eSim Summer Fellowship 2026

On

Development of Digital IP Models for eSim

Submitted by

Beer Mohammed Irfan Z

B.E. Electronics Engineering
(VLSI Design and Technology)

RMK Engineering College, Chennai

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

Submitted on

July 4, 2026

Acknowledgement

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K for their valuable guidance, coordination, and technical inputs during the internship.Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to realworld circuit modeling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Contents

Acknowledgement **1**

1 Introduction **5**
1.1 About FOSSEE 5
1.2 About eSim 5
1.3 Objectives of the Internship 5

2 Literature Survey **7**
2.1 Ngspice 7
2.2 KiCad 7
2.3 SPICE Subcircuits 7

3 Overview of IP Design and Digital Accelerators **8**
3.1 Introduction to IP Blocks 8
3.2 Need for IP-Based Design 8
3.3 Types of IP Blocks Implemented 9
3.4 Design Methodology in eSim 9
3.5 Summary 9

4 Tools and Technologies Used **10**
4.1 eSim 10
4.2 Ngspice 10
4.3 KiCad 10
4.4 LaTeX and Overleaf 10

5 Systolic Array Accelerator (2×2) **11**
5.1 Design Study 11
5.2 Design Implementation 12
5.3 Test Bench Design 14
5.4 Simulation Results 16

6 Digital Lock Detector **18**
6.1 Design Study 18
6.2 Design Implementation 18
6.3 Test Bench Design 20
6.4 Simulation Results 22

7 FIR Filter Accelerator (8-Tap) 24
7.1 Design Study . . . 24
7.2 Design Implementation . . . 24
7.3 Testbench Design . . . 27
7.4 Simulation Results . . . 28

8 Matrix Multiplication Accelerator 31
8.1 Design Study . . . 31
8.2 Design Implementation . . . 32
8.3 Design Implementation . . . 33
8.4 Test Bench Design . . . 34
8.5 Simulation Results . . . 36

9 Clock Failure Detector 38
9.1 Design Study . . . 38
9.2 Design Implementation . . . 38
9.3 Test Bench Design . . . 41
9.4 Simulation Results . . . 43

10 Reed-Muller Encoder 45
10.1 Design Study . . . 45
10.2 Design Implementation . . . 45
10.3 Test Bench Design . . . 47
10.4 Simulation Results . . . 48

11 Sequence Alignment Detector 50
11.1 Design Study . . . 50
11.2 Design Implementation . . . 50
11.3 Test Bench Design . . . 51
11.4 Simulation Results . . . 53

12 RISC-V Pipeline Hazard Unit 54
12.1 Design Study . . . 54
12.2 Design Implementation . . . 54
12.3 Test Bench Design . . . 56
12.4 Simulation Results . . . 58

13 Instruction Cache Controller 59
13.1 Design Study . . . 59
13.2 Design Implementation . . . 60
13.3 Test Bench Design . . . 62
13.4 Simulation Results . . . 64

14 Digital Twin Lockstep 66
14.1 Design Study . . . 66
14.2 Design Implementation . . . 67
14.3 Test Bench Design . . . 69
14.4 Simulation Results . . . 72

15 Conclusion **73**
15.1 Summary of Contributions 73
15.2 Learning Outcomes 74
15.3 Future Scope 74

Bibliography **76**

Chapter 1

Introduction

1.1 About FOSSEE

FOSSEE (Free/Libre and Open Source Software for Education) is a project promoted by the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education, Government of India. The main objective of the FOSSEE project is to encourage the use of Free and Open Source Software (FOSS) tools in academic institutions and improve the quality of education through open-source technologies. The project is coordinated at IIT Bombay and supports various open-source software platforms used in engineering, science, and education.

FOSSEE conducts workshops, internships, training programs, and research activities to help students and educators gain practical exposure to open-source tools. It also provides opportunities for students to contribute to real-world projects involving software development, circuit design, simulation, documentation, and VLSI-related activities.

1.2 About eSim

eSim is an open-source Electronic Design Automation (EDA) tool developed by the FOSSEE project, IIT Bombay. It is mainly used for circuit design, simulation, analysis, and PCB development. eSim integrates various open-source software tools such as KiCad, Ngspice, and Scilab to provide a complete environment for electronic circuit simulation and testing.

Using eSim, users can create schematic diagrams, perform SPICE simulations, analyze waveforms, and design PCB layouts efficiently. The software supports analog, digital, and mixed-signal circuit simulations, making it useful for students, researchers, and engineers.

1.3 Objectives of the Internship

The main objective of this internship was to gain practical knowledge and hands-on experience in electronic circuit simulation and IC subcircuit development using the

eSim tool. The internship focused on understanding SPICE modeling techniques, analyzing IC datasheets, and implementing accurate subcircuit models for simulation purposes.

Another important objective was to contribute to the open-source ecosystem by developing and verifying IC subcircuits that can be used by students and researchers in future projects. The internship also aimed to improve technical skills related to circuit analysis, debugging, simulation verification, and documentation.

Throughout the internship, multiple IC subcircuits were successfully designed and tested using eSim and Ngspice simulation tools.

Chapter 2

Literature Survey

eSim is an open-source Electronic Design Automation (EDA) tool developed under the FOSSEE project at IIT Bombay. It is designed for schematic capture, circuit simulation, PCB design, and analysis of electronic circuits. eSim integrates several open-source tools such as KiCad, Ngspice, and Scilab to provide a complete environment for electronic system design.

2.1 Ngspice

Ngspice is an open-source mixed-signal circuit simulator widely used for analog and digital circuit analysis. It is based on the SPICE simulation engine and is integrated with eSim for performing circuit simulations and waveform analysis.

In this internship, Ngspice played a major role in validating the behavior of implemented IC subcircuits. The simulator helped in debugging errors, verifying output waveforms, and checking the functionality of SPICE models developed during the internship.

2.2 KiCad

KiCad is an open-source software suite used for schematic capture and PCB design. It is integrated with eSim to provide a user-friendly interface for designing electronic circuits.

The schematic editor in KiCad was used during the internship to design and verify electronic circuits associated with the IC subcircuits.

2.3 SPICE Subcircuits

SPICE subcircuits are reusable circuit models used to represent integrated circuits and complex electronic components in simulation environments. Subcircuits simplify circuit design by allowing designers to use predefined models instead of creating the entire internal circuitry repeatedly.

Chapter 3

Overview of IP Design and Digital Accelerators

3.1 Introduction to IP Blocks

In modern digital system design, IP (Intellectual Property) blocks are reusable hardware modules implemented using Hardware Description Languages (HDL) such as Verilog. These IPs are widely used in FPGA, ASIC, and system-on-chip (SoC) design methodologies.

In this internship, various digital IPs were designed, implemented, and verified using the eSim platform integrated with Verilog and Ngspice simulation support.

These IP blocks focus on computation, control, and signal processing tasks, enabling efficient hardware acceleration.

3.2 Need for IP-Based Design

IP-based hardware design is essential in modern electronic systems due to the following advantages:

- Reusability of hardware modules across different systems
- Faster development of complex digital architectures
- Reduced design complexity through modular approach
- Efficient hardware acceleration for computation-heavy tasks
- Easier verification and testing using standardized testbenches

3.3 Types of IP Blocks Implemented

The following digital IP blocks were designed and implemented during the internship:

- Systolic Array Accelerator (2x2)
- Matrix Multiplication Accelerator
- FIR Filter Accelerator (8-tap)
- Digital Lock Detector
- Clock Failure Detector
- Sequence Alignment Detector
- Reed-Muller Encoder
- RISC-V Pipeline Hazard Detection Unit
- Instruction Cache Controller
- Digital Twin Lockstep Comparator

3.4 Design Methodology in eSim

The IP development process was carried out using the eSim environment with Verilog HDL integration. The general design flow included:

- Understanding architecture and functional requirements of each IP
- Developing Verilog HDL design modules
- Creating testbenches for functional verification
- Simulating designs using Ngspice integration
- Validating outputs against expected behavior

Each IP block was individually tested and verified to ensure correct functionality under different input conditions.

3.5 Summary

A total of ten digital IP blocks were successfully implemented and validated. These designs cover a wide range of applications including digital signal processing, error detection, processor control logic, and parallel computation architectures.

Chapter 4

Tools and Technologies Used

4.1 eSim

eSim is an open-source Electronic Design Automation (EDA) tool developed by the FOSSEE project at IIT Bombay. It is used for schematic design, circuit simulation, waveform analysis, and PCB design.

4.2 Ngspice

Ngspice is an open-source SPICE-based circuit simulator used for analog, digital, and mixed-signal simulations. It is one of the core simulation engines integrated with eSim.

4.3 KiCad

KiCad is an open-source software suite used for electronic schematic capture and PCB design. KiCad is integrated with eSim and works together with Ngspice for circuit simulation purposes.

4.4 LaTeX and Overleaf

LaTeX is a document preparation system widely used for technical and scientific documentation. Overleaf is an online LaTeX editor that provides a collaborative environment for creating professional reports and research papers. In this internship, Overleaf was used for preparing and formatting the internship report.

Chapter 5

Systolic Array Accelerator (2×2)

5.1 Design Study

The Systolic Array Accelerator is a highly parallel hardware architecture specifically designed to accelerate matrix multiplication, convolution, and other repetitive arithmetic operations commonly encountered in digital signal processing (DSP), artificial intelligence (AI), machine learning (ML), image processing, and scientific computing. Unlike conventional sequential processors, a systolic array performs computations concurrently using multiple interconnected Processing Elements (PEs), thereby significantly improving computational throughput while reducing execution time.

The primary objective of this work was to study the design principles and implementation methodology of a 2×2 systolic array accelerator using Verilog HDL and verify its functionality within the eSim simulation environment. The study focused on understanding the dataflow architecture, synchronization mechanism, pipelined execution, and parallel processing characteristics that distinguish systolic arrays from traditional processor architectures.

A systolic array consists of a regular arrangement of Processing Elements connected in a grid-like structure. Each PE performs a Multiply-and-Accumulate (MAC) operation, which forms the fundamental arithmetic operation in matrix multiplication. During every clock cycle, each processing element receives an input operand and a weight value, performs multiplication, accumulates the partial result, and forwards the operands to neighboring processing elements. This rhythmic movement of data through the array resembles the pumping action of the human heart, which is the origin of the term *systolic*.

The architecture enables efficient data reuse because input operands propagate across multiple processing elements without repeatedly accessing external memory. This significantly reduces memory bandwidth requirements, lowers power consumption, and improves computational efficiency. Such characteristics make systolic arrays the preferred hardware architecture in modern AI accelerators such as Tensor Processing Units (TPUs), neural network processors, and FPGA-based matrix multiplication engines.

Before implementation, an extensive design study was conducted to understand the timing behavior of the architecture. Particular emphasis was placed on the synchronization of data movement between neighboring processing elements, accumulation of partial products, propagation of operands through pipeline stages, and generation of the final output matrix after the required number of clock cycles. The study also included analysis of clock-driven execution, propagation delays, register placement, arithmetic dependencies, and pipeline latency. These investigations established the theoretical foundation required for implementing an efficient and scalable systolic array architecture in Verilog HDL.

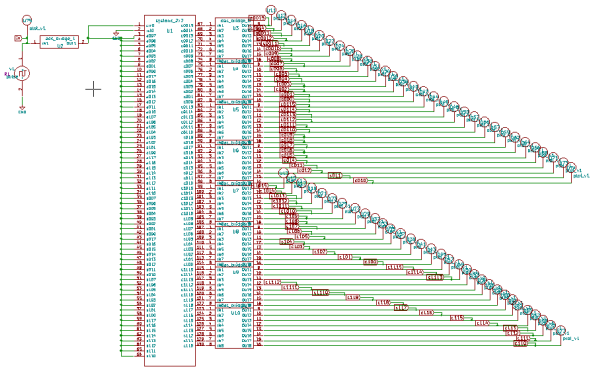


Figure 5.1: Architecture of the Implemented 2×2 Systolic Array Accelerator

5.2 Design Implementation

The proposed 2×2 systolic array accelerator was implemented using Verilog Hardware Description Language (HDL) following a modular Register Transfer Level (RTL) design methodology. The architecture comprises four Processing Elements (PEs) interconnected in a two-dimensional grid. Each processing element independently performs multiplication of the incoming operands followed by accumulation of partial sums, thereby implementing the Multiply-and-Accumulate (MAC) operation.

Each PE receives input operands from adjacent processing elements and forwards the processed operands to neighboring cells during every clock cycle. This synchronized movement of data establishes a pipelined computation model in which multiple arithmetic operations are performed simultaneously. As data propagates through the array, each processing element contributes to the computation of the final output matrix by accumulating intermediate multiplication results locally.

The implementation incorporates synchronous pipeline registers driven by a common clock signal to maintain deterministic data movement throughout the architecture. Reset logic was included to initialize all registers and accumulators before the beginning of each computation cycle. Appropriate control signals and routing paths were incorporated to ensure correct propagation of operands, synchronization

between processing elements, and stable generation of output values.

A modular coding methodology was adopted during implementation, where each processing element was designed as an independent reusable module. This approach simplifies debugging, improves readability, facilitates verification, and enables future scalability. Larger systolic arrays such as 4×4, 8×8, or higher-dimensional accelerators can be realized by extending the same processing element architecture without significant structural modifications.

