EDA322 Digital Design Lab

LAB 4

Designed by: Angelos Arelakis; Anurag Negi, Ioannis Sourdis

The goal of this lab is to implement the controller for the ChAcc processor. In the previous labs, you implemented the datapath of ChAcc. The controller is the last missing part before having a fully functional simple processing unit. The controller is the "brain" of the processor since it orchestrates all the different processor modules in a way that useful operations can be executed on the processor. Before starting the lab, please do the preparation as described below.

Preparation

Preparing for the fourth lab requires to:

1. Completed lab3.
2. Study Sections 2 and 3 in the processor's specification document (processor.pdf).
3. Study the lecture material of up to the previous study week.
4. Prepare in a document the following:
    a. Draw the state transitions for a Mealy type finite-state machine (FSM) for the processor's controller. Your FSM should cover all the instructions of the processor's ISA.
    b. Answer to the following question: How do a Moore and a Mealy type FSMs differ regarding the output timing? Use the output "pcLd" to explain your answer (see Table 3, processor.pdf).
    c. Answer to the following questions: What is the difference between the following VHDL processes? What is role of the sensitivity list of a process?
        i. Tempo1: process (enable, reset)
        ii. Tempo2: process (clock, enable, reset)
    d. In the VHDL code below, what will be the values of out_1, out_2 and out_3 when a=1, b=0 and c=1? If in the next set of inputs only b changes to 1 what will be the output values?

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

entity prep is
port(   a, b, c:        in std_logic;
        out_1, out_2, out_3:   out std_logic);
```

```
end prep;

architecture behave of prep is

  signal temp_s: std_logic;

begin

  proccess: process(a,b,c)
    variable temp_v: std_logic;
  begin

            temp_v := a and b;
            out_1  <= temp_v  xor c;

            temp_s <= a and b;
            out_2  <= temp_s  xor c;
  end process;

            out_3 <= temp_s xor c;

end behave;
```

Introduction

The controller is implemented as a synchronous sequential circuit making use of a Finite State Machine (FSM). The processor's specification document (processor.pdf) describes thoroughly the working of the controller as well as which signals and when must be set or reset. Therefore a **deep knowledge** of the **controller's specifications (described in processor.pdf)** is required to make the implementation in a reasonable amount of time. After implementing the controller, add it to the project of lab3 and replace the previously used mock controller. The correctness of the controller will be tested by simulating the whole implemented processor in ModelSim running the provided testbench "TestbenchLab4.vhd". In the simulation, you will need to initialize the memories with the new mif files.

This lab assignment requires you to do the following **three** tasks:

1. Implement the ChAcc processor controller using a Finite State Machine (FSM).
2. Replace the mock controller with the new controller.
3. Simulate the whole processor by running the provided Testbench.

Implementing the controller – Task 1

The entity of the controller (is named as *procController*) is given below. The controller's interface (inputs/outputs) is detailed in the processor's specification document (processor.pdf).

