Semester Long Internship Report

On

Designing of IP Blocks

& Verification using Mixed Signal Simulation in eSim

Submitted by

Srinivasa Reddy Badam

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 3, 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**, **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 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 advancement of Electronic Design Automation (EDA) tools has revolutionized the way electronic circuits are designed, simulated, and verified. Among these, open-source tools play a crucial role in making advanced EDA capabilities accessible to students, educators, and researchers without the burden of expensive licenses. The FOSSEE project at IIT Bombay has taken significant steps to promote such open-source tools globally.

One of the key open-source tools developed and promoted under FOSSEE is **eSim**, which integrates multiple open-source EDA tools to provide a unified platform for circuit design, simulation, and PCB layout. Various tools like NgSpice, Makerchip, KiCad, and GHDL work together under eSim to offer a complete design environment.

1.1 eSim

eSim is an open-source Electronic Design Automation (EDA) tool developed by FOSSEE, IIT Bombay. The primary objective of eSim is to provide a completely free and open-source alternative to commercial circuit design and simulation software, thereby reducing dependency on expensive proprietary tools in academia and research.

eSim integrates multiple open-source tools to provide a comprehensive design and simulation environment. KiCad is used for schematic capture and PCB design, while NgSpice performs analog circuit simulation. For digital simulation, GHDL is integrated supporting VHDL-based designs, and Makerchip enables online Transaction Level Verilog (TL-Verilog) based digital design and verification. Python support is also available in eSim, allowing users to perform customized simulations and generate netlists.

One of the key features of eSim is its **Subcircuit feature**, which allows complex designs to be modularized into reusable blocks, simplifying the design of large systems. Additionally, eSim supports device modeling, enabling users to define and simulate custom semiconductor devices using real-world parameters. As part of its continuous development, eSim also supports **SkyWater SKY130 PDK**, allowing users to perform simulations using an open-source 130nm process design kit suitable for analog, mixed-signal, and digital IC design research.

By supporting a fully open-source toolchain, eSim empowers students, educators, and researchers to gain hands-on experience with industry-relevant EDA tools and contribute actively to the open-source hardware ecosystem.

1.2 NgSpice

NgSpice is an open-source circuit simulator integrated into eSim for performing analog and mixed-signal simulations. Based on the SPICE simulation engine, NgSpice supports DC, AC, transient, and parametric analyses, making it suitable for analyzing a wide range of circuits.

It allows users to simulate circuit behavior using industry-standard SPICE models, including support for subcircuits and behavioral modeling. In eSim, NgSpice works as the backend simulator for schematics created using KiCad, providing waveform outputs for voltage, current, and frequency responses. Its flexibility and accuracy make it a powerful tool for verifying designs before hardware implementation.

1.3 Makerchip

Makerchip is an integrated platform designed to simplify digital circuit design by offering browser-based and desktop-based environments for coding, simulation, and debugging digital hardware designs. It supports multiple hardware description languages including Verilog, SystemVerilog, and Transaction-Level Verilog (TL-Verilog), allowing flexibility for users at various levels of abstraction.

In eSim, Makerchip is interfaced through the Makerchip IDE for digital design and verification. The platform integrates several open-source and proprietary tools to provide a rich set of features such as real-time simulation, waveform viewing, and code linting, thereby improving design accuracy and productivity.

1.4 KiCad

KiCad is an open-source PCB design and schematic capture tool integrated within eSim for creating circuit schematics and generating PCB layouts. It allows users to design multi-layer boards, define custom footprints, and perform design rule checks to ensure design correctness before fabrication.

In eSim, KiCad acts as the primary schematic editor, allowing users to graphically build circuits by placing and interconnecting components from extensive open-source libraries. The designed schematics can directly be used for both simulation and PCB layout generation. KiCad also supports features such as 3D visualization of PCBs, Gerber file export for manufacturing, and electrical rule checking to identify design issues early. Its seamless integration within eSim enables a smooth transition from schematic design to simulation and physical realization, offering a complete design-to-fabrication workflow entirely within an open-source environment.

1.5 GHDL

GHDL is an open-source simulator for VHDL, a hardware description language widely used in digital circuit design. It supports the complete IEEE VHDL standard, allowing designers to simulate, verify, and debug VHDL-based digital systems effectively.

In eSim, GHDL is integrated to enable digital simulation alongside analog simulation provided by NgSpice. Users can model digital circuits using VHDL, which are then simulated using GHDL to verify logical functionality and timing behavior. This integration allows eSim to handle mixed-signal designs, where the analog components are simulated by NgSpice and the digital components by GHDL, providing a comprehensive platform for complex system design. The combination of these tools allows designers to validate both analog and digital subsystems within a unified simulation environment.

Chapter 2

Features of eSim

eSim offers a comprehensive set of features that make it a powerful open-source alternative for electronic design automation. Some of the key features include:

- **Open-source and Free:** eSim is fully open-source, allowing unrestricted access without licensing costs, making it ideal for academic and research purposes.
- **Integrated Toolchain:** Combines multiple open-source tools such as KiCad for schematic capture and PCB design, NgSpice for analog simulation, GHDL for digital simulation, and Makerchip for advanced digital design.
- **Mixed-Signal Simulation:** Supports both analog and digital simulation, allowing users to design and simulate mixed-signal circuits efficiently.
- **Subcircuit Feature:** Enables hierarchical and modular design by allowing complex circuits to be broken into reusable subcircuits.
- **Device Modeling:** Supports custom device modeling, allowing users to simulate real-world semiconductor devices using user-defined parameters.
- **SkyWater SKY130 PDK Support:** Provides access to open-source 130nm process design kit for IC design and simulation.
- **Python Integration:** Allows automation, scripting, and advanced analysis using Python interfaces.
- **User-Friendly Interface:** Offers an intuitive GUI that simplifies circuit creation, simulation setup, and result analysis, making it suitable for both beginners and advanced users.
- **Cross-Platform Support:** Compatible with major operating systems like Linux and Windows.
- **Active Community and Documentation:** Extensive documentation, tutorials, and community support provided through FOSSEE ensure smooth learning and troubleshooting.

2.1 Mixed-Signal Simulation using NGVeri

This part involves using eSim's NGVeri feature for mixed-signal simulation by integrating analog circuits with custom-designed Verilog digital IP blocks. NGVeri enables simultaneous simulation of analog (NgSpice) and digital (Verilator) domains, allowing realistic verification of mixed-signal systems.

In this work, several reusable digital IP blocks such as arithmetic units, sequential logic modules, and control circuits were designed in Verilog HDL and integrated into eSim. These IP blocks were then interfaced with analog circuits to validate their behavior in practical mixed-signal environments. This approach ensures that digital logic interacts correctly with real-world analog signals, improving system-level reliability and design accuracy.

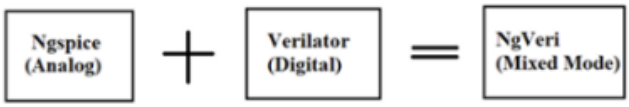

Figure 2.1: NGVeri mixed-signal simulation flow

2.1.1 Approach

- **Digital IP Design:** Reusable digital modules such as adders, multiplexers, counters, flip-flops, and control logic were designed using synthesizable Verilog HDL.
- **RTL Verification:** Each digital IP block was functionally verified using Icarus Verilog and GTKWave to ensure correct logical behavior before integration.
- **NGVeri Model Generation:** Verified Verilog modules were imported into NGVeri. The RTL code was compiled using Verilator to generate NGVeri-compatible mixed-signal models.
- **Analog–Digital Integration:** The generated digital IP blocks were instantiated within eSim schematics and connected with analog circuits designed using KiCad and NgSpice components.
- **Mixed-Signal Simulation:** Co-simulation was performed where analog signals from NgSpice interacted with digital logic executed by Verilator, enabling simultaneous analog–digital behavior analysis.
- **Result Verification:** Simulation outputs such as waveforms, timing relationships, and logic transitions were analyzed using eSim plotting tools to validate correct mixed-signal operation.

- **Reusable IP Validation:** The digital IP blocks were tested across multiple mixed-signal scenarios to confirm their reusability and reliability within the eSim design environment.

Chapter 3

Problem Statement

The objective of this project is to design and develop reusable Verilog-based Digital IP blocks and verify their functionality using mixed-signal simulation in eSim. The developed IPs are integrated into eSim through NGVeri to enable co-simulation of digital logic with analog circuits. This approach ensures correct interaction between digital modules and real-world analog signals, improving reliability and enabling system-level validation of digital designs within the eSim environment.

3.1 Digital IP Blocks Development

Digital IP blocks form the fundamental building units of complex digital systems. In this work, commonly used digital modules such as arithmetic circuits, sequential elements, and control logic were designed as reusable RTL IPs in Verilog HDL. These IPs were structured to be modular, synthesizable, and compatible with NGVeri integration inside eSim.

The developed IP blocks were intended for repeated use across different mixed-signal design scenarios. Each IP was verified independently and later validated within analog–digital co-simulation setups in eSim.

3.1.1 Approach

- **IP Selection:** Frequently used digital modules such as adders, multiplexers, counters, flip-flops, registers, and control units were identified as reusable IP candidates.
- **RTL Design:** Selected IP blocks were implemented using synthesizable Verilog HDL following modular and hierarchical design practices.
- **Functional Verification:** Testbenches were created and simulated using Icarus Verilog and GTKWave to verify correct logical functionality and timing behavior.

- **IP Packaging:** Verified RTL modules were organized as reusable digital IP blocks suitable for integration into the eSim–NGVeri flow.

3.2 Mixed-Signal Simulation using NGVeri

To validate the developed digital IP blocks in realistic operating conditions, mixed-signal simulation was performed using eSim's NGVeri framework. NGVeri enables simultaneous co-simulation of analog circuits using NgSpice and digital logic using Verilator, allowing interaction between continuous analog signals and discrete digital logic.

In this work, the designed digital IP blocks were integrated into eSim schematics and interfaced with analog components. This enabled verification of digital logic behavior under analog stimulus conditions, ensuring correct functionality in mixed-signal environments.

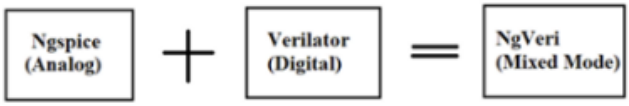

Figure 3.1: NGVeri flow for mixed-signal simulation

3.2.1 Approach

- **NGVeri Model Generation:** Verified Verilog RTL modules were compiled using Verilator to generate NGVeri-compatible digital simulation models.
- **Analog–Digital Integration:** Generated digital IP models were instantiated within eSim schematics and connected with analog circuits created using KiCad and NgSpice components.
- **Co-Simulation Setup:** Mixed-signal simulation environment was configured where NgSpice handled analog behavior and Verilator executed digital logic concurrently.
- **Mixed-Signal Verification:** Digital IP outputs were validated under analog input conditions, ensuring correct logic transitions, timing response, and functional interaction.
- **System-Level Validation:** The reusable IP blocks were tested across multiple mixed-signal scenarios to confirm reliability and compatibility within the eSim design ecosystem.

Chapter 4

Digital IP Design and Integration

Digital IP blocks enable modular and scalable digital system design by encapsulating commonly used functionality into reusable components. In this work, two FIFO designs were implemented and verified using mixed-signal simulation in eSim: an Asynchronous FIFO and a Synchronous FIFO.

4.1 Asynchronous FIFO Design and Implementation

4.1.1 General Description

The Asynchronous FIFO is a digital memory buffer that allows reliable data transfer between two independent clock domains. It is widely used in digital systems where producer and consumer modules operate at different clock frequencies or phases. The FIFO uses separate write and read clocks and maintains data integrity through Gray-code pointer synchronization and dual-clock memory architecture.

The designed FIFO supports parameterizable data width and depth, making it reusable across different digital systems. It provides status flags such as *full* and *empty* to control data flow between asynchronous domains. The architecture ensures metastability-safe pointer transfer using two-flip-flop synchronizers.

4.1.2 Features

- Independent write and read clock domains
- Gray-code pointer synchronization
- Dual-port memory architecture
- Full and Empty status flags
- Metastability-safe CDC operation
- Parameterizable width and depth

4.1.3 Control Logic

Condition	Write Operation	Read Operation
Full = 0	Write Enabled	–
Full = 1	Write Disabled	–
Empty = 0	–	Read Enabled
Empty = 1	–	Read Disabled

4.1.4 IP Symbol in eSim

The Verilog-based Asynchronous FIFO was imported into eSim through NGVeri, generating a reusable digital IP symbol with dual clock ports, data bus, and status outputs.

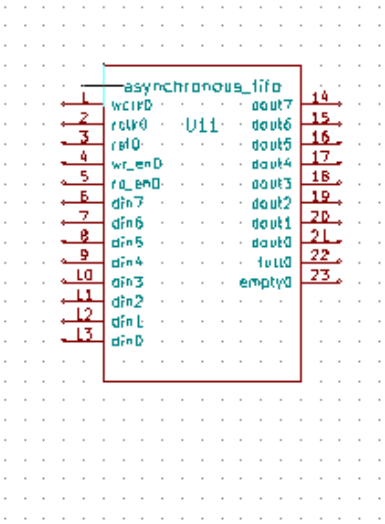

Figure 4.1: Asynchronous FIFO IP Symbol in eSim

4.1.5 Circuit

The FIFO circuit consists of dual-port memory, Gray-coded read and write pointer generators, and two-stage synchronizers for safe clock-domain crossing. Separate clock domains operate independently while synchronized pointers ensure correct full and empty detection.

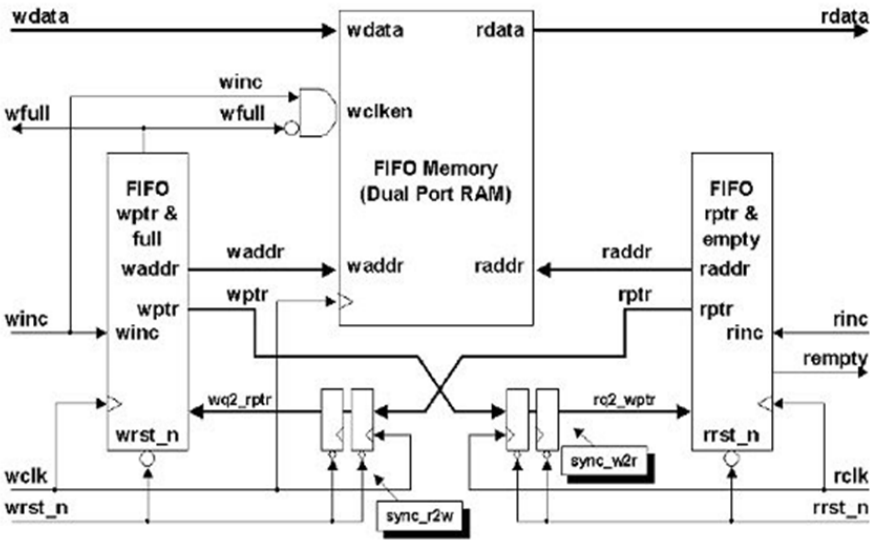

Figure 4.2: Asynchronous FIFO internal architecture

4.1.6 eSim Schematic Integration

The Asynchronous FIFO digital IP was instantiated in an eSim schematic and interfaced with clock sources, data stimulus, and observation probes to enable mixed-signal co-simulation using NGVeri.

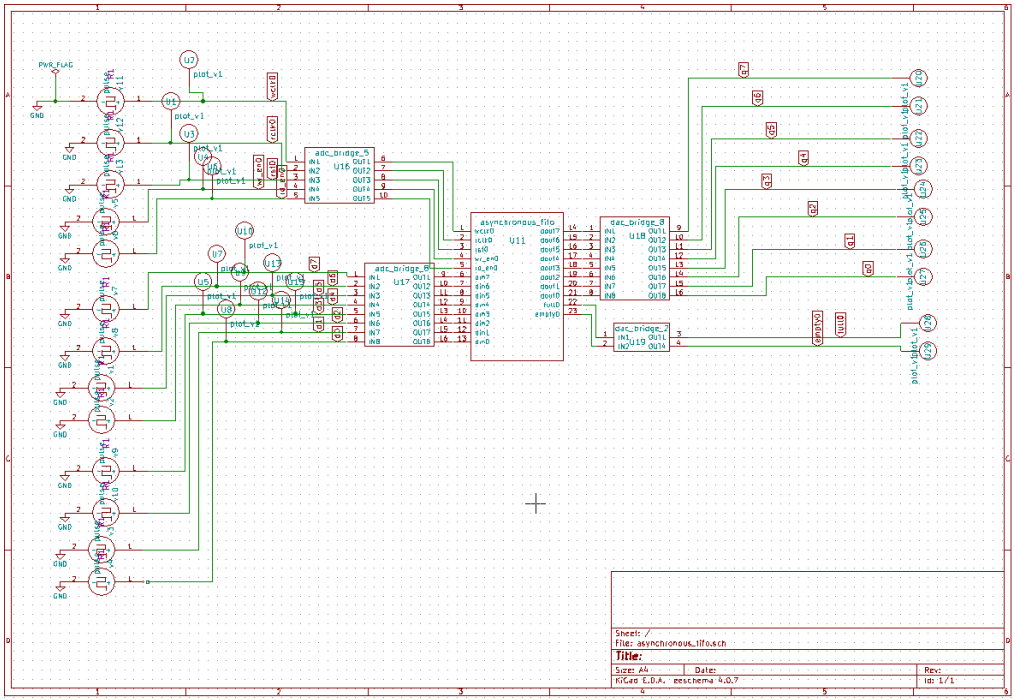

Figure 4.3: eSim Schematic of Asynchronous FIFO Integration

4.1.7 RTL Implementation

The Asynchronous FIFO Digital IP was implemented in Verilog HDL using dual clock domains and Gray-coded pointer synchronization. Two-stage synchronizers were used to safely transfer pointer values across clock domains, ensuring metastability-safe operation. The FIFO full condition is detected by comparing the write pointer with the inverted MSBs of the synchronized read pointer, while the empty condition occurs when both pointers are equal.

The RTL description of the asynchronous FIFO is shown below.

```
module asynchronous_fifo (
    input  wclk,
    input  rclk,
    input  rst,
    input  wr_en,
    input  rd_en,
    input  [7:0] din,
    output reg [7:0] dout,
    output full,
    output empty
);

localparam ADDR_WIDTH = 3;
localparam DEPTH = 8;

reg [7:0] mem [0:7];

reg [3:0] wptr_bin, rptr_bin;
reg [3:0] wptr_gray, rptr_gray;

reg [3:0] wptr_gray_sync1, wptr_gray_sync2;
reg [3:0] rptr_gray_sync1, rptr_gray_sync2;

wire [3:0] wptr_bin_next;
wire [3:0] wptr_gray_next;
wire [3:0] rptr_bin_next;
wire [3:0] rptr_gray_next;

assign wptr_bin_next  = wptr_bin + (wr_en & ~full);
assign wptr_gray_next = (wptr_bin_next >> 1) ^ wptr_bin_next;

assign rptr_bin_next  = rptr_bin + (rd_en & ~empty);
assign rptr_gray_next = (rptr_bin_next >> 1) ^ rptr_bin_next;

initial begin
    wptr_bin = 0;
    rptr_bin = 0;
    wptr_gray = 0;
    rptr_gray = 0;
    dout = 0;
end
```

Figure 4.4: Asynchronous FIFO RTL — Write and Pointer Logic

```
// WRITE DOMAIN
always @(posedge wclk) begin
    if (rst) begin
        wptr_bin  <= 0;
        wptr_gray <= 0;
    end else begin
        wptr_bin  <= wptr_bin_next;
        wptr_gray <= wptr_gray_next;

        if (wr_en && !full)
            mem[wptr_bin[2:0]] <= din;
    end
end

// READ DOMAIN
always @(posedge rclk) begin
    if (rst) begin
        rptr_bin  <= 0;
        rptr_gray <= 0;
        dout      <= 0;
    end else begin
        rptr_bin  <= rptr_bin_next;
        rptr_gray <= rptr_gray_next;

        if (rd_en && !empty)
            dout <= mem[rptr_bin[2:0]];
    end
end

// Synchronizers
always @(posedge wclk) begin
    if (rst) begin
        rptr_gray_sync1 <= 0;
        rptr_gray_sync2 <= 0;
    end else begin
        rptr_gray_sync1 <= rptr_gray;
        rptr_gray_sync2 <= rptr_gray_sync1;
    end
end

always @(posedge rclk) begin
    if (rst) begin
        wptr_gray_sync1 <= 0;
        wptr_gray_sync2 <= 0;
    end else begin
        wptr_gray_sync1 <= wptr_gray;
        wptr_gray_sync2 <= wptr_gray_sync1;
    end
end

assign empty = (rptr_gray == wptr_gray_sync2);

assign full =
    (wptr_gray_next ==
        {~rptr_gray_sync2[3:2],
          rptr_gray_sync2[1:0]});

endmodule
```

Figure 4.5: Asynchronous FIFO RTL — Read Logic and Synchronizers

4.1.8 Digital Simulation Results

Functional verification of the FIFO RTL was first performed using digital simulation. The waveform confirms correct write-read sequencing, pointer progression, and full/empty flag behavior.

