Semester Long Internship Report

On

Designing of IP Blocks

& Verification using Mixed Signal Simulation in eSim

Submitted by

Yamini Menda

B.Tech (Electronics and Communication)

Rajiv Gandhi University of Knowledge Technologies, Nuzvid

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator,
Department of Aerospace Engineering,
Indian Institute of Technology Bombay

June 4, 2026

Acknowledgment

I would like to express my deepest gratitude to **Prof. Prabhu Ramachandran** for providing me the opportunity to be a part of the **FOSSEE** internship program and for supporting open-source engineering tool development at IIT Bombay. His guidance and vision have encouraged students and researchers to actively contribute to the open-source ecosystem.

I would also like to acknowledge **Prof. Kannan M. Moudgalya** for his foundational role in establishing and nurturing the **FOSSEE** initiative. His contributions toward open-source education and the creation of the **FOSSEE** fellowship framework have directly shaped the 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.

I would also like to thank my fellow students and friends at **Rajiv Gandhi University of Knowledge Technologies (RGUKT)** for their encouragement, support, and constructive discussions throughout this internship. Their motivation and cooperation contributed to a positive learning environment and helped me successfully complete this work.

I would also like to acknowledge and thank **Mr. P. Shyam**, Assistant Professor and EITP Dean, **Rajiv Gandhi University of Knowledge Technologies (RGUKT)**, for his constant support, encouragement, and guidance throughout this internship. His valuable advice and motivation have been instrumental in helping me manage academic and project responsibilities effectively and in successfully completing this internship.

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 real-world circuit modelling 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

Chapter 1

Introduction

The rapid growth of modern electronic systems has significantly increased the complexity of digital and analog circuit design. Designing reliable hardware systems requires extensive verification and testing before physical implementation to minimize design errors, reduce development cost, and improve overall system performance. Therefore, **Electronic Design Automation (EDA)** tools play an essential role in enabling efficient circuit modeling, simulation, verification, and analysis.

Although several commercial EDA tools are available for industrial applications, their high licensing cost often limits accessibility for students, researchers, and academic institutions. To address this challenge, open-source EDA platforms have emerged as cost-effective and flexible alternatives for electronic system development.

One such platform is **eSim**, developed under the **FOSSEE (Free/Libre and Open Source Software for Education)** project at **IIT Bombay**. eSim is an open-source EDA tool designed for circuit design, simulation, mixed-signal verification, and PCB development. It integrates multiple open-source simulation and verification tools into a single environment, enabling users to design and analyze **analog, digital, and mixed-signal systems** efficiently.

During this internship, eSim was utilized for the **design, implementation, and verification of multiple digital Intellectual Property (IP) blocks** using **mixed-signal simulation techniques**. Various reusable and industry-relevant IP blocks were modeled using **Verilog HDL**, integrated as subcircuits, and verified through waveform analysis in both **eSim** and **Vivado**. The internship provided practical exposure to **digital system design, mixed-signal simulation workflows, IP integration, and hardware verification methodologies**.

1.1 Overview of eSim

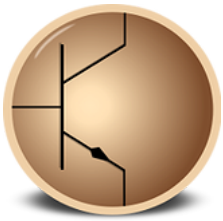


Figure 1.1: eSim Logo

eSim is an open-source **Electronic Design Automation (EDA)** software developed under the **FOSSEE** project for circuit design, simulation, and PCB development. It provides an integrated environment for creating, simulating, and verifying **analog, digital, and mixed-signal circuits**. By combining several open-source tools into a unified platform, eSim simplifies the overall electronic design process.

The platform supports the complete electronic design workflow, beginning with **schematic creation**, followed by **simulation and verification**, and extending to **PCB design implementation**. This integrated approach helps users analyze circuit functionality before hardware realization, thereby reducing implementation errors and improving reliability.

1.2 Tools Integrated in eSim

eSim integrates multiple open-source tools within a single environment to support the complete electronic system design workflow, including schematic capture, simulation, digital verification, device modeling, and PCB implementation. These integrated tools collectively enable efficient analysis and verification of analog, digital, and mixed-signal circuits.

1.2.1 Ngspice

Ngspice is the primary simulation engine integrated within eSim for performing **analog, digital, and mixed-signal circuit simulations**. It is an open-source SPICE-based simulator widely used for analyzing circuit behavior under different operating conditions.

Ngspice supports several simulation techniques such as **DC analysis**, **AC analysis**, **transient analysis**, and **operating point analysis**. These analyses enable designers to study voltage, current, timing characteristics, frequency response, and overall circuit performance before physical hardware implementation. By allowing early-stage verification, Ngspice helps reduce design errors and improves circuit reliability.

1.2.2 NgVeri

NgVeri is a digital simulation and verification tool integrated with eSim for analyzing hardware designs written in **Verilog and SystemVerilog**. It provides an efficient environment for functional verification of digital circuits through waveform generation and behavioral analysis.

NgVeri enables users to validate the correctness of digital logic, observe signal transitions, and identify functional errors during the design phase. It plays a significant role in verifying reusable **digital Intellectual Property (IP) blocks** and ensuring proper operation before hardware synthesis.

1.2.3 NGHDL

NGHDL is an integrated simulation framework used for verifying digital systems modeled using the **VHDL hardware description language**. It allows designers to simulate VHDL-based digital circuits and analyze their functionality through waveform inspection.

In addition to digital verification, NGHDL supports interaction between analog and digital domains, making it suitable for **mixed-signal system simulation**. This capability helps designers validate complex hardware systems involving both analog and digital components.

1.2.4 Makerchip-App

Makerchip-App is a browser-based digital design and simulation platform integrated with eSim to simplify hardware design workflows. It provides an interactive environment for designing, simulating, debugging, and visualizing digital circuits.

The platform supports rapid prototyping of hardware logic and enables users to observe real-time simulation results, making it particularly useful for learning, experimentation, and functional verification of digital systems.

1.2.5 Model Builder

The **Model Builder** in eSim enables users to create, customize, and modify simulation models for semiconductor devices such as **diodes, BJTs, MOSFETs, JFETs, and IGBTs**. Accurate device modeling is essential for obtaining realistic simulation behavior and improving circuit analysis precision.

This feature allows designers to define custom electrical parameters and adapt simulation models according to specific design requirements, thereby enhancing overall simulation flexibility and accuracy.

1.2.6 Subcircuit Builder

The **Subcircuit Builder** is a useful feature in eSim that allows users to create **reusable hierarchical circuit modules**. Frequently used circuit components or functional blocks can be converted into subcircuits and reused across multiple projects.

This modular design approach improves circuit organization, reduces design complexity, minimizes repetitive work, and supports scalable hardware development. During this internship, multiple digital **IP blocks** were developed and integrated using the subcircuit framework in eSim.

1.2.7 PCB Design Tools

For Printed Circuit Board (PCB) implementation, eSim integrates **CvPcb** and **Pcbnew**. **CvPcb** is used to assign appropriate component footprints to schematic symbols, ensuring compatibility between circuit design and physical hardware implementation.

Pcbnew is the PCB layout editor used for board design, component placement, routing, and optimization of electrical connections. It enables designers to generate manufacturable PCB layouts after successful circuit verification.

Thus, the integration of these tools makes eSim a comprehensive and flexible **open-source Electronic Design Automation (EDA) platform** suitable for academic learning, research activities, digital system design, mixed-signal verification, and hardware prototyping.

Chapter 2

Features of eSim

eSim provides an integrated environment for electronic circuit design, simulation, verification, and PCB implementation. It combines multiple open-source tools to support analog, digital, and mixed-signal system design. The major features of eSim are listed below:

1. **Schematic Design:** eSim provides a schematic editor called Eeschema, which enables users to create and modify electronic circuit schematics efficiently. It supports component placement, wiring, annotation, and electrical rule checking (ERC).

2. **Circuit Simulation:** eSim integrates Ngspice for circuit simulation, allowing users to verify circuit functionality before hardware implementation. It supports DC analysis, AC analysis, transient analysis, and mixed-signal simulation.

3. **PCB Layout Design:** eSim supports PCB implementation through Pcbnew, enabling users to convert schematics into PCB layouts with routing and track optimization.

4. **Component Footprint Association:** The CvPcb tool is used to assign footprints to schematic components, ensuring compatibility between circuit schematics and PCB layouts.

5. **KiCad to Ngspice Conversion:** eSim provides a KiCad-to-Ngspice converter for converting schematic netlists into a format compatible with Ngspice for simulation.

6. **Device Model Builder:** The Model Builder enables users to create and modify simulation models for devices such as diodes, BJTs, MOSFETs, JFETs, and IGBTs.

7. **Subcircuit Builder:** The Subcircuit Builder allows users to create reusable circuit modules and hierarchical circuit designs, reducing design complexity.

8. **Mixed Signal Simulation Support:** eSim supports digital and mixed-signal verification through NgVeri and NGHDL, enabling simulation using Verilog, SystemVerilog, and VHDL.

9. **Component Libraries:** eSim includes a wide range of component libraries consisting of analog, digital, hybrid, power, source, and user-defined components for flexible circuit development.

10. **Open Source and Cross Platform Support:** eSim is an open-source EDA tool accessible to students, educators, researchers, and hardware designers, supporting multiple operating systems.

Chapter 3

Design Methodology

3.1 Problem Identification

The problem identification phase focused on understanding the functionality, importance, and implementation requirements of various **digital Intellectual Property (IP) blocks** used in modern digital systems. Since IP blocks are fundamental building units in **embedded systems, FPGA, ASIC, and System-on-Chip (SoC)** architectures, proper analysis was essential before implementation.

The following methodology was adopted:

- Studied the role and industrial relevance of selected **digital IP blocks**.
- Identified the functional requirements, input-output behavior, and expected system responses of each design.
- Analyzed timing requirements, synchronization constraints, arbitration logic, signal conditioning, and control mechanisms.
- Examined the feasibility of implementation and verification using the **eSim mixed-signal simulation environment**.
- Defined design objectives for functional implementation and validation.

3.2 RTL Design and Verilog Implementation

After identifying the design requirements, the selected IP blocks were modeled using **Verilog Hardware Description Language (HDL)** at the **Register Transfer Level (RTL)**. The focus was on developing synthesizable and reusable hardware architectures.

The following design flow was followed:

- Studied the operational principle and architecture of each IP block.

- Designed RTL logic using **Verilog HDL** for implementing control, timing, synchronization, and sequential operations.
- Developed modular and reusable architectures suitable for hardware implementation.
- Used **Makerchip-App** integrated with eSim for understanding digital logic behavior and interactive verification wherever applicable.
- Ensured logical correctness and design consistency before schematic-level implementation.

3.3 Schematic Design and Subcircuit Development in eSim

The designed IP blocks were implemented at the schematic level using the **eSim platform**. To improve modularity and design reuse, the implemented circuits were converted into reusable subcircuits.

The following procedure was adopted:

- Created schematic representations using the **eSim schematic editor**.
- Used the **Subcircuit Builder** to develop reusable **IP modules**.
- Integrated **ADC** and **DAC** blocks for enabling mixed-signal interaction between analog and digital domains.
- Configured signal interconnections, clock inputs, reset logic, and control signals according to functional requirements.
- Verified schematic structure and module connectivity before simulation.

3.4 Mixed-Signal Simulation

The designed IP blocks were validated through **mixed-signal simulation** using the integrated toolchain available in eSim. Simulation was performed to observe functional correctness, timing behavior, and signal interaction under different operating conditions.

The following simulation methodology was followed:

- Used **NgVeri** for simulation and functional verification of digital circuits written in **Verilog HDL**.

- Used **Ngspice** to perform mixed-signal simulation involving interaction between digital circuits and analog interfaces.
- Used **NGHDL** wherever VHDL compatibility and digital verification support were required within the mixed-signal environment.
- Applied different clock conditions, control signals, and test cases to evaluate circuit behavior.
- Observed waveform responses, signal transitions, synchronization behavior, and timing characteristics for functional analysis.

3.5 Verification and Analysis

The final verification stage ensured that the implemented IP blocks performed according to the expected theoretical behavior. The outputs obtained through simulation were analyzed to confirm functional correctness and reliability.

The following analysis procedure was followed:

- Compared simulation outputs with expected design functionality.
- Analyzed waveform responses generated using **NgVeri** and **Ngspice**.
- Verified timing relationships, synchronization mechanisms, control operations, and signal integrity.
- Identified and corrected possible functional mismatches through iterative testing.
- Validated the overall performance of the designed IP blocks for reliable mixed-signal system integration.

Chapter 4

IP Blocks

4.1 Clock Gating Controller

4.1.1 Introduction

Clock gating is a widely used low-power design technique in modern digital systems to reduce unnecessary dynamic power consumption. Since clock signals continuously toggle regardless of circuit activity, inactive modules consume power even when not performing useful operations. A **Clock Gating Controller** selectively enables or disables clock propagation to functional blocks based on control signals, thereby reducing switching activity and improving power efficiency.

In this work, an **edge-triggered Clock Gating Controller** was designed and implemented using Verilog HDL. The controller supports multiple gated clock outputs and incorporates a dedicated **Test Mode** to facilitate functional verification and timing analysis.

4.1.2 Importance of Clock Gating Controller

- To reduce dynamic power consumption caused by continuous clock switching.
- To prevent unnecessary toggling in idle functional blocks.
- To improve overall power efficiency in digital systems.
- Widely used in low-power processors, SoCs, and ASIC designs.

4.1.3 Schematic Implementation

The Clock Gating Controller consists of a main clock input, enable control signals, reset logic, and a test mode control signal. The enable signals determine whether a gated clock output remains active or disabled during normal operation.

The schematic implementation of the controller is shown in Figure 4.1. The design includes edge-triggered enable registration to ensure stable clock gating and avoid glitches during switching.

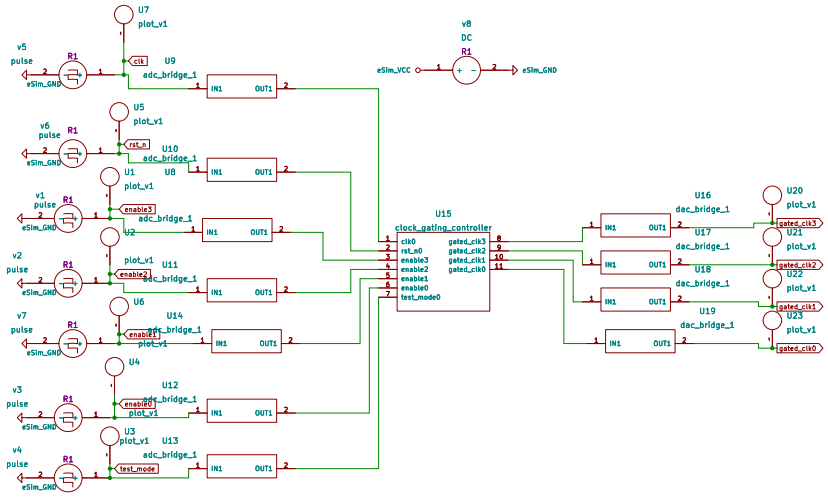


Figure 4.1: Schematic of the Clock Gating Controller

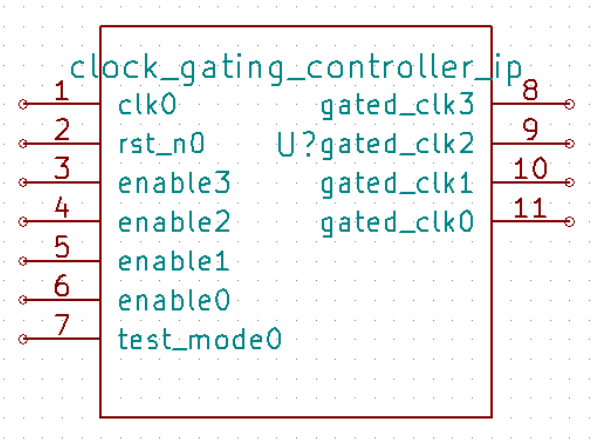


Figure 4.2: IP module of Clock Gating Controller

4.1.4 RTL Design and Verilog Implementation

The Clock Gating Controller was implemented using synthesizable Verilog HDL. The design follows an edge-triggered approach where enable signals are registered at the positive edge of the clock, and a generate block is used to implement scalable clock gating for multiple outputs.

