Analog Hardware for Solving the Boolean Satisfiability Problem

Project report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

Submitted by

Sejal Singh (2010110578)

Under Supervision of

Dr. Venkatnarayan Hariharan
Department of Electrical Engineering

Department of Electrical Engineering
School of Engineering
Shiv Nadar Institution of Eminence

(April 2024)

Candidate Declaration

I hereby declare that the thesis entitled "Analog Hardware for Solving the Boolean Satisfiability Problem" was submitted for the B. Tech. degree program. This thesis has been written in my own words. I have adequately cited and referenced the sources.

Sejal Singh
(2010110578)

Certificate

It is certified that the work contained in the project report titled "Analog Hardware for Solving the Boolean Satisfiability Problem," by "Sejal Singh" has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Venkatnarayan Hariharan
Dept. of Electrical Engineering
School of Engineering
Shiv Nadar Institution of Eminence
Date: 25/04/2024

Abstract

Techniques to solve the Boolean Satisfiability problem (SAT) lend themselves to various applications in computer-aided design, circuit design, automated reasoning, formal verification, etc. The SAT problem falls under the class of NP (Non-deterministic Polynomial time) problems, hence, solving them on digital computers is intractable as the increase in the number of variables results in an exponential increase in solution time. This hurdle has led to the development of many alternative hardware solutions, one of which, is a novel technique based on the solution of suitably designed analog circuits. In this work, the implementation of a hardware-software technique for a 3-SAT solution is demonstrated by leveraging a methodology that was presented in prior work that showed a hardware-hardware implementation [1].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Boolean Satisfiability (SAT) problem is a decision problem that involves determining if an interpretation exists that satisfies a given Boolean formula. At its core, SAT is about finding a way to assign truth values (true or false) to variables in a Boolean expression, so the entire expression evaluates to true.

1.1 CNF Encoding:

CNF Encoding, short for Conjunctive Normal Form Encoding, is a pivotal technique in representing Boolean formulas, especially within the context of solving SAT problems. A formula in CNF is expressed as a conjunction (AND) of several clauses, where each clause is a disjunction (OR) of literals (a variable or its negation). This form is particularly advantageous for SAT solvers because it simplifies the logical structure of the problem, making it more amenable to algorithmic analysis and solutions [2]. For example, a 3-SAT problem, which is an SAT problem where each clause contains exactly three literals, might have a clause like C5=$\{A + B + \overline{C}\}$ showcasing the CNF structure. This encoding not only standardizes the approach for problem-solving across various computational methods, including analog and digital computing but also serves as a fundamental stepping stone for the application of efficient solving algorithms, be it through classical digital means or through innovative analog circuits that leverage the continuous dynamics of physical systems to find solutions to these NP-complete problems.

1.2 NP-Completeness:

NP-completeness is a fundamental concept in computational theory that categorizes decision problems based on their complexity and solvability. The significance of NP-completeness lies in its application across a wide array of fields, from cryptography to network design, where understanding the complexity of problems is crucial for determining the feasibility of their solutions. The SAT problem was the first to be proven NP-complete. This classification implies two main aspects: any instance of an NP problem can be transformed into an instance of SAT in polynomial time, and if a polynomial-time algorithm exists for SAT, it exists for all problems in NP. This revelation, known as the

Cook-Levin Theorem [3], underscores the fundamental importance of SAT in computational theory and the complexity of finding efficient solutions.

1.3 SAT:

SAT is pivotal in both theoretical computer science and practical computing applications. It is crucial in various domains such as electronic design automation, verification, artificial intelligence, and optimization problems. The inherent difficulty of solving SAT problems efficiently due to their NP-complete nature, coupled with their widespread applicability, has spurred significant research into finding more effective solving techniques.

1.4 Need for 3-SAT from *k*-SAT:

The significance of 3-SAT within this context lies in its ability to be reduced to any other problem in the NP class, illustrating the interconnected complexity of these problems. Additionally, the restriction to three literals per clause in 3-SAT problems presents a balance between computational tractability and complexity, making it an ideal candidate for exploring the limits of algorithmic efficiency and optimization. Moreover, 3-SAT's role is pivotal in understanding the computational landscape of k-SAT problems, where $k > 3$, as any k-SAT problem, can be polynomially reduced to a 3-SAT problem, thus establishing a uniform approach to tackling these challenges. This reduction process not only underscores the universality of 3-SAT as a core problem in computational theory but also highlights its importance in devising strategies and algorithms for solving broader classes of satisfiability problems.

Chapter 2

Literature Survey

The paper [1] covers various aspects of solving SAT problems, especially focusing on analog circuits' potential to solve NP-complete problems like k-SAT efficiently. It highlights the transition from digital to analog computation paradigms due to the limitations of Moore's law [4], exploring alternatives like quantum computing and neuromorphic computing. It delves into the principles of analog computing, where software algorithms are expressed as dynamical systems, emphasizing the deterministic continuous-time dynamical system (CTDS) proposed for solving SAT.

The paper also reviews the development of AC-SAT, an analog hardware SAT solver based on CTDS [5], offering significant speedups over digital solvers for hard k-SAT problems. It also discusses the modular and programmable nature of *AC-SAT*, designed to extend easily to larger problem sizes, and its potential implications for various fields by providing efficient solutions to NP-complete problems

2.1 Types of SAT Solvers:

SAT solvers can be broadly categorized into two types: complete and incomplete solvers. Complete solvers, such as DPLL (Davis-Putnam-Logemann-Loveland) algorithms [6] and CDCL (Conflict-Driven Clause Learning) [7], guarantee to find a solution if one exists or proves unsatisfiability. Incomplete solvers, including stochastic and heuristic-based solvers like WalkSAT [8], may find solutions more quickly but do not guarantee a solution or unsatisfiability.

Past efforts in hardware-based SAT solvers have explored FPGA-based accelerations [9], significantly speeding up the Boolean Constraint Propagation (BCP) component crucial to modern SAT solvers, reporting speed improvements over traditional software solvers like MiniSat [10]. Additionally, custom digital integrated circuits have been developed to enhance efficiency in processing the implication graph and generating conflict clauses, achieving notable performance gains. However, these hardware approaches, designed primarily for digital computation paradigms, still offer substantial room for improvement in their efficiency and speed due to the inherent limitations of adapting algorithms optimized for digital environments.

2.2 Analog Circuit Approaches for SAT Solving:

Recent advancements have introduced analog circuit approaches for solving SAT problems, offering a novel paradigm compared to traditional digital computing methods. These approaches exploit the continuous nature of analog circuits to represent and manipulate SAT problem instances, potentially overcoming the limitations of digital computational models. The paper discusses the development of efficient analog circuits that implement a deterministic Continuous Time Dynamical System (CTDS) to solve SAT problems, showing promise in handling NP-complete problems more efficiently in some cases [5]. According to the literature, analog circuits produce the best time improvements when compared to software SAT solvers and software solvers for CTDS equations.

Table 2.1: Performance comparison of `AC-SAT`, Software CTDS and MiniSAT [1]

Variables	**Time for `AC-SAT` (s)**	**Time for Software CTDS (s)**	**Time for MiniSAT (s)**
N=10	4x10^{-}9	4.40x10^{-}4	2.3x10^{-}4
N=20	7x10^{-}9	3.91x10^{-}3	2.4x10^{-}4
N=30	10^{-}8	1.62x10^{-}2	2.8x10^{-}4
N=40	1.3x10^{-}8	5.22x10^{-}2	3.1x10^{-}4
N=50	1.4x10^{-}8	1.13x10^{-}1	3.7x10^{-}4

2.3 CTDS:

The paper [5] explores a novel approach to solving k-SAT problems by mapping them into a deterministic continuous-time dynamical system. This method uniquely correlates its attractors with the k-SAT solution clusters. The system is shown to find solutions for satisfiable formulas even in traditionally hard problem regimes, such as random 3-SAT and locked occupation problems, thanks to its hyperbolic character. Despite finding solutions in polynomial continuous time, the method involves exponential fluctuations in its energy function, highlighting a balance between efficiency and the computational demand of solving NP-complete problems through analog means.

The CTDS approach as discussed embodies a significant shift from traditional digital computation methods for SAT solving by exploiting the inherent parallelism and continuous nature of analog circuits. This enables the system to potentially find solutions faster and more efficiently for certain classes of SAT problems, including those that are NP-complete, by navigating the search space in a continuous manner rather than through discrete steps. The paper highlights the feasibility and benefits of this approach, including its modular and programmable design, which allows for scalability and adaptability to different problem sizes and complexities. By converting the logical structure of SAT problems into a continuous-time dynamical system, CTDS offers a novel pathway to tackling the intrinsic difficulty of NP-complete problems. The exploration of analog circuits for SAT solving opens up new avenues for research in both the fields of computational theory and electronic design, promising advancements in the efficiency and capabilities of hardware-based solvers for complex computational challenges.

