Solving Binary Multivariate Quadratic Systems on 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

Frank Kennedy

Date: 4/26/2023

Project Advisor: Dr. Koksal Mus

Acknowledgements

I would like to thank Professor Koksal Mus for being my advisor during this MQP and for his endless support allowing me to complete this project. Professor Mus was always available and ready to answer whatever questions I had for this project, and his insight proved to be invaluable during these terms. I would also like to thank Worcester Polytechnic Institute and the Electrical and Computer Engineering for giving me the opportunity to participate in this project. Finally, I would like to thank James McAllese, Liam Stearns, and Carlton Mugo for their research on the topic and working towards a linear system solver provided me with an amazing starting point for my work. Their work made learning about the systems engaging and provided me with a better starting point when developing my code instead of working purely from scratch.

Abstract

The goal of this project was to design a program that would be able to solve a given set of quadratic equations with binary coefficients that would effectively reduce the complexity required to solve the system normally. To achieve this, the system was converted into a matrix, which was then simplified by using a modified version of Gaussian elimination in order to find partial solutions that were then combined into a full solution once tested across the entire system. To verify the calculated solutions, an exhaustive search was also completed to ensure validity. The code for both the modified Gaussian (the recursive method) and the exhaustive method can be seen in the Appendix.

Table of Contents

Table of Figures

Background

The aim of this project was to develop a program that could be used to efficiently solve binary quadratic systems on low resource hardware. To achieve this, the given system would first be converted into a matrix that would allow the program to use linear algebra to compute the solutions. Ordinarily, when using linear algebra to solve a system of equations, there can either be no solutions, one solution, or an infinite number of solutions. In the case of a binary system, where the values are either 0 or 1, the infinite number of solutions that could have resulted are brought down to 2^n possible solutions, where n represents the number of variables in the system. For this project, while a quadratic system is being used, it varies very slightly than a linear system when solving. As previously mentioned, in a binary system, the only applicable values are 0 and 1. Due to this, anything multiplied by 0 is 0, and anything multiplied by 1 is itself, which simulates a more linear system. However, when looking at the case of 2^n possible solutions, the n here now represents the n linear variables present in the system, none of the quadratic.

To achieve this goal, a modified version of Gaussian elimination and an exhaustive search will be used. As written about in a paper by Wen Wang, and implemented by Liam Stearns and Carlton Mugo, to solve a large system converted into a matrix, one should break the matrix into smaller segments and solve for the segments produced individually. Due to the limitations of Gaussian elimination, which can result in numbers beyond just 0 and 1, the exhaustive search is used in conjunction to help verify the solutions as they are found. While this does work with linear systems, in a quadratic system, more adjustments must be made.

Linear Solving Improvements

While reviewing the solving process for the linear equation system, it was seen that having more subsections, or smaller pieces to find partial solutions for would ultimately help reduce the complexity of the system as a whole. Ordinarily, the system would have 2^n potential solutions, where n is equal to the number of variables featured in the system. However, by reducing the system into subcategories, the complexity of the system would decrease as the pieces increased. As an example, in a system with eight variables, the number of possible solutions available is 256. If the system could be broken up into four different groups, the number of possible solutions is now 2^{n-s}, where s is equivalent to the number of groups. This results in 16 possible solutions, which is drastically better than the initial amount.

Additionally, circumstances where equations were exact except for a few variables after the recursive sorting was done were also observed. To explore this relationship, the team used two equations, with equation 1 being written as 0101 0110 = 1, and equation 2 as 0100 0110 = 0. To establish a relationship between possible solutions and a case like this, a table was created to compare the coefficients and the potential solutions to the answers provided in the original equation. For a difference in one variable, the following rules were established below. In the table below, the Rest category defines what the solution adds to using bitwise addition with the exception of the variable in question. The coefficients and right-hand side categories refer to what the variable in question is and what their associated answer is. The solution category states whether a solution exists or not, and the possible solution section states what the variable in question must be masked with in order for the solution, in conjunction with the rest of the solution, to exist in the problem.

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 1: Rules for Two Equations with One Differing Variable

These rules help simplify the solution finding process even more. By finding what the variable must be in the solution, the team is able to remove it fully from the equation by making proper adjustments through adding the variable to the right-hand side. The removal of this variable

helps shrink the complexity of the search, making what was originally 2^8 possible solutions now 2^7 possible solutions.

To refine this process even further, the team additionally tested this theory with two differing coefficients. Like the trials above, the same equations were utilized, however one coefficient was changed so that the two different coefficient case could be tested. This meant using equation 1 as 0101 0110 = 1 and equation 2 as 0100 0110 = 0 (for example, equation 1 would be listed as 0101 1110 = 1, and equation 2 as 0100 0110 = 0). In addition to this, the team also tested these conditions under a variety of factors, such as differing variables and differing right hand sides, or the same coefficients and same right-hand sides. This was more complex than the one differing coefficient but led to more cohesive discoveries found below. In the following tables, the blue font stands for the original equations, green are the coefficients in question, and red is for the solution coefficients that are being changed. For a solution to be viable, it must work for both equations. This is signified by having two 1s in the RHS column. The first 1/0 signifies equation 1 and the second 1/0 signifies equation 2. If a solution works, it will be highlighted in green. Below is the first case, where the coefficients for each equation were the same, but the right-hand sides were different.

	A1	A2	A3	A4	A5	A6	A7	A8	RHS
E1	0	1	0	1	1	1	1	0	1
E2	0	1	0	0	0	1	1	0	0
	A1	A2	A3	A4	A5	A6	A7	A8	E1|E2
Solutions	1	1	1	0	0	1	1	1	Y|N
	1	1	1	0	1	1	1	1	N N
	1	1	1	1	0	1	1	1	N N
	1	1	1	1	1	1	1	1	Y N
	1	0	1	0	0	1	1	1	N Y
	1	0	1	0	1	1	1	1	Y Y
	1	0	1	1	0	1	1	1	Y Y
	1	0	1	1	1	1	1	1	N Y

Figure 2: Possible Solutions for Equations with Same Coefficients but Different Right-Hand Sides

As seen above, for this case, a solution is only present when the rest of the solution is equivalent to 0, and the coefficients (the ones in red) are either 01 or 10. Using this method, the team observed other cases featured in the table below.

Case #	Coefficients E1 = RHS	Coefficients E2 =RHS	Rest	Solutions
1	11 = 1	00 = 0	0	01, 10
2	11 = 1	00 = 1	1	00, 11
3	11 = 0	00 = 1	1	01, 10
4	11 = 0	00 = 0	0	00, 11
5	10 = 1	01 = 0	1	01
			0	10
6	10 = 1	01 = 1	1	00
			0	11
7	10 =0	01 = 1	1	10
			0	01
8	10 = 0	01 = 0	1	11
			0	00
9	00 = 1	11 = 0	1	01, 10
10	00 = 1	11 = 1	1	00, 11
11	00 = 0	11 = 1	0	01, 10
12	00 = 0	11 = 0	0	00, 11
13	01 = 1	10 = 0	1	10
			0	01
14	01 = 1	10 = 1	1	00
			0	11
15	01 = 0	10 = 1	1	01
			0	10
16	01 = 0	10 = 0	1	11
			0	00

Figure 3: Rules for Two Equations with Two Differing Variables

When reading the chart, the first column shows the case number, the second shows the coefficients in the first equation and what the equation equals to, the third column shows the coefficients of the second equation in question and that equations right hand side, the fourth column shows what the rest of the solution needs to be equal to for it to work, and the final column shows what the coefficients in question need to be in order for a solution to be found. As an example, when looking at case 1, equation 1 (or E1), would consist of xxx11xxx = 1 and equation 2 would be xxx00xxx = 0. For a solution to work in case 1, the rest of the solution must be equal to 0 (when you XOR the placeholder x's in the solution after multiplying it by the original coefficients, they must result to 0) and the variables in question can be either 01 or 10. By multiplying these variables by the ones in each of the equations in case 1, the solution will result in the given right-hand sides (RHS) seen in the original equations, allowing one to store these values to be used later in the overall solution. Like the initial observation with the single differing variable, solving this allows the team to simplify the equations even further by eliminating two variables from the system entirely.

Recursive Search Program

The recursive search program primarily deals with dividing the system matrix into smaller pieces, which in turn decreases the complexity of the system. As completed before by Liam Stearns and Carlton Mugo, the recursive search program for this project will also rely on first organizing the linear portions of the into a more simplified matrix. In order to do this, the linear portion of the matrix is treated as its own separate section. When the separation is completed, it is then organized into three different parts, an upper triangle consisting of zeros, the independent variables, and the dependent variables. The independent variables here serve to separate the zeros from the dependent variables and are the ones that will be solved for when deciding the partial solutions. Through this method, it is ideal to isolate the independent variables so that the initial equation does not contain dependent variables since the dependent variables add to the complexity of the search function. As the function proceeds down the different equations, the solved independent variables will become part of the dependent variable sets of the subsequent equations. This division can be seen more clearly in Figure 1 below.

Sort the Matrix to Create an Upper Triangle of 0s and a Section of Grouped 1s on Each Row

S_n	...	S_3	S_2	S_1	RHS	
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1 or 0}	E_1
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1 or 0}	E_2
{0, 0, 0, ... , 0}	{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1 or 0}	E_3
{0, 0, 0, ... , 0}	{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1 or 0}	...
{1, 1, 1, ... , 1}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1s & 0s}	{1 or 0}	E_n

Figure 4: Matrix Sorted into Independent and Dependent Variables

Like the linear binary system, the quadratic approach organizes the matrix in a similar fashion. To start this process, linear weights are assigned to each equation. The weight of each equation is determined by the number of ones an equation has in its linear section, excluding the right-hand side's value (referred to as RHS). An equation with a single linear variable will have a weight of 1 while an equation with four linear variables will have a weight of 4. Once a weight for each equation is determined, the matrix would then be organized in ascending order. Here, the smallest weights would be placed at the top of the matrix and the largest weights would be placed towards the bottom. This reduces the need for subsets, which were used in the linear recursive search algorithm in order to achieve the same result: positioning equations with lesser weights and shared variables above heavier ones. Additionally, this allows for the smaller groupings of ones to be found earlier on, decreasing the runtime. Once this step is completed, the program then focuses on developing the upper triangle of zeros. To do this, the ones in each row are pushed as far right as possible to replace the zeros located there. For the initial equation, as it only has a single one, this one will be seen in the rightmost position. As the program steps down to the next equation, that column would then be ignored in subsequent reorganizations until the upper zero triangle is formed.

