Indian Institute of Technology Bombay

EE 705 VLSI Design Lab

Assignment III Part A

---

# Implementation of AXI Protocol using AXI BRAM Controller and BRAM IPs

---

*Submitted By:*

Bala Murugan S

Reg No: 24M1173

*Course Instructor:*

Prof. Laxmeesha Somappa

## Contents

**1 Interfacing BRAM with a Xilinx AXI BRAM Controller IP** **4**
1.1 Block Design Overview . . . . . 4
1.2 Testbench Code . . . . . 5
1.3 Post Implementation Timing Simulation . . . . . 7

**2 Controller to Access AXI-BRAM Interface** **7**
2.1 Code for the Controller . . . . . 7
2.2 Block Design . . . . . 10
2.3 Test COE File . . . . . 10
2.4 Testbench Code . . . . . 11
2.5 Post Implementation Timing Simulation . . . . . 11

## List of Figures

1 Block Design of AXI BRAM Controller and BRAM IP . . . . . . . . . . 4
2 Output Waveform demonstrating Write and Read Operations . . . . . 7
3 Block design of Controller interfaced with AXI-BRAM IP . . . . . . . . 10
4 Image of a Test Vector File . . . . . . . . . . . . . . . . . . . . . . . . . 10
5 Output Waveform demonstrating Sequential Read Operations . . . . . 11

# 1 Interfacing BRAM with a Xilinx AXI BRAM Controller IP

## 1.1 Block Design Overview

The testbench (Master) should generate suitable AXI-4 lite signals to perform the following operations:

- Write to the memory (single address)
- Read back from the same address

I started creating the block design by adding the AXI BRAM Controller IP and initialized it to AXI4 Lite Mode and added BRAM IP in Mode Controller. Then added a Clocking Wizard to give clock signals and reset signals. Then I made the required pins external so that I can access them and view them in the simulation using testbench (Master). I generated the output products and created an HDL wrapper, which was instantiated as `uut` in the testbench. This can be noticed in the attached code discussed in the next subsection. The following image shows the block design.