Chapter 3

Work Done

3.1 Mathematical Framework

To make SAT solvers using analog hardware, one must first find a way to express an instance of SAT as a continuous equation. This is done in [5], where a CNF is expressed as a Continuous Time Dynamical System (CTDS). After developing the continuous equation, a circuit is developed, whose behavior models the CTDS behavior ***asymptotically***, even if intermediate trajectory points of the circuit response may not necessarily model the CTDS behavior. I describe below the methodology of [5] based upon which I have built the present implementation. For the sake of continuity, I present some background, but in the interest of brevity and to focus more on our implementation, I skip the details and urge the reader to look up [5] for more details.

3.1.1 From CNF to CTDS

To solve a $k-$SAT problem one aims to find an assignment to N Boolean variables such that they satisfy the Boolean formula F. One can represent F as a CNF with M clauses and at most k literals per clause. To facilitate the CTDS theory, the following are defined in [5]:

1. A Boolean formula F with N variables $x_i \in \{0,1\}, i = 1,2,..,N$, within M clauses $C_m, m = 1,2,..,M$, formed by the disjunction of k literals (variables or their complements).

$$F = \bigwedge_m^M C_m \text{ where } C_m = (x_i \vee x_{i2}, \ldots, \vee x_{ik})$$

2. An analog variable $s_i \in [-1,1]$, associated with the Boolean variable x_i such that $s_i = -1$ when x_i is FALSE and $s_i = 1$ when x_i is TRUE.

3. An MxN matrix $C = \{c_m, i\}$, with $c_{m,i} = 1$ when x_i appears in the clause C_m, $c_{m,i} = -1$ when its negation $\bar{x}_i$ appears in the clause C_m and $c_{m,i} = 0$ when neither appears in the clause C_m.

With the above as an input specification, two functions are defined in [5] [1] as:
1. An analog function $K_m(\mathbf{s}) \in [0, 1]$,

$$K_m(\mathbf{s}) = 2^{-k} \prod_{i=1}^{N} (1 - c_{m,i} s_i) \tag{3.1}$$

2. A potential energy function $V(\mathbf{s}) \in [0, 1]$,

$$V(\mathbf{s}, \mathbf{a}) = \sum_{m=1}^{M} a_m K_m^2 \tag{3.2}$$

SAT is thus redefined as, a search in $\mathbf{s}$ for $V(\mathbf{s}) = 0$ where $\mathbf{s}$ is the global minimum for V and the solution to k-SAT. To overcome the risk of the algorithm getting stuck at a local minima, [1] introduces the auxiliary variable, $a_m \in (0, \infty)$.

From (3.1) and (3.2), [1] derives:
1. Gradient descent on V:

$$\frac{ds_i}{dt} = \sum_{m=1}^{M} a_m D_{m,i} \tag{3.3}$$

where

$$D_{m,i} = 2K_m c_{m,i} \prod_{j=1 j \neq i}^{N} (1 - c_{m,j} s_j) \tag{3.4}$$

2. Exponential growth is driven by the level of non-SAT in K_m:

$$\dot{a_m} = \frac{da_m}{dt} = a_m K_m, \; m = 1, ..., M \tag{3.5}$$

3.1.2 From CTDS to Circuit

In [1], the authors develop the CTDS as an analogous Analog Circuit in three circuit blocks: the signal dynamics circuit (SDC), the auxiliary variable circuit (AVC), and the digital verification circuit (DVC). These blocks are assembled and inputs to them are decided based on the CNF for which SAT is to be determined. *Fig.3.1* shows the block diagram of all the circuit blocks and their connections. Refer to [1] for a more detailed explanation.

3.2 Implementation

In this section I discuss how I implemented the circuits as described in [1]. My simulations of the `AC-SAT` are implemented at the transistor level in Cadence Virtuoso [11] using a 45nm technology model. The supply voltage is $V_{DD} = 1.2V$ and the transistor sizes are

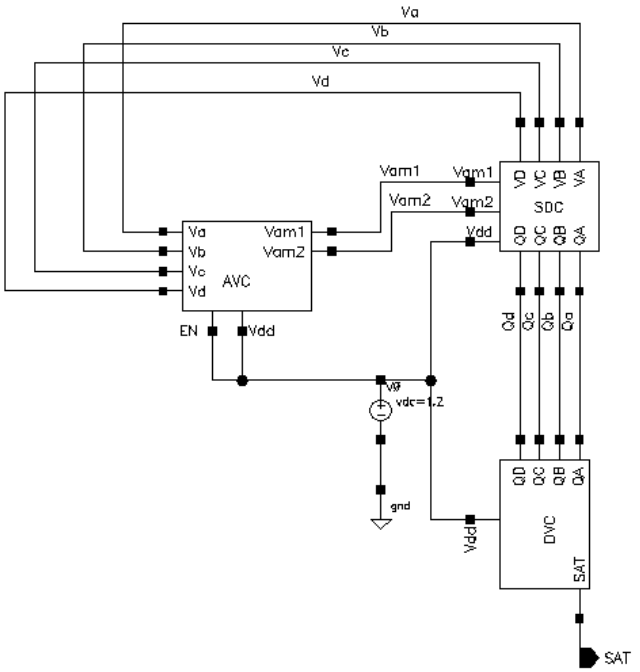

Figure 3.1: Block diagram based on [1].

set to $L = 45nm$ and $W = 120nm$ except in certain circuits where sizes are selected according to the transistor's specific role. This is the first step in coming up with a generalized netlisting script. After manually implementing the circuit for set CNFs, I ensure that it simulates correctly, after which, I acquire its netlists. These netlists are then templated to automate the creation of other netlists as per the requirements of the CNF.

3.2.1 Signal Dynamics Circuit (SDC)

For a 3-SAT problem, one can infer from (3.5) that,

$$D_{m,i} = 2^{-2} \text{ x } c_{m,i}(1 - c_{m,i}s_i)(1 - c_{m,i2}s_{i2})^2(1 - c_{m,i3}s_{i3})$$

$$D_{m,i} = \begin{cases} 2^{-2}(1 - \mathrm{s}_i)(1 - \mathrm{c}_{m,i2}\mathrm{s}_{i2})^2(1 - \mathrm{c}_{m,i3}\mathrm{s}_{i3})^2 & \text{if} \mathrm{c}_{m,i} = 1 \\ 0 & \text{if} \mathrm{c}_{m,i} = 0 \\ 2^{-2}(-1 - \mathrm{s}_i)(1 - \mathrm{c}_{m,i2}\mathrm{s}_{i2})^2(1 - \mathrm{c}_{m,i3}\mathrm{s}_{i3})^2 & \text{if} \mathrm{c}_{m,i} = -1 \end{cases} \tag{3.6}$$

$$C\frac{dV_i}{dt} = \sum_{m=1}^{M} I_{m,i} \tag{3.7}$$

$$I_{m,i} = \begin{cases} (\mathrm{V}_{DD} - \mathrm{V}_i)/(\mathrm{R}_{m,i}||\mathrm{R}_{am} + \mathrm{R}_{m,i_2} + \mathrm{R}_{m,i_3}) & \text{if} \mathrm{c}_{m,i} = 1 \\ 0 & \text{if} \mathrm{c}_{m,i} = 0 \\ (\mathrm{GND} - \mathrm{V}_i)/(\mathrm{R}_{m,i}||\mathrm{R}_{am} + \mathrm{R}_{m,i_2} + \mathrm{R}_{m,i_3}) & \text{if} \mathrm{c}_{m,i} = -1 \end{cases} \tag{3.8}$$

From (3.7), [1] develops a circuit analogy with the current-voltage relations described in

