Improvements on Solving Binary MQs on an FPGA

A Major Qualifying Project
Submitted to the Faculty of Worcester Polytechnic Institute
In partial fulfillment of requirements for the Degree of Bachelor of Science in Electrical and Computer Engineering

By
Matthew Lund & Andrew Gray

Date: 2/21/2023
Project Advisor: Dr. Köksal Mus

Table of Contents

Table of Figures

Abstract

This project is a part of a larger project with the goal of implementing algorithms to solve systems of binary quadratic equations using a recursive search on FPGAs. Our goal was to implement an exhaustive search method for a binary quadratic system of equations as a proof of concept, as well as expanding upon a modified GRASP (Generic Search Algorithm for the Satisfiability Problem) algorithm created in Python to improve the speed and reduce complexity of the decision making. Look up tables (LUTs) created by the group covered cases for two equations with one, two, or three differences between them. The tables and modules for the project are written in Verilog and SystemVerilog, with one additional module created in VHDL. Some work was also completed in Python. For future use, the exhaustive solver can easily be upgraded to scale the number of terms and solution size. The CDCL (Conflict Driven Clause Learning) is still a work in progress at the time of this paper. Although Boolean Satisfiability problem is a never-ending research problem, our group did make improvements in the solving that could be expanded upon by further research and improvements.

Background

The Solution of Boolean multivariate quadratic (MQ) systems, aka Boolean Satisfiability Problem (SAT) is a popular problem in Cryptography. The SAT (satisfiability) problem involves determining whether a given logical expression, represented as a system of boolean equations, can be satisfied by finding a combination of truth values for the variables that makes the entire expression true at the same time. The MQ problem takes this premise and asks if there is a solution to a system of binary, quadratic systems, similar to the one shown in Figure 1.

$$X1 + X2 + X3 + X1X2 = 1$$
$$X1 + X2 + X1X2 + X2X3 = 1$$
$$X2 + X1X2 + X2X3 + X1X3 = 0$$

Figure 1: An Example System of Binary Quadratic Equations of 3 variables

The MQ problem itself is stated to be used in regard to post-quantum cryptography as “the building block of the Multivariate public key crypto systems (MPKCS)” (Bellini, et al. 2022) such as Rainbow, LUOV. These public keys have specific parameters that help to determine the hardness of the MQ problem given as well as the specific time and space complexities. (Bellini, et al. 2022):

1. Size of Finite Field (usually 2)
2. Number of variables
3. Number of polynomials

Knowing this, there have been numerous attempts at solving the MQ as well as the SAT problem such as the "Crossbred" algorithm, different variations of the AmoebaSAT algorithm, as well as looking back at algorithms created by previous MQP teams such as the one made by Frank Kennedy to solve linear systems via exhaustive searching as the first step towards solving quadratic systems. The goal of this project is to see what can be enhanced from Frank's algorithm as well as laying the groundwork for the next steps in the implementation on an FPGA.

Implementations of MQ/SAT Problems

"Crossbred" Algorithm

The "Crossbred" algorithm developed by French cryptographers Antoine Joux and Vanessa Vitse, focused on solving systems of quadratic binary polynomials using Macaulay matrices, much like the FXL/BooleanSolve algorithm it is based on (Joux 2018). A Macaulay matrix is able to show the coefficient weight of each linear and quadratic term.

$$\begin{array}{c} \begin{array}{cccccccccccc} X_1X_2 & X_1X_3 & X_1X_4 & X_1 & X_2X_3 & X_2X_4 & X_2 & X_3X_4 & X_3 & X_4 & 1 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 2: An Example Degree 2 Macaulay Matrix (Joux 2018)

The main problem with a lot of solving algorithms that involve the use of this matrix is linear algebra that is "performed 2^{n-k} times," where *n* is the number of unknown terms and *k* is the number of known variables that can be taken out of the system (Joux 2018). Joux and Vitse decided to mitigate this issue by performing this portion after the elimination of the known variables.

The basic premise would be to first construct the matrix in alphabetical order, like the matrix in Figure 2, and compute the last rows of the constructed matrix's reduced row echelon form. By being able to just compute the last rows of the system, excluding variables with the X1 term from the reduced row echelon form as shown in Figure 3, we can solve for the rest via exhaustive search methods, and then checking the solutions with the equations that contain X1 (Joux 2018).

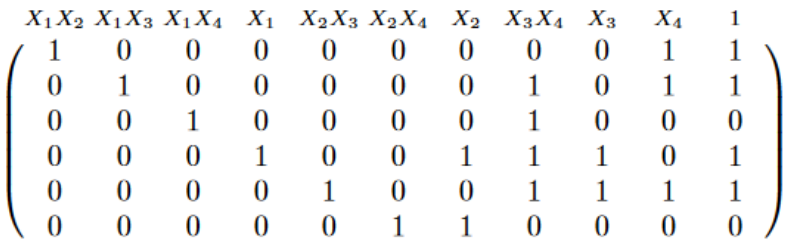


Figure 3: Reduced Row Echelon of Figure 2's Matrix (Joux 2018)

However, it is not necessary to eliminate all of the *k* variables from the system. Joux and Vitse explain that a more refined version of the algorithm involves ordering the columns of the matrix in **graded reverse lexicographic** (GRevLex) order, with all quadratic terms first before the linear terms, creating a row echelon form of the matrix shown in Figure 4. From the last 3 rows, we see that all the equations have X1, X2, and X3 in degree 1. This allows us to assign X4 to whatever we want and solve for the other variables, theoretically eliminating them from the search.

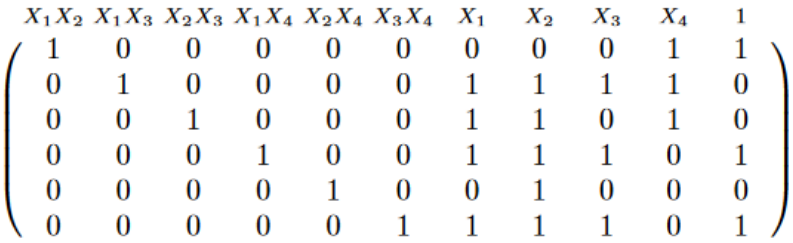


Figure 4: GRevLex Row Echelon Form of Matrix (Joux 2018)

In terms of implementation, Joux and Vitse tackled the Fukuoka Type I MQ challenges issued in 2015 to assess the hardness of solving the systems of equations (Joux 2018). They used a network of Opteron and Xeon processors and were able to solve challenges using up to 74 differing variables taking an estimated maximum of 300,000 hours to solve (Joux 2018).

AmoebaSAT

The AmoebaSAT algorithm is a cooperation between software and the hardware of an FPGA based on amoeba cell biology. Primarily used for Internet of Things (IoT) oriented applications, requiring the processing of many variables, the algorithm is based on how an amoeba can grow and move from light signals called "Bounceback signals" (Ngyuen, et al. 2020). These signals are sets of rules that dictate that each variable cannot be both 1 and 0 at the same time, all literals cannot be 0, and rules to resolve situations where a variable cannot be either 0 or 1. These decisions end up consuming a lot of memory to operate.

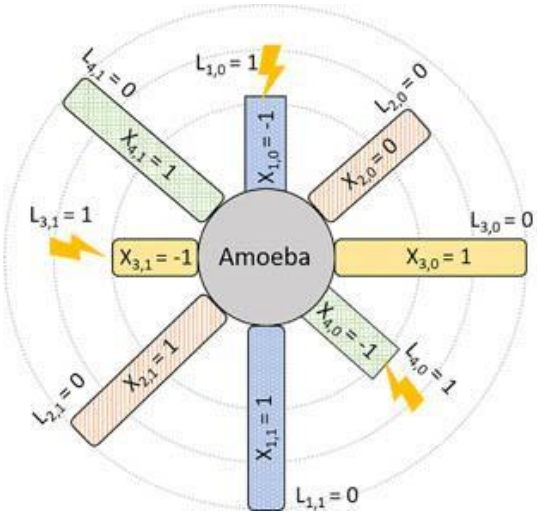


Figure 5: AmoebaSAT model for 4 Variable SAT Case (Nguyen, et al. 2020)

An iteration of this algorithm known as AmoebaSATslim (ASATslim) is able to reduce the amount of memory it uses by omitting certain rules from the bounce back signals and instead, implementing them as temporary signals on a branch-by-branch basis. Although it will need nearly the same number of iterations as the original ASAT algorithm, due to memory issues being mitigated to a degree, ASATslim can handle more variables (Ngyuen, et al. 2020). An even more "evolved" version of this algorithm known as ASATone further reduces the computational resources needed by representing variables as single branches in a more instance-based method.