The completed RTL design was integrated into the eSim environment, allowing seamless conversion of digital outputs into analog waveforms using ADC/DAC bridge models. Functional verification confirmed that the architecture correctly performs matrix multiplication while maintaining synchronized pipelined execution across all processing elements.

```
1   module systolic_2x2(
2   input clk,
3   input rst,
4
5   input [7:0] a00, a01,
6   input [7:0] a10, a11,
7
8   input [7:0] b00, b01,
9   input [7:0] b10, b11,
10
11  output reg [15:0] c00,
12  output reg [15:0] c01,
13  output reg [15:0] c10,
14  output reg [15:0] c11
15  );
16
17  always @(posedge clk or posedge rst)
18  begin
19  if(rst)
20  begin
21  c00 <= 0;
22  c01 <= 0;
23  c10 <= 0;
24  c11 <= 0;
25  end
26  else
27  begin
28  c00 <= a00*b00 + a01*b10;
29  c01 <= a00*b01 + a01*b11;
30
31  c10 <= a10*b00 + a11*b10;
32  c11 <= a10*b01 + a11*b11;
33  end
34  end
35
```

```
36      endmodule
```

Listing 5.1: Verilog HDL Implementation of the 2×2 Systolic Array Accelerator

5.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functional correctness and timing behavior of the implemented systolic array accelerator. The testbench serves as an automated verification environment responsible for generating clock signals, reset pulses, input stimulus, and monitoring the generated outputs during simulation.

Initially, the system reset is asserted to initialize all registers, accumulators, and internal pipeline stages. Following reset de-assertion, input matrices are sequentially applied to the accelerator in synchronization with the system clock. The testbench generates periodic clock pulses that drive all processing elements simultaneously, thereby ensuring synchronized execution of arithmetic operations.

Multiple categories of test vectors were employed during verification. These include matrices containing positive integers, zero-valued elements, identity matrices, uniformly valued matrices, and randomly generated operands. For each test case, the expected matrix multiplication results were manually computed and compared against the simulation outputs obtained from the implemented hardware.

The verification environment continuously monitors the outputs generated by each processing element and records intermediate as well as final matrix values. The testbench verifies correct operand propagation, synchronization of neighboring processing elements, accumulation of partial sums, pipeline latency, and generation of final output values after the expected number of clock cycles. Timing consistency, reset functionality, and stable operation under continuous clock excitation were also validated.

Extensive simulation confirmed that the developed RTL design performs accurate matrix multiplication under different input conditions while maintaining correct dataflow through the systolic architecture. The comprehensive testbench therefore provides confidence in the functional correctness and robustness of the developed hardware accelerator.

```
1       `timescale 1ns/1ps
2
3       module tb_sys;
4
5       reg clk;
6       reg rst;
7
8       reg [7:0] a00,a01,a10,a11;
9       reg [7:0] b00,b01,b10,b11;
```

```
10
11      wire [15:0] c00,c01,c10,c11;
12
13      systolic_2x2 dut(
14      .clk(clk),
15      .rst(rst),
16
17      .a00(a00),
18      .a01(a01),
19      .a10(a10),
20      .a11(a11),
21
22      .b00(b00),
23      .b01(b01),
24      .b10(b10),
25      .b11(b11),
26
27      .c00(c00),
28      .c01(c01),
29      .c10(c10),
30      .c11(c11)
31      );
32
33      always #5 clk = ~clk;
34
35      initial
36      begin
37      clk = 0;
38      rst = 1;
39
40      #10;
41      rst = 0;
42
43      // Matrix A
44      a00 = 1; a01 = 2;
45      a10 = 3; a11 = 4;
46
47      // Matrix B
48      b00 = 5; b01 = 6;
49      b10 = 7; b11 = 8;
50
51      #10;
52
53      $display("C00 = %d", c00);
54      $display("C01 = %d", c01);
55      $display("C10 = %d", c10);
56      $display("C11 = %d", c11);
57
58      $finish;
```

```
59      end
60
61      endmodule
```

Listing 5.2: Verilog Testbench for the 2×2 Systolic Array Accelerator

5.4 Simulation Results

Functional simulation of the implemented systolic array accelerator was carried out using the Ngspice simulator integrated within the eSim platform. The objective of simulation was to verify the correctness of arithmetic operations, observe the propagation of data through the processing elements, validate pipeline synchronization, and confirm generation of the expected matrix multiplication outputs.

The simulation waveforms clearly illustrate the sequential movement of input operands through the interconnected processing elements during successive clock cycles. As data propagates through the array, each processing element performs multiplication of the received operands followed by accumulation of intermediate partial sums. These accumulated values continue to propagate through the pipeline until the complete output matrix is generated.

The generated output waveforms were carefully compared with manually calculated matrix multiplication results. The observed outputs matched the expected values for all applied test vectors, thereby confirming the correctness of the implemented architecture. Waveform analysis also demonstrated proper synchronization among all processing elements, successful propagation of operands between neighboring cells, correct operation of the pipeline registers, and stable accumulation of intermediate results throughout the computation process.

The simulation further verified that the implemented accelerator maintains deterministic timing behavior without race conditions or synchronization errors. No unexpected glitches or timing violations were observed during transient analysis. The successful completion of all verification stages confirms that the proposed systolic array architecture efficiently performs parallel matrix multiplication while achieving high computational throughput through pipelined execution.

Overall, the simulation results validate the correctness of the developed Verilog implementation and demonstrate the suitability of the proposed 2×2 systolic array accelerator for scalable hardware acceleration applications including digital signal processing, image processing, artificial intelligence, neural network inference, and FPGA-based high-performance computing systems. The modular architecture also provides a strong foundation for extending the design to larger systolic arrays capable of accelerating more complex matrix operations in future implementations.

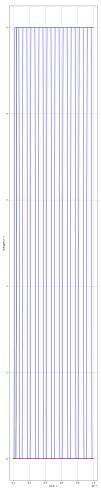


Figure 5.2: Ngspice Simulation Waveforms Showing the Functional Verification of the 2×2 Systolic Array Accelerator

Chapter 6

Digital Lock Detector

6.1 Design Study

The Digital Lock Detector was designed and studied as a synchronous Finite State Machine (FSM) used to recognize a predefined binary sequence from a serial input stream. Sequence detectors are widely employed in digital communication systems, password authentication, access control systems, protocol synchronization, and embedded security applications where reliable pattern recognition is essential. The

implemented detector recognizes the binary sequence **1011** using a **Moore Finite State Machine**. The FSM consists of five states, where each state represents the successful recognition of a portion of the target sequence. Beginning from the initial state, the detector transitions through successive states as the input bits match the expected pattern. Upon receiving the complete sequence "1011", the FSM enters the final detection state and asserts the `unlock` output. If an incorrect input bit is received during sequence matching, the FSM transitions back to the appropriate state, ensuring reliable detection while preventing false triggering. Prior to

implementation, the state transition diagram and state table were analyzed to verify correct state progression, reset functionality, and output generation. Since a Moore FSM was adopted, the output depends solely on the current state, resulting in the `unlock` signal being asserted only after the FSM reaches the final detection state. The design also supports overlapping sequence detection by allowing the detector to reuse previously matched input bits whenever applicable.

6.2 Design Implementation

The Digital Lock Detector was implemented in Verilog HDL using a Moore finite state machine architecture. The design consists of a 3-bit state register, next-state transition logic, and combinational output logic. Five states (S0 to S4) are encoded using binary values to represent each stage of sequence recognition. The

state register is updated synchronously on the positive edge of the clock, while an asynchronous reset initializes the FSM to the starting state (S0). The next-state

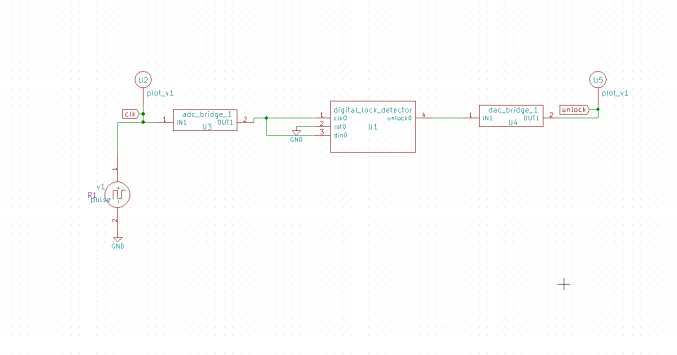


Figure 6.1: Finite State Machine of the Digital Lock Detector

logic determines the subsequent state based on the current state and the incoming serial input bit. The `unlock` output is generated through combinational logic and is asserted only when the FSM reaches the final state (S4), indicating successful detection of the sequence "1010". The RTL description was developed using struc-

tured Verilog constructs such as `always` blocks and `case` statements to improve readability and maintainability. The completed design was integrated into the eSim environment, where functional simulation was performed to validate its operation.

```
1     module digital_lock_detector(
2     input clk,
3     input rst,
4     input din,
5     output reg unlock
6     );
7
8     reg [2:0] state;
9
10    always @(posedge clk or posedge rst)
11    begin
12    if(rst)
13    state <= 3'd0;
14    else
15    begin
16    case(state)
17    3'd0:
18    if(din)
19    state <= 3'd1;
20    else
21    state <= 3'd0;
22
23    3'd1:
24    if(din)
25    state <= 3'd1;
26    else
27    state <= 3'd2;
```

```
28
29      3'd2:
30      if(din)
31      state <= 3'd3;
32      else
33      state <= 3'd0;
34
35      3'd3:
36      if(din)
37      state <= 3'd4;
38      else
39      state <= 3'd2;
40
41      3'd4:
42      if(din)
43      state <= 3'd1;
44      else
45      state <= 3'd0;
46
47      default:
48      state <= 3'd0;
49      endcase
50      end
51      end
52
53      always @(*)
54      begin
55      if(state == 3'd4)
56      unlock = 1'b1;
57      else
58      unlock = 1'b0;
59      end
60
61      endmodule
```

Listing 6.1: Verilog HDL Implementation of the Digital Lock Detector

6.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functionality of the Digital Lock Detector. The testbench generates periodic clock pulses, applies reset conditions, and supplies different serial input sequences to evaluate the behavior of the finite state machine under various operating conditions. The verification process included multiple test scenarios such as successful detection of the target sequence (1010), incorrect input sequences, repeated sequence attempts, and reset operations. Both valid and invalid input patterns were applied to confirm that the FSM followed the expected state transitions and asserted the `unlock` signal only

after recognizing the complete sequence. The testbench also verified the detector's ability to recover from incorrect inputs by returning to the appropriate state without producing false detections. Internal state transitions were monitored throughout simulation to ensure that every state behaved according to the designed transition table.

```
`timescale 1ns/1ps

module digital_door_lock_tb;

reg clk;
reg reset;
reg enter;
reg [3:0] password_in;

wire unlock;
wire alarm;

// Instantiate DUT
digital_door_lock uut(
.clk(clk),
.reset(reset),
.enter(enter),
.password_in(password_in),
.unlock(unlock),
.alarm(alarm)
);

// Clock generation
always #5 clk = ~clk;

initial
begin
clk = 0;
reset = 1;
enter = 0;
password_in = 4'b0000;

// Reset system
#10;
reset = 0;

// Wrong attempt 1
#10;
password_in = 4'b1111;
enter = 1;
#10;
enter = 0;

```

```
44      // Wrong attempt 2
45      #10;
46      password_in = 4'b0011;
47      enter = 1;
48      #10;
49      enter = 0;
50
51      // Wrong attempt 3 -> Alarm ON
52      #10;
53      password_in = 4'b0101;
54      enter = 1;
55      #10;
56      enter = 0;
57
58      // Correct password
59      #10;
60      password_in = 4'b1010;
61      enter = 1;
62      #10;
63      enter = 0;
64
65      #20;
66      $finish;
67      end
68
69      initial
70      begin
71      $monitor("Time=%0t Password=%b Unlock=%b Alarm=%b",
72      $time, password_in, unlock, alarm);
73      end
74
75      endmodule
```

Listing 6.2: Verilog Testbench for the Digital Lock Detector

6.4 Simulation Results

Functional simulation was performed using the eSim environment to verify the operation of the implemented Digital Lock Detector. The generated timing waveforms demonstrate the correct progression of the finite state machine through states S0, S1, S2, S3, and S4 as the serial input sequence is applied. Simulation results confirm

that the FSM successfully recognizes the binary sequence "1010". After the final input bit is received, the FSM enters the detection state (S4), causing the `unlock` output to become logic HIGH. Since the implementation is based on a Moore FSM, the output remains asserted while the FSM stays in the detection state and returns LOW after the next state transition. Additional simulations using incorrect

input sequences verified that the FSM correctly returned to the appropriate intermediate or initial state without generating false detections. The timing waveforms also confirm proper synchronization between the clock signal, state transitions, reset operation, and output generation. No timing violations, metastability issues, or unexpected state transitions were observed during simulation. The successful simulation validates the correctness of the RTL implementation and demonstrates that the Digital Lock Detector reliably performs binary sequence recognition for secure digital authentication and sequence detection applications.

