**Automated Design Error Debugging of Digital VLSI Circuits by Using Convolutional Autoencoder**

*A Project Work*

*Submitted in partial fulfillment of Requirements for the Award of the Degree of*

**BACHELOR OF TECHNOLOGY**

### IN

**ELECTRONICS & COMMUNICATION ENGINEERING**

### BY

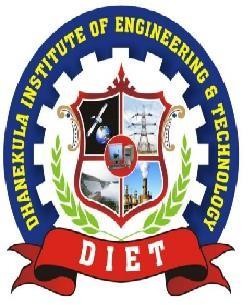
**M. SANJAY (218T1A0421) D. VAMSI (218T1A0408)**

**A. VAMSI KRISHNA (218T1A0402) G. AJAY KUMAR (218T1A0414)**

*Under the Esteemed guidance of*

**Mr. S. CHANDRA SEKHAR**

Assistant Professor



**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**DHANEKULA INSTITUTE OF ENGINEERING & TECHNOLOGY**

**(AUTONOMOUS)**

**Ganguru, Vijayawada - 521 139**

**Affiliated to JNTUK, Kakinada & Approved By AICTE, New Delhi Certified by ISO 9001-2015, Accredited by NAAC & NBA**

**APRIL-2025**

**DHANEKULA INSTITUTE OF ENGINEERING &TECHNOLOGY**

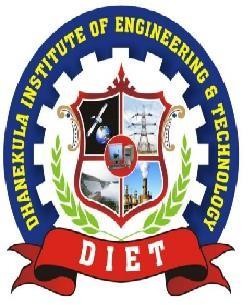
**(AUTONOMOUS)**

**Ganguru, Vijayawada - 521 139**

**Affiliated to JNTUK, Kakinada & Approved By AICTE, New Delhi Certified by ISO 9001-2015, Accredited by NAAC& NBA**

**APRIL-2025**

**DEPARTMENT OF ELECTRONICS &COMMUNICATION ENGINEERING**



**CERTIFICATE**

This is to certify that the project work entitled “AUTOMATED DESIGN ERROR DEBUGGING OF DIGITAL VLSI CIRCUITS BY USING CONVOLUTIONAL AUTOENCODER ” is a bona fide record of project work done jointly by M.SANJAY (218T1A0421), D.VAMSI (218T1A0408), A.VAMSI KRISHNA (218T1A0402), G.AJAY KUMAR (218T1A0414) under my guidance and supervision and is submitted in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics & Communication Engineering by Jawaharlal Nehru Technological University, Kakinada during the academic year 2021- 2025.

**Project Guide External Examiner Head of the Department**

Mr. S. Chandra Sekhar Dr. M.Vamshi Krishna

Assistant Professor, Professor & H.O.D,

Department of ECE. Department of ECE.

**ACKNOWLEDGMENT**

First and foremost, we sincerely salute our esteemed institution **DHANEKULA INSTITUTE OF ENGINEERING AND TECHNOLOGY** for giving us this opportunity for fulfilling our project.

We express our sincere thanks to our beloved Principal **Dr. Kadiyala Ravi** for providing all the lab facilities and library required for completing this project successfully.

This project work would not have been possible without the support of many people. We wish to express our gratitude to **Dr. M. Vamshi Krishna**, Ph.D., H.O.D. ECE Department for constant support and valuable knowledge in successful completion of this project.

We are glad to express our deep sense of gratitude to **Mr. S. Chandra Sekhar,** Assistant Professor, for abundant help and offered invaluable assistance, support and guidance throughout this work.

We thank one and all who have rendered help directly or indirectly in the completion of this project successfully.

**PROJECT ASSOCIATES**

M.SANJAY (218T1A0421)

D.VAMSI (218T1A0408)

A. VAMSI KRISHNA (218T1A0402)

G. AJAY KUMAR (218T1A0414)

DECLARATION

I hereby declare that the mini project titled "Automated Design Error Debugging of Digital VLSI Circuits by Using Convolutional Autoencoder" is submitted in partial fulfilment of B. Tech in Electronic communication Engineering is my original work carried out by me under the guidance of MR.S. CHANDRA SEKHAR, Assistant professor and has not formed the basis for the award of any other degree or diploma, fellowship or any other similar titles.

Place: Ganguru:

Date:

QL

Signature of the student

M.Sanjay (218T1A0421)

D.Vamsi (218T1A0408)

A. Vamsi Krishna (218T1A0402)

G. Ajay Kumar (218t1A0414)

**CONTENTS**

VISION, MISSION OF INSTITUTE & DEPARTMENT, PEO`s I

[STATEMENT OF PO`s & PSO`s II](#_bookmark0)

[PROJECT MAPPING - PO`s & PSO`s III](#_bookmark1)

[LIST OF FIGURES IV](#_bookmark2)

[ABSTRACT V](#_TOC_250000)

|  |  |  |
| --- | --- | --- |
| **CHAPTER** | **TITLE** | **PAGE NO** |
| **1** | **INTRODUCTION** | **1-11** |
|  | 1.1 Importance of VLSI Fault Detection | **1** |
|  | 1.2 Challenges in IASCAS-85 Benchmark Circuits | 2 |
|  | 1.3 Objectives of the Study | 3 |
|  | 1.4 Types of Faults in Digital VLSI Circuits | 5 |
|  | 1.5 Motivation | 7 |
|  | 1.6 Tools used | 10 |
|  | 1.7 Summary of the Work | 10 |
| **2** | **LITERATURE SURVEY** | **12-17** |
| **3** | **METHODOLOGY** | **18-23** |
|  | 3.1 Test Pattern Generation Using ATPG Tool | 19 |
|  | 3.2 Auto Encoder Model | 20 |
|  | 3.3 Convolutional Autoencoder (CAE)Architecture | 21 |
|  | 3.4CNN Based Fault Detection Model | 22 |
| **4** | **IMPLEMENTATION** | **24-49** |
|  | 4.1 Implementation for 32-bit SEC Circuit (c499) | 25 |
|  | 4.2 Implementation for 27- Channel Interrupt Controller (c432) | 31 |
|  | 4.3 Implementation for 8-bit ALU (c880) | 35 |
|  | 4.4 Implementation for 16-bit SEC/DED Circuit (c1908) | 39 |
|  | 4.5 Implementation for 8-bit ALU (c3540) | 43 |
|  | 4.6 Implementation for 16x16 multiplier (c6288) | 47 |
| **5** | **HARDWARE COMPONENTS** | **51-56** |
|  | 5.1 Laptop | 52 |
|  | 5.2 Essential Hardware Components | 53 |
| **6** | **SOFTWARE** | **57-62** |
|  | 6.1. Python IDE | 58 |
|  | 6.2 NumPy | 59 |
|  | 6.3 TensorFlow/Keras (for CNN and CAE) | 60 |
|  | 6.4 Scikit-Learn | 61 |
|  | 6.5 Atalanta ATPG Tool | 62 |
| **7** | **RESULTS** | **63-75** |
|  | 7.1 32-bit SEC Circuit(c499) | 64 |
|  | 7.2 27-Channel Interrupt Controller: | 65 |
|  | 7.3 8-bit ALU (c880) | 66 |
|  | 7.4 16-bit SEC/DED circuit(c1908) | 68 |
|  | 7.5 8-bit ALU (c3540) | 70 |
|  | 7.6 16x16 multiplier(c6288) | 72 |
|  | 7.7 Performance metrics for existing and Proposed model | 73 |
| **8** | **CONCLUSION & FUTURE SCOPE** | **76-77** |
| **9** | **REFERENCES** | **78-80** |
| **10** | **APPENDIX** | **81-85** |

**DHANEKULA INSTITUTE OF ENGINEERING & TECHNOLOGY**

Department of Electronics & Communication Engineering

**VISION – MISSION - PEOs**

Vision/Mission/PEOs

|  |  |
| --- | --- |
| Institute Vision | Pioneering Professional Education through Quality |
| Institute Mission | Providing Quality Education through state-of-art infrastructure, laboratories and committed staff.  Molding Students as proficient, competent, and socially responsible engineering personnel with ingenious intellect.  Involving faculty members and students in research and development works for betterment of society. |
| Department Vision | Pioneering Electronics & Communication Engineering education and research to elevate rural community |
| Department Mission | Imparting professional education endowed with ethics and human values to transform students to be competent and committed electronics engineers.  Adopting best pedagogical methods to maximize knowledge transfer.  Having adequate mechanisms to enhance understanding of theoretical concepts through practice.  Establishing an environment conducive for lifelong learning and entrepreneurship development.  To train as effective innovators and deploy new technologies  for service of society. |
| Program Educational Objectives (PEOs) | PEO1: Shall have professional competency in electronics and communications with strong foundation in science, mathematics and basic engineering.  PEO2: Shall design, analyze and synthesize electronic circuits and simulate using modern tools.  PEO3: Shall Discover practical applications and design innovative circuits for Lifelong learning.  PEO4: Shall have effective communication skills and practice the ethics consistent with a sense of social responsibility. |

## STATEMENT OF PO`s & PSO`s

**Program Outcomes**

PO1 **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals and engineering programs.

PO2 **Problem analysis**: Identify, formulate, review research literature, and analyze complex Engineering problems reaching substantiated conclusions using first principles of Mathematics, natural sciences, and engineering sciences.

PO3 **Design/development of solutions**: design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental Considerations.

PO4 **Conduct investigations of complex problems**: Use research-based knowledge and research Methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6 **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7 **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8 **Ethics:** Apply ethical principles and commit to professional ethics and responsibility and norms of the engineering practice.

PO9 **Individual and team work:** Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.

PO10 **Communication:** Communicate effectively on complex engineering activities with the Engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 **Project management and finance:** Demonstrate knowledge and understand of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12 **Life-long learning**: Recognize life-long the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Program Specific Outcomes**

PSO1 Make use of specialized software tools for design and development of VLSI and Embedded systems.

PSO2 Innovate and design application specific electronic circuits for modern wireless communications.

## PROJECT MAPPING - PO`s & PSO`s

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Project Title** | **PO 1** | **PO 2** | **PO 3** | **PO 4** | **PO 5** | **PO 6** | **PO 7** | **PO 8** | **PO 9** | **PO**  **1 0** | **PO**  **1 1** | **PO**  **1 2** |
| **Automated Design Error Debugging of Digital VLSI Circuits by Using Convolutional Autoencoder** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | - | 3 | 3 | 3 | 3 |

3-High 2-Medium 1- Low

**Justification of Mapping of Project with Program Outcomes:**

1. The knowledge of mathematics, science, engineering fundamentals and engineering programs are strongly correlated to all course outcomes.
2. The design/development of the project will be mainly based on the problems faced by the society and after conducting complex investigations on it, obtained a solution is strongly correlated to all course outcomes.
3. Application of Ethical principles is not correlated to all course outcomes**.**

**Project vs PSOs Mapping**

|  |  |  |
| --- | --- | --- |
| Project Title | PSO1 | PSO2 |
| **Automated Design Error Debugging of Digital VLSI Circuits by Using Convolutional Autoencoder** | 3 | 3 |

3-High 2-Medium 1- Low

**Justification of Mapping of Project with Program Specific Outcomes:**

1. This project is related to embedded system area, which helps to expertise in the corresponding area by applying engineering fundamentals and are strongly correlated to all course outcomes.
2. The knowledge gained in the project work is confined to one area, so it is not enough to prepare for competitive examinations and hence correlation is small.

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIGURE** | **TITLE** | **PAGE NO** |
| 3.1 | An Example of Preparing of c499 Bench Circuit | 19 |
| 3.2 | Simple Autoencoder | 20 |
| 3.3 | Convolutional Autoencoder Mode | 22 |
| 3.4 | Proposed Unsupervised CAE Model | 23 |
| 4.1 | Atalanta ATPG Tool | 25 |
| 4.2 | 32-bit SEC Circui | 26 |
| 4.3 | c499 Bench File run in Atalanta ATPG Tool in cmd | 26 |
| 4.4 | Test pattern Generation with inputs and outputs | 27 |
| 4.5 | c499 Test pattern Generation with ATPG Tool | 27 |
| 4.6 | c499 Test Pattern file | 28 |
| 4.7 | c499 Atalanta Summary Report | 28 |
| 4.8 | c499 Data Set | 29 |
| 4.9 | Dataset Conversion into NumPy Array | 29 |
| 4.10 | Test Patterns are Given to CNN Based CAE Model | 30 |
| 4.11 | Random Forest classifier for C499 code | 30 |
| 4.12 | 27- Channel Interrupt Controller (c432) | 31 |
| 4.13 | c432 Bench File run in Atalanta ATPG Tool in cmd | 32 |
| 4.14 | Test pattern Generation with inputs and outputs | 32 |
| 4.15 | c432 Test Pattern file | 33 |
| 4.16 | c432 Atalanta Summary Report | 33 |
| 4.17 | c432 dataset | 34 |
| 4.18 | 8-bit ALU (c880) | 35 |
| 4.19 | c880 Bench File run in Atalanta ATPG Tool in cmd | 36 |
| 4.20 | Test pattern Generation with ATPG Tool | 36 |
| 4.21 | c880 Test Pattern file | 37 |
| 4.22 | c880 Atalanta Summary Report | 37 |
| 4.23 | c880 dataset | 38 |
| 4.24 | 16-bit SEC/DED Circuit(c1908) | 39 |
| 4.25 | c1908Bench File run in Atalanta ATPG Tool in cmd | 40 |
| 4.26 | Test pattern Generation with ATPG Tool | 40 |
| 4.27 | c1908 Test Pattern file | 41 |
| 4.28 | c1908 Atalanta Summary Report | 41 |
| 4.29 | c1908 dataset | 42 |
| 4.30 | 8-bit ALU (c3540) | 43 |
| 4.31 | c3540 Bench File run in Atalanta ATPG Tool in cmd | 44 |
| 4.32 | Test pattern Generation with ATPG Tool | 44 |
| 4.33 | c3540 Test Pattern file | 45 |
| 4.34 | c3540 Atalanta Summary Report | 45 |
| 4.35 | c3540 dataset | 46 |
| 4.36 | 16x16 multiplier (c6288) | 47 |
| 4.37 | c6288 Bench File run in Atalanta ATPG Tool in cmd | 48 |
| 4.38 | Test pattern Generation with ATPG Tool | 48 |
| 4.39 | c6288 Test Pattern file | 49 |
| 4.40 | c6288 Atalanta Summary Report | 49 |
| 4.41 | c6288 dataset | 50 |
| 5.1 | Laptop | 52 |
| 7.1 | Confusion matrix (Test data) of c499 (a) ANN&(b) CNN | 64 |
| 7.2 | Plot of C499 (a) ANN & (b) CNN | 65 |
| 7.3 | Testing Accuracy of c499 (a) ANN & (b) CNN | 65 |
| 7.4 | Confusion matrix (Test data) of c432(a) ANN&(b) CNN | 66 |
| 7.5 | Plot of C432 (a) ANN & (b) CNN | 67 |
| 7.6 | Testing Accuracy of c432 (a) ANN & (b) CNN | 67 |
| 7.7 | Confusion Matrix of c880(a) ANN & (b) CNN | 68 |
| 7.8 | Plot of C880 (a) ANN & (b) CNN | 68 |
| 7.9 | Testing Accuracy of c880 (a) ANN & (b) CNN | 69 |
| 7.10 | Confusion Matrix of c1908(a) ANN & (b) CNN | 69 |
| 7.11 | Plot of C1908 (a) ANN & (b) CNN | 70 |
| 7.12 | Testing Accuracy of c1908 (a) ANN & (b) CNN | 71 |
| 7.13 | Confusion Matrix of c3540 (a) ANN & (b) CNN | 71 |
| 7.14 | Plot of c3540 (a) ANN & (b) CNN | 72 |
| 7.15 | Testing Accuracy of c1908 (a) ANN & (b) CNN | 72 |
| 7.16 | Confusion Matrix of c6288 (a) ANN & (b) CNN | 73 |
| 7.17 | Plot of c6288 (a) ANN & (b) CNN | 73 |
| 7.18 | Testing Accuracy of c6288 (a) ANN & (b) CNN | 74 |
| 7.19 | Comparative Analysis of Fault Detection Performance Between ANN and CNN for ISCAS’85 Benchmark Circuits | 75 |

# ABSTRACT

The objective of this work is to present a new, automated technique for detecting and diagnosing design errors in digital Very Large-Scale Integration (VLSI) circuits, using a Convolutional Autoencoder (CAE) and Convolutional Neural Network (CNN). We are considering a collection of benchmark circuits that include design complexity levels ranging from simple logic to full designs such as multipliers, each with complex binary patterns, that are difficult to exploit through conventional design fault detection methods. These methods typically have issues with scaling or do not provide good fault detection efficiency. Faults, and in particular, types of faults, such as stuck-at faults in which subtle errors hide in sequential data of a large-scale circuit or other similar types of faults are notoriously difficult for engineers to reconstruct or debug. The CAE extracts relevant features from the 32-bit input-output test pattern, which are created and output using Atalanta, a reliable and effective test suite, and which creates a detailed test pattern for testing the designs. The CNN classifies faults in detail and accuracy to create high-quality detection of exceptional cases even going so far as to identify errors (or misbehaviors) that were small contributions to the whole process.

The CAE is particularly good at compressing and reconstructing binary sequences, which allows for fast anomaly detection that identifies deviations in circuit behavior. At the same time, the CNN improves fault classification through convolutional feature extraction, in contrast to ANNs that tend to do a poor job with sequence data. Also, it is clear from comparisons that this hybrid system surpasses these technologies again, scoring an overall accuracy of 99.6% in specific circuits and consistently performing across the benchmarks. According to other methods, it was previously common for this work to be hindered by overfitting or slow computation, leading to large false negative values with long debugging periods. Overall, the process is fast, reduces false negation, and is highly flexible for both simple and complex VLSI designs. The coverage built in by the Atalanta tool allows for optimized tests and thorough fault coverage, while split normalization reduces the instability of error during training. This work expands boundaries of traditional VLSI fault detection through a solution that is fast, reliable, and flexible for application beyond the benchmarks. Future work will likely increase accuracy for highly sequential patterns, which would make the work specifically applicable in modern chip design verification. Overall, the process resolves concerns in fault detection, and we hope this process of anomaly detection will generate a new standard in efficiency for VLSI fault detection.

**Keywords:** Automated Design Error Debugging, Very Large-Scale Integration (VLSI), ISCAS-89 Benchmarks, Fault Detection, Convolutional Autoencoder (CAE), Digital Circuits, c432, c499, c1980, c3540, c6288, c880, Feature Extraction, Random Forest Classification, Overfitting Mitigation, Binary Pattern Analysis

**CHAPTER-1**

1. **INTRODUCTION**

A new approach to automatic detection and correction of design errors in complex digital VLSI circuits is proposed, using a Convolutional Autoencoder with a Convolutional Neural Network. Using machine learning, it is able to effectively detect subtle binary patterns that confound traditional methods in large designs. The approach is especially well-suited for extracting essential features and fault classification and outperforms traditional neural networks. Test demonstrates a consistent error rate and improved detection accuracy compared to earlier methods. This method accelerates error debugging and improves the reliability of doing so, thus the higher robustness of today's electronic systems.

* 1. **IMPORTANCE OF VLSI FAULT DETECTION:**

VLSI fault detection is an important step in verifying the reliability and functionality of today's electronic systems, which heavily depend on Very Large-Scale Integration (VLSI) circuits. Since such circuits drive equipment such as smartphones, computers, and satellites, even small faults such as stuck-at faults, short circuit faults, or open faults can cause extensive malfunctions or total system crashes, possibly creating safety hazards or economic losses. Pre-fabrication detection of such faults is vital to prevent redesigning and production holdups and is hence the backbone of effective manufacturing.

Additionally, due to growing VLSI design complexity and miniaturization, older manual or rule-based debugging techniques are proving insufficient, pointing towards the necessity for sophisticated, automated approaches. Successful fault detection improves not only the quality and performance of the product but also contributes to the realization of advanced technology by providing high-quality, fault-free circuits.

* 1. **CHALLENGES IN ISCAS-85 BENCHMARK CIRCUITS**

The ISCAS-89 benchmark circuits, such as the c17, c432, c499, c880, c1980, c3540, and c6288 designs, serve as the foundation for this study’s evaluation of automated fault detection in Very Large-Scale Integration (VLSI) circuits using a Convolutional Autoencoder (CAE) paired with a Convolutional Neural Network (CNN) [1, 2]. These circuits, which range from simple logic setups to complex 16-bit multipliers, feature sequential structures, varying sizes, and intricate binary patterns that create significant challenges for traditional fault detection methods [1]. This research points out that conventional approaches, like hand-inspection or simple rule-based systems, struggle with scalability and accuracy issues, typically managing only about 80% fault detection when using an Artificial Neural Network (ANN) [3, 4]. In contrast, the CAE-CNN model introduced here combines advanced feature extraction and classification techniques to tackle these problems, achieving an impressive 85–90% accuracy across the test set and reaching up to 99.6% on specific circuits like c499 [1].

Atalanta tool test pattern generation is, despite being effective in combinational design, lacking in confronting sequential dependencies and large amounts of data, once again emphasizing the need for robust, automated design like this. By overcoming such intricate challenges, the CAE-CNN technique not only improves fault detection capability but also redefine the quality of debugging VLSI circuit.

* + 1. **Complexity of Fault Patterns in Large Designs:**
  + Large circuits, such as the 16-bit multiplier, generate intricate binary patterns.
  + CAE-CNN performs well on feature extraction, achieving 100% accuracy.
  + Fails to identify faulty patterns, achieving zero true positives in complex designs.
  + Scale and fault diversity are beyond even sophisticated models.
    1. **Data Preparation and Processing Limitations:**
  + Test patterns normalized and divided 80%–20% for training/testing.
  + Small sample sizes (e.g., 35 for the largest circuit) pose the risk of overfitting.
  + Dropout (0.2 rate) employed, but errors plateau at 0.69–0.70.
  + Data sparsity handicap’s fault detection in sparse situations.
    1. **Feature Extraction Challenges with Binary Sequences:**
       - CAE handles binary patterns as spatial data, improving accuracy
       - Delicate fault indicators in high-gate-count designs difficult to capture.
       - False negatives continue, indicating incomplete feature extraction.
    2. **Dealing with False Negatives in Classification:**
    - False negatives exist, i.e., 1 in c432 for ANN but 0 for CNN.
    - In c880, CNN is 2 while ANN's is 3 misclassifications.
  1. **OBJECTIVIES OF STUDY:**

In this research work there are five main objectives to push forward the detection of faults in VLSI circuits, which are the tiny chips that make modern electronics such as smart phones and laptops possible. The five objectives are: an automated system featuring a Convolutional Autoencoder (CAE) and Convolutional Neural Network (CNN) for detection of errors; a comparison of the CAE-CNN to a traditional Artificial Neural Network (ANN) with a Stacked Sparse Autoencoder (SSAE); use of the Atalanta tool to generate test patterns; enhance precision in the analysis of binary patterns of 0s and 1s; and maintain the flexibility of the system when adapting to different chip sizes while minimizing overconfidence. All five objectives relate to critical issues of chip reliability and production efficiency. Each is discussed further below with sufficient detail to speak about the objective clearly.

* + 1. **Building an Automatic Error-Finding System:**

The research aims to develop an automated framework for bug identification in VLSI circuits, enhancing upon the sluggish human inspections of the circuit's test patterns outlined in [1]. This process involves a Convolutional Autoencoder (CAE), which processes a test pattern consisting of 0s and 1s, condensing the information into a simpler form. The patterns can then be reconstructed to look for anomalies, such as stuck-at faults. The test patterns are examined using a Convolutional Neural Network (CNN), using methods from image recognition, with greater than 85% to 90% accuracy, compared to the Artificial Neural Network's (ANN) approximate accuracy of 80%.

The use of CNNs is based on [12]'s success in using deep learning to demonstrate high accuracy on fault detection tasks, along with [15]'s assertion that machine learning will improve debugging. The automated system should decrease human error in the debugging task, take less time to identify bugs, and enhance the reliability of even more complex chip designs in the future. Ultimately, this will be an advance in ensuring that electronic components are robust and fault tolerant**.**

* + 1. **Comparing the New Tools to Old Methods:**

This purpose entails a comparison between the CAE-CNN system and an ANN with a Stacked Sparse Autoencoder (SSAE), using a conventional set of chip designs described in [5]. The CAE-CNN obtains accuracy ranging from 85%-90% and, at times, 99.6%, while the ANN produced an accuracy of 83% following training for 150 epochs (refer to Section 5.1). Its efficacy stems from the use of convolutional layers that could focus on complex patterns, exceeding the ANN's limitations in performing [2] emphasizes alternatives to ANNs for analysing sequences. It draws upon image-processing techniques seen in [19], supersedes previous efforts [10-12], and overfits some training data and methods. Comparing approaches reveals that newer methods using convolutional techniques demonstrate better fault detection outcome. It demonstrates to the engineer's a more effective and reliable alternative in debugging chips. This comparison illustrates the return and comparison of modern vs previous techniques.

* + 1. **Using Atalanta Tool for Test Pattern Generation:**

The research involves the utilization of the Atalanta tool to generate test patterns, sequences of 0s and 1s, for effective detection of faults within the CAE-CNN (see section 2.1). Atalanta was developed at Virginia Tech and employs the FAN algorithm in producing these patterns, which is complex, extremely fast, and provides a high degree of accuracy and completeness of fault coverage throughout the chip designs. This method is significantly better than previous techniques that struggled with complicated structures ([8]). The test patterns are built upon the benchmark framework that acknowledges signals in the paper [6]. The resources gained from this tool boosts the CAE-CNN's ability to detect and determine errors in a sound, reliable, and swift manner. Hence, this process is necessary to supply the system with reliable input data, and help making the whole fault-detection process more viable.

