ES3F1 High Performance Embedded Systems Design

Lab 2

1 Introduction

Lab 2 is an expansion of the first lab, using the Digilent Zybo Z7-20 FPGA board. In this Lab we'll use Vivado HLS (High Level Synthesis) to generate HW components that calculate the matrix multiplication. Moreover, we'll explore how to exploit the parallelism in the calculations using HLS pragmas and how different directive factors affect the FPGA resources occupied by the component and its performance. Next, we'll use Vivado to use the generated HW blocks with the ZYNQ PS. Finally, we'll use Xilinx SDK to run the calculate the matrix multiplication and measure the execution time of each method.

2 Vivado HLS

In this section we will create the HW components that calculate the matrix multiplication. The matrices dimensions for the rest of this lab will be set to 5×5.

2.1 Create an HLS project

1. Start Vivado HLS 2017.2 and create a new project.
2. Set the Location and the name of the project. Click next.
3. In the **Add/remove C-based source files** window, click on the **NEW FILE** and create 2 source files, a C language source file (.c) and its header file(.h).
4. In the next window, we need to create a another C language file (.c) that will act as the testbench of the previously created source file. Once done, click next.
5. Set the Clock Period to 10, by default this number is in ns. From the **Part Selection** section find and select the **Zybo Z7-20** board. Click Finish to create your project.

On the left hand side, you should be able to find the files we have created under the **Source** and **Test Bench** tabs.

2.2 Unoptimized implementation

Source file

1. Open the .c source file and at first include its header file.

2. Create a void function with 3 parameters, each parameter must be a 2D array of 5x5. Two of those parameters are the two input arrays and the other is the result of the matrix multiplication.

   ```
   i.e. void array_mult(int in_a[ROWS][COLS],int in_b[ROWS][COLS],int res[ROWS][COLS]){
   ```

3. Define the interfaces of the inputs and outputs using HLS pragmas. In this case we'll use the AXI-Lite interface for all our ports. The example below sets the return port of the function as axilite, in this case the return port includes the inputs and outputs that are used to control the accelerator or to provide a status of its operation. The bundle option lets us create and name a group for this port or to even add it to existing bundles.

   ```
   i.e. #pragma HLS INTERFACE s_axilite port=return bundle=CRTL
   ```

   Using the example provided, apply the HLS INTERFACE pragma to the rest of the input and output ports of the function. Group the two input arrays to the same bundle i.e. DATA_IN, while the output array has it's own separate bundle i.e. DATA_OUT.

4. For the rest of the function, write the code (for-loops) that implements the array multiplication.You may reuse and adjust your code from Lab 1.

Header file

1. Open the header file under the sources section.
2. Add any libraries that you may be using in your .c source file.
3. Define any constants that you may be using.
4. Add your function declaration.

Test Bench

1. Open your testbench file and create a **main** function that returns an **int** and has no parameters.
2. Declare four 2D arrays, two of them will be used as inputs, one of them will be used to calculate in software the expected result and final one will be used to store the result that we get from the function in the source file.
3. Populate the input data for the two input arrays. You may reuse your code from Lab 1.
   NOTE: When using the rand() function to populate the data for the two input arrays DO NOT sample the timer to do so as this will result in an error in one of the design flow steps (in co-simulation). Instead use a fixed number as your seed to the rand() function.
4. Call the function that you created in the source file with arguments the two input arrays and an empty array in which the result of the HW design will be stored.

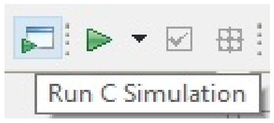
```
i.e. array_mult(in_a_tb,in_b_tb,mult_acc_tb);
```

5. Write the code the that calculates the array multiplication and store the result in an array.
6. Check that the two results, the array that was generated in SW, step 5, and the result from HW design, are the same. If they are, return 0, otherwise return 1.

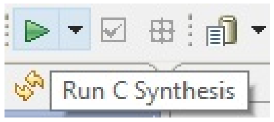
HLS implementation flow

Now we will go through the Vivado HLS implementation flow to make the design's functional verification, synthesis and implementation.

