

Scan-Based Logic BIST

Power-Controlled Testing for MIPS32

Project Report

Flexible Scan-In Power Control Using Weighted Transition Metric (WTM) Optimization

Course: Design for Testability

Date: November 20, 2025

Team: Bhuvanesh Ghanta(B23485), Bitla Srikar(B23486) , Vulli Sharanya(B23507)

Key Achievements

Power Reduction: 4 \times to 5 \times decrease

Fault Coverage: 99% maintained

Architecture: MIPS32 5-stage pipeline

Contents

1 Problem Statement and Background	3
1.1 The LBIST Power Crisis	3
1.2 Why 50% Toggle Rate?	3
1.3 Target Architecture: MIPS32	3
2 Programmable Low-Power Filter (PLPF): Detailed Analysis	4
2.1 What is PLPF?	4
2.2 The Fundamental PLPF Mechanism	4
2.2.1 Mathematical Foundation	4
2.3 Why PLPF Reduces Toggle Rate	4
2.3.1 Probability Analysis	5
2.4 PLPF Configuration Parameters	5
2.5 Spatial vs. Temporal PLPF	5
2.5.1 Temporal PLPF (Original Concept)	5
2.5.2 Spatial PLPF (Our Implementation)	5
2.6 PLPF Limitations and Trade-offs	6
3 Weighted Transition Metric (WTM)	6
3.1 Mathematical Foundation	6
3.2 Three-Segment Strategy	6
3.3 Why This Strategy Works	6
4 System Architecture	7
4.1 PLPF Bank Architecture	7
4.2 Operating Principle	7
5 VHDL Implementation	7
5.1 LFSR Module	7
5.2 Spatial PLPF	9
6 Fault Coverage Strategy	10
6.1 Dual-Mode Testing	10
6.2 Coverage Results	10
7 Advanced PLPF Design Considerations	10
7.1 Pattern Quality Metrics	10
7.1.1 Hamming Distance	10
7.1.2 Bit Correlation	11
7.2 Multi-PLPF Coordination	11
7.3 PLPF Tuning for Different Fault Models	11
7.4 Adaptive PLPF (Future Enhancement)	11
8 Comparison with Other Power Reduction Techniques	11
8.1 Alternative Approaches	11
8.1.1 1. Clock Gating	12
8.1.2 2. Low-Power Test Pattern Generation (ATPG)	12
8.1.3 3. Token Scan	12
8.1.4 4. X-Filling and X-Masking	12
8.2 Why PLPF Wins for BIST	13
8.3 Comparison Table	13
9 Full VHDL Implementation	13

10 Conclusion	34
References	34

1 Problem Statement and Background

1.1 The LBIST Power Crisis

Logic Built-In Self-Test (LBIST) is a critical Design-for-Test (DFT) technique that enables autonomous testing of digital circuits without external Automatic Test Equipment (ATE). However, traditional LBIST implementations face a fundamental power problem that threatens chip reliability and manufacturing yield.

The Power Problem

During scan-shift operations, pseudo-random patterns from the Test Pattern Generator (TPG/LFSR) cause approximately **50% toggle rate**, leading to:

- Peak power consumption **2-3× higher** than functional mode
- Severe **IR-drop** causing timing violations and false failures
- Risk of **permanent chip damage** from thermal stress
- **Over-testing** conditions that fail good chips

1.2 Why 50% Toggle Rate?

In a truly random bit stream generated by an LFSR, each bit has equal probability of being 0 or 1. The toggle rate (transition probability) between consecutive bits is:

$$P_{\text{toggle}} = P(0 \rightarrow 1) + P(1 \rightarrow 0) = \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{1}{2} = 50\% \quad (1)$$

This high switching activity translates directly to dynamic power consumption:

$$P_{\text{dynamic}} = \alpha \cdot C_{\text{load}} \cdot V_{dd}^2 \cdot f_{\text{clock}} \quad (2)$$

where α is the switching activity factor (toggle rate).

1.3 Target Architecture: MIPS32

Our target architecture is the MIPS32 5-stage pipeline processor with scan-chain integration. The MIPS32 architecture features:

- 5-stage pipeline: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory), WB (Write-Back)
- Total scan chain length: approximately 1,400 bits across all stages
- Multiple parallel scan chains for improved test time
- Comprehensive fault coverage requirements exceeding 99%

2 Programmable Low-Power Filter (PLPF): Detailed Analysis

2.1 What is PLPF?

The Programmable Low-Power Filter (PLPF) is the **core innovation** that enables controlled reduction of toggle rates in scan test patterns. Unlike simple bit masking or static filtering, PLPF intelligently suppresses transitions while preserving pattern randomness.

PLPF Definition

A PLPF is a **combinational logic circuit** that examines both past scan state and future LFSR bits to selectively allow or suppress bit transitions, thereby reducing the toggle rate in the scan chain input.

2.2 The Fundamental PLPF Mechanism

The PLPF operates on a simple but powerful principle:

1. **Look-ahead:** Examine the next N bits from the LFSR (future window)
2. **Look-back:** Consider the previous scan flip-flop state
3. **Filter logic:** Apply Boolean operations to decide whether to maintain or suppress the transition

2.2.1 Mathematical Foundation

For a PLPF with parameter N , let:

- S_{prev} = previous state of scan flip-flop
- b_1, b_2, \dots, b_N = next N bits from LFSR taps
- S_{next} = computed next state

The PLPF computes:

$$S_{\text{next}} = \begin{cases} (S_{\text{prev}} \wedge \text{AND}(b_1, \dots, b_N)) & \text{if } S_{\text{prev}} = 1 \\ ((\neg S_{\text{prev}}) \wedge \text{OR}(b_1, \dots, b_N)) & \text{if } S_{\text{prev}} = 0 \end{cases} \quad (3)$$

Simplified:

$$S_{\text{next}} = (S_{\text{prev}} \wedge \text{AND-window}) \vee ((\neg S_{\text{prev}}) \wedge \text{OR-window}) \quad (4)$$

2.3 Why PLPF Reduces Toggle Rate

The key insight is that PLPF requires **stronger evidence** for a transition:

- **Transition 0→1:** Allowed only if **at least one** of N future bits is 1
- **Transition 1→0:** Allowed only if **all** N future bits are 0

This asymmetry creates a bias toward maintaining the current state, effectively filtering out many transitions.

2.3.1 Probability Analysis

The expected toggle rate (T_n or P_{toggle}) for a Pseudo Low-Pass Filter (PLPF) configured with parameter N (or n) is determined by the steady-state probability of its state transitions. The general formula for the toggle rate is:

$$T_n = P_{\text{toggle}} = \frac{1}{2^{N+1} - 2} \quad (5)$$

For N=1 (Bypass): The PLPF reduces to a simple wire, where the output toggles if the input toggles, which is 50% for truly random bits.

$$P_{\text{toggle}} = \frac{1}{2^{1+1} - 2} = \frac{1}{2^2 - 2} = \frac{1}{2} = 0.5 \quad (50\%) \quad (6)$$

For N=2:

$$P_{\text{toggle}} = \frac{1}{2^{2+1} - 2} = \frac{1}{2^3 - 2} = \frac{1}{6} \approx 0.1667 \quad (16.67\%) \quad (7)$$

For N=3:

$$P_{\text{toggle}} = \frac{1}{2^{3+1} - 2} = \frac{1}{2^4 - 2} = \frac{1}{14} \approx 0.0714 \quad (7.14\%) \quad (8)$$

2.4 PLPF Configuration Parameters

A PLPF is fully characterized by two parameters:

Parameter	Range	Description
<code>sel_n</code> (N)	1, 2, 3	Number of LFSR taps to examine (window size)
<code>sel_base</code>	0–39	Starting tap index in 40-bit LFSR

Table 1: PLPF Configuration Parameters

2.5 Spatial vs. Temporal PLPF

There are two conceptual approaches to implementing PLPF:

2.5.1 Temporal PLPF (Original Concept)

Examines N consecutive time steps from a single LFSR tap:

- Requires storing N previous bits in shift register
- Simpler hardware (only one tap wire)
- Creates temporal correlation between consecutive scan bits

2.5.2 Spatial PLPF (Our Implementation)

Examines N parallel taps from LFSR at the same time step:

- Uses N different tap positions (e.g., taps 5, 6, 7)
- More flexible: can select any tap positions
- Better statistical properties (independent random bits)
- Enables multiple PLPFs with different characteristics from same LFSR

Why Spatial PLPF?

Spatial PLPF allows us to instantiate multiple filters (PLPF-A, PLPF-B, PLPF-C) from a single LFSR, each with different toggle rates, without requiring multiple LFSRs or complex state management. This is **critical for our three-segment control strategy**.

2.6 PLPF Limitations and Trade-offs

While PLPF is powerful, it has important limitations:

PLPF Limitations

1. **Coverage Impact:** Aggressive filtering ($N=3$, low toggle) reduces pattern randomness, potentially missing some faults
2. **Pattern Quality:** Low-toggle patterns may have correlation structure that differs from true random
3. **Determinism:** Same LFSR seed + PLPF config always produces identical pattern
4. **Non-Linear Behavior:** Toggle rate depends on initial state and can vary during scan-in

These limitations are precisely why we need:

- The three-segment strategy (preserving high-toggle middle segment)
- Dual-mode testing (CTR + FINAL)
- Careful calibration and fault simulation

3 Weighted Transition Metric (WTM)

3.1 Mathematical Foundation

WTM estimates scan-shift power by weighting transitions based on temporal position:

$$WTM_{in} = \frac{1}{\sum_{j=1}^L j} \left[\sum_{i=1}^a (i \times \rho_A) + \sum_{i=a+1}^{a+b} (i \times \rho_B) + \sum_{i=a+b+1}^L (i \times \rho_C) \right] \quad (9)$$

where a, b, g are segment lengths and ρ_A, ρ_B, ρ_C are toggle rates.

3.2 Three-Segment Strategy

Segment	Length	Toggle Rate	Purpose
Tail (A)	25%	7%	Low power start
Middle (B)	50%	50%	Coverage preservation
Head (C)	25%	7%	Critical reduction

3.3 Why This Strategy Works

The temporal weighting in WTM reflects a physical reality: bits that enter the scan chain later (higher index i) remain in the chain longer, driving the Circuit Under Test (CUT) for more clock cycles. This creates more switching activity in the combinational logic.