After this linear reorganization is completed, the quadratic variables would then be added to the final matrix. Here, these values will not undergo the same organization, but will instead be placed in the same order as seen in the linear portion. As an example, if we have the linear variables X1, X2, X3, with X3 featured in the rightmost column of the linear matrix, the first quadratic section would feature all values including X3 (X3X1 and X3X2 in that order), then

X2 quantities (X2X1). There would not be an X1 section as those quantities have already been accounted for.

Ordinarily, to solve for the partial solutions at this point, one would compare the first section of independent variables to the right-hand side. In the quadratic system, while this can be done, it fails to incorporate the quadratic variables featured in the system, which can alter the validity of the solution. To alleviate this issue, the quadratic recursive function will first start by setting the initial linear value to either 0 or 1. While this will extend the runtime of the system compared to the previous linear approach, this will help by reducing the complexity of the system overall. As mentioned previously, when multiplied by 0 or 1, the resulting value will either be 0 or the multiplicand. By doing this, as a partial solution is found, it can then be applied to the rest of matrix, drastically reducing it. A variable of 0 would result in its erasure in the system, while a variable of 1 would result in the value being multiplied by, turning the quadratic value into a linear one which can then be added to the linear section. Verifying the partial solutions found will still act the same as the linear code, however. To do so, the solution is first masked with the variables in the equation, which then undergo bitwise addition in order to gain one value, the right-hand side. After comparing this value to the actual right-hand side of the equations, if they both are equal, the partial solution is stored, and the program works on the next equation. If a partial solution does fail, the program goes back to the previous equation and searches for another valid solution. As the aim of this program is to reduce the complexity of the system, the aim of the independent variables is to make sure only one variable is present in each group. This means that there are only 2^1 possible solutions for it, and if both fail, there are no valid solutions for the system.

Example System

Below is an example of a set of eleven quadratic equations organized into a matrix before being used in the recursive search:

	X1	X2	X3	X4	X5	X1X2	X1X3	X1X4	X1X5	X2X3	X2X4	X2X5	X3X4	X3X5	X4X5	RHS
E1	0	1	1	0	1	0	1	1	1	0	0	0	0	0	0	1
E2	1	1	1	0	0	0	1	0	1	0	0	1	1	0	1	1
E3	1	1	0	1	0	0	1	1	1	1	1	1	0	1	1	0
E4	1	0	1	1	1	0	0	1	0	0	0	1	1	1	0	0
E5	1	1	1	0	1	1	1	0	1	1	1	0	0	0	1	0
E6	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1
E7	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1
E8	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0
E9	1	1	1	0	0	0	1	0	0	1	0	1	0	0	1	0
E10	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1
E11	1	1	0	1	0	0	1	0	0	0	1	0	1	0	1	0

Figure 5: Matrix Organization of Quadratic System

To begin the organization, first only the linear portion of the equations are considered. Here, the quadratic portions are removed, and weights are assigned to each equation.

	X1	X2	X3	X4	X5	RHS	Weights
E1	0	1	1	0	1	1	3
E2	1	1	1	0	0	1	3
E3	1	1	0	1	0	0	3
E4	1	0	1	1	1	0	4
E5	1	1	1	0	1	0	4
E6	0	0	1	1	0	1	2
E7	0	0	0	1	0	1	1
E8	0	0	0	1	1	0	2
E9	1	1	1	0	0	0	3
E10	0	1	0	1	0	1	2
E11	1	1	0	1	0	0	3

Figure 6: Linear Matrix with Assigned Weights

With the weights now determined, the matrix is then reorganized with the lesser weights on top and the heavier weights at the bottom.

	X1	X2	X3	X4	X5	RHS
E7	0	0	0	1	0	1
E8	0	0	0	1	1	0
E6	0	0	1	1	0	1
E3	1	1	0	1	0	0
E10	0	1	0	1	0	1
E1	0	1	1	0	1	1
E2	1	1	1	0	0	1
E9	1	1	1	0	0	0
E11	1	1	0	1	0	0
E4	1	0	1	1	1	0
E5	1	1	1	0	1	0

Figure 7: Linear Matrix Organized by Weight

Now that the matrix has been organized by weight, it can now be reorganized to form the upper triangle grouping, independent variable grouping, and dependent variable grouping. To do this, starting with the first equation in the matrix, if a 1 is found, the entire column swaps places with the first 0. In this system, to begin the process, X5 and X4 will swap positions. When looking at the subsequent equations, now that X4 has already been moved to its required spot, it will not be considered in future organizations. The result of this can be seen below. For clarity, the colors used in Figure 1 will be applied here to better highlight the similarities. They will be reverted in future examples.

	X1	X2	X3	X5	X4	RHS
E7	0	0	0	0	1	1
E8	0	0	0	1	1	0
E6	0	0	1	0	1	1
E3	1	1	0	0	1	0
E10	0	1	0	0	1	1
E1	0	1	1	1	0	1
E2	1	1	1	0	0	1
E9	1	1	1	0	0	0
E11	1	1	0	0	1	0
E4	1	0	1	1	1	0
E5	1	1	1	1	0	0

Figure 8: Linear Matrix After Column Swapping

Now that the linear matrix has been fully organized, the quadratic variables are added back to the full matrix to be used for the search process.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		1					1		1		0	0	0	0	1	1
E8	1		1								0	0	0	1	1	0
E6				1		1	1	1	1		0	0	1	0	1	1
E3		1	1	1	1	1	1	1		1	1	1	0	0	1	0
E10				1							0	1	0	0	1	1
E1		1		1			1				0	1	1	1	0	1
E2		1		1	1				1	1	1	1	1	0	0	1
E9		1	1		1					1	1	1	1	0	0	0
E11		1						1	1	1	1	1	0	0	1	0
E4					1	1	1		1		1	0	1	1	1	0
E5	1	1	1	1				1		1	1	1	1	1	0	0

Figure 9: Full Quadratic Matrix to be used in the Recursive Search

With this done, to begin, the first variable, X4, will be set to 0 in the initial search. Due to this, the column can be ignored as anything multiplied by 0 results in 0. The same will be applied to all X4 terms.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
							0	0	0	0					0	
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		1									0	0	0	0		1
E8	1		1								0	0	0	1		0
E6				1		1					0	0	1	0		1
E3		1	1	1	1	1					1	1	0	0		0
E10				1							0	1	0	0		1
E1		1		1							0	1	1	1		1
E2		1		1	1						1	1	1	0		1
E9		1	1		1						1	1	1	0		0
E11		1									1	1	0	0		0
E4					1	1					1	0	1	1		0
E5	1	1	1	1							1	1	1	1		0

Figure 10: Setting X4 Terms to Zero

Now that this step has been completed, the program can now search for the first partial solution. As seen in equation 7, or E7, for the right-hand side to be valid, X3X1 must be equal to 1. This means that both X3 and X1 individually equal 1 and can be accounted for in the potential solution. With these values now found, X3 and X1 are set to 1, reducing their quadratic terms to linear ones. These linear values are then added to the linear section of the whole matrix, being replaced with zeros to signify a change in value (in the linear section, a red value signifies this change, the quadratic portion does this with a grey value). This is seen in the following matrix:

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
	X2	X1	X2	X5		X5	0	0	0	0	1		1		0	
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		0									1	0	0	0		1
E8	0		0								0	0	0	1		0
E6				0		0					0	0	1	0		1
E3		0	0	0	1	0					0	0	0	0		0
E10				0							0	1	0	1		1
E1		0		0							1	1	1	0		1
E2		0		0	1						0	1	1	1		1
E9		0	0		1						0	0	1	0		0
E11		0									0	1	0	0		0
E4					1	0					1	0	1	0		0
E5	0	0	0	0							0	1	1	0		0

Figure 11: Updated Matrix with X1=X3=1 and X4=0

With this completed, the program will now step down to the next equation, Equation 8, and find the next partial solution. As seen here, X5 is the only variable in question. For the partial solution to work, X5 must equal 0. With this now found, all X5 terms will be set to 0, removing them from the matrix entirely.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
	X2	X1	X2	0	0	0	0	0	0	0	1		1	0	0	
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		0									1	0	0			1
E8	0		0								0	0	0			0
E6											0	0	1			1
E3		0	0								0	0	0			0
E10											0	1	0			1
E1		0									1	1	1			1
E2		0									0	1	1			1
E9		0	0								0	0	1			0
E11		0									0	1	0			0
E4											1	0	1			0
E5	0	0	0								0	1	1			0

Figure 12: Updated Matrix with X1=X3=1 and X4=X5=0

Now, while the previous values so far have been solved for, the adjustments made to the matrix have broken the independent variable grouping that was defined previously. To overcome this, the program will just reorganize the rows to replace the required independent variable. Since X2 is the only variable required, equation 3 and equation 10 will swap places. This could have been done in an early phase of the searching algorithm, but as this only affected one variable, in this example was left towards the end. In the code itself, this anomaly will be accounted for as it arises.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
	X2	X1	X2	0	0	0	0	0	0	0	1		1	0	0	
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		0									1	0	0			1
E8	0		0								0	0	0			0
E6											0	0	1			1
E10											0	1	0			1
E3		0	0								0	0	0			0
E1		0									1	1	1			1
E2		0									0	1	1			1
E9		0	0								0	0	1			0
E11		0									0	1	0			0

E4											1	0	1			0
E5	0	0	0								0	1	1			0

Figure 13: Reorganized Matrix to Redefine Independent Variables

With this modification made, the program will see that in equation 10, for the right-hand side to be valid, X2 must be equal to 1. Now that X2 has been decided, the full solution can be tested to verify its validity: X1 = X2 = X3 = 1 and X4 = X5 =0. As X5 and X4 are equal to zero, to simplify this test below, only the values for X1, X2, and X3 will be shown. In the full code, every value will be used, but in the simplified matrix produced, these values have all be converted to a linear representation. To verify the solution, it will be first be masked to the values in the given variable slot and added together through bitwise addition. If the answer and RHS are equivalent, the solution is valid for that equation. If the two values are not equal, the solution fails for the whole system. This can be seen below.