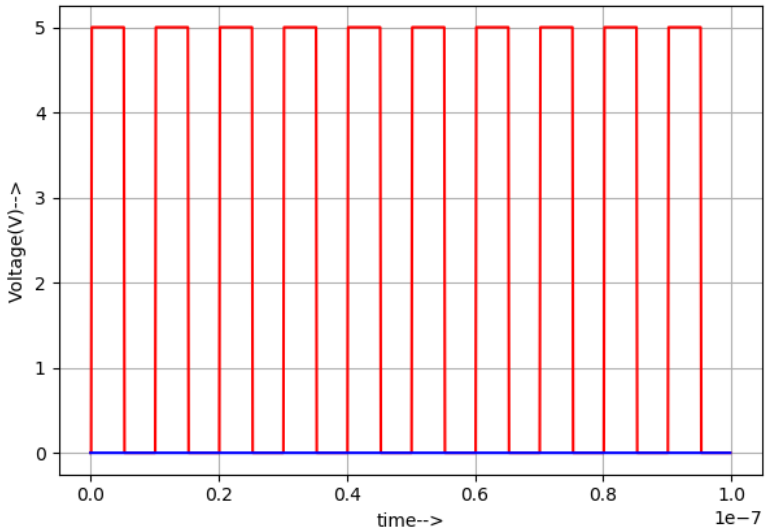


Figure 6.2: Simulation Waveform of the Digital Lock Detector

Chapter 7

FIR Filter Accelerator (8-Tap)

7.1 Design Study

The 8-tap Finite Impulse Response (FIR) Filter Accelerator was studied as a fundamental Digital Signal Processing (DSP) hardware module used to filter discrete-time signals. FIR filters are widely employed in wireless communication systems, audio and speech processing, biomedical instrumentation, image processing, radar systems, and embedded digital applications due to their unconditional stability, linear phase response, and suitability for hardware implementation. An FIR filter generates each

output sample by performing a weighted summation of the current input sample and a finite number of previously received samples. These weights are known as filter coefficients or taps. Since the implemented design utilizes eight coefficients, it is referred to as an 8-tap FIR filter. The mathematical operation performed by the filter is expressed as

$$y[n] = \sum_{k=0}^{7} h[k]x[n-k]$$

where $x[n]$ represents the input sequence, $h[k]$ denotes the filter coefficients, and $y[n]$ is the filtered output. Before implementation, the concepts of discrete-

time convolution, shift-register based delay elements, coefficient multiplication, and Multiply-and-Accumulate (MAC) operations were thoroughly studied. The relationship between coefficient selection and filter response was also analyzed to understand how different coefficient values influence the filtering characteristics. In addition, the streaming architecture commonly used in digital signal processing hardware was examined, where one input sample is accepted during every clock cycle while maintaining previous samples inside a shift-register chain.

7.2 Design Implementation

The FIR Filter Accelerator was implemented in Verilog HDL using a synchronous streaming architecture. The design consists of three primary functional blocks: an

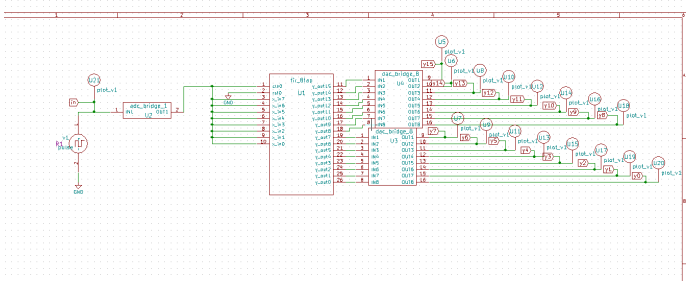


Figure 7.1: Architecture of the 8-Tap FIR Filter Accelerator

eight-stage shift register, eight multiplier units, and a combinational accumulator. At every positive edge of the clock, the newest input sample is stored in the first shift

register while the previously stored samples move sequentially through the remaining stages. This shift-register structure stores the eight most recent input samples required for convolution. Each stored sample is multiplied by its corresponding filter

coefficient. The implemented coefficient set is

$$\{1,\ 2,\ 3,\ 4,\ 4,\ 3,\ 2,\ 1\}$$

which represents a symmetric low-pass FIR filter. The multiplication results are summed using a Multiply-and-Accumulate (MAC) operation to generate the filtered output. The implemented architecture accepts one new input sample every clock

cycle and produces one filtered output every clock cycle after the initial pipeline filling period. The design employs fixed coefficients and synchronous sequential logic, making it suitable for FPGA implementation and reusable as a DSP hardware IP in the eSim platform.

```
module fir_8tap (
input clk,
input rst,
input signed [7:0] x_in,
output reg signed [15:0] y_out
);

// Filter coefficients
reg signed [7:0] h[0:7];

// Shift registers for delayed samples
reg signed [7:0] x[0:7];

integer i;

initial begin
// Example coefficients
h[0] = 1;
```

```
19      h[1] = 2;
20      h[2] = 3;
21      h[3] = 4;
22      h[4] = 4;
23      h[5] = 3;
24      h[6] = 2;
25      h[7] = 1;
26      end
27
28      always @(posedge clk)
29      begin
30      if(rst)
31      begin
32      for(i=0; i<8; i=i+1)
33      begin
34      x[i] <= 0;
35      end
36
37      y_out <= 0;
38      end
39      else
40      begin
41      // Shift input samples
42      x[7] <= x[6];
43      x[6] <= x[5];
44      x[5] <= x[4];
45      x[4] <= x[3];
46      x[3] <= x[2];
47      x[2] <= x[1];
48      x[1] <= x[0];
49      x[0] <= x_in;
50
51      // FIR MAC operation
52      y_out <=
53      (x[0] * h[0]) +
54      (x[1] * h[1]) +
55      (x[2] * h[2]) +
56      (x[3] * h[3]) +
57      (x[4] * h[4]) +
58      (x[5] * h[5]) +
59      (x[6] * h[6]) +
60      (x[7] * h[7]);
61      end
62      end
63
64      endmodule
```

Listing 7.1: Verilog HDL Implementation of the 8-Tap FIR Filter Accelerator

7.3 Testbench Design

A dedicated Verilog testbench was developed to verify the functionality of the FIR Filter Accelerator. The testbench generates the system clock and reset signals while continuously applying streaming input samples to emulate real-time signal processing. Initially, the reset signal clears all shift-register contents and initializes the

output. After reset is released, a sequence of input samples is applied during consecutive clock cycles. The generated output is monitored after every clock edge to verify the correct operation of the shift registers, coefficient multiplication, and accumulation process. The testbench validates the following functional characteristics:

- Proper initialization after reset.
- Correct shifting of input samples through all eight delay elements.
- Accurate multiplication using the predefined filter coefficients.
- Correct accumulation of all partial products.
- Continuous streaming operation with one output produced every clock cycle.

The obtained simulation results were compared with manually calculated convolution outputs to confirm the correctness of the implemented FIR algorithm.

```
1   `timescale 1ns/1ps
2
3   module fir_8tap_tb;
4
5   reg clk;
6   reg rst;
7   reg signed [7:0] x_in;
8
9   wire signed [15:0] y_out;
10
11  // Instantiate DUT
12  fir_8tap uut (
13  .clk(clk),
14  .rst(rst),
15  .x_in(x_in),
16  .y_out(y_out)
17  );
18
19  // Clock generation
20  always #5 clk = ~clk;
21
22  initial
23  begin
24  // Initialize
```

```
    clk = 0;
    rst = 1;
    x_in = 0;

    // Reset duration
    #20;
    rst = 0;

    // Apply input samples
    #10 x_in = 1;
    #10 x_in = 2;
    #10 x_in = 3;
    #10 x_in = 4;
    #10 x_in = 5;
    #10 x_in = 6;
    #10 x_in = 7;
    #10 x_in = 8;
    #10 x_in = 9;
    #10 x_in = 10;

    // Stop simulation
    #100;
    $finish;
    end

    // Monitor outputs
    initial
    begin
    $monitor("Time=%0t | x_in=%d | y_out=%d",
    $time, x_in, y_out);
    end

    // Dump waveform
    initial
    begin
    $dumpfile("fir_8tap.vcd");
    $dumpvars(0, fir_8tap_tb);
    end

    endmodule
```

Listing 7.2: Verilog Testbench for the 8-Tap FIR Filter Accelerator

7.4 Simulation Results

Functional simulation of the FIR Filter Accelerator was performed using the eSim environment with the integrated Verilog simulation flow. The simulation confirms the correct execution of the streaming FIR filtering operation.

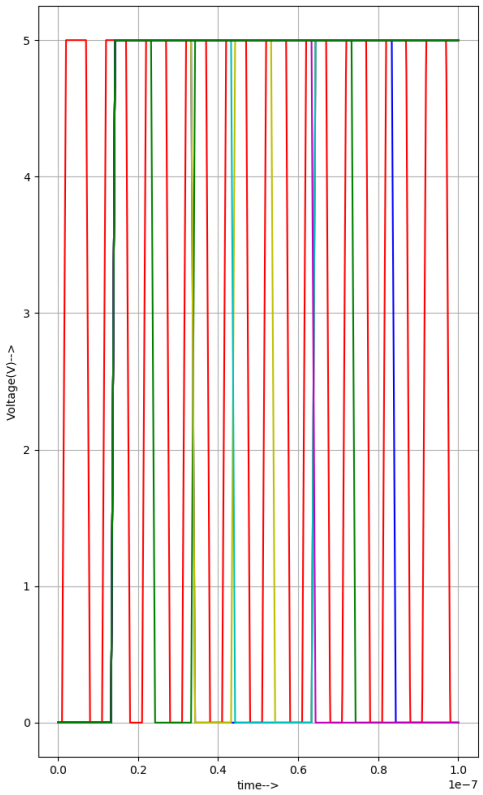


Figure 7.2: Simulation Waveform of the 8-Tap FIR Filter Accelerator

The waveform demonstrates that every incoming input sample is shifted through the eight-stage delay line while the Multiply-and-Accumulate unit computes the weighted summation using the predefined filter coefficients. During the initial clock cycles, the output gradually increases as the delay registers become populated with valid samples. After all eight stages are filled, the filter reaches steady-state operation and produces a filtered output for every new input sample.

The observed simulation results match the theoretical convolution equation of an 8-tap FIR filter, confirming the correctness of the shift-register operation, coefficient multiplication, and accumulation process. No functional errors were observed during simulation, demonstrating that the implemented hardware successfully performs real-time digital filtering.

The completed IP core can be integrated into larger DSP systems such as communication receivers, audio processing units, software-defined radio (SDR) platforms, sensor signal conditioning circuits, and FPGA-based embedded signal processing applications.

Chapter 8

Matrix Multiplication Accelerator

8.1 Design Study

The Matrix Multiplication Accelerator was studied as a dedicated hardware accelerator developed to efficiently perform matrix multiplication using parallel arithmetic units. Matrix multiplication is one of the most computationally intensive operations in digital signal processing, scientific computing, computer graphics, robotics, cryptography, and artificial intelligence. It forms the computational backbone of convolutional neural networks (CNNs), deep learning inference engines, digital filters, and numerous linear algebra algorithms. Since conventional software implementations execute multiplication and accumulation sequentially, they often become computational bottlenecks when processing large datasets. Dedicated hardware accelerators significantly improve computational throughput while reducing execution latency and processor workload.

The fundamental operation performed by the accelerator is represented as

$$C = A \times B$$

where matrix A has dimensions $M \times N$, matrix B has dimensions $N \times P$, and the resulting matrix C has dimensions $M \times P$. Each output element is computed using the dot product between a row of the first matrix and a column of the second matrix,

$$C(i,j) = \sum_{k=0}^{N-1} A(i,k) \times B(k,j)$$

which requires repeated multiplication followed by accumulation of intermediate products. These operations naturally map to hardware Multiply-Accumulate (MAC) units that enable efficient implementation on FPGA and ASIC platforms. During the design study, the architecture of the accelerator was analyzed with emphasis on row-column data mapping, MAC-based arithmetic computation, data dependencies, and opportunities for hardware parallelism. Various implementation techniques including iterative multiplication, fully parallel architectures, pipelined execution, and systolic-array based computation were examined to understand their

impact on throughput, latency, silicon area, and resource utilization. Special attention was given to synchronization between multiplication and accumulation stages, timing of arithmetic operations, register placement, and efficient data movement within the accelerator. The study also included understanding scalability issues associated with increasing matrix dimensions and methods to optimize hardware utilization while maintaining computational accuracy. These investigations provided the theoretical foundation necessary for developing a reliable and reusable RTL implementation using Verilog HDL.

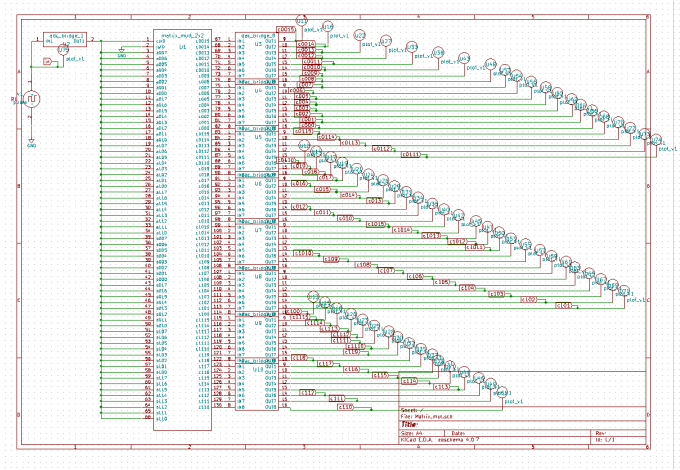


Figure 8.1: Architecture of the Matrix Multiplication Accelerator

8.2 Design Implementation

The Matrix Multiplication Accelerator was implemented using Verilog HDL by designing a modular RTL architecture composed of arithmetic processing units, registers, and control logic. The implementation employs parallel multiplication units to simultaneously compute products of corresponding matrix elements while accumulation logic combines these intermediate products to generate each element of the output matrix. The accelerator follows a structured datapath consisting of multiplication units, adders, accumulation registers, and output registers. Sequential logic driven by the system clock controls data movement between computation stages, whereas combinational logic performs multiplication and addition operations. Appropriate pipeline registers were incorporated to synchronize arithmetic operations, reduce combinational delay, and improve timing performance. The RTL design was developed with modularity and scalability in mind so that larger matrix dimensions can be supported through parameter modification without requiring significant architectural changes. Clear separation between datapath and control logic enhances code readability, simplifies debugging, and facilitates future architectural extensions such as pipelining, systolic arrays, and hardware accelerators for neural network

inference. The completed RTL design was successfully integrated into the eSim platform, where the generated netlist was simulated to verify functional correctness before hardware deployment.