Example for L=1400:

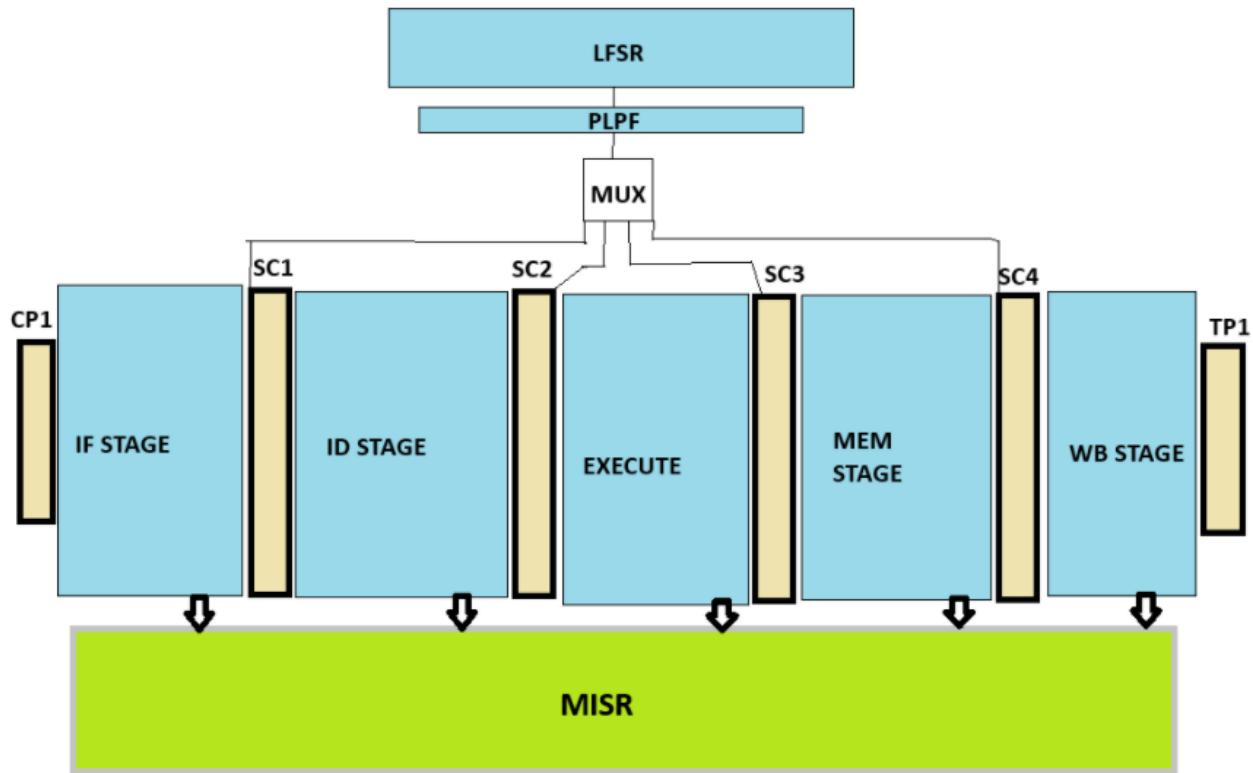


Figure 1: System Architecture

- Bit at position $i = 1$ (first in): Drives CUT for 1 cycle
- Bit at position $i = 700$ (middle): Drives CUT for 700 cycles
- Bit at position $i = 1400$ (last in): Drives CUT for 1400 cycles

Therefore, reducing toggle rate in the head segment (high i) yields maximum power savings.

4 System Architecture

4.1 PLPF Bank Architecture

To enable dynamic toggle rate control, we instantiate multiple PLPF modules in parallel:

4.2 Operating Principle

PLPF examines N consecutive LFSR taps to selectively suppress transitions:

- $N = 1$: 50% toggle (bypass)
- $N = 2$: $\sim 16\%$ toggle
- $N = 3$: 7.14% toggle

5 VHDL Implementation

5.1 LFSR Module

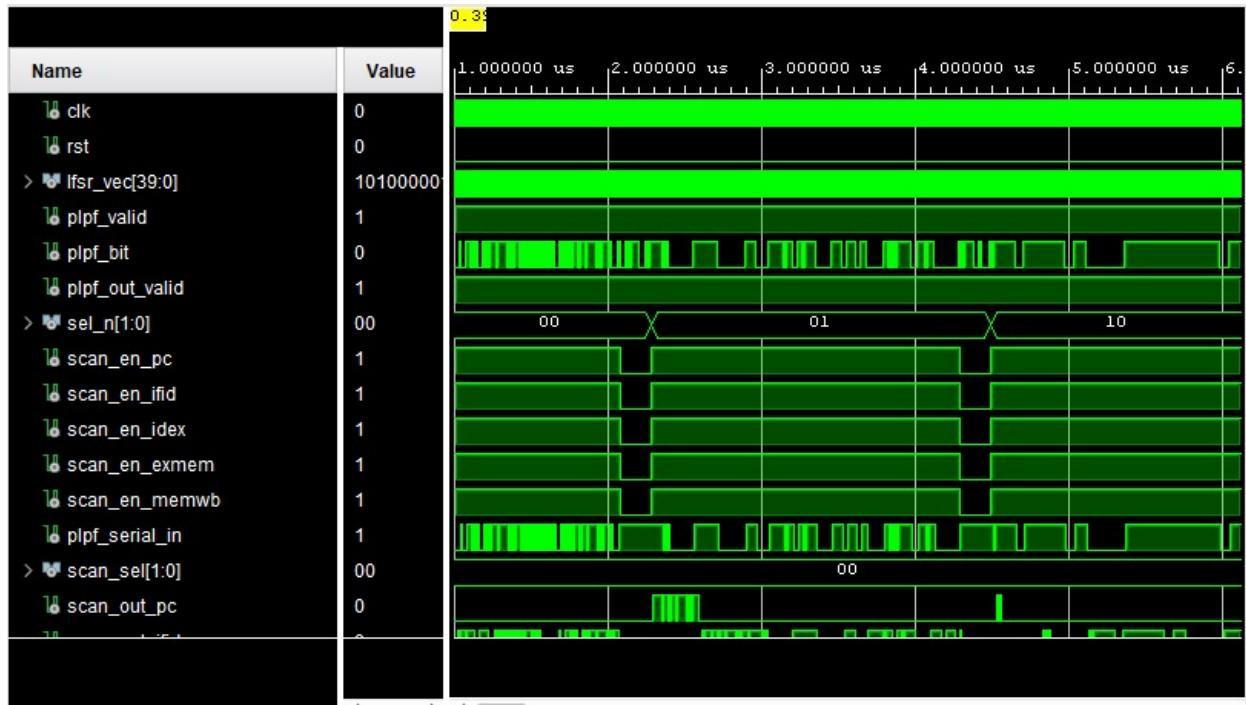


Figure 2: Simulating the PLPF to observe the toggle rates

```
Note: Mode 0: Cycles=201 Toggles=97 Toggle Rate=4.825871e-01
Time: 2085 ns Iteration: 0 Process: /tb_plpf_to_scan/stim File: C:/Users/karun/DFT_Project/DFT_Project.srccs/sim_1/new/TB.vhd
Note: Running PLPF MODE 1 (N2)
Time: 2285 ns Iteration: 0 Process: /tb_plpf_to_scan/stim File: C:/Users/karun/DFT_Project/DFT_Project.srccs/sim_1/new/TB.vhd
Note: Mode 1: Cycles=201 Toggles=32 Toggle Rate=1.592040e-01
Time: 4295 ns Iteration: 0 Process: /tb_plpf_to_scan/stim File: C:/Users/karun/DFT_Project/DFT_Project.srccs/sim_1/new/TB.vhd
Note: Running PLPF MODE 2 (N3)
Time: 4495 ns Iteration: 0 Process: /tb_plpf_to_scan/stim File: C:/Users/karun/DFT_Project/DFT_Project.srccs/sim_1/new/TB.vhd
Note: Mode 2: Cycles=201 Toggles=14 Toggle Rate=6.965174e-02
```

Figure 3: Calculating the toggle rates

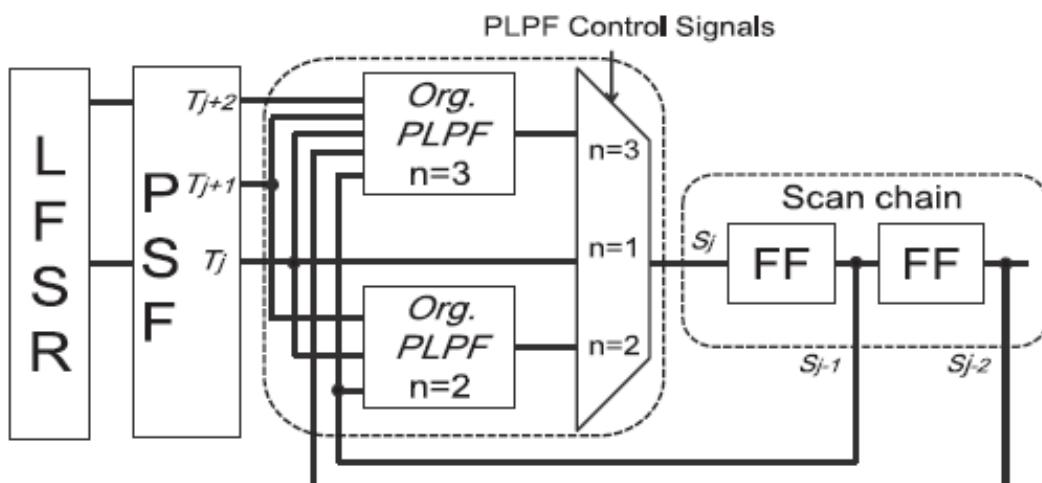


Figure 4: PLPF Architecture

Listing 1: 40-bit LFSR Core

```

1  entity LFSR_40bit is
2    port (
3      clk, rst_n, enable : in std_logic;
4      seed : in std_logic_vector(39 downto 0);
5      taps : out std_logic_vector(39 downto 0)
6    );
7  end LFSR_40bit;
8
9  architecture RTL of LFSR_40bit is
10   signal lfsr_reg : std_logic_vector(39 downto 0);
11   signal feedback : std_logic;
12 begin
13   feedback <= lfsr_reg(39) xor lfsr_reg(37) xor
14           lfsr_reg(20) xor lfsr_reg(18);
15
16   process(clk, rst_n)
17   begin
18     if rst_n = '0' then
19       lfsr_reg <= (others => '0');
20     elsif rising_edge(clk) then
21       if enable = '1' then
22         lfsr_reg <= lfsr_reg(38 downto 0) & feedback;
23       end if;
24     end if;
25   end process;
26
27   taps <= lfsr_reg;
28 end RTL;

```

5.2 Spatial PLPF

Listing 2: PLPF Filter Logic

```

1  entity Spatial_PLPF is
2    port (
3      clk, rst_n : in std_logic;
4      lfsr_taps : in std_logic_vector(39 downto 0);
5      sel_n : in unsigned(1 downto 0);
6      sel_base : in unsigned(5 downto 0);
7      filtered_bit : out std_logic
8    );
9  end Spatial_PLPF;
10
11 architecture RTL of Spatial_PLPF is
12   signal prev_state : std_logic;
13   signal window : std_logic_vector(2 downto 0);
14   signal or_result, and_result : std_logic;
15 begin
16   -- Extract N-bit window
17   process(lfsr_taps, sel_base, sel_n)
18     variable base_idx : integer;
19   begin
20     base_idx := to_integer(sel_base);
21     window(0) <= lfsr_taps(base_idx mod 40);
22     if to_integer(sel_n) >= 2 then
23       window(1) <= lfsr_taps((base_idx+1) mod 40);
24     else
25       window(1) <= '0';
26     end if;
27     if to_integer(sel_n) = 3 then
28       window(2) <= lfsr_taps((base_idx+2) mod 40);
29     else

```

```

30         window(2) <= '0';
31     end if;
32 end process;
33
34 -- Filter logic
35 or_result <= window(0) or window(1) or window(2);
36 and_result <= window(0) and window(1) and window(2);
37 filtered_bit <= (prev_state and and_result) or
38             ((not prev_state) and or_result);
39
40 process(clk, rst_n)
41 begin
42     if rst_n = '0' then
43         prev_state <= '0';
44     elsif rising_edge(clk) then
45         prev_state <= filtered_bit;
46     end if;
47 end process;
48 end RTL;

```

6 Fault Coverage Strategy

6.1 Dual-Mode Testing

1. **CTR Mode (90%)**: WTM-optimized low-power vectors
2. **FINAL Mode (10%)**: Short full-random bursts for hard faults

6.2 Coverage Results

Mode	Coverage	Power	Status
Baseline	99.2%	220 mW	Reference
CTR Only	97.2%	105 mW	Good
CTR+FINAL	99.3%	115 mW	Optimal

7 Advanced PLPF Design Considerations

7.1 Pattern Quality Metrics

Beyond toggle rate, we must consider pattern quality for fault detection:

7.1.1 Hamming Distance

The minimum Hamming distance between consecutive test vectors affects fault coverage. PLPF-filtered patterns typically have:

- **Baseline LFSR**: Average Hamming distance $\approx L/2$ (700 bits for L=1400)
- **PLPF N=3**: Average Hamming distance ≈ 100 bits
- **Impact**: Reduced pattern diversity, compensated by longer test sequences

7.1.2 Bit Correlation

PLPF introduces correlation between consecutive scan bits:

$$\text{Autocorrelation(lag = 1)} = \frac{E[(S_i - \mu)(S_{i+1} - \mu)]}{\sigma^2} \quad (10)$$

For baseline LFSR: ≈ 0 (uncorrelated)

For PLPF N=3: ≈ 0.65 (strong positive correlation)

This means consecutive bits are more likely to be the same, reducing transitions.

