Frankenstein and Frankenstein's Monster: 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
Coco Mao, Andrew Gray, Patrick Hunter, Joshua Eben, Matthew Lund, Samuel David

Date: 4/25/2024
Project Advisor: Dr. Köksal Muş

Table of Contents

Table of Figures

Abstract

The goal of this project was to solve the Boolean Satisfiability Problem (SAT) by implementing an exhaustive search method for a binary quadratic system of equations on FPGA hardware. We first used Python to construct a proof-of-concept CDCL solver for quadratic Boolean systems, which we named the Frankenstein algorithm. Additionally, we drafted look up tables (LUTs) that analyzed two equations and defined variables based on one, two, or three term differences between the pair of equations. These LUTs were created because for these equations, the terms that differ have a limited number of solutions, so some variables can be defined early. We also created an algorithm that determines the decision order for variables based on their frequency and the current state of the solver. After that, the team worked on converting the Python code to Verilog. Our group successfully implemented the LUTs and some of Frankenstein in Verilog and System Verilog, with one additional module created in VHDL. Although SAT is a continuous research problem, our group created a solid foundation that leaves plenty of room for future groups to expand on.

Introduction

The Boolean Satisfiability Problem (SAT) is a popular problem in cryptography. The SAT problem determines if a Boolean system can be satisfied by finding a combination of assignments for the variables that makes the entire system true. Additionally, the Multivariate Quadratic (MQ) problem builds on the SAT problem by introducing quadratic terms to Boolean systems. This project views Boolean systems as a quadratic system of equations, where every equation must be satisfied for the system to be satisfied. An example of our Boolean system interpretation is shown in Figure 1.

$$X_1 + X_2 + X_3 + X_1X_2 = 1$$
$$X_1 + X_2 + X_1X_2 + X_2X_3 = 1$$
$$X_2 + X_1X_2 + X_2X_3 + X_1X_3 = 0$$

Figure 1: Example Boolean system with three variables

The purpose of our MQP is to prepare for post-quantum cryptography by researching and improving current multivariate public key crypto systems, such as Rainbow and LUOV. In a public key, the size of the finite field, number of variables, and number of polynomials determine the hardness of the MQ problem given, as well as the specific time and space complexities.

There have been numerous attempts at solving the MQ problem as well as the SAT problem, such as the Crossbred algorithm, the AmoebaSAT algorithm, as well as algorithms created by previous iterations of this project, such as the one made by Frank Kennedy. Frank's goal was to solve linear systems using an efficient exhaustive search. The goal of this project is to improve 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 was developed by French cryptographers Antoine Joux and Vanessa Vitse, and it focused on solving systems of quadratic binary polynomials using Macaulay matrices, much like the FXL/BooleanSolve algorithm it is based on (Joux 2018). A Macaulay matrix illustrates the presence of each linear and quadratic term in the Boolean system.

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

Figure 2: An example degree two Macaulay matrix (Joux 2018)

The main problem with a lot of solving algorithms that involve the use of this matrix is the number of calculations performed. For one Boolean system, an algorithm may perform up to 2^{n-k} calculations, 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 were able to mitigate this issue by eliminating known variables as they are discovered.

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

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

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

However, it is not necessary to eliminate all the known 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 order, with all quadratic terms first before the linear terms, creating a row echelon form of the matrix shown in Figure 4. From the last 3 rows, we see that all the equations have X1, X2, and X3 in degree 1. This allows us to assign X4 to whatever we want and solve for the other variables, theoretically eliminating them from the search.

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

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 in Internet of Things applications, the algorithm is based on how an amoeba can grow and move from light signals called "Bounceback signals" (Ngyuen, et al. 2020). These signals establish a set of rules, those being that each variable cannot be both 1 and 0 at the same time, all literals cannot be 0, and other rules to resolve situations where a variable cannot be either 0 or 1. These decisions end up consuming a lot of memory to operate.

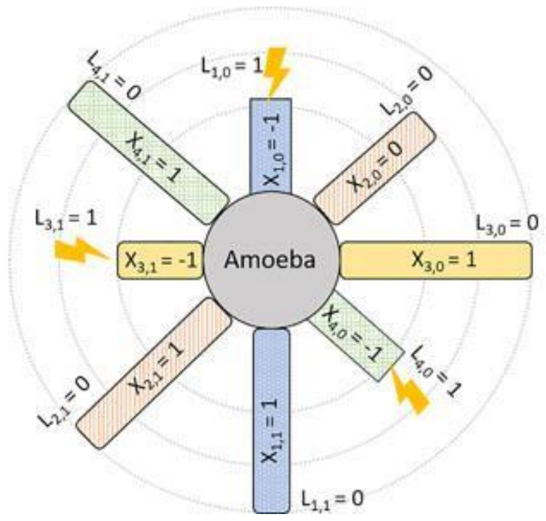


Figure 5: AmoebaSAT model for 4 Variable SAT Case (Nguyen, et al. 2020)

An iteration of this algorithm known as AmoebaSATslim (ASATslim) reduces the amount of memory it uses by omitting certain rules from the bounce back signals and instead implements them as temporary signals on a branch-by-branch basis. Although it will need nearly the same number of iterations as the original AmoebaSAT algorithm, due to memory issues being mitigated to a degree, ASATslim is capable of handling more iterations compared to AmoebaSAT. An evolution of this algorithm, ASATone, further reduces the computational resources needed by representing variables as single branches.

This version of the ASAT algorithm was able to implement *h* copies of the uf50-100.cnf 3-SAT instance, a set of 50*h* variables and 218*h* that only have one solution on a Zynq Ultrascale+ FPGA. When compared to a software implementation using a Ryzen 3960X 24 Core CPU, the FPGA was anywhere between 3 and 15 times faster while using less power, running under ten watts. Parallelization was important to the FPGA's speed, with multiple instances of ASATone running at once.

Frank Kennedy's Implementation

Frank Kennedy's implementation was built off previous work from WPI students Liam Stearns, Carlton Mugo, and James McAleese, groups from the past two years. The main idea of Frank's recursive algorithm was to split the system into smaller groups. Essentially, each subgroup is able to be solved independently of every other group. This decreases the number of solutions to 2^{n-s}, where n is the total number of variables, and s is equivalent to the number of groups. Kennedy created these groups by finding partial solutions to the linear terms of the equations. Kennedy started his improvements by observing when two different equations differed only in one variable. Frank then found what the solution of that variable would be when the right hand side, presence of the differing variable, and the anticipated sum of every other variable in the equation were manipulated for each of the two equations.

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 assigns a weight, the number of linear variables present, to each equation. After weighing each equation, the equations are then sorted from lowest 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 column, removing that column from other reorganizations. The reorganization process repeats until an upper right-hand triangle is formed, which creates a decision order on how to assign the variables. This first variable can be set to 0 first, and then check every equation to see if it they are unsatisfied. If no equations are unsatisfied after the last variable is decided, then the solution is considered SAT. If an equation is unsatisfied, then the first variable to be set to 0 in the current variable assignments is set to 1 and the process starts from that assignment. If no solution is found after a complete exhaustive search, then the system is considered UNSAT.

Although Kennedy was unable to finish or implement this searching algorithm, he states that the algorithm has $2^{n/2}$ solutions, which is a significant improvement when compared to the 2^{n} complexity of exhaustive searching. From his report, Kennedy 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 he setup the matrices and equations from a memory file instead. Kennedy suggested that creating a lookup table of solutions for cases in which two variables differ would be worth investigating. The only issue he saw with this method would be that there would be a significant jump in memory usage and the board he was using did not have enough non-volatile flash to store this data, suggesting that a more powerful board might be necessary.

The Frankenstein Algorithm

GRASP

GRASP is an exhaustive search algorithm that learns from mistakes it makes and prevents them from occurring in the future. GRASP learns by creating a decision tree, which remembers the order of each decided variable, allowing for easier backtracking. Backtracking occurs after a conflict, a scenario where an equation has become unsatisfied. This is where the term Conflict Driven Clause Learning (CDCL) comes from, as GRASP is the earliest example of a CDCL. GRASP was finished in 1996, and the success of GRASP cemented the CDCL architecture as the standard for SAT solvers.

Figure 6 below shows an example of a binary decision tree. The tree begins by deciding the most frequent variable and then decides variables in order of most frequent to least. The decisions are stored in the tree, as well as the decision level that they were decided at. The Frankenstein algorithm is based on the CDCL architecture, including the use of a decision tree similar to the one GRASP uses.

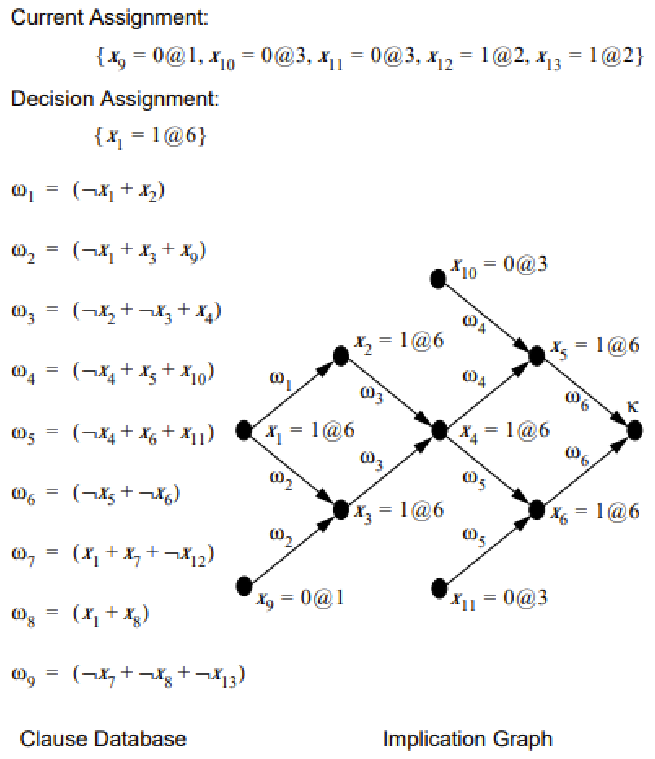

Current Assignment:
$\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\}$
Decision Assignment:
$\{x_1 = 1@6\}$
$\omega_1 = (\neg x_1 + x_2)$
$\omega_2 = (\neg x_1 + x_3 + x_9)$
$\omega_3 = (\neg x_2 + \neg x_3 + x_4)$
$\omega_4 = (\neg x_4 + x_5 + x_{10})$
$\omega_5 = (\neg x_4 + x_6 + x_{11})$
$\omega_6 = (\neg x_5 + \neg x_6)$
$\omega_7 = (x_1 + x_7 + \neg x_{12})$
$\omega_8 = (x_1 + x_8)$
$\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$
$x_{10} = 0@3$
$x_2 = 1@6$
$x_5 = 1@6$
$x_1 = 1@6$
$x_4 = 1@6$
κ
$x_3 = 1@6$
$x_6 = 1@6$
$x_9 = 0@1$
$x_{11} = 0@3$
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
Clause Database
Implication Graph

Figure 6: GRASP example implication graph (Silva & Sakallah, 1996)

Our Process

The Frankenstein algorithm can be broken down into several steps, all of which are described below.

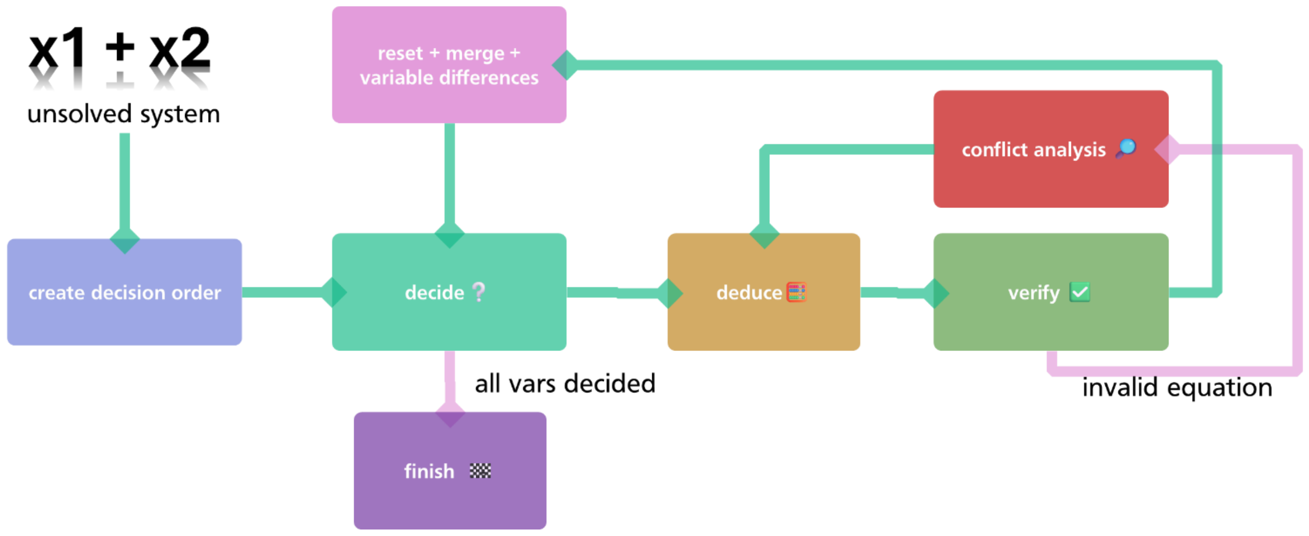


Figure 7: Flowchart of the solver algorithm

Decide

In the decide block, the algorithm determines which variable should be decided. The decision order of the variables is determined using our decision order algorithm, and the value of the variable is chosen randomly. After a decision, the algorithm moves to the deduce block. If the decision stage is entered after all the variables have been decided, then it is assumed that the algorithm has reached a valid assignment of variables, and the algorithm finishes.

