

Laboratory Exercise #3

Creating a Custom Hardware IP and Interfacing it with
Software

Name: Daniel Horan

UIN: 527005307

ECEN 449-504

I. Introduction

In this lab, we will utilize Vivado's 'Create and Package IP' feature to design a custom peripheral for integer multiplication within the Zynq Processing System. Our goal is twofold: first, to develop this hardware module, and second, to craft software in the SDK for its operation. This exercise underscores the vital interplay between hardware and software in FPGA-based co-designs.

II. Procedure

In the initial phase, we set up a Vivado project for the ZYBO Z7-10 board. The primary goal was the development of a custom peripheral for integer multiplication using the 'Create and Package IP' tool. This required the integration of the ARM Cortex A9 processor from the Zynq Chip and the introduction of a multiplication block as a replacement for the traditional GPIO modules.

Following hardware design and export, attention was directed to the Vitis development environment. An application was developed to allocate values to the registers 'slv reg0' and 'slv reg1', and then extract the computational output from 'slv reg2'. The 'picocom' serial console application was used for real-time output visualization.

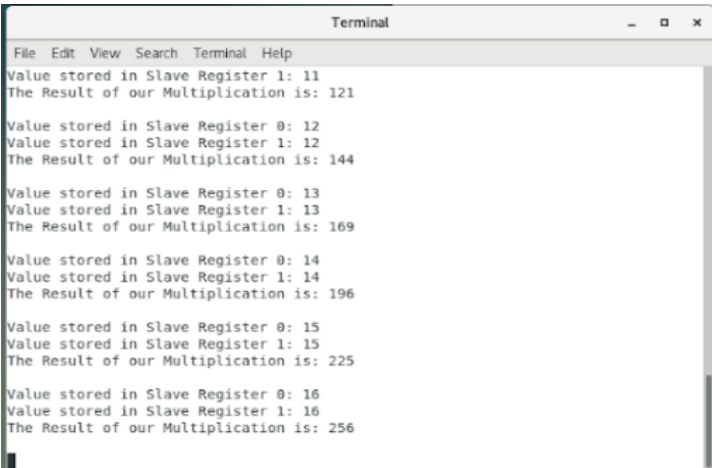
The final step involved a demonstration of the integrated system, emphasizing the interaction between the hardware module and its associated software. The procedure was accompanied by hints to aid in troubleshooting and refinement.

III. Results

During this lab, the main task was to create and integrate a custom IP module for integer multiplication using Vivado. The process started with defining the custom IP, followed by its integration into the Zynq Processing System.

While creating the custom IP and its integration was straightforward, challenges arose due to my misunderstanding of instructions. Luckily, all of my questions were resolved during the second lab session.

Upon completion, the setup was demonstrated to the TA, confirming the correct integration of the custom IP with the Zynq system and the software's successful interaction. The outputs can be seen below:



```
Terminal
File Edit View Search Terminal Help
Value stored in Slave Register 1: 11
The Result of our Multiplication is: 121

Value stored in Slave Register 0: 12
Value stored in Slave Register 1: 12
The Result of our Multiplication is: 144

Value stored in Slave Register 0: 13
Value stored in Slave Register 1: 13
The Result of our Multiplication is: 169

Value stored in Slave Register 0: 14
Value stored in Slave Register 1: 14
The Result of our Multiplication is: 196

Value stored in Slave Register 0: 15
Value stored in Slave Register 1: 15
The Result of our Multiplication is: 225

Value stored in Slave Register 0: 16
Value stored in Slave Register 1: 16
The Result of our Multiplication is: 256
```

IV. Conclusion

In this lab, we employed Vivado to design and integrate a custom IP for integer multiplication within the Zynq Processing System. While the creation and integration of the custom IP were largely successful, there were challenges due to instruction ambiguities. Overcoming these issues emphasized the importance of precise interpretation in complex system designs. Demonstrating the final setup to the TA validated our efforts. Overall, this lab reinforced the complexities of FPGA-based co-design and underlined the significance of detailed execution in both hardware and software aspects.

