Chapter 2

INTRODUCTION

VLSI (Very Large Scale Integration) refers to the process of integrating thousands to millions of transistors on a single silicon chip. It forms the backbone of modern electronics, enabling the development of compact, high-performance devices such as processors, memory chips, and communication modules.

VLSI design involves creating digital circuits and systems using hardware description languages like Verilog or VHDL. The design process includes specification, modeling, synthesis, and implementation on hardware platforms like FPGAs or ASICs.

Verification is a critical phase in the VLSI design cycle. It ensures that the designed hardware functions correctly as per the intended specifications before manufacturing. Using advanced techniques like simulation, testbenches, assertions, and formal verification, errors can be detected early in the design flow.

Together, VLSI design and verification play a vital role in developing reliable and efficient digital systems for real-world applications in computing, communication, automotive, and consumer electronics.

2.1 Understanding Design Verification (DV)

Design Verification (DV) is a key step in the VLSI (Very Large Scale Integration) design flow. It involves verifying that the RTL (Register Transfer Level) design behaves as intended and meets the specifications provided by the architecture team. After the design is implemented using hardware description languages like Verilog or VHDL, verification ensures its logical correctness before the design proceeds to synthesis and physical implementation.

Verification can include both functional and formal approaches. Functional verification uses simulations with testbenches, while formal verification mathematically proves properties of the design. Without proper verification, design bugs may remain undetected and could cause device failure after fabrication, resulting in significant cost, time, and resource loss.

2.2 Critical Role of Verification in VLSI Development

In modern VLSI design, chips are composed of millions to billions of transistors and numerous functional blocks. As a result, **verification has become the most time-consuming and resource-intensive phase**, often taking up to 60–70% of the total development time.

DV ensures that each block within the chip and their interactions are functionally correct under all conditions, including edge cases. It also ensures that the design adheres to interface protocols, timing constraints, and power efficiency requirements. Verification protects against logical flaws that may otherwise go undetected until late in the design cycle or post-manufacturing.

Thus, the verification phase is not just a checkpoint—it's an essential guardrail ensuring the overall success of the silicon.

2.3 Why Verification is Essential in Chip Design

The importance of design verification lies in its **ability to detect errors early** in the design cycle, minimizing the risk of failures post-fabrication. Key reasons for its importance include:

- **Reduces Cost of Errors:** Post-silicon bugs can lead to re-spins, which cost millions of dollars and months of delay.
- **Improves Design Reliability:** A fully verified chip performs reliably in all operating conditions.
- **Ensures Functionality:** Confirms that the chip does exactly what it is designed to do, according to specification.
- **Compliance Assurance:** Helps the design meet industry and protocol standards (e.g., SPI, I2C, AMBA).
- **Faster Time-to-Market:** Efficient verification helps reduce the debug cycle and accelerates product delivery.

2.4 Responsibilities of a Design Verification Engineer

A Design Verification Engineer plays a pivotal role in the success of a chip design project. Their responsibilities include:

- **Understanding Functional Specifications:** Gaining a complete understanding of how the design is supposed to work.
- **Creating Verification Plans:** Developing test strategies that outline what needs to be tested and how.
- **Developing Testbenches:** Writing code in SystemVerilog, Verilog, or using verification methodologies like UVM (Universal Verification Methodology) to simulate and monitor the behavior of the design.
- **Running Simulations and Debugging:** Simulating the design to identify issues, using waveform analysis and logs to isolate problems.

- **Measuring Coverage:** Using coverage metrics (functional, code, branch coverage) to evaluate the thoroughness of verification and identify untested areas.

A good DV engineer combines digital design knowledge, programming skills, and analytical thinking to ensure the design is bug-free and ready for tape-out.

2.5 Verification Strategies and Techniques in Practice

In modern VLSI projects, various techniques are employed to ensure complete verification:

- **Simulation-Based Verification:** This is the most common technique where a testbench applies a range of input scenarios to the DUT (Design Under Test) to verify its functionality. Simulation tools like Vivado, ModelSim, or QuestaSim are typically used.
- **Constrained Random Verification:** Random test vectors are generated within user-defined constraints. This method helps uncover unexpected bugs and corner cases that directed tests might miss.
- **Formal Verification:** Uses formal methods like theorem proving or model checking to mathematically prove the correctness of the design. This is particularly useful for checking properties like deadlock freedom, FIFO order, or protocol compliance.
- **Assertion-Based Verification (ABV):** SystemVerilog assertions (SVA) are embedded in RTL or testbenches to continuously monitor and validate specific conditions or sequences. They help in pinpointing bugs early during simulation.
- **Coverage-Driven Verification:** Measures how much of the design is exercised by tests using code coverage (statements, branches) and functional coverage (specific feature checks). It guides engineers to improve test quality and completeness.

Chapter 3

TASK PERFORMED

3.1 Digital Electronics

Digital Electronics is the branch of electronics that deals with circuits and systems which operate on discrete voltage levels—commonly represented as binary 0s and 1s. Unlike analog systems, which process continuous signals, digital systems are more reliable, noise-tolerant, and easier to scale with technology. The fundamental components of digital systems are logic gates, which are used to perform binary operations and build more complex circuits like arithmetic units, control units, memory blocks, and processors.

In Digital Electronics, information is represented, processed, and stored using digital signals. It covers key concepts such as number systems, Boolean algebra, logic gate operations, combinational and sequential circuit design, memory units, and data flow control—all of which are crucial in building real-world digital systems.

3.1.1 Core Concepts in Digital Electronics

The following topics were covered during the internship, each playing a critical role in VLSI design and verification:

- **Number Systems and Conversions**

  Covers binary, decimal, octal, and hexadecimal systems and how to convert between them. Essential for data representation in hardware and memory addressing.