EQ	X1	X2	X3	Ans	RHS
E7	1	0	0	1	1
E8	0	0	0	0	0
E6	0	0	1	1	1
E10	0	1	0	1	1
E3	0	0	0	0	0
E1	1	1	1	1	1
E2	0	1	1	0	1
E9	0	0	1	1	0
E11	0	1	0	1	0
E4	1	0	1	0	0
E5	0	1	1	1	0

Figure 14: Testing the Full Solution

As seen in the figure above, the given solution works for all the equations except for equation 2, equation 9, and equation 11. This means that having X4 = 0 as the initial case does not work and the program should test the case where X4 = 1. Here, while they are featured as separate cases, the program will work on both together as it goes, like the linear recursive method. To begin this

case, the program will start by setting all X4 terms to 1 and adding them to their respective linear components. In addition to this however, since the X4 linear values are guaranteed, these values are added to the right-hand side of the equation to reduce the complexity of the matrix. This can be seen below.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
							X1	X2	X3	X5					1	+X4
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7		1					0		0		1	0	1	0		0
E8	1		1								0	0	0	1		1
E6				1		1	0	0	0		1	1	0	0		0
E3		1	1	1	1	1	0	0		0	0	0	0	1		1
E10				1							0	1	0	0		0
E1		1		1			0				1	1	1	1		1
E2		1		1	1				0	0	1	1	0	1		1
E9		1	1		1					0	1	1	1	1		0
E11		1						0	0	0	1	0	1	1		1
E4					1	1	0		0		0	0	0	1		1
E5	1	1	1	1				0		0	1	0	1	0		0

Figure 15: Updated Initial Matrix Setting X4 =1

With the first matrix now organized, the program can now solve for the partial solutions like it did in the initial zero case. Starting with equation 7, for the right-hand side to be valid, X1 and X3 must be equal to 0. With this found, all X1 and X3 values can be excluded from the system.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
	X1=0	0	0	X1=0		X3=0	0	X2	0	X5	0		0		1	+X4
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7												0		0		0
E8												0		1		1
E6								0				1		0		0
E3					1			0		0		0		1		1
E10												1		0		0
E1												1		1		1
E2					1					0		1		1		1
E9					1					0		1		1		0
E11								0		0		0		1		1
E4					1							0		1		1
E5								0		0		0		0		0

Figure 16: Updated Matrix with X1 = X3 =0 and X4 =1

With the updated matrix, the program will now look at equation 8 to solve for X5. As X5 is now the only variable in equation 8, for the right-hand side to be valid, X5 must equal 1. The program will now update the matrix again as it did before.

	X2 Terms	X3 Terms		X5 Terms			X4 Terms				Linear					
	X1=0	0	0	X1=0	X2	X3=0	0	X2	0	X5	0		0	1	1	+X4
Eq	X2X1	X3X1	X3X2	X5X1	X5X2	X5X3	X4X1	X4X2	X4X3	X4X5	X1	X2	X3	X5	X4	RHS
E7												0		0		0
E8												0		1		1
E6								0				1		0		0
E3					0			0		0		1		1		1
E10												1		0		0
E1												1		1		1
E2					0					0		0		1		1
E9					0					0		0		1		0
E11								0		0		0		1		1
E4					0							1		1		1
E5								0		0		0		0		0

Figure 17: Updated Matrix with X1 = X3 = 0 and X4 = X5 =1

While this new matrix does not appear to fit the standard independent variable grouping seen in previous examples, as it only features X2 and X5, these columns can be brought closer

together to better visualize the grouping. Despite this, the matrix allows for the program to find the partial solution for X2 through equation 6. Seen here, for the right-hand side to be valid, X2 must be equal to 0. With this found, the program now has a partial solution with X1 = X2 = X3 = 0 and X4 = X5 = 1. As the X4 values have already been added to the right-hand side, the resulting X5 values can be tested against the partial solution. This will also be done by masking the partial solution with the X5 values. This can be seen below.

Eq	X5	Ans	RHS
E7	0	0	0
E8	1	1	1
E6	0	0	0
E3	1	1	1
E10	0	0	0
E1	1	1	1
E2	1	1	1
E9	1	1	0
E11	1	1	1
E4	1	1	1
E5	0	0	0

Figure 18: Testing the Full Solution

As seen here, the partial solution fails when applied to equation 9, which means this system does not have a solution.

Recursive Search Program Code

The recursive search program code used for the quadratic system is based off of the program Liam Stearns and Carlton Mugo completed for a linear problem set with modifications to mask and incorporate the nonlinear terms in. To begin the code, classes are made to establish the different equations present in the system, an info matrix to store the values of the equations in to later be used in simplification and row manipulation, and a linear matrix for the purely linear portions of the equations. Both the info matrix and equation classes include variables that discern the equation number, variables in the system, and the equations respective right-hand side, but the equation class includes an extra parameter, the linear weight of the equation, which is used to sort the equations in order of lightest to heaviest in the code. The linear matrix class functions like the info matrix class, but as the name implies, only considers the linear portions of the given equations. The equation number and right-hand side are also included here.

The next part of the code is sum matrix, which takes an equation and organizes it based on its weight in a temporary matrix. In the code, the sum matrix helps organize the full matrix, which represents the entire system, and the linear matrix, which is primarily used to develop the upper triangle and independent portions of the matrix that will later be used to solve the system. Here, the full matrix is used as a placeholder, as the values will not change until the independent variables are found. This only applies to the linear portion of the matrix since the quadratic values are untouched, but the matrix is still included to visually ensure that the organization step worked properly.

With the matrices now sorted based on weight, the program now starts to organize the matrices based on the 1's and 0's. The goal of this step is to clearly define the independent variables featured in the system that will then be prioritized in the solving mechanism. To do this, the code first steps through each cell in the linear matrix starting with the rightmost cell and searches for the nearest 1 in the row. If a 1 is found and there is a 0 before it, those columns are swapped, and the index variable of the code is increased, highlighting a position change. This allows the code in future rows to ignore the already changed columns, preventing further alterations. While this step does only change the positions found in the linear matrix, the full matrix is changed just by setting the initial columns equal to the linear matrix. Due to this, the full quadratic matrix now has the properly organized linear portions featuring the upper triangle, independent variables, and dependent variables, without changing the set quadratic ones.

Due to time restraints however, the code was not able to be finished in terms of the solving mechanism. To accomplish this however, a future team should consider making functions that will help reduce the matrix by making the initial lone independent variable either a 0 or a 1. In this quadratic example, when a variable is set to 0, all associated variables are also reduced to 0. This ultimately means that are removed from the matrix as they do not matter to the overall system anymore, but this can be done through masking. By checking the zero condition first, it allows the team to focus along one branch of the binary at a time, reducing the number of checks the code goes through overall. The other function setting the variable to 1 makes it so all associated variables are set to the multiplicand, turning what used to be quadratic variable into linear ones. These can be added to their respective terms, and the initial

independent variable can be added to their respective right-hand sides, as the quantity is now set. Like the zero case, it now allows for further simplification. This can be achieved through masking the variables as the code continues, but these cases should be observed initially.

Exhaustive Search Program

In addition to the recursive search algorithm, an exhaustive search algorithm was developed in order to gauge how efficient the alternative program was compared to testing every possible binary solution there was. To do this, the exhaustive search program would cycle between every solution available, test them against the given system, and store whichever solutions were valid. Solutions that failed to work would be discarded. To produce all the possible solutions that could be used for the system, a binary counter was utilized. For the system used in this program, it contained five linear variables, resulting in 2^5 possible solutions. The binary counter itself used this number to produce the resulting 5-bit numbers, starting with 0 and ending with 31, in order to test all 32 possible solutions.

Like the linear model developed by Liam Stearns and Carlton Mugo, when given a possible solution, the code would first mask the variables with the potential solution and then add them together using a bitwise or in order to produce a temporary answer. This answer would then be set equal to the original right hand side value of the given equation to verify if it worked, and if it did, the solution would be stored. The program differs when it is applied to the nonlinear portion of the equation, however. Unlike before, these portions require a double masking to account for the two variables being used. For example, when looking at x_1x_2, when masking this value, you must mask it with the solution for x1 and x2 before completing the bitwise addition, allowing for full coverage of the system.

Results

As of now, the effectiveness of the recursive search program is unable to be calculated due to the solving mechanism not being completed. This being said, the search program is based on the same process used to solve the linear set of equations, which reduces the possible number of solutions from 2^n down to $2^{n/2}$, which is a significant complexity drop. This is what the program for the quadratic system should theoretically result to, especially since it aims at converting the quadratic variables it has into as many linear parts as possible. The exhaustive search function also remained the same as the one used previously in the linear set. Despite completing two different things, the complexity stayed the same at 2^n.

While significant progress has been achieved, the code can still be improved to reduce the complexity even more. Like Liam Stearns and Carlton Mugo, I utilized many hard coded values in order to establish the equations and matrices used in the setup portion of the code. It would be beneficial if this could be avoided, streamlining the process even further. I also believe that the solving mechanism can be improved more by developing cases like in the linear portion where one or two variables differ. This would allow for table checks, reducing the complexity even further since hard solution values have already been established. An issue that can arise from this, however, is the memory that the board in use has. An Artix-7 Basys 3 FPGA, the board I utilize, has 32 megabits of non-volatile flash, which could be used up quickly as the complexity of the equations increases. While the methods developed for this project aim at reducing this already, quadratic equations grow much more rapidly than linear equations do when a single variable is added. This should be considered when continuing with the project.

Conclusion

During this project, the goal was to develop a program that would be able to solve binary quadratic systems of equations in a way that would effectively reduce the complexity required to solve the system out normally. While the code was not finished due to time constraints, the team was able to effectively develop a method to do so based on the research of Liam Stearns and Carlton Mugo. Through their research, and modifications made during this term, the goal of the method is to turn the quadratic portions of the system into linear segments, allowing for a reduction in not only the system, but in the complexity of the solution as well. Theoretically, this complexity should match the one found by Mugo and Stearns, $2^{n/2}$, but more work is needed to verify this. I hope the foundation developed during this project can be used by future groups in the completion of this program, and even further, as something that can be improved upon further.

References

1. Bard, G. V. (2009). Algebraic cryptanalysis. Springer Science & Business Media.
2. Bardet, M., Faugère, J.-C., Salvy, B., & Spaenlehauer, P.-J. (2013). On the complexity of solving quadratic Boolean systems. Journal of Complexity, 29(1), 53–75.10.1016/j.jco.2012.07.001
3. Keinänen, M., De, U., Keinänen, M., & Oy, M. (2005). SOLVING BOOLEAN EQUATION SYSTEMS
4. McAleese, J. (2021.). (rep.). Solving Systems of Linear Equations over GF(2) on FPGAs.
5. Mugo, C,. & Stearns, L. (2022.). (rep.). Solving Linear Binary Systems on FPGAs.
6. Wang, W., Szefer, J., & Niederhagen, R. (2016). Solving large systems of linear equations over GF(2) on FPGAs. 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 1–7. 10.1109/ReConFig.2016.7857188

Appendix

The code developed and used for this project can be seen below. Everything was coded on EDA Playground and can be simulated on the site without the need for an extra program or physical device.