This version of the ASAT algorithm was able to be implemented *h* copies of the uf50-100.cnf 3-SAT instance, a set of variables that only have one solution with 50*h* variables and 218*h* clauses on a Zynq Ultrascale+ FPGA and compared with a software implementation using a Ryzen 3960X 24 Core CPU (Nguyen 2020). When compared to the software implementation, the FPGA was able to be anywhere between 3 and 15 times faster while sipping power using under ten watts (Nguyen 2020). By utilizing the ability to execute multiple instances through parallelization on FPGAs, they are best suited for these multiple variable problems.

Frank Kennedy's Implementation

Frank Kennedy's implementation has been built off of previous work from WPI students Liam Stearns, Carlton Mugo, and James McAleese, from the past two years with the main idea of his recursive algorithm was to split the system into smaller groups decreasing the number of solutions to "2^{n-s}, where *s* is equivalent to the number of groups" (Kennedy, 2023). Kennedy's goal was to create a program to solve a set of quadratic equations via partial solutions to the

linear terms of the equations. Kennedy started his improvements by observing when different equations would be nearly identical besides one variable, while also observing the Right Hand Side (RHS) variable, also known as the solution of the equation, along with a Rest variable that is defined as "what the solution adds to using bitwise addition with the exception of the variable in question" (Kennedy, 2023) to see if there is a possible solution and what the said solution would be. Figure 6 below shows Kennedy's table for two equations with a one variable difference. These rules are able to assist in Kennedy's recursive algorithm, eliminating sets that need to be figured out in the rest of the algorithm.

Rest	Coefficients (E1 & E2)	Right Hand Side (E1 & E2)	Solution	Possible Solution
0	00	00	Yes	0
0	01	00	No	N/A
0	10	00	No	N/A
0	11	00	No	N/A
0	00	01	No	N/A
0	01	01	Yes	0
0	10	01	No	N/A
0	11	01	No	N/A
0	00	10	No	N/A
0	01	10	No	N/A
0	10	10	Yes	1
0	11	10	No	N/A
0	00	11	No	N/A
0	01	11	No	N/A
0	10	11	No	N/A
0	11	11	Yes	1
1	00	00	No	N/A
1	00	01	Yes	1
1	00	10	Yes	0
1	00	11	Yes	0
1	01	00	Yes	1
1	01	01	No	N/A
1	01	10	Yes	0
1	01	11	Yes	0
1	10	00	Yes	1
1	10	01	Yes	1
1	10	10	No	N/A
1	10	11	Yes	0
1	11	00	Yes	1
1	11	01	Yes	1
1	11	10	Yes	0
1	11	11	No	N/A

Figure 6: Rules for One Different Variable (Kennedy, 2023)

In terms of Kennedy's recursive searching algorithm for the sets of equations, he intended on splitting the system matrix into smaller pieces, treating the linear portion as its own section. Kennedy only assigns weights to each linear portion of the equations, organizing the equations from lowest weight to highest weight. This organized form of the matrix is then further reorganized "starting with the first equation in the matrix, if a 1 is found, the entire column swaps places with the first 0, " removing that column from other reorganizations, and repeated for the following equations to form an upper right-hand triangle, creating a priority on how to search the variables (Kennedy 2023). This first variable can be set to 0 at first and evaluate the rest of the terms to see if it was a viable solution in the first place. If it works, then the solution is considered SAT and the work is done. If not, then the first term is set to 1 and the process starts all over again. If there continues to not be a solution found, then the system can be considered UNSAT, meaning that there are no solutions.

Although Kennedy was unable to finish or implement this searching algorithm, he states that the algorithm has a theorized number of solutions of $2^{n/2}$, which is a significant improvement when compared to the 2^n complexity of exhaustive searching. From his report, Kennedy also states that he "utilized many hard coded values in order to establish the equations and matrices used in the setup portion of the code," and that the process could be made more efficient if there was less hard coding and a possible reading from a memory file occurred instead. Kennedy also suggested that creating a lookup table of solutions for cases in which two variables differ would also be beneficial in checking solutions for complexity reduction. The only issue he saw with this method would be that there would be a significant jump in the memory usage and the board he was using did not have enough non-volatile flash to store this data and that a new board should be looked into.

GRASP

During the project's duration, part of our group completed a modified implementation of Generic Search Algorithm for the Satisfiability problem (GRASP) in Python. GRASP is made with a decision tree that bases later decision off earlier decisions. The decision tree helps reduce the number of solutions worked through, as large portions of the tree can be eliminated at once. The decision tree also allows for easier backtracking. This backtracking is a result of a conflict, where several variables have been chosen but the solution is no longer viable. This is where the term Conflict Driven Clause Learning (CDCL) comes from, as GRASP is the earliest example of a CDCL. GRASP was finished in 1996 and there have been several improvements in both time complexity and ease of design. GRASP sets an important basis for future CDCLs and improvements in our own project itself.

Figure 7 below shows an example binary decision tree. The beginning of the tree chooses the most involved variable (variable that occurs the most) and then continues from there. The decisions are stored in the database, as well as the level that they are decided at. The tree will eventually reach a point where a solution has been found and the system is satisfiable, or all viable options have been tested and the system is unsatisfiable. Although the algorithm improves a lot when compared to an exhaustive solver, there is still some randomness (Silva & Sakallah, 1996).

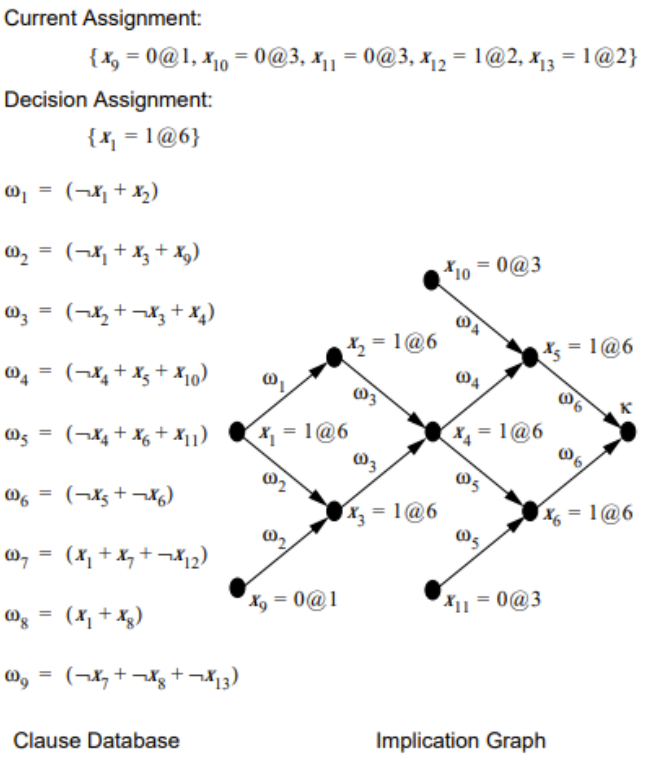


Figure 7: GRASP example implication graph with decisions and levels (Silva & Sakallah, 1996)

Improving Solving Lookup Table to Account for Quadratics and Two Term Differences

Building off of Kennedy's work on comparing two equations with one differing linear variable, we decided to create a simulation written in SystemVerilog in the Xilinx Vivado ISE to generate all of the combinations of coefficients for the equations of the format shown in Figure 7, excluding the Rest and RHS. The aim was to be able to create a lookup table of common solutions for when equations may appear very similar besides two coefficients that involve linear or quadratic terms, hence the exclusion of Rest and RHS. The goal is that this lookup table would be able to be referenced in recursive searching algorithms when breaking down equations

with differences in three or more terms, decreasing the complexity of solving these kinds of systems.

