FPGA BASED SAT SOLVER PROJECT

A CORE COURSE PROJECT REPORT

Submitted by

S V PRASATH (22EC097)

PRAVEEN G (22EC088)

This project report is submitted as part of the requirements for the completion of the Core Course Project of the curriculum

ELECTRONICS AND COMMUNICATION ENGINEERING

CHENNAI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution, Affiliated to Anna University, Chennai)

OCTOBER 2024

CHENNAI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution, Affiliated to Anna University, Chennai)

BONAFIDE CERTIFICATE

Certified this project report **“FPGA BASED SAT-SOLVER PROJECT”** is a bonafide work of **“S V PRASATH (22EC097) and PRAVEEN G(22EC088)”** who carried out this Core Course project work under my supervision.

Dr.P.Sureshkumar M.E, Ph.D	Dr R Sureshkumar M.E,Ph.D
Professor and Head,	Assistant Professor,
Department of Electronics and Communication Engineering	Department of Electronics and Communication Engineering
Chennai Institute of Technology	Chennai Institute of Technology
Chennai - 600069	Chennai - 600069

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

We express our gratitude to our Chairman **Shri.P.SRIRAM** and all trust members of Chennai institute of technology for providing the facility and opportunity to do this project as a part of our undergraduate course.We are grateful to our Principal **Dr.A.RAMESH M.E, Ph.D.** for providing us the facility and encouragement during the course of our work.We sincerely thank our Head of the Department, **Dr. P SURESH KUMAR M.E., Ph.D** Department of Electronics and Communication Engineering for having provided us valuable guidance, resources and timely suggestions throughout our work.We would like to extend our thanks to our Faculty coordinators of the Department of Electronics and Communication Engineering, for their valuable suggestions throughout this project.We wish to extend our sincere thanks to all Faculty members of the Department of Electronics and Communication Engineering for their valuable suggestions and their kind cooperation for the successful completion of our project.We wish to acknowledge the help received from the Lab Instructors of the Department of Electronics and Communication Engineering and others for providing valuable suggestions and for the successful completion of the project.

S V PRASATH (22EC097)

PRAVEEN G (22EC088)

PREFACE

I , S V PRASATH and PRAVEEN G ,students of Electronics and Communication Engineering require to do a Project to enhance my knowledge. The purpose of core course Project is to acquaint the students with practical application of theoretical concept taught to me during my course period.It was a great opportunity to have close comparison of theoretical concept in practical field. This report may depict deficiencies on my part but still it is an account of my effort.The output of my analysis is summarised in a shape of Industrial Project the content of report shows the details of sequence of these. This is my Core Course Project report which I have prepared for the sake of my Third year Project. Being an engineer, I should help the society for inventing something new by utilising my knowledge which can help them to solve their problem.

ABSTRACT

This project focuses on designing and implementing an FPGA-based SAT solver, aimed at solving Boolean Satisfiability Problems (SAT) expressed in Conjunctive Normal Form (CNF). The solver is built using VHDL and programmed onto an FPGA board, enabling efficient hardware-based parallelism for solving SAT problems. The SAT solver determines whether a given CNF formula is satisfiable, and if so, returns a satisfying assignment of values to the variables.The core algorithm is based on the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, a complete algorithm that either finds a solution or proves that none exists. The FPGA design implements this algorithm, ensuring completeness while optimizing the decision-making process through advanced heuristics like Variable State Independent Decaying Sum (VSIDS) to improve performance by efficiently selecting decision variables.By utilizing the parallel processing capabilities of FPGA, this implementation aims to outperform traditional CPU-based SAT solvers, especially for large problem instances. The combination of the DPLL algorithm with FPGA's hardware optimizations enables faster traversal of the search space, making this solution particularly suitable for high-performance applications in hardware verification, circuit design, and optimization tasks.

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Boolean Satisfiability Problem (SAT) is a well-known decision problem that has significant applications in fields such as hardware verification, optimization, artificial intelligence, and cryptography. The problem requires determining whether there exists an assignment of truth values to variables that satisfies a given Boolean formula, typically represented in Conjunctive Normal Form (CNF). Despite being NP-complete, SAT solving is central to numerous computational tasks, and as such, efficient solvers are in high demand.

Traditional SAT solvers, implemented in software and run on CPUs or GPUs, often face performance limitations due to the inherent sequential nature of their processing. These limitations become more evident as the problem size grows, and solving large-scale SAT problems becomes computationally expensive. The need for faster and more efficient solvers has led to exploring hardware-based approaches, particularly the use of Field-Programmable Gate Arrays (FPGAs), which are known for their parallel processing capabilities.

In this project, we aim to design an FPGA-based SAT solver that can efficiently handle large CNF formulas by utilizing the parallel architecture of an FPGA. The solver will be based on the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, which is a well-established complete algorithm for SAT solving. A complete algorithm ensures that either a solution will be found, or it will be conclusively proven that no solution exists. By implementing the DPLL algorithm on an FPGA, the solver will be able to process multiple potential solutions simultaneously, greatly improving speed and efficiency.

To further enhance performance, the design incorporates optimization techniques such as variable selection heuristics, specifically the Variable State Independent Decaying Sum (VSIDS) heuristic, which helps in efficiently selecting the decision variables during the solving process. These optimizations reduce the search space and improve the solver's ability to handle complex instances.

The primary goal of this project is to build the SAT solver using VHDL and program it onto an FPGA board, leveraging the board's parallel processing capabilities to achieve significant performance improvements over software-based solvers. By combining a hardware-based approach with a proven SAT-solving algorithm and strategic optimizations, this FPGA-based SAT solver aims to offer a powerful and efficient solution for solving large-scale satisfiability problems.

1.2 PROJECT OBJECTIVES

The main objective of this project is to design and implement a SAT solver using VHDL, targeting an FPGA board to exploit its parallel processing capabilities. The solver will be based on the complete DPLL (Davis–Putnam–Logemann–Loveland) algorithm, ensuring that it can either find a satisfying assignment or conclusively prove that no solution exists. Additionally, the design will incorporate optimization techniques such as Variable State Independent Decaying Sum (VSIDS) heuristics to improve the efficiency of decision variable selection, thereby enhancing the solver's performance in handling complex SAT problems.

Key aspects of the project include:

1. Design and Implementation of FPGA-based SAT Solver:

Develop a SAT solver using VHDL, specifically targeting FPGA hardware to leverage its parallel processing capabilities. The solver will be capable of processing Boolean formulas in Conjunctive Normal Form (CNF) to determine their satisfiability.

2. Implementation of DPLL Algorithm:

Utilize the Davis–Putnam–Logemann–Loveland (DPLL) algorithm as the core solving mechanism. The algorithm will be implemented in a way that guarantees completeness, ensuring that the solver either finds a satisfying solution or proves that none exists.

3. Optimization of Variable Selection:

Incorporate advanced heuristics such as the Variable State Independent Decaying Sum (VSIDS) to optimize the selection of decision variables. This will enhance the solver’s efficiency by reducing the search space during the solving process.

4. Hardware Parallelism Exploitation:

Fully exploit the parallel architecture of the FPGA to enable concurrent processing of potential solutions, improving performance compared to traditional CPU-based solvers, particularly for large and complex SAT instances.

5. Scalability and Performance Testing:

Ensure that the FPGA-based SAT solver is scalable and can handle large CNF formulas. The solver will be tested for performance across varying problem sizes, comparing its efficiency and speed with software-based solvers.

6. Resource-efficient Design:

Develop an FPGA design that optimizes the use of logic elements, memory, and other hardware resources, allowing for a balance between performance and resource consumption on the FPGA board.

7. Real-world Application Potential:

Demonstrate the applicability of the FPGA-based SAT solver in practical scenarios such as hardware verification, optimization problems, and other computational tasks that rely on efficient SAT solving.

CHAPTER 2

LITERATURE REVIEW

The Boolean Satisfiability Problem (SAT) is a fundamental challenge in computer science, with far-reaching implications in various domains, including artificial intelligence, verification of hardware and software systems, and cryptography. Over the decades, researchers have developed a variety of algorithms and methodologies to address SAT, resulting in a rich body of literature that highlights both the theoretical and practical advancements in this area.

1. SAT Solving Algorithms:

The evolution of SAT solving algorithms can be traced back to the early 1970s, with the development of the DPLL algorithm by Davis, Putnam, Logemann, and Loveland. This algorithm introduced a systematic approach to explore the solution space by recursively assigning truth values to variables and backtracking when necessary. DPLL’s completeness and soundness properties have made it a cornerstone for subsequent developments in SAT solving. Various improvements and extensions to the DPLL algorithm have been proposed, such as Conflict-Driven Clause Learning (CDCL), which enhances the efficiency of the search process by learning from conflicts encountered during the solving process (Marques-Silva & Sakallah, 1996).

2. Heuristic Approaches:

Heuristic methods play a crucial role in enhancing the performance of SAT solvers. The choice of decision variables significantly affects the solver’s efficiency, and several heuristics have been developed for this purpose. The Variable State Independent Decaying Sum (VSIDS) heuristic is one such technique that dynamically selects variables based on their activity levels, promoting variables that are involved in recent conflicts (Zhang et al., 2001). The use of heuristics has shown to improve the performance of SAT solvers on large and complex instances, as demonstrated by various empirical studies comparing different heuristic strategies.

3. Parallel and Distributed SAT Solving:

The increasing complexity of SAT problems has prompted research into parallel and distributed SAT solving techniques. Parallel SAT solvers leverage multiple processors or cores to explore the solution space concurrently, thereby improving solving times for large instances. Several parallel SAT solvers, such as Plingeling and CryptoMiniSat, have been developed, employing different strategies for distributing the workload and synchronizing results (Heule et al., 2011). FPGAs (Field-Programmable Gate Arrays) have also been explored for SAT solving due to their ability to perform highly parallel computations. FPGA-based SAT solvers, like those proposed by Pomeranz et al. (2010), demonstrate significant speed-ups over traditional CPU-based solvers, particularly for specific problem classes.

4. Hardware Implementations:

The implementation of SAT solvers on hardware platforms, especially FPGAs, has gained attention due to the need for high-performance computing in real-time applications. Research in this area focuses on developing hardware-efficient representations of the DPLL algorithm and optimizing the use of FPGA resources. FPGA-based designs benefit from parallel processing capabilities, allowing multiple assignments to be evaluated simultaneously (Xiang et al., 2016). These hardware implementations have shown promising results, particularly in applications requiring rapid SAT solving, such as model checking and formal verification.

5. Applications of SAT Solving:

SAT solving techniques have found applications across various domains. In hardware verification, SAT solvers are employed to check the correctness of circuits and ensure that designs meet specified requirements (Biere et al., 2009). In artificial intelligence, SAT solvers play a vital role in planning and reasoning tasks, where they are used to solve complex decision-making problems. The ability to efficiently solve SAT problems has also implications in optimization, where SAT solvers can be used to find optimal configurations or resource allocations under certain constraints.