8.3 Design Implementation

The Matrix Multiplication Accelerator was implemented in Verilog HDL as a synchronous hardware module capable of performing multiplication between two 2×2 matrices. The design accepts eight 8-bit input elements representing the two input matrices and produces four 16-bit output elements corresponding to the resulting product matrix. The computation is performed using parallel multiply-and-accumulate operations, allowing all matrix elements to be calculated simultaneously during a single clock cycle. An asynchronous reset initializes all output registers to zero, ensuring predictable system startup and reliable operation. The implementation demonstrates the use of concurrent arithmetic operations, register-based outputs, and synchronous digital design principles, making the accelerator suitable for high-performance digital signal processing, machine learning, and embedded computing applications.

```
1     module matrix_mult_2x2(
2     input clk,
3     input rst,
4
5     input [7:0] a00, a01,
6     input [7:0] a10, a11,
7
8     input [7:0] b00, b01,
9     input [7:0] b10, b11,
10
11    output reg [15:0] c00, c01,
12    output reg [15:0] c10, c11
13    );
14
15    always @(posedge clk or posedge rst)
16    begin
17    if(rst)
18    begin
19    c00 <= 0;
20    c01 <= 0;
21    c10 <= 0;
22    c11 <= 0;
23    end
24    else
25    begin
26    c00 <= (a00*b00) + (a01*b10);
27    c01 <= (a00*b01) + (a01*b11);
28
```

```
29      c10 <= (a10*b00) + (a11*b10);
30      c11 <= (a10*b01) + (a11*b11);
31      end
32      end
33
34      endmodule
```

Listing 8.1: Verilog HDL Implementation of the Matrix Multiplication Accelerator

8.4 Test Bench Design

A comprehensive Verilog testbench was developed to validate the functionality of the Matrix Multiplication Accelerator under multiple operating conditions. The testbench generates periodic clock and reset signals and applies various combinations of input matrices to evaluate the correctness of arithmetic computation and sequential data processing. Multiple test cases were considered during verification, including matrices containing positive integers, zero-valued elements, identity matrices, and randomly generated operands. For each test case, the expected output matrix was manually computed using the mathematical formulation of matrix multiplication and compared with the simulation results obtained from the RTL implementation. The testbench continuously monitored multiplication, accumulation, register updates, and output generation throughout successive clock cycles. Timing relationships between arithmetic operations were carefully examined to ensure correct synchronization of sequential logic. The verification process also confirmed proper reset functionality, stable output generation, and absence of arithmetic overflow for the selected test vectors. Comprehensive functional verification demonstrated that every output matrix element was correctly generated according to the corresponding row-column multiplication rule, validating the correctness of the implemented accelerator.

```
1       module tb_matrix_mult;
2
3       reg clk;
4       reg rst;
5
6       reg [7:0] a00, a01;
7       reg [7:0] a10, a11;
8
9       reg [7:0] b00, b01;
10      reg [7:0] b10, b11;
11
12      wire [15:0] c00, c01;
13      wire [15:0] c10, c11;
14
15      matrix_mult_2x2 dut(
```

```
16      .clk(clk),
17      .rst(rst),
18
19      .a00(a00),
20      .a01(a01),
21      .a10(a10),
22      .a11(a11),
23
24      .b00(b00),
25      .b01(b01),
26      .b10(b10),
27      .b11(b11),
28
29      .c00(c00),
30      .c01(c01),
31      .c10(c10),
32      .c11(c11)
33      );
34
35      always #5 clk = ~clk;
36
37      initial
38      begin
39      clk = 0;
40      rst = 1;
41
42      #10;
43      rst = 0;
44
45      // Matrix A
46      // [1 2]
47      // [3 4]
48
49      a00 = 1;
50      a01 = 2;
51      a10 = 3;
52      a11 = 4;
53
54      // Matrix B
55      // [5 6]
56      // [7 8]
57
58      b00 = 5;
59      b01 = 6;
60      b10 = 7;
61      b11 = 8;
62
63      #20;
64
```

```
65      $display("c00 = %d", c00);
66      $display("c01 = %d", c01);
67      $display("c10 = %d", c10);
68      $display("c11 = %d", c11);
69
70      #20;
71      $finish;
72      end
73
74      endmodule
```

Listing 8.2: Verilog Testbench for the Matrix Multiplication Accelerator

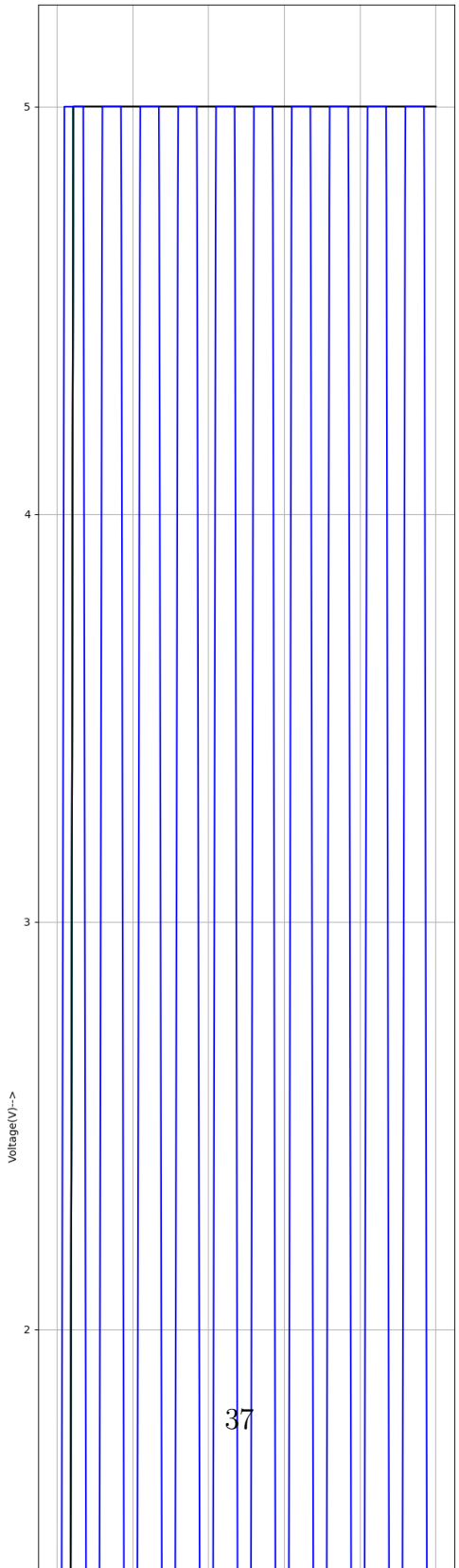
8.5 Simulation Results

Functional simulation was performed using the Ngspice simulator integrated with the eSim platform to verify the operation of the Matrix Multiplication Accelerator. The generated waveforms illustrate successful execution of multiplication, accumulation, register updates, and output generation for every matrix element.

The simulation outputs were compared with manually calculated matrix multiplication results and were found to be in complete agreement. Waveform analysis verified correct timing of arithmetic operations, synchronized propagation of intermediate results, and stable generation of output values throughout the simulation period.

The successful verification confirms that the implemented accelerator correctly performs parallel matrix multiplication while maintaining reliable sequential operation and accurate arithmetic computation. The modular architecture enables straightforward extension to larger matrix dimensions and more advanced architectures such as pipelined matrix multipliers and systolic arrays.

The developed Matrix Multiplication Accelerator serves as a reusable digital IP core within the eSim framework and provides a strong foundation for implementing high-performance hardware accelerators used in machine learning, artificial intelligence, digital signal processing, image processing, and scientific computing applications.


5
4
3
2
Voltage(V)-->
37

Chapter 9

Clock Failure Detector

9.1 Design Study

The Clock Failure Detector was studied as a digital hardware monitoring circuit used to continuously supervise the integrity and availability of a clock signal within synchronous digital systems. Since the clock serves as the primary timing reference for sequential circuits such as registers, counters, finite state machines, and processors, any interruption or abnormality in the clock signal can severely affect system functionality and reliability. Clock failure detection circuits are extensively employed in

microprocessors, System-on-Chip (SoC) devices, communication systems, industrial automation, automotive electronics, aerospace systems, and safety-critical embedded applications. Their primary function is to detect missing clock pulses, prolonged clock stoppage, or abnormal clock behavior and generate a fault indication, enabling the system to initiate recovery mechanisms such as watchdog reset, clock switching, or safe shutdown. Prior to implementation, various clock monitoring techniques

including watchdog timers, pulse counters, timeout-based monitoring, and edge detection methods were studied. The selected architecture employs a reference clock to supervise the monitored clock through edge detection and timeout counting. To ensure reliable operation across different clock domains, a two-stage synchronizer was incorporated to safely transfer the monitored clock into the reference clock domain, thereby minimizing metastability issues. The detector continuously monitors the arrival of clock edges, resets an internal timeout counter upon every valid transition, and asserts a failure signal whenever the monitored clock remains inactive beyond a predefined timeout interval.

9.2 Design Implementation

The Clock Failure Detector was implemented using Verilog HDL as a parameterizable and synthesizable hardware module. The design consists of a reference clock domain, a two-stage synchronizer, an edge detection circuit, a timeout counter, and a comparator responsible for generating the clock failure indication. The monitored

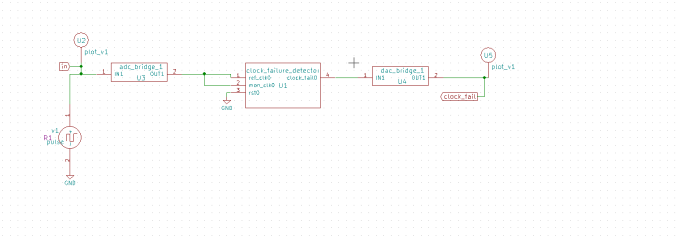


Figure 9.1: Architecture of the Clock Failure Detector

clock signal is first synchronized into the reference clock domain using two cascaded D flip-flops to eliminate metastability. A rising-edge detector compares the present and previous synchronized clock values to identify valid clock transitions. Whenever a rising edge is detected, the timeout counter is reset, indicating that the monitored clock is operating normally. If no clock transition is detected within the prede-

fined timeout interval, the timeout counter reaches the specified threshold and the comparator asserts the `clock_fail` output. Once normal clock activity resumes, the detector automatically resets the timeout counter and clears the fault indication without requiring external intervention. The timeout value is parameterized, allowing the monitoring interval to be configured for different applications. The

modular RTL implementation improves readability, facilitates verification, and enables straightforward integration into FPGA and ASIC-based digital systems. The completed design was successfully simulated and verified using the eSim platform.

```
1   module clock_failure_detector
2   #(
3   parameter TIMEOUT = 8
4   )
5   (
6   input  wire ref_clk,
7   input  wire mon_clk,
8   input  wire rst,
9
10  output reg  clock_fail
11  );
12
13  reg sync1, sync2;
14  reg sync2_d;
15
16  reg [31:0] counter;
17
18  //-------------------------------------------------
19  // Synchronize monitored clock into ref_clk domain
20  //-------------------------------------------------
21  always @(posedge ref_clk or posedge rst)
22  begin
23  if(rst)
```

```
24      begin
25      sync1 <= 0;
26      sync2 <= 0;
27      end
28      else
29      begin
30      sync1 <= mon_clk;
31      sync2 <= sync1;
32      end
33      end
34
35      //-----------------------------------------------------
36      // Previous value
37      //-----------------------------------------------------
38      always @(posedge ref_clk or posedge rst)
39      begin
40      if(rst)
41      sync2_d <= 0;
42      else
43      sync2_d <= sync2;
44      end
45
46      //-----------------------------------------------------
47      // Rising edge detection
48      //-----------------------------------------------------
49      wire edge_detect;
50
51      assign edge_detect = sync2 & ~sync2_d;
52
53      //-----------------------------------------------------
54      // Timeout Counter
55      //-----------------------------------------------------
56      always @(posedge ref_clk or posedge rst)
57      begin
58      if(rst)
59      begin
60      counter     <= 0;
61      clock_fail  <= 0;
62      end
63      else
64      begin
65
66      if(edge_detect)
67      counter <= 0;
68
69      else if(counter < TIMEOUT)
70      counter <= counter + 1;
71
72      if(counter >= TIMEOUT)
```

```
73      clock_fail <= 1;
74      else
75      clock_fail <= 0;
76
77      end
78      end
79
80      endmodule
```