X1	X2	X3	X4	X1X2	X2X3	X3X4	REST	RHS
0	0	1	0	1	0	0	0	1

Figure 8: Equation Format for Weight of 2 Solving with Example Coefficients

Using this specific set of terms, I am able to cover all of these five different scenarios:

1. Two different linear terms (X1, X3),
2. One linear and one quadratic **without** a shared term (X1, X2X3),
3. One linear and one quadratic **with** a shared term (X1, X1X2),
4. Two quadratics **without** a shared term (X1X2, X3X4),
5. Two quadratics **with** a shared term (X1X2, X2X3).

This equation format also uses the assumption that there are more terms that we are not looking at that are the same between each equation that both reduce down to a simple "Rest" term of either 1 or 0, similar to Frank's research. The RHS can be either the same or different for each equation and thus is not accounted for in the binary weight as well as the rest.

In order for me to figure out what every combination of weight of two was for the seven coefficients, I started with a simple bit-counting simulation in Verilog that would use a seven-bit counter as an input and output a 1-bit flag that says if the input value has a binary weight of 2. From there, I was able to construct the table shown in Figure 8 that shows the combination of two sets of coefficients that follow the following rules:

1. The coefficients can NOT be the same value (blacked out in Figure 8)
2. The combination of coefficients can not be covered twice
   (i.e (EQ1 = 0000011, EQ2 = 0000101) and (EQ1 = 0000101, EQ2 = 0000011)

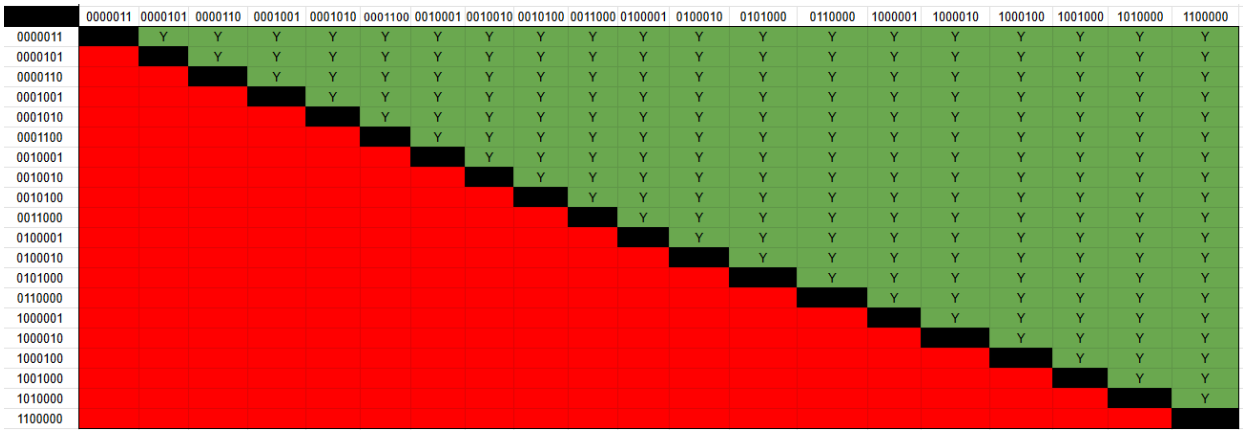


Figure 9: Valid Combinations of Coefficients

The table demonstrates that there are 190 total valid combinations of coefficients that can be iterated through, alongside the different variations of RHS (00, 01, 10, and 11 for {EQ1,EQ2}, respectively) and Rest (0 or 1), resulting in a total of 1,520 systems in the format portrayed in Figure 9.

```
{Coefficient_EQ1,REST, RHS1}
{Coefficient_EQ2,REST, RHS2}
```

Figure 10: Format for Two Equations with Weight Two in Code

Knowing all of this, I was able to create a look-up table to iterate two counters through all of the valid coefficients from Figure 8 and form a simulation that would also iterate through the combinations of the RHS for the equations via a two-bit value (RHS[1] = RHS2 and RHS[0] = RHS1), as well as the rest value, outputting all data on the number of total systems created, what the equations are, all valid combinations of X1, X2, X3, and X4 iterated via a 4-bit counter mapped to each linear term (the quadratic terms are just the linear terms multiplied by each other through the use of bitwise AND), as well as the number of solutions for each system. The systems are solved via conducting a bitwise-XOR on both equations generated, then performing a bitwise-AND on the resulting coefficients and the possible solution made by the 4-bit counter. From there, the resulting string is XOR'd with each other and compared to the final RHS value. If the values are the same, then it is labeled a valid solution and displayed in the console of the simulation. The code for finding the weight of two, look-up table, solving module, simulation, and the results can be found in Appendices B, C, D, E, and F respectively. At the end of my time on this project, I was also able to create the look-up table for cases involving a weight of three and modified the weight-finding code to parametrize the binary weight. The code for these will be in Appendix I and J respectively.

Choosing a New Board

To perform such operations for row and column operations that act similar to Gaussian elimination, as well as the storage of multiple arrays, we will need to utilize more memory on the FPGA. With taking Frank Kennedy's input and recommendations on getting a better board than the Digilent Basys-3 board with the Artix-7 A35 FPGA on-board (Kennedy, 2023), we decided to look for a board that fulfilled these requirements:

1. Has more than 32 megabits of non-volatile flash
2. At least 5 times as many logic cells as Basys-3 (A35 has 33,280)
3. At least 5 times as much Block RAM (BRAM) (5 * 1800 kilobits on Basys-3)

When accounting for these requirements, a board with the Artix-7 A200 FPGA, more specifically, Digilent's Nexys Video board would be the best for this application. The A200

Artix-7 FPGA has **215,360 logic cells**, covering the second requirement, and has **13 megabits** of BRAM, fulfilling the third requirement. The board itself has **32 mega*bytes*** of non-volatile flash on board when compared to the 32 mega*bits* on the Basys-3, fulfilling the first requirement.

COMPARE / Reset	XC7A35T	XC7A200T
Logic Cells	33,280	215,360
DSP Slices	90	740
Memory	1,800	13,140
GTP 6.6Gb/s Transceivers	4	16
I/O Pins	250	500

Figure 11: Comparison of A35 and A200 FPGA (Xilinx Website)

Using this board would allow for implementation of systems of equations with a larger number of equations and a numerous number of differing terms, executing programs in a reasonable amount of time.

Aside from this board chosen, the team wanted to explore implementing code using Python onto a Xilinx Zynq 7000 System-on-Chip (SoC) board, more specifically the PYNQ-Z2 board. PYNQ itself is an open-source project developed by Advanced Micro Devices (AMD) that takes Python and any associated libraries and translates it to hardware description language during run-time or for parallelization of the code. At the time of writing, the board is still a work in progress. The actual board did not work directly with python as expected. The modules are being converted to Verilog to be able to run properly on the board. Not everything of the module needs to be converted between languages, but the board was not as “plug and play” as expected. Some of the Python programs work as expected, but much of the top level module has to be converted into Verilog and SystemVerilog. This is a different effort from the full CDCL in SystemVerilog. The two boards can be used to compare a Python implementation vs a full HDL implementation and determine the performance differences.

Exhaustive Searching

When looking at the previous exhaustive search code from Frank Kennedy, the team and I saw that there were many areas that could be improved. The first one that was shown was the creation of classes, column-swapping, and weight assignments, which would not impact the time complexity of 2^n, where *n* is the number of linear terms. The classes also contained more information than what was needed for exhaustive search to be performed.

After consideration, we were able to create a new packed structure 16 bits long for the equation's coefficients, with *n* for the implementation being five linear terms (5 linear terms + 10 quadratic terms + RHS = 16). This structure is used to create a packed array of 16 hard-coded equations generated with the use of a random number generator online to perform the exhaustive search. Although the initial idea was to use a clocked 16-bit linear feedback shift register (LFSR) to create the array, there ended up being some cross-clock domain synchronization issues making some equations have all values of X (don't care in SystemVerilog) and concerns of true randomness not being possible that resorted to the use of the hard-coded equations discussed. While this hard-coded solution is not the final idea, the team intends to move this implementation to read off of an SD card loaded onto the Nexys board, which will take a longer amount of time to determine a proper way of doing so in the future.