- **Logic Gates**

  Includes basic gates (AND, OR, NOT) and universal gates (NAND, NOR), along with XOR/XNOR. These gates form the foundation for all digital circuit design.

- **Boolean Algebra**

  Teaches logical expression simplification using rules and theorems like DeMorgan's Laws. Crucial for minimizing logic in VLSI circuits.

- **Karnaugh Maps (K-Maps)**

  Visual tools for reducing Boolean expressions to minimal form. Helps in optimizing combinational logic.

- **Combinational Circuits**

  Circuits without memory where output depends only on current inputs. Includes:

  • Adders and Subtractors

- Multiplexers/Demultiplexers
- Encoders/Decoders
- Code Converters

- **Sequential Circuits**

  Circuits with memory where output depends on current inputs and previous states. Includes:
  - Flip-Flops (SR, D, JK, T)
  - Counters (Up, Down, Synchronous, Asynchronous)
  - Shift Registers

- **Flip-Flops and Latches**

  Basic memory elements used to store binary data. Flip-flops are triggered by clock edges, making them essential in FSMs and pipelines.

- **Counters**

  Used for counting pulses and events. Can be synchronous or asynchronous. Important in digital clocks, timers, and sequence generators.

- **Shift Registers**

  Used for data storage and transfer. Support parallel-to-serial and serial-to-parallel data movement, often used in communication systems.

3.2 C Programming

C is a structured, high-performance programming language widely used in embedded systems and low-level programming. It forms the backbone of many system-level applications and is essential for engineers working close to hardware. Its syntax and logic flow are also helpful for understanding HDLs like Verilog.

3.2.1 Importance of C Programming in VLSI

In the VLSI domain, C is used to write firmware, model hardware behavior before RTL implementation, and develop supporting tools for verification. A strong foundation in C improves a VLSI engineer's ability to understand data handling, memory usage, and structured programming, which are directly applicable when working with FSMs, protocols, and memory controllers.

3.2.2 Topics Covered

- **C Program Structure**

  Learned the standard structure of a C program including #include directives, main() function, and return types.
- **Data Types and Variables**

  Worked with basic types like int, char, float, and user-defined types using struct.
- **Operators and Expressions**

  Understood arithmetic, relational, logical, bitwise, and assignment operators for writing expressions.
- **Control Statements**

  Practiced decision-making using if-else, switch-case, and looping with for, while, do-while.
- **Functions**

  Explored how to define and call functions with and without parameters and return values.
- **Arrays and Strings**

  Learned declaration and manipulation of one-dimensional arrays and string functions like strlen, strcpy.
- **Pointers**

  Gained experience in pointer arithmetic, arrays and pointers relationship, and dynamic memory allocation using malloc and free.

3.3 Hardware Description Language(HDL)

HDL refers to Hardware description language. A high-level computer language that can model, represent and simulate the digital design. HDL is a language that describes the behavior or structure of the digital circuits. Textual representation of a digital logic design. HDLs resemble high-level programming languages such as C or Python, but it's a fundamental difference: Statements in HDL code involve parallel operation, whereas programming languages represent sequential operation.

Characteristics of HDL

- **Simulation:** Process in which the functionality of the design is verified using suitable test vectors.
- **Logic Synthesis:** Process in which HDL Code is translated into an equivalent gate-level schematic
- **Reusability:** A working recipe for completing your next design quickly by reusing existing code.

3.3.1 Verilog

Verilog is a hardware description language (HDL) used to model electronic systems. It allows designers to describe the structure and behavior of digital circuits and systems such as processors, memory units, and other logic-based components.

➢ History of Verilog

- **Developed by:** Phil Moorby at Gateway Design Automation (1984)
- **Acquired by:** Cadence Design Systems (1990)
- **Standardized by:** IEEE as **IEEE 1364** in 1995, updated in 2001 and 2005.
- **System Verilog:** An extension of Verilog developed in the early 2000s, standardized as **IEEE 1800**.

➢ Key Concepts in Verilog

1. Design Abstraction Levels

Verilog can describe systems at different levels:

- Behavioral (high-level, algorithmic)
- Register Transfer Level (RTL) (describes data flow between registers)
- Gate Level (describes logic gates and interconnections)
- Switch Level (lowest level, uses transistors)

2. Modules

The building blocks in Verilog are called modules. Each module can represent a logic circuit.

➢ Verilog Syntax Overview

1. Modules

Used to encapsulate design components.

Syntax:

```
module module_name (port_list);
  // Declarations
  // Logic
endmodule
```

2. Ports

- input, output, inout
- Can be scalar or vector (e.g., [3:0] for a 4-bit bus)

3. Data Types

- wire: Used for combinational logic

- reg: Used to store values (sequential logic)
- integer, real, time, parameter, etc.

4. Operators

- Arithmetic: +, -, *, /
- Logical: &&, ||, !
- Bitwise: &, |, ^, ~
- Relational: ==, !=, <, >

5. Procedural Blocks

- always: Executes repeatedly when sensitivity list changes
- initial: Executes once at time 0

6. Control Statements

- if, else
- case, casex, casez
- for, while, repeat, forever

3.3.2 System Verilog

System Verilog is a **hardware description and verification language (HDVL)**. It extends Verilog by adding powerful features for:

- **Design modeling** (like enhanced data types, interfaces, assertions)
- **Verification** (like classes, randomization, functional coverage, assertions)
- **Testbench automation** (UVM methodology, object-oriented verification)

It was standardized as **IEEE 1800** (in 2005, 2009, 2012, 2017).

3.3.1 System Verilog for Design

1. Enhanced Data Types

- **logic**: Replaces reg and wire with more consistent behavior
- **bit [7:0]**: For unsigned logic (0 or 1)
- **int**, **shortint**, **longint**: Signed integers
- **enum**: Enumerated types

2. Interfaces

Used to group related signals together, simplifying module connections.