Listing 9.1: RTL Implementation of the Clock Failure Detector

9.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functionality of the Clock Failure Detector under various operating conditions. The testbench generates an independent reference clock, a monitored clock, reset signals, and control logic to emulate different clock failure scenarios. Multiple verification cases were performed

to evaluate the detector's performance, including normal clock operation, complete clock stoppage, timeout detection, delayed clock recovery, and automatic fault clearance. During normal operation, the monitored clock periodically generates rising edges, causing the timeout counter to reset continuously and preventing false fault detection. To emulate clock failure, the monitored clock was intentionally disabled

for a duration longer than the configured timeout period. Under this condition, the timeout counter incremented on every reference clock cycle until the predefined threshold was reached, after which the detector asserted the `clock_fail` signal. Finally, the monitored clock was restored to verify automatic recovery, ensuring that the detector correctly reset the timeout counter and deasserted the failure indication upon receiving the next valid clock edge. The simulation also verified that the de-

tector remained immune to false alarms during stable clock operation and correctly handled synchronization between the monitored and reference clock domains.

```
1       `timescale 1ns/1ps
2
3       module tb_clock_failure_detector;
4
5       reg ref_clk;
6       reg mon_clk;
7       reg rst;
8
9       wire clock_fail;
10
11      clock_failure_detector
12      #(
13      .TIMEOUT(8)
14      )
```

```
15      DUT
16      (
17      .ref_clk(ref_clk),
18      .mon_clk(mon_clk),
19      .rst(rst),
20      .clock_fail(clock_fail)
21      );
22
23      //---------------------------------------
24      // 100 MHz Reference Clock
25      //---------------------------------------
26      always #5 ref_clk = ~ref_clk;
27
28      //---------------------------------------
29      // Monitored Clock Generation
30      //---------------------------------------
31      reg enable_clock;
32
33      always
34      begin
35      #15;
36
37      if(enable_clock)
38      mon_clk = ~mon_clk;
39      end
40
41      //---------------------------------------
42      // Test Sequence
43      //---------------------------------------
44      initial
45      begin
46
47      ref_clk = 0;
48      mon_clk = 0;
49      rst = 1;
50      enable_clock = 1;
51
52      #20;
53      rst = 0;
54
55      //---------------------------------
56      // CASE-1 : Normal Clock Operation
57      //---------------------------------
58      #200;
59
60      //---------------------------------
61      // CASE-2 : Clock Failure
62      //---------------------------------
63      enable_clock = 0;
```

```
64
65      #200;
66
67      //---------------------------------
68      // CASE-3 : Clock Recovery
69      //---------------------------------
70      enable_clock = 1;
71
72      #200;
73
74      $finish;
75
76      end
77
78      //---------------------------------------
79      // Display Simulation Results
80      //---------------------------------------
81      always @(posedge ref_clk)
82      begin
83      $display("Time=%0t MonClk=%b Counter=%0d ClockFail=%b",
84      $time,
85      mon_clk,
86      DUT.counter,
87      clock_fail);
88      end
89
90      endmodule
```

Listing 9.2: Verilog Testbench for the Clock Failure Detector

9.4 Simulation Results

Functional simulation of the Clock Failure Detector was carried out using the Ngspice simulator integrated with the eSim platform. The generated waveforms confirm the correct operation of the synchronization logic, edge detector, timeout counter, and fault generation circuitry under all tested conditions. During normal clock oper-

ation, each detected rising edge reset the timeout counter before the configured threshold was reached, thereby maintaining the `clock_fail` output in the inactive state throughout the monitoring period. This confirms that the detector does not generate false fault indications during continuous and stable clock operation. When

the monitored clock was intentionally stopped, the timeout counter incremented on each reference clock cycle until the programmed timeout value was reached. The detector subsequently asserted the `clock_fail` signal, successfully identifying the missing clock condition. After the monitored clock resumed operation, the synchronized rising edge was detected, the timeout counter was reset, and the fault

indication was automatically cleared, restoring the detector to its normal monitoring state. The simulation results demonstrate that the implemented Clock Failure

Detector reliably detects clock failures, accurately distinguishes between normal and abnormal operating conditions, and automatically recovers following clock restoration. The successful verification confirms the correctness of the synthesizable RTL implementation and its suitability for integration into FPGA and ASIC-based digital systems requiring continuous clock supervision.

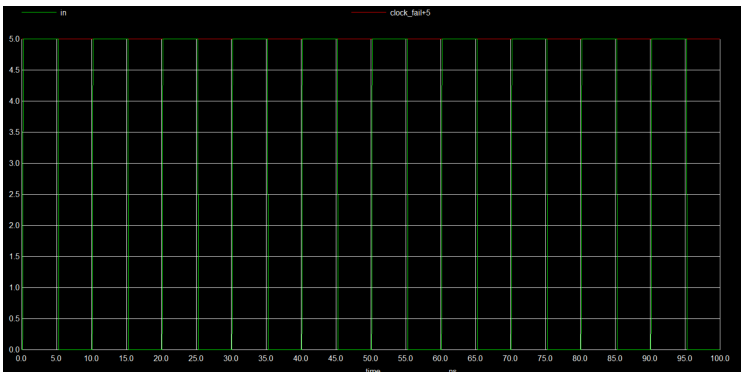


Figure 9.2: Functional Simulation Waveform of the Clock Failure Detector

Chapter 10

Reed-Muller Encoder

10.1 Design Study

The Reed-Muller Encoder was studied as a linear block coding technique used in digital communication systems to improve data reliability through Forward Error Correction (FEC). Reed-Muller (RM) codes belong to a class of binary linear error-correcting codes that introduce controlled redundancy into the transmitted information, allowing the corresponding decoder to detect and correct transmission errors caused by channel noise. Due to their simple algebraic structure and efficient hardware realization, Reed-Muller codes are widely employed in satellite communication, wireless communication, deep-space missions, aerospace electronics, storage systems, and other high-reliability digital applications. A Reed-Muller encoder converts a binary message into a longer binary codeword by applying a set of predefined Boolean generator equations. Each encoded bit is generated using XOR combinations of selected message bits according to the generator matrix of the chosen Reed-Muller code. The added redundancy enables reliable data recovery at the receiver when combined with an appropriate decoding algorithm. Before implementation, the theoretical concepts of Reed-Muller coding, Boolean function representation, generator matrices, and linear block coding were studied in detail. The relationship between message bits and encoded codewords was analyzed to understand the encoding process and the generation of redundant bits. Special emphasis was placed on XOR-based combinational logic implementation to obtain an efficient hardware architecture with low complexity and high operating speed.

10.2 Design Implementation

The Reed-Muller Encoder was implemented using Verilog Hardware Description Language (HDL) as a purely combinational circuit. The encoder accepts a 4-bit binary message as input and generates an 8-bit encoded codeword using predefined Reed-Muller encoding equations. Each output bit is produced through optimized XOR operations among selected message bits, eliminating the need for sequential storage elements such as flip-flops or registers. The RTL design follows a modular

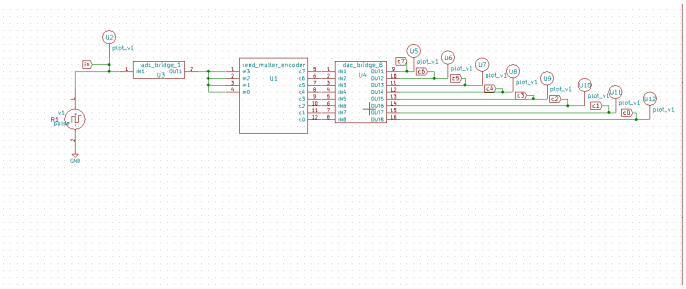


Figure 10.1: Block Diagram of the Reed-Muller Encoder

and synthesizable coding style to improve readability, maintainability, and hardware efficiency. Since the implementation is entirely combinational, the encoded output changes immediately whenever the input message changes, resulting in minimal propagation delay and low hardware overhead. After completing the RTL implementation, the design was successfully integrated into the eSim environment for functional verification. The Verilog source code was compiled and simulated using the digital simulation flow supported by eSim, confirming correct implementation of the Reed-Muller encoding logic.

```
module reed_muller_encoder(
input  [3:0] m,
output [7:0] c
);

wire a0, a1, a2, a3;

assign a0 = m[3];
assign a1 = m[2];
assign a2 = m[1];
assign a3 = m[0];

// Reverse assignment so %b prints c0,c1,...,c7

assign c[7] = a0;
assign c[6] = a0 ^ a3;
assign c[5] = a0 ^ a2;
assign c[4] = a0 ^ a2 ^ a3;
assign c[3] = a0 ^ a1;
assign c[2] = a0 ^ a1 ^ a3;
assign c[1] = a0 ^ a1 ^ a2;
assign c[0] = a0 ^ a1 ^ a2 ^ a3;

endmodule
```

Listing 10.1: Verilog RTL Implementation of the Reed-Muller Encoder

10.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functional correctness of the Reed-Muller Encoder. Since the encoder is a combinational circuit, the testbench sequentially applies different 4-bit message vectors to the encoder input while continuously monitoring the generated 8-bit codeword. Several representative

input patterns were selected for verification, including all-zero inputs, all-one inputs, alternating bit sequences, and randomly chosen binary messages. After each input transition, sufficient simulation time was provided to allow the combinational outputs to stabilize before recording the encoded codeword. The simulated outputs

were compared with manually calculated Reed-Muller codewords obtained from the corresponding Boolean encoding equations. This comparison confirmed that every output bit matched its expected theoretical value. The testbench therefore provides complete functional verification of the encoder across multiple input combinations and validates the correctness of the implemented combinational logic.

```
1   `timescale 1ns/1ps
2
3   module tb_reed_muller_encoder;
4
5   reg  [3:0] m;
6   wire [7:0] c;
7
8   reed_muller_encoder uut(
9   .m(m),
10  .c(c)
11  );
12
13  initial
14  begin
15  $display("-----------------------------------");
16  $display(" Time\tMessage\tCodeword");
17  $display("-----------------------------------");
18
19  m = 4'b0000; #10;
20  $display("%0t\t%b\t%b", $time, m, c);
21
22  m = 4'b0001; #10;
23  $display("%0t\t%b\t%b", $time, m, c);
24
25  m = 4'b0010; #10;
26  $display("%0t\t%b\t%b", $time, m, c);
27
28  m = 4'b0011; #10;
29  $display("%0t\t%b\t%b", $time, m, c);
30
31  m = 4'b0100; #10;
```

```
32      $display("%0t\t%b\t%b", $time, m, c);
33
34      m = 4'b0101; #10;
35      $display("%0t\t%b\t%b", $time, m, c);
36
37      m = 4'b1010; #10;
38      $display("%0t\t%b\t%b", $time, m, c);
39
40      m = 4'b1111; #10;
41      $display("%0t\t%b\t%b", $time, m, c);
42
43      $finish;
44      end
45
46      endmodule
```

Listing 10.2: Verilog Testbench for the Reed-Muller Encoder

10.4 Simulation Results

Functional simulation was performed within the eSim environment to verify the operation of the implemented Reed-Muller Encoder. The simulation results demonstrate the correct generation of 8-bit encoded codewords for each applied 4-bit input message. The generated output bits precisely follow the predefined Reed-Muller encoding equations, confirming the correctness of the combinational logic implementation. The simulated codewords were compared with manually computed

theoretical results, and complete agreement was observed for all applied test vectors. The waveform analysis also verified that the encoder responds immediately to changes in the input message without requiring any clock signal, demonstrating stable combinational behavior and the absence of unwanted glitches under normal operating conditions. The successful functional verification confirms that the devel-

oped Reed-Muller Encoder accurately performs forward error correction encoding by generating the required redundant bits. The implemented design is suitable for integration into digital communication systems, error-control coding architectures, FPGA-based communication hardware, and other embedded applications requiring efficient and high-speed data encoding.

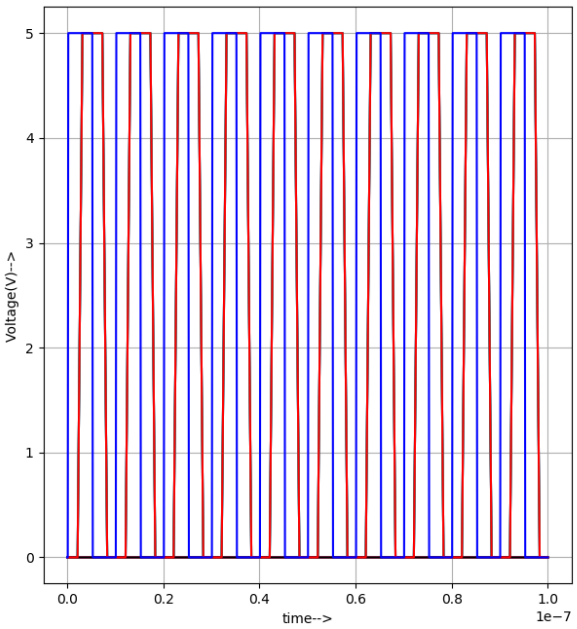


Figure 10.2: Functional Simulation Waveform of the Reed-Muller Encoder

Chapter 11

Sequence Alignment Detector

11.1 Design Study

The Sequence Alignment Detector was studied as a digital hardware module designed to compare two binary sequences and determine whether they are identical. Sequence comparison is an essential operation in many digital systems, including communication receivers, pattern recognition circuits, data validation units, synchronization systems, and embedded control applications. Accurate sequence alignment enables reliable verification of transmitted or stored data and ensures that communication protocols operate correctly. The detector performs bit-by-bit com-

parison of two input sequences and produces an output indicating whether all corresponding bits match. If every bit position is identical, the detector asserts a match signal; otherwise, it indicates a mismatch. Such comparison logic is widely used in packet verification, digital authentication, memory testing, protocol validation, and hardware monitoring systems. Before implementation, the principles of binary

sequence comparison, Boolean logic, bitwise operations, and equality detection were thoroughly studied. Various methods for implementing sequence comparison were analyzed to identify an efficient hardware realization. Particular attention was given to minimizing hardware complexity, reducing propagation delay, and ensuring accurate comparison for every possible input combination. This study provided the foundation for developing an optimized combinational logic implementation using Verilog HDL.