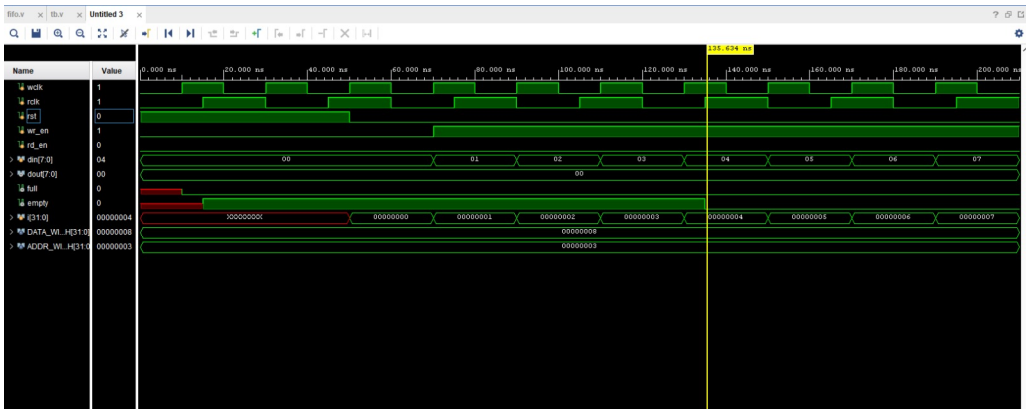

Figure 4.6: Digital Simulation Waveform of Asynchronous FIFO

4.1.9 Mixed-Signal Simulation Results

Mixed-signal co-simulation was performed in eSim using NGVeri, where the FIFO RTL executed in Verilator while asynchronous clocks and stimulus were generated through NgSpice. The results confirm correct operation across independent clock domains.

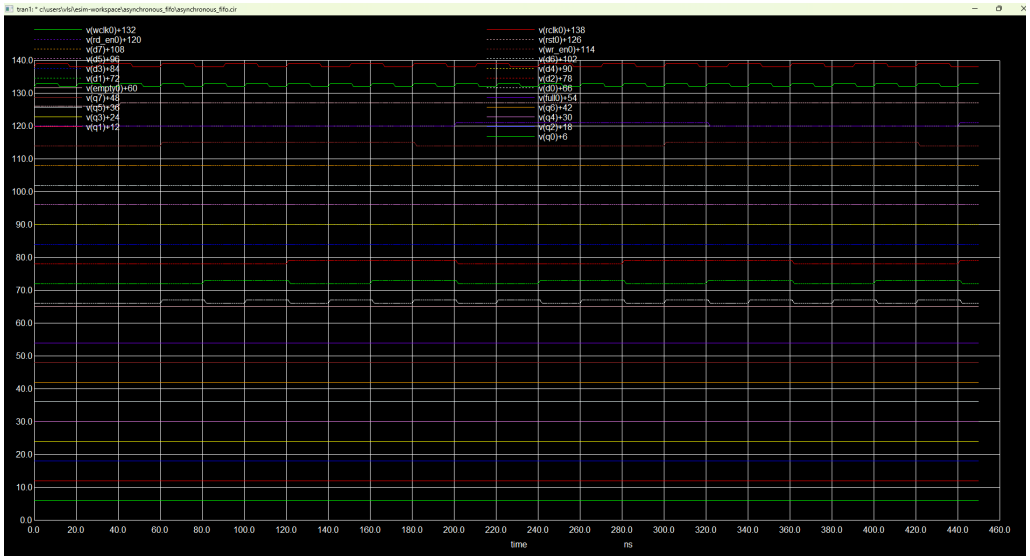

Figure 4.7: Mixed-Signal Simulation Waveforms in eSim

4.1.10 Conclusion

The designed Asynchronous FIFO Digital IP enables reliable data transfer between independent clock domains while maintaining data integrity and metastability safety. Gray-code pointer synchronization and dual-clock architecture ensure robust operation across varying clock frequencies. The IP was successfully verified using both digital and mixed-signal simulation in eSim and can be reused in complex digital and mixed-signal system designs.

4.2 Synchronous FIFO Design and Implementation

4.2.1 Introduction

FIFO (First-In First-Out) is a memory structure used for temporary data storage where the data that is written first is read first. FIFOs are commonly used in digital systems to handle data buffering between modules operating at different processing speeds.

A **Synchronous FIFO** operates with a single clock signal for both write and read operations. Since both operations share the same clock domain, the design is simpler and avoids issues related to clock domain crossing.

4.2.2 Why FIFO is Used

FIFO buffers are widely used in digital systems for several purposes:

- Temporary storage of streaming data
- Data rate matching between producer and consumer modules
- Decoupling processing stages in pipelines
- Preventing data loss when one module is faster than another
- Efficient communication between hardware blocks

4.2.3 Why Use Synchronous FIFO When Frequency is Same

Even when two modules operate at the same clock frequency, the **data rate** at which they produce or consume data may be different.

Example Scenario

Consider two modules:

- Module A produces data every clock cycle.
- Module B processes data once every few cycles.

Even though both modules share the same clock, the consumer cannot process incoming data immediately. Without buffering, data loss would occur.

A synchronous FIFO solves this problem by temporarily storing the produced data until the consumer is ready to read it.

$$\text{Producer} \rightarrow \text{FIFO Buffer} \rightarrow \text{Consumer}$$

This ensures smooth data flow and prevents overflow or data loss.

4.2.4 FIFO Architecture

The synchronous FIFO consists of the following components:

- Memory array
- Write pointer
- Read pointer
- Full detection logic
- Empty detection logic

Memory Array

The memory array stores incoming data values.

$$mem[0 : DEPTH - 1]$$

Each memory location stores an 8-bit data value.

4.2.5 Design Parameters

- Data width: 8 bits
- FIFO depth: 8 locations
- Address width: 3 bits
- Pointer width: 4 bits (extra bit used for wrap detection)

4.2.6 Pointer Mechanism

Two pointers control FIFO operations:

- Write Pointer (wptr)
- Read Pointer (rptr)

The write pointer indicates the memory location where new data will be written. The read pointer indicates the location from which data will be read.

An additional most significant bit (MSB) is used to detect wrap-around conditions.

4.2.7 FIFO Working Flow

The FIFO operates in two phases:

Write Operation

- When **wr_en** is asserted and FIFO is not full
- Data is written into memory
- Write pointer increments

Read Operation

- When **rd_en** is asserted and FIFO is not empty
- Data is read from memory
- Read pointer increments

4.2.8 Empty Condition

The FIFO is empty when both pointers are equal.

$$empty = (wptr = rptr)$$

This indicates that no valid data is available for reading.

4.2.9 Full Condition

The FIFO is full when:

- Address bits of write and read pointers are equal
- Most significant bits are different

$$full = (wptr[ADDR{-}1:0] = rptr[ADDR{-}1:0]) \wedge (wptr[ADDR] \neq rptr[ADDR])$$

This indicates that the memory buffer has been completely filled.

4.2.10 Empty and Full Condition Explanation with Example

In a FIFO memory, it is essential to determine when the buffer is empty or full to prevent incorrect read or write operations.

Empty Condition Example

The FIFO is considered empty when the write pointer and read pointer are equal.

$$empty = (wptr = rptr)$$

This means that there is no valid data available in the FIFO for reading.

Example of Empty Condition:

Assume the FIFO depth is 8. The pointers contain an extra most significant bit (MSB) to detect wrap-around conditions.

Initial state after reset:

Write Pointer	Read Pointer	FIFO Status
0000	0000	EMPTY

Since both pointers are equal, the FIFO contains no data.

If one data value is written:

Operation	Pointer Value
Write Data	wptr = 0001
Read Pointer	rptr = 0000

Now the FIFO contains one data element.

After reading that data:

Write Pointer	0001
Read Pointer	0001

Since both pointers match again, the FIFO becomes empty.

Full Condition Example

The FIFO becomes full when the lower address bits of the write and read pointers are equal but their most significant bits are different.

$$full = (wptr[ADDR{-}1:0] = rptr[ADDR{-}1:0]) \wedge (wptr[ADDR] \neq rptr[ADDR])$$

This condition indicates that the write pointer has wrapped around and reached the read pointer position.

Example of Full Condition:

Assume FIFO depth = 8.

Initial state:

$$\text{wptr} = 0000$$
$$\text{rptr} = 0000$$

Write eight values sequentially:

Write Count	Write Pointer
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

At this point:

$$\text{wptr} = 1000$$
$$\text{rptr} = 0000$$

Checking the full condition:

- Address bits are equal
- MSB bits are different

Therefore, the FIFO is full.

Why an Extra MSB Bit is Required

If only address bits were used, both full and empty conditions would appear identical.

Example:

$$\text{wptr} = 000$$
$$\text{rptr} = 000$$

This could represent either:

- FIFO empty
- FIFO full

To resolve this ambiguity, designers add one extra pointer bit that toggles whenever the pointer wraps around the FIFO memory.

4.2.11 Pointer Wrap-Around Mechanism

The additional MSB bit helps detect whether the pointer has wrapped around the memory buffer.

Example:

State	Write Pointer	Read Pointer
Initial	0000	0000
After 8 Writes	1000	0000
After 8 Reads	1000	1000

This mechanism allows accurate detection of full and empty states.

4.2.12 How Hardware Designers Detect FIFO Full in ASIC/FPGA Designs

In ASIC and FPGA implementations, FIFO full detection must be reliable and efficient. Hardware designers typically use pointer comparison logic.

The full condition is detected using the following method:

1. Compare the lower address bits of write and read pointers.
2. Check if the most significant bits of the pointers are different.
3. If both conditions are satisfied, the FIFO is full.

This can be implemented using simple combinational logic gates.

Example Verilog logic:

```
assign full = (wptr[ADDR-1:0] == rptr[ADDR-1:0]) &&
              (wptr[ADDR] != rptr[ADDR]);
```

This logic ensures that:

- Write operations stop when the FIFO is full
- Overflow conditions are prevented

Similarly, the empty condition is implemented as:

```
assign empty = (wptr == rptr);
```

These conditions are commonly used in industry-standard FIFO architectures for FPGA and ASIC designs.

4.2.13 Timing Behavior

All FIFO operations occur on the rising edge of the clock.

- Write occurs on positive clock edge when wr_en is active
- Read occurs on positive clock edge when rd_en is active

This ensures synchronous operation and predictable timing.

4.2.14 Advantages of Synchronous FIFO

- Simple design compared to asynchronous FIFO
- No clock domain crossing issues
- Efficient for pipeline buffering
- Lower hardware complexity
- High throughput data transfer

4.2.15 Applications

Synchronous FIFOs are used in many digital systems including:

- Digital signal processing pipelines
- FPGA communication interfaces
- Network packet buffering
- CPU pipeline stages
- Hardware accelerators

4.2.16 Verilog Implementation

```
module sync_fifo (
    input  wire        clk,
    input  wire        rst,
    input  wire        wr_en,
    input  wire        rd_en,
    input  wire [7:0]  din,
    output reg  [7:0]  dout,
    output wire        full,
    output wire        empty
);
```

```
11
12 parameter DEPTH = 8;
13 parameter ADDR  = 3;
14
15 reg [7:0] mem [0:DEPTH-1];
16
17 reg [ADDR:0] wptr;
18 reg [ADDR:0] rptr;
19
20 always @(posedge clk or posedge rst) begin
21     if (rst)
22         wptr <= 0;
23     else if (wr_en && !full) begin
24         mem[wptr[ADDR-1:0]] <= din;
25         wptr <= wptr + 1;
26     end
27 end
28
29 always @(posedge clk or posedge rst) begin
30     if (rst) begin
31         rptr <= 0;
32         dout <= 0;
33     end
34     else if (rd_en && !empty) begin
35         dout <= mem[rptr[ADDR-1:0]];
36         rptr <= rptr + 1;
37     end
38 end
39
40 assign empty = (wptr == rptr);
41
42 assign full  = (wptr[ADDR-1:0] == rptr[ADDR-1:0]) &&
43                (wptr[ADDR] != rptr[ADDR]);
44
45 endmodule
```

Listing 4.1: Synchronous FIFO Verilog Code

4.2.17 Synchronous FIFO Block Diagram

The synchronous FIFO architecture consists of a memory array, read pointer, write pointer, and control logic for full and empty detection. The write pointer indicates the next location to store data, while the read pointer indicates the next location from which data will be read.

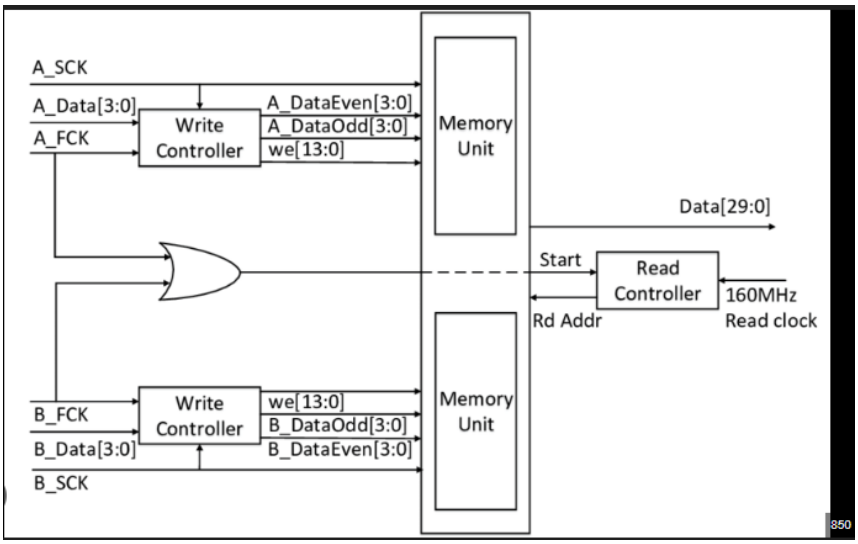

Figure 4.8: Block Diagram of Synchronous FIFO

The main components of the FIFO are:

- **Memory Array**: Stores incoming data elements.
- **Write Pointer**: Points to the memory location where the next data will be written.
- **Read Pointer**: Points to the location from which data will be read.
- **Control Logic**: Determines the FIFO status such as full or empty.

4.2.18 IP Module Implementation

In FPGA-based designs, FIFO functionality is often implemented using built-in IP cores provided by FPGA vendors such as Xilinx or Intel. These IP cores provide optimized implementations using dedicated memory resources such as Block RAM (BRAM).

Typical FIFO IP features include:

- Configurable data width
- Configurable FIFO depth
- Full and empty status flags
- Almost full and almost empty signals
- Built-in error detection

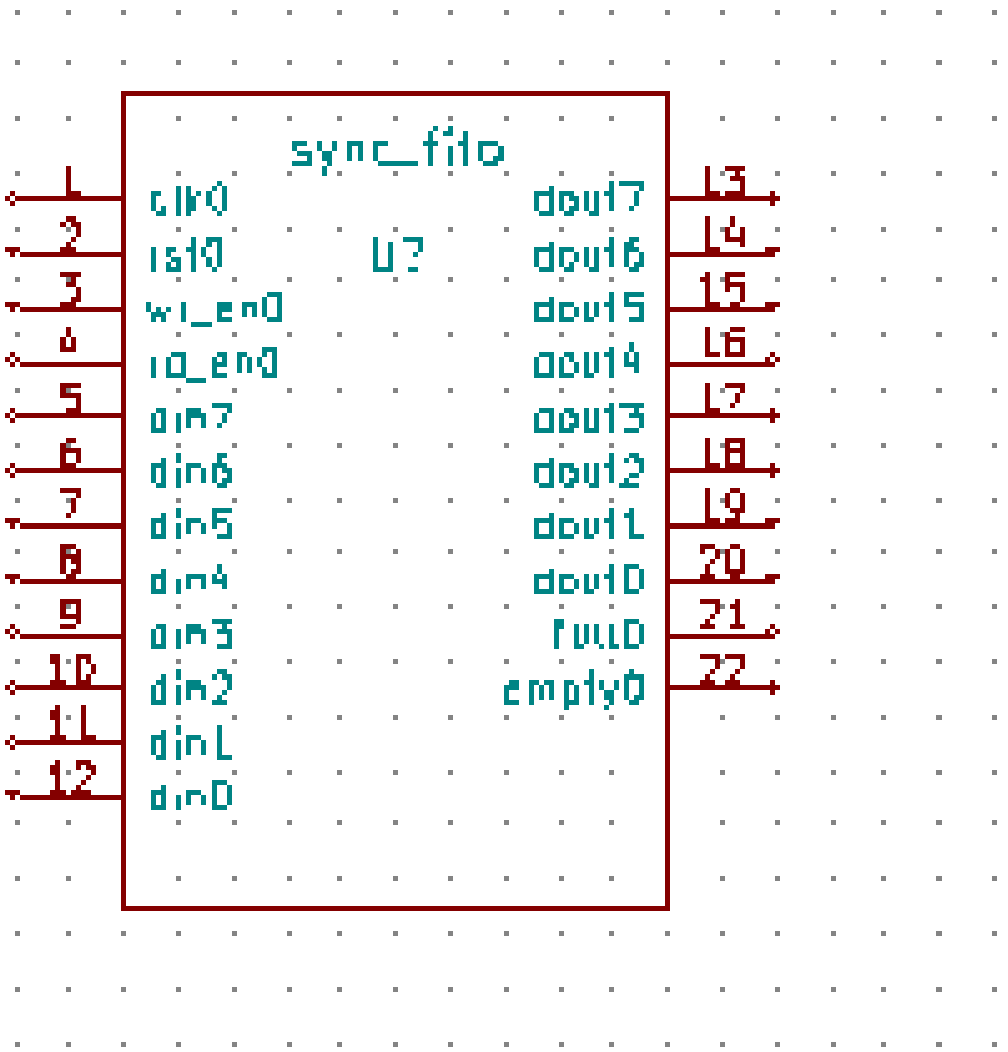

Figure 4.9: eSim IP Module of Synchronous FIFO

The IP core automatically generates the required RTL and optimized hardware implementation.

4.2.19 eSim Schematic Implementation

The synchronous FIFO was also simulated using the eSim environment. In this setup, the Verilog module was integrated with analog stimulus sources and digital bridges.

The schematic includes:

- Clock source (pulse generator)
- Reset signal
- Write enable signal
- Read enable signal

- Data input bus
- DAC bridges for observing output signals

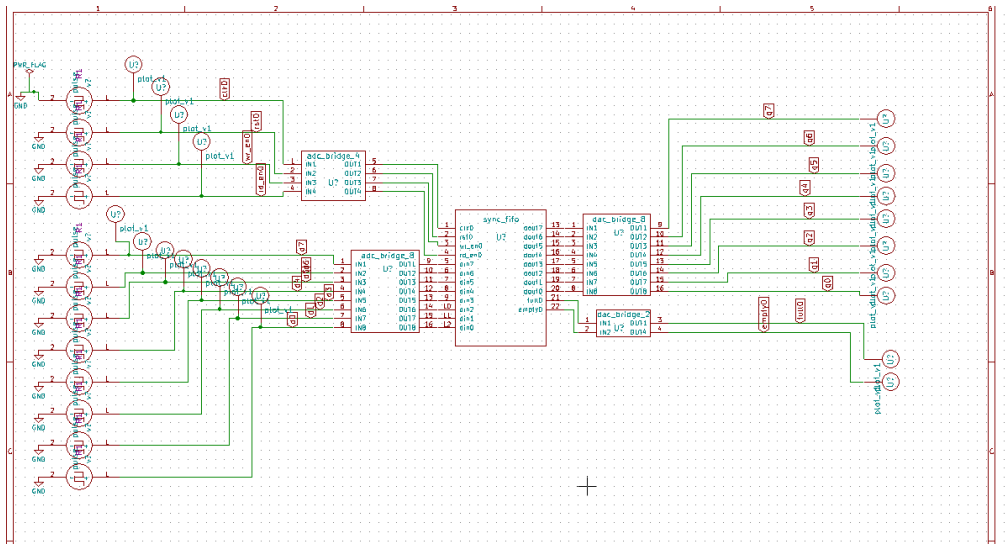

Figure 4.10: eSim Schematic of Synchronous FIFO

The eSim environment allows mixed-signal simulation by combining Verilog digital modules with analog sources and waveform visualization through Ngspice.