EDA Playground Quadratic Exhaustive Code: https://www.edaplayground.com/x/LGwM

Incomplete EDA Playground Quadratic Recursive Code:

https://www.edaplayground.com/x/rEAW

Verilog Exhaustive Search Code

```
`timescale 1ns/1ns
//Create a module which organizes a given matrix to solve a linear system of equations
//Create a class which keeps track of an equation and any information tied to it
class equation;
 int eqnum;//Tells us which equation
 int x1;
 int x2;
 int x3;
 int x4;
 int x5;
 int x1x2;
 int x1x3;
 int x1x4;
 int x1x5;
 int x2x3;
 int x2x4;
 int x2x5;
```

```
  int x3x4;
  int x3x5;
  int x4x5;
  int rhs;//rhs value for equation
  int linearWeight;//weight of the equation (num of 1's excluding rhs)


  //Function within the class to display the sum for a given equation
  function void weight_display();
    $display("\teqnum = %0d, sum = %0d", eqnum, linearWeight);
  endfunction

endclass

//Create a class which keeps track of a given rows information for the matrix
//This makes it easier to perform row and column adjustments
class info_matrix;
  int eqnum;
  int c1;
  int c2;
  int c3;
  int c4;
  int c5;
  int c12;
  int c13;
  int c14;
  int c15;
  int c23;
  int c24;
  int c25;
  int c34;
  int c35;
```

```
  int c45;
  int rhs;
  //int temp;

  //Function used to display a matrix row
  function void matrix_display();
    $display("\tE%0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d |
%0d | %0d ", eqnum, c45, c35, c34, c25, c24, c23, c15, c14, c13, c12, c5, c4, c3, c2, c1, rhs);
   // $display("\t----------------------------------------");
  endfunction
endclass

module exhaustive_search;
  equation sum_matrix[1:11];//an array to sort based on eq weights
  equation eq;
  info_matrix full_matrix[1:11];//an array which acts as our matrix (deals with 1 dimension)
  info_matrix i_matrix;
  int t_matrix [1:11][1:15];//temp 2d matrix
  int final_matrix [1:11][1:15];
  int t_array [1:11]; //hold temp values
  int t_value;//holds the value for a given variable for swapping
  int t_rhs;//holds a temp rhs value when performing row operations
  int matrix_max;
  int bin_count;
  int t_bin_count;
  int bin_index;
  int num_operations;

  reg [5:1] binary = 5'd0;
  reg [5:1] t_binary = 5'd0;
  reg [5:1] t_solution = 5'd0;
  int solution_count;
```

```
reg eq_rhs1 = 1'b0;
reg eq_rhs2 = 1'b0;
reg eq_rhs3 = 1'b0;
reg eq_rhs4 = 1'b0;
reg eq_rhs5 = 1'b0;
reg eq_rhs6 = 1'b0;
reg eq_rhs7 = 1'b0;
reg eq_rhs8 = 1'b0;
reg eq_rhs9 = 1'b0;
reg eq_rhs10 = 1'b0;
reg eq_rhs11 = 1'b0;

int t_eqnum;
int index;
int index_array [1:11];
int search_pos;
int pos_array [1:15];//keeps track of where variables sit in our matrix
int pos_c_array [1:15];
int subset_found;//variable to tell us if a subset for organization has been found

initial begin
//initialize all of our equations
  eq = new();
  eq.eqnum = 1; // 0 1 1 0 1 0 1 1 1 0 0 0 0 0 0 1
  eq.x1 = 0;
  eq.x2 = 1;
  eq.x3 = 1;
  eq.x4 = 0;
  eq.x5 = 1;
  eq.x1x2 = 0;
  eq.x1x3 = 1;
  eq.x1x4 = 1;
```

```
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 3;
sum_matrix[1] = eq;

eq = new();
eq.eqnum = 2; //1 1 1 0 0 0 1 0 1 0 0 1 1 0 1 1
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 1;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 1; //1
eq.linearWeight = 3;
sum_matrix[2] = eq;
```

```
eq = new();
eq.eqnum = 3; // 1 1 0 1 0 0 1 1 1 1 1 1 0 1 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 1;
eq.x1x5 = 1;
eq.x2x3 = 1;
eq.x2x4 = 1;
eq.x2x5 = 1;
eq.x3x4 = 0;
eq.x3x5 = 1;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
sum_matrix[3] = eq;

eq = new();
eq.eqnum = 4; // 1 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0
eq.x1 = 1;
eq.x2 = 0;
eq.x3 = 1;
eq.x4 = 1;
eq.x5 = 1;
eq.x1x2 = 0;
eq.x1x3 = 0;
eq.x1x4 = 1;
eq.x1x5 = 0;
```

```
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 1;
eq.x3x5 = 1;
eq.x4x5 = 0;
eq.rhs = 0;
eq.linearWeight = 4;
sum_matrix[4] = eq;

eq = new();
eq.eqnum = 5; // 1 1 1 0 1 1 1 0 1 1 1 0 0 0 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 1;
eq.x1x2 = 1;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 1;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 4;
sum_matrix[5] = eq;

eq = new();
```

```
eq.eqnum = 6; // 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 1;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 0;
eq.x1x4 = 1;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 1;
eq.x4x5 = 0;
eq.rhs = 0;
eq.linearWeight = 2;
sum_matrix[6] = eq;

eq = new();
eq.eqnum = 7; // 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 1
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 1;
eq.x1x5 = 0;
eq.x2x3 = 0;
```

```
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 1;
sum_matrix[7] = eq;

eq = new();
eq.eqnum = 8; // 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 1;
eq.x1x2 = 1;
eq.x1x3 = 0;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 1;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 0;
eq.linearWeight = 2;
sum_matrix[8] = eq;

eq = new();
eq.eqnum = 9; // 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 0
```

```
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 1;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
sum_matrix[9] = eq;

eq = new();
eq.eqnum = 10; // 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1
eq.x1 = 0;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 0;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
```

```
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 2;
sum_matrix[10] = eq;

eq = new();
eq.eqnum = 11; //1 1 0 1 0 0 1 0 0 0 1 0 1 0 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 0;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
sum_matrix[11] = eq;

//Assigns values to "cells" of the matrix
foreach(sum_matrix[i])begin
  i_matrix = new();
```

```
i_matrix.eqnum = sum_matrix[i].eqnum;
i_matrix.c1 = sum_matrix[i].x1;
i_matrix.c2 = sum_matrix[i].x2;
i_matrix.c3 = sum_matrix[i].x3;
i_matrix.c4 = sum_matrix[i].x4;
i_matrix.c5 = sum_matrix[i].x5;
i_matrix.c12 = sum_matrix[i].x1x2;
i_matrix.c13 = sum_matrix[i].x1x3;
i_matrix.c14 = sum_matrix[i].x1x4;
i_matrix.c15 = sum_matrix[i].x1x5;
i_matrix.c23 = sum_matrix[i].x2x3;
i_matrix.c24 = sum_matrix[i].x2x4;
i_matrix.c25 = sum_matrix[i].x2x5;
i_matrix.c34 = sum_matrix[i].x3x4;
i_matrix.c35 = sum_matrix[i].x3x5;
i_matrix.c45 = sum_matrix[i].x4x5;
i_matrix.rhs = sum_matrix[i].rhs;
//fills up our temp 2d array with initial values
t_matrix[i][1] = sum_matrix[i].x1;
t_matrix[i][2] = sum_matrix[i].x2;
t_matrix[i][3] = sum_matrix[i].x3;
t_matrix[i][4] = sum_matrix[i].x4;
t_matrix[i][5] = sum_matrix[i].x5;
t_matrix[i][6] = sum_matrix[i].x1x2;
t_matrix[i][7] = sum_matrix[i].x1x3;
t_matrix[i][8] = sum_matrix[i].x1x4;
t_matrix[i][9] = sum_matrix[i].x1x5;
t_matrix[i][10] = sum_matrix[i].x2x3;
t_matrix[i][11] = sum_matrix[i].x2x4;
t_matrix[i][12] = sum_matrix[i].x2x5;
t_matrix[i][13] = sum_matrix[i].x3x4;
t_matrix[i][14] = sum_matrix[i].x3x5;
```

```
    t_matrix[i][15] = sum_matrix[i].x4x5;
    full_matrix[i] = i_matrix;
  end

  $display("\t   | x4x5| x3x5| x3x4| x2x5| x2x4| x2x3| x1x5| x1x4| x1x3| x1x2| x5| x4| x3| x2| x1|rhs");

  foreach(full_matrix[i])begin
    full_matrix[i].matrix_display();
  end
  $display("============================================================");
  solution_count = 0;

  $display("x5|x4|x3|x2|x1");
  while(bin_index <= 2**5)begin
    if(bin_index >= 2**5)begin