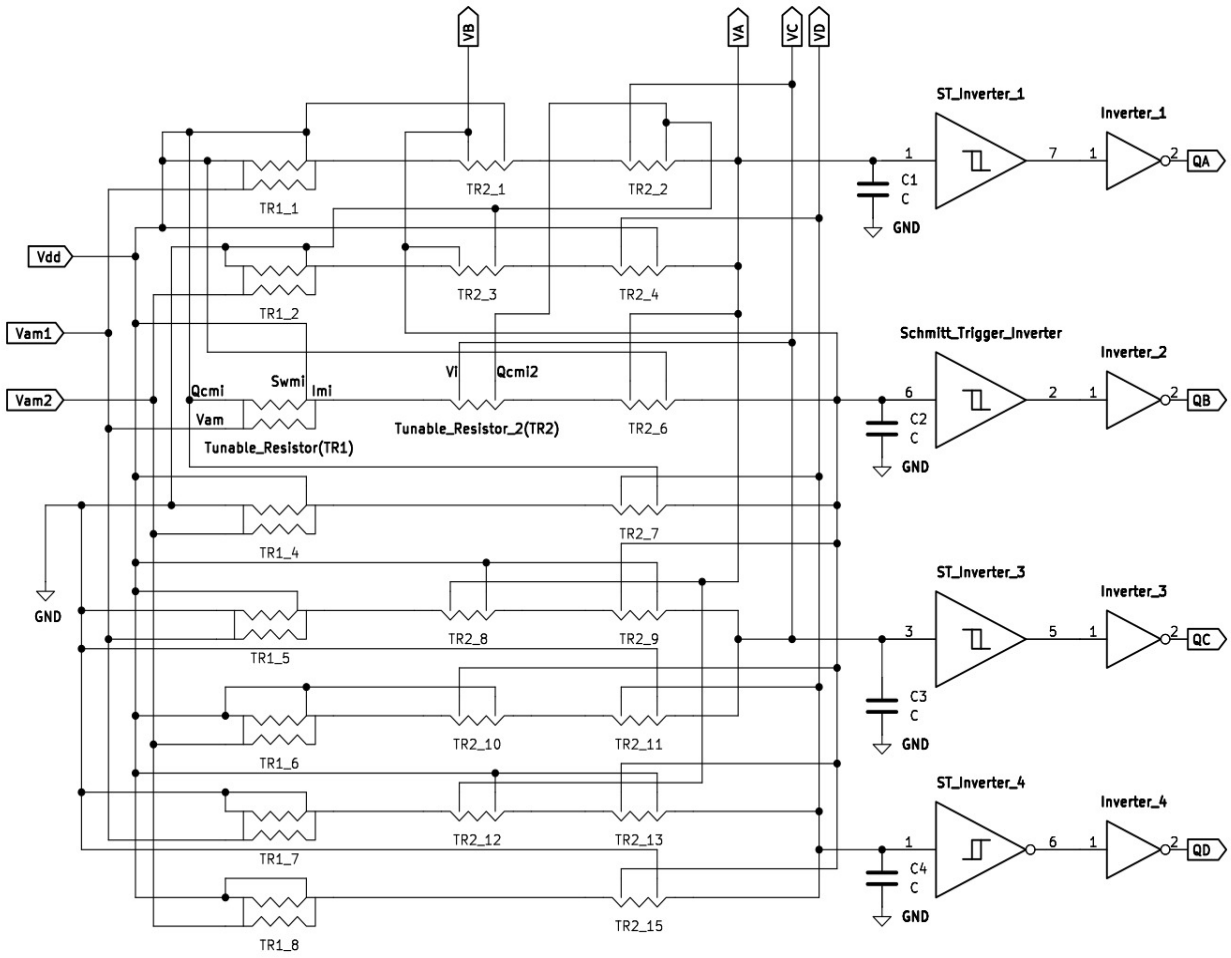

Figure 3.2: SDC schematic for CNF:$\{\{A, B, \overline{C}\}\{\overline{B}, D\}\}$

(3.9). The resistances mentioned in the formula are obtained through tunable resistors which resist current through the circuit based on the level of satisfiability of the variable. Each resistor $R_{m,i}, R_{m,i_2}$, and R_{m,i_3} correspond to the three variables of one clause in the *3*-SAT instance, such that this circuit block is replicated with different $R_{m,i}, R_{m,i_2}$, and R_{m,i_3} conditions for each variable. This voltage output signal of this circuit is analogous to our s_i variable, as defined earlier, which is the solution to k-SAT. The SDC block comprises a combination of these circuits, one for each variable, whose output end is also connected to a Schmitt trigger inverter to obtain the digital value of s_i. My implementation of the SDC circuit is shown in *Fig.3.2.*

3.2.2 Auxiliary Variable Circuit (AVC)

The AVC is a simple circuit with an operational amplifier, which has the three tunable resistors in series given to it as inputs. The operational amplifier outputs the voltage analogy of a_m, V_{a_m}, which follows an exponential growth driven by the level of non-SAT to ensure that none of the solutions get stuck in the local minima of the potential energy function in (3.2). Each clause has its own AVC circuit to generate a_m as per the SAT

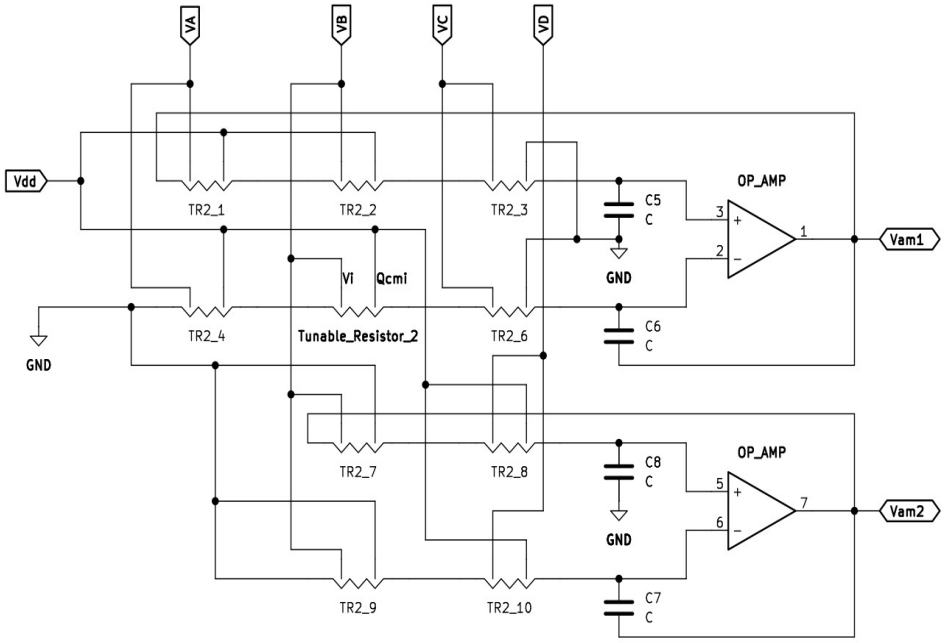

Figure 3.3: AVC schematic for CNF:$\{\{A, B, \overline{C}\}\{\overline{B}, D\}\}$

conditions of its variables. The first order differential equation of V_{a_m} [1] is given as,

$$C\frac{dV_{a,m}}{dt} = V_{a,m}/(R_{m,i_1} + R_{m,i_2} + R_{m,i_3}) \tag{3.9}$$

The simulation uses a Verilog-A behavioral model of an ideal op-amp [12]. The schematic of my AVC implementation is shown in *Fig.3.3*. There is an upper bound of V_{DD} on V_{a_m} by setting the supply voltage of the op-amp to V_{DD}. When any one variable in the clause is satisfied, the V_{a_m} of that clause is saturated to a certain voltage. The waveform of one such case is shown in *Fig.3.4*, where one can see V_{a_m} rising exponentially until CNF is satisfied.

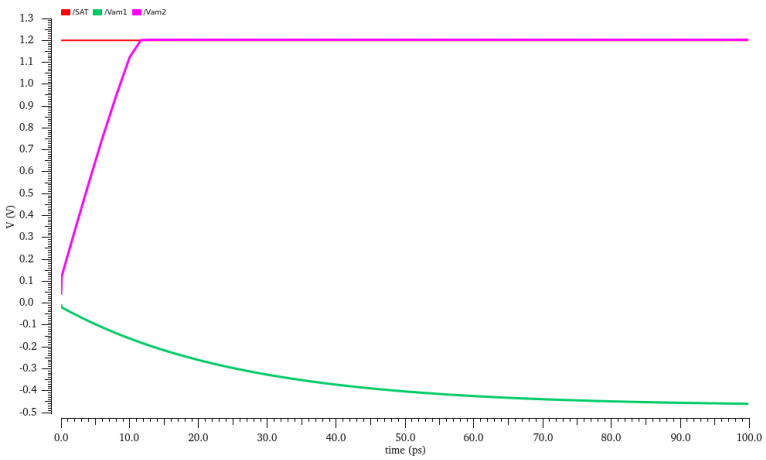

Figure 3.4: Virtuoso simulation for the waveform of V_{a_m}s over time.

