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

Date: 12/15/2023
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. The goal of this part was to implement an exhaustive search method for a binary quadratic system of equations as a proof of concept, as well as creating a lookup table to store the solutions to two binary quadratic equations that were similar to each other besides two variables. For a lookup-table, recursion-based approach for equations that were similar to each other, a table was created through simulations created using the hardware description languages Verilog and SystemVerilog to find solutions for those similar equations that were differing in a combination of two variables, with different combinations of linear and quadratic terms accounted for. For the future verification of produced solutions via an improved recursive search, a low-resource utilization method of exhaustive searching has been developed and implemented on upgraded hardware that can be scaled up both in terms of linear terms present, as well as the number of equations, with a method of displaying via a terminal being developed in the future. All of the code from the Vivado project can be found in the Appendix.

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.

$$X1 + X2 + X3 + X4 + X3X2 + X3X5 = 1$$
$$X2 + X3 + X5 + X1X4 + X4X5 = 0$$
$$X1 + X4 + X5 + X3X4 + X3X5 + X4X5 = 0$$

Figure 1: An Example System of Binary Quadratic Equations

The MQ problem itself is stated to be used in regard to post-quantum cryptography as "the building block of the Multivariate public key cryptosystems (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):

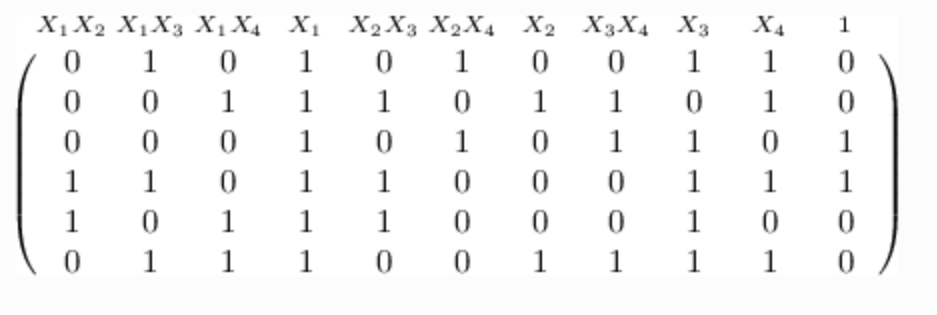
1. Size of Finite Field (usually 2)
2. Number of variables
3. Number of polynomials

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 through the use of 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.



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
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

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 chicking the solutions with the equations that contain X1 (Joux 2018).

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
1	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	1	0	1	1
0	0	1	0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	1	1	0	1
0	0	0	0	1	0	0	1	1	1	1
0	0	0	0	0	1	1	0	0	0	0

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 of 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.

$$
\begin{array}{cccccccccc}
X_1X_2 & X_1X_3 & X_2X_3 & X_1X_4 & X_2X_4 & X_3X_4 & X_1 & X_2 & X_3 & X_4 & 1 \\
\left(\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1
\end{array}\right)
\end{array}
$$

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 can not be both 1 and 0 at the same time, all literals can not be 0, and rules to resolve situations where a variable can not be either 0 or 1. These decisions end up consuming a lot of memory to operate.

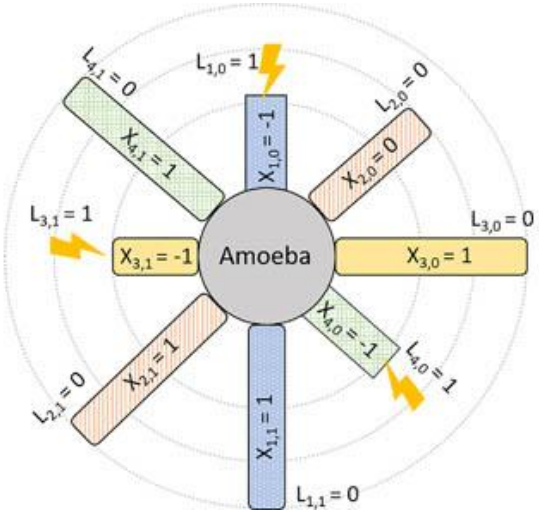


Figure 5: AmoebaSAT model for 4 Variable SAT Case (Nguyen, et al. 2020)

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 bounceback 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). Kenndey'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.