6. Future Directions:

The literature indicates a continued interest in improving SAT solving efficiency through algorithmic enhancements, heuristic developments, and hardware implementations. Future research may focus on integrating machine learning techniques to adaptively refine heuristics based on problem characteristics or exploring new architectures that further leverage the parallelism of modern hardware platforms. Additionally, the ongoing development of hybrid approaches that combine different solving techniques could lead to more robust and versatile SAT solvers.

In summary, the literature on SAT solving is vast and encompasses a variety of methodologies, heuristics, and applications. The integration of FPGA technology into SAT solving represents a promising direction that could further enhance solving capabilities, making it an exciting area for continued research and development.

Citations:

[1] Handbook of Satisfiability - IOS Press
[2] A Computing Procedure for Quantification Theory - Journal of the ACM
[3] SAT Solving with Conflict-Driven Clause Learning - Journal of Automated Reasoning
[4] GRASP – A New Search Algorithm for SAT - ICCAD '96
[5] Hardware-Based Algorithms for SAT - ICCD 2010
[6] A Novel FPGA-Based SAT Solver - IEEE Transactions on VLSI Systems
[7] An Efficient SAT Solver Based on the VSIDS Heuristic

CHAPTER 3
RESEARCH METHODOLOGY

This research involves developing and testing algorithms for solving satisfiability problems (SAT). In particular, two key reduction techniques, **FindUnitClause** and **FindPureLiteral**, will be examined in conjunction with branching and backtracking strategies. These methods play critical roles in reducing the complexity of SAT problems, ensuring efficient search for solutions, or determining when a formula is unsatisfiable (UNSAT). The methodology of this research is divided into several stages to ensure thorough exploration and validation of the proposed techniques.

3.1 PROBLEM IDENTIFICATION AND SCOPE

Satisfiability (SAT) is a classic computational problem where the task is to determine if a given logical formula can be satisfied. A formula is said to be satisfiable if there exists an assignment of truth values to its variables such that the entire formula evaluates to true. The importance of SAT in fields like computer science, artificial intelligence, and operations research has motivated the exploration of advanced techniques to solve SAT more efficiently.

In this research, we focus on SAT problems expressed in Conjunctive Normal Form (CNF), where a formula is a conjunction (AND) of clauses, each of which is a disjunction (OR) of literals. A literal is either a variable or its negation. The goal is to test the effectiveness of two commonly used reduction techniques, **FindUnitClause** and **FindPureLiteral**, in reducing the size of the problem before making branching decisions. Additionally, the efficiency of branching and backtracking strategies will be analyzed in solving SAT problems after reductions.

3.2 OVERVIEW OF THE SAT SOLVER TECHNIQUES

3.2.1 Reduction Techniques

- **FindUnitClause:** This method identifies clauses with exactly one unassigned literal. When such a clause is found, the value of that literal is determined automatically since it must be true for the clause to hold. If the literal is positive, it is assigned a value of 1 (true). If the literal is negative, it is assigned a value of 0 (false). The assignment simplifies the overall formula by reducing the number of literals in other clauses.
- **FindPureLiteral:** This method identifies literals that appear only in one form throughout the entire formula. For example, if a variable appears only as a positive literal (never as a negative), it is known as a pure literal. Pure literals can be assigned a truth value (1 for positive literals, 0 for negative) because assigning a different value would falsify the formula.

These reduction techniques offer two key advantages:

1. **Efficiency:** The techniques operate in linear time, providing fast identification of literals to simplify the problem.
2. **Soundness:** These reductions are logically sound, meaning they maintain the correctness of the SAT formula while reducing its size.

3.2.2 Branching and backtracking strategies

When no further reductions are possible, a decision is made to assign a value to an unassigned variable. This decision introduces a form of "guessing" into the process. The goal is to make an intelligent choice about which variable to assign and which value (0 or 1) to assign to that variable.

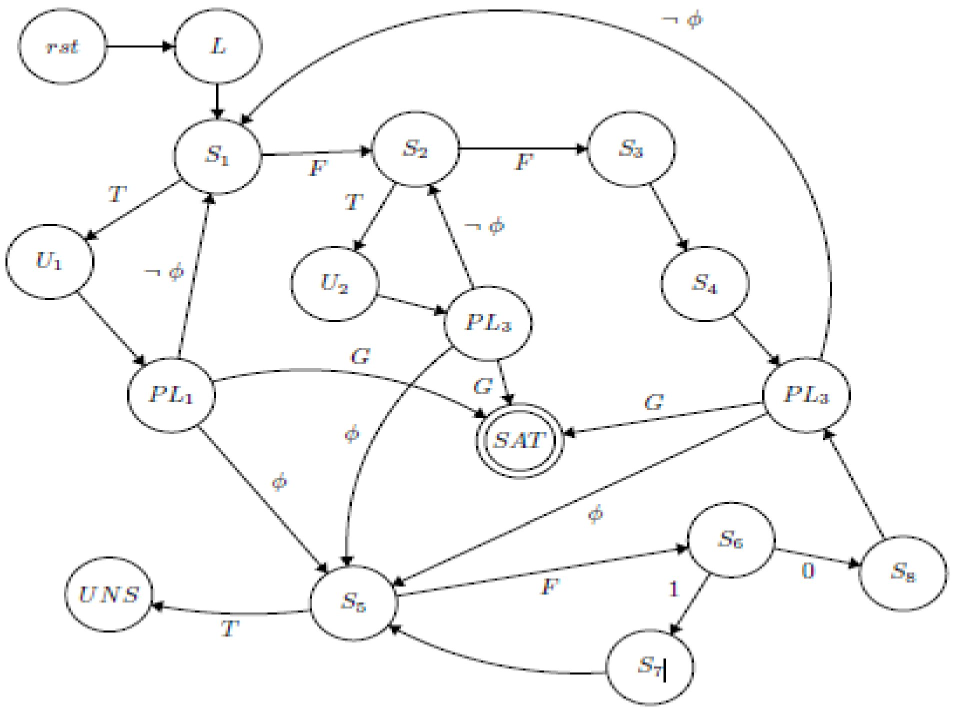


Fig 1 : State Diagram

If all clauses are satisfied, the formula is SAT, and the current variable assignment represents a solution. If a conflict occurs, meaning a clause becomes empty (i.e., all literals in the clause are falsified), backtracking is required. The most recent decision variable is revisited, and its value is inverted. If both possible values for the decision variable have been tried, the solver backtracks further, undoing decisions until a new variable can be assigned.

3.2.3 Criteria For Success

The solver concludes successfully if it finds a satisfying assignment for the formula, meaning all clauses are satisfied. If all possible assignments have been tested, and no satisfying assignment is found, the solver concludes that the formula is unsatisfiable (UNSAT).

3.3 RESEARCH METHODOLOGY STEPS

The research methodology is divided into several stages, focusing on design, implementation, testing, and analysis of the SAT solver.

3.3.1 Design of the Algorithm

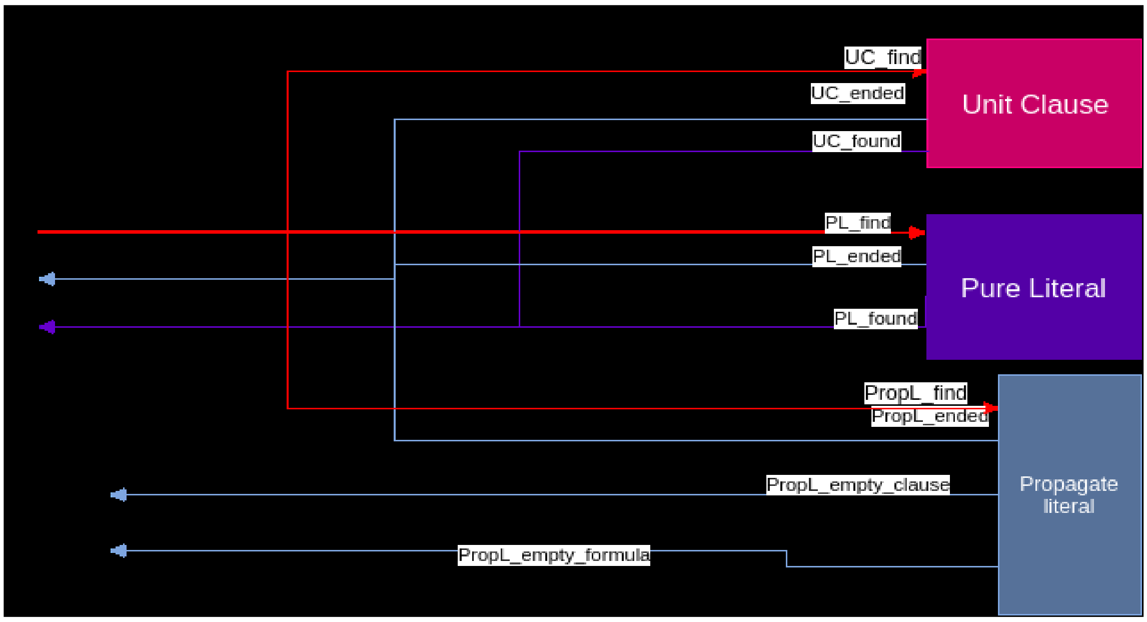


Fig 2 : Architecture

The first step involves the design of a SAT solver incorporating the **FindUnitClause** and **FindPureLiteral** reduction techniques. This stage will include defining the following components:

- **Data Structures:** Efficient data structures for storing clauses and literals will be selected to facilitate fast lookup and manipulation during reductions and decision-making.
- **Propagation Rules:** The rules for propagating variable assignments throughout the formula will be formalized.
- **Branching Heuristics:** A heuristic for selecting which variable to assign during branching will be implemented. Popular heuristics, such as the **Variable State Independent Decaying Sum (VSIDS)**, may be evaluated for incorporation into the solver.

3.3.2 Implementation

The algorithm will be implemented using **VHDL (VHSIC Hardware Description Language)**, which is a language commonly used for describing the behavior of electronic systems, including digital logic and SAT solvers. VHDL will be used to implement the SAT solver as a hardware-based solution. This approach will allow for parallelization of certain tasks and increased speed in processing large formulas, especially on Field-Programmable Gate Arrays (FPGAs).

The VHDL implementation will focus on:

- **Reduction Techniques:** Implementing the **FindUnitClause** and **FindPureLiteral** methods for detecting and applying reductions.
- **Branching and Backtracking:** Implementing decision-making for unassigned variables and the backtracking process when conflicts arise.
- **Propagation:** Efficient propagation of the effects of variable assignments throughout the formula.

3.3.3 Testing and Validation

The VHDL-based SAT solver will be tested on a variety of CNF formulas of varying sizes and complexity. These formulas will be sourced from:

- **Standard Benchmarks:** Publicly available SAT benchmarks (e.g., from SAT competitions) will be used to test the solver on real-world and challenging problems.
- **Generated Instances:** Randomly generated CNF formulas with controlled properties (e.g., number of variables, clause length) will be used to explore how the solver performs under different conditions.

