

ECEN 449 - Lab Report

Lab Number: 3

Lab Title: Creating a Custom Hardware IP and Interfacing it with Software

Section Number: 511

Student's Name: Kris Gavvala

Student's UIN: 929000158

Date: 3/5/2025

TA: Gautham Nemani

Introduction:

This lab is an instructional walkthrough on creating and importing a customIP module for a Zynq Processing System based system. The 'Create and Package IP' feature in Vivado is utilized to develop a custom integer multiplication peripheral. The peripheral will be then integrated into a microprocessor system and software will be used to interact with it using the SDK.

Procedure:

Create Zynq Base System

Create a new project in Vivado and select hardware using the 'Board' tab. Check connectivity to the board and proceed onto the next step.

Create the new block design

Create a new block design 'Multiply' -> add IP -> add ZYNQ7Processing System' IP

Re-customize the processor IP just generated and import the .tcl file from the Tamu Ecen directory to presets-> apply configuration. Uncheck all peripheral I/O pins. Edit the IP and enable the UART peripheral I/O pin, making sure it is the only one enabled.

Create the Multiply IP

Click on 'Tools' -> 'Create and Package New IP' -> 'Create a new AXI4 peripheral' and click on 'Next'. Select the parameters for the peripheral interface:

Interface type : Lite

Interface mode: slave

Data Width: 32

Number of Registers: 4

Select the 'edit IP' option and click next

IP logic function implementation

open the 'multiply v1 0 S00 AXI.v' file from the sources tab. Comment out all code that writes to slv_reg2 register. Towards the bottom of the file, add code for the multiplier logic given in figure 1 that does the following:

```
// on a positive clock edge
```

```
    // if reset is low assign slv_reg2 and tmp_reg to 0
```

```
    // else assign tmp_reg to the product of slv_reg0 and slv_reg1 and assign slv_reg2 to tmp_reg
```

Re-package the IP and close the project.

Add the newly created IP to the processor system block diagram, click 'Run Connection Automation' -> regenerate layout

After creating the HDL wrapper and generating the bitstream, launch the Vitis IDE tool and create the multiply_test project from the generated .xsa file.

Edit the helloworld.c file

to write to the slv1 and slv0 registers and read their product from the slv2 register:

- First get the base address of the AXI port on the Multiply IP from the xparameters.h file
- Read and write functions for Multiply are declared in Multiply.h
- Since each of the registers are 32 bits and the slv_0, slv_1, slv_2 memory is contiguous, we use base_address + offset 0 to write to the address of slv_0, base_address + offset 4 to write to slv_1, and base_address + offset 8 to read from the slv_2 address.
- Using a for loop, different values are written to the slave 0 and 1 addresses and the result is read from the slave 2 address using the functions in multiply.h

Results:

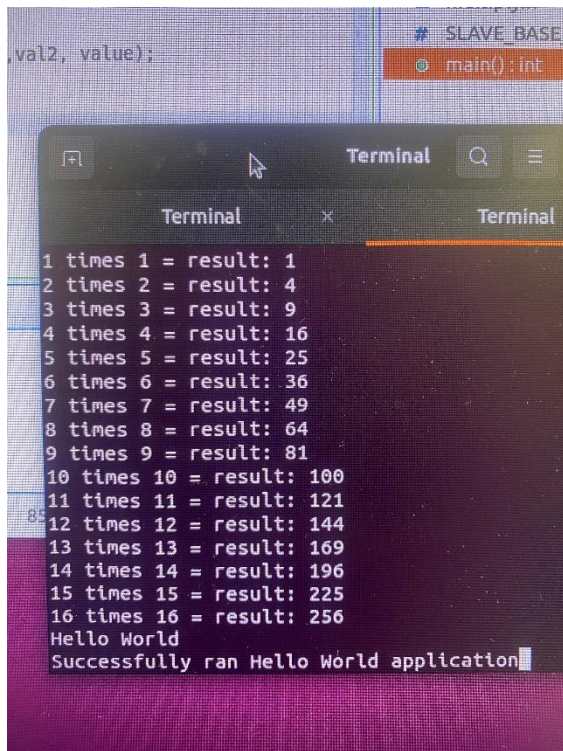
After programming the processing system with the helloworld.c code, the computation from Multiply could be read from the terminal showing the result of slave 0 * slave 1 as the squares from 1 – 16 ($2*2 = 4$, $3*3 = 9$, $4*4 = 16$)

Conclusion:

This lab gave a good tutorial in creating a custom hardware IP and testing it with software. It also gave more practice using the xparameters.h file to find correct read/write addresses and using c language to program the processor system.