3. Always Comb / Always FF

New always_comb, always_ff, always_latch blocks improve synthesis clarity.

3.3.2 System Verilog for Verification

1. Classes and OOP

Supports classes, inheritance, polymorphism, and constructors.

2. Randomization

Built-in constrained random testing.

3. Functional Coverage

Measure which values/scenarios were exercised during simulation.

4. Assertions (SVA)

Used to validate behavior during simulation.

5. Testbench Components

System Verilog testbenches often follow a layered approach.

3.4 Project

- **Advanced eXtensible Interface (AXI) verification using Verilog**

3.4.1 Advanced eXtensible Interface (AXI) verification using Verilog

Verification Language: Verilog

Duration: 2 Months

1. Description:

- The **Advanced eXtensible Interface (AXI)** is a high-performance, high-frequency, synchronous, and burst-based on-chip communication protocol, part of the ARM AMBA (Advanced Microcontroller Bus Architecture) family. AXI is widely used in **System-on-Chip (SoC)** designs to facilitate communication between master and slave components such as processors, memory interfaces, and IP blocks.
- AXI operates using four primary signals:
  - **AW (Write Address Channel) –** carries the address and control information for write transactions.
  - **W (Write Data Channel) –** carries the data being written to the slave.
  - **B (Write Response Channel) –** carries the response from the slave after write completion.
  - **AR (Read Address Channel)** – carries the address and control information for read transactions.
  - **R (Read Data Channel)** – carries the data returned from the slave along with response info.

- AXI provides support for separate **read** and **write** data channels, allowing **out-of-order transactions** and **high throughput**. It is commonly used in **VLSI** and **FPGA** verification environments using **Verilog or SystemVerilog**, often along with **UVM (Universal Verification Methodology)**.

2. Steps Followed to Build SPI Protocol:

- Studied the AXI protocol and its five channels: **AW, W, B, AR, and R**
- Designed the **AXI master module** in Verilog using FSMs to generate valid read/write transactions
- Created the **AXI slave module** with proper response generation and data storage logic
- Integrated master and slave modules into a **top-level AXI system**
- Wrote a **testbench** to simulate AXI transactions and monitor protocol handshakes (VALID/READY)
- Performed simulation in **Vivado/ModelSim** and verified transaction sequences through waveform analysis

3. Methodology:

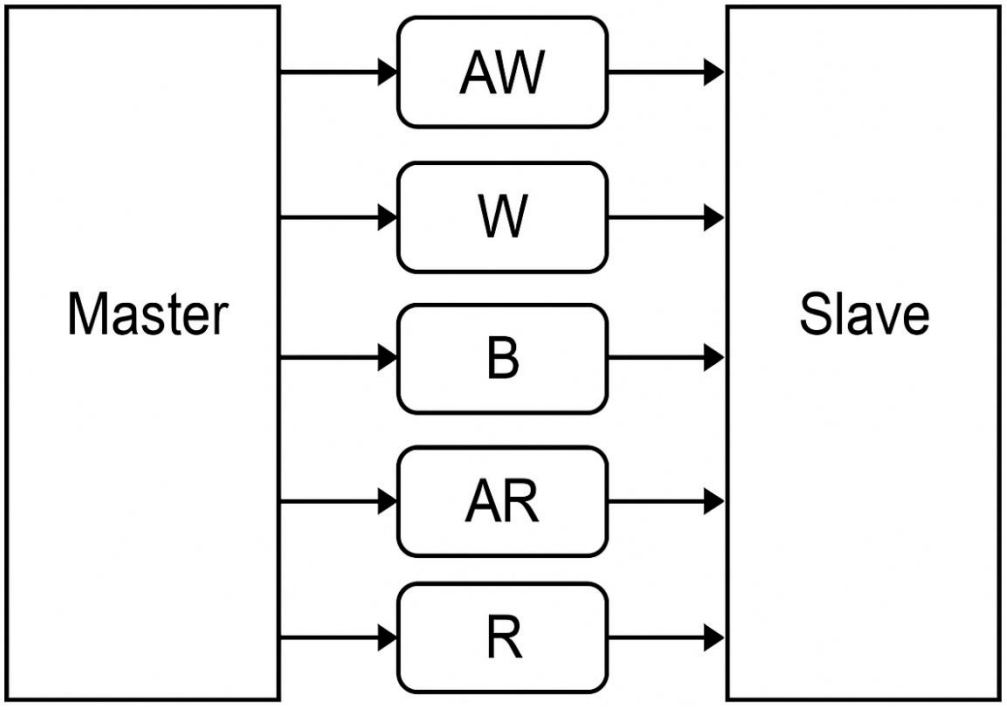


Figure 3.4.1: Methodology of AXI

4. Code Components:

AXI Master – Code Components

- **clk, rst (Inputs)**

  Synchronizes logic and registering signals.

- **aw_addr, aw_valid (Inputs)**

  Drive the write address channel.

- **w_data, w_valid (Inputs)**

  Address the write data channel.

- **b_ready (Input)**

  Facilitates the write response channel.

- **ar_addr, ar_valid (Inputs)**

  Drive the read address channel.

- **r_ready (Input)**

  Enables data reception on the read data channel.

- **state (FSM control)**

  Implements a state machine with IDLE, LOAD, TRANSFER, and DONE states.

AXI Slave – Code Components

- **clk, rst (Inputs)**

  Used to synchronize internal behavior and reset logic.

- **aw_ready, b_resp (Outputs)**

  Drive the write address and write response channels.

- **w_ready (Output)**

  Supports the write data channel.

