Lab Exercise 4
Pipelining with VHDL and implementation of a SoC on the Xilinx Zynq-7020

INF3430/INF4431 Autumn 2018
Version 2.3/3.10.2018

Task 1

Adding and multiplying numbers involves extending the number of bits in the sum and in the product. This is the case when we assume that the magnitude of the numbers are maximum.

Exercise 1:

We have two numbers a and b which is 16 bits each.

How many bits is the result of the sum of these two numbers (i.e. a + b):

a) 16 bits b) 17 bits or c) 18 bits

Exercise 2:

How many bits is the result of mulitiplying these two 16 bits numbers (i.e. a * b):

a) 16 bits b) 32 bits or c) 33 bits

Exercise 3:

We now have four numbers: a, b, c,and d who are 16 bits each.

How many bits is the result of adding all these numbers a + b + c + d,

a) 16 bits b) 17 bits or c) 18 bits

Exercise 4:

We now have a additional number e which is 16 bits.

How many bits is the result of (a + b + c + d) * e

a) 16 bits b) 32 bits c) 33 bits or d) 34 bits

In the module compute shown below, the sum of the numbers a, b, c and d is calculated as 18 bits, and then multiplied with the 16-bit input e when dvalid is equal ‘1’. The output result with 32 bits is set to max value equal to x"FFFFFFFF" (i.e. all bits set to '1') when the result is greater than x"FFFFFFFF" and the signal max is then set to '1' at the same time. The rvalid signal is active high when a valid data is output. The given compute module is simulated with the given testbench tb_compute.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compute is
  port
    (rst    : in  std_logic;
     clk    : in  std_logic;
     a      : in  std_logic_vector(15 downto 0);
     b      : in  std_logic_vector(15 downto 0);
     c      : in  std_logic_vector(15 downto 0);
     d      : in  std_logic_vector(15 downto 0);
     e      : in  std_logic_vector(15 downto 0);
     dvalid : in  std_logic;
     result : out std_logic_vector(31 downto 0);
     max    : out std_logic;
     rvalid : out std_logic);
end entity compute;

architecture rtl of compute is
begin
  process (rst, clk) is
    variable addresult_i   : unsigned(17 downto 0);
    variable multresult_i : unsigned(33 downto 0);
  begin
```

```
    if rst = '1' then
      result <= (others => '0');
      max    <= '0';
      rvalid <= '0';

    elsif rising_edge(clk) then
      if (dvalid = '1') then
        addresult_i := (unsigned("00" & a) + unsigned("00" & b)) +
                       (unsigned("00" & c) + unsigned("00" & d));
        multresult_i := addresult_i * unsigned(e);
        if (multresult_i(33 downto 32) = "00") then
          result <= std_logic_vector(multresult_i(31 downto 0));
          max    <= '0';
        else
          result <= (others => '1');
          max    <= '1';
        end if;
      else
        result <= (others => '0');
        max    <= '0';
      end if;
      rvalid <= dvalid;
    end if;
  end process;

