Indian Institute of Technology Bombay

EE 705 VLSI Design Lab

Assignment III Part B

Implementation of AXI-BRAM

Submitted By:

Bala Murugan S

Reg No: 24M1173

Course Instructor:

Prof. Laxmeesha Somappa

Contents

List of Figures

1 Controller to Access AXI-BRAM Interface

I created a 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.

1.1 Design of 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. After reading the 32 bit value stored in the BRAM, it is divided into two registers `a[15:0]` and `b[15:0]`, which are connected to Dadda Multiplier. The result is fetched back from that block and written to the BRAM using the designed controller module. This is achieved by an implementation of state machine which waits for the respective signals such as `rvalid`, which splits `rdata` into a and b inputs then, the signals `awvalid`, `wvalid`, and the respective write address with proper `wstrb` is given. Then the `bready` signal is made `high` and waits for `bvalid` to become `high`. This ensures the write operation is completed. Then again the cycle starts from reading the next address in BRAM and the cycle continues until every value which must be multiplied is read and product is written back. Then the products are read and loaded into a register called `dataout`. I have defined six states such as `IDLE`, `ADDR_READ`, `WAIT_RVALID`, `MULTIPLY`, `WAIT_BVALID` and `DONE`. The `IDLE` state is used for initialization of the signals, then it is moved to `ADDR_READ` state, where the `arvalid` signals is set high. The next state for `ADDR_READ` 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 split the `rdata` into a and b registers and moves to `MULTIPLY` state. In this state, the appropriate values of `awvalid`, `wvalid`, `awaddr`, `wstrb` and `bready` are given. Then after that it moves to `WAIT_BVALID` state. There it waits for `bvalid` which ensures that write operation is completed. Then it moves back to `ADDR_READ` state. This cycle continues until the last valid address is read.