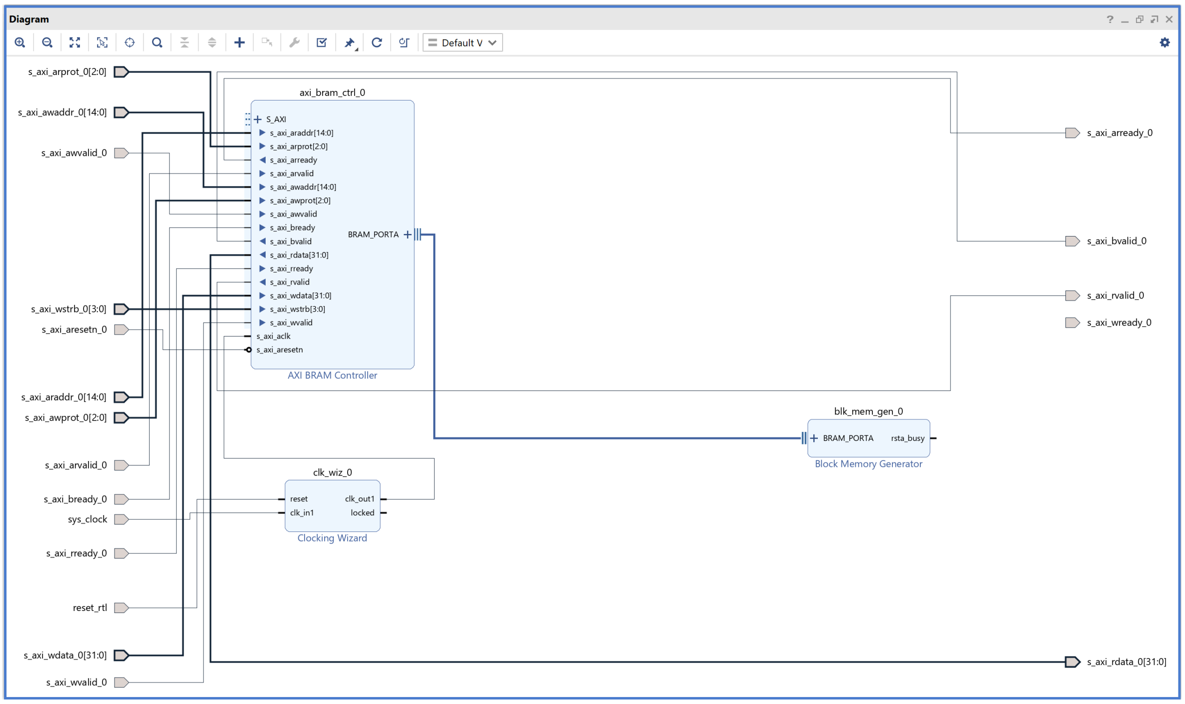


Figure 1: Block Design of AXI BRAM Controller and BRAM IP

## 1.2 Testbench Code

The following is the testbench code that I have written to execute write and read operations. The write address is given after making the `awvalid` signal high, then `wdata` is given after making `wvalid` signal high and waits for `bvalid` so that the write operation is complete. Then for read operation, the `araddr` is given after making `arvalid` high and waits for `rvalid` signal to become high to fetch `rdata`, which indicates that the read operation is completed successfully.

```
‘timescale 1 ns / 1 ps
module axi_controller_wrapper_tb ;
    reg clk_in1_0;
    reg reset_0 ;
    reg s_axi_aresetn_0 ;
    reg [14:0] s_axi_awaddr_0 ;
    reg [2:0] s_axi_awprot_0 ;
    reg s_axi_awvalid_0 ;
    reg s_axi_bready_0 ;
    reg [31:0] s_axi_wdata_0 ;
    reg [3:0] s_axi_wstrb_0 ;
    reg s_axi_wvalid_0 ;
    wire s_axi_bvalid_0 ;
    reg [14:0] s_axi_araddr_0 ;
    reg [2:0] s_axi_arprot_0 ;
    reg s_axi_arvalid_0 ;
    reg s_axi_rready_0 ;
    wire [31:0] s_axi_rdata_0 ;
    wire s_axi_rvalid_0 ;
// Instantiation of axi_controller_wrapper uut
axi_controller_wrapper uut (
    .reset_0(reset_0),
    .s_axi_araddr_0(s_axi_araddr_0),
    .s_axi_aresetn_0(s_axi_aresetn_0),
    .s_axi_arprot_0(s_axi_arprot_0),
    .s_axi_arvalid_0(s_axi_arvalid_0),
    .s_axi_awaddr_0(s_axi_awaddr_0),
    .s_axi_awprot_0(s_axi_awprot_0),
    .s_axi_awvalid_0(s_axi_awvalid_0),
    .s_axi_bready_0(s_axi_bready_0),
    .s_axi_rdata_0(s_axi_rdata_0),
    .s_axi_rready_0(s_axi_rready_0),
    .s_axi_wdata_0(s_axi_wdata_0),
    .s_axi_wstrb_0(s_axi_wstrb_0),
    .s_axi_wvalid_0(s_axi_wvalid_0),
    .s_axi_bvalid_0(s_axi_bvalid_0),
    .s_axi_rvalid_0(s_axi_rvalid_0),
    .clk_in1_0(clk_in1_0)
);
    always #5 clk_in1_0 = ~clk_in1_0 ;
    initial begin
    reset_0 = 1;
    s_axi_aresetn_0 = 0;
    s_axi_araddr_0 = 0;
    s_axi_arprot_0 = 0;
    s_axi_arvalid_0 = 0;
    s_axi_awaddr_0 = 0;
    s_axi_awprot_0 = 0;
    s_axi_awvalid_0 = 0;
    s_axi_bready_0 = 0;
    s_axi_rready_0 = 0;
    s_axi_wdata_0 = 0;
    s_axi_wstrb_0 = 0;
    s_axi_wvalid_0 = 0;
    clk_in1_0 = 0;
    #5;
    reset_0 = 0;
    s_axi_aresetn_0 = 1;
    #5
    // Write operation
    s_axi_wvalid_0 = 1; // write data valid
    s_axi_bready_0 = 1; // Ready to accept write response
    s_axi_awvalid_0 = 1; // write address valid
    s_axi_awprot_0 = 3'b000 ;
    s_axi_awaddr_0 = 15'd100 ; // Address
    s_axi_wstrb_0 = 4'b1111 ; // All bytes valid
    s_axi_wdata_0 = 32'd233456 ; // Write Data
    while (!s_axi_bvalid_0) begin
    #1;
    end
    s_axi_awvalid_0 = 0;
    s_axi_wvalid_0 = 0;
    s_axi_wstrb_0 = 4'b0000 ;
    s_axi_wdata_0 = 32'd0;
    // Read operation
    #10
    s_axi_arprot_0 = 3'b000 ;
    s_axi_arvalid_0 = 1; // read address valid
    s_axi_rready_0 = 1; // Ready to accept read data
    s_axi_araddr_0 = 15'd100 ; // Read Address
    while (!s_axi_rvalid_0) begin
    #10;
    end
    #100;
    $finish ;
end
endmodule
```

## 1.3 Post Implementation Timing Simulation

The following image shows the output waveform of the above testbench code after implementing the design.