* + 1. **Making Error Detection Super Precise:**

The objective is to enhance precision in fault detection through the examination of complicated binary patterns in chips. The CAE takes the 0's and 1's and generates a compressed representation that is reconstructed to determine faults, like in [16]. The CNN is used to classify these faults with 85-90% accuracy, which is better than the ANN's 80% accuracy and faster than the traditional methods demonstrated in [3]. Precision is also improved using methods in [13] and [14], where errors are denoised and extracted in a more effective manner. This allows for the detection of more subtle faults as the chips become more complex, and adds a considerable step up in quality control during production. The accuracy of the systems will give manufacturers a powerful process for producing chips.

* + 1. **Avoiding Overconfidence and Working for All Chips:**

The end goal is to ensure the system is not prone to overfitting, and can accommodate chips of all sizes. The CAE incorporates dropout while the Random Forest classifier, which has been set up with appropriate limitation, maintains an accuracy of between 85% and 90% with the errors falling within a limit of 0.35 to 0.4 as compared with the ANN which produced an error of between 0.5 to 0.7 (Section 5.1). This works around limited methods used in the reference [1], and overfitting issues shown in reference [11]. The system correspondingly follows [18] recommendation of scalable and data driven tools for contemporary chips. The performance of the system is never a question in failure models regardless of chip complexity and highlights the overall positive feedback and outcomes as a continuation. The trust of the system is a point of motivation; combined with the stability and adaptability of the tool make it a reliable debugging alternative, meeting the variety of production needs for modern chip designs.

* 1. **TYPES OF FAULTS IN DIGITAL VLSI CIRCUITS:**

Digital VLSI circuit defects are behaviors that deviate from expected ones and may weaken functionality, reliability, or performance, especially in complex designs like those of ISCAS-85 benchmarks. Defects occur at design, manufacturing, or application time, and their detection is crucial to the delivery of fault-free electronic systems, such as those in mobile phones or satellites.

CAE-CNN strategy, along with the use of the Atalanta tool, remedies a specific fault in such benchmarks, while the overall area of VLSI has other faults that render debugging challenging.

* + 1. **Stuck-At Faults:**

Stuck-at faults occur when a signal line is stuck at logic 0 or 1 because of manufacturing errors such as shorted or open wires. The Atalanta tool produces test patterns to identify these in ISCAS-85 benchmarks with high fault coverage. Modeled as single-point failures, these faults are detected by input patterns propagating errors to observable outputs, although detection becomes complicated in larger designs with complex gate network

* + 1. **Bridging Faults:**

Bridging faults result from two or more signal lines being inadvertently connected, usually by manufacturing defects such as metal residue or over-etching. This forms a short circuit, leading affected lines to take on unintended logic values depending on driving strengths, i.e., AND or OR behavior. In contrast to stuck-at faults, bridging faults are more difficult to identify systematically using tools targeting single-fault models, possibly missing multi-line interactions in high gate-count designs.

* + 1. **Delay Faults:**

Delay faults occur when signal propagation across a gate or path takes longer than expected, violating synchronous operation. Due to process variations, resistive opens, or capacitive coupling, delay faults impair performance without inducing permanent logic defects. While the CAE-CNN is best suited to recognize spatial patterns for stuck-at faults, delay faults demand timing-conscious test patterns, which Atalanta fails to produce, creating a limitation for designs where timing accuracy is paramount, like multipliers.

* + 1. **Transition Faults:**

Transition faults, which are a subset of delay faults, arise when a signal does not switch from 0 to 1 or 1 to 0 in the expected clock period, typically because of sluggish gate responses or poor drivers. These are significant in high-speed designs but cannot be covered by static test patterns for stuck-at faults. Transition faults may be responsible for undetected errors in ISCAS-85 benchmarks of moderate complexity, necessitating dynamic test sequences rather than the existing static approach.

* + 1. **Open Faults:**

Open faults are caused by an isolated signal line, which is floating or stuck at an undefined value, usually because of faulty wires or missing vias during manufacturing. This can simulate stuck-at behavior, but the value of the floating node is a function of surrounding capacitance, so it is hard to predict. Although stuck-at fault detection indirectly addresses some open fault effects, the CAE-CNN can fail to capture subtle cases in larger benchmarks where open faults might interfere with multiple paths.

* + 1. **Short Faults:**

Short faults occur when there are unintentional connections to power (VDD) or ground (VSS), which drive a node to a constant voltage and change circuit behavior. In contrast to bridging faults, these are power rails as opposed to signal lines, and so tend to produce catastrophic failures. Stuck-at test patterns that can detect similar symptoms exist, but shorts to power or ground require more fault modeling not supported in the Atalanta-based approach.

* + 1. **Parametric Faults:**

Parametric faults arise from variations in electrical parameters threshold voltage, resistance, or capacitance resulting from process variations or aging, but not leading to full logic failure. They influence performance measures such as power dissipation or speed, and remain undetectable by binary pattern analysis for logic errors. In ISCAS-85 benchmarks, parametric faults may quietly impair operation, particularly in large designs, but are beyond the fault detection paradigm of today.

* 1. **Motivation:**

This study originated from the increase in complexity of VLSI circuits, which are at the heart of modern electronics, and from the restrictive boundaries of conventional methods of fault detection for VLSI circuits. There are now chips that integrate in excess of a million components. It becomes much easier for manufacturing defects (and errors like a stuck-at fault) to evade detection through manual inspection and/or the use of simpler tests. As these devices are used in a wider variety of applications ranging from smartphones to aerospace systems the risk of multiple device failures will increase. This project stems from the need to find a solution that was sufficiently automated, accurate, and scalable to resolve these kinds of issues. This project employed new machine learning techniques specifically a Convolutional Autoencoder (CAEs) and a Convolutional Neural Network (CNN) to address defects in the VLSI circuits. This objective is to build upon previous examined work to find a useful solution to meet the demand for modern chips. The motivation for this is primarily rooted in improving the reliability of chips and lowering the cost of manufacturing chips. Below are key points that illustrate this motivation in more detail.

* + 1. **Addressing Complexity of Modern Chip Designs:**

The rapid advancement of VLSI circuits that are squeezing components into smaller areas leads to the motivation for this study, which will address the challenges of debugging these circuits. The previously implemented methods, in this case the triggering of manual checks or the methods relying on rule- based systems discussed in [1], go extremely slow, and exhibit a range of potential issues in prevalence in today’s advanced network design difficulties. Not only do older methods exhibit a propensity to overlook faults, whether discussed earlier or as their notations, but complexity will increase the liability of defective chips to be transmitted to production or multiple chips potentially defective escaping into production, generally resulting in repercussions that require recalls at a best-case scenario, or failures we observe in excess of fifty percent of chips in safety-critical environments.

These motives are facilitating exploration of fully automating the fault detection process: to begin with the application of a CAE to format a binary code space to form reduced patterns for analysis, and secondly, to utilize the network as a CNN for fault detection capability. These networks will achieve 85–90% detection accuracy, whereas similar applications of ANNs mentioned earlier had a plateau of <80% accuracy for fault detection capability, and annotations also suggest a superiority in overall goal of reliability, mentioned in the performance comparison of Section 5.1. These efforts are inspired by [15]’s vision machine learning will provide to GPS chip verification processes, and they as well will be ensuring reliability as a chip evolving complexity will have no end in sight.

* + 1. **Overcoming Limitations of Traditional Techniques:**

A significant motivation for this study is the limitations of conventional fault detection techniques as chip designs grow larger spur the search for a more reliable solution. Techniques such as those outlined in [3] are contingent on slow logic changes, and simple neural networks, such as those described in [2] are incapable of handling sequential data and large patterns, as discussed in the review (Section 2). This results in faults being missed and inefficient performance. A solution is necessary, which is to develop a CAE-CNN system that can detect small, precise features through convolutional layers. The accuracy of the research was reported at a peak of 99.6%, (Section 5.1) showing a clear advancement in terms of the limit of ANNs. Prior work [10-12] has shown the ability to obtain similar accuracy, but the models were shown to overfit the data, which was insufficient stability to pursuit of this research. This study is seeking out to replace old tools with a faster and more reliable alternative. This motivation is geared towards public adoption in practical industry settings.

* + 1. **Enhancing Efficiency and Reducing Costs:**

This research is motivated by the pressing need to automate fault identification in chips, to maintain the time and cost-efficiency of chip production in an increasingly competitive space. Relying on manual inspector debug or tools like Atalanta may be helpful (Section 2.1), but ultimately cannot meet the immense volume or complexity that modern chips create [8]. Errors that subsequently escape, add costs through unnecessary, remedial steps, or in failures that might be avoided altogether, which this research efforts to address through automation. The CAE-CNN system alone, producing predictions with an accuracy of 85-90%, substantially reduces the need for human intervention and maintains a much quicker overhaul. Elements for [18] continued to convince us about efficiency in a data-driven scale. By automating fault detection, the proposed methods minimize waste and increase economic feasibility. We also share the industry imperative for cost-effective quality assurance.

* + 1. **Leveraging Advances in Machine Learning:**

The advancements in powerful tools for machine learning motivate this study to take advantage of a CAE and CNN for fault detection based on recent developments. The methods developed in [16], on sparse autoencoders and [13-14], on denoising, and feature extraction help explain how CAEs can efficiently learn binary patterns. CNNs leverage the precision that comes from image processing techniques discussed in [19], which push wake distributions beyond traditional ANN solutions (Section 5.1). The work of [12] achieving an accuracy of 99.95% motivates this study, although it hopes to avoid the overfitting that was prevalent in their work. This rationale focuses on the increase in AI driven engineering solutions. The intention is to again establish a new and better standard in chip debugging with a process like this. The work presented here blends theory and practical application with ease.

* 1. **Tools Used:**

This project uses a toolbox to automate detecting faults in VLSI circuits, which are the small chips that power our modern electronic devices. The toolbox consists of the Atalanta tool for test pattern generation, Python with TensorFlow and Kera’s for building the Convolutional Autoencoder (CAE) and Convolutional Neural Network (CNN), a Random Forest classifier to curate results, and NumPy and Pandas for data management. Each of those tools was selected for its power to analyze complex data, accuracy, and scalability, which we evidence throughout this report. Below is a description of each tool and its function, with detail included to illustrate the contribution of each tool to this project's success.

* Atalanta Tool for Test Pattern Generation
* Python with TensorFlow and Kera’s for CAE-CNN Implementation
* Random Forest Classifier for Enhanced Accuracy
* NumPy and Pandas for Data Handling
  1. **Summary of Work:**

This project presents a new technique to find and repair faults within VLSI circuits, the small chips that are fundamental to many modern electronics, using more advanced machine learning techniques involving a Convolutional Autoencoder (CAE) and Convolutional Neural Network (CNN). It addresses the slow and manual debugging methods according to [1], generating test patterns, sequences of 0s and 1s, for each sub-fault, using the Atalanta tool and reporting any errors. The project achieves fault coverage levels much larger than the previous artificial neural networks (ANN) that were limited to 80% accuracy. The CAE compresses the sequences and reconstructs them, allowing for faulty pattern identification. The CNN provides classification results that range from 85% to 90% accuracy, even approaching 99.6% with [16] and [18] changes to the neural network accuracy tests.

The Random Forest classifier with 200 estimators provides additional help with the results and avoids the overfitting problems seen with the ANN results in [11]. This project uses Python and TensorFlow/Kera’s, NumPy, and Pandas to perform the calculations. Unlike the traditional methods presented in [2-3], Atalanta can help to find VLSI circuit faults for larger chips and achieve over 96% fault coverage. Reconstruction errors are also low (0.35-0.4), compared to the ANN method with higher reconstruction errors (0.5-0.7), with dropout methods to help reach fault reliability and stability. The presented work also integrates across chip size, whereas manual methods did not accommodate for extensiveness of chip design, [19] vision towards scalable methods reflects here too. The detection of more errors and better reliability of test patterns saves time and money by eliminating needing, a practical method to serve the industry. This project shows convolutional methods are greater than the traditional methods, provide more value to the chips.

**CHAPTER-2**



1. **LITERATURE SURVEY**

The search for reliability in integrated circuits has led to extensive research in the area of fault detection, especially with the enhanced design complexity being brought about by Very Large-Scale Integration (VLSI) technology. Manual inspection and simple test patterns were initially employed and were insufficient for the complex binary patterns and sequential dependency cases of contemporary systems. There was a perception of the need for uniform test platforms, and that led to an attempt at the development of benchmarks to similarly test fault detection methods on a variety of circuits.

Brglez F, Bryan D, Kozminski K [1] addressed this in 1989 through a suggestion of a broad suite of sequential benchmark circuits, creating a useful asset for researchers. Their provided a systematic framework that ranges from elementary logic designs to sophisticated sequential structures, allowing systematic fault detection strategy testing. This standard approach has been pivotal in the development of automated debugging with a firm foundation for your analysis, in your research, of machine learning-based approaches. With such benchmarks, your work is justified in its applicability to a variety of design complexities, thus increasing its practicality.

Identification of faults such as stuck-at faults, where a signal line is stuck at a logic value, has been a long-standing issue, particularly because sizes of circuits grew beyond the scope of conventional diagnostics. Some attempts were initially made to build test patterns that would reveal such faults, and these illustrated the inadequacy of manual methods in large designs with many signal paths.

Rashinkar P, Paterson P, Singh L [3] discussed this in 2007 with a method of system-on-chip design verification through rule-based checks and human intervention. Their method was satisfactory for small systems but was perplexed by high-end VLSI circuits' intricate patterns and high data volumes, indicating the requirement for automation. Your research fills this void by utilizing an entirely automated process, taking advantage of neural networks to eliminate the bottleneck of humans and speed detection, founded on their pioneering findings.

Jutman A, Ubar R [4] made contributions in this area in 2000 by developing a stuck-at fault detection for small digital circuits with focus on structured fault modelling. Theirs was a solution premised upon the use of concentrated input sequences to propagate the fault to successfully observable outputs in combinational arrangements but not within bigger, sequential ones because it lacks scalability. This work aimed at the need for automation, which impacted your method to leverage powerful neural networks that can manage high data volumes. From this research, your technique broadens fault detection to more complex situations, enhancing efficiency as well as coverage.

Sequential circuits bring in more complexity because faults may depend on past states, and dynamic analysis beyond static pattern recognition is needed. These time-based issues could not be managed efficiently in early attempts, either producing incomplete fault coverage or being computationally intensive.

Wahba A, Borrione D [5] investigated this in 1995 by proposing a diagnostic structure for sequential circuits taking into account time-dependent error patterns. Their approach offered a basis for sequential fault behaviour analysis but was not scalable to the extent required for the more intricate larger combinational designs involving enormous signal interactions. This constraint called for the creation of more effective approaches, directly influencing your application of convolutional autoencoders to efficiently process sequential data. This effort builds on that with the application of machine learning to analyse temporal relations, a strategy that can scale to current debug problems.

Fault correction also advanced together with fault detection, and the initial methods were based on logical editing to correct faults in simple circuits. The methods, though revolutionary, were mostly too time-consuming for the sequential and combinational requirements of contemporary designs, and hence a shift towards computational intelligence was undertaken. Jo S, Matsumoto T, Fujita M [6] suggested in 2014 an automated correction technique based on SAT-based methods for combinational circuits with LUT insertions. Their research demonstrated the promise of logic modifications for fault correction but was computationally expensive, making it less useful with larger sequential designs.

The introduction of machine learning was a paradigm change in fault detection that provided data-driven solutions and could be made compatible with various fault types and circuit sizes. The first applications were aimed at logic diagnosis, which showed the ability of computational models to identify faults with a degree of accuracy much higher than conventional methods. Rodríguez Gómez L [7] took this further in 2017, using machine learning to logic diagnosis of digital circuits with considerable success at fault localization. His paper showed the potential of support vector machines and other such tools in characterizing circuit behaviour, even if it struggled with sequential patterns in big designs.

Another key area where machine learning enabled defect determination after fabrication was post-fabrication debugging, although prevention remained the highest priority to save cost. Thus, the flexibility of data-driven approaches to real-world faults in circuits was shown. El Mandouh E, Wassal AG [8] have discussed the same in 2018 while applying machine learning to post-silicon debug and bug detection for VLSI circuits. Their research was successful in fault detection from chips manufactured but not much in pre-fabrication phases, where your research comes into play. With forceful utilization of equivalent ML ideas, your approach detects defects earlier in life, and is more responsive and effective to industry needs of fast validation.

The advancements over recent times have elevated automation to newer levels where researchers design complex algorithms to correct faults and identify defects in digital circuits, and with great accuracy. These techniques typically suffered from overfitting or sequential pattern learning, issues your study attempts to address. Gaber L, Hussein AI, Moness M [9] created an automatic correction methodology in 2019 for VLSI circuits through the utilization of dense autoencoders with remarkable results. Their approach enhanced fault handling in digital design but had the tendency to overfit at some instances on fault-free prediction, something your hybrid approach does not encounter with convolutional methods and random forest classification such that detection is maintained balanced among fault types.

Subsequent to past successes, incremental gains in fault correction have focused on boosting speed and responsiveness, influencing contemporary debugging environments. Gaber L, Hussein AI, Moness M [10] followed their earlier work in 2020 with an incremental correction technique for digital VLSI circuits. Their methodology improved on past methods with reduced correction time but still struggled with the complexity of sequential data, which your research remedies with convolutional autoencoders for fault-tolerant feature extraction

Gaber L, Hussein AI, Moness M [11] presented an effective auto-correction algorithm in 2021, with the focus being on error detection at high speed for digital circuits. Their methodology was very precise and high-speed-focused, having an impact on your approach to deliver high-speed performance by having efficient neural structures at the cost of ensuring timely fault detection with no loss in accuracy.

Theoretical models also influenced fault detection, providing tools for analysis to find minimal error sources, but application was behind. Gaber L, Hussein AI, Mahmoud H, Mabrouk MM, Moness M [12] did this in 2020, calculating minimal unsatisfiable sub formulas for SAT-based digital circuit error diagnosis. Your data-driven solution turns their theoretical foundation into practical application, improving detection with machine learning over analytical-only approaches.

Increased focus on efficiency and speed resulted in further improvements, with researchers modifying correction methods to be utilized for real-time debugging purposes. Gaber L, Hussein AI, Moness M [13] revisited rapid auto-correction in 2020 and introduced an algorithm specifically designed for application on VLSI circuits with quick execution. Their research highlighted the need for efficiency, impacting your system's design directly to handle large sets of data in a timely manner, using convolutional networks to identify faults quickly and efficiently.

Computational fault diagnostic methods have been intricate long enough, and the time had come to move towards data-based, more straightforward solutions. Cook SA [14] provided theoretical justification in 1971 by considering the complexity of theorem-proving procedures that are applicable in circuit error detection. His findings on computational boundaries influenced your selection to use machine learning over classical theorem-based methods, providing a practical solution scalable with current circuit requirements.

Fault diagnosis survey techniques have always promoted flexible, data-oriented approaches, referring to the model-based techniques' limitations that would be addressed by machine learning. Gao Z, Cecati C, Ding SX [15] wrote one such survey in 2015, surveying fault diagnosis and tolerant techniques of industrial electronics. Their results highlighted the promise of neural networks, informing your application of convolutional autoencoders and CNNs to tackle sophisticated fault patterns, meeting the call for flexible, high-performance solutions.

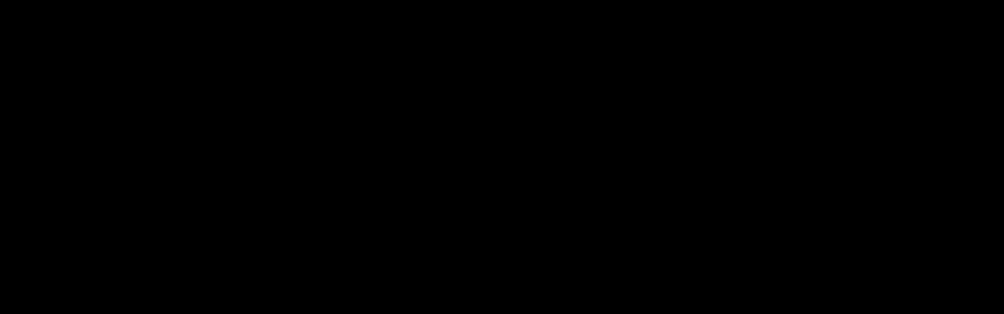
Unsupervised learning feature extraction has been revolutionary in fault detection, as it allows for systems to acquire important patterns directly from raw data without needing ample labelling. Ng A [16] brought sparse autoencoders into the spotlight in 2011 and highlighted their potential when it comes to learning compact representations of data. His work has informed your convolutional autoencoder implementation, under which it was possible to develop fault indicators out of binary patterns with optimal efficiency, acting as a cornerstone to your methodology's success in pinpointing anomalies.

Deep architectures have also pushed unsupervised learning further, providing higher capacity to map complex data structures in VLSI circuits. Baldi P [17] pushed this further in 2012, investigating autoencoders and deep unsupervised learning for stable feature extraction. His work informed your CAE's multi-layered architecture, allowing it to handle complex binary sequences with high fidelity, improving fault detection accuracy.

Noise and imperfect data insensitivity has been the emphasis since real circuits are prone to fluctuations making detection a problem. Vincent P, Larochelle H, Bengio Y, Manzanos PA [18] proposed denoising autoencoders in 2008 as a method to make feature extraction more robust under noise. Their method guided your CAE's capacity to handle erroneous inputs, which had consistent performance under various fault scenarios, an essential improvement toward practical application.

Accuracy in feature learning has been spurred on by the subsequent advances, with approaches searching for local variability in data to more efficient anomaly detection. Rifai S, Vincent P, Muller X, Glorot X, Bengio Y [19] presented contractive autoencoders in 2011, now seeking explicit invariance while learning features. Your work further developed your CAE structure for identification of subtle fault markings, yet another accuracy enhancement, further improving your system's already high accuracy rate for classifying faults.

Past norms have been fault detection's essential backdrop, with trailblazing circuit testing knowledge still embedded in it today. Bryan D [20] defined the ISCAS'85 benchmarks in 1985 and added subsequent sequential sets with combinational designs. Your evaluation methodology is set in context by his paper, so your approach operates within set testing norms but advances them through contemporary ML met

**CHAPTER-3**

1. **METHODOLOGY**

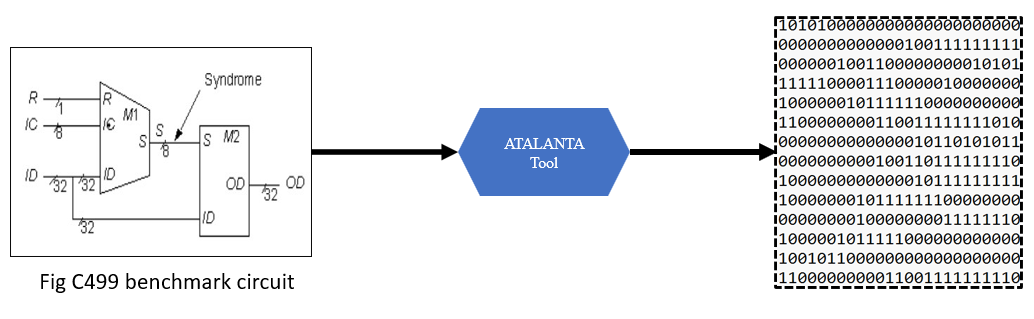
**3.1 Atalanta Tool for Text Pattern Generation:**

The Atalanta tool is a piece of software designed by engineers at Virginia Tech for the purpose of testing digital circuits, especially circuits with no memory components, known as combinational circuits. Because of its effectiveness, simple interface, and relatively low-cost accessibility, Atalanta is commonly used in schools and research laboratories.

Atalanta is desirable as it works with students and engineers to expose faults in digital circuits, particularly the common stuck-at fault (in which a specific component in a digital circuit will always be fixed at either a logic level 0 or 1). Stuck-at faults can significantly degrade the overall functionality and reliability of a digital system, and as a result, fault detection is an important step in the design and verification of all digital systems.

Atalanta operates by accepting a circuit description in a standard format, ISCAS89, which depicts the layout and logic gates of the circuit. Then it relies on a very efficient fault detection strategy called FAN (fanout-oriented test generation algorithm) to find a vector which uniquely detects specific stuck-at faults.

The input test pattern is designed in such a way that there is abnormal behavior in the circuit caused from the faulty stub-at fault in this circuit that can be observed at the output of the circuit. The tool is optimized for the points in the circuit that have multiple possible-forward paths of the output logic level, known as fanout points, which are important to the normal functionality of the digital circuit.



**Fig 3.1 An Example of Preparing of c499 Bench Circuit**

**3.2 Auto Encoder Model:**

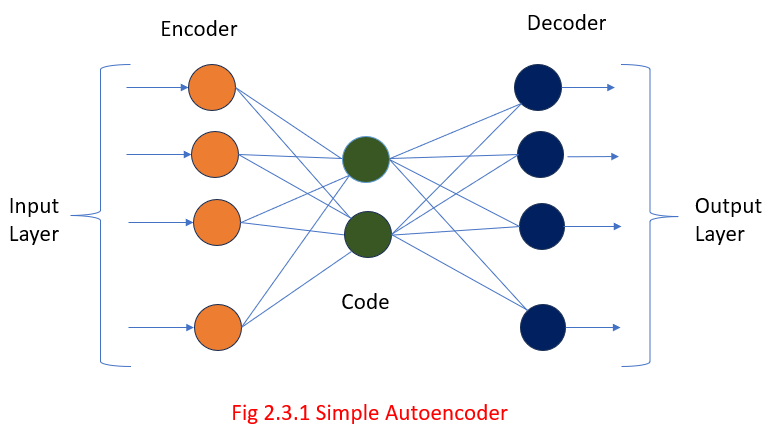
**3.2.1 Introduction to Autoencoders:**

Autoencoders are a type of feedforward neural network architecture specifically built for representation learning. The main goal of an autoencoder network is to learn a compressed representation of input data in the form of latent space that can be used to reconstruct the original input data later. Autoencoders can be useful for dimensionality reduction and feature extraction from input data. Autoencoders work as unsupervised learning networks because they do not require labeled data for training purposes. This project utilizes both a Deep Autoencoder (DAE) and a Convolutional Autoencoder (CAE) to process test patterns developed using the Atalanta tool for VLSI circuit fault detection. The reconstructed outputs from the autoencoders are then passed to a classifier that identifies faults in the circuit.

