Design and Verification of AXI4-Lite Slave Interface Protocol using System Verilog

(by CH. ANIRUDH)

1. Introduction

The **AXI4-Lite** (Advanced Extensible Interface - Lite) is a subset of the **AMBA** AXI4 protocol designed for simple, low-bandwidth memory-mapped communication in modern **SoC** (System-on-Chip) **architectures**. It provides a lightweight, single transaction, register-based **communication** interface, making it ideal for scenarios where **complex** burst transactions are not required. AXI4-Lite is commonly used in **peripheral** control registers, **embedded systems**, and low-speed data transfers within larger AXI-based **interconnects**.

Unlike the full AXI4 protocol, AXI4-Lite supports only a single **beat** per transaction, meaning no burst **operations** are allowed. This simplicity reduces the complexity of **slave** peripherals, making it a preferred choice for memory-mapped slave devices in **FPGA** and **ASIC** designs. The **protocol** also ensures **independent** read and write channels, allowing simultaneous transactions while maintaining efficient control over the **bus**.

This project focuses on the design and implementation of an AXI4-Lite Slave module using Verilog/System Verilog. The implementation includes **RTL** coding, **verification**, **simulation**, and **synthesis** to ensure **functional** correctness and **optimized** resource **utilization**. A structured testbench is also developed to verify the protocol compliance, making use of System Verilog verification techniques such as **interfaces**, **transactions**, and **synchronization** mechanisms.

2. AXI4-Lite Protocol Overview

The AXI4-Lite protocol is a simplified version of AXI4, providing a **straightforward** interface for register-mapped accesses. It is mainly used for accessing control registers and simple peripherals where **high**-performance burst transactions are not needed. The protocol divides **communication** into independent read and write channels, ensuring data **integrity** and **pipeline** efficiency.

Key Features of AXI4-Lite:

- Simple address-based read and write transactions
- No burst support (single data transaction per request)
- Separate read and write address/data channels
- Uses a **handshaking** mechanism (VALID & READY signals)
- Ideal for low-power and low-bandwidth applications

Each transaction follows a request-response mechanism, where the master initiates the request, and the slave responds accordingly. The interface consists of five main channels:

1. Address Channel – Transfers the write address from the Write master to the slave.
2. Write Data Channel – Transfers the data to be written.
3. Write Response Channel – Sends an acknowledgment for the write transaction.

4. Read Address Channel – Transfers the read address to the slave.
5. Read Data Channel – Sends the requested data to the master.

Unlike AXI4, where burst transactions are supported, AXI4-Lite ensures only a single address and data transfer per transaction, reducing the complexity of the protocol. This makes it ideal for memory-mapped control registers and peripheral interfaces such as **GPIOs**, **UART** controllers, and timers.

Another critical feature of AXI4-Lite is its valid-ready **handshake mechanism**, which ensures proper synchronization between the master and the slave. Each transaction occurs only when both **VALID** and **READY** signals are asserted, ensuring data integrity. This feature helps in designing **robust**, low-latency communication between **master** and **slave** components.

This protocol is widely used in FPGA-based SoCs, embedded processors, and custom ASIC implementations due to its simplicity and efficiency. It enables designers to implement low-overhead interconnects while maintaining **compatibility** with AXI-based interconnect architectures.

3. AXI4-Lite Design Implementation

The design of the AXI4-Lite Slave module follows a modular approach, ensuring proper **functionality** and ease of **verification**. The implementation involves developing register-based memory mapping, where an AXI4-Lite Master can read from and **write** to **registers** within the slave module.

Design Approach:

- The AXI4-Lite Slave receives read and write requests from the master and responds accordingly.
- It includes address decoding logic, ensuring correct data transfer to the requested register location.
- The module ensures synchronization between control signals (VALID, READY, RVALID, WVALID).
- Read/write operations are implemented using combinational and sequential logic to ensure accurate timing and data integrity.

The RTL implementation consists of:

- Write Address Decoder – Captures the write address and control signals.
- Write Data Handler – Stores the received data into internal registers.
- Read Address Handler – Fetches the requested data from registers.
- Read Data Response – Sends the appropriate response to the master.

