

Lab 3: Creating a Custom Hardware IP and Interfacing it with Software

Alana Ordonez
UIN: 428008825
ECEN 449 506
Date: October 7, 2022

I. Introduction

We did this lab to have experience with creating a custom IP module, specifically for a Zynq Processing System based system. It was meant to reinforce our skills with creating block diagrams in Vivado and transferring the bitstream to the SDK, where we could use our software knowledge to create a multiplier.

II. Procedure

A block design named “multiply” was created to include the ZYNQ7 Processing System and customized it by importing `ZYBO_Z7_B2.tcl`, a file to add preset configurations for the processing system, and changing the peripheral I/O pins to only use UART1. A new AXI4 peripheral, also named “multiply,” was created and changed such that all the code within `multiply_v1_0_S00_AXI.v` does not allow the third software register, our output, to write. User logic was added to the end of the file such that `tmp_reg` temporarily holds the result of the multiplication and then loads it into `slv_reg2`. This allowed for the new “multiply” IP to be added to the block diagram before the HDL wrapper was created and the bitstream was generated.

After launching this in the SDK, `helloworld.c` was modified to ultimately write values to `slv_reg0` and `slv_reg1` then store the values into `slv_reg2`, printing each register’s values in the terminal. The file was written such that the values being passed as factors are numbers 0 through 10, and the values are being multiplied by itself (so squaring the numbers). The ZYBO Z7-10 was then programmed with the bitstream file, and the results were outputted to the terminal using ‘picocom.’

III. Results

Unlike the previous labs, there were no visual indications on the ZYBO Z7-10 that the code was functioning properly (past syntax and linkage errors). The results were outputted on the terminal, first displaying what value is stored in `slv_reg0`, then `slv_reg1`, then what the product of the two values is. The first number is 0, and it stops at 10.

Based on this output versus the intention of the code, I believe that the code functions correctly, as it displays all the squares of integers 0 through 10.

IV. Conclusion

Aside from building on the foundation of our Vivado user knowledge, we were able to get more experience with creating our own custom IPs, using the SDK, and programming software. It was different from previous labs such that we didn’t have to include an `.xdc` file to configure the physical components of the ZYBO Z7-10 to encourage an output. Instead, we created a multiplier using a custom IP and were able to view the results printed to the terminal.

V. Questions

- (a) *Recall that 'slv_reg0', 'slv_reg1', and 'slv_reg2' are all 32-bit integers. 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.*

If *slv_reg0* and *slv_reg1* were greater than 32 bit integers, or if one were the maximum 32-bit integer and the other were a digit greater than 1, or if the product of the two ended up creating a number larger than 32 bits, then the common type of computation error that would arise would be an overflow error. To correct this, the size of the registers could be changed from 32 bits to 64 bits during the creation of the peripheral with a Verilog implementation such as the following:

```
[63:0] slv_reg0;  
[63:0] slv_reg1;  
[63:0] slv_reg2;
```