The Verilog implementation of the design is provided in the following section. The architecture is modular and suitable for FPGA and ASIC-based digital systems.

The RTL design generated from the implementation is shown below.

```
1  `timescale 1ns / 1ps
2
3  module Clock_gating_controller (
4      input  wire                  clk,
5      input  wire                  rst_n,
6      input  wire [3:0]   enable,
7      input  wire                  test_mode,
8
9      output wire [3:0]   gated_clk
10 );
11
12     reg [3:0] enable_reg;
13
14     always @(posedge clk or negedge rst_n) begin
15         if (!rst_n)
16             enable_reg <= 0;
17         else
18             enable_reg <= enable;
19     end
20
21     // Clock gating
22     genvar j;
23     generate
24         for (j = 0; j <4; j = j + 1) begin : GATING
25             assign gated_clk[j] = test_mode ? clk : (clk &
                   enable_reg[j]);
26         end
27     endgenerate
28
29 endmodule
```

Listing 4.1: Clock Gating Controller Verilog Code

4.1.5 Functional Description

The Clock Gating Controller operates in two modes: **Normal Mode** and **Test Mode**.

- **Normal Mode:** Gated clocks follow their enable signals to reduce switching activity and save power.
- **Test Mode:** All gated clocks follow the main clock, enabling easier verification and debugging.

To ensure glitch-free operation, enable signals are sampled using edge-triggered registers before clock gating is performed.

4.1.6 Simulation Results and Verification

The Clock Gating Controller was verified using **eSim** and **Vivado** simulations. Test cases were used to evaluate Normal and Test modes.

In Normal Mode, gated clocks followed the enable signals, while in Test Mode all clocks followed the main clock. The results confirmed correct functionality of the design.

Simulation Results for Clock Gating Controller (Key Cases)

Clock Edge	Enable	Test Mode	Gated Clock[3:0]	Observation
Positive Edge	0000	0	0000	All clocks disabled
Positive Edge	1000	0	1000	Only MSB toggles
Positive Edge	1111	0	1111	All clocks enabled per enable bits
Positive Edge	xxxx	1	1111	Test mode active, all clocks follow clk
Negative Edge	any	0 or 1	Hold previous value	No update occurs

Figure 4.3: Simulation Results for Key cases

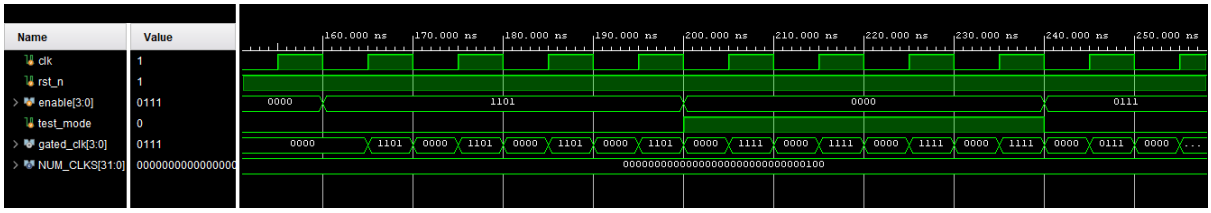


Figure 4.4: Simulation Results in Vivado

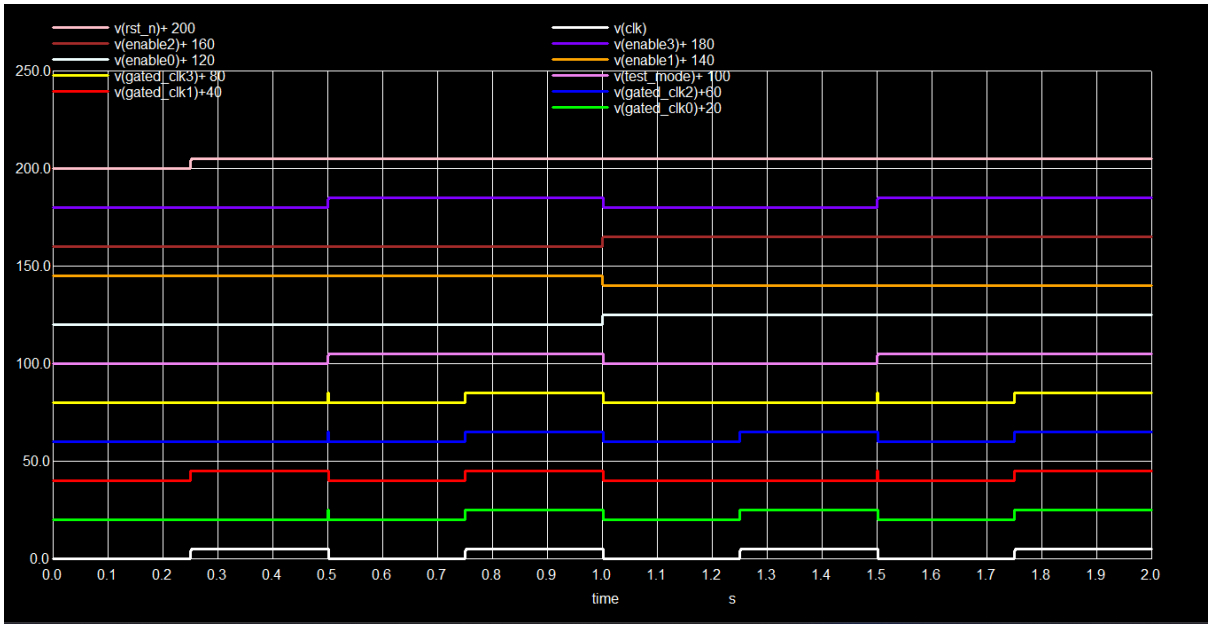


Figure 4.5: Simulation Results in eSim

4.1.7 Applications

Clock gating techniques are widely used in several low-power digital systems and integrated circuits. Some major applications include:

- Low-power processors and microcontrollers
- System-on-Chip (SoC) architectures
- Battery-powered embedded systems
- ASIC and FPGA-based power optimization
- Design-for-Test (DFT) and timing verification environments

4.1.8 Summary

The Clock Gating Controller was successfully designed and implemented to reduce dynamic power consumption by selectively controlling clock propagation. The controller supports both Normal and Test operating modes, ensuring power efficiency as well as ease of verification. Simulation results validated the expected functionality of the design, demonstrating its suitability for low-power FPGA and ASIC-based digital systems.

4.2 Interrupt Controller

4.2.1 Introduction

An Interrupt Controller is a digital circuit used to manage multiple interrupt requests and prioritize them for a processor. It enables efficient handling of asynchronous events by selecting the highest-priority interrupt when multiple requests occur simultaneously.

In this work, a 4-input Interrupt Controller was designed and implemented using Verilog HDL.

4.2.2 Importance of Interrupt Controller

- Enables priority-based interrupt handling
- Reduces processor overhead compared to polling
- Improves system response time
- Supports multiple interrupt sources efficiently

4.2.3 Schematic Implementation

The Interrupt Controller consists of four interrupt request inputs, a priority encoder, and control logic to generate interrupt and encoded ID outputs.

The schematic representation is shown in Figure 4.9.

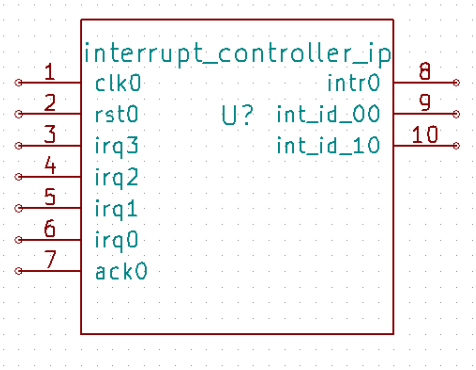


Figure 4.6: IP Module

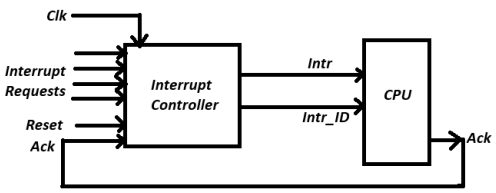


Figure 4.7: Block Diagram

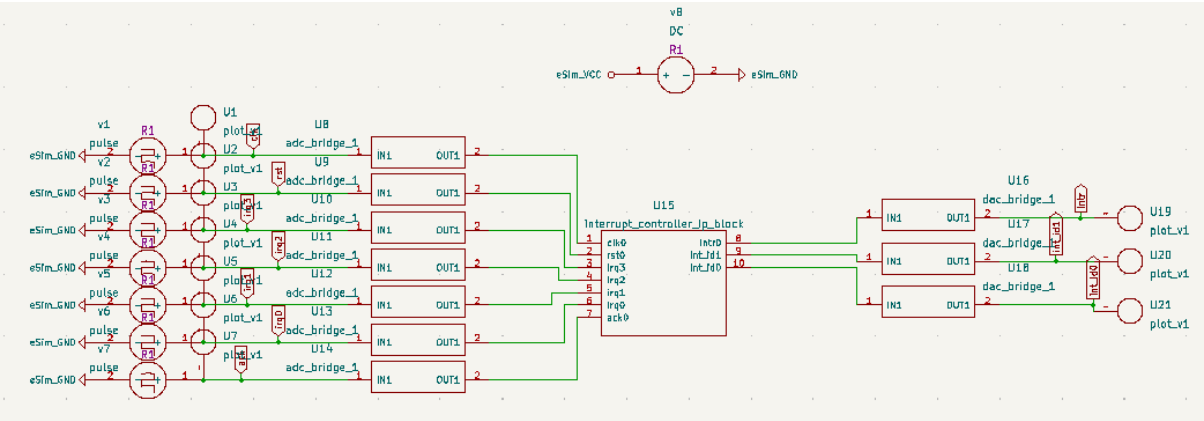


Figure 4.8: Schematic of Interrupt Controller

4.2.4 Functional Description

The Interrupt Controller operates by continuously monitoring interrupt request inputs.

- **Normal Operation:** The controller selects the highest-priority active interrupt and generates the corresponding interrupt signal and ID.
- **Acknowledgement:** The interrupt is cleared when the processor sends an acknowledge signal.

Interrupt Requests	Highest Priority	int_id
0001	IRQ0	00
001x	IRQ1	01
01xx	IRQ2	10
1xxx	IRQ3	11
0000	No Interrupt	--

Figure 4.9: Priority Encoding

4.2.5 RTL Design and Verilog Implementation

The Interrupt Controller was implemented using synthesizable Verilog HDL. The design uses a priority encoder and FSM-based control logic to generate interrupt signals and encoded interrupt IDs.

```
module Interrupt_Controller_ip_block(
    input  wire        clk,
    input  wire        rst,
    input  wire [3:0]  irq,
    input  wire        ack,
    output reg         intr,
    output reg [1:0]   int_id
);
    parameter IDLE  = 2'b00;
    parameter SERVE = 2'b01;
    reg [1:0] state, next_state;
    reg [1:0] priority_id;
    reg valid_irq;

    always @(*) begin
        valid_irq = 1'b1;
        casez (irq)
            4'b1???: priority_id = 2'd3;
            4'b01??: priority_id = 2'd2;
            4'b001?: priority_id = 2'd1;
            4'b0001: priority_id = 2'd0;
            default: begin
                priority_id = 2'd0;
                valid_irq = 1'b0;
            end
        endcase
    end
    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= IDLE;
        else
            state <= next_state;
    end
    always @(*) begin
        case (state)
            IDLE:  next_state = (valid_irq) ? SERVE : IDLE;
            SERVE: next_state = (ack) ? IDLE : SERVE;
            default: next_state = IDLE;
        endcase
    end
    always @(posedge clk or posedge rst) begin
```

```
        if (rst) begin
            intr <= 0;
            int_id <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (valid_irq) begin
                        intr <= 1;
                        int_id <= priority_id;
                    end else
                        intr <= 0;
                end
                SERVE: begin
                    intr <= 1;
                    if (ack)
                        intr <= 0;
                end
            endcase
        end
    end
endmodule
```

Listing 4.2: Interrupt Controller Verilog Code

4.2.6 Simulation Results and Verification

The design was verified using **eSim** and **Vivado**. The controller correctly handled single and multiple interrupt requests and selected the highest-priority interrupt in all cases.

In multiple request conditions, the priority order was:

$$IRQ_3 > IRQ_2 > IRQ_1 > IRQ_0$$

The simulation results confirmed correct interrupt generation, priority selection, and acknowledgement behavior.