In this design, FSM (Finite State Machine) logic is not used, as the implementation follows a combinational and sequential logic-based approach to manage data transfers efficiently. The design is implemented in Verilog/System Verilog, ensuring modularity and reusability.

Fig(a): AXI4-Lite Slave

RTL CODE:

```
module axi4_lite_slave #(
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 32
)(
    input logic ACLK,
    input logic ARESETn,

    // Write Address Channel
    input logic [ADDR_WIDTH-1:0] AWADDR,
```

```
    input logic AWVALID,
    output logic AWREADY,

    // Write Data Channel
    input logic [DATA_WIDTH-1:0] WDATA,
    input logic [DATA_WIDTH/8-1:0] WSTRB,
    input logic WVALID,
    output logic WREADY,

    // Write Response Channel
    output logic [1:0] BRESP,
    output logic BVALID,
    input logic BREADY,

    // Read Address Channel
    input logic [ADDR_WIDTH-1:0] ARADDR,
    input logic ARVALID,
    output logic ARREADY,

    // Read Data Channel
    output logic [DATA_WIDTH-1:0] RDATA,
    output logic [1:0] RRESP,
    output logic RVALID,
    input logic RREADY
);

    // Internal registers
    logic [DATA_WIDTH-1:0] mem [0:255]; // 256 x 32-bit memory
    logic [1:0] resp;

    initial begin
    for (int i = 0; i < 256; i++) begin
        mem[i] = 32'h0000_0000;
```

```
    end
  end

    // Write Address Channel
    always_ff @(posedge ACLK or negedge ARESETn) begin
        if (!ARESETn) begin
            AWREADY <= 1'b0;
        end else if (AWVALID && !AWREADY) begin
            AWREADY <= 1'b1;
        end else begin
            AWREADY <= 1'b0;
        end
    end

    // Write Data Channel
    always_ff @(posedge ACLK or negedge ARESETn) begin
        if (!ARESETn) begin
            WREADY <= 1'b0;
        end else if (WVALID && !WREADY) begin
            WREADY <= 1'b1;
            // Write data to memory
            for (int i = 0; i < DATA_WIDTH/8; i++) begin
                if (WSTRB[i]) begin
                    mem[AWADDR][i*8 +: 8] <= WDATA[i*8 +: 8];
                end
            end
             $display("Memory Write: Addr = %h, Data = %h", AWADDR, WDATA);
        end else begin
            WREADY <= 1'b0;
        end
    end

    // Write Response Channel
```

```
  always_ff @(posedge ACLK or negedge ARESETn) begin
    if (!ARESETn) begin
        BVALID <= 1'b0;
        BRESP <= 2'b00; // OKAY response
    end else if (WREADY && WVALID) begin
        BVALID <= 1'b1;
    end else if (BREADY && BVALID) begin
        BVALID <= 1'b0;
    end
end

// Read Address Channel
always_ff @(posedge ACLK or negedge ARESETn) begin
    if (!ARESETn) begin
        ARREADY <= 1'b0;
    end else if (ARVALID && !ARREADY) begin
        ARREADY <= 1'b1;
    end else begin
        ARREADY <= 1'b0;
    end
end

// Read Data Channel
always_ff @(posedge ACLK or negedge ARESETn) begin
    if (!ARESETn) begin
        RVALID <= 1'b0;
        RDATA <= '0;
        RRESP <= 2'b00; // OKAY response
    end else if (ARREADY && ARVALID) begin
        RVALID <= 1'b1;
        RDATA <= mem[ARADDR];
        $display("Memory Read: Addr = %h, Data = %h", ARADDR, mem[ARADDR]);
    end else if (RREADY && RVALID) begin
```

```
        RVALID <= 1'b0;
      end
    end

endmodule
```