Post-lab Deliverables:

output of the terminal (picocom).



```
val2, value);
# SLAVE_BASE
main():int

Terminal
Terminal

1 times 1 = result: 1
2 times 2 = result: 4
3 times 3 = result: 9
4 times 4 = result: 16
5 times 5 = result: 25
6 times 6 = result: 36
7 times 7 = result: 49
8 times 8 = result: 64
9 times 9 = result: 81
10 times 10 = result: 100
11 times 11 = result: 121
12 times 12 = result: 144
13 times 13 = result: 169
14 times 14 = result: 196
15 times 15 = result: 225
16 times 16 = result: 256
Hello World
Successfully ran Hello World application
```

[6 points.] Answers to the following questions:

(a) Recall that 'slv reg0', 'slv reg1', and 'slv reg2' are all 32-bit registers. What values of 'slv reg0' and 'slv reg1' would produce incorrect results from the multiplication block? What is the name commonly assigned to this type of computation error, and how would you correct this? Provide a Verilog example and explain what you would change during the creation of the corrected peripheral.

Answer: if the values of the registers were larger than 32 bits, then the results would be incorrect because the number could overflow and change other values in slv_ addresses.

The error is overflow. To fix this you could increase the bit width of the operand and result registers

```
reg [63:0] tmp_reg;           // 64-bit temporary register
reg [63:0] slv_2;             // Store full 64-bit result
```

(b) In this exercise, we wrote the multiplier and the multiplicand to the input registers, followed by a read from the output register for the multiplication result. Is it possible that we end up reading the output register before the correct result is available? Why?

Answer:

Yes, it is possible. There is no signal that indicates if a result from multiplication is valid or not. The result could be either 0 after reset or from a previous computation.

(c) While creating the multiply IP, we kept the interface mode as "Slave". Suppose we change this mode to "Master", do you think it will impact our experiment? Why?

Answer: Yes it will impact it. With Multiply being master, it would need to handle all data read and write to slv_1, slv_2, slv_0. It would need its own logic to handle that.

APPENDIX

multiply_v1_0_S00_AXI

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

```
module multiply_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

// 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
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 strobes are used to


```

        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
    end
    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

// 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;
    end
end

```

```

        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

reg[0:C_S_AXI_DATA_WIDTH - 1] tmp_reg;    // initialize temporary register
always @(posedge S_AXI_ACLK) begin        //on every rising clock signal
    if( S_AXI_ARESETN == 1'b0) begin       //check if global reset signal is high
        slv_reg2 <=0;                      //reset the temporary and slave registers
    end
end

```

```

        tmp_reg <= 0;
    end
    else begin
        //if rest is not high
        tmp_reg <= slv_reg0 * slv_reg1;    //multiply values in slave 1 and slave 0 and write to
temporary register
        slv_reg2 <= tmp_reg;                //write from temporary register to slave 2
    end

end

// User logic ends

Endmodule

```

Helloworld.c

```

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

#define SLAVE_BASE_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR

// declare the base address for slv_0, 1,2 of Multiply peripheral

int main()

```

```

{
    init_platform();

    //write to slv1 and slv0

    u32 input; // input to slv_0, slv_1
    for ( input = 0x00000000; input < 0x00000011; input++ ){
        // write to slv_0 and slv_2
        MULTIPLY_mWriteReg(SLAVE_BASE_ADDR, 0, input);
        MULTIPLY_mWriteReg(SLAVE_BASE_ADDR, 4, input);
        //read slv_0 and 1 values to print them later
        u32 val1 = MULTIPLY_mReadReg(SLAVE_BASE_ADDR, 0);
        u32 val2 = MULTIPLY_mReadReg(SLAVE_BASE_ADDR, 4);

        //read from slv_reg2
        u32 value = MULTIPLY_mReadReg(SLAVE_BASE_ADDR, 8);

        xil_printf("%u times %u = result: %u\n\r", val1, val2, value); // print results
    }
    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}

```