      $display("Number Of Operations = %d", num_operations);
      break;
    end
    binary = bin_index;

    //eq_rhs1 =
(binary[1]&full_matrix[1].c1)+(binary[2]&full_matrix[1].c2)+(binary[3]&full_matrix[1].c3)+(binary[4]&full_matrix[1]
.c4)+(binary[5]&full_matrix[1].c5)+(binary[6]&full_matrix[1].c6)+(binary[7]&full_matrix[1].c7);

    eq_rhs1 =
(binary[1]&full_matrix[1].c1)+(binary[2]&full_matrix[1].c2)+(binary[3]&full_matrix[1].c3)+(binary[4]&full_matrix[1]
.c4)+(binary[5]&full_matrix[1].c5)+(binary[1]&binary[2]&full_matrix[1].c12)+(binary[1]&binary[3]&full_matrix[1].c
13)+(binary[1]&binary[4]&full_matrix[1].c14)+(binary[1]&binary[5]&full_matrix[1].c15)+(binary[2]&binary[3]&full_
matrix[1].c23)+(binary[2]&binary[4]&full_matrix[1].c24)+(binary[2]&binary[5]&full_matrix[1].c25)+(binary[3]&bina
ry[4]&full_matrix[1].c34)+(binary[3]&binary[5]&full_matrix[1].c35)+(binary[4]&binary[5]&full_matrix[1].c45);

    eq_rhs2 =
(binary[1]&full_matrix[2].c1)+(binary[2]&full_matrix[2].c2)+(binary[3]&full_matrix[2].c3)+(binary[4]&full_matrix[2]
.c4)+(binary[5]&full_matrix[2].c5)+(binary[1]&binary[2]&full_matrix[2].c12)+(binary[1]&binary[3]&full_matrix[2].c
13)+(binary[1]&binary[4]&full_matrix[2].c14)+(binary[1]&binary[5]&full_matrix[2].c15)+(binary[2]&binary[3]&full_
matrix[2].c23)+(binary[2]&binary[4]&full_matrix[2].c24)+(binary[2]&binary[5]&full_matrix[2].c25)+(binary[3]&bina
ry[4]&full_matrix[2].c34)+(binary[3]&binary[5]&full_matrix[2].c35)+(binary[4]&binary[5]&full_matrix[2].c45);
```

```
    eq_rhs3 =
(binary[1]&full_matrix[3].c1)+(binary[2]&full_matrix[3].c2)+(binary[3]&full_matrix[3].c3)+(binary[4]&full_matrix[3]
.c4)+(binary[5]&full_matrix[3].c5)+(binary[1]&binary[2]&full_matrix[3].c12)+(binary[1]&binary[3]&full_matrix[3].c
13)+(binary[1]&binary[4]&full_matrix[3].c14)+(binary[1]&binary[5]&full_matrix[3].c15)+(binary[2]&binary[3]&full_
matrix[3].c23)+(binary[2]&binary[4]&full_matrix[3].c24)+(binary[2]&binary[5]&full_matrix[3].c25)+(binary[3]&bina
ry[4]&full_matrix[3].c34)+(binary[3]&binary[5]&full_matrix[3].c35)+(binary[4]&binary[5]&full_matrix[3].c45);


    eq_rhs4 =
(binary[1]&full_matrix[4].c1)+(binary[2]&full_matrix[4].c2)+(binary[3]&full_matrix[4].c3)+(binary[4]&full_matrix[4]
.c4)+(binary[5]&full_matrix[4].c5)+(binary[1]&binary[2]&full_matrix[4].c12)+(binary[1]&binary[3]&full_matrix[4].c
13)+(binary[1]&binary[4]&full_matrix[4].c14)+(binary[1]&binary[5]&full_matrix[4].c15)+(binary[2]&binary[3]&full_
matrix[4].c23)+(binary[2]&binary[4]&full_matrix[4].c24)+(binary[2]&binary[5]&full_matrix[4].c25)+(binary[3]&bina
ry[4]&full_matrix[4].c34)+(binary[3]&binary[5]&full_matrix[4].c35)+(binary[4]&binary[5]&full_matrix[4].c45);


    eq_rhs5 =
(binary[1]&full_matrix[5].c1)+(binary[2]&full_matrix[5].c2)+(binary[3]&full_matrix[5].c3)+(binary[4]&full_matrix[5]
.c4)+(binary[5]&full_matrix[5].c5)+(binary[1]&binary[2]&full_matrix[5].c12)+(binary[1]&binary[3]&full_matrix[5].c
13)+(binary[1]&binary[4]&full_matrix[5].c14)+(binary[1]&binary[5]&full_matrix[5].c15)+(binary[2]&binary[3]&full_
matrix[5].c23)+(binary[2]&binary[4]&full_matrix[5].c24)+(binary[2]&binary[5]&full_matrix[5].c25)+(binary[3]&bina
ry[4]&full_matrix[5].c34)+(binary[3]&binary[5]&full_matrix[5].c35)+(binary[4]&binary[5]&full_matrix[5].c45);


    eq_rhs6 =
(binary[1]&full_matrix[6].c1)+(binary[2]&full_matrix[6].c2)+(binary[3]&full_matrix[6].c3)+(binary[4]&full_matrix[6]
.c4)+(binary[5]&full_matrix[6].c5)+(binary[1]&binary[2]&full_matrix[6].c12)+(binary[1]&binary[3]&full_matrix[6].c
13)+(binary[1]&binary[4]&full_matrix[6].c14)+(binary[1]&binary[5]&full_matrix[6].c15)+(binary[2]&binary[3]&full_
matrix[6].c23)+(binary[2]&binary[4]&full_matrix[6].c24)+(binary[2]&binary[5]&full_matrix[6].c25)+(binary[3]&bina
ry[4]&full_matrix[6].c34)+(binary[3]&binary[5]&full_matrix[6].c35)+(binary[4]&binary[5]&full_matrix[6].c45);


    eq_rhs7 =
(binary[1]&full_matrix[7].c1)+(binary[2]&full_matrix[7].c2)+(binary[3]&full_matrix[7].c3)+(binary[4]&full_matrix[7]
.c4)+(binary[5]&full_matrix[7].c5)+(binary[1]&binary[2]&full_matrix[7].c12)+(binary[1]&binary[3]&full_matrix[7].c
13)+(binary[1]&binary[4]&full_matrix[7].c14)+(binary[1]&binary[5]&full_matrix[7].c15)+(binary[2]&binary[3]&full_
matrix[7].c23)+(binary[2]&binary[4]&full_matrix[7].c24)+(binary[2]&binary[5]&full_matrix[7].c25)+(binary[3]&bina
ry[4]&full_matrix[7].c34)+(binary[3]&binary[5]&full_matrix[7].c35)+(binary[4]&binary[5]&full_matrix[7].c45);


    eq_rhs8 =
(binary[1]&full_matrix[8].c1)+(binary[2]&full_matrix[8].c2)+(binary[3]&full_matrix[8].c3)+(binary[4]&full_matrix[8]
.c4)+(binary[5]&full_matrix[8].c5)+(binary[1]&binary[2]&full_matrix[8].c12)+(binary[1]&binary[3]&full_matrix[8].c
13)+(binary[1]&binary[4]&full_matrix[8].c14)+(binary[1]&binary[5]&full_matrix[8].c15)+(binary[2]&binary[3]&full_
matrix[8].c23)+(binary[2]&binary[4]&full_matrix[8].c24)+(binary[2]&binary[5]&full_matrix[8].c25)+(binary[3]&bina
ry[4]&full_matrix[8].c34)+(binary[3]&binary[5]&full_matrix[8].c35)+(binary[4]&binary[5]&full_matrix[8].c45);
```

```
    eq_rhs9 =
(binary[1]&full_matrix[9].c1)+(binary[2]&full_matrix[9].c2)+(binary[3]&full_matrix[9].c3)+(binary[4]&full_matrix[9]
.c4)+(binary[5]&full_matrix[9].c5)+(binary[1]&binary[2]&full_matrix[9].c12)+(binary[1]&binary[3]&full_matrix[9].c
13)+(binary[1]&binary[4]&full_matrix[9].c14)+(binary[1]&binary[5]&full_matrix[9].c15)+(binary[2]&binary[3]&full_
matrix[9].c23)+(binary[2]&binary[4]&full_matrix[9].c24)+(binary[2]&binary[5]&full_matrix[9].c25)+(binary[3]&bina
ry[4]&full_matrix[9].c34)+(binary[3]&binary[5]&full_matrix[9].c35)+(binary[4]&binary[5]&full_matrix[9].c45);


    eq_rhs10 =
(binary[1]&full_matrix[10].c1)+(binary[2]&full_matrix[10].c2)+(binary[3]&full_matrix[10].c3)+(binary[4]&full_matri
x[10].c4)+(binary[5]&full_matrix[10].c5)+(binary[1]&binary[2]&full_matrix[10].c12)+(binary[1]&binary[3]&full_mat
rix[10].c13)+(binary[1]&binary[4]&full_matrix[10].c14)+(binary[1]&binary[5]&full_matrix[10].c15)+(binary[2]&bina
ry[3]&full_matrix[10].c23)+(binary[2]&binary[4]&full_matrix[10].c24)+(binary[2]&binary[5]&full_matrix[10].c25)+(
binary[3]&binary[4]&full_matrix[10].c34)+(binary[3]&binary[5]&full_matrix[10].c35)+(binary[4]&binary[5]&full_m
atrix[10].c45);


    eq_rhs11 =
(binary[1]&full_matrix[11].c1)+(binary[2]&full_matrix[11].c2)+(binary[3]&full_matrix[11].c3)+(binary[4]&full_matri
x[11].c4)+(binary[5]&full_matrix[11].c5)+((binary[1]&binary[2])&full_matrix[11].c12)+((binary[1] &
binary[3])&full_matrix[11].c13)+((binary[1]&binary[4])&full_matrix[11].c14)+((binary[1] &
binary[5])&full_matrix[11].c15)+((binary[2]&binary[3])&full_matrix[11].c23)+((binary[2] &
binary[4])&full_matrix[11].c24)+((binary[2]&binary[5])&full_matrix[11].c25)+((binary[3] &
binary[4])&full_matrix[11].c34)+((binary[3]&binary[5])&full_matrix[11].c35)+((binary[4] &
binary[5])&full_matrix[11].c45);


    num_operations++;


    if(((eq_rhs1 == full_matrix[1].rhs) && (eq_rhs2 == full_matrix[2].rhs) && (eq_rhs3 == full_matrix[3].rhs) &&
(eq_rhs4 == full_matrix[4].rhs) && (eq_rhs5 == full_matrix[5].rhs) && (eq_rhs6 == full_matrix[6].rhs) && (eq_rhs7
== full_matrix[7].rhs) && (eq_rhs8 == full_matrix[8].rhs) && (eq_rhs9 == full_matrix[9].rhs) && (eq_rhs10 ==
full_matrix[10].rhs) && (eq_rhs11 == full_matrix[11].rhs)) == 1)


      begin

      $display("Sol =  %b  %b  %b  %b  %b", binary[5], binary[4], binary[3],binary[2], binary[1]);

    end

   //$display("Solution = %b", binary);

    bin_index++;

  end

 end

endmodule
```