Deduce

In the deduce block, equations are checked to see if they have one remaining variable. If there is one variable, then that variable is solved for since every other variable in the equation has a definition and this variable must have a certain assignment, or else the equation will become unsatisfied. Deduction repeats until there are no more variables that can be deduced. After deduction ends, the algorithm moves to the verification block.

Verify

At the verification step, all equations with all their variables defined are checked to see if they are satisfied with the current variable assignment. If any of these equations are unsatisfied after substituting every variable, then a conflict has arisen, and the algorithm moves to the conflict analysis block. If not, then the algorithm moves to the reset + merge block.

Conflict analysis/resolution

When a conflict occurs, the algorithm will append a clause that prevents the current variable assignments from ever occurring. Afterwards, all variable assignments up to and including the last decided variable are reverted, and the root of the level's definition, the last decided variable, is flipped. Finally, the matrix is restored to how it was before the assignment. After conflict analysis, the algorithm moves back to the deduce block.

Reset + Merge

Occurring just before the decision block, the algorithm will first check if the assignment should be reset if the clauses it is finding are not useful. When a reset occurs, the learned clauses are not removed, and only the variable assignments are reset. For merging, if there is a quadratic term that is still inside the window with only one defined variable with a definition of 1, then in the matrix**,** the quadratic term's bit will be XORed with the undefined variable's column in the matrix.

For example, in a system where x1 is defined to be 1, the term x1x2 essentially interpreted as x2. The matrix represents the presence of every term in each equation, so XORing x1x2 into x2 is the algorithm's way of interpreting x1x2 as x2.

Implementation Concepts

In addition to these main steps, a few auxiliary systems were created to help organize and improve the algorithm.

Column arrays

In the matrix, column indices are stored in a separate array. Our algorithm will frequently modify the order of columns, since physically swapping every column in the matrix will consume a lot of processing power. The column array maps the current index of the column to its original index, indices representing the current column order and values representing the column indices in the original matrix. The column order of the original matrix is never modified, and its data is only modified during the merging process.

The window

The window represents the area of the matrix the algorithm is looking at. When parsing the matrix, blocks such as deduction, verification, and merging will only look at columns within the window. For a window of size n, the window will start at the RHS, the leftmost column, and observe every column up to the n-1th column from the left. The purpose of the window is to exclude defined quadratic terms from evaluation since they will be merged into the linear columns during the merge process. Note that the column array is used to move the defined quadratic terms to the end of the column order.

Decision Trees

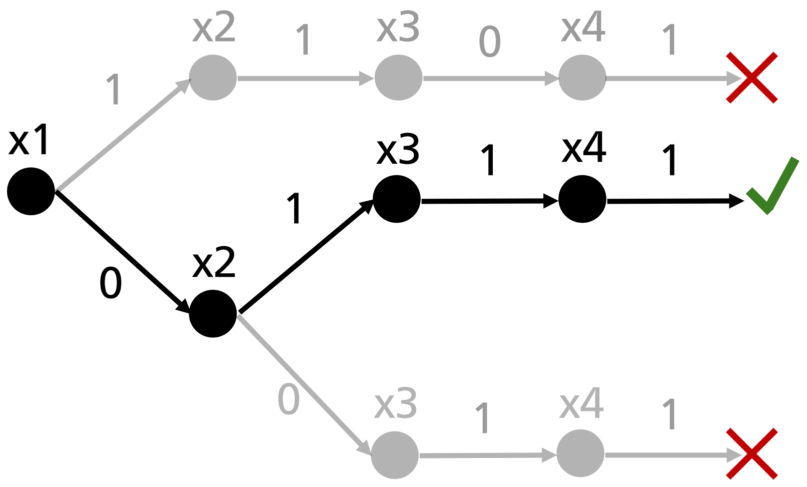


Figure 8: Decision tree, visualized

Decision trees help organize the variable assignments into different decision levels**,** which are used during the reversion process. In the decision tree shown in Figure 8, each branch represents a different decision level, while non-branches are variable assignments that belong on the same level. x1 represents the root of the lowest decision level, while x2 and x3 in the valid branch represent the second and third levels, respectively. Roots are always variables that were decided in the decide block. A root's outgoing nodes are either roots of higher decision levels, or deduced variables on the root's current decision level.

Watched Variables

For an equation to be verified, it must have zero variables that are undefined. Similarly, for a variable to be deduced from an equation, the equation must have one undefined variable. With this logic, if an equation does not meet these requirements, then verification or deduction can be skipped for the equation. The watched variable system will keep track of at most two undefined variables for every equation and added clause. If a watched variable is defined, then for every equation that had that watch variable, it will be assigned a new undefined variable. Eventually, equations will have less than two watch variables, which marks them eligible for deduction or verification. The watched variable system is much faster than the deduction and verification processes, so this significantly speeds up the algorithm.

Linear Block Distance

Basically, when a conflict clause is added to the system, the amount of decision levels present is the linear block distance for the clause. For resetting, the average LBD is calculated after some amount (around 300-700) of iterations of the algorithm. If the LBD of the clause being added is higher than the average by a significant amount (aka, the LBD average is multiplied by a number greater than 1), then the search restarts**.**

Decision Algorithm

The Decision Order is determined by scoring the n variables based on how much Hamming weight they contribute to the entire m x n equation system with linear term matrix L (m x n) and quadratic term matrix Q (m x $\frac{n(n-1)}{2}$). For the sake of reducing time in the algorithm, we calculate the hamming weights of every column $L(i)$ and $Q(ij)$, L itself, and $Q(i)$: the matrix of quadratic x_i terms in Q at the beginning of the algorithm. As we go down the decision tree, this allows us the need to only change L itself, and $Q(i)$ instead of recalculating the weights for every branch.

Variables are rated for substitution by score. The total score $S(i)$ of variable x_i is a combination of the two possible changes of the hamming weight of the equation system by substituting it as either 0 or 1. When the variables are ready to be ranked, S(i) is sorted to determine the top k variables. The calculation of the value of k is explored in a later section. Denoting p_i as the probability of x_i being 1, and the zero and one scores as $S_0(i)$ and $S_1(i)$, $S(i)$ can be expressed as follows:

$$S(i) = (1 - p_i)S_0(i) \; + \; p_i S_1(i)$$

For the scope of this project, p is 0.5 universally, meaning an equal chance of any x being 0 or 1. If further developments in the algorithm include a way to guess a reasonable p_i value, the equation can be adjusted easily. $S_0(i)$ and $S_1(i)$ can be expressed as:

$$S_0(i) \; = \; W\big(Q(i)\big) \; + \; W\big(L(i)\big)$$

$$S_1(i) \; = \; W\big(Q(i)\big) \; + \; W\big(L(i)\big) + \Big(W\big(L_N(i)\big) \; - \; W\big(L_N(i) \oplus Q(i)\big)\Big)$$

$$S_1(i) \; = \; S_0(i) + \; \Big(W\big(L_N(i)\big) \; - \; W\big(L_N(i) \oplus Q(i)\big)\Big)$$

Where W is Hamming weight, $Q(i)$ is the matrix of quadratic x_i terms (m x $n - 1$), $L(i)$ is the x_i term (m x 1), and $L_N(i)$ is the matrix of all linear terms except x_i (m x $n - 1$). Substituting these $S_0(i)$ and $S_1(i)$ values into $S(i)$, we obtain the following:

$$S(i) = (1 - p_i)S_0(i) + p_i\left(S_0(i) + \left(W\big(L_N(i)\big) - W\big(L_N(i) \oplus Q(i)\big)\right)\right)S(i)$$
$$= \big((1 - p_i)S_0(i) + p_iS_0(i)\big) + p_i\left(W\big(L_N(i)\big) - W\big(L_N(i) \oplus Q(i)\big)\right)$$
$$S(i) = S_0(i) + p_i\left(W\big(L_N(i)\big) - W\big(L_N(i) \oplus Q(i)\big)\right)$$
$$S(i) = S_0(i) + p_i\Delta W\big(L_N(i)\big)$$

For both possible values of x_i , the equation system's hamming weight will decrease by $S_0(i)$, as it represents columns being XORed into L or RHS. However, for only the case where $x_i = 1$, L changes by the Hamming Distance between $L_N(i)$ and $Q(i)$.

Because of our initial calculations, $S_0(i)$ can be calculated in $O(1)$ time. However, because of the dimensions of $Q(i)$ and $L_N(i)$, calculating $\Delta W\big(L_N(i)\big)$ with pinpoint accuracy is $O(mn)$. This makes determining $S(i)$ take $O(mn)$ time by proxy. Despite this, there is a way to get around this by estimating $\Delta W\big(L_N(i)\big)$ instead, in $O(1)$ time. As such, the algorithm is free to choose if it wants speed or accuracy.

How do we predict the value of $W(V_1 \oplus V_2)$ in terms of $W(V_1)$ and $W(V_2)$? While we can't get an exact value in $O(1)$, $W(V_1)$ and $W(V_2)$ could potentially influence $W(V_1 \oplus V_2)$. First, we identify which vector has less hamming weight, and denote that weight as L_W, and the other weight as H_WLet T_W be the maximum value W(V) can be. Then,

$$W\left(V_1 \bigoplus V_2\right) \cong (H_W - \frac{H_W}{T_W}L_W + \frac{T_W - H_W}{T_W}L_W)$$

The RHS of this equation can be rearranged into either :

$$(H_W + (1 - \frac{2H_W}{T_W})L_W) = (L_W + (1 - \frac{2L_W}{T_W})H_W)$$

Using this in our equation for $S(i)$,

$$S(i) = S_0(i) + p_i\Delta W\big(L_N(i)\big)$$
$$S(i) = S_0(i) + p_i(W\big(L_N(i)\big) - W\big(L_N(i) \oplus Q(i)\big)$$

$$S(i) \cong S_0(i) + p_i\left(W\big(L_N(i)\big) - \left(W\big(L_N(i)\big) + \left(1 - \frac{2W\big(L_N(i)\big)}{\mathrm{m(n-1)}}\right) W\big(Q(i)\big)\right)\right)$$

$$E(i) = S_0(i) - p_i\left(1 - \frac{2W\big(L_N(i)\big)}{\mathrm{m(n-1)}}\right) W\big(Q(i)\big)$$

$$E(i) = W\big(Q(i)\big) + W\big(L(i)\big) - p_i\left(1 - \frac{2W\big(L_N(i)\big)}{\mathrm{m(n-1)}}\right) W\big(Q(i)\big)$$

$$= W\big(L(i)\big) + (1 - p_i\left(1 - \frac{2W\big(L_N(i)\big)}{\mathrm{m(n-1)}}\right)) W\big(Q(i)\big)$$

Using this formula, $S(i)$ can be estimated in O(1) time.

Substitution

Suppose the algorithm has decided to substitute a specific number k of variables $v_1, v_2, \ \dots \ v_k$ as bitSequence $b_1, b_2, \ \dots \ b_k$. This can either be the result of a lookup table or sorted variable score array. Regardless of the source of the decision, new system of size $n - k$ will be created with matrices NL and NQ and iterated through.

First, create the NQ and NL matrices by extracting all the unsubstituted columns in Q and L . Alter the hamming weights of the variables in NQ by subtracting the weights of the appropriate quadratic columns eliminated.

$$\Delta W\big(NQ(i)\big) = -\sum_{j=1}^{k} W\left(Q\big(iv_j\big)\right)$$

$$\Delta W(NL) = -\sum_{j=1}^{k} W\left(L\big(v_j\big)\right)$$

While altering the non-substituted variable quadratic weights is $O\big(k(n-k)\big)$, and can become $O(n^2)$ if k is chosen as some factors times n, it is a one-time operation that only has to be done before the iterations.

Case 1: Lookup table/Substituted Variable

Alter NL accordingly based on the variables substituted as 1. If this isn't the first iteration use the result of the previous iteration bit sequence XORed with the current iteration bit sequence to make the alteration quicker. If the linear term of variable v_i does not change, its respective XOR value will be zero. Otherwise, it's 1, and you XOR it with the appropriate quadratic column.

Case 2: Sorted Variable Score Array

Define $bitSeq_k$ as : $0bitSeq_{k-1}(0),\ \dots\ 0bitSeq_{k-1}(2^{k-1}-1), 1bitSeq_{k-1}(2^{k-1}-1),\ \dots\ 1bitSeq_{k-1}(0)$

Ex. $bitSeq_1$= 0,1 $bitSeq_2$ = 00,01,11,10 $bitSeq_3$ = 000,001,011,010,110,111,101,100

Iterate through all 2^kpotential bit sequences. Because of how $bitSeq$ is designed, when XORing a previous iteration the result will always be all zero except for 1 bit c_b. XOR the appropriate quadratic columns in Q containing c_b onto NL. Now the remaining variables can be sorted by score, in $O((n-k)\log(n-k))$ time.

Once RHS is changed accordingly, the new system is ready for testing. The number of XORs on RHS depend on $W(bitSeq)$, as the quadratic columns in Q containing c_b with the other variable being substituted as 1 are the ones being XORed. This makes the RHS calculation $O(W(bitSeq))$ per iteration, not $O(n-k)$ every single time like NL .

