Indian Institute of Technology Bombay

EE 705 VLSI Design Lab

Project Part B

Final Integration of RISCV

Course Instructor:

Prof. Laxmeesha Somappa

Team Members:

Algorithm Avengers	Silicon Syndicate	RAF
Srinivas Nagireddy (24M1159) Shanmukhi Ganesh Sai (24M1158) Harisingh Nenavath (24M1133) Ritesh Kumar (24M1160)	Bala Murugan S (24M1173) Lakshmideep Chowdary M (24M1150) Juhi Bharti (24M1144)	Sumit Pal (24M1192)

1 Final Integration of RISCV

1.1 Introduction

This project involves building a RISC-V processor by integrating multiple Verilog modules to form a functional CPU. The key components include:

- riscv_core – The main CPU core that fetches and executes RISC-V instructions.
- riscv_top – The top-level wrapper that connects all major components.
- riscv_soc - The very topmost module having core, soc, arb_conv.
- soc.v – The system-on-chip module, integrating the core with memory, peripherals, and buses.
- axi4lite_axi4_conv.v – A bus converter that bridges AXI4-Lite and full AXI4 for efficient data transfers.
- irq_ctrl.v – The interrupt controller, managing external events and prioritizing CPU alerts.

By integrating these modules, we create a working RISC-V CPU that can run programs and interact with external devices. This report explains how each block works.

1.2 Sources of different blocks integrated

AXI4 modules (`axi4_arb, axi4_retime and axi4_lite_tap`) - Provided by TAs

SOC Components	**RISC-V Core Components**
UART - Silicon Syndicate	ALU+DECODE - Algorithm Avengers
SPI - RAM_705	FETCH - Panchajanya
GPIO - RISCCEY	CSR - Bitstream Builders
Timer - Team Vajra	LSU - Electric_Elite
IRQ_CTRL - Provided by TAs	EXEC - Grp123

Table 1: SoC and RISCV_Core Components

2 The integration of RISCV_CORE components

2.1 Module Descriptions

CSR (Control and Status Registers)

The Control and Status Register module implements the RISC-V architecture specification for processor state management. It handles all machine-mode CSRs including `mstatus` for interrupt control, `mepc` for exception program counters, `mcause` for trap reasons, and `mtvec` for trap vector addressing. The module supports CSR instructions like CSRRW (read-write), CSRRS (read-set), and CSRRC (read-clear), enabling software to manipulate these registers and manages interrupts through the `intr_i` input. When exceptions occur, it updates status registers and triggers trap handling via `branch_csr_request_o`. The CSR also supports the `MRET` instruction for returning from exceptions.

Fetch Unit

The module instantiates the riscv_Fetch submodule to handle instruction fetching from the cache. It handles branching, flushing, or invalidating of the instruction cache. It provides valid instruction and updates program counter.

Decode Unit

The decode stage processes all RV32I instruction formats (R/I/S/B/U/J), extracting opcodes, register indices, and immediate values. It generates control signals for the execution pipelines and manages operand forwarding from register files or previous results. The unit coordinates pipeline flow by handling stalls from busy functional units and branches from executed instructions. It maintains execution correctness by properly routing writeback results to the register file.

ALU (Arithmetic Logic Unit)

The ALU performs all arithmetic and logical operations for the RISC-V core, executing instructions like ADD, SUB, AND, OR, and comparisons. It takes operand values from the decoder and generates results with status flags. The unit supports both register-register and immediate operations, with proper sign-extension for immediate values. Its outputs feed directly into the EXEC stage for further processing.

EXEC (Execute) Module

The EXEC stage manages instruction execution flow, handling branches, jumps, and result forwarding. It processes ALU outputs, makes branch decisions (BEQ, BNE, etc.), and controls pipeline behavior. The module generates writeback signals for register

updates and stall signals when necessary. It coordinates with the decode stage to maintain correct program flow during branches and exceptions.

LSU (Load/Store Unit)

The Load-Store Unit executes all RV32I memory operations including byte, halfword and word accesses. It supports instructions like LW (load word), LH (load halfword), LB (load byte), SW (store word), SH (store halfword), and SB (store byte). It calculates effective addresses by combining register operands with instruction offsets, while verifying proper alignment for each access type. The LSU interfaces with memory through dedicated address and data signals `(mem_addr_o, mem_data_wr_o)` and reports misaligned accesses as exceptions. For multi-cycle memory operations, it manages pipeline stalls to maintain data consistency.

2.2 Testbench Code for riscv_core.v

The testbench verifies the `riscv_core.v` by sequentially executing different instruction types (arithmetic, branches, jumps, load/store) with proper timing. It initializes the core with a NOP instruction, then tests CSR operations, ALU functions, and memory operations while monitoring memory interfaces and control signals.

```
// Reset sequence
#30 rst_i = 1;
#10 mem_i_accept_i = 1; mem_i_valid_i = 1;
// Test different instructions with mem_i_inst_i

// immediate instructions
#20 mem_i_inst_i = {12'd2,5'd1,3'd0,5'd10,7'b0010011}; // ADDI instruction with imm = 2
// Load instruction
#20 mem_i_inst_i = {7'd10,5'd4,3'd0,5'd3,7'b0000011}; // LH with Imm 10, and R3 + 10 Address to load in R4
mem_d_accept_i = 1;
mem_d_data_rd_i = 32'd80;
mem_d_resp_tag_i = 11'h084;
mem_d_ack_i = 1'b1;
#80;


// ADD instruction
#20 mem_i_inst_i = {7'h00,5'd10,5'd1,3'd0,5'd10,7'b0110011}; // ADD instruction

#20 mem_i_inst_i = {20'd2,5'd10, 7'b0110111}; // LUI instruction with imm = 2

// branch instructions
#20 mem_i_inst_i = {7'h00,5'd2,5'd1,3'd0,4'b1,1'b0,7'b1100011}; // BEQ instruction with imm = 8192, {opcode_opcode_i[31], opcode_opcode_i[7], opcode_opcode_
```

Figure 1: Main Snippet of the Testbench

2.3 Post Synthesis Timing Simulation

The following image shows the output waveform of the above testbench code after the design is synthesized. The following three images shows the ouptut signals observed from differentt blocks such as `DECODE`, `EXEC` and `LSU`.

