Design for Testability
Project: Logic BIST Implementation

Rupak Antani
2022102045
rupak.antani@students.iiit.ac.in

Abhinav S
2022102037
abhinav.s@students.iiit.ac.in

Himanshu Gupta
2022102002
himanshu.gupta@students.iiit.ac.in

Akshat Tiwari
2022102043
akshat.tiwari@students.iiit.ac.in

***Abstract*—Testing is a critical process in ensuring that integrated circuits (ICs) function correctly and reliably. Traditional methods like external Automatic Test Equipment (ATE) and scan-based testing can detect faults effectively but are often costly and time-consuming. Logic Built-In Self-Test (Logic BIST) is an essential technique for automating the testing of combinational circuits, offering a self-contained, efficient, and cost-effective solution. This project focuses on implementing a Logic BIST system for the ISCAS '85 c432 circuit. Key components include a Type-2 Modular Form Linear Feedback Shift Register (LFSR)-based Test Pattern Generator (TPG), a phase Shifter for generating high fault coverage test patterns, a mode selection circuit that switches between normal operation and BIST mode, a Multiple Input Signature Register-based Output Response Analyzer (ORA) for signature generation, and a BIST Controller for fault detection. Python script is used to generate tap points for the phase shifter. The design and simulation, carried out using Verilog and demonstrate the system's effectiveness in detecting faults and achieving high fault coverage, showcasing the potential of Logic BIST for scalable and efficient circuit testing.**

I. Introduction

Testing is an important step in designing and producing digital circuits to make sure they work correctly and reliably. It helps find manufacturing defects, random errors, and performance problems that could affect how the circuit functions. As integrated circuits (ICs) become more advanced, the need for faster and more efficient testing methods has increased. Traditional testing methods, like using external Automatic Test Equipment (ATE) or scan-based testing, are effective but can be expensive, time-consuming, and depend on external tools. To overcome these challenges, new techniques like Built-In Self-Test (BIST), a part of Design for Testability (DFT), have become popular.

BIST is a smart testing method that allows circuits to test themselves while they are operating. It has three main parts: a Test Pattern Generator (TPG) to create test signals, an Output Response Analyzer (ORA) to check the circuit's output, and a control system to detect errors. This method applies test signals to the circuit, checks the outputs, and identifies faults without needing external equipment. By having the testing capability built into the circuit itself, BIST saves money, reduces testing time, and improves how well faults are detected. It is also great for testing circuits in real-time or in the field.

The Logic BIST technique can be used in various applications such as embedded systems, FPGA-based designs, and ASICs, making it a versatile tool in modern circuit testing.

II. Scope of this Project

The scope of this project is to design and implement a Logic Built-In Self-Test (Logic BIST) system for the ISCAS '85 c432 combinational circuit. The features of this project are as follows:

1) Develop key components of the Logic BIST system, including:
   - Test Pattern Generator (TPG) for efficient test pattern generation.
   - A phase shifter to enhance fault coverage by producing diverse test patterns.
   - A mode selection circuit to switch seamlessly between normal operation and BIST mode.
   - Output Response Analyzer (ORA) for compact signature generation and fault detection.
   - A BIST Controller to manage and coordinate the testing process.
2) Any stuck-at fault can be introduced in any or the wires of the circuit using a python script and accordingly we can do the testing process.
3) Utilize a Python script to determine optimized tap points for the phase shifter to improve fault coverage efficiency.
4) Simulate and validate the Logic BIST system using Verilog, demonstrating its capability to detect faults effectively while ensuring high fault coverage.
5) Though we have implemented it on only a single circuit, the BIST architecture developed by us can be easily extended to any circuit by making small changes in the TPG and phase shifter.

III. Overview of Logic BIST Architecture

The architecture of a Logic Built-In Self-Test (BIST) system is designed to perform self-testing within a chip, ensuring fault detection without the need for external testing equipment. Figure 1 illustrates a conventional Logic BIST architecture. The key components of this architecture are as follows:

A. Test Pattern Generator (TPG)

The TPG is responsible for generating test patterns, either pseudo-random or deterministic, that are applied to the Circuit Under Test (CUT). It is typically implemented using a Linear Feedback Shift Register (LFSR) for compact and efficient test pattern generation.

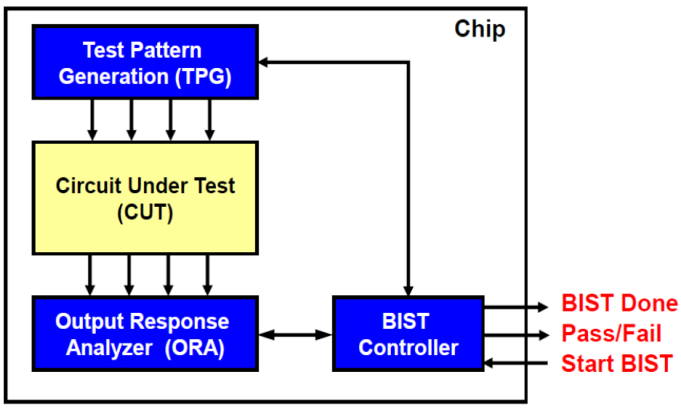

Fig. 1. Logic Built-In-Self-Test (BIST) Architecture

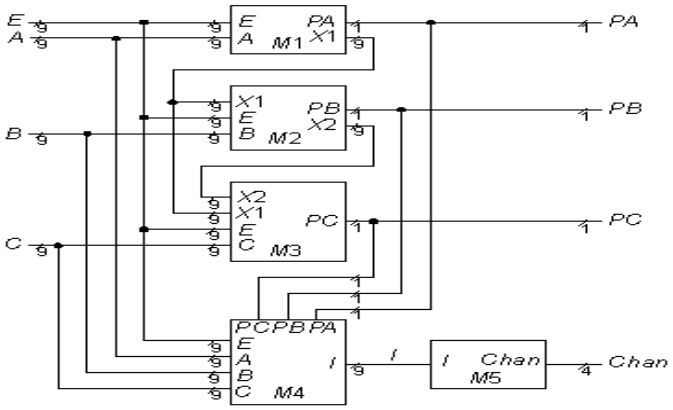

Fig. 2. ISCAS '85 c432

B. Circuit Under Test (CUT)

The CUT represents the part of the circuit being tested. Test patterns generated by the TPG are applied to the CUT, and its responses are captured and analyzed.