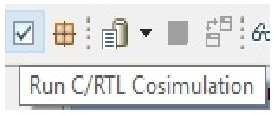
1. Define the top function of the design by opening the **Project** menu and then click on the **Project Settings**. In the window that opens up, click on the **Synthesis** tab on the left hand side and then click on **Browse**. This will open up another window, that lists the available functions of the project. Select your top function (In this case it should be only one).
2. Run the C simulation, this can be done by clicking on the **Project** menu and then **Run C simulation** or by clicking on the icon shown below.

   Doing so, the tool runs the testbench in SW and lets us find out any issues in the code before advancing to the following stages.
3. Next, proceed to the **Synthesis** of our design code by either clicking on the icon shown below or by clicking on the **Solution** menu and then **Run C Synthesis –> Active Solution**

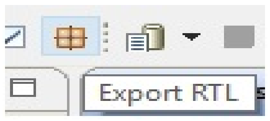
   

4. Once the Synthesis has finished, run the **C/RTL Cosimulation** by clicking on the icon shown below or by clicking on the **Solution** menu and then **Run C/RTL Cosimulation**. In the window that opens up, click OK to the default options.

The C/RTL Cosimulation generates a testbench in HDL which uses the generated RTL code of our design. The results of the RTL simulation are then compared with the expected results to verify the functionality of the generated RTL code.

5. The last step is to **Export the RTL** by clicking on the icon shown below or by clicking on the **Solution** menu and then **Export RTL.**



In the window that opens up make sure the **Vivado synthesis,place and route** are checked before clicking OK.

2.3 Optimized implementation

To exploit the parallelism in matrix multiplication and effectively reduce the latency of our design, we'll a Vivado HLS pragmas to unroll the loop computations. The syntax of the unroll pragma is defined as follows:

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

The keywords **factor**, **region** and **skip_exit_ check** are optional and the last two are out of the scope of this lab. This enables us to partially unroll, by defining a factor, or to fully unroll the loop that the pragma is placed in, by omitting any factor definition.

1. Create an new HLS project with new source files and test bench. (Give them different names from what you named your files before, to differentiate.)

2. Copy and paste the code from the unoptimized version to your new files.

3. Name your function differently and change your code accordingly.

4. Try different placements of the unroll pragma (fully or partial) and synthesize your design. Try using the pragma in more than one loop.

5. What happens to the synthesis results? (Utilization estimates and latency)

6. Unroll fully the two inner loops of your source file and go through the HLS implementation flow as described previously.

3 Vivado

In this section we will use Vivado to create a new block design that implements the system (PS-PL).

1. Start Vivado and by following the same process as in Lab 1 create a new project.
2. Add the IP repository of the unoptimized design by clicking on the **Window** menu and selecting the **IP Catalog**. Right click on the new window that opens up and click on the **Add Repository**. Browse to the directory where your HLS unoptimized project is saved and the select the **/solution1/impl/ip** directory.
3. In the same way, add the IP of the optimized design and then you can close the window.
4. Create and open a new block design.
5. By right clicking in the Diagram's area add the following IPs:
   (a) ZYNQ7 Processing System
   (b) Processor System Reset
   (c) Concat
   (d) The unoptimized IP core for matrix multiplication
   (e) The optimized IP core for matrix multiplication
6. Run Block Automation
7. Double click on the **ZYNQ7 Processing System**. On the window that opens up, select the **Interupts** tab from the left hand side. Enable and then expand the **Fabric Interrupts** and then, from the **PL-PS Interrupt Ports**, enable the **IRQ_ F2P**[**15:0**] and then click OK.
8. Make the following connections :
   (a) The **FCLK_CLK0** of the **ZYNQ7 Processing System** to the **M_AXI_GP0_ACLK** of the same IP.
   (b) The **FCLK_RESET0_N** of the **ZYNQ7 Processing System** to the **ext_reset_in** of the **Processor System Reset**.
   (c) The **dout** of the **Concat** to the **IRQ_F2P** of the **ZYNQ7 Processing System**.
   (d) The **interrupt** outputs of BOTH HLS IPs to the **In0** and **In1** inputs of the **Concat**
9. **Run Connection Automation**, make sure to select all ports.
10. Validate your block design
11. Create the HDL wrapper

12. Generate the bitsream

13. After the bitsream has been generated, export the hardware (make sure that you included the bistream) and launch SDK.

