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

Samuel Huang
ECEN 449

## 1. Introduction

This lab introduces the process of creating and importing custom IP modules for Zynq Processing System based systems. Through creating a Package IP in Vivado, the lab develops a custom peripheral for integer multiplication. This package then gets integrated into a microprocessor system, in which developed software interacts by driving the input, and reading the output of the peripheral. Ultimately this lab demonstrates a hardware and software co-design, by creating an integer multiplication peripheral which is driven by a c language program.

## 2. Procedure

The first part of this lab sets up the environment to create a multiplication peripheral. By creating a custom peripheral that sets up an AXI lite slave that contains 4 x 32-bit read/write registers, the lab then adds the appropriate Verilog code to drive the hardware logic of the system. With this, the template hardware peripheral for the multiplication peripheral has functionality that can be added to the PS system through exporting the design and bitstream to the FPGA.

Once the PS system has generated the layout of the custom multiplication peripheral, software can be applied to drive the input, and read the output. Software driven values are written to the two input slave registers , processed at the hardware level, written to the output slave register, and then read by the software. For the software to function properly it must locate the memory addresses where the multiplication peripherals are stored. Using functions found in the included in xparameters.h and multiply.h, slave register 0 and 1 can be written to and the result in slave register 2 can be read. With this the for-loop writing different values from 0 to 16 for each input register can be implemented.

## 3. Results

The first part of the lab creates a custom peripheral which performs integer multiplication of two input slave registers and writes the result into an output slave register. This Peripheral is the hardware component of the lab, which gets mapped onto the FPGA. The software component of the lab creates a software program which uses the base address of the peripheral and a for-loop to write values from 0-16 into each of the input slave registers, and subsequently reads the product from the output slave register. Ultimately this lab creates a software/hardware codesign to create an integer multiplier.

## 4. Conclusion

This lab taught me how to build a custom peripheral and apply it to a generated block design in Vivado. It showed me how to customize the base peripheral to apply the desired hardware logic to it. The process showed where the Vivado code that drives the logic of the hardware is placed in the system, and how to specify which registers get applied. The lab not just taught me how to create the hardware functionality of the system, but further showed how software can be used to drive it.

The lab also taught me where to look to include the necessary functions, to properly connect the software to the hardware. The software part of the lab taught me how to read values into and out of registers on the peripheral. Ultimately the lab showed me how to create a software and hardware co-design system that establishes the functionality in the hardware which then gets driven by a software component.

## 5. Questions

Since the maximum size that the output slave register ‘slvreg2’ can store is 32-bits which is equal to 2,147,483,647. Any input value into the ‘slvreg0’ and ‘slvreg1’, whose product exceeds this value would result in  overflow which would create an inaccurate result. The easiest way to fix this would be creating a check in the Verilog that sees if the product of the input registers result in an overflow, and sets a flag value if it does. To ensure that the peripheral works with this change, it would have to be modified to handle the overflow tag by writing to the fourth register.

```
reg [0:(2*C_S_AXI_DATA_WIDTH)-1] product;
 // 64-bit register for multiplication to handle potential overflow


 always @(posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0) begin
       slvreg2 <= 0;           // reset slvreg2 to 0
       product <= 0;           // reset product to 0
       slvreg3 <= 0;           // reset slvreg3 to 0
    end
    else begin
       // perform multiplication and check for overflow
       product = slvreg0 * slvreg1;
       if (product > 32'h7FFFFFFF) begin  // check if result exceeds 2,147,483,647
          slvreg3 <= 1;  // Set overflow flag
       end
       else begin
          slvreg2 <= product;   // update slvreg2 with the valid product
          slvreg3 <= 0;         // clear overflow flag
```

```
        end
      end
    end
```

Both the multiplier and multiplicand are written and read at the same time, since they use non-blocking assignments. They both occur during the same clock cycle, so if the multiplication takes longer than a single clock cycle the read output will be the old product. This means the correct result would not be available, and the read value is wrong.

Changing from "Slave" to "Master" would impact the experiment. We used an AXI Slave Interface mode since it only responds to requests from the software, which writes and reads from the registers. An AXI Master Interface would mean that the Multiply IP would be responsible for writing and reading its own input and outputs, making it autonomous. This would mean that the software would no longer be the control flow of the system.

## 6. Appendix

Lab4.c

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

#define BASE_ADDR ((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR) // gets
the base address to properly orient

#define SLV_REG0_OFFSET 0x00 // offset of 0 since this is the first
reg
#define SLV_REG1_OFFSET 0x04 // offset of 4 bytes for the 32 bit reg
#define SLV_REG2_OFFSET 0x08 // offset of 8 bytes for the 32 bit reg
following the first offset

int main() {
    // Initialization and cleanup
    init_platform(); // initializes board
    cleanup_platform(); // prepare board for new application


    u32 value0, value1, result;


    // Loop through values 0 to 16 on both Reg0 and Reg1 for full
testing
    for (value0 = 0; value0 <= 16; value0++) {
        for (value1 = 0; value1 <= 16; value1++) {
            // set reg0 to value0 and reg1 to value1
            MULTIPLY_mWriteReg(BASE_ADDR, SLV_REG0_OFFSET, value0);
            MULTIPLY_mWriteReg(BASE_ADDR, SLV_REG1_OFFSET, value1);
            printf("Written values: slv_reg0 = %u, slv_reg1 = %u\n",
value0, value1);
            // multiplication occurs at hardware level
            // result is what is read from reg2
            result = MULTIPLY_mReadReg(BASE_ADDR, SLV_REG2_OFFSET);
            printf("Multiplication result stored in slv_reg2: %u\n",
result);
        }
    }
    // prepare board for new application
    cleanup_platform();


    return 0;
}
```