Testbench:

```
interface axi4_lite_if (input logic ACLK, input logic ARESETn);
    logic [31:0] AWADDR;
    logic AWVALID;
    logic AWREADY;

    logic [31:0] WDATA;
    logic [3:0] WSTRB;
    logic WVALID;
    logic WREADY;

    logic [1:0] BRESP;
    logic BVALID;
    logic BREADY;

    logic [31:0] ARADDR;
    logic ARVALID;
    logic ARREADY;

      logic [31:0] RDATA;
    logic [1:0] RRESP;
    logic RVALID;
```

```
    logic RREADY;
endinterface


class transaction;
    rand logic [31:0] addr;
    rand logic [31:0] data;
    rand logic [3:0] strb;
    rand logic is_write;
endclass

class driver;
    mailbox drv_mbx;
    virtual axi4_lite_if vif;
task run();
    forever begin
        transaction tr;
        drv_mbx.get(tr);
        if (tr.is_write) begin
            $display("Driver: Write Transaction. Addr = %h, Data = %h", tr.addr, tr.data);
            vif.AWADDR <= tr.addr;
            vif.AWVALID <= 1'b1;
            vif.WDATA <= tr.data;
            vif.WSTRB <= tr.strb;
            vif.WVALID <= 1'b1;
            @(posedge vif.ACLK);
            while (!vif.AWREADY || !vif.WREADY) @(posedge vif.ACLK);
            vif.AWVALID <= 1'b0;
            vif.WVALID <= 1'b0;
            @(posedge vif.ACLK);
            while (!vif.BVALID) @(posedge vif.ACLK);
            vif.BREADY <= 1'b1;
            @(posedge vif.ACLK);
```

```
            vif.BREADY <= 1'b0;
        end else begin
            $display("Driver: Read Transaction. Addr = %h", tr.addr);
            vif.ARADDR <= tr.addr;
            vif.ARVALID <= 1'b1;
            @(posedge vif.ACLK);
            while (!vif.ARREADY) @(posedge vif.ACLK);
            vif.ARVALID <= 1'b0;
            @(posedge vif.ACLK);
            while (!vif.RVALID) @(posedge vif.ACLK);
            vif.RREADY <= 1'b1;
            @(posedge vif.ACLK);
            vif.RREADY <= 1'b0;
        end
    end
endtask

endclass

class monitor;
    mailbox mon_mbx;
    virtual axi4_lite_if vif;

    task run();
        forever begin
            transaction tr;
            @(posedge vif.ACLK);
            if (vif.AWVALID && vif.AWREADY) begin
                tr = new();
                tr.addr = vif.AWADDR;
                tr.is_write = 1'b1;
                mon_mbx.put(tr);
            end
```