end architecture rtl;
```

It turns out that there are timing errors during implementation in the selected technology and clock frequency. The compute module has to be changed to a new module compute_pipeline with architecture rtl that only have the add operations in one clock period, and the multiply and the compare operation (i.e. multresult_i(33 downto 32) = "00") in the following clock period to achieve the timing requirement (i.e. pipelined operations). Multiple add operations can be performed in one clock period.

Exercise 5:

Implement the new module compute_pipeline in synthesizable VHDL based on the given compute rtl architecture.

The given testbench tb_compute_pipelined shall be used to ensure that the compute_pipeline architecture works as required.

Exercise 6:

How many registers/flip-flops are used in the module compute?

a) 32 b) 33 c) 34 or d) 35

Exercise 7:

How many registers/flip-flops are used in the module compute_pipelined

a) 32 b) 68 c) 69 d) 70 or e) 103

Hint: You can use Vivado synthesis tool to check the answers to exercise 6 and 7.

Task 2

In this lab you will implement a simple application in the processing system (PS) and connect it to FPGA hardware modules that includes the motor controller (MC) module designed in lab 3, so that you can change the set-point dynamically from the software in the PS.

IMPORTANT: This lab requires that all VHDL source files to be compiled with VHDL-2008 for simulation.

Set Motor Controller Set-Point using Zynq Processing System

<u>Goal:</u> The goal of this exercise is to successfully implement a bare metal application running on the processing system on the Zynq, which interfaces to the switches and Motor Controller (MC) implemented in lab 3. The motor set-point can be set either from the switches (SW 0-6) or it can be automatically updated with set-points from the Processing System (PS).

<u>Required for approved delivery:</u> The final Vivado project with the implemented system, successful presentation of the working hardware to a lab supervisor and answers to the questions in the lab.

System Architecture

In this lab you will learn how to use the Zynq Processing System (PS) and how to interface it with the Programmable Logic (PL). This hybrid solution allows implementing powerful system-on-chips and integrating the benefits of microprocessors and FPGA.

The system that you will assemble in this lab is made of different blocks:

- Zynq Processing System (PS)
- AXI interconnect
- AXI2PIF Bridge module
- Register Module (lab4reg)
- 7-bit MUX
- Motor Controller module (MC, from lab 3)

The **Zynq PS** is interfaced to the rest of the modules via an AXI interconnect. You have to create a block design containing these 2 IPs (and a Reset module) as explained in section 5 of the Cookbook.

The **AXI2PIF** Bridge allows to write and read data to and from specific addresses on a shared bus.

The **Register Module** is connected to the bus at a specific address, so that the PS can read and write data to the register using AXI communication. Although in this case only a single register module is connected to the bus, the same architecture can be extended to several modules and it is commonly used in industrial applications.

The **7-bit MUX** is used to choose between two possible ways of setting controlling the motor. The SEL of the MUX comes from SW 7 of the Zedboard. When SW7 is "0", then the set-point corresponds to the 7 bits on the remaining switches of the board – SW 6 to 0, when it is set to 1 the set-point must be mapped to the Register Module value.

The **Motor Controller (MC)** is the same module you implemented in lab 3.

The structure of the system is shown in the figure below.

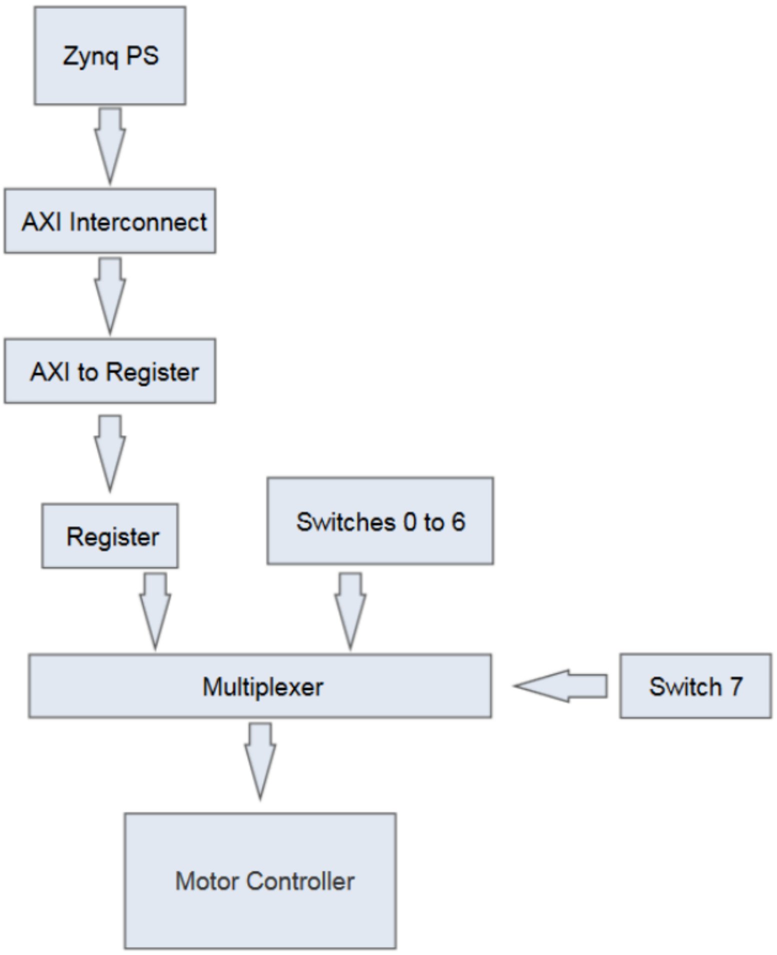

Figure
Modules and connections of the system

Tasks

1) Perform top level simulation with the given AXI4Lite Bus Functional Model (BFM) to simulate AXI4LITE read and write accesses.

   Create a Modelsim project in the “sim” folder. Use the testbench found in “tb” folder to simulate transactions on the AXI bus using functions included as a part of the BFM. The required source files are given in the “src” folder. A dummy model is used in the testbench instead of the processor system; which will be added in task 3. Write different values to the LAB4REG_SETPOINT register and observe how the motor responds (**motor_beh.vhd** is used for simulation of the motor/position sensor).