From there, I XOR all of the equations together, similar to my simulation in the improving linear solving section. This is to determine what terms are needed to be accounted for to solve the system. In regard to the terms, I use five of the switches on the Nexys board to assign values of 1 or 0 to the linear terms X1 to X5 and assign the quadratic values via a bitwise AND between these linear terms. All of these terms are concatenated into a 15-bit long string as shown in the code snippet in Figure 11 to easily perform a bitwise AND between them and the term coefficients (bits 16 to 1) of the XORed equations.

```
wire X1, X2, X3 ,X4 ,X5 ,X1X2, X1X3, X1X4, X1X5, X2X3, X2X4, X2X5, X3X4, X3X5, X4X5;
wire [4:0] Terms = sw[4:0];
assign X1 = Terms[0];    //Assign Linear Terms
assign X2 = Terms[1];
assign X3 = Terms[2];
assign X4 = Terms[3];
assign X5 = Terms[4];

//Assigning Quad Terms
assign X1X2 = X1 & X2;
assign X1X3 = X1 & X3;
assign X1X4 = X1 & X4;
assign X1X5 = X1 & X5;
assign X2X3 = X2 & X3;
assign X2X4 = X2 & X4;
assign X2X5 = X2 & X5;
assign X3X4 = X3 & X4;
assign X3X5 = X3 & X5;
assign X4X5 = X4 & X5;

wire [14:0] term_string = {X1, X2, X3, X4 ,X5, X1X2, X1X3, X1X4, X1X5, X2X3, X2X4, X2X5, X3X4, X3X5, X4X5}; //easier for computation
```

Figure 12: Term Assignment Code Snippet

Once the AND operation has been completed and stored in a 15-bit wire, all of the bits in it are XORed together for the addition to compare to the RHS value created from XORing all of the initial equation coefficients' RHS to determine if the combination of switches is solves the system. If it is a solution, the set of LEDs corresponding to the switches will turn on, indicating that it is a solution; if not, then an LED not controlled by the LED will be turned on, indicating that it does not provide a solution. As my solution was only concerned with combinational logic, a clock was not required for my implementation to operate. Of course, with wanting to read from an SD card in the future, there will arise the need to use a clock and ensure that the SD card is fully read from to ensure the system of equations is properly solved.

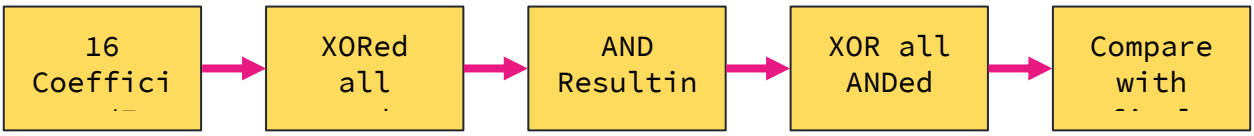


Figure 13: Exhaustive Searching Flow Diagram

When the solution flag within the code is 1, the team also wants a string to be outputted saying that a solution has been found at the value of the terms. This is currently being worked on and utilizes the UART protocol to operate while encoded our string into ASCII text for communication purposes. At the time of this report, we are able to print out the equations into a terminal and some of the solutions.

Exhaustive Searching Results

To first test to see if the exhaustive searching works, I simulated my searching module, updating the terms register of the simulation every 10 nanoseconds, and comparing it to what the good_out wire, the representation of the LEDs, gave. As shown in Figure 13, any time that the wire had a hexadecimal value of 20 (binary value of 100000), that would mean that the combination of terms was not a solution to the system. Conversely, when the wire equals the terms register, that indicates that the combination is a valid solution. Figure 14 shows that my implementation uses little-to-no resources of the board, meaning that this implementation can be scaled to include many more linear terms, thus more quadratic terms as well, up to however many switches that you would want to control the value of the linear terms.

To prove that the simulation is correct, I also generated the bitstream and programmed the Nexys board to run the exhaustive search code. Figure 15 shows that a hex value of 0a (X2 and X4 = 1) does not provide a solution to the system, while a value of 12 (X2 and X5 = 1) is a solution and turns on the LEDs shown in Figure 16.