1.2 State Transition Diagram

The following image shows the state diagram of the above implemented design,

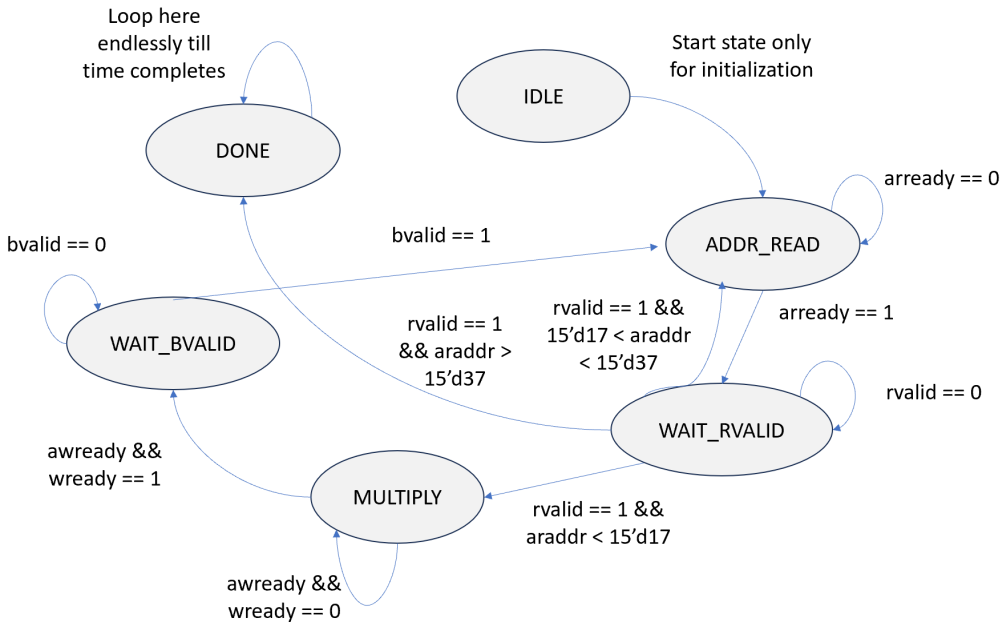

Figure 1: State Transition Diagram

1.3 Controller Code

The following code is written based on the above state transition diagram,

```
`timescale 1ns / 1ps
module controller(
    input clk,
    input reset,
    input awready,
    input bvalid,
    input rvalid,
    input wready,
    input aready,
    input [31:0] rdata,
    input [31:0] result,
    output reg [31:0] wdata,
    output reg [15:0] a,
    output reg [15:0] b,
    output reg [14:0] araddr,
    output reg [14:0] awaddr,
    output reg [2:0] arprot,
    output reg [2:0] awprot,
    output reg arvalid,
    output reg awvalid,
```

```
    output reg [3:0] wstrb,
    output reg wvalid,
    output reg rready,
    output reg rstn,
    output reg bready,
    output reg [31:0] dataout
);
  localparam IDLE        = 3'b000,
             ADDR_READ   = 3'b001,
             WAIT_RVALID = 3'b010,
             MULTIPLY    = 3'b011,
             WAIT_BVALID = 3'b100,
             DONE        = 3'b101;
  reg [2:0] state, next_state;
  reg [31:0] prevrdata;
  reg [14:0] writeaddr;
  // Next-state logic
  always @(*) begin
    case (state)
      IDLE: begin
        next_state = ADDR_READ;
      end
      ADDR_READ: begin
        if (arready)
          next_state = WAIT_RVALID;
        else
          next_state = ADDR_READ;
      end
      WAIT_RVALID: begin
        if (rvalid) begin
          if (araddr < 15'd17)
            next_state = MULTIPLY;
          else if (araddr > 15'd17 && araddr < 15'd45)
            next_state = ADDR_READ;
          else
            next_state = DONE;
        end else begin
          next_state = WAIT_RVALID;
        end
      end
      MULTIPLY: begin
           if(awready && wready)
              next_state = WAIT_BVALID;
           else
              next_state = MULTIPLY;
      end
      WAIT_BVALID: begin
           if(bvalid)
              next_state = ADDR_READ;
           else
              next_state = WAIT_BVALID;
      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  <= 0;
      araddr  <= 15'd0;
      arprot  <= 3'b000;
      awprot <= 3'b000;
      dataout <= 32'd0;
      awaddr <= 15'd0;
      awvalid <= 0;
      wvalid <= 0;
      bready <= 0;
      wdata <= 32'd0;
      wstrb <= 4'd0;
      a <= 16'd0;
      b <= 16'd0;
      writeaddr <= 15'd20;
    end else begin
      state <= next_state;
      rstn  <= 1;
      case (state)
        IDLE: begin
          arvalid <= 0;
          rready  <= 0;
          araddr  <= araddr;
        end
        ADDR_READ: begin
          arvalid <= 1;
          araddr <= araddr;
          rready <= 1;
        end
        WAIT_RVALID: begin
          if (rvalid) begin
            arvalid <= 0;
            rready <= 0;
            if(araddr < 15'd17) begin
              a <= rdata[31:16];
              b <= rdata[15:0];
            end else if (araddr > 15'd17 && araddr < 15'd41) begin
              dataout <= rdata;
              araddr <= araddr + 4;
            end else
              araddr <= araddr;
```

```
          end else begin
            araddr <= araddr;
          end
        end
        MULTIPLY: begin
          awvalid <= 1;
          wvalid <= 1;
          if(awready && wready)begin
             awaddr <= writeaddr;
             wdata <= result;
             wstrb = 4'b1111;
             bready <= 1;
          end
        end
       WAIT_BVALID: begin
          if(bvalid)begin
            bready <= 0;
            awaddr <= 15'd0;
            writeaddr <= writeaddr + 4;
            araddr <= araddr + 4;
            wdata <= 32'd0;
            wstrb <= 4'b0000;
            awvalid <= 0;
            wvalid <= 0;
          end
        end
        DONE: begin
          arvalid <= 0;
          rready  <= 0;
          araddr  <= araddr;
        end
        default: begin
          arvalid <= 0;
          rready  <= 0;
          araddr  <= araddr;
        end
      endcase
    end
  end
endmodule
```