- **ar_ready, r_resp (Outputs)**

  Drive the read address and read response channels.

- **r_data (Output)**

  Supplies read data to the master.

- **reg_file (Output)**

  Internal memory/register array for read/write operations (e.g., reg [31:0] mem [0:255]).

- **write_enable / read_enable (Internal signals)**

  Signals used to control internal read/write operations.

Testbench – Code Components

- **clk (Register)**

  Testbench-generated clock signal used to drive both master and slave modules.

- **rst (Register)**

  Reset signal used to initialize all registers and FSMs at the start of simulation.

- **start (Register)**

  Control signal to begin AXI communication by triggering the master.

- **master_data_in (Register)**

  16-bit input data given to the AXI master for transmission.

- **master_data_out (Wire)**

  Captures the 64-bit data received by the master from the slave during simulation.

- **slave_received (Wire)**

  Shows the data received by the slave from the master during the AXI transfer.

- **master_data, slave_data (Wires)**

  Capture data exchanged during communication.

- **Clock Generator (always #5 clk = ~clk)**

  Creates a periodic clock signal with a 10-time-unit period (50% duty cycle).

- **Initial Block (initial begin ... end)**

  Controls the sequence of test actions: reset, apply inputs, trigger start, and wait for completion.

- **wait(done) Statement**

  Pauses the simulation until the AXI master completes the 8-bit data transfer.

- **$monitor Statements**

  Used to display the values of master_data, and slave_data during simulation for debugging and verification.

- **$finish Statement**

  Ends the simulation once the AXI transfer is complete and data is displayed.

5. Verilog code

5.1 AXI_slave module

```
module axi_slave(
input clk,arvalid,res_n,
input [1:0]arburst,
input [2:0]arsize,
input [3:0]arlen,
input [4:0]araddr,
output arready,
input awvalid,
input [4:0]awaddr,
output awready,
input [3:0]awlen,
input [1:0]awburst,
input [2:0]awsize,
input wvalid,
input [15:0]wdata,
output wready,
input wlast,
input bready,
output bvalid,
output bresp,
output reg [15:0]rdata,
output rresp,rlast,
input rready,
output rvalid
   );

   parameter [3:0]IDLE = 4'b0001,SETUP = 4'b0010, PREACCESS = 4'b0100,ACCESS =
4'b1000,
                     SETUPW = 4'b0011, PREACCESSW = 4'b0111,ACCESSW = 4'b1111,
                     WTERMINATE = 4'b0101;
   reg [3:0]current_state,next_state=IDLE;
   reg [1:0]burst;
   reg [2:0]size;
   reg [3:0]len;
   reg [31:0]addr;
   reg last;
   reg [15:0]memory[31:0];
   initial
     begin
       memory[0] = 16'hffff;
       memory[1] = 16'h1111;
       memory[2] = 16'h2222;
       memory[3] = 16'h1234;
       memory[4] = 16'h7890;
       memory[5] = 16'h1111;
     end
```

```
always@(posedge clk,negedge res_n)
begin
if(!res_n)
  current_state <= IDLE;
else
  current_state <= next_state;
end

always@(current_state,arvalid,rready,awvalid,wvalid,bready)
begin
case(current_state)
IDLE : begin
        last = 0;
        rdata = 32'b0;
        if(arvalid)
          next_state <= SETUP;
        else if(awvalid)
          next_state <= SETUPW;
        else
          next_state <= IDLE;
       end

SETUPW : begin
          if(awvalid)
            begin
              burst = awburst;
              size = awsize;
              len = awlen + 1;
              addr =  awaddr;
              next_state <= PREACCESSW;
            end
          else
            next_state <= IDLE;
        end

PREACCESSW : begin
              if(wvalid)
                next_state <= ACCESSW;
              else
                next_state <= PREACCESSW;
             end

ACCESSW : begin
           if(len != 4'd0)
             begin
               if(wlast)
                 next_state <= WTERMINATE;
               else
                 next_state <= PREACCESSW;
               case(size)
                 3'b000: memory[addr] = {8'd0,wdata[7:0]} ;
```

```
                3'b001: memory[addr] = wdata;
                default : memory[addr] = 16'd0;
              endcase
              case(burst)
                2'b00: addr = addr;
                2'b01: addr = addr + 1;
                default : addr = addr;
              endcase
              len = len - 1;
            end
          else
            next_state <= WTERMINATE;
        end

WTERMINATE : begin
            if(bready)
              next_state <= IDLE;
            else
              next_state <= WTERMINATE;
          end

SETUP : begin
        if(arvalid)
          begin
            burst = arburst;
            size = arsize;
            len = arlen + 1;
            addr = araddr;
            next_state <= PREACCESS;
          end
        else
          next_state <= IDLE;
      end
PREACCESS : begin
            rdata = 32'd0;
            if(rready)
              next_state <= ACCESS;
            else
              next_state <= PREACCESS;
          end
ACCESS : begin
         if(len != 4'd0)
           begin
             if(len == 4'd1)
               begin
                 last = 1;
                 next_state <= IDLE;
               end
             else
               next_state <= PREACCESS;
             case(size)
```

```
                3'b000 : rdata = (memory[addr] & 16'h00ff);
                3'b001 : rdata = memory[addr];
                default : rdata = 16'd0;
              endcase
              case(burst)
                2'b00 : addr = addr;
                2'b01 : addr = addr + 1;
                default : addr = addr;
              endcase
              len = len - 1;
            end
          else
            next_state <= IDLE;
        end
    default : next_state <= IDLE;
    endcase
    end

assign aready = (current_state == SETUP);
    assign rresp = (current_state == ACCESS);
    assign rlast = (rresp && last);
    assign rvalid = (current_state == ACCESS);
    assign awready = (current_state == SETUPW);
    assign wready = (current_state == ACCESSW);
    assign bresp = (current_state == WTERMINATE);
    assign bvalid = (current_state == WTERMINATE);

  endmodule
```