```
entity procController is
    Port (    master_load_enable: in STD_LOGIC;
              opcode : in  STD_LOGIC_VECTOR (3 downto 0);
              neq : in STD_LOGIC;
              eq : in STD_LOGIC;
              CLK : in STD_LOGIC;
              ARESETN : in STD_LOGIC;
              pcSel : out  STD_LOGIC;
              pcLd : out  STD_LOGIC;
              instrLd : out  STD_LOGIC;
              addrMd : out  STD_LOGIC;
              dmWr : out  STD_LOGIC;
              dataLd : out  STD_LOGIC;
              flagLd : out  STD_LOGIC;
              accSel : out  STD_LOGIC;
              accLd : out  STD_LOGIC;
              im2bus : out  STD_LOGIC;
              dmRd : out  STD_LOGIC;
              acc2bus : out  STD_LOGIC;
              ext2bus : out  STD_LOGIC;
              dispLd: out STD_LOGIC;
              aluMd : out STD_LOGIC_VECTOR(1 downto 0));
end procController;
```

The rest of the controller is built using a finite state machine (FSM), where a state transition takes place when an executed instruction goes from one datapath stage into another. Recall that the datapath is divided into 5 stages: Fetch (FE), Decode (DE), Decode* (DE*), Execute (EX) and Memory (ME) but only a subset of them is eventually used by each executed instruction. For example, the Decode* stage is only used by the index instructions because they access the data memory twice. See processor.pdf for more details.

Therefore, when designing the FSM, it must be taken into account that different states are required based on each instruction, thus the state transition may vary. The used stages and the exact number of them (summarized in Table 2 at processor.pdf) are known in the Decode stage when the *opcode* is sent to the controller. In addition, particular control signals (Controller's outputs) must be set or reset in every state (stage). This is summarized in Table 3 in processor's specification document (processor.pdf).
There are two FSM design alternatives:

- <u>Moore-type</u>: There are few states (FE and DE) that are shared by all the instructions. For the rest, there is one state sub-diagram per instruction. The outputs are determined based on the current state and are independent of the inputs.

- Mealy-type: The same states are used by all instructions. However, in Mealy-type FSMs the output signals (e.g., *flagLd*) depend on both the input and the current state.

Figures 1 and 2 depict an example part of the designed controller using a Moore-type and a Mealy-type FSM respectively. The former is easier to design but will result in a large number of states complicating the implemented design. On the other hand, the latter is more elegant since it keeps the number of states low. You can use either of the design alternatives but it is strongly recommended to use the Mealy-type FSM.

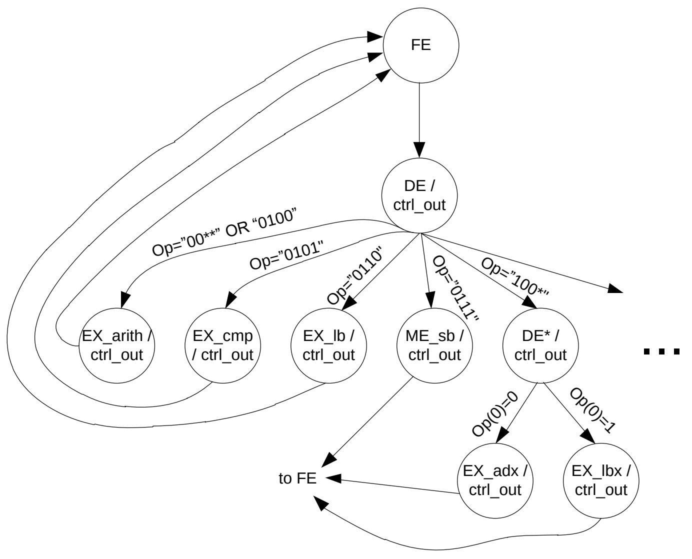

Figure 1: Part of a Moore-type FSM example of the controller

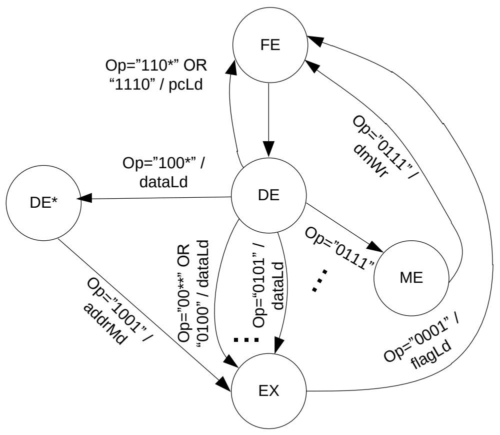

Figure 2: Part of a Mealy-type FSM example of the controller

When reset (ARESETN) is enabled ('0'), a state transition from any state to "FE" must take place and all the output control signals must be reset. When ARESETN is released, the program starts from the beginning as PC is set to 0.

You now have all the important information to design and implement the controller of the ChAcc processor. Use the project of Lab3 or create a new one to implement the controller. Create a new vhdl file and copy the Entity Declaration, exactly as it was given in the text box at the beginning of this lab assignment, into that file. Then, follow the steps:

1. Design the FSM on a paper including all the states and which outputs must be generated in each particular state depending on the FSM type.
2. Implement the FSM in VHDL using ModelSim. It is strongly recommended that you implement the FSM as a sequential circuit that consists of two parts: a) the combinational and b) the "memory" element that is actually implemented using registers. Because of this, the "memory" part of the FSM is controlled by *master_load_enable* like the registers (see the processor's specification document). Use behavioral or dataflow design style, but note that state assignment and minimization are required, if you select the dataflow design style.

Hints:

For the state of the FSM you have to implement a register, for this reason you are going to need two signals and a process. The state signals will have to be of type State_type, declare a type as follows:

type State_type is (FE, DE1, DE2, EX, ME);

signal curr_state , next_state : State_type;

The process will be similar to the one you used for the register in Lab3, but on reset the ***curr_state*** must be set to the ***FE*** state.