K value decision

The value of k in the algorithm is determined by calculating the theoretical complexity of solving 2^ksystems of size $n-k$. This can be calculated as the total time to calculate every RHS, every NL, and solving every subsystem, as NQ is the same for every new equation system. On every iteration, The time complexity for NL is constantly $m(n-k)$, but for RHS it varies. Using bitSeq, the time complexity is $mW(v)$. Therefore, the total time can be expressed with the formula:

$$T(k) = R_T\left(\sum_{h=0}^{k} mh_k\, C_h\right) + 2^k\left(L_T m(n-k) + F(n-k)\right)$$

Where R_T and L_T are constants chosen based on how long it takes to perform the L and RHS parts of the algorithm, and F is the average time it takes to solve a system. For the sake of simplicity, let $F(n) = F_T 2^{\phi n}$, as most MQ SAT problems are generally $F(n) = O(2^{\kappa n})$for some value κ . Now all of $T(k)$ is differentiable except for the first R_T part. However, SIMGA can be easily calculated because adding the h and k – h term results in RtmkCh, and since RT is a constant anyway, we'll use it to tank the ½ that should be there.

$$T(k) = R_T mk2^k \; + \; 2^k\big(L_T m(n \,-\, k) \; + \; F(n-k)\big)$$
$$T(k) = 2^k(mkR_T + m(n-k)L_T) \; + \; F_T 2^{\phi(n-k)+k}$$
$$T(k) = m2^k(kR_T + (n-k)L_T) \; + \; F_T 2^{\phi n} 2^{(1-\phi)k}$$

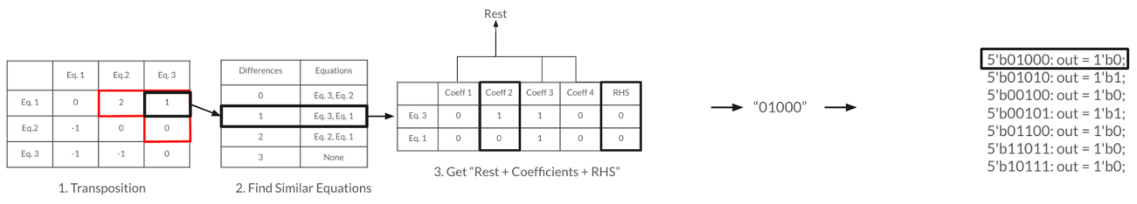
$$\frac{dT}{dk} = m2^k\big(\ln 2\,(k(R_T - L_T) + nL_T) + (R_T - L_T)\big) + \ln 2\, F_T 2^{\phi n}(1-\phi)2^{(1-\phi)k} = 0$$
$$m\left(k(R_T - L_T) + nL_T + \frac{(R_T - L_T)}{\ln 2}\right) + F_T 2^{\phi n}(1-\phi)2^{(-\phi)k} = 0$$
$$F_T 2^{\phi n}(1-\phi)2^{(-\phi)k} = m\left(k(L_T - R_T) - nL_T + \frac{(L_T - R_T)}{\ln 2}\right)$$

This equation is unsolvable, also due to the scope of this project we don't have a clear range of values for L_T R_T ϕ and F_T. For now, we propose 2 methods to determine k : calculate the difference between the left and right side in the equation and whatever value is lowest is the optimal k. Or select k such that the new system is approximately half the size of the previous system.

$$k(n-1) - \frac{k(k-1)}{2} + k = \frac{n(n+1)}{4}$$
$$\frac{k^2}{2} \; - \; \left(n \; + \; \frac{1}{2}\right)k + \frac{n(n+1)}{4} = 0$$
$$k = \left(n \; + \; \frac{1}{2}\right) - \sqrt{\frac{n(n+1)}{2} + \frac{1}{4}} \cong n\left(1 - \frac{1}{\sqrt{2}}\right)$$

Look Up Table for Python Implementation

From Kennedy's work, the team decided that LUTs could further provide speed up the reduction when solving the SAT problem. The aim is to identify similar equations and deduce a solution using LUTs. We concluded that the differences above three would not provide the necessary speed up intended for the algorithm, thus LUTs were only created for differences up to three. The figure below shows how the LUTs are utilized in the Python Implementation. These LUTs lay the foundational work for Verilog conversion. However, we decided to forgo those implementations as it did not fit our scope. That process can be found in Appendix A.

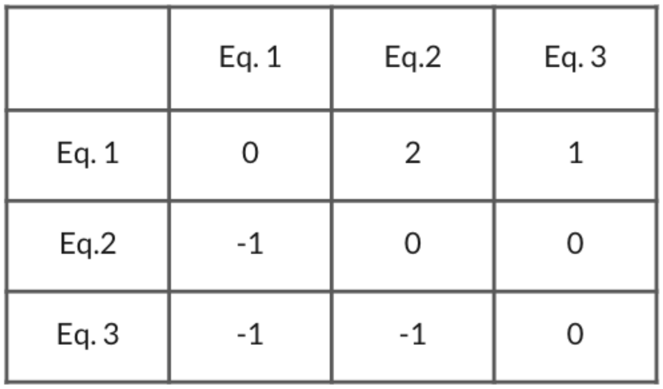


```
5'b01000: out = 1'b0;
5'b01010: out = 1'b1;
5'b00100: out = 1'b0;
5'b00101: out = 1'b1;
5'b01100: out = 1'b0;
5'b11011: out = 1'b0;
5'b10111: out = 1'b0;
```

Figure 9: Flowchart of LUTs

Identifying Differences

The system of equations is XORed by its transposed matrix. By XORing, the difference of each equation can be shown. In the table below, the matrix stores the number of different variables between every single equation. From the table, we can conclude that Eq 1 and Eq 2 have two differences as its algebraic sum of the XOR result is 2. Note that only the upper triangle of the matrix is used, the bottom half is ignored as they are identical. The matrix is used in later steps to sort the equations by their varying differences (See step 2 of Figure 9).



	Eq. 1	Eq.2	Eq. 3
Eq. 1	0	2	1
Eq.2	-1	0	0
Eq. 3	-1	-1	0

Figure 10: Table of Differences

Deduction

Case 0

When the difference between two equations is zero, it means that the equations are equal. Thus, the corresponding row and column are deleted from the matrix, and the equation is removed from the original system.

Intermediate Steps

To use the Case 1, Case 2, Case 3 LUTs, it is necessary to differentiate the terms (whether it is quadratic or linear, dependent or independent).

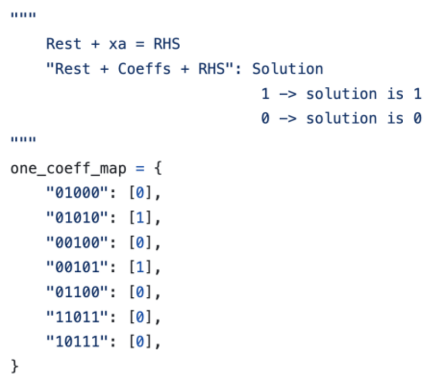
- A linear term is when the term is associated with only one variable. Ex: x1, x2, x3
- A quadratic term is when the term is associated with more than one variable. Ex: x1x2, x2x3
- A dependent equation is when the terms in the equation have a dependency on each other. Ex: x2 + x2x3 + x2x4
- An independent equation is when the terms in the equation have no dependency on each other. Ex: x3 + x7 + x5x6

The team developed an algorithm that allowed the categorization of the equations based on the combination of terms. Therefore, allowing the use of these LUTs automatically.

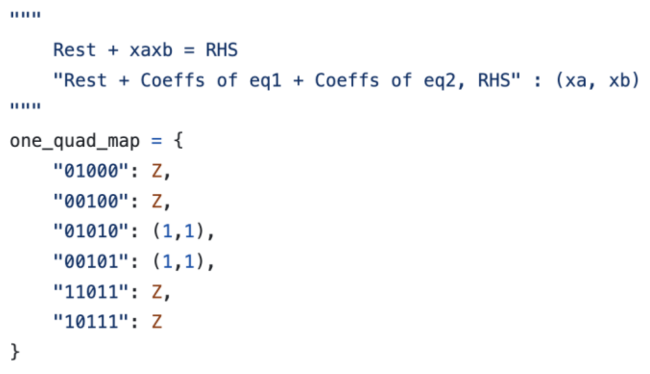
It is also necessary to assume a value for Rest. Rest is the value for the rest of the equation which is the same. Rest can be either 0 or 1, which is unknown to the solver.

Case 1

In Case 1, there is only one variable that is different between the two equations. This variable can either be a linear term or a quadratic term shown below. Here, the variable Z indicates that pairs of solutions that make up to zero. Thus, in one_quad_map, (xa, xb) can be the combination that is: (0,0), (1,0), (0,1).



```
"""
    Rest + xa = RHS
    "Rest + Coeffs + RHS": Solution
                            1 -> solution is 1
                            0 -> solution is 0
"""
one_coeff_map = {
    "01000": [0],
    "01010": [1],
    "00100": [0],
    "00101": [1],
    "01100": [0],
    "11011": [0],
    "10111": [0],
}
```



```
"""
    Rest + xaxb = RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2, RHS" : (xa, xb)
"""
one_quad_map = {
    "01000": Z,
    "00100": Z,
    "01010": (1,1),
    "00101": (1,1),
    "11011": Z,
    "10111": Z
}
```

Figure 11: LUTs for Varying 1 Difference

Case 2

In Case 2, there are five variations of terms that can make up the cases. The LUTs follow the same guidelines are the Case 1 LUTs. However, due to the variations, it is possible to multiple sets of solution for different equations. The LUTs can be found in Appendix L.

Case 3

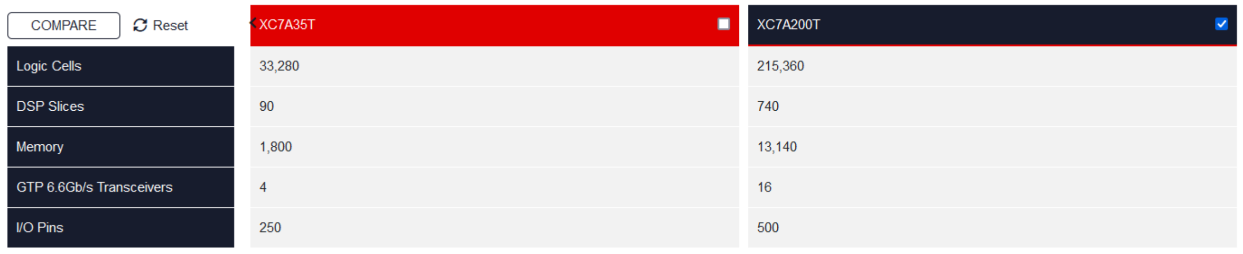
In Case 3, there are thirteen variations of terms that can make up the three differences. Case 3 LUTs simplifies the equations to Case 2 or Case 1. However, in some cases, it must use reclusiveness as the equation can only simplify Case 3 LUTs to other types of Case 3 LUTs. These LUTs can be found in Appendix L.

Choosing a New Board

Due to the high computational and memory costs of this project, 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, we decided to look for a board that fulfills 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)

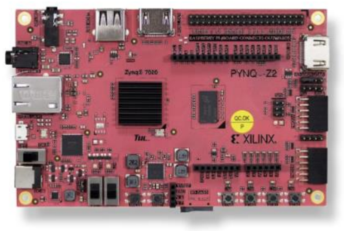
When accounting for these requirements, a board with the Artix-7 A200 FPGA, more specifically, Digilent's Nexys Video board would be the best for this application. The A200 Artix-7 FPGA has 215,360 logic cells, covering the second requirement, and has 13 megabits of BRAM, fulfilling the third requirement. 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.



	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 12: 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.



Pynq Z2 Board

```
$299 (lower-end range)
ZYNQ XC7Z020-1CLG400C FPGA SoC
650MHz dual-core Cortex-A9 processor

FPGA Logic on board similar to Artix-7
     630 KB BRAM

512 MB 525MHz(1050 MT/s) DDR3
16 MB Flash and MicroSD Slot

Designed to be used with PYNQ, an open
source framework allowing for programming of
the SoC with Python
```

Nexys Video Artix-7 FPGA

```
$549 (mid range)
Nexys Video Artix-7 FPGA
13 Mbits BRAM, 32MB Flash, microSD
card slot for expandable
non-volatile storage

Xilinx Artix®-7 XC7A200T-1SBG484C

On-board OLED display (can use for
debugging/testing)

512 MB 400MHZ(800 MT/s) DDR3
```

Figure 13: Comparison of PynqZ2 board and Nexys FPGA

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.

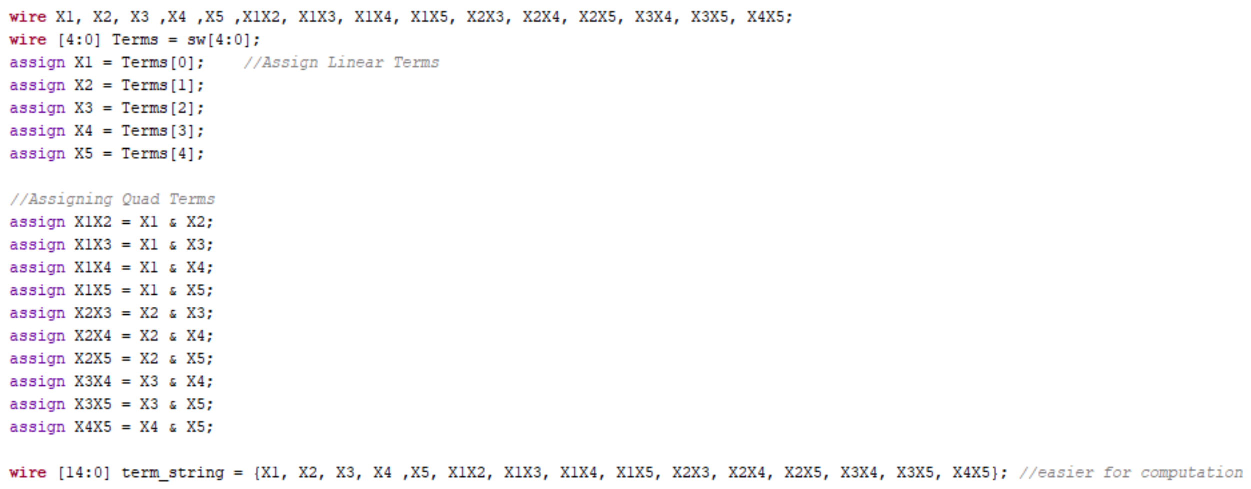
The modules are being converted to Verilog to be able to run properly on the board. Not everything of the module needs to be converted between languages, but the board was not as “plug and play” as expected. Some of the Python programs work as expected, but much of the top-level module must be converted into Verilog and System Verilog. This is a different effort from the full CDCL in System Verilog. The two boards can be used to compare a Python implementation vs a full HDL implementation and determine the performance differences.