C. Output Response Analyzer (ORA)

The ORA evaluates the responses from the CUT. It compresses the output data into a signature, which is compared against a reference signature to determine if faults exist. A Multiple Input Signature Register (MISR) is often used for this purpose.

D. BIST Controller

The BIST Controller orchestrates the operation of the Logic BIST system. It initiates the test process, monitors the progress, and determines the final test result (Pass/Fail). It also has control signals such as "Start BIST" and "BIST Done" to manage the testing process.

This architecture is built directly into the chip, allowing it to test itself without needing external equipment. It can perform testing on its own, making it a cost-effective way to find defects or faults in circuits during manufacturing or normal operation.

IV. Circuit Under Test (CUT) - ISCAS '85 c432

The Logic BIST system is implemented for ISCAS '85 c432 interrupt controller circuit shown in Fig 2. The c432 interrupt controller is a 27-channel system designed to handle interrupt requests from three input buses, labeled A, B, and C. Each of these buses consists of 9 bits (or channels), and a fourth 9-bit bus, labeled E, is used to enable or disable the interrupt requests for the respective channels. The controller prioritizes and acknowledges interrupt requests based on a set of predefined rules.

The priority hierarchy is as follows:

- Bus A has the highest priority, followed by Bus B, and Bus C has the lowest priority.
- Within each bus, a higher index corresponds to a higher priority. For example, A[i] has a higher priority than A[j] if $i > j$.
- If $E[i] = 0$, the corresponding channels A[i], B[i], and C[i] are ignored.

The interrupt controller produces seven outputs: PA, PB, PC, and Chan[3:0]. PA, PB, and PC indicate which of the three buses have acknowledged interrupt requests, while Chan[3:0] specifies the channel of the bus that has been acknowledged. Only the channel with the highest priority from the requesting bus with the highest priority is acknowledged.

For example, consider the following interrupt requests:

- A[4] = 1, A[2] = 1
- B[6] = 1
- C[4] = 1

In this case, the interrupt controller will acknowledge A[4] because it belongs to the highest-priority bus (Bus A), and within Bus A, it has the highest index. Therefore, the output will be:

- PA = 1
- PB = 0
- PC = 0
- Chan[3:0] = 4'b0100

V. Implementation of Logic BIST

In this section, we will explain the system design of the Logic Built-In Self-Test (BIST) implementation. The goal is to create an efficient and reliable testing system that can work automatically within a chip. Key components of the BIST system, such as the Test Pattern Generator (TPG), Phase Shifter, Mode Select, Circuit Under Test (CUT), and the Output Response Analyzer (ORA), are designed to help detect faults accurately. The design focuses on making the testing process efficient, with minimal impact on the overall system, while ensuring all components work together to find and identify problems in integrated circuits. This design forms the basis for the implementation and testing steps that follow.

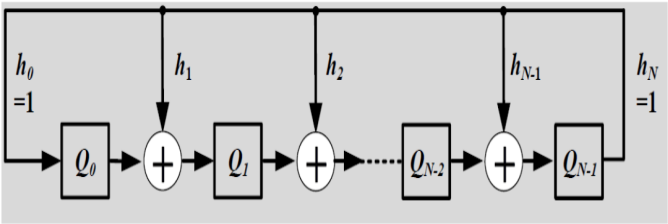

Fig. 3. Type-2 Modular Form LFSR-based TPG

A. Test Pattern Generator (TPG)

The **Test Pattern Generator (TPG)** is an essential part of Built-In Self-Test (BIST) systems. Its main job is to create input test patterns that are used to test the Circuit Under Test (CUT). The TPG aims to produce test inputs that can detect different types of faults in the CUT, such as stuck-at faults, bridging faults, and other combinational errors. A well-designed TPG ensures that the CUT is thoroughly tested without needing exhaustive manual checks. The TPG generates test patterns based on requirements like fault coverage, randomness, and efficiency, and it must work smoothly with other parts of the BIST system.

In most modern BIST systems, the TPG is built using a **Linear Feedback Shift Register (LFSR)**. The LFSR is used because it can efficiently generate a large number of pseudo-random test patterns. It works by shifting bits in a register and feeding the output back into the register, creating a sequence of bits that looks random but is actually determined by an initial seed value. This method allows for long, unpredictable sequences, which are important for detecting faults.

For the implementation, a Type-2 Modular Form Linear-Feedback Shift Register (LFSR) is used for the TPG. Figure 3 shows the TPG based on a Type-2 Modular Form LFSR.

The characteristic polynomial of the LFSR can be expressed as:

$$P(x) = \sum_{i=0}^{N} h_i x^i$$

Where:

- $h_i = 1$ if feedback exists at position i, and $h_i = 0$ if there is no feedback at position i.
- $h_0 = 1$ and $h_N = 1$, ensuring that feedback occurs at both ends of the shift register.

N represents the order of the LFSR.

The characteristic matrix M of a Type 2 LFSR is as follows:

$$T = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & h_0 \\ 1 & 0 & 0 & \cdots & 0 & h_1 \\ 0 & 1 & 0 & \cdots & 0 & h_2 \\ 0 & 0 & 1 & \cdots & 0 & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & h_{N-2} \\ 0 & 0 & 0 & \cdots & 1 & h_{N-1} \end{pmatrix}$$

The next state Q^+ of the LFSR is obtained by applying the transformation matrix T to the current state Q, i.e.,

$$Q^+ = TQ$$

, where Q^+ and Q are row matrices representing the next state and current state of the flip-flops in the LFSR respectively.