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 variables, I 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 7: Equation Format for Weight of 2 Solving with Example Coefficients

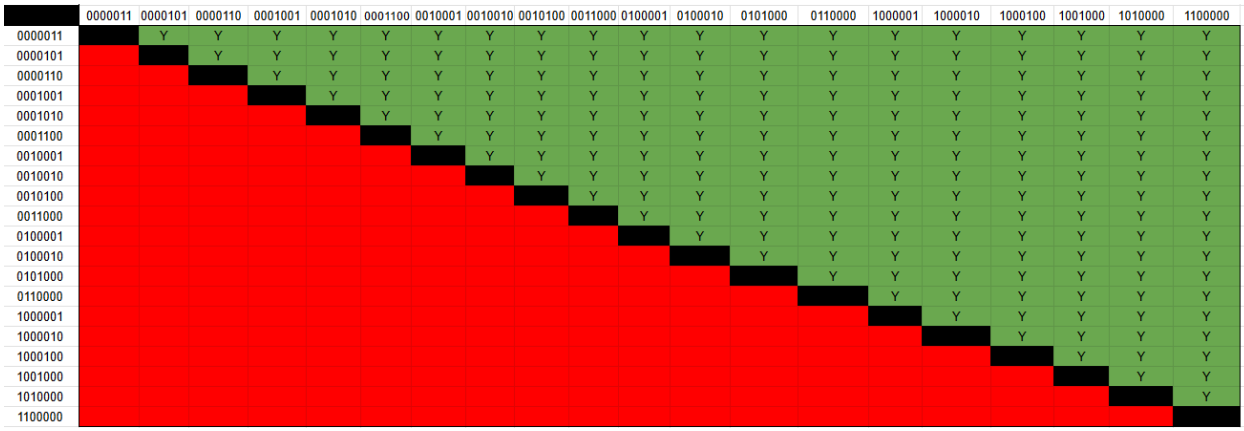
Using this specific set of terms, I am able to cover all of these five different scenarios:

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)



	0000011	0000101	0000110	0001001	0001010	0001100	0010001	0010010	0010100	0011000	0100001	0100010	0101000	0110000	1000001	1000010	1000100	1001000	1010000	1100000
0000011		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0000101			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0000110				Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0001001					Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0001010						Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0001100							Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0010001								Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0010010									Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0010100										Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0011000											Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
0100001												Y	Y	Y	Y	Y	Y	Y	Y	Y
0100010													Y	Y	Y	Y	Y	Y	Y	Y
0101000														Y	Y	Y	Y	Y	Y	Y
0110000															Y	Y	Y	Y	Y	Y
1000001																Y	Y	Y	Y	Y
1000010																	Y	Y	Y	Y
1000100																		Y	Y	Y
1001000																			Y	Y
1010000																				Y
1100000																				

Figure 8: 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 9: 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), I 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 10: 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. Although the board has been ordered, it has not been delivered or tested at the time of this report.

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, I was 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 regards 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 11: 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.

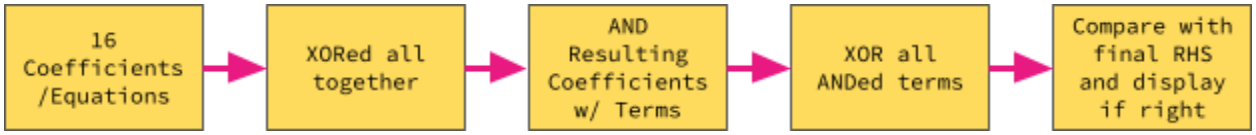


Figure 12: Exhaustive Searching Flow Diagram

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.