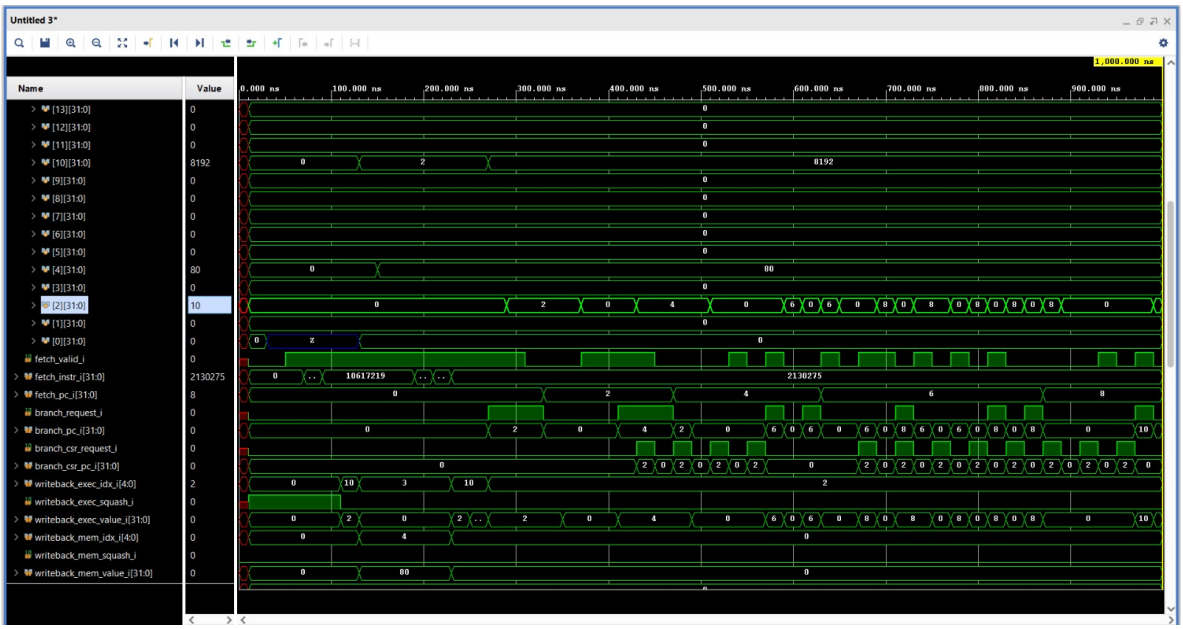

Figure 2: Output signals of DECODE Block

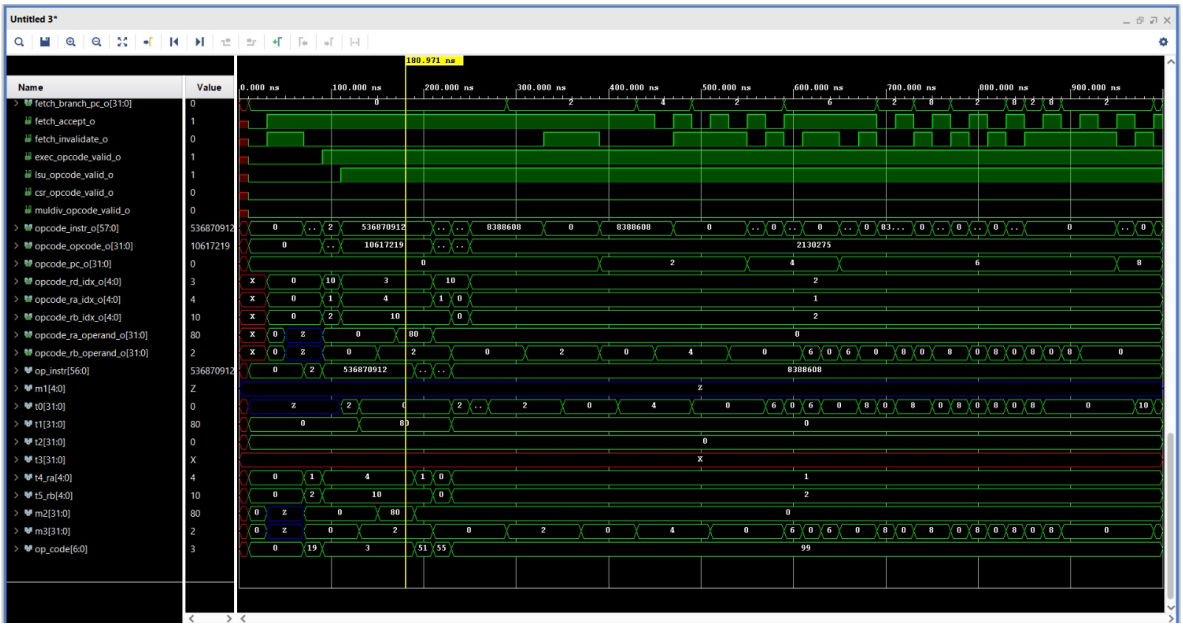

Figure 3: Output signals of EXEC Block

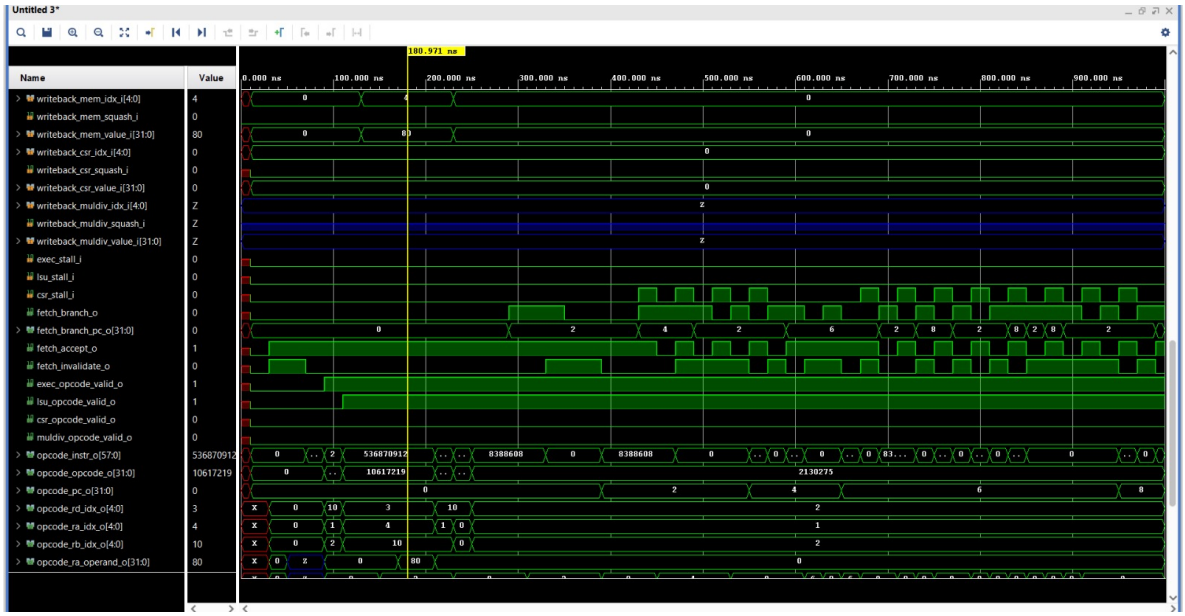

Figure 4: Output signals of LSU Block

3 The integration of SOC components

Peripheral 0: Interrupt Controller

The central interrupt manager handles hardware interrupts from all peripherals through four prioritized channels. It implements an AXI4-Lite interface for configuration, featuring individual interrupt masking (IER), status monitoring (ISR/IPR), and acknowledgment (IAR) registers. The controller resolves concurrent interrupts using fixed hardware priority (channel 0 highest) and provides vector address reporting (IVR) for efficient CPU servicing. A master enable bit (MER) allows global interrupt control.

Peripheral 1: Timer

This programmable 32-bit timer module offers precise timing functions through a free-running counter and programmable comparator. Accessed via AXI4-Lite, it generates periodic interrupts for real-time scheduling and timeouts. The implementation includes automatic reload capability and preserves timing accuracy during CPU sleep states through clock domain synchronization.

Peripheral 2: UART Lite

Supporting asynchronous serial communication, this peripheral features 16-byte deep FIFOs for both transmission and reception. Its AXI4-Lite interface allows baud rate configuration, status monitoring, and interrupt setup. The module automatically handles start/stop bits and includes parity error detection, triggering interrupts on data events (RX full/TX empty) and line status changes.

Peripheral 3: SPI Controller

Operating in master mode, this SPI interface connects to flash memory and sensors with configurable clock characteristics (CPOL/CPHA). The AXI4-Lite accessible control registers manage chip select lines (SSR), data transfer (DTR/DRR), and interrupt generation (IPISR). Support includes programmable clock prescaling and dual-buffered data registers for continuous transfers.

Peripheral 4: GPIO

The 32-bit general-purpose I/O module provides pin-level control through AXI4-Lite accessible registers. Each bit supports independent direction configuration (gpio_direction_q), with atomic set/clear operations (output_set_q/output_clr_q) for output manipulation. The input change detector generates maskable interrupts (int_mask_q) with configurable trigger conditions (int_mode_q/int_level_q), while status registers (gpio_input_q) reflect real-time pin states.

Retime Block

Whenever the Lite Tap sees the peripheral address other than the valid peripheral addresses available, the AXI Full signals are routed to the retime block which fetches the Instructions from the BRAM. It takes a block of data from BRAM and stores in a buffer which is then routed back one by one to the ICACHE block through AXI LITE TAP and AXI ARBITER.