So, if Q_{seed} is the seed, the initial state of the LFSR, the the state after K cycles is as follows:

$$Q_K^+ = T^K Q_{seed}$$

If the order of the LFSR is N, meaning it has N flip-flops, we want the values of these N bits to cover all possible values from 1 to $2^N - 1$. To achieve this, the characteristic polynomial must be a primitive polynomial.

A primitive polynomial of degree N must satisfy the following conditions:

- It must be irreducible.
- It should have an odd number of terms.
- It should divide $1 + x^{2^N - 1}$; $N > 3$.

The length of the sequence produced by the LFSR is denoted as L_c, and is given by:

$$L_c = 2^N - 1$$

When a primitive polynomial is chosen for the LFSR, the next state after L_c cycles is:

$$Q_{L_c}^+ = T^{L_c} Q_{\text{seed}} = Q_{\text{seed}}$$

This is because an N^{th}-order LFSR returns to its initial state after $2^{N-1} - 1$ cycles. This implies that:

$$T^{L_c} = I$$

and

$$T^{L_c+1} = T$$

The $c432$ circuit has 36 inputs, which implies that we need a Type-2 modular form LFSR of order 36. For this, the chosen primitive polynomial is

$$P(x) = 1 + x^5 + x^{12} + x^{25} + x^{36}$$

```
module tpg_36_1(
    input rst, clk,
    output reg q0, q1, q2, q3, q4, q5, q6, q7,
        q8, q9, q10, q11, q12, q13, q14, q15,
        q16, q17, q18,
    q19, q20, q21, q22, q23, q24, q25, q26,
        q27, q28, q29, q30, q31, q32, q33, q34,
        q35
);
```

Listing 1. Verilog instantiation of Type-2 Modular Form LFSR TPG

- **rst (Reset) Input:**
  Type: `input rst`
  The reset input ensures that the LFSR-based TPG initializes to a known state, the seed .
- **clk (Clock) Input:**
  Type: `input clk`
  This is the clock signal that synchronizes the sequential logic, driving the state transitions of the LFSR.
- **q0 to q35 (LFSR State Bits) Outputs:**
  Type: `output reg q0, q1, q2, ..., q35`
  These are the 36 output bits representing the state of the LFSR. Each output corresponds to one flip-flop in the LFSR, and its value is updated every clock cycle to reflect the next state.

The next state of each flip-flop ($q0, q1, \ldots, q35$) is determined using the companion matrix T, corresponding to the characteristic polynomial employed, along with the next state equation.

The LFSR has some limitations due to its pattern length and the way its flip-flops depend on each other. The length of the sequence generated by an LFSR is $2^N - 1$, which can lead to very long test times, especially for larger LFSRs. Additionally, LFSRs have structural dependency where the next state of each flip-flop depends on the current state of its neighboring flip-flops. The next state of the flip-flop is the current state of the neighbouring flip-flop if feedback does not exist. This dependency limits the randomness of the sequence and may reduce the effectiveness of the test pattern generator (TPG) in finding faults.

B. Phase Shifter

The test length L_c can become very large, especially for circuits with many inputs. So, we test the circuit with a limited number of cases, to improve test time. When only a limited number of test cases are used, the structural dependency in LFSRs may cause many faults to go undetected. To address this, a phase shifter can be employed to eliminate the structural dependency, enhancing the randomness of the sequence. This improvement helps generate more effective test patterns, making it possible to detect faults more efficiently and reduce the overall test time.

To implement a phase shift, we need to determine the tap points that need to be added (XORed) to achieve the desired phase shift. Suppose we want to shift the output of the k^{th} flip-flop of the LFSR-based TPG by n cycles. The tap points can be represented by the matrix

$$X = BT^n$$

where T is the companion matrix of the LFSR used in the TPG, and B is a row vector with a '1' at the k^{th} position (corresponding to the k^{th} flip-flop) and '0' at all other positions. To achieve the phase shift, we XOR the outputs of the flip-flops corresponding to the positions indicated by the tap points ('1') in matrix X. This ensures that the output of the k^{th} flip-flop is shifted by n cycles.

In order to find the tap points for achieving the desired phase shift, a python script is used as shown below:

```python
import numpy as np

def create_lfsr_matrix():
    T = np.zeros((36, 36), dtype=int)
    feedback_positions = [0, 5, 12]
    for pos in feedback_positions:
        T[pos, 35] = 1
    for i in range(1, 36):
        T[i, i - 1] = 1
    return T

def xor_matrix_multiply(B, T):
    result = np.zeros(B.shape, dtype=int)
    for i in range(B.shape[0]):
        for j in range(T.shape[1]):
            xor_sum = 0
            for k in range(T.shape[0]):
                xor_sum ^= (B[i, k] & T[k, j])
            result[i, j] = xor_sum
    return result

def create_identity_matrix():
    return np.eye(36, dtype=int)

def matrix_exponentiation(T, n):
    result = create_identity_matrix()
    if n >= 1:
        result = T
    while n > 1:
        result = xor_matrix_multiply(result,
            result)
        n //= 2
    return result

T = create_lfsr_matrix()
np.set_printoptions(threshold=np.inf)
print("Matrix T is:")
print(T)

n = int(input("Enter the power n (must be a
    power of 2): "))
B = np.zeros((1, 36), dtype=int)
position_of_one = int(input("Enter the position
    of 1 (0-35): "))
B[0, position_of_one] = 1

T_power_n = matrix_exponentiation(T, n)
result = xor_matrix_multiply(B, T_power_n)

print("The result of BT^n is:")
print(result)

positions_of_ones = np.where(result[0] == 1)[0]
print("Positions with value 1 are:",
    positions_of_ones)
```

Listing 2. Python Script for finding tap-points for phase shifter

- **position_of_one**
  Represents the position of the bit we want to phase shift.
- **n**: This represents the phase shift, i.e., the number of clock cycles by which we want the bit to be shifted. For easy computation, this is always taken as a power of 2.
- **create_lfsr_matrix**
  Generates the companion matrix T of the Type-2 LFSR based TPG described in the previous section.