1.4 Block Design

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

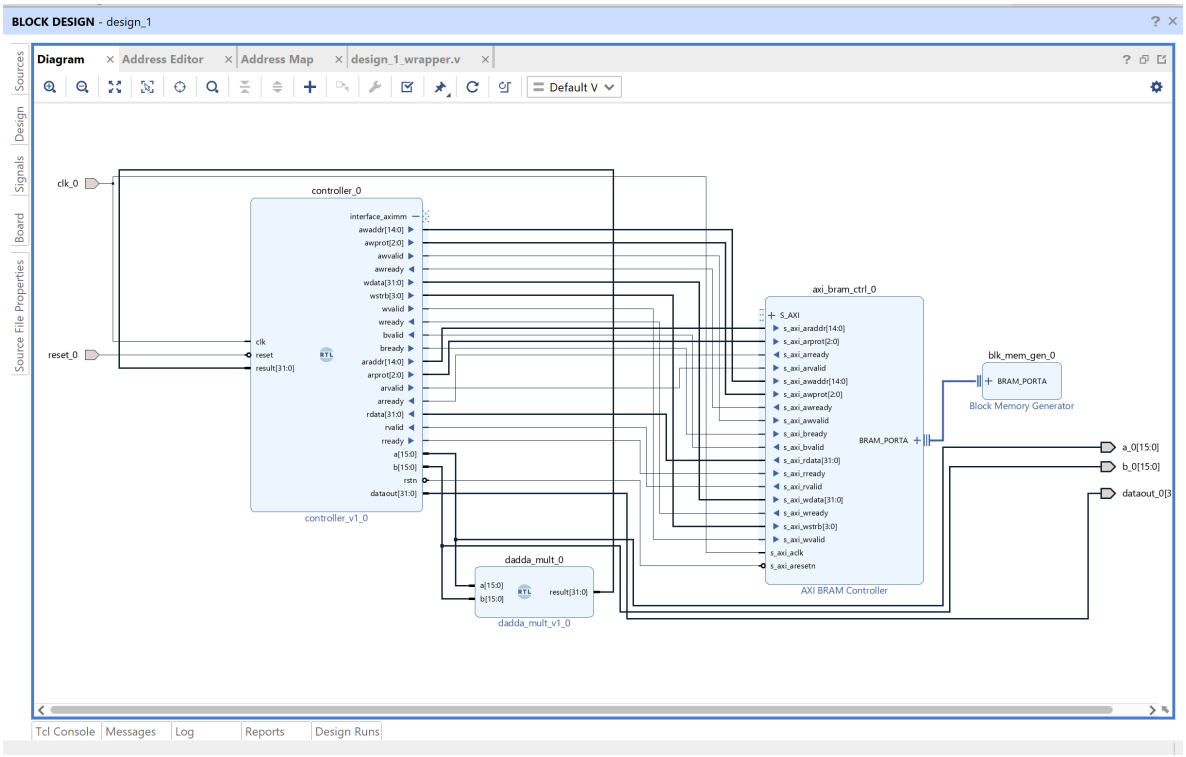

Figure 2: Block design of Controller and Dadda Multiplier interfaced with AXI-BRAM IP

1.5 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 Controller address is incremented by 4.

```
memory_initialization_radix=16;
memory_initialization_vector=
0010004D,
0,
0,
0,
0040007C,
0,
0,
0,
00070008,
0,
0,
0,
143C01B7,
0,
0,
0,
007D01F4;
```

Figure 3: Image of a Test Vector File

1.6 Testbench Code

Then I wrote a simple testbench to verify my design.

```
`timescale 1ns / 1ps
module design_1_wrapper_tb;
  reg         clk_0;
  reg         reset_0;
  wire [31:0] dataout_0;
  wire [15:0] a_0;
  wire [15:0] b_0;
  design_1_wrapper dut (
    .clk_0(clk_0),
    .reset_0(reset_0),
    .dataout_0(dataout_0),
    .a_0(a_0),
    .b_0(b_0)
  );

  initial begin
    clk_0 = 1;
    forever #20 clk_0 = ~clk_0;
  end

  initial begin
    reset_0 = 1;
    #10;
    reset_0 = 0;
  end

  always @(posedge clk_0) begin
    $display("Time: %0t, dataout_0 = %h", $time, dataout_0);
  end

  initial begin
    #2500;
    $finish;
  end

endmodule
```

1.7 Behavioural Simulation

The following image shows the output waveform of the above testbench code after the behavioural simulation. In the following image it can be seen that the data I gave in the `.coe` file is read one by one, and products are computed, the products are read in the data_out register. The multiplicand and multiplier are in the a and b registers (radix is changed to unsigned decimal).