4.2.20 Simulation Results

The synchronous FIFO design was verified through simulation. The waveform confirms the correct behavior of write and read operations.

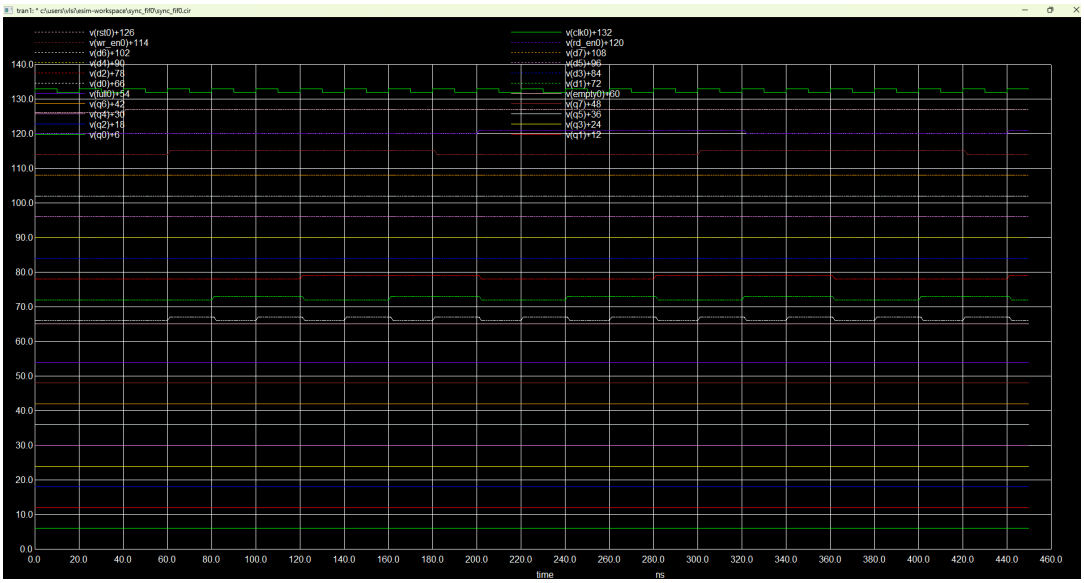

Figure 4.11: Simulation Waveform on eSim Showing FIFO Write and Read Operations

The simulation demonstrates the following behavior:

- Data is written sequentially when the write enable signal is asserted.
- The FIFO correctly stores incoming data in memory.
- Data is read in the same order in which it was written.
- The **full** signal becomes active when the FIFO memory is completely filled.
- The **empty** signal becomes active when all stored data has been read.

4.2.21 Observation from Simulation

From the waveform analysis:

- FIFO correctly maintains the First-In First-Out ordering.
- No overflow occurs when the FIFO becomes full.
- No underflow occurs when the FIFO becomes empty.
- The read and write pointers operate synchronously with the clock.

These results confirm the correct functional behavior of the synchronous FIFO design.

4.2.22 Conclusion

A synchronous FIFO was successfully designed and verified. The design efficiently manages data buffering between modules operating within the same clock domain. The pointer-based mechanism ensures correct data ordering while detecting full and empty states reliably. Such FIFO structures are essential building blocks in modern digital systems.

4.3 Frequency Multiplier Design and Implementation

4.3.1 Introduction

A frequency multiplier is a digital circuit used to generate an output signal with a frequency that is a multiple of the input signal frequency. Frequency multipliers are widely used in digital communication systems, clock generation circuits, and high-speed digital processors.

In this project, a simple frequency doubler is implemented using an XOR gate and a flip-flop. The XOR gate detects transitions of the input clock signal and produces pulses at both rising and falling edges. These pulses toggle a flip-flop, producing an output clock signal with twice the input frequency.

4.3.2 Why Frequency Multipliers are Used

Frequency multipliers are important in digital systems for several reasons:

- Generating higher frequency clocks from a stable reference clock
- Synchronizing different subsystems in digital processors
- Clock generation in communication systems
- Timing control in digital hardware
- High-speed interface support

4.3.3 Block Diagram

The block diagram of the frequency multiplier architecture is shown in Figure 4.29.

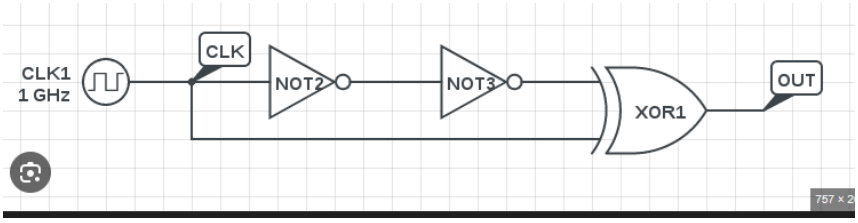

Figure 4.12: Circuit Diagram of XOR + Flip-Flop Frequency Multiplier

The main blocks include:

- Delay element (NOT gate)
- XOR gate for edge detection
- Toggle flip-flop for generating the doubled clock

4.3.4 Working Principle

The input clock is applied directly to the XOR gate. A delayed version of the clock is created using an inverter. The XOR gate compares these two signals and generates pulses whenever the signals differ.

Since the signals differ during both the rising and falling edges of the clock, the XOR gate produces pulses at twice the rate of the input clock.

These pulses are then used to toggle a flip-flop which produces the final output clock signal with double the input frequency.

4.3.5 IP Module

The implemented Verilog module used for the frequency multiplier is shown in Figure 4.13.

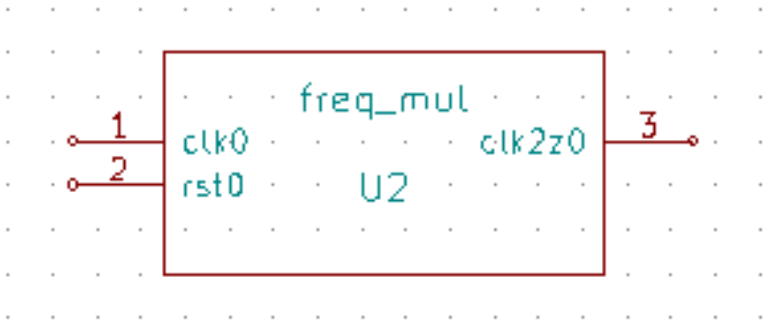

Figure 4.13: IP Module Representation of Frequency Multiplier

4.3.6 eSim Schematic

The implementation of the frequency multiplier using digital components in eSim is shown in Figure 4.30.

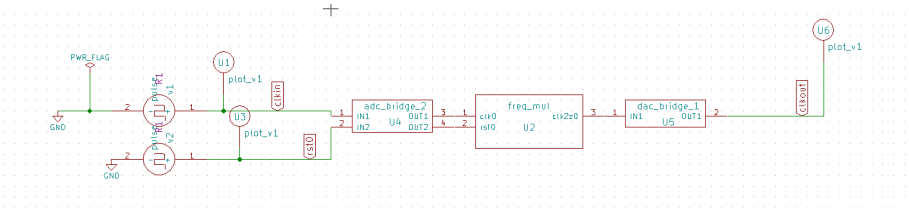

Figure 4.14: eSim Schematic of Frequency Multiplier

The schematic consists of:

- Digital clock generator
- NOT gate acting as delay element
- XOR gate acting as edge detector

- Flip-flop for toggling
- Output probe for waveform visualization

4.3.7 Verilog Implementation

The Verilog implementation of the frequency doubler is shown below.

```
module freq_mul_x2_xor_ff (
    input  wire clk,
    input  wire rst,
    output reg  clk2x
);

    wire clk_d;
    wire edge_pulse;

    not #1 U1 (clk_d, clk);
    xor U2 (edge_pulse, clk, clk_d);

    always @(posedge edge_pulse or posedge rst) begin
        if (rst)
            clk2x <= 0;
        else
            clk2x <= ~clk2x;
    end

endmodule
```

4.3.8 Simulation Waveform

The simulation waveform of the design is shown in Figure 4.15.

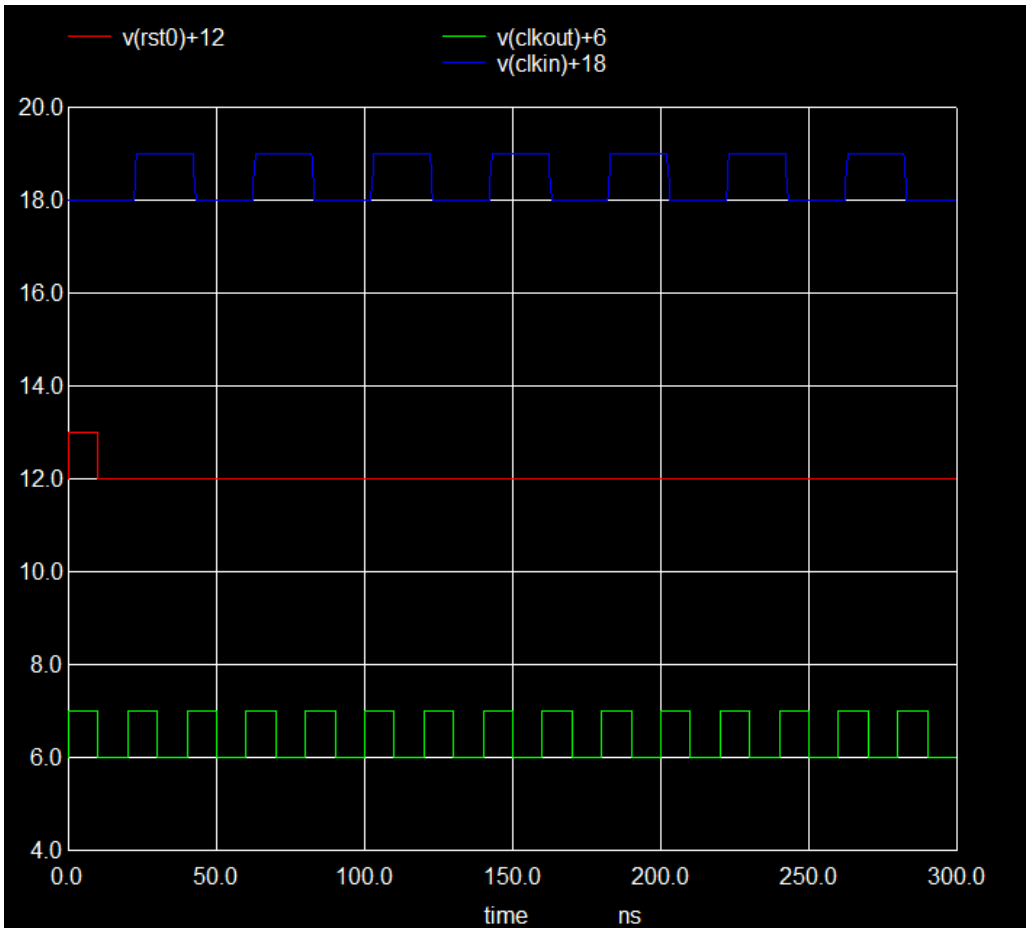

Figure 4.15: Simulation Waveform on eSim Showing Frequency Doubling

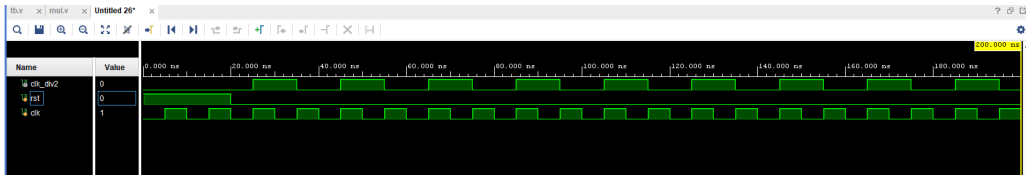

Figure 4.16: Simulation Waveform on Vivado Showing Frequency Doubling

From the waveform:

- Input clock period = 20 ns
- Input frequency = 50 MHz
- Output clock period = 10 ns
- Output frequency = 100 MHz

This verifies that the circuit successfully doubles the input frequency.

4.3.9 Advantages

The proposed design has several advantages:

- Simple digital logic implementation
- Low hardware complexity
- Fast edge detection
- Easy integration with digital systems
- Suitable for simulation and educational purposes

4.3.10 Applications

Frequency multipliers are used in many digital systems including:

- Clock generation circuits
- Digital signal processing systems
- Communication systems
- Microprocessor clock management
- FPGA and ASIC designs

4.3.11 Conclusion

A digital frequency multiplier based on XOR logic and a toggle flip-flop was successfully implemented and verified through simulation. The circuit generates pulses at both edges of the input clock and toggles a flip-flop to produce an output clock with twice the frequency of the input clock. The design demonstrates an efficient and simple approach for frequency multiplication in digital systems.

4.4 Transposed Fir Filter Design and Implementation

4.4.1 Introduction

A Finite Impulse Response (FIR) filter is a type of digital filter whose output depends only on the current and past input samples. Unlike IIR filters, FIR filters do not have feedback paths which makes them inherently stable.

The general FIR equation is:

$$y[n] = \sum_{k=0}^{N-1} b_k x[n-k]$$

where

- $x[n]$ = input signal
- $y[n]$ = output signal
- b_k = filter coefficients
- N = number of filter taps

FIR filters are widely used in digital signal processing applications such as communication systems, audio filtering, biomedical signal processing and image processing.

4.4.2 Why FIR Filters are Used

FIR filters are widely used because of the following advantages:

- Guaranteed stability
- Linear phase response
- Easy implementation in hardware
- No feedback loops
- Highly suitable for FPGA and ASIC designs

4.4.3

sectionIdeal Filter Response

The design of a FIR filter usually begins with an **ideal frequency response**. For example, an ideal low-pass filter has the following frequency response:

- Passes frequencies below the cutoff frequency

- Rejects frequencies above the cutoff frequency

The impulse response of an ideal low-pass filter is given by the sinc function:

$$h[n] = \frac{\sin(\omega_c n)}{\pi n}$$

where ω_c is the cutoff frequency.

However, the sinc function extends infinitely in both directions. Since hardware implementations require a finite number of coefficients, the impulse response must be truncated.

4.4.4 FIR Filter Architecture

The implemented FIR filter uses a **Direct Form (Tapped Delay Line)** architecture.

The architecture consists of:

- Delay elements (shift registers)
- Multipliers
- Adders
- Filter coefficients

Each input sample passes through delay elements and is multiplied by filter coefficients. The resulting products are added together to produce the filtered output.

4.4.5

sectionWindowing Technique

Truncating the ideal impulse response directly leads to ripples in the frequency response, known as the **Gibbs phenomenon**.

To reduce these ripples, window functions are used. The window function smoothly truncates the infinite impulse response.

The final FIR coefficients are obtained as:

$$b[n] = h_{ideal}[n] \times w[n]$$

where $w[n]$ is the window function.

Common window functions include:

- Rectangular Window
- Hamming Window
- Hanning Window

- Blackman Window
- Kaiser Window

For example, the Hamming window is defined as:

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$$

Windowing reduces spectral leakage and improves stopband attenuation.

4.4.6 Linear Phase Property

One of the most important advantages of FIR filters is their ability to achieve a **linear phase response**.

Linear phase means that all frequency components of the signal experience the same delay, preserving the waveform shape of the signal.

This occurs when the filter coefficients are symmetric:

$$b_k = b_{N-k}$$

The coefficients used in this design are symmetric, which ensures linear phase behavior.

4.4.7

sectionHardware Implementation Considerations

FIR filters are well suited for FPGA and ASIC implementations due to their parallel architecture.

Important implementation features include:

- Pipelined multiplier-add structures
- Efficient mapping to FPGA DSP slices
- Parallel processing of filter taps
- Possibility of coefficient symmetry optimization

Because FIR filters do not contain feedback loops, they are easier to pipeline and achieve higher clock frequencies in hardware implementations.

4.4.8 FIR Filter Block Diagram

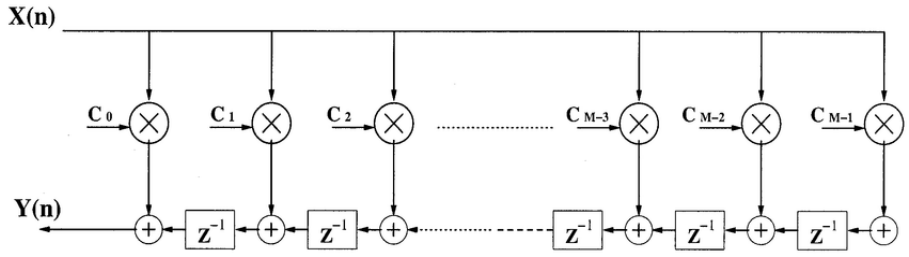

Figure 4.17: FIR Filter Block Diagram

The block diagram consists of delay elements z^{-1}, multipliers and adders which compute the weighted sum of current and past input samples.

4.4.9

sectionWorking Flow

The FIR filter operates as follows:

1. Input sample enters the filter.
2. Previous samples shift through the delay registers.
3. Each delayed sample is multiplied by its corresponding coefficient.
4. All multiplier outputs are summed together.
5. The result becomes the filtered output.

4.4.10

sectionCoefficient Selection

The FIR filter uses symmetric coefficients:

$$1,\ 3,\ 6,\ 10,\ 15,\ 18,\ 18,\ 15,\ 10,\ 6,\ 3,\ 1$$

These coefficients implement a low-pass filtering behavior and help in removing high-frequency noise.