Exhaustive Searching

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

After consideration, we were able to create a new packed structure 16 bits long for the equation's coefficients, with *n* for the implementation being five linear terms (5 linear terms + 10 quadratic terms + RHS = 16). This structure is used to create a packed array of 16 hard-coded equations generated with the use of a random number generator online to perform the exhaustive search. Although the initial idea was to use a clocked 16-bit linear feedback shift register (LFSR) to create the array, there ended up being some cross-clock domain synchronization issues making some equations have all values of X (don't care in System Verilog) 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 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, XOR all of the equations together, similar to the simulation in the improving linear solving section. This is to determine what terms are needed to be accounted for to solve the system. In regard to the terms, we 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 14 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 14: 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 solving 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 Matthew's solution was only concerned with combinational logic, a clock was not required for Matthew's 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.

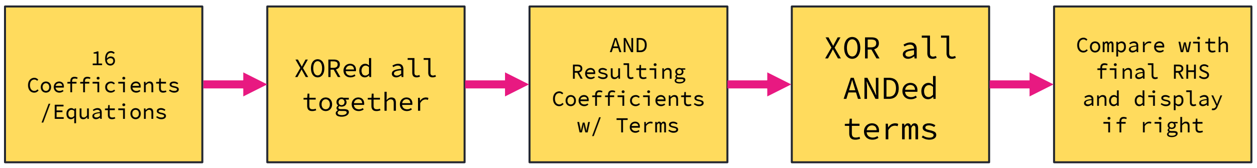

16 Coefficients /Equations
XORed all together
AND Resulting Coefficients w/ Terms
XOR all ANDed terms
Compare with final RHS and display if right

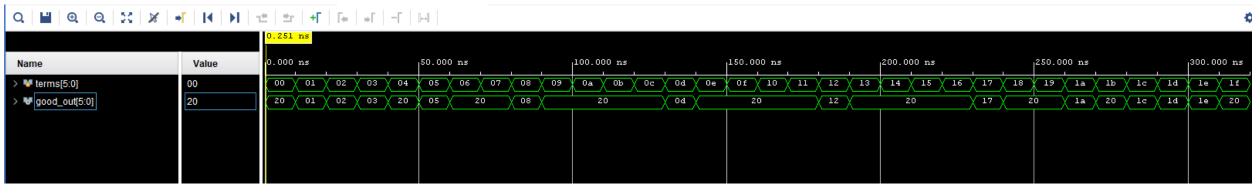
Figure 15: Exhaustive Searching Flow Diagram

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

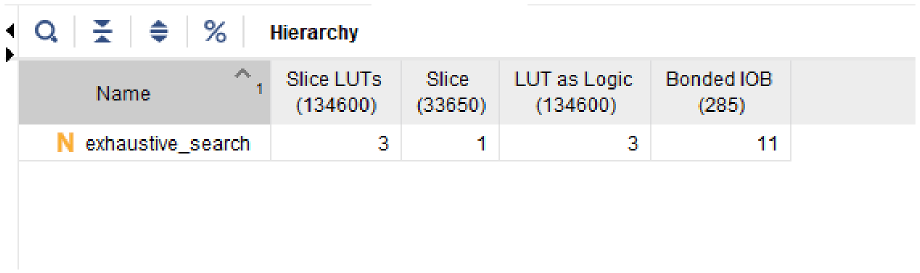
First, to test if the exhaustive searching works, we simulated the searching module, updating the terms register of the simulation every 10 nanoseconds, and comparing it to what the good_out wire, the representation of the LEDs, gave. As shown in Figure 13, any time that the wire had a hexadecimal value 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 Matthew's 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, we 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.



```
Name           Value
terms[5:0]     00     00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
good_out[5:0]  20     20 01 02 03 20 05 20 08 20 0d 20 12 20 17 20 1a 20 1c 1d 1e 20
```

Figure 16: Exhaustive Search Vivado Simulation Results



Name	Slice LUTs (134600)	Slice (33650)	LUT as Logic (134600)	Bonded IOB (285)
exhaustive_search	3	1	3	11

Figure 17: Exhaustive Search Resource Utilization

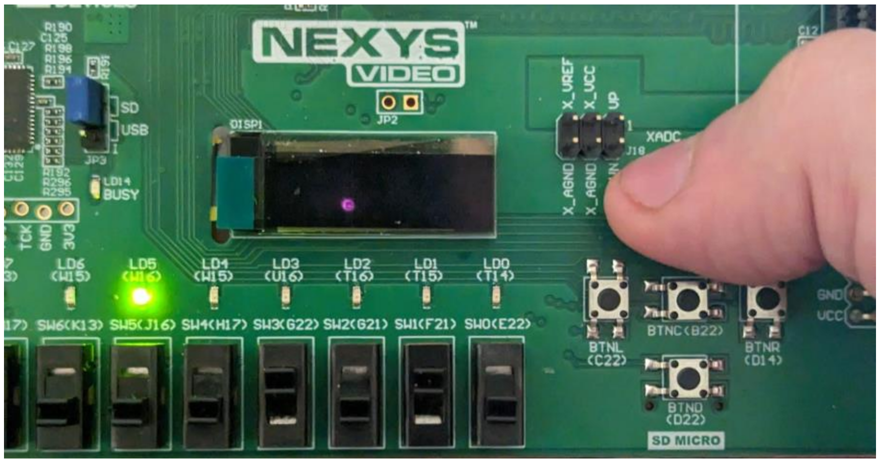


Figure 18: X2 and X4 = 1 on Board

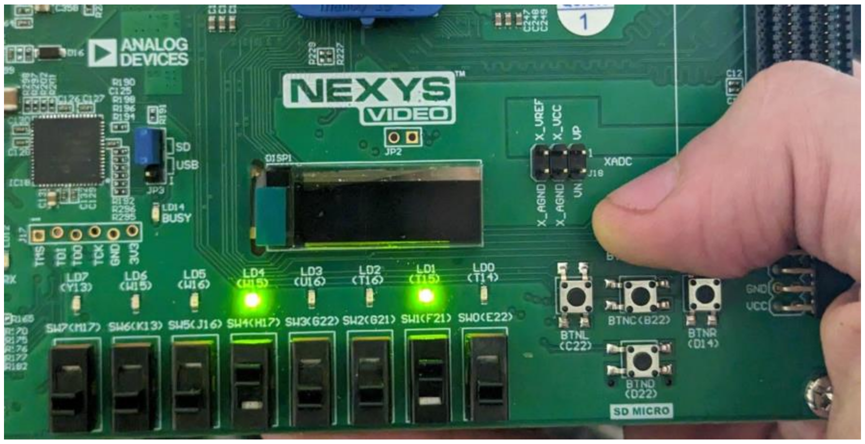
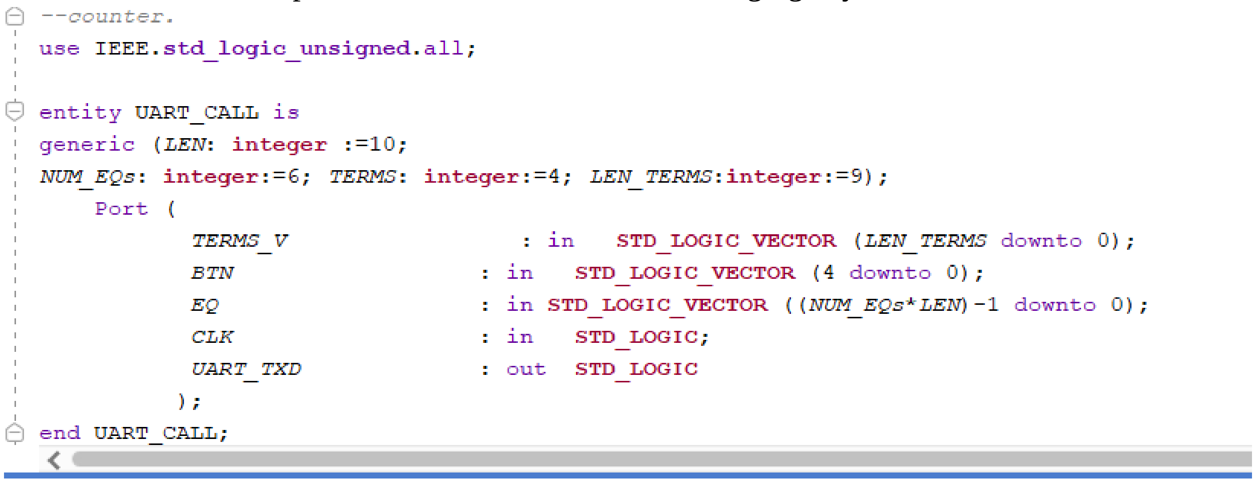


Figure 19: X2 and X5 = 1 on Board

Exhaustive Searching Expanded

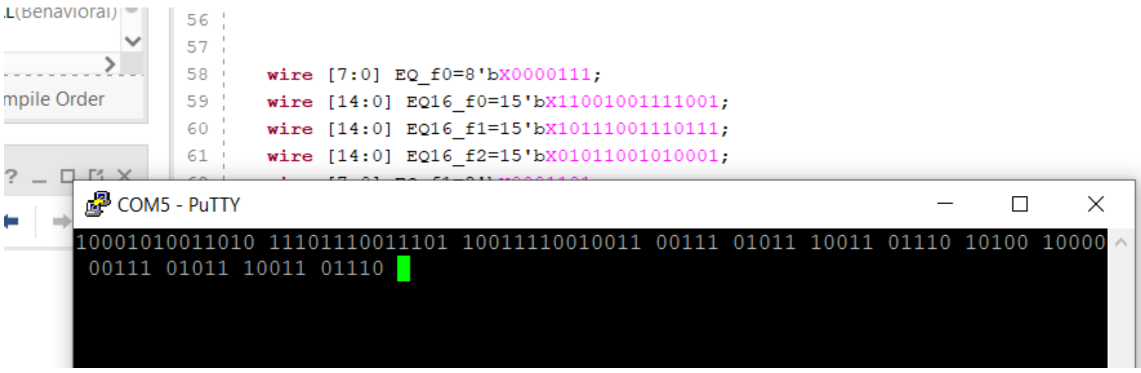
In order to meet the challenge of solving larger and more complex systems, the exhaustive solver was expanded to eight variables (as a proof of concept), which creates an individual equation size of 35 bits. The solution length is the same length as the number of variables, which in this case is 8 terms long. Eight was chosen just to show expandability, the solver is adjustable for more. To meet the need for showing larger solutions, UART printing was used. The starting code provided by Digilent (Appendix K) in VHDL served as a good example for how the UART printing worked. The code was modified to accept parameters based on the equation length and size, as shown in FIGURE X. The parameters allow the VHDL module to be called from the Verilog top level. There was a focus on parametrization so that it could be easily changed. The length of the terms, number of equations, and number of variables are all modifiable from the top level of the module, without changing any VHDL code.