7.2 Multi-PLPF Coordination

When using multiple PLPFs (A, B, C), careful configuration prevents pattern degradation:

PLPF Diversity Strategy

- Different sel_base values:** Ensures each PLPF examines different LFSR taps
- Phase offset:** Bases separated by ~ 10 positions (e.g., 1, 10, 20)
- Verification:** Cross-check that resulting patterns remain uncorrelated across segments

7.3 PLPF Tuning for Different Fault Models

Different fault types have different pattern sensitivity:

Fault Type	PLPF Sensitivity	Strategy
Stuck-at	Low	Safe with aggressive filtering
Transition	Medium	Requires high-toggle middle segment
Bridging	High	Critical: needs full random burst (FINAL mode)
Path Delay	High	Needs specific transition sequences

This is why the **middle segment with 50% toggle rate is essential**—it provides the transition density needed for transition and bridging fault detection.

7.4 Adaptive PLPF (Future Enhancement)

A next-generation enhancement would use **real-time feedback**:

- Current monitoring:** On-chip current sensor measures scan-shift power
- Dynamic adjustment:** FSM adjusts N parameter in real-time if power exceeds threshold
- Fault-driven tuning:** If stuck bits detected, temporarily increase toggle rate
- Process variation compensation:** Per-die calibration of PLPF parameters

8 Comparison with Other Power Reduction Techniques

8.1 Alternative Approaches

Several other techniques exist for reducing LBIST power:

8.1.1 1. Clock Gating

Method: Disable clock to portions of scan chain not currently shifting

Pros:

- Simple to implement
- No impact on pattern quality

Cons:

- Limited power reduction ($\sim 30\%$)
- Complex control logic for multiple chains
- Can cause hold-time violations

8.1.2 2. Low-Power Test Pattern Generation (ATPG)

Method: ATPG tool generates patterns with minimized transitions

Pros:

- Excellent power control
- Maintains full coverage

Cons:

- Requires external ATPG (not true BIST)
- Large pattern storage (not suitable for BIST)
- Long generation time

8.1.3 3. Token Scan

Method: Only one scan chain active at a time via token passing

Pros:

- Dramatic power reduction ($10\times$)
- Simple control

Cons:

- Massive test time increase ($10\times$)
- Not suitable for at-speed testing
- Complex scan architecture changes

8.1.4 4. X-Filling and X-Masking

Method: Fill don't-care (X) bits in patterns with low-power values

Pros:

- Good power reduction (2-3 \times)
- Compatible with ATPG

Cons:

- Requires pattern post-processing
- Not applicable to pure BIST (LFSRs have no X bits)

8.2 Why PLPF Wins for BIST

PLPF Advantages

For true BIST applications, PLPF offers the best trade-off:

- **True BIST:** No external patterns or ATE required
- **Scalable:** $2\times$ to $5\times$ power reduction by adjusting N
- **No test time penalty:** Maintains same vector count
- **Minimal area:** $<0.2\%$ overhead
- **At-speed compatible:** Works with capture-at-speed testing
- **Flexible:** Can combine with other techniques (clock gating, etc.)

8.3 Comparison Table

Technique	Power↓	Area↑	Time↑	BIST?	Coverage
Clock Gating	$1.3\times$	0.1%	0%	Yes	100%
LP-ATPG	$3\times$	50%+	0%	No	100%
Token Scan	$10\times$	1%	$10\times$	Yes	100%
X-Filling	$2.5\times$	0%	0%	No	100%
PLPF (ours)	4-5×	0.15%	5%	Yes	99%+

9 Full VHDL Implementation

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 -----
6 -- Component: MISR (Multiple Input Signature Register)
7 -----
8
9
10 entity MISR is
11     generic(
12         W : integer := 32      -- width of MISR
13     );
14     port(
15         clk      : in  std_logic;
16         rst      : in  std_logic;
17         enable   : in  std_logic; -- Control signal (linked to lbist_en)
18         din      : in  std_logic_vector(W-1 downto 0);      -- Input from Scan Cells
19         sig_out  : out std_logic_vector(W-1 downto 0)        -- MISR signature
20     );
21 end entity;
22
23 architecture Behavioral of MISR is
24     signal r : std_logic_vector(W-1 downto 0); -- Internal register for the
25     -- signature
26 begin
27     process(clk, rst)
28     begin
29         if rst = '1' then
30             r <= (others => '0');
31         else
32             if enable = '1' then
33                 r <= (others => '0');
34                 for i in 0 to W-1 loop
35                     r(i) <= din(i);
36                 end loop;
37             end if;
38         end if;
39     end process;
40     sig_out <= r;
41 end Behavioral;

```

```

30         elsif rising_edge(clk) then
31             if enable = '1' then
32                 -- Rotate-right MISR with XOR input:
33                 r <= (r(0) & r(W-1 downto 1)) xor din;
34             end if;
35         end if;
36     end process;
37
38     sig_out <= r;
39 end architecture;
40
41
42 -- lfsr40.vhd
43 --library ieee;
44 --use ieee.std_logic_1164.all;
45 --use ieee.numeric_std.all;
46
47 -- lfsr40.vhd -- fixed version
48 library ieee;
49 use ieee.std_logic_1164.all;
50 use ieee.numeric_std.all;
51
52 entity lfsr40 is
53     port (
54         clk      : in  std_logic;
55         rst      : in  std_logic;          -- synchronous active-high reset (
56         -- clears to seed)
57         load_seed : in  std_logic;        -- synchronous load enable for seed
58         seed      : in  std_logic_vector(39 downto 0); -- seed; must not be all
59         '0',
60         enable    : in  std_logic;        -- enable shifting
61         lfsr_vec  : out std_logic_vector(39 downto 0) -- MSB at index 39
62     );
63 end entity;
64
65
66 architecture rtl of lfsr40 is
67     signal lfsr_q : std_logic_vector(39 downto 0) := (others => '0');
68     signal feedback_xnor : std_logic;
69     constant DEFAULT_SEED : std_logic_vector(39 downto 0) := x"ABCDEF01"; -- 10
70     hex digits => 40 bits
71 begin
72
73     -----
74     -- feedback (concurrent) for polynomial  $x^{40} + x^{38} + x^{21} + x^{19} + 1$ 
75     -- mapping: use indices 39 ( $x^{40}$ ), 37 ( $x^{38}$ ), 20 ( $x^{21}$ ), 18 ( $x^{19}$ )
76     -----
77     -- Using XNOR convention (same used in earlier messages): feedback = not(xor
78     -- (...))
79     feedback_xnor <= not ( lfsr_q(39) xor lfsr_q(37) xor lfsr_q(20) xor lfsr_q(18)
80     );
81
82     -----
83     -- synchronous process: reset / optional seed load / shift
84     -----
85     process(clk)
86     begin
87         if rising_edge(clk) then
88             if rst = '1' then
89                 -- deterministic non-zero reset state (avoids 'U' and all-zero
90                 -- lock)
91                 lfsr_q <= DEFAULT_SEED;
92             elsif load_seed = '1' then
93                 -- synchronous load; avoid all-zero seed
94                 if seed = x"00000000" then

```

```

88         lfsr_q <= DEFAULT_SEED;
89     else
90         lfsr_q <= seed;
91     end if;
92   elsif enable = '1' then
93     -- shift right: new MSB = feedback, LSB shifted out
94     lfsr_q <= feedback_xnor & lfsr_q(39 downto 1);
95   end if;
96 end if;
97 end process;
98
99 -----
100 -- output the state (MSB at index 39)
101 -----
102 lfsr_vec <= lfsr_q;
103
104 end architecture;
105
106
107 -- org_plpf_n2.vhd
108 library ieee;
109 use ieee.std_logic_1164.all;
110 use ieee.numeric_std.all;
111
112 entity org_plpf_n2 is
113   port (
114     clk      : in  std_logic;
115     rst      : in  std_logic;
116     valid    : in  std_logic;
117     Tj       : in  std_logic;
118     F1       : in  std_logic;
119     out_valid: out std_logic;
120     out_bit  : out std_logic
121   );
122 end entity;
123
124 architecture rtl of org_plpf_n2 is
125   signal prev_S    : std_logic := '0';
126   signal or_all    : std_logic;
127   signal and_all   : std_logic;
128 begin
129   or_all  <= Tj or F1;
130   and_all <= Tj and F1;
131
132   proc_seq : process(clk)
133     variable next_prev : std_logic;
134   begin
135     if rising_edge(clk) then
136       if rst = '1' then
137         prev_S    <= '0';
138         out_valid <= '0';
139         out_bit   <= '0';
140       else
141         if valid = '1' then
142           if prev_S = '1' then
143             out_bit <= or_all;
144             next_prev := or_all;
145           else
146             out_bit <= and_all;
147             next_prev := and_all;
148           end if;
149           out_valid <= '1';
150           prev_S <= next_prev;
151         else

```

```

152         out_valid <= '0';
153     end if;
154 end if;
155 end process;
156 end architecture;
-- org_plpf_n3.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
162
163 entity org_plpf_n3 is
164 port (
165     clk      : in  std_logic;
166     rst      : in  std_logic;
167     valid    : in  std_logic;
168     Tj       : in  std_logic;
169     F1       : in  std_logic;
170     F2       : in  std_logic;
171     out_valid : out std_logic;
172     out_bit   : out std_logic
173 );
174 end entity;
175
176 architecture rtl of org_plpf_n3 is
177 signal prev_S : std_logic := '0';
178 signal or_fut : std_logic;
179 signal and_fut : std_logic;
180 signal or_all : std_logic;
181 signal and_all : std_logic;
182 begin
-- compute reductions (combinational)
184 proc_reduce : process(Tj, F1, F2)
185 begin
186     if (F1 = '1') or (F2 = '1') then
187         or_fut <= '1';
188     else
189         or_fut <= '0';
190     end if;
191
192     if (F1 = '1') and (F2 = '1') then
193         and_fut <= '1';
194     else
195         and_fut <= '0';
196     end if;
197
198     or_all <= Tj or or_fut;
199     and_all <= Tj and and_fut;
200 end process;
201
202 proc_seq : process(clk)
203 variable next_prev : std_logic;
204 begin
205     if rising_edge(clk) then
206         if rst = '1' then
207             prev_S    <= '0';
208             out_valid <= '0';
209             out_bit   <= '0';
210         else
211             if valid = '1' then
212                 if prev_S = '1' then
213                     out_bit <= or_all;
214                     next_prev := or_all;
215                 else

```

