ECEN 749

Laboratory Exercise #3

Creating a Custom Hardware IP and Interfacing it with Software

Name: Pranav Anantharam

UIN: 734003886

Course: ECEN 749 Microprocessor System Design

Section: 603

Introduction:

The third laboratory exercise was designed to introduce us to the process of creating and importing custom IP modules for Zynq Processing System based systems using Vivado. We were tasked with developing a custom peripheral for integer multiplication, integrating it into a microprocessor system, and creating software to interact with it using the SDK. Through this practical exercise, we gained valuable insight into hardware/software co-design, a fundamental aspect of building specialized embedded systems. This skill is vital across industries requiring tailored hardware solutions, equipping us with essential knowledge for tackling real-world engineering tasks.

Procedure:

1) Create a working directory in the home folder.
2) Load the environment variables on the workstation using appropriate commands and open Vivado 2022.1.
3) Create a new project. Select 'Boards' and choose the 'Zybo Z7-10' board option.
4) Create a new block design and name the design 'multiply'.
5) Right click within the diagram tab and add the 'ZYNQ7 Processing System' IP to our design.
6) Download the 'ZYBO_Z7_B2_.tcl' file.
7) Open the 'Re-customize IP' window and import the TCL file. Uncheck all the peripheral I/O pins.
8) Enable only the UART 1 pin. Select 'Tools' and select 'Create and Package New IP'.
9) Assign a name and version number to the peripheral. Select the interface and confirm.
10) Select 'Edit IP' and select 'Finish'. The peripheral has been generated by Vivado.
11) Open a new Vivado window that contains the peripheral created and open the 'multiply_v1_0_S00_AXI.v' file.
12) Comment out any code that writes to 'slv_reg2' to deactivate the write capabilities to the third software register.
13) Insert the lines of code given in the manual in the appropriate location in the source code.
14) In Package IP tab, select 'Review and Package' and select 'Re-package'.
15) Select the Vivado window which has the PS system and add the 'multiply' IP to the PS system.
16) Run connection automation and select the 'Regenerate Layout' option to clean up the circuit.
17) Select 'Validate Design' to check for errors in the design. Create HDL wrapper and generate bitstream.
18) Once the bitstream has been generated, export the hardware design (.xsa file).
19) Launch Vitis IDE and create a new 'Application Project'.
20) In the Platform page, select the 'Create a new hardware platform' option and select the exported hardware design file.
21) Give an application project name and select the 'Hello World' template.
22) Edit the helloworld.c file and write the code to write values to registers 'slv_reg0' and 'slv_reg1' and read the multiplication result from 'slv_reg2'.
23) In Vitis IDE, build the hardware platform project and software system project. The application binary file will be created.
24) Finally, Program the FPGA board by selecting the 'Program Device' option.
25) Use picocom serial console application to view the output of printf statements on the machine using the commands given in the manual.

(Source Code and Output Screenshots given in Appendix Section)

Results:

A custom IP was built for integer multiplication using AXI4 peripheral and used ARM processing system present in the Zybo board. Code was written in C to test the functionality of the designed IP. We make use of the 'MULTIPLY_mWriteReg' function and appropriate register offset values to write input numbers into 'slv_reg0' and 'slv_reg1' registers. The multiplier IP we constructed computes the multiplication result and stores it in register 'slv_reg2'. We use 'MULTIPLY_mReadReg' function to read the multiplication result from the 'slv_reg2' register. In the C code, we calculate the product of all combinations of numbers input ranging from 0 to 16.

Outputs were obtained and displayed on a serial console using picocom application. The outputs were demonstrated to the TA as well.

(Source Code and Output Screenshots given in Appendix Section)

Conclusion:

In conclusion, the third laboratory exercise provided us with hands-on experience in creating and integrating custom IP modules, essential for Zynq Processing System based systems. By developing a custom peripheral for integer multiplication and integrating it into a microprocessor system, we gained practical insight into hardware/software co-design. This exercise not only equipped us with valuable skills for building specialized embedded systems but also emphasized the importance of tailored hardware solutions in various industries. Moving forward, this knowledge will serve us well in tackling real-world engineering challenges with confidence and expertise.