Verilog Code for Recursive Search Function

```
`timescale 1ns/1ns

//Create a class which keeps track of an equation and any information tied to it
class equation;
 int eqnum;//Tells us which equation
 //int x0;//values for x0-x7
 int x1;
 int x2;
 int x3;
 int x4;
 int x5;
 int x1x2;
 int x1x3;
 int x1x4;
 int x1x5;
 int x2x3;
 int x2x4;
 int x2x5;
 int x3x4;
 int x3x5;
 int x4x5;
 int rhs;//rhs value for equation
 int linearWeight;//weight of the linear portion of equation (num of 1's excluding rhs)


 //Function within the class to display the sum for a given equation
 function void weight_display();
  $display("\teqnum = %0d, sum = %0d", eqnum, linearWeight);
 endfunction

endclass
```

```
//Create a class which keeps track of a given rows information for the matrix
//This makes it easier to perform row and column adjustments
class info_matrix;
  int eqnum;
  int c1;
  int c2;
  int c3;
  int c4;
  int c5;
  int c12;
  int c13;
  int c14;
  int c15;
  int c23;
  int c24;
  int c25;
  int c34;
  int c35;
  int c45;
  int rhs;


  //Function used to display a matrix row
  function void matrix_display();
    $display("\tE%0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d ", eqnum, c45, c35, c34, c25, c24, c23, c15, c14, c13, c12, c5, c4, c3, c2, c1, rhs);
   // $display("\t---------------------------------------");
  endfunction


endclass
```

```
//Class used to organize matrix based on linear portions only
 class linear_matrix;
 int eqnum;
 int c1;
 int c2;
 int c3;
 int c4;
 int c5;
 int rhs;

  //Function used to display a matrix row
 function void linMatrix_display();
  $display("\tE%0d | %0d | %0d | %0d | %0d | %0d | %0d ", eqnum, c5, c4, c3, c2, c1, rhs);
 // $display("\t---------------------------------------");
 endfunction

 endclass


module recursive_solve;

 equation sum_matrix[1:11];//an array to sort based on eq weights
 equation eq;
 info_matrix full_matrix[1:11];//an array which acts as our matrix (deals with 1 dimension)
 linear_matrix linearFull_matrix[1:11];
 info_matrix i_matrix;
 linear_matrix lin_matrix;
 int t_matrix [1:11][1:15];//Area where temp matrix is stored
 int l_matrix [1:11][1:5]; //Linear matrix values
 int final_matrix [1:11][1:15];//Final matrix is used to reference to the matrix before any row operations
 int t_array [1:11]; //hold temp values
```

```
int l_array [1:11];
int l_value;
int l_rhs;

int t_value;//holds the value for a given variable for swapping
int t_rhs;//holds a temp rhs value when performing row operations
int matrix_max;//Matrix Max is the number of rows - 1 that we are finding partial solutions for
int bin_count;//Used when counting in binary
int bin_index;//Monitors Search FSM Position (aka which row we are operating on)
int num_operations; //Number of times we compare a partial solution against an rhs
int num_fullsol; //Number of times a full solution is checked
int bin_count_array [11:1];//Used to keep track of partial solutions for each equation

reg [5:1] binary = 5'd0;
reg [5:1] t_binary = 5'd0;
reg [5:1] t_solution = 5'd0;
reg eq_rhs = 1'b0;

int t_eqnum;//
int index;//Used for checking tracking right most 0 position
int index_array [1:11];//stores right most 0 position for each row
int pos_array [1:5];//keeps track of where variables sit in our matrix


initial begin
//initialize all of our equations
  eq = new();   // x1 x2 x3 x4 x5 x1x2 x1x3 x1x4 x1x5 x2x3 x2x4 x2x5 x3x4 x3x5 x4x5 rhs
  eq.eqnum = 1; // 0 1 1 0 1 0 1 1 1 0 0 0 0 0 0 1
  eq.x1 = 0;
  eq.x2 = 1;
  eq.x3 = 1;
```

```
eq.x4 = 0;
eq.x5 = 1;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 1;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 3;
sum_matrix[1] = eq;

eq = new();
eq.eqnum = 2; //1 1 1 0 0 0 1 0 1 0 0 1 1 0 1 1
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 1;
eq.x3x5 = 0;
```

```
eq.x4x5 = 1;
eq.rhs = 1; //1
eq.linearWeight = 3;
sum_matrix[2] = eq;

eq = new();
eq.eqnum = 3; // 1 1 0 1 0 0 1 1 1 1 1 1 0 1 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 1;
eq.x1x5 = 1;
eq.x2x3 = 1;
eq.x2x4 = 1;
eq.x2x5 = 1;
eq.x3x4 = 0;
eq.x3x5 = 1;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
sum_matrix[3] = eq;

eq = new();
eq.eqnum = 4; // 1 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0
eq.x1 = 1;
eq.x2 = 0;
eq.x3 = 1;
eq.x4 = 1;
```

```
eq.x5 = 1;
eq.x1x2 = 0;
eq.x1x3 = 0;
eq.x1x4 = 1;
eq.x1x5 = 0;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 1;
eq.x3x5 = 1;
eq.x4x5 = 0;
eq.rhs = 0;
eq.linearWeight = 4;
sum_matrix[4] = eq;

eq = new();
eq.eqnum = 5; // 1 1 1 0 1 1 1 0 1 1 1 0 0 0 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 1;
eq.x1x2 = 1;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 1;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 1;
```

```
eq.rhs = 0;
eq.linearWeight = 4;
sum_matrix[5] = eq;

eq = new();
eq.eqnum = 6; // 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 1;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 0;
eq.x1x4 = 1;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 1;
eq.x4x5 = 0;
eq.rhs = 1;
eq.linearWeight = 2;
sum_matrix[6] = eq;

eq = new();
eq.eqnum = 7; // 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 1
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
```

```
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 1;
eq.x1x5 = 0;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 1;
sum_matrix[7] = eq;

eq = new();
eq.eqnum = 8; // 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0
eq.x1 = 0;
eq.x2 = 0;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 1;
eq.x1x2 = 1;
eq.x1x3 = 0;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 1;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 0;
```

```
eq.linearWeight = 2;
sum_matrix[8] = eq;

eq = new();
eq.eqnum = 9; // 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 1;
eq.x4 = 0;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 1;
eq.x2x4 = 0;
eq.x2x5 = 1;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
sum_matrix[9] = eq;

eq = new();
eq.eqnum = 10; // 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1
eq.x1 = 0;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
```

```
eq.x1x3 = 0;
eq.x1x4 = 0;
eq.x1x5 = 1;
eq.x2x3 = 0;
eq.x2x4 = 0;
eq.x2x5 = 0;
eq.x3x4 = 0;
eq.x3x5 = 0;
eq.x4x5 = 0;
eq.rhs = 1; //1
eq.linearWeight = 2;
sum_matrix[10] = eq;

eq = new();
eq.eqnum = 11; //1 1 0 1 0 0 1 0 0 0 1 0 1 0 1 0
eq.x1 = 1;
eq.x2 = 1;
eq.x3 = 0;
eq.x4 = 1;
eq.x5 = 0;
eq.x1x2 = 0;
eq.x1x3 = 1;
eq.x1x4 = 0;
eq.x1x5 = 0;
eq.x2x3 = 0;
eq.x2x4 = 1;
eq.x2x5 = 0;
eq.x3x4 = 1;
eq.x3x5 = 0;
eq.x4x5 = 1;
eq.rhs = 0;
eq.linearWeight = 3;
```

```
sum_matrix[11] = eq;

//Sorts the equations with the heaviest equations at the top
sum_matrix.sort with (item.linearWeight);

//Assigns values to "cells" of the matrix
foreach(sum_matrix[i])begin
  i_matrix = new();
  i_matrix.eqnum = sum_matrix[i].eqnum;

  i_matrix.c1 = sum_matrix[i].x1;
  i_matrix.c2 = sum_matrix[i].x2;
  i_matrix.c3 = sum_matrix[i].x3;
  i_matrix.c4 = sum_matrix[i].x4;
  i_matrix.c5 = sum_matrix[i].x5;
  i_matrix.c12 = sum_matrix[i].x1x2;
  i_matrix.c13 = sum_matrix[i].x1x3;
  i_matrix.c14 = sum_matrix[i].x1x4;
  i_matrix.c15 = sum_matrix[i].x1x5;
  i_matrix.c23 = sum_matrix[i].x2x3;
  i_matrix.c24 = sum_matrix[i].x2x4;
  i_matrix.c25 = sum_matrix[i].x2x5;
  i_matrix.c34 = sum_matrix[i].x3x4;
  i_matrix.c35 = sum_matrix[i].x3x5;
  i_matrix.c45 = sum_matrix[i].x4x5;
  i_matrix.rhs = sum_matrix[i].rhs;
  //fills up our temp 2d array with initial values

  t_matrix[i][1] = sum_matrix[i].x1;
  t_matrix[i][2] = sum_matrix[i].x2;
  t_matrix[i][3] = sum_matrix[i].x3;
  t_matrix[i][4] = sum_matrix[i].x4;
```

```
  t_matrix[i][5] = sum_matrix[i].x5;
  t_matrix[i][6] = sum_matrix[i].x1x2;
  t_matrix[i][7] = sum_matrix[i].x1x3;
  t_matrix[i][8] = sum_matrix[i].x1x4;
  t_matrix[i][9] = sum_matrix[i].x1x5;
  t_matrix[i][10] = sum_matrix[i].x2x3;
  t_matrix[i][11] = sum_matrix[i].x2x4;
  t_matrix[i][12] = sum_matrix[i].x2x5;
  t_matrix[i][13] = sum_matrix[i].x3x4;
  t_matrix[i][14] = sum_matrix[i].x3x5;
  t_matrix[i][15] = sum_matrix[i].x4x5;
  full_matrix[i] = i_matrix;

  lin_matrix = new();
  lin_matrix.eqnum = sum_matrix[i].eqnum;
  lin_matrix.c1 = sum_matrix[i].x1;
  lin_matrix.c2 = sum_matrix[i].x2;
  lin_matrix.c3 = sum_matrix[i].x3;
  lin_matrix.c4 = sum_matrix[i].x4;
  lin_matrix.c5 = sum_matrix[i].x5;
  lin_matrix.rhs = sum_matrix[i].rhs;

  l_matrix[i][1] = sum_matrix[i].x1;
  l_matrix[i][2] = sum_matrix[i].x2;
  l_matrix[i][3] = sum_matrix[i].x3;
  l_matrix[i][4] = sum_matrix[i].x4;
  l_matrix[i][5] = sum_matrix[i].x5;

  linearFull_matrix[i] = lin_matrix;
end

foreach(full_matrix[i])begin
```

```
 full_matrix[i].c1 = t_matrix[i][1];
 full_matrix[i].c2 = t_matrix[i][2];
 full_matrix[i].c3 = t_matrix[i][3];
 full_matrix[i].c4 = t_matrix[i][4];
 full_matrix[i].c5 = t_matrix[i][5];
 full_matrix[i].c12 = t_matrix[i][6];
 full_matrix[i].c13 = t_matrix[i][7];
 full_matrix[i].c14 = t_matrix[i][8];