- (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?*

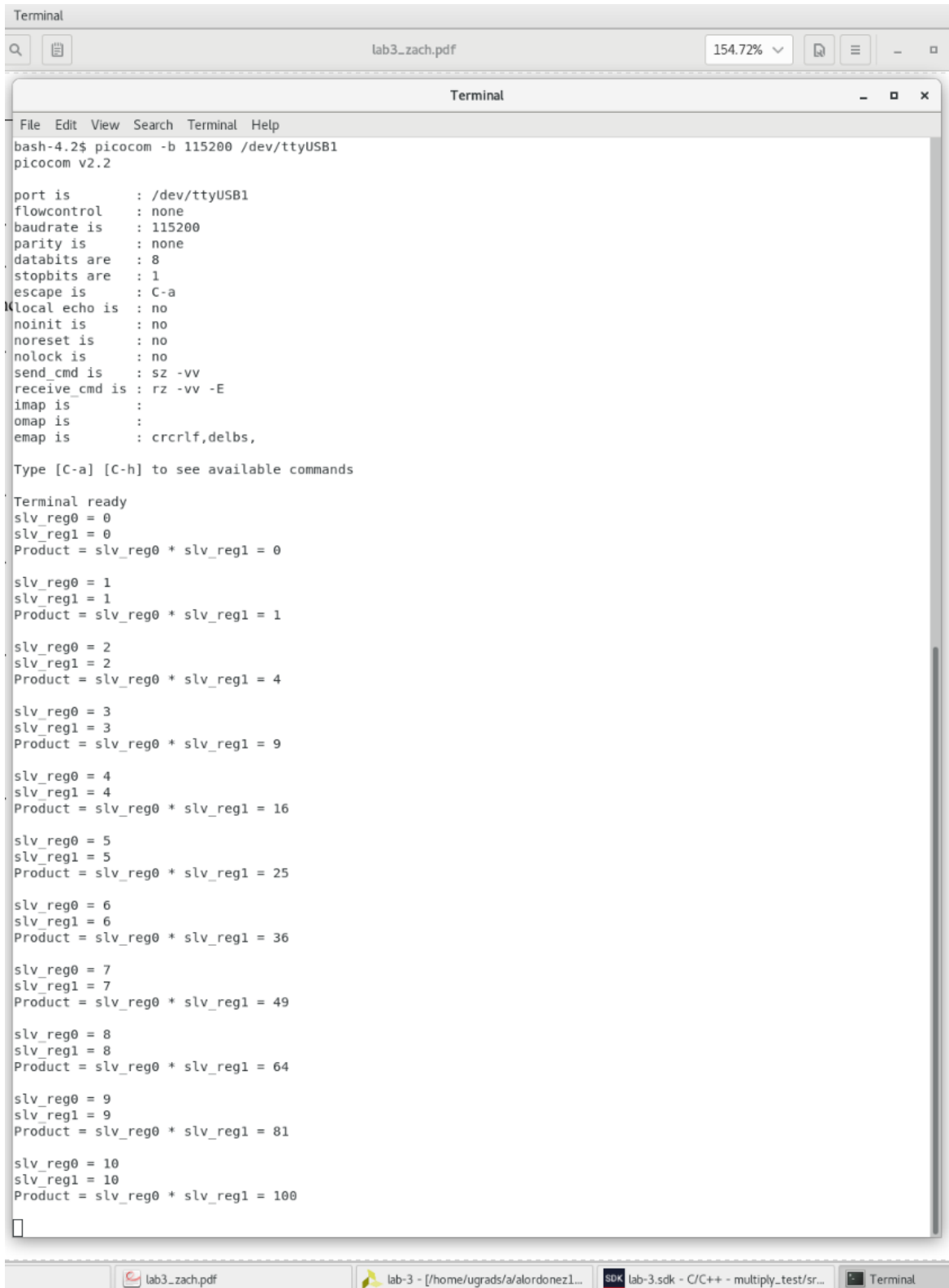
That might happen if one of the factors was not written correctly, so it might not properly enter the register by the time the output register is read. But because the commands execute in order, and if all the values entered are correct and do not exceed 32 bits, then we don't really expect to see that very often.

- (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?*

No, I don't think it will. Both Master and Slave can read input registers and write to output registers, and since our program isn't very complex that requires the Master to drive any of these transactions, our output for this lab is essentially the same.

VI. Appendix

Output



```
Terminal
lab3_zach.pdf 154.72%
Terminal
File Edit View Search Terminal Help
bash-4.2$ picocom -b 115200 /dev/ttyUSB1
picocom v2.2

port is       : /dev/ttyUSB1
flowcontrol   : none
baudrate is   : 115200
parity is     : none
databits are  : 8
stopbits are  : 1
escape is     : C-a
local echo is : no
noinit is     : no
noreset is    : no
nolock is     : no
send_cmd is   : sz -vv
receive_cmd is : rz -vv -E
imap is       :
omap is       :
emap is       : crcrlf,delbs,

Type [C-a] [C-h] to see available commands

Terminal ready
slv_reg0 = 0
slv_reg1 = 0
Product = slv_reg0 * slv_reg1 = 0

slv_reg0 = 1
slv_reg1 = 1
Product = slv_reg0 * slv_reg1 = 1

slv_reg0 = 2
slv_reg1 = 2
Product = slv_reg0 * slv_reg1 = 4

slv_reg0 = 3
slv_reg1 = 3
Product = slv_reg0 * slv_reg1 = 9

slv_reg0 = 4
slv_reg1 = 4
Product = slv_reg0 * slv_reg1 = 16

slv_reg0 = 5
slv_reg1 = 5
Product = slv_reg0 * slv_reg1 = 25

slv_reg0 = 6
slv_reg1 = 6
Product = slv_reg0 * slv_reg1 = 36

slv_reg0 = 7
slv_reg1 = 7
Product = slv_reg0 * slv_reg1 = 49

slv_reg0 = 8
slv_reg1 = 8
Product = slv_reg0 * slv_reg1 = 64

slv_reg0 = 9
slv_reg1 = 9
Product = slv_reg0 * slv_reg1 = 81

slv_reg0 = 10
slv_reg1 = 10
Product = slv_reg0 * slv_reg1 = 100