- **xor_matrix_multiply**
  Multiplies two matrices in $GF(2)$.
- **matrix_exponentiation**
  Computes T^n, taking n to be a power of 2.

Using the Python script, the tap points are calculated for each input. In this implementation, the shift values for each bit are as follows:

The first bit is shifted by 0 cycles, the second bit is shifted by 8^1 cycles, the third bit is shifted by 8^2 cycles, continuing this pattern, the ninth bit is shifted by 8^8 cycles. These bits correspond to Bus A.

The next group of bits, from the tenth to the eighteenth, are shifted by the following amounts: the tenth bit is shifted by 2, the eleventh bit by 2×8^1, the twelfth bit by 2×8^2, continuing this pattern, the eighteenth bit is shifted by 2×8^8. These bits correspond to Bus B.

The bits from the nineteenth to the twenty-seventh are shifted by the following amounts: the nineteenth bit is shifted by 4, the twentieth bit by 4×8^1, the twenty-first bit by 4×8^2, continuing this pattern, the twenty-seventh bit is shifted by 4×8^8. These bits correspond to Bus C.

Finally, the bits from the twenty-eighth to the thirty-sixth are shifted by the following amounts: the twenty-eighth bit is shifted by 8, the twenty-ninth bit by 8×8^1, the thirtieth bit by 8×8^2, continuing this pattern, the thirty-sixth bit is shifted by 8×8^8. These bits correspond to the Enable signals.

The phase-shifted module implemented in Verilog is shown below:

```
module phaseshifter(
    input q0, q1, q2, q3, q4, q5, q6, q7, q8,
        q9, q10, q11, q12, q13, q14, q15, q16,
        q17, q18,
    q19, q20, q21, q22, q23, q24, q25, q26,
        q27, q28, q29, q30, q31, q32, q33, q34,
        q35,
    output N1, N4, N8, N11, N14, N17, N21, N24,
        N27, N30, N34, N37, N40, N43, N47, N50,
        N53, N56, N60, N63,
    N66, N69, N73, N76, N79, N82, N86, N89,
        N92, N95, N99, N102, N105, N108, N112,
        N115
);
```

- **q0 to q35 Inputs:**
  Type: `input q0, q1, q2, ..., q35`
  36-bit input from LFSR-based TPG.
- **N1, N4, N8 , . . . , N112, N115 outputs:**
  Type: `output N1, N4, N8 , ..., N112, N115`
  36-bit phase shifted version of the the sequence from TPG. This will go as input to Circuit-Under-Test in test mode.

For instance, the 20th input corresponding to $N63$ is phase-shifted by 32 cycles. By setting $n = 32$ and position_of_one $=$ 19, as the counting starts from 0, the resulting output is shown in Figure 4.

The tap points are 11, 18, 23, and 35. Therefore, the assignment

```
Enter the power n (must be a power of 2): 32
Enter the position of 1 (0-35): 19
The result of BT^n is:
[[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1]]
Positions with value 1 are: [11 18 23 35]
```

Fig. 4. Python Output of Phase Shifter Script

```
assign N63 = q11^q18^q23^q35;
```

is done to generate N63.

C. Mode Select Circuit

The Mode Select Circuit receives both the normal input from the Circuit-Under-Test (CUT) and the input from the TPG. When the circuit is in test mode, it passes the TPG input; otherwise, it passes the normal input.

The verilog implementation of Mode Select Circuit is shown below:

```
module mode(
    input wire BIST,
    output wire N1_sel, N4_sel, N8_sel,
        N11_sel, N14_sel, N17_sel, N21_sel,
        N24_sel, N27_sel, N30_sel,
                N34_sel, N37_sel, N40_sel,
                    N43_sel, N47_sel, N50_sel,
                    N53_sel, N56_sel, N60_sel,
                    N63_sel,
                N66_sel, N69_sel, N73_sel,
                    N76_sel, N79_sel, N82_sel,
                    N86_sel, N89_sel, N92_sel,
                    N95_sel,
                N99_sel, N102_sel, N105_sel,
                    N108_sel, N112_sel,
                    N115_sel,
    input wire N1, N4, N8, N11, N14, N17, N21,
        N24, N27, N30,
                N34, N37, N40, N43, N47, N50,
                    N53, N56, N60, N63,
                N66, N69, N73, N76, N79, N82,
                    N86, N89, N92, N95,
                N99, N102, N105, N108, N112,
                    N115,
    input wire N1_1, N4_1, N8_1, N11_1, N14_1,
        N17_1, N21_1, N24_1, N27_1, N30_1,
                N34_1, N37_1, N40_1, N43_1,
                    N47_1, N50_1, N53_1, N56_1,
                    N60_1, N63_1,
                N66_1, N69_1, N73_1, N76_1,
                    N79_1, N82_1, N86_1, N89_1,
                    N92_1, N95_1,
                N99_1, N102_1, N105_1, N108_1,
                    N112_1, N115_1
);
    // Assign values to _sel based on the value
        of BIST
    assign x_sel   = (BIST) ? x_1   : x;
    // In Code, there is a separe line for each
        input, i.e, x is replaced for each
        input, x is just a placeholder.

endmodule
```

Listing 3. Verilog Instantiation of mode select circuit

- **BIST Input:**
  Decides the mode of operation of the CUT. When BIST=0, the circuit undergoes normal operation. When BIST=1, it is in test mode.
- **Inputs N1, N4, . . . , N115:**
  Normal input of the CUT.
- **Inputs N1_1, N4_1, . . . , N115_1:**
  TPG input.
- **Inputs N1_sel, N4_sel, . . . , N115_sel:**
  Selected output based on the BIST input. N1, N4 , . . . , N115 is selected if BIST = 0 (Normal Operation). N1_1, N4_1, . . . , N115_1 is selected if BIST = 1 (Test Mode).