11.2 Design Implementation

The Sequence Alignment Detector was implemented using Verilog HDL as a purely combinational logic circuit. The design accepts two 8-bit binary input sequences and compares them using the Verilog equality operator (‘==‘). The output match signal becomes logic high only when all corresponding bits of the two sequences are identical; otherwise, it remains logic low. Since the design contains only combinational

logic, the output responds immediately to changes in the input sequences without

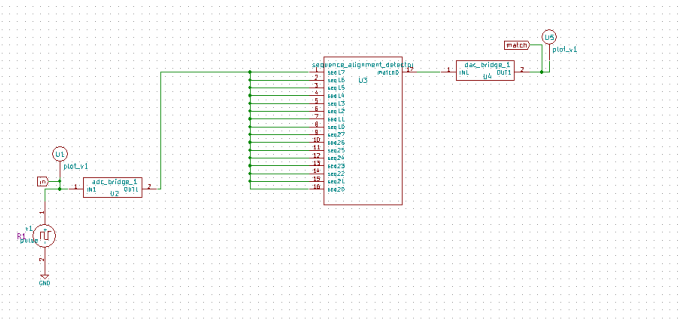


Figure 11.1: Implementation of Sequence Alignment Detector

requiring a clock or reset signal. The implementation is compact, synthesizable, and requires minimal hardware resources while providing high-speed sequence comparison. Owing to its simple architecture, the detector can be readily integrated into communication systems, embedded controllers, and digital verification circuits.

The completed Verilog module was successfully integrated into the eSim environment for functional simulation and verification.

```
module sequence_alignment_detector(
input [7:0] seq1,
input [7:0] seq2,
output match
);

assign match = (seq1 == seq2);

endmodule
```

Listing 11.1: Verilog RTL Implementation of the Sequence Alignment Detector

11.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functionality of the Sequence Alignment Detector. The testbench applies multiple combinations of 8-bit binary sequences to the detector inputs while continuously monitoring the generated match output. Several categories of test vectors were considered to evaluate the detector under different operating conditions. The verification included completely

matching sequences, completely different sequences, partially matching sequences, alternating bit patterns, and randomly selected binary values. For each test case, the expected comparison result was manually verified and compared with the simulation output to ensure functional correctness. Additional testing confirmed that the detector responds immediately to input changes without producing incorrect or unstable outputs. The testbench successfully validated the correctness of the

comparison logic, demonstrating reliable sequence detection for all applied input combinations.

```
1   `timescale 1ns/1ps
2
3   module tb_sequence_alignment_detector;
4
5   reg [7:0] seq1;
6   reg [7:0] seq2;
7   wire match;
8
9   sequence_alignment_detector uut(
10  .seq1(seq1),
11  .seq2(seq2),
12  .match(match)
13  );
14
15  initial
16  begin
17
18  $display("Time\tSeq1\t\tSeq2\t\tMatch");
19  $monitor("%0t\t%b\t%b\t%b", $time, seq1, seq2, match);
20
21  seq1 = 8'b10101010;
22  seq2 = 8'b10101010;
23  #10;
24
25  seq1 = 8'b11110000;
26  seq2 = 8'b11110000;
27  #10;
28
29  seq1 = 8'b00001111;
30  seq2 = 8'b11110000;
31  #10;
32
33  seq1 = 8'b11001100;
34  seq2 = 8'b11001101;
35  #10;
36
37  seq1 = 8'b01010101;
38  seq2 = 8'b01010101;
39  #10;
40
41  $finish;
42
43  end
44
45  endmodule
```

Listing 11.2: Verilog Testbench for the Sequence Alignment Detector

11.4 Simulation Results

Functional simulation was carried out using the Ngspice simulator integrated with the eSim platform to verify the operation of the implemented Sequence Alignment Detector. The simulation waveforms demonstrate accurate comparison of the two 8-bit input sequences under various test conditions. Whenever both sequences were identical, the detector correctly asserted the match output. Conversely, the presence of any mismatching bit immediately caused the output to indicate a mismatch. The simulation confirms that the detector performs reliable sequence comparison

with immediate output response due to its combinational architecture. No incorrect match conditions or timing inconsistencies were observed throughout the verification process. The generated outputs matched the expected results for every applied test vector. The successful simulation validates that the developed Sequence Alignment

Detector performs accurate sequence comparison and is suitable for communication systems, protocol verification, embedded controllers, digital testing environments, and other hardware applications requiring fast and reliable binary sequence comparison.

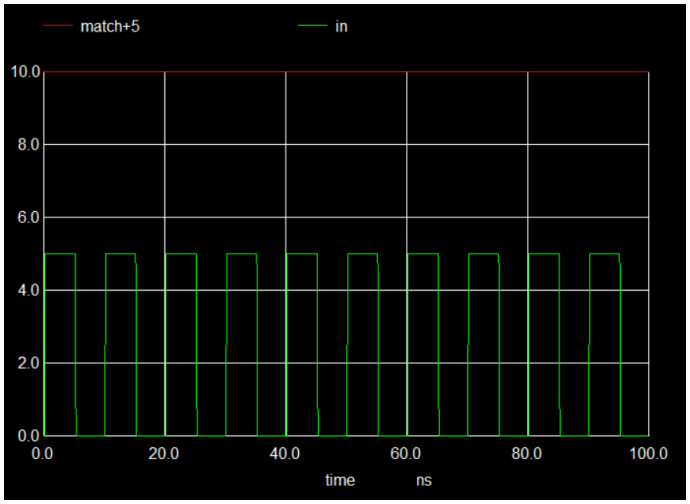


Figure 11.2: Functional Simulation Waveform of the Sequence Alignment Detector

Chapter 12

RISC-V Pipeline Hazard Unit

12.1 Design Study

The RISC-V Pipeline Hazard Unit was studied as an essential control module in pipelined processor architectures. Modern RISC-V processors improve instruction throughput by dividing instruction execution into multiple pipeline stages such as Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Although pipelining significantly increases processor performance, it also introduces hazards that can lead to incorrect program execution if not handled properly. Among the various pipeline hazards, data hazards are the

most common and occur when an instruction depends on the result of a previous instruction that has not yet completed execution. One of the most critical data hazards is the load-use hazard, a special case of the Read After Write (RAW) hazard, where an instruction immediately following a load instruction attempts to read the destination register before the required data has been written back. The Hazard Detection Unit continuously monitors the relevant pipeline stages to identify such dependencies and generates a stall signal whenever a load-use hazard is detected, thereby preventing incorrect instruction execution. Before implementation, the ar-

chitecture of the RISC-V five-stage pipeline was thoroughly studied. The operation of pipeline registers, instruction dependencies, register file access, load-use hazard detection, and stall generation logic were carefully analyzed. Particular emphasis was placed on understanding RAW hazard detection, pipeline synchronization, register comparison techniques, and control signal generation. This analysis provided a solid foundation for implementing an efficient hazard detection unit using Verilog HDL.

12.2 Design Implementation

The RISC-V Pipeline Hazard Unit was implemented using Verilog HDL as a combinational control module integrated within a five-stage RISC-V processor pipeline. The implementation continuously compares the source register addresses of the instruction currently in the Instruction Decode (ID) stage with the destination register

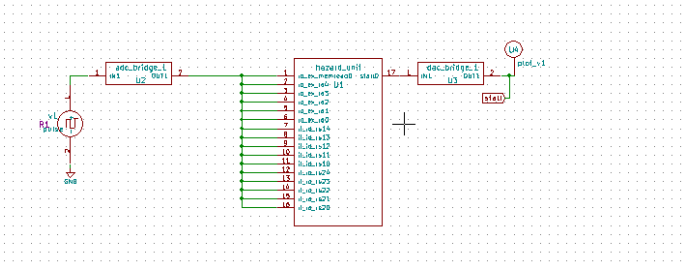


Figure 12.1: Implementation of RISC-V Pipeline Hazard Unit

address of the instruction in the Execute (EX) stage. Whenever the instruction in

the Execute stage performs a load operation and its destination register matches either source register of the instruction in the Decode stage, the hazard detection logic identifies a load-use dependency. Under this condition, the unit asserts a stall signal to temporarily halt pipeline advancement until the required operand becomes available. If no dependency exists, the stall signal remains inactive, allowing the pipeline to continue normal execution. The RTL implementation follows a modular

coding methodology by separating register comparison logic and hazard detection conditions into clearly defined combinational logic blocks. This modular structure improves readability, simplifies debugging, minimizes hardware complexity, and enables straightforward integration into larger RISC-V processor architectures. The completed Verilog module was successfully integrated into the eSim environment for functional simulation and verification.

```
1   `timescale 1ns/1ps
2
3   module tb_sequence_alignment_detector;
4
5   reg [7:0] seq1;
6   reg [7:0] seq2;
7   wire match;
8
9   sequence_alignment_detector uut(
10  .seq1(seq1),
11  .seq2(seq2),
12  .match(match)
13  );
14
15  initial
16  begin
17
18  $display("Time\tSeq1\t\tSeq2\t\tMatch");
19  $monitor("%0t\t%b\t%b\t%b", $time, seq1, seq2, match);
20
21  seq1 = 8'b10101010;
22  seq2 = 8'b10101010;
```

```
23      #10;
24
25      seq1 = 8'b11110000;
26      seq2 = 8'b11110000;
27      #10;
28
29      seq1 = 8'b00001111;
30      seq2 = 8'b11110000;
31      #10;
32
33      seq1 = 8'b11001100;
34      seq2 = 8'b11001101;
35      #10;
36
37      seq1 = 8'b01010101;
38      seq2 = 8'b01010101;
39      #10;
40
41      $finish;
42
43      end
44
45      endmodule
```

Listing 12.1: Verilog Testbench for the Sequence Alignment Detector

12.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functional correctness of the implemented RISC-V Pipeline Hazard Unit. The testbench applies various combinations of source and destination register addresses while controlling the load instruction indicator to evaluate the hazard detection logic under different operating conditions. Multiple verification scenarios were considered, including independent

instructions, Read After Write (RAW) hazards, load-use hazards, matching and non-matching register addresses, disabled load operations, and accesses involving register x0. The applied test vectors were carefully selected to evaluate the ability of the hazard detection unit to correctly identify load-use dependencies under different operating conditions. The generated stall signal was continuously moni-

tored throughout the simulation. The observed outputs were compared with the expected pipeline behavior derived from manual analysis of the instruction dependencies. This comprehensive verification confirmed that the hazard detection logic correctly detects load-use hazards while avoiding unnecessary pipeline stalls during normal instruction execution.

```
1       `timescale 1ns/1ps
2
```

```
3       module tb_sequence_alignment_detector;
4
5       reg [7:0] seq1;
6       reg [7:0] seq2;
7       wire match;
8
9       sequence_alignment_detector uut(
10      .seq1(seq1),
11      .seq2(seq2),
12      .match(match)
13      );
14
15      initial
16      begin
17
18      $display("Time\tSeq1\t\tSeq2\t\tMatch");
19      $monitor("%0t\t%b\t%b\t%b", $time, seq1, seq2, match);
20
21      seq1 = 8'b10101010;
22      seq2 = 8'b10101010;
23      #10;
24
25      seq1 = 8'b11110000;
26      seq2 = 8'b11110000;
27      #10;
28
29      seq1 = 8'b00001111;
30      seq2 = 8'b11110000;
31      #10;
32
33      seq1 = 8'b11001100;
34      seq2 = 8'b11001101;
35      #10;
36
37      seq1 = 8'b01010101;
38      seq2 = 8'b01010101;
39      #10;
40
41      $finish;
42
43      end
44
45      endmodule
```

Listing 12.2: Verilog Testbench for the Sequence Alignment Detector

12.4 Simulation Results

Functional simulation was performed using the Ngspice simulator integrated with the eSim platform to verify the operation of the implemented RISC-V Pipeline Hazard Unit. The simulation waveforms demonstrate successful detection of load-use hazards by continuously monitoring the destination register of the instruction in the Execute stage and comparing it with the source register addresses of the instruction in the Decode stage. The waveform analysis confirms that the hazard

detection logic accurately identifies load-use dependencies and asserts the stall signal whenever the required operand is not yet available. In the absence of register dependencies, the stall signal remains inactive, allowing uninterrupted pipeline execution. The simulation also verified that the hazard unit responds immediately to changes in the input signals because of its combinational architecture. Throughout

the verification process, the generated stall signal matched the expected behavior of a standard five-stage RISC-V pipeline for every applied test case. No false hazard detection or unnecessary stall generation was observed during simulation. The successful verification demonstrates that the implemented RISC-V Pipeline Hazard Unit provides reliable load-use hazard detection and can be integrated into larger RISC-V processor architectures to ensure correct pipeline execution with minimal hardware overhead.