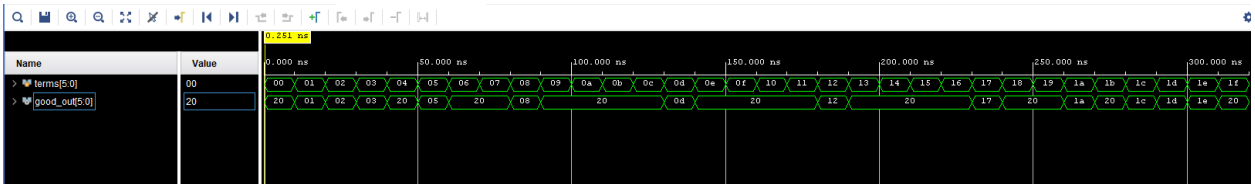


Figure 14: Exhaustive Search Vivado Simulation Results

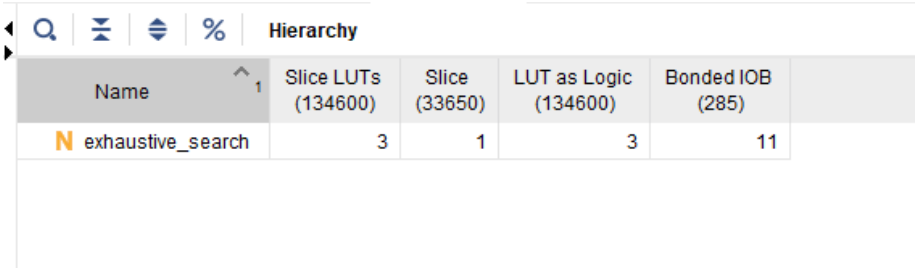


Figure 15: Exhaustive Search Resource Utilization

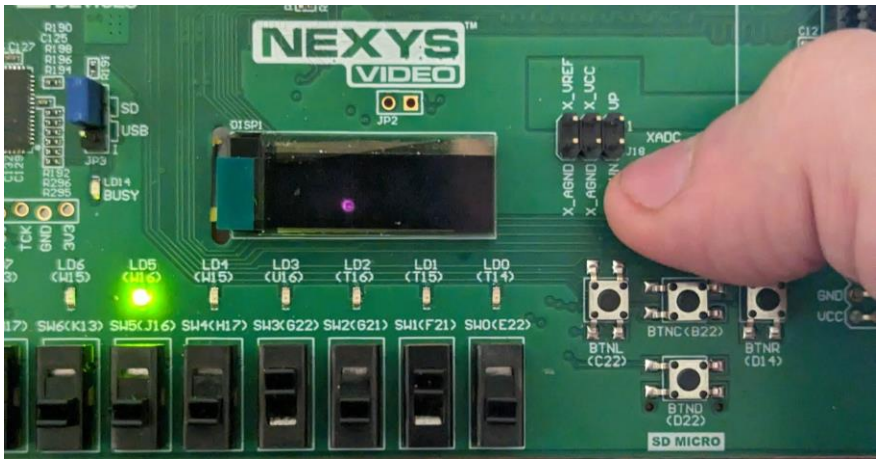


Figure 16: X2 and X4 = 1 on Board

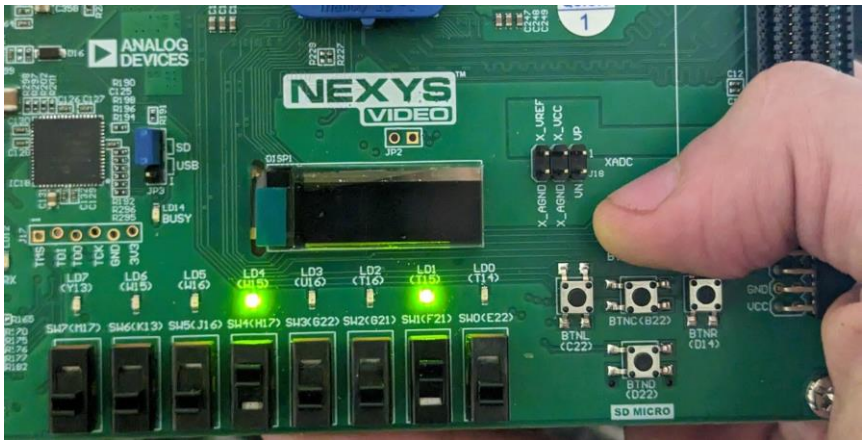


Figure 17: X2 and X5 = 1 on Board

Exhaustive Searching Expanded

In order to truly meet the challenge of solving larger and more complex systems, the exhaustive solver was expanded to eight variables (as a proof of concept), which creates an individual equation size of 36 bits. The solution length is the same length as the number of variables, which in this case is 8 terms long. Eight was chosen just to show expandability, the solver is adjustable for more. To meet the need to show larger solutions, UART printing was used. The starting code provided by Digilent in VHDL served as a good starting point for how the UART printing worked (Digilent, 2016**)**. The code was modified to accept parameters based on the equation length and size, as shown in figure 18. The parameters allow the VHDL module to be called from the Verilog top level module. There was a focus on parameterization so that the printing could be easily changed. The length of the terms, number of equations, and number of variables are all modifiable from the top level of the module, without changing any VHDL code.

```
--counter.
use IEEE.std_logic_unsigned.all;

entity UART_CALL is
generic (LEN: integer :=10;
NUM_EQs: integer:=6; TERMS: integer:=4; LEN_TERMS:integer:=9);
    Port (
           TERMS_V                  : in   STD_LOGIC_VECTOR (LEN_TERMS downto 0);
           BTN                    : in   STD_LOGIC_VECTOR (4 downto 0);
           EQ                     : in STD_LOGIC_VECTOR ((NUM_EQs*LEN)-1 downto 0);
           CLK                    : in   STD_LOGIC;
           UART_TXD               : out  STD_LOGIC
          );
end UART_CALL;
```

Figure 18: UART printing module in VHDL

Figure 19 shows an example output with 5 variables. The equations are printed first, then the solutions are shown. Because of the limitations of Verilog, the solutions wire length must be determined at compile time. This means that the solutions wire length must be guessed. Figure 19 shows what happens because of this. When the solutions wire is longer than the number of solutions, the solutions are just repeated. If the system has no solutions, the entire solutions wire is all 0's. The don't care (X) is added in to be written over, due to VHDL restrictions on loops.

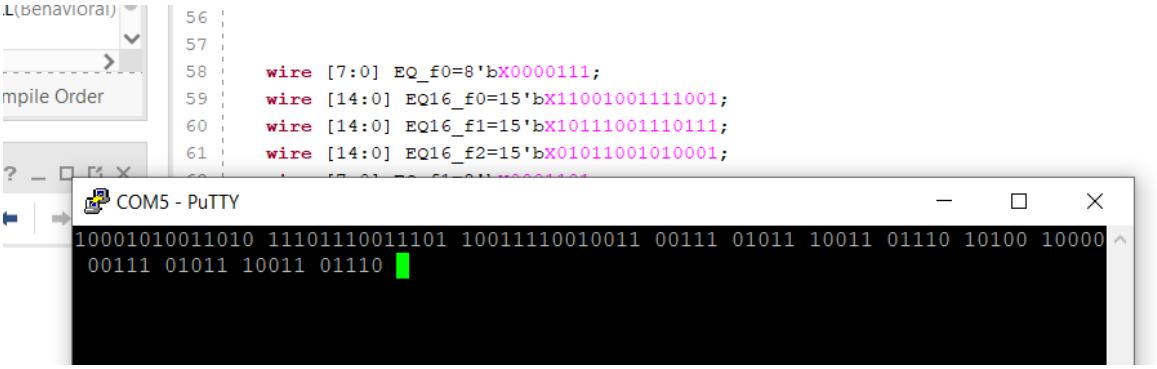


Figure 19: Example printing with input equations shown

The exhaustive solver is similar to the previous exhaustive solver, where a terms wire is created and iterates through all possible options. If the solution works, it is appended to the solutions and later sent to the UART printing VHDL module. The specific benefit of adding the UART printing and slightly modifying the exhaustive solver is that the project is now expandable to a much larger set of equations. The modules have parameters to be easily changed.

The problem with exhaustive searching is the high complexity as the solutions are expanded. The actual running of the module is still less than 2 seconds, but the process to create (synthesis, implementation, and bitstream generation) took about 15 minutes with 8 variables and 10 equations. Although the system can be expanded, it will take exponentially longer to build. The team needed another solution.

GRASP Python Implementation

Due to the exponential complexity of the exhaustive solver, another implementation was created. The group initially split into two teams, one working on a CDCL implementation in python and one group working on the FPGA exhaustive search. After the CDCL in python finished, the group convened to work together on converting the files to SystemVerilog. This includes, the look up tables, operations on the look up tables, and other important CDCL functions. This presented a unique set of challenges, as the original code contained many functionalities that do not exist in SystemVerilog, such as dynamic arrays (appending to an array of unknown length). As of the end of C term, the modules are mostly completed but the code itself has not been fully run on a board yet, just a few individual modules. Figure 20 and 21 show two of the same modules, one in Python and one in SystemVerilog. The module takes in an NxM matrix and returns an NxN matrix with the differences between equations. For example, xor_transpose[2][4] will show the number of differences between equations 2 and 4. This shows the difficulty in converting modules, as many python functionalities are not available in SystemVerilog. For example, the inputs in SystemVerilog must be more explicitly defined, including the size of arrays.

```python
def xor_transpose(matrix: Matrix, window):
    """
        transpose matrix and xor

        @param original: 2D Matrix/Array

        :return: 2D Array of xor between original and transposed array
    """
    # transposed = np.transpose(original)
    # debug_print(transposed)

    # M1 = np.array(original)
    # M2 = np.array(transposed)
    result = np.empty((len(matrix.equations), len(matrix.equations)), dtype = np.int64)

    for eq1 in range(len(matrix.equations)):
        for eq2 in range(eq1 + 1, len(matrix.equations)):
            #XOR both
            xDiff = 0
            for i in range(1, window):
                xDiff = xDiff + ((matrix.access(eq1, i) + matrix.access(eq2, i)) % 2)
            result[eq1][eq2] = xDiff

    return result
```