D. Circuit-Under-Test

It takes in the selected input from Mode Select module and produces the 9 outputs of the CUT, i.e, ISCAS '85 c432.

```
module c432 (
    N1, N4, N8, N11, N14, N17, N21, N24, N27,
        N30,
    N34, N37, N40, N43, N47, N50, N53, N56,
        N60, N63,
    N66, N69, N73, N76, N79, N82, N86, N89,
        N92, N95,
    N99, N102, N105, N108, N112, N115, N223,
        N329, N370, N421,
    N430, N431, N432
);
```

Listing 4. c432 Module Definition

- **Inputs N1, N4, . . . , N115:**
  36-bit input to c432 benchmark circuit. This is selected output of mode circuit.
- **Output N223, N329, . . . , N432:**
  7-bit output of c432 benchmark circuit.

E. Output Response Analyzer (ORA)

The **Output Response Analyzer (ORA)** is an essential component in the Logic (BIST) implementation. Its role is to generate a compact form of the output bits obtained from the circuit under test. This compact representation is called a **signature**. The generation of signature is important in order to reduce the memory required to store the correct outputs for all the test patterns. For example, in the case of c432 circuit, 7 output bits are generated. However, we generate a 4-bit signature from it. This reduction does lead to issues such as aliasing but it is important to balance between memory requirement and probability of aliasing.

1) Serial Input Signature Register(SISR)

The SISR is a circuit used to produce signature from serial input bits. Its structure is similar to a Type-2 LFSR but it has an additional input as shown in figure 5. The input bits are given from the leftmost input. Suppose if there are N input bits to the SISR and it produces an M-bit signature, the M bits of the SISR are assigned certain initial value and then after N cycles, the value of those M bits is the final signature for that particular input and output test pattern. In this way, the SISR helps in space compaction of bits.

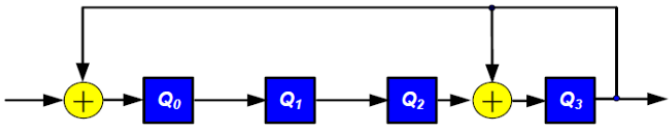

Fig. 5. SISR ciruit

2) Multiple Input Signature Register(MISR)

Multiple Input Signature Register (MISR) is another signature generation circuit. It differs from SISR as it takes multiple outputs patterns instead of a single output pattern to generate the signature as shown in the figure 6. It allows the ORA to efficiently compress large amounts of output data into a small, fixed-size signature that still provides sufficient information to detect faults. In the circuit shown in figure 6, the flip-flops are initially assigned certain bit values. After this, 4 consecutive output patterns having N bits each are given as inputs from the circuit under test as inputs and after N cycles the bits stored in the 4 flip-flops gives us the signature.

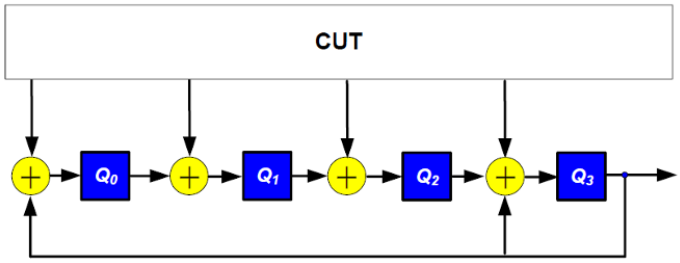

Fig. 6. MISR circuit

3) Advantages of MISR over SISR

The use of MISR as an ORA has several advantages over SISR:

- **Parallelism and Efficiency**: MISR handles multiple data inputs simultaneously, which allows parallel compression of test responses from multiple outputs. This reduces the overall test time significantly, especially in complex circuits with many outputs. Whereas SISR processes only one input at a time which has a higher latency.
- **Scalability**: MISR can be scaled for circuits that have many outputs easily however it is not possible to scale an SISR circuit.
- **Fault Detection Capability**: MISR provides a much more robust fault coverage compared to SISR since it compresses responses from multiple outputs into a signature while maintaining sensitivity to various fault patterns.

4) Implementation of MISR in Verilog

The following Verilog code implements the ORA using a 4-bit MISR. It processes the outputs from the Circuit-Under-Test (CUT) and generates a 4-bit signature.

```
always @(posedge clk1)
begin
    if(count == 7)
    begin
        count = -1;
        q <= 4'b1101;

        s[0] = q[0];
        s[1] = q[1];
        s[2] = q[2];
        s[3] = q[3];
    end
    else
    begin
        q[0] <= N1[count] ^ q[3];
        q[1] <= N2[count] ^ q[0];
        q[2] <= N3[count] ^ q[1];
        q[3] <= N4[count] ^ q[2] ^ q[3];
    end
    count = count + 1;
end
```

Listing 5. Verilog Code for ORA Using MISR

In this code, N1, N2, N3 and N4 are the 4 input patterns and q is the MISR bits. Hence, the value of the bits stored in the MISR are updated each cycle by performing XOR operations and we get the final signature after 7 cycles. Note that the clock used for MISR is different from the clock used for giving inputs.

F. BIST Controller

The BIST Controller is responsible for checking whether the signature generated for an input test pattern is correct or not. It contains a memory which stores the correct signature for each input pattern. It compares the generated signature with the corresponding stored signature and based on that shows whether the BIST is pass or fail.