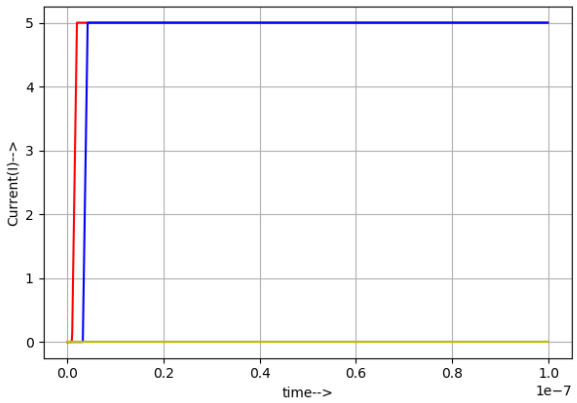


Figure 12.2: Simulation of RISC-V Pipeline Hazard Unit

Chapter 13

Instruction Cache Controller

13.1 Design Study

The Instruction Cache Controller was studied as an essential component of modern processor architectures that improves instruction fetch performance by reducing memory access latency. In pipelined processors, instruction fetching is performed continuously, and frequent accesses to external memory can significantly reduce system performance because of its relatively high access time. An instruction cache serves as a small, high-speed memory located between the processor and instruction memory, storing recently accessed instructions to minimize repeated memory accesses. The primary function of the Instruction Cache Controller is to determine

whether the requested instruction is already available in the cache. This is accomplished by comparing the tag portion of the requested instruction address with the stored cache tag while checking the corresponding valid bit. If both conditions are satisfied, a cache hit occurs and the instruction is immediately supplied to the processor. Otherwise, a cache miss is generated, and new instruction data supplied from the external memory interface is stored in the cache before being returned to the processor. Before implementation, the concepts of cache organization, address de-

composition, tag comparison, valid-bit management, cache hit and miss mechanisms, and cache update operations were thoroughly studied. Different cache organizations, including direct-mapped, set-associative, and fully associative caches, were also examined to understand their hardware complexity, performance characteristics, and implementation trade-offs. This study provided the theoretical foundation for implementing a simplified direct-mapped Instruction Cache Controller using Verilog HDL.

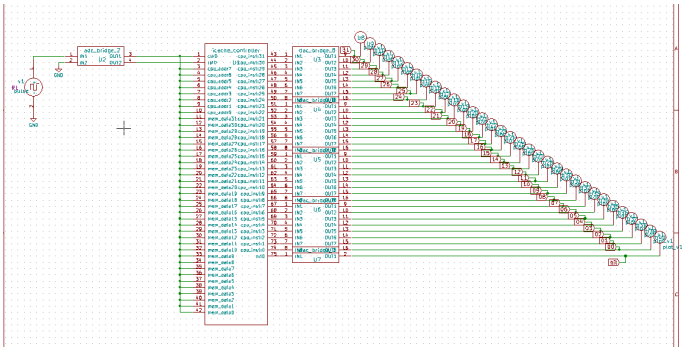


Figure 13.1: Implementation of the Instruction Cache Controller

13.2 Design Implementation

The Instruction Cache Controller was implemented using Verilog HDL as a direct-mapped cache consisting of cache data memory, tag memory, valid-bit storage, tag comparison logic, and cache control circuitry. The controller accepts an instruction address from the processor, extracts the corresponding tag and index fields, and accesses the appropriate cache line. During each cache access, the stored tag associated with the selected cache index is compared with the requested tag. If the tags match and the valid bit is asserted, a cache hit is generated and the cached instruction is immediately delivered to the processor. If either the tag comparison fails or the valid bit is not set, a cache miss occurs. In this case, the instruction data provided through the external memory data input is written into the selected cache line, the corresponding tag and valid bit are updated, and the fetched instruction is forwarded to the processor. The implementation employs synchronous sequential logic for cache updating and valid-bit management, while combinational comparison logic performs hit detection. This modular architecture improves readability, maintainability, and scalability. The completed design was successfully integrated into the eSim environment for functional simulation and verification.

```
1     module icache_controller (
2     input clk,
3     input rst,
4
5     input [7:0] cpu_addr,
6     output reg [31:0] cpu_instr,
7     output reg hit,
8
9     input [31:0] mem_data
10    );
11
12    reg [31:0] cache_data [0:15];
13    reg [3:0] cache_tag [0:15];
14    reg       valid [0:15];
15
```

```
    wire [3:0] index;
    wire [3:0] tag;

    assign index = cpu_addr[3:0];
    assign tag   = cpu_addr[7:4];

    integer i;

    always @(posedge clk or posedge rst)
    begin
    if(rst)
    begin
    for(i=0; i<16; i=i+1)
    begin
    cache_data[i] <= 0;
    cache_tag[i]  <= 0;
    valid[i]      <= 0;
    end

    cpu_instr <= 0;
    hit <= 0;
    end

    else
    begin

    // Cache Hit
    if(valid[index] &&
    cache_tag[index] == tag)
    begin
    cpu_instr <= cache_data[index];
    hit <= 1;
    end

    // Cache Miss
    else
    begin
    cache_data[index] <= mem_data;
    cache_tag[index]  <= tag;
    valid[index]      <= 1;

    cpu_instr <= mem_data;
    hit <= 0;
    end
    end
    end

    endmodule
```

Listing 13.1: Verilog RTL Implementation of the Instruction Cache Controller

13.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify the functional correctness of the Instruction Cache Controller. The testbench generates clock and reset signals while applying various instruction addresses and corresponding memory data values to evaluate cache operation under different scenarios. Several verification cases

were considered, including cache initialization, cache misses, cache hits, repeated instruction accesses, cache updating, and valid-bit operation. Different instruction addresses were applied to verify correct extraction of the cache index and tag fields, proper tag comparison, and accurate cache line selection. During cache miss conditions, external memory data values were supplied to emulate instruction fetches, allowing verification of cache update functionality. The testbench further verified

that once an instruction was written into the cache following a cache miss, subsequent accesses to the same address generated cache hits and returned the stored instruction correctly. The observed simulation results matched the expected behavior for all verification scenarios, confirming the correctness of the implemented design.

```
1     module tb_icache_controller;
2
3     reg clk;
4     reg rst;
5     reg [7:0] cpu_addr;
6     reg [31:0] mem_data;
7
8     wire [31:0] cpu_instr;
9     wire hit;
10
11    icache_controller DUT (
12    .clk(clk),
13    .rst(rst),
14    .cpu_addr(cpu_addr),
15    .cpu_instr(cpu_instr),
16    .hit(hit),
17    .mem_data(mem_data)
18    );
19
20    // Clock Generation
21    always #5 clk = ~clk;
22
23    initial
24    begin
25
26    clk = 0;
27    rst = 1;
28    cpu_addr = 0;
29    mem_data = 0;
```

```
30
31      #10;
32      rst = 0;
33
34      //------------------------------------------------
35      // Access Address 0x12 (Cache Miss)
36      //------------------------------------------------
37      cpu_addr = 8'h12;
38      mem_data = 32'hDEADBEEF;
39
40      #10;
41
42      $display("Addr=%h Instr=%h Hit=%b",
43      cpu_addr, cpu_instr, hit);
44
45      //------------------------------------------------
46      // Access Same Address Again (Cache Hit)
47      //------------------------------------------------
48      cpu_addr = 8'h12;
49
50      #10;
51
52      $display("Addr=%h Instr=%h Hit=%b",
53      cpu_addr, cpu_instr, hit);
54
55      //------------------------------------------------
56      // Access New Address (Cache Miss)
57      //------------------------------------------------
58      cpu_addr = 8'h34;
59      mem_data = 32'h12345678;
60
61      #10;
62
63      $display("Addr=%h Instr=%h Hit=%b",
64      cpu_addr, cpu_instr, hit);
65
66      //------------------------------------------------
67      // Access Same Address Again (Cache Hit)
68      //------------------------------------------------
69      cpu_addr = 8'h34;
70
71      #10;
72
73      $display("Addr=%h Instr=%h Hit=%b",
74      cpu_addr, cpu_instr, hit);
75
76      #20;
77      $finish;
78
```

```
79      end
80
81      endmodule
```

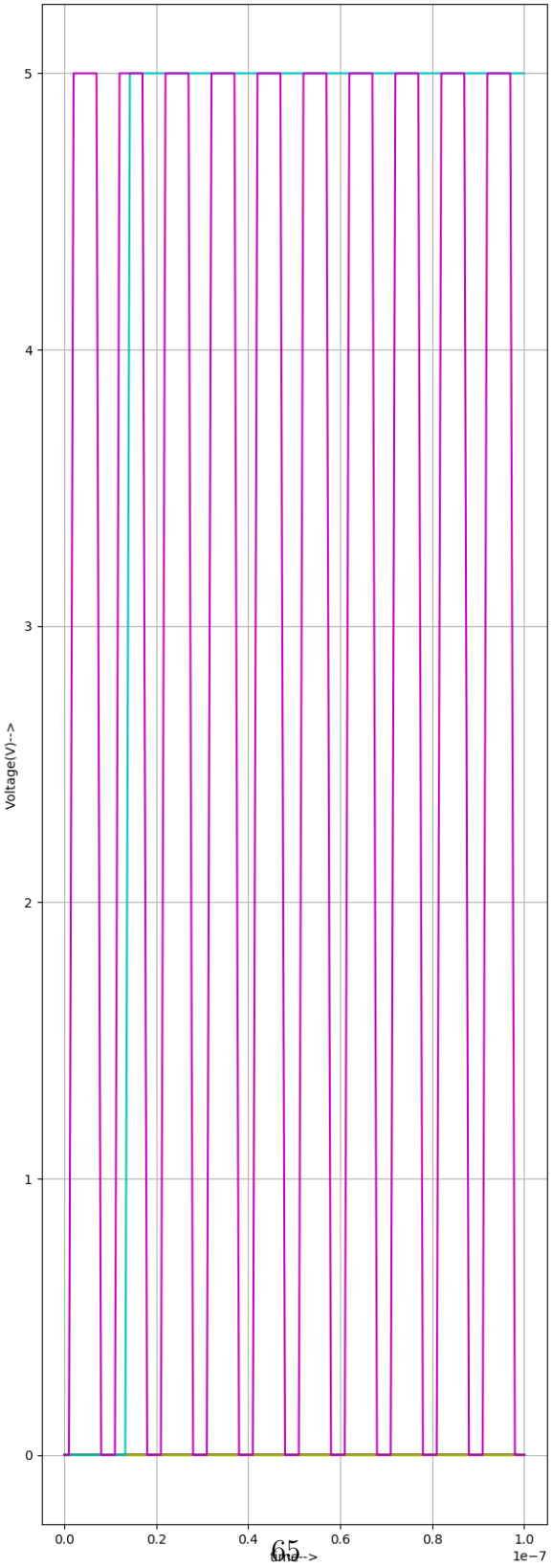
Listing 13.2: Verilog Testbench for the Instruction Cache Controller

13.4 Simulation Results

Functional simulation was performed using the Ngspice simulator integrated with the eSim platform to evaluate the behavior of the Instruction Cache Controller. The simulation results demonstrate correct cache operation by accurately distinguishing cache hits from cache misses through tag comparison and valid-bit verification. Dur-

ing cache hit conditions, the requested instruction was successfully retrieved from the cache without requiring cache updating. During cache miss conditions, the externally supplied instruction data was correctly written into the cache along with its associated tag and valid bit. Subsequent accesses to the same instruction address produced cache hits, confirming successful cache updating and storage. The simula-

tion waveforms verified proper operation of the cache indexing logic, tag comparison circuitry, valid-bit management, cache update mechanism, and instruction output generation. All observed hit and miss conditions matched the expected cache behavior, and no functional errors or synchronization issues were observed throughout the simulation. The successful verification demonstrates that the implemented Instruction Cache Controller operates correctly and provides an efficient mechanism for reducing instruction fetch latency in digital processor architectures.


Voltage(V)-->
5
4
3
2
1
0
0.0
0.2
0.4
0.6
0.8
1.0
1e−7

Chapter 14

Digital Twin Lockstep

14.1 Design Study

The Digital Twin Lockstep architecture was studied as a fault-tolerant hardware technique widely employed in safety-critical and mission-critical embedded systems such as aerospace electronics, automotive controllers, industrial automation, railway signaling systems, and medical devices. The primary objective of the lockstep mechanism is to improve system reliability by executing identical operations simultaneously on two independent processing units and continuously comparing their outputs. Any mismatch between the outputs indicates the occurrence of a hardware fault, transient error, or data corruption, allowing the system to detect failures before they propagate further. The implemented Digital Twin Lockstep system consists of two identical processing modules operating synchronously under a common clock and reset signal. Both modules receive the same input data and normally execute identical operations, producing matching outputs. One processing module functions as the reference Digital Under Test (DUT), while the second module acts as the digital twin. A dedicated comparator continuously compares the outputs generated by both modules at every clock cycle. To facilitate functional verification, the digital twin includes a fault injection mechanism controlled through a `fault_enable` signal. During normal operation, both modules compute identical results, whereas enabling the fault injection modifies the output of the twin by producing an incorrect value. The comparator detects this difference and asserts a mismatch signal, thereby validating the fault detection capability of the lockstep architecture. Before implementation, concepts including lockstep execution, hardware redundancy, synchronous execution, comparator design, fault injection techniques, clock synchronization, reset handling, and hardware fault detection were thoroughly studied. These studies provided the theoretical foundation for implementing the complete Digital Twin Lockstep architecture using Verilog HDL.