4.4.11

sectionVerilog Implementation

The FIR filter is implemented using Verilog HDL.

subsectionFIR Filter Verilog Code

```
module fir_filter (
    input clk,
    input rst,
    input signed [7:0] x,
    output signed [15:0] y
);

parameter signed [7:0] b0  = 8'd1;
parameter signed [7:0] b1  = 8'd3;
parameter signed [7:0] b2  = 8'd6;
parameter signed [7:0] b3  = 8'd10;
parameter signed [7:0] b4  = 8'd15;
parameter signed [7:0] b5  = 8'd18;
parameter signed [7:0] b6  = 8'd18;
parameter signed [7:0] b7  = 8'd15;
parameter signed [7:0] b8  = 8'd10;
parameter signed [7:0] b9  = 8'd6;
parameter signed [7:0] b10 = 8'd3;
parameter signed [7:0] b11 = 8'd1;

reg signed [7:0] x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11;

wire signed [15:0]
m0,m1,m2,m3,m4,m5,
m6,m7,m8,m9,m10,m11;

assign m0  = b0  * x;
assign m1  = b1  * x1;
assign m2  = b2  * x2;
assign m3  = b3  * x3;
assign m4  = b4  * x4;
assign m5  = b5  * x5;
assign m6  = b6  * x6;
assign m7  = b7  * x7;
assign m8  = b8  * x8;
assign m9  = b9  * x9;
assign m10 = b10 * x10;
assign m11 = b11 * x11;

assign y = m0+m1+m2+m3+m4+m5+
           m6+m7+m8+m9+m10+m11;

```

```
43 always @(posedge clk or posedge rst)
44 begin
45     if(rst)
46         {x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11} <= 0;
47     else begin
48         x1<=x;
49         x2<=x1;
50         x3<=x2;
51         x4<=x3;
52         x5<=x4;
53         x6<=x5;
54         x7<=x6;
55         x8<=x7;
56         x9<=x8;
57         x10<=x9;
58         x11<=x10;
59     end
60 end
61
62 endmodule
```

4.4.12 Fir Ip module

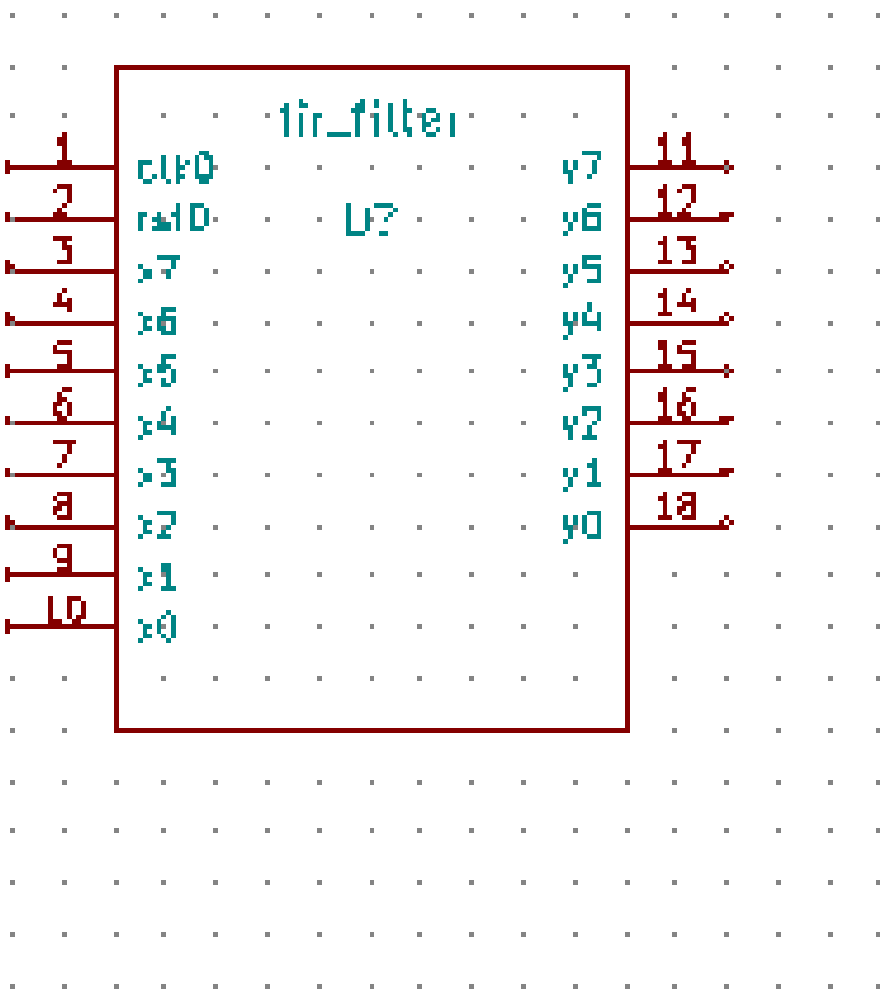

Figure 4.18: eSim Mixed Signal FIR Filter Schematic

4.4.13 eSim Schematic

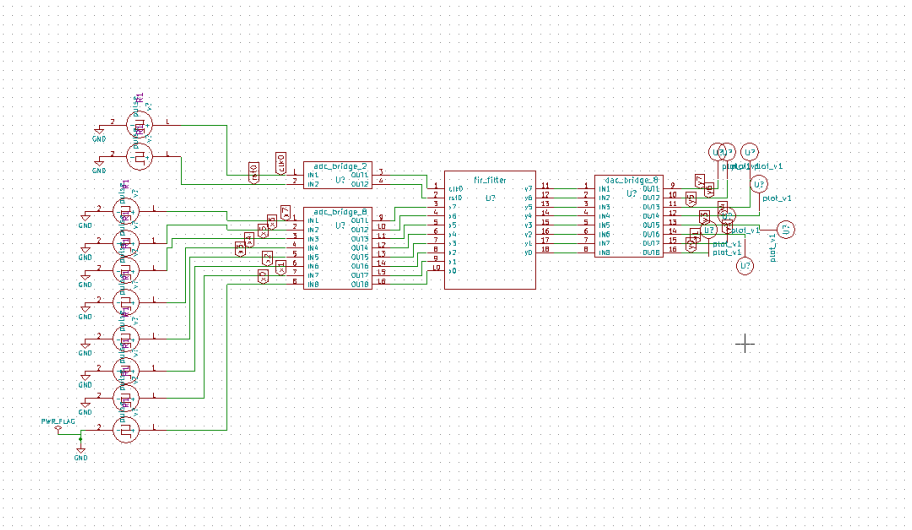

Figure 4.19: eSim Mixed Signal FIR Filter Schematic

The schematic integrates the Verilog FIR filter IP with analog sources using ADC and DAC bridges for mixed signal simulation.

4.4.14 Simulation Waveforms

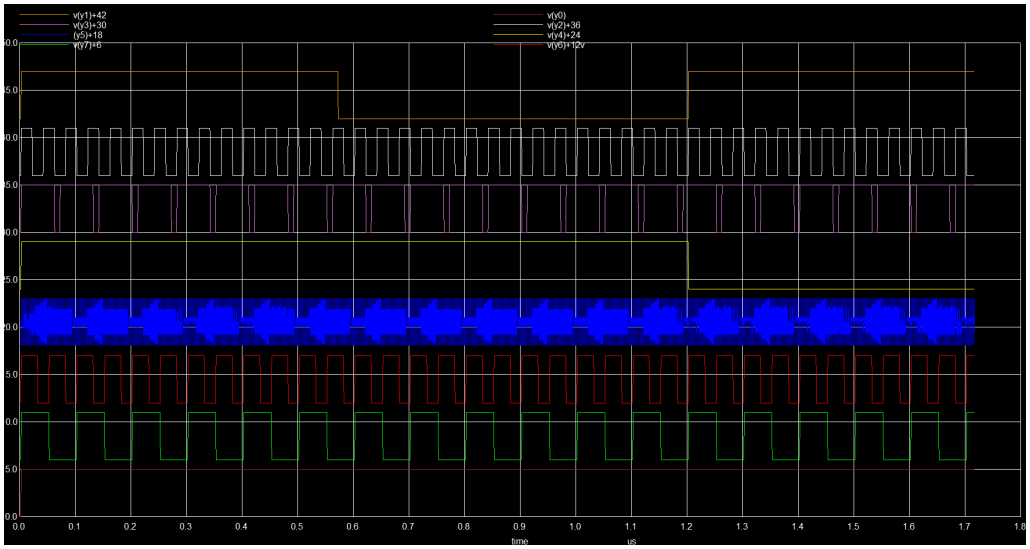

Figure 4.20: FIR Filter Simulation Waveforms

The waveform shows clock signal, input samples and filtered output bits generated by the FIR filter.

4.4.15 Timing Explanation

At every clock cycle:

- Input sample enters the delay register
- Previous samples move through delay stages
- Multipliers compute weighted products
- Adders generate the final output

The delay line introduces latency equal to the filter order.

4.4.16 Advantages

- Always stable
- Linear phase response
- Easy FPGA implementation
- Parallel architecture
- High numerical accuracy

4.4.17 Applications

FIR filters are used in:

- Audio signal processing
- ECG filtering
- Communication systems
- Radar signal processing
- Image processing

4.4.18 Conclusion

A 12-tap FIR filter was successfully designed and implemented using Verilog HDL. The design was simulated using eSim mixed signal simulation environment. The results demonstrate correct filtering behavior and verify the functionality of the FIR architecture.

4.5 Booth-Wallace Multiplier Design and Implementation

4.5.1 Introduction

A multiplier is one of the most important arithmetic blocks in digital systems, signal processing units, microprocessors, DSP architectures, and AI accelerators. Multiplication is used in convolution, filtering, matrix operations, MAC units, FFT, and many real-time applications. Since multiplication is a time-consuming operation compared to addition or subtraction, high-speed multiplier architectures are widely used in VLSI design.

In this work, a 4-bit signed Booth-Wallace multiplier is designed and implemented using Verilog HDL and simulated in eSim. The design combines Booth recoding for efficient partial product generation and Wallace tree reduction using Carry Save Adders (CSA) for faster addition of partial products.

4.5.2 Why Multipliers are Used

Multipliers are used in almost every digital computation system where repeated arithmetic operations are required. Some of the major applications are listed below:

- Digital Signal Processing (DSP)
- FIR and IIR filters
- FFT and DFT processors
- Multiply-Accumulate (MAC) units
- Microprocessors and ALUs
- Image and video processing
- AI/ML accelerators

A normal array multiplier is simple, but it is slower due to longer addition delay. To improve speed and reduce the number of partial products, Booth encoding and Wallace tree reduction are preferred.

4.5.3 Booth Multiplication Concept

Booth multiplication is an efficient signed multiplication technique. It examines the multiplier bits in pairs and decides whether to add, subtract, or ignore the multiplicand.

For radix-2 Booth encoding, the following rules are used:

Q_i	Q_{i-1}	Operation
0	0	No operation
0	1	Add multiplicand
1	0	Subtract multiplicand
1	1	No operation

Table 4.1: Radix-2 Booth Encoding Rules

Thus, Booth encoding reduces unnecessary partial products and handles signed numbers naturally.

4.5.4 Why Booth Encoding is Used

The main reasons for using Booth encoding are:

- Reduces the number of effective partial products
- Supports signed multiplication directly
- Improves multiplier efficiency
- Reduces switching activity and power consumption
- Suitable for high-speed arithmetic units

4.5.5 Wallace Tree Reduction

After Booth recoding generates the partial products, these products must be added efficiently. If they are added one by one using normal adders, the delay becomes large. Wallace tree reduction solves this problem by compressing multiple partial product rows in parallel.

The Wallace tree uses:

- Full Adders for 3:2 compression
- Half Adders where required
- Carry Save Adder stages to avoid immediate carry propagation

The CSA reduces three input rows into two output rows:

$$x + y + z = \text{sum} + \text{carry} \tag{4.1}$$

The carry is shifted by one bit to the left, and final addition is done only at the last stage. This significantly improves speed.

4.5.6 Carry Save Adder Principle

A Carry Save Adder does not propagate carry immediately across all bit positions. Instead, for each bit position:

$$\text{sum}_i = x_i \oplus y_i \oplus z_i \tag{4.2}$$

$$\text{carry}_i = x_i y_i + y_i z_i + x_i z_i \tag{4.3}$$

Thus, three numbers are reduced into two numbers, which are later added using a final carry propagate adder.

4.5.7 Architecture of Booth-Wallace Multiplier

The complete architecture consists of the following stages:

1. Booth encoder
2. Partial product generation
3. Wallace tree reduction using CSA stages
4. Final carry propagate adder

The flow is:

Booth Encoding → Partial Products → CSA Reduction → Final Adder → Product

4.5.8 Booth-Wallace Multiplier Block Diagram

Figure 4.21 shows the overall block diagram of the Booth-Wallace multiplier.

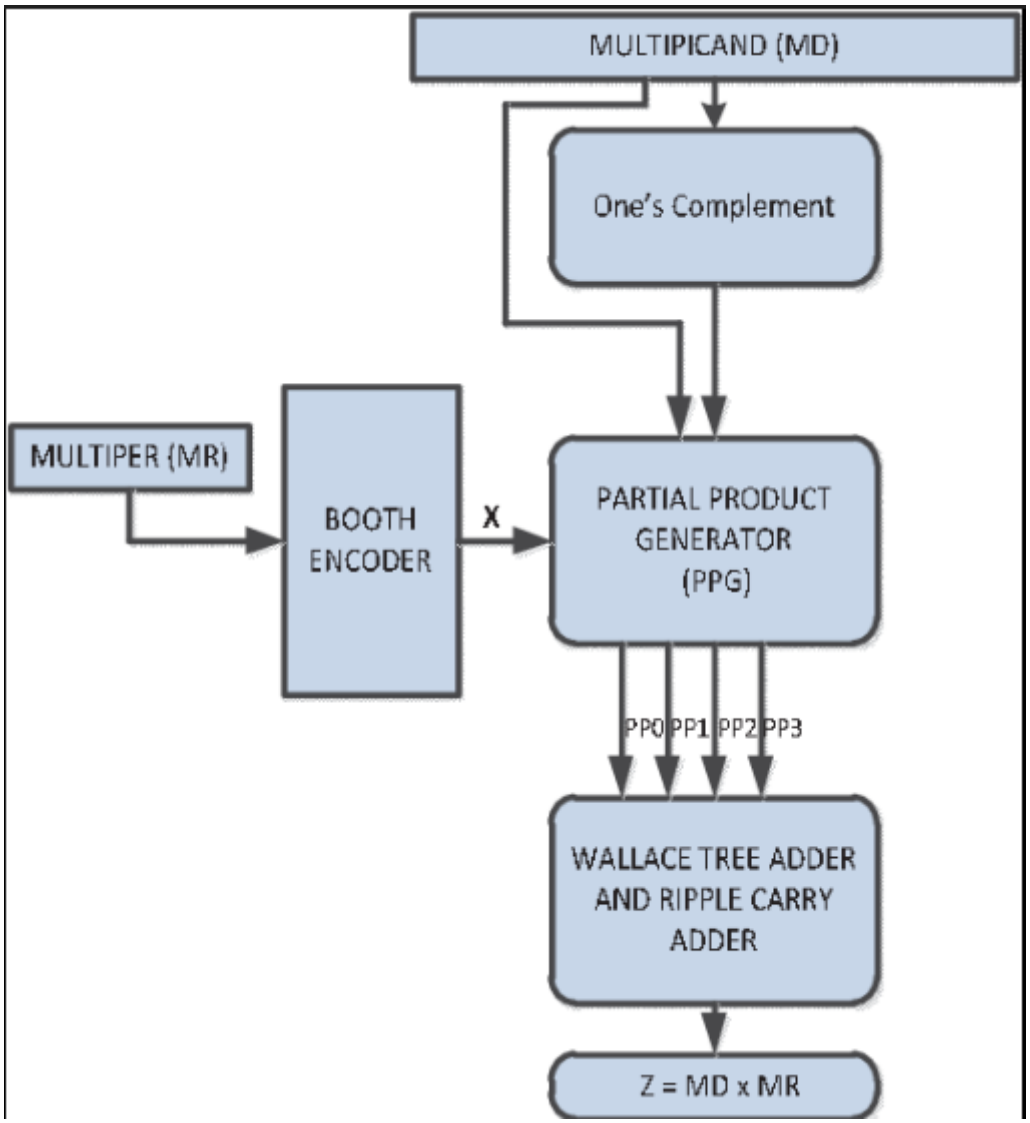

Figure 4.21: Block Diagram of 4-bit Booth-Wallace Multiplier

4.5.9 Working of the Multiplier

Let the multiplicand be A and multiplier be B. Booth encoding first checks pairs of multiplier bits and generates signed partial products. These partial products are shifted according to bit position. The Wallace tree then compresses them in stages using Carry Save Adders.

For example:

- Stage 1: PP_0, PP_1, and PP_2 are reduced to S_1 and C_1
- Stage 2: S_1, C_1, and PP_3 are reduced to S_2 and C_2
- Final stage: $S_2 + C_2$ gives the final product

4.5.10 Design Example

Consider:

$$A = 4 = 0100$$

$$B = -5 = 1011$$

Expected product:

$$P = 4 \times (-5) = -20$$

8-bit two's complement result:

$$P = 11101100$$

This result is verified in simulation.

4.5.11 IP Module

The RTL or IP module of the designed Booth-Wallace multiplier is shown in Figure 4.22. It consists of two 4-bit inputs and one 8-bit output.

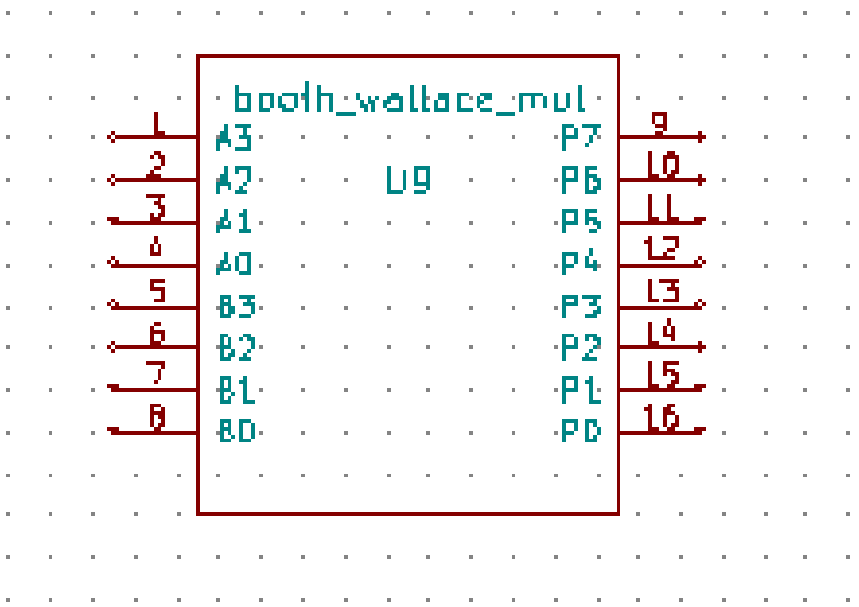

Figure 4.22: RTL/IP Module of Booth-Wallace Multiplier

4.5.12 Verilog Design Description

The Verilog code contains the following modules:

- `booth_wallace_4x4` : Top module
- `csa_8bit` : 8-bit Carry Save Adder

- `full_adder` : 1-bit full adder

The top module performs Booth recoding, generates partial products, reduces them using two CSA stages, and computes the final 8-bit product.

4.5.13 eSim Schematic

The eSim schematic of the Booth-Wallace multiplier is shown in Figure 4.23. The input bits are connected through ADC bridges and the output bits are observed through DAC bridges for waveform visualization and result verification.

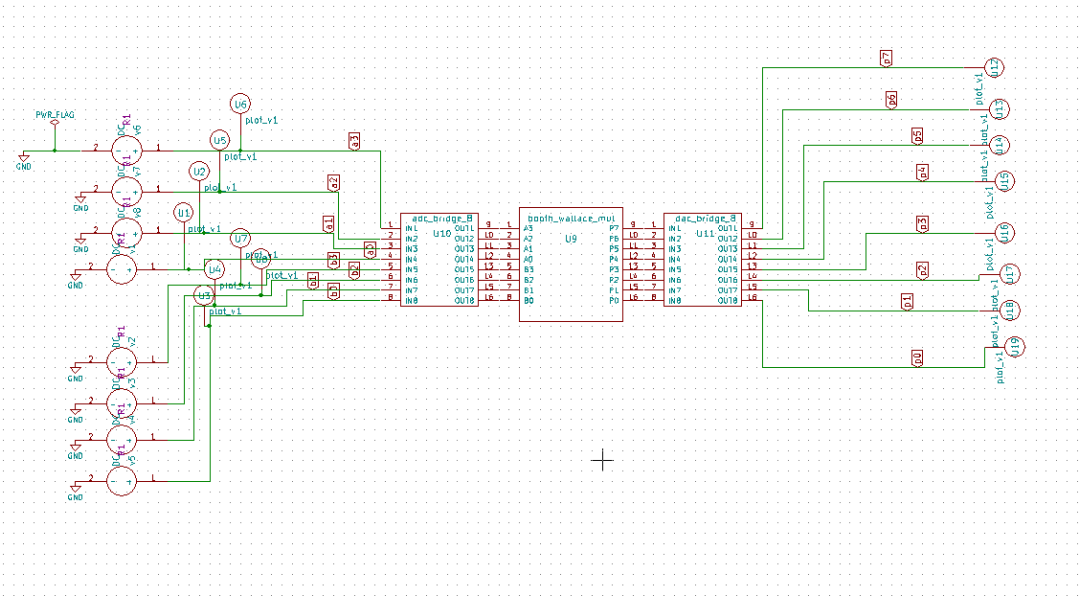

Figure 4.23: eSim Schematic of Booth-Wallace Multiplier

4.5.14 Simulation Waveforms

The simulation waveforms for the multiplier are shown in Figure 4.37. For the applied input combination $A = 4$ and $B = -5$, the output product is obtained as $P = -20$.