 full_matrix[i].c15 = t_matrix[i][9];
 full_matrix[i].c23 = t_matrix[i][10];
 full_matrix[i].c24 = t_matrix[i][11];
 full_matrix[i].c25 = t_matrix[i][12];
 full_matrix[i].c34 = t_matrix[i][13];
 full_matrix[i].c35 = t_matrix[i][14];
 full_matrix[i].c45 = t_matrix[i][15];
end

$display("\t   | x4x5| x3x5| x3x4| x2x5| x2x4| x2x3| x1x5| x1x4| x1x3| x1x2| x5| x4| x3| x2| x1| rhs");
foreach(full_matrix[i])begin
 full_matrix[i].matrix_display();

end
$display("=================================================");

//Copy Matrix Values to Final Matrix for future referencing
for(int i = 1; i <= 11; i++) begin
 for(int j = 1; j <= 5; j++) begin //15
   final_matrix[i][j] = l_matrix[i][j]; //t_matrix
 end
end

//stores info as of what variable is in what column
```

```
foreach(pos_array[i])begin
  pos_array[i] = i;
end

foreach(linearFull_matrix[i])begin
  linearFull_matrix[i].c1 = l_matrix[i][1];
  linearFull_matrix[i].c2 = l_matrix[i][2];
  linearFull_matrix[i].c3 = l_matrix[i][3];
  linearFull_matrix[i].c4 = l_matrix[i][4];
  linearFull_matrix[i].c5 = l_matrix[i][5];
end



/*$display("\t   | x4x5| x3x5| x3x4| x2x5| x2x4| x2x3| x1x5| x1x4| x1x3| x1x2| x5| x4| x3| x2| x1| rhs");
//$display("\t----------------------------------------");
foreach(full_matrix[i])begin
  full_matrix[i].matrix_display();

end   */

$display("\t   | x5| x4| x3| x2| x1| rhs");
//$display("\t----------------------------------------");
foreach(linearFull_matrix[i])begin
  linearFull_matrix[i].linMatrix_display();

end
$display("================================================");


//Placing ones on right hand side of matrix (Should only focus on linear portion for now)
```

```
            index = 1; //we want to maintain the index throughout the rows, so we only set it to 0 beforehand
  //here max num j == 5
   foreach(linearFull_matrix[j])begin
     foreach(l_matrix[j][i])begin //for each cell in the matrix
       if(i>=index)begin//as long as i is beyond the index position (rightmost 0 pos)
        if(l_matrix[j][i] == 1)begin//if we find a 1, we swap the entire column of the rightmost 0 and the current 1 column

          l_array[j] = l_matrix[j][index];
          l_array[j+1] = l_matrix[j+1][index];
          l_array[j+2] = l_matrix[j+2][index];
          l_array[j+3] = l_matrix[j+3][index];
          l_array[j+4] = l_matrix[j+4][index];

          l_value = pos_array[index];

          l_matrix[j][index] = l_matrix[j][i];
          l_matrix[j+1][index] = l_matrix[j+1][i];
          l_matrix[j+2][index] = l_matrix[j+2][i];
          l_matrix[j+3][index] = l_matrix[j+3][i];
          l_matrix[j+4][index] = l_matrix[j+4][i];

          pos_array[index] = pos_array[i];

          l_matrix[j][i] = l_array[j];
          l_matrix[j+1][i] = l_array[j+1];
          l_matrix[j+2][i] = l_array[j+2];
          l_matrix[j+3][i] = l_array[j+3];
          l_matrix[j+4][i] = l_array[j+4];

          pos_array[i] = l_value;//updates the reference for which variables are in which column
        // $display("row = %0d ,index = %0d, i = %0d", j,index, i);
```

```
        index++;
      end
      end

    end
    index_array[j] = index;
    //$display("index = %0d", index_array[j]);
  end

 // foreach(pos_array[i])begin
   // $display("pos %0d = x%0d", i, pos_array[i]);
  // end


  //Updates the new matrix column positons ie [i][[pos_array[0]] after grouping 1s
  foreach(linearFull_matrix[i])begin
    //full_matrix[i].c0 = final_matrix[i][pos_array[0]];
    linearFull_matrix[i].c1 = final_matrix[i][pos_array[1]];
    linearFull_matrix[i].c2 = final_matrix[i][pos_array[2]];
    linearFull_matrix[i].c3 = final_matrix[i][pos_array[3]];
    linearFull_matrix[i].c4 = final_matrix[i][pos_array[4]];
    linearFull_matrix[i].c5 = final_matrix[i][pos_array[5]];
  end

  $display("\t   | x%0d| x%0d| x%0d| x%0d| x%0d|rhs", pos_array[5], pos_array[4], pos_array[3], pos_array[2],
pos_array[1]);

  linearFull_matrix[1].linMatrix_display();
  linearFull_matrix[2].linMatrix_display();
  linearFull_matrix[3].linMatrix_display();
  linearFull_matrix[4].linMatrix_display();
  linearFull_matrix[5].linMatrix_display();
```

```
linearFull_matrix[6].linMatrix_display();
linearFull_matrix[7].linMatrix_display();
linearFull_matrix[8].linMatrix_display();
linearFull_matrix[9].linMatrix_display();
linearFull_matrix[10].linMatrix_display();
linearFull_matrix[11].linMatrix_display();

$display("================================================");
        bin_count = 0;

foreach(full_matrix[i])begin
  //full_matrix[i].c0 = final_matrix[i][pos_array[0]];
  full_matrix[i].c1 = final_matrix[i][pos_array[1]];
  full_matrix[i].c2 = final_matrix[i][pos_array[2]];
  full_matrix[i].c3 = final_matrix[i][pos_array[3]];
  full_matrix[i].c4 = final_matrix[i][pos_array[4]];
  full_matrix[i].c5 = final_matrix[i][pos_array[5]];
  full_matrix[i].c12 = t_matrix[i][6];
  full_matrix[i].c13 = t_matrix[i][7];
  full_matrix[i].c14 = t_matrix[i][8];
  full_matrix[i].c15 = t_matrix[i][9];
  full_matrix[i].c23 = t_matrix[i][10];
  full_matrix[i].c24 = t_matrix[i][11];
  full_matrix[i].c25 = t_matrix[i][12];
  full_matrix[i].c34 = t_matrix[i][13];
  full_matrix[i].c35 = t_matrix[i][14];
  full_matrix[i].c45 = t_matrix[i][15];
end

//Displays the matrix when fully organzed
$display("\t  | x4x5| x3x5| x3x4| x2x5| x2x4| x2x3| x1x5| x1x4| x1x3| x1x2| x%0d| x%0d| x%0d| x%0d|
x%0d|rhs", pos_array[5], pos_array[4], pos_array[3], pos_array[2], pos_array[1]);
```

```
  full_matrix[1].matrix_display();
  full_matrix[2].matrix_display();
  full_matrix[3].matrix_display();
  full_matrix[4].matrix_display();
  full_matrix[5].matrix_display();
  full_matrix[6].matrix_display();
  full_matrix[7].matrix_display();
  full_matrix[8].matrix_display();
  full_matrix[9].matrix_display();
  full_matrix[10].matrix_display();
  full_matrix[11].matrix_display();

  $display("================================================");

  //Fills the array bin_count_array with a corresponding values as of the max size for a partial solution
  foreach(bin_count_array[i])begin
    if((index_array[i]-index_array[i-1])==1)begin
      matrix_max = i-1;
      $display("matrix_max = %0d", matrix_max);
    end
    bin_count_array[i] = 0;
  end
end


//Starts the search portion of recursive solve.
initial begin
  bin_count = 1;
  num_operations = 0;
  num_fullsol = 0;
  bin_index = 1;
```

```
$display("\tx%0d|x%0d|x%0d|x%0d|x%0d|x%0d|x%0d", pos_array[5], pos_array[4], pos_array[3], pos_array[2], pos_array[1]);

   //begin our FSM with exit state being when first row is fully exhausted

   while(binary <= ((2**index_array[1])-1))begin



    // function void matrix_display();

  // $display("\tE%0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d | %0d ", eqnum, c45, c35, c34, c25, c24, c23, c15, c14, c13, c12, c5, c4, c3, c2, c1, rhs);

  // $display("\t----------------------------------------");

 // endfunction



   /*  function void zero();

      if (pos_array[1] == full_matrix[i].c1) begin

        full_matrix[i].c1 = 0

        full_matrix[i].c12 = 0

        full_matrix[i].c13 = 0

        full_matrix[i].c14 = 0

        full_matrix[i].c15 = 0

      end else if (pos_array[1] == full_matrix[i].c2) begin

        full_matrix[i].c2 = 0

        full_matrix[i].c12 = 0

        full_matrix[i].c23 = 0

        full_matrix[i].c24 = 0

        full_matrix[i].c25 = 0

      end else if (pos_array[1] == full_matrix[i].c3) begin

        full_matrix[i].c3 = 0

        full_matrix[i].c13 = 0

        full_matrix[i].c23 = 0

        full_matrix[i].c34 = 0
```

```
    full_matrix[i].c35 = 0
   end else if (pos_array[1] == full_matrix[i].c4) begin
    full_matrix[i].c4 = 0
    full_matrix[i].c14 = 0
    full_matrix[i].c24 = 0
    full_matrix[i].c34 = 0
    full_matrix[i].c45 = 0
   end
   end
 endfunction */

/* function one();
   if pos_array[1] = full_matrix[i].c1 begin

    else if pos_array[1] = full_matrix[i].c2;
    full_matrix[i].c2 = 0;
    full_matrix[i].c12 = 0;
    full_matrix[i].c23 = 0;
    full_matrix[i].c24 = 0;
    full_matrix[i].c25 = 0;
    else if pos_array[1] = full_matrix[i].c3;
    full_matrix[i].c3 = 0;
    full_matrix[i].c23 = 0;
    full_matrix[i].c34 = 0;
    full_matrix[i].c35 = 0;
    else if pos_array[1] = full_matrix[i].c4;
    full_matrix[i].c4 = 0;
    full_matrix[i].c34 = 0;
    full_matrix[i].c45 = 0;
   end
 endfunction */
```

```
    if(bin_index == 0 && binary <= ((2**index_array[1])-1))begin//Checking first row partial solution as long as it is not completely exaughsted

      num_operations++;

      binary = bin_count_array[1];//Binary is our binary representation of our Binary counter

      if(binary > ((2**index_array[1])-1))begin //If we reach limit for partial solution, break the search

        break;

      end

      //$display("bintest %0b", binary[0]);

      //$display("eq_rhs = %0b : rhs = %0b", eq_rhs, full_matrix[0].rhs);

      //$display("Index = %0d, Value = %3b " , 0, binary);