Key metrics for evaluation will include:

- **Execution Time:** How long the solver takes to find a solution or conclude that the formula is UNSAT.
- **Number of Decisions and Backtracks:** How many decisions and backtracks the solver makes before reaching a conclusion.
- **Effectiveness of Reductions:** How much the formula is simplified by the reduction techniques, measured by the number of clauses and literals removed.

3.3.4 Analysis of Results

The results from the testing phase will be analyzed to determine the effectiveness of the reduction techniques and the branching/backtracking strategies. This analysis will focus on the following:

- **Impact of FindUnitClause and FindPureLiteral:** How much these reductions contribute to solving the formula efficiently.

- **Effectiveness of Branching Heuristics:** How different heuristics impact the solver's performance.
- **Scalability:** How the solver performs as the size of the CNF formula increases, particularly in terms of execution time and memory usage.

3.3.5 Improvements and Optimization

Based on the results of the analysis, improvements and optimizations will be made to the solver. Potential areas for optimization include:

- **Improved Heuristics:** Exploring more sophisticated branching heuristics to reduce the number of decisions and backtracks.
- **Parallelization:** Investigating how to leverage the parallel processing capabilities of hardware (e.g., FPGAs) to enhance performance.
- **Memory Optimization:** Reducing the memory footprint of the solver, particularly for large CNF formulas.
- This research aims to provide a deeper understanding of how reduction techniques and branching strategies impact the performance of SAT solvers, with a focus on VHDL-based implementation. By focusing on **FindUnitClause** and **FindPureLiteral**, along with efficient backtracking and decision-making processes, this research will contribute to the development of faster and more reliable SAT solving methods. The use of VHDL allows for the exploration of hardware-accelerated SAT solvers, offering potential performance gains over software-based solutions.

CHAPTER 4

PROPOSED METHODOLOGY

In this section, we explore two commonly used reduction methods in satisfiability (SAT) problems: FindUnitClause and FindPureLiteral. These methods serve to simplify the formula before employing more advanced decision-making and backtracking techniques. By detecting easily satisfiable parts of the formula early on, these methods help improve efficiency and reduce the problem size, allowing faster convergence to a solution or an unsatisfiability result (UNSAT).

4.1 REDUCTION TECHNIQUES

Reduction techniques play a pivotal role in simplifying a SAT formula by identifying and addressing literals that can be resolved deterministically. These reductions are crucial because they reduce the complexity of the problem, minimize the number of decisions, and, in some cases, avoid unnecessary backtracking. Below, we discuss two primary reduction techniques used in SAT solvers:

4.1.1 FindUnitClause:

This method identifies clauses that contain exactly one unassigned literal, known as unit clauses. A unit clause is a clause in which all but one literal has been assigned a value, leaving only one free literal that must satisfy the clause. Since there is only one way to satisfy this clause, the value of the remaining literal is determined as follows:

- If the remaining literal is positive, it must be assigned a value of 1 (true).

- If the remaining literal is negative, it must be assigned a value of 0 (false).

Once this literal is assigned, the clause is satisfied and can be removed from the formula. Additionally, the assigned literal is propagated through the formula: any clause that contains this literal is satisfied, and the literal can be removed from other clauses in which it appears. This process simplifies the remaining formula, making it easier to detect other unit clauses or pure literals.

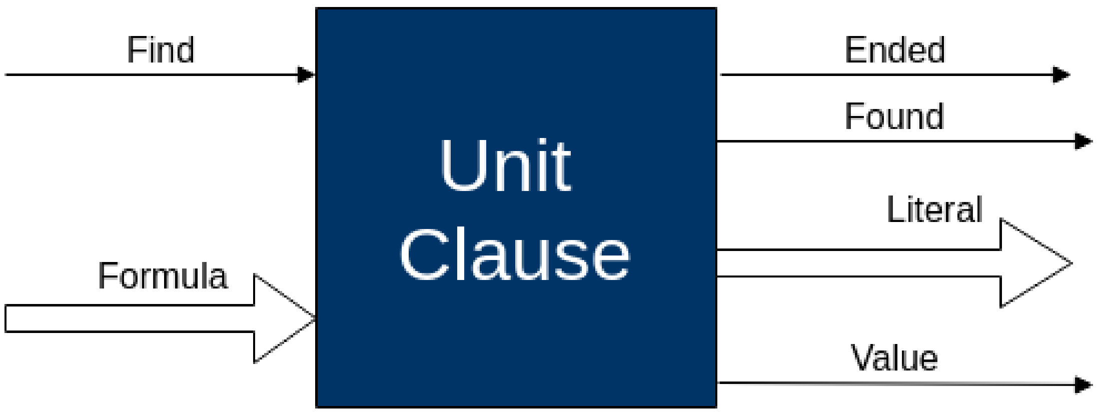


Fig 3 : Unit Clause Logic

The main advantage of this technique is its linear time complexity, meaning it can detect and resolve unit clauses quickly, leading to substantial reductions in the size of the formula. It is especially useful in large formulas where resolving trivial cases early can prevent unnecessary branching decisions.

4.1.2 FindPureLiteral:

A pure literal is a variable that appears consistently as either a positive or negative literal throughout the entire formula. For instance, if a variable appears only in its positive form **(e.g., \(x_1\))** and never in its negated form **(\(\neg x_1\)),** it is a pure literal. Similarly, if a variable appears only in its negated form, it is a negatively pure literal.

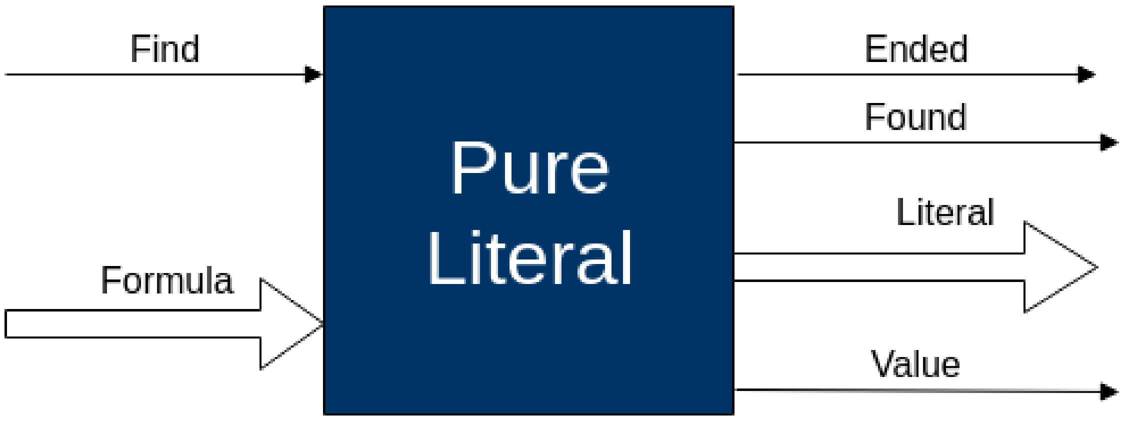


Fig 4 : Pure Literal Logic

In this reduction technique, any pure literal is assigned a truth value that satisfies all occurrences of that literal in the formula:

- If a literal is pure positive (i.e., it appears only as \(x_1\)), it is assigned the value 1 (true).

- If a literal is pure negative (i.e., it appears only as \(\neg x_1\)), it is assigned the value 0 (false).

Assigning a pure literal its truth value simplifies the formula because all clauses containing the pure literal are satisfied and can be removed. Like in the case of unit clauses, this simplification reduces the number of clauses and literals that need to be considered in subsequent decisions.

Pure literal elimination is also performed in linear time and is effective in reducing the problem's complexity without requiring any guessing or backtracking.

4.1.3 The Reduction Process and Propagation

Once the reduction techniques have been applied, the formula is simplified by eliminating clauses that are satisfied by the assigned literals. This process is referred to as propagation. As literals are assigned values, their consequences are propagated throughout the formula, leading to the removal of satisfied clauses and the simplification of other clauses by removing false literals.

For example, if a literal \(x_1 = 1\) is assigned, all clauses where \(x_1\) appears are satisfied, and those clauses are removed. Furthermore, any instance of \(\neg x_1\) in other clauses is removed, as it is now false. This process might reveal new unit clauses or pure literals, which can trigger further reductions. By propagating the effects of each assignment, the solver reduces the complexity of the problem, making it easier to find a satisfying assignment or detect a conflict.

4.1.4 Deciding the Branch and Backtracking

After applying reduction techniques, if no further simplifications are possible, the solver proceeds to the decision phase. In this phase, the solver assigns a value to an unassigned variable based on heuristics or predefined strategies. The decision-making process is crucial because it introduces branching in the search space, exploring possible assignments in an attempt to satisfy the formula.

A decision involves selecting an unassigned variable and assigning it a truth value (either 0 or 1). The choice of which variable to assign and what value to assign to it is typically guided by heuristics designed to make the most "informed" decisions. These heuristics aim to minimize the need for future backtracking by selecting variables that are likely to lead to the quickest resolution of the problem.

Once a decision is made, the assigned value is propagated through the formula as described earlier. Clauses that are satisfied by this decision are removed, and false literals are eliminated from other clauses. The formula is further simplified, potentially revealing more reductions or triggering further decisions.

4.1.5 Conflict Detection and Backtracking

A conflict occurs when all literals in a clause are falsified, meaning no assignment can satisfy the clause. This results in an empty clause, signaling that the current variable assignments have led to an unsatisfiable state. When a conflict arises, the solver must backtrack to correct previous decisions.

Backtracking is the process of undoing recent decisions and exploring alternative assignments. Specifically, the solver revisits the most recent decision variable and inverts its value (if it was assigned 1, it is reassigned to 0, and vice versa). The process of backtracking involves:

1. Erasing actions associated with the most recent decision, including variable assignments and any propagations that occurred as a result of that decision.

2. Inverting the decision variable and propagating its new value through the formula.

3. Checking for further conflicts: If another conflict occurs, the solver continues to backtrack further, undoing previous decisions until it finds a variable that can be reassigned.

4.1.6 Exhaustion of Decisions

If backtracking exhausts all possible assignments for a variable (i.e., both 0 and 1 have been tried), the solver continues backtracking to earlier decision variables. This recursive process of decision-making and backtracking continues until one of the following outcomes is reached:

- SAT (Satisfiable Formula): If all clauses are satisfied, the formula is declared SAT, and the current variable assignment represents a valid solution.

- UNSAT (Unsatisfiable Formula): If all possible assignments have been explored and no satisfying assignment has been found, the formula is declared UNSAT. This indicates that there is no way to assign truth values to the variables that satisfy all clauses.