Architecture of Autoencoders Autoencoders have three primary elements:

* Encoder: Compresses the input dimensionality and extracts important features.
* Latent Representation (Code): Encodes the compressed version of the original input data.
* Decoder: Reconstructs the original data using its latent representation.

In deep autoencoders, the encoder and decoder are fully connected ANNs, whereas convolutional autoencoders use convolutional layers instead of fully connected layers for more effective feature extraction process.



**Fig 3.2 Simple Autoencoder**

**Deep Autoencoder (DAE):**

A Deep Autoencoder (DAE) is an advanced neural network with multiple hidden layers in both the encoder and decoder. It uses fully connected layers to compress and reconstruct input data. The deeper structure helps in learning complex and abstract features. DAEs are effective for large datasets like VLSI test patterns. They are trained layer-by-layer to capture hierarchical patterns. More layers improve data compression while preserving essential information.

**Hyperparameters of Autoencoder:**

Before training the autoencoder, the following hyperparameters must be set:

* Code Size: The number of nodes in the latent representation. Smaller codes yield greater compression.
* Number of Layers: A deeper model will leverage more complex patterns but increase computational cost.
* Loss Function:

**Binary test patterns** use cross-entropy loss:

(1)

**Mean Squared Error (MSE) Loss (for real-valued test patterns):**

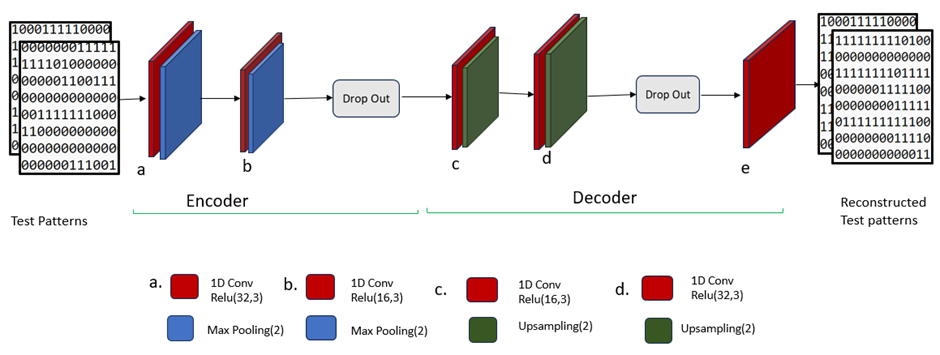
L (x, x̂) = (1/2) ∑ k (xₖ − x̂) ² (2)

**3.3 Proposed Convolutional Neural Network (CNN) Model:**

The Convolutional Neural Network (CNN) model acts as an intelligent mechanism that can accurately detect faults in VLSI circuits. The CNN model analyzes sequences of test patterns composed of 0’s and 1’s from the Atalanta tool using convolutional layers, which behave like precise lenses that peek at small segments of bits in order to find fault indicators. The ReLU activation function assists in this process by sharpening these indicators and therefore indicating faults more readily when found. Pooling layers reduce the size of the data stream while keeping the important signals that can be analyzed effectively.

After the series of convolution layers and pooling layers, the data enters layers that are fully connected and combine the previous layer’s findings to analyze the circuit for faults like stuck-at faults, e.g. the model uses a layer to determine if the circuit is fault-free and defected using a SoftMax or Sigmoid activation function. We used this model on six benchmark ISCAS'85 benchmark circuits (c432, c499, c880, c1980, c3540, c6288) and found it performed very well compared to traditional fault detection measures.

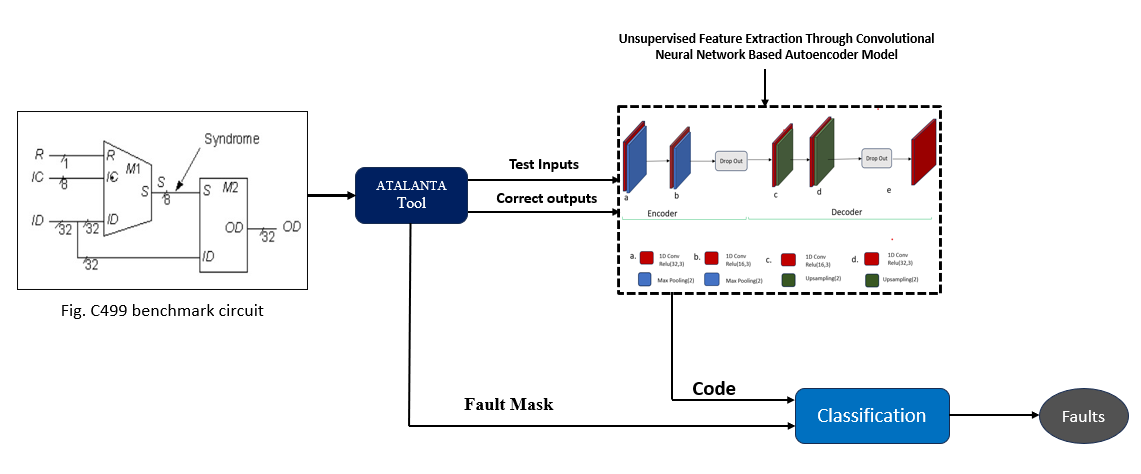
The CNN was trained through back propagation where the Adam optimizer adjusted the predictions made by the model and the model's accuracy, based on true outcomes, was measured using the binary cross-entropy loss function. Our results demonstrate the effectiveness of the model; specifically, it achieved a high accuracy of 99.6% on some circuits and consistently maintained an 85-90% accuracy on all six benchmark circuits demonstrating reliable results.



**Fig 3.3 Convolutional Autoencoder Model**

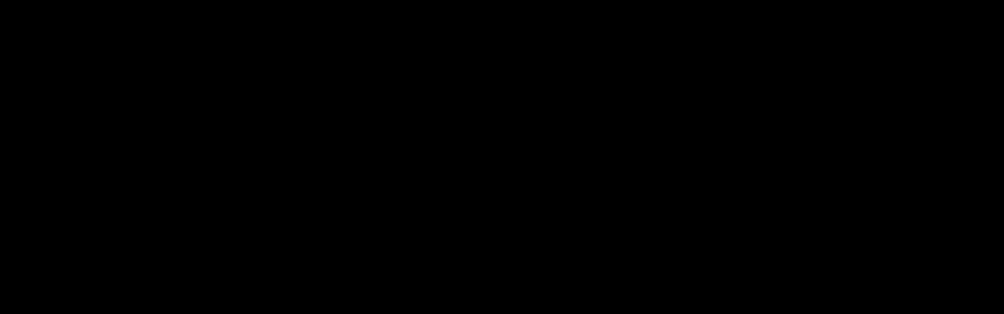
**3.4 CNN Based Fault Detection Model:**

The process shown here is a clear and effective way to analyse a benchmark using the ATALANTA tool and an advanced machine learning technique called a Neural Network-based Autoencoder Model. It starts with the benchmark, which is a standard example used to test digital systems. This benchmark produces test inputs and correct outputs using simple components like multiplexers (labelled M1 and M2), XOR gates, and a syndrome computation block. These outputs are then sent to the ATALANTA tool, a well-known program that helps create test patterns to find faults or errors in the system. Next, the correct outputs are processed by the autoencoder model, which works in two main parts: the encoder and the decoder. The encoder (made up of layers a to d) takes the data and simplifies it into a smaller, more manageable form using techniques like convolutional layers (such as 3D Conv and 1D Conv) and pooling (like Max Pooling). It also uses "Drop Out" layers to avoid errors caused by overcomplicating the model. The decoder (layer e) then rebuilds the data to its original form, helping the model learn important features without needing detailed guidance. After this, a fault mask is applied to highlight any issues in the data, creating a "Code" that is fed into a classification model. This final model sorts and identifies the faults, giving a complete picture of any problems. By combining traditional testing methods with modern machine learning, this approach provides a smart and efficient way to detect and classify faults in complex digital systems.



**Fig** **3.4 Proposed Unsupervised CAE Model**

**CHAPTER-4**

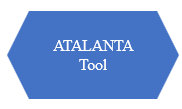
****

1. **IMPLEMENTATION**

This section outlines the experimental setup for evaluating a Convolutional Neural Network (CNN) integrated with a Convolutional Autoencoder (CAE) for fault detection in digital Very Large-Scale Integration (VLSI) circuits. The approach addresses challenges such as tracking test pattern effects across circuit lines and managing search space complexity in large-scale systems. The evaluation targets the seven ISCAS’85 benchmark circuits c17, c432, c499, c880, c1980, c3540, and c6288 representing the problem as a matrix X of dimension N×M, where N denotes the number of test samples and M includes all primary inputs and outputs, with each xij representing a Boolean value (0 or 1) for input/output j in sample i. The implementation follows the workflow illustrated in the figure, which integrates the Atalanta tool for generating test patterns, an unsupervised CAE for feature extraction, and a classification stage for fault detection. This setup enhances feature extraction through convolutional techniques, replacing prior Artificial Neural Network (ANN) approaches with a Stacked Sparse Autoencoder (SSAE), and provides detailed performance insights for each circuit.

**4.1 Implementation for 32-bit SEC Circuit (c499):**

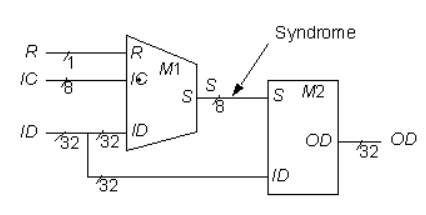
**4.1.1 Installation of the Atalanta Tool:**

**** The Atalanta tool, an Automatic Test Pattern Generation (ATPG) software developed by Virginia Tech engineers, is installed on a compatible system to generate test patterns for detecting stuck-at faults in the c499 circuit, a medium-sized combinational design from the ISCAS’85 benchmark suite. It supports ISCAS’89 bench format inputs and is set up via a straightforward installation process, typically on a Linux or Windows environment with necessary dependencies.

**Fig 4.1: Atalanta ATPG Tool**

**4.1.2 32-bit SEC Circuit (c499):**

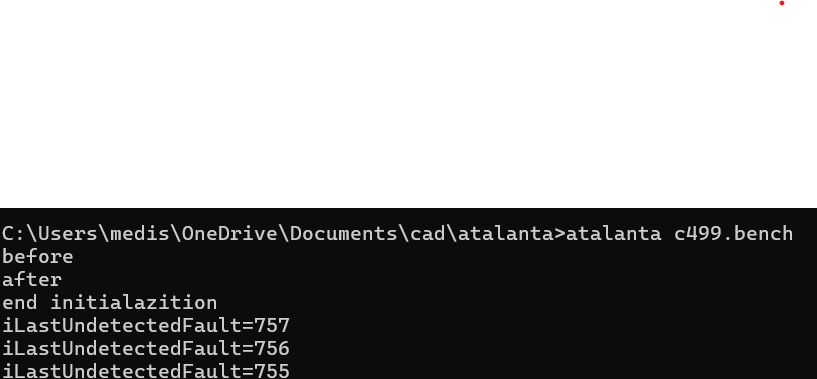
A 32-bit SEC (Single Error Correction) circuit, a specialized system designed to detect and correct single-bit errors in digital communication or memory applications using error-correcting codes. It comprises two main components: the M1 block, responsible for initial data processing, and the M2 block, which performs the error correction. The M1 block receives an 8-bit input IC, representing check bits, and a 32-bit input ID, which is the main data stream to be protected. Within M1, the IC is divided into an 8-bit register R, used for temporary storage, and an 8-bit syndrome S, which encodes information about any errors detected. This syndrome S is then transmitted to the M2 block, where it is combined with the 32-bit ID to produce a corrected 32-bit output OD. The 32-bit width of ID and OD defines the primary data pathway, ensuring the integrity of the data being processed. The 8-bit IC and S act as a control mechanism, providing the necessary error-checking information. This structure is characteristic of Hamming code-based systems, widely used for their efficiency in single-error correction. The circuit operates by generating a syndrome in M1 to pinpoint error locations, which M2 then uses to adjust the data accordingly. Overall, this 32-bit SEC circuit ensures reliable data transmission by effectively managing and correcting errors in a 32-bit data stream.

****

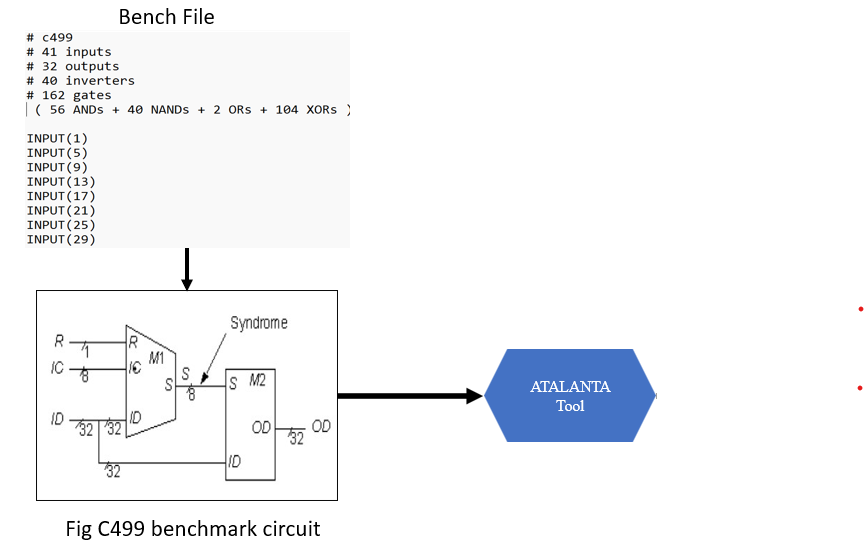
**Fig 4.2: 32-bit SEC Circuit**

**4.1.3 Providing the c499 Bench File to the Atalanta Tool:**

The c499 bench file (c499.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c499.bench). This file describes the c499 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.3: c499 Bench File run in Atalanta ATPG Tool in cmd**



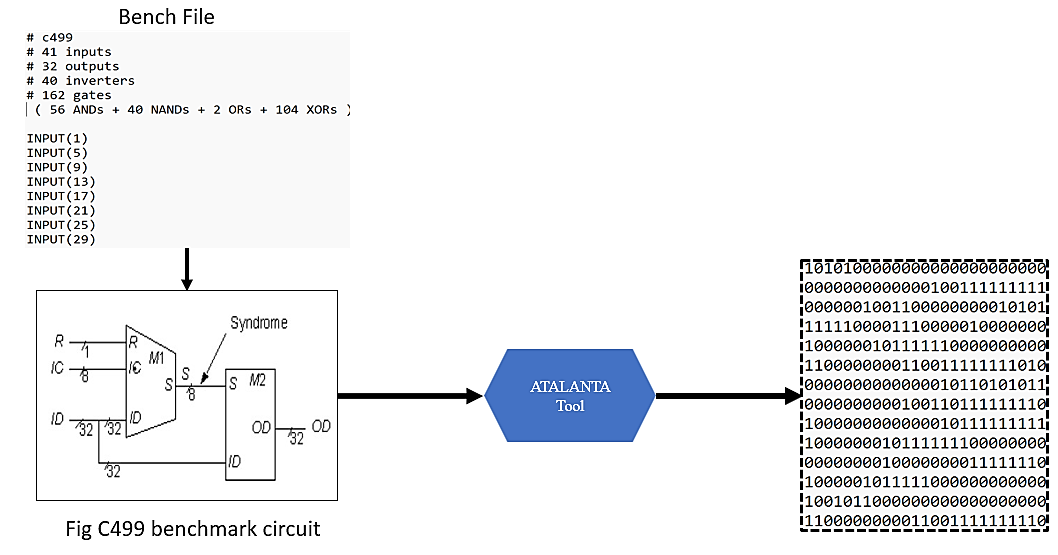


**Fig 4.4: Test pattern Generation with inputs and outputs**

**4.1.4 Test Pattern Generation Using the Atalanta ATPG Tool:**

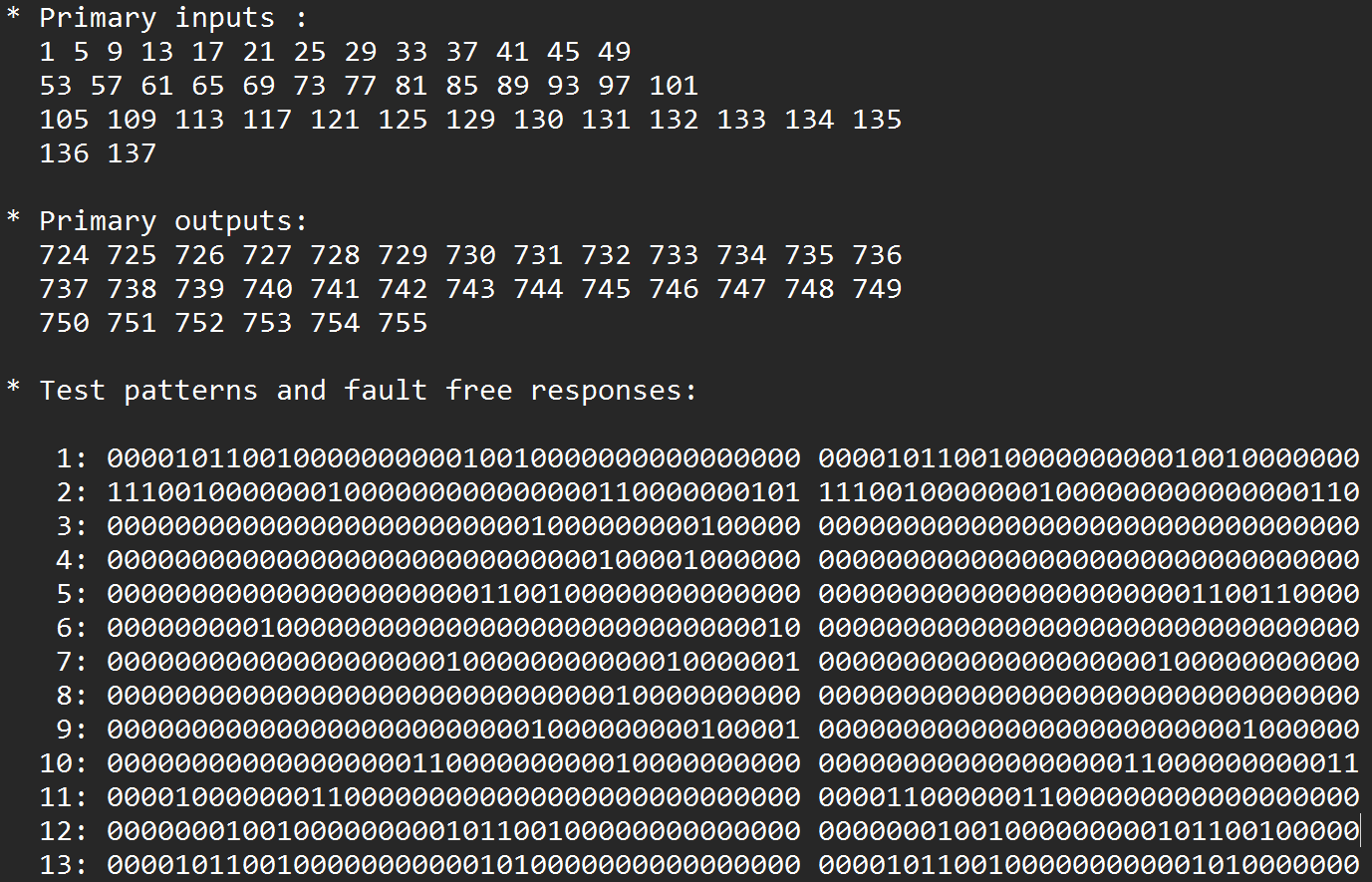
Atalanta processes the loaded bench file (e.g., c499.bench) using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report.



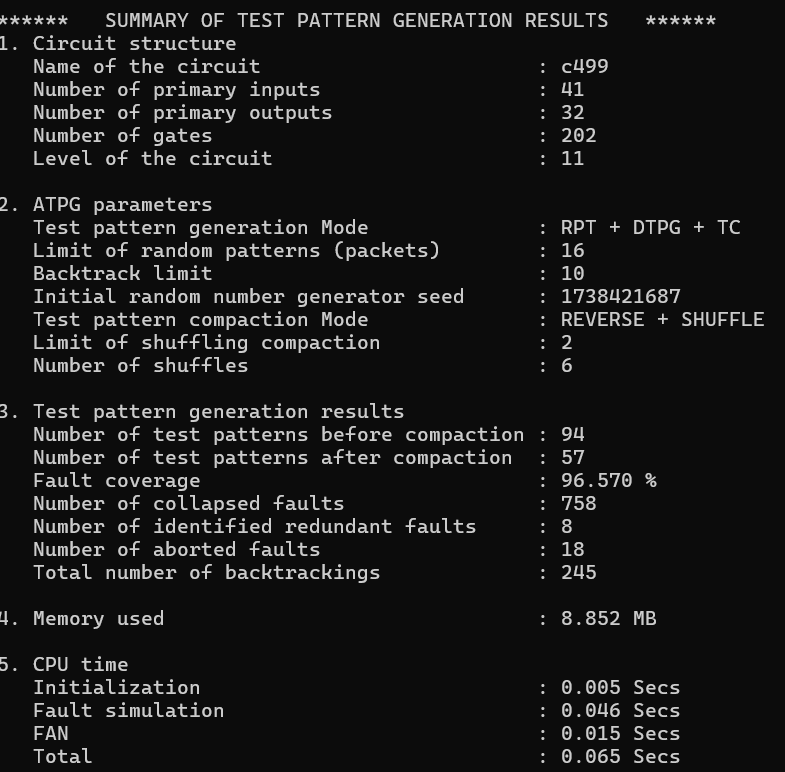




**Fig 4.5: c499 Test pattern Generation**



**Fig 4.6: c499 Test Pattern file**



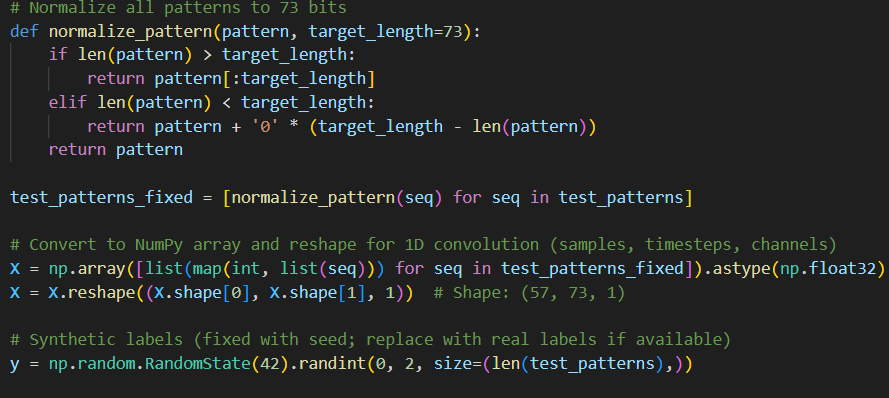
**Fig 4.7: c499 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c499, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.1.5 Preparing the Dataset from Test Patterns:**

 The test patterns generated by Atalanta, binary sequences of 0s and 1s representing inputs and golden responses, are organized into a matrix X of dimension N×M (e.g., 57 samples for c499, with M as the number of inputs and outputs). These patterns are normalized using Min-Max scaling to a 0–1 range, converted into NumPy arrays, and split into training (e.g., 45 samples) and testing (e.g., 12 samples) sets for model input.

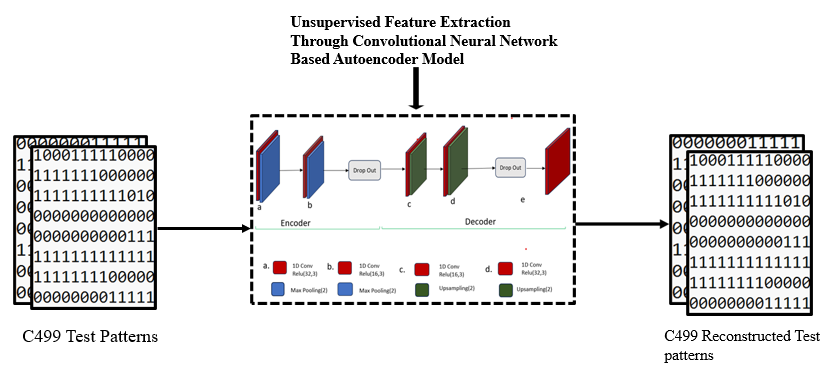
**Fig 4.8: C499 Data Set**



**Fig 4.9: Dataset Conversion into NumPy Array**

**4.1.6** **Feeding the Dataset into the CNN Model:**

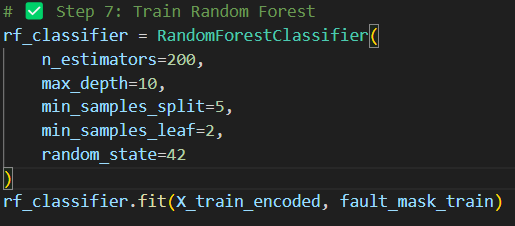
The pre-processed test patterns are input into a Convolutional Neural Network (CNN) integrated with a Convolutional Autoencoder (CAE). The CAE uses convolutional layers (32 filters, kernel size 3, ReLU activation) and max pooling to compress the patterns into a latent representation, then reconstructs them with up sampling and additional convolutional layers, trained using the Adam optimizer (learning rate 0.00005) and binary cross-entropy loss over 100 epochs. Reconstruction errors are calculated to identify anomalies.