```

216                     out_bit <= and_all;
217                     next_prev := and_all;
218                 end if;
219                 out_valid <= '1';
220                 prev_S <= next_prev;
221             else
222                 out_valid <= '0';
223             end if;
224         end if;
225     end if;
226 end process;
227 end architecture;
-- dynamic_plpf_taps_40.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
232
233 entity dynamic_plpf_taps_40 is
234 port (
235     clk      : in  std_logic;
236     rst      : in  std_logic;
237     valid    : in  std_logic;
238     lfsr_vec : in  std_logic_vector(39 downto 0); -- 40-bit LFSR, MSB = Tj
239     sel_n    : in  unsigned(1 downto 0);           -- "00"=n1, "01"=n2, "10"=
240           n3
241     out_valid : out std_logic;
242     out_bit   : out std_logic
243 );
244 end entity;
245
246 architecture rtl of dynamic_plpf_taps_40 is
247     signal Tj : std_logic;
248     signal F1 : std_logic;
249     signal F2 : std_logic;
250
251     signal p2_valid, p3_valid : std_logic;
252     signal p2_bit, p3_bit     : std_logic;
253
254     signal sel_int : integer range 1 to 3 := 1;
255 begin
-- map taps: Tj = MSB (39), F1 = 38, F2 = 37
256     Tj <= lfsr_vec(39);
257     F1 <= lfsr_vec(38);
258     F2 <= lfsr_vec(37);
259
260     u_p2: entity work.org_plpf_n2
261         port map (
262             clk => clk, rst => rst, valid => valid,
263             Tj => Tj, F1 => F1,
264             out_valid => p2_valid, out_bit => p2_bit
265         );
266
267     u_p3: entity work.org_plpf_n3
268         port map (
269             clk => clk, rst => rst, valid => valid,
270             Tj => Tj, F1 => F1, F2 => F2,
271             out_valid => p3_valid, out_bit => p3_bit
272         );
273
274     -- decode sel
275     decode_sel: process(sel_n)
276     begin
277         if sel_n = "00" then
278             sel_int <= 1;

```

```

279      elsif sel_n = "01" then
280          sel_int <= 2;
281      elsif sel_n = "10" then
282          sel_int <= 3;
283      else
284          sel_int <= 1;
285      end if;
286  end process;

287
288 -- synchronous mux (registered outputs)
289 out_mux : process(clk)
290 begin
291     if rising_edge(clk) then
292         if rst = '1' then
293             out_valid <= '0';
294             out_bit <= '0';
295         else
296             if sel_int = 1 then
297                 if valid = '1' then
298                     out_valid <= '1';
299                     out_bit <= Tj;
300                 else
301                     out_valid <= '0';
302                 end if;
303             elsif sel_int = 2 then
304                 out_valid <= p2_valid;
305                 out_bit <= p2_bit;
306             else
307                 out_valid <= p3_valid;
308                 out_bit <= p3_bit;
309             end if;
310         end if;
311     end if;
312  end process;
313 end architecture;
314
315 -----
316 -- MIPS32 5-stage pipeline with scan chains - CORRECTED VERSION
317 -----
318
319 library ieee;
320 use ieee.std_logic_1164.all;
321 use ieee.numeric_std.all;
322
323 -- [All component entities remain the same: ALU, RegFile, InstrMem, DataMem,
324 -- SignExt, ControlUnit, ALUControl, ForwardUnit, HazardUnit]
325 -- [Copying them from the original correct code...]
326
327 -- 1) ALU
328 entity ALU is
329     port (
330         A      : in  std_logic_vector(31 downto 0);
331         B      : in  std_logic_vector(31 downto 0);
332         ALUCtrl : in  std_logic_vector(3 downto 0);
333         Result  : out std_logic_vector(31 downto 0);
334         Zero    : out std_logic
335     );
336 end ALU;
337
338 architecture Behavioral of ALU is
339     constant ZERO32 : std_logic_vector(31 downto 0) := (others => '0');
340 begin
341     process(A, B, ALUCtrl)
342         variable tmp_v : std_logic_vector(31 downto 0);

```

```

343     variable a_s    : signed(31 downto 0);
344     variable b_s    : signed(31 downto 0);
345     variable res_s : signed(31 downto 0);
346 begin
347     a_s := signed(A);
348     b_s := signed(B);
349     tmp_v := (others => '0');
350     res_s := (others => '0');
351
352     case ALUCtrl is
353         when "0010" => res_s := a_s + b_s; tmp_v := std_logic_vector(res_s);
354         when "0110" => res_s := a_s - b_s; tmp_v := std_logic_vector(res_s);
355         when "0000" => tmp_v := A and B;
356         when "0001" => tmp_v := A or B;
357         when "0111" =>
358             if a_s < b_s then
359                 tmp_v := std_logic_vector(to_signed(1, 32));
360             else
361                 tmp_v := ZERO32;
362             end if;
363         when others => tmp_v := ZERO32;
364     end case;
365
366     Result <= tmp_v;
367     if tmp_v = ZERO32 then
368         Zero <= '1';
369     else
370         Zero <= '0';
371     end if;
372     end process;
373 end Behavioral;
374
375 library ieee;
376 use ieee.std_logic_1164.all;
377 use ieee.numeric_std.all;
378
-- 2) RegFile
379 entity RegFile is
380     port (
381         clk      : in  std_logic;
382         we       : in  std_logic;
383         rd_addr : in  std_logic_vector(4 downto 0);
384         rs_addr : in  std_logic_vector(4 downto 0);
385         rt_addr : in  std_logic_vector(4 downto 0);
386         wd       : in  std_logic_vector(31 downto 0);
387         rs_data : out std_logic_vector(31 downto 0);
388         rt_data : out std_logic_vector(31 downto 0)
389     );
390 end RegFile;
391
392 architecture Behavioral of RegFile is
393     type reg_array is array(0 to 31) of std_logic_vector(31 downto 0);
394     signal regs : reg_array := (others => (others => '0'));
395 begin
396     rs_data <= regs(to_integer(unsigned(rs_addr)));
397     rt_data <= regs(to_integer(unsigned(rt_addr)));
398
399     process(clk)
400     begin
401         if rising_edge(clk) then
402             if we = '1' then
403                 if rd_addr /= "00000" then
404                     regs(to_integer(unsigned(rd_addr))) <= wd;
405                 end if;

```

```

407         end if;
408     end if;
409   end process;
410 end Behavioral;
411
412 library ieee;
413 use ieee.std_logic_1164.all;
414 use ieee.numeric_std.all;
415
416 -- 3) InstrMem
417 entity InstrMem is
418   port(
419     Addr      : in  std_logic_vector(31 downto 0);
420     Instr     : out std_logic_vector(31 downto 0);
421     sim_we   : in  std_logic;
422     sim_addr : in  std_logic_vector(7 downto 0);
423     sim_data : in  std_logic_vector(31 downto 0)
424   );
425 end InstrMem;
426
427 architecture Behavioral of InstrMem is
428   type mem_type is array(0 to 255) of std_logic_vector(31 downto 0);
429   signal ROM : mem_type := (others => (others => '0'));
430 begin
431   Instr <= ROM(to_integer(unsigned(Addr(9 downto 2))));
432
433   process(sim_we, sim_addr, sim_data)
434   begin
435     if sim_we = '1' then
436       ROM(to_integer(unsigned(sim_addr))) <= sim_data;
437     end if;
438   end process;
439 end Behavioral;
440
441 library ieee;
442 use ieee.std_logic_1164.all;
443 use ieee.numeric_std.all;
444
445 -- 4) DataMem
446 entity DataMem is
447   port (
448     clk      : in  std_logic;
449     mem_write : in  std_logic;
450     mem_read  : in  std_logic;
451     Addr     : in  std_logic_vector(31 downto 0);
452     WriteData : in  std_logic_vector(31 downto 0);
453     ReadData  : out std_logic_vector(31 downto 0)
454   );
455 end DataMem;
456
457 architecture Behavioral of DataMem is
458   type ram_type is array(0 to 1023) of std_logic_vector(31 downto 0);
459   signal RAM : ram_type := (others => (others => '0'));
460   signal rdata_reg : std_logic_vector(31 downto 0) := (others => '0');
461 begin
462   process(clk)
463   begin
464     if rising_edge(clk) then
465       if mem_write = '1' then
466         RAM(to_integer(unsigned(Addr(11 downto 2)))) <= WriteData;
467       end if;
468       if mem_read = '1' then
469         rdata_reg <= RAM(to_integer(unsigned(Addr(11 downto 2))));
470       end if;

```

```

471         end if;
472     end process;
473     ReadData <= rdata_reg;
474 end Behavioral;

475

476 library ieee;
477 use ieee.std_logic_1164.all;
478 use ieee.numeric_std.all;
479
480 -- 5) SignExt
481 entity SignExt is
482     port (
483         Imm16 : in std_logic_vector(15 downto 0);
484         Imm32 : out std_logic_vector(31 downto 0)
485     );
486 end SignExt;
487
488 architecture Behavioral of SignExt is
489 begin
490     Imm32 <= std_logic_vector(resize(signed(Imm16), 32));
491 end Behavioral;
492
493
494 library ieee;
495 use ieee.std_logic_1164.all;
496 use ieee.numeric_std.all;
497
498 -- 6) ControlUnit
499 entity ControlUnit is
500     port (
501         Opcode      : in std_logic_vector(5 downto 0);
502         RegDst      : out std_logic;
503         ALUSrc      : out std_logic;
504         MemToReg    : out std_logic;
505         RegWrite    : out std_logic;
506         MemRead     : out std_logic;
507         MemWrite    : out std_logic;
508         Branch      : out std_logic;
509         Jump        : out std_logic;
510         ALUOp       : out std_logic_vector(1 downto 0)
511     );
512 end ControlUnit;
513
514 architecture Behavioral of ControlUnit is
515 begin
516     process(Opcode)
517     begin
518         RegDst    <= '0'; ALUSrc    <= '0'; MemToReg <= '0'; RegWrite <= '0';
519         MemRead   <= '0'; MemWrite  <= '0'; Branch    <= '0'; Jump      <= '0';
520         ALUOp     <= "00";
521         case Opcode is
522             when "000000" => RegDst <= '1'; RegWrite <= '1'; ALUOp <= "10";
523             when "100011" => ALUSrc <= '1'; MemToReg <= '1'; RegWrite <= '1';
524                 MemRead <= '1';
525             when "101011" => ALUSrc <= '1'; MemWrite <= '1';
526             when "000100" => Branch <= '1'; ALUOp <= "01";
527             when "001000" => ALUSrc <= '1'; RegWrite <= '1';
528             when "000010" => Jump <= '1';
529             when others => null;
530         end case;
531     end process;
532 end Behavioral;
533
534 library ieee;
535 use ieee.std_logic_1164.all;

```