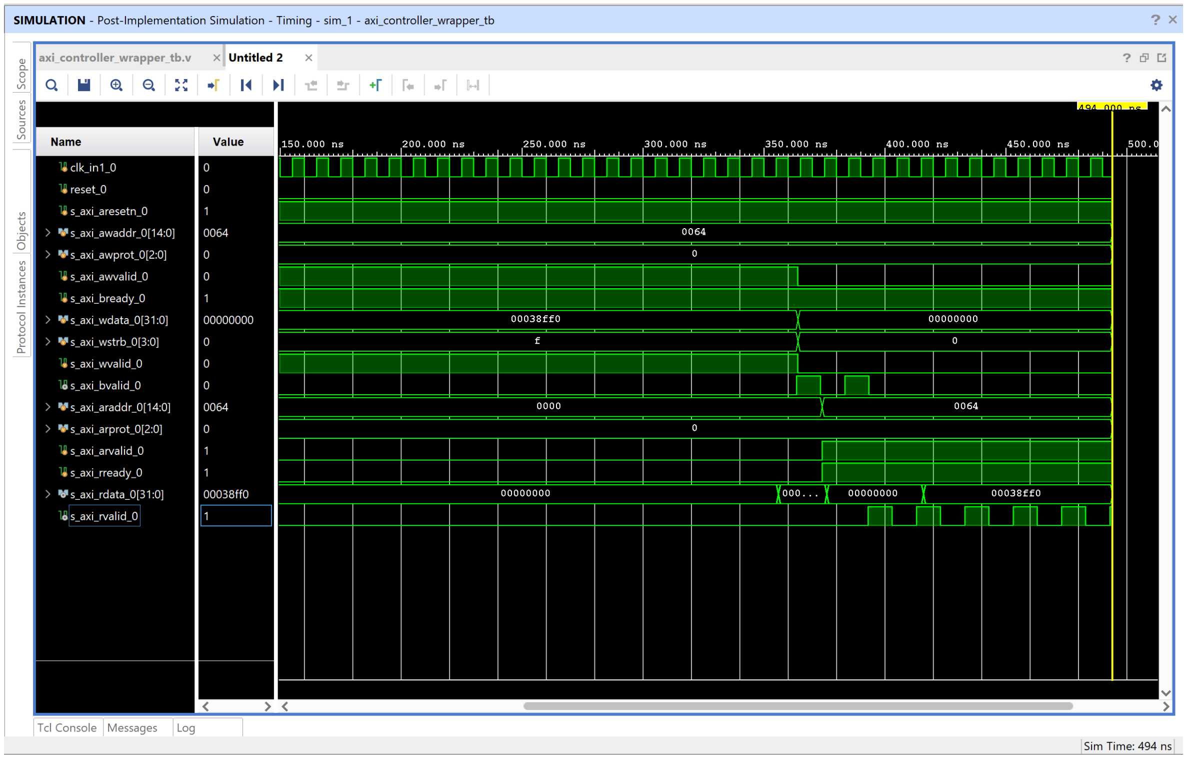


Figure 2: Output Waveform demonstrating Write and Read Operations

## 2 Controller to Access AXI-BRAM Interface

Here, I have to create a new controller module/design that will interface with the AXI-BRAM to read all data from the initialized BRAM. First of all, the BRAM used here is initialized by loading a `.coe` file. The BRAM is used in standalone mode.