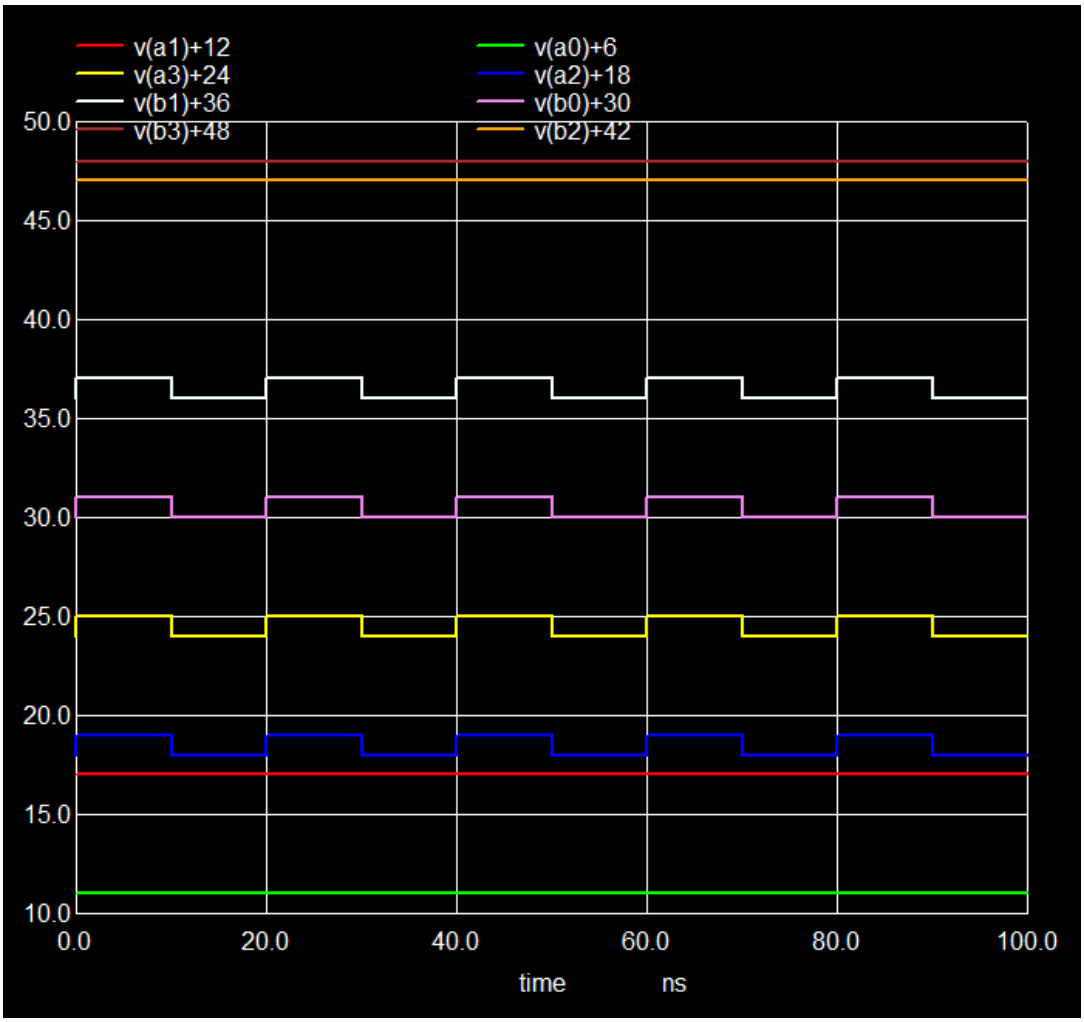

Figure 4.24: Input Simulation Waveform on Esim of Booth-Wallace Multiplier

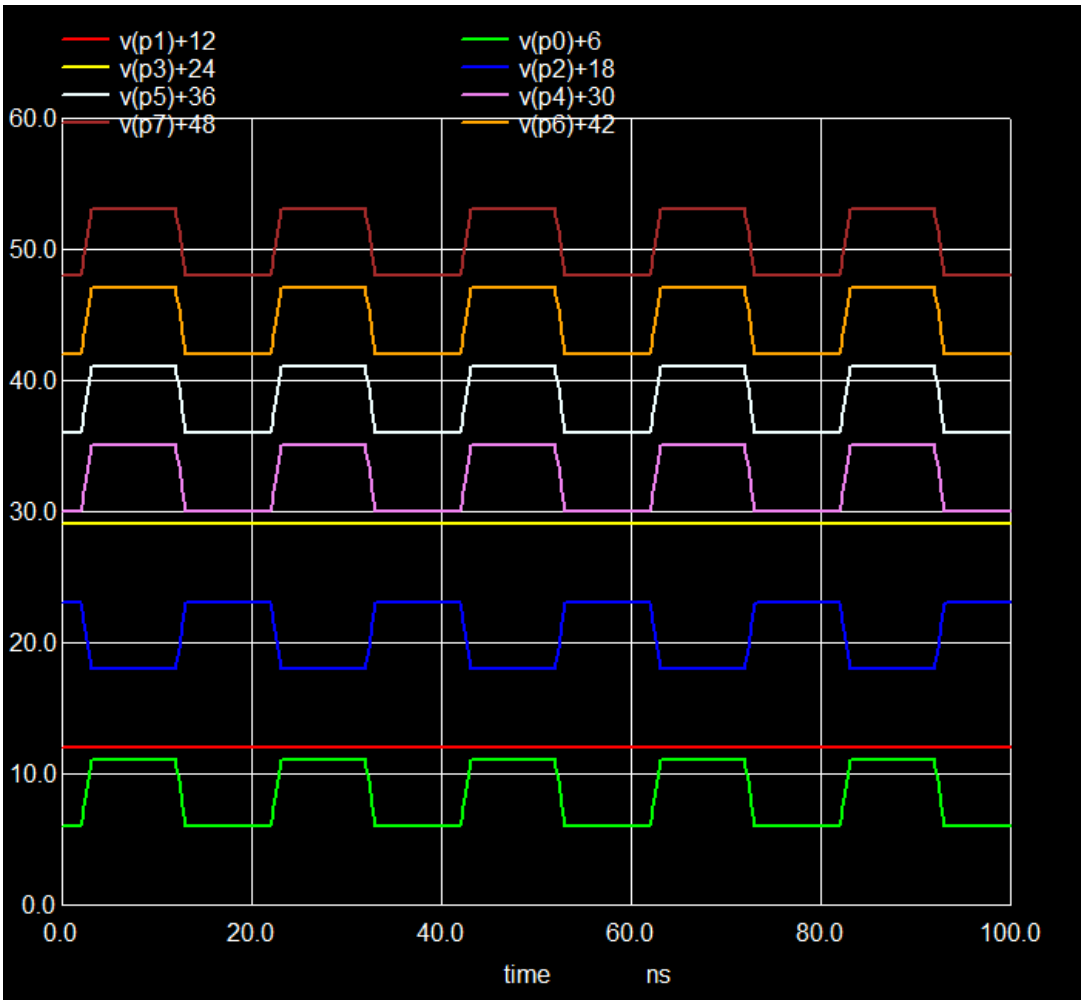

Figure 4.25: Output Simulation Waveform on Esim of Booth-Wallace Multiplier

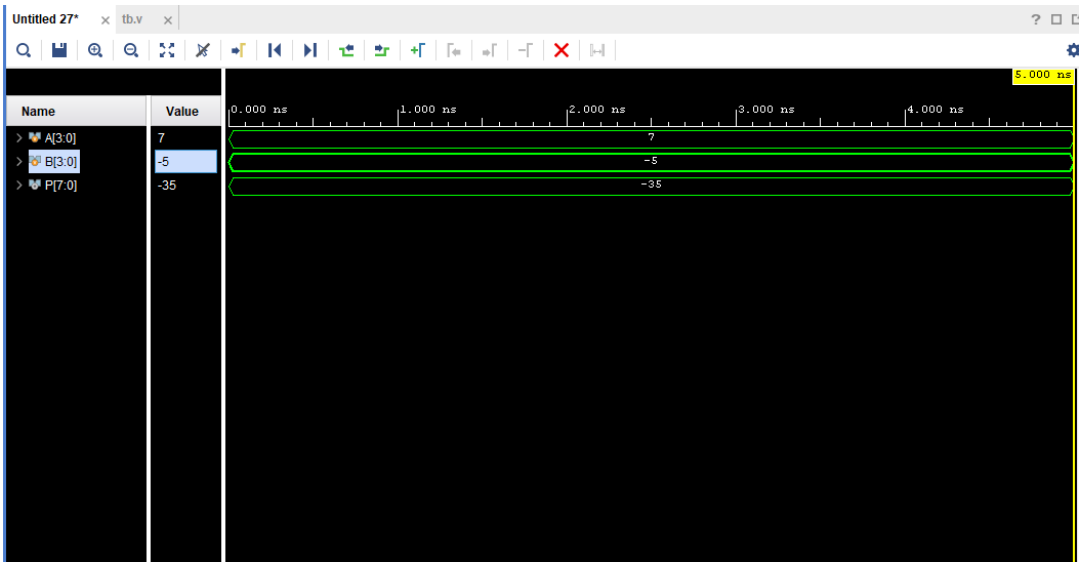

Figure 4.26: Simulation Waveform on Vivado of Booth-Wallace Multiplier

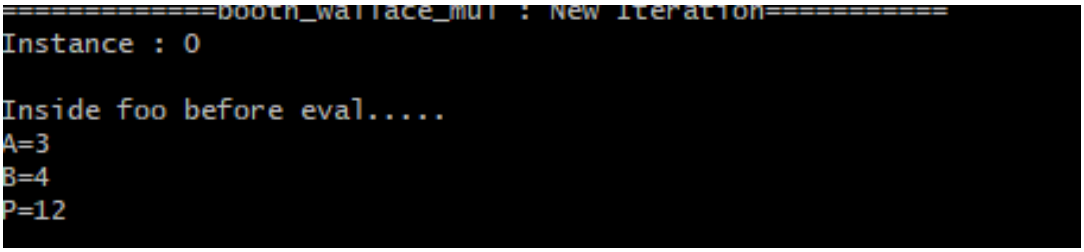

Figure 4.27: Console Simulation Output on Esim of Booth-Wallace Multiplier

4.5.15 Advantages

The Booth-Wallace multiplier offers the following advantages:

- High speed multiplication
- Reduced number of partial products
- Efficient signed multiplication
- Lower delay compared to array multipliers
- Good architecture for DSP and MAC units
- Suitable for ASIC and FPGA implementations

4.5.16 Applications

This multiplier can be used in:

- FIR filter implementation
- DSP processors
- MAC units
- ALU design
- Embedded processors
- Machine learning accelerators
- Image processing hardware

4.5.17 Conclusion

A 4-bit signed Booth-Wallace multiplier was successfully designed and implemented using Verilog HDL and simulated in eSim. Booth encoding was used for efficient signed partial product generation, while Wallace tree reduction with Carry Save Adders improved the speed of accumulation. The design was verified through simulation and produced correct signed multiplication results. This architecture is efficient, synthesizable, and suitable for high-speed arithmetic applications in VLSI and FPGA-based systems.

4.6 Design and Implementation of Divider Ip

4.6.1 Introduction

Division is an essential arithmetic operation used in digital processors, signal processing systems, and embedded hardware. Compared to addition or multiplication, division requires multiple steps and therefore efficient algorithms are required for hardware implementation.

In this work, an **8-bit divider** is implemented using the **restoring division algorithm** in Verilog HDL. The design is simulated and verified through waveform analysis.

4.6.2

sectionWhy Dividers are Used

Dividers are widely used in digital systems for:

- Arithmetic operations in processors and ALUs
- Digital Signal Processing (DSP)
- Frequency scaling and clock generation
- Scientific and engineering computations
- Hardware accelerators

Efficient divider architectures improve system performance and resource utilization in FPGA and ASIC designs.

4.6.3 Restoring Division Algorithm

The restoring division algorithm performs division using iterative shift and subtraction operations.

Algorithm steps:

1. Initialize remainder to zero.
2. Shift left the remainder and bring the next dividend bit.
3. Compare the remainder with the divisor.
4. If remainder $\geq$ divisor:
    - Subtract divisor from remainder
    - Set quotient bit to 1
5. Otherwise set quotient bit to 0.

6. Repeat the process for all bits of the dividend.

For an 8-bit divider, the operation requires **8 clock cycles**.

4.6.4 Block Diagram

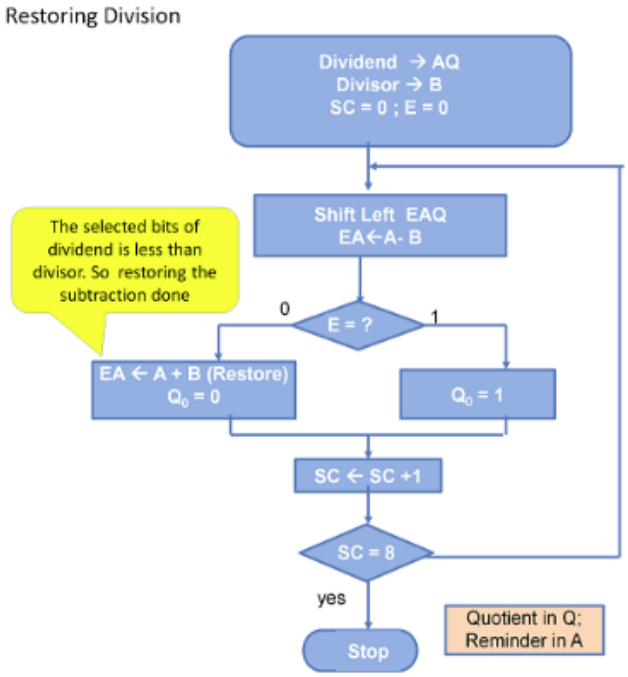

Figure 4.28: Block Diagram of Divider Architecture

The main components include:

- Shift registers
- Subtractor
- Comparator
- Control logic
- Remainder register

4.6.5 IP Module

The divider IP block interface is shown below:

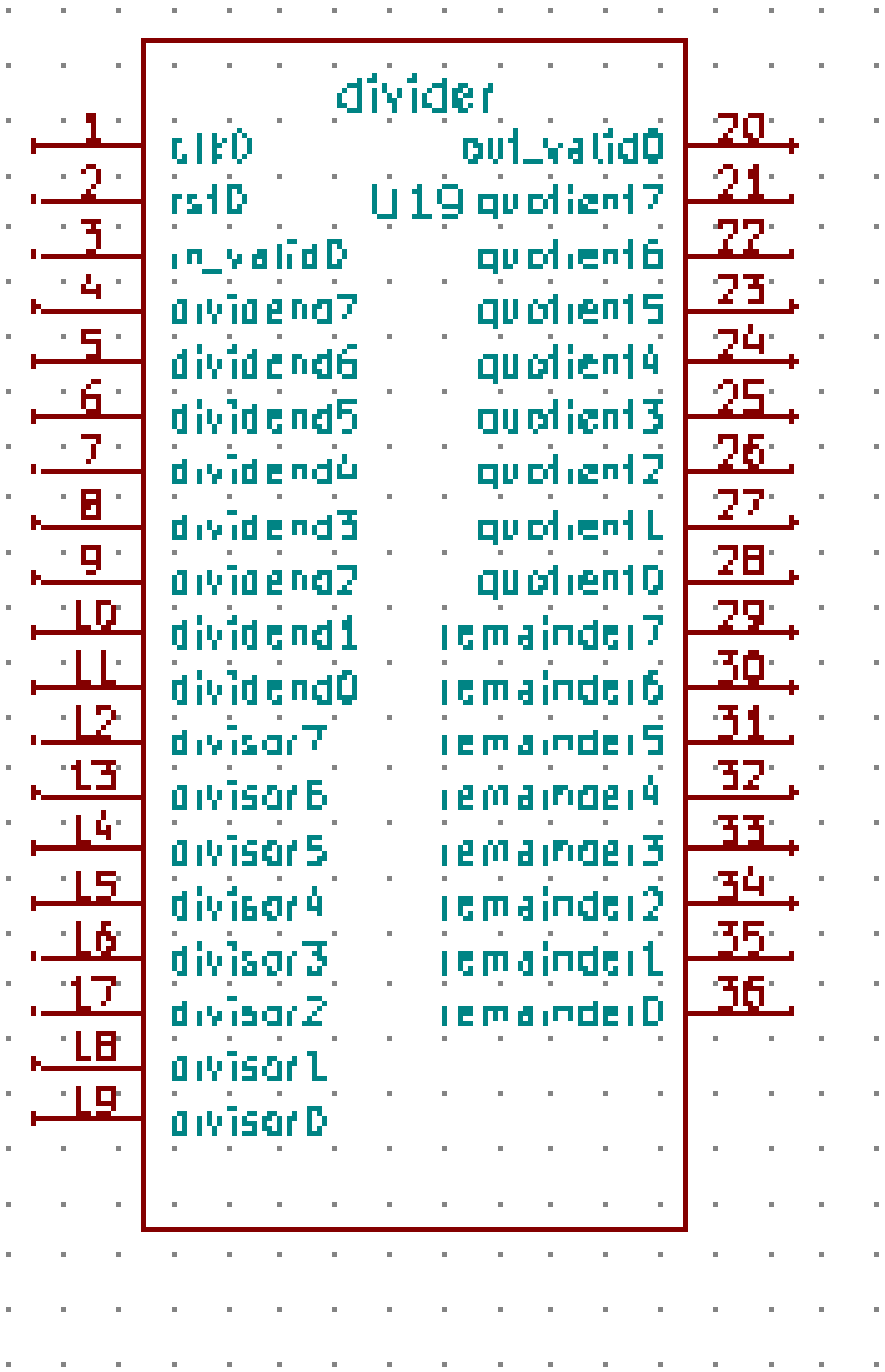

Figure 4.29: IP Module Diagram of Divider Architecture

4.6.6 Verilog Implementation

The following Verilog code implements the restoring division algorithm.

```
module divider_8bit (
    input  wire clk,
    input  wire rst,
    input  wire start,
    input  wire [7:0] dividend,
```

```
6       input  wire [7:0] divisor,
7       output reg  [7:0] quotient,
8       output reg  [7:0] remainder,
9       output reg  done
10  );
11
12      reg [7:0] divisor_reg;
13      reg [7:0] quotient_reg;
14      reg [8:0] remainder_reg;
15      reg [3:0] count;
16      reg busy;
17
18      reg [8:0] temp_rem;
19
20      always @(posedge clk or posedge rst) begin
21          if (rst) begin
22              quotient <= 0;
23              remainder <= 0;
24              divisor_reg <= 0;
25              quotient_reg <= 0;
26              remainder_reg <= 0;
27              count <= 0;
28              busy <= 0;
29              done <= 0;
30          end
31          else begin
32              done <= 0;
33
34              if (start && !busy) begin
35                  busy <= 1;
36                  count <= 8;
37                  divisor_reg <= divisor;
38                  quotient_reg <= dividend;
39                  remainder_reg <= 0;
40              end
41
42              else if (busy) begin
43
44                  temp_rem = {remainder_reg[7:0], quotient_reg
                        [7]};
45                  quotient_reg <= {quotient_reg[6:0], 1'b0};
46
47                  if (temp_rem >= divisor_reg) begin
48                      remainder_reg <= temp_rem - divisor_reg;
49                      quotient_reg[0] <= 1'b1;
```

```
                    end
                    else begin
                        remainder_reg <= temp_rem;
                    end

                    count <= count - 1;

                    if (count == 1) begin
                        busy <= 0;
                        quotient <= quotient_reg;
                        remainder <= remainder_reg[7:0];
                        done <= 1;
                    end
                end
            end
        end

endmodule
```