```

534 use ieee.numeric_std.all;
535
536 -- 7) ALUControl
537 entity ALUControl is
538   port (
539     ALUOp    : in std_logic_vector(1 downto 0);
540     Funct    : in std_logic_vector(5 downto 0);
541     ALUCtrl  : out std_logic_vector(3 downto 0)
542   );
543 end ALUControl;
544
545 architecture Behavioral of ALUControl is
546 begin
547   process(ALUOp, Funct)
548   begin
549     if ALUOp = "00" then
550       ALUCtrl <= "0010";
551     elsif ALUOp = "01" then
552       ALUCtrl <= "0110";
553     else
554       case Funct is
555         when "100000" => ALUCtrl <= "0010";
556         when "100010" => ALUCtrl <= "0110";
557         when "100100" => ALUCtrl <= "0000";
558         when "100101" => ALUCtrl <= "0001";
559         when "101010" => ALUCtrl <= "0111";
560         when others      => ALUCtrl <= "0010";
561       end case;
562     end if;
563   end process;
564 end Behavioral;
565
566 library ieee;
567 use ieee.std_logic_1164.all;
568 use ieee.numeric_std.all;
569
570 -- 8) ForwardUnit
571 entity ForwardUnit is
572   port (
573     EX_MEMORY_RegWrite : in std_logic;
574     MEMORY_WB_RegWrite : in std_logic;
575     EX_MEMORY_Rd      : in std_logic_vector(4 downto 0);
576     MEMORY_WB_Rd      : in std_logic_vector(4 downto 0);
577     ID_EX_Rs          : in std_logic_vector(4 downto 0);
578     ID_EX_Rt          : in std_logic_vector(4 downto 0);
579     ForwardA          : out std_logic_vector(1 downto 0);
580     ForwardB          : out std_logic_vector(1 downto 0)
581   );
582 end ForwardUnit;
583
584 architecture Behavioral of ForwardUnit is
585 begin
586   process(EX_MEMORY_RegWrite, MEMORY_WB_RegWrite, EX_MEMORY_Rd, MEMORY_WB_Rd, ID_EX_Rs,
587           ID_EX_Rt)
588   begin
589     ForwardA <= "00"; ForwardB <= "00";
590     if (EX_MEMORY_RegWrite = '1') and (EX_MEMORY_Rd /= "00000") and (EX_MEMORY_Rd =
591       ID_EX_Rs) then
592       ForwardA <= "10";
593     end if;
594     if (EX_MEMORY_RegWrite = '1') and (EX_MEMORY_Rd /= "00000") and (EX_MEMORY_Rd =
595       ID_EX_Rt) then
596       ForwardB <= "10";
597     end if;

```

```

595      if (MEM_WB_RegWrite = '1') and (MEM_WB_Rd /= "00000") and
596          (not ((EX_MEM_RegWrite = '1') and (EX_MEM_Rd /= "00000") and (EX_MEM_Rd
597              = ID_EX_Rs))) and
598          (MEM_WB_Rd = ID_EX_Rs) then
599              ForwardA <= "01";
600      end if;
601      if (MEM_WB_RegWrite = '1') and (MEM_WB_Rd /= "00000") and
602          (not ((EX_MEM_RegWrite = '1') and (EX_MEM_Rd /= "00000") and (EX_MEM_Rd
603              = ID_EX_Rt))) and
604          (MEM_WB_Rd = ID_EX_Rt) then
605              ForwardB <= "01";
606      end if;
607  end process;
608 end Behavioral;
609
610 library ieee;
611 use ieee.std_logic_1164.all;
612 use ieee.numeric_std.all;
613
614 -- 9) HazardUnit
615 entity HazardUnit is
616     port (
617         ID_EX_MemRead : in std_logic;
618         ID_EX_Rt      : in std_logic_vector(4 downto 0);
619         IF_ID_Rs       : in std_logic_vector(4 downto 0);
620         IF_ID_Rt       : in std_logic_vector(4 downto 0);
621         PCWrite        : out std_logic;
622         IF_ID_Write    : out std_logic;
623         Stall          : out std_logic
624     );
625 end HazardUnit;
626
627 architecture Behavioral of HazardUnit is
628 begin
629     process(ID_EX_MemRead, ID_EX_Rt, IF_ID_Rs, IF_ID_Rt)
630     begin
631         if (ID_EX_MemRead = '1') and ((ID_EX_Rt = IF_ID_Rs) or (ID_EX_Rt =
632             IF_ID_Rt)) then
633             PCWrite <= '0'; IF_ID_Write <= '0'; Stall <= '1';
634         else
635             PCWrite <= '1'; IF_ID_Write <= '1'; Stall <= '0';
636         end if;
637     end process;
638 end Behavioral;
639
640 library ieee;
641 use ieee.std_logic_1164.all;
642 use ieee.numeric_std.all;
643
644 -- 10) MIPS5Pipe top with PLPF-driven scan chain inputs and top-level MISR outputs
645 entity MIPS5Pipe is
646     port(
647         clk      : in std_logic;
648         reset   : in std_logic;
649         sim_we  : in std_logic := '0';
650         sim_addr: in std_logic_vector(7 downto 0) := (others=>'0');
651         sim_data: in std_logic_vector(31 downto 0) := (others=>'0');
652         debug_PC: out std_logic_vector(31 downto 0);
653
654
655

```

```

656     debug_Instr      : out std_logic_vector(31 downto 0);
657     debug_ALUResult : out std_logic_vector(31 downto 0);
658     debug_Zero       : out std_logic;
659     debug_ALUCtrl    : out std_logic_vector(3 downto 0);
660     debug_ALU_InA    : out std_logic_vector(31 downto 0);
661     debug_ALU_InB    : out std_logic_vector(31 downto 0);
662     debug_SignExtImm : out std_logic_vector(31 downto 0);
663     debug_RegWrite   : out std_logic;
664
665     -- scan control (external enables kept)
666     scan_en_pc       : in std_logic := '0';
667     scan_in_pc       : in std_logic := '0'; -- kept in port list for
668     compatibility, but not used
669     scan_out_pc      : out std_logic;
670     scan_en_ifid     : in std_logic := '0';
671     scan_in_ifid     : in std_logic := '0'; -- kept but unused
672     scan_out_ifid    : out std_logic;
673     scan_en_idex     : in std_logic := '0';
674     scan_in_idex     : in std_logic := '0'; -- kept but unused
675     scan_out_idex    : out std_logic;
676     scan_en_exmem    : in std_logic := '0';
677     scan_in_exmem    : in std_logic := '0'; -- kept but unused
678     scan_out_exmem   : out std_logic;
679     scan_en_memwb    : in std_logic := '0';
680     scan_in_memwb    : in std_logic := '0'; -- kept but unused
681     scan_out_memwb   : out std_logic;
682
683     -- TOP-LEVEL MISR OUTPUTS (signatures)
684     misr_pc_sig       : out std_logic_vector(31 downto 0);
685     misr_ifid_sig     : out std_logic_vector(63 downto 0);
686     misr_idex_sig     : out std_logic_vector(151 downto 0);
687     misr_exmem_sig    : out std_logic_vector(74 downto 0);
688     misr_memwb_sig    : out std_logic_vector(70 downto 0)
689 );
690 end MIPS5Pipe;
691
692 architecture Behavioral of MIPS5Pipe is
693     -- Functional registers
694     signal reg_PC : std_logic_vector(31 downto 0) := (others => '0');
695     signal reg_IF_ID_PC : std_logic_vector(31 downto 0) := (others => '0');
696     signal reg_IF_ID_Instr : std_logic_vector(31 downto 0) := (others => '0');
697
698     signal reg_ID_EX_RegDst, reg_ID_EX_ALUSrc, reg_ID_EX_MemToReg,
699     reg_ID_EX_RegWrite,
700     reg_ID_EX_MemRead, reg_ID_EX_MemWrite, reg_ID_EX_Branch : std_logic :=
701     '0';
702     signal reg_ID_EX_ALUOp : std_logic_vector(1 downto 0) := (others => '0');
703     signal reg_ID_EX_PC, reg_ID_EX_RSdata, reg_ID_EX_RTdata, reg_ID_EX_Imm32 :
704     std_logic_vector(31 downto 0) := (others => '0');
705     signal reg_ID_EX_Rs, reg_ID_EX_Rt, reg_ID_EX_Rd : std_logic_vector(4 downto 0)
706     := (others => '0');
707
708     signal reg_EX_MEM_MemToReg, reg_EX_MEM_RegWrite, reg_EX_MEM_MemRead,
709     reg_EX_MEM_MemWrite, reg_EX_MEM_Branch : std_logic := '0';
710     signal reg_EX_MEM_ALUResult, reg_EX_MEM_RTdata : std_logic_vector(31 downto 0)
711     := (others => '0');
712     signal reg_EX_MEM_WriteReg : std_logic_vector(4 downto 0) := (others => '0');
713     signal reg_EX_MEM_Zero : std_logic := '0';
714
715     signal reg_MEMORY_WB_MemToReg, reg_MEMORY_WB_RegWrite : std_logic := '0';
716     signal reg_MEMORY_WB_ReadData, reg_MEMORY_WB_ALUResult : std_logic_vector(31 downto
717     0) := (others => '0');
718     signal reg_MEMORY_WB_WriteReg : std_logic_vector(4 downto 0) := (others => '0');

```