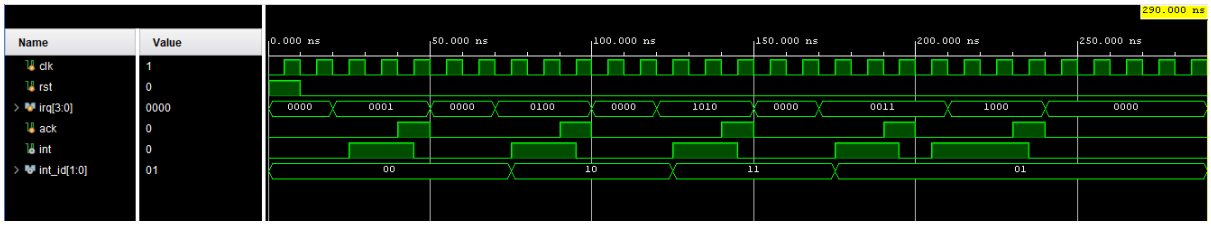


Figure 4.10: Simulation Results in Vivado

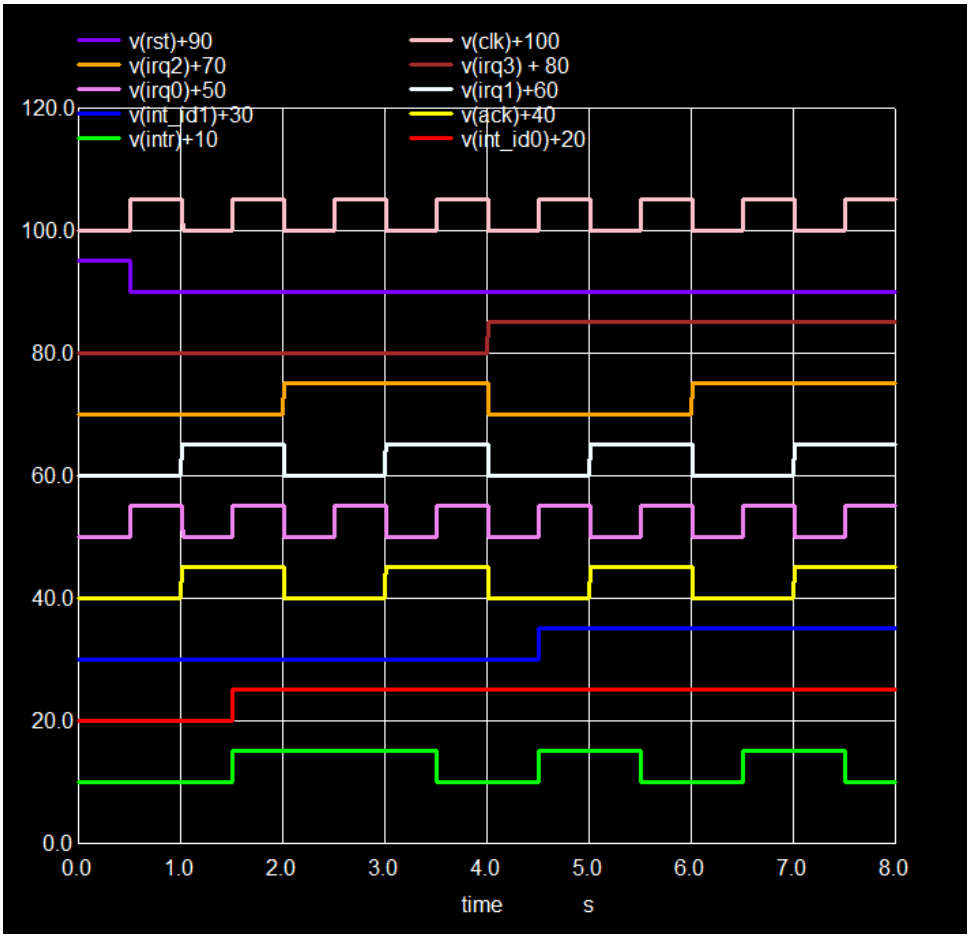


Figure 4.11: Simulation Results in eSim

4.2.7 Applications

- Microprocessor systems
- Embedded controllers
- FPGA-based systems
- Communication systems

4.2.8 Summary

The Interrupt Controller was successfully designed and verified using Verilog HDL. It efficiently handles multiple interrupt requests using priority-based selection and provides correct interrupt signaling and encoding. The design is suitable for basic digital and embedded system applications.

4.3 Universal Shift Register

4.3.1 Introduction

A Universal Shift Register is a sequential circuit capable of performing multiple data operations such as hold, shift left, shift right, and parallel load. It is widely used in digital systems for data storage and data movement applications.

In this work, an 8-bit Universal Shift Register was designed and implemented using Verilog HDL with mode-controlled operation.

4.3.2 Importance of Universal Shift Register

- Supports multiple data operations in a single circuit
- Useful for serial and parallel data conversion
- Reduces hardware complexity by combining functionalities
- Widely used in communication and processor-based systems

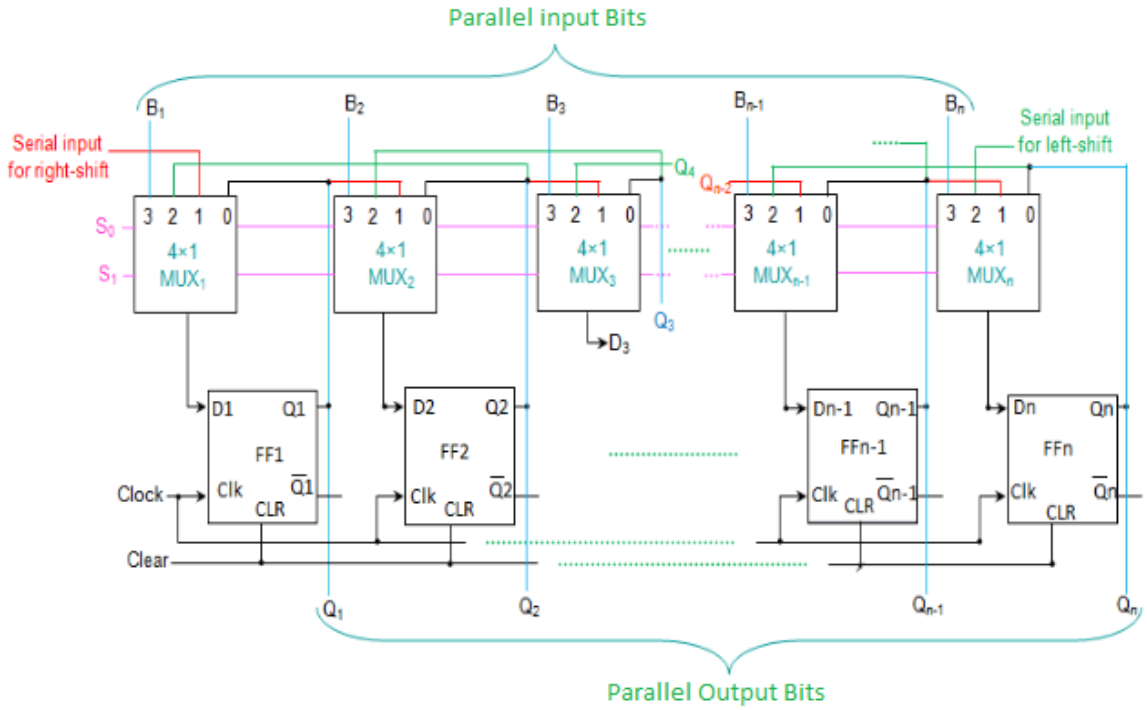


Figure 4.12: Block Diagram of Universal Shift Register

4.3.3 Schematic Implementation

The Universal Shift Register consists of an 8-bit register with mode control logic, allowing selection between hold, shift left, shift right, and parallel load operations.

The schematic implementation is shown in Figure 4.13.

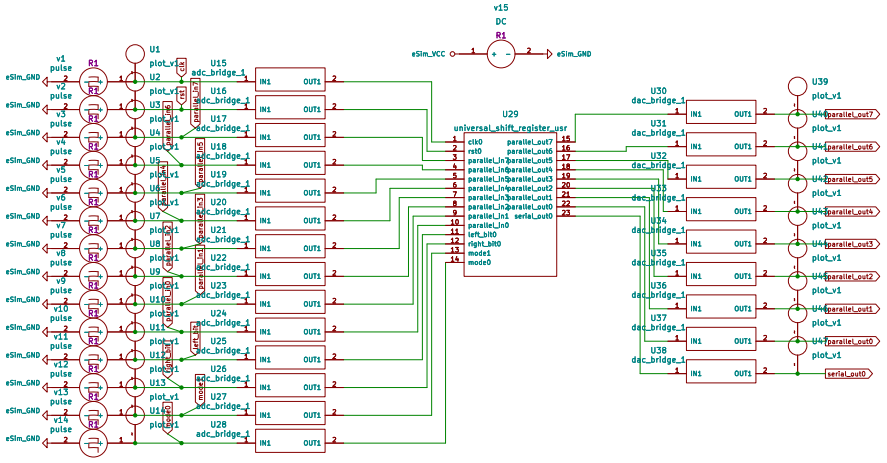


Figure 4.13: Schematic of Universal Shift Register

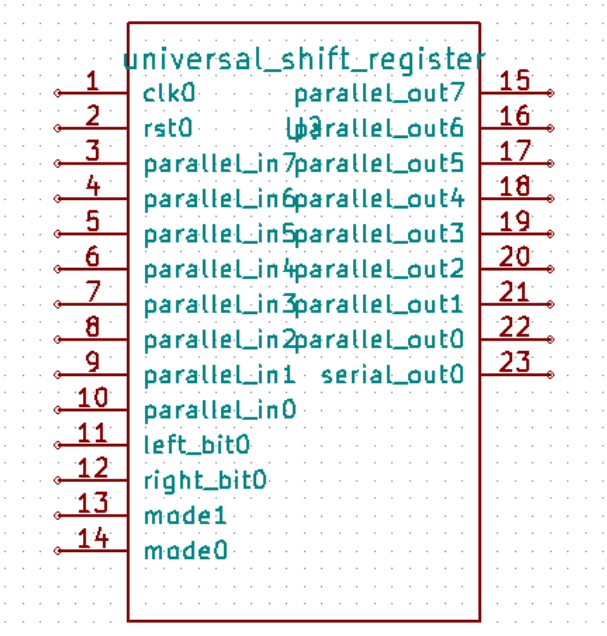


Figure 4.14: IP Module of USR

4.3.4 RTL Design and Verilog Implementation

The Universal Shift Register was implemented using synthesizable Verilog HDL. The design uses a case-based control structure to perform different shift and load operations based on a 2-bit mode input.

```
module Universal_Shift_Register_usr(
    input clk, rst,
    input [7:0] parallel_in,
    input left_bit, right_bit,
    input [1:0] mode,
    output reg [7:0] parallel_out,
    output reg serial_out
);
    localparam HOLD = 2'b00,
               SHIFT_LEFT = 2'b01,
               SHIFT_RIGHT = 2'b10,
               PARALLEL_LOAD = 2'b11;
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            parallel_out <= 8'b0;
            serial_out <= 1'b0;
        end else begin
            case (mode)
                HOLD: parallel_out <= parallel_out;
                SHIFT_LEFT: begin
                    serial_out <= parallel_out[7];
                    parallel_out <= {parallel_out[6:0],
                       left_bit};
                end
                SHIFT_RIGHT: begin
                    serial_out <= parallel_out[0];
                    parallel_out <= {right_bit, parallel_out
                       [7:1]};
                end
                PARALLEL_LOAD: parallel_out <= parallel_in;
                default: parallel_out <= parallel_out;
            endcase
        end
    end
endmodule
```

Listing 4.3: Universal Shift Register Verilog Code

4.3.5 Functional Description

The Universal Shift Register operates based on a 2-bit mode control signal:

- **Hold Mode:** Retains previous data
- **Shift Left Mode:** Shifts data left with serial input
- **Shift Right Mode:** Shifts data right with serial input
- **Parallel Load Mode:** Loads 8-bit input data directly

Mode[1:0]	Operation	Description
00	Hold	Retains the current data without any change.
01	Shift Left	Shifts the stored data one bit to the left and inserts **left_bit** into the LSB.
10	Shift Right	Shifts the stored data one bit to the right and inserts **right_bit** into the MSB.
11	Parallel Load	Loads the 8-bit input data **parallel_in** directly into the register.

Figure 4.15: Operations of Universal Shift Register

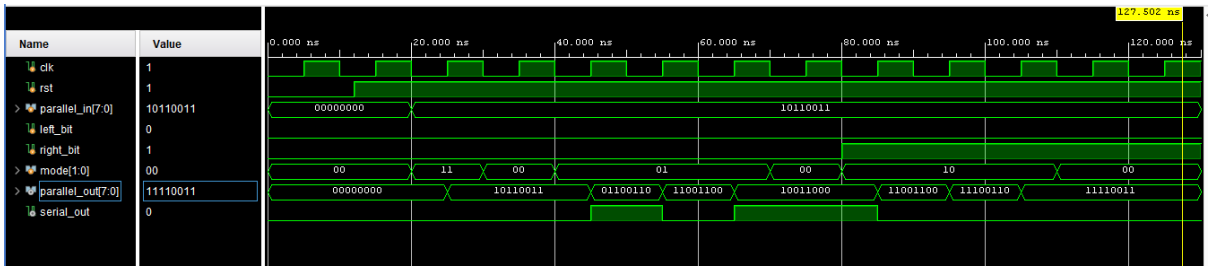
4.3.6 Simulation Results and Verification

The design was verified using simulation tools, and all modes were tested successfully. The register correctly performed data hold, shift operations, and parallel loading.

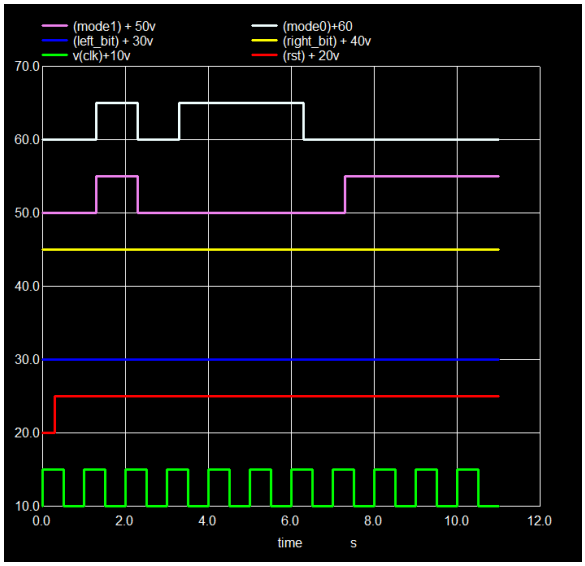
The results confirmed correct functionality for all test cases.