4.6.7 eSim Schematic

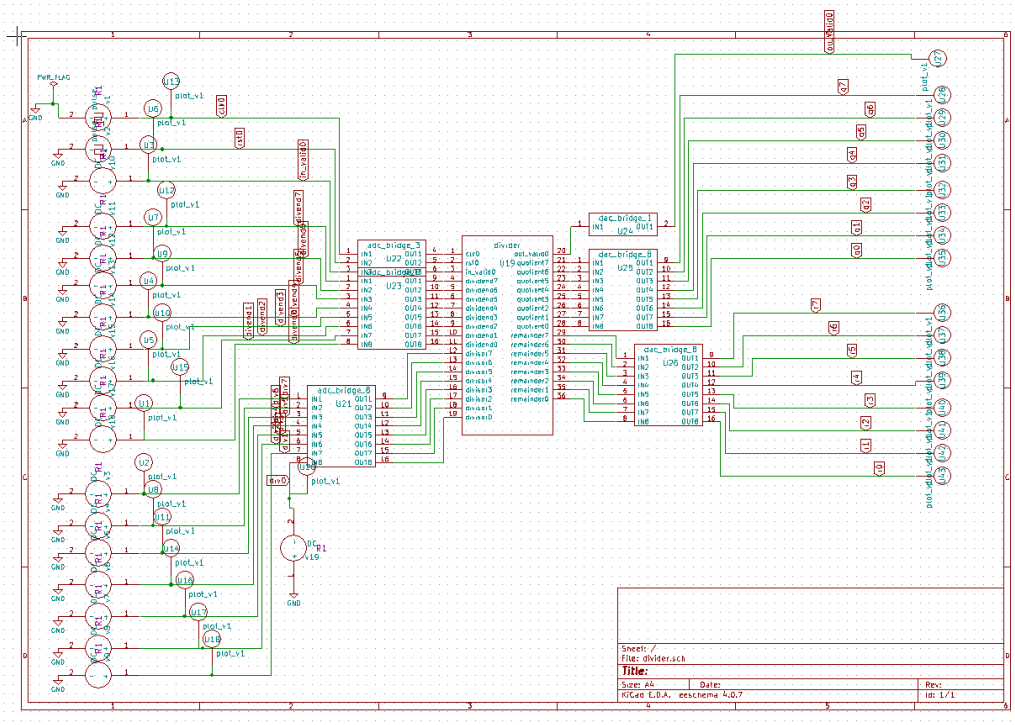

Figure 4.30: eSim Schematic of Divider

4.6.8 Simulation Waveforms

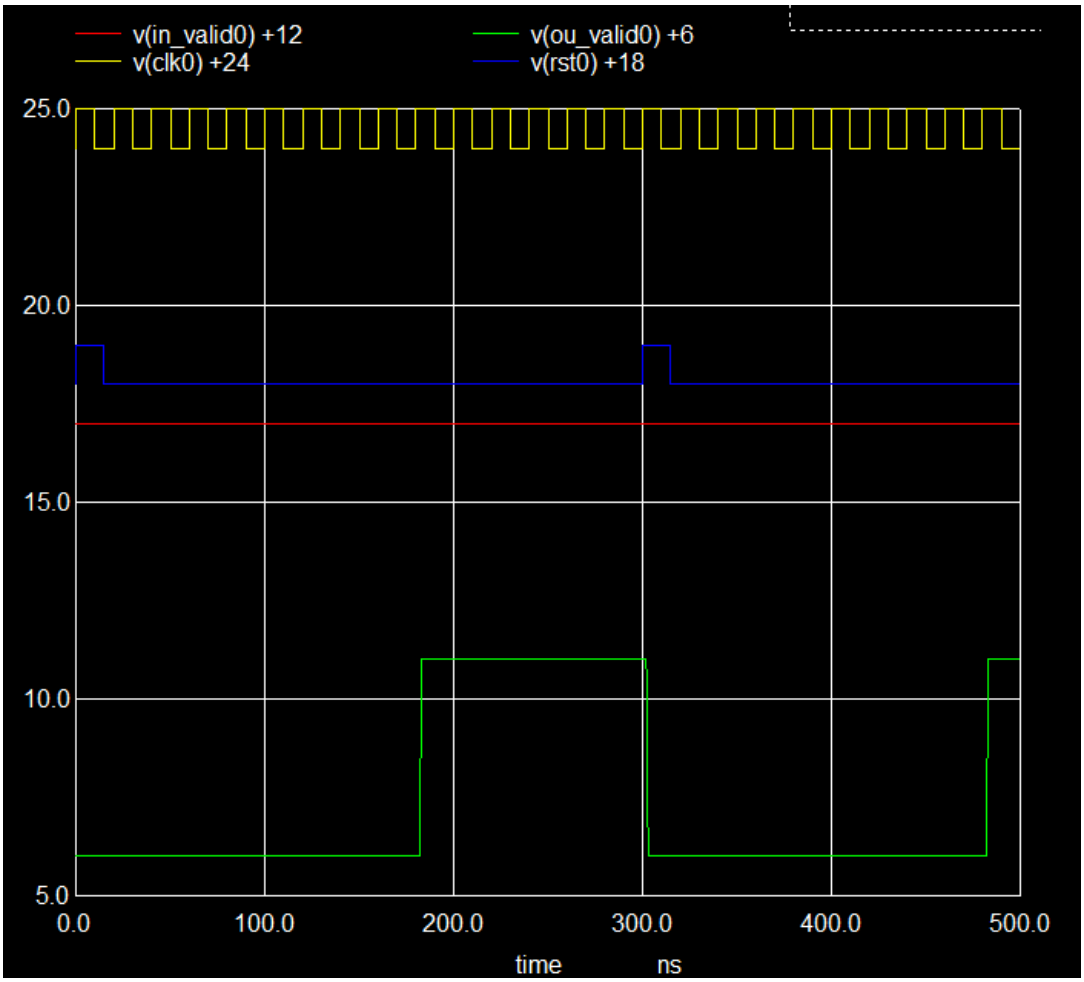

Figure 4.31: Simulation Waveforms on esim of Divider

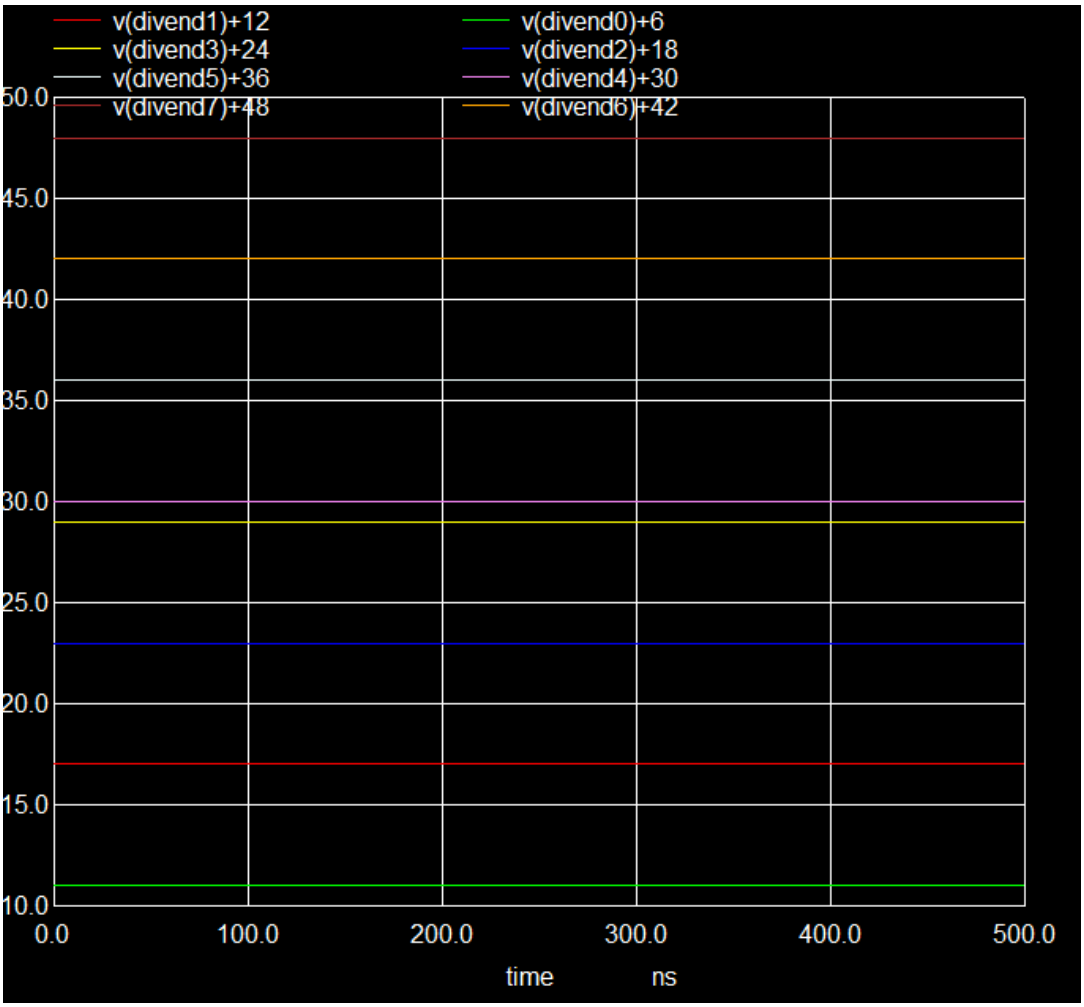

Figure 4.32: Dividend Simulation Waveforms on esim of Divider

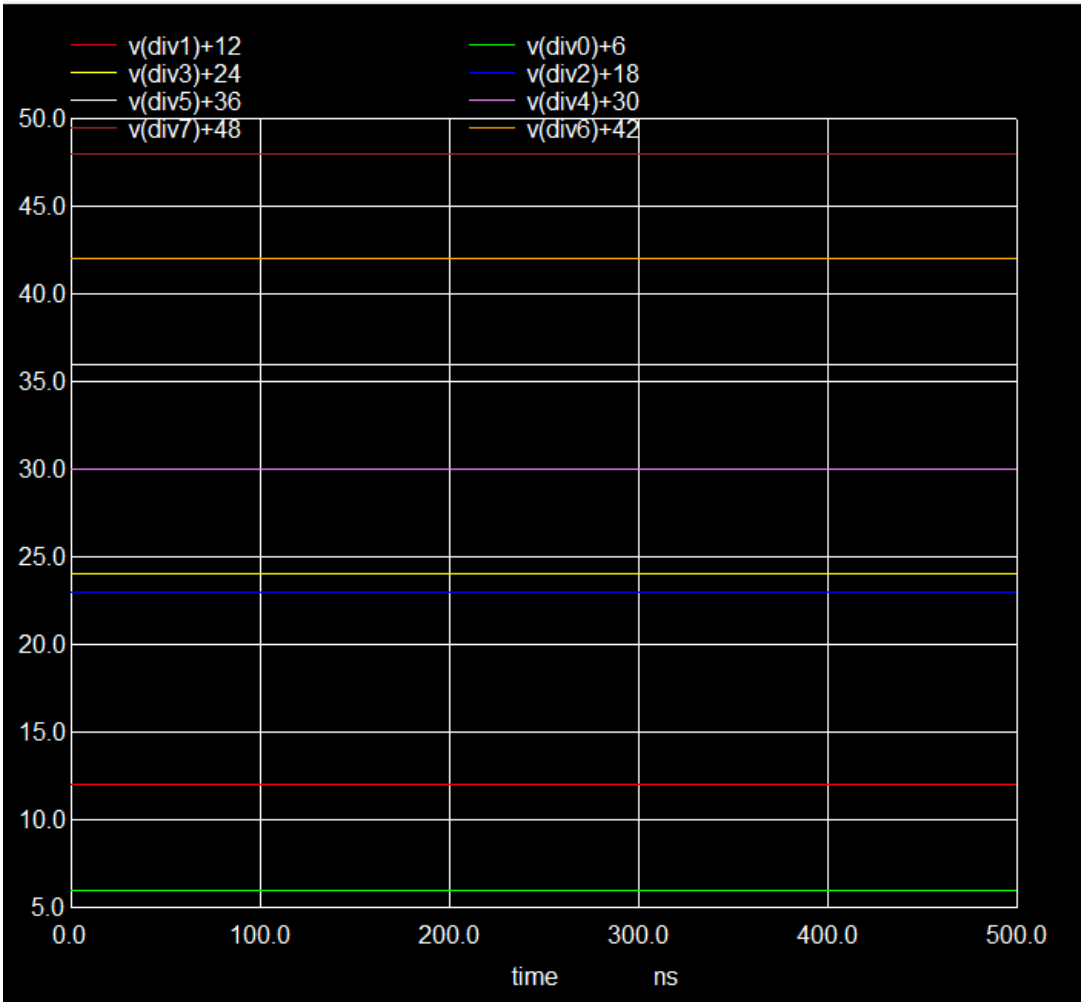

Figure 4.33: Divisor Simulation Waveforms on esim of Divider

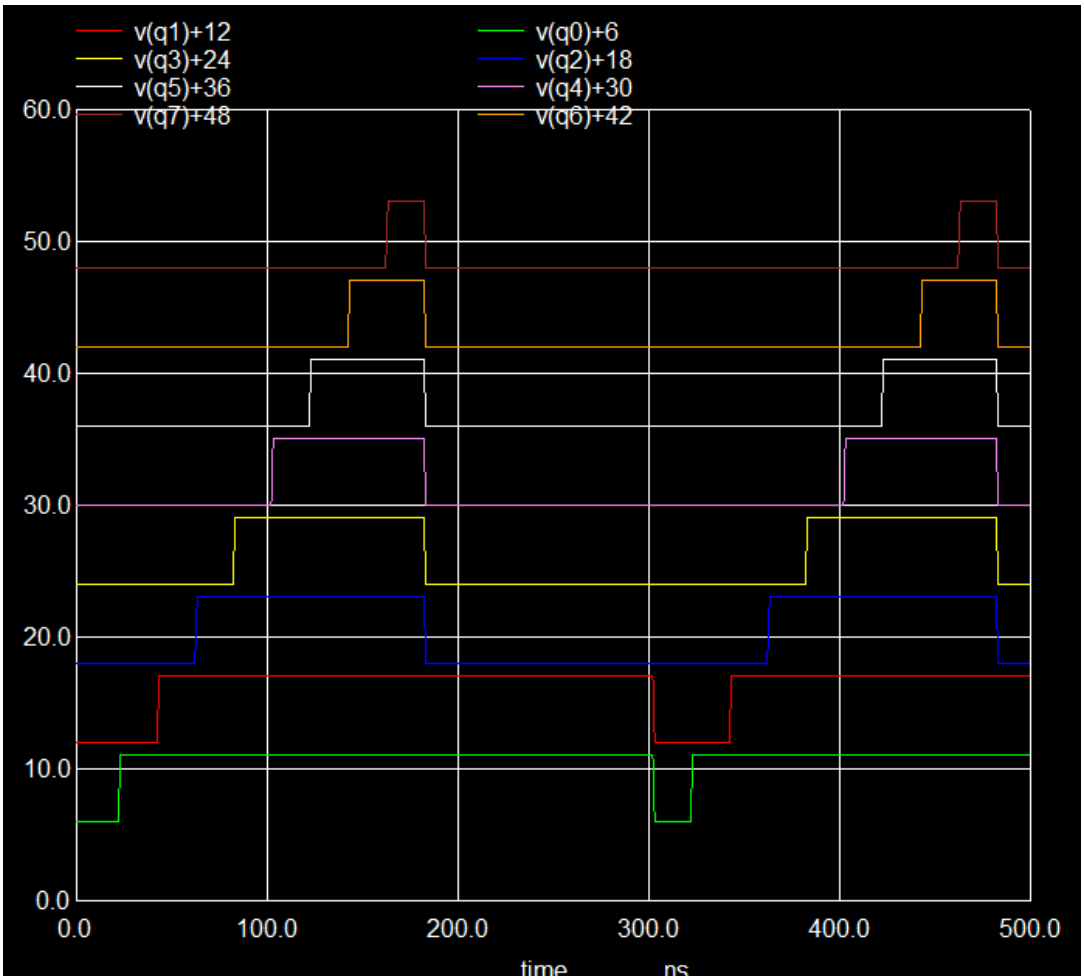

Figure 4.34: Quiotient Simulation Waveforms on esim of Divider

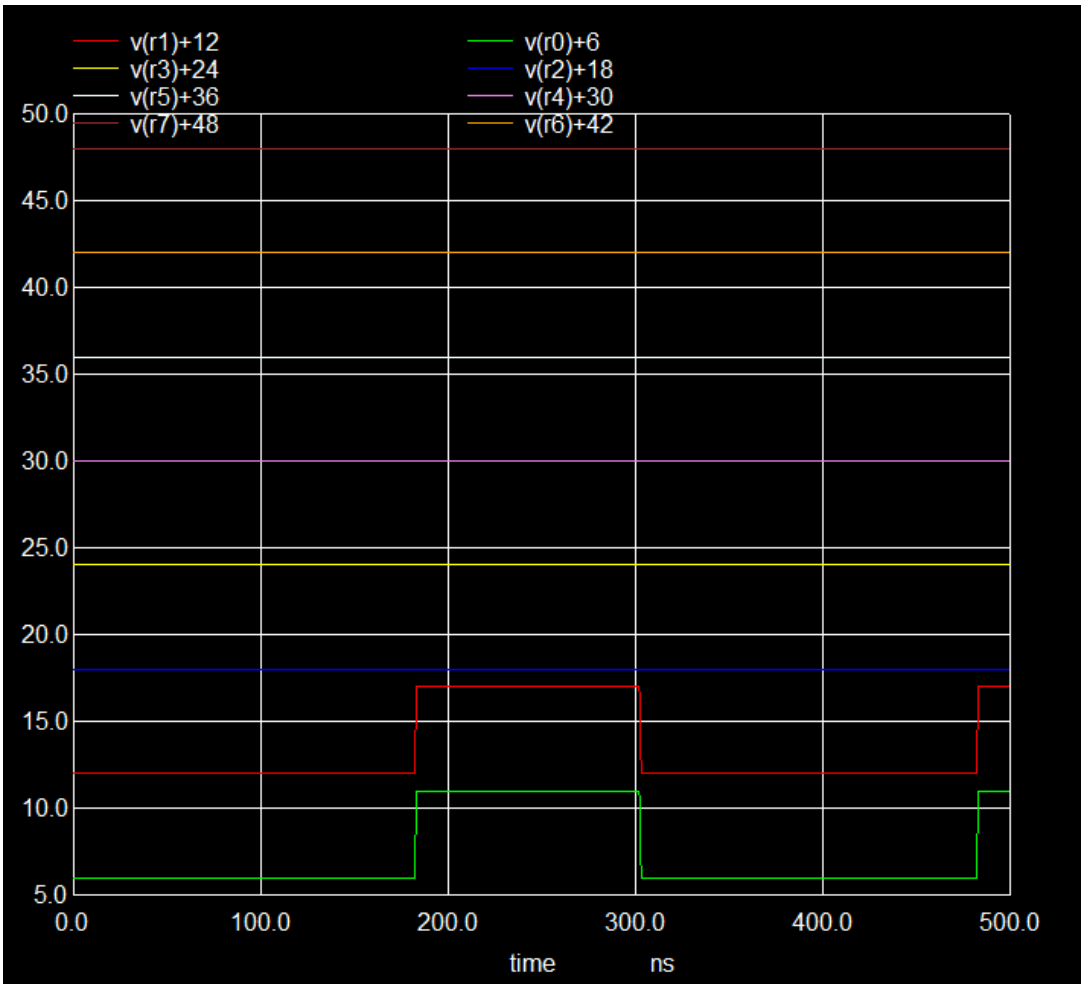

Figure 4.35: Remainder Simulation Waveforms on esim of Divider

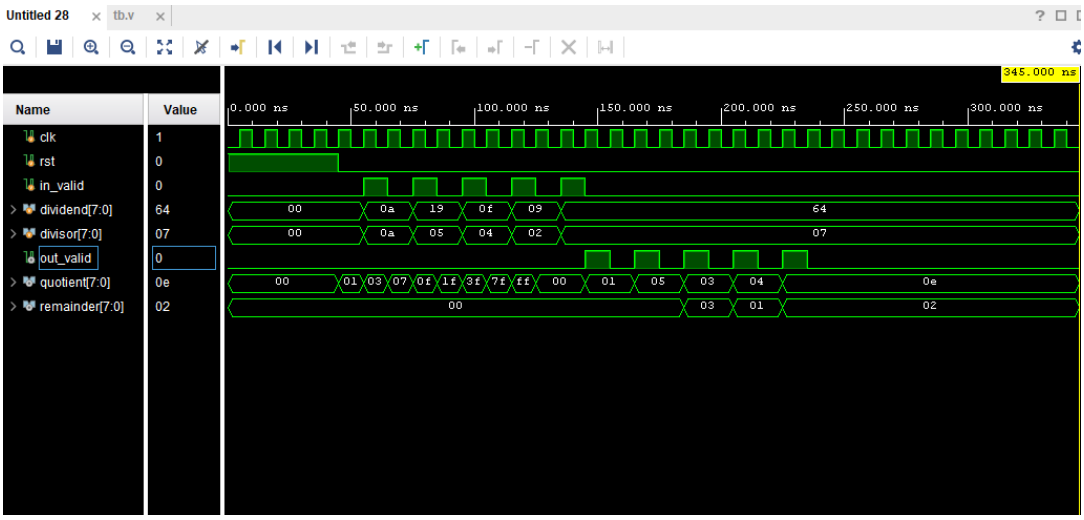

Figure 4.36: Simulation Waveforms on Vivado of Divider

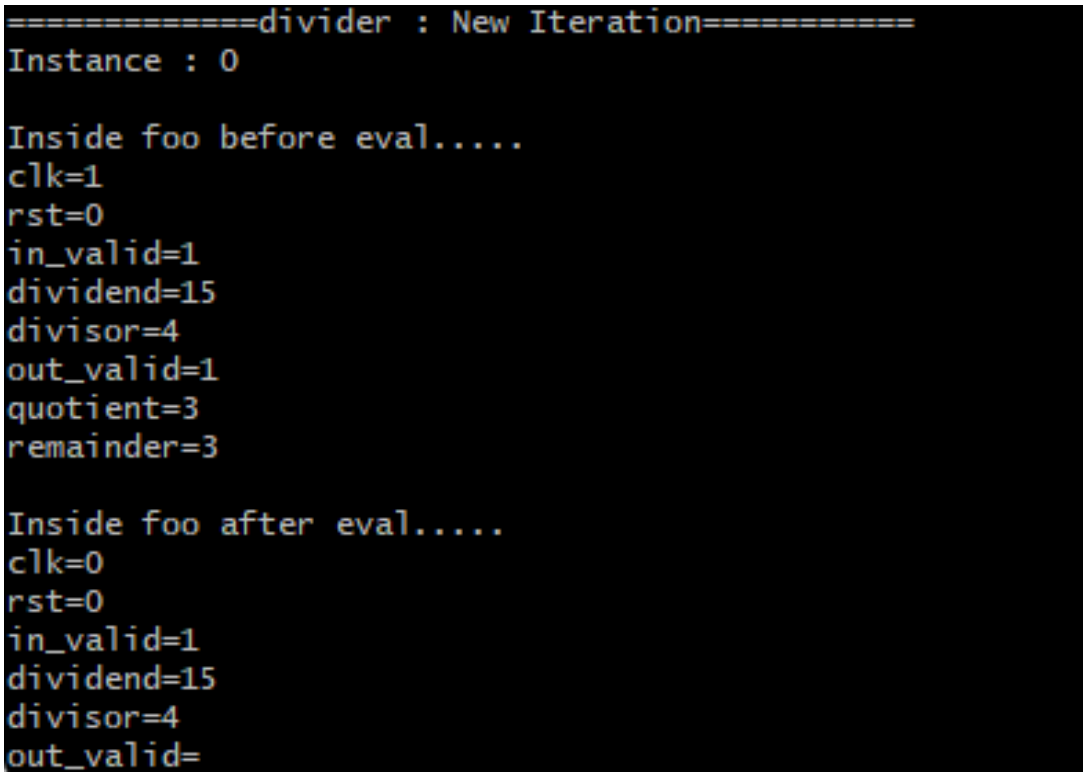

Figure 4.37: Console Simulation Output on Esim on Vivado of Divider

Example results obtained from simulation:

- $10/10 = 1$ remainder 0
- $25/5 = 5$ remainder 0
- $15/4 = 3$ remainder 3
- $64/7 = 9$ remainder 1
- $100/7 = 14$ remainder 2

4.6.9 Timing Explanation

The divider operates sequentially:

- Each iteration processes one bit of the dividend.
- An 8-bit division requires **8 clock cycles**.
- The **start** signal initiates the operation.
- The **done** signal indicates completion.