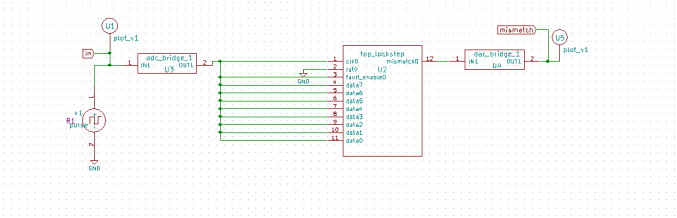


Figure 14.1: Implementation of Digital Twin Lockstep

14.2 Design Implementation

The Digital Twin Lockstep system was implemented using Verilog HDL as three independent modules: a processing block, a lockstep comparator, and a top-level integration module. Two identical processing blocks execute concurrently using the same clock, reset, and input data. The DUT always performs the intended operation by incrementing the input data by one, while the digital twin performs the same operation under normal conditions. A configurable fault injection mechanism

is incorporated into the twin processing module using the `fault_enable` control signal. When this signal is deasserted, both processing units generate identical outputs. When asserted, the twin intentionally produces an incorrect output by incrementing the input by two instead of one. This controlled deviation emulates a hardware fault without modifying the RTL design, allowing both normal and fault conditions to be evaluated using the same hardware implementation. The outputs

generated by both processing modules are continuously monitored by the lockstep comparator. During each clock cycle, the comparator compares the two outputs and generates a mismatch signal whenever they differ. The modular RTL organization simplifies verification, improves readability, and allows future enhancements such as Triple Modular Redundancy (TMR), majority voting, error recovery mechanisms, or more sophisticated fault injection models. After successful implementation, the complete design was integrated into the eSim environment for functional simulation and verification.

```
1     module top_lockstep(
2     input clk,
3     input rst,
4     input fault_enable,
5     input [7:0] data,
6     output mismatch
7     );
8
9     wire [7:0] dut_result;
10    wire [7:0] twin_result;
11
12    // DUT (Always Correct)
13    digital_twin_lockstep DUT(
14    .clk(clk),
```

```
15      .rst(rst),
16      .fault_enable(1'b0),
17      .in_data(data),
18      .out_data(dut_result)
19      );
20
21      // Twin (Fault Controllable)
22      digital_twin_lockstep TWIN(
23      .clk(clk),
24      .rst(rst),
25      .fault_enable(fault_enable),
26      .in_data(data),
27      .out_data(twin_result)
28      );
29
30      // Comparator
31      lockstep_comparator CMP(
32      .clk(clk),
33      .rst(rst),
34      .dut_out(dut_result),
35      .twin_out(twin_result),
36      .mismatch(mismatch)
37      );
38
39      endmodule
40
41
42      //------------------------------------------------
43      // Digital Twin Module
44      //------------------------------------------------
45      module digital_twin_lockstep(
46      input clk,
47      input rst,
48      input fault_enable,
49      input [7:0] in_data,
50      output reg [7:0] out_data
51      );
52
53      always @(posedge clk or posedge rst)
54      begin
55      if (rst)
56      out_data <= 8'd0;
57      else
58      begin
59      if (fault_enable)
60      out_data <= in_data + 8'd2;   // Faulty Behaviour
61      else
62      out_data <= in_data + 8'd1;   // Correct Behaviour
63      end
```

```
64      end
65
66      endmodule
67
68
69      //-------------------------------------------------
70      // Lockstep Comparator
71      //-------------------------------------------------
72      module lockstep_comparator(
73      input clk,
74      input rst,
75      input [7:0] dut_out,
76      input [7:0] twin_out,
77      output reg mismatch
78      );
79
80      always @(posedge clk or posedge rst)
81      begin
82      if (rst)
83      mismatch <= 1'b0;
84      else
85      mismatch <= (dut_out != twin_out);
86      end
87
88      endmodule
```

Listing 14.1: Verilog RTL Implementation of the Digital Twin Lockstep System

14.3 Test Bench Design

A comprehensive Verilog testbench was developed to verify both the functional correctness and fault detection capability of the implemented Digital Twin Lockstep architecture. The testbench generates synchronized clock and reset signals while applying identical input data to both processing modules through the top-level design. Multiple verification scenarios were implemented, including reset verification, normal

operation, boundary value testing, random input testing, and fault injection. During normal operation, the `fault_enable` signal remains deasserted, allowing both processing modules to execute identical operations and produce matching outputs. Consequently, the comparator continuously reports a mismatch value of zero. To

evaluate the fault detection mechanism, the `fault_enable` signal is asserted during selected test cases. This intentionally modifies the output generated by the twin processing module while the DUT continues normal execution. The comparator immediately detects the difference between the two outputs and asserts the mismatch signal. After disabling fault injection, both processing modules resume identical operation and the mismatch signal automatically returns to zero. The testbench

displays the applied input data, fault injection status, and comparator output after each test vector, allowing straightforward verification of the implemented lockstep architecture under both normal and faulty operating conditions.

```
`timescale 1ns/1ps

module tb_lockstep;

reg clk;
reg rst;
reg fault_enable;
reg [7:0] data;

wire mismatch;

// DUT
top_lockstep uut(
.clk(clk),
.rst(rst),
.fault_enable(fault_enable),
.data(data),
.mismatch(mismatch)
);

//------------------------------------------------
// Clock Generation
//------------------------------------------------
always #5 clk = ~clk;


//------------------------------------------------
// Task to Apply Input Data
//------------------------------------------------
task apply_data(input [7:0] value);
begin
@(posedge clk);
data = value;

@(posedge clk); // Wait for output to settle

$display("Time=%0t Data=%0d Fault=%b Mismatch=%b",
$time, value, fault_enable, mismatch);
end
endtask


//------------------------------------------------
// Test Sequence
//------------------------------------------------
initial
```

```
47      begin
48      clk = 0;
49      rst = 1;
50      data = 0;
51      fault_enable = 0;
52
53      $display("\n===== RESET =====");
54
55      repeat(2) @(posedge clk);
56      rst = 0;
57
58      @(posedge clk);
59      $display("Reset done, mismatch=%b", mismatch);
60
61      //-------------------------------------------------
62      // TC1 : Normal Operation
63      //-------------------------------------------------
64      $display("\n===== NORMAL OPERATION =====");
65      apply_data(10);
66      apply_data(20);
67      apply_data(30);
68
69      //-------------------------------------------------
70      // TC2 : Boundary Values
71      //-------------------------------------------------
72      $display("\n===== BOUNDARY TEST =====");
73      apply_data(0);
74      apply_data(255);
75
76      //-------------------------------------------------
77      // TC3 : Random Inputs
78      //-------------------------------------------------
79      $display("\n===== RANDOM TEST =====");
80      repeat(5)
81      apply_data($urandom_range(0,255));
82
83      //-------------------------------------------------
84      // TC4 : Fault Injection
85      //-------------------------------------------------
86      $display("\n===== FAULT INJECTION =====");
87      fault_enable = 1;
88
89      apply_data(15);
90      apply_data(50);
91      apply_data(90);
92
93      fault_enable = 0;
94
95      #20;
```

```
96      $finish;
97      end
98
99      endmodule
```

Listing 14.2: Verilog Testbench for the Digital Twin Lockstep System

14.4 Simulation Results

Functional simulation was performed using the Ngspice simulator integrated with the eSim platform to verify the Digital Twin Lockstep architecture. During normal operation, both processing modules received identical input data while the fault injection mechanism remained disabled. As expected, both modules generated identical outputs, and the comparator consistently maintained the mismatch signal at logic zero. The simulation was further extended by enabling the built-in

fault injection mechanism through the `fault_enable` signal. Under these conditions, the twin processing module intentionally generated an incorrect output, producing a mismatch with respect to the DUT. The comparator successfully detected this discrepancy during the same clock cycle and asserted the mismatch signal, confirming correct fault detection behavior. Simulation results were obtained for reset

operation, normal functional inputs, boundary values, randomly generated input vectors, and multiple fault injection scenarios. In all normal operating conditions, the comparator reported no mismatch, whereas every injected fault was successfully detected. After disabling fault injection, both processing modules resumed identical execution and the mismatch signal automatically returned to zero. These

results demonstrate that the implemented Digital Twin Lockstep architecture provides reliable real-time fault detection while maintaining correct operation under fault-free conditions. The modular implementation and configurable fault injection mechanism make the design suitable for educational verification, hardware reliability studies, and safety-oriented digital system development.

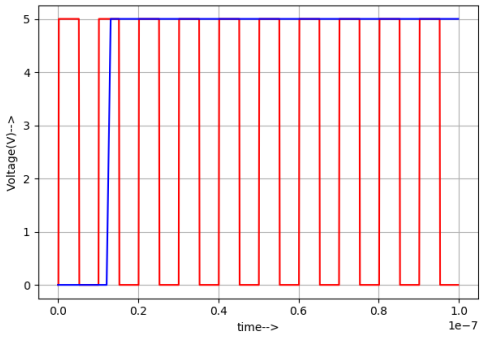


Figure 14.2: Simulation of Digital Twin Lockstep

Chapter 15

Conclusion

15.1 Summary of Contributions

During the Semester Long Internship under the FOSSEE Project at the Indian Institute of Technology Bombay, significant contributions were made towards the development, implementation, and verification of reusable digital Intellectual Property (IP) cores for the eSim platform. The work focused on designing hardware modules using Verilog HDL and validating their functionality through comprehensive testbenches and simulations.

The major contributions of this internship are summarized below:

- Designed and implemented a **2×2 Systolic Array Accelerator** for efficient parallel matrix computations using pipelined multiply-and-accumulate operations.
- Developed a **Digital Lock Detector** based on a finite state machine (FSM) capable of detecting predefined binary sequences with reliable state transitions.
- Implemented an **8-Tap FIR Filter Accelerator** employing multiply-and-accumulate architecture for digital signal processing applications.
- Designed a **Matrix Multiplication Accelerator** to perform high-speed matrix operations using parallel computation techniques.
- Developed a **Clock Failure Detector** capable of identifying missing or abnormal clock pulses for system reliability monitoring.
- Implemented a **Reed-Muller Encoder** using combinational logic to support error control coding applications.
- Designed a **Sequence Alignment Detector** to compare binary input sequences and determine matching conditions.
- Developed a **RISC-V Pipeline Hazard Unit** for detecting data hazards and controlling pipeline stalls during processor execution.
- Implemented an **Instruction Cache Controller** to improve processor performance through efficient cache hit and miss management.

- Designed a **Digital Twin Lockstep** module for fault detection by comparing outputs from redundant processing units operating simultaneously.
- Developed comprehensive Verilog testbenches for each IP module to validate functional correctness under different operating scenarios.
- Successfully integrated and verified all digital IPs using the eSim platform, ensuring correct functionality and reusability for future digital system development.
- Improved proficiency in Verilog HDL, digital system design, RTL development, hardware verification, debugging, and open-source EDA tools through practical implementation.

15.2 Learning Outcomes

The internship significantly enhanced both technical knowledge and practical implementation skills. The major learning outcomes include:

- Understanding the complete digital IP development flow using Verilog HDL.
- Designing combinational and sequential digital circuits using modular coding techniques.
- Developing reusable and parameterized RTL modules for digital systems.
- Creating comprehensive Verilog testbenches for functional verification.
- Performing simulation and debugging using the eSim platform.
- Improving knowledge of finite state machine (FSM) based digital system design.
- Understanding processor-related digital modules such as hazard detection units and instruction cache controllers.
- Learning good RTL coding practices, documentation standards, and hardware verification methodologies.
- Gaining practical experience in open-source EDA tools and collaborative software development.

15.3 Future Scope

The work carried out during this internship can be extended in several directions to improve functionality and applicability.

- Integration of all developed IP blocks into a complete System-on-Chip (SoC) architecture.

- FPGA implementation and hardware validation using Xilinx or Intel FPGA development boards.
- Addition of industry-standard interfaces such as AXI4, APB, or AHB to improve interoperability.
- Optimization of power, area, and timing using RTL optimization and synthesis techniques.
- Development of configurable and parameterized versions of the implemented IPs.
- Incorporation of Design-for-Testability (DFT), Built-In Self-Test (BIST), and fault-tolerant architectures.
- Verification using advanced methodologies such as Universal Verification Methodology (UVM) and SystemVerilog assertions.
- Development of additional reusable IP cores to expand the eSim digital IP library for educational and research purposes.

Bibliography

[1] Samir Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Edition, Pearson Education, 2003.

[2] M. Morris Mano and Michael D. Ciletti, *Digital Design*, 6th Edition, Pearson Education, 2018.

[3] Stephen Brown and Zvonko Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 3rd Edition, McGraw-Hill Education, 2014.

[4] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th Edition, Morgan Kaufmann, 2014.

[5] David A. Patterson and Andrew Waterman, *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon LLC, 2017.

[6] John G. Proakis and Dimitris G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 4th Edition, Pearson Education, 2007.

[7] Shu Lin and Daniel J. Costello, *Error Control Coding*, 2nd Edition, Pearson Education, 2004.

[8] AMD Xilinx, *Vivado Design Suite User Guide*, Available at: `https://docs.xilinx.com`

[9] FOSSEE Project, *eSim Documentation*, Indian Institute of Technology Bombay. Available at: `https://esim.fossee.in`

[10] FOSSEE Project, Indian Institute of Technology Bombay. Available at: `https://fossee.in`

[11] The ngspice Development Team, *Ngspice User Manual*. Available at: `https://ngspice.sourceforge.io`

[12] KiCad Development Team, *KiCad EDA Documentation*. Available at: `https://www.kicad.org`

[13] IEEE Xplore Digital Library. Available at: `https://ieeexplore.ieee.org`

[14] Accellera Systems Initiative, *Verilog and SystemVerilog Standards*. Available at: `https://www.accellera.org`

[15] Michael John Sebastian Smith, *Application-Specific Integrated Circuits*, Addison-Wesley, 1997.