Step	Mode	Binary Output	Decimal Output
Initial Load	11	10110011	179
Hold	00	10110011	179
Left Shift 1	01	01100110	102
Left Shift 2	01	11001100	204
Left Shift 3	01	10011000	152
Hold	00	10011000	152
Right Shift 1	10	11001100	204
Right Shift 2	10	11100110	230
Right Shift 3	10	11110011	243

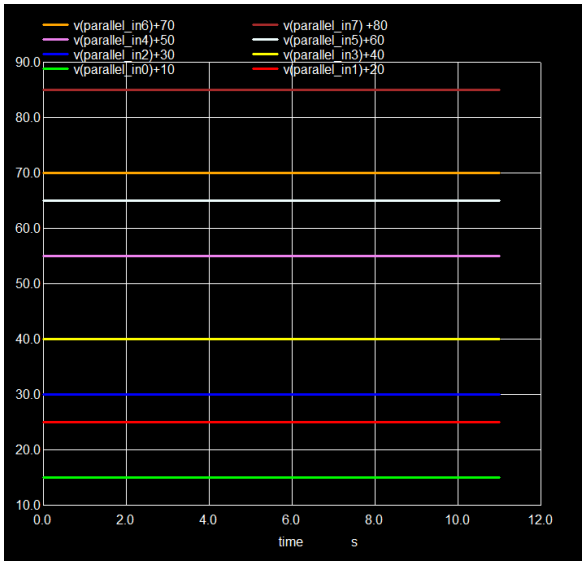
Figure 4.16: Observed Output Sequence of USR



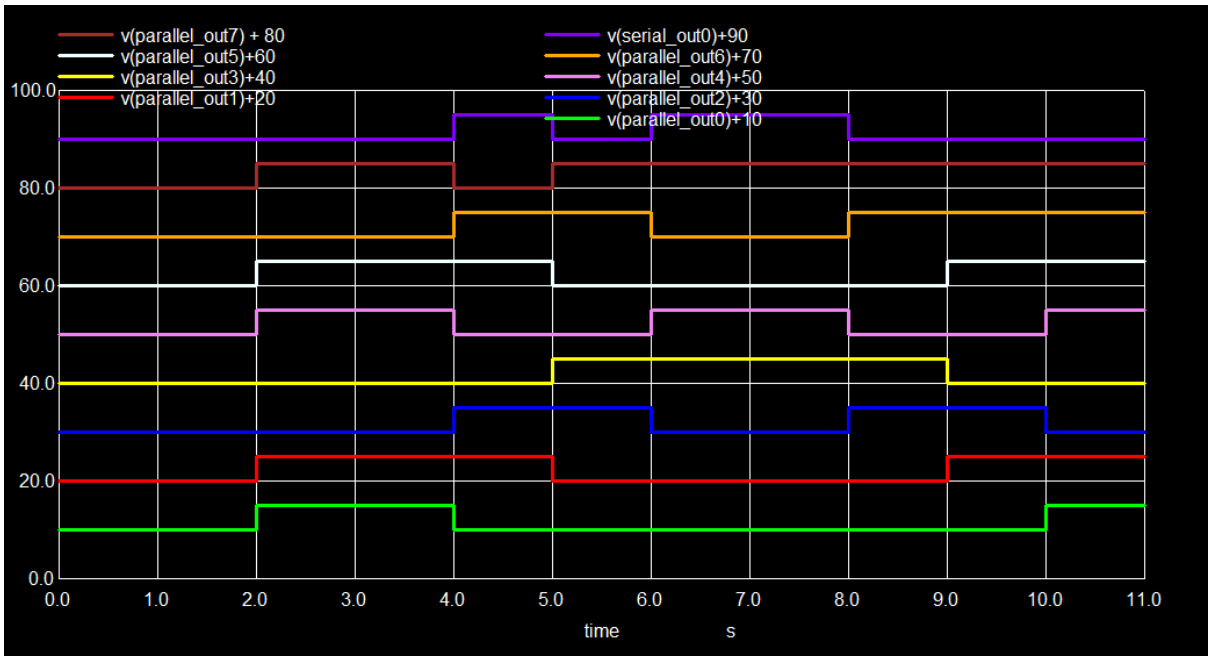
(a) Simulation Results in Vivado



(b) Input Simulation Setup in eSim



(c) Input Data



(d) Output Data

Figure 4.17: Simulation and I/O Results of the Universal Shift Register

4.3.7 Applications

- Serial-to-parallel data conversion
- Data buffering and temporary storage
- Digital communication systems
- Processor register and data path design

4.3.8 Summary

The Universal Shift Register was successfully designed and verified using Verilog HDL. The design supports multiple data operations within a single IP block, making it suitable for flexible data handling in digital systems.

4.4 Round Robin Arbiter

4.4.1 Introduction

A Round Robin Arbiter is a digital circuit used to allocate shared resources among multiple requesters in a fair and cyclic manner. Unlike fixed-priority arbitration, it ensures equal access by dynamically rotating priority, thereby preventing starvation.

In this work, a 4-input Round Robin Arbiter was designed and implemented using Verilog HDL.

4.4.2 Importance of Round Robin Arbiter

- Ensures fair resource allocation among multiple requesters
- Prevents starvation through cyclic priority rotation
- Improves shared resource utilization
- Widely used in bus and memory arbitration systems

4.4.3 Schematic Implementation

The Round Robin Arbiter consists of request inputs, grant outputs, and a rotating pointer mechanism to dynamically update priority after every successful grant.

The schematic implementation is shown below.

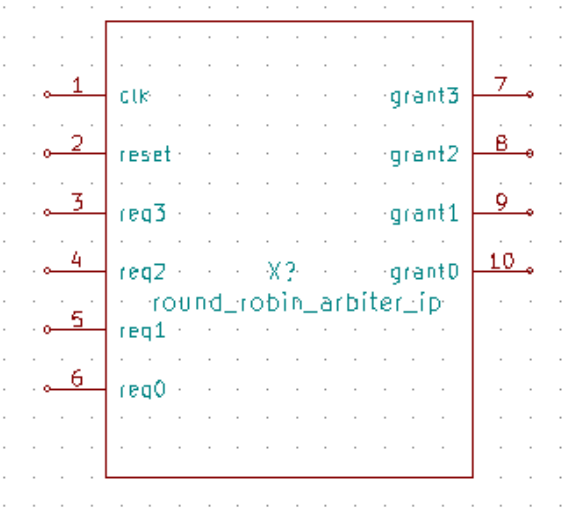


Figure 4.18: IP Module of Round Robin Arbiter

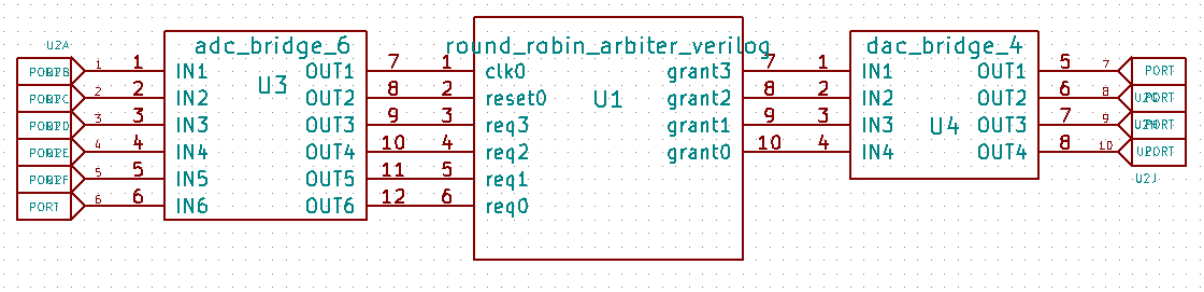


Figure 4.19: Subcircuit integration with ADC and DAC blocks

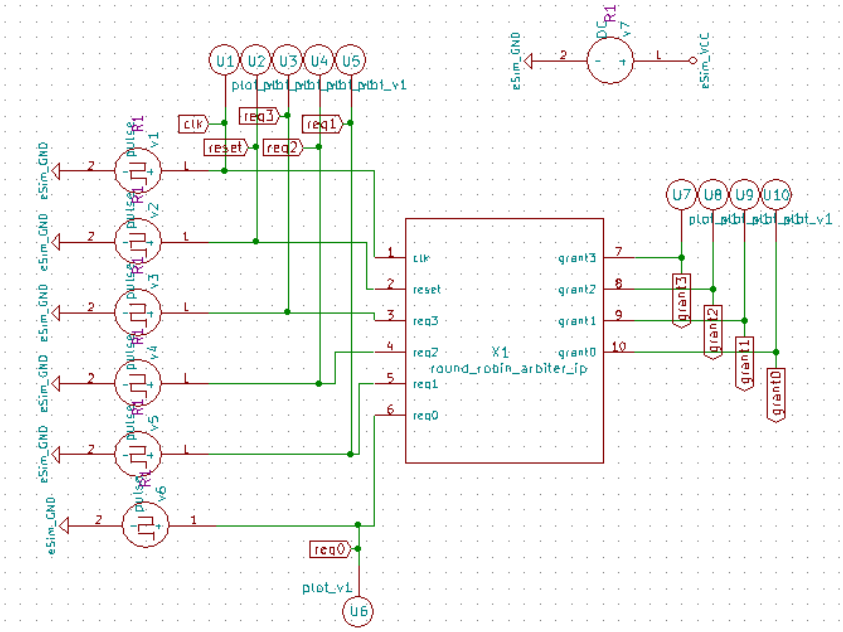


Figure 4.20: Schematic of Round robin Arbiter

4.4.4 Functional Description

The Round Robin Arbiter operates using a rotating priority mechanism.

- Active requests are checked cyclically based on pointer position.
- The first valid request receives the grant signal.
- The pointer updates after every successful grant to ensure fairness.
- If no request is active, no grant signal is generated.

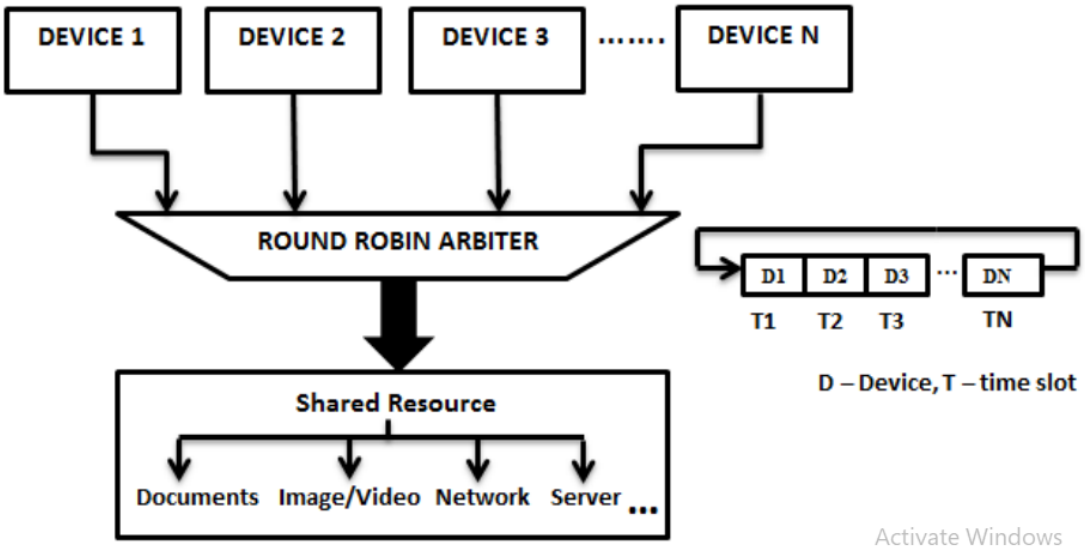


Figure 4.22: Block Diagram of Round Robin Arbiter

Cycle	Pointer	Request (req[3:0])	Grant (grant[3:0])	Explanation
1	0	1111	0001	Starts at 0 → grants req[0]
2	1	1111	0010	Next priority → grants req[1]
3	2	1111	0100	Next → grants req[2]
4	3	1111	1000	Next → grants req[3]
5	0	1010	0010	req[0]=0 → skips → grants req[1]
6	2	1010	1000	req[2]=0 → skips → grants req[3]
7	0	0000	0000	No request → no grant
8	0	0101	0001	Starts at 0 → grants req[0]

Figure 4.21: Working Sequence of Round Robin Arbiter

4.4.5 RTL Design and Verilog Implementation

The Round Robin Arbiter was implemented using synthesizable Verilog HDL. The design uses a pointer-based approach to rotate priority dynamically and generate grant signals based on active requests.

Listing 4.4: Verilog Code for Round Robin Arbiter

```
module round_robin_arbiter(input clk,input reset,input [3:0] req,output
    reg [3:0] grant);
reg [1:0] pointer;
always @(posedge clk or posedge reset) begin
 if (reset) begin pointer<=2'b00; grant<=4'b0000; end
 else begin
  case(pointer)
   2'd0: begin
    if(req[0]) begin grant<=4'b0001; pointer<=2'd1; end
    else if(req[1]) begin grant<=4'b0010; pointer<=2'd2; end
    else if(req[2]) begin grant<=4'b0100; pointer<=2'd3; end
    else if(req[3]) begin grant<=4'b1000; pointer<=2'd0; end
    else grant<=4'b0000;
   end
   2'd1: begin
    if(req[1]) begin grant<=4'b0010; pointer<=2'd2; end
    else if(req[2]) begin grant<=4'b0100; pointer<=2'd3; end
    else if(req[3]) begin grant<=4'b1000; pointer<=2'd0; end
    else if(req[0]) begin grant<=4'b0001; pointer<=2'd1; end
    else grant<=4'b0000;
   end
   2'd2: begin
    if(req[2]) begin grant<=4'b0100; pointer<=2'd3; end
    else if(req[3]) begin grant<=4'b1000; pointer<=2'd0; end
    else if(req[0]) begin grant<=4'b0001; pointer<=2'd1; end
    else if(req[1]) begin grant<=4'b0010; pointer<=2'd2; end
    else grant<=4'b0000;
   end
   2'd3: begin
    if(req[3]) begin grant<=4'b1000; pointer<=2'd0; end
    else if(req[0]) begin grant<=4'b0001; pointer<=2'd1; end
    else if(req[1]) begin grant<=4'b0010; pointer<=2'd2; end
    else if(req[2]) begin grant<=4'b0100; pointer<=2'd3; end
    else grant<=4'b0000;
   end
  endcase
 end
end
endmodule
```

4.4.6 Simulation Results and Verification

The Round Robin Arbiter was verified using **Vivado** and **eSim**. Different request combinations were tested to validate grant generation and pointer rotation.

The simulation results confirmed correct cyclic priority allocation and fair resource sharing without starvation.

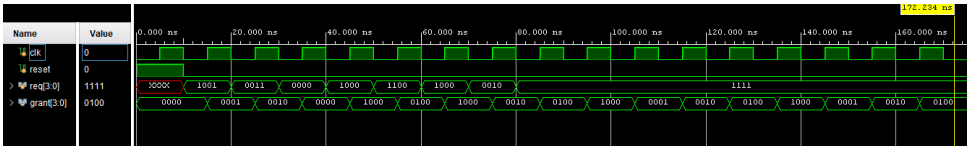


Figure 4.23: Simulation Results in Vivado

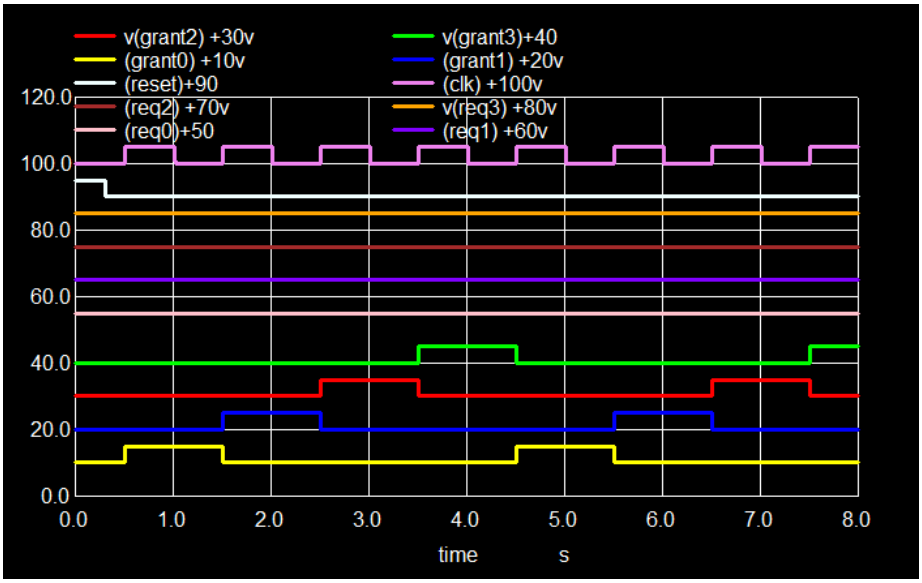


Figure 4.24: Simulation Results in eSim

4.4.7 Applications

- Bus arbitration systems
- Shared memory access control
- Processor scheduling
- Network routers and communication systems

4.4.8 Summary

The Round Robin Arbiter was successfully designed and verified using Verilog HDL. The rotating priority mechanism ensures fair resource allocation while preventing starvation, making the design suitable for arbitration in FPGA and ASIC-based digital systems.