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

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

Figure 20: UART printing module in VHDL

Figure 21 shows an example output with 5 variables. The equations are printed first, then the solutions are shown. Because of the limitations of Verilog, the solutions wire length must be determined at compile time. This means that the solutions wire length must be guessed. Figure X shows what happens as a result of this. When the solutions wire is longer than the number of solutions, the solutions are just repeated. If the system has no solutions, the entire solutions wire is all 0's. The dont care (X) is added in to be written over, due to VHDL restrictions on for loops.



```
wire [7:0] EQ_f0=8'bX0000111;
wire [14:0] EQ16_f0=15'bX11001001111001;
wire [14:0] EQ16_f1=15'bX10111001110111;
wire [14:0] EQ16_f2=15'bX01011001010001;
```

```
10001010011010 11101110011101 10011110010011 00111 01011 10011 01110 10100 10000
 00111 01011 10011 01110
```

Figure 21: Example output with input equations shown

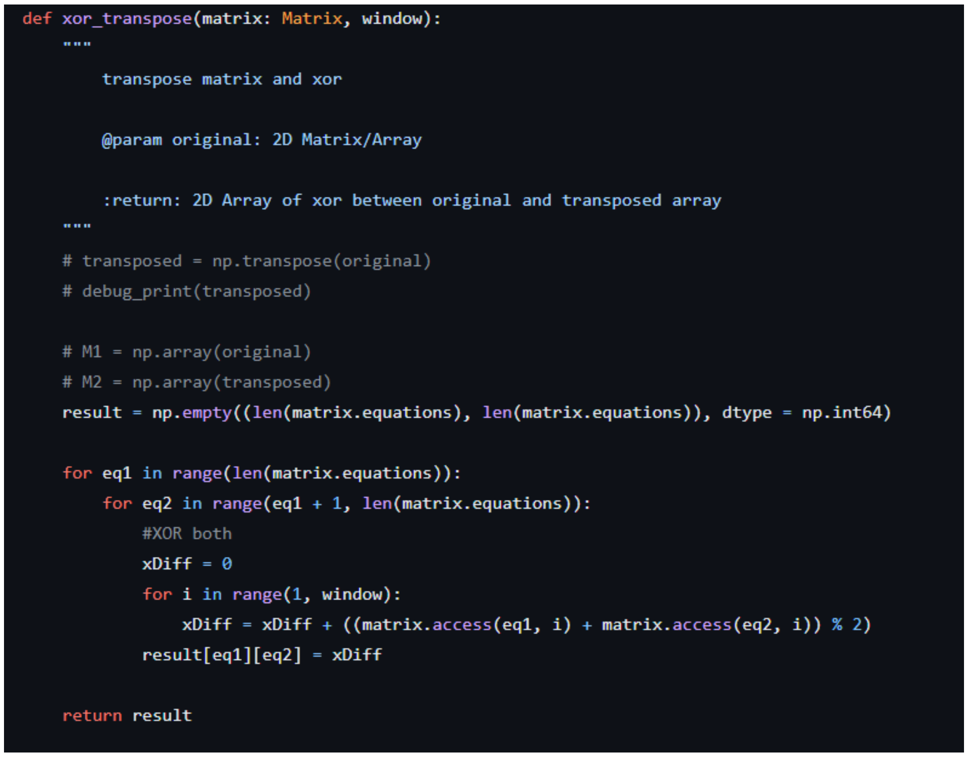
The exhaustive solver is similar to the previous exhaustive solver, where a terms wire is created and iterates through all possible options. If the solution works, it is appended to the solutions and later sent to the UART printing VHDL module. The specific benefit of adding the UART printing and slightly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

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

LUTs in Python to Verilog Conversion

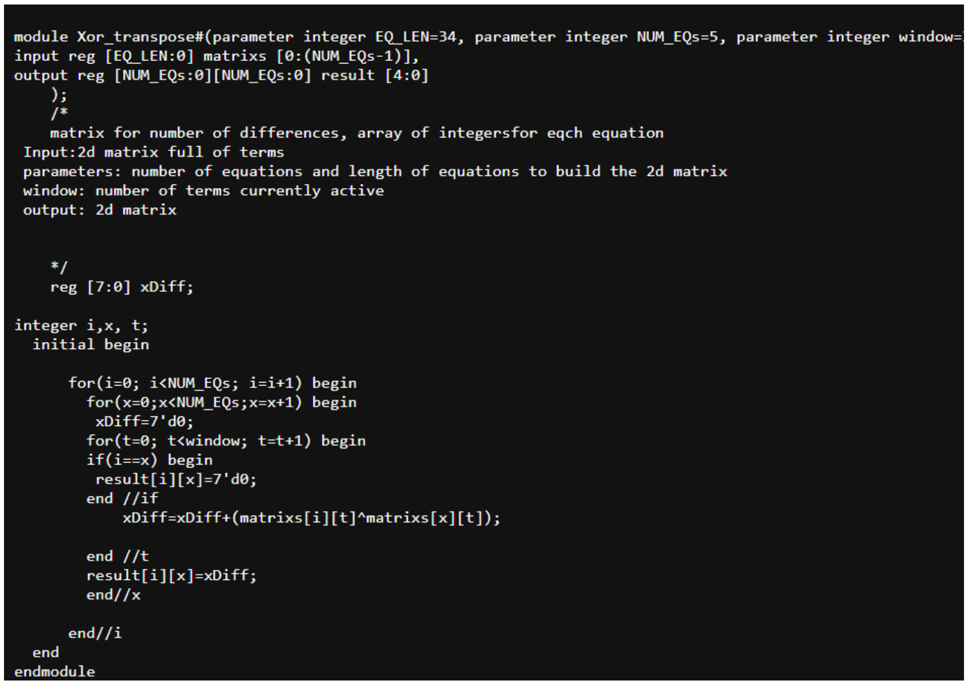
After the CDCL in python finished, the group convened to work together on converting the files to System Verilog. This includes, the LUTs, operations on the LUTs, and other important CDCL functions. This presented a unique set of challenges, as the original code contained many functionalities that do not exist in System Verilog, such as dynamic arrays.

As of the end of this MQP, the modules are mostly completed but the code itself has not been run on a board yet. Figure 22 and 23 show two of the same modules, one in python and one in System Verilog. The module takes in an NxM matrix and returns an NxN matrix with the differences between equations. For example, xor_transpose[2][4] will show the number of differences between equations 2 and 4. This shows the difficulty in converting modules, as many python functionalities are not available in System Verilog. For example, the inputs in System Verilog must be more explicitly defined, including the size of arrays.



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

        @param original: 2D Matrix/Array

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

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

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

    return result
```

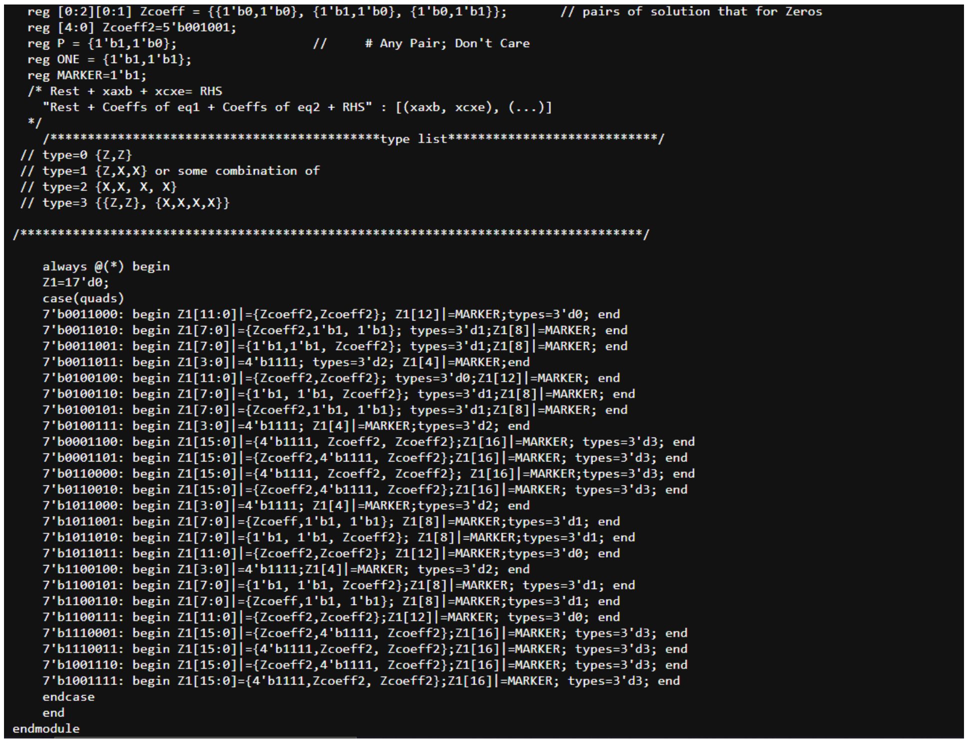
Figure 22: XOR transpose in Python



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


    */
    reg [7:0] xDiff;

integer i,x, t;
  initial begin

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

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

      end//i
  end
endmodule
```

Figure 23: XOR transpose in Verilog

Another major accomplishment of the group was the conversion of additional LUTs from python to System Verilog. These LUTs cover cases for one, two, and three differences between equations. Currently the Case 3 LUTs are not implemented with recursiveness, however, it is an expansion of the previous section mentioning differences between two equations. They cover different types of cases such as differences being quadratic dependent (xaxb + xaxc, where one of the differences is a shared term and the other is not). Although the LUTs are written, they still require testing to ensure accuracy and performance. There are a number of important tables that speed up the calculation complexity. Figure 24 shows an example of one table below.



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

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

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

Figure 24: LUT for 2 quadratic independent differences

Converting Frankenstein to Verilog

Alongside the equation difference algorithm, the Frankenstein algorithm was also set to be converted to Verilog. To accomplish this, a modular approach was used, converting only small parts of the algorithm at a time into Verilog. The PYNQ board was helpful in testing these modules, as it allowed for the hardware modules to replace their Python counterparts in the algorithm and essentially use the Python version of the algorithm to test the hardware.

Debugging with PYNQ-Z2

The PYNQ-Z2 board runs a Jupyter server on its CPU, which allows for Python code to be ran on it. Additionally, it allows for a bitstream to be loaded into the FPGA fabric. To communicate between the FPGA fabric and CPU, the AXI4 Protocol is necessary, and communication was done via shared registers between the fabric and the CPU. In hardware synthesis, each module was wrapped with an AXI4 Lite wrapper, which handled AXI4 signal processing and linked the shared registers to their respective port definitions in the module. Each AXI4 wrapped module was connected to an AXI interconnect module, which was connected to the ZYNQ7 Processing

System module that is responsible for connecting the CPU to the bitstream. A block diagram of all the Verilog-translated modules created by the end of the project are shown in Figure ??.

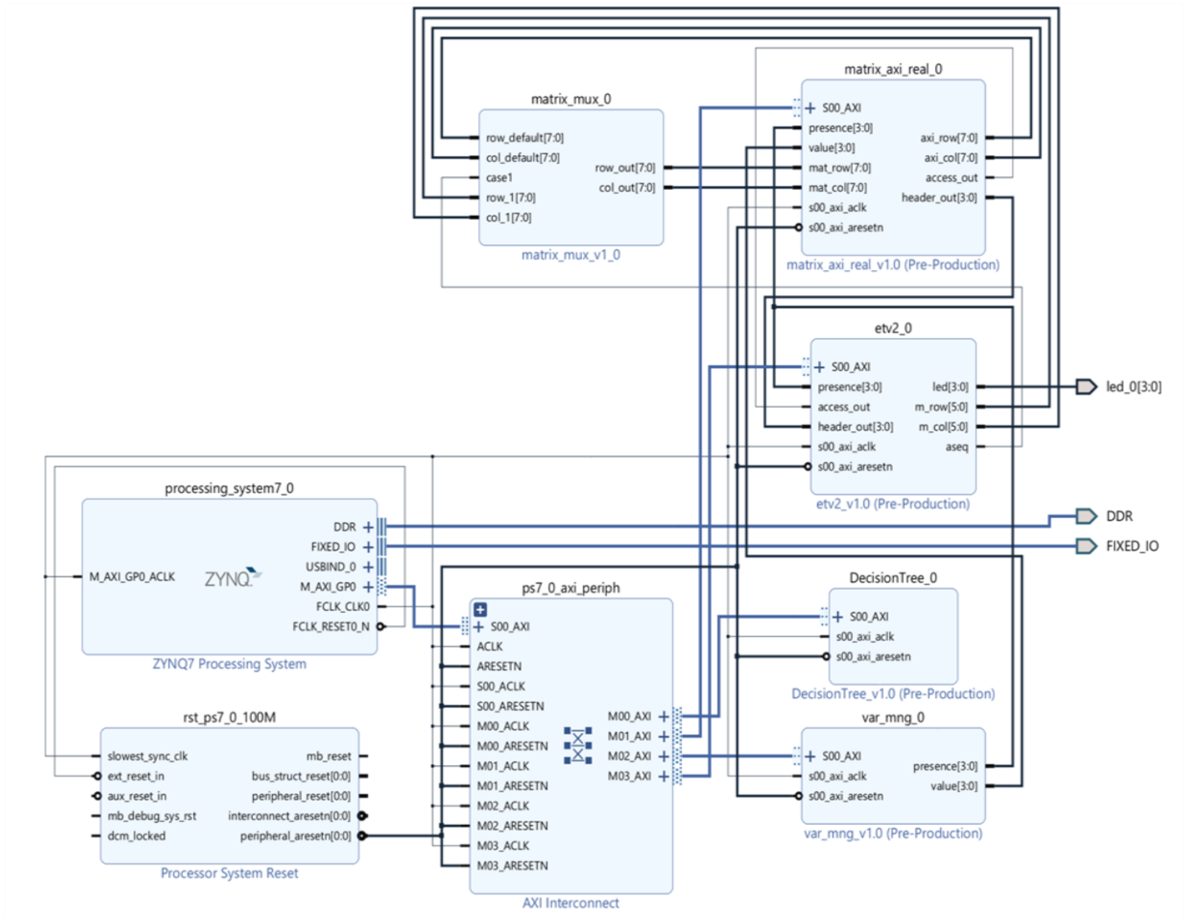


Figure 25: Block diagram of the Frankenstein HDL implementation

Limitations of the PYNQ-Z2 board

While debugging hardware with the PYNQ-Z2 did prove useful in translating some of the modules to HDL, it also had some serious drawbacks. There were a lot of discrepencies between testing the module in Vivado simulations and with register manipulation in Jupyter. Most of the time, a module could work perfectly in a Vivado simulation, but on actual hardware it does not carry out its intended behavior. Debugging modules with these problems is tedious compared to the simulations, as any clues about the bug can be drawn only from the shared registers. Additionally, making any changes to the HDL requires the bitstream to be regenerated, which would take almost 15 minutes to generate and load into the PYNQ board. For working modules at runtime, the AXI4 protocol ended up significantly hindering the performance of the HDL implementation. Multiple AXI4Lite signals were written and read each time a module was used, and the modules that were converted to HDL were the most frequently used parts, so the HDL implementation was already making thousands of AXI4 calls every second. Thankfully, once the entire algorithm is in the FPGA fabric, there would be no need to wrap the modules in AXI4, since the modules can directly connect with one another.