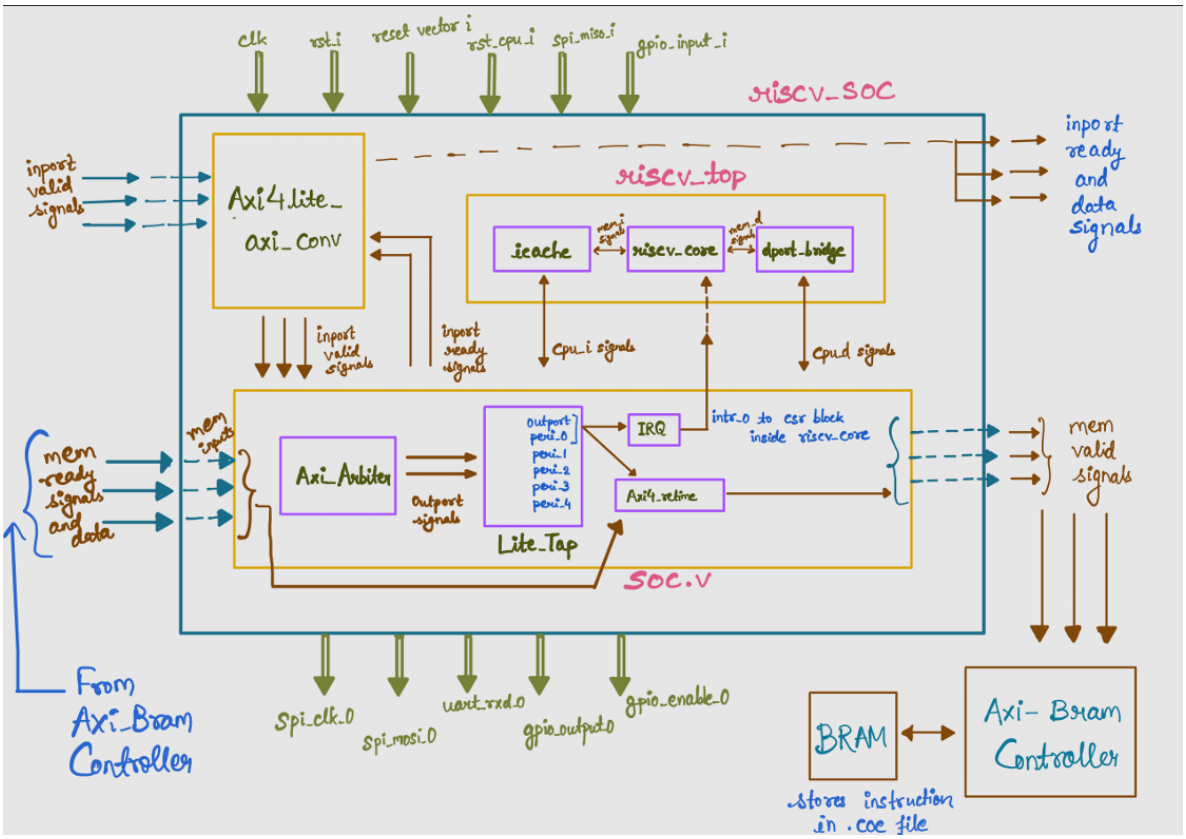

Figure 5: Block Diagram for final integration

3.1 Testbench code for Integration of SOC components

The testbench successfully validated integration of all peripherals (Periph0-Periph4) through AXI4-Lite transactions. All components demonstrated correct register access patterns and interrupt signaling. Post-synthesis timing simulations confirmed stable operation.

```
`timescale 1ns/1ps
`include "GPIORegDef.v"

module soc_tb;

    // Inputs
```

```
    reg             clk_i;
    reg             rst_i;
    reg             inport_awvalid_i;
    reg   [31:0]    inport_awaddr_i;
    reg   [3:0]     inport_awid_i;
    reg   [7:0]     inport_awlen_i;
    reg   [1:0]     inport_awburst_i;
    reg             inport_wvalid_i;
    reg   [31:0]    inport_wdata_i;
    reg   [3:0]     inport_wstrb_i;
    reg             inport_wlast_i;
    reg             inport_bready_i;
    reg             inport_arvalid_i;
    reg   [31:0]    inport_araddr_i;
    reg   [3:0]     inport_arid_i;
    reg   [7:0]     inport_arlen_i;
    reg   [1:0]     inport_arburst_i;
    reg             inport_rready_i;
    reg             mem_awready_i;
    reg             mem_wready_i;
    reg             mem_bvalid_i;
    reg   [1:0]     mem_bresp_i;
    reg   [3:0]     mem_bid_i;
    reg             mem_arready_i;
    reg             mem_rvalid_i;
    reg   [31:0]    mem_rdata_i;
    reg   [1:0]     mem_rresp_i;
    reg   [3:0]     mem_rid_i;
    reg             mem_rlast_i;
    reg             cpu_i_awvalid_i;
    reg   [31:0]    cpu_i_awaddr_i;
    reg   [3:0]     cpu_i_awid_i;
    reg   [7:0]     cpu_i_awlen_i;
    reg   [1:0]     cpu_i_awburst_i;
    reg             cpu_i_wvalid_i;
    reg   [31:0]    cpu_i_wdata_i;
    reg   [3:0]     cpu_i_wstrb_i;
    reg             cpu_i_wlast_i;
    reg             cpu_i_bready_i;
    reg             cpu_i_arvalid_i;
    reg   [31:0]    cpu_i_araddr_i;
    reg   [3:0]     cpu_i_arid_i;
    reg   [7:0]     cpu_i_arlen_i;
    reg   [1:0]     cpu_i_arburst_i;
    reg             cpu_i_rready_i;
    reg             cpu_d_awvalid_i;
    reg   [31:0]    cpu_d_awaddr_i;
    reg   [3:0]     cpu_d_awid_i;
    reg   [7:0]     cpu_d_awlen_i;
    reg   [1:0]     cpu_d_awburst_i;
    reg             cpu_d_wvalid_i;
    reg   [31:0]    cpu_d_wdata_i;
```

```
    reg  [3:0]    cpu_d_wstrb_i;
    reg           cpu_d_wlast_i;
    reg           cpu_d_bready_i;
    reg           cpu_d_arvalid_i;
    reg  [31:0]   cpu_d_araddr_i;
    reg  [3:0]    cpu_d_arid_i;
    reg  [7:0]    cpu_d_arlen_i;
    reg  [1:0]    cpu_d_arburst_i;
    reg           cpu_d_rready_i;
    reg           spi_miso_i;
    reg           uart_txd_i;
    reg  [31:0]   gpio_input_i;

    // Outputs
    wire          intr_o;
    wire          inport_awready_o;
    wire          inport_wready_o;
    wire          inport_bvalid_o;
    wire [1:0]    inport_bresp_o;
    wire [3:0]    inport_bid_o;
    wire          inport_arready_o;
    wire          inport_rvalid_o;
    wire [31:0]   inport_rdata_o;
    wire [1:0]    inport_rresp_o;
    wire [3:0]    inport_rid_o;
    wire          inport_rlast_o;
    wire          mem_awvalid_o;
    wire [31:0]   mem_awaddr_o;
    wire [3:0]    mem_awid_o;
    wire [7:0]    mem_awlen_o;
    wire [1:0]    mem_awburst_o;
    wire          mem_wvalid_o;
    wire [31:0]   mem_wdata_o;
    wire [3:0]    mem_wstrb_o;
    wire          mem_wlast_o;
    wire          mem_bready_o;
    wire          mem_arvalid_o;
    wire [31:0]   mem_araddr_o;
    wire [3:0]    mem_arid_o;
    wire [7:0]    mem_arlen_o;
    wire [1:0]    mem_arburst_o;
    wire          mem_rready_o;
    wire          cpu_i_awready_o;
    wire          cpu_i_wready_o;
    wire          cpu_i_bvalid_o;
    wire [1:0]    cpu_i_bresp_o;
    wire [3:0]    cpu_i_bid_o;
    wire          cpu_i_arready_o;
    wire          cpu_i_rvalid_o;
    wire [31:0]   cpu_i_rdata_o;
    wire [1:0]    cpu_i_rresp_o;
    wire [3:0]    cpu_i_rid_o;
```

```
    wire            cpu_i_rlast_o;
    wire            cpu_d_awready_o;
    wire            cpu_d_wready_o;
    wire            cpu_d_bvalid_o;
    wire [1:0]      cpu_d_bresp_o;
    wire [3:0]      cpu_d_bid_o;
    wire            cpu_d_arready_o;
    wire            cpu_d_rvalid_o;
    wire [31:0]     cpu_d_rdata_o;
    wire [1:0]      cpu_d_rresp_o;
    wire [3:0]      cpu_d_rid_o;
    wire            cpu_d_rlast_o;
    wire            spi_clk_o;
    wire            spi_mosi_o;
    wire            spi_cs_o;
    wire            uart_rxd_o;
    wire [31:0]     gpio_output_o;
    wire [31:0]     gpio_output_enable_o;
    //register used to read data
    reg [31:0] read_data;

    // Instantiate the Unit Under Test (UUT)
    soc uut (
        // Inputs
        .clk_i(clk_i),
        .rst_i(rst_i),
        .inport_awvalid_i(inport_awvalid_i),
        .inport_awaddr_i(inport_awaddr_i),
        .inport_awid_i(inport_awid_i),
        .inport_awlen_i(inport_awlen_i),
        .inport_awburst_i(inport_awburst_i),
        .inport_wvalid_i(inport_wvalid_i),
        .inport_wdata_i(inport_wdata_i),
        .inport_wstrb_i(inport_wstrb_i),
        .inport_wlast_i(inport_wlast_i),
        .inport_bready_i(inport_bready_i),
        .inport_arvalid_i(inport_arvalid_i),
        .inport_araddr_i(inport_araddr_i),
        .inport_arid_i(inport_arid_i),
        .inport_arlen_i(inport_arlen_i),
        .inport_arburst_i(inport_arburst_i),
        .inport_rready_i(inport_rready_i),
        .mem_awready_i(mem_awready_i),
        .mem_wready_i(mem_wready_i),
        .mem_bvalid_i(mem_bvalid_i),
        .mem_bresp_i(mem_bresp_i),
        .mem_bid_i(mem_bid_i),
        .mem_arready_i(mem_arready_i),
        .mem_rvalid_i(mem_rvalid_i),
        .mem_rdata_i(mem_rdata_i),
        .mem_rresp_i(mem_rresp_i),
        .mem_rid_i(mem_rid_i),
```

```
        .mem_rlast_i(mem_rlast_i),
        .cpu_i_awvalid_i(cpu_i_awvalid_i),
        .cpu_i_awaddr_i(cpu_i_awaddr_i),
        .cpu_i_awid_i(cpu_i_awid_i),
        .cpu_i_awlen_i(cpu_i_awlen_i),
        .cpu_i_awburst_i(cpu_i_awburst_i),
        .cpu_i_wvalid_i(cpu_i_wvalid_i),
        .cpu_i_wdata_i(cpu_i_wdata_i),
        .cpu_i_wstrb_i(cpu_i_wstrb_i),
        .cpu_i_wlast_i(cpu_i_wlast_i),
        .cpu_i_bready_i(cpu_i_bready_i),
        .cpu_i_arvalid_i(cpu_i_arvalid_i),
        .cpu_i_araddr_i(cpu_i_araddr_i),
        .cpu_i_arid_i(cpu_i_arid_i),