4.5 Serializer–Deserializer (SerDes)

4.5.1 Introduction

A Serializer–Deserializer (SerDes) system is used to enable efficient data communication by converting parallel data into serial form and reconstructing it back at the receiver. This approach reduces interconnection complexity, minimizes wiring overhead, and improves transmission efficiency in digital systems.

In this work, a 4-bit Serializer and Deserializer were designed and implemented using Verilog HDL.

4.5.2 Importance of SerDes

- Reduces the number of interconnections in digital systems
- Enables efficient high-speed data transmission
- Minimizes area and hardware complexity
- Widely used in communication and VLSI systems

4.5.3 Schematic Implementation

The SerDes system consists of a serializer, deserializer, and ADC/DAC interfaces for mixed-signal simulation. The serializer converts parallel data into serial form, while the deserializer reconstructs the original parallel data.

The schematic implementation is shown below.

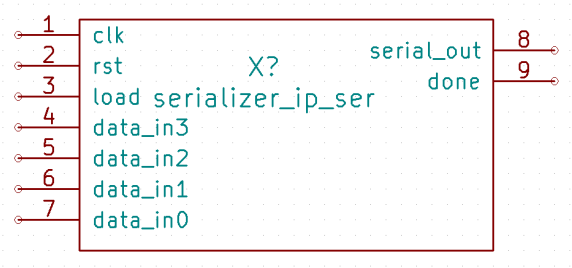


Figure 4.25: Serializer IP module

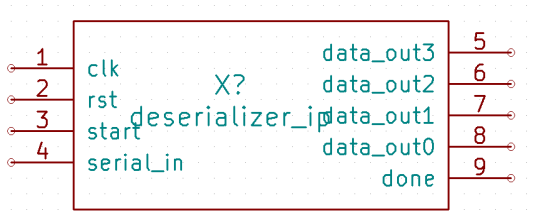


Figure 4.26: Deserializer IP module

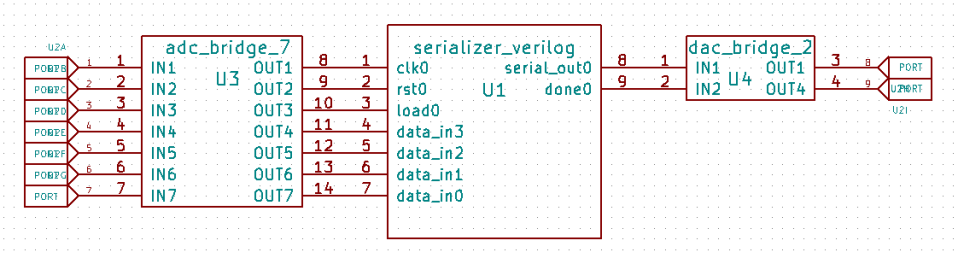


Figure 4.27: Serializer Subcircuit with ADC/DAC

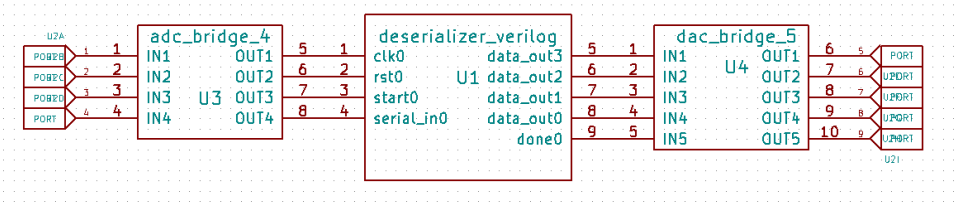


Figure 4.28: Deserializer Subcircuit with ADC/DAC

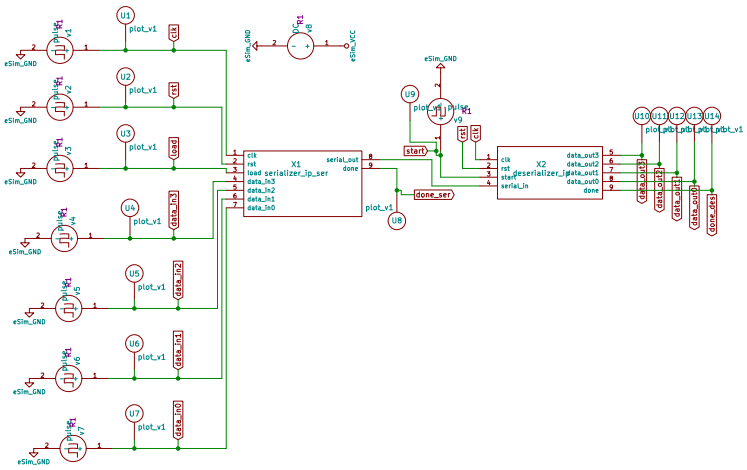


Figure 4.29: Schematic of SerDes

4.5.4 RTL Design and Verilog Implementation

The SerDes system was implemented using synthesizable Verilog HDL. A shift register-based approach was used for serialization and deserialization to ensure synchronized serial data transfer.

The Verilog implementation of the Serializer and Deserializer is provided below.

```
module serializer_verilog (
    input clk, rst, load,
    input [3:0] data_in,
    output reg serial_out, done
);
    reg [3:0] shift_reg;
    reg [2:0] count;
    reg active;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            shift_reg <= 0;
            count <= 0;
            done <= 0;
            active <= 0;
            serial_out <= 0;
        end
        else begin
            if (load && !active) begin
                shift_reg <= data_in;
                count <= 0;
                done <= 0;
                active <= 1;
            end
            else if (active) begin
                serial_out <= shift_reg[0];
                shift_reg <= shift_reg >> 1;
                count <= count + 1;
                if (count == 3) begin
                    done <= 1;
                    active <= 0;
                end
            end
            else begin
                done <= 0;
                serial_out <= 0;
            end
        end
    end
endmodule
```

Listing 4.5: Verilog Code for Serializer

```verilog
module deserializer_verilog (
    input clk,
    input rst,
    input start,
    input serial_in,
    output reg [3:0] data_out,
    output reg done
);
    reg [3:0] shift_reg;
    reg [2:0] count;
    reg active, serial_in_d;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            shift_reg <= 0;
            count <= 0;
            done <= 0;
            active <= 0;
            data_out <= 4'b0000;
            serial_in_d <= 0;
        end
        else begin
            serial_in_d <= serial_in;
            done <= 0;
            if (start && !active) begin
                count <= 0;
                active <= 1;
            end
            else if (active) begin
                shift_reg <= {serial_in_d, shift_reg[3:1]};
                count <= count + 1;
                if (count == 3) begin
                    data_out <= {serial_in_d, shift_reg[3:1]};
                    done <= 1;
                    active <= 0;
                end
            end
        end
    end
endmodule
```

Listing 4.6: Verilog Code for Deserializer

4.5.5 Functional Description

The SerDes system performs data conversion between parallel and serial domains.

- **Serializer:** Converts 4-bit parallel input data into serial form.
- **Deserializer:** Reconstructs serial input data into parallel output.
- **ADC/DAC Interfaces:** Enable mixed-signal simulation in eSim.

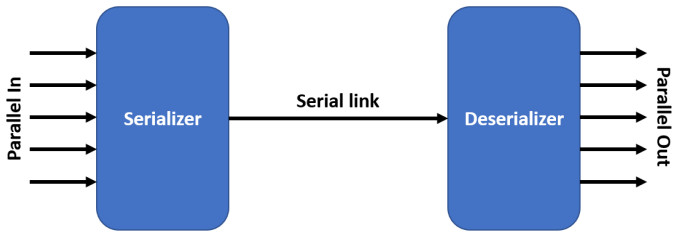


Figure 4.30: Block Diagram of SerDes

4.5.6 Simulation Results and Verification

The Serializer–Deserializer system was verified using **eSim** and **Vivado**. Different test cases were applied to validate correct serialization and deserialization of data.

The simulation results confirmed proper serial data transfer and successful reconstruction of the original parallel data.

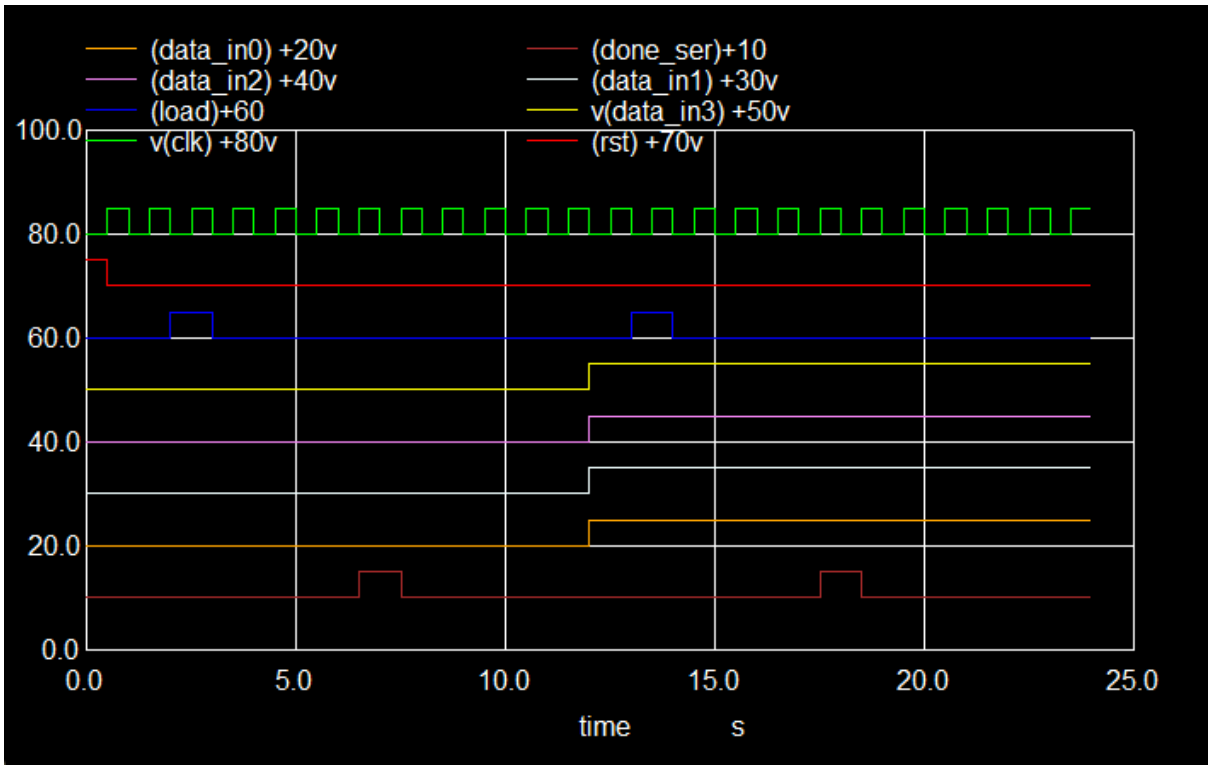


Figure 4.31: Simulation Results in eSim

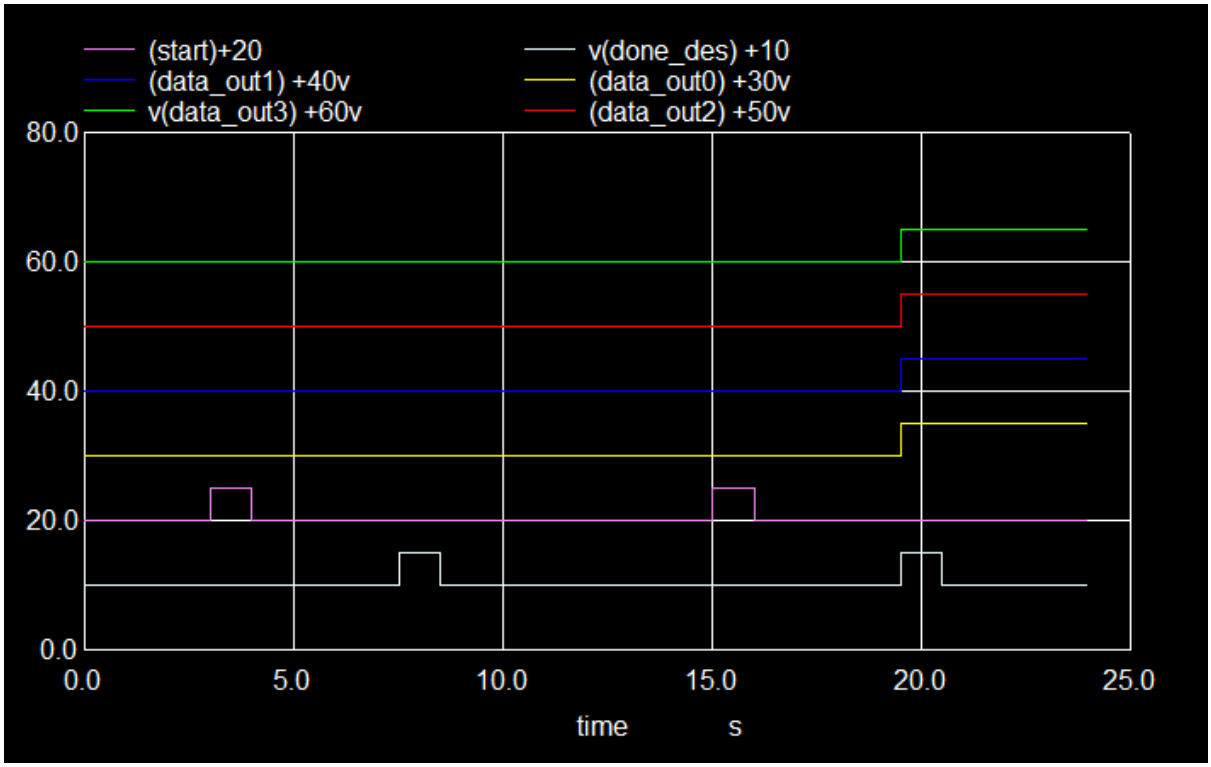


Figure 4.32: Simulation Results in eSim

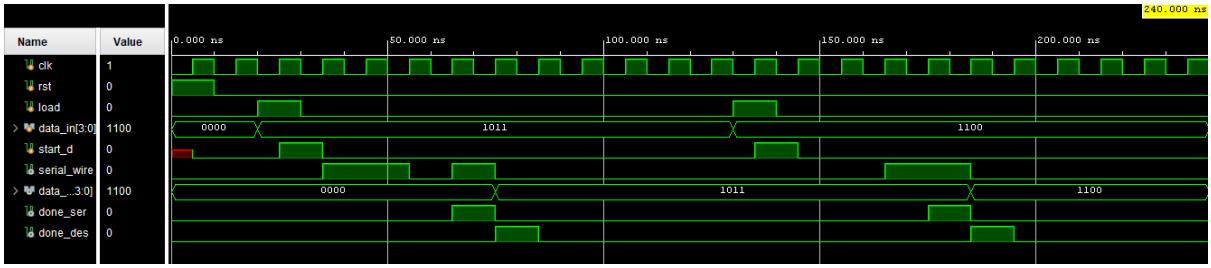


Figure 4.33: Simulation Results in Vivado

4.5.7 Applications

- High-speed communication interfaces
- VLSI and SoC interconnect systems
- Embedded and networking systems
- Mixed-signal communication architectures

4.5.8 Summary

The Serializer–Deserializer system was successfully designed and verified using Verilog HDL. The implementation enabled reliable parallel-to-serial and serial-to-parallel conversion, making the design suitable for communication and mixed-signal digital systems.