4.1.7 Propagate Literal

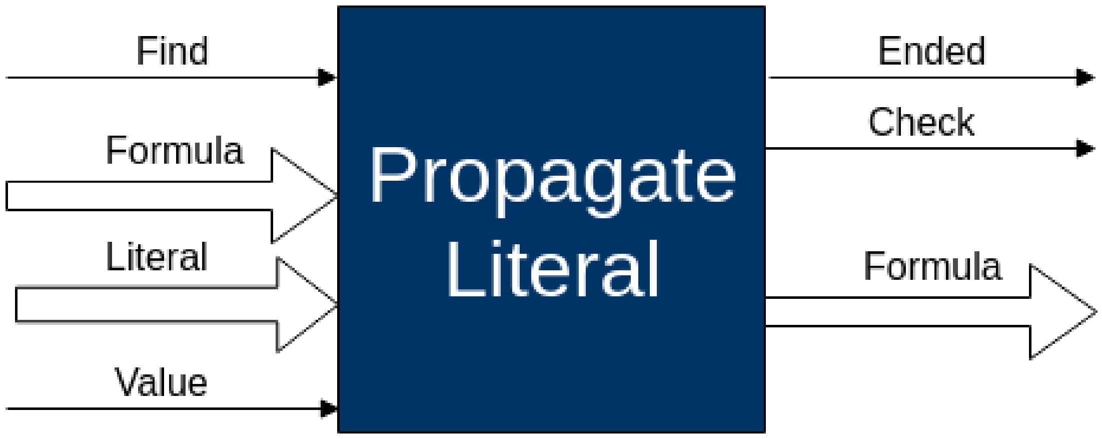
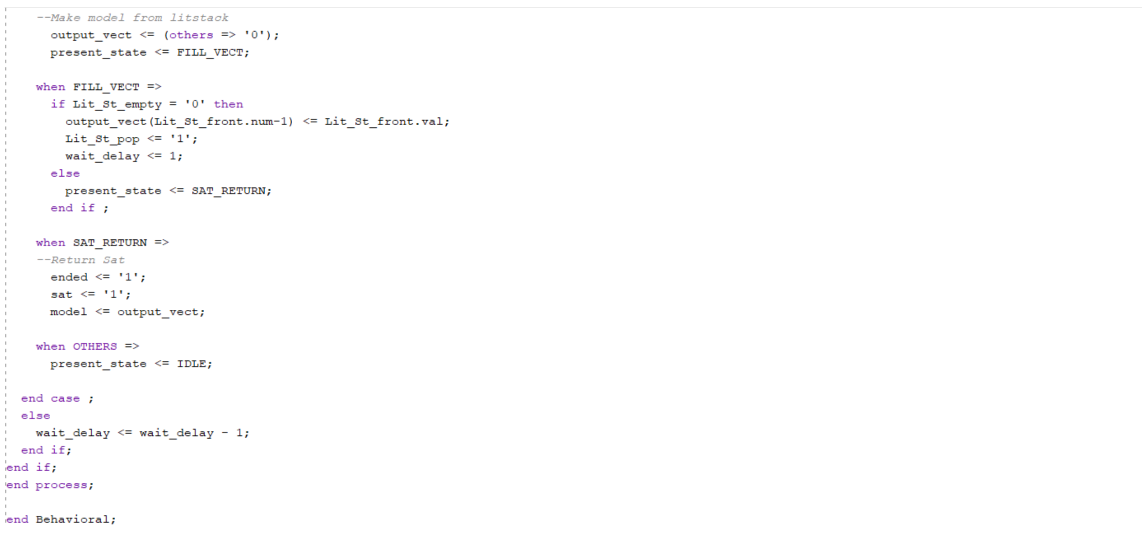


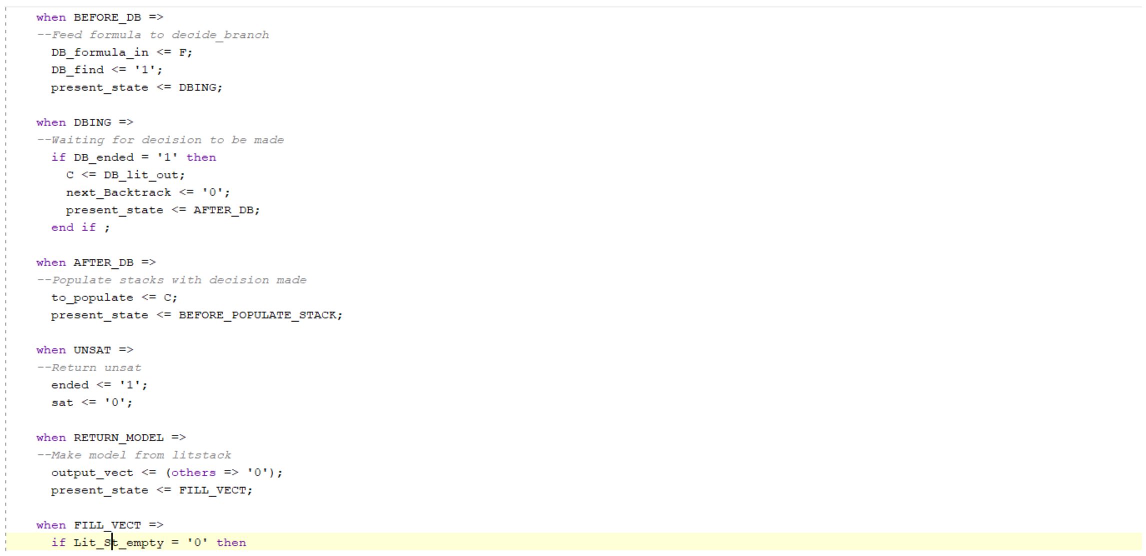
Fig 5 : Propagate Literal Logic

The **Propagate Literal** function is a key mechanism in SAT solvers that updates the formula after a literal is assigned a truth value. Upon propagation, all clauses containing the literal in a satisfying position are immediately removed from the formula, as they no longer need evaluation. Simultaneously, the negation of this literal is eliminated from any clauses where it appears, since it is now false. This often reduces the size of the clauses, possibly creating new unit clauses, which in turn triggers further propagation. The process is crucial for streamlining the search, as it efficiently narrows down the problem space, and it typically operates in linear time relative to the number of affected clauses, maintaining computational efficiency.

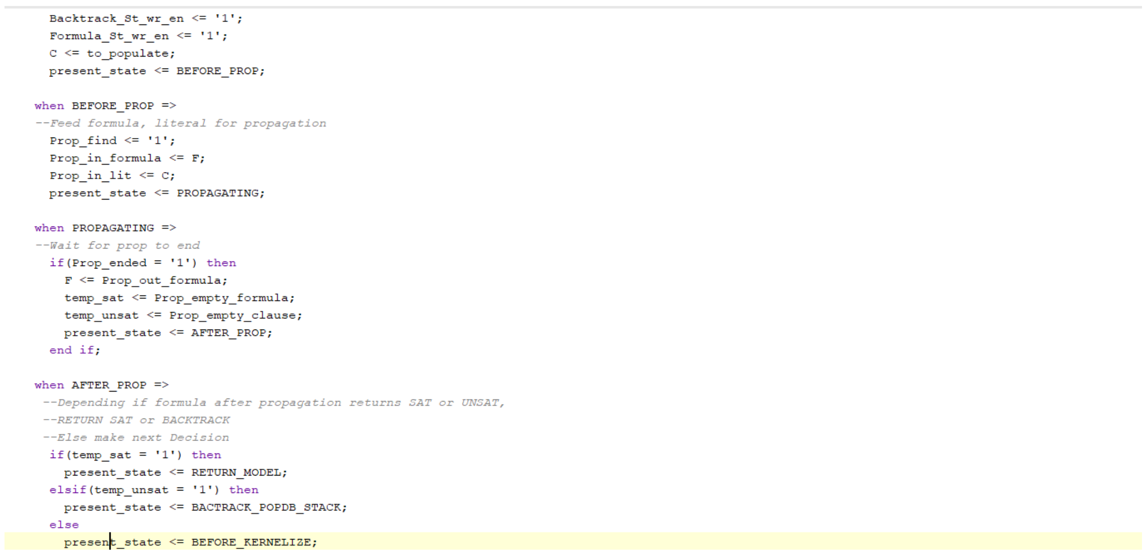
4.2 PROGRAM - VHDL



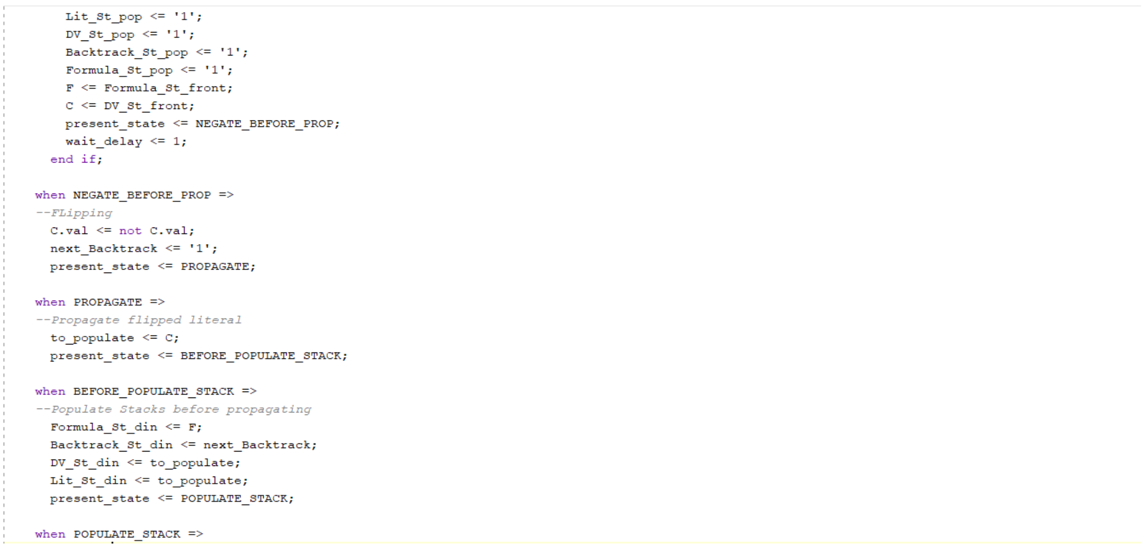
```
    --Make model from litstack
      output_vect <= (others => '0');
      present_state <= FILL_VECT;

    when FILL_VECT =>
      if Lit_St_empty = '0' then
        output_vect(Lit_St_front.num-1) <= Lit_St_front.val;
        Lit_St_pop <= '1';
        wait_delay <= 1;
      else
        present_state <= SAT_RETURN;
      end if ;

    when SAT_RETURN =>
    --Return Sat
      ended <= '1';
      sat <= '1';
      model <= output_vect;

    when OTHERS =>
      present_state <= IDLE;

  end case ;
  else
    wait_delay <= wait_delay - 1;
  end if;
end if;
end process;

end Behavioral;
```