### 2.1 Code for the Controller

I have designed a controller which will generate AXI4 Lite control Signals and it will read the data stored in BRAM in a sequential manner through the use of AXI-BRAM Controller. This is achieved by an implementation of state machine which waits for the respective signals such as `rvalid` and sets up a new address in the address bus. This ensures the read operation is completed. I have defined four states such as `IDLE`, `SEND_ADDR`, `WAIT_RVALID`, `DONE`. The `IDLE` state is used for initialization of the signals, then it is moved to `SEND_ADDR` state, where the `arvalid` signals are set high. The next state for SEND_ADDR is WAIT_RVALID which checks whether the **rvalid** is high or not, if it is low then it will hold **arvalid** and the respective **araddr** in the same value, else it will increment the **araddr** by 4 since the memory is byte addressable and moves to SEND_ADDR state. This cycle continues until the last valid address is read.

```
`timescale 1ps / 1ps
module controller(
    input           clk,
    input           reset,
    input           rvalid,
    input           arready,
    output reg [14:0] araddr,
    output reg [2:0]  arprot,
    output reg        arvalid,
    output reg        rready,
    output reg        rstn
);
  localparam IDLE        = 2'b00,
             SEND_ADDR   = 2'b01,
             WAIT_RVALID = 2'b10,
             DONE        = 2'b11;
  reg [1:0] state, next_state;
  // Next-state logic
  always @(*) begin
    case (state)
      IDLE: begin
        next_state = SEND_ADDR;
      end
      SEND_ADDR: begin
        if (arready)
          next_state = WAIT_RVALID;
        else
          next_state = SEND_ADDR;
      end
      WAIT_RVALID: begin
        if (rvalid) begin
          if (araddr < 15'd66)
            next_state = SEND_ADDR;
          else
            next_state = DONE;
        end else begin
          next_state = WAIT_RVALID;
        end
      end
      DONE: begin
        next_state = DONE;
      end
      default: next_state = IDLE;
    endcase
  end
  // Sequential logic
  always @(posedge clk or posedge reset) begin
    if (reset) begin
      state    <= IDLE;
      rstn     <= 0;
      arvalid <= 0;
      rready   <= 1;
      araddr   <= 15'd0;
      arprot   <= 3'b000;
    end else begin
      state <= next_state;
      rstn  <= 1;
      case (state)
        IDLE: begin
          arvalid <= 0;
          rready  <= 1;
          araddr  <= araddr;
        end

        SEND_ADDR: begin
          arvalid <= 1;
          araddr  <= araddr;
        end

        WAIT_RVALID: begin
          arvalid <= 0;
          if (rvalid) begin
            if (araddr < 15'd66)
              araddr <= araddr + 4;
            else
              araddr <= araddr;
          end else begin
            araddr <= araddr;
          end
        end

        DONE: begin
          arvalid <= 0;
          rready  <= 0;
          araddr  <= araddr;
        end

        default: begin
          arvalid <= 0;
          rready  <= 1;
          araddr  <= araddr;
        end
      endcase
    end
  end
endmodule
```

## 2.2 Block Design

The following image shows how the above controller code is interfaced with the AXI BRAM controller IP. Then the block design is generated an HDL wrapper is created.

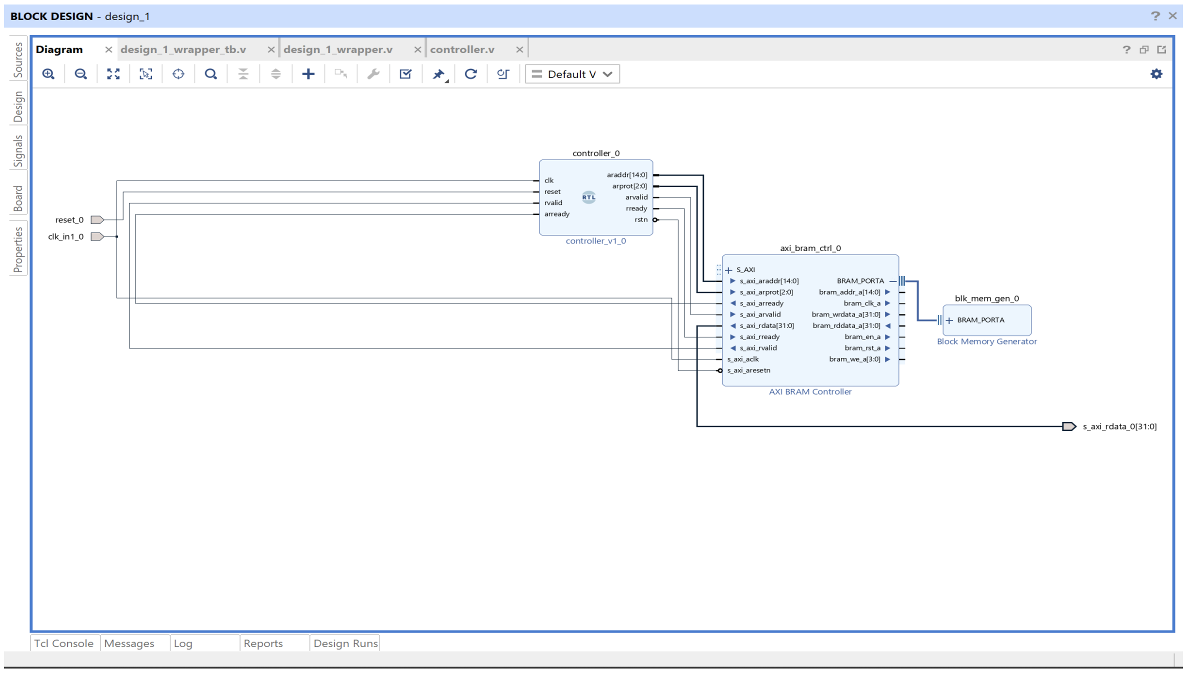
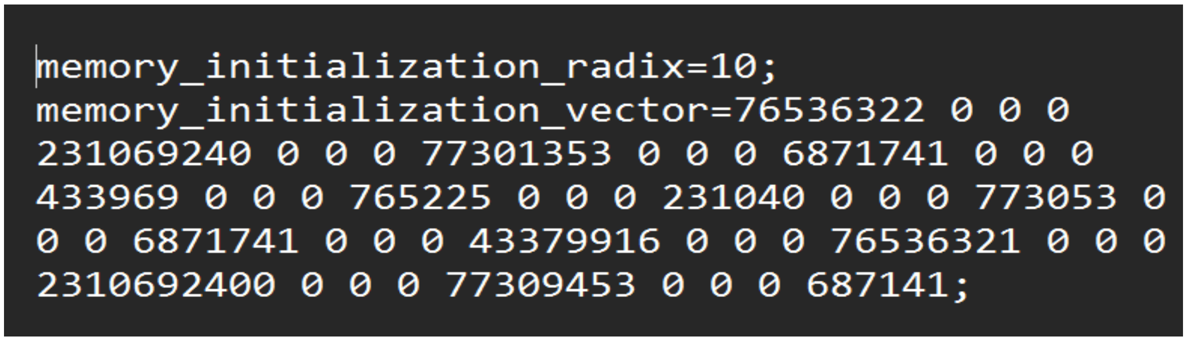


