ECEN 749 LAB REPORT

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

Tanmay Verma

09.21.2017

INTRODUCTION

The purpose of this lab is to familiarize you with the process of creating and importing a custom IP module for a Zynq Processing System based system. We will be using the 'Create and Package IP' in Vivado to develop a custom peripheral for performing integer multiplication. We will then integrate the integer multiplication peripheral into a microprocessor system and develop software to interact with the peripheral using the SDK. This lab serves as a simple hardware/software co-design example.

PROCEDURE

The following steps were carried out:

1. A new block design was created in the project.
2. In the diagram, 'ZYNQ7 Processing System' IP was added.
3. Provided xml file was used to re-configure the PS IP.
4. Only UART1 was "enabled" in the Peripheral I/O.
5. A new AXI4 Peripheral named "multiply" was created.
6. The user logic for multiplication was added in the "multiply" block.
7. The peripheral was imported in the PS block design.
8. Connection Automation was run and the layout was regenerated.
9. HDL wrapper was created and bitstream was generated.
10. The bitstream was generated and SDK was launched.
11. A C code was written to write values to the registers 'slv_reg0' and 'slv_reg1' and read the multiplication result from 'slv_reg2'.
12. The program was then run on PS and picocom on ttyUSB1 was used to monitor the output.

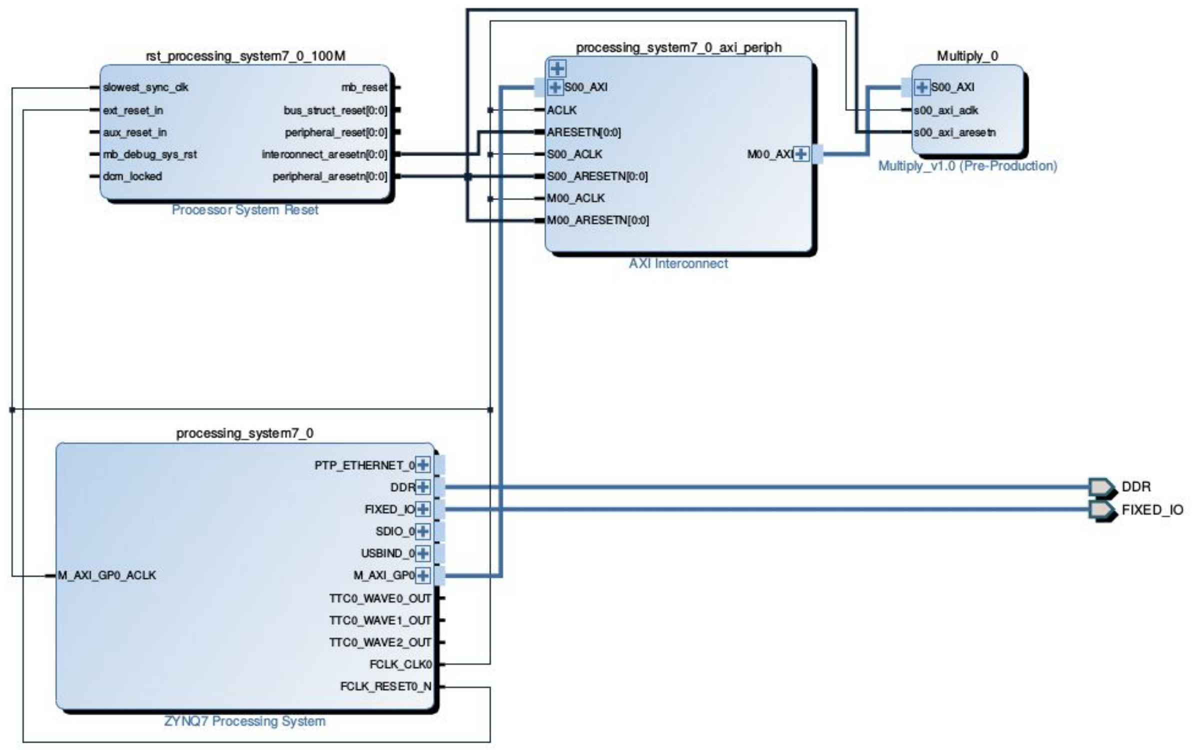
RESULTS

The software on PS interacted with the multiply IP designed in verilog to successfully implement Multiply operation on the two numbers. The C code wrote to the two input registers of Multiply IP. The IP hardware written in verilog multiplied the two values and stored the result in output register. The C code then read the values at the three registers and printed their values.

CONCLUSION

The exercise provided a very useful introductory example of Hardware/Software Co-design where a part operation was carried out on PS and rest on the custom IP.