Conclusion

During our time on the project, our goals were to solve a Boolean system of equations with linear and quadratic terms, expand on the two equation variable differences started by Frank, and create a working solver on an FPGA. We researched other CDCL solvers to create our own CDCL solver. We also created LUTs used to analyze potential solutions from two equations with up to three different variables. Additionally, we started implementation of the Frankenstein algorithm on an FPGA, and fully implemented the LUTs on hardware. Although the project is unfinished, a significant amount of progress was made, and future MQP teams can pick up where we left off.

The Python implementation successfully implements CDCL concepts and the decision order algorithm. The mixed algorithm (Python and FPGA) takes much longer to execute than the regular Python implementation. However, this is likely due to delay from the AXI4 communication between the FPGA fabric and the CPU, which is only used for testing on the PYNQ board.

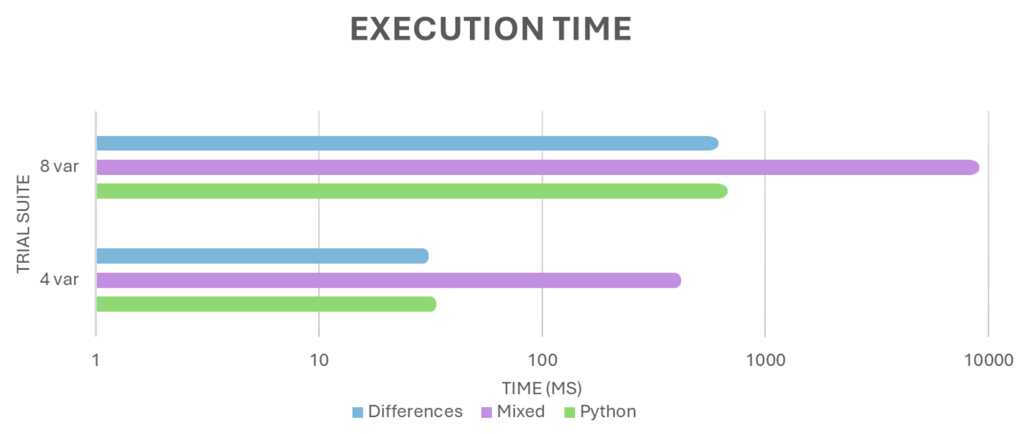


Figure 26: Results from all implementations

The pure Python implementation, mixed Python/FPGA implementation, and the pure Python with equation difference checking were ran against each other on the PYNQ board in a variety of trial suites, and their execution time was measured. Each trial suite had 100 Boolean systems, where each system had the number of variables listed in Figure 26. Our recommendation for the future of this project would be to finish the Verilog implementation of our algorithm, along with improving anything we have done for the implementation.

Acknowledgements

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

References

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

Appendix A: Improving Solving Look Up Table to Account for Quadratics and Two Term Differences by Matthew Lund

Building from Kennedy's work on comparing two equations with one differing linear variable, we decided to create a simulation written in System Verilog in the Xilinx Vivado ISE to generate all of the combinations of coefficients for the equations of the format shown in Figure a, 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 a: Equation Format for Weight of 2 Solving with Example Coefficients

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

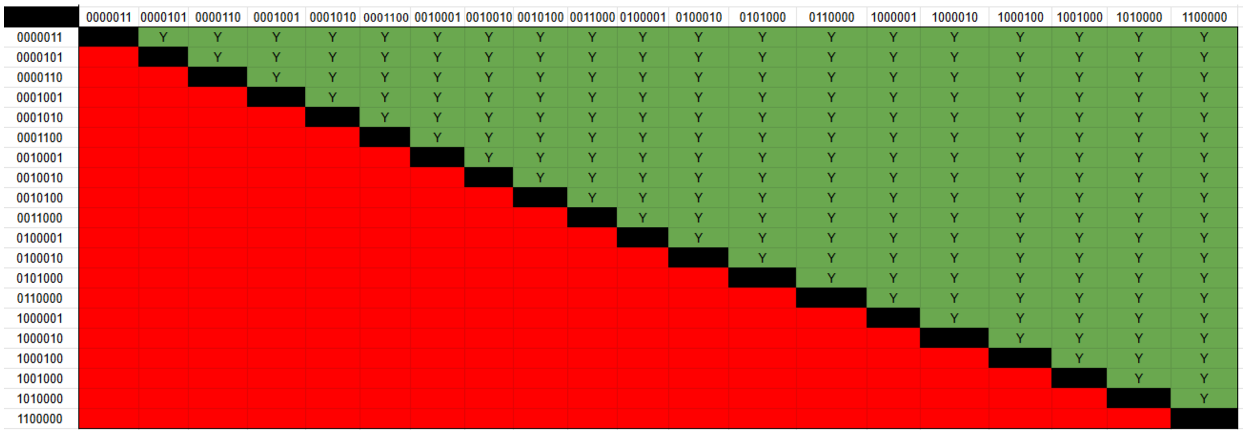
1. Two different linear terms (X1, X3),
2. One linear and one quadratic **without** a shared term (X1, X2X3),
3. One linear and one quadratic **with** a shared term (X1, X1X2),
4. Two quadratics **without** a shared term (X1X2, X3X4),
5. Two quadratics **with** a shared term (X1X2, X2X3).

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 b)

2. The combination of coefficients cannot 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 b: 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 c.

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

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

Knowing all of this, Matthew 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 Matthew's time on this project, Matthew 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.

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
```

Appendix K: Digilent Starting Code

https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-video-basic-user-demo/start

Appendix L: Case 2 and 3 LUTs

```
# Constants
Z = [(0,0), (1,0), (0,1)]     # pairs of solution that for Zeros
P = (1,0)                     # Any Pair; Don't Care
ONE = (1,1)

"""
   Rest + xa + xb = RHS
   "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xa, xb), (...)]
```

```
"""
two_coeff_map = {
    "0011000": [(0,0)],
    "0011010": [(0,1)],
    "0011001": [(1,0)],
    "0011011": [(1,1)],
    "0100100": [(0,0)],
    "0100110": [(1,0)],
    "0100101": [(0,1)],
    "0100111": [(1,1)],
    "0001100": [(1,1), (0,0)],
    "0001101": [(0,1), (1,0)],
    "0110000": [(1,1), (0,0)],
    "0110010": [(0,1), (1,0)],
    "1011000": [(1,1)],
    "1011001": [(0,1)],
    "1011010": [(1,0)],
    "1011011": [(0,0)],
    "1100100": [(1,1)],
    "1100101": [(1,0)],
    "1100110": [(0,1)],
    "1100111": [(0,0)],
    "1110001": [(0,1), (1,0)],
    "1110011": [(1,1), (0,0)],
    "1001110": [(0,1), (1,0)],
    "1001111": [(1,1), (0,0)]
}

"""
    Rest + xa + xbxc = RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xa, xbxc), (...)]
"""
two_linear_quad_independent = {
    "0011000": [(0,Z)],
    "0011010": [(0,1,1)],
    "0011001": [(1,Z)],
    "0011011": [(1,1,1)],
    "0100100": [(0,Z)],
    "0100110": [(1,Z)],
    "0100101": [(0,1,1)],
    "0100111": [(1,1,1)],
    "0001100": [(1,1,1), (0,Z)],
    "0001101": [(0,1,1), (1,Z)],
    "0110000": [(1,1,1), (0,Z)],
    "0110010": [(0,1,1), (1,Z)],
    "1011000": [(1,1,1)],
    "1011001": [(0,1,1)],
    "1011010": [(1,Z)],
    "1011011": [(0,Z)],
    "1100100": [(1,1,1)],
    "1100101": [(1,Z)],
    "1100110": [(0,1,1)],
    "1100111": [(0,Z)],
    "1110001": [(0,1,1), (1,Z)],
    "1110011": [(1,1,1), (0,Z)],
    "1001110": [(0,1,1), (1,Z)],
    "1001111": [(1,1,1), (0,Z)]
}

"""
    Rest + xaxb + xaxc = RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xa, xb, xc), (...)]
"""
two_quad_dependent = {
    "0011000": [(0,P), (1,0,0)],
    "0011010": [(1,0,1)],
    "0011001": [(1,1,0)],
    "0011011": [(1,1,1)],
    "0100100": [(0,P,P), (1,0,0)],
    "0100110": [(1,1,0)],
    "0100101": [(1,0,1)],
    "0100111": [(1,1,1)],
    "0001100": [(0,P,P), (1,1,1)],
    "0001101": [(1,0,1), (1,1,0)],
    "0110000": [(0,P,P), (1,1,1)],
    "0110010": [(1,0,1), (1,1,0)],
    "1011000": [(1,1,1)],
    "1011010": [(1,1,0)],
    "1011001": [(1,0,1)],
    "1011011": [(0,P,P), (1,0,0)],
    "1100100": [(1,1,1)],
    "1100101": [(1,1,0)],
    "1100110": [(1,0,1)],
    "1100111": [(0,P,P), (1,0,0)],
    "1110001": [(1,1,0), (1,0,1)],
    "1110011": [(0,P,P), (1,0,0), (1,1,1)],
    "1001110": [(1,1,0), (1,0,1)],
    "1001111": [(0,P,P), (1,0,0), (1,1,1)]
}

"""
    Rest + xa + xaxb = RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xa, xb), (...)]
"""
two_linear_quad_dependent = {
    "0011000": [(0,P)],
    "0011010": [],
    "0011001": [(1,P)],
    "0011011": [(1,1)],
    "0100100": [(0,P)],
    "0100110": [(1,P)],
    "0100101": [],
    "0100111": [(1,1)],
    "0001100": [(1,1), (0,P)],
    "0001101": [(1,0)],
    "0110000": [(1,1),(0,P)],
    "0110010": [(1,0)],
    "1011000": [(1,1)],
    "1011001": [],
    "1011010": [(1,0)],
    "1011011": [(0,P)],
    "1100100": [(1,1)],
    "1100101": [(1,0)],
    "1100110": [],
    "1100111": [(0,P)],
    "1110001": [(1,1), (0,P)],
    "1110011": [(1,0), (0,P)],
    "1001110": [(1,1), (0,P)],
    "1001111": [(1,0), (0,P)]
}
```

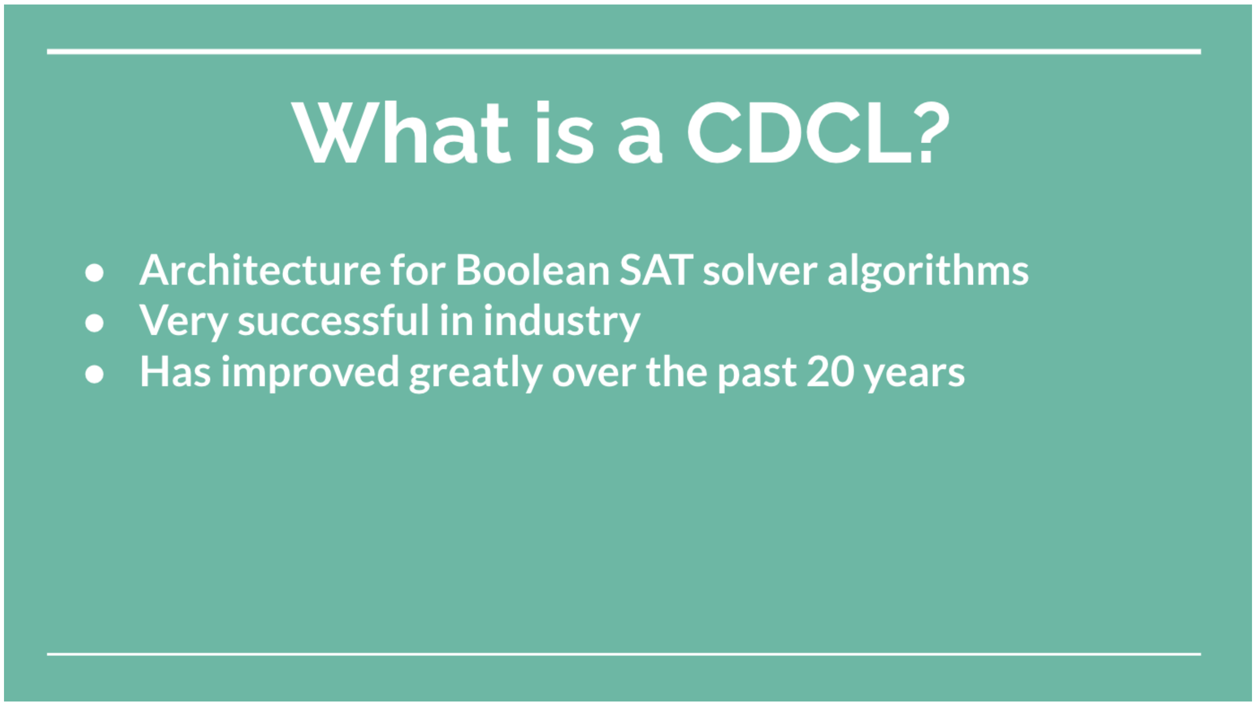
```
"""
    Rest + xaxb + xcxe= RHS
    "Rest + Coeffs of eq1 + Coeffs of eq2 + RHS" : [(xaxb, xcxe), (...)]
"""
two_quad_independent = {
    "0011000": [(Z,Z)],
    "0011010": [(Z,1,1)],
    "0011001": [(1,1,Z)],
    "0011011": [(1,1,1,1)],
    "0100100": [(Z,Z)],
    "0100110": [(1,1,Z)],
    "0100101": [(Z,1,1)],
    "0100111": [(1,1,1,1)],
    "0001100": [(1,1,1,1),(Z,Z)],
    "0001101": [(Z,1,1),(1,1,Z)],
    "0110000": [(1,1,1,1),(Z,Z)],
    "0110010": [(Z,1,1),(1,1,Z)],
    "1011000": [(1,1,1,1)],
    "1011001": [(Z,1,1)],
    "1011010": [(1,1,Z)],
    "1011011": [(Z,Z)],
    "1100100": [(1,1,1,1)],
    "1100101": [(1,1,Z)],
    "1100110": [(Z,1,1)],
    "1100111": [(Z,Z)],
    "1110001": [(Z,1,1),(1,1,Z)],
    "1110011": [(1,1,1,1),(Z,Z)],
    "1001110": [(Z,1,1),(1,1,Z)],
    "1001111": [(1,1,1,1),(Z,Z)]
}