V. Questions

- a) Multiplying maximum values of 32-bit registers 'slv reg0' and 'slv reg1' (both being $2^{32} - 1$) will produce a result that requires 64 bits. Storing this in a 32-bit 'slv reg2' will lead to "overflow". To correct this, 'slv reg2' should be expanded to 64 bits. In Verilog:

```
reg [63:0] slv_reg2;
```

Adjustments in peripheral logic will also be needed to handle the increased data width.

- b) Yes, it's possible to read the output register before the correct result is available due to propagation delays inherent in digital circuits. When writing values to 'slv reg0' and 'slv reg1' and immediately reading 'slv reg2', the multiplication operation might not have completed yet because of these delays. Ensuring synchronization between writing to input registers and reading from output registers is crucial to avoid such issues.
- c) Yes, changing the interface mode from "Slave" to "Master" will impact the experiment. In the "Slave" mode, the multiply IP waits for commands or data from a controlling entity (Master). If switched to "Master" mode, the multiply IP would actively initiate transactions, which could disrupt the expected data flow and synchronization in the experiment, as the setup and interactions designed in the lab are based on the multiply IP functioning as a slave.

VI. Appendix

helloworld.c:

```
#include <stdio.h>
#include <xparameters.h>
#include <multiply.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"

int main()
{
    init_platform();

    int multiplicationOutput;
    const int BASE_ADDRESS = XPAR_MULTIPLY_0_S00_AXI_BASEADDR;
    const int MAX_VALUE = 16;

    for (int counter = 0; counter <= MAX_VALUE; counter++){
```

```

//Stores the first value to multiply in the first register and print its value
MULTIPLY_mWriteReg(BASE_ADDRESS, 0, counter);
printf("Value stored in Slave Register 0: %d\n", counter);

//Stores the second value to multiply in the second register and print its value
MULTIPLY_mWriteReg(BASE_ADDRESS, 4, counter);
printf("Value stored in Slave Register 1: %d\n", counter);

//Retrieves and print the result from the multiplication operation
multiplicationOutput = MULTIPLY_mReadReg(BASE_ADDRESS, 8);
printf("The Result of our Multiplication is: %d\n\n", multiplicationOutput);
}

//Restores the platform's initial state
cleanup_platform();
return 0;
}

```

multiply.v:

```
`timescale 1 ns / 1 ps
```

```

module multiply_1_v1_0_S00_AXI #
(
    // Users to add parameters here

    // User parameters ends
    // Do not modify the parameters beyond this line

    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH = 4
)
(
    // Users to add ports here

    // User ports ends
    // Do not modify the ports beyond this line

    // Global Clock Signal
    input wire S_AXI_ACLK,
    // Global Reset Signal. This Signal is Active LOW
    input wire S_AXI_ARESETN,
    // Write address (issued by master, accepted by Slave)
    input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
    // Write channel Protection type. This signal indicates the
    // privilege and security level of the transaction, and whether