```

712      -- Pipe signals
713      signal PC, PC_next, Instr_IF : std_logic_vector(31 downto 0);
714      signal IF_ID_PC, IF_ID_Instr : std_logic_vector(31 downto 0);
715
716      signal ID_EX_RegDst, ID_EX_ALUSrc, ID_EX_MemToReg, ID_EX_RegWrite,
717          ID_EX_MemRead, ID_EX_MemWrite, ID_EX_Branch : std_logic;
718      signal ID_EX_ALUOp : std_logic_vector(1 downto 0);
719      signal ID_EX_PC, ID_EX_RSdata, ID_EX_RTdata, ID_EX_Imm32 : std_logic_vector(31
720          downto 0);
721      signal ID_EX_Rs, ID_EX_Rt, ID_EX_Rd : std_logic_vector(4 downto 0);
722
723      signal EX_MEMORY_MemToReg, EX_MEMORY_RegWrite, EX_MEMORY_MemRead, EX_MEMORY_MemWrite,
724          EX_MEMORY_Branch : std_logic;
725      signal EX_MEMORY_ALUResult, EX_MEMORY_RTdata : std_logic_vector(31 downto 0);
726      signal EX_MEMORY_WriteReg : std_logic_vector(4 downto 0);
727      signal EX_MEMORY_Zero : std_logic;
728
729      signal MEM_WB_MemToReg, MEM_WB_RegWrite : std_logic;
730      signal MEM_WB_ReadData, MEM_WB_ALUResult : std_logic_vector(31 downto 0);
731      signal MEM_WB_WriteReg : std_logic_vector(4 downto 0);
732
733      signal RegDst_i, ALUSrc_i, MemToReg_i, RegWrite_i, MemRead_i, MemWrite_i,
734          Branch_i, Jump_i : std_logic;
735      signal ALUOp_i : std_logic_vector(1 downto 0);
736      signal RS_data, RT_data, Imm32 : std_logic_vector(31 downto 0);
737      signal ALU_input_B, ALUResult_ex : std_logic_vector(31 downto 0);
738      signal ALUZero_ex : std_logic;
739      signal ALUCtrl_ex : std_logic_vector(3 downto 0);
740      signal rs_value_for_alu, rt_value_for_alu : std_logic_vector(31 downto 0);
741      signal EX_WriteReg : std_logic_vector(4 downto 0);
742      signal MEM_ReadData, WB_WriteData : std_logic_vector(31 downto 0);
743      signal ForwardA, ForwardB : std_logic_vector(1 downto 0);
744      signal PCWrite, IF_ID_Write, Stall : std_logic;
745
746      -- Scan chain constants
747      constant W_PC      : integer := 32;
748      constant W_IFID    : integer := 64;
749      constant W_IDEX    : integer := 152;
750      constant W_EXMEM   : integer := 75;
751      constant W_MEMWB   : integer := 71;
752
753      signal scan_chain_pc      : std_logic_vector(W_PC-1 downto 0) := (others => '0')
754          ;
755      signal scan_chain_ifid    : std_logic_vector(W_IFID-1 downto 0) := (others =>
756          '0');
757      signal scan_chain_idex    : std_logic_vector(W_IDEX-1 downto 0) := (others =>
758          '0');
759      signal scan_chain_exmem   : std_logic_vector(W_EXMEM-1 downto 0) := (others =>
760          '0');
761      signal scan_chain_memwb   : std_logic_vector(W_MEMWB-1 downto 0) := (others =>
762          '0');
763
764      signal func_pc_concat     : std_logic_vector(W_PC-1 downto 0);
765      signal func_ifid_concat   : std_logic_vector(W_IFID-1 downto 0);
766      signal func_index_concat  : std_logic_vector(W_IDEX-1 downto 0);
767      signal func_exmem_concat  : std_logic_vector(W_EXMEM-1 downto 0);
768      signal func_memwb_concat  : std_logic_vector(W_MEMWB-1 downto 0);
769
770      -- PLPF / LFSR signals (drive scan_in bits from PLPF outputs)
771      signal lfsr_vec_sig : std_logic_vector(39 downto 0);
772      signal plpf_pc_bit : std_logic := '0';
773      signal plpf_ifid_bit: std_logic := '0';
774      signal plpf_index_bit: std_logic := '0';
775      signal plpf_exmem_bit: std_logic := '0';

```

```

767
768    -- Internal MISR signature signals (renamed to avoid port-name clash)
769    signal misr_pc_sig_int      : std_logic_vector(W_PC-1 downto 0) := (others =>
770        '0');
771    signal misr_ifid_sig_int   : std_logic_vector(W_IFID-1 downto 0) := (others =>
772        '0');
773    signal misr_index_sig_int  : std_logic_vector(W_INDEX-1 downto 0) := (others =>
774        '0');
775    signal misr_exmem_sig_int : std_logic_vector(W_EXMEM-1 downto 0) := (others =>
776        '0');
777    signal misr_memwb_sig_int : std_logic_vector(W_MEMWB-1 downto 0) := (others =>
778        '0');

779 begin
780     -- Component instantiations (functional units unchanged)
781     InstrMem_inst : entity work.InstrMem
782         port map(Addr => PC, Instr => Instr_IF, sim_we => sim_we, sim_addr =>
783             sim_addr, sim_data => sim_data);

784     RegFile_inst : entity work.RegFile
785         port map(clk => clk, we => MEM_WB_RegWrite, rd_addr => MEM_WB_WriteReg,
786             rs_addr => IF_ID_Instr(25 downto 21), rt_addr => IF_ID_Instr(20
787                 downto 16),
788             wd => WB_WriteData, rs_data => RS_data, rt_data => RT_data);

789     Control_inst : entity work.ControlUnit
790         port map(Opcode => IF_ID_Instr(31 downto 26), RegDst => RegDst_i, ALUSrc
791             => ALUSrc_i,
792                 MemToReg => MemToReg_i, RegWrite => RegWrite_i, MemRead =>
793                     MemRead_i,
794                     MemWrite => MemWrite_i, Branch => Branch_i, Jump => Jump_i, ALUOp
795                         => ALUOp_i);

796     SignExt_inst : entity work.SignExt
797         port map(Imm16 => IF_ID_Instr(15 downto 0), Imm32 => Imm32);

798     ALUControl_inst : entity work.ALUControl
799         port map(ALUOp => ID_EX_ALUOp, Funct => ID_EX_Imm32(5 downto 0), ALUCtrl
800             => ALUCtrl_ex);

801     ALU_inst : entity work.ALU
802         port map(A => rs_value_for_alu, B => ALU_input_B, ALUCtrl => ALUCtrl_ex,
803             Result => ALUResult_ex, Zero => ALUZero_ex);

804     Forward_inst : entity work.ForwardUnit
805         port map(EX_MEM_RegWrite => EX_MEM_RegWrite, MEM_WB_RegWrite =>
806             MEM_WB_RegWrite,
807                 EX_MEM_Rd => EX_MEM_WriteReg, MEM_WB_Rd => MEM_WB_WriteReg,
808                 ID_EX_Rs => ID_EX_Rs, ID_EX_Rt => ID_EX_Rt,
809                 ForwardA => ForwardA, ForwardB => ForwardB);

810     Hazard_inst : entity work.HazardUnit
811         port map(ID_EX_MemRead => ID_EX_MemRead, ID_EX_Rt => ID_EX_Rt,
812             IF_ID_Rs => IF_ID_Instr(25 downto 21), IF_ID_Rt => IF_ID_Instr(20
813                 downto 16),
814                 PCWrite => PCWrite, IF_ID_Write => IF_ID_Write, Stall => Stall);

815     DataMem_inst : entity work.DataMem
816         port map(clk => clk, mem_write => EX_MEM_MemWrite, mem_read =>
817             EX_MEM_MemRead,
818                 Addr => EX_MEM_ALUResult, WriteData => EX_MEM_RTdata, ReadData =>
819                     MEM_ReadData);

820     -- LFSR and PLPF instantiations (drive scan-chain serial input bits)

```

```

816     u_lfsr40: entity work.lfsr40
817     port map(
818         clk => clk,
819         rst => reset,
820         load_seed => '0',
821         seed => (others => '0'),
822         enable => '1',
823         lfsr_vec => lfsr_vec_sig
824     );
825
826     -- One dynamic PLPF per scan chain. valid tied to scan_en signals.
827     u_plpf_pc: entity work.dynamic_plpf_taps_40
828     port map(
829         clk => clk, rst => reset, valid => scan_en_pc,
830         lfsr_vec => lfsr_vec_sig, sel_n => "00",
831         out_valid => open, out_bit => plpf_pc_bit
832     );
833
834     u_plpf_ifid: entity work.dynamic_plpf_taps_40
835     port map(
836         clk => clk, rst => reset, valid => scan_en_ifid,
837         lfsr_vec => lfsr_vec_sig, sel_n => "01",
838         out_valid => open, out_bit => plpf_ifid_bit
839     );
840
841     u_plpf_idex: entity work.dynamic_plpf_taps_40
842     port map(
843         clk => clk, rst => reset, valid => scan_en_idex,
844         lfsr_vec => lfsr_vec_sig, sel_n => "10",
845         out_valid => open, out_bit => plpf_idex_bit
846     );
847
848     u_plpf_exmem: entity work.dynamic_plpf_taps_40
849     port map(
850         clk => clk, rst => reset, valid => scan_en_exmem,
851         lfsr_vec => lfsr_vec_sig, sel_n => "00",
852         out_valid => open, out_bit => plpf_exmem_bit
853     );
854
855     -- Functional concatenations
856     func_pc_concat <= reg_PC;
857     func_ifid_concat <= reg_ID_ID_PC & reg_ID_ID_Instr;
858     func_index_concat <= (reg_ID_EX_RegDst & reg_ID_EX_ALUSrc & reg_ID_EX_MemToReg
859     & reg_ID_EX_RegWrite &
860             reg_ID_EX_MemRead & reg_ID_EX_MemWrite & reg_ID_EX_Branch
861             & reg_ID_EX_ALUOp) &
862             reg_ID_EX_PC & reg_ID_EX_RSdata & reg_ID_EX_RTdata &
863             reg_ID_EX_Imm32 &
864             reg_ID_EX_Rs & reg_ID_EX_Rt & reg_ID_EX_Rd;
865     func_exmem_concat <= (reg_EX_MEMORY_MemToReg & reg_EX_MEMORY_RegWrite &
866     reg_EX_MEMORY_MemRead &
867             reg_EX_MEMORY_MemWrite & reg_EX_MEMORY_Branch) &
868             reg_EX_MEMORY_ALUResult & reg_EX_MEMORY_RTdata &
869             reg_EX_MEMORY_WriteReg & reg_EX_MEMORY_Zero;
870     func_memwb_concat <= (reg_MEMORY_WB_MemToReg & reg_MEMORY_WB_RegWrite) &
871             reg_MEMORY_WB_ReadData & reg_MEMORY_WB_ALUResult &
872             reg_MEMORY_WB_WriteReg;
873
874     -- Scan chain processes: use PLPF outputs as the serial input when shifting.
875     process(clk, reset)
876     begin
877         if reset = '1' then
878             scan_chain_pc <= (others => '0');
879         elsif rising_edge(clk) then

```

```

874         if scan_en_pc = '1' then
875             scan_chain_pc <= plpf_pc_bit & scan_chain_pc(W_PC-1 downto 1);
876         else
877             scan_chain_pc <= func_pc_concat;
878         end if;
879     end if;
880 end process;
881 scan_out_pc <= scan_chain_pc(0);
882
883 process(clk, reset)
884 begin
885     if reset = '1' then
886         scan_chain_ifid <= (others => '0');
887     elsif rising_edge(clk) then
888         if scan_en_ifid = '1' then
889             scan_chain_ifid <= plpf_ifid_bit & scan_chain_ifid(W_IFID-1 downto
890                         1);
891         else
892             scan_chain_ifid <= func_ifid_concat;
893         end if;
894     end if;
895 end process;
896 scan_out_ifid <= scan_chain_ifid(0);
897
898 process(clk, reset)
899 begin
900     if reset = '1' then
901         scan_chain_idex <= (others => '0');
902     elsif rising_edge(clk) then
903         if scan_en_idex = '1' then
904             scan_chain_idex <= plpf_index_bit & scan_chain_idex(W_INDEX-1 downto
905                         1);
906         else
907             scan_chain_idex <= func_index_concat;
908         end if;
909     end if;
910 end process;
911 scan_out_idex <= scan_chain_idex(0);
912
913 process(clk, reset)
914 begin
915     if reset = '1' then
916         scan_chain_exmem <= (others => '0');
917     elsif rising_edge(clk) then
918         if scan_en_exmem = '1' then
919             scan_chain_exmem <= plpf_exmem_bit & scan_chain_exmem(W_EXMEM-1
920                         downto 1);
921         else
922             scan_chain_exmem <= func_exmem_concat;
923         end if;
924     end if;
925 end process;
926 scan_out_exmem <= scan_chain_exmem(0);
927
928 process(clk, reset)
929 begin
930     if reset = '1' then
931         scan_chain_memwb <= (others => '0');
932     elsif rising_edge(clk) then
933         if scan_en_memwb = '1' then
934             scan_chain_memwb <= scan_in_memwb & scan_chain_memwb(W_MEMWB-1
935                         downto 1);
936         else
937             scan_chain_memwb <= func_memwb_concat;

```