4.6 Handshake Pulse Synchronizer

4.6.1 Introduction

A Handshake Pulse Synchronizer is a Clock Domain Crossing (CDC) circuit used to safely transfer single-cycle pulses between asynchronous clock domains. Since direct pulse transfer may lead to metastability and pulse loss, the synchronizer combines a **2-Flip-Flop (2FF) synchronizer** with a **handshaking mechanism** to ensure reliable pulse transfer.

The design converts an input pulse into a level-based request signal, synchronizes it to the destination clock domain, and reconstructs the pulse using edge detection. An acknowledge signal completes the handshake process.

4.6.2 Importance of Handshake Pulse Synchronizer

- Enables safe pulse transfer across asynchronous clock domains.
- Reduces metastability and pulse loss issues.
- Ensures glitch-free and reliable pulse synchronization.
- Widely used in CDC and SoC-based digital systems.

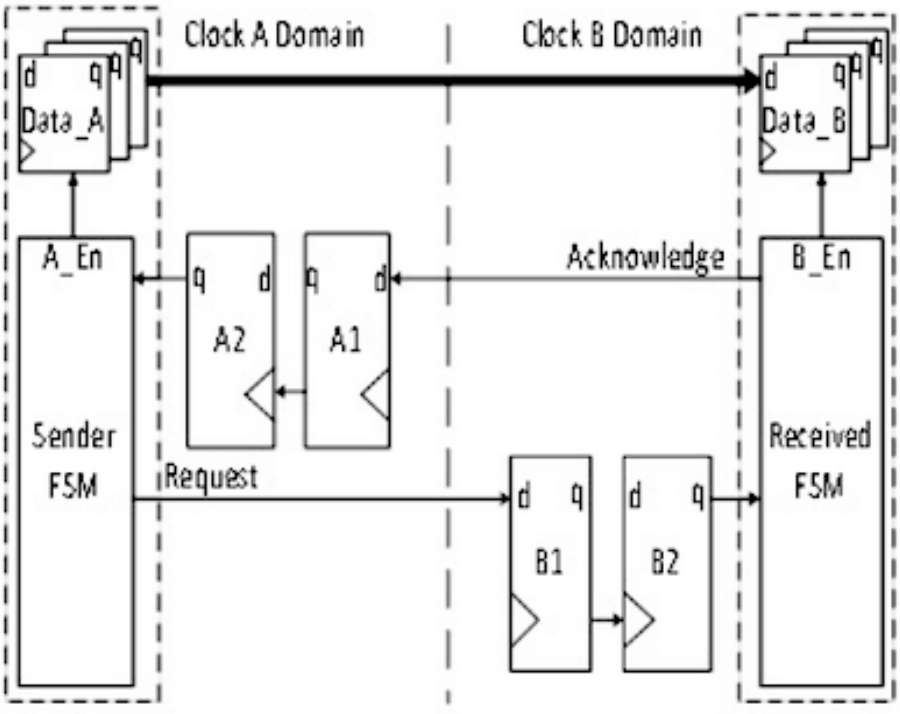


Figure 4.34: Block Diagram of Handshake Pulse Synchronizer

4.6.3 Schematic Implementation

The Handshake Pulse Synchronizer consists of source and destination clock domains connected through a request–acknowledge mechanism. A 2FF synchronizer is used to safely synchronize signals between clock domains, while edge detection logic generates the synchronized output pulse.

The schematic implementation and IP block representation are shown below.

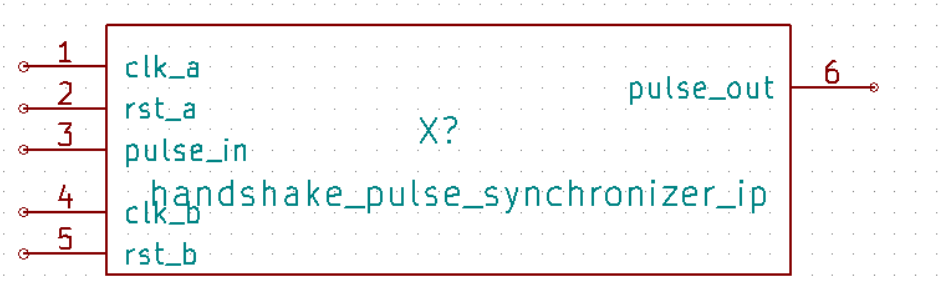


Figure 4.35: Handshake Pulse Synchronizer IP module

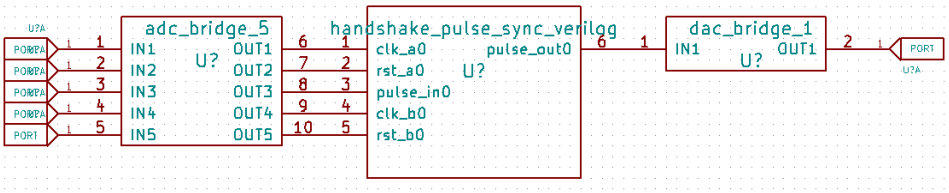


Figure 4.36: Subcircuit integrating ADC and DAC blocks

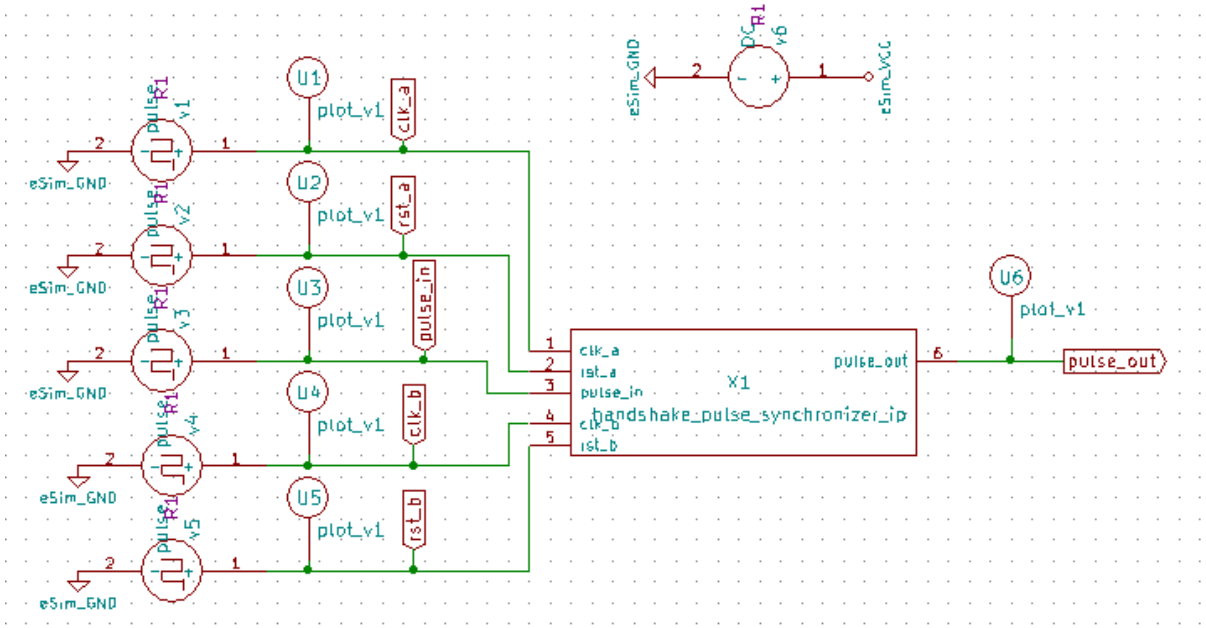


Figure 4.37: Schematic setup for Handshake Pulse Synchronizer

4.6.4 RTL Design and Verilog Implementation

The Handshake Pulse Synchronizer was implemented using synthesizable Verilog HDL. The design uses `always` blocks to synchronize request and acknowledge signals across clock domains using 2FF stages. Edge detection logic is used in the destination domain to generate a synchronized output pulse.

```
module handshake_pulse_sync_verilog (
    input wire clk_a,
    input wire rst_a,
    input wire pulse_in,
    input wire clk_b,
    input wire rst_b,
    output wire pulse_out
);

// Source Domain
reg req;
reg ack_sync1, ack_sync2;
wire ack;

// Synchronize acknowledge signal
always @(posedge clk_a or posedge rst_a) begin
    if (rst_a) begin
        ack_sync1 <= 0;
        ack_sync2 <= 0;
    end
    else begin
        ack_sync1 <= ack;
        ack_sync2 <= ack_sync1;
    end
end

// Request generation
always @(posedge clk_a or posedge rst_a) begin
    if (rst_a)
        req <= 0;
    else if (pulse_in)
        req <= 1;
    else if (ack_sync2)
        req <= 0;
end
```

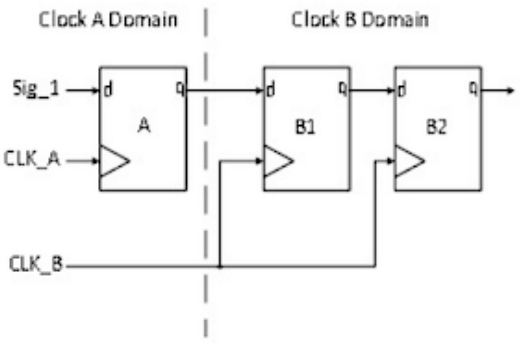
```
36
37 // Destination Domain
38 reg req_sync1, req_sync2;
39 reg req_d;
40 reg ack_reg;
41
42 assign ack = ack_reg;
43
44 // Synchronize request signal
45 always @(posedge clk_b or posedge rst_b) begin
46     if (rst_b) begin
47         req_sync1 <= 0;
48         req_sync2 <= 0;
49     end
50     else begin
51         req_sync1 <= req;
52         req_sync2 <= req_sync1;
53     end
54 end
55
56 // Edge detection
57 always @(posedge clk_b or posedge rst_b) begin
58     if (rst_b)
59         req_d <= 0;
60     else
61         req_d <= req_sync2;
62 end
63
64 assign pulse_out = req_sync2 & ~req_d;
65
66 // Acknowledge generation
67 always @(posedge clk_b or posedge rst_b) begin
68     if (rst_b)
69         ack_reg <= 0;
70     else if (req_sync2)
71         ack_reg <= 1;
72     else
73         ack_reg <= 0;
74 end
75 endmodule
```

Listing 4.7: Handshake Pulse Synchronizer Verilog Code

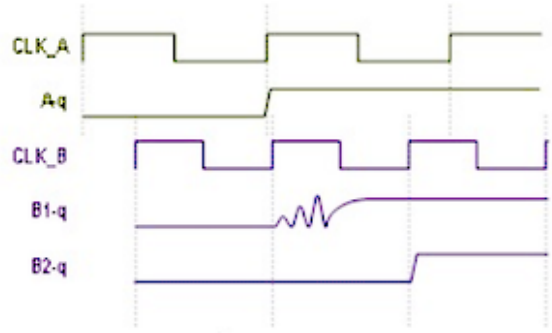
4.6.5 Functional Description

The Handshake Pulse Synchronizer transfers pulses between two asynchronous clock domains using a request–acknowledge mechanism.

- **Request Generation:** The input pulse sets a request signal in the source domain.
- **Synchronization:** The request signal is synchronized to the destination domain using a 2FF synchronizer.
- **Pulse Generation:** Edge detection generates a single-cycle synchronized output pulse.
- **Acknowledge Signal:** An acknowledge signal is returned to reset the request and complete the handshake.



(a) 2-FF Synchronizer



(b) Timing of 2-FF Synchronizer

4.6.6 Simulation Results and Verification

The Handshake Pulse Synchronizer was verified using **eSim** and **Vivado** simulations. The synchronized output pulse was generated correctly for each valid input pulse without pulse duplication or loss.

The simulation results confirmed reliable Clock Domain Crossing and metastability-safe pulse transfer.

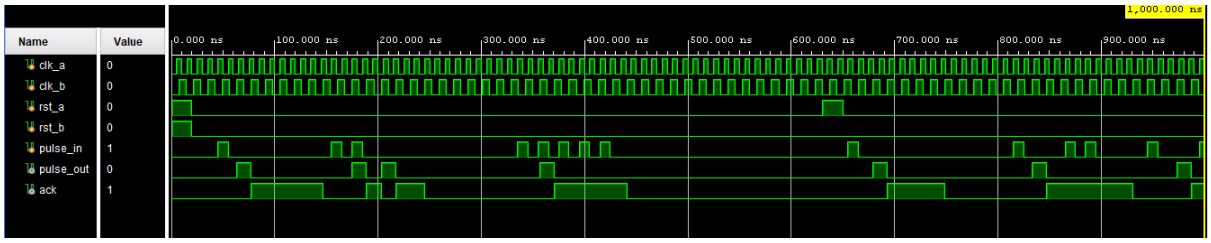


Figure 4.39: Simulation Results in Vivado

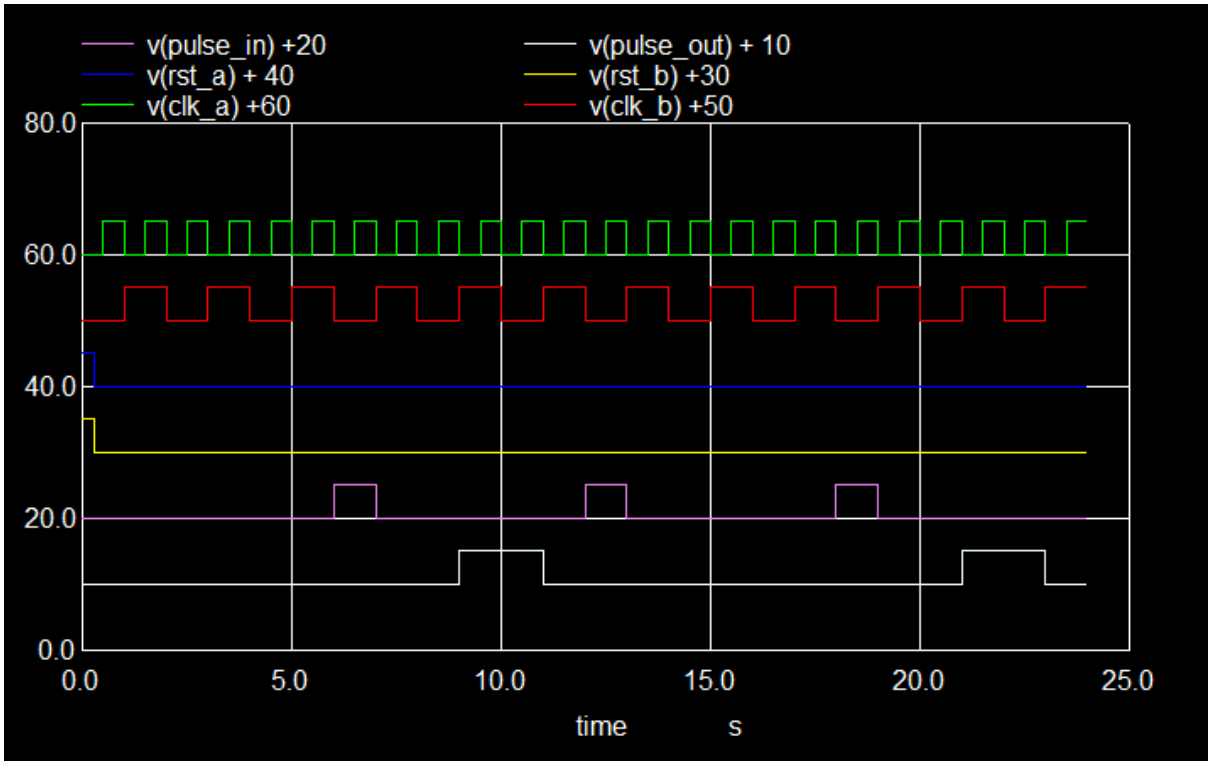


Figure 4.40: Simulation Results in eSim

4.6.7 Applications

- Clock Domain Crossing (CDC) circuits
- System-on-Chip (SoC) communication
- Interrupt signal synchronization
- FIFO control logic
- Multi-clock digital systems

4.6.8 Summary

The Handshake Pulse Synchronizer was successfully designed and implemented for reliable pulse transfer across asynchronous clock domains. By combining 2FF synchronization and handshaking, the design ensures metastability-safe, glitch-free, and lossless pulse synchronization. Simulation results validated the expected functionality, making the design suitable for CDC-based FPGA and ASIC applications.