5.2 AXI_slave Testbench

```
module axi_slave_tb;
reg clk,arvalid,res_n;
reg [1:0]arburst;
reg [2:0]arsize;
reg [3:0]arlen;
reg [4:0]araddr;
wire aready;
wire [15:0]rdata;
wire rresp,rlast;
reg rready;
wire rvalid;

reg awvalid;
reg [4:0]awaddr;
wire awready;
reg [3:0]awlen;
reg [1:0]awburst;
reg [2:0]awsize;
```

```
reg wvalid;
reg [15:0]wdata;
wire wready;
reg wlast;

reg bready;
wire bvalid;
wire bresp;


axi_slave uut(clk,arvalid,res_n,arburst,arsize,arlen,araddr,arready,awvalid ,awaddr,

awready,awlen,awburst,awsize,wvalid,wdata,wready,wlast,bready,bvalid,bresp,rdata,rresp,rlast,rread
y,rvalid);

always #5 clk = ~clk;
initial
begin
res_n = 0;
clk = 0;
arvalid = 0;
arburst = 2'b00;
arsize = 3'b000;
arlen = 4'b0000;
araddr = 5'b00000;
rready = 0;
awvalid = 0;
awburst = 2'b00;
awsize = 3'b000;
awlen = 4'b0000;
awaddr=5'b00000;
wlast = 0;
bready = 0;
wdata = 0;
wvalid = 0;
end

initial
begin
#30 res_n=1;
#5 arvalid = 1;arburst = 2'b01;arsize = 3'b000;arlen = 4'b0011;araddr=5'b00000;
#10;
#10 arvalid = 0;arburst = 2'b00;arsize = 3'b000;arlen = 4'b0000;araddr=5'b00000;
#10 rready = 1;
#10;
#10;
#10;
#10;
#10;
#10;
#10;
#10;
```

```
#10;
#10 rready = 0;
#10;
#10 res_n = 1;
#10 awvalid = 1;awburst = 2'b01;awsize =3'b001;awlen=4'b0011;awaddr = 5'b00000;
#10;
#10 awvalid = 0;awburst = 2'b00;awsize = 3'b000;awlen = 4'b0000;awaddr=5'b00000;
#10 wvalid = 1;wdata = 16'hff11;
#10;
#10 wdata = 16'h11aa;
#10;
#10 wdata = 16'h0011;
#10;
#10 wdata = 16'h1110;wlast = 1;
#10;
#10 wlast = 0;
#10 bready = 1;
#10 bready = 0;
#10;
#10 res_n=1;
#10 arvalid = 1;arburst = 2'b01;arsize = 3'b001;arlen = 4'b0011;araddr=5'b00000;
#10;
#10 arvalid = 0;arburst = 2'b00;arsize = 3'b000;arlen = 4'b0000;araddr=5'b00000;
#10 rready = 1;
#10;
#10;
#10;
#10;
#10;
#10;
#10;
#10;
#10;
#10 rready = 0;
#10;
 $finish;
end
endmodule
```