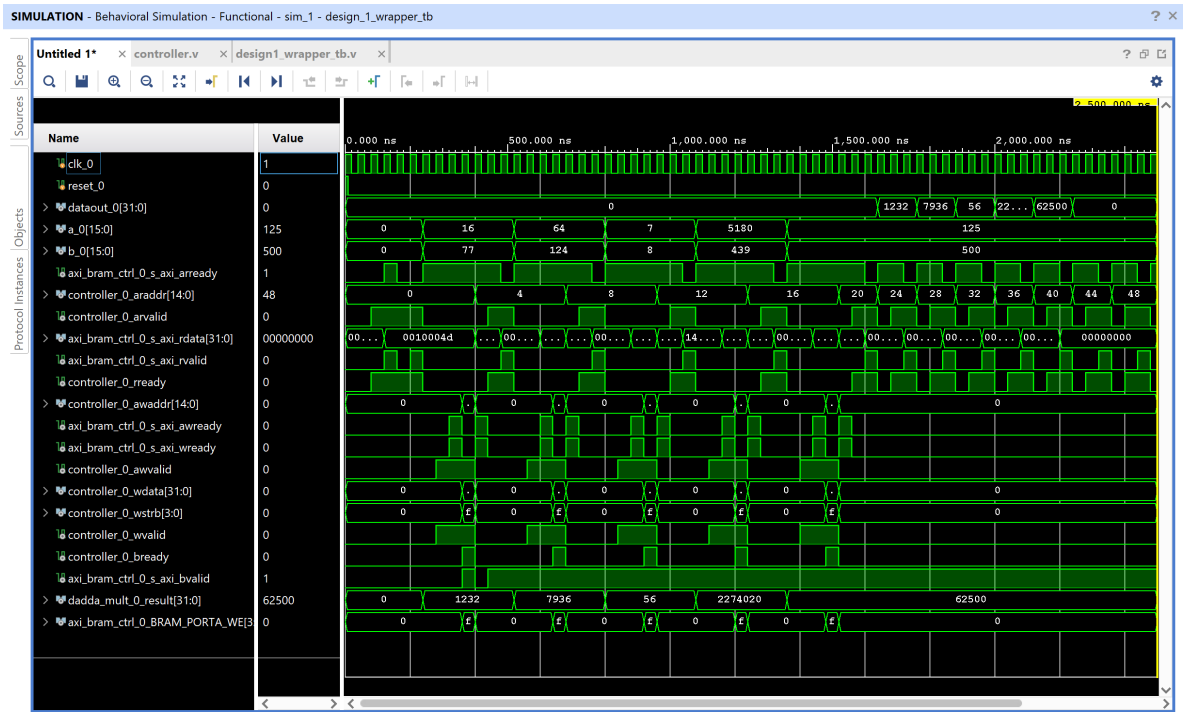

Figure 4: Behavioural Simulation

1.8 Post Implementation Timing Simulation

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

Figure 5: Post Implementation Timing Simulation

1.9 Final Block Design

After the verification of the designed controller using a testbench, I created a new project, attached those source files, and made a complete block diagram including the Clocking Wizard, Integrated Logic Analyzer to see all the output and write AXI signals.