4 Xilinx SDK

In this section we will use Xilinx SDK to calculate the matrix multiplication in software and on the two HLS designs. We will also measure the execution time of each method and print their runtime. You can reuse your code from Lab 1 and make the appropriate adjustments to it.

1. Make sure that the drivers generated by HLS for the two accelerators can be found under the **design_1_wrapper_hw_platform_0** tab.

2. Open one of the header files and have a look at the available functions. Their name should be in the form of **x<name of the HLS function>.h**.

3. Create a new application project as described in Lab 1.

4. Open the .c file of the application project and paste your code from Lab 1.

5. Modify your code so that the pointers to the arrays used are in the form of a single pointer and not double.

6. Include the two header files of each HLS implementation. Their names can be found from Step 1 and they should be in the form of **x<name of the HLS function>.h**

7. Add 4 variables to measure the start and time of the HLS implementations.

8. Declare a definition struct for each of the accelerators. The struct name can be found in the header files of each HLS implementation and should be in the following form: **X<capital(N)ame of the hls function>**

```
i.e. XArray_mult array_mult_hw;
```

9. Declare a configuration struct pointer for each accelerator. The struct names can be found in the header files of each HLS implementation and should be in the following form: **X<capital(N)ame of the hls function>_Config**

```
i.e. XArray_mult_Config *array_mult_hw_cfg;
```

10. Load the configuration of each accelerator using the LookupConfig function to each pointer struct. The function can be found in the header file of each HLS implementation and should be in the following form: **X<<capital(N)>ame of the hls function>_ LookupConfig**. The function takes as an input the HLS implementation's device id which can be found in the **xparameters.h** file and is in the following form **XPAR_<NAME OF THE HLS FUNCTION>_0_DEVICE_ID**

```
i.e. array_mult_hw_cfg = XArray_mult_LookupConfig(XPAR_ARRAY_MULT_0_DEVICE_ID);
```

11. Initialize each accelerator instance struct with its configurations. This can be done using the **CfgInitialize** function. The function can be found in the header file of each HLS implementation and is usually in the following form: **X<capital(N)ame of the hls function>_CfgInitialize**. The function takes as an input the address of the accelerator's definition struct and its configuration.

    ```
    i.e. XArray_mult_CfgInitialize(&array_mult_hw, array_mult_hw_cfg);
    ```

12. Send the input data to each of the accelerator's input using the **X<Name of the HLS function> _Write_<port name>_Bytes** function which can be found in the header files of the HLS implementations.

    ```
    i.e. XArray_mult_Write_in_a_Bytes(&array_mult_hw,0,x,sizeof(x)*(rows*cols));
    ```

13. Start the computation of the unoptimized accelerator and wait until the computation ends. This is the part of the code that you need to measure its execution time. '

    ```
    XTime_GetTime(&tStart_hw);//start time
    XArray_mult_Start(&array_mult_hw);//starts the accelerator
    while(!XArray_mult_IsDone(&array_mult_hw));//computation ends check
    XTime_GetTime(&tEnd_hw);//end time
    ```

14. Transfer the result of the unoptimized accelerator to an array.

    ```
    i.e. XArray_mult_Read_res_Bytes (&array_mult_hw,0,hw_res,sizeof(int)*(rows*cols));
    ```

15. Do the same for the optimized accelerator.

16. Check that the three arrays are the same.

17. Print the execution time of the three computation methods. Compare the execution time of each HLS implementation with its theoretical one.