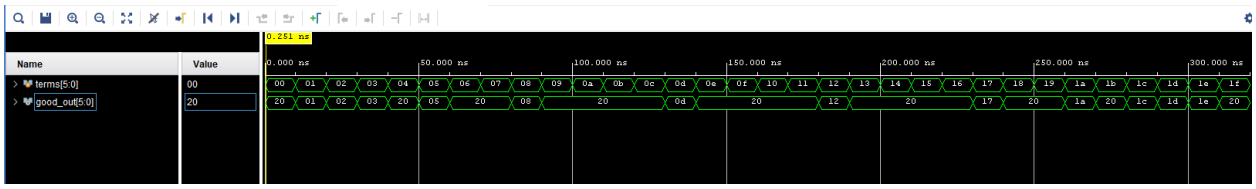


Figure 13: Exhaustive Search Vivado Simulation Results

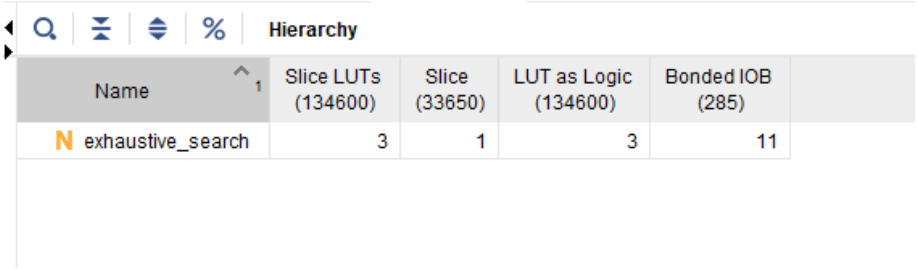


Figure 14: Exhaustive Search Resource Utilization

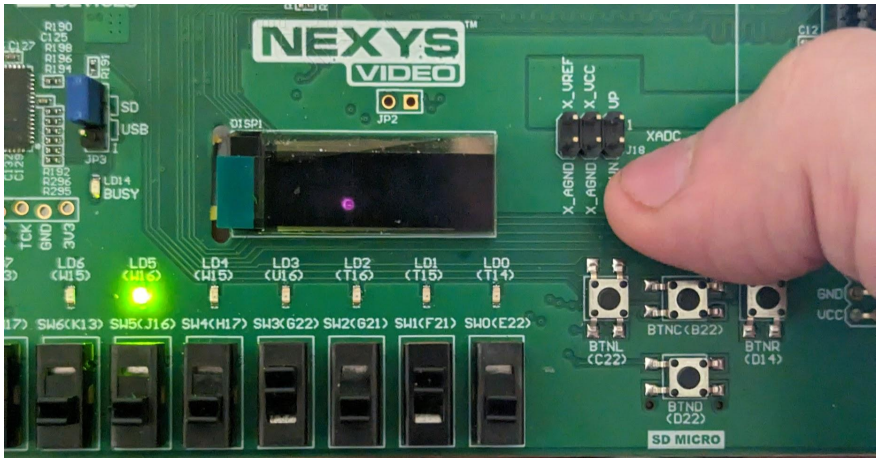


Figure 15: X2 and X4 = 1 on Board

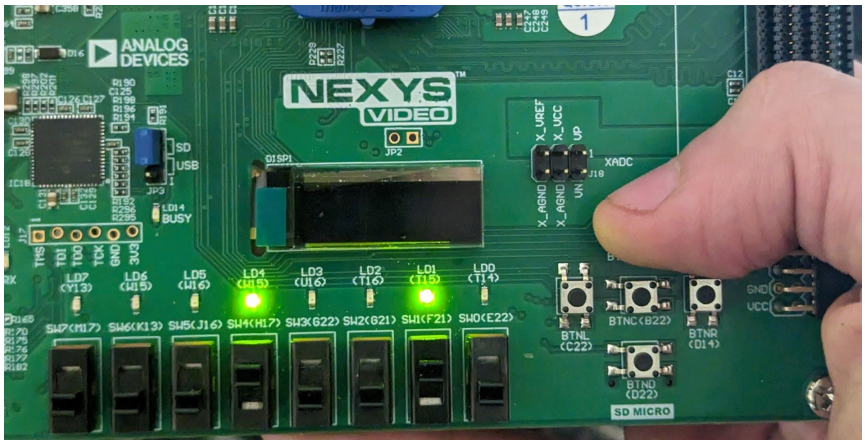


Figure 16: X2 and X5 = 1 on Board

Conclusion

During my 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. I was able to aid the team in implementing an exhaustive search algorithm on a system of 16 equations with five linear terms and ten quadratic terms, with progress starting on being able to use an upgraded board, containing a magnitude of more computational resources and memory, to report what solutions are satisfiable in a terminal. Using a system of three linear terms and three quadratic terms, it was possible to generate a table of 1,521 situations of two equations with a difference of two terms excluding the RHS and Rest terms to aid in a possible lookup table-based solution to recursive searching. I was also able to parametrize the weight-finding code and create the lookup table for a theoretical search for two equations with a difference of three variables. I believe that the improvements that were made on the foundation started by Frank Kennedy and the implementations on the new FPGA as well as the PYNQ SoC board will lay excellent groundwork for future research in this field.

Acknowledgements

Firstly I would like to thank my advisor Professor Köksal Mus and the rest of my team: Samuel David, Coco Mao, Andrew Gray, Patrick Hunter, and Joshua Eben for their extensive support and assistance in completing this much of the project. Professor Mus's knowledge was instrumental to my success and the combined efforts of the team were able to help me to achieve the goals that I wanted to get done during my time on the project, while also leaving room for improvement and further development for the future. Further gratitude is extended to Worcester Polytechnic Institute as well as the entirety of the Electrical and Computer Engineering department and faculty for providing me the opportunity to participate, along with the cherished experiences and knowledge I gained during my undergraduate studies. Finally, I would also like to thank Frank Kennedy for the extensive research and work on a starting point for solving quadratic systems, as well as worthwhile recommendations for improvement that were taken into consideration.

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.
- 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

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
```