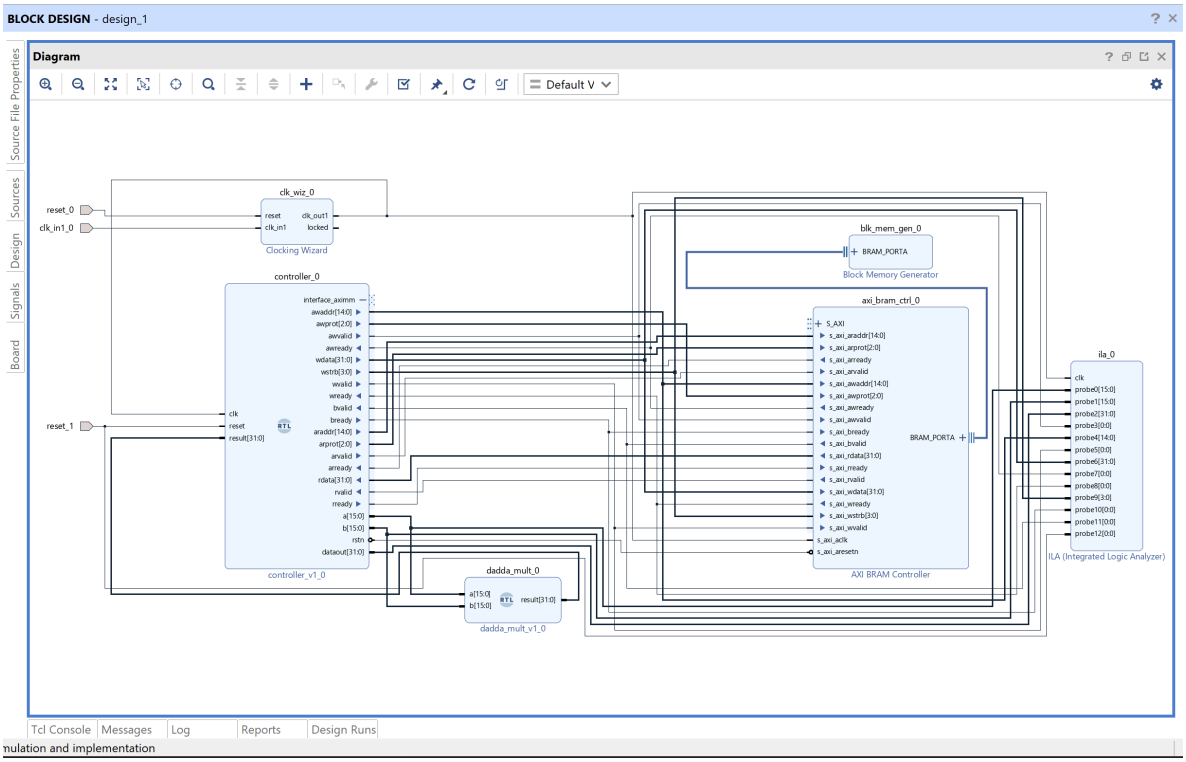

Figure 6: Block design of Controller, AXI-BRAM IP with ILA

1.10 Generation of Bitstream

After saving the design, I created a HDL wrapper, and made it as top level entity. Then I generated the bitstream, the bitstream successful dialog box is shown below.

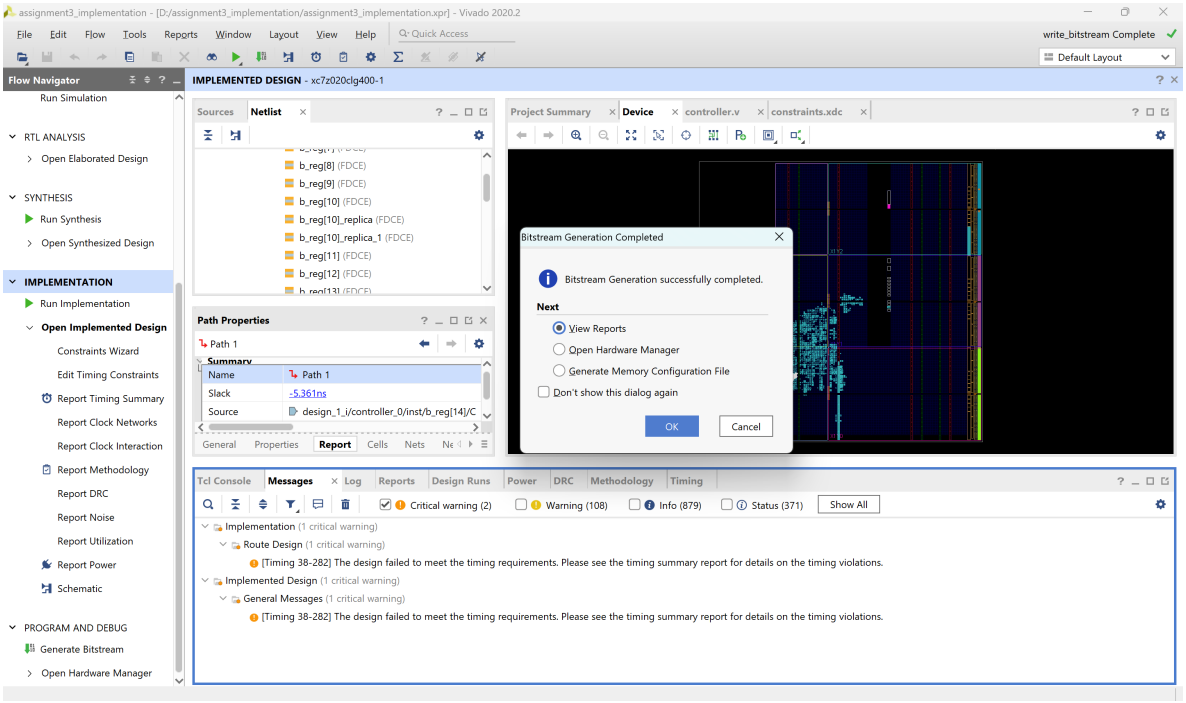

Figure 7: Bitstream Generation Successful Dialog Box