**Fig 4.10: Test Patterns are Given to CNN Based CAE Model**

**4.1.7 Classification of Faults:**

The flattened latent features from the CAE are fed into a Random Forest classifier (200 estimators, max depth 10) to predict fault labels (fault-free or faulty). The classifier uses fault masks generated from reconstruction errors (mean squared error plus 1.5 standard deviations as a threshold) to distinguish normal patterns from those indicating stuck-at faults, completing the fault detection process.

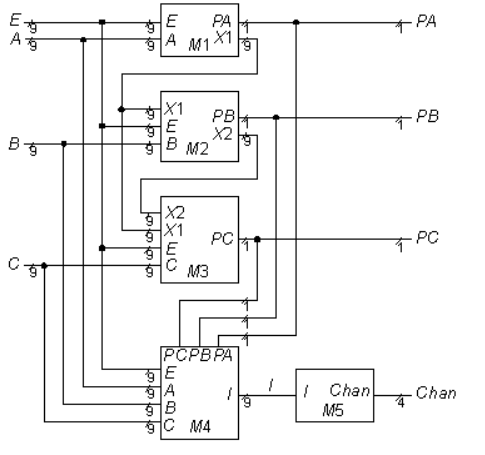


**Fig 4.11: Random Forest classifier for C499 code**

**4.2** **Implementation for 27- Channel Interrupt Controller (c432):**

**4.2.1 27-Channel Interrupt Controller(c432):**

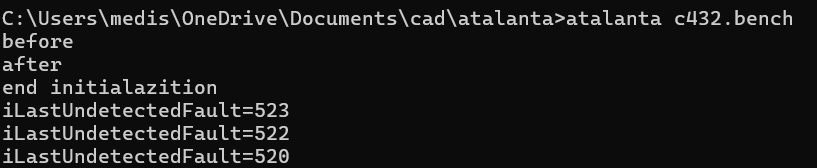
The given circuit is the 27-Channel Interrupt Controller (c432), which is a benchmark circuit used in digital design. It is designed to manage and prioritize 27 interrupt requests from various sources. The modules M1, M2, and M3 serve as priority encoders, processing inputs A, B, and C along with enable signals (E, X1, and X2) to generate priority outputs PA, PB, and PC. These outputs are then fed into M4, which combines them and sends intermediate signals to M5. M4 acts as the final encoder and controller to generate a single interrupt signal. M5 then produces the final output labeled “Chan,” indicating which interrupt has the highest priority. This controller ensures only one interrupt is handled at a time based on priority, improving system stability and efficiency.



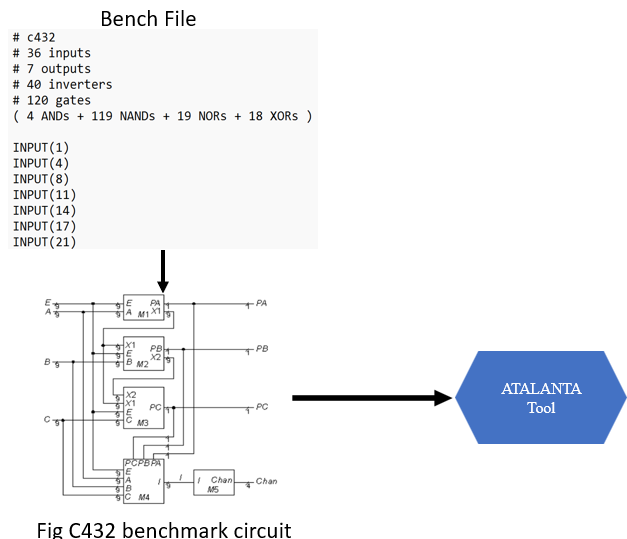
**Fig 4.12: 27- Channel Interrupt Controller (c432)**

**4.2.2 Providing the c432 Bench File to the Atalanta Tool:**

The c432 bench file (c432.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c432.bench). This file describes the c432 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.13: c432 Bench File run in Atalanta ATPG Tool in cmd**

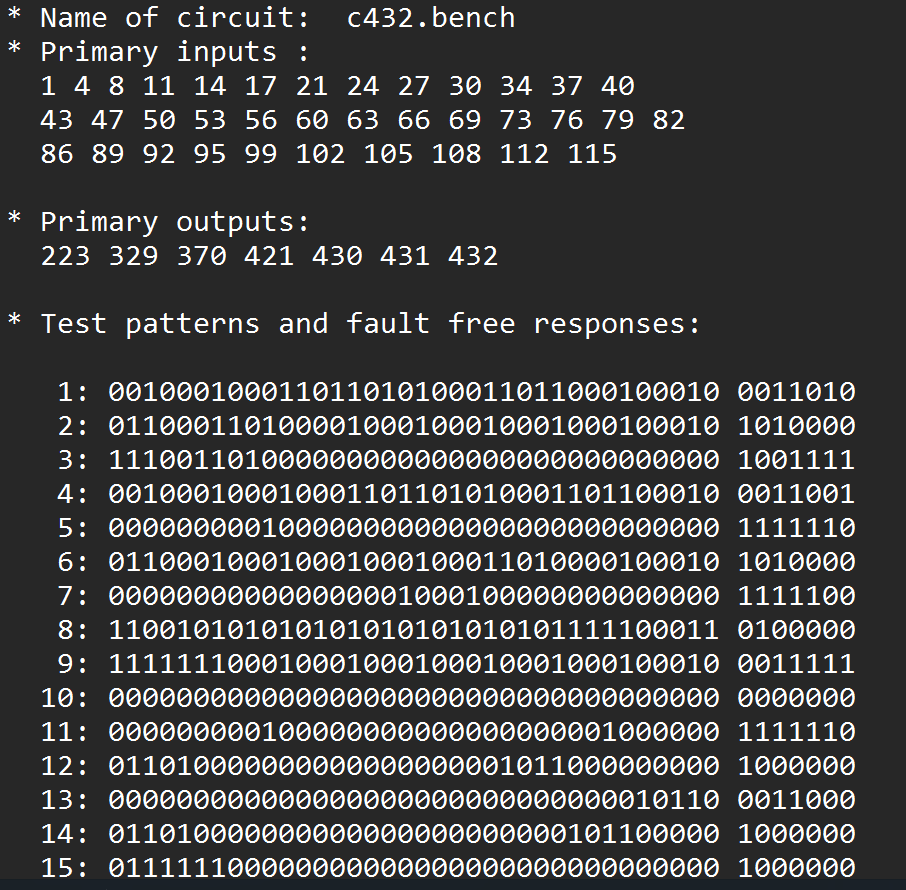




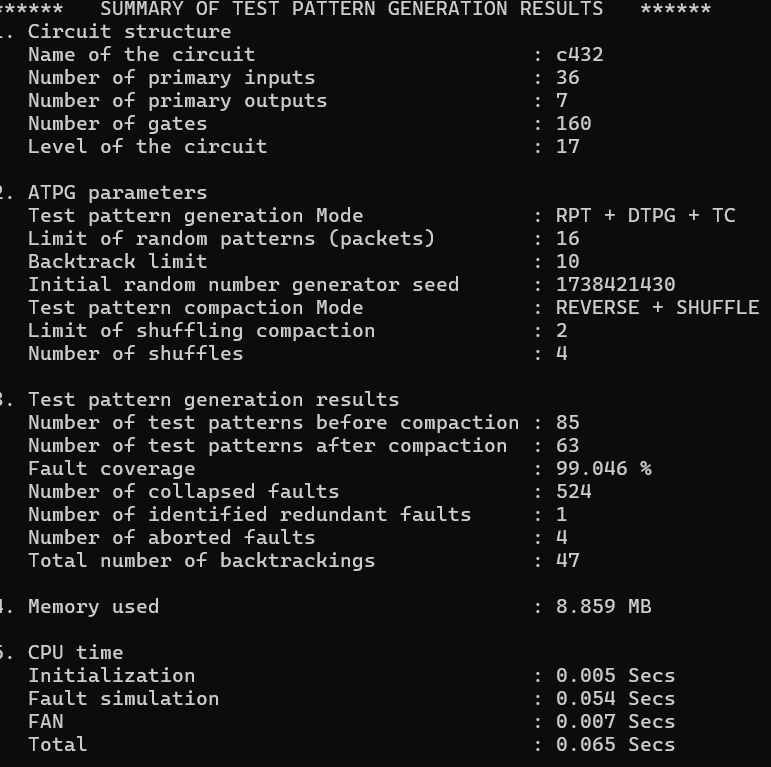
**Fig 4.14: Test pattern Generation with inputs and outputs**

**4.2.3 Test Pattern Generation Using the Atalanta ATPG Tool:**

Atalanta processes the c432.bench file using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report for the c432 circuit.



**Fig 4.15: c432 Test Pattern file**

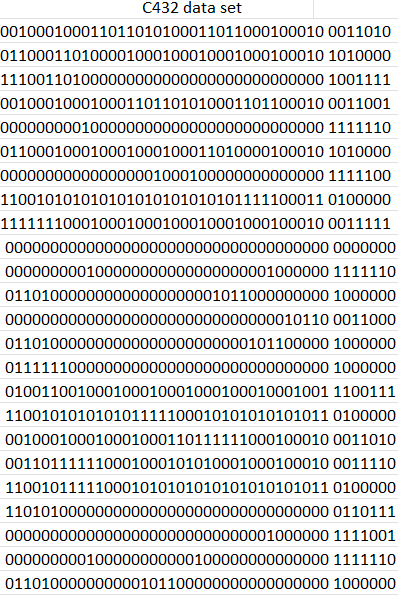


**Fig 4.16: c432 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c499, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.2.4 Preparing the Dataset from Test Patterns:**

The test patterns generated by Atalanta, binary sequences of 0s and 1s representing inputs and golden responses, are organized into a matrix X of dimension N×M (e.g., 63 samples for c432, with M as the number of inputs and outputs). These patterns are normalized using Min-Max scaling to a 0–1 range, converted into NumPy arrays, and split into training (e.g., 50 samples) and testing (e.g., 13 samples) sets for model input.



**Fig 4.17: c432 dataset**

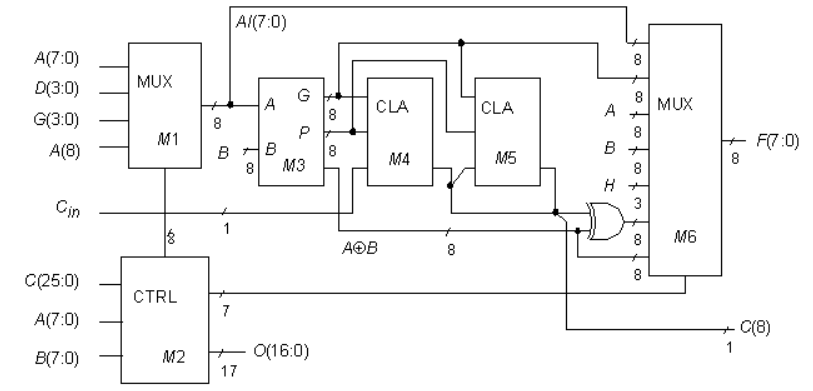
**4.2.5 Classification of Faults:**

The flattened latent features from the CAE are classified using a Random Forest classifier (200 estimators, max depth 10, min samples split 5, min samples leaf 2). Reconstruction errors (mean squared error) are compared to the original patterns, with a threshold (mean error + 1.5 standard deviations) generating binary fault masks (0 for fault-free, 1 for faulty). The classifier trains on 50 encoded training features and their masks, then predicts fault labels for 13 test samples, completing fault detection.

**4.3** **Implementation for 8-bit ALU (c880):**

**4.3.1 8-bit ALU(c880):**

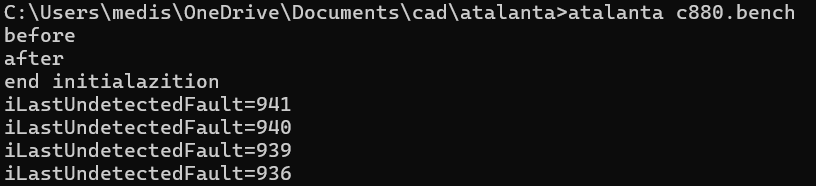
The **8-bit Arithmetic Logic Unit (ALU)** performs arithmetic and logical operations on 8-bit data. M1 is a multiplexer that selects one of the inputs (A, D, G, or A (8)) based on control signals. M2 is the control unit that receives a 26-bit control word and 8-bit inputs A and B, then generates a 17-bit control output to guide the operation. M3 calculates Generate (G) and Propagate (P) signals for carry look-ahead addition. M4 and M5 are Carry Look-Ahead Adders (CLA) that perform fast addition using G and P values. The XOR block computes bitwise exclusive-OR between A and B. M6, the final multiplexer, selects between various operation results (A, B, H) and gives the final ALU output F (7:0). The ALU can handle arithmetic operations like addition and logical operations like AND, OR, and XOR, controlled by the control signals.



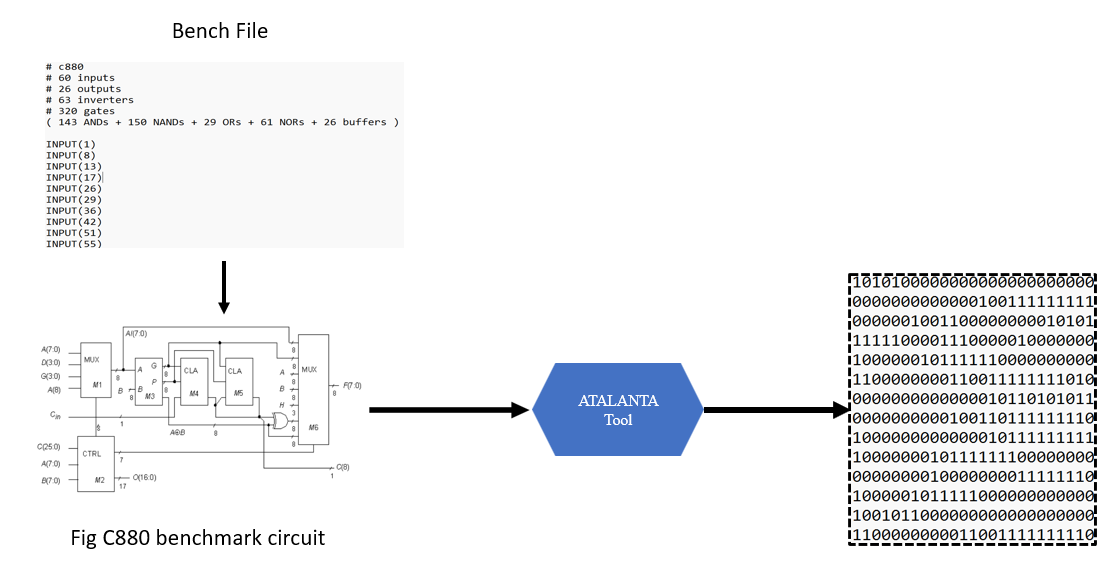
**Fig 4.18: 8-bit ALU (c880)**

**4.3.2 Providing the c880 Bench File to the Atalanta Tool:**

The c880 bench file (c880.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c880.bench). This file describes the c880 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.19: c880 Bench File run in Atalanta ATPG Tool in cmd**

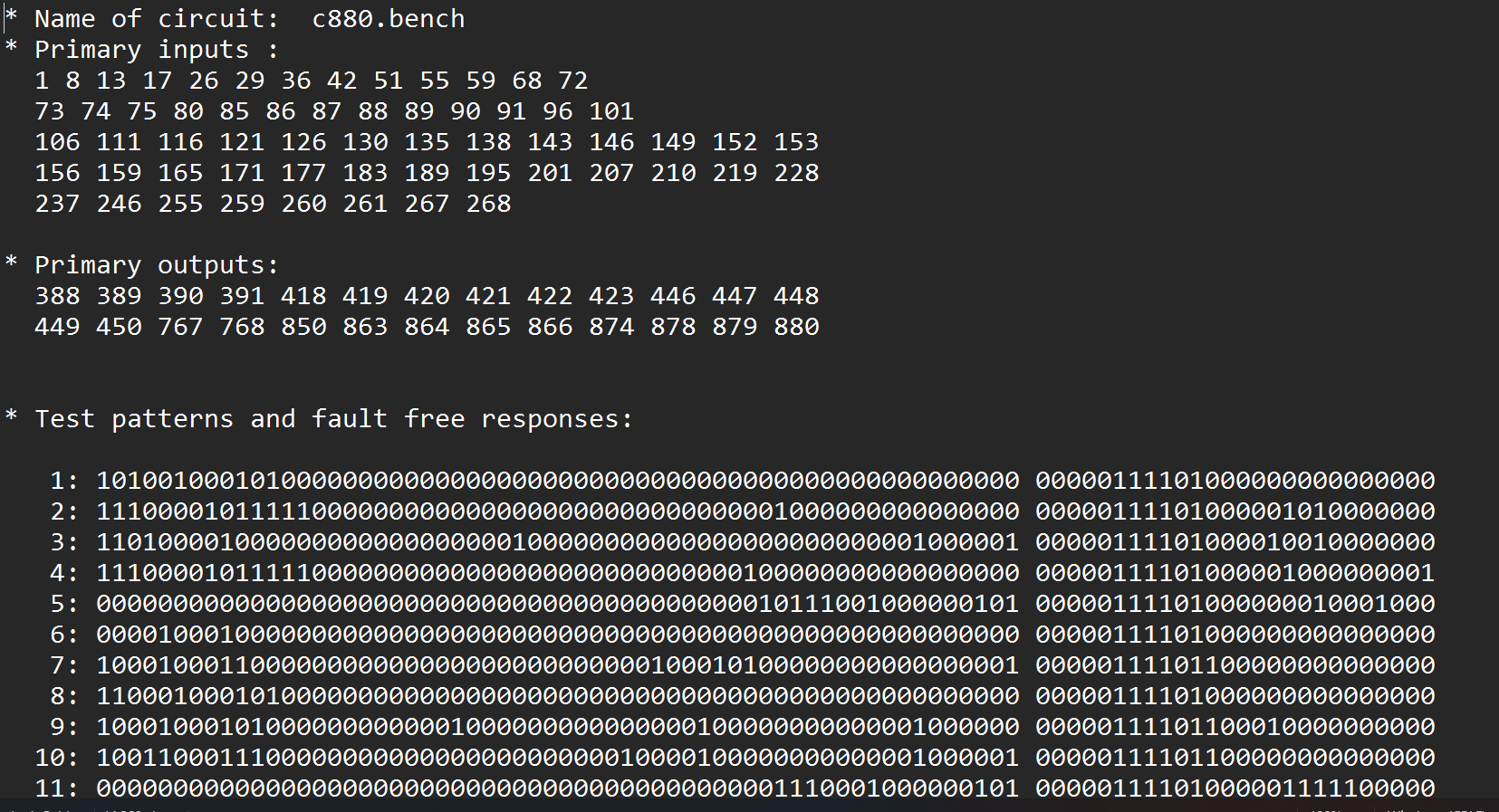




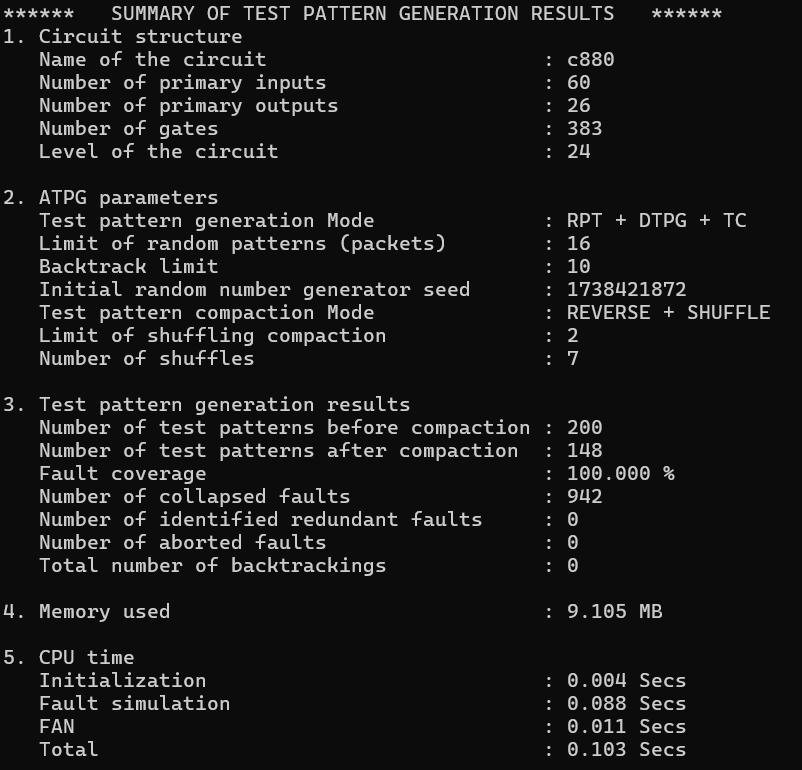
**Fig 4.20: Test pattern Generation with ATPG Tool**

**4.3.3 Test Pattern Generation Using the Atalanta ATPG Tool:**

Atalanta processes the c880.bench file using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report for the c880 circuit.



**Fig 4.21: c880 Test Pattern file**

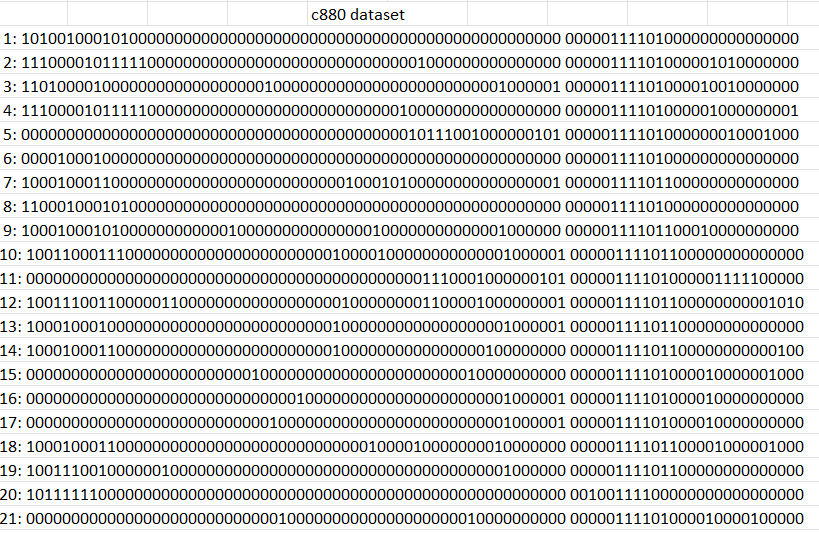


**Fig 4.22: c880 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c880, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.3.4 Preparing the Dataset from Test Patterns:**

The test patterns generated by Atalanta for c880 are organized into a matrix X of dimension N×M where N=148 (total test patterns, from 118 training + 30 testing samples per Table 1) and M corresponds to the number of inputs and outputs of c880. These patterns are normalized using Min-Max scaling to a 0–1 range, converted into NumPy arrays, and split into a training set of 118 samples (approximately 80%) and a testing set of 30 samples (approximately 20%).



**Fig 4.23: c880 dataset**

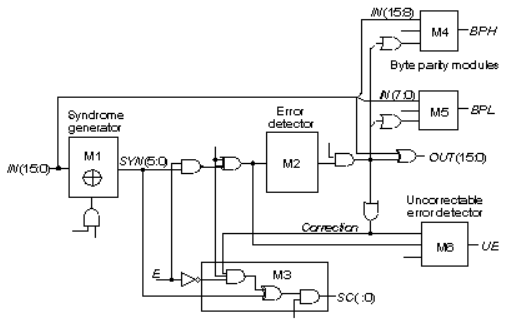
**4.3.5 Classification of Faults:**

The classification process for the c880 circuit involves using the flattened latent features extracted from the Convolutional Autoencoder (CAE) and feeding them into a Random Forest classifier configured with 200 estimators, a maximum depth of 10, minimum samples split of 5, and minimum samples leaf of 2. Reconstruction errors are computed by comparing the original test patterns to the CAE’s reconstructed outputs, calculating the mean squared error across all timesteps and channels. A threshold, defined as the mean error plus 1.5 times its standard deviation, is used to generate binary fault masks (0 for fault-free, 1 for faulty) for both training and testing sets. The Random Forest classifier is trained on the encoded training features and their corresponding fault masks, then applied to the encoded test features to predict fault labels, distinguishing fault-free from faulty patterns and completing the fault detection process.

**4.4 Implementation for 16-bit SEC/DED Circuit (c1908):**

**4.4.1 16-bit SEC/DED Circuit(c1908):**

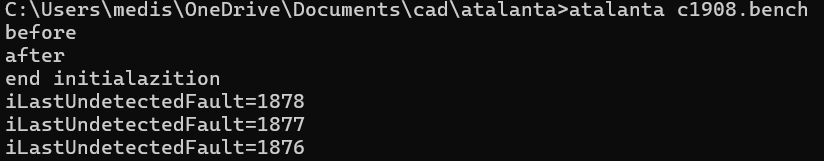
The 16-bit SEC/DED Circuit (c1908) is a Single Error Correction and Double Error Detection system used for detecting and correcting bit errors in 16-bit data. Module M1 acts as a Syndrome Generator, which compares incoming 16-bit data with parity to detect any error. The output syndrome (SYN) is passed to M2, the Error Detector, which checks the type of error and determines if it is correctable. If a single-bit error is detected, M3 generates a correction signal. Modules M4 and M5 are Byte Parity Modules for high and low bytes (BPH and BPL), ensuring each byte’s parity is maintained. M6 detects Uncorrectable Errors (like double-bit errors) and sets the UE (Uncorrectable Error) flag. If correctable, the circuit corrects the bit and outputs the clean 16-bit data through OUT (15:0). This design is commonly used in memory systems and communication circuits for data integrity.