4.7 Programmable Timer with Interrupt

4.7.1 Introduction

A Programmable Timer with Interrupt is a digital timing IP block used to generate accurate delays and interrupt signals in embedded systems and digital hardware. It performs a countdown operation using a programmable count value and generates an interrupt pulse when the terminal count reaches zero.

Since timing control is essential in **System-on-Chip (SoC)** architectures, processors, and real-time digital systems, programmable timers are widely used for scheduling, time-out generation, and periodic event control. The developed timer supports both **one-shot** and **periodic** operating modes for flexible timing applications.

4.7.2 Importance of Programmable Timer with Interrupt

- Provides accurate and programmable delay generation.
- Generates interrupt signals without continuous processor monitoring.
- Supports both one-shot and periodic timing operations.
- Suitable for embedded processors and real-time systems.

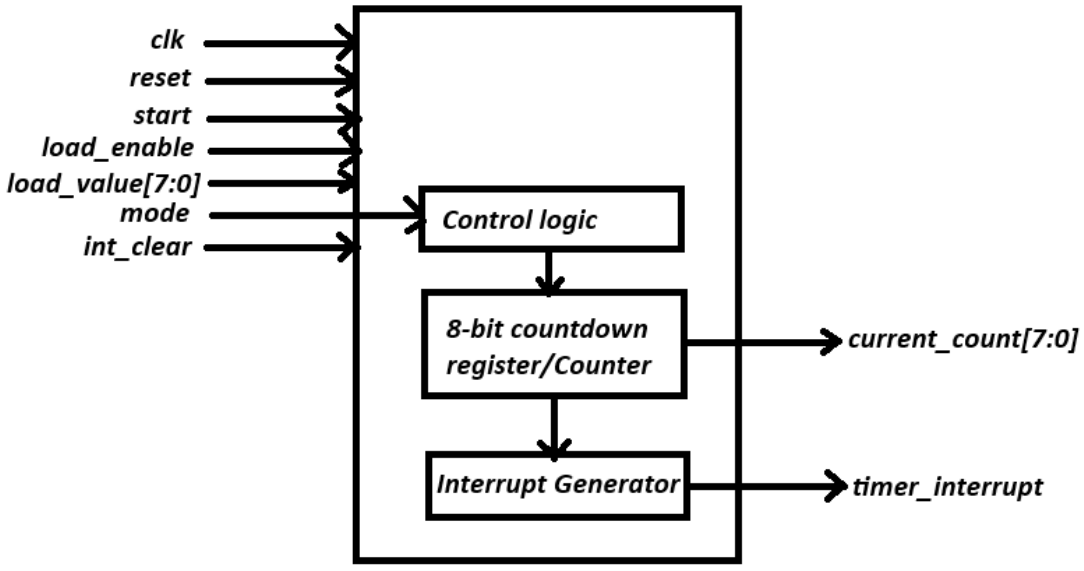


Figure 4.41: Block Diagram of Programmable Timer

4.7.3 Schematic Implementation

The Programmable Timer with Interrupt consists of a programmable counter, control logic, and interrupt generation circuitry. The timer loads a user-defined count value and decrements synchronously with the applied clock signal. Once the count reaches zero, an interrupt pulse is generated.

The timer supports two operating modes:

- **One-shot mode:** Generates a single interrupt and stops operation.
- **Periodic mode:** Automatically reloads the count value and continues repetitive timing cycles.

The schematic implementation and IP block representation are shown below.

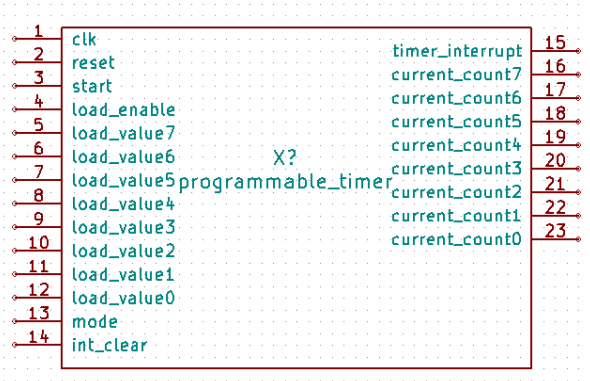


Figure 4.42: Programmable timer IP module

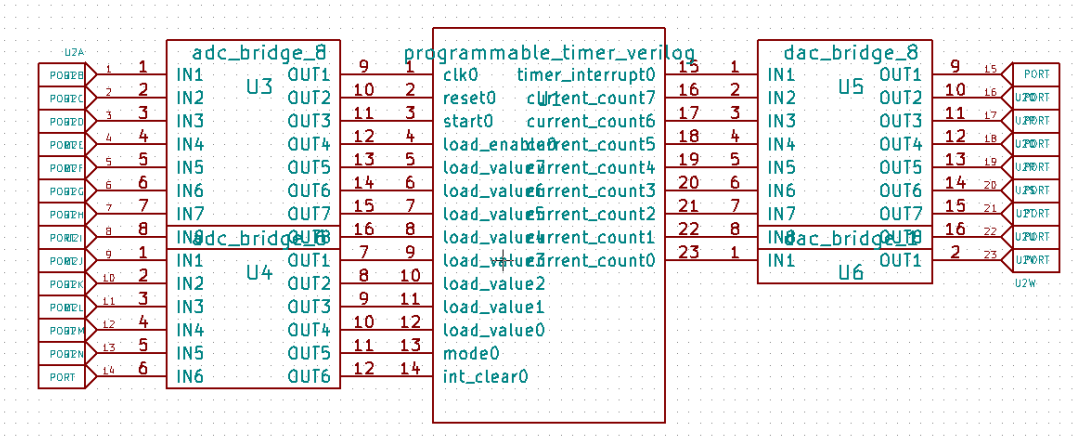


Figure 4.43: Subcircuit Integrating ADC and DAC Blocks

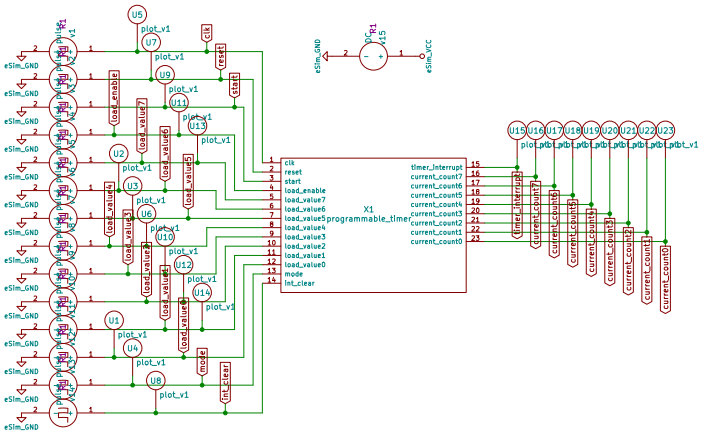


Figure 4.44: Schematic setup for Programmable Timer with Interrupt

Timer Mode Selection

Mode	Operation	Description
0	One-Shot Mode	Counts once and stops
1	Periodic Mode	Reloads and repeats counting

Figure 4.45: Programmable Mode Selection

4.7.4 RTL Design and Verilog Implementation

The Programmable Timer with Interrupt was implemented using synthesizable Verilog HDL. The design uses an `always` block triggered at the positive edge of the clock to control timer loading, countdown operation, and interrupt generation.

The timer initially loads a programmable count value through the `load_enable` signal. Once the `start` signal is enabled, the timer decrements its count value on every clock cycle. At terminal count, the `timer_interrupt` signal is asserted. In periodic mode, the timer reloads automatically, while in one-shot mode, operation stops after interrupt generation.

Listing 4.8: Programmable Timer with Interrupt Verilog Code

```
module programmable_timer_verilog(
    input wire clk,
    input wire reset,
    input wire start,
    input wire load_enable,
    input wire [7:0] load_value,
    input wire mode,
    input wire int_clear,
    output reg timer_interrupt,
    output reg [7:0] current_count
);
reg running;
always @(posedge clk or posedge reset) begin
    if (reset) begin
        current_count <= 8'd0;
        timer_interrupt <= 1'b0;
        running <= 1'b0;
    end
    else begin
        timer_interrupt <= 1'b0;
        if (load_enable) begin
            current_count <= load_value;
            running <= 1'b0;
        end
        else if (start) begin
            running <= 1'b1;
        end
        if (running) begin
            if (current_count > 0)
                current_count <= current_count - 1;
            else begin
                timer_interrupt <= 1'b1;
                if (mode)
                    current_count <= load_value;
                else
                    running <= 1'b0;
            end
        end
        if (int_clear)
            timer_interrupt <= 1'b0;
    end
end
endmodule
```

4.7.5 Functional Description

The Programmable Timer with Interrupt performs synchronous countdown operation using a programmable input value.

- **Count Loading:** The timer loads an external count value using the `load_enable` signal.
- **Countdown Operation:** After the `start` signal is enabled, the timer decrements on each positive clock edge.
- **Interrupt Generation:** When the count reaches zero, a single-cycle interrupt pulse is generated.
- **Mode Selection:** The timer either stops (one-shot mode) or reloads automatically (periodic mode).

S.No	Test Condition	Observed Result
1	Reset signal activated	Current count and interrupt initialized to logic low
2	Load enable asserted	Programmed binary count value loaded successfully
3	Start signal enabled	Timer started synchronous countdown operation
4	Countdown sequence	Count changed as 5 → 4 → 3 → 2 → 1 → 0
5	Terminal count reached	Interrupt pulse generated for one clock cycle
6	One-shot mode	Timer stopped after single interrupt generation
7	Periodic mode	Timer reloaded automatically and repeated countdown

Figure 4.46: Observed output for several test conditions

4.7.6 Simulation Results and Verification

The Programmable Timer with Interrupt was verified using **eSim** and **Vivado** simulations. Different timer operations such as programmable count loading, countdown execution, interrupt generation, and mode selection were tested.

The simulation results confirmed correct timer operation for both one-shot and periodic modes. The generated interrupt pulse matched the expected behavior at terminal count, validating the reliability of the design.

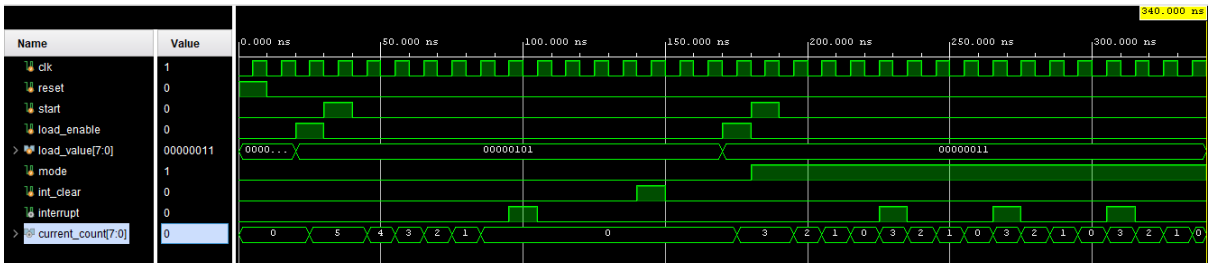


Figure 4.47: Simulation Result in Vivado

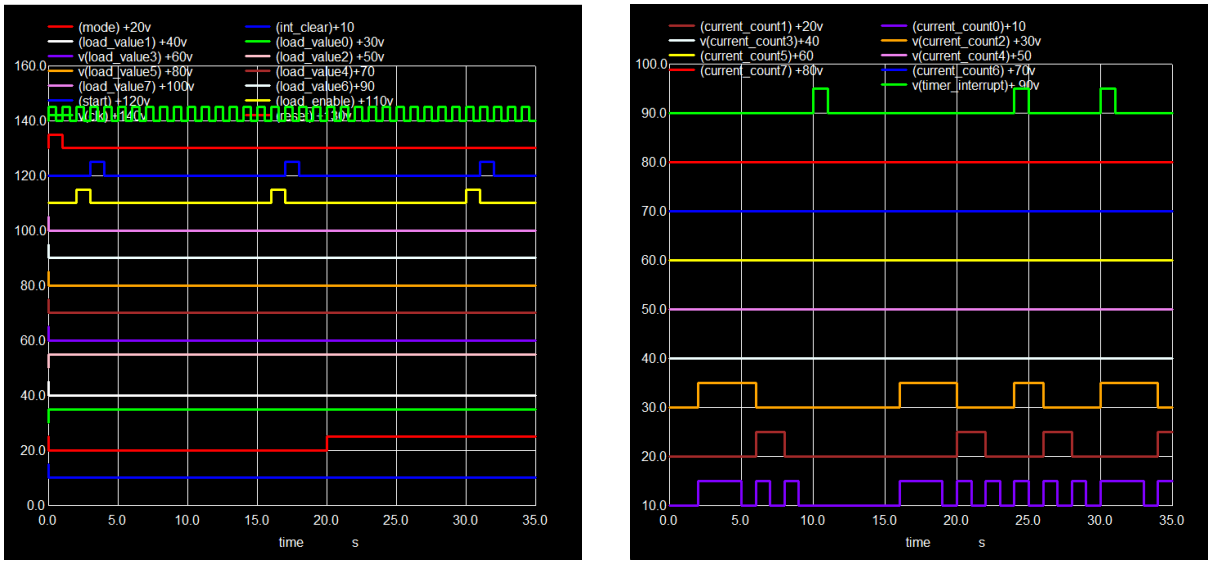


Figure 4.48: Timer Results in eSim showing One-shot and Periodic modes

4.7.7 Applications

- Embedded processor timer modules
- Task scheduling and timeout generation
- Watchdog timer implementation
- Periodic event triggering systems
- Interrupt-based real-time monitoring

4.7.8 Summary

The Programmable Timer with Interrupt was successfully designed and implemented for programmable delay generation and interrupt control. By supporting both one-shot and periodic modes, the design provides flexible timing functionality for digital systems. Simulation results validated correct countdown behavior and interrupt generation, making the timer suitable for embedded, FPGA, and ASIC-based applications.