        .cpu_i_arlen_i(cpu_i_arlen_i),
        .cpu_i_arburst_i(cpu_i_arburst_i),
        .cpu_i_rready_i(cpu_i_rready_i),
        .cpu_d_awvalid_i(cpu_d_awvalid_i),
        .cpu_d_awaddr_i(cpu_d_awaddr_i),
        .cpu_d_awid_i(cpu_d_awid_i),
        .cpu_d_awlen_i(cpu_d_awlen_i),
        .cpu_d_awburst_i(cpu_d_awburst_i),
        .cpu_d_wvalid_i(cpu_d_wvalid_i),
        .cpu_d_wdata_i(cpu_d_wdata_i),
        .cpu_d_wstrb_i(cpu_d_wstrb_i),
        .cpu_d_wlast_i(cpu_d_wlast_i),
        .cpu_d_bready_i(cpu_d_bready_i),
        .cpu_d_arvalid_i(cpu_d_arvalid_i),
        .cpu_d_araddr_i(cpu_d_araddr_i),
        .cpu_d_arid_i(cpu_d_arid_i),
        .cpu_d_arlen_i(cpu_d_arlen_i),
        .cpu_d_arburst_i(cpu_d_arburst_i),
        .cpu_d_rready_i(cpu_d_rready_i),
        .spi_miso_i(spi_miso_i),
        .uart_txd_i(uart_txd_i),
        .gpio_input_i(gpio_input_i),

        // Outputs
        .intr_o(intr_o),
        .inport_awready_o(inport_awready_o),
        .inport_wready_o(inport_wready_o),
        .inport_bvalid_o(inport_bvalid_o),
        .inport_bresp_o(inport_bresp_o),
        .inport_bid_o(inport_bid_o),
        .inport_arready_o(inport_arready_o),
        .inport_rvalid_o(inport_rvalid_o),
        .inport_rdata_o(inport_rdata_o),
        .inport_rresp_o(inport_rresp_o),
        .inport_rid_o(inport_rid_o),
        .inport_rlast_o(inport_rlast_o),
        .mem_awvalid_o(mem_awvalid_o),
        .mem_awaddr_o(mem_awaddr_o),
```

```
        .mem_awid_o(mem_awid_o),
        .mem_awlen_o(mem_awlen_o),
        .mem_awburst_o(mem_awburst_o),
        .mem_wvalid_o(mem_wvalid_o),
        .mem_wdata_o(mem_wdata_o),
        .mem_wstrb_o(mem_wstrb_o),
        .mem_wlast_o(mem_wlast_o),
        .mem_bready_o(mem_bready_o),
        .mem_arvalid_o(mem_arvalid_o),
        .mem_araddr_o(mem_araddr_o),
        .mem_arid_o(mem_arid_o),
        .mem_arlen_o(mem_arlen_o),
        .mem_arburst_o(mem_arburst_o),
        .mem_rready_o(mem_rready_o),
        .cpu_i_awready_o(cpu_i_awready_o),
        .cpu_i_wready_o(cpu_i_wready_o),
        .cpu_i_bvalid_o(cpu_i_bvalid_o),
        .cpu_i_bresp_o(cpu_i_bresp_o),
        .cpu_i_bid_o(cpu_i_bid_o),
        .cpu_i_arready_o(cpu_i_arready_o),
        .cpu_i_rvalid_o(cpu_i_rvalid_o),
        .cpu_i_rdata_o(cpu_i_rdata_o),
        .cpu_i_rresp_o(cpu_i_rresp_o),
        .cpu_i_rid_o(cpu_i_rid_o),
        .cpu_i_rlast_o(cpu_i_rlast_o),
        .cpu_d_awready_o(cpu_d_awready_o),
        .cpu_d_wready_o(cpu_d_wready_o),
        .cpu_d_bvalid_o(cpu_d_bvalid_o),
        .cpu_d_bresp_o(cpu_d_bresp_o),
        .cpu_d_bid_o(cpu_d_bid_o),
        .cpu_d_arready_o(cpu_d_arready_o),
        .cpu_d_rvalid_o(cpu_d_rvalid_o),
        .cpu_d_rdata_o(cpu_d_rdata_o),
        .cpu_d_rresp_o(cpu_d_rresp_o),
        .cpu_d_rid_o(cpu_d_rid_o),
        .cpu_d_rlast_o(cpu_d_rlast_o),
        .spi_clk_o(spi_clk_o),
        .spi_mosi_o(spi_mosi_o),
        .spi_cs_o(spi_cs_o),
        .uart_rxd_o(uart_rxd_o),
        .gpio_output_o(gpio_output_o),
        .gpio_output_enable_o(gpio_output_enable_o)
    );
    //for master1
         task write1;
        input [31:0] addr;
        input [31:0] data;
        begin

   @(posedge clk_i);
        inport_awaddr_i = addr;
        inport_awvalid_i = 1;
```

```
        inport_wdata_i = data;
        inport_wstrb_i = 4'b1111;
        inport_awlen_i=8'b00;
        inport_wvalid_i = 1;
        inport_awid_i=4'b0000;
        inport_wlast_i=1'b1;
        inport_awburst_i=2'b00;
        @(posedge clk_i);
        // Wait for handshake signals
        wait(inport_awready_o && inport_wready_o);
        @(posedge clk_i);
        inport_awvalid_i = 0;
        inport_wvalid_i = 0;
        //inport_awaddr_i = 32'b0;
        //inport_wdata_i = 32'b0;

        inport_wlast_i=1'b0;
        //cpu_i_awid_i=4'b0000;

        // Wait for write response
        inport_bready_i = 1;
        wait(inport_bvalid_o);
        repeat(2)@(posedge clk_i);
        inport_bready_i = 0;

        end
    endtask

     task read1;
        input [31:0] addr;
        output [31:0] data;
        begin
           // Address Phase
            @(posedge clk_i);
            inport_arvalid_i = 1;
            inport_araddr_i = addr;
            inport_arid_i=4'b0000;
            inport_arburst_i=2'b01;
            inport_arlen_i=8'h00;
            @(posedge clk_i);
            wait(inport_arready_o);
            @(posedge clk_i);
            inport_arvalid_i = 0;

            // Data Phase
            inport_rready_i = 1;
            wait(inport_rvalid_o);
            @(posedge clk_i);
            data = inport_rdata_o;
            @(posedge clk_i);

            inport_rready_i = 0;
```

```
            // Add spacing between transactions
            repeat(2) @(posedge clk_i);
        end
    endtask
  //for master2
     task write3;
       input [31:0] addr;
       input [31:0] data;
       begin
            @(posedge clk_i);
            cpu_i_awvalid_i = 1;
            cpu_i_awaddr_i = addr;
            cpu_i_awid_i=4'b1000;
            cpu_i_awlen_i=8'h00;
            cpu_i_awburst_i=2'b01;
             // Data Phase
            cpu_i_wvalid_i = 1;
            cpu_i_wdata_i = data;
            cpu_i_wstrb_i=4'b1111;
            cpu_i_wlast_i=1'b1;
            @(posedge clk_i);
            wait(cpu_i_wready_o&cpu_i_awready_o);
            @(posedge clk_i);
            cpu_i_wvalid_i = 0;
            cpu_i_awvalid_i = 0;
            cpu_i_wlast_i=0;
            // Response Phase
            cpu_i_bready_i = 1;
            wait(cpu_i_bvalid_o);
            @(posedge clk_i);
            cpu_i_bready_i = 0;
            // Add spacing between transactions
            repeat(2) @(posedge clk_i);
        end
    endtask

     task read3;
        input [31:0] addr;
        output [31:0] data;
        begin
           // Address Phase
            @(posedge clk_i);
            cpu_i_arvalid_i = 1;
            cpu_i_araddr_i = addr;
            cpu_i_arid_i=4'b1000;
            cpu_i_arburst_i=2'b01;
            cpu_i_arlen_i=8'h00;
            @(posedge clk_i);
            wait(cpu_i_arready_o);
            @(posedge clk_i);
            cpu_i_arvalid_i = 0;
```

```
            // Data Phase
            cpu_i_rready_i = 1;
            wait(cpu_i_rvalid_o);
            @(posedge clk_i);
            data = cpu_i_rdata_o;
            @(posedge clk_i);

            cpu_i_rready_i = 0;