6. AXI architecture

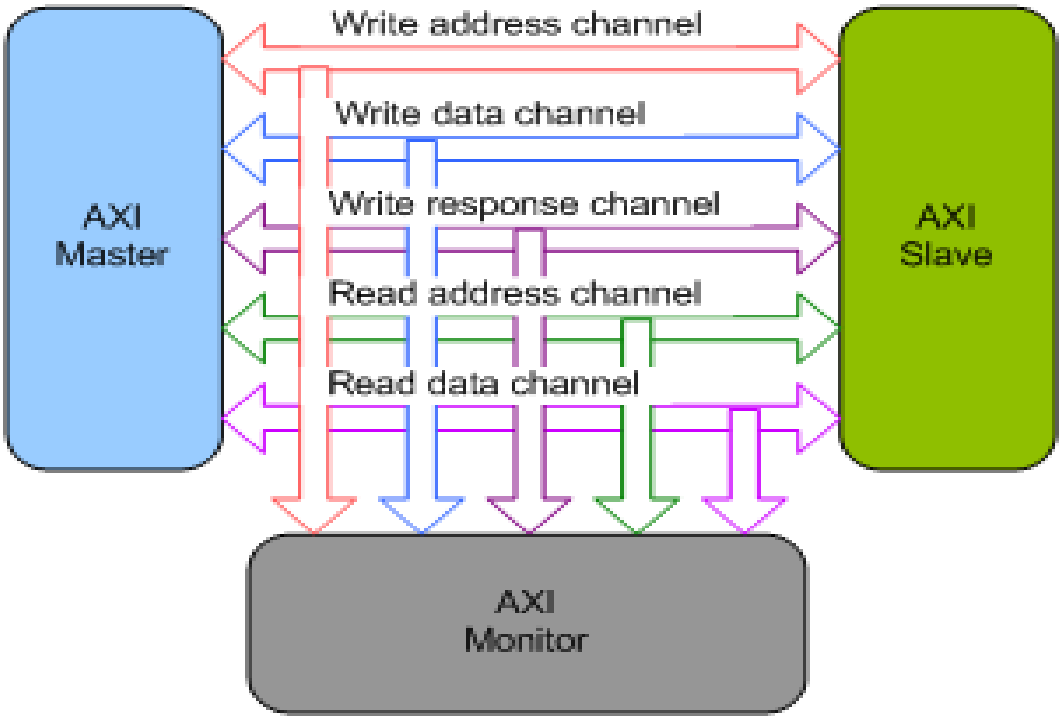
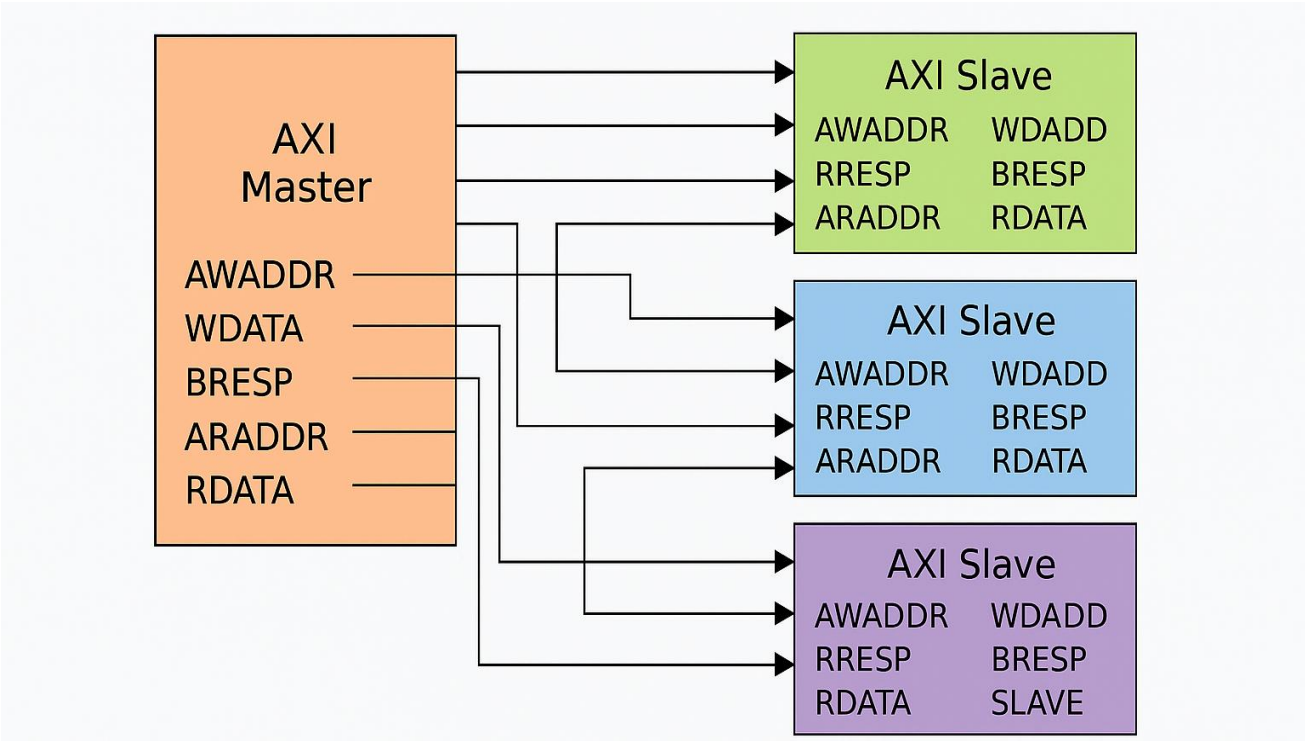


Figure 3.4.2: AXI protocol architecture



6. AXI Data Transfer – Working Flow (Read or Write Request from Master to Slave)

Step-by-Step Data Transfer Process

- Step 1: **Address Phase (Read or Write Request)**
  The **AXI Master** sends a read **(ar*)** or write **(aw*)** address request. The slave sees **ar_valid** or **aw_valid** high and responds with **ar_ready** or **aw_ready**.
- Step 2: **Write Data Phase (Write Only)**
  For write transactions, the master begins sending data on the **w_data** bus. **w_valid** is asserted by the master, and the slave acknowledges with **w_ready**.
- Step 3: **Read Data Phase (Read Only)**
  For read transactions, the slave places data on **r_data**, asserts **r_valid**, and waits for **r_ready** from the master. It sends multiple beats if **ar_len** > 0 using burst type rules (e.g., INCR).
- Step 4**: Response Phase**
  For writes, after receiving all write data and **w_last**, the slave asserts:
  - **b_valid** to indicate a write response is ready.
  - **b_resp** to communicate OKAY or error status.

  For reads, it asserts:
  - **r_resp** (response status per read beat)
  - **r_valid** for each beat.
- Step 5: **Transaction Completion**
  Both The **transaction ends** when the master acknowledges the response:

  For write: **b_ready** high → slave clears **b_valid**

  For read: **r_ready** high when **r_valid** is high → slave clears **r_valid**.

  Slave resets internal flags or moves to the IDLE state.