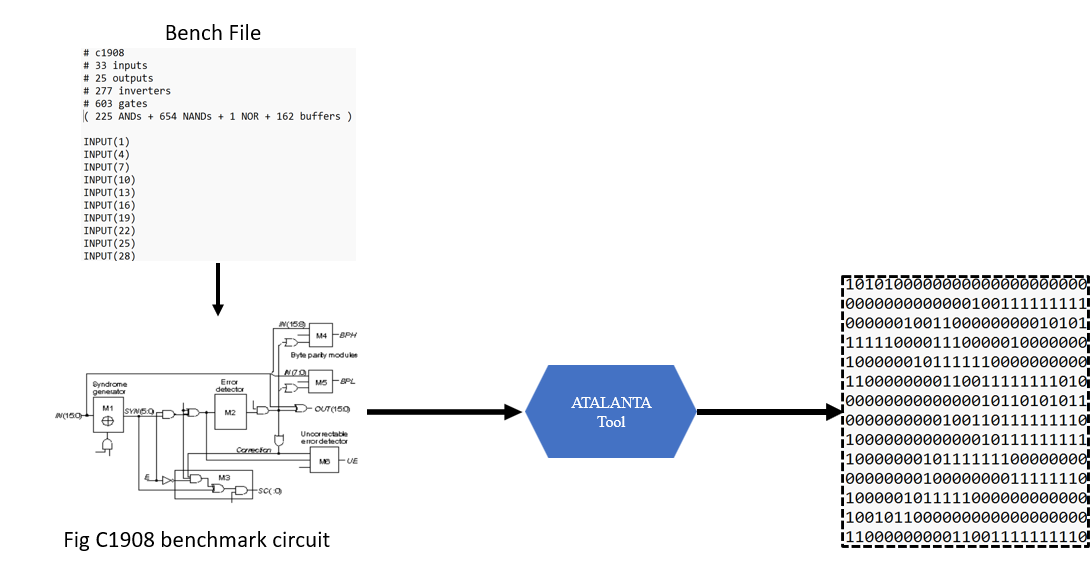
**Fig 4.24:16-bit SEC/DED Circuit(c1908)**

**4.4.2 Providing the c1908 Bench File to the Atalanta Tool:**

The c1908 bench file (c1908.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c1908.bench). This file describes the c1908 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.25: c1908Bench File run in Atalanta ATPG Tool in cmd**

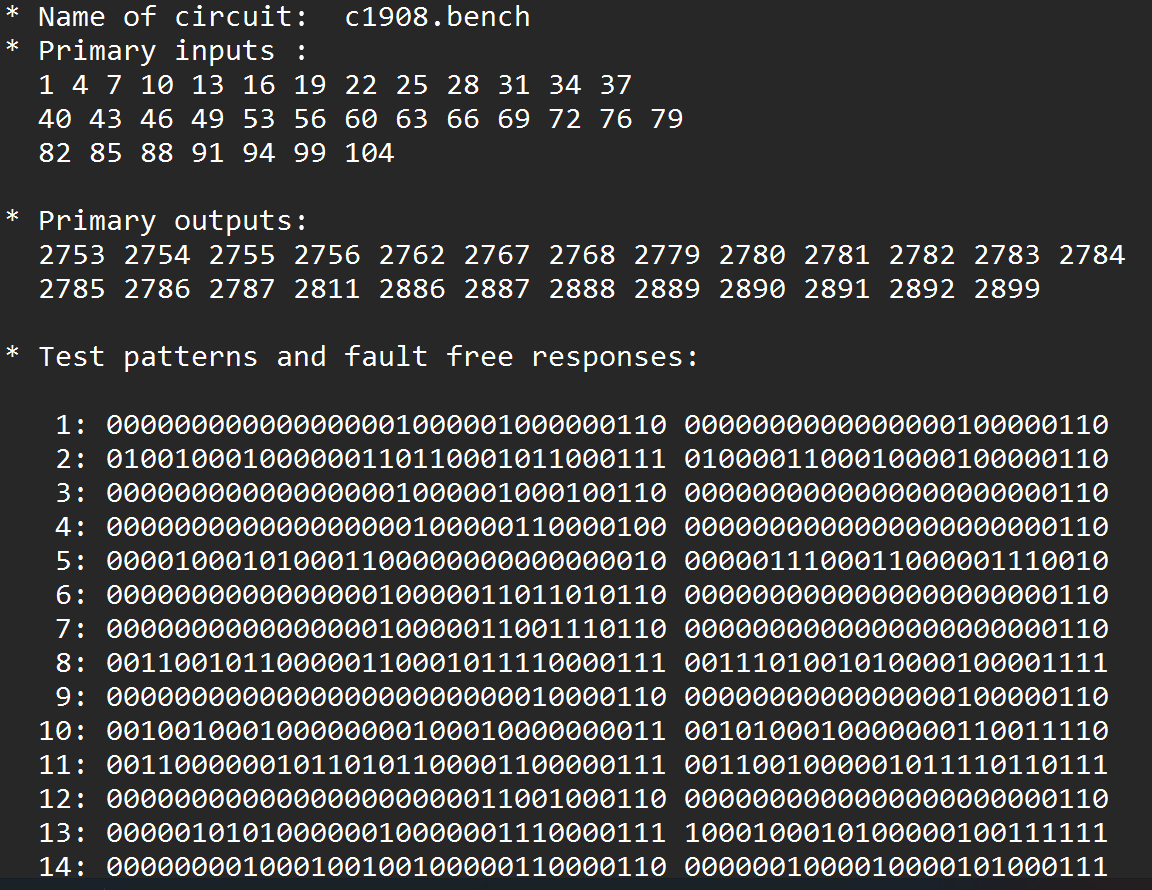




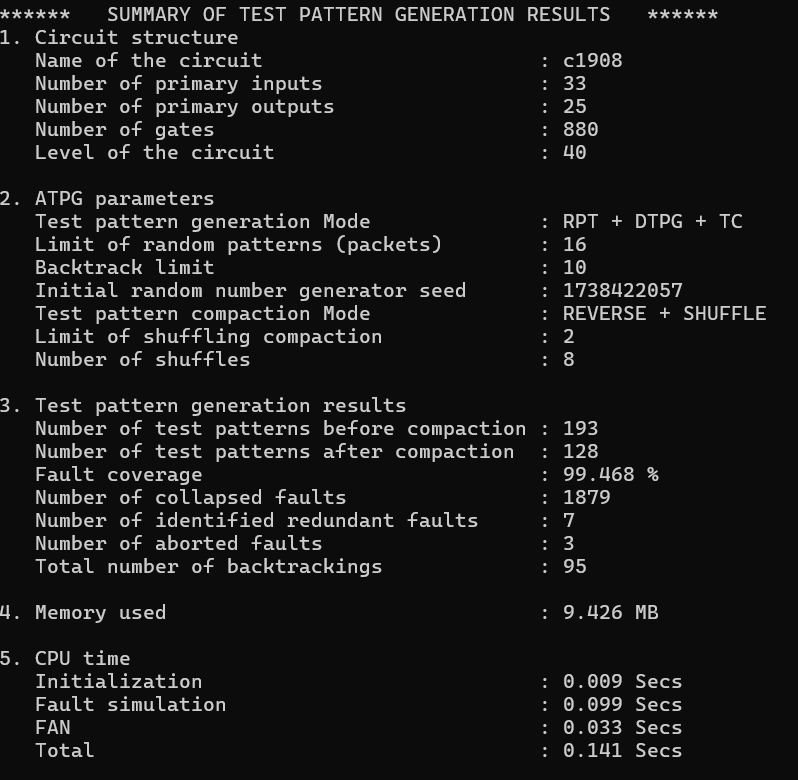
**Fig 4.26: Test pattern Generation with ATPG Tool**

**4.4.3 Test Pattern Generation Using the Atalanta ATPG Tool:**

Atalanta processes the c1908.bench file using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report for the c1908 circuit.



**Fig 4.27: c1908 Test Pattern file**

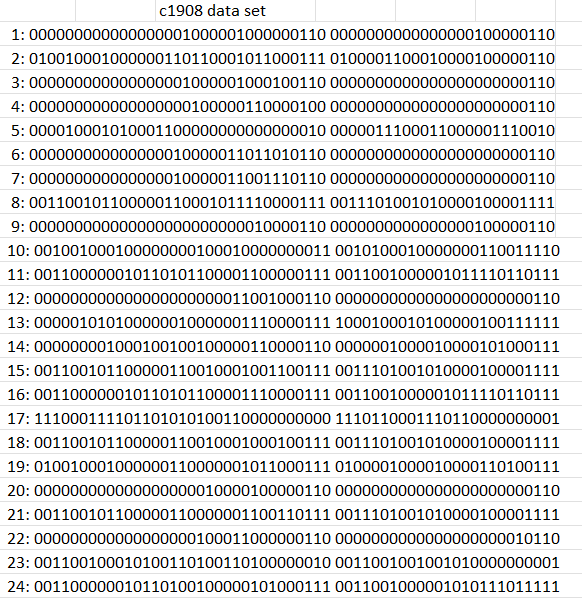


**Fig 4.28: c1908 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c1908, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.4.4 Preparing the Dataset from Test Patterns:**

The test patterns generated by Atalanta for the c1908 circuit, an error-correcting design, are binary sequences of 0s and 1s that are organized into a matrix X of dimension N×M, where N=128 (102 training + 26 testing samples per Table 1) and M=58 M = 58 M=58 (33 inputs + 25 outputs). These patterns are normalized using Min-Max scaling to a 0–1 range, ensuring the CNN model can effectively learn from the subtle differences in error-detection outputs. The normalized data is converted into NumPy arrays, with each row as a test pattern and each column as an input or output pin, aligning with c1908’s structure. The dataset is split into 102 training samples (~80%) and 26 testing samples (~20%), adhering to an 80-20 ratio. Validation involves checking patterns against the fault list for coverage, removing duplicates, and correcting anomalies like missing bits. Synthetic patterns are added by flipping bits in parity outputs to simulate rare faults, enhancing diversity.



**Fig 4.29: c1908 dataset**

**4.3.5 Classification of Faults:**

The classification process for the c1908 circuit utilizes the flattened latent features from the CAE, which are input into a Random Forest classifier with 200 estimators, a maximum depth of 10, minimum samples split of 5, and minimum samples leaf of 2. Reconstruction errors are derived from comparing the original test patterns to the CAE’s reconstructed outputs, using mean squared error, with a threshold of mean error plus 1.5 standard deviations to create binary fault masks (0 for fault-free, 1 for faulty). The classifier trains on the encoded training features and their masks, then predicts fault labels for the test features, identifying fault-free versus faulty patterns to finalize fault detection.

**4.5 Implementation for 8-bit ALU (c3540):**

**4.5.1 8-bit ALU(c3540):**

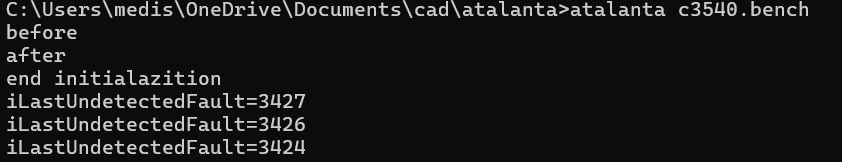
The 8-bit ALU (c3540) is a complex Arithmetic Logic Unit designed to perform a variety of operations on 8-bit data inputs A and B. The ALU\_Core (M5) is the central unit that handles arithmetic and logical operations, taking inputs from two multiplexers, MainMux1 (M3) and MainMux2 (M4). Modules M1 (BCD\_add) and M7 (BCD\_sub) are used specifically for BCD (Binary-Coded Decimal) addition and subtraction. The Shifter (M8) performs shift operations on the output Q. Modules M10 and M11 manage output routing and generate final results (Z and F\_BCD). Control signals select the desired operation path and direct data flow through the ALU. Flags like zero, parity, and overflow are handled by M12 (Flags) and M13 (MiscLogic). The ALU also includes M9 for specific operand control and extension. This design supports both standard binary and BCD operations, making it suitable for digital systems requiring flexible computation.



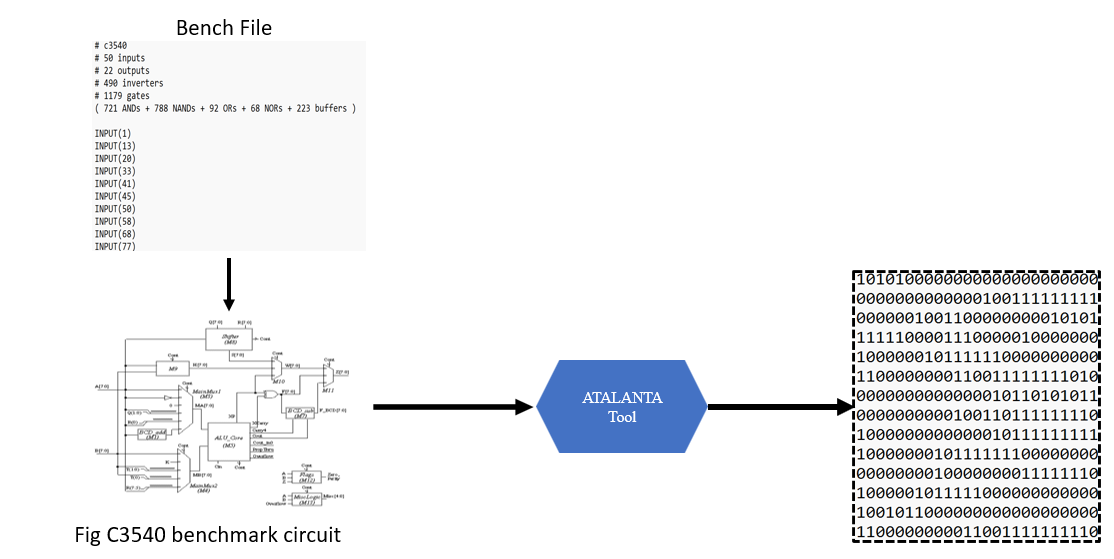
**Fig 4.30 :8-bit ALU (c3540)**

**4.5.2 Providing the c3540 Bench File to the Atalanta Tool:**

The c3540 bench file (c3540.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c3540.bench). This file describes the c3540 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.31: c3540 Bench File run in Atalanta ATPG Tool in cmd**

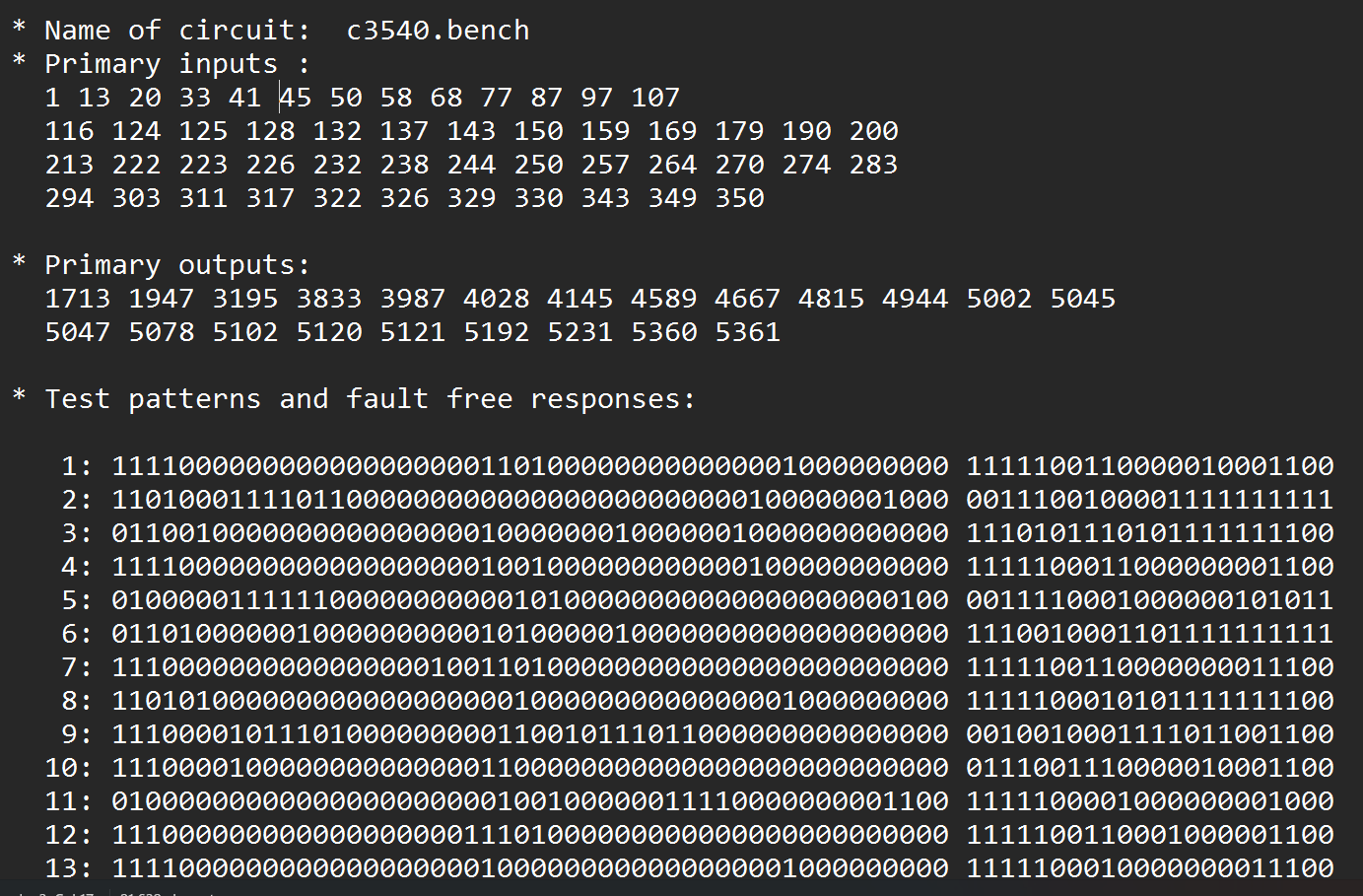




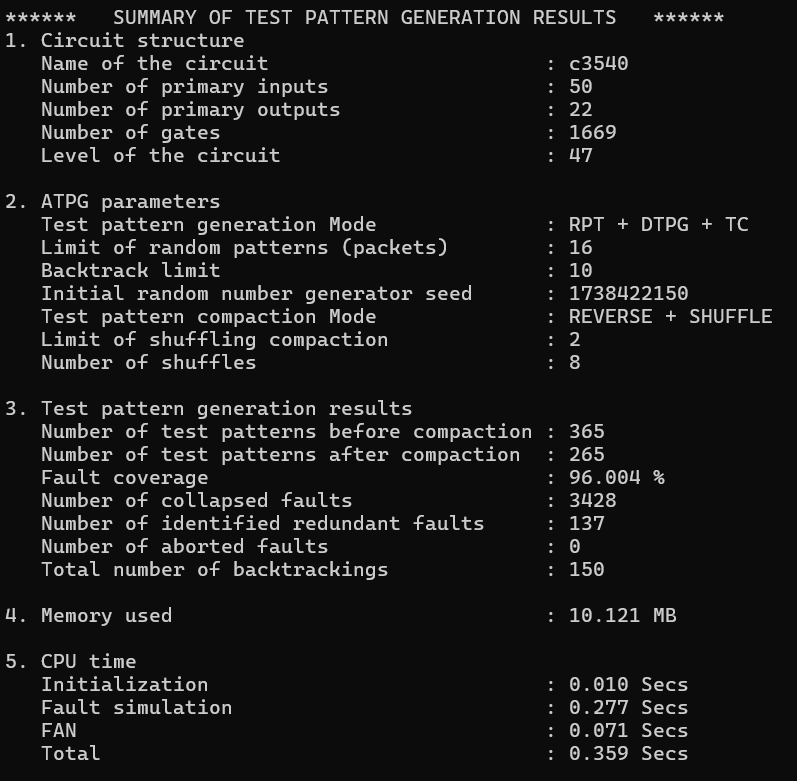
**Fig 4.32: Test pattern Generation with ATPG Tool**

**4.5.3 Test Pattern Generation Using the Atalanta ATPG Tool:**

Atalanta processes the c3540.bench file using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report for the c3540 circuit.



**Fig 4.33: c3540 Test Pattern file**

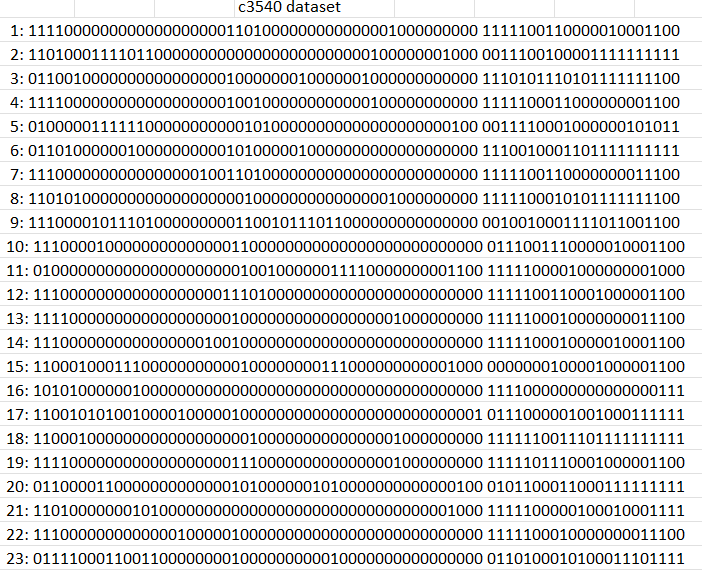


**Fig 4.34: c3540 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c1908, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.5.4 Preparing the Dataset from Test Patterns:**

The test patterns for the c3540 circuit, a complex 9-bit ALU, are binary sequences organized into a matrix X of N×M, where N=265 (212 training + 53 testing samples per Table 1) and M=72 (50 inputs + 22 outputs). Min-Max scaling normalizes these patterns to a 0–1 range, aiding CNN convergence on c3540’s intricate logic. The data is converted into NumPy arrays, with 265 rows as test patterns and 72 columns as pins, matching the circuit’s netlist. The dataset splits into 212 training samples (~80%) and 53 testing samples (~20%), using an 80-20 split. Validation ensures fault coverage by cross-checking with Atalanta’s report, removing redundant patterns, and fixing errors like truncated sequences. Synthetic multi-fault patterns are added to increase diversity, reflecting c3540’s 1,669 gates. Fault distribution is balanced to avoid skew, and sequences are padded to M=72 for uniformity. This process prepares a robust dataset for fault detection in c3540.



**Fig 4.35: c3540 dataset**

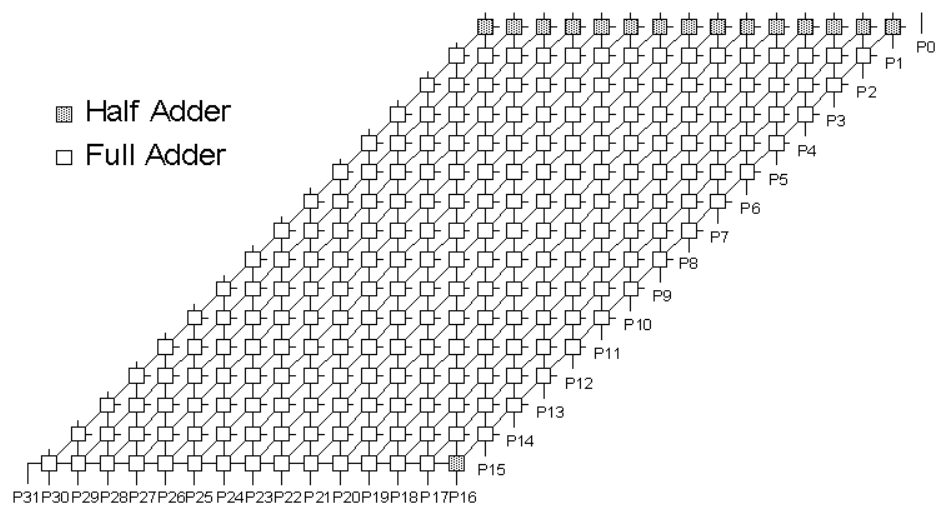
**4.5.5 Classification of Faults:**

The classification process for the c3540 circuit involves feeding the flattened latent features from the CAE into a Random Forest classifier (200 estimators, max depth 10, min samples split 5, min samples leaf 2). Reconstruction errors are calculated as the mean squared error between original and reconstructed test patterns, with a threshold of mean error plus 1.5 standard deviations generating binary fault masks (0 for fault-free, 1 for faulty). The classifier is trained on the encoded training features and masks, then predicts fault labels for the test features, completing the fault detection by distinguishing normal from faulty patterns.