3.2.3 Digital Verification Circuit (DVC)

This block determines if a solution to the CNF has been found. It is implemented through an array of $3M$ XOR gates and an array of M NAND gates. The inputs to the DVC are the digital representations of the s_i signals, as output from the SDC, and the $c_{m,i}$ signals as obtained from the C matrix *(defined in 3.1.1 From CNF to CTDS)*. If the variable i, exists in the clause m, in its non-negated form, the digital voltage associated with $c_{m,i}$, $Q_{c_{m,i}} = V_{DD}$ and if it exists in its negated form, the digital voltage associated with $c_{m,i}$, $Q_{c_{m,i}} = GND$. These two inputs are to the battery of XOR gates and if the value of the variable matches with the form it exists in, in the CNF, a false value is sent to the NAND gate. If every NAND gate gets at least one false value, the signal passes through the AND gate outputting a true, which means that the CNF is satisfiable. In other words, every clause of the CNF must have at least one variable that has a value following the form it exists in, for that CNF to be satisfiable. The implementation of the DVC is shown in *Fig.3.5.*

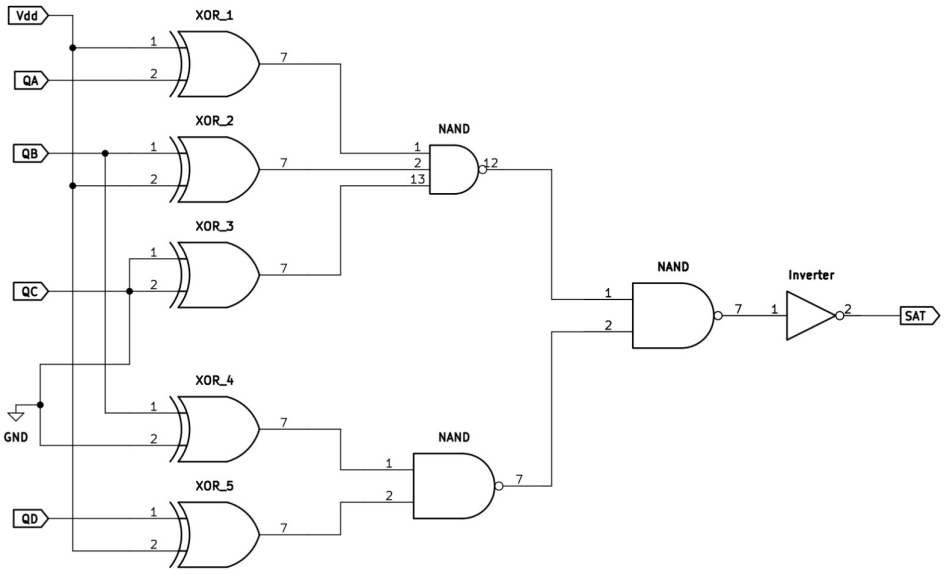

Figure 3.5: DVC schematic for CNF:$\{\{A, B, \overline{C}\}\{\overline{B}, D\}\}$

3.2.4 Simulation Results

This implementation successfully determines if the CNF is satisfiable and outputs both SAT and the solution at which SAT is discovered. *Figs. 3.6-3.8* show waveforms of some of the simulation results. If the CNF is satisfiable the voltage output of the DVC is high, i.e., $SAT = 1$ ($\approx 1.2V$), else, $SAT = 0$ ($\approx 0V$). Table 1 describes the different waveforms as per their CNFs and shows the solutions at which SAT/UNSAT is found.

Table 3.1: Simulation results for different CNFs run manually through Virtuoso

Figure	**CNF**	**SAT/UNSAT**	**Solution**
Fig.3.6	$\{\{A, B, C'\}\{B', D\}\}$	SAT	1011
Fig.3.7	$\{\{A, B, C\}\{B', C'\}\{D\}\}$	SAT	1101
Fig.3.8	$\{\{A, B, C\}\{B', C'\}\{D\}\{D'\}\}$	UNSAT	1100

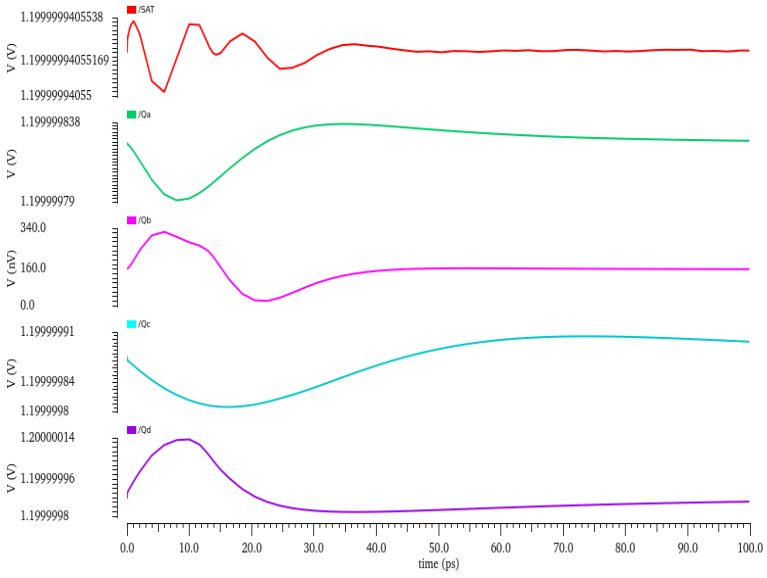

Figure 3.6: Simulation results for CNF: $\{\{A, B, \overline{C}\}\{\overline{B}, D\}\}$.

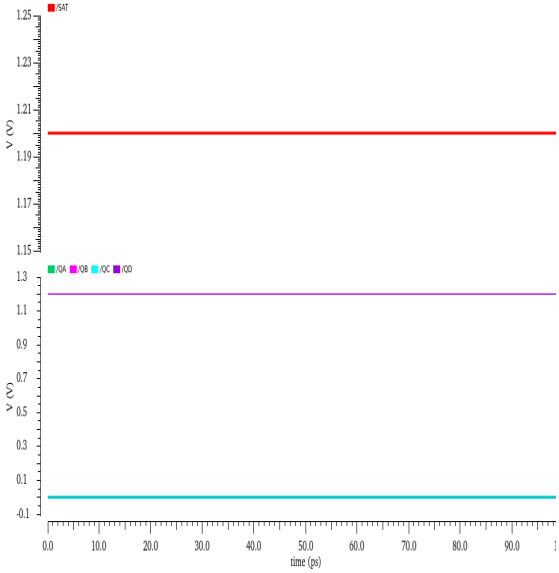

Figure 3.7: Simulation results for CNF: $\{\{A, B, C\}\{\overline{B}, \overline{C}\}\{D\}\}$.

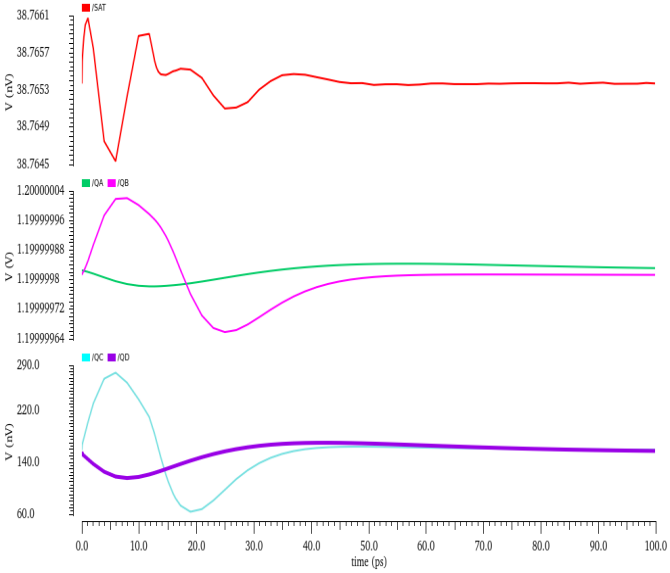

Figure 3.8: Simulation results for CNF: $\{\{A, B, C\}\{\overline{B}, \overline{C}\}\{D\}\{\overline{D}\}\}$.

3.3 The Automated Hardware-Software Solution

Upon understanding the connections between circuit elements at the lowest level of the hierarchy, the process of writing a script for netlisting becomes apparent. Netlisting the scripts as per the M and N values of the CNF makes the process of circuit generation a lot simpler than manually creating the circuit in the GUI of a tool like Cadence Virtuoso, which would be a lot harder for CNFs with a large number of clauses and variables, as it is in real-world applications. My ***hardware-software*** solution is implemented through a Python script designed to generate a netlist for the Analog SAT solver circuit as in [1]. The following steps break down the functionality of the code:

1. **Input:** The script reads the CNF through a text file, into which the user may input the CNF in a suitable format (as shown in *Fig.* 5.1). The code accepts the CNF input as a matrix of M rows and N columns. The rows represent the M clauses and the columns represent the N variables. This representation of the CNF as a matrix is better explained in Section 3.1.1 of this report where the $C_{m,i}$ matrix is defined.