```
fsm : process(CLK, ARESETN)
if ARESETN = '0' then
...
Else
...
End process
```

After you have implemented the state register you need two more processes for the FSM to be complete. The first process will set the value of ***next_state*** based on the ***opcode*** and the ***curr_state***.

```
next_state_process : process( curr_state, opcode)
begin

        case( curr_state ) is
                when FE =>
                        ...
                when DE1 =>
end process ;
```

The final process you are going to need to implement the FSM will take care of the outputs of the controller based on the values of the ***curr_state***, and the inputs to the controller such as ***opcode, EQ, NEQ***

```
output_process : process( curr_state, opcode, EQ, NEQ )
begin
...
end process;
```

Testing the controller – Tasks 2 and 3

Connect the new controller to the datapath that you implemented in Lab3, by replacing the mock controller with the new one. Compile with ModelSim and correct any syntax errors. Then verify the correctness of the controller by simulating the whole implemented processor. Run the provided testbench (TestbenchLab4.vhd) in ModelSim. Please don't make any changes to the testbench. You need to download the new *.mif* files to your project directory and initialize the instruction and data memories. The "inst_mem.mif" contains the lab4test code (lab4test.pdf contains the assembly code) in machine code, while the "data_mem.mif" contains the initial state of the data memory. The testbench must run for approximately 700ns. The test will be successful if your data memory content matches the one in "DM_after_exec.pdf" after completing the test run.

If the test fails, this can be due to a combination of errors. Debug your design by first checking the controller. Load its inputs, outputs and the state signals to the waveform and re-run the testbench. Check that the state transitions happen correctly for each simulated instruction and the correct output signals are generated based on the provided specifications (see Tables 2 and 3 in processor.pdf). If this is correct, then the problem may be on the datapath. Recall that you could not verify its correct operation due to the lack of a realistic controller. Check the value of processor's main output signals or the rest of internal signals by adding them in the waveform.

Demonstration

Tasks to be done for successfully completing this lab:

1. Write VHDL to implement the controller module of the ChAcc processor. Show the code to the instructor.
2. Connect the controller to the datapath of lab3.
3. Simulate the processor using the provided testbench (run for at least 700ns) and the provided .mif files. The data memory content (after the simulation) must match the one of the provided file "DM_after_exec". Demonstrate a working simulation to the instructor.

Evaluation:

The instructor will check for the following:

Task#	Coding style	Simulation
1	X	
2	X	
3		X

Make sure that when you are done with the lab, you have demonstrated all checked aspects of each task. This is necessary for successful completion of the lab.

Lab report:

In the final lab report, write one section describing what you did in this lab. More specifically, show the FSM of the controller by presenting the diagram you drew. Which design decisions did you make and why? Also include few waveforms, where you show that the controller runs correctly for some particular instructions using the provided testbench.

Learning outcome:

After completing this lab, you should be able to:

- Draw FSMs for non-trivial problems like the controller of a simple processor.
- Implement an FSM in VHDL using dataflow or behavioral design style.
- Debug a full design.

Hints and Tips

Problem with getting correct values in the Data memory

There are several issues that might cause your design to not perform as expected.
Here is a list of things which might help you with debugging your controller:

1- Check if your controller goes through all states as expected. Upon arrival of a new opcode, in which state should your controller be?

2- Try to understand the input instructions and the sequence of outputs you expect. Spot the element of the data memory that is wrong at the end of the simulation. Examine the test program and understand what it does in every step. Pinpoint the instruction that was responsible for writing the particular value in the data memory. Check that the particular instruction was ran correctly (the controller went through the appropriate states and activated the appropriate outputs at the correct times). If all of the above is correct, it is possible that either (a) some previous instruction produced a wrong result which is currently written in the memory instead of the correct one or (b) that some datapath component that takes part in the execution of the instruction works wrong. Keep tracing back the error to the actual cause.

3- Remember the hint from lab 2 PM, about red lines in your waveforms. If there are such red lines, focus on them before going into detailed examination of other signals' values.

4- Check the sequence of the Flag register (FReg). (MSB) to least significant bit (LSB) order, since it is common to scramble these signals at some connection:

    a. Ovf: It indicates overflow in the ALU operation.
    b. NEQ: Indicates that the two ALU input operands are not equal.
    c. EQ: Indicates that the two ALU input operands are equal.
    d. Zero: Indicates that the ALU data output is zero.

5- In the controller, state transition should not start before reset signal is released.

6- EQ and NEQ are two of the inputs that should be connected to the controller from the output of FReg (be careful not to connect them from the output of ALU).