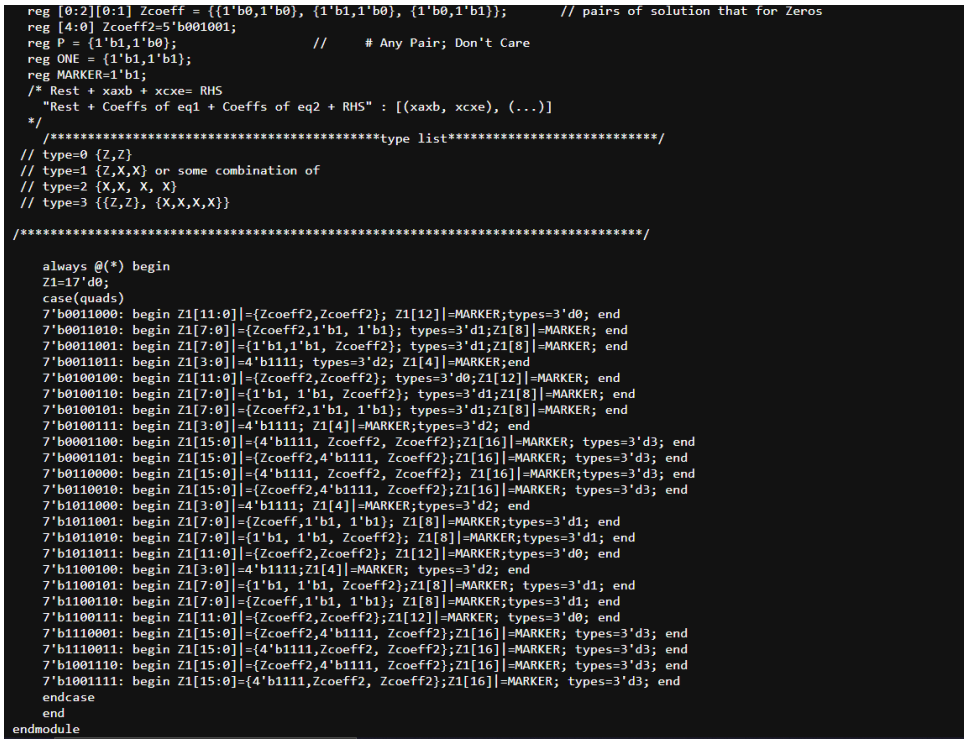
Figure 20: XOR transpose in python

```
module Xor_transpose#(parameter integer EQ_LEN=34, parameter integer NUM_EQs=5, parameter integer window=
input reg [EQ_LEN:0] matrixs [0:(NUM_EQs-1)],
output reg [NUM_EQs:0][NUM_EQs:0] result [4:0]
    );
    /*
    matrix for number of differences, array of integersfor eqch equation
 Input:2d matrix full of terms
 parameters: number of equations and length of equations to build the 2d matrix
 window: number of terms currently active
 output: 2d matrix


    */
    reg [7:0] xDiff;

integer i,x, t;
  initial begin

      for(i=0; i<NUM_EQs; i=i+1) begin
        for(x=0;x<NUM_EQs;x=x+1) begin
         xDiff=7'd0;
        for(t=0; t<window; t=t+1) begin
        if(i==x) begin
         result[i][x]=7'd0;
        end //if
            xDiff=xDiff+(matrixs[i][t]^matrixs[x][t]);

        end //t
        result[i][x]=xDiff;
        end//x

      end//i
  end
endmodule
```

Figure 21: XOR transpose in SystemVerilog

Another major accomplishment of the group was the conversion of additional look up tables from python to SystemVerilog. These look up tables cover cases for one, two, and three differences between equations. The tables can be used recursively and are an expansion of the previous section mentioning differences between two equations. They cover different types of

cases such as differences being quadratic dependent (xaxb + xaxc, where one of the differences is a shared term and the other is not). There are a number of important tables that speed up the calculation complexity. Figure 22 shows an example of one table below.



```
  reg [0:2][0:1] Zcoeff = {{1'b0,1'b0}, {1'b1,1'b0}, {1'b0,1'b1}};       // pairs of solution that for Zeros
  reg [4:0] Zcoeff2=5'b001001;
  reg P = {1'b1,1'b0};                  //     # Any Pair; Don't Care
  reg ONE = {1'b1,1'b1};
  reg MARKER=1'b1;
  /* Rest + xaxb + xcxe= RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xaxb, xcxe), (...)]
  */
    /*****************************************type list*************************/
 // type=0 {Z,Z}
 // type=1 {Z,X,X} or some combination of
 // type=2 {X,X, X, X}
 // type=3 {{Z,Z}, {X,X,X,X}}

/*************************************************************************/

    always @(*) begin
    Z1=17'd0;
    case(quads)
    7'b0011000: begin Z1[11:0]|={Zcoeff2,Zcoeff2}; Z1[12]|=MARKER;types=3'd0; end
    7'b0011010: begin Z1[7:0]|={Zcoeff2,1'b1, 1'b1}; types=3'd1;Z1[8]|=MARKER; end
    7'b0011001: begin Z1[7:0]|={1'b1,1'b1, Zcoeff2}; types=3'd1;Z1[8]|=MARKER; end
    7'b0011011: begin Z1[3:0]|=4'b1111; types=3'd2; Z1[4]|=MARKER;end
    7'b0100100: begin Z1[11:0]|={Zcoeff2,Zcoeff2}; types=3'd0;Z1[12]|=MARKER; end
    7'b0100110: begin Z1[7:0]|={1'b1, 1'b1, Zcoeff2}; types=3'd1;Z1[8]|=MARKER; end
    7'b0100101: begin Z1[7:0]|={Zcoeff2,1'b1, 1'b1}; types=3'd1;Z1[8]|=MARKER; end
    7'b0100111: begin Z1[3:0]|=4'b1111; Z1[4]|=MARKER;types=3'd2; end
    7'b0001100: begin Z1[15:0]|={4'b1111, Zcoeff2, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b0001101: begin Z1[15:0]|={Zcoeff2,4'b1111, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b0110000: begin Z1[15:0]|={4'b1111, Zcoeff2, Zcoeff2}; Z1[16]|=MARKER;types=3'd3; end
    7'b0110010: begin Z1[15:0]|={Zcoeff2,4'b1111, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b1011000: begin Z1[3:0]|=4'b1111; Z1[4]|=MARKER;types=3'd2; end
    7'b1011001: begin Z1[7:0]|={Zcoeff,1'b1, 1'b1}; Z1[8]|=MARKER;types=3'd1; end
    7'b1011010: begin Z1[7:0]|={1'b1, 1'b1, Zcoeff2}; Z1[8]|=MARKER;types=3'd1; end
    7'b1011011: begin Z1[11:0]|={Zcoeff2,Zcoeff2}; Z1[12]|=MARKER;types=3'd0; end
    7'b1100100: begin Z1[3:0]|=4'b1111;Z1[4]|=MARKER; types=3'd2; end
    7'b1100101: begin Z1[7:0]|={1'b1, 1'b1, Zcoeff2};Z1[8]|=MARKER; types=3'd1; end
    7'b1100110: begin Z1[7:0]|={Zcoeff,1'b1, 1'b1}; Z1[8]|=MARKER;types=3'd1; end
    7'b1100111: begin Z1[11:0]|={Zcoeff2,Zcoeff2};Z1[12]|=MARKER; types=3'd0; end
    7'b1110001: begin Z1[15:0]|={Zcoeff2,4'b1111, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b1110011: begin Z1[15:0]|={4'b1111,Zcoeff2, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b1001110: begin Z1[15:0]|={Zcoeff2,4'b1111, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    7'b1001111: begin Z1[15:0]={4'b1111,Zcoeff2, Zcoeff2};Z1[16]|=MARKER; types=3'd3; end
    endcase
    end
endmodule
```

Figure 22: Look Up Table for 2 quadratic independent differences

Conclusion

During our time on the project, the overall goal was to construct improvements to solving systems of similar equations besides a combination of linear and/or quadratic terms, implementing the quadratic exhaustive search, and upgrading the hardware used for implementation. We were able to implement an exhaustive solver that used the peripherals of the board to change equations, and a more advanced exhaustive solver which can be scaled for larger equations. In addition, we were able to help advance our group in their implementation of a CDCL FPGA solver by completing a significant amount of the modules and testing the individual modules. Although the project remains unfinished, that is the nature of the research. We tried to implement the best way to push the solutions to be faster and faster, by both improving hardware and improving the approach to the problem.

Acknowledgements

Firstly we would like to thank my advisor Professor Köksal Mus and the rest of our team: Samuel David, Coco Mao, Patrick Hunter, and Joshua Eben for their extensive support and assistance in completing this much of the project. Professor Mus's experience in the FPGA field was extremely beneficial whenever the group needed it or improvements could be made to the existing code. We would also like to thank the Electrical and Computer Engineering Department at WPI for providing us with the opportunity to work on many hands-on projects throughout our undergraduate experience. Additionally, we would like to thank Frank Kennedy for providing a good starting point for the project and helping us select our board parameters. Finally, we would like to thank the sources cited and numerous forum posts on how to fix our niche issues when creating our program.