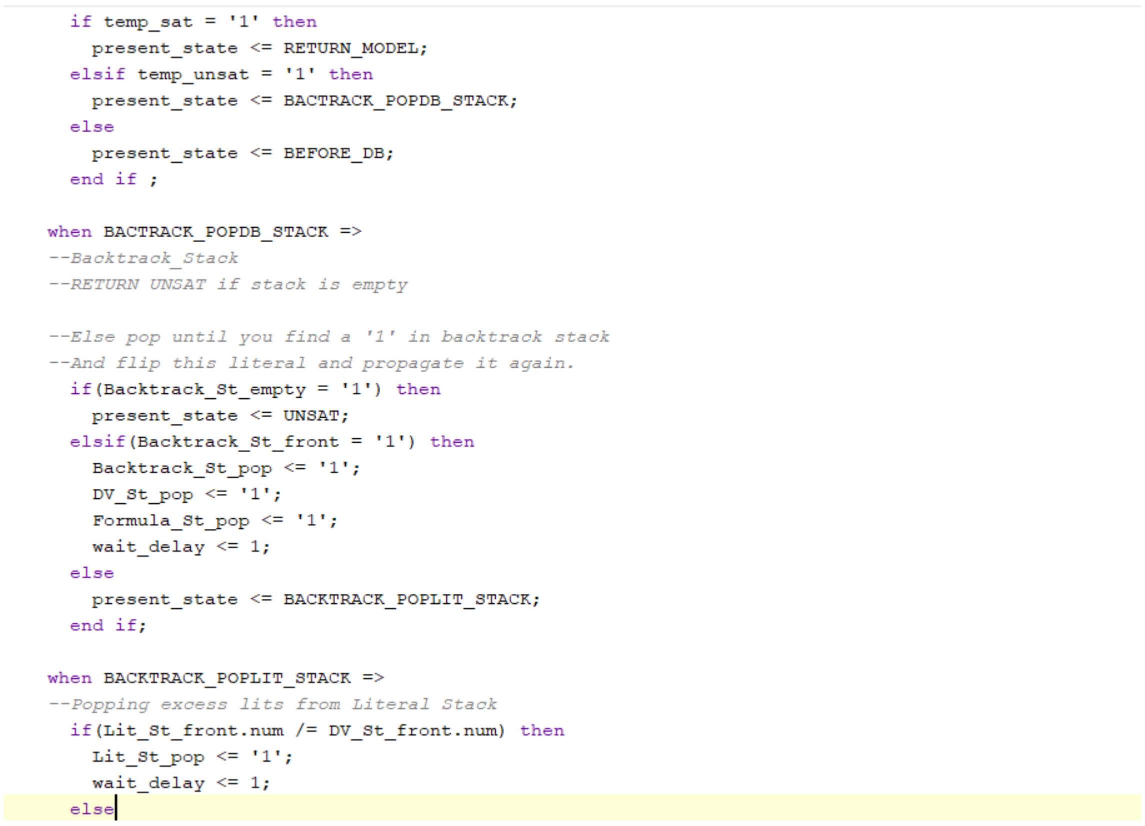
```
    when BEFORE_DB =>
    --Feed formula to decide_branch
      DB_formula_in <= F;
      DB_find <= '1';
      present_state <= DBING;

    when DBING =>
    --Waiting for decision to be made
      if DB_ended = '1' then
        C <= DB_lit_out;
        next_Backtrack <= '0';
        present_state <= AFTER_DB;
      end if ;

    when AFTER_DB =>
    --Populate stacks with decision made
      to_populate <= C;
      present_state <= BEFORE_POPULATE_STACK;

    when UNSAT =>
    --Return unsat
      ended <= '1';
      sat <= '0';

    when RETURN_MODEL =>
    --Make model from litstack
      output_vect <= (others => '0');
      present_state <= FILL_VECT;

    when FILL_VECT =>
      if Lit_St_empty = '0' then
```



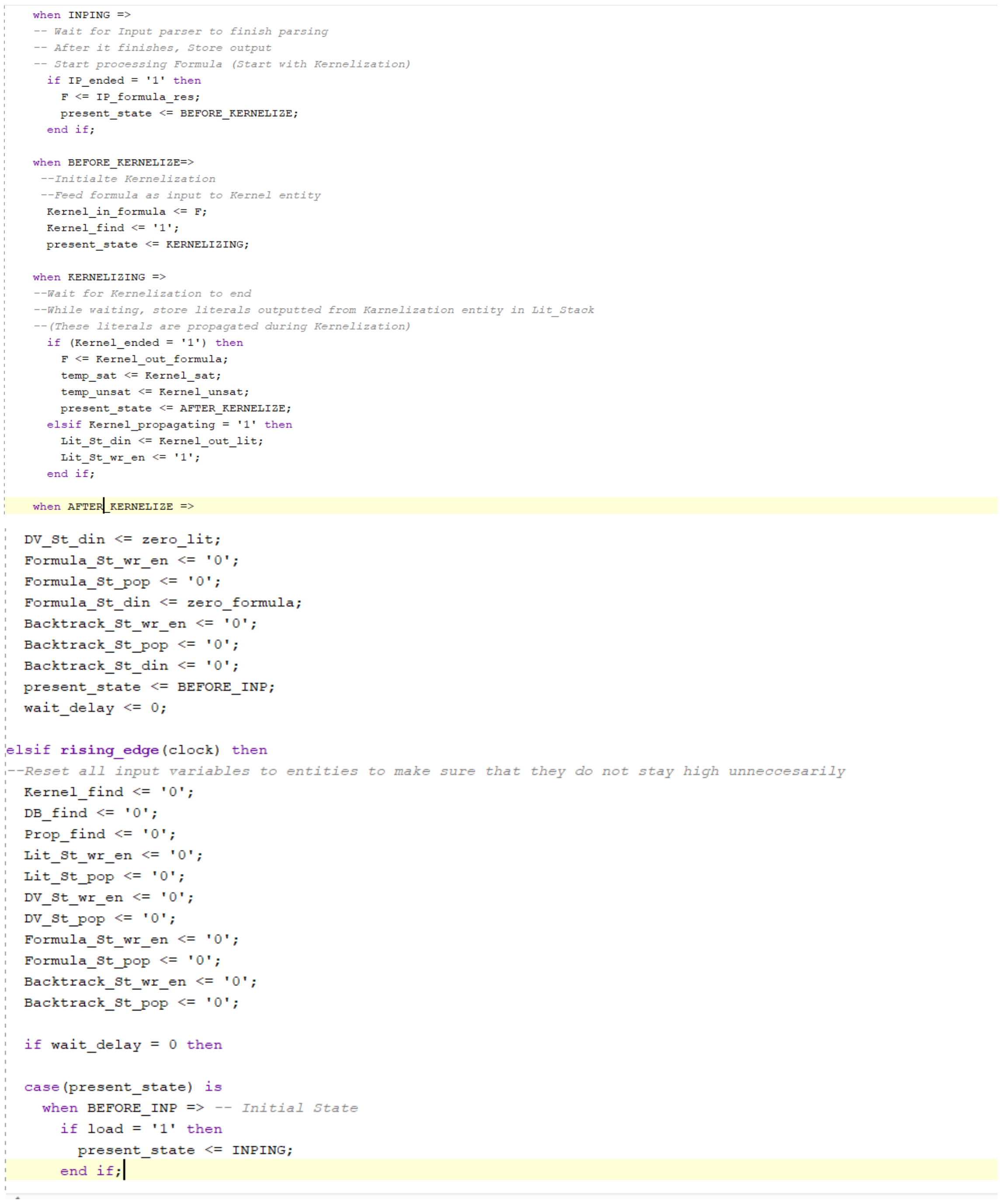
```
      Backtrack_St_wr_en <= '1';
      Formula_St_wr_en <= '1';
      C <= to_populate;
      present_state <= BEFORE_PROP;

    when BEFORE_PROP =>
    --Feed formula, literal for propagation
      Prop_find <= '1';
      Prop_in_formula <= F;
      Prop_in_lit <= C;
      present_state <= PROPAGATING;

    when PROPAGATING =>
    --Wait for prop to end
      if(Prop_ended = '1') then
        F <= Prop_out_formula;
        temp_sat <= Prop_empty_formula;
        temp_unsat <= Prop_empty_clause;
        present_state <= AFTER_PROP;
      end if;

    when AFTER_PROP =>
     --Depending if formula after propagation returns SAT or UNSAT,
     --RETURN SAT or BACKTRACK
     --Else make next Decision
      if(temp_sat = '1') then
        present_state <= RETURN_MODEL;
      elsif(temp_unsat = '1') then
        present_state <= BACTRACK_POPDB_STACK;
      else
        present_state <= BEFORE_KERNELIZE;
```