2. **Parsing the matrix:** Before parsing, the script first displays the input matrix as a Product-Of-Sums expression (POS). The POS representation is similar to the CNF representation of a Boolean Formula. After this, the script parses through the expression to find the index of the first and last non-zero variable of each M clause. This parsing is done for the sub-circuit netlisting.

3. **Sub-circuit netlisting:** Once we have the first and last non-zero variables of the clauses, this part of the script generates netlists for the SDC, AVC, and DVC as per their circuits as described in Section 3.2.

4. **Creating templates for reusable blocks:** After that it appends the template in the netlist for blocks like logic gates, tunable resistors, and switches that are repeatedly used in the circuit.

5. **Generating the AVC block:** This script then writes netlists of all internal connections of the AVC block according to the variable information of each clause, as obtained on parsing the matrix.

6. **Generating the AVC top element:** The script creates the top module for the AVC block connecting internal AVC element pins to the external circuit. This AVC circuit is then connected to the other blocks, namely, the SDC and DVC.

7. **Generating the SDC element:** The script generates the circuit and netlist connections for the SDC block.

8. **Generating elements for the DVC block:** The script generates circuit elements for the DVC block. A subcircuit for an M-input (M clauses) NAND gate is also created for the DVC circuit.

9. **Generating the DVC element:** The script creates all the circuit connections for the DVC circuit block.

10. **Netlist instantiation:** Post the netlisting of all circuit blocks, the script instantiates the top-level modules for the SDC, AVC, and DVC blocks to dynamically assemble the overall circuit.

11. **Analysis and results:** After netlisting, the script sets up analysis for the simulation of the circuit by initiating Spectre simulation. Post simulation the script extracts the SAT solution and prints it.

Table 3.2: Simulation results for different CNFs run through Python script

POS No.	**CNF**	**SAT/UNSAT**	**Solution**
1	$\{\{A, B, C'\}\{B', D\}\}$	SAT	1011
2	$\{\{A, B, C\}\{B', C'\}\{D\}\}$	SAT	1101
3	$\{\{A, B, C\}\{B', C'\}\{D\}\{D'\}\}$	UNSAT	1000
4	$\{\{A, B'\}\{A', C, D\}\{B, D'\}\}$	SAT	0010
5	$\{\{A, B'\}\{C', D\}\}$	SAT	1001
6	$\{\{A, B, C'\}\{A', B', C\}\{A', B, C\}\}$	SAT	000
7	$\{\{A', B', C'\}\{D, E', F'\}\{A, B, C\}\}$	SAT	100100
8	$\{\{A', B', D\}\{D', A', B\}\{D', B, A\}\}$	SAT	0000
9	$\{\{A, B, C\}\{D, E, F'\}\{F, D', C'\}\{A', B', C'\}\}$	SAT	110010
10	$\{\{A, B, C\}\{A', B', C'\}\{A, B, D'\}\{A', B', D\}\}$	SAT	0010
11	$\{\{A\}\{A', B'\}\{B\}\}$	UNSAT	11
12	$\{\{A, B, C\}\{A, B, C'\}\{A, B', C\}$ $\{A, B', C'\}\{A', B, C\}\{A', B, C'\}$ $\{A', B', C\}\{A', B', C'\}\}$	UNSAT	000
13	$\{\{A, B, C'\}\{B', D, E'\}\{F, G, E'\}\{C, B, D'\}\}$	SAT	1000011

The Python code of the ***hardware-software*** implementation developed has been made available in the appendix section of this report and also at this github link:
https://github.com/SejalS01/3-SATSolver.git

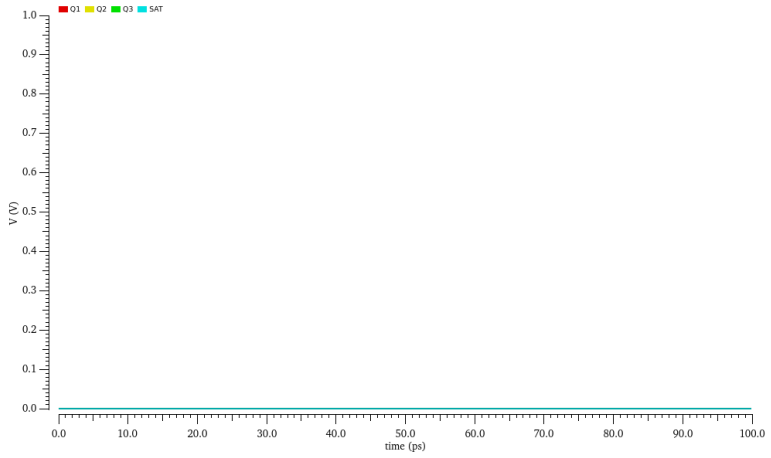

Figure 3.9: Simulation results for POS 12

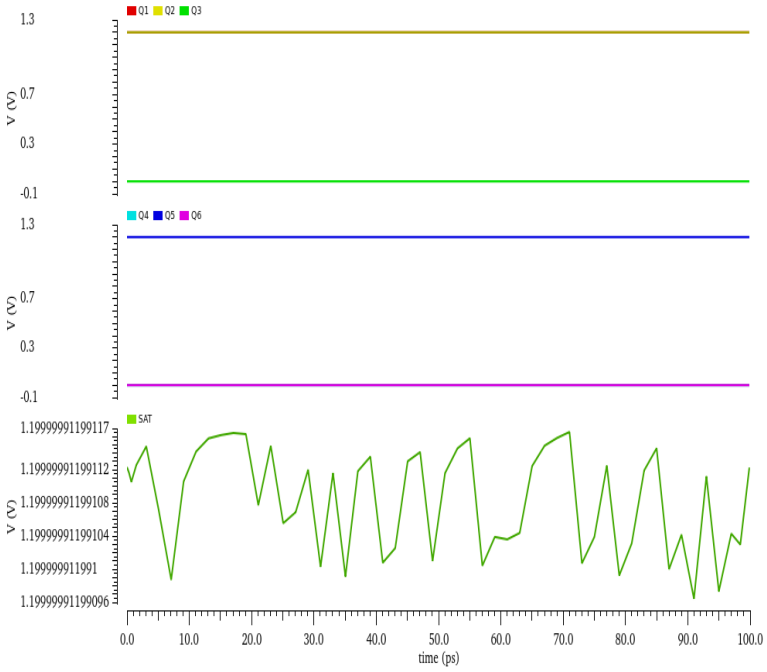

Figure 3.10: Simulation results for POS 9

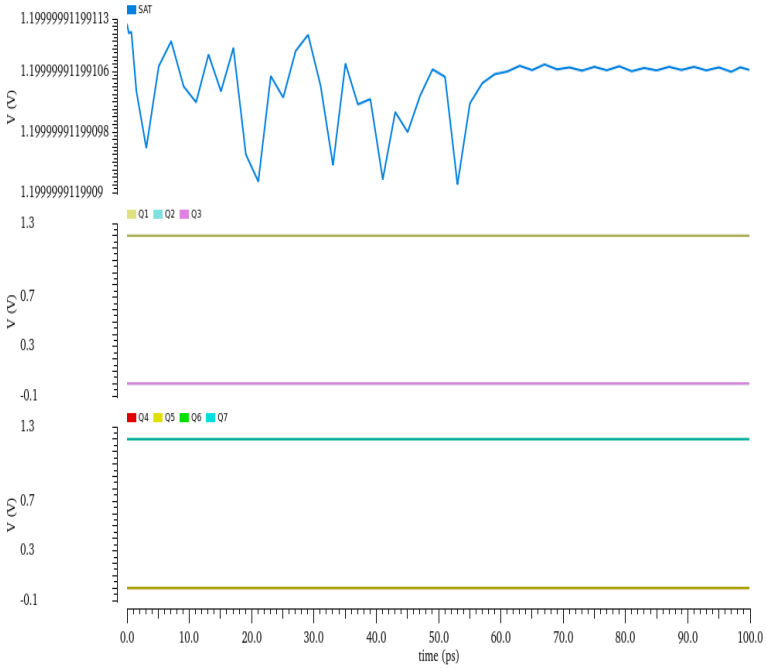

Figure 3.11: Simulation results for POS 13

Chapter 4

Conclusion

This report on "Analog Hardware for Solving the Boolean Satisfiability Problem" underscores the effectiveness and potential of the developed ***hardware-software*** approach. This methodology leverages a hybrid design that dynamically generates and simulates circuit netlists tailored to the requirements of specific Conjunctive Normal Form (CNF) inputs. The implementation successfully extends the capabilities of the analog circuit solution *AC-SAT*, which was originally conceived as a pure hardware solution. The project highlights several advantages of integrating software flexibility with hardware speed. The approach allows for rapid prototyping and testing of SAT solutions, offering a significant improvement in solving times compared to traditional software solvers. The dynamic generation of netlists based on input CNFs facilitates a more adaptable and scalable solution to address a variety of SAT instances efficiently.