Appendix

Appendix A: Works Cited

- A. H. N. Nguyen, M. Aono, & Y. Hara-Azumi. (2020). FPGA-Based Hardware/Software Co-Design of a Bio-Inspired SAT Solver. *IEEE Access, 8,* 49053–49065. https://doi.org/10.1109/ACCESS.2020.2980008
- Bellini, E., Makarim, R. H., Sanna, C., & Verbel, J. (2022). An Estimator for the Hardness of the MQ Problem. In L. Batina & J. Daemen (Eds.), *Progress in Cryptology—AFRICACRYPT 2022* (pp. 323–347). Springer Nature Switzerland.
- Digilent (2016). *Nexys Video gpio*. Nexys Video GPIO - Digilent Reference. https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-video-basic-user-demo/start
- Joux, A., & Vitse, V. (2018). A Crossbred Algorithm for Solving Boolean Polynomial Systems. In J. Kaczorowski, J. Pieprzyk, & J. Pomykała (Eds.), *Number-Theoretic Methods in Cryptology* (pp. 3–21). Springer International Publishing.
- Kennedy, F. (2023). *Solving Binary MQs on FPGA*. Worcester Polytechnic Institute; Digital WPI. https://digital.wpi.edu/show/k643b455z
- Silva, J., & Sakallah, K. (1996). *Grasp—a new search algorithm for satisfiability*. Computer Science Carnegie Mellon. https://www.cs.cmu.edu/~emc/15-820A/reading/grasp_iccad96.pdf

Appendix B: Finding Binary Weight of Two Code

```
module hamming #(parameter length = 7)( //Defines how many bits long testing
        input [length-1:0] counter, //arrays start @ 0
        output flag2
        );

        integer i;
        reg [length -1:0] weight;

        always @ (counter) begin
        weight = 0;
        for(i = 0; i <= length; i = i + 1)begin
        if(counter[i] == 1'b1) begin
                weight = weight + counter[i];
        end
        end
        end

        assign flag2 = (weight > 1 && weight < 3) ? 1'b1 : 1'b0;

endmodule
```

Appendix C: Binary Weight of Two Lookup Table Code

```
module weight2_lut(
	input [4:0] counter,
	output reg [6:0] coeff
	);

	always @ (counter) begin
	case(counter)
		5'd0: coeff <= 7'd3;
		5'd1: coeff <= 7'd5;
		5'd2: coeff <= 7'd6;
		5'd3: coeff <= 7'd9;
		5'd4: coeff <= 7'd10;
		5'd5: coeff <= 7'd12;
		5'd6: coeff <= 7'd17;
		5'd7: coeff <= 7'd18;
		5'd8: coeff <= 7'd20;
		5'd9: coeff <= 7'd24;
		5'd10: coeff <= 7'd33;
		5'd11: coeff <= 7'd34;
		5'd12: coeff <= 7'd40;
		5'd13: coeff <= 7'd48;
		5'd14: coeff <= 7'd65;
		5'd15: coeff <= 7'd66;
		5'd16: coeff <= 7'd68;
		5'd17: coeff <= 7'd72;
		5'd18: coeff <= 7'd80;
		5'd19: coeff <= 7'd96;
		default: coeff <= 7'd3;
	endcase
	end
endmodule
```

Appendix D: Two Equation Weight of Two Solving Module Code

```
module solver(
	input [3:0] terms,
	input [8:0] Co_1, Co_2,
	output solved
	);

	wire X1, X2, X3, X4;
	assign {X4, X3, X2, X1} = terms[3:0];

	//X1 + X2 + X3 + X1X2 + X2X3 + X3X4 + Rest = RHS
	//7 Terms + Rest + RHS = 9 Bit long Equations

	wire X1X2 = X1 & X2;
	wire X2X3 = X2 & X3;
	wire X3X4 = X4 & X3;

	wire [8:0] temp_eq = Co_1 ^ Co_2;   //XOR arrays

	wire result = (X1 & temp_eq[8]) ^ (X2 & temp_eq[7]) ^ (X3 & temp_eq[6]) ^(X4 *
temp_eq[5]) ^(X1X2 & temp_eq[4]) ^(X2X3 & temp_eq[3]) ^(X3X4 & temp_eq[2]) ^
temp_eq[1];

	assign solved = (result == temp_eq[0]) ? 1'b1 : 1'b0;

endmodule
```

Appendix E: Two Equation Weight of Two Solving Simulation Code

```
`timescale 1ns / 1ps

module weight2_table(

    );
    reg [4:0] eq1_count, eq2_count;   //iterate for loop
    reg [2:0] RHS;  //for equations
    wire RHS1, RHS2;
    assign {RHS2, RHS1} = RHS[1:0];
    reg [1:0] Rest;   //for equations
    wire [6:0] eq1_co, eq2_co;  //output of lut
    integer sys_num, solutions_num; //for # of systems and solutions per system
    reg [4:0] terms;          //for iterating through X1-X4
    wire solved;    //for saying if solution

    //Equations
    wire[8:0] eq1, eq2;
    assign eq1 = {eq1_co, Rest[0], RHS1};
    assign eq2 = {eq2_co, Rest[0], RHS2};

    weight2_lut eq1_coefficient(
    .counter(eq1_count),
    .coeff(eq1_co)
    );

    weight2_lut eq2_coefficient(
    .counter(eq2_count),
    .coeff(eq2_co)
    );

    solver algorithm(
    .terms(terms[3:0]),
    .Co_1(eq1),
    .Co_2(eq2),
    .solved(solved)
    );
```

```
        initial begin
        eq1_count = 0;
        eq2_count = 1; //avoid "squared" positions
        RHS = 3'b000;
        Rest = 2'b00;
        sys_num = 0;
        solutions_num = 0;
        terms = 0;
        #20;
        while(Rest[1] != 1'b1) begin
        while(RHS[2] != 1'b1) begin
                while(eq1_count <= 18) begin
                while(eq2_count <= 19) begin
                sys_num = sys_num + 1;
                $display("System # %d", sys_num);
                $display("EQ1 : %b", eq1);
                $display("EQ2 : %b", eq2);
                while(terms[4] != 1) begin
                        if(solved) begin
                        solutions_num = solutions_num + 1;
                        $display("Solution : X1 = %b X2 = %b X3 = %b X4 = %b", terms[0],
terms[1], terms[2], terms[3]);
                        end
                        #5 terms = terms + 1;
                end
                terms = 0;
                solutions_num = 0;
                eq2_count = eq2_count + 1'b1;
                end
                eq1_count = eq1_count + 1'b1;
                eq2_count = eq1_count + 1'b1;
                end
                RHS = RHS + 1'b1;
                eq1_count = 0;
                eq2_count = 1;
        end
```

```
        Rest = Rest + 1'b1;
        RHS = 3'b000;
        eq1_count = 0;
        eq2_count = 1;
        end
        $stop;
        end


endmodule
```

Appendix F: Two Equation Weight of Two Solving Results

Due to the brevity of the results, a link to the text on Github has been provided.