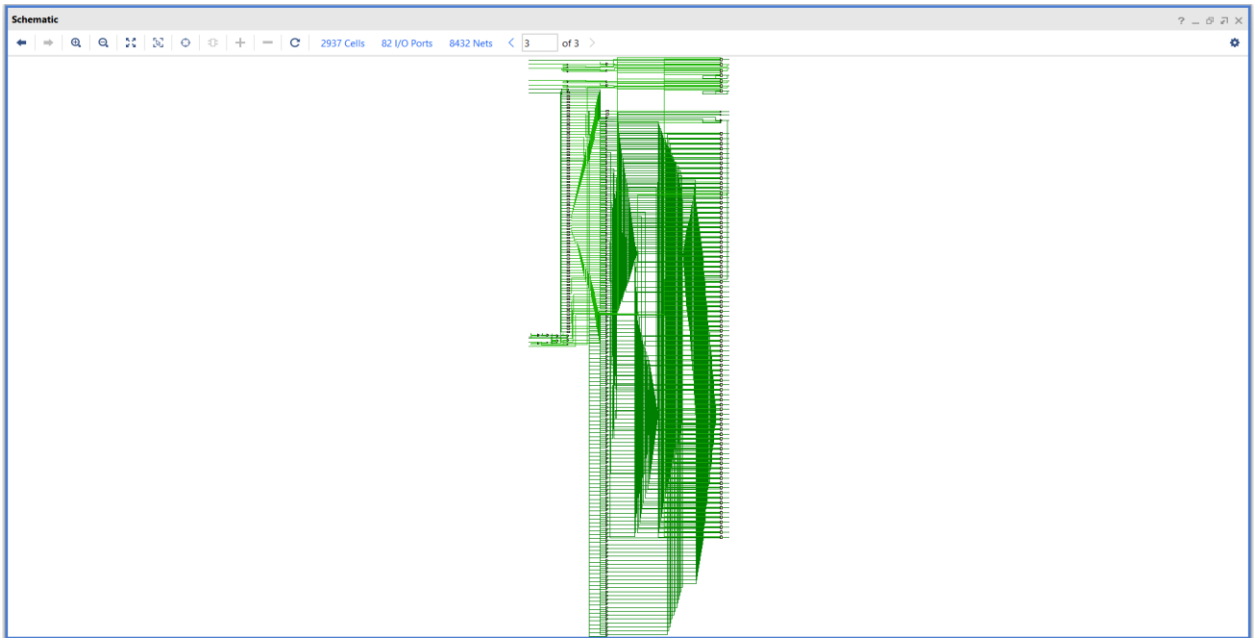
```
        if (vif.ARVALID && vif.ARREADY) begin
          tr = new();
          tr.addr = vif.ARADDR;
          tr.is_write = 1'b0;
          mon_mbx.put(tr);
        end
      end
    endtask
endclass

class scoreboard;
    mailbox scb_mbx;
    transaction tr;

    task run();
      forever begin
        scb_mbx.get(tr);
        if (tr.is_write) begin
          $display("Write Transaction: Addr = %h, Data = %h", tr.addr, tr.data);
        end else begin
          $display("Read Transaction: Addr = %h", tr.addr);
        end
      end
    endtask
endclass

module tb;
    logic ACLK;
    logic ARESETn;

    axi4_lite_if vif(ACLK, ARESETn);

    axi4_lite_slave dut (
```

```
    .ACLK(ACLK),
    .ARESETn(ARESETn),
    .AWADDR(vif.AWADDR),
    .AWVALID(vif.AWVALID),
    .AWREADY(vif.AWREADY),
    .WDATA(vif.WDATA),
    .WSTRB(vif.WSTRB),
    .WVALID(vif.WVALID),
    .WREADY(vif.WREADY),
    .BRESP(vif.BRESP),
    .BVALID(vif.BVALID),
    .BREADY(vif.BREADY),
    .ARADDR(vif.ARADDR),
    .ARVALID(vif.ARVALID),
    .ARREADY(vif.ARREADY),
    .RDATA(vif.RDATA),
    .RRESP(vif.RRESP),
    .RVALID(vif.RVALID),
    .RREADY(vif.RREADY)
);

initial begin
    ACLK = 0;
    forever #5 ACLK = ~ACLK;
end

initial begin
    ARESETn = 0;
    #20;
    ARESETn = 1;
end

initial begin
```

```
    driver drv;
    monitor mon;
    scoreboard scb;
    mailbox drv_mbx, mon_mbx, scb_mbx;

    drv = new();
    mon = new();
    scb = new();

    drv_mbx = new();
    mon_mbx = new();
    scb_mbx = new();

    drv.drv_mbx = drv_mbx;
    mon.mon_mbx = mon_mbx;
    scb.scb_mbx = scb_mbx;

    drv.vif = vif;
    mon.vif = vif;
    // Generate transactions and send to driver
  fork
    begin
      transaction tr;
      tr = new();
      tr.addr = 32'h0000_1000; // Example address
      tr.data = 32'h1234_5678; // Example data
      tr.strb = 4'b1111;     // All bytes valid
      tr.is_write = 1'b1;    // Write transaction
      drv_mbx.put(tr);

      #100; // Wait for write to complete

      tr = new();
```