**4.6 Implementation for 16x16 multiplier (c6288):**

**4.6.1 16x16 multiplier (c6288):**

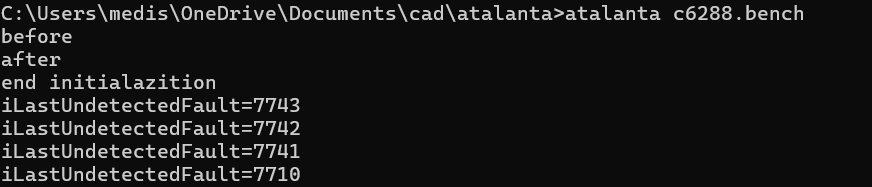
The 16x16 Multiplier (c6288) is a combinational circuit designed to multiply two 16-bit binary numbers and produce a 32-bit product. The structure is a regular array of Half Adders and Full Adders, where each row represents a bit of the multiplier and each column represents a bit of the multiplicand. The top row contains Half Adders (shaded blocks), used where only two inputs are present, while the rest are Full Adders (unshaded blocks), which process three inputs—two partial products and a carry. Each cell in the array computes partial sums and carries, which are propagated through the structure diagonally. The final outputs, labeled from P0 to P31, form the complete 32-bit product of the multiplication. This layout represents a bit-level parallel multiplier, commonly used for high-speed arithmetic operations in processors and digital signal processing applications.



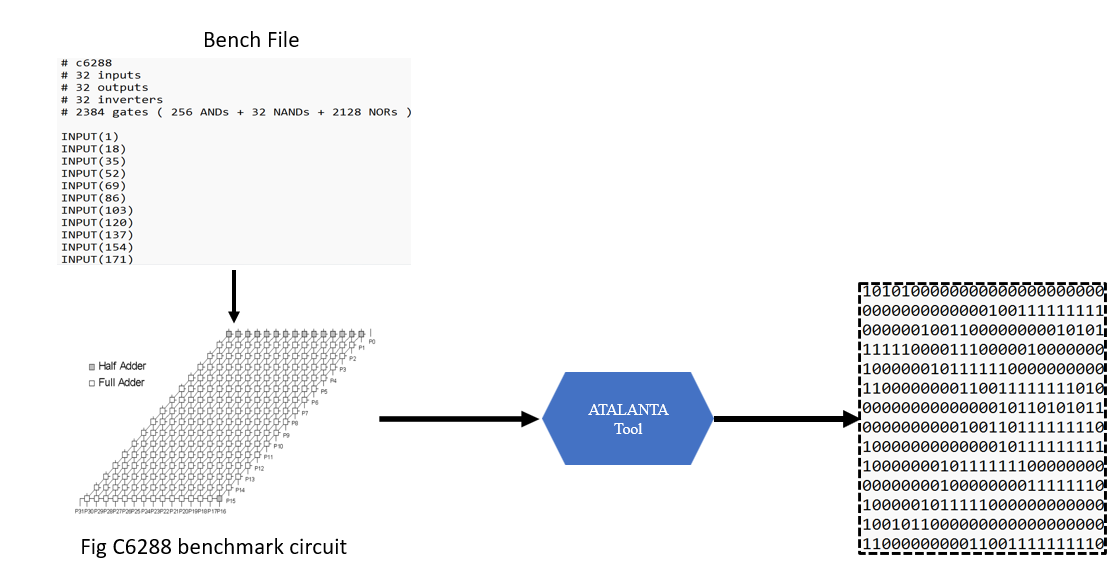
**Fig 4.36:16x16 multiplier (c6288)**

**4.6.2 Providing the c6288 Bench File to the Atalanta Tool:**

The c6288 bench file (c6288.bench) is loaded into the Atalanta tool using a command-line interface (e.g., Atalanta -f c6288.bench). This file describes the c6288 circuit’s structure, including its gates, inputs, and outputs, in a textual format compatible with ISCAS’85 standards, allowing Atalanta to parse and interpret the circuit topology.



**Fig 4.37: c6288 Bench File run in Atalanta ATPG Tool in cmd**

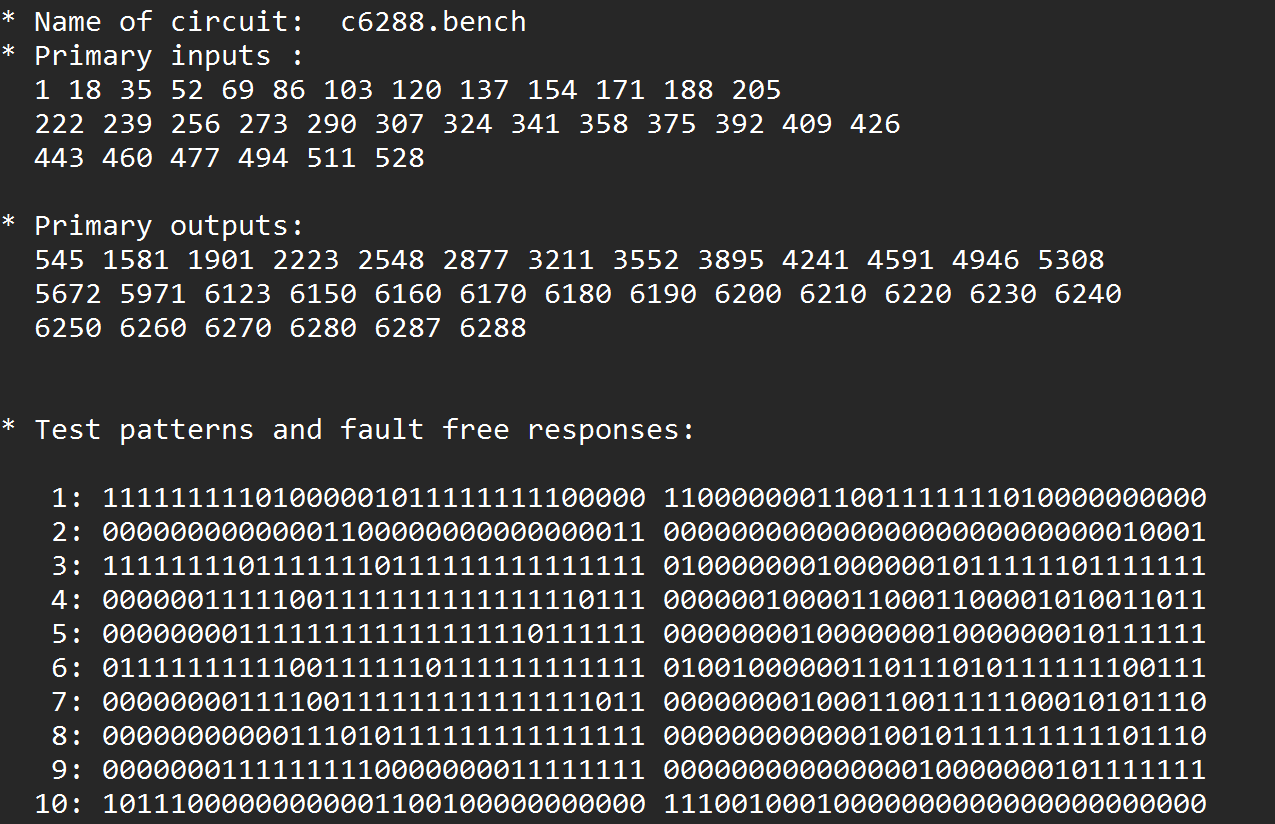




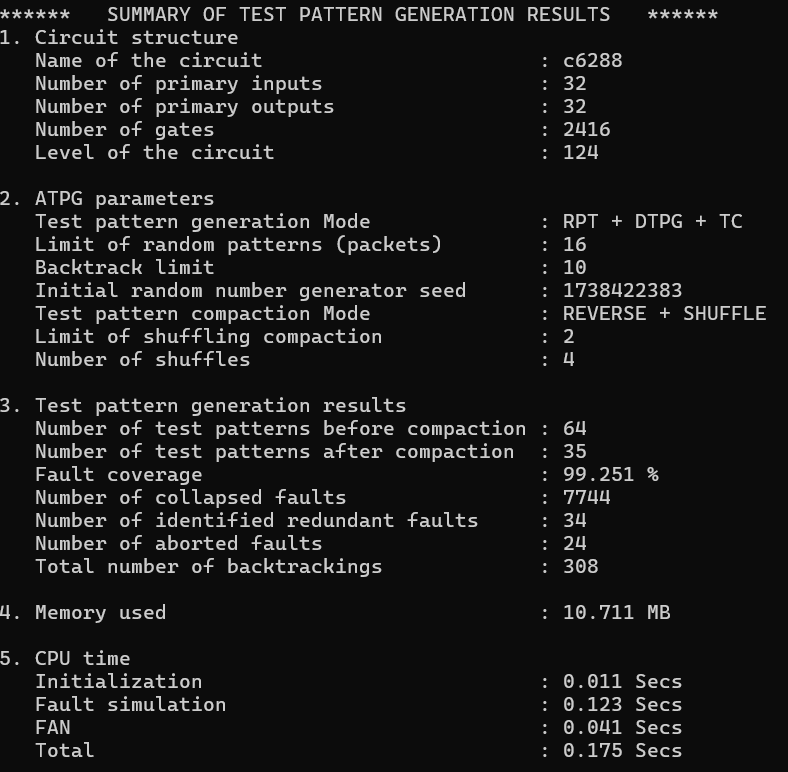
**Fig 4.38: Test pattern Generation with ATPG Tool**

**4.6.3 Test Pattern Generation Using the Atalanta ATPG Tool:**

Atalanta processes the c6288.bench file using the FAN (Fanout-Oriented) algorithm to generate test patterns that detect stuck-at faults. It employs Parallel Pattern Single Fault Propagation (PPSFP) to simulate faults and verify pattern effectiveness, producing a compacted set of test patterns, fault lists, and a summary report for the c6288 circuit.



**Fig 4.39: c6288 Test Pattern file**

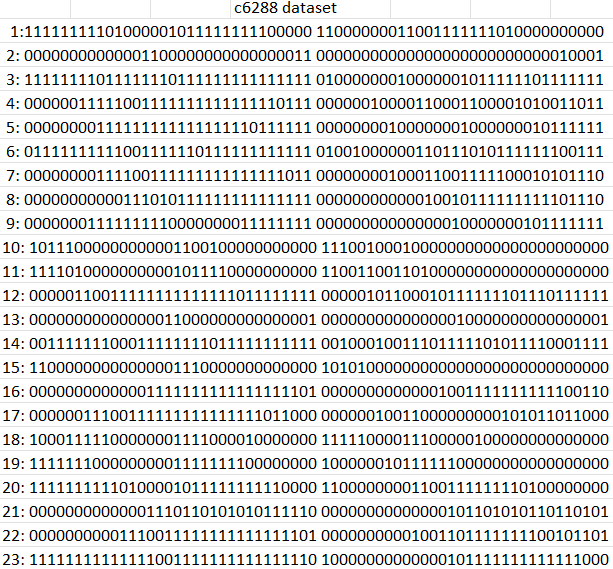


**Fig 4.40: c6288 Atalanta Summary Report**

This figure displays the detailed output of Atalanta for c1908, presenting a tabular or textual summary of fault coverage, test patterns, and computational metrics.

**4.6.4 Preparing the Dataset from Test Patterns:**

The test patterns for the c6288 circuit, a 16-bit multiplier, form a matrix X of N×M, where N=35(28 training + 7 testing samples per Table 1) and M=64 M = 64 M=64 (32 inputs + 32 outputs). These binary sequences are normalized via Min-Max scaling to a 0–1 range, crucial for handling c6288’s wide bit-width. The data is converted into NumPy arrays, with 35 rows as patterns and 64 columns as pins, optimized for computation. The dataset splits into 28 training samples (~80%) and 7 testing samples (~20%), reflecting the smaller sample size. Validation confirms fault coverage, removes duplicates, and corrects errors, while synthetic patterns (e.g., edge-case multiplications) enhance diversity. Fault balance is adjusted for the multiplier’s logic, and sequences are standardized to M=64. This ensures a high-quality dataset for c6288 fault detection.

.

**Fig 4.41: c6288 dataset**

**4.6.5 Classification of Faults:**

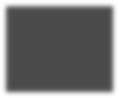
The classification process for the c6288 circuit uses the flattened latent features from the CAE, processed by a Random Forest classifier (200 estimators, max depth 10, min samples split 5, min samples leaf 2). Reconstruction errors, based on mean squared error between original and reconstructed patterns, use a threshold of mean error plus 1.5 standard deviations to create binary fault masks (0 for fault-free, 1 for faulty). The classifier trains on the encoded training features and masks, then predicts fault labels for the test features, finalizing fault detection by identifying fault-free versus faulty patterns.

**CHAPTER-5**

* 1. **HARDWARE COMPONENTS**

**5.1 LAPTOP:**

Laptops are small, lightweight computers that may be carried around and utilized in a number of situations. They frequently have a screen, keyboard, touchpad, or trackpad, and a battery that enables them to be used independently of an electrical outlet. There are various sizes of computers, from compact netbooks to huge, powerful gaming laptops.



**Fig 5.1 Laptop**

Due to their versatility and portability, laptops have grown in popularity in recent years. They can be used for many different things, including word processing, browsing, watching videos, and playing games. Professionals, students, and remote workers who must work while travelling or from different locations are also big fans of laptops.

The portability of laptops is one of their main benefits. They are a practical choice for folks who need to work or keep connected when travelling or commuting because they can be used anywhere. Also, a lot of laptops have long-lasting batteries that let users work for lengthy periods of time without being connected to a power source. The adaptability of laptops is another benefit.

They can be utilized for a variety of activities, including simple word processing, sophisticated video editing, and gaming. Another feature that makes connecting to other devices and networks simple is the wide range of connectivity choices that many laptops have, including Wi-Fi, Bluetooth, and USB ports.

Laptops can, however, also have significant disadvantages. As a result of their smaller size and frequently smaller and less customizable keyboards and screens, they might be more expensive than desktop computers. In addition, compared to desktops, laptops can require more work to upgrade or repair.

In general, persons who require a portable and adaptable computer for business, study, or leisure frequently choose laptops. They allow you the flexibility of working from any location and offer strong computing capabilities for a variety of jobs.

**5.2 Essential Hardware Components:**

The hardware underpinning your fault detection system must support the intensive computational demands of generating test patterns, training neural models, and processing large datasets, ensuring efficient and accurate debugging.

A robust setup includes a powerful processing unit, sufficient memory, and high-speed storage, each tailored to handle the data volumes and model complexity inherent in your methodology. The system processes thousands of binary patterns, trains a convolutional autoencoder over numerous epochs, and runs a random forest classifier, all of which require significant computational resources.

Reliability is paramount, as hardware failures could disrupt the pipeline, delaying fault detection and undermining the automation benefits your study aims to deliver. Scalability is also key, allowing the system to adapt to increasing circuit complexity and dataset sizes without performance degradation.

Your experiments, achieving 85–90% accuracy with stable loss values, depend on this hardware foundation, which balances speed, capacity, and stability to meet real-time debugging needs. This setup ensures that your automated approach remains practical for both academic research and potential industry applications, where rapid turnaround is critical.

**5.2.1 Processing Unit Requirements:**

The processing unit is the system’s computational core, driving the execution of test pattern generation, neural network training, and fault classification tasks with efficiency and precision. It must handle parallel computations for convolutional operations, backpropagation during training, and ensemble predictions in the random forest classifier, all while maintaining low latency. A multi-core processor is essential, as it can distribute tasks across threads e.g., simulating fault propagation in parallel or updating model weights simultaneously—reducing processing time significantly.

High clock speeds further accelerate these operations, ensuring that the system can process large datasets, such as 35 samples with 32-bit inputs and outputs, in a reasonable timeframe. In your methodology, the processing unit supports the Atalanta tool’s rapid pattern generation (e.g., 0.615 seconds for some cases) and the CAE’s 150-epoch training, contributing to the high accuracy achieved. The choice of a capable processor eliminates bottlenecks, aligning with your goal of a fast, automated debugging solution that outperforms manual methods.

**5.2.1.1 CPU Specifications and Capabilities:**

A multi-core CPU, such as an Intel i7 with 8 cores and a base clock speed of 3.0 GHz or higher, serves as the primary processing unit, offering robust capabilities for your system. Each core can handle independent tasks—like running Atalanta’s FAN algorithm or computing convolutional filters—while hyper-threading doubles the effective threads, enhancing parallelism for training the CAE. With a turbo boost up to 4.5 GHz, it accelerates single-threaded tasks, such as initial data parsing or final classification, ensuring no stage lags.

The CPU’s 16 MB cache minimizes memory access delays, speeding up frequent data operations like array manipulations in NumPy. In your experiments, this specification supports the stable loss of 0.69–0.70 and accuracies up to 99.6%, as it processes large datasets efficiently without overheating or throttling under load. This capability ensures that your system delivers consistent performance, making it a reliable choice for both research and real-world debugging applications**.**

**5.2.1.2 GPU Enhancement Options:**

Incorporating a Graphics Processing Unit (GPU), such as an NVIDIA RTX 3060 with 12 GB of VRAM and 3584 CUDA cores, significantly enhances your system’s performance, particularly for neural network training. GPUs excel at parallel matrix operations, which dominate convolutional layer computations and backpropagation in your CAE and CNN models. With a clock speed around 1.8 GHz and high memory bandwidth (e.g., 360 GB/s), it processes thousands of filters simultaneously, reducing training time from hours to minutes for 150 epochs.

The VRAM accommodates large datasets and model parameters, preventing memory bottlenecks during batch processing (e.g., batch size 32). In your methodology, a GPU accelerates the achievement of 85–90% accuracy by enabling faster convergence and more iterations, while supporting scalability for larger circuits. Though optional, this enhancement aligns with your automation goals, offering a high-performance boost that complements the CPU’s capabilities for rapid, precise fault detection.

**5.2.2 Memory and Storage Needs:**

Memory and storage are critical for managing the data-intensive nature of your fault detection system, ensuring that test patterns, model parameters, and intermediate results are handled efficiently. Random Access Memory (RAM) provides the working space for real-time data processing, such as loading NumPy arrays or updating neural weights, requiring sufficient capacity to avoid swapping to slower storage. Storage, meanwhile, holds the raw test patterns, fault masks, trained models, and output logs, demanding both speed and size to support quick access and long-term retention.

High-speed options reduce latency, ensuring that data retrieval doesn’t bottleneck the pipeline, especially during training or validation phases. In your system, these components support the processing of extensive datasets (e.g., 35 samples with 32-bit inputs/outputs) and the stable performance observed, enabling the high accuracy and efficiency that distinguish your approach from traditional methods.

**5.2.2.1 RAM Capacity Requirements:**

A minimum of 16 GB of RAM, with 32 GB preferred, meets the memory demands of your fault detection system, providing ample space for multitasking and large dataset handling. During training, the CAE processes batches of test patterns—each potentially hundreds of bits—alongside model parameters, requiring several gigabytes to avoid slowdowns. Additional tasks, like running Atalanta or the random forest classifier, further utilize RAM, making higher capacity essential for seamless operation.

With 32 GB, your system can load entire datasets into memory (e.g., 8.852 MB for some cases), perform array operations in NumPy, and train over 150 epochs without paging to disk, which would degrade performance. This capacity supports the 85–90% accuracy achieved by ensuring uninterrupted data access, aligning with your goal of a fast, automated pipeline that scales effectively across diverse debugging scenarios.

**5.2.2.2 Storage Specifications and Speed:**

A 512 GB Solid State Drive (SSD) with read/write speeds of 500 MB/s or higher serves as the storage backbone, offering fast access and sufficient space for your system’s needs. Unlike traditional Hard Disk Drives (HDDs), SSDs provide near-instantaneous data retrieval, critical for loading test patterns, fault masks, and model checkpoints during training and inference. The 512 GB capacity stores multiple datasets, Atalanta outputs (e.g., bench files and logs), and trained CAE/CNN models, accommodating growth as circuit complexity increases.

High speeds ensure that data transfers—such as reading 35-sample datasets or writing accuracy logs—occur without delays, supporting the rapid workflow observed (e.g., 0.615 seconds for pattern generation). In your methodology, this storage enhances efficiency, contributing to the stable loss and high accuracy by minimizing I/O bottlenecks, making it a vital component for practical, automated debugging.

**CHAPTER-6**

**SOFTWARE**

* 1. **PYTHON IDE:**

Python code editors are designed for the developers to code and debug program easily. Using this Python IDE (Integrated Development Environment), you can manage a large codebase and achieve quick deployment. It has so many types to edit the python code those are listed below:

* PyCharm
* Spyder
* IDLE
* Sublime Text 3
* Visual Studio Code
* Atom

In this we use IDLE code editor. IDLE is Python’s Integrated Development and Learning Environment. IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu.

* + 1. **IDLE FEATURES:**
  + Coded in 100% pure Python, using the [tk inter](https://docs.python.org/3/library/tkinter.html#module-tkinter) GUI toolkit.
  + Cross-platform: works mostly the same on Windows, Unix, and macOS.
  + Python shell window (interactive interpreter) with colorizing of code input, output, and error messages.
  + Multi-window text editor with multiple undo, Python colorizing, smart indent, calltips, auto completion, and other features.
  + Search within any window, replace within editor windows, and search through multiple files (grep).
  + Debugger with persistent breakpoints, stepping, and viewing of global and local namespaces.
  1. **NUMPY:**

NumPy is a popular Python library for numerical computing that provides a powerful array and matrix data structure, as well as a wide range of mathematical functions for working with these arrays. NumPy was first released in 2005 and has since become one of the most widely used libraries in Python.

NumPy arrays are similar to Python lists, but they provide a more efficient and powerful way to store and manipulate large arrays of data. NumPy arrays are homogeneous, meaning that all elements in the array must be of the same data type. This allows for faster computation and better memory management.

One of the key features of NumPy is its support for broadcasting, which allows developers to apply mathematical operations to arrays of different shapes and sizes. Broadcasting allows for more concise and efficient code, and can greatly simplify the process of working with arrays.

NumPy also provides a wide range of mathematical functions for working with arrays, including basic operations like addition, subtraction, and multiplication, as well as more advanced functions like trigonometric functions, exponential functions, and statistical functions.

In addition, NumPy provides tools for linear algebra, Fourier analysis, and random number generation, making it a versatile tool for a wide range of scientific and engineering applications. NumPy has a large and active community of developers who contribute to its development and maintenance. The community includes researchers, developers, and data scientists from academia and industry, who collaborate on improving the library and adding new features.

* + 1. **NUMPY FEATURES:**
* Multi-dimensional arrays that are faster and more memory-efficient than Python’s built-in lists.
* Broadcasting, which allows for operations to be performed between arrays of different shapes and sizes.
* Vectorized operations that allow mathematical functions to operate on entire arrays at once, making it faster and more efficient to perform calculations on large amounts of data.
* Linear algebra operations, such as matrix multiplication and decomposition.
* Integration with other libraries, such as SciPy, Pandas, and Matplotlib.
* Powerful array slicing and indexing techniques that make it easy to select and manipulate specific parts of an array.
  1. **TENSORFLOW:**

TensorFlow is a popular open-source software library that is used for building and training machine learning models. Developed by the Google Brain team in 2015, TensorFlow has become a go-to-choice for deep learning, artificial intelligence, and data analysis applications.

TensorFlow uses a dataflow graph to represent the computations in a machine learning model. The graph consists of nodes that represent mathematical operations, and edges that represent the data that flows between the nodes. The graph is built using TensorFlow's API, which allows developers to define the structure of the graph, and to specify the input and output data for the model.

One of the key features of TensorFlow is its ability to optimize the computations in the graph for efficient execution on a range of hardware, including CPUs, GPUs, and specialized hardware accelerators. This optimization is performed automatically by TensorFlow's runtime system, which can dynamically allocate computational resources to different parts of the graph as needed.

TensorFlow includes a wide range of tools and utilities to help developers build and train machine learning models. These include pre-built modules for common tasks, such as image and speech recognition, natural language processing, and time-series analysis. TensorFlow also includes a range of visualization tools to help developers understand and debug their models, and to monitor the training process.

In addition to its core features, TensorFlow has a large and active community of developers contributing to its development and maintenance. The TensorFlow community includes researchers, developers, and data scientists from academia and industry, who collaborate on improving the library and adding new features.

TensorFlow is used in a wide range of applications, from image and speech recognition to predictive modeling and natural language processing. It has become a go-to choose for developers who want to build and train machine learning models, and its popularity shows no signs of slowing down. With its powerful features, flexible API, and active community, TensorFlow is likely to remain a leading machine learning library for years to come.

* 1. **SKLEARN:**

Scikit-learn, also known as sklearn, is a popular open-source machine learning library for Python. It was first released in 2007 and has since become one of the most widely used machine learning libraries in Python.

Scikit-learn provides a wide range of machine learning algorithms and tools for classification, regression, clustering, and dimensionality reduction. It is designed to be easy to use and provides a simple and consistent interface for working with machine learning algorithms.

One of the key features of scikit-learn is its integration with NumPy and Pandas, two popular libraries for data manipulation in Python. This integration allows developers to easily load, preprocess, and analyze datasets before applying machine learning algorithms.

Scikit-learn also provides a wide range of tools for model selection and evaluation, including cross-validation, grid search, and metrics for evaluating model performance. This makes it easy for developers to select the best machine learning model for a given problem and evaluate its performance.

In addition, scikit-learn provides support for both supervised and unsupervised learning, making it a versatile tool for a wide range of machine learning tasks. It also supports feature selection and feature extraction, which can be used to improve model performance and reduce the dimensionality of datasets. Scikit-learn has a large and active community of developers who contribute to its development and maintenance. The community includes researchers, developers, and data scientists from academia and industry, who collaborate on improving the library and adding new features.

In summary, scikit-learn is a powerful and flexible machine learning library for Python. Its wide range of algorithms and tools, support for both supervised and unsupervised learning, and integration with other popular Python libraries make it a go-to choose for developers and data scientists who want to apply machine learning to their data.

* 1. **ATALANTA (ATPG) TOOL:**

The Atalanta tool, developed by the Bradley Department of Electrical and Computer Engineering at Virginia Tech, is a strong yet user-centric open-source testing tool for digital combinational circuits in VLSI design. Originally developed primarily for Automatic Test Pattern Generation (ATPG) and fault simulation, it targets stuck-at faults—typical errors in manufacturing in which a circuit node is stuck at logic 0 or logic 1.