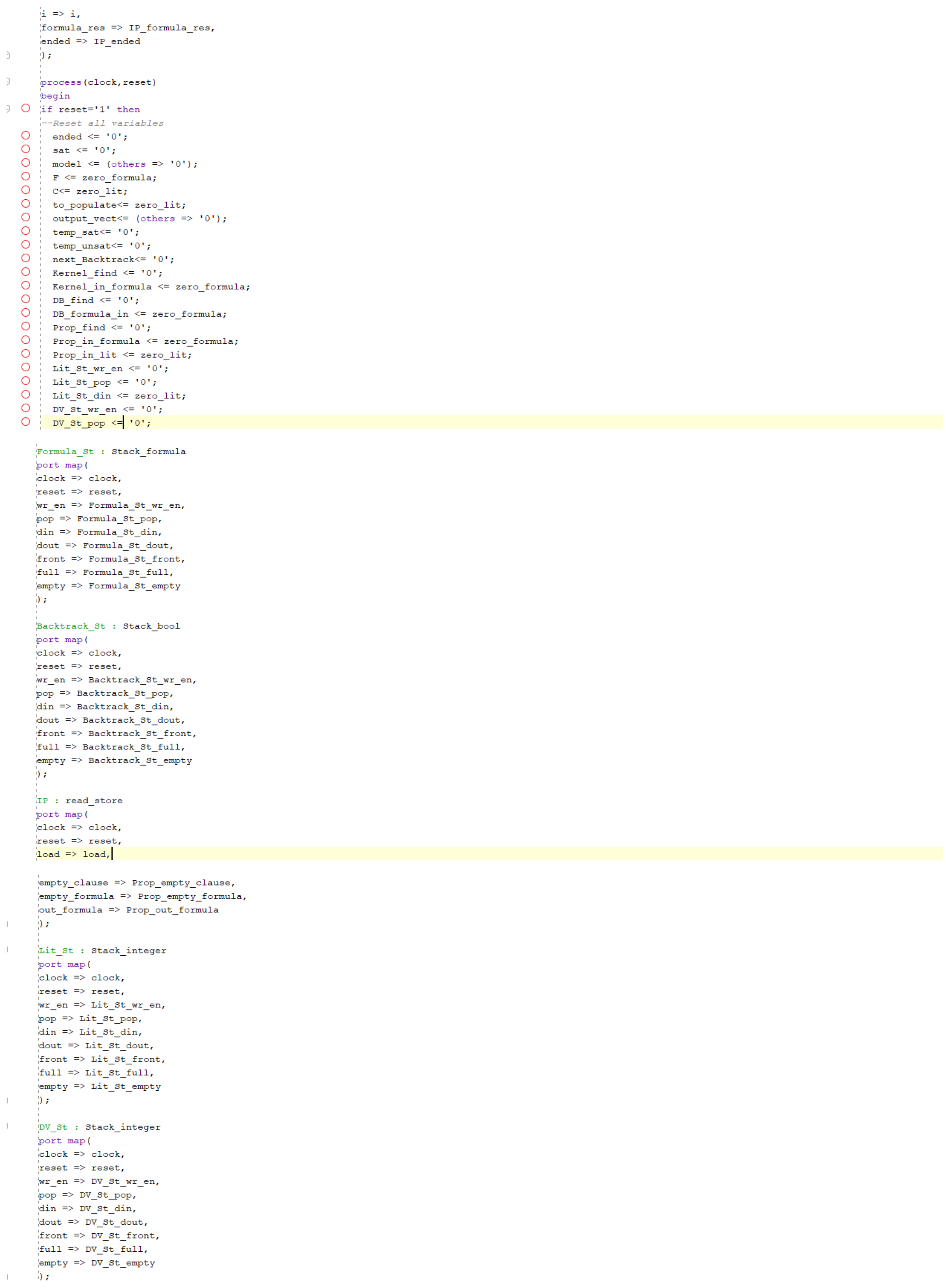
```
      Lit_St_pop <= '1';
      DV_St_pop <= '1';
      Backtrack_St_pop <= '1';
      Formula_St_pop <= '1';
      F <= Formula_St_front;
      C <= DV_St_front;
      present_state <= NEGATE_BEFORE_PROP;
      wait_delay <= 1;
    end if;

  when NEGATE_BEFORE_PROP =>
  --FLipping
    C.val <= not C.val;
    next_Backtrack <= '1';
    present_state <= PROPAGATE;

  when PROPAGATE =>
  --Propagate flipped literal
    to_populate <= C;
    present_state <= BEFORE_POPULATE_STACK;

  when BEFORE_POPULATE_STACK =>
  --Populate Stacks before propagating
    Formula_St_din <= F;
    Backtrack_St_din <= next_Backtrack;
    DV_St_din <= to_populate;
    Lit_St_din <= to_populate;
    present_state <= POPULATE_STACK;

  when POPULATE_STACK =>
```



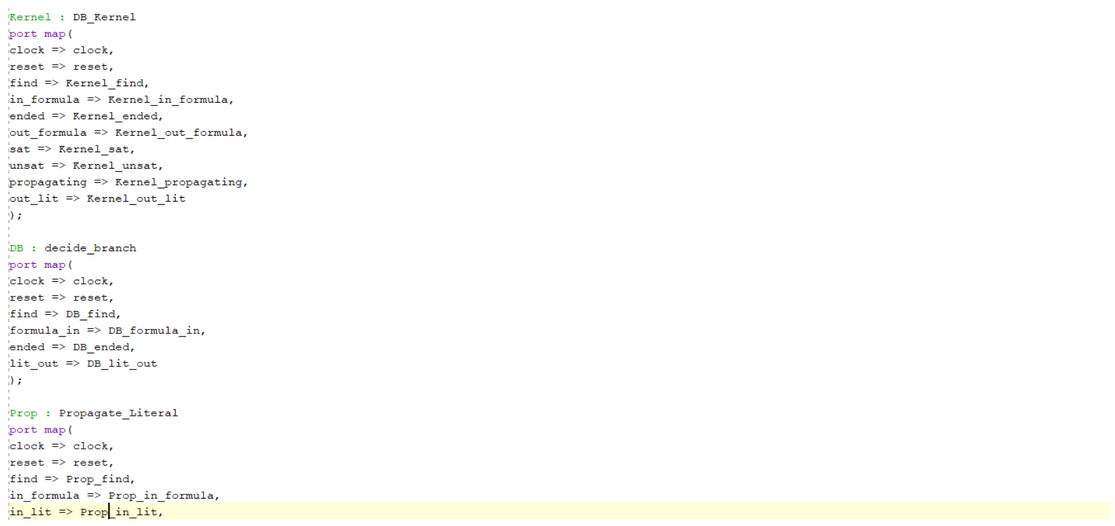
```
    if temp_sat = '1' then
      present_state <= RETURN_MODEL;
    elsif temp_unsat = '1' then
      present_state <= BACTRACK_POPDB_STACK;
    else
      present_state <= BEFORE_DB;
    end if ;

  when BACTRACK_POPDB_STACK =>
  --Backtrack Stack
  --RETURN UNSAT if stack is empty

  --Else pop until you find a '1' in backtrack stack
  --And flip this literal and propagate it again.
    if(Backtrack_St_empty = '1') then
      present_state <= UNSAT;
    elsif(Backtrack_St_front = '1') then
      Backtrack_St_pop <= '1';
      DV_St_pop <= '1';
      Formula_St_pop <= '1';
      wait_delay <= 1;
    else
      present_state <= BACKTRACK_POPLIT_STACK;
    end if;

  when BACKTRACK_POPLIT_STACK =>
  --Popping excess lits from Literal Stack
    if(Lit_St_front.num /= DV_St_front.num) then
      Lit_St_pop <= '1';
      wait_delay <= 1;
    else
```



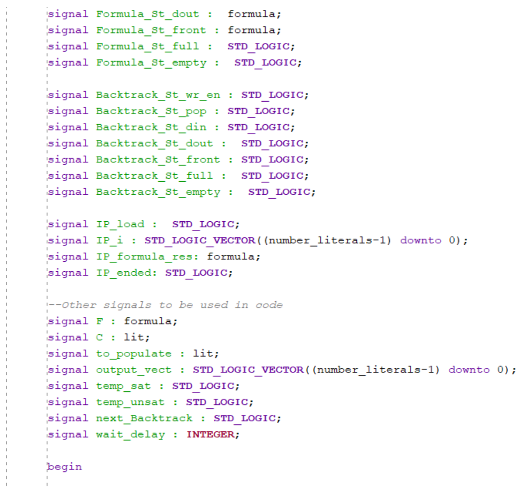
```
    when INPING =>
    -- Wait for Input parser to finish parsing
    -- After it finishes, Store output
    -- Start processing Formula (Start with Kernelization)
      if IP_ended = '1' then
        F <= IP_formula_res;
        present_state <= BEFORE_KERNELIZE;
      end if;

    when BEFORE_KERNELIZE=>
     --Initialte Kernelization
     --Feed formula as input to Kernel entity
      Kernel_in_formula <= F;
      Kernel_find <= '1';
      present_state <= KERNELIZING;

    when KERNELIZING =>
    --Wait for Kernelization to end
    --While waiting, store literals outputted from Karnelization entity in Lit_Stack
    --(These literals are propagated during Kernelization)
      if (Kernel_ended = '1') then
        F <= Kernel_out_formula;
        temp_sat <= Kernel_sat;
        temp_unsat <= Kernel_unsat;
        present_state <= AFTER_KERNELIZE;
      elsif Kernel_propagating = '1' then
        Lit_St_din <= Kernel_out_lit;
        Lit_St_wr_en <= '1';
      end if;

    when AFTER_KERNELIZE =>

  DV_St_din <= zero_lit;
  Formula_St_wr_en <= '0';
  Formula_St_pop <= '0';
  Formula_St_din <= zero_formula;
  Backtrack_St_wr_en <= '0';
  Backtrack_St_pop <= '0';
  Backtrack_St_din <= '0';
  present_state <= BEFORE_INP;
  wait_delay <= 0;

elsif rising_edge(clock) then
--Reset all input variables to entities to make sure that they do not stay high unnececesarily
  Kernel_find <= '0';
  DB_find <= '0';
  Prop_find <= '0';
  Lit_St_wr_en <= '0';
  Lit_St_pop <= '0';
  DV_St_wr_en <= '0';
  DV_St_pop <= '0';
  Formula_St_wr_en <= '0';
  Formula_St_pop <= '0';
  Backtrack_St_wr_en <= '0';
  Backtrack_St_pop <= '0';

  if wait_delay = 0 then

  case(present_state) is
    when BEFORE_INP => -- Initial State
      if load = '1' then
        present_state <= INPING;
      end if;
```



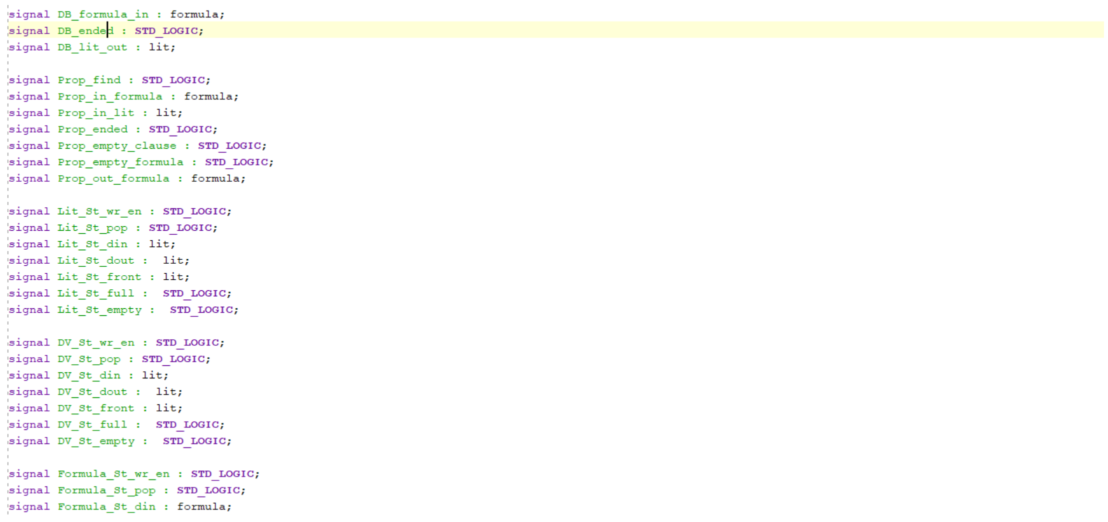
```
i => i,
formula_res => IP_formula_res,
ended => IP_ended
);

process(clock,reset)
begin
if reset='1' then
--Reset all variables
  ended <= '0';
  sat <= '0';
  model <= (others => '0');
  F <= zero_formula;
  C<= zero_lit;
  to_populate<= zero_lit;
  output_vect<= (others => '0');
  temp_sat<= '0';
  temp_unsat<= '0';
  next_Backtrack<= '0';
  Kernel_find <= '0';
  Kernel_in_formula <= zero_formula;
  DB_find <= '0';
  DB_formula_in <= zero_formula;
  Prop_find <= '0';
  Prop_in_formula <= zero_formula;
  Prop_in_lit <= zero_lit;
  Lit_St_wr_en <= '0';
  Lit_St_pop <= '0';
  Lit_St_din <= zero_lit;
  DV_St_wr_en <= '0';
  DV_St_pop <= '0';

Formula_St : Stack_formula
port map(
clock => clock,
reset => reset,
wr_en => Formula_St_wr_en,
pop => Formula_St_pop,
din => Formula_St_din,
dout => Formula_St_dout,
front => Formula_St_front,
full => Formula_St_full,
empty => Formula_St_empty
);

Backtrack_St : Stack_bool
port map(
clock => clock,
reset => reset,
wr_en => Backtrack_St_wr_en,
pop => Backtrack_St_pop,
din => Backtrack_St_din,
dout => Backtrack_St_dout,
front => Backtrack_St_front,
full => Backtrack_St_full,
empty => Backtrack_St_empty
);

IP : read_store
port map(
clock => clock,
reset => reset,
load => load,

empty_clause => Prop_empty_clause,
empty_formula => Prop_empty_formula,
out_formula => Prop_out_formula
);

Lit_St : Stack_integer
port map(
clock => clock,
reset => reset,
wr_en => Lit_St_wr_en,
pop => Lit_St_pop,
din => Lit_St_din,
dout => Lit_St_dout,
front => Lit_St_front,
full => Lit_St_full,
empty => Lit_St_empty
);

DV_St : Stack_integer
port map(
clock => clock,
reset => reset,
wr_en => DV_St_wr_en,
pop => DV_St_pop,
din => DV_St_din,
dout => DV_St_dout,
front => DV_St_front,
full => DV_St_full,
empty => DV_St_empty
);
```