In conclusion, this project has effectively demonstrated the practical implementation of the AC-SAT hardware solution for solving the Boolean Satisfiability Problem (SAT), which plays an instrumental role in the verification of complex circuitry across various domains such as computer-aided design and formal verification. Through the exploration of analog solutions to tackle the computationally intensive k-SAT problems, this initiative not only highlights the limitations of digital computation in handling NP-complex problems but also showcases the potential of analog approaches in managing such challenges more efficiently.

The successful development of a Python script that dynamically generates analog hardware netlists, coupled with the ability to simulate them in Cadence Spectre, represents a significant stride towards bridging the gap between software-based SAT solvers and their hardware counterparts. This integration facilitates a more streamlined and efficient verification process, enhancing the reliability of the solutions provided.

Furthermore, the project's ability to extract and present solutions in a user-friendly format ensures that the benefits of this sophisticated approach can be widely accessed and utilized by practitioners in the field. The reliability assessments and feasibility studies conducted as part of this project underscore the viability of extending traditional software SAT solvers into the hardware domain, opening up new avenues for future research and development in circuit design and automated reasoning.

Chapter 5

Future Prospects

Listed below are some more ideas that can be incorporated to make the ***hardware-software*** interface smoother:

- This project used a split Windows/Linux development platform by running the code on a Windows system and then uploading the generated netlist to Cadence Virtuoso for running the Spectre simulation and then displaying the outputs using tools such as the Viva available in the Virtuoso Suite. Moving the development platform to a complete Linux machine can be a scope for improvement. This will enable us to have a push-button run which will greatly enhance usability.

- This implementation does not make use of certain deterministic features of other solvers such as conditioning on unit clauses or conditioning on pure literals in the CNF. Several theoretical techniques, such as eliminating unitary clauses, conditioning pure literals, implementing clause satisfiability constraints, etc., used to simplify CNF instances, can be added to the script.

- Moreover, as discussed, since algorithms to reduce k-SAT to 3-SAT in polynomial time already exist, we can add those to this script to extend the usability of the solver to all k-SAT problems.

- In [1] several alternative AVC implementations are discussed that conserve area and power. Some remove the operational amplifier completely and obtain results in the solver of the same order as with an op-amp. The script has not focused on these aspects because it was not meant as a hardware-hardware solution (ie. one realized as an ASIC or FPGA). But if needed, it can be modified to optimize for that aspect also, albeit it is arguable if that can be deemed an optimization in a hardware-software-targeted implementation.

Bibliography

[1] X. Yin, B. Sedighi, M. Varga, M. Ercsey-Ravasz, Z. Toroczkai, and X. S. Hu, "Efficient analog circuits for boolean satisfiability," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 155–167, 2018.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, vol. 336. [Online]. Available: https://doi.org/10.3233/FAIA336

[3] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[4] M. Waldrop, "More than moore," *Nature*, vol. 530, 2016.

[5] M. Ercsey-Ravasz and Z. Toroczkai, "Optimization hardness as transient chaos in an analog approach to constraint satisfaction," *Nature Physics*, vol. 7, no. 12, pp. 966–970, 2011. [Online]. Available: https://doi.org/10.1038/nphys2105

[6] A. Darwiche and K. Pipatsrisawat, "Complete algorithms," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 101–132. [Online]. Available: https://doi.org/10.3233/FAIA200986

[7] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 133–182. [Online]. Available: https://doi.org/10.3233/FAIA200987

[8] B. Selman, H. A. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," in *Cliques, Coloring, and Satisfiability*, 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:3215289

[9] K. Bousmar, F. Monteiro, Z. Habbas, S. Dellagi, and A. Dandache, "A pure hardware k-sat solver architecture for fpga based on generic tree-search," in *2017 29th International Conference on Microelectronics (ICM)*, 2017, pp. 1–5.

[10] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[11] Cadence Design Systems, Inc., "Cadence eda virtuoso suite." [Online]. Available: https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-design/virtuoso-ade-suite.html

[12] M. Brinson, "Qucs: Verilog-a modular macromodel for operational amplifiers," Tech. Rep., 2007.

Appendix

```
# Example Input File for CNF Matrix
# The matrix represents the CNF: {A + B'}{A' + C + D}
# Each row represents a clause, and each column represents a variable where:
# 1 means the variable is present, -1 means the variable is negated
# 0 means the variable is absent.
# Matrix values (enter row by row, use space to separate values in a row):
# An example of the input written in the text file looks like this:
1 -1 0 0
-1 0 1 1

# Notes:
#  Ensure each line corresponds to one clause in your CNF formula.
#  This file format supports an arbitrary number of variables, but each line must
#  have the same number of entries for consistency.
#  This format allows easy parsing and usage
#  in simulations or solvers requiring CNF input.
```

Figure 5.1: Sample input file for CNF: $\{\{A, \overline{B}\}\{\overline{A}, C, D\}\}$