Design



Verilog Code for Custom IP

Only the User Logic portion is shown as requested: all statement s writing to slv_reg3 were commented out.

```
// Add user logic here


    reg [0:C_S_AXI_DATA_WIDTH-1] tmp_reg;

    always @(posedge S_AXI_ACLK) begin

            if(S_AXI_ARESETN == 1'b0) begin         //Resets the output and temp register when reset is
enabled.

            slv_reg2 <= 0;

            tmp_reg <= 0;

            end

            else begin

            tmp_reg <= slv_reg0 * slv_reg1;         //calculates the value of product.

            slv_reg2 <= tmp_reg;

            end

            end

    // User logic ends
```

C code

```
#include "xil_io.h"

#include <stdio.h>

#include <stdint.h>

#include "platform.h"

#include "xparameters.h"

#include "Multiply.h"


#define COUNT    17  //The upper limit on the input value

#define WAIT_TIME 10000000

/* This function is used to add a delay */

int delay()

{

   volatile int temp = 0;

   while(temp < WAIT_TIME) {

             temp++;

   }

   return(0);

}


int main()

{

            init_platform();

            int i;

            u32 count = 0;


            u32 reg_0;

            u32 reg_1;
            u32 reg_2;


            while(count <  COUNT) //Will keep on repeating the multiplication operation for operands in
[0, 1, 2, 3 … 16]

            {

             delay();

/*Writes  count value on reg_0 */

                        MULTIPLY_mWriteReg((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR,
(unsigned)MULTIPLY_S00_AXI_SLV_REG0_OFFSET, count);


/*Writes count value on reg_1 */

                        MULTIPLY_mWriteReg((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR,
(unsigned)MULTIPLY_S00_AXI_SLV_REG1_OFFSET, count);


/* Reads the values  all the three registers, with the third holding the results*/

            reg_0 = MULTIPLY_mReadReg((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR,
(unsigned)MULTIPLY_S00_AXI_SLV_REG0_OFFSET); //read operand1  from reg 0

            reg_1 = MULTIPLY_mReadReg((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR,
(unsigned)MULTIPLY_S00_AXI_SLV_REG1_OFFSET); //read operand2  from reg 1

            reg_2 = MULTIPLY_mReadReg((u32)XPAR_MULTIPLY_0_S00_AXI_BASEADDR,
(unsigned)MULTIPLY_S00_AXI_SLV_REG2_OFFSET); //read result  from reg 2


            printf(" reg_0 = %d \t reg_1= %d \t result(reg_3) %d \n ", (int)reg_0, (int)reg_1, (int)reg_2);

            count++;


            cleanup_platform();

            return 0;

}
```

PICOCOM OUTPUT

```
reg_0 = 0     reg_1= 0     result(reg_3) 0
reg_0 = 1     reg_1= 1     result(reg_3) 1
reg_0 = 2     reg_1= 2     result(reg_3) 4
reg_0 = 3     reg_1= 3     result(reg_3) 9
reg_0 = 4     reg_1= 4     result(reg_3) 16
reg_0 = 5     reg_1= 5     result(reg_3) 25
reg_0 = 6     reg_1= 6     result(reg_3) 36
reg_0 = 7     reg_1= 7     result(reg_3) 49
reg_0 = 8     reg_1= 8     result(reg_3) 64
reg_0 = 9     reg_1= 9     result(reg_3) 81
reg_0 = 10     reg_1= 10     result(reg_3) 100
reg_0 = 11     reg_1= 11     result(reg_3) 121
reg_0 = 12     reg_1= 12     result(reg_3) 144
reg_0 = 13     reg_1= 13     result(reg_3) 169
reg_0 = 14     reg_1= 14     result(reg_3) 196
reg_0 = 15     reg_1= 15     result(reg_3) 225
reg_0 = 16     reg_1= 16     result(reg_3) 256
```

QUESTIONS

(a) What is the purpose of the tmp_reg from the Verilog code provided in lab, and what happens if this register is removed from the code?

The tmp_reg is added in the critical path to prevent setup violation of slv_reg2.If slv_reg2 is directly connected with the combinatorial logic of multiplier then it is highly like that a intermediary erroneous data gets registered on the output.

(b) 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.

For large values of slv_reg0 and slv_reg1, the result can exceed 32bit size on slv_reg3 loss of data beyond 32bits. The result read is erroneous.

Such an error is called an OVERFLOW. If they both are 32bit of size then their multiplication will result in 64bit value at max. We can use both slv_reg2 and slv_reg3 to store the result, where slv_reg2 stores lower 32bits and slv_reg3 stores upper 32bits.

```
reg [0: 2*C_S_AXI_DATA_WIDTH-1] tmp_reg;

always @(posedge S_AXI_ACLK) begin

        if(S_AXI_ARESETN == 1'b0) begin        //Resets the output and temp register when reset is
enabled.

        slv_reg2 <= 0;

        tmp_reg <= 0;

        end

        else begin

        tmp_reg <= slv_reg0 * slv_reg1;        //calculates the value of product.

        slv_reg2 <= tmp_reg[C_S_AXI_DATA_WIDTH-1:0];

        slv_reg3<= tmp_reg[2*C_S_AXI_DATA_WIDTH-1:C_S_AXI_DATA_WIDTH];

        end

        end

// User logic ends
```