The following Verilog code shows the implementation of BIST controller in Verilog.

```
initial begin
    memory[0] =
    ↪ {36'b000000000000000000000000000000000000,
    ↪ 4'b0000};
    memory[1] =
    ↪ {36'b011110100111011110001010000101000000,
    ↪ 4'b1000};
    memory[2] =
    ↪ {36'b101011000111100100001110010111101101,
    ↪ 4'b1000};
    memory[3] =
    ↪ {36'b011011010101100000101010000101101000,
    ↪ 4'b0011};
    memory[4] =
    ↪ {36'b010110100001111100110101000110111000,
    ↪ 4'b0111};
    memory[5] =
    ↪ {36'b101001000100111000101001100010000000,
    ↪ 4'b1100};
    \\ Other signatures are also stored
    ↪ here

    end

    always @(posedge clk)
    begin
        found = 0;
        for (i = 0; i < 63; i = i + 1)
        begin
            if (N_new == memory[i][39:4])
            begin
                memory_N = memory[i][39:4];
                memory_s = memory[i][3:0];
                if (s == memory[i][3:0])
                begin
                    found = 1;
                end
            end
        end
    end
```

Listing 6. BIST Controller Verilog Code

In the above code, first, the correct signature corresponding to each input test pattern is stored is a memory. Next, the generated signature(s) and the stored signature are compared. If they are equal, 'found' is assigned to 1 and it is a pass case else it is fail and 'found' is assigned the value 0. Note that, we have not stored the signature for all possible input test patterns. We have only stored for few intial test patterns in order to check the functioning of the BIST controller.

VI. Simulation Results

To validate the functionality of the Built-In Self-Test (BIST) implementation, it is essential to inject stuck-at faults into the c432 benchmark circuit. This can be achieved using the 'force' function. However, the 'force' function operates only on registers and not on wires. Therefore, the structural Verilog code is modified into a behavioral representation. To automate the fault injection process, the Python code shown below is used:

```
search_text="//Marker"
replace_text = ""

n = int(input("Enter number of stuck at faults
    you want to introduce in the test circuit:
    "))
for _ in range(n):
    node = input("Enter node name where you
        want to introduce stuck at fault (Names
        of all nodes given in comment): ")
    bit = int(input("Enter whether you want
        Stuck-at-1 or Stuck-at-0 fault at that
        node: "))
    if bit == 1 or bit == 0:
        # Append the formatted string with
            "force" before user_choice
        replace_text += f'force {node}
            ={bit};\n'
    else:
        print("Invalid value")

print(replace_text);
fp1 = open("c432_1.v",'r')
fp2 = open("c432_sa.v",'w')
data = fp1.read()
data = data.replace(search_text,replace_text)
fp2.write(data)
```

Listing 7. Python script for intjecting stuck-at faults

The program begins by prompting the user to input the number of stuck-at faults they wish to inject into the circuit. For each fault, the program asks for the name of the node and whether a stuck-at-0 or stuck-at-1 fault should be introduced. Based on the inputs, the 'replace_text' string is updated with the appropriate command in the format 'force node_name = node_value;'. After collecting all the faults, the program modifies the c432 benchmark circuit. It replaces a predefined marker, '\\Marker', placed inside the 'initial' block with the generated 'replace_text' containing the force function calls, thereby injecting the specified stuck-at faults into the circuit and making a temporary copy of the c432 circuit with the faults injected, which is used for testing.

```
initial begin
    //Marker
end
```

Listing 8. Initial block with Marker in c432

For example, suppose we give n=2 as the number of stuck-at-faults, we want to inject. Suppose, we want to inject stuck-at-0 fault at node N329 and stuck-at-1 fault at node N360. Fig 7 shows the output of the python script for the above described faults.

```
Enter number of stuck at faults you want to introduce in the test circuit: 2
Enter node name where you want to introduce stuck at fault (Names of all nodes given in comment): N329
Enter whether you want Stuck-at-1 or Stuck-at-0 fault at that node: 0
Enter node name where you want to introduce stuck at fault (Names of all nodes given in comment): N360
Enter whether you want Stuck-at-1 or Stuck-at-0 fault at that node: 1
force N329 =0;
force N360 =1;
```

Fig. 7. Output of Python Script for Injecting Faults

The modified initial block with the faults injected is as follows:

```
initial begin
force N329 =0;
force N360 =1;
end
```

Listing 9. Modified initial block in c432 with stuck-at-faults injected

When the process of BIST is started, the Test Pattern Generator(TPG) generates the input test pattern. Next, the phase shifter shifts the test patterns generated by the TPG. Now, these test patterns are given to the Circuit Under Test(CUT) and it generates 7 bits as output. Next, the signature is generated for each test pattern using the MISR-based Output Response Analyzer. Lastly, the BIST Controller generates the 'found' bit which shows whether the BIST for that particular input pattern is pass or fail.

A. Test Pattern Generator(TPG)

Figure 7 shows the 36-bit input pattern q0 to q35 produced by the Test Pattern Generator (TPG) by using a Type-2 based LFSR. A new input pattern is generated every cycle and in this way, it generates all the 2^{36} test patterns. Also, we can notice from the generated pattern that there are a lot of dependencies present between the different input patterns which prevents us from detecting certain faults.

