EDA322Digital Design Lab

LAB2

Designed by: Angelos Arelakis; Anurag Negi, Ioannis Sourdis

The goal of this lab is to implement the top level design of the ChAcc processor. In the previous lab, you implemented the ALU module with a ripple-carry adder (RCA) or a carry look-ahead adder (CLA). Other modules that may be useful from previous labs are the muxes. Before starting the lab, please do the preparation as described below.

Preparation

Preparing for the third lab requires to:

1. Complete lab1.
2. Listen to the corresponding coding tutorials.
3. Study Sections 1 and 2 in the processor's specification document (processor.pdf).
4. Study the lecture material of up to the previous study week.
5. Download "Lab2 preparation – Questions.pdf" and answer the questions.

Introduction

The top-level design of ChAcc processor includes a) the implementation of datapath modules such as memory, registers and the bus, b) initialization of memory components and c) connecting all the datapath components together, assuming the controller as a black box. The datapath of ChAcc is depicted in Figure 1.

The lab requires you to do the following **three** tasks:

1. Implement the storage components (register and memory) and verify them with ModelSim.
2. Implement the bus using multiplexors and verify its correct operation.
3. Connect all the modules of the top-level design of the ChAcc processor. Assume the controller as a black-box. For this purpose, a mock controller is provided. Due to this assumption, you will not be able to verify the correct operation of the whole datapath.

Open ModelSim and create a new project.

The submitted code is checked for plagiarism