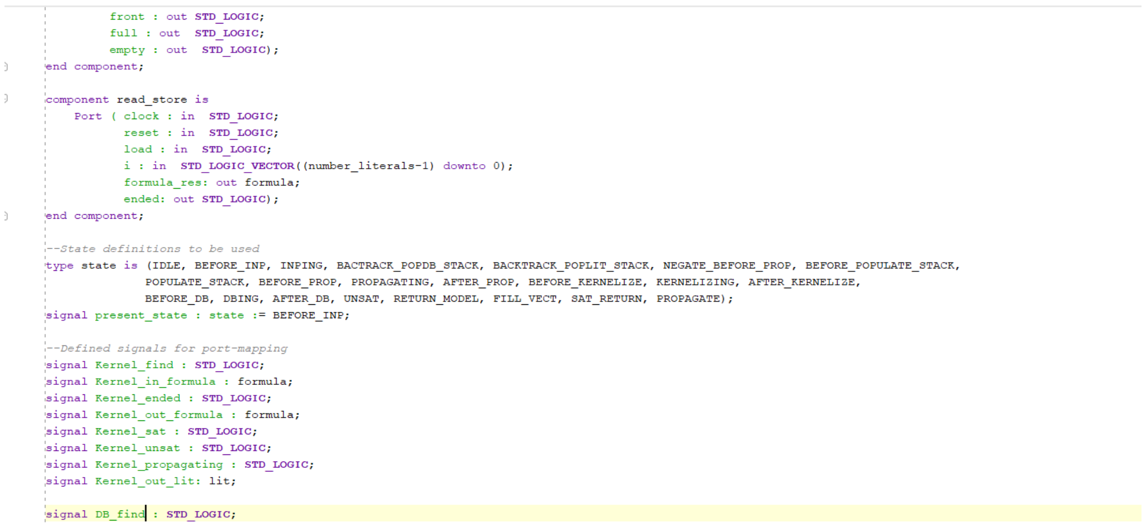
```
Kernel : DB_Kernel
port map(
clock => clock,
reset => reset,
find => Kernel_find,
in_formula => Kernel_in_formula,
ended => Kernel_ended,
out_formula => Kernel_out_formula,
sat => Kernel_sat,
unsat => Kernel_unsat,
propagating => Kernel_propagating,
out_lit => Kernel_out_lit
);

DB : decide_branch
port map(
clock => clock,
reset => reset,
find => DB_find,
formula_in => DB_formula_in,
ended => DB_ended,
lit_out => DB_lit_out
);

Prop : Propagate_Literal
port map(
clock => clock,
reset => reset,
find => Prop_find,
in_formula => Prop_in_formula,
in_lit => Prop_in_lit,
```



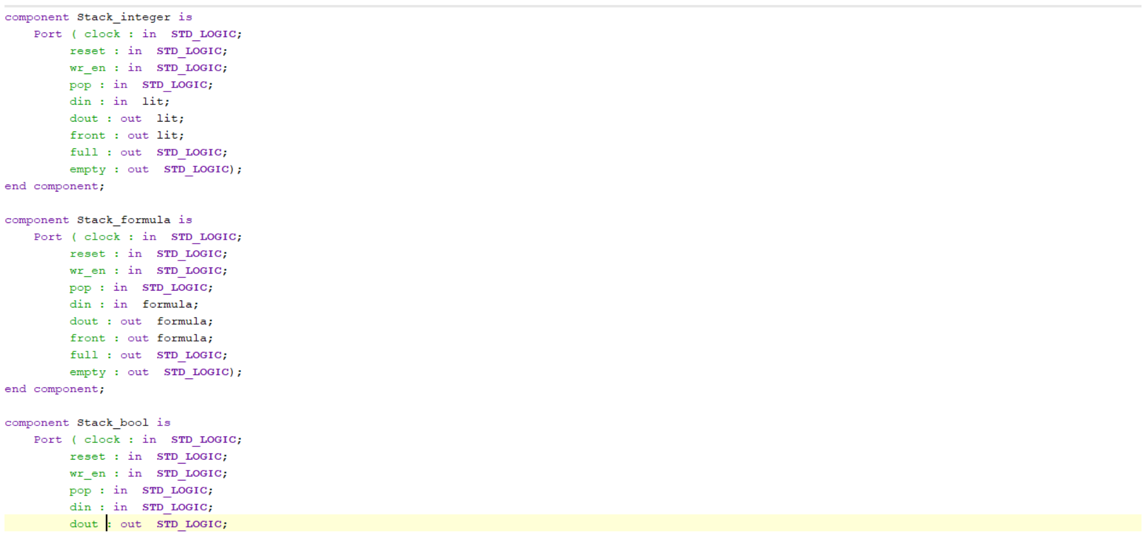
```
signal Formula_St_dout :  formula;
signal Formula_St_front : formula;
signal Formula_St_full :  STD_LOGIC;
signal Formula_St_empty :  STD_LOGIC;

signal Backtrack_St_wr_en : STD_LOGIC;
signal Backtrack_St_pop : STD_LOGIC;
signal Backtrack_St_din : STD_LOGIC;
signal Backtrack_St_dout :  STD_LOGIC;
signal Backtrack_St_front : STD_LOGIC;
signal Backtrack_St_full :  STD_LOGIC;
signal Backtrack_St_empty :  STD_LOGIC;

signal IP_load :  STD_LOGIC;
signal IP_i : STD_LOGIC_VECTOR((number_literals-1) downto 0);
signal IP_formula_res: formula;
signal IP_ended: STD_LOGIC;

--Other signals to be used in code
signal F : formula;
signal C : lit;
signal to_populate : lit;
signal output_vect : STD_LOGIC_VECTOR((number_literals-1) downto 0);
signal temp_sat : STD_LOGIC;
signal temp_unsat : STD_LOGIC;
signal next_Backtrack : STD_LOGIC;
signal wait_delay : INTEGER;

begin
```



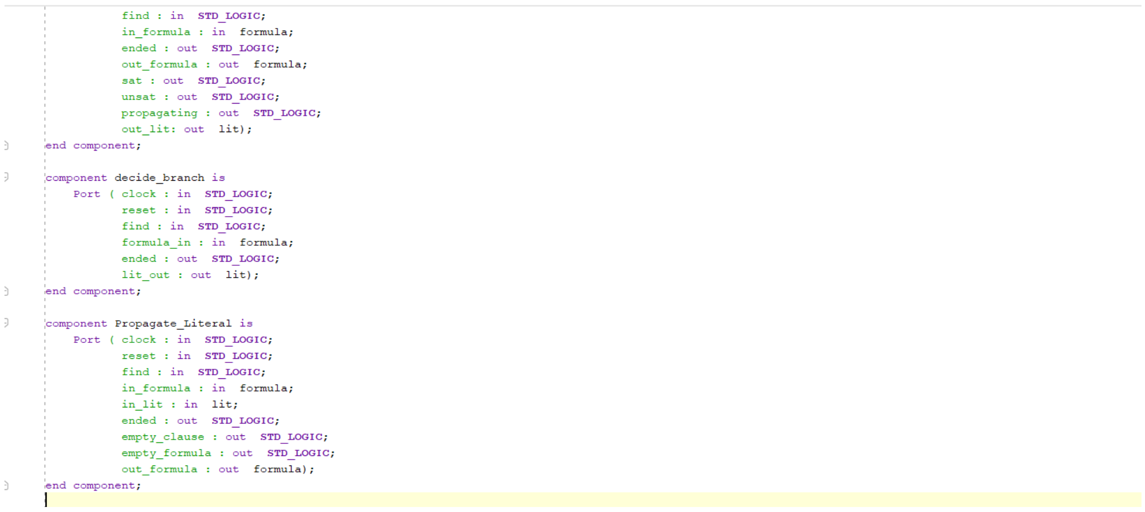
```
signal DB_formula_in : formula;
signal DB_ended : STD_LOGIC;
signal DB_lit_out : lit;

signal Prop_find : STD_LOGIC;
signal Prop_in_formula : formula;
signal Prop_in_lit : lit;
signal Prop_ended : STD_LOGIC;
signal Prop_empty_clause : STD_LOGIC;
signal Prop_empty_formula : STD_LOGIC;
signal Prop_out_formula : formula;

signal Lit_St_wr_en : STD_LOGIC;
signal Lit_St_pop : STD_LOGIC;
signal Lit_St_din : lit;
signal Lit_St_dout :  lit;
signal Lit_St_front : lit;
signal Lit_St_full :  STD_LOGIC;
signal Lit_St_empty :  STD_LOGIC;

signal DV_St_wr_en : STD_LOGIC;
signal DV_St_pop : STD_LOGIC;
signal DV_St_din : lit;
signal DV_St_dout :  lit;
signal DV_St_front : lit;
signal DV_St_full :  STD_LOGIC;
signal DV_St_empty :  STD_LOGIC;

signal Formula_St_wr_en : STD_LOGIC;
signal Formula_St_pop : STD_LOGIC;
signal Formula_St_din : formula;
```