```

helloworld.c

```
system.hdf system.mss xil_io.h xil_printf.h stdio.h xparameters.h multiply_selftest.c multiply_test.elf helloworld.c ⌵

/* Copyright (C) 2009 - 2014 Xilinx, Inc. All rights reserved. */

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * -----
 * | UART TYPE   BAUD RATE                                |
 * -----
 * | uarts550    9600
 * | uartlite    Configurable only in HW design
 * | ps7_uart    115200 (configured by bootrom/bsp)
 */

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

/*
 * Alana Ordonez
 * UIN 428008835
 *
 * Code Description:
 * This code creates a multiplier that squares integers 0 through 10 and outputs them to the console.
 */

int main()
{
    init_platform();

    int product;
    int i;

    for (i = 0; i <= 10; i++) {
        // #define MULTIPLY_mWriteReg(BaseAddress, ReOffset, Data)

        MULTIPLY_mWriteReg(XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 0, i); // first factor
        printf("slv_reg0 = %d\n", i);
        MULTIPLY_mWriteReg(XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 4, i); // second factor
        printf("slv_reg1 = %d\n", i);
        product = MULTIPLY_mReadReg(XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 8); // product of two factors
        printf("Product = slv_reg0 * slv_reg1 = %d\n\n", product);
    }

    cleanup_platform();
    return 0;
}
```

multiply_v1_0_S00_AXI.v

object/multiply_v1_0_project.xpr] - Vivado 2018.3

multiply_v1_0_project - [/home/ugrads/a/alordonez1/lab-3/lab-3.tmp/multiply_v1_0_project/multiply_v1_0_project.xpr] - Vivado 2018.3

Project Summary x Package IP - multiply x multiply_v1_0_S00_AXI.v x

/home/ugrads/a/alordonez1/p_repo/multiply_1.0/hdl/multiply_v1_0_S00_AXI.v

🔍 🏠 ⬅ ➡ ✂ 📄 📁 ❌ // 📖 ?

```
363 : end
364 :
365 : // Implement memory mapped register select and read logic generation
366 : // Slave register read enable is asserted when valid address is available
367 : // and the slave is ready to accept the read address.
368 : assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
369 : always @(*)
370 : begin
371 : // Address decoding for reading registers
372 : case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
373 : 2'h0 : reg_data_out <= slv_reg0;
374 : 2'h1 : reg_data_out <= slv_reg1;
375 : 2'h2 : reg_data_out <= slv_reg2;
376 : 2'h3 : reg_data_out <= slv_reg3;
377 : default : reg_data_out <= 0;
378 : endcase
379 : end
380 :
381 : // Output register or memory read data
382 : always @(posedge S_AXI_ACLK)
383 : begin
384 : if ( S_AXI_ARESETN == 1'b0 )
385 : begin
386 : axi_rdata <= 0;
387 : end
388 : else
389 : begin
390 : // When there is a valid read address (S_AXI_ARVALID) with
391 : // acceptance of read address by the slave (axi_arready),
392 : // output the read data
393 : if (slv_reg_rden)
394 : begin
395 : axi_rdata <= reg_data_out; // register read data
396 : end
397 : end
398 : end
399 :
400 : // Add user logic here
401 :
402 : reg [0:C_S_AXI_DATA_WIDTH - 1] tmp_reg;
403 : always @(posedge S_AXI_ACLK) begin
404 : if (S_AXI_ARESETN == 1'b0) begin
405 : slv_reg2 <= 0;
406 : tmp_reg <= 0;
407 : end
408 : else begin
409 : tmp_reg <= slv_reg0 * slv_reg1;
410 : slv_reg2 <= tmp_reg;
411 : end
412 : end
413 :
414 : // User logic ends
415 :
416 : endmodule
417 :
```