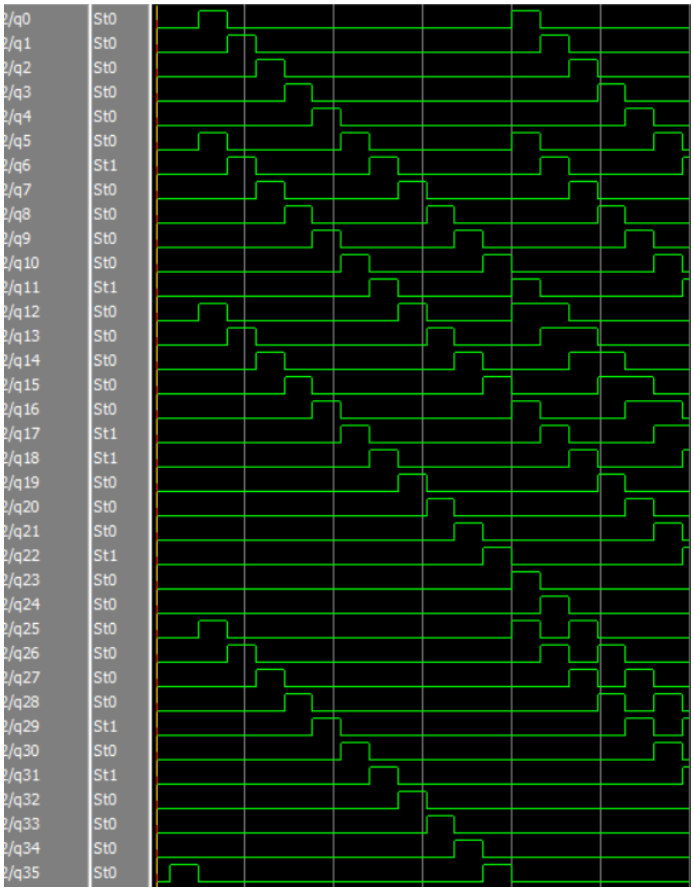

Fig. 8. Input Pattern Generated by TPG

B. Phase Shifter

Figure 8 shows the input test pattern generated after passing the original test pattern through the phase shifter. It can be noticed that the randomness of the input patterns has increased significantly and hence there are no or negligible dependencies between consecutive test patterns. This helps in increasing the fault coverage.

C. Circuit Under Test(CUT)

Figure 9 shows the outputs generated by the c432 Ciruit Under Test(CUT) on giving the input test pattern generated by the phase shifter. The output of this circuit consists of 7 bits.

D. Output Response Analyzer

Figure 10 shows the 4 patterns, which are basically the outputs produced by the circuit under test, that are given to the MISR as inputs to generate the signature. Figure 11 shows the 4-bit signature generated(s) by the MISR for each input test pattern. Note that, the signature for an input test pattern is displayed in the next clock cycle as 1 clock cycle is required to generate the signature.

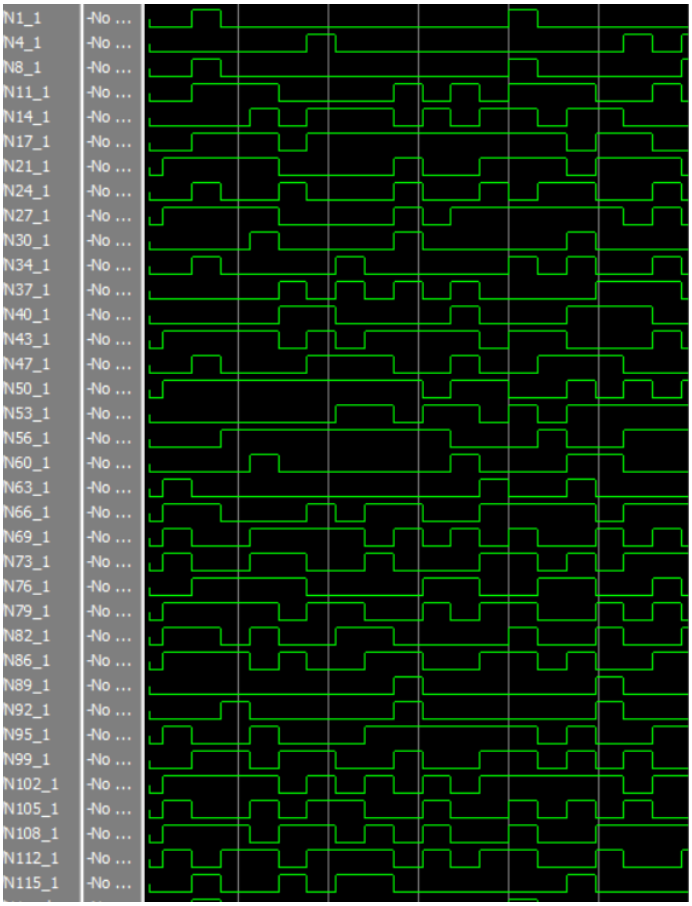

Fig. 9. Phase Shifted Input Test Pattern

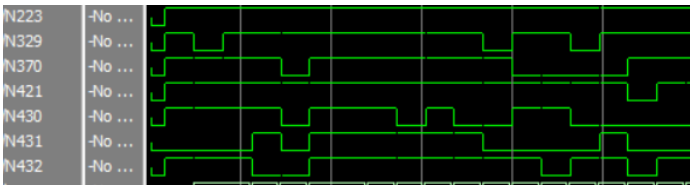

Fig. 10. Output generated by the c432 circuit

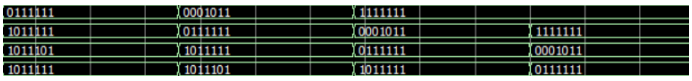

Fig. 11. Inputs to the MISR used as ORA

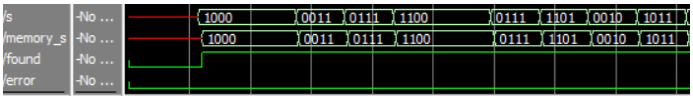

Fig. 12. Generated signature(s), Correct signature(memory_s), Found bit and error bit in absence of any stuck-at faults

E. BIST Controller

The correct signature for a particular test pattern is stored in a memory and it is compared with the generated signature to check if there any fault or not. Figure 11 shows the