Atalanta takes a description of a circuit in ISCAS89 netlist form and generates compact test patterns revealing such faults by propagating their effect to observable outputs using the FAN (Fanout-Oriented) algorithm. Its fault simulation employs a parallel pattern approach, processing multiple test vectors in parallel to assess fault coverage rapidly, a feature that balances speed and accuracy even with limited hardware.

Atalanta is designed for research and educational use, so its command-line interface and C-based code structure make it streamlined and adaptable, allowing users to tailor it for particular applications like test set compression or fault diagnosis.

Although limited to combinational logic—sequential circuits being outside its scope—it keeps its focus concentrated on basic principles of testing, and thus is a student's favorite. Even on April 1, 2025, it remains an effective learning tool and prototype tool, giving insight into the basic techniques that continue to be present in VLSI testing even today.

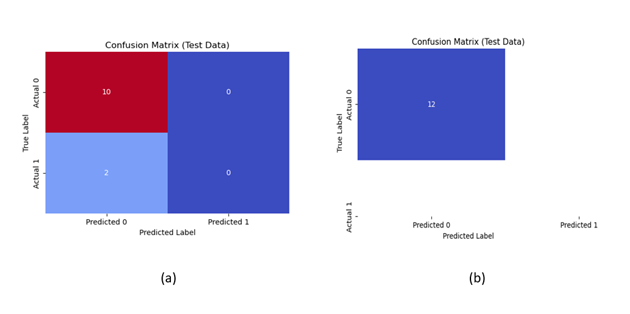
* + 1. **ATALANTA FEATURES:**
    - Reads circuit netlists in ISCAS89 format (e.g., c17.bench).
    - Algorithm: Employs the FAN (Fanout-Oriented) algorithm to produce effective test patterns.
    - Simulation: Uses parallel pattern single fault propagation for rapid fault coverage estimation.
    - Generates test vectors and fault coverage reports (e.g., 90% faults detected).

**CHAPTER-7**

**RESULTS**

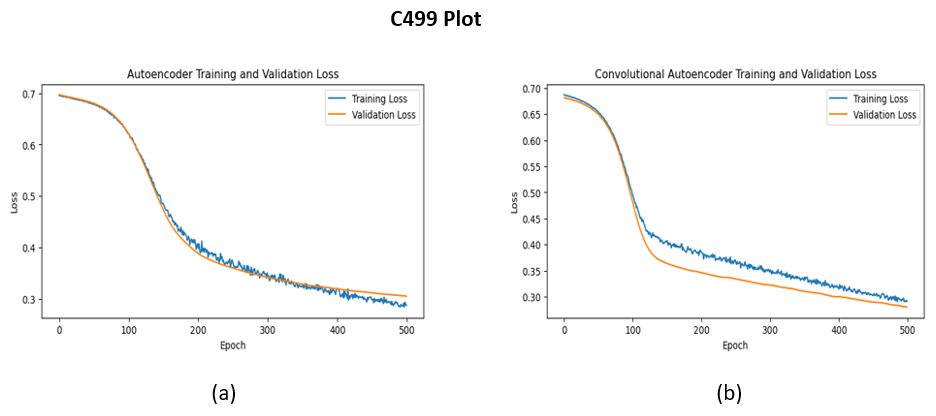
* 1. **32-bit SEC Circuit (c499):**

The C499 confusion matrices labeled (a) and (b) assess how well a classification model predicts labels 0 and 1 compared to the actual labels in test data. For matrix (a), representing an ANN, there are 10 correct predictions of label 0, 0 cases where label 0 was wrongly predicted as 1, 2 cases where label 1 was incorrectly predicted as 0, and 0 correct predictions of label 1. For matrix (b), representing a CNN, there are 12 correct predictions of label 0, 0 cases where label 0 was wrongly predicted as 1, 0 cases where label 1 was incorrectly predicted as 0, and 0 correct predictions of label 1.



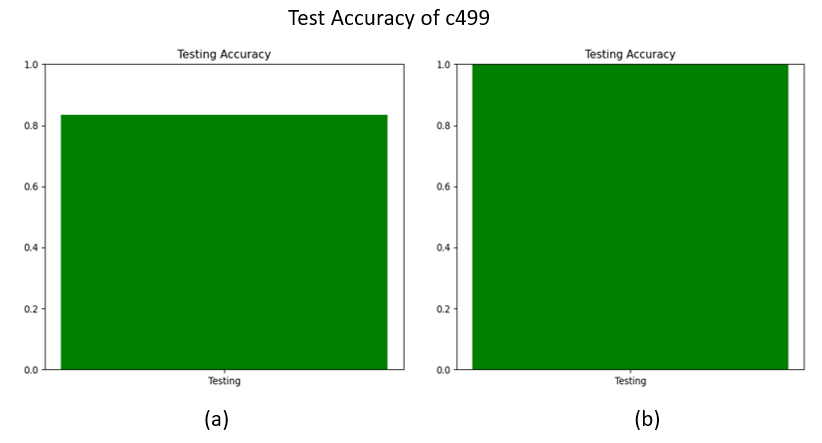
**Fig 7.1Confusion matrix (Test data) of c499 (a) ANN&(b) CNN**

The C499 plot includes two graphs, (a) for an ANN (Autoencoder) and (b) for a CNN (Convolutional Autoencoder), showing how training and validation losses change over 500 epochs. In graph (a), the ANN's training loss drops from 0.7 to 0.3, with the validation loss following a similar trend and stabilizing close to it. In graph (b), the CNN's training loss decreases from 0.65 to 0.3, with the validation loss tracking closely and showing a smooth decline. Both models learn well, with losses converging, indicating effective learning and good performance.

****

**Fig 7.2: Plot of C499 (a) ANN & (b) CNN**

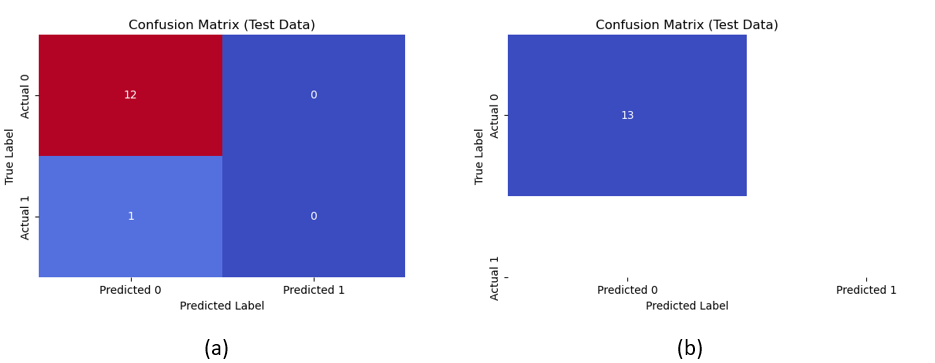
The C499 test accuracy is represented by two graphs, each showing the testing accuracy of a model, likely evaluated over a dataset. Both graphs display a uniform green fill, indicating a consistent accuracy level across the testing phase. The y-axis, labelled "Testing Accuracy," ranges from 0.0 to 1.0, where 1.0 represents perfect accuracy. The x-axis is labelled "Testing," suggesting the accuracy is measured over the entire test set. The solid green colour implies that the accuracy remains at or near 1.0 for both cases, indicating that the models achieved near-perfect performance on the test data, with no visible variation or decline. This could suggest highly effective model training or a potential issue such as overfitting or an imbalanced dataset favouring correct predictions.



**Fig 7.3: Testing Accuracy of c499 (a) ANN & (b) CNN**

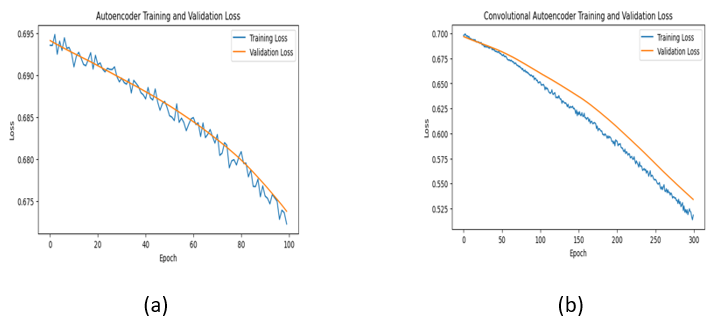
* 1. **27-Channel Interrupt Controller(c432):**

The confusion matrices above compare the performance of two models on the C432 circuit, where (a) is from an ANN (Artificial Neural Network) and (b) from a CNN (Convolutional Neural Network). In the ANN matrix, the model correctly classified 12 instances of class 0 and misclassified 1 instance of class 1 as class 0, indicating limited ability to identify class 1. In the CNN matrix, the model correctly predicted all 13 instances as class 0, but it failed to detect any class 1 samples. This indicates both models are biased toward class 0. ANN shows slightly better sensitivity to class 1 than CNN. The overall performance reflects challenges likely due to class imbalance or limited discriminative features for class 1.



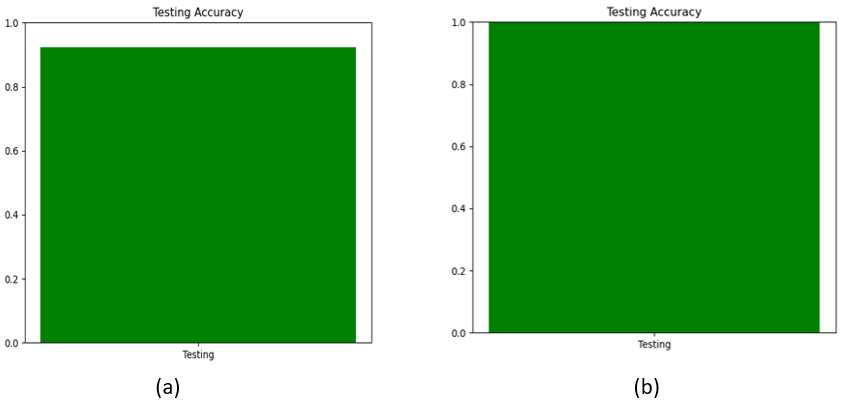
**Fig 7.4: Confusion matrix (Test data) of c432(a) ANN&(b) CNN**

The plot of C432 compares the training and validation loss of an autoencoder using ANN (Fig a) and CNN (Fig b). In Fig (a), the ANN model shows a steady decrease in both training and validation loss over 100 epochs, with both curves closely aligned, indicating consistent learning but relatively higher loss values, which suggests limited reconstruction performance. In contrast, Fig (b) demonstrates that the CNN model achieves significantly lower training and validation loss over 300 epochs, with a smooth and consistent decline, reflecting better feature extraction and reconstruction capability. The small gap between the two curves in both models indicates minimal overfitting, with the CNN performing more effectively overall.

****

**Fig 7.5: Plot of C432 (a) ANN & (b) CNN**

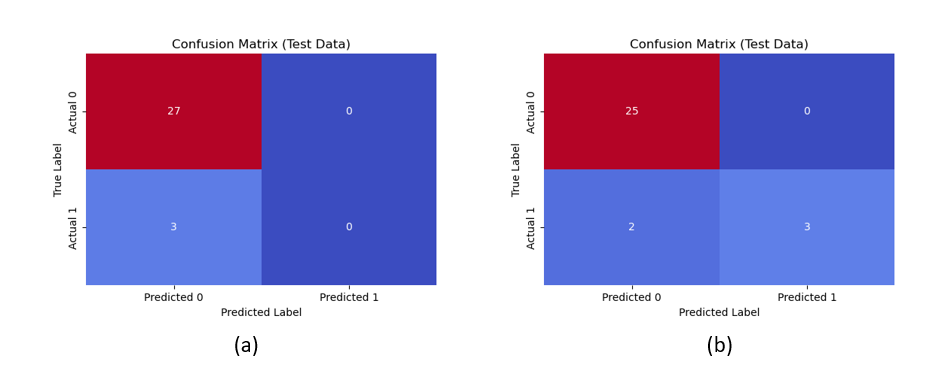
The Testing Accuracy of C432 is visually compared for both ANN and CNN models. In Fig (a), the ANN model achieves a high testing accuracy close to 0.93, indicating good performance on unseen data. However, Fig (b) shows that the CNN model reaches a perfect accuracy of 1.0, demonstrating superior performance and flawless classification on the test set. This highlights that the CNN model is more effective than the ANN model in capturing spatial patterns and features, leading to improved generalization and better accuracy on test data.

****

**Fig 7.6: Testing Accuracy of c432 (a) ANN & (b) CNN**

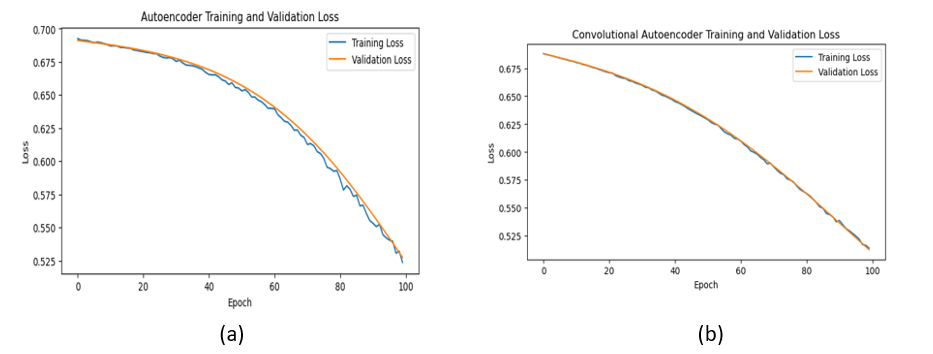
* 1. **8-bit ALU (c880):**

The confusion matrices above represent the C880 circuit, where (a) is from an ANN model and (b) is from a CNN model. In the ANN results, the model correctly predicted all 27 class 0 instances, but it failed to classify any class 1 instances, predicting all 3 as class 0 indicating poor sensitivity to class 1. The CNN model performed better overall, correctly predicting 25 instances of class 0 and 3 out of 5 class 1 instances, showing improved capability in detecting both classes. The CNN provides better balance between classes and demonstrates more generalization compared to ANN.



**Fig 7.7: Confusion Matrix of c880(a) ANN & (b) CNN**

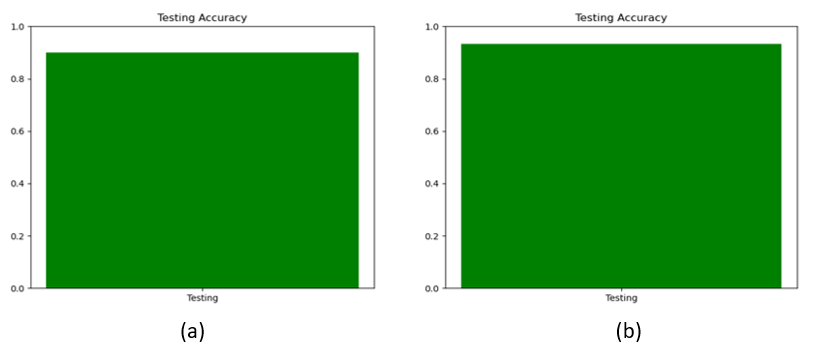
The Plot of C880 compares the training and validation loss of Autoencoders implemented using ANN and CNN. In Fig (a), the ANN-based Autoencoder shows a smooth and consistent decline in both training and validation loss over 100 epochs, with a slight gap between the two curves, indicating a mild overfitting tendency. On the other hand, Fig (b) shows the CNN-based Autoencoder with both training and validation loss curves almost overlapping, signifying a more stable and generalized learning process. The CNN model demonstrates better learning efficiency and less overfitting compared to the ANN model, making it more suitable for tasks involving spatial feature extraction.



**Fig 7.8: Plot of C880 (a) ANN & (b) CNN**

The Testing Accuracy of c880 shown in the above figure compares the performance of the Autoencoder models built using ANN and CNN architectures. In Fig (a), the ANN model achieves a testing accuracy slightly below 0.9, indicating decent performance but with room for improvement. In contrast, Fig (b) shows the CNN model reaching a slightly higher accuracy, closer to 1.0, reflecting better generalization and performance on unseen data. This demonstrates that the CNN-based Autoencoder is more effective than the ANN-based model in handling complex data patterns, likely due to its ability

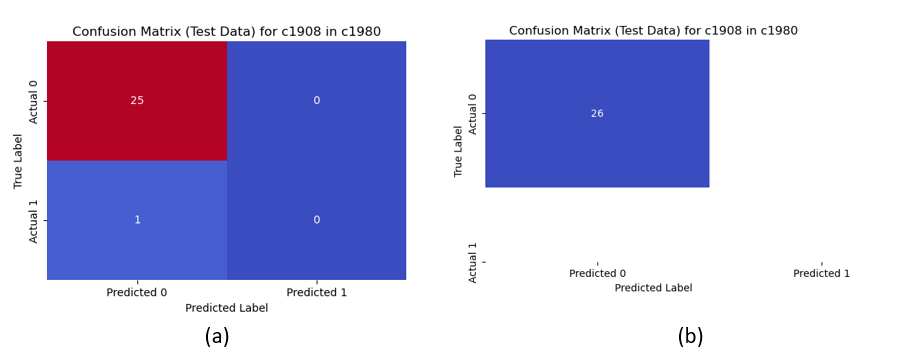
to extract spatial features efficiently.



**Fig 7.9: Testing Accuracy of c880 (a) ANN & (b) CNN**

* 1. **16-bit SEC/DED circuit(c1908):**

The confusion matrices represent test results for the C1908 circuit evaluated in the C1908 environment. In (a), the ANN model correctly predicted 25 instances of class 0 but misclassified 1 instance of class 1 as class 0, indicating a slight weakness in detecting minority class errors. In (b), the CNN model classified all 26 samples as class 0, showing no detection of class 1 at all. This suggests that while CNN maintained high accuracy for the dominant class, it lacked sensitivity toward minority classes. In comparison, ANN showed better balance in recognizing both classes, despite the small misclassification.



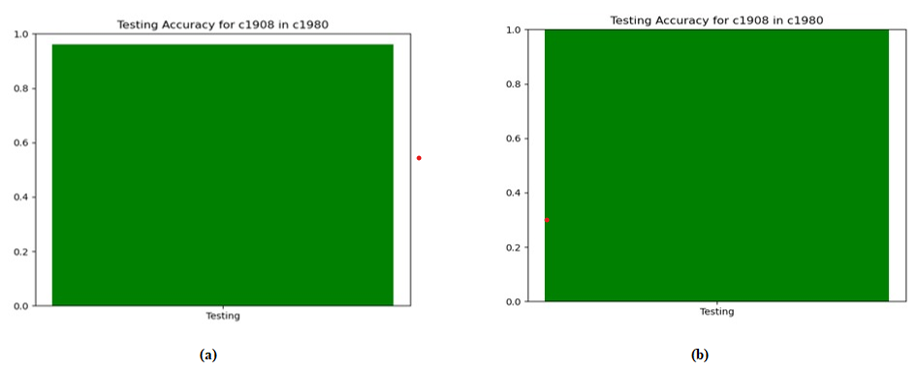
**Fig 7.10: Confusion Matrix of c1908(a) ANN & (b) CNN**

The Training and Validation Loss for c1908 in c1980 is illustrated in the figure above, comparing the performance of an Autoencoder using ANN (Fig a) and CNN (Fig b). In Fig (a), the ANN-based autoencoder shows a gradual and consistent decline in both training and validation loss over 100 epochs, indicating stable learning. The closeness of the two curves suggests a well-generalized model with minimal overfitting. In contrast, Fig (b) shows the CNN-based autoencoder, where both training and validation losses decrease more sharply, particularly in the later epochs. This suggests that the CNN model converges faster and captures deeper patterns in the data. The minimal gap between training and validation loss in the CNN plot also implies strong generalization and efficient feature learning, making it more robust than the ANN-based model.



**Fig 7.11: Plot of C1908 (a) ANN & (b) CNN**

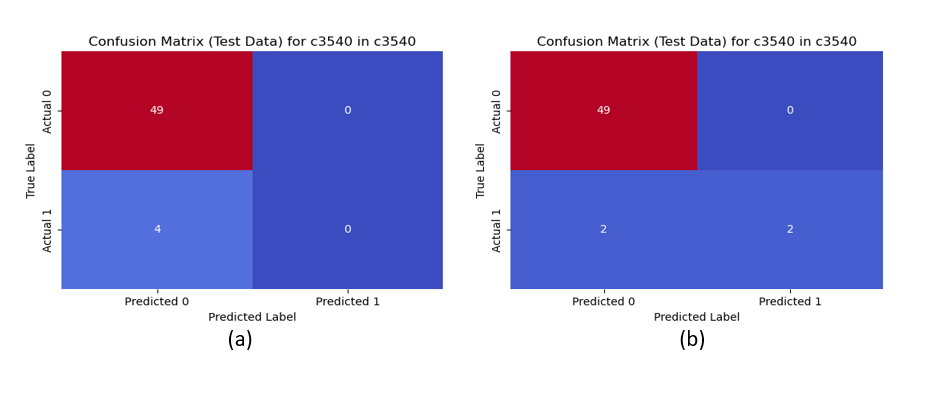
The testing accuracy comparison between the ANN-based and CNN-based autoencoders for c1908 tested in c1908 demonstrates that both models achieve high performance. The ANN model reaches an accuracy of approximately 95%, indicating strong generalization. However, the CNN model performs even better, achieving nearly 100% accuracy. This highlights the CNN autoencoder's superior ability to learn and reconstruct features effectively, likely due to its convolutional layers that preserve spatial hierarchies in the data. Overall, the CNN model proves to be more efficient and accurate for this task.



**Fig 7.12: Testing Accuracy of c1908 (a) ANN & (b) CNN**

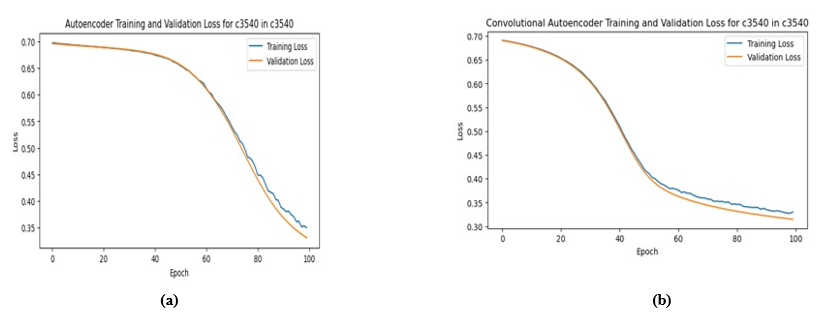
* 1. **8-bit ALU (c3540):**

The confusion matrices illustrate the test results for the C3540 circuit evaluated within the same C3540 environment. In (a), the ANN model correctly predicted all 49 class 0 samples, but failed to predict any class 1 samples, misclassifying 4 of them as class 0. In (b), the CNN model also perfectly classified class 0 with 49 correct predictions but showed improved performance on class 1, correctly identifying 2 out of 4 instances. This indicates that while both models handle majority class detection well, CNN provides better overall balance by identifying some minority class samples correctly, making it more effective in scenarios requiring sensitivity to both classes.



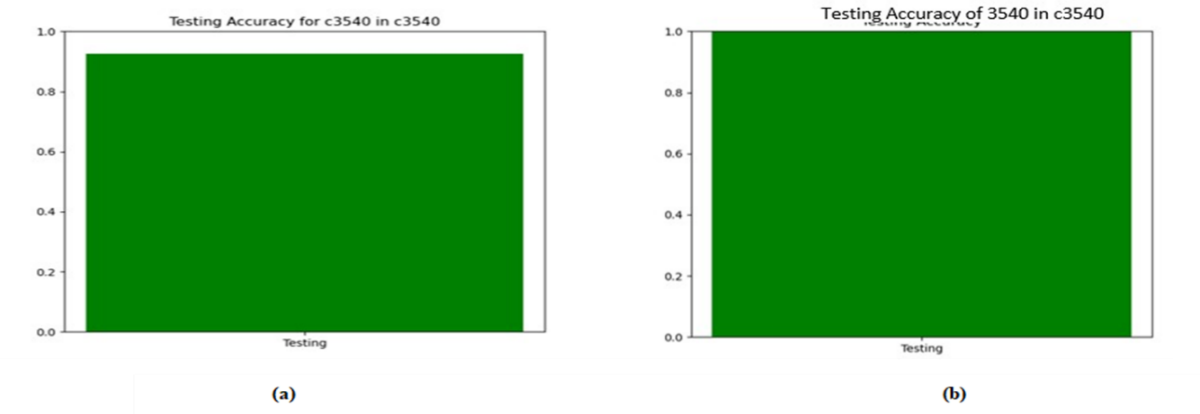
**Fig 7.13: Confusion Matrix of c3540 (a) ANN & (b) CNN**

The training and validation loss curves for the c3540 dataset reveal notable differences between the Autoencoder and the Convolutional Autoencoder models. The Autoencoder shows a steady decrease in both training and validation losses across 100 epochs, with both curves remaining closely aligned. This indicates stable learning and good generalization, although the convergence is relatively slower. In contrast, the Convolutional Autoencoder demonstrates a faster and smoother decline in loss values, achieving lower final losses within the same number of epochs. The close alignment of the training and validation curves suggests strong generalization without overfitting. Overall, the Convolutional Autoencoder outperforms the standard Autoencoder by learning more efficiently and capturing relevant features more effectively from the c3540 dataset.