```

```

// the transaction is a data access or an instruction access.
input wire [2 : 0] S_AXI_AWPROT,
// Write address valid. This signal indicates that the master signaling
// valid write address and control information.
input wire S_AXI_AWVALID,
// Write address ready. This signal indicates that the slave is ready
// to accept an address and associated control signals.
output wire S_AXI_AWREADY,
// Write data (issued by master, accepted by Slave)
input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
// Write strobes. This signal indicates which byte lanes hold
// valid data. There is one write strobe bit for each eight
// bits of the write data bus.
input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
// Write valid. This signal indicates that valid write
// data and strobes are available.
input wire S_AXI_WVALID,
// Write ready. This signal indicates that the slave
// can accept the write data.
output wire S_AXI_WREADY,
// Write response. This signal indicates the status
// of the write transaction.
output wire [1 : 0] S_AXI_BRESP,
// Write response valid. This signal indicates that the channel
// is signaling a valid write response.
output wire S_AXI_BVALID,
// Response ready. This signal indicates that the master
// can accept a write response.
input wire S_AXI_BREADY,
// Read address (issued by master, accepted by Slave)
input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
// Protection type. This signal indicates the privilege
// and security level of the transaction, and whether the
// transaction is a data access or an instruction access.
input wire [2 : 0] S_AXI_ARPROT,
// Read address valid. This signal indicates that the channel
// is signaling valid read address and control information.
input wire S_AXI_ARVALID,
// Read address ready. This signal indicates that the slave is
// ready to accept an address and associated control signals.
output wire S_AXI_ARREADY,
// Read data (issued by slave)
output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
// Read response. This signal indicates the status of the
// read transfer.
output wire [1 : 0] S_AXI_RRESP,
// Read valid. This signal indicates that the channel is
// signaling the required read data.

```

```

        output wire S_AXI_RVALID,
        // Read ready. This signal indicates that the master can
        // accept the read data and response information.
        input wire S_AXI_RREADY
    );

    // AXI4LITE signals
    reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_awaddr;
    reg axi_awready;
    reg axi_wready;
    reg [1 : 0] axi_bresp;
    reg axi_bvalid;
    reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_araddr;
    reg axi_arready;
    reg [C_S_AXI_DATA_WIDTH-1 : 0] axi_rdata;
    reg [1 : 0] axi_rresp;
    reg axi_rvalid;

    // Example-specific design signals
    // local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
    // ADDR_LSB is used for addressing 32/64 bit registers/memories
    // ADDR_LSB = 2 for 32 bits (n downto 2)
    // ADDR_LSB = 3 for 64 bits (n downto 3)
    localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
    localparam integer OPT_MEM_ADDR_BITS = 1;
    //-----
    //-- Signals for user logic register space example
    //-----
    //-- Number of Slave Registers 4
    reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg0;
    reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1;
    reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2;
    reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg3;
    wire slv_reg_rden;
    wire slv_reg_wren;
    reg [C_S_AXI_DATA_WIDTH-1:0] reg_data_out;
    integer byte_index;
    reg aw_en;

    // I/O Connections assignments

    assign S_AXI_AWREADY = axi_awready;
    assign S_AXI_WREADY = axi_wready;
    assign S_AXI_BRESP = axi_bresp;
    assign S_AXI_BVALID = axi_bvalid;
    assign S_AXI_ARREADY = axi_arready;
    assign S_AXI_RDATA = axi_rdata;
    assign S_AXI_RRESP = axi_rresp;

```

```

assign S_AXI_RVALID      = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_awready <= 1'b0;
            aw_en <= 1'b1;
        end
    else
        begin
            if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
                begin
                    // slave is ready to accept write address when
                    // there is a valid write address and write data
                    // on the write address and data bus. This design
                    // expects no outstanding transactions.
                    axi_awready <= 1'b1;
                    aw_en <= 1'b0;
                end
            else if (S_AXI_BREADY && axi_bvalid)
                begin
                    aw_en <= 1'b1;
                    axi_awready <= 1'b0;
                end
            else
                begin
                    axi_awready <= 1'b0;
                end
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_awaddr <= 0;
        end
    else
        begin

```

```

    if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
    begin
        // Write Address latching
        axi_awaddr <= S_AXI_AWADDR;
    end
end

// Implement axi_wready generation
// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
    begin
        axi_wready <= 1'b0;
    end
    else
    begin
        if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID && aw_en )
        begin
            // slave is ready to accept write data when
            // there is a valid write address and write data
            // on the write address and data bus. This design
            // expects no outstanding transactions.
            axi_wready <= 1'b1;
        end
        else
        begin
            axi_wready <= 1'b0;
        end
    end
end

// Implement memory mapped register select and write logic generation
// The write data is accepted and written to memory mapped registers when
// axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
strokes are used to
// select byte enables of slave registers while writing.
// These registers are cleared when reset (active low) is applied.
// Slave register write enable is asserted when valid address and data are available
// and the slave is ready to accept the write address and write data.
assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready &&
S_AXI_AWVALID;

always @( posedge S_AXI_ACLK )