Figure 3: Block design of Controller interfaced with AXI-BRAM IP

## 2.3 Test COE File

The following image shows the `.coe` file that I loaded into the standalone BRAM. The testvector file must have readable values at every fourth index starting from 0000, since in an AXI-BRAM Contorller address is incremented by 4.



```
memory_initialization_radix=10;
memory_initialization_vector=76536322 0 0 0
231069240 0 0 0 77301353 0 0 0 6871741 0 0 0
433969 0 0 0 765225 0 0 0 231040 0 0 0 773053 0
0 0 6871741 0 0 0 43379916 0 0 0 76536321 0 0 0
2310692400 0 0 0 77309453 0 0 0 687141;
```

Figure 4: Image of a Test Vector File

## 2.4 Testbench Code

```
`timescale 1ns / 1ps
module design_1_wrapper_tb;
  reg          clk_in1_0;
  reg          reset_0;
  wire [31:0] dataout_0;
  design_1_wrapper dut (
    .clk_in1_0(clk_in1_0),
    .reset_0(reset_0),
    .dataout_0(dataout_0)
  );
  initial begin
    clk_in1_0 = 1;
    forever #10 clk_in1_0 = ~clk_in1_0;
  end
  initial begin
    reset_0 = 1;
    #5;
    reset_0 = 0;
  end
  always @(posedge clk_in1_0) begin
    $display("Time: %0t, dataout_0 = %h", $time, dataout_0);
  end
  initial begin
    #25000;
    $finish;
  end
endmodule
```

## 2.5 Post Implementation Timing Simulation

The following image shows the output waveform of the above testbench code after the implementation of the design. In the following image it can be seen that the data I gave in the `.coe` file is read one by one.

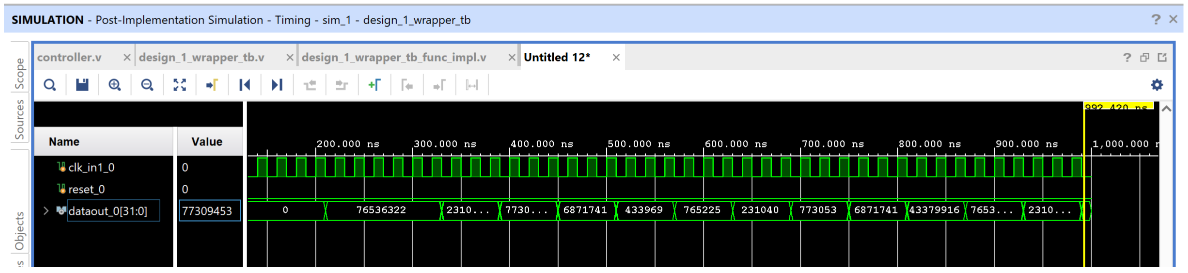


Figure 5: Output Waveform demonstrating Sequential Read Operations