"""
    Look Up Tables for Case 3

    case_diff_ID
    EX: case_3_1 will be the first case of three different variables

"""
```

```
# the code cant run without you
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
season_of_two = {
    "111000":"000111",
    "110001":"001110",
    "101010":"010101",
    "100011":"011100"
}

#3 independent
""" xa + xb + xc
    Example:
        Rest + xa + xb + xc, xa = 1, Rest = 0
        0 + 0*1 + 0*xb + 0*xc = 0 -> 0 + 0*xb + 0*xc = 0
        0 + 1*1 + 1*xb + 1*xc = 0 -> 1*1 + 1*xb + 1*xc = 0
    "Rest + Coeff1 + Coeff2"
    Solution is [(new equation, xa=?),(...)]: [("R1 R2 Coeff1(2) Coeff2(2)", xa), (...)]
"""
case_3_1= {
    "0000111":[("010011",1),("00011",0)],
    "0001110":[("010110",1),("00110",0)],
    "0010101":[("011001",1),("01001",0)],
    "0011100":[("011100",1),("01100",0)],
    "1000111":[("100011",1),("10011",0)],
    "1001110":[("100110",1),("10110",0)],
    "1010101":[("101001",1),("11001",0)],
    "1011100":[("101100",1),("11100",0)],
}

#2 independent + 1 quad doubly dependent
""" xa + xb + xaxb
    Example:
        Rest + xa + xb + xaxb, xa = 1, Rest = 0
        0 + 0*1 + 0*xb + 0*1*xb = 0 -> 0 + 0*xb + 0*xb = 0
        0 + 1*1 + 1*xb + 1*1*xb = 0 -> 1*1 + 1*xb + 1*xb = 0
    "Rest + Coeff1 + Coeff2"
    Solution is [(new eq1, eq2, xa=?),(...)]: [("R1 Coeff1" "R2 Coeff2", xa), ("R1 Coeff1 Coeff2",
xa)]
        "Rest xb xaxb(1*xb)", xa=1
        "2 xb", xa = 1
        "Rest xb", xa=0
"""
case_3_2= {
    "0000111":[("00","10",1),("001",0)],
    "0001110":[("01","11",1),("001",0)],
    "0010101":[("01","11",1),("010",0)],
    "0011100":[("00","10",1),("010",0)],
    "1000111":[("10","00",1),("101",0)],
    "1001110":[("11","01",1),("101",0)],
    "1010101":[("11","01",1),("110",0)],
    "1011100":[("10","00",1),("110",0)]
}

#2 independent + 1 quad dependent
""" xa + xb + xaxc
    Example:
        Rest + xa + xb + xaxc, xa = 1, Rest = 0
        0 + 0*1 + 0*xb + 0*1*xc = 0 -> 0 + 0*xb + 0*xc = 0
        0 + 1*1 + 1*xb + 1*1*xc = 0 -> 1*1 + 1*xb + 1*xc = 0
    "Rest + Coeff1 + Coeff2"
    Solution is [(new eq1, eq2, xa=?),(...)]: [("R1 Coeff1" "R2 Coeff2", xa), (...)]
        "Rest xb xaxc(1*xc)", xa=1
        "Rest xb", xa=0 "R1 Coeff1 Coeff2"
"""
case_3_3= {
    "0000111":[("000","111",1),("001",0)],
    "0001110":[("001","110",1),("001",0)],
    "0010101":[("010","101",1),("010",0)],
    "0011100":[("011","100",1),("010",0)],
    "1000111":[("100","011",1),("101",0)],
    "1001110":[("101","010",1),("101",0)],
    "1010101":[("110","001",1),("110",0)],
    "1011100":[("111","000",1),("110",0)]
}

#should be the same as case 1
```

```
#2 independent + 1 quad independent
""" xa + xb + xcxd
   Example:
      Rest + xa + xb + xcxd, xa = 1, Rest = 0
      0 + 0*1 + 0*xb + 0*xcxd = 0 -> 0 + 0*xb + 0*xcxd = 0
      0 + 1*1 + 1*xb + 1*xcxd = 0 -> 1*1 + 1*xb + 1*xcxd = 0
   "Rest + Coeff1 + Coeff2"
   Solution is [(new eq1, eq2, xa=?),(...)]: [("R1 R2 Coeff1(2) Coeff2(2)", xa), (...)]
"""
case_3_4= {
   "0000111":[("010011",1),("00011",0)],
   "0001110":[("010110",1),("00110",0)],
   "0010101":[("011001",1),("01001",0)],
   "0011100":[("011100",1),("01100",0)],
   "1000111":[("100011",1),("10011",0)],
   "1001110":[("100110",1),("10110",0)],
   "1010101":[("101001",1),("11001",0)],
   "1011100":[("101100",1),("11100",0)],
}

#TODO what do we do about the 0 case since it just eliminates everything
#1 dependent + 2 mutually dependent quads
""" xa + xaxb + xaxc
   Example:
      Rest + xa + xaxb + xaxc, xa = 1, Rest = 0
      0 + 0*1 + 0*1*xb + 0*1*xc = 0 -> 0 + 0*xb + 0*xc = 0
      0 + 1*1 + 1*1*xb + 1*1*xc = 0 -> 1*1 + 1*xb + 1*xc = 0
   "Rest + Coeff1 + Coeff2"
   Solution is [(new eq1, eq2, xa=?),(...)]: [("R1 R2 Coeff1(2) Coeff2(2)", xa), (...)]
"""
case_3_5= {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
   "0000111":[("000","111",1),("","",0)],
   "0001110":[("001","110",1),("","",0)],
   "0010101":[("010","101",1),("","",0)],
   "0011100":[("011","100",1),("","",0)],
   "1000111":[("100","011",1),("","",0)],
    "1001110":[("101","010",1),("","",0)],
    "1010101":[("110","001",1),("","",0)],
    "1011100":[("111","000",1),("","",0)]
}

# xa + xaxb + xcxd
# 1 dependent + 1 quad dependent + 1 quad independent
""" xa + xaxb + xcxd
    Example:
        Rest + xa + xaxb + xcxd, xa = 1, Rest = 0
        0 + 0*1 + 0*1*xb + 0*xcxd = 0 -> 0 + 0*xb + 0*xcxd = 0
        0 + 1*1 + 1*1*xb + 1*xcxd = 0 -> 1*1 + 1*xb + 1*xcxd = 0
    "Rest + Coeff1 + Coeff2"
    Solution is [(new eq1, eq2, xa=?),(...)]: [("R1 R2 Coeff1(2) Coeff2(2)", xa), (...)]
"""
case_3_6= {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("000","111",1),("001",0)],
    "0001110":[("001","110",1),("010",0)],
    "0010101":[("010","101",1),("001",0)],
    "0011100":[("011","100",1),("010",0)],
    "1000111":[("100","011",1),("100","011",0)],
    "1001110":[("101","010",1),("101","010",0)],
    "1010101":[("110","001",1),("110","001",0)],
    "1011100":[("111","000",1),("111","000",0)]
}

# xa + xbxc + xdxe
# 1 independent + 2 independent quads
case_3_7= {
    "0000111":[("000","111",1),("00011",0)],
    "0001110":[("001","110",1),("00110",0)],
    "0010101":[("010","101",1),("01001",0)],
    "0011100":[("011","100",1),("01100",0)],
    "1000111":[("100","011",1),("10011",0)],
    "1001110":[("101","010",1),("10110",0)],
    "1010101":[("110","001",1),("11001",0)],
    "1011100":[("111","000",1),("11100",0)]
}

# xaxb + xaxc + xaxd
# 3 mutually dependent quads
case_3_8= {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("0000111",1),("","",0)],
    "0001110":[("0001110",1),("","",0)],
    "0010101":[("0010101",1),("","",0)],
    "0011100":[("0011100",1),("","",0)],
    "1000111":[("1000111",1),("","",0)],
    "1001110":[("1001110",1),("","",0)],
    "1010101":[("1010101",1),("","",0)],
    "1011100":[("1011100",1),("","",0)]
}

# xaxb + xaxc + xdxe
# 2 dependent quads + 1 indep quad
case_3_9= {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("0000111",1),("001",0)],
    "0001110":[("0001110",1),("010",0)],
    "0010101":[("0010101",1),("001",0)],
    "0011100":[("0011100",1),("010",0)],
    "1000111":[("1000111",1),("101",0)],
    "1001110":[("1001110",1),("110",0)],
    "1010101":[("1010101",1),("101",0)],
    "1011100":[("1011100",1),("110",0)]
}

# xaxb + xcxd + xexf
# 3 independent quads
case_3_10= {
```

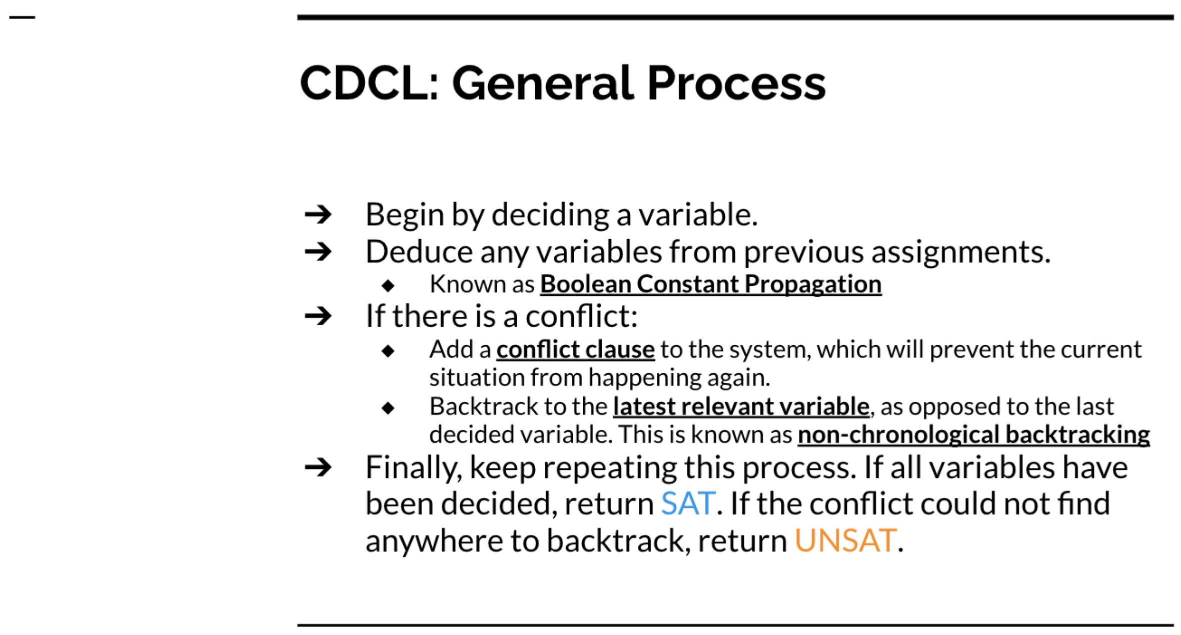
```
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("0000111",1),("00011",0)],
    "0001110":[("0001110",1),("00110",0)],
    "0010101":[("0010101",1),("01001",0)],
    "0011100":[("0011100",1),("01100",0)],
    "1000111":[("1000111",1),("10011",0)],
    "1001110":[("1001110",1),("10110",0)],
    "1010101":[("1010101",1),("11001",0)],
    "1011100":[("1011100",1),("11100",0)]
}

# xaxb + xbxc + xcxa
# triangle
case_3_11= {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("0000111",1),("001",0)],
    "0001110":[("0001110",1),("001",0)],
    "0010101":[("0010101",1),("010",0)],
    "0011100":[("0011100",1),("010",0)],
    "1000111":[("1000111",1),("101",0)],
    "1001110":[("1001110",1),("101",0)],
    "1010101":[("1010101",1),("110",0)],
    "1011100":[("1011100",1),("110",0)]
}


#xaxb + xaxc + xbxd
#2 groups of quads dependent
case_3_12 = {
# pair_1 = [(0,0,0),(1,1,1)]
# pair_2 = [(0,0,1),(1,1,0)]
# pair_3 = [(0,1,0),(1,0,1)]
# pair_4 = [(0,1,1),(1,0,0)]
    "0000111":[("0000111",1),("001",0)],
    "0001110":[("0001110",1),("010",0)],
    "0010101":[("0010101",1),("001",0)],
    "0011100":[("0011100",1),("010",0)],
    "1000111":[("1000111",1),("101",0)],
    "1001110":[("1001110",1),("110",0)],
    "1010101":[("1010101",1),("101",0)],
    "1011100":[("1011100",1),("110",0)]
}