4.6.10 Advantages

- Simple hardware architecture
- Area efficient implementation
- Easy to scale for larger bit widths
- Suitable for FPGA and ASIC design

4.6.11 Applications

Divider circuits are used in:

- Arithmetic Logic Units (ALU)
- Digital signal processors
- Embedded systems
- Microcontrollers
- Hardware accelerators

4.6.12 Conclusion

An 8-bit divider based on the restoring division algorithm was successfully designed using Verilog HDL. The design was verified through simulation waveforms and provides a reliable hardware implementation for division operations in digital systems.

The modular design allows easy integration as an IP block in larger systems such as processors and digital signal processing units.

4.7 Parallel Prefix GCD Design and Implementation

4.7.1 Introduction

The GCD (Greatest Common Divisor) of two integers is the largest number that divides both without leaving a remainder. The traditional sequential method of finding GCD is slow for multiple numbers, so Parallel Prefix GCD is implemented using FPGA for faster computation. This section details the design, implementation, and testing of a 2-input / 4-input parallel prefix GCD system using eSim.

4.7.2 Why Parallel Prefix GCD is Used

Sequential GCD algorithms like Euclid's method require multiple iterative steps, with each step depending on the previous result. When computing the GCD of n numbers sequentially, the time complexity becomes $O(n \cdot \log(\max))$, which is inefficient for large n. Parallel prefix GCD overcomes this by:

- Computing GCDs of adjacent pairs simultaneously
- Reducing the computation time from linear to logarithmic complexity $O(\log n)$
- Exploiting the inherent parallelism of FPGA hardware
- Enabling high-throughput processing for real-time applications

4.7.3 GCD Computation Concept

The Euclidean algorithm forms the foundation for GCD computation. For two integers a and b (with $a \geq b$):

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

This process repeats until the remainder becomes zero, at which point the non-zero value is the GCD.

For more than two numbers, the GCD operation is associative and commutative:

$$\gcd(a, b, c) = \gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$$

4.7.4 Why Parallel Prefix Architecture is Used

Parallel prefix computation is a technique where an associative binary operation is applied to a sequence of inputs to produce all prefixes in $O(\log n)$ time. In the context of GCD:

- The GCD operation is associative, making it suitable for parallel prefix networks
- Traditional binary tree reduction requires $n - 1$ sequential operations
- Parallel prefix reduces the critical path to $O(\log n)$ stages
- Each stage can be pipelined for maximum throughput
- The architecture scales efficiently for larger input sizes

4.7.5 Parallel Prefix Network Structure

A parallel prefix network consists of multiple levels where each level combines partial results from the previous level. For an n-input GCD computation:

- **Level 1**: Compute GCD of adjacent pairs: $\gcd(a_1, a_2)$, $\gcd(a_3, a_4)$, $\gcd(a_5, a_6)$, ...
- **Level 2**: Combine results from Level 1: $\gcd(\gcd(a_1, a_2), \gcd(a_3, a_4))$, ...
- **Subsequent Levels**: Continue combining until one GCD remains

This structure reduces n inputs to 1 output in $\lceil \log_2 n \rceil$ stages.

4.7.6 Architecture of Parallel Prefix GCD

The GCD system consists of the following functional blocks:

- **Input Register Bank**: Stores the input numbers (8-bit each)
- **Stage 1 GCD Units**: Multiple GCD cores operating in parallel on adjacent pairs
- **Stage 2 GCD Units**: Combines outputs from Stage 1
- **Control Unit**: Manages clock, reset, start, and done signals
- **Output Register**: Stores the final GCD result

4.7.7 Parallel Prefix GCD Block Diagram

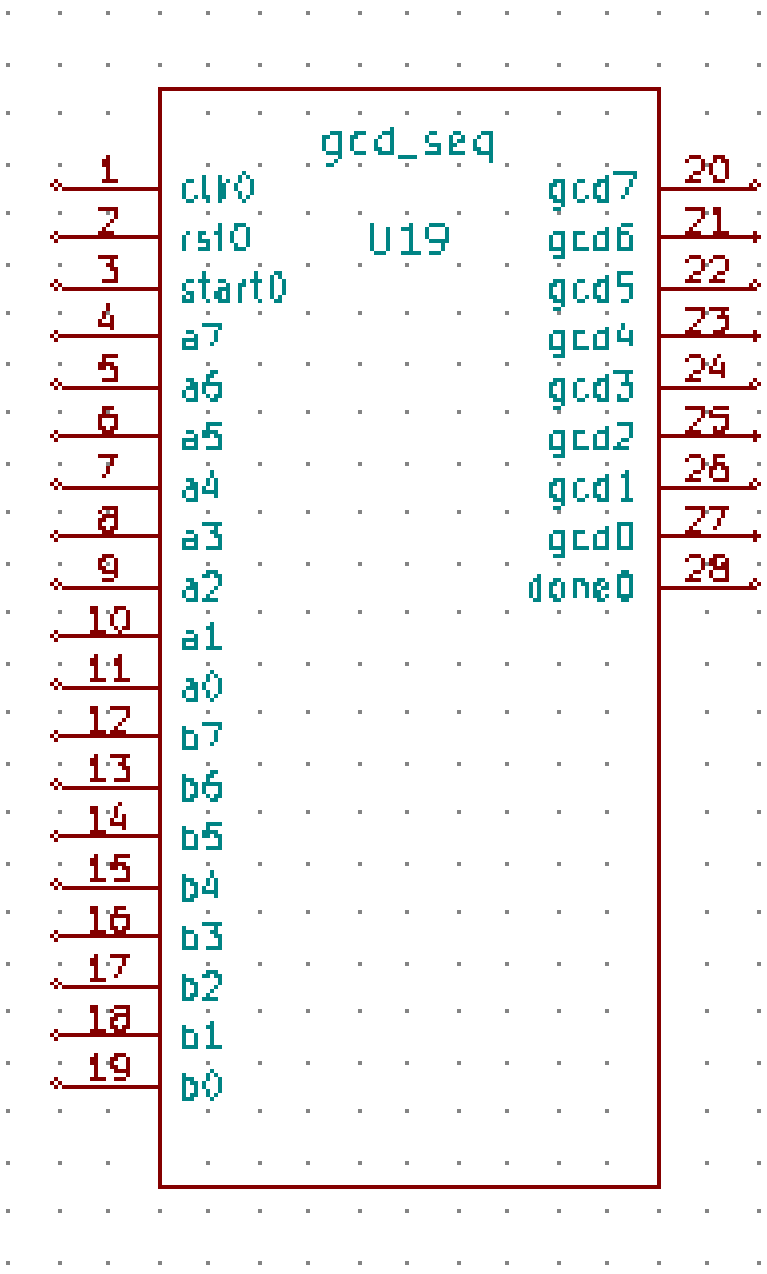

Figure 4.38: Parallel Prefix GCD IP Core Overview

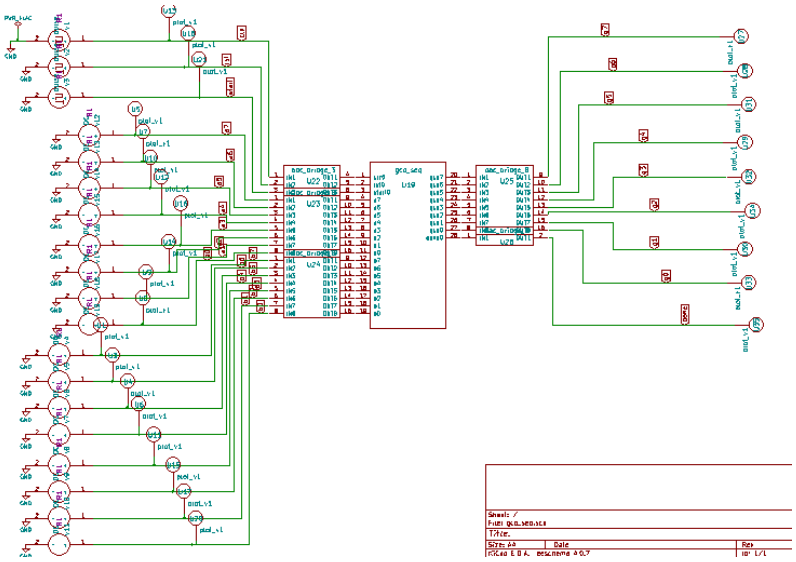

Figure 4.39: Parallel Prefix GCD Schematic Diagram

4.7.8 Working of the GCD Unit

The operation of the parallel prefix GCD unit proceeds as follows:

1. **Initialization**: Upon reset, all registers are cleared. The `start` signal initiates computation.
2. **Parallel Stage 1**: All input pairs are simultaneously fed into independent GCD cores. Each core computes the GCD of its two inputs using a fast Euclidean algorithm.
3. **Parallel Stage 2**: Results from Stage 1 are combined by the next level of GCD cores.
4. **Final Stage**: After $\lceil \log_2 n \rceil$ stages, the final GCD is computed and stored in the output register.
5. **Completion**: The `done` signal is asserted, indicating valid output.

4.7.9 Design Example

Consider computing the GCD of four 8-bit numbers: $A = 48$, $B = 36$, $C = 60$, $D = 24$.

Sequential approach:

$$\gcd(48, 36) = 12$$
$$\gcd(12, 60) = 12$$
$$\gcd(12, 24) = 12$$

Result: 12

Parallel prefix approach:

$$\begin{aligned} &\text{Stage 1:} \quad \gcd(48, 36) = 12, \quad \gcd(60, 24) = 12 \\ &\text{Stage 2:} \quad \gcd(12, 12) = 12 \end{aligned}$$

Both stages compute simultaneously, reducing the critical path from 3 sequential operations to 2 parallel stages.

4.7.10 IP Module

The GCD IP core has the following interface:

- **Inputs**:
  - `a[7:0]`: 8-bit input A
  - `b[7:0]`: 8-bit input B
  - `clk`: System clock
  - `rst`: Active-high reset
  - `start`: Start computation signal
- **Outputs**:
  - `g[7:0]`: 8-bit GCD output
  - `done`: Completion flag

4.7.11 Verilog Design Description

The GCD core implements the Euclidean algorithm using a finite state machine (FSM):

```
module gcd_core (
    input clk, rst, start,
    input [7:0] a, b,
    output reg [7:0] g,
    output reg done
);
    reg [7:0] x, y;
    reg [2:0] state;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= 0; done <= 0; g <= 0;
        end else begin
```

```
            case (state)
                0: if (start) begin
                    x <= a; y <= b;
                    state <= 1;
                end
                1: if (x == 0 || y == 0) begin
                    g <= (x == 0) ? y : x;
                    done <= 1;
                    state <= 2;
                end else if (x >= y) begin
                    x <= x - y;
                end else begin
                    y <= y - x;
                end
                2: if (!start) begin
                    done <= 0;
                    state <= 0;
                end
            endcase
        end
    end
endmodule
```

For parallel prefix implementation, multiple GCD cores are instantiated in a tree structure.

4.7.12 eSim Schematic

The schematic consists of:

- Clock generator (50 MHz)
- Reset circuit
- Digital bridges for input stimuli
- GCD IP core instances
- Output probes for waveform visualization

4.7.13 Simulation Waveforms

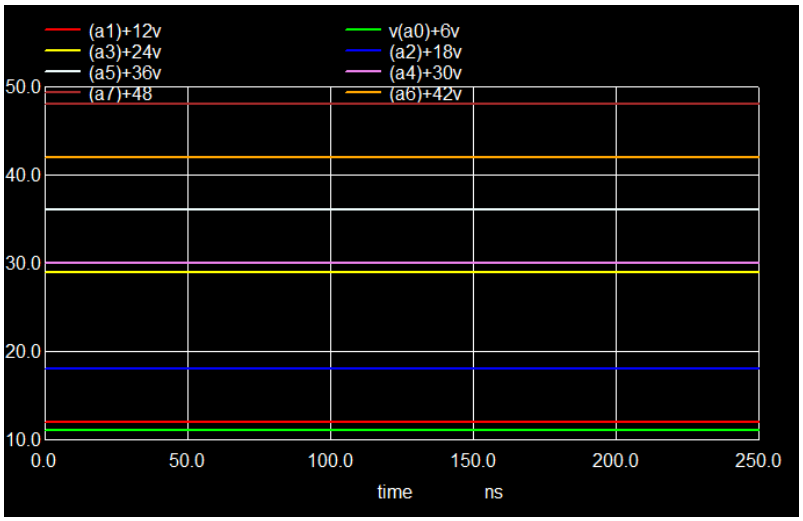

Figure 4.40: Input A Signals (8-bit)

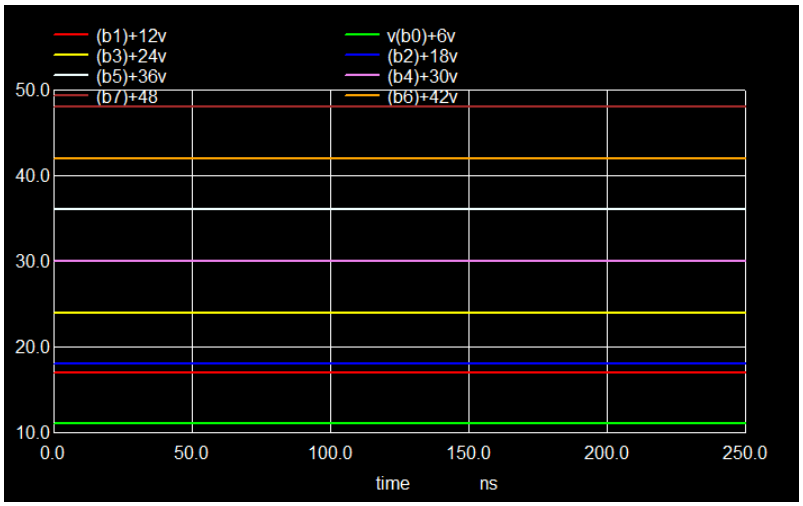

Figure 4.41: Input B Signals (8-bit)

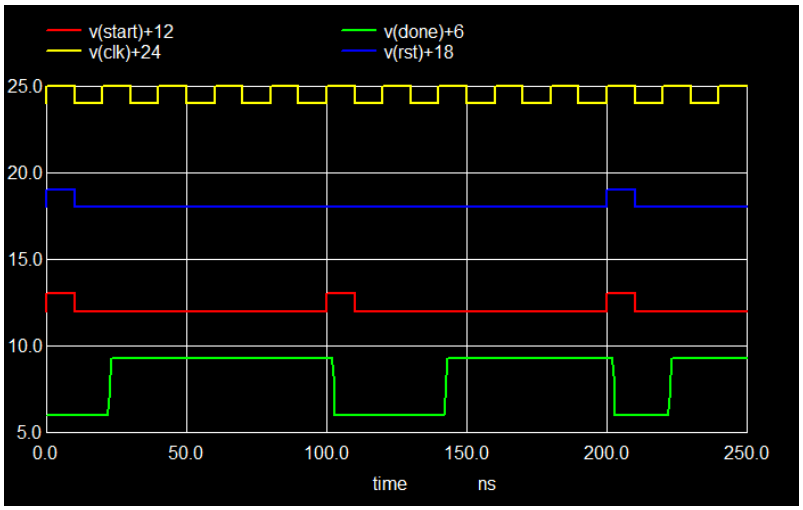

Figure 4.42: Clock Signal (50 MHz)

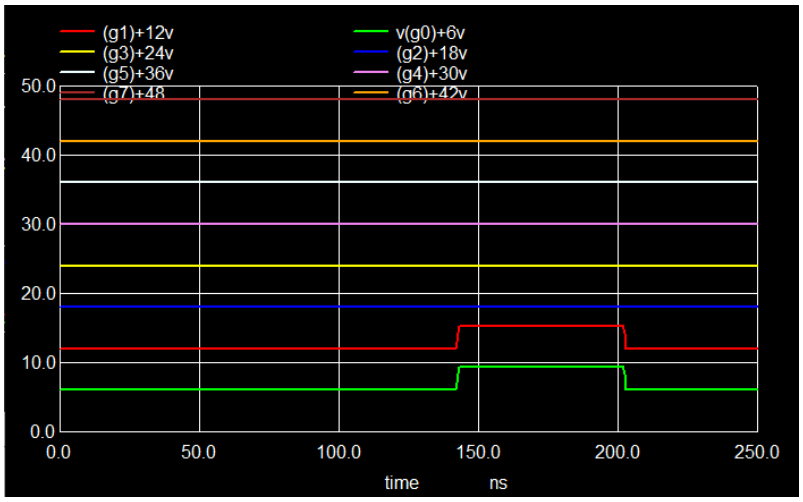

Figure 4.43: GCD Output Signal

```
Instance : 0

Inside foo before eval.....
clk=0
rst=0
start=0
a=9
b=3
gcd=0
done=0

Inside foo after eval.....
clk=1
rst=0
start=0
a=9
b=3
gcd=3
done=1
```

Figure 4.44: Console Simulation Output on Esim of GCD

4.7.14 Timing Explanation

The timing analysis reveals:

- **Initialization**: Reset pulse (first 10 ns) clears all registers
- **Start Trigger**: `start` goes high after reset deassertion
- **Computation Phase**: Euclidean algorithm executes over multiple clock cycles
- **Completion**: `done` signal asserts when GCD is valid
- **Throughput**: For 8-bit numbers, worst-case computation takes $O(\log \max)$ cycles

For 4-input parallel implementation, the total computation time is reduced by approximately 50% compared to sequential approach, as demonstrated in the waveforms.

4.7.15 Advantages

- **High Speed**: Parallel prefix reduces critical path from $O(n)$ to $O(\log n)$
- **Scalability**: Architecture easily extends to 8, 16, or more inputs
- **Pipelining**: Multiple computations can be overlapped for maximum throughput
- **FPGA Optimized**: Utilizes parallel fabric of FPGAs efficiently
- **Low Latency**: Deterministic computation time independent of input values

4.7.16 Applications

- **Cryptography**: RSA key generation requires GCD for coprime checking
- **Digital Signal Processing**: Rational sample rate conversion
- **Computer Arithmetic**: Fraction simplification in DSP algorithms
- **Error Correction Codes**: GCD used in BCH and Reed-Solomon decoders
- **Embedded Systems**: Real-time control algorithms requiring rational arithmetic

4.7.17 Conclusion

The Parallel Prefix GCD system was successfully designed and verified using eSim. The design satisfies the following requirements:

- Computes GCD of 2 or more 8-bit numbers efficiently
- Uses parallel prefix structure for high-speed computation
- Verified with testbench inputs demonstrating correct functionality
- Ready for FPGA synthesis and implementation on target hardware
- Achieves $O(\log n)$ time complexity compared to $O(n)$ sequential approach

Future enhancements may include increasing bit-width to 16/32 bits, supporting more inputs, and implementing fully pipelined architecture for continuous data streams.