This section contains the python script for the ***hardware-software*** implementation.

```
"""
#########################################################################
Copyright (c) 2024, Shiv Nadar University, Delhi NCR, India. All
   Rights Reserved. Permission to use, copy, modify and
   distribute this software for educational, research, and
   not-for-profit purposes, without fee and without a signed
   license agreement, is hereby granted, provided that this
   paragraph and the following two paragraphs appear in all
   copies, modifications, and distributions.IN NO EVENT SHALL
   SHIV NADAR UNIVERSITY BE LIABLE TO ANY PARTY FOR DIRECT,
   INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
   INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS
   SOFTWARE. SHIV NADAR UNIVERSITY SPECIFICALLY DISCLAIMS ANY
   WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
   WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS PROVIDED "AS IS".
   SHIV NADAR UNIVERSITY HAS NO OBLIGATION TO PROVIDE
   MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

#########################################################################
   """
```

```
def main():


    def input_matrix(file_path):
        """
    Read a matrix from a file, excluding comment lines and empty
       lines.

    This function opens a file from the specified path and reads
       its contents to form a matrix.
    Each line in the file represents a row in the matrix, but
       lines that start with '#' (comments)
    or are completely empty are ignored. Each row is split into
       integers, forming the matrix.

    Parameters:
    file_path (str): The path to the file containing the matrix
       data.

    Returns:
    list[list[int]]: A 2D list of integers forming the matrix
       read from the file.
    """
        with open(file_path, 'r') as file:
        # Read all lines in the file
            lines = file.readlines()

        # Initialize the matrix
            matrix = []

        # Convert each line to a row in the matrix, ignoring
           lines that are comments or empty
            for line in lines:
                if not line.startswith('#') and line.strip() !=
                   '':
                    row = list(map(int, line.strip().split()))
                    matrix.append(row)

        return matrix



    def display_pos(matrix):
        """
    Display the matrix as clauses in either POS (Positive) or
       CNF (Conjunctive Normal Form) notation.
```

```
This function takes a matrix where each row represents a
   clause, and each element in the row
represents the state of a variable in the clause (1, -1, or
   0). It then formats and prints
each clause in a readable logical expression where 'xN'
   represents the N-th variable:
- 'xN' for a positive variable (1),
- 'xN'' (xN prime) for a negative variable (-1),
- Skips the variable if its state is 0.

Parameters:
matrix (list[list[int]]): The matrix where each row
   represents a clause.
"""
    for row in matrix:
        clause = []
        for i, val in enumerate(row):
            if val == 1:
                clause.append(f"x{i+1}")
            elif val == -1:
                clause.append(f"x{i+1}'")
        print(f"({' OR '.join(clause)})")



def find_first_non_zero(clause):
    """
Find the index of the first non-zero variable in a clause.

This function scans through a list representing a clause and
   returns the index of the first element
that is non-zero. This is useful in algorithms that need to
   identify the first significant variable
in a clause for further processing.

Parameters:
clause (list): The clause represented as a list of variable
   states.

Returns:
int or None: The index of the first non-zero variable, or
   None if all variables are zero.
"""
    for index, var in enumerate(clause):
        if var != 0:
            return index
    return None
```

```
#function to find the index of the last non-zero variable
""" def find_last_non_zero(clause):
    last_index = None
    for index, var in enumerate(clause):
        if var != 0:
            last_index = index
    return last_index"""



def generate_avc_element(clause_matrix):
    """
Generates the netlist for an Auxiliary Variable Circuit
   (AVC) based on the given matrix of clauses.

This function constructs a SPICE-like netlist describing the
   internal connections of an AVC block.
Each row in the input matrix represents a clause, and each
   element in the row indicates the state
of a variable in the clause (non-zero means the variable is
   active). The function iterates over
each clause, adds subcircuit headers, and creates
   connections based on the state of each variable.
It handles the first non-zero variable in each clause
   specially to differentiate its connections.
Other non-zero variables are connected in series using
   transistors. The function also adds components
like operational amplifiers and capacitors specific to each
   clause's subcircuit.

Parameters:
clause_matrix (list[list[int]]): A matrix where each row
   represents a clause and each element
in a row represents the state of a variable in the clause (1
   for positive, -1 for negative, 0 for inactive).

Returns:
str: A string containing the complete netlist for the AVC
   based on the given clause matrix.
"""
    #Function to write netlist of internal connections of
       AVC block according to every clause
    avc_netlist = ""
    net_counter=1 #to keep count of the nets in each
       subcircuit
    for clause_id, clause in enumerate(clause_matrix):
        # Add header for the subcircuit
        avc_netlist += f"* Clause {clause_id+1} element in
           AVC\n"
```

```
avc_netlist += f".subckt avc{clause_id+1}
   Vam{clause_id+1} EN"
for var_id, var_state in enumerate(clause):
    if var_state != 0:
        avc_netlist += f" V{var_id+1}"
avc_netlist += "\n"

# Handle the first non-zero variable differently
first_non_zero_index = find_first_non_zero(clause)
for var_id, presence in enumerate(clause):
    if presence != 0:
    # Handle the first non-zero variable differently
       as its connections are different (its one
       terminal is connected to either gnd or the sw)
        if var_id == first_non_zero_index:
            connection = 'Vdd' if presence == 1 else
               '0'
            avc_netlist += f"X{clause_id}_{var_id}
               net0 net1 {connection} V{var_id+1}
               TR2\n"
            avc_netlist += f"X{clause_id}_{var_id+1}
               0 net101 {connection} V{var_id+1}
               TR2\n"
            pass
        else: #for the other non-zero variables,
           connection of TR2 instances in series
            if presence != 0:
                connection = 'Vdd' if presence == 1
                   else '0'
                # Adjust the instance numbering to
                   avoid repetition
                avc_netlist +=
                   f"X1{clause_id}_{net_counter}
                   net{net_counter}
                   net{net_counter+1} {connection}
                   V{var_id+1} TR2\n"
                avc_netlist +=
                   f"X2{clause_id}_{net_counter}
                   net{net_counter+100}
                   net{net_counter+101} {connection}
                   V{var_id+1} TR2\n"
                net_counter += 1  # Increment for
                   the next net name
            pass

# Add the opamp and branch components (sw,
   capacitors) for the clause element
avc_netlist += f"XOP{clause_id} net{net_counter}
   net{net_counter+100} Vam{clause_id+1} mod_amp\n"
```

```
            avc_netlist += f"C{clause_id}_0 net{net_counter} 0
                1p\n"
            avc_netlist += f"C{clause_id}_1 net{net_counter+100}
                Vam{clause_id+1} 1p\n"
            avc_netlist += f"XI{clause_id}_2 net0
                Vam{clause_id+1} EN sw\n"
            avc_netlist += f".ends avc{clause_id+1}\n\n"
        return avc_netlist



def generate_avc_top_element(clause_matrix):
    """
Generates the top-level netlist for an Auxiliary Variable
    Circuit (AVC) using a matrix of clauses.

This function constructs the top-level netlist for an AVC by
    defining a subcircuit that integrates
multiple internal AVC subcircuits, each corresponding to a
    clause from the clause matrix. It creates
connections between the inputs of the top-level AVC block
    and the inputs of each internal AVC
subcircuit, as well as manages the output nets.

The function iterates over the variables and clauses in the
    matrix to generate appropriate connections
for each AVC subcircuit within the top-level AVC block. It
    includes the necessary inputs for each
clause subcircuit based on active variables in the clause
    and instantiates each internal AVC subcircuit.

After constructing the top-level AVC netlist, this function
    calls `generate_avc_element` to get
the netlists for individual AVC clauses and combines them
    with the top-level netlist to form
a complete AVC system netlist.

Parameters:
clause_matrix (list[list[int]]): A matrix where each row
    represents a clause, and each element
in a row represents the state of a variable in the clause (1
    for positive, -1 for negative, 0 for inactive).

Returns:
str: A string containing the complete netlist for the AVC,
    including both the top-level and
individual clause-level netlists.
"""
    #function for creating a top module for the AVC block
```

```
        connecting the internal avc element pins to outside
        circuit
    top_avc_netlist = "* Top Level AVC Block\n"
    top_avc_netlist += ".subckt AVC EN"

    # Add all Vi inputs to the top-level AVC block
    for var_id in range(len(clause_matrix[0])):
        top_avc_netlist += f" V{var_id+1}"
    for clause_id, clause in enumerate(clause_matrix):
        top_avc_netlist += f" Vam{clause_id+1}"
    top_avc_netlist += "\n"

    # Include individual AVC clause subcircuits
    for clause_id, clause in enumerate(clause_matrix):
        # Connect Vi inputs to corresponding AVC{clause_id}
            subcircuit inputs
        inputs = " ".join([f"V{var_id+1}" for var_id,
            var_state in enumerate(clause) if var_state != 0])
        # Instantiate each AVC{clause_id} with the
            corresponding inputs and an internal Vam net
        top_avc_netlist += f"Xavc{clause_id+1}
            Vam{clause_id+1} EN {inputs} avc{clause_id+1}\n"
    top_avc_netlist += f".ends AVC\n"

    # Combine with the individual AVC clauses netlists
    avc_netlist = generate_avc_element(clause_matrix)
    complete_netlist = avc_netlist + top_avc_netlist
    return complete_netlist



def generate_sdc_element(matrix):
    """
Generates a SPICE netlist for a Signal Dynamic Circuit (SDC)
    block using a matrix
where each row represents a different clause and each column
corresponds to a variable in the CNF.

This function constructs a subcircuit for the SDC block,
    where each variable has a dedicated
branch that handles different states according to the
    matrix. The netlist includes transistor
instances (TR1 and TR2) for logic processing, resistance for
    convergence, and capacitors for
stability or delay purposes. The output is modulated through