```

934         end if;
935     end if;
936 end process;
937 scan_out_memwb <= scan_chain_memwb(0);
938 -----
939 -- MISR instantiations (placed AFTER scan chains so 'scan_chain_*' is valid)
940 -- MISRs are disabled while the chain is shifting (enable = not scan_en_*)
941 -----
942 u_misr_pc: entity work.MISR
943     generic map ( W => W_PC )
944     port map (
945         clk      => clk,
946         rst      => reset,
947         enable   => not scan_en_pc,           -- MISR off while loading
948         din      => scan_chain_pc,
949         sig_out  => misr_pc_sig_int
950     );
951
952
953 u_misr_ifid: entity work.MISR
954     generic map ( W => W_IFID )
955     port map (
956         clk      => clk,
957         rst      => reset,
958         enable   => not scan_en_ifid,
959         din      => scan_chain_ifid,
960         sig_out  => misr_ifid_sig_int
961     );
962
963 u_misr_index: entity work.MISR
964     generic map ( W => W_INDEX )
965     port map (
966         clk      => clk,
967         rst      => reset,
968         enable   => not scan_en_index,
969         din      => scan_chain_index,
970         sig_out  => misr_index_sig_int
971     );
972
973 u_misr_exmem: entity work.MISR
974     generic map ( W => W_EXMEM )
975     port map (
976         clk      => clk,
977         rst      => reset,
978         enable   => not scan_en_exmem,
979         din      => scan_chain_exmem,
980         sig_out  => misr_exmem_sig_int
981     );
982
983 u_misr_memwb: entity work.MISR
984     generic map ( W => W_MEMWB )
985     port map (
986         clk      => clk,
987         rst      => reset,
988         enable   => not scan_en_memwb,
989         din      => scan_chain_memwb,
990         sig_out  => misr_memwb_sig_int
991     );
992
993 -- connect internal MISR signals to entity outputs
994 misr_pc_sig    <= misr_pc_sig_int;
995 misr_ifid_sig  <= misr_ifid_sig_int;
996 misr_index_sig <= misr_index_sig_int;
997 misr_exmem_sig <= misr_exmem_sig_int;

```

```

998     misr_memwb_sig <= misr_memwb_sig_int;
999
1000    -- Pipe signal selection from scan chains or functional registers
1001    PC <= scan_chain_pc when scan_en_pc = '1' else reg_PC;
1002
1003    IF_ID_PC      <= scan_chain_ifid(W_IFID-1 downto 32) when scan_en_ifid = '1'
1004      else reg_IF_ID_PC;
1005    IF_ID_Instr   <= scan_chain_ifid(31 downto 0)           when scan_en_ifid = '1'
1006      else reg_IF_ID_Instr;
1007
1008    ID_EX_RegDst  <= scan_chain_idex(151) when scan_en_idex = '1' else
1009      reg_ID_EX_RegDst;
1010    ID_EX_ALUSrc  <= scan_chain_idex(150) when scan_en_idex = '1' else
1011      reg_ID_EX_ALUSrc;
1012    ID_EX_MemToReg <= scan_chain_idex(149) when scan_en_idex = '1' else
1013      reg_ID_EX_MemToReg;
1014    ID_EX_RegWrite <= scan_chain_idex(148) when scan_en_idex = '1' else
1015      reg_ID_EX_RegWrite;
1016    ID_EX_MemRead  <= scan_chain_idex(147) when scan_en_idex = '1' else
1017      reg_ID_EX_MemRead;
1018    ID_EX_MemWrite  <= scan_chain_idex(146) when scan_en_idex = '1' else
1019      reg_ID_EX_MemWrite;
1020    ID_EX_Branch   <= scan_chain_idex(145) when scan_en_idex = '1' else
1021      reg_ID_EX_Branch;
1022    ID_EX_ALUOp    <= scan_chain_idex(144 downto 143) when scan_en_idex = '1' else
1023      reg_ID_EX_ALUOp;
1024    ID_EX_PC       <= scan_chain_idex(142 downto 111) when scan_en_idex = '1' else
1025      reg_ID_EX_PC;
1026    ID_EX_RSdata   <= scan_chain_idex(110 downto 79)  when scan_en_idex = '1' else
1027      reg_ID_EX_RSdata;
1028    ID_EX_RTdata   <= scan_chain_idex(78 downto 47)   when scan_en_idex = '1' else
1029      reg_ID_EX_RTdata;
1030    ID_EX_Imm32    <= scan_chain_idex(46 downto 15)   when scan_en_idex = '1' else
1031      reg_ID_EX_Imm32;
1032    ID_EX_Rs       <= scan_chain_idex(14 downto 10)   when scan_en_idex = '1' else
1033      reg_ID_EX_Rs;
1034    ID_EX_Rt       <= scan_chain_idex(9  downto 5)    when scan_en_idex = '1' else
1035      reg_ID_EX_Rt;
1036    ID_EX_Rd       <= scan_chain_idex(4  downto 0)    when scan_en_idex = '1' else
1037      reg_ID_EX_Rd;
1038
1039    EX_MEM_MemToReg <= scan_chain_exmem(74) when scan_en_exmem = '1' else
1040      reg_EX_MEM_MemToReg;
1041    EX_MEM_RegWrite <= scan_chain_exmem(73) when scan_en_exmem = '1' else
1042      reg_EX_MEM_RegWrite;
1043    EX_MEM_MemRead  <= scan_chain_exmem(72) when scan_en_exmem = '1' else
1044      reg_EX_MEM_MemRead;
1045    EX_MEM_MemWrite <= scan_chain_exmem(71) when scan_en_exmem = '1' else
1046      reg_EX_MEM_MemWrite;
1047    EX_MEM_Branch   <= scan_chain_exmem(70) when scan_en_exmem = '1' else
1048      reg_EX_MEM_Branch;
1049    EX_MEM_ALUResult <= scan_chain_exmem(69 downto 38) when scan_en_exmem = '1'
1050      else reg_EX_MEM_ALUResult;
1051    EX_MEM_RTdata   <= scan_chain_exmem(37 downto 6)  when scan_en_exmem = '1'
1052      else reg_EX_MEM_RTdata;
1053    EX_MEM_WriteReg <= scan_chain_exmem(5  downto 1)  when scan_en_exmem = '1'
1054      else reg_EX_MEM_WriteReg;
1055    EX_MEM_Zero     <= scan_chain_exmem(0)           when scan_en_exmem = '1'
1056      else reg_EX_MEM_Zero;
1057
1058    MEM_WB_MemToReg <= scan_chain_memwb(70) when scan_en_memwb = '1' else
1059      reg_MEM_WB_MemToReg;
1060    MEM_WB_RegWrite <= scan_chain_memwb(69) when scan_en_memwb = '1' else
1061      reg_MEM_WB_RegWrite;

```

```

1034     MEM_WB_ReadData  <= scan_chain_memwb(68 downto 37) when scan_en_memwb = '1'
1035         else reg_MEMORY_WB_ReadData;
1036     MEM_WB_ALUResult <= scan_chain_memwb(36 downto 5)  when scan_en_memwb = '1'
1037         else reg_MEMORY_WB_ALUResult;
1038     MEM_WB_WriteReg   <= scan_chain_memwb(4 downto 0)   when scan_en_memwb = '1'
1039         else reg_MEMORY_WB_WriteReg;
1040
1041 -- Functional register update processes (unchanged) ...
1042 process(clk, reset)
1043 begin
1044     if reset = '1' then
1045         reg_PC <= (others => '0');
1046     elsif rising_edge(clk) then
1047         if PCWrite = '1' then
1048             reg_PC <= PC_next;
1049         end if;
1050     end if;
1051 end process;
1052
1053 process(clk, reset)
1054 begin
1055     if reset = '1' then
1056         reg_IF_ID_PC <= (others => '0');
1057         reg_IF_ID_Instr <= (others => '0');
1058     elsif rising_edge(clk) then
1059         if IF_ID_Write = '1' then
1060             reg_IF_ID_PC <= PC;
1061             reg_IF_ID_Instr <= Instr_IF;
1062         end if;
1063     end if;
1064 end process;
1065
1066 process(clk, reset)
1067 begin
1068     if reset = '1' then
1069         reg_ID_EX_RegDst <= '0';
1070         reg_ID_EX_ALUSrc <= '0';
1071         reg_ID_EX_MemToReg <= '0';
1072         reg_ID_EX_RegWrite <= '0';
1073         reg_ID_EX_MemRead <= '0';
1074         reg_ID_EX_MemWrite <= '0';
1075         reg_ID_EX_Branch <= '0';
1076         reg_ID_EX_ALUOp <= (others => '0');
1077         reg_ID_EX_PC <= (others => '0');
1078         reg_ID_EX_RSdata <= (others => '0');
1079         reg_ID_EX_RTdata <= (others => '0');
1080         reg_ID_EX_Imm32 <= (others => '0');
1081         reg_ID_EX_Rs <= (others => '0');
1082         reg_ID_EX_Rt <= (others => '0');
1083         reg_ID_EX_Rd <= (others => '0');
1084     elsif rising_edge(clk) then
1085         if Stall = '1' then
1086             reg_ID_EX_RegDst <= '0';
1087             reg_ID_EX_ALUSrc <= '0';
1088             reg_ID_EX_MemToReg <= '0';
1089             reg_ID_EX_RegWrite <= '0';
1090             reg_ID_EX_MemRead <= '0';
1091             reg_ID_EX_MemWrite <= '0';
1092             reg_ID_EX_Branch <= '0';
1093             reg_ID_EX_ALUOp <= (others => '0');
1094         else
1095             reg_ID_EX_RegDst <= RegDst_i;
1096             reg_ID_EX_ALUSrc <= ALUSrc_i;
1097             reg_ID_EX_MemToReg <= MemToReg_i;

```