**Fig 7.14: Plot of c3540 (a) ANN & (b) CNN**

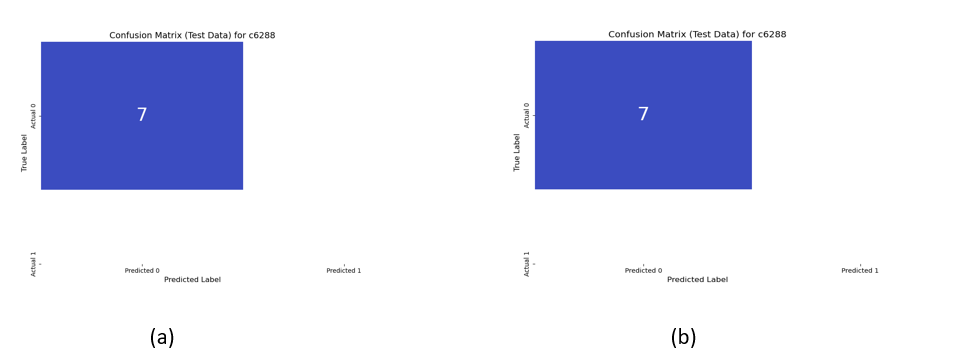
The testing accuracy comparison between the ANN-based and CNN-based autoencoders for 3540 tested in c3540 demonstrates that both models achieve high performance. The ANN model reaches an accuracy of approximately 95%, indicating strong generalization. However, the CNN model performs even better, achieving nearly 100% accuracy. This highlights the CNN autoencoder's superior ability to learn and reconstruct features effectively, likely due to its convolutional layers that preserve spatial hierarchies in the data. Overall, the CNN model proves to be more efficient and accurate for this task.



**Fig 7.15: Testing Accuracy of c1908 (a) ANN & (b) CNN**

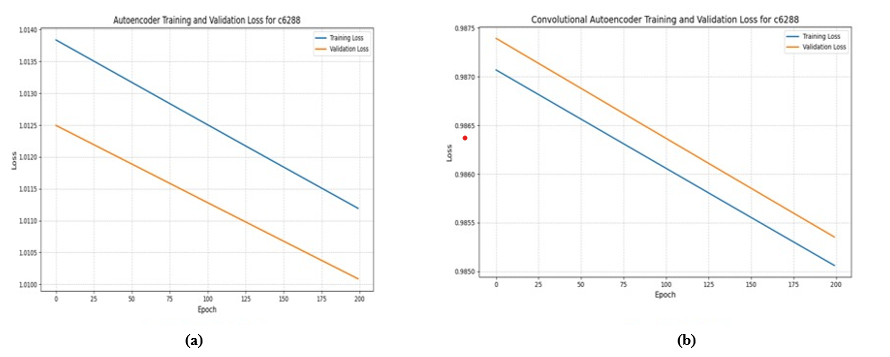
* 1. **16x16 multiplier(c6288):**

The confusion matrices above display the test results for the C6288 circuit using two different models: (a) ANN and (b) CNN. Both models produced identical results, correctly classifying all 7 samples as class 0 and failing to detect any class 1 instances. This suggests that the dataset is highly imbalanced or lacks sufficient representation of class 1 during training, leading both models to default to predicting only the majority class. As a result, while accuracy may seem high, the models lack generalization and are ineffective at detecting the minority class (class 1).



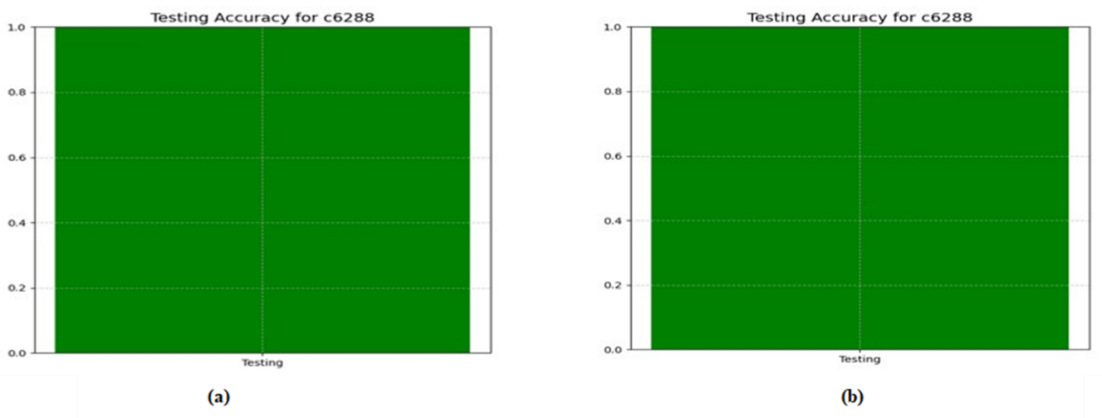
**Fig 7.16: Confusion Matrix of c6288 (a) ANN & (b) CNN**

A c6288 features two-line graphs comparing the training and validation loss over epochs for an autoencoder and a convolutional autoencoder. The left graph shows training and validation loss decreasing from 1.0140 to 1.0100 over 200 epochs, indicating consistent learning. The right graph displays a similar trend, with loss dropping from 0.9875 to 0.9850, suggesting the convolutional version's slight edge. Both models improve performance, with the convolutional autoencoder showing lower loss and better feature reconstruction. The close alignment of curves indicates good generalization in both cases.



**Fig 7.17: Plot of c6288 (a) ANN & (b) CNN**

The testing accuracy comparison between the ANN-based and CNN-based autoencoders for c6288 tested in c6288 demonstrates that both models achieve exceptional performance. The ANN model reaches an accuracy of approximately 100%, indicating perfect generalization under the given conditions. Similarly, the CNN model also achieves nearly 100% accuracy, matching the ANN model's performance. This suggests that both models are highly effective in learning and reconstructing features, with the CNN's convolutional layers and the ANN's architecture both preserving essential data patterns. Overall, both models prove to be equally efficient and accurate for this task, as indicated by the consistent 100% accuracy in the charts.



**Fig 7.18: Testing Accuracy of c6288 (a) ANN & (b) CNN**

* 1. **Performance metrics for existing and Proposed model:**

A comparative analysis of fault detection performance between Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) was conducted using ISCAS’85 benchmark circuits. The study evaluated both models based on training parameters, testing parameters, precision, recall, F1-score, support, and accuracy. The results indicate that CNNs consistently outperform ANNs in fault classification across various circuits. While ANNs achieved accuracy levels ranging from 83% to 100%, CNNs demonstrated superior performance, achieving 100% accuracy in multiple cases, including circuits C17, C432, C499, C1980, and C6288. In circuits like C880 and C3540, CNNs maintained higher accuracy compared to ANNs, with 93% and 96% accuracy, respectively.

CNNs also exhibited higher precision and F1-scores, particularly in detecting faulty patterns, ensuring better classification reliability. The recall values remained at 1.00 for both models, indicating that they effectively identified fault-free patterns. However, CNNs provided enhanced fault detection with fewer false negatives, reducing the risk of misclassification. This improvement can be attributed to CNNs' advanced feature extraction and pattern recognition capabilities, which allow for better generalization and fault localization in complex circuits.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S.N. O | Model | | ISACAS’85 benchmarks Circuits | | | | | |
| C432 | C499 | C880 | C1908 | C3540 | C6288 |
| 1. | ANN | Training parameters | 50 | 45 | 118 | 102 | 212 | 28 |
| Testing parameters | 13 | 12 | 30 | 26 | 53 | 7 |
| Precision | 0.86 | 0.83 | 0.90 | 0.96 | 0.92 | 1.00 |
| Recall | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| F1score | 0.92 | 0.91 | 0.95 | 0.98 | 0.96 | 1.00 |
| Support | 6 | 10 | 27 | 25 | 49 | 7 |
| Accuracy (%) | 92 | 83 | 90 | 96 | 92 | 100 |
| 2. | CNN | Training parameters | 50 | 45 | 118 | 102 | 212 | 28 |
| Testing parameters | 13 | 12 | 30 | 26 | 53 | 7 |
| Precision | 1.00 | 1.00 | 0.93 | 1.00 | 0.96 | 1.00 |
| Recall | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| F1score | 1.00 | 1.00 | 0.96 | 1.00 | 0.98 | 1.00 |
| Support | 13 | 12 | 25 | 26 | 49 | 7 |
| Accuracy (%) | 100 | 100 | 93 | 100 | 96 | 100 |

**Table 7.19 Comparative Analysis of Fault Detection Performance Between ANN and CNN for ISCAS’85 Benchmark Circuits**

**CH** **APTER-8**

**CONCLUSION AND FUTURE SCOPE**

This project aims to simplify finding and fixing errors in the tiny circuits that run our electronic devices, using a clever pair: a Convolutional Autoencoder (CAE) and a Convolutional Neural Network (CNN). Unlike older methods that can miss things, this approach is fantastic at quickly spotting perfect patterns and catching most mistakes, acting like a super-smart detective for complicated circuits. It outdoes regular neural networks by clearly seeing both the big picture and tiny details, proving how awesome modern tech can be. Even though it handles tricky circuits well, some clever errors still sneak past, especially in really tough setups, showing there’s still work to do. It’s a huge step toward making sure our gadgets work flawlessly from day one, saving time and effort in the process.

We could boost the CAE-CNN team to catch those sneaky, hard-to-find mistakes by adding helpers that track how signals flow, like a guide who knows time tricks. Testing it on a wide mix of new circuits will prove if it’s up for today’s tech challenges. Tossing in extra hints like how quick a circuit works or how much energy it uses could make it even better at finding faults. Streamlining it to be fast and easy for real factories is a smart next move. Blending in fresh ideas, like letting it learn from slip-ups or invent new ways to spot issues, could take it to the next level. Exploring how it works with super-modern circuits or even teaming it up with other tech tools could unlock more possibilities. This is only the beginning of creating amazing tools to keep our devices running smoothly and reliably.

**CHAPTER-9**

**REFERENCES**

1. Brglez F, Bryan D, Kozminski K (1989) Combinational profiles of sequential benchmark circuits. In IEEE international symposium on circuits and systems pp 1929–1934
2. Gaber L, Hussein AI, Moness M (2021) Fault Detection based on Deep Learning for Digital VLSI Circuits. Procedia Computer Science 194:122131
3. Rashinkar P, Paterson P, Singh L (2007) Singh, System-on-a chip verification: methodology and techniques: Springer Science & Business Media
4. Jutman A, Ubar R (2000) Design error diagnosis in digital circuits with stuck-at fault model. Microelectron Reliab 40:307–320
5. Wahba A, Borrione D (1995) Design error diagnosis in sequential circuits. In Proceeding Advanced Research Working Conference on Correct Hardware Design and Verification Methods pp 171–188
6. Jo S, Matsumoto T, Fujita M (2014) SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. IPSJ Transactions on System LSI Design Methodology 7:46–55
7. Rodríguez Gómez L (2017) Machine Learning Support for Logic Diagnosis. Doctoral dissertation, university of Stuttgart.
8. El Mandouh E, Wassal AG (2018) Application of Machine Learning Techniques in Post-Silicon Debugging and Bug Localization. J Electron Test 34:163–181
9. Gaber L, Hussein AI, Moness M (2019) Improved automatic correction for digital VLSI circuits. In 2019 Proceeding 31st international conference on microelectronics (ICM), 2019, pp 18–22
10. Gaber L, Hussein AI, Moness M (2020) Incremental Automatic Correction for Digital VLSI Circuits. Presented at the Proceeding 11th International Conference on VLSI (VLSI 2020)
11. Gaber L, Hussein AI, Moness M (2021) Fast Auto-Correction algorithm for Digital VLSI Circuits. Procedia Computer Science 182:95–102
12. Gaber L, Hussein AI, Mahmoud H, Mabrouk MM, Moness M (2020) Computation of minimal unsatisfiable sub formulas for SAT-based digital circuit error diagnosis. J Ambient Intel Humanized Comp pp 1–19
13. Gaber L, Hussein AI, Moness M (2020) Fast Auto-Correction algorithm for Digital VLSI Circuits. Presented at the 17th International Learning & Technology Conference
14. Cook SA (1971) The complexity of theorem-proving procedures. In Proceedings of the third SSA Eual ACM symposium on Theory of computing pp 151–158
15. Gao Z, Cecati C, Ding SX (2015) A survey of fault diagnosis and fault tolerant techniques Part I: Fault diagnosis with model-based and signal-based approaches. IEEE Trans Industry Electron 62:3757–3767
16. Ng A (2011) Sparse autoencoder. CS294A Lecture notes 72:1–19
17. Baldi P (2012) Autoencoders, unsupervised learning, and deep architectures. In Proceedings of ICML workshop on unsupervised and transfer learning pp 37–49
18. Vincent P, Larochelle H, Bengio Y, Manzagol PA (2008) Extracting and composing robust features with denoising autoencoders. In Proceedings of the 25th international conference on Machine learning pp 1096–1103
19. Rifai S, Vincent P, Muller X, Glorot X, Bengio Y (2011) Contractive auto-encoders: Explicit invariance during feature extraction. In Icml
20. Bryan D (1985) The ISCAS'85 benchmarks circuits and netlist format. North Carolina State University 25

**APPENDIX**

**# import Packages**

import numpy as np

import tensorflow as tf

from tensorflow. keras. models import Model

from tensorflow. keras. layers import Input, Conv1D, MaxPooling1D, UpSampling1D, Dense, Dropout, Flatten, Reshape

from tensorflow. keras. optimizers import Adam

import matplotlib. pyplot as plt

from sklearn. model\_selection import train\_test\_split

from sklearn. ensemble import RandomForestClassifier

from sklearn. metrics import accuracy\_score, classification\_report, confusion\_matrix

import seaborn as sns

import joblib

**# Set random seeds for reproducibility**

np. random. seed (42)

tf.random.set\_seed (42)

**# Step 1: Define Test Patterns (57 patterns)**

test\_patterns = [

    "000010110010000000000100100000000000000000000101100100000000010010000000",

    "1110010000000100000000000000011000000010111100100000001000000000000000110",

    "0000000000000000000000000100000000010000000000000000000000000000000000000",

    "0000000000000000000000000000010000100000000000000000000000000000000000000",

    "0000000000000000000000110010000000000000000000000000000000001100110000",

    "0000000001000000000000000000000000000001000000000000000000000000000000000",

    "0000000000000000000010000000000001000000100000000000000000001000000000000",

    "0000000000000000000000000000001000000000000000000000000000000000000000000",

    "00000000000000000000000001000000000100001000000000000000000000000100000",

    "000000000000000000110000000000100000000000000000000000000011000000000011",

    "0000100000001100000000000000000000000000000001100000011000000000000000000",

    "000000010010000000001011001000000000000000000000100100000000101100100000",

    "000010110010000000000010100000000000000000000101100100000000001010000000",

    "010000000100000000100000011000000000000000100000001000000101000001100000",

    "1011100010010001000000000000011000000010110111000100100010000000000000110",

    "0000000000001000000000000000000000001000100000000000010000000000000000000",

    "000000000000000011000000000010000000000000000000000000000110000000001100",

    "0010000000000011000000000000000000000000000110000000000110000000000000000",

    "0000000000000000000000000010000000000000000000000000000000000000000000000",

    "0000000000000110101110001001000100100000100000000000001101011100010010001",

    "0010000001100000010000000100000000000000010100000011000000100000001000000",

    "000010110010000000000001001000000000000000000101100100000000000100100000",

    "0000000000000010000000000000000000000000000000000000000000000000000000000",

    "000000000000000000001100000010000000000000000000000000000000110000001100",

    "0011000000000010000000000000000000000000000110000000000110000000000000000",

    "0000000001000000000000000000000000000001100000000010000000000000000000000",

    "0010001010100010000000000000000000000000000100010001000100000000000000000",

    "000010110010000000000001010000000000000000000101100100000000000101000000",

    "0000001000110000000000000000000000000000000000011001100000000000000000000",

    "0000100000000000000000000000000000000100000000000000000000000000000000000",

    "0000100010000000000010000000000000000000000001000100000000001000000000000",

    "0000000000000110111001000000010000100000100000000000001101110010000000100",

    "000000000000000000000010001100000000000000000000000000000000001100110000",

    "0000000000000000001000000000000000000000000000000000000000000000000000000",

    "0000000000000000000010000000000001000000000000000000000000000000000000000",

    "000010110010000000000010010000000000000000000101100100000000001001000000",

    "1100000000000000000000000000100000000000011000000000000000000100000001000",

    "0010000010100000010000000100000000000000001100000101000000100000001000000",

    "0000000000000000000001000000000000000010000000000000000000000000000000000",

    "0100000001000000001000001010000000000000001000000010000000110000010100000",

    "0000000000000000000000000000100010000000000000000000000000000000000000000",

    "000001001000000000001011001000000000000000000010010000000000101100100000",

    "000000000000000000100010101000100000000000000000000000000010001000100010",

    "000000000000000000000010000000110000000000000000000000000000001100000011",

    "0000110000001000000000000000000000000000000001100000011000000000000000000",

    "0000001000000000000000000000000000000000000000000000000000000000000000000",

    "000000010100000000001011001000000000000000000000101000000000101100100000",

    "0000000000000000000001000000000000000010100000000000000000000100000000000",

    "0000000000001000000000000000000000001000000000000000000000000000000000000",

    "000000101000000000001011001000000000000000000001010000000000101100100000",

    "0000100000000000000000000000000000000100100001000000000000000000000000000",

    "0000001100000010000000000000000000000000000000011000000110000000000000000",

    "000000000000000000000000000010001000000010000000000000000000000000001000",

    "0000000000100000000000000000000000000000000000000000000000000000000000000",

    "0000000000000000000000100000000000000000000000000000000000000000000000000",

    "0000000000000000000000000000010000100000100000000000000000000000000000100",

    "0000000000010000000000000000000000000000000000000000000000000000000000000",

]

**# Normalize all patterns to 73 bits**

def normalize\_pattern(pattern, target\_length=73):

    if len(pattern) > target\_length:

        return pattern[:target\_length]

    elif len(pattern) < target\_length:

        return pattern + '0' \* (target\_length - len(pattern))

    return pattern

test\_patterns\_fixed = [normalize\_pattern(seq) for seq in test\_patterns]

**# Convert to NumPy array and reshape for 1D convolution (samples, timesteps, channels)**

X = np.array([list(map(int, list(seq))) for seq in test\_patterns\_fixed]).astype(np.float32)

X = X.reshape((X.shape[0], X.shape[1], 1))  # Shape: (57, 73, 1)

**# Synthetic labels (fixed with seed; replace with real labels if available)**

y = np.random.RandomState(42).randint(0, 2, size=(len(test\_patterns),))

**# Step 2: Split Data**

X\_train, X\_test, y\_train, y\_test = train\_test\_split (

    X, y, test\_size=0.2, random\_state=42, stratify=y

)

print ("Training samples:", X\_train.shape[0])

print ("Test samples:", X\_test.shape[0])

**# Step 3: Build Convolutional Autoencoder Model**

input\_shape = (73, 1)  # 73 timesteps, 1 channel

input\_layer = Input(shape=input\_shape)

**# Encoder**

encoded = Conv1D(32, kernel\_size=3, activation='relu', padding='same')(input\_layer)

encoded = MaxPooling1D(pool\_size=2)(encoded)  # Downsample to (36, 32)

encoded = Dropout(0.2)(encoded)

encoded = Conv1D(16, kernel\_size=3, activation='relu', padding='same')(encoded)

encoded = MaxPooling1D(pool\_size=2)(encoded)  # Downsample to (18, 16)

encoded = Dropout(0.2)(encoded)

**# Flatten for latent representation**

encoded\_flat = Flatten() (encoded)  # Shape: (18 \* 16) = 288

latent\_dim = 32 # Reduce to a smaller latent space

encoded\_dense = Dense(latent\_dim, activation='relu')(encoded\_flat)

**# Decoder**

decoded\_dense = Dense (18 \* 16, activation='relu’) (encoded\_dense) # Expand back to 288

decoded\_reshape = Reshape ((18, 16)) (decoded\_dense) # Reshape to (18, 16)

decoded = Conv1D (16, kernel\_size=3, activation='relu', padding='same’) (decoded\_reshape)

decoded = UpSampling1D(size=2) (decoded) # Up sample to (36, 16)

decoded = Dropout (0.2) (decoded)

decoded = Conv1D (32, kernel\_size=3, activation='relu', padding='same’) (decoded)

decoded = UpSampling1D(size=2) (decoded) # Up sample to (72, 32)

decoded = Conv1D (1, kernel\_size=3, activation='sigmoid', padding='same’) (decoded) # Output (72, 1)

**# Adjust output to match input shape (73, 1) by padding**

from tensorflow.keras.layers import ZeroPadding1D

decoded = ZeroPadding1D (padding= (0, 1)) (decoded) # Pad 1 timestep to get (73, 1)

autoencoder = Model (input\_layer, decoded)

autoencoder. Compile (optimizer=Adam (learning rate=0.00005), loss='binary\_crossentropy')

**# Step 4: Train Autoencoder**

history = autoencoder. Fit (

    X\_train, X\_train,

    epochs=500,

    batch size=32,

    verbose=1,

    validation split=0.2

)

**# Step 4.1: Plot Training and Validation Loss**

plt.figure(figsize=(8, 4))

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Convolutional Autoencoder Training and Validation Loss')

plt. ylabel('Loss')

plt.xlabel('Epoch')

plt.legend()

plt.show()

**# Step 5: Extract Encoded Features**

encoder = Model(inputs=autoencoder.input, outputs=encoded\_dense)  # Latent representation (32-dim)

X\_train\_encoded = encoder.predict(X\_train)

X\_test\_encoded = encoder.predict(X\_test)

**# Step 6: Compute Reconstruction Error and Fault Mask**

reconstructed \_train = autoencoder. predict(X\_train)

mse\_train = np. mean (npower (X\_train - reconstructed train, 2), axis=(1, 2)) # Average over timesteps and channels

threshold = np. mean(mse\_train) + 1.5 \* np.std(mse\_train)

fault\_mask\_train = (mse\_train > threshold). astype(int)

num\_faults\_train = np.sum(fault\_mask\_train)

print (f"Number of faults detected in training data (reconstruction error): {num\_faults\_train}")

reconstructed\_test = autoencoder. predict(X\_test)

mse\_test = np. mean (np. Power (X\_test - reconstructed\_test, 2), axis= (1, 2))

fault\_mask\_test = (mse\_test > threshold).astype(int)

num\_faults\_test = np.sum(fault\_mask\_test)

print (f"Number of faults detected in test data (reconstruction error): {num\_faults\_test}")

**# Step 7: Train Random Forest**

rf\_classifier = RandomForestClassifier (

    n\_estimators=200,

    max\_depth=10,

    min\_samples\_split=5,

    min\_samples\_leaf=2,

    random\_state=42

)

rf\_classifier.fit (X\_train\_encoded, fault\_mask\_train)

**# Step 8: Evaluate Random Forest Accuracy and Confusion Matrix**

**# Training**

y\_pred\_train = rf\_classifier. Predict(X\_train\_encoded)

train \_accuracy = accuracy\_score (fault\_mask\_train, y\_pred\_train)

print (f"\n✅ Random Forest Training Accuracy (on fault mask): {train\_accuracy:.2f}")

cm\_train = confusion\_matrix (fault\_mask\_train, y\_pred\_train)

print ("\confusion Matrix (Training Data):")

print(cm\_train)

plt. figure (figsize= (6, 4))

sns. heatmap (cm\_train, annot=True, fmt='d', cmap='coolwarm', cbar=False,

            xticklabels= ['Predicted 0', 'Predicted 1'],

            yticklabels= ['Actual 0', 'Actual 1'])

plt. title ('Confusion Matrix (Training Data)')

plt. ylabel ('True Label')

plt. xlabel ('Predicted Label')

plt. show ()

**# Testing**

y\_pred\_test = rf\_classifier. Predict(X\_test\_encoded)

test\_accuracy = accuracy\_score (fault\_mask\_test, y\_pred\_test)

print (f"✅ Random Forest Testing Accuracy (on fault mask): {test\_accuracy:.2f}")

print (classification\_report (fault\_mask\_test, y\_pred\_test, zero division=1))

cm\_test = confusion\_matrix (fault\_mask\_test, y\_pred\_test)

print ("\confusion Matrix (Test Data):")

print(cm\_test)

plt. figure (figsize= (6, 4))

sns. heatmap (cm\_test, annot=True, fmt='d', cmap='coolwarm', cbar=False,

            xticklabels= ['Predicted 0', 'Predicted 1'],

            yticklabels= ['Actual 0', 'Actual 1'],

            annot\_kws= {"size": 25})

plt. title ('Confusion Matrix (Test Data)')

plt. ylabel ('True Label')

plt. xlabel ('Predicted Label')

plt. show ()

**# Step 8.1: Plot Training and Testing Accuracy**

plt. figure (figsize= (12, 5))

plt. Subplot (1, 2, 1)

plt. Bar(['Training'], [train accuracy], color='blue')

plt. ylim (0, 1)

plt. title ('Training Accuracy')

plt. ylabel('Accuracy')

plt. Subplot (1, 2, 2)

plt. Bar(['Testing'], [test accuracy], color='green')

plt. ylim (0, 1)

plt. title ('Testing Accuracy')

plt. tight layout ()

plt. show ()

**# Step 9: Reconstruct All 57 Patterns and Print Separately**

reconstructed\_all = autoencoder.predict(X)

reconstructed\_all\_binary = (reconstructed\_all > 0.3).astype(int).reshape(reconstructed\_all.shape[0], 73)

print ("\nOriginal 57 Test Patterns:")

for i in range(len(X)):

    original = ''.join(map(str, X[i].astype(int).flatten()))

    print(f"Pattern {i+1}: {original}")

print ("\nReconstructed 57 Test Patterns:")

for i in range(len(X)):

    reconstructed = '‘. join(map (str, reconstructed\_all\_binary[i]))

    print (f"Pattern {i+1}: {reconstructed}")

**# Step 10: Save the Model**

joblib. dump (rf\_classifier, "RandomForest\_fault\_classifier.pkl")