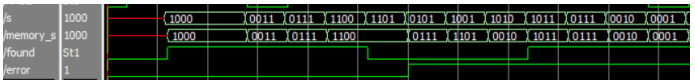

Fig. 13. Generated signature(s), Correct signature(memory_s), Found bit and error bit in presence of stuck-at faults

generated signature(s), correct signature which is stored in the memory(memory_s), the found bit and the error bit when there is no stuck-at fault present in the circuit. The 'found' bit shows that whether the generated signature is same as the correct signature(1) or not(0) and the error bit shows whether we have detected any error in the BIST testing process or not. We can notice that the found bit is always 1 for fault-free case as the generated signature is always correct. Also, the error bit is always 0 which shows we have not encountered any error in the testing. Figure 12 shows the same set of outputs as figure 11 and for the same sequence of test patterns but for a circuit which has certain stuck-at faults. We can notice that the generated signature is not same as the correct signature for one of the input test pattern due to which the found bit is 0 for 4 clock cycles due to which, the error bit also becomes 1 showing that one or more errors are present in the circuit under test. The output produced is wrong only for first of the 4 test patterns, however as that output is used to generate signature for itself as well the next 3 test patterns, we are getting wrong signature for 4 cycles instead of a single cycle.

0.8em

VII. Problems Faced and Solution

1) **Problem -** The input test pattern that was being generated by the TPG has many dependencies which prevents us from detecting certain faults.
Solution - We added a Phase Shifter which can shift the test patterns generated by the TPG and can generate a more random test pattern sequence.

2) **Problem -** Initially, we had used SISR in the ORA in order to generate signatures. However, SISR has cetain disadvantes as highlighted earlier.
Solution - We changed the ORA from SISR to MISR as MISR is a better and more reliable method to produce signatures.

3) **Problem -** The circuit under test has 36 input bits and hence 2^{36} input test patterns. Also, each one of them will have a 4-bit signature. But the storage of the signatures for such a large number of input patterns will require a very high amount of memory, which is not feasible.
Solution - In order to resolve this problem, we can store the signatures for only the initial few test patterns. The Phase shifter implemented ensures that the test patterns used, whose signatures are stores are not in anyway dependent, thereby ensuring sure that fault coverage is high. Through these test patterns, it is possible to verify the working of the BIST circuitry.

VIII. Limitations

- Logic BIST implemetation has to be designed for a particular circuit as the order of LFSR-based TPG depends on number of inputs of the CUT , order of LFSR-based ORA depends on the number of outputs of the CUT, signatures to be stored in memory in BIST controller depends on the logic or function of the CUT. The implementation done in the project works only for the c432 benchmark circuit. Adapting to a different circuit requires changing TPG, ORA, signatures stored in memory in BIST controller as well as the phase shift to be provided to each input.

IX. Instructions to Run Code

- **python3 saf.py** : Runs python script which inject faults. First, enter number of stuck-at-faults to introduce in ISCAS '85 c432 circuit. For fault-free simulation, give 0. Otherwise, give some positive value. Then, give the node names where stuck-at-faults need to be injected, with the corresponding stuck-at-value (stuck-at-0 or stuck-at-1). This will create a temporary copy, **c432_sa.v** of the c432 circuit with the faults injected.
- Ensure all Verilog files are available in the working directory. Use a Verilog simulator such as **ModelSim** (Preferred). Compile and simulate the **c432_tb_2.v** file. Check and Verify Output with and without introducing stuck-at-faults. If Error becomes 1, then it means the Logic BIST has detected the fault. Else, it means the fault is not detected. Here, either the circuit is a good circuit, or the faults injected are not detectable (Test Escapes).

X. Individual Contribution

- **Rupak Antani (2022102045):**Implementation- Output Response Analyzer, Test Pattern Generator, Report- Output Response Analyzer, BIST Controller, Simulation Results
- **Abhinav S (2022102037):** Implementation - Fault Injection , Phase Shifter (Python and Verilog), Mode Select, Report - CUT, Phase Shifter, Test Pattern Generator, Fault Injection
- **Himanshu Gupta (2022102002):** Implementation - BIST Controller, Report - Mode Select, Scope of project, Problems Faced and Solution, Limitations, Formatting
- **Akshat Tiwari (2022102043):** Implementation - Testbench , Wrapper Module, Report - Introduction, Overview of Logic BIST Architecture, Readme, Instructions to run code

Acknowledgment

We would like to express our sincere gratitude to our professor, Usha Gogineni, for imparting invaluable knowledge and guidance throughout the course. The concepts and principles shared in class provided the foundation for this project, and their continued support and encouragement were instrumental in bringing this work to fruition.

References

The ISCAS '85 c432 benchmark circuit used for implementing the logic BIST architecture is referenced [1][2]. The references for implementation of Logic Built-In Self-Test (BIST) are taken from the VLSI Testing lectures by Prof. James Chien-Mo Li of Nanyang Technological University [3][4]. A research paper published by Janusz Rajski and Jerzy Tyszer is used for phase shifter implementation [5].

1. Circuit Reference (ISCAS-85 C432):- http://web.eecs.umich.edu/~jhayes/iscas.restore/c432.html

2. ISCAS '85 c432 benchmark-Verilog Code:- https://pld.ttu.ee/~maksim/benchmarks/iscas85/verilog/c432.v

3. VLSI Testing Lecture-13 :- http://cc.ee.ntu.edu.tw/~cmli/VLSItesting/lecturenote/13.zip

4. VLSI Testing Lecture-14 :- http://cc.ee.ntu.edu.tw/~cmli/VLSItesting/lecturenote/14.zip

5. Design of Phase Shifters for BIST:- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=670871&tag=1

6. Logic BIST/LFSR (Section 3.4):- https://www.sciencedirect.com/topics/computer-science/build-in-self-test