4.8 Debounce Controller

4.8.1 Introduction

A Debounce Controller is a digital signal-conditioning circuit used to eliminate unwanted glitches and bounce noise generated by mechanical switches, push buttons, sensors, and asynchronous digital inputs. During signal transitions, these devices may produce multiple temporary fluctuations that can lead to false triggering and unreliable system behavior if directly processed.

The Debounce Controller ensures stable signal processing by validating an input transition only after it remains unchanged for a predefined number of clock cycles. In this design, a **programmable debounce mechanism** is implemented using a **2-bit mode selection input**, allowing different debounce delays for flexible operation under varying noise conditions.

4.8.2 Importance of Debounce Controller

- Eliminates glitches and switch bounce noise from digital inputs.
- Prevents false triggering in sequential and control circuits.
- Provides stable and reliable digital signal conditioning.
- Offers programmable debounce delay for different operating environments.
- Widely used in embedded systems, FPGA, and ASIC-based digital designs.

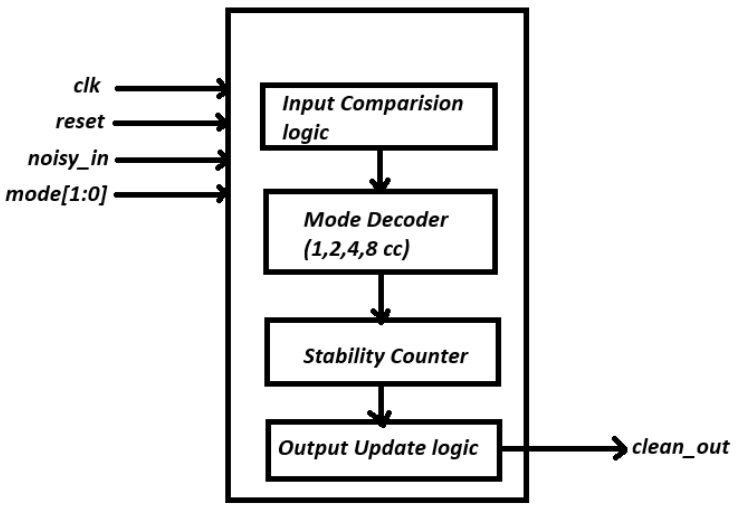


Figure 4.49: Block Diagram of Debounce Controller

4.8.3 Schematic Implementation

The Debounce Controller is designed as a synchronous digital filtering circuit that continuously monitors the noisy input signal and updates the clean output only after validating signal stability for a selected number of clock cycles.

The design includes clock, reset, noisy input, and mode selection signals. The selected debounce delay determines the number of stable clock cycles required before updating the filtered output signal.

The schematic implementation and IP block representation are shown below.

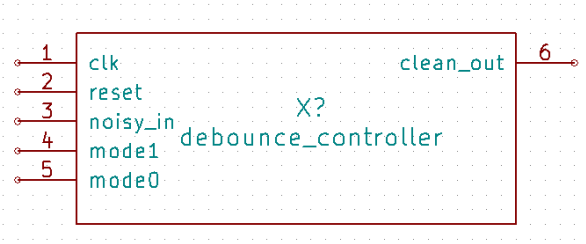


Figure 4.50: Debounce Controller IP module

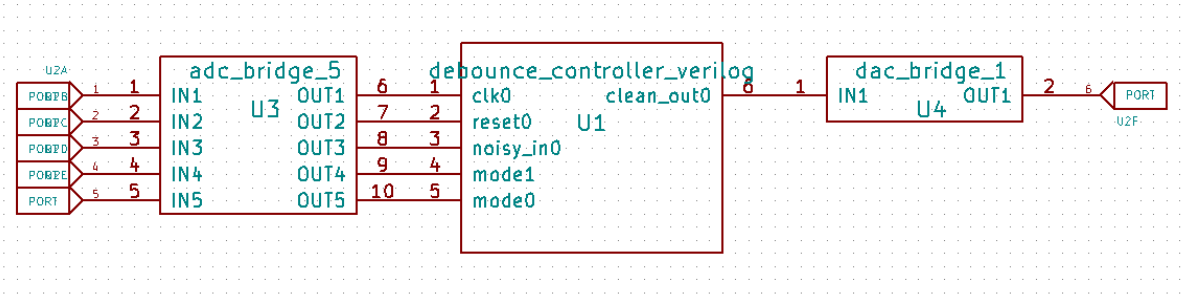


Figure 4.51: Subcircuit integrating ADC and DAC blocks

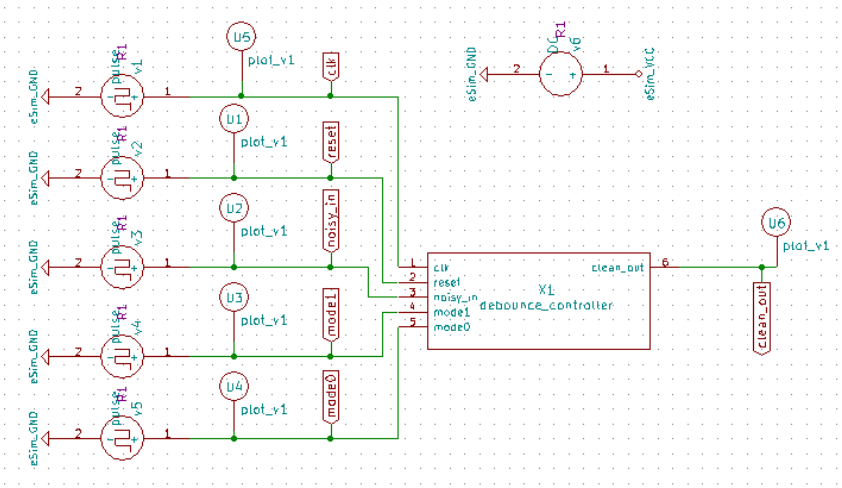


Figure 4.52: Schematic of Debounce Controller

4.8.4 Functional Description

The Debounce Controller filters noisy digital inputs by validating signal transitions only after a predefined stability period.

- **Input Monitoring:** The controller continuously checks the noisy input signal at every clock cycle.
- **Stability Detection:** If a new input value is detected, an internal counter begins counting stable clock cycles.
- **Mode-Based Delay Selection:** A 2-bit mode input determines the debounce delay of 1, 2, 4, or 8 clock cycles.
- **Output Update:** The clean output is updated only after the input remains stable for the selected duration.
- **Glitch Rejection:** Short transient pulses and bounce noise are ignored automatically.

Mode[1:0]	Stable Clock Cycles	Operation
00	1 Cycle	Fast Response
01	2 Cycles	Normal Filtering
10	4 Cycles	Strong Filtering
11	8 Cycles	Maximum Filtering

Figure 4.53: Debounce Delay Selection

4.8.5 RTL Design and Verilog Implementation

The Debounce Controller was implemented using synthesizable Verilog HDL. The design continuously compares the noisy input signal with the candidate stable value and uses a counter-based validation mechanism to confirm stable transitions.

A `mode` input selects the required debounce delay corresponding to 1, 2, 4, or 8 clock cycles. Only after the input remains stable for the selected duration is the output updated.

The Verilog implementation of the design is provided below.

```verilog
module debounce_controller_verilog(
    input clk,
    input reset,
    input noisy_in,
    input [1:0] mode,
    output reg clean_out
);
    reg [3:0] count;
    reg candidate;
    reg [3:0] stable_cycles;
    // Mode decode
    always @(*) begin
        case (mode)
            2'b00: stable_cycles = 1;
            2'b01: stable_cycles = 2;
            2'b10: stable_cycles = 4;
            2'b11: stable_cycles = 8;
        endcase
    end
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            clean_out <= 0;
            candidate <= 0;
            count <= 0;
        end
        else begin
            if (noisy_in != candidate) begin
                candidate <= noisy_in;
                count <= 1;
            end
            else if (count < stable_cycles) begin
                count <= count + 1;
            end
            if (count == stable_cycles)
                clean_out <= candidate;
        end
    end
endmodule
```

Listing 4.9: Verilog Code for Debounce Controller

4.8.6 Simulation Results and Verification

The Debounce Controller was verified using both **eSim** and **Vivado** simulations. The controller successfully filtered short glitches and updated the clean output only after the selected debounce duration.

Simulation results confirmed correct operation for all four debounce modes and demonstrated reliable signal stabilization without false triggering.

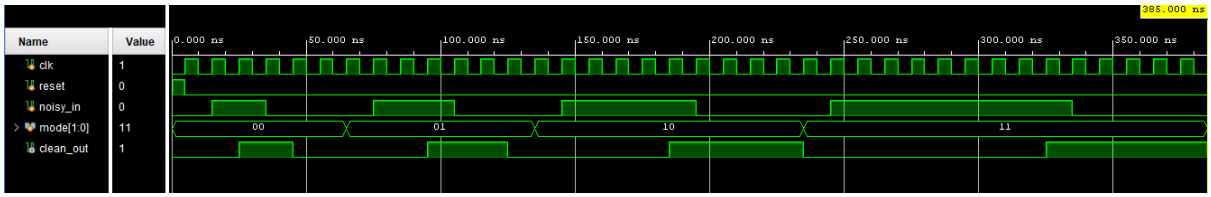
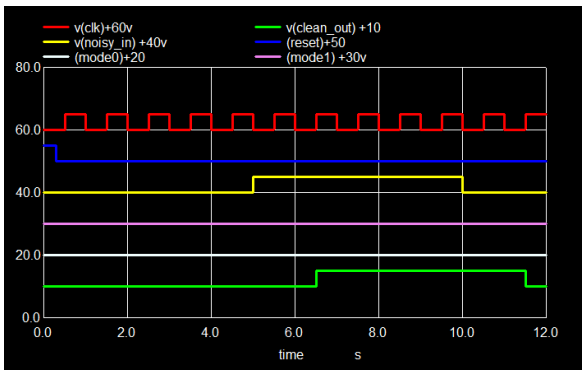
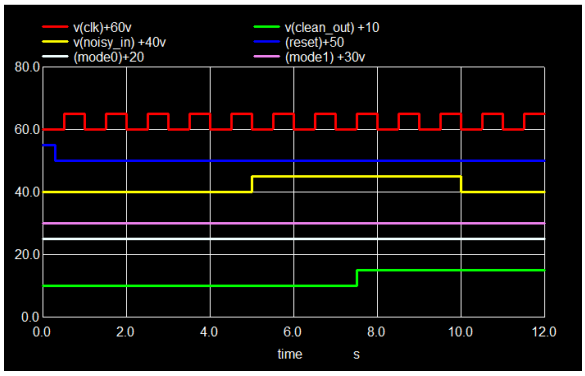


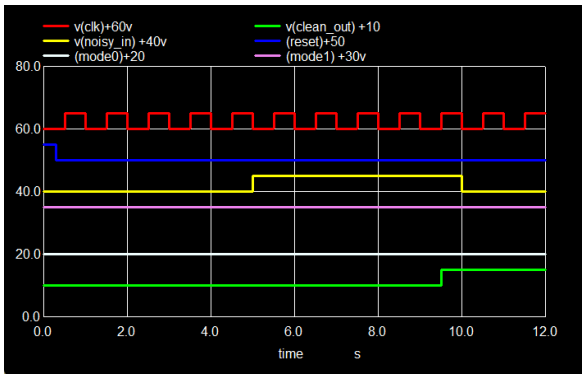
Figure 4.54: Simulation Results in Vivado



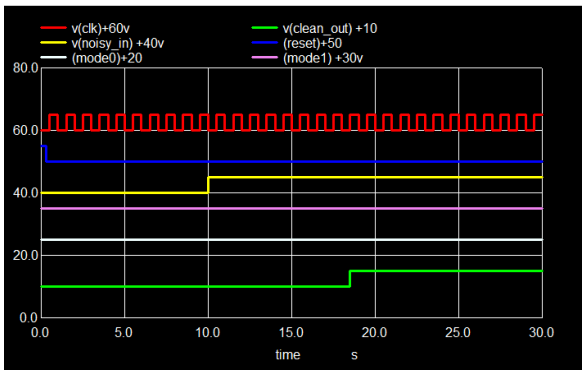
(a) Mode 00 (1 Clock Cycle)



(b) Mode 01 (2 Clock Cycles)



(c) Mode 10 (4 Clock Cycles)



(d) Mode 11 (8 Clock Cycles)

Figure 4.55: Debounce Controller Operation for Different Modes

4.8.7 Applications

- Push-button and switch interfacing
- Sensor signal conditioning
- Embedded system input stabilization
- FPGA and ASIC digital designs
- Industrial automation and control systems
- Noise filtering in communication interfaces

4.8.8 Summary

The Debounce Controller was successfully designed and implemented for reliable filtering of noisy digital input signals. By incorporating programmable debounce delays, the design effectively suppresses glitches and validates only stable input transitions. Simulation results verified correct operation across all debounce modes, making the controller suitable for reusable FPGA and ASIC-based digital signal-conditioning applications.

Chapter 5

Conclusion

This project successfully designed and verified multiple digital **Intellectual Property (IP) blocks** using **Verilog HDL** in the eSim environment. The implemented designs addressed important digital system requirements such as clock optimization, arbitration, serial communication, clock domain crossing, programmable timing, and signal conditioning.

The developed IPs include the Clock Gating Controller, Interrupt Controller, Round Robin Arbiter, Serializer–Deserializer (SerDes), Handshake Pulse Synchronizer, Programmable Timer with Interrupt, and Debounce Controller. Each module was implemented using synthesizable Verilog architecture and verified through mixed-signal simulation, with results matching the expected functionality.

This project provided practical exposure to **RTL design**, modular IP development, sequential logic, timing control, synchronization techniques, and digital verification. In addition, the integration of **ADC and DAC blocks** in eSim enabled mixed-signal validation and system-level analysis.

Overall, the developed IP blocks are reliable and reusable, making them suitable for FPGA and ASIC-based applications. This work also establishes a strong foundation for future learning in **VLSI design** and **System-on-Chip (SoC)** development.