    inverters and Schmitt triggers
to produce a cleaned and stabilized output signal for each
    variable.
```

```
Parameters:
matrix (list[list[int]]): A 2D list where each row
   represents a clause and each element
                          in a row represents the state of a
                             variable (1 for positive, -1 for
                             negative,
                          and 0 for inactive).

Returns:
str: A string containing the complete netlist for the SDC
   block. This netlist can be used
     directly in SPICE simulations to model the described
        configurable logic behavior.
"""
    #function for generating the SDC block connections
    netlist = "*Element in SDC\n"
    netlist += f".subckt SDC"
    #header for the subcircuits
    # Assuming each row has the same number of columns
    net_counter = 1  # Unique identifier for each net in the
       netlist
    for var_id in range(num_variables):
        netlist += f" Q{var_id+1}"
        netlist += f" V{var_id+1}"
    for clause_id, clause in enumerate(matrix):
        netlist += f" Vam{clause_id+1}"
    netlist += "\n"
    #creating the clause-wise branches for each variable
       separately
    for var_id in range(num_variables):
        netlist += f"*Variable {var_id+1} branches\n"
        previous_nets = []

        #for creating the TR1 instance for every clause
           where this particular variable is present
        for clause_id, clause in enumerate(matrix):
            var_presence = clause[var_id]
            if var_presence != 0:
                state = 'Vdd' if var_presence == 1 else '0'
                netlist += f"XTR1_{var_id+1}_{clause_id}
                   net{net_counter} {state} Vdd
                   Vam{clause_id+1} TR1\n"
                current_net = f"net{net_counter}"
                net_counter += 1

                # Add TR2 instances for every other variable
                   in the clause
                for other_var_id, other_var_presence in
                   enumerate(clause):
```

```
                    if other_var_id != var_id and
                       other_var_presence != 0:
                        other_state = 'Vdd' if
                           other_var_presence == 1 else '0'
                        netlist +=
                           f"XTR2_{varid+1}{clause_id}{other_var_id+1}
                           {current_net} net{net_counter}
                           {other_state} V{other_var_id+1}
                           TR2\n"
                        current_net = f"net{net_counter}"
                        net_counter += 1
                previous_nets.append(current_net)

        if previous_nets:
            # Convergence point for the branches of this
               variable across all clauses
            convergence_net = f"net{net_counter}"
            net_counter += 1

            for end_net in previous_nets:
                netlist += f"R{var_id+1}_{end_net} {end_net}
                   {convergence_net} 0\n"

            # Final connections to capacitor and Schmitt
               trigger for this variable through the
               convergence point
            netlist += f"C{var_id+1} {convergence_net} 0
               500f\n"
            netlist += f"XS_{var_id+1} Q{var_id+1}bar
               {convergence_net} INVschmitt\n"
            netlist += f"XInv_{var_id+1} Q{var_id+1}
               Q{var_id+1}bar INV\n"
    netlist += f".ends SDC\n"
    return netlist



def generate_n_input_nand_netlist(n):
    """
Generates a SPICE netlist for an n-input NAND gate using
   CMOS technology.

The function constructs the SPICE netlist description for an
   n-input NAND gate,
including both PMOS and NMOS transistor networks. The PMOS
   transistors are connected
in parallel to ensure that the output is high when any input
   is low. The NMOS
transistors are connected in series, ensuring that the
```

```
    output is low only when all
inputs are high.

The netlist includes a subcircuit definition with input and
    output terminals,
transistor definitions for both PMOS and NMOS, and
    connections for the transistors.
Each transistor is defined with specific model parameters
    and dimensions appropriate
for 45 nm technology nodes.

Parameters:
n (int): The number of inputs to the NAND gate, specifying
    how many PMOS and NMOS
          transistors will be included in the netlist.

Returns:
str: A string containing the complete SPICE netlist for the
    described n-input NAND gate.
      This netlist can be used directly in SPICE simulations.
"""
    header = f"* SPICE Netlist for a {n}-input NAND gate\n"
    subckt = f".subckt NAND{n} " + " ".join([f"in{i}" for i
       in range(1, n+1)]) + " Out \n"

    # Generate PMOS transistors
    pmos_transistors = ""
    for i in range(1, n + 1):
        pmos_transistors += f"MP{i-1} Out i{i} Vdd Vdd
           g45p1svt w=120n l=45n\n"

    # Generate NMOS transistors
    nmos_transistors = ""
    for i in range(1, n + 1):
        if i == 1:
            source = "0"
        else:
            source = f"net{i-2}"

        if i == n:
            drain = "Out"
        else:
            drain = f"net{i-1}"

        nmos_transistors += f"MN{i-1} {drain} i{i} {source}
           0 g45n1svt w=120n l=45n\n"

    footer = ".ends NAND" + str(n) + "\n\n"
```

```
    # Combine all parts
    netlist = header + subckt + pmos_transistors +
       nmos_transistors + footer
    return netlist



def generate_dvc_element(matrix):
    """
Generates a SPICE netlist for a Digital Verification Circuit
   (DVC) from a given matrix.

This function translates a matrix, where each row represents
   a clause in CNF, into a DVC using
XOR and NAND gates to simulate the clause logic. The matrix
   uses 1 for true, -1 for false, and 0
for an inactive variable. Each clause is processed to create
   a subcircuit with XOR gates representing
the variable states and a NAND gate combining these states.
   The overall output of all clauses is
combined using another NAND gate to produce a single output
   indicating SATisfiability.

Parameters:
matrix (list[list[int]]): Matrix where each row represents a
   clause with integers indicating variable
                          states (1 for true, -1 for false,
                             0 for inactive).

Returns:
str: A complete netlist string for the DVC that can be used
   in SPICE simulations.
"""
    #function for creating DVC block connections
    netlist = "* SPICE Netlist for Digital Verification
       Circuit (DVC)\n"
    #header for subcircuit
    netlist += f".subckt DVC SAT" #output pin SAT
    for var_id in range(num_variables):
        netlist += f" Q{var_id+1}" #input pins
    netlist += f"\n"
    net_counter = 1  # To create unique net names
    clause_nand_outputs = []  # To collect the outputs of
       each clause's NAND gate

    # Iterate through each clause in the CNF
    for clause_id, clause in enumerate(matrix):
        clause_variables = [idx + 1 for idx, var in
           enumerate(clause) if var != 0]
```

```
        current_clause_nets = []

        # Add XOR gates for each variable present in the
            clause
        for var_id in clause_variables:
            var_state = 'Vdd' if clause[var_id - 1] == 1
                else '0'
            xor_output_net = f"net{net_counter}"
            netlist += f"X{clause_id}_{var_id} Q{var_id}
                {var_state} {xor_output_net} XOR\n"
            current_clause_nets.append(xor_output_net)
            net_counter += 1

        # Add an NAND gate for the clause
        nand_inputs = ' '.join(current_clause_nets)
        nand_output_net = f"net{net_counter}"
        clause_nand_outputs.append(nand_output_net)
        if len(clause_variables) == 1:
        # If only one variable in the clause, use an
            inverter (INV) instead of a NAND gate
            netlist += f"XINV{clause_id} {nand_inputs}
                {nand_output_net} INV\n"
        else:
        # For more than one variable, use a NAND gate with
            the appropriate number of inputs
            netlist += f"XNAND{clause_id} {nand_inputs}
                {nand_output_net}
                NAND{len(clause_variables)}\n"

        net_counter += 1   # Increment counter to get name
            for output

# Assuming clause_nand_outputs contains the outputs of
    all clause's NAND gates
# Add an M-input NAND gate for the final SAT output
m_nand_inputs = ' '.join(clause_nand_outputs)
nand_output_net = f"net{net_counter}"
netlist += f"XNAND_final {m_nand_inputs}
    {nand_output_net} NAND{len(matrix)}\n"
net_counter += 1

# Inverter (NOT gate) to invert the NAND output to get
    AND indicator
netlist += f"XINV_final SAT {nand_output_net} INV\n"
net_counter += 1
netlist += f".ends DVC\n"
return netlist
```

```
#Commands to run
matrix =
   input_matrix('C:\\Users\\Sejal\\Desktop\\Input_File.txt')
display_pos(matrix)

#source file containing the definition of reusable blocks
   like tunable resisitors (i.e. TR1 & TR2) and the logic
   gates used
source_file_path = 'C:\\Users\\Sejal\\Desktop\\SNU\\Sem
   8\\upload github\\blocks.ckt'
#destination file in which the final netlist will be written
destination_file_path =
   "C:\\Users\\Sejal\\Desktop\\netlist.sp"

#contents of source file being copied into  destination file
   line by line
with open(source_file_path, 'r') as source_file,
   open(destination_file_path, 'w') as destination_file:
    for line in source_file:
        destination_file.write(line)

num_variables = len(matrix[0]) #no of variables in CNF
num_clauses=len(matrix) #no of clauses in CNF

netlist = f"\n"
if num_clauses>3:
    netlist += generate_n_input_nand_netlist(num_clauses)

#combining all the elements in a single netlist
netlist +=
   generate_sdc_element(matrix)+"\n"+generate_avc_top_element(matrix)
   +"\n"+generate_dvc_element(matrix)+"\n\n"

#adding the top module instantiation for SDC, AVC, DVC blocks
netlist += f"*Top module instantiation\n"
netlist += f"X1"

# Add all input and output pins to the top-level SDC block
for var_id in range(num_variables):
        netlist += f" Q{var_id+1}"
        netlist += f" V{var_id+1}"
for clause_id, clause in enumerate(matrix):
        netlist += f" Vam{clause_id+1}"
netlist +=" SDC\n"
netlist += f"X2 Vdd"

# Add all Vi inputs to the top-level AVC block
for var_id in range(len(matrix[0])):
```

```
            netlist += f" V{var_id+1}"

        # Add all Vam outputs to the top-level AVC block
        for clause_id, clause in enumerate(matrix):
            netlist += f" Vam{clause_id+1}"
        netlist += f" AVC\n"

        # Add all input and output pins to the top-level DVC block
        netlist += f"X3 SAT"
        for var_id in range(num_variables):
                netlist += f" Q{var_id+1}"
        netlist += f" DVC\n"

        #adding the analysis and printing of the solution
        for var_id in range(num_variables):
            netlist += f".print tran V(Q{var_id+1})\n"
        netlist += f".print tran V(SAT)\n.end\n"

        #appending it all in destination file
        with open(destination_file_path, 'a') as destination_file:
            destination_file.write(netlist)


if __name__ == "__main__":
    main()
```