            // Add spacing between transactions
            repeat(2) @(posedge clk_i);
        end
    endtask
//for master3
  task write2;
        input [31:0] addr;
        input [31:0] data;
        begin
            @(posedge clk_i);
            cpu_d_awvalid_i = 1;
            cpu_d_awaddr_i = addr;
            cpu_d_awid_i=4'b0100;
            cpu_d_awlen_i=8'h00;
            cpu_d_awburst_i=2'b01;
             // Data Phase
            cpu_d_wvalid_i = 1;
            cpu_d_wdata_i = data;
            cpu_d_wstrb_i=4'b1111;
            cpu_d_wlast_i=1'b1;
             @(posedge clk_i);
            wait(cpu_d_wready_o&cpu_d_awready_o);
            @(posedge clk_i);
            cpu_d_wvalid_i = 0;
            cpu_d_awvalid_i = 0;
            cpu_i_wlast_i=0;
            // Response Phase
            cpu_d_bready_i = 1;
            wait(cpu_d_bvalid_o);
            @(posedge clk_i);
            cpu_d_bready_i = 0;
            // Add spacing between transactions
            repeat(2) @(posedge clk_i);
        end
    endtask

     task read2;
        input [31:0] addr;
        output [31:0] data;
        begin
           // Address Phase
            @(posedge clk_i);
```

```
            cpu_d_arvalid_i = 1;
            cpu_d_araddr_i = addr;
            cpu_d_arid_i=4'b0100;
            cpu_d_arburst_i=2'b01;
            cpu_d_arlen_i=8'h00;
            @(posedge clk_i);
            wait(cpu_d_arready_o);
            @(posedge clk_i);
            cpu_d_arvalid_i = 0;

            // Data Phase
            cpu_d_rready_i = 1;
            wait(cpu_d_rvalid_o);
            @(posedge clk_i);
            data = cpu_d_rdata_o;
            @(posedge clk_i);

            cpu_d_rready_i = 0;

            // Add spacing between transactions
            repeat(2) @(posedge clk_i);
        end
    endtask

    // Clock generation
    initial begin
        clk_i = 0;
        forever #5 clk_i = ~clk_i; // 50MHz clock
    end

    // Reset generation
    initial begin
        rst_i = 0;
        #100;
        rst_i = 1;
    end

    // Test sequence
    initial begin
        // Initialize all inputs
        inport_awvalid_i = 0;
        inport_awaddr_i = 0;
        inport_awid_i = 0;
        inport_awlen_i = 0;
        inport_awburst_i = 0;
        inport_wvalid_i = 0;
        inport_wdata_i = 0;
        inport_wstrb_i = 0;
        inport_wlast_i = 0;
        inport_bready_i = 0;
        inport_arvalid_i = 0;
        inport_araddr_i = 0;
```

```
        inport_arid_i = 0;
        inport_arlen_i = 0;
        inport_arburst_i = 0;
        inport_rready_i = 0;
        mem_awready_i = 0;
        mem_wready_i = 0;
        mem_bvalid_i = 0;
        mem_bresp_i = 0;
        mem_bid_i = 0;
        mem_arready_i = 0;
        mem_rvalid_i = 0;
        mem_rdata_i = 0;
        mem_rresp_i = 0;
        mem_rid_i = 0;
        mem_rlast_i = 0;
        cpu_i_awvalid_i = 0;
        cpu_i_awaddr_i = 0;
        cpu_i_awid_i = 0;
        cpu_i_awlen_i = 0;
        cpu_i_awburst_i = 0;
        cpu_i_wvalid_i = 0;
        cpu_i_wdata_i = 0;
        cpu_i_wstrb_i = 0;
        cpu_i_wlast_i = 0;
        cpu_i_bready_i = 0;
        cpu_i_arvalid_i = 0;
        cpu_i_araddr_i = 0;
        cpu_i_arid_i = 0;
        cpu_i_arlen_i = 0;
        cpu_i_arburst_i = 0;
        cpu_i_rready_i = 0;
        cpu_d_awvalid_i = 0;
        cpu_d_awaddr_i = 0;
        cpu_d_awid_i = 0;
        cpu_d_awlen_i = 0;
        cpu_d_awburst_i = 0;
        cpu_d_wvalid_i = 0;
        cpu_d_wdata_i = 0;
        cpu_d_wstrb_i = 0;
        cpu_d_wlast_i = 0;
        cpu_d_bready_i = 0;
        cpu_d_arvalid_i = 0;
        cpu_d_araddr_i = 0;
        cpu_d_arid_i = 0;
        cpu_d_arlen_i = 0;
        cpu_d_arburst_i = 0;
        cpu_d_rready_i = 0;
        spi_miso_i = 0;
        uart_txd_i = 1;
        gpio_input_i = 0;

        // Wait for reset to complete
```

```
        @(posedge rst_i);

        #20;
////////////Testing IRQ controller///////////////
         write1({32'h90000008},32'd15);
         read1(32'h90000000,read_data);

 /////////////////Testing GPIO////////////////////
   write1({24'h940000, `GPIO_DIRECTION}, 32'h0000FFFF);
   write1({24'h940000, `GPIO_OUTPUT}, 32'h0000A5A5);
   write2({24'h940000, `GPIO_OUTPUT_SET}, 32'h0000FF00);
   write3({24'h940000, `GPIO_OUTPUT_CLR}, 32'h000000A5);
   #20;
   gpio_input_i = 32'hA5A50000;
   #20;
   read1({24'h940000, `GPIO_INPUT}, read_data);
////////////////Testing UART//////////////////////
        // Simulate UART RX input
        #8000 uart_txd_i = 0; // start bit
        #8000 uart_txd_i = 1;
        #8000 uart_txd_i = 1;
        #8000 uart_txd_i = 1;
        #8000 uart_txd_i = 0;
        #8000 uart_txd_i = 1;
        #8000 uart_txd_i = 0;
        #8000 uart_txd_i = 0;
        #8000 uart_txd_i = 1;
        #8000 uart_txd_i = 1; // stop bit
 // Write TX Register (0x04) with data 0x55 ('U' ASCII)

        write1(32'h92000004,32'd55);
 // Write TX Register (0x0C) with data 0x10 ( enable interrupt)
        @(posedge clk_i)
        write2(32'h9200000C,32'd16);
        #200;
// Read Status Register
        read2(32'h92000008,read_data);
        //Reading ISR for IRQ_CTRL
        read1(32'h90000000,read_data);

// Read RX Buffer
        wait(read_data[1]);
        //making read request from same master
        read1(32'h92000000,read_data);
        //reading ISR form IRQ_ctrl
        write1({32'h9000000C},32'd2);
        read1(32'h90000000,read_data);


   #20;
 /////////////////Testing Timer///////////
 //Write Operations
```

```
  write1({32'h9100000C}, 32'h3A5A5000);
  write1({32'h91000008}, 32'h00000000);
  write2({32'h91000018}, 32'h000A5000);
  write3({32'h91000018}, 32'h005A5000);
  write3({32'h91000014}, 32'h00000004);
  //Read Operations
  read3({32'h91000008}, read_data);
  read1({32'h9100000C}, read_data);
  read3({32'h91000018}, read_data);
  read2({32'h91000014}, read_data);



//////////////////Testing SPI //////////////
  //enable global interrupt
 write1({32'h9300001C}, 32'h8000000);
 //enable tx interrupt
 write2({32'h93000028}, 32'h0000004);
 //enabling spi&master mode
 write3({32'h93000060}, 32'h00000C6);
 //writing data to dtr register
 write2({32'h93000068}, 32'h00000A5);
 write2({32'h93000068}, 32'h0000000);
 #400;
 spi_miso_i=1'b1;
 #1000;
 spi_miso_i=1'b0;
 #400
 //Reading from drr register
 read1({32'h9300006C}, read_data);


    end
endmodule
```