Chapter 5

RESULTS & CONCLUSION

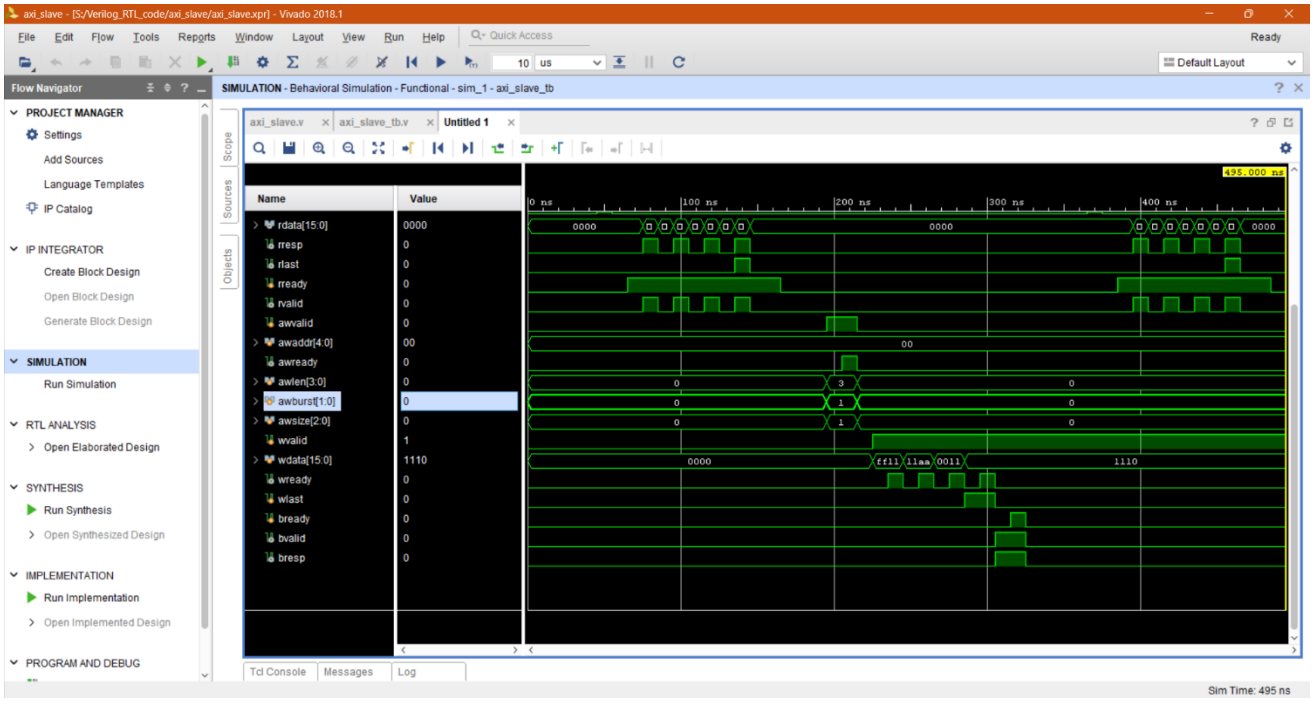


Figure 5.1: Output Simulation of AXI Protocol (Read operation)

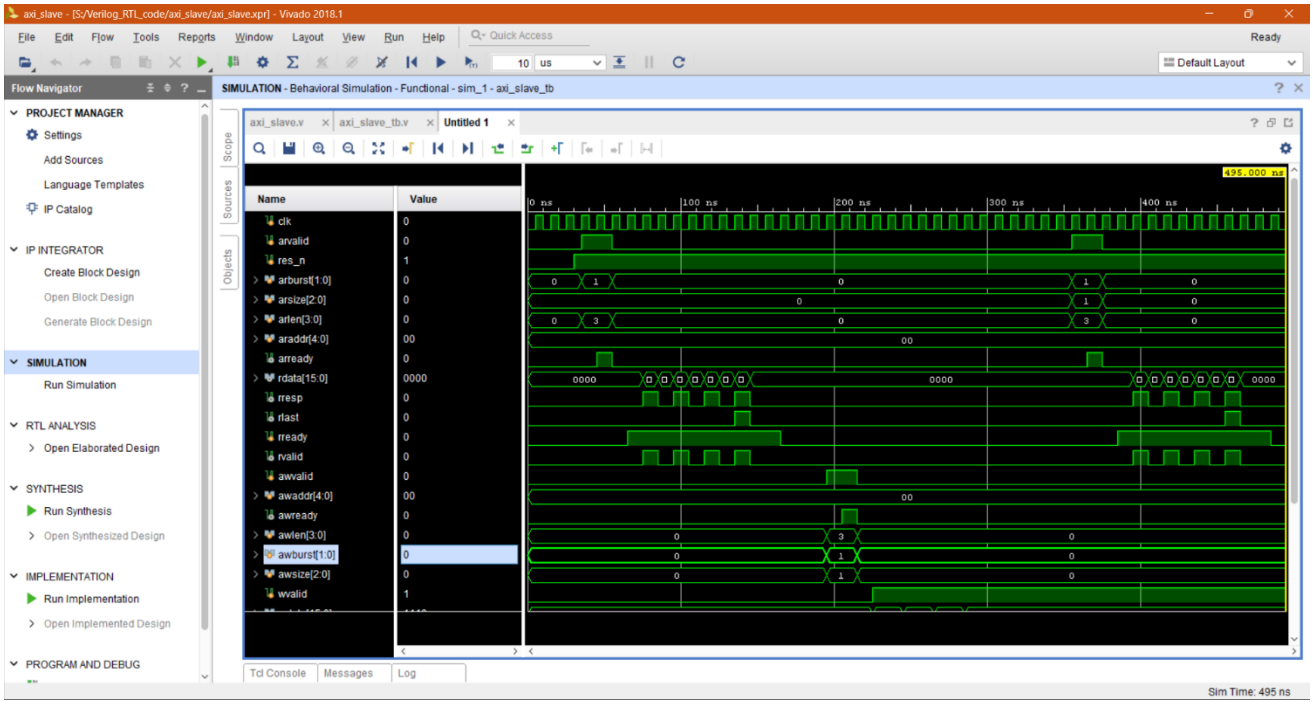


Figure 5.2: Output Simulation of AXI Protocol (Write operation)

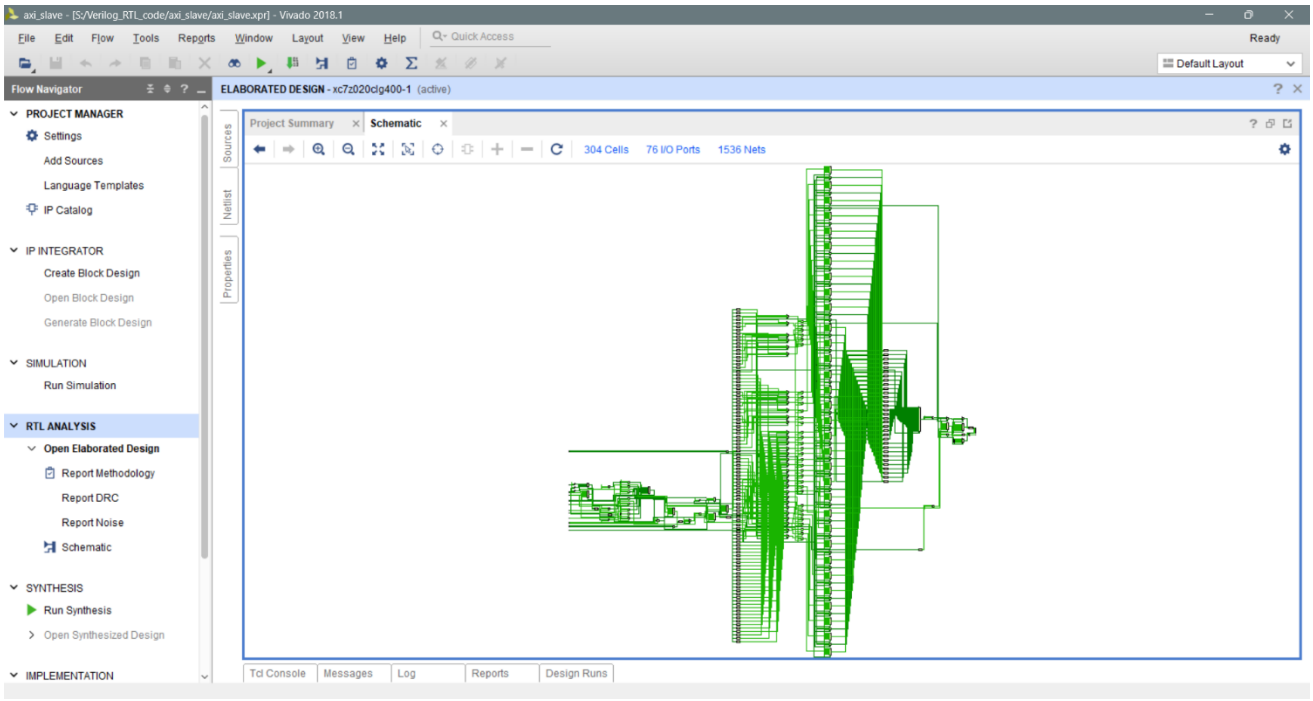


Figure 5.3: Eloberated Schematic Design of AXI

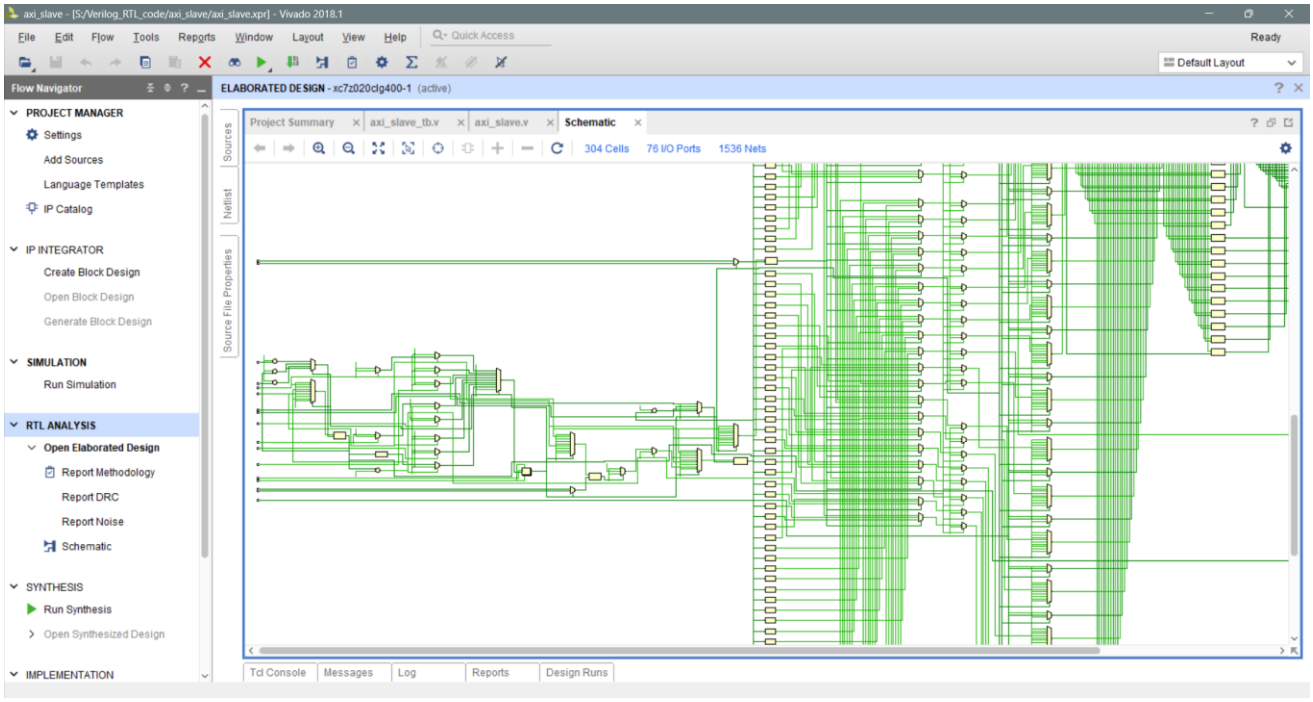


Figure 5.4: Eloberated Schematic Design of AXI