Appendix G: Exhaustive Search Code

```
module exhaustive_search(
	//input clk, reset_n,
	input [4:0] sw,
	output [5:0] led
	);

	// Define packed struct for equation coefficients
	typedef struct packed {
	logic [15:0] coefficient;
	} EquationCoeff;

	//16 equations with 15 coefficients + RHS
	EquationCoeff EQ_Matrix [0:15];   //X1 + X2 + X3 + X4 + X5 + X1X2 + X1X3 + X1X4 + X1X5 + X2X3 + X2X4 + X2X5 + X3X4 + X3X5 + X4X5 = RHS

	wire X1, X2, X3 ,X4 ,X5 ,X1X2, X1X3, X1X4, X1X5, X2X3, X2X4, X2X5, X3X4, X3X5, X4X5;
	wire [4:0] Terms = sw[4:0];
	assign X1 = Terms[0];		//Assign Linear Terms
	assign X2 = Terms[1];
	assign X3 = Terms[2];
	assign X4 = Terms[3];
	assign X5 = Terms[4];

	//Assigning Quad Terms
	assign X1X2 = X1 & X2;
	assign X1X3 = X1 & X3;
	assign X1X4 = X1 & X4;
	assign X1X5 = X1 & X5;
	assign X2X3 = X2 & X3;
	assign X2X4 = X2 & X4;
	assign X2X5 = X2 & X5;
	assign X3X4 = X3 & X4;
	assign X3X5 = X3 & X5;
	assign X4X5 = X4 & X5;
```

```
        wire [14:0] term_string = {X1, X2, X3, X4 ,X5, X1X2, X1X3, X1X4, X1X5, X2X3,
X2X4, X2X5, X3X4, X3X5, X4X5}; //easier for computation

        //Equations
        assign EQ_Matrix[0].coefficient = 16'b01001_1011_001_11_0_1;  //X2 + X5 + X1X2 +
X1X4 + X1X5 + X2X5 + X3X4 + X3X5 = 1
        assign EQ_Matrix[1].coefficient = 16'b10110_0001_101_01_1_0;
        assign EQ_Matrix[2].coefficient = 16'b11001_1010_010_10_0_1;
        assign EQ_Matrix[3].coefficient = 16'b11110_1100_111_00_1_0;
        assign EQ_Matrix[4].coefficient = 16'b01011_0011_011_10_1_0;
        assign EQ_Matrix[5].coefficient = 16'b01000_0100_110_01_0_0;
        assign EQ_Matrix[6].coefficient = 16'b00000_0101_101_01_0_1;
        assign EQ_Matrix[7].coefficient = 16'b01101_1001_111_01_1_0;
        assign EQ_Matrix[8].coefficient = 16'b00100_0100_101_01_0_1;
        assign EQ_Matrix[9].coefficient = 16'b11100_0000_100_00_1_0;
        assign EQ_Matrix[10].coefficient = 16'b10110_1000_101_10_1_1;
        assign EQ_Matrix[11].coefficient = 16'b00101_1111_100_00_1_0;
        assign EQ_Matrix[12].coefficient = 16'b11111_1011_010_00_0_0;
        assign EQ_Matrix[13].coefficient = 16'b01101_0011_101_01_0_0;
        assign EQ_Matrix[14].coefficient = 16'b00001_0010_001_00_0_1;
        assign EQ_Matrix[15].coefficient = 16'b10100_0111_100_11_0_1;

        //XOR all coefficients together
        wire [15:0] EQ_XOR = (EQ_Matrix[0].coefficient ^ EQ_Matrix[1].coefficient ^
EQ_Matrix[2].coefficient ^ EQ_Matrix[3].coefficient ^
                EQ_Matrix[4].coefficient ^ EQ_Matrix[5].coefficient ^ EQ_Matrix[6].coefficient
^ EQ_Matrix[7].coefficient ^
                EQ_Matrix[8].coefficient ^ EQ_Matrix[9].coefficient ^
EQ_Matrix[10].coefficient ^ EQ_Matrix[11].coefficient ^
                EQ_Matrix[12].coefficient ^ EQ_Matrix[13].coefficient ^
EQ_Matrix[14].coefficient ^ EQ_Matrix[15].coefficient);

        //AND all coefficients and terms together
        wire [14:0] EQ_AND = EQ_XOR[15:1] & term_string;

        //XOR EQ_AND together
```

```
        wire EQ_SUM = (EQ_AND[14] ^ EQ_AND[13] ^ EQ_AND[12] ^ EQ_AND[11] ^
EQ_AND[10]
                ^ EQ_AND[9] ^ EQ_AND[8] ^ EQ_AND[7] ^ EQ_AND[6] ^ EQ_AND[5]
                ^ EQ_AND[4] ^ EQ_AND[3] ^ EQ_AND[2] ^ EQ_AND[1] ^ EQ_AND[0]);

        //Check to see if sum = RHS of XOR'd equations
        wire solution = (EQ_SUM == EQ_XOR[0]) ? 1'b1 : 1'b0;

        assign led = (solution) ? {1'b0,Terms[4:0]} : 6'b1_00000;    //display the solution on the
LEDs

        //Used for Simulation Purposes (working on a way to do this on FPGA)
  /*initial begin
        $display("Solving System of Equations");
        $display("Equation 1: %b", EQ_Matrix[0].coefficient);
        $display("Equation 2: %b", EQ_Matrix[1].coefficient);
        $display("Equation 3: %b", EQ_Matrix[2].coefficient);
        $display("Equation 4: %b", EQ_Matrix[3].coefficient);
        $display("Equation 5: %b", EQ_Matrix[4].coefficient);
        $display("Equation 6: %b", EQ_Matrix[5].coefficient);
        $display("Equation 7: %b", EQ_Matrix[6].coefficient);
        $display("Equation 8: %b", EQ_Matrix[7].coefficient);
        $display("Equation 9: %b", EQ_Matrix[8].coefficient);
        $display("Equation 10: %b", EQ_Matrix[9].coefficient);
        $display("Equation 11: %b", EQ_Matrix[10].coefficient);
        $display("Equation 12: %b", EQ_Matrix[11].coefficient);
        $display("Equation 13: %b", EQ_Matrix[12].coefficient);
        $display("Equation 14: %b", EQ_Matrix[13].coefficient);
        $display("Equation 15: %b", EQ_Matrix[14].coefficient);
        $display("Equation 16: %b", EQ_Matrix[15].coefficient);
        end

        always @(*) begin
        if(solution) begin
        $display("Solution Found: X1 = %b, X2 = %b, X3 = %b, X4 = %b, X5 = %b", Terms[0],
Terms[1], Terms[2], Terms[3], Terms[4]);
        end
```

```
        else begin
        $display("Solution not found at this state");
        end
        end*/

endmodule
```

Appendix H: Exhaustive Search Simulation Code

```
`timescale 1ns / 1ps


module exhaust_sim();
	reg [4:0] terms;
	wire [5:0] good_out;   //showing what terms work

	exhaustive_search uut(
	.sw(terms),
	.led(good_out)
	);

	initial begin
	terms <= 5'b00000;
		repeat(31) begin	//go until 5'b11111
			if(terms == 5'b11111) begin
				$stop;
			end
			#10 terms <= terms + 1'b1;
		end
	end
endmodule
```

Appendix I: Weight of Three Lookup Table

```
module weight3_lut(
	input [4:0] counter,
	output reg [6:0] coeff
	);

	always @ (counter) begin
	case(counter)
	5'd0: coeff <= 7'd7;
	5'd1: coeff <= 7'd11;
	5'd2: coeff <= 7'd13;
	5'd3: coeff <= 7'd14;
	5'd4: coeff <= 7'd19;
	5'd5: coeff <= 7'd21;
	5'd6: coeff <= 7'd22;
	5'd7: coeff <= 7'd25;
	5'd8: coeff <= 7'd26;
	5'd9: coeff <= 7'd28;
	5'd10: coeff <= 7'd35;
	5'd11: coeff <= 7'd37;
	5'd12: coeff <= 7'd38;
	5'd13: coeff <= 7'd41;
	5'd14: coeff <= 7'd42;
	5'd15: coeff <= 7'd44;
	5'd16: coeff <= 7'd49;
	5'd17: coeff <= 7'd50;
	5'd18: coeff <= 7'd52;
	5'd19: coeff <= 7'd56;
	5'd20: coeff <= 7'd67;
	5'd21: coeff <= 7'd69;
	5'd22: coeff <= 7'd70;
	5'd23: coeff <= 7'd73;
	5'd24: coeff <= 7'd74;
	5'd25: coeff <= 7'd76;
	5'd26: coeff <= 7'd81;
	5'd27: coeff <= 7'd82;
	5'd28: coeff <= 7'd84;
	5'd29: coeff <= 7'd88;
	5'd30: coeff <= 7'd97;
```

```
        5'd31: coeff <= 7'd98;
        5'd32: coeff <= 7'd100;
        5'd33: coeff <= 7'd104;
        5'd34: coeff <= 7'd112;
        default: coeff <= 7'd3;
        endcase
        end
endmodule
```

Appendix J: Parametrized Binary Weight Code

```
module weightfinding#(parameter length = 7, parameter weight = 3)(
	input [length-1:0] counter, //arrays start @ 0
	output flag
	);

	integer i;
	reg [length -1:0] weight_count;

	always @ (counter) begin
		weight_count = 0;
		for(i = 0; i <= length; i = i + 1)begin
			if(counter[i] == 1'b1) begin
				weight_count = weight_count + counter[i];
			end
		end
	end

	assign flag = (weight_count > (weight-1) && weight_count < (weight+1) ) ? 1'b1 : 1'b0;

endmodule
```