      //t_solution[0] = binary[0];//stores the binary counter value into our partial solution for testing

      t_solution[1] = binary[1];

      t_solution[2] = binary[2];

      //determines the value of the left side of the matrix by masking the partial solution with the matrix values and summing them together

      eq_rhs = (full_matrix[bin_index].c1 & t_solution[1]) + (full_matrix[bin_index].c2 & t_solution[2]) + (full_matrix[bin_index].c3 & t_solution[3]) + (full_matrix[bin_index].c4 & t_solution[4]) + (full_matrix[bin_index].c5 & t_solution[5]) + (full_matrix[bin_index].c12 & t_solution[1] & t_solution[2]) + (full_matrix[bin_index].c13 & t_solution[1] & t_solution[3]) + (full_matrix[bin_index].c14 & t_solution[1] & t_solution[4]) + (full_matrix[bin_index].c15 & t_solution[1] & t_solution[5]) + (full_matrix[bin_index].c23 & t_solution[2] & t_solution[3]) + (full_matrix[bin_index].c24 & t_solution[2] & t_solution[4]) + (full_matrix[bin_index].c25 & t_solution[2] & t_solution[5]) + (full_matrix[bin_index].c34 & t_solution[3] & t_solution[4]) + (full_matrix[bin_index].c35 & t_solution[3] & t_solution[5]) + (full_matrix[bin_index].c45 & t_solution[4] & t_solution[5]);

      if(eq_rhs == full_matrix[bin_index].rhs)begin//compares Left hand size sum to right hand side value

        //$display("Solution = %b", t_solution);

        bin_index = 2; //Tells FSM to move onto next row

       end

      bin_count_array[1] = bin_count_array[1] + 1;//Increments our partial solution


    end

    if(bin_index == 1 )begin

                    num_operations++;

      //$display("eq_rhs = %0b : rhs = %0b", eq_rhs, full_matrix[0].rhs);
```

```
      if(bin_count_array[1] == (2**(index_array[2]-index_array[2-1])))begin //if weve seen every possible soultion for this partial solution, reset partial solution incrementation and return to previous row
        bin_count_array[1] = 0;
        bin_index = 1;
      end

      if(bin_index != 1)begin
        //$display("Index = %0d, Value = \t%2b " , 1, bin_count_array[1]);

        t_binary = bin_count_array[1];
        t_solution[3] = t_binary[1];//stores the binary counter value into our partial solution for testing
        t_solution[4] = t_binary[2];
        //determines the value of the left side of the matrix by masking the partial solution with the matrix values and summing them together
        eq_rhs = (full_matrix[bin_index].c1 & t_solution[1]) + (full_matrix[bin_index].c2 & t_solution[2]) + (full_matrix[bin_index].c3 & t_solution[3]) + (full_matrix[bin_index].c4 & t_solution[4]) + (full_matrix[bin_index].c5 & t_solution[5]) + (full_matrix[bin_index].c12 & t_solution[1] & t_solution[2]) + (full_matrix[bin_index].c13 & t_solution[1] & t_solution[3]) + (full_matrix[bin_index].c14 & t_solution[1] & t_solution[4]) + (full_matrix[bin_index].c15 & t_solution[1] & t_solution[5]) + (full_matrix[bin_index].c23 & t_solution[2] & t_solution[3]) + (full_matrix[bin_index].c24 & t_solution[2] & t_solution[4]) + (full_matrix[bin_index].c25 & t_solution[2] & t_solution[5]) + (full_matrix[bin_index].c34 & t_solution[3] & t_solution[4]) + (full_matrix[bin_index].c35 & t_solution[3] & t_solution[5]) + (full_matrix[bin_index].c45 & t_solution[4] & t_solution[5]);
        if(eq_rhs == full_matrix[bin_index].rhs)begin//compares Left hand size sum to right hand side value
          //$display("Solution = %b", t_solution);
          bin_index = 3;//Tells FSM to move onto next row
        end
        bin_count_array[1] = bin_count_array[1] + 1;//Increments our partial solution

      end



    end
    if(bin_index == 3)begin
      num_operations++;
```

```
      if(bin_count_array[2] == (2**(index_array[3]-index_array[3-1])))begin//if weve seen every possible soultion for this partial solution, reset partial solution incrementation and return to previous row
        bin_count_array[2] = 0;
        bin_index = 2;
      end


      if(bin_index != 2)begin
        //$display("Index = %0d, Value = \t\t%2b " , 2, bin_count_array[2]);
        t_binary = bin_count_array[2];
        t_solution[5] = t_binary[1];//stores the binary counter value into our partial solution for testing

        //determines the value of the left side of the matrix by masking the partial solution with the matrix values and summing them together
        eq_rhs = (full_matrix[bin_index].c1 & t_solution[1]) + (full_matrix[bin_index].c2 & t_solution[2]) + (full_matrix[bin_index].c3 & t_solution[3]) + (full_matrix[bin_index].c4 & t_solution[4]) + (full_matrix[bin_index].c5 & t_solution[5]) + (full_matrix[bin_index].c12 & t_solution[1] & t_solution[2]) + (full_matrix[bin_index].c13 & t_solution[1] & t_solution[3]) + (full_matrix[bin_index].c14 & t_solution[1] & t_solution[4]) + (full_matrix[bin_index].c15 & t_solution[1] & t_solution[5]) + (full_matrix[bin_index].c23 & t_solution[2] & t_solution[3]) + (full_matrix[bin_index].c24 & t_solution[2] & t_solution[4]) + (full_matrix[bin_index].c25 & t_solution[2] & t_solution[5]) + (full_matrix[bin_index].c34 & t_solution[3] & t_solution[4]) + (full_matrix[bin_index].c35 & t_solution[3] & t_solution[5]) + (full_matrix[bin_index].c45 & t_solution[4] & t_solution[5]);
        if(eq_rhs == full_matrix[bin_index].rhs)begin//compares Left hand size sum to right hand side value
          //$display("Solution = %b", t_solution);
          bin_index = 4;//Tells FSM to move onto next row
        end
        bin_count_array[2] = bin_count_array[2] + 1;//Increments our partial solution

      end

    end

    if(bin_index == 4)begin
      num_operations++;
```

```
    if(bin_count_array[3] == (2**(index_array[4]-index_array[4-1])))begin//if weve seen every possible soultion for this partial solution, reset partial solution incrementation and return to previous row

      bin_count_array[3] = 0;

      bin_index = 3;

    end



    if(bin_index != 4 )begin

      //$display("Index = %0d, Value = \t\t\t%1b " , 3, bin_count_array[3]);

      t_binary = bin_count_array[3];


      //determines the value of the left side of the matrix by masking the partial solution with the matrix values and summing them together

      eq_rhs = (full_matrix[bin_index].c1 & t_solution[1]) + (full_matrix[bin_index].c2 & t_solution[2]) + (full_matrix[bin_index].c3 & t_solution[3]) + (full_matrix[bin_index].c4 & t_solution[4]) + (full_matrix[bin_index].c5 & t_solution[5]) + (full_matrix[bin_index].c12 & t_solution[1] & t_solution[2]) + (full_matrix[bin_index].c13 & t_solution[1] & t_solution[3]) + (full_matrix[bin_index].c14 & t_solution[1] & t_solution[4]) + (full_matrix[bin_index].c15 & t_solution[1] & t_solution[5]) + (full_matrix[bin_index].c23 & t_solution[2] & t_solution[3]) + (full_matrix[bin_index].c24 & t_solution[2] & t_solution[4]) + (full_matrix[bin_index].c25 & t_solution[2] & t_solution[5]) + (full_matrix[bin_index].c34 & t_solution[3] & t_solution[4]) + (full_matrix[bin_index].c35 & t_solution[3] & t_solution[5]) + (full_matrix[bin_index].c45 & t_solution[4] & t_solution[5]);


      if(eq_rhs == full_matrix[bin_index].rhs)begin//compares Left hand size sum to right hand side value

        num_fullsol++;//increments number of full solutions which have been tested against rhs and remaining equations

        //determines the value of the left side of the matrix by masking the full solution with the matrix values and summing them together

        eq_rhs = (full_matrix[bin_index].c1 & t_solution[1]) + (full_matrix[bin_index].c2 & t_solution[2]) + (full_matrix[bin_index].c3 & t_solution[3]) + (full_matrix[bin_index].c4 & t_solution[4]) + (full_matrix[bin_index].c5 & t_solution[5]) + (full_matrix[bin_index].c12 & t_solution[1] & t_solution[2]) + (full_matrix[bin_index].c13 & t_solution[1] & t_solution[3]) + (full_matrix[bin_index].c14 & t_solution[1] & t_solution[4]) + (full_matrix[bin_index].c15 & t_solution[1] & t_solution[5]) + (full_matrix[bin_index].c23 & t_solution[2] & t_solution[3]) + (full_matrix[bin_index].c24 & t_solution[2] & t_solution[4]) + (full_matrix[bin_index].c25 & t_solution[2] & t_solution[5]) + (full_matrix[bin_index].c34 & t_solution[3] & t_solution[4]) + (full_matrix[bin_index].c35 & t_solution[3] & t_solution[5]) + (full_matrix[bin_index].c45 & t_solution[4] & t_solution[5]);

        if(eq_rhs == full_matrix[bin_index+1].rhs)begin//if the masking and then sum of the full solution and last row values equals the rhs of last eq

          num_operations++;
```

```
          //display our valid solution
           $display("Sol =  %b  %b  %b  %b  %b",t_solution[5], t_solution[4], t_solution[3],t_solution[2], t_solution[1]);
         end
       end
       //increment partial solution
                    bin_count_array[3] = bin_count_array[3] + 1;
     end

   end

  end

  $display("Number Of Operations = %d", num_operations);
  $display("Number Of Full Solution Checks = %d", num_fullsol);

  end

endmodule
```