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:

When we give inputs greater than 16 bits for both the input variables, the output variable will have a value greater than 32 bits, which would result in **'overflow'** of the output variable since it is also defined as a 32-bit integer. We can correct this by increasing the 'slv_reg2' register size to 64 bits. However, we will still need to ensure that the input bits for both inputs do not exceed 32 bits in length in that case.

To correct the peripheral, the following changes can be made in the 'multiply_v1_0_S00_AXI.v' Verilog code:

```
//------------------------------------------------
//-- 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;
```

Here, when defining parameters for an AXI4 peripheral, we specify the width to be 32 bits. These registers are 31 bits long, indicated by [31:0]. If we define it as 64 bits, the register size will become [63:0]. Thus, by increasing the size of the register, we can correct the output of the peripheral.

(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:

It is possible that we end up reading the incorrect output from the output register in the case where the inputs are lengthy, resulting in a computational delay of the multiplier circuit. If the read command executes before the final output is stored in the output register, we end up reading a garbage value instead of the expected output. This case can be avoided by introducing a delay before the read command, ensuring that the output is ready before the read command executes.

(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:

The main difference between the "Master" and "Slave" modes in an AXI4 peripheral is that the one assigned with the "Master" interface mode initiates the read or write commands, while the one with the "Slave" interface mode responds to these commands. So, in our module, the multiplier IP is the custom peripheral that we have developed, and to test its functionality, we need to send commands to it, to which it responds, such as read and write calls. Making it a "Master" would affect the functionality, as we wouldn't be able to test the peripheral itself.

Appendix:

(Screenshots of Source Code and Output Screenshots)

Picocom Serial Console Output Screenshots:

```
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
First number = 0
Second number = 0
Result = 0

First number = 0
Second number = 1
Result = 0
```

```
First number = 1
Second number = 0
Result = 0

First number = 1
Second number = 1
Result = 1

First number = 1
Second number = 2
Result = 2

First number = 1
Second number = 3
Result = 3

First number = 1
Second number = 4
Result = 4

First number = 1
Second number = 5
Result = 5

First number = 1
Second number = 6
Result = 6
```

```
First number = 16
Second number = 9
Result = 144

First number = 16
Second number = 10
Result = 160

First number = 16
Second number = 11
Result = 176

First number = 16
Second number = 12
Result = 192

First number = 16
Second number = 13
Result = 208

First number = 16
Second number = 14
Result = 224

First number = 16
Second number = 15
Result = 240

First number = 16
Second number = 16
Result = 256


Thanks for using picocom
bash-4.2$
```

Source Code:

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


int main()
{
    init_platform();

    int i,j;
    // Multiplication result
    int result = 0;

    // Outer loop for first number
    for( i = 0; i <= 16; i++ )
    {
        // Write first number into slv_reg0
        MULTIPLY_mWriteReg( XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 0, i );

        // Inner loop for second number
        for( j = 0; j <= 16; j++)
        {
            printf("First number = %d \n", i);
```

```c
    // Outer loop for first number
    for( i = 0; i <= 16; i++ )
    {
        // Write first number into slv_reg0
        MULTIPLY_mWriteReg( XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 0, i );

        // Inner loop for second number
        for( j = 0; j <= 16; j++)
        {
            printf("First number = %d \n", i);

            // Write second number into slv_reg1
            MULTIPLY_mWriteReg( XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 4, j );
            printf("Second number = %d \n", j);

            // Read multiplication result from slv_reg2
            result = MULTIPLY_mReadReg( XPAR_MULTIPLY_0_S00_AXI_BASEADDR, 8 );

            // Print output on console
            printf("Result = %d \n\n", result);
        }
    }

    cleanup_platform();
    return 0;
}
```