```



```

begin
  if ( S_AXI_ARESETN == 1'b0 )
  begin
    slv_reg0 <= 0;
    slv_reg1 <= 0;
    //slv_reg2 <= 0;
    slv_reg3 <= 0;
  end
else begin
  if (slv_reg_wren)
  begin
    case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
      2'h0:
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
          if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 0
            slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
          end
        2'h1:
          for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 1
              slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
          2'h2:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 2
                //slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              end
            2'h3:
              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                  // Respective byte enables are asserted as per write strobes
                  // Slave register 3
                  slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                end
            default : begin
              slv_reg0 <= slv_reg0;
              slv_reg1 <= slv_reg1;
              // slv_reg2 <= slv_reg2;

```

```

        slv_reg3 <= slv_reg3;
    end
endcase
end
end
end

// Implement write response logic generation
// The write response and response valid signals are asserted by the slave
// when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
// This marks the acceptance of address and indicates the status of
// write transaction.

always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
    begin
        axi_bvalid <= 0;
        axi_bresp <= 2'b0;
    end
    else
    begin
        if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
S_AXI_WVALID)
        begin
            // indicates a valid write response is available
            axi_bvalid <= 1'b1;
            axi_bresp <= 2'b0; // 'OKAY' response
        end
        // work error responses in future
    else
    begin
        if (S_AXI_BREADY && axi_bvalid)
            //check if bready is asserted while bvalid is high)
            //(there is a possibility that bready is always asserted high)
            begin
                axi_bvalid <= 1'b0;
            end
        end
    end
end
end

// Implement axi_arready generation
// axi_arready is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_arready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.

```

```

always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_arready <= 1'b0;
            axi_araddr <= 32'b0;
        end
    else
        begin
            if (~axi_arready && S_AXI_ARVALID)
                begin
                    // indicates that the slave has accepted the valid read address
                    axi_arready <= 1'b1;
                    // Read address latching
                    axi_araddr <= S_AXI_ARADDR;
                end
            else
                begin
                    axi_arready <= 1'b0;
                end
            end
        end
    end
end

// Implement axi_arvalid generation
// axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_ARVALID and axi_arready are asserted. The slave registers
// data are available on the axi_rdata bus at this instance. The
// assertion of axi_rvalid marks the validity of read data on the
// bus and axi_rresp indicates the status of read transaction. axi_rvalid
// is deasserted on reset (active low). axi_rresp and axi_rdata are
// cleared to zero on reset (active low).
always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_rvalid <= 0;
            axi_rresp <= 0;
        end
    else
        begin
            if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
                begin
                    // Valid read data is available at the read data bus
                    axi_rvalid <= 1'b1;
                    axi_rresp <= 2'b0; // 'OKAY' response
                end
            else if (axi_rvalid && S_AXI_RREADY)
                begin

```

```

        // Read data is accepted by the master
        axi_rvalid <= 1'b0;
    end
end
end

// Implement memory mapped register select and read logic generation
// Slave register read enable is asserted when valid address is available
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0 : reg_data_out <= slv_reg0;
        2'h1 : reg_data_out <= slv_reg1;
        2'h2 : reg_data_out <= slv_reg2;
        2'h3 : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
    endcase
end

// Output register or memory read data
always @( posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
    begin
        axi_rdata <= 0;
    end
    else
    begin
        // When there is a valid read address (S_AXI_ARVALID) with
        // acceptance of read address by the slave (axi_arready),
        // output the read data
        if (slv_reg_rden)
        begin
            axi_rdata <= reg_data_out;    // register read data
        end
    end
end

// Add user logic here

// User logic ends

endmodule

```