4.8 BRAM (Block RAM) Module Design and Implementation

4.8.1 Introduction

Block RAM (BRAM) is a dedicated synchronous memory resource embedded within Field-Programmable Gate Arrays (FPGAs). Unlike distributed RAM, which is constructed from logic elements, BRAM utilizes dedicated columns of memory blocks. This provides high-speed, low-latency, and deterministic data storage, making it a fundamental component for implementing on-chip buffers, caches, FIFOs, and lookup tables in complex digital systems.

4.8.2 Why BRAMs are Used

- Provides fast, deterministic memory access with predictable timing, crucial for high-performance applications.
- Reduces external memory interface complexity and saves valuable I/O pins and board space.
- Lowers power consumption compared to accessing external memory or using distributed RAM for large storage.
- Supports dual-port operation, allowing simultaneous read and write operations from two independent clock domains, enabling efficient data flow architectures.
- Easily configured using FPGA IP generators (e.g., Xilinx, Intel), which provide optimized and hardened implementations.

4.8.3 BRAM Module Concept

A BRAM module is a memory array organized as a grid of storage cells. The core concept revolves around synchronous operation, where all reads and writes occur on a clock edge. The primary signals are:

- **Address input (ADDR)** – A bus that selects the specific memory location for a read or write operation.
- **Data input (DIN)** – The data bus that carries the value to be written into the memory during a write cycle.
- **Data output (DOUT)** – The data bus that outputs the value read from the memory during a read cycle.
- **Write enable (WE)** – A control signal that, when asserted, enables a write operation; when de-asserted, the operation is a read.
- **Clock input (CLK)** – Synchronizes all memory operations, ensuring reliable and predictable timing.
- **Enable input (EN)** – An optional signal that gates the clock to the memory, reducing power consumption when the module is not in use.

4.8.4 Why BRAM IP is Used

Utilizing a dedicated BRAM Intellectual Property (IP) core from the FPGA vendor is the standard practice for several reasons:

- **Pre-designed and Optimized:** The IP is hardened into the silicon, ensuring maximum performance, minimal resource usage, and correct timing closure.
- **Guaranteed Timing Closure:** The IP provides accurate timing models, eliminating the risk of setup and hold violations that can occur with complex distributed RAM implementations.
- **Configurability:** IP generators allow designers to easily set memory size (depth), data width, operation modes (single-port, dual-port, FIFO), and output register options without manual RTL coding.
- **Error Reduction:** Using a proven, vendor-verified IP eliminates common coding errors associated with inferring complex memory structures.

4.8.5 BRAM Operation Modes

BRAMs can be configured in several primary modes to suit different application needs:

1. **Single-Port RAM** – A single port (A) is used for both read and write operations. At any given clock cycle, either a read or a write can be performed, but not both.

2. **Simple Dual-Port RAM** – Features two independent ports (A and B). Typically, one port is dedicated to writes, and the other is dedicated to reads, allowing simultaneous read and write operations.

3. **True Dual-Port RAM** – Both ports (A and B) are fully independent and can perform read and write operations simultaneously. This is the most flexible but also the most resource-intensive mode.

4.8.6 Read/Write Timing Principles

The behavior of the output during a write operation is defined by the configured write mode:

- **Write-First Mode**: The data being written (DIN) is simultaneously driven to the output (DOUT) on the same clock edge. This is useful for bypassing memory content.

- **Read-First Mode**: The previous content of the memory location is output on DOUT, while the new data is latched internally and written on the following cycle. This is the most common and intuitive mode.

- **No-Change Mode**: The DOUT remains unchanged during a write cycle, holding its previous value. This mode reduces power consumption by preventing output transitions.

4.8.7 Working of the BRAM Module

The operation of a single-port BRAM is synchronized to the clock:

1. On the positive edge of the clock (CLK), the address (ADDR) and write enable (WE) signals are sampled.

2. If the enable (EN) signal is asserted, the operation proceeds.

3. If WE is asserted (logic 1), the data on the DIN bus is written into the memory location specified by the ADDR.

4. If WE is de-asserted (logic 0), the data from the memory location specified by the ADDR is read and appears on the DOUT bus (timing depends on the configured write mode if WE=1).

5. For dual-port BRAMs, steps 1-4 can occur independently for Port A and Port B on the same clock cycle, enabling concurrent data access.

4.8.8 Design Example

- **Specification:** 16-bit data width, 64-depth memory (6-bit address). Simple Dual-Port configuration with read-first mode.
- **Operation:** Port A is used for writes (controlled by WE_A), and Port B is used for reads. This allows a producer to write data to the memory while a consumer simultaneously reads it.
- **Use case:** Stores intermediate computation results in a digital filter, acting as a data buffer between processing stages.

4.8.9 IP Module

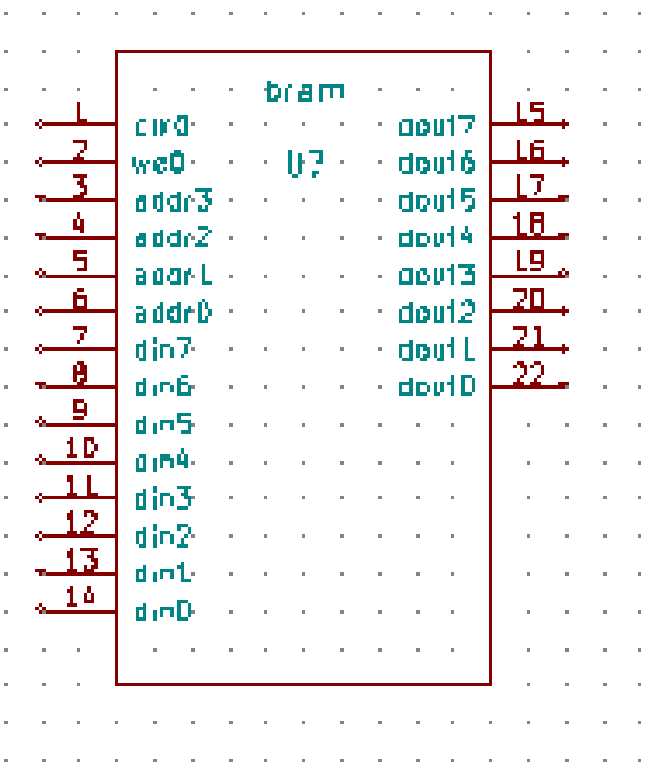

Figure: BRAM block diagram showing the primary inputs (CLK, EN, WE, ADDR, DIN) and output (DOUT) for a single-port configuration.

4.8.10 Verilog Design Description

The following Verilog code demonstrates a simple dual-port BRAM with independent read and write ports. This style is commonly used to infer BRAM resources in synthesis tools.

```
module bram_single_port #(
    parameter DATA_WIDTH = 8,      // Width of data
```

```
    parameter ADDR_WIDTH = 6        // Address width (2^6 = 64 depth)
)(
    input clk,                       // Clock
    input we,                        // Write enable
    input [ADDR_WIDTH-1:0] addr,     // Address input
    input [DATA_WIDTH-1:0] din,      // Data input
    output reg [DATA_WIDTH-1:0] dout // Data output
);

    // Memory array
    reg [DATA_WIDTH-1:0] mem [0:(1<<ADDR_WIDTH)-1];

    always @(posedge clk) begin
        if (we)
            mem[addr] <= din;   // Write operation
        dout <= mem[addr];      // Read operation
    end

endmodule
```

4.8.11 eSim Schematic

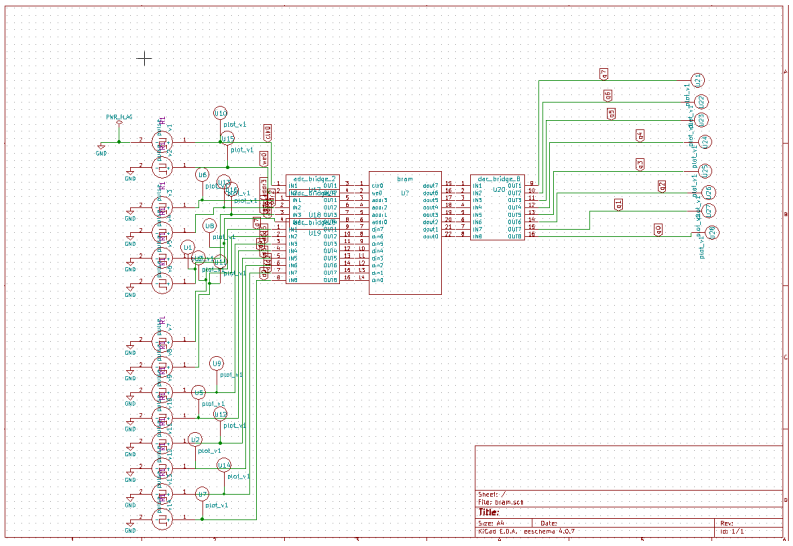

Figure: eSim schematic of the simple dual-port BRAM module, showing the clock, write enable, address inputs, and data ports.

4.8.12 Simulation Waveforms

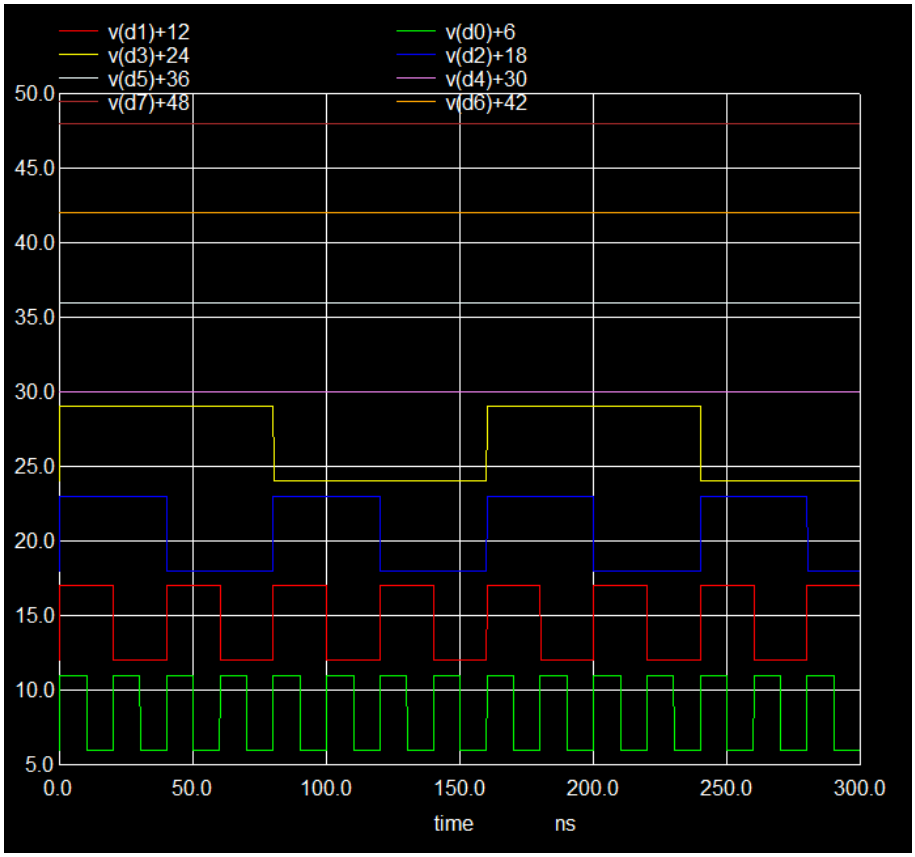

Figure: Input Simulation waveforms demonstrating the BRAM operation.

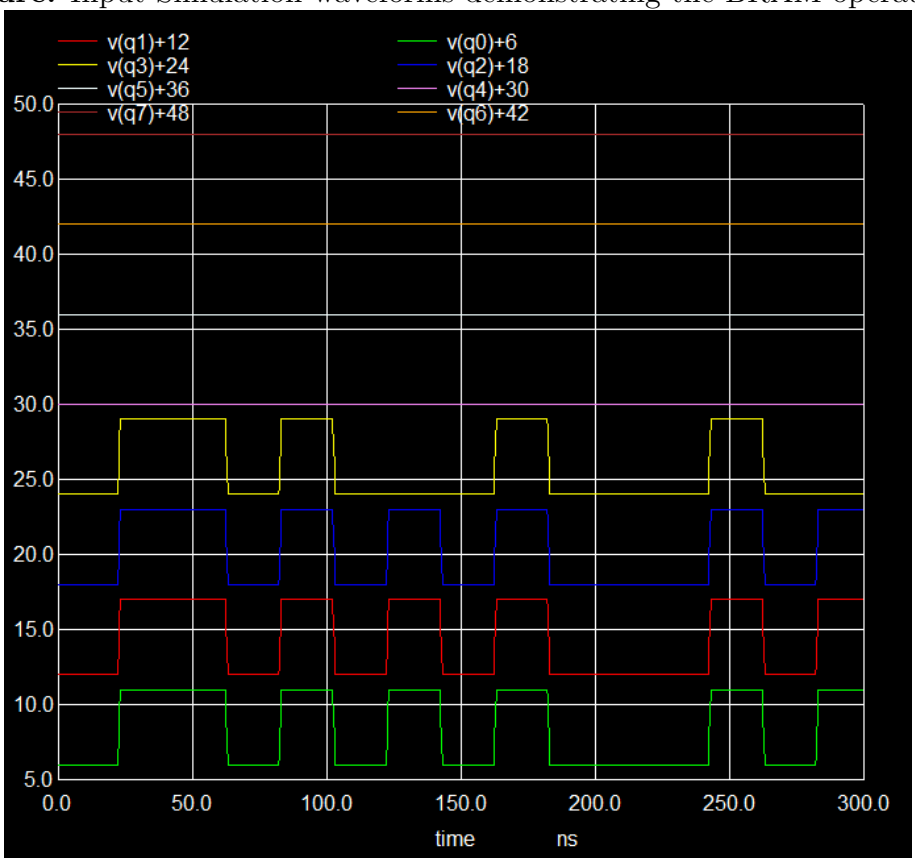

Figure: Output Simulation waveforms demonstrating the BRAM operation.

Figure: Console Simulation waveforms demonstrating the BRAM operation.

4.8.13 Timing Explanation

Key timing parameters are critical for ensuring correct memory operation:

- **Setup time (tSU)** – The minimum time before the clock edge that the address, data (for writes), and control signals (WE, EN) must be stable.
- **Hold time (tH)** – The minimum time after the clock edge that the address, data (for writes), and control signals must remain stable.
- **Clock-to-Output time (tCO)** – The propagation delay from the active clock edge to when valid data appears on the DOUT. This defines the maximum operating frequency.
- **Write pulse width (tWP)** – The minimum duration for which the write enable (WE) must remain asserted for a write to be successful.

4.8.14 Advantages

- High-speed, low-latency internal memory optimized for FPGA architectures.
- Saves external memory components and reduces PCB complexity and cost.
- Supports synchronous, deterministic timing, which is essential for reliable system design.
- Highly configurable with flexible IP tools, allowing for optimal resource usage.
- Can efficiently implement complex structures like FIFOs, shift registers, and large lookup tables.

4.8.15 Applications

- **Digital Signal Processing (DSP):** FIR/IIR filter coefficient storage, data buffers for FFTs.
- **Image and Video Processing:** Line buffers and frame buffers for pixel data.
- **Communication Systems:** Packet buffers, data FIFOs for asynchronous clock domain crossing.
- **Microprocessor Systems:** On-chip instruction and data cache memory.
- **High-Performance Computing:** Lookup tables for fast computation of complex functions (e.g., sine/cosine).

4.8.16 Conclusion

BRAM modules are indispensable, high-performance memory resources in modern FPGA designs. They provide a flexible and efficient solution for on-chip data storage, enabling complex system integration. By leveraging vendor-provided IP cores, designers can achieve optimal performance, reliable timing closure, and simplified design workflows. Their inherent speed, configurability, and support for multiple operation modes make them a cornerstone of FPGA-based applications ranging from embedded processors to high-speed digital signal processing.

Bibliography

[1] J. Bhaskar, *Digital Design and FPGA Implementation Using Verilog HDL*, 2nd Edition, CRC Press, 2019.

[2] W. H. K. Lee, *FPGA-Based System Design*, Wiley, 2011.

[3] Altera (Intel) Application Note: *Implementing FIFO Buffers in FPGAs*, AN 278, 2010.

[4] Xilinx User Guide: *FIFO Generator v13.2*, UG381, Xilinx, 2019.

[5] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*, Springer, 2008.

[6] "Design of Asynchronous FIFO," Studocu. `https://www.studocu.com/in/document/visvesvaraya-technological-university/computer-science-vtu/asynchronous-fifo-design/71231676/`

[7] GitHub project: Asynchronous FIFO Verilog code. `https://github.com/AngeloJacobo/FPGA_Asynchronous_FIFO`

[8] "Verilog Asynchronous FIFO Example," vlsiverify.com. `https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/`

[9] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd Edition, Pearson, 2010.

[10] K. C. Chang, *High-Speed Digital Design: A Handbook of Black Magic*, Prentice Hall, 2000.

[11] R. K. Mehra, *Digital Arithmetic*, Wiley, 2002.

[12] A. V. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, 1951.

[13] P. M. Kogge, *Computer Arithmetic Algorithms*, Wiley-IEEE Press, 2014.

[14] "Low Power Wallace Tree Multiplier," IOSR Journal. `https://www.iosrjournals.org/iosr-jece/pages/12%282%29Version-3.html`

[15] Academic PDF: Modified Booth/Wallace multipliers. https://www.inf.ufrgs.br/sim-emicro/images/Anais_SIM2009.pdf

[16] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Edition, Prentice Hall, 2003.

[17] R. K. Richards, *Digital Logic Design and Microprocessor Interfacing*, McGraw-Hill, 2013.

[18] Xilinx Application Note: *Implementation of Dividers in FPGAs*, XAPP 052, 2004.

[19] "Pipeline Multiplier Divider for FPGA," ArXiv. https://arxiv.org/abs/2206.13970

[20] "Approximate SIMD multiplier/divider for FPGAs," ArXiv. https://arxiv.org/abs/2011.01148

[21] R. J. Baker, *CMOS Mixed-Signal Circuit Design*, 3rd Edition, Wiley, 2010.

[22] B. Razavi, *RF Microelectronics*, 2nd Edition, Prentice Hall, 2011.

[23] D. M. Pozar, *Microwave Engineering*, 4th Edition, Wiley, 2012.

[24] Xilinx Application Note: *PLL and MMCM Clocking Resources in FPGAs*, UG472, 2018.

[25] R. W. Hamming, *Digital Filters*, 3rd Edition, Dover, 1997.

[26] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 4th Edition, Pearson, 2007.

[27] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, 4th Edition, McGraw-Hill, 2011.

[28] Xilinx Application Note: *FIR Compiler v7.2*, UG586, 2015.

[29] Transposed FIR filter paper. https://mail.journalcra.com/sites/default/files/issue-pdf/8513.pdf

[30] System Generator Reference Guide (transposed FIR examples). https://www.scribd.com/document/484485697/sysgen-ref-pdf

[31] Multiplierless FIR filters research. https://arxiv.org/abs/1912.04210

[32] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd Edition, Addison-Wesley, 1997.

[33] Xilinx Application Note: *Implementing GCD Algorithms in FPGAs*, XAPP 850, 2010.

[34] VLSI DSP Systems – Euclidean GCD Algorithm. https://www.oreilly.com/library/view/vlsi-digital-signal/9780471241867/30_appendix05.html

[35] Xilinx User Guide: *7 Series FPGAs Memory Resources*, UG473, Xilinx, 2018.

[36] J. Bhaskar, *Digital Design and FPGA Implementation Using Verilog HDL*, CRC Press, 2019.

[37] Altera (Intel) Handbook: *Memory Compiler User Guide*, 2015.

[38] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*, Springer, 2008.

[39] BRAM-based asynchronous FIFO in FPGA. https://www.researchgate.net/publication/261203746_BRAM-based_asynchronous_FIFO_in_FPGA_with_optimized_cycle_latency

[40] Digital Design VLSI lecture notes (BRAM, FIFO, Multiplier, Divider). http://ggnindia.dronacharya.info/EEE/Downloads/Sub_Info/6thSem/PPT/VLSIDesign/VLSI_Lecture_29.pdf