3.2 Post Synthesis Timing Simulation

The testbench checks all the main parts of the system-on-chip working together. It tests the three AXI master ports by reading and writing to all the peripherals, making sure the communication protocol works correctly. We verified that interrupts from the GPIO and UART modules properly reach the CPU through the interrupt controller. The test sequences cover normal operations as well as special cases like back-to-back transfers from different masters.

Figure 6: Inputs are given from inport

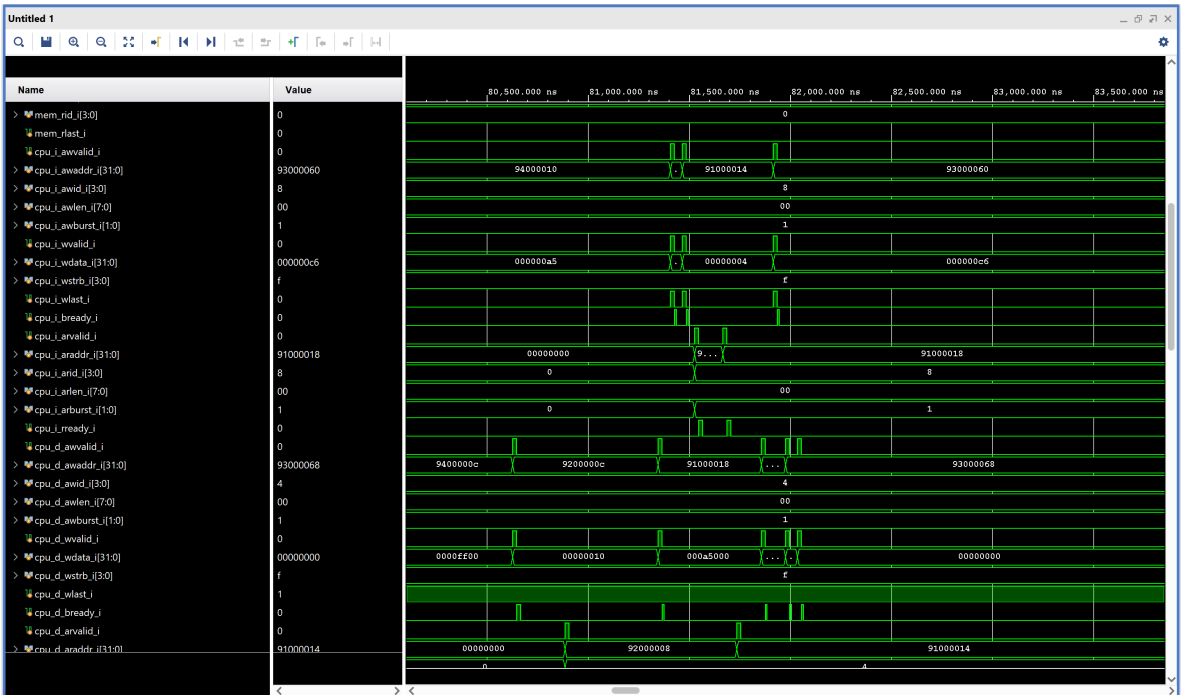

Figure 7: Observed Output signals

4 Integration of RISCV_Top Module

4.1 Modules Used

The `riscv_top.v` integrates three components:

- `icache.v`: Instruction cache module (TA-provided)
- `riscv_core.v`: Main processor core (our implementation)
- `dport_bridge.v`: Debug port interface (TA-provided)

4.2 Module Interconnection

The `riscv_top.v` integrates the processor core with its memory subsystem through the following architecture:

- **Instruction Cache Hierarchy:**
  - `icache.v` manages cache operations as the controller
  - `icache_data_ram.v` stores actual instruction data
  - `icache_tag.v` handles address tagging and comparison
- **Core Interface:**
  - `riscv_core.v` connects to the cache for instruction fetching
  - Cache hits return instructions in 1 cycle, misses trigger memory access
- **Debug System:**
  - `dport_bridge.v` provides external debug access
  - Maintains coherency between debug probes and cache contents

Data Flow

The integrated system operates through two primary pathways:

1. *Normal Execution:* Core requests → Cache check → Fast hit return or memory fetch
2. *Debug Access:* Debugger commands → Bridge → Bypass or flush cache as needed

5 Integration of RISC-V SoC Module

5.1 Modules Used

The `riscv_soc.v` integrates three key components:

- `axi4lite_axi4_conv.v`: AXI4-Lite to AXI4 protocol converter (TA-provided) for bridging different bus standards
- `riscv_top.v`: Complete RISC-V processor subsystem (our implementation) containing the core and caches
- `soc.v`: System-on-Chip infrastructure (our implementation) for peripheral interconnect and memory mapping

Module Interconnections

- **AXI Bridge (`axi4lite_axi4_conv.v`)**:
  - Converts lightweight AXI4-Lite transactions from peripherals to full AXI4 for memory access
  - Bridges between peripheral bus (PS) and memory system (PL)
- **Processor Subsystem (`riscv_top.v`)**:
  - Connects to AXI4 interface for instruction/data memory access
  - Cache-coherent requests routed through the bridge
  - Debug port exposed to SoC infrastructure
- **SoC Infrastructure (`soc.v`)**:
  - Memory-maps peripherals (UART, GPIO, etc.) to AXI4-Lite
  - Routes interrupts from peripherals to the core
  - Manages clock/reset domains for all components

System Operation

1. *Processor Access*:
   - Core generates AXI4 requests → Bridge converts for peripheral access
   - Cache misses go through bridge to main memory
2. *Peripheral Access*:
   - Devices use AXI4-Lite → Bridge converts to AXI4 → Memory controller

5.2 Block Diagram

The BRAM Generator (`blk_mem_gen_0`) and AXI_BRAM Controller (`axi_bram_ctrl_0`) handle memory storage and access in the `riscv_soc`. The BRAM acts as a simple on-chip memory, while the AXI_BRAM Controller allows the CPU to read and write data using the AXI bus. This setup makes sure the processor can easily access memory for instructions and data during operation.