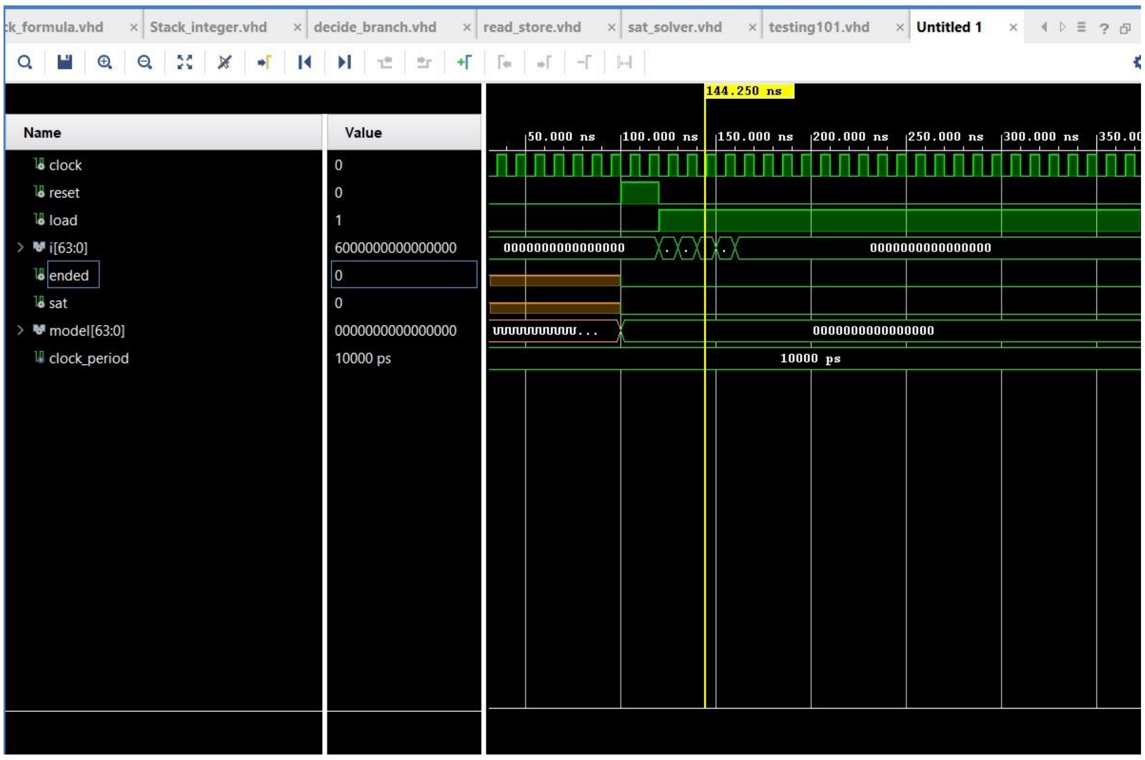
```
        front : out STD_LOGIC;
        full : out  STD_LOGIC;
        empty : out  STD_LOGIC);
end component;

component read_store is
    Port ( clock : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           load : in  STD_LOGIC;
           i : in  STD_LOGIC_VECTOR((number_literals-1) downto 0);
           formula_res: out formula;
           ended: out STD_LOGIC);
end component;

--State definitions to be used
type state is (IDLE, BEFORE_INP, INPING, BACTRACK_POPDB_STACK, BACKTRACK_POPLIT_STACK, NEGATE_BEFORE_PROP, BEFORE_POPULATE_STACK,
              POPULATE_STACK, BEFORE_PROP, PROPAGATING, AFTER_PROP, BEFORE_KERNELIZE, KERNELIZING, AFTER_KERNELIZE,
              BEFORE_DB, DBING, AFTER_DB, UNSAT, RETURN_MODEL, FILL_VECT, SAT_RETURN, PROPAGATE);
signal present_state : state := BEFORE_INP;

--Defined signals for port-mapping
signal Kernel_find : STD_LOGIC;
signal Kernel_in_formula : formula;
signal Kernel_ended : STD_LOGIC;
signal Kernel_out_formula : formula;
signal Kernel_sat : STD_LOGIC;
signal Kernel_unsat : STD_LOGIC;
signal Kernel_propagating : STD_LOGIC;
signal Kernel_out_lit: lit;

signal DB_find : STD_LOGIC;
```



```
component Stack_integer is
    Port ( clock : in  STD_LOGIC;
         reset : in  STD_LOGIC;
         wr_en : in  STD_LOGIC;
         pop : in  STD_LOGIC;
         din : in  lit;
         dout : out  lit;
         front : out lit;
         full : out  STD_LOGIC;
         empty : out  STD_LOGIC);
end component;

component Stack_formula is
    Port ( clock : in  STD_LOGIC;
         reset : in  STD_LOGIC;
         wr_en : in  STD_LOGIC;
         pop : in  STD_LOGIC;
         din : in  formula;
         dout : out  formula;
         front : out formula;
         full : out  STD_LOGIC;
         empty : out  STD_LOGIC);
end component;

component Stack_bool is
    Port ( clock : in  STD_LOGIC;
         reset : in  STD_LOGIC;
         wr_en : in  STD_LOGIC;
         pop : in  STD_LOGIC;
         din : in  STD_LOGIC;
         dout : out  STD_LOGIC;
```



```
           find : in  STD_LOGIC;
           in_formula : in  formula;
           ended : out  STD_LOGIC;
           out_formula : out  formula;
           sat : out  STD_LOGIC;
           unsat : out  STD_LOGIC;
           propagating : out  STD_LOGIC;
           out_lit: out  lit);
end component;

component decide_branch is
    Port ( clock : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           find : in  STD_LOGIC;
           formula_in : in  formula;
           ended : out  STD_LOGIC;
           lit_out : out  lit);
end component;

component Propagate_Literal is
    Port ( clock : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           find : in  STD_LOGIC;
           in_formula : in  formula;
           in_lit : in  lit;
           ended : out  STD_LOGIC;
           empty_clause : out  STD_LOGIC;
           empty_formula : out  STD_LOGIC;
           out_formula : out  formula);
end component;
```



Name	Value
clock	0
reset	0
load	1
i[63:0]	6000000000000000
ended	0
sat	0
model[63:0]	0000000000000000
clock_period	10000 ps

```
0000000000000000                0000000000000000
UUUUUUUUUUUU...                 0000000000000000
                    10000 ps
```

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

6.1 CONCLUSION

In conclusion, the application of reduction techniques like FindUnitClause and FindPureLiteral, combined with efficient propagation and backtracking strategies, forms the backbone of an effective SAT solver. These methods simplify the problem space, reduce unnecessary computation, and allow for intelligent decision-making during the search process. By progressively refining the formula and dynamically adjusting variable assignments, the solver can either efficiently arrive at a satisfying solution or conclusively determine that the formula is unsatisfiable. This structured approach not only improves performance but also ensures scalability for handling larger and more complex SAT problems.

6.2 FUTURE ENHANCEMENT

Several opportunities exist to enhance the efficiency and scalability of SAT solvers. One potential direction is the integration of machine learning techniques to predict optimal decision heuristics, enabling more informed variable selections and reducing the need for backtracking. Additionally, incorporating parallel processing can significantly speed up the search by allowing multiple branches of the decision tree to be explored simultaneously. Improving clause learning mechanisms during conflict resolution could further minimize repeated errors and guide the solver more effectively. Finally, advancements in hybrid solvers, which combine SAT techniques with other constraint-solving methods, offer promising avenues for handling even more complex, real-world problems. These enhancements could broaden the solver's applicability in fields like hardware verification, AI, and cryptography.

REFERENCES

1. Chen et al. (2010). Automated design debugging with maximum satisfiability.

2. CN103678745B (2016-09-28). Cross-platform multi-level integrated design system for FPGA.

3. CN105301984A (2016-02-03). FPGA-based power electronic simulation system and method.

4. JP2004527036A (2004-09-02). Design verification method and device for complex IC without logic simulation.

5. CN104615808B (2018-07-03). A kind of test method and reference model device of hardware computation component to be tested.

6. Bailey et al. (2019). A mixed-signal RISC-V signal analysis SoC generator with a 16-nm FinFET instance.

7. Alqudah et al. (2020). Parallel implementation of genetic algorithm on FPGA using Vivado high level synthesis.

8. CN105740206A (2016-07-06). SAT automatic integrated solver based on FPGA.

9. Leong et al. (2001). A bitstream reconfigurable FPGA implementation of the WSAT algorithm.

10. CN106777729A (2017-05-31). A kind of algorithms library simulation and verification platform implementation method based on FPGA.

11. US11023635B1 (2021-06-01). Sequence of frames generated by emulation and waveform reconstruction using the sequence of frames.

12. CN109581206B (2020-12-11). Integrated circuit fault injection attack simulation method based on partial scanning.

PO& PSO ATTAINMENT:

PO No.	GRADUATE ATTRIBUTE	ATTAINED	JUSTIFICATION
PO1	Engineering Knowledge	Yes	Demonstrates a solid understanding of engineering principles and applies them effectively to solve complex problems.
PO2	Problem Analysis	Yes	Utilizes critical thinking and analytical skills to identify, formulate, and solve engineering problems.
PO3	Design /Development of Solution	Yes	Applies creative and systematic approaches in designing and developing solutions that meet specified requirements.
PO4	Conduct Investigation of Complex Problems	Yes	Employs research methodologies and technical tools to investigate complex engineering issues and derive evidence-based solutions.
PO5	Modern Tool Usage	Yes	Utilizes current engineering tools and technologies effectively for analysis, design, and implementation in engineering practice.
PO6	The Engineer and Society	Yes	Recognizes the impact of engineering solutions on societal issues, promoting the welfare of individuals and communities.
PO7	Environment and Sustainability	Yes	Understands the importance of sustainable practices in engineering and incorporates environmental considerations in design and development.
PO8	Ethics	Yes	Demonstrates ethical behavior and professional responsibility in engineering practices and decision-making processes
PO9	Individual and Team Work	Yes	Individual: Code implementation, data preprocessing, model training. Team: Idea brainstorming, algorithm selection, result analysis, documentation, .
PO 10	Communication	Yes	Clear communication of complex concepts fosters understanding and collaboration.
PO 11	Project Management and Finance	Yes	Applies project management principles and financial practices to effectively manage engineering projects within budget and time constraints.
PO 12	Life-Long Learning	Yes	Commits to continuous personal and professional development through lifelong learning and staying updated with emerging technologies.

PSO No.	GRADUATE ATTRIBUTE	ATTAINED	JUSTIFICATION
PSO 1	To analyze, design and develop quality solutions in Communication Engineering by adapting The emerging technologies.	**YES**	Equips graduates with the skills necessary to integrate advanced technologies into communication systems, enhancing performance and functionality.
PSO 2	To innovate ideas and solutions for real time problems in industrial and domestic automation using Embedded & IOT tools.	**YES**	Encourages creativity and technical proficiency in developing innovative automation solutions that address practical challenges in various sectors.