# xa + xbxc + xbxd
case_3_13 = {
    "0000111":[("000","111",1),("00011",0)],
    "0001110":[("001","110",1),("00110",0)],
    "0010101":[("010","101",1),("01001",0)],
    "0011100":[("011","100",1),("01100",0)],
    "1000111":[("100","011",1),("10011",0)],
    "1001110":[("101","010",1),("10110",0)],
    "1010101":[("110","001",1),("11001",0)],
    "1011100":[("111","000",1),("11100",0)]
}
```

Appendix M: CDCL History



What is a CDCL?

- Architecture for Boolean SAT solver algorithms
- Very successful in industry
- Has improved greatly over the past 20 years



CDCL: General Process

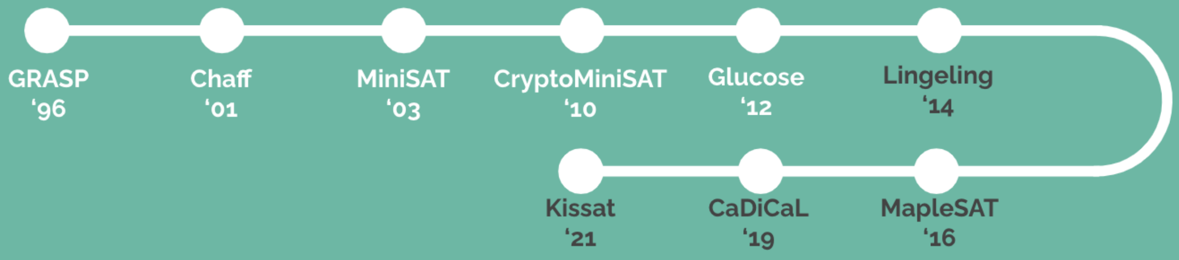
➔ Begin by deciding a variable.

➔ Deduce any variables from previous assignments.
  - Known as **Boolean Constant Propagation**

➔ If there is a conflict:
  - Add a **conflict clause** to the system, which will prevent the current situation from happening again.
  - Backtrack to the **latest relevant variable**, as opposed to the last decided variable. This is known as **non-chronological backtracking**

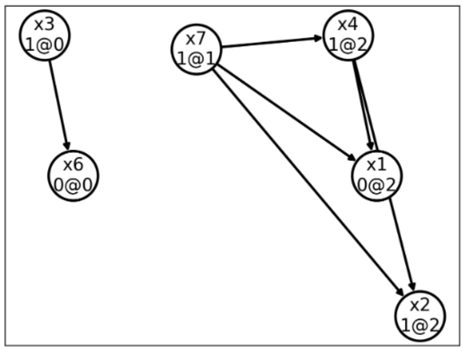
➔ Finally, keep repeating this process. If all variables have been decided, return SAT. If the conflict could not find anywhere to backtrack, return UNSAT.

Evolution of CDCL



GRASP '96

The introduction of CDCL

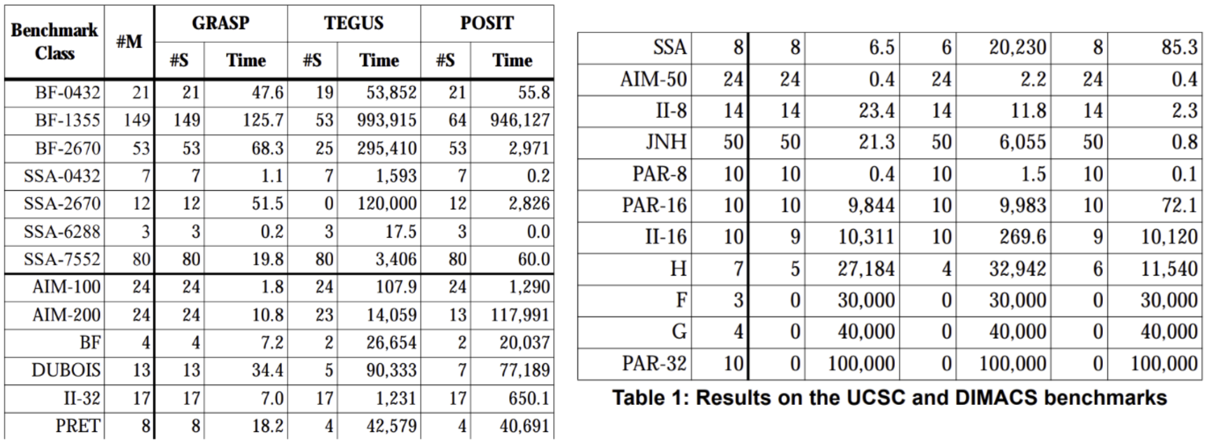


Example of an Implication Graph

GRASP was the first CDCL algorithm, created by João P. Marques Silva and Karem A. Sakallah in 1996.

GRASP works by deciding the most involved variables, deducing any variables from current assignments, and adding conflict clauses and non chronological backtracking on conflicts.

Compared to POSIT and TEGUS, state-of-the-art algorithms at the time, GRASP significantly outperformed each one on most tests.



Benchmark Class	#M	GRASP		TEGUS		POSIT	
		#S	Time	#S	Time	#S	Time
BF-0432	21	21	47.6	19	53,852	21	55.8
BF-1355	149	149	125.7	53	993,915	64	946,127
BF-2670	53	53	68.3	25	295,410	53	2,971
SSA-0432	7	7	1.1	7	1,593	7	0.2
SSA-2670	12	12	51.5	0	120,000	12	2,826
SSA-6288	3	3	0.2	3	17.5	3	0.0
SSA-7552	80	80	19.8	80	3,406	80	60.0
AIM-100	24	24	1.8	24	107.9	24	1,290
AIM-200	24	24	10.8	23	14,059	13	117,991
BF	4	4	7.2	2	26,654	2	20,037
DUBOIS	13	13	34.4	5	90,333	7	77,189
II-32	17	17	7.0	17	1,231	17	650.1
PRET	8	8	18.2	4	42,579	4	40,691
SSA	8	8	6.5	6	20,230	8	85.3
AIM-50	24	24	0.4	24	2.2	24	0.4
II-8	14	14	23.4	14	11.8	14	2.3
JNH	50	50	21.3	50	6,055	50	0.8
PAR-8	10	10	0.4	10	1.5	10	0.1
PAR-16	10	10	9,844	10	9,983	10	72.1
II-16	10	9	10,311	10	269.6	9	10,120
H	7	5	27,184	4	32,942	6	11,540
F	3	0	30,000	0	30,000	0	30,000
G	4	0	40,000	0	40,000	0	40,000
PAR-32	10	0	100,000	0	100,000	0	100,000

Table 1: Results on the UCSC and DIMACS benchmarks

Chaff '01 - Watch Variables

Chaff brings a few improvements to the CDCL architecture.

First, Chaff introduces the concept of watch variables for BCP. Since a clause of N literals cannot become a unit clause until N-1 literals are decided, two variables in the clause can be "watched", meaning that the clause will be checked for BCP/Conflicts only when one of the two watch variables are determined. This speeds up the BCP process (which is typically 80-90% of the execution time for CDCL!)

Chaff '01 - VSIDS

Second, Chaff introduces the Variable State Independent Decaying Sum, its decision algorithm. All variables are assigned a score based on how many times it appears in each clause initially, and its score increases as more clauses are added. Additionally, each score is halved after a certain period of time. This makes the decision algorithm prioritize added clauses over the initial clauses.

Finally, Chaff supports restarts, which erases all decisions. These restarts preserve learnt clauses, and these restarts have shown to improve performance a bit.

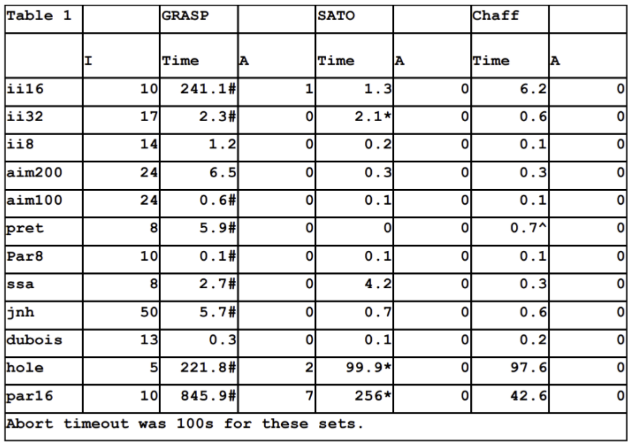


Table 1		GRASP		SATO		Chaff	
	I	Time	A	Time	A	Time	A
ii16	10	241.1#	1	1.3	0	6.2	0
ii32	17	2.3#	0	2.1*	0	0.6	0
ii8	14	1.2	0	0.2	0	0.1	0
aim200	24	6.5	0	0.3	0	0.3	0
aim100	24	0.6#	0	0.1	0	0.1	0
pret	8	5.9#	0	0	0	0.7^	0
Par8	10	0.1#	0	0.1	0	0.1	0
ssa	8	2.7#	0	4.2	0	0.3	0
jnh	50	5.7#	0	0.7	0	0.6	0
dubois	13	0.3	0	0.1	0	0.2	0
hole	5	221.8#	2	99.9*	0	97.6	0
par16	10	845.9#	7	256*	0	42.6	0
Abort timeout was 100s for these sets.							

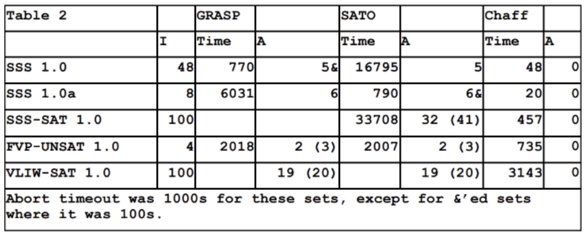


Table 2		GRASP		SATO		Chaff	
	I	Time	A	Time	A	Time	A
SSS 1.0	48	770	5&	16795	5	48	0
SSS 1.0a	8	6031	6	790	6&	20	0
SSS-SAT 1.0	100			33708	32 (41)	457	0
FVP-UNSAT 1.0	4	2018	2 (3)	2007	2 (3)	735	0
VLIW-SAT 1.0	100		19 (20)		19 (20)	3143	0
Abort timeout was 1000s for these sets, except for &'ed sets where it was 100s.							

MiniSAT ‘03

Niklas Eén, Niklas Sörensson

Algorithm wise, MiniSAT does not offer any new additions over Chaff. However, the paper offers a detailed implementation of the algorithm in C++, which could potentially be useful if we implement a CDCL algorithm on an FPGA.

—

Clause Deletion

The MiniSAT paper also offers a good explanation on clause deletion, a feature introduced in the Chaff algorithm. Basically, after a restart, some clauses will be deleted to limit the search space. Clauses are tracked similar to the VSIDS decision system, where activity is tracked low-activity clauses are ejected from the clause database. The only exception are locked clauses, which cannot be removed from the system at its current state.

CryptoMiniSAT ‘10

CryptoMiniSAT is built off of MiniSAT, but it introduced A LOT of changes (MiniSAT’s line count is ~2000, while CMSAT’s is ~50000)

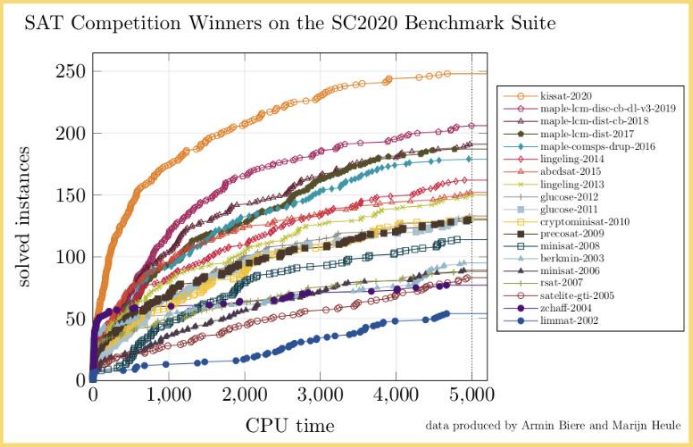
Unfortunately, there isn’t really any source of information that explains these new features in detail. The only paper found described these features briefly.

First, CMSAT handles XOR clauses independently from other clauses. A “forest” of Binary XOR clauses is constructed, which determines if variables should be equivalent or non-equivalent to others. Regular XOR clauses are also periodically XORed together to create binary clauses.

Next, regular clauses are strengthened, subsumed? and variables are eliminated after each conflict analysis. Also, binary clauses are solved before any other clause.

CryptoMiniSAT ‘10

Finally, CMSAT detects sub-problems in the system, and performs a separate sub-search on those problems. If a subproblem fails, then the whole system is UNSAT.



This graph illustrates the performance of different SAT solvers with the 2020 SAT Competition suite, with CMSAT being yellow. It clearly is outclassed by newer algorithms, and since it is much more complex than the previous algorithms, it may not be worthwhile to research thoroughly if newer algorithms are better documented.

—

Glucose '12

Glucose introduces the concept of **Linear Block Distance (LBD)**, which measures the amount of **decision levels** in a **clause**. Clauses with a **lower** LBD are more useful to the solver, and clauses with a LBD of 2 are known as **glue clauses**. If a clause has too high an LBD, it is **removed** from the system.

Glucose also introduces a restart policy based off of LBD. Basically, it looks at the N most recent clauses in the **current buffer**, computes the average LBD, and compares it with the **global buffer**'s average LBD. If it is greater than the global, it will **restart the search, but keep the learned clauses.** The restart may be **postponed** if it it is close to a decision.

One topic addressed by Glucose is **parallelization**. Multiple different CDCL solvers attempt to solve the system, but they each have the ability to **share clauses** with the others. Since importing the wrong clause could hurt the other solvers, only **unary, glue clauses, or clauses that "survived" more than two conflict analyses** are shared with other solvers. An imported clause is also not used for deduction until it is falsified, which **promotes** it to a regular clause.

Finally, **sequential solving** was covered by Glucose. This is the idea of assuming a variable's value from the learned clauses (somehow?) and removing initial clauses no longer necessary as a result of the assumption.