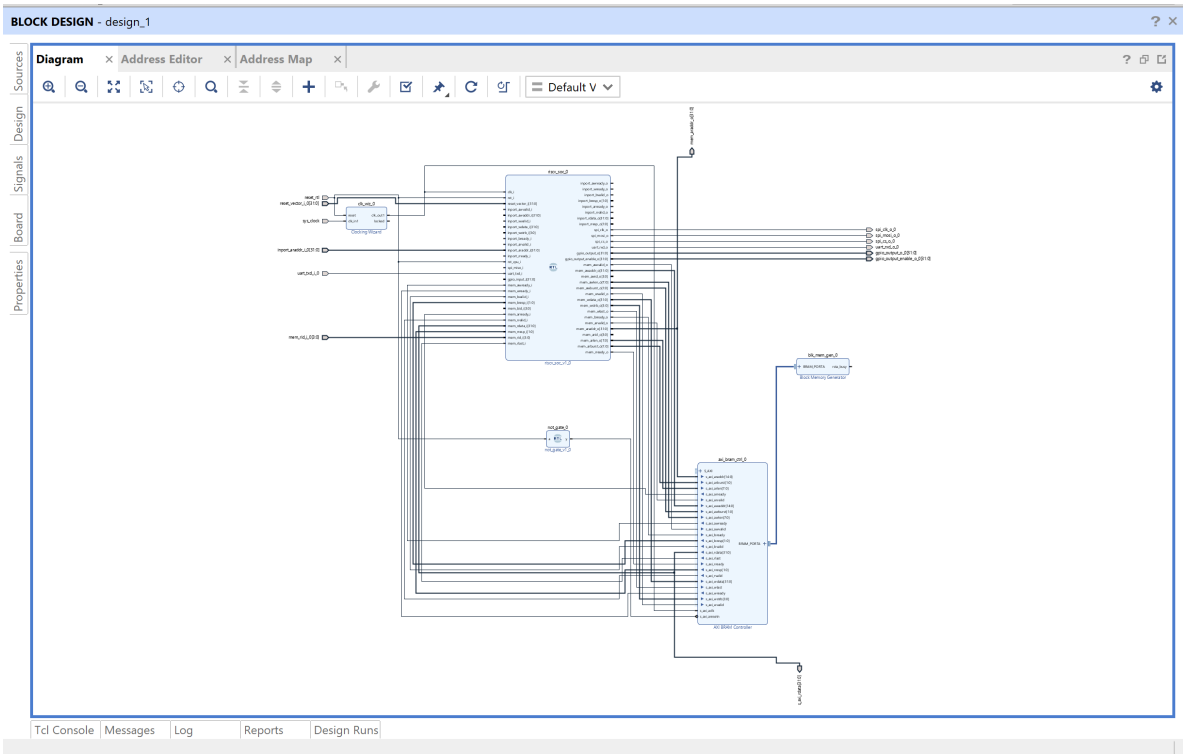

Figure 8: Block Diagram for integration of BRAM with `riscv_soc`

5.3 Assembly Language Program

We performed a series of operations to verify the functionality of the SoC and observe outputs via the memory-mapped **GPIO** peripheral. The GPIO is accessed by writing specific values to a designated base address using the `sw` (store word) instruction.

We first computed the base address for the GPIO as follows:

```
addi  x6, x0, 0x18    # x6 = 0x18 (24 in decimal)
addi  x5, x0, 0x94    # x5 = 0x94
sll   x5, x5, x6      # x5 = x5 << 24 = 0x94000000
```

- The instruction `sll x5, x5, x6` performs a **logical left shift** of the value `0x94` by 24 bits.

- This gives us the **GPIO base address**:

`x5 = 0x94 << 24 = 0x94000000`

This value is used as the base for all GPIO-related store instructions.
The following instructions are responsible for outputting data to the GPIO:

```
sw x18, 0x08, x5    # *(0x94000000 + 0x08) = x18 = 0x00000064
sw x22, 0x08, x5    # *(0x94000000 + 0x08) = x22 = 0x000000AC
sw x23, 0x08, x5    # *(0x94000000 + 0x08) = x23 = 0x00000090
```

Each `sw` instruction writes to the same address offset (`0x08`) of the GPIO base, so the value is **overwritten** with each new write. The **final observed value** at GPIO offset `0x08` will be:
`Final Value at 0x94000008 = 0x00000090`

However, all three values (`0x64`, `0xAC`, `0x90`) are written sequentially and can be observed during program execution. The following code shows the whole assembly level program and they are converted into hexvalues using the Venus compiler and stored as a `.coe` and imported into the Block Memory Generator.

```
loop:
addi x6, x0, 0x18
addi x1, x0, 0x90
addi x2, x0, 0x91
addi x3, x0, 0x92
addi x4, x0, 0x93
addi x5, x0, 0x94
sll x1, x1, x6      #90000000
sll x2, x2, x6      #91000000
sll x3, x3, x6      #92000000
sll x4, x4, x6      #93000000
sll x5, x5, x6      #94000000
# peripherals
addi x9, x0, 0xFF
addi x7, x0, 0x01   #GPIO
addi x8, x0, 0x09
addi x10, x0, 0x08
sll x9, x9, x6
addi x11, x0, 16    #UART
addi x12, x0, 0x36  #UART
sw x7, 0x1C, x1
sw x8, 0x08, x1
addi x13, x0, 0x80  #SPI
sub x21, x0, x7     #GPIO
sw x9, 0x0C, x2
sw x10, 0x08, x2
addi x14, x0, 4     #SPI
sll x13, x13, x6    #SPI
sw x11, 0x0C, x3    #UART
sw x12, 0x04, x3    #UART
```

```
addi x15, x0, 0xC6  #SPI
addi x16, x0, 0x76  #SPI
sw x13, 0x1c, x4    #SPI
sw x14, 0x28, x4    #SPI
sw x15, 0x60, x4    #SPI
sw x16, 0x68, x4    #SPI
sw x21, 0x00, x5    #GPIO
#some basic operations
add x22,x12,x16
mul x23,x8,x11
addi x18,x0, 0x64
addi x19,x0,0x70
addi x26,x0,0x65
sw x18, 0x08, x5     #GPIO
sw x22, 0x08, x5     #GPIO
sw x23, 0x08, x5     #GPIO
bne x18,x19,loop
```

5.4 Testbench code for riscv_soc

The following testbench is written and it just initializes the default signals and maintains `mem_rid` as 4'd8 to properly route back the `rvalid` signal to the proper master. The instructions stored in BRAM must be sent to the ICACHE block by the Retime block and AXI Arbiter.

```
module design_tb();
    reg clk_i;
    reg rst_i;
    reg [31:0] reset_vector_i_0;

    // AXI interface signals
    reg [31:0] inport_araddr_i_0;
    wire [31:0]gpio_output_o_0;
    wire [31:0]gpio_output_enable_o_0;
    wire spi_clk_o_0;
    wire spi_cs_o_0;
    reg spi_miso_i_0;
    wire spi_mosi_o_0;
    wire [31:0] s_axi_rdata;
    wire [31:0] inport_rdata_o_0;
    // Memory interface signals
    reg [3:0] mem_rid_i_0;
    wire [31:0] mem_araddr_o;
    // UART
    wire uart_rxd_o_0;
    reg uart_txd_i_0;
    // Instantiate DUT
    design_1_wrapper dut (
        .sys_clock(clk_i),
```

```
        .reset_rtl(rst_i),
        .reset_vector_i_0(reset_vector_i_0),
        .inport_araddr_i_0(inport_araddr_i_0),
        .mem_rid_i_0(mem_rid_i_0),
        .uart_rxd_o_0(uart_rxd_o_0),
        .uart_txd_i_0(uart_txd_i_0),
        .gpio_output_o_0(gpio_output_o_0),
        .gpio_output_enable_o_0(gpio_output_enable_o_0),
        .spi_clk_o_0(spi_clk_o_0),
        .spi_cs_o_0(spi_cs_o_0),
        .spi_mosi_o_0(spi_mosi_o_0),
        .s_axi_rdata(s_axi_rdata)

    );
    // Clock generation
    always #5 clk_i = ~clk_i;
    // Test sequence
    initial begin
        // Initialize signals
        clk_i = 0;
        rst_i = 0;
        inport_araddr_i_0 = 0;
        uart_txd_i_0 = 0;
        mem_rid_i_0 = 4'b0000;
        reset_vector_i_0 = 32'h0;
        // Reset pulse
        #20 rst_i = 1;
        #20 rst_i = 0;
        mem_rid_i_0 = 4'd8;
        // Finish test
        #50000;
        $finish;
    end
endmodule
```

5.5 Post Synthesis Timing Simulation

The following image shows the output waveform of the above testbench code after the design is synthesized. The following image shows the ouptut signals observed signals from the GPIO Peripheral.