   Include the pos_seg7_ctrl module developed in lab3 in lab4 (i.e. compile all VHDL files for the pos_seg7_ctrl module)!

   Note: select all files in your Questa project and set them to **VHDL-2008**.

2) Add a 32-bit test register and a 16-bit test register in the register module (lab4reg_rtl.vhd) and lab4 package (lab4_pck.vhd), and add the commented RAM module to the PIF bus. Simulate write and read register accesses to the register module and RAM module. Use the included RAM register based model (**ram_lab4_rtl.vhd**) for RAM simulation and synthesis in this task.

3) Create a block design named **lab4processor** containing the Zynq Processing System, the AXI Interconnect, and the Processor Reset Module. Follow the instructions of the cookbook (Section 5). Remove the dummy model used in task 1 (i.e. **lab4processor_dmy.vhd**) in the top level module (**lab4_top_str.vhd**) and add the instantiation of the processor module (i.e. **lab4processor.bd**). Use the given constraints file (.xdc), and add the generated clock and clock group from lab3. You can find the clock using Synthesis -> report Clock Network.

4) You shall in this task change the RAM to a Xilinx IP block RAM with the same name as the RAM RTL model (i.e. **ram_lab4**) as explained in the Cookbook chapter 5.3. The RAM has to be a single port BRAM with 32-bit width and a depth of 1024 words. **It must handle byte access with 8-bit byte size (i.e. tick Byte Write Enable and select Byte Size 8-bits). Remove the tick for Primitives Output Register and add a tick for RSTA pin to make the generated RAM equal to the ram_lab4_rtl VHDL model**. Simulate write and read register accesses to the block RAM module with the generated Xilinx block RAM netlist VHDL simulation model (i.e. compile the VHDL file **ram_lab4_sim_netlist.vhdl (**in Vivado older than version 2015.3 **ram_lab4_funcsim.vhdl) instead of ram_lab4_rtl.vhd**).

5) Use the RAM VHDL RTL model (**ram_lab4_rtl.vhd**) for synthesis and implementation and then change the RAM to the Xilinx IP RAM (i.e. remove ram_lab4_rtl.vhd from Vivado and add the generated file **ram_lab4.xci**) and rerun synthesis and implementation. Open the technology viewer in Vivado after implementation and compare used resources. What is the difference in resource usage between the register model and the Xilinx IP block RAM model?

6) Write a C program with the Xilinx SDK (follow the steps in section 5.1 to 5.3 of the cookbook) in order to dynamically set a new set-point for the motor between 0X00 and 0X7f (i.e. 127 with 7 bits available). When writing to the register, check with a read operation that the written value is correct.
Remember to add a delay with the sleep function between write operations to let the motor move to the previous position before a new value is written. Sleep(1) waits for 1 second and sleep(2) waits for 2 seconds etc. Calibrate the delay so that the motor is able to reach the set position.

7) (**Mandatory for INF4431 and optional for INF3430**)

Add a System_ILA logic debug module in the block design to observe read/write accesses on the AXI4Lite interconnect to the AXI4PIFB module. Add an ILA logic debug module to the top level VHDL code to observe read/write accesses on the AXI4LITE interconnect, the PIF bus and observe the setpoint value to the motor controller module. Use the same managed IP project as you used for the RAM to create the ILA IP. Usage of the ILA and the logic analyzer in Vivado is shown in a lecture (i.e. the Xilinx Video).

8) (**Mandatory for INF4431 and optional for INF3430**)

Write a small C program that first writes 20 motor positions to the RAM module and then use the values in the RAM to move the motor.

9) (**Optional**)

Add a new register module with a 32-bit test register and a 16-bit test register. This will require some modification to the bridge module between AXI and PIF (axi4pifb_rtl.vhd). You must add a new module base address in the address map (lab4_pck.vhd), and then update the decoding of module addresses in the bridge.

Hint: Use “LAB4_REG” as a template.