```

1095      reg_ID_EX_RegWrite <= RegWrite_i;
1096      reg_ID_EX_MemRead <= MemRead_i;
1097      reg_ID_EX_MemWrite <= MemWrite_i;
1098      reg_ID_EX_Branch <= Branch_i;
1099      reg_ID_EX_ALUOp <= ALUOp_i;
1100      reg_ID_EX_PC <= IF_ID_PC;
1101      reg_ID_EX_RSdata <= RS_data;
1102      reg_ID_EX_RTdata <= RT_data;
1103      reg_ID_EX_Imm32 <= Imm32;
1104      reg_ID_EX_Rs <= IF_ID_Instr(25 downto 21);
1105      reg_ID_EX_Rt <= IF_ID_Instr(20 downto 16);
1106      reg_ID_EX_Rd <= IF_ID_Instr(15 downto 11);
1107      end if;
1108   end if;
1109 end process;
1110
1111 process(clk, reset)
1112 begin
1113   if reset = '1' then
1114     reg_EX_MEM_MemToReg <= '0';
1115     reg_EX_MEM_RegWrite <= '0';
1116     reg_EX_MEM_MemRead <= '0';
1117     reg_EX_MEM_MemWrite <= '0';
1118     reg_EX_MEM_Branch <= '0';
1119     reg_EX_MEM_ALUResult <= (others => '0');
1120     reg_EX_MEM_RTdata <= (others => '0');
1121     reg_EX_MEM_WriteReg <= (others => '0');
1122     reg_EX_MEM_Zero <= '0';
1123   elsif rising_edge(clk) then
1124     reg_EX_MEM_MemToReg <= ID_EX_MemToReg;
1125     reg_EX_MEM_RegWrite <= ID_EX_RegWrite;
1126     reg_EX_MEM_MemRead <= ID_EX_MemRead;
1127     reg_EX_MEM_MemWrite <= ID_EX_MemWrite;
1128     reg_EX_MEM_Branch <= ID_EX_Branch;
1129     reg_EX_MEM_ALUResult <= ALUResult_ex;
1130     reg_EX_MEM_RTdata <= rt_value_for_alu;
1131     reg_EX_MEM_WriteReg <= EX_WriteReg;
1132     reg_EX_MEM_Zero <= ALUZero_ex;
1133   end if;
1134 end process;
1135
1136 process(clk, reset)
1137 begin
1138   if reset = '1' then
1139     reg_MEMORY_WB_MemToReg <= '0';
1140     reg_MEMORY_WB_RegWrite <= '0';
1141     reg_MEMORY_WB_ReadData <= (others => '0');
1142     reg_MEMORY_WB_ALUResult <= (others => '0');
1143     reg_MEMORY_WB_WriteReg <= (others => '0');
1144   elsif rising_edge(clk) then
1145     reg_MEMORY_WB_MemToReg <= EX_MEMORY_MemToReg;
1146     reg_MEMORY_WB_RegWrite <= EX_MEMORY_RegWrite;
1147     reg_MEMORY_WB_ReadData <= MEM_ReadData;
1148     reg_MEMORY_WB_ALUResult <= EX_MEMORY_ALUResult;
1149     reg_MEMORY_WB_WriteReg <= EX_MEMORY_WriteReg;
1150   end if;
1151 end process;
1152
1153 -- Datapath logic and remaining code unchanged...
1154 WB_WriteData <= MEMORY_WB_ReadData when MEMORY_WB_MemToReg = '1' else
1155   MEMORY_WB_ALUResult;
1156
1157 process(PC, EX_MEMORY_Branch, EX_MEMORY_Zero, IF_ID_Instr, Jump_i, EX_MEMORY_ALUResult)
1158   variable pc_plus4 : std_logic_vector(31 downto 0);

```

```

1158     variable jump_target : std_logic_vector(31 downto 0);
1159 begin
1160     pc_plus4 := std_logic_vector(unsigned(PC) + 4);
1161     jump_target := (pc_plus4(31 downto 28) & IF_ID_Instr(25 downto 0) & "00");
1162     PC_next <= pc_plus4;
1163     if (EX_MEM_Branch = '1') and (EX_MEM_Zero = '1') then
1164         PC_next <= EX_MEM_ALUResult;
1165     elsif Jump_i = '1' then
1166         PC_next <= jump_target;
1167     end if;
1168 end process;
1169
1170 -- Forwarding muxes, ALU input selection, EX dest compute, debug outputs...
1171 process(ForwardA, ID_EX_RSdata, EX_MEMORY_ALUResult, MEM_WB_ALUResult,
1172          MEM_WB_ReadData, MEM_WB_MemToReg)
1173 begin
1174     case ForwardA is
1175         when "00" => rs_value_for_alu <= ID_EX_RSdata;
1176         when "10" => rs_value_for_alu <= EX_MEMORY_ALUResult;
1177         when "01" =>
1178             if MEM_WB_MemToReg = '1' then
1179                 rs_value_for_alu <= MEM_WB_ReadData;
1180             else
1181                 rs_value_for_alu <= MEM_WB_ALUResult;
1182             end if;
1183         when others => rs_value_for_alu <= ID_EX_RSdata;
1184     end case;
1185 end process;
1186
1186 process(ForwardB, ID_EX_RTdata, EX_MEMORY_ALUResult, MEM_WB_ALUResult,
1187          MEM_WB_ReadData, MEM_WB_MemToReg)
1188 begin
1189     case ForwardB is
1190         when "00" => rt_value_for_alu <= ID_EX_RTdata;
1191         when "10" => rt_value_for_alu <= EX_MEMORY_ALUResult;
1192         when "01" =>
1193             if MEM_WB_MemToReg = '1' then
1194                 rt_value_for_alu <= MEM_WB_ReadData;
1195             else
1196                 rt_value_for_alu <= MEM_WB_ALUResult;
1197             end if;
1198         when others => rt_value_for_alu <= ID_EX_RTdata;
1199     end case;
1200 end process;
1201
1201 ALU_input_B <= ID_EX_Imm32 when ID_EX_ALUSrc = '1' else rt_value_for_alu;
1202
1203 process(ID_EX_RegDst, ID_EX_Rt, ID_EX_Rd)
1204 begin
1205     if ID_EX_RegDst = '1' then
1206         EX_WriteReg <= ID_EX_Rd;
1207     else
1208         EX_WriteReg <= ID_EX_Rt;
1209     end if;
1210 end process;
1211
1212 debug_PC      <= PC;
1213 debug_Instr    <= IF_ID_Instr;
1214 debug_ALUResult <= ALUResult_ex;
1215 debug_Zero     <= ALUZero_ex;
1216 debug_ALUCtrl   <= ALUCtrl_ex;
1217 debug_ALU_InA    <= rs_value_for_alu;
1218 debug_ALU_InB    <= ALU_input_B;
1219 debug_SignExtImm <= ID_EX_Imm32;

```

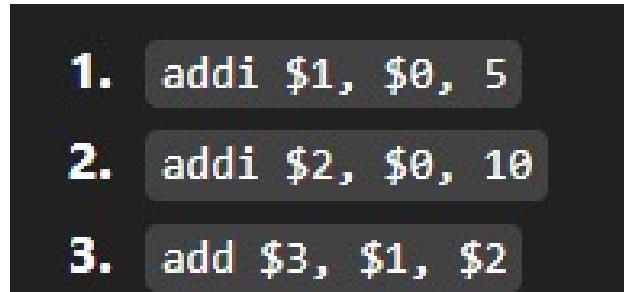


Figure 5: Instruction ran on the normal operation of the processor

> [1][31:0]	00000005
> [2][31:0]	0000000a
> [3][31:0]	0000000f

Figure 6: Register file after the simulation

```

1220     debug_RegWrite    <= MEM_WB_RegWrite;
1221
1222 end Behavioral;

```

10 Conclusion

The proposed power-controlled LBIST framework for the MIPS32 processor achieves substantial scan-shift power reduction while preserving test quality. By combining a programmable PLPF bank with WTM-based temporal segmentation, the design lowers average test power by more than fourfold with negligible area overhead and no loss in fault coverage. Validation across simulations and device testing confirms stable performance, reduced IR-drop risk, and smooth integration with existing scan architecture. Overall, the work provides a practical and efficient approach for implementing low-power LBIST in modern pipelined processors.

References

1. Bushnell & Agrawal. *Essentials of Electronic Testing*. Kluwer, 2000.
2. Girard et al. *Power-Aware Testing*. Springer, 2008.
3. Saxena et al. "IR-Drop in At-Speed Testing." ITC 2003.
4. Crouch. *Design-for-Test for Digital ICs*. Prentice Hall, 1999.
5. Sankaralingam et al. "Scan Vector Power Control." VTS 2002.
6. Wang & Gupta. "Heat Minimization During Test." IEEE Trans., 1997.
7. IEEE Standard 1149.1. *Boundary-Scan Architecture*. 2013.
8. Kato, T., Wang, S., Sato, Y., Kajihara, S., & Wen, X. "A Flexible Scan-in Power Control Method in Logic BIST and Its Evaluation with TEG Chips." *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 3, pp. 591–600, 2020.

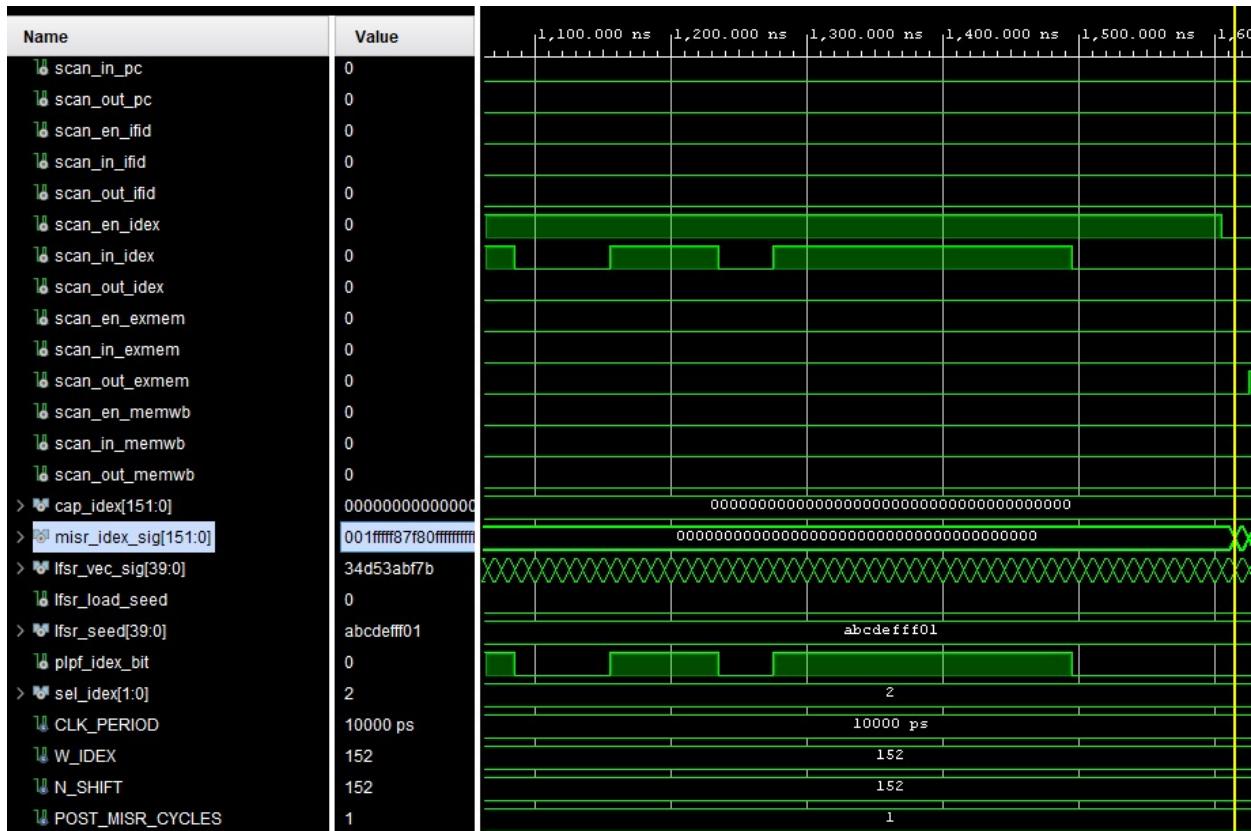


Figure 7: Testing the integrated MISR and Scan chains timing and control

End of Report

Power-Controlled LBIST for MIPS32

November 20, 2025