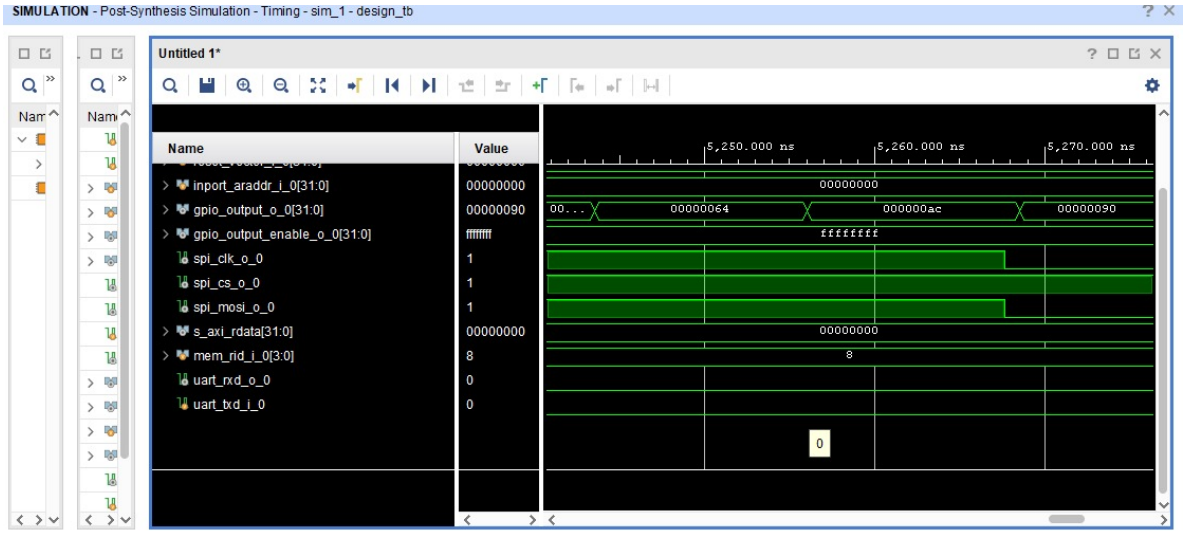

Figure 9: GPIO signals showing the result of our operations in Assembly Code

6 FPGA Implementation of RISC-V SoC Module

The same riscv_soc module integrated in last section is connected with Virtual Input/Output(VIO) and Integrated Logic Analyzer(ILA) for implementing in FPGA. The Block diagram for the same is shown below.

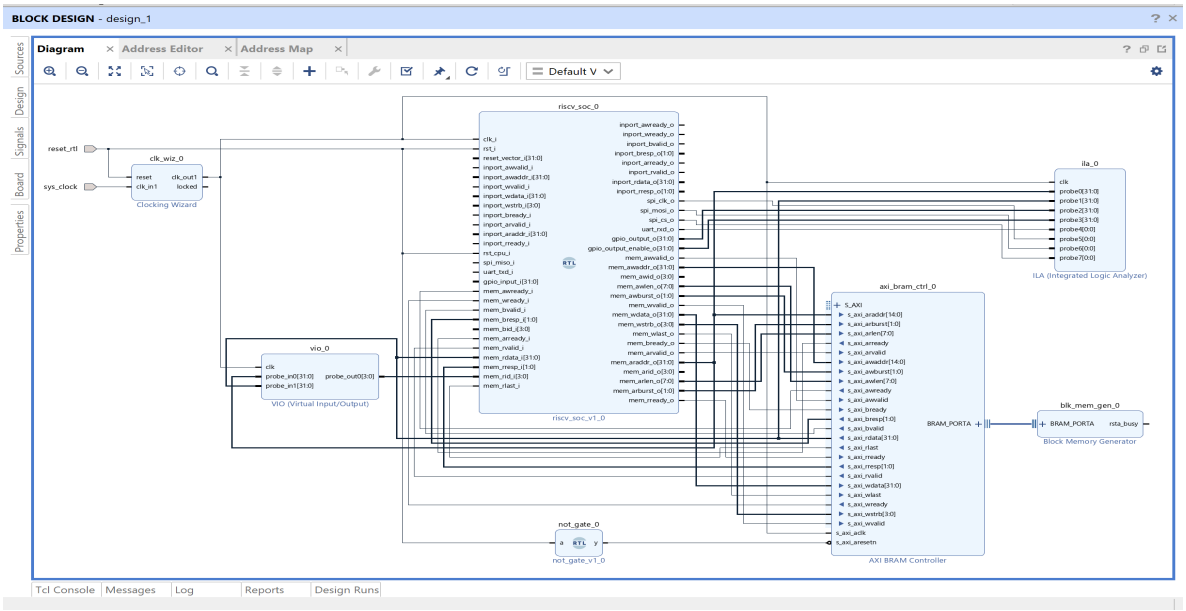

Figure 10: Block Diagram of riscv_soc connected with VIO and ILA

After successful generation of bitstream file we uploaded that in the `pynq-z2` board and through the ILA window we observed the followng output.

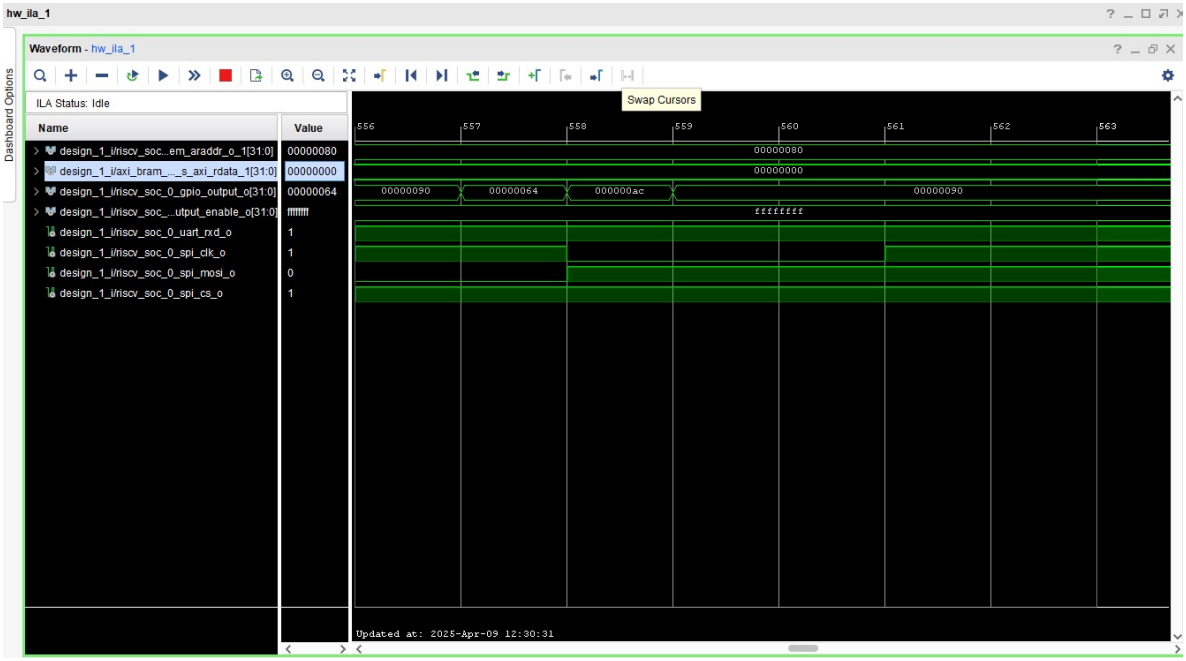

Figure 11: FPGA Implementation of `riscv_soc` module

7 Conclusion

Thus we first checked the `soc.v` with proper testbench and the `riscv_core` block, both are synthesized as expected. Then we went for the final integration thus the RISCV processor is designed according to the specifications and implemented successfully on the `pynq-z2` board.