishing polynomials
    Write ideal declarations

Function main():
    Prompt user for BLIF file path
    Prompt user for output path
    Parse BLIF file
    Perform topological sort on gates
    Set ring size (default 2)
    Write .sing file
    Print completion message

Execute main()
```

VI. Concepts Learned

A. Mathematical Ideals

Ideals ensure the accuracy of electronic systems In circuit testing and verification. Ideal membership testing involves using mathematical models to assess whether a circuit aligns with specified ideal characteristics. Engineers compare these models with real-world circuit implementations to identify discrepancies to help detect potential faults in the design. This approach enhances precision and provides a systematic framework for validating complex circuits, contributing to the development of high-quality electronic systems.

B. Gröbner Basis

Gröbner Basis in-circuit testing and verification is a powerful method that involves algebraic techniques to analyze and validate complex electronic systems. Gröbner Bases provide a systematic way to address polynomial equations representing circuit behaviors. Gröbner Bases are a fundamental concept in algebraic geometry and computer algebra systems, and they play a crucial role in solving systems of polynomial equations. The primary idea behind Gröbner Bases is to provide a systematic method for transforming a set of polynomials into a more manageable and structured form.

C. blif to sing conversion

By using a python script to parse our mapped blif files (lib2.genlib), we can convert the list of gates contained in the blif file to a list of corresponding polynomials for our sing file. By splitting the blif file line by line and parsing for key words such as ".gate," "XOR," "NAND," and "INV" to create the different polynomial equations to model logic gates over Q. For example, "XOR" uses the equation z - a - b + 2ab, and "AND" uses the equation z = a * b for the corresponding polynomials. Python makes this very easy with its parsing functionalities.

D. Ideal Membership Testing

Ideal membership testing is a mathematical approach used in circuit verification to determine whether a given circuit satisfies ideal properties or conditions. It involves algebraic structures known as ideals, particularly in the context of polynomial rings. An ideal is a subset of the polynomial ring that consists of all polynomials that satisfy certain conditions. In the context of circuit verification, the ideal encapsulates the set of polynomials representing the desired ideal behavior of the circuit. The core of the process involves determining whether a given polynomial, representing the behavior of the actual circuit, belongs to the ideal. If the polynomial is an element of the ideal, it indicates that the circuit satisfies the ideal properties. Otherwise, it suggests a deviation from the expected behavior.

E. Miters

Miters play a crucial role in circuit testing and verification. A model of the circuit is integrated into the testing process, allowing for comprehensive analysis. The miter compares the expected behavior with the actual circuit response using an xor gate to highlighting any discrepancies based on the model. By using miters, we can identify and address potential issues before moving to further stages of development or production, ensuring that the circuit functions as intended.

VII. Conclusion

We started with the idea of verifying multiplier circuits, ranging from 2-bit to 32-bit. To do so, we wrote our .sing files for 2, 3, and 4-bit multipliers, then began using ABC to generate .blif files for any higher order multipliers than that. These .blif files could not be directly verified due to their size, so a method had to be derived that would allow conversion from .blif to .sing while maintaining functionality. The solution was a Python script written to parse the .blif file and produce a corresponding .sing file. This method took up a lot of time, but in the end, organizing the terms in reverse topological term order was a task we never overcame. It meant that the 16-bit and 32-bit multipliers could be converted to .sing, but these .sing files contained bugs that prevented verification. Throughout our work on the project, we explored everything from basic circuit functionality and circuit traversal to the ins and outs of the different verification methods. We also would consider ourselves much more fluent in .blif and .sing files, as we spent quite a while with them. The experience of taking a designed circuit and proceeding to take it through the full verification process was a great help in wrapping our heads around the work of circuit verification, and what to expect from future verification attempts.

References

[1] D. Ritirc, A. Biere and M. Kauers, "Column-wise verification of multipliers using computer algebra," 2017 Formal Methods in Computer Aided Design (FMCAD), Vienna, Austria, 2017, pp. 23-30, doi: 10.23919/FMCAD.2017.8102237.

[2] T. Pruss, P. Kalla and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction From Combinational Circuits for Verification Over Finite Fields," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 7, pp. 1206-1218, July 2016, doi: 10.1109/TCAD.2015.2501301.

[3] Jerry R. Burch. 1991. Using BDDs to verify multipliers. In Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC '91). Association for Computing Machinery, New York, NY, USA, 408–412. https://doi.org/10.1145/127601.127703