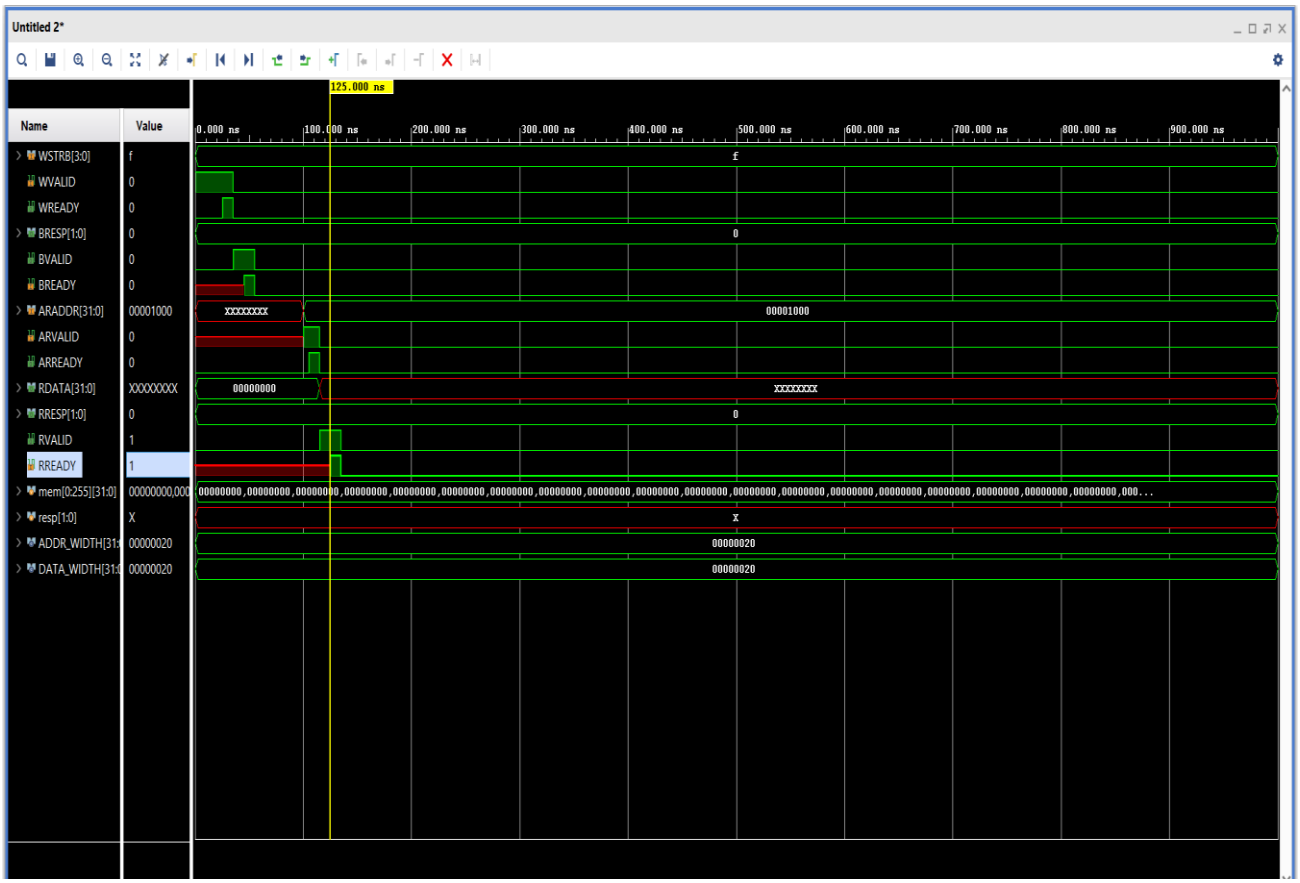
```
        tr.addr = 32'h0000_1000; // Example address
        tr.is_write = 1'b0;      // Read transaction
        drv_mbx.put(tr);
      end



        drv.run();
        mon.run();
        scb.run();
      join


  end
endmodule
```

Elaborated Design:



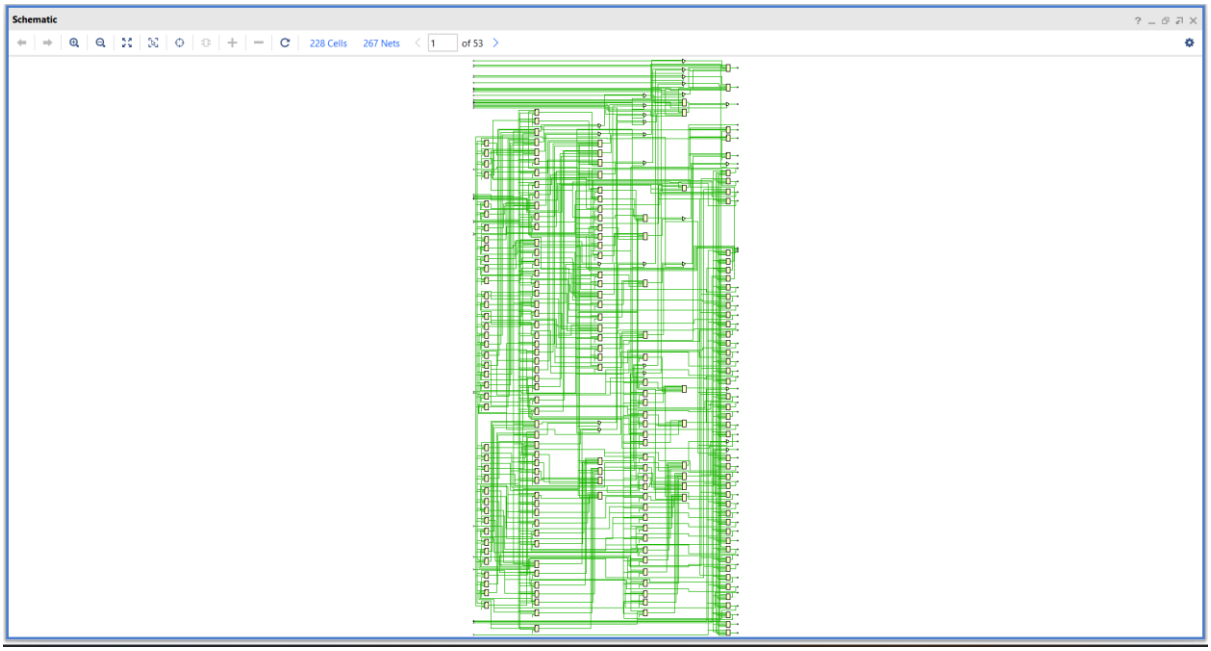
Fig(b): AXI4-Lite Elaborated design

Simulation:



Fig(b): AXI4-Lite Simulation

Synthesis:



Fig(c): AXI4-Lite Synthesis

4. Advantages and Disadvantages

Advantages of AXI4-Lite:

- Simple Implementation – No burst support, reducing complexity.
- Efficient Resource Utilization – Uses fewer logic gates compared to full AXI4.
- Independent Read/Write Channels – Enables concurrent transactions.
- Handshaking Mechanism – Ensures data integrity with VALID and READY signals.
- Scalability – Easily integrated into AXI-based SoCs.

Disadvantages of AXI4-Lite:

- No Burst Transactions – Only supports single data transactions, limiting bandwidth.
- Higher Latency for Large Transfers – Since it doesn't support bursts, multiple transactions take longer.
- Limited Throughput – Designed for low-speed control applications rather than high-speed data transfer.

Despite these limitations, AXI4-Lite is widely used in embedded systems and peripheral control applications due to its ease of use and efficient design.

5. Applications of AXI4-Lite

The AXI4-Lite protocol is used in various embedded systems, FPGA designs, and ASIC implementations where simple register-based communication is required. Some common applications include:

- Peripheral Interfaces – Used for GPIO, UART, SPI, I2C controllers in SoC designs.
- Memory-Mapped Registers – Controls configuration registers in embedded devices.
- Low-Speed Data Transfer – Suitable for slow, memory-mapped transactions in processors.
- Embedded Processors – Implemented in ARM-based SoCs for control registers.
- FPGA-Based System Designs – Utilized in Xilinx & Intel FPGAs for processor-to-peripheral communication.

AXI4-Lite is a critical component of modern SoC architectures, ensuring efficient register-based communication while maintaining compatibility with the AMBA AXI ecosystem.

6. Conclusion

This project successfully implements an AXI4-Lite Slave module using Verilog/SystemVerilog, along with testbench verification, simulation, and synthesis. The structured design approach ensures efficient memory-mapped communication between the master and the slave.
Future enhancements can include:

- Extending the design to AXI4 with burst transactions
- Implementing an AXI4-Lite Master module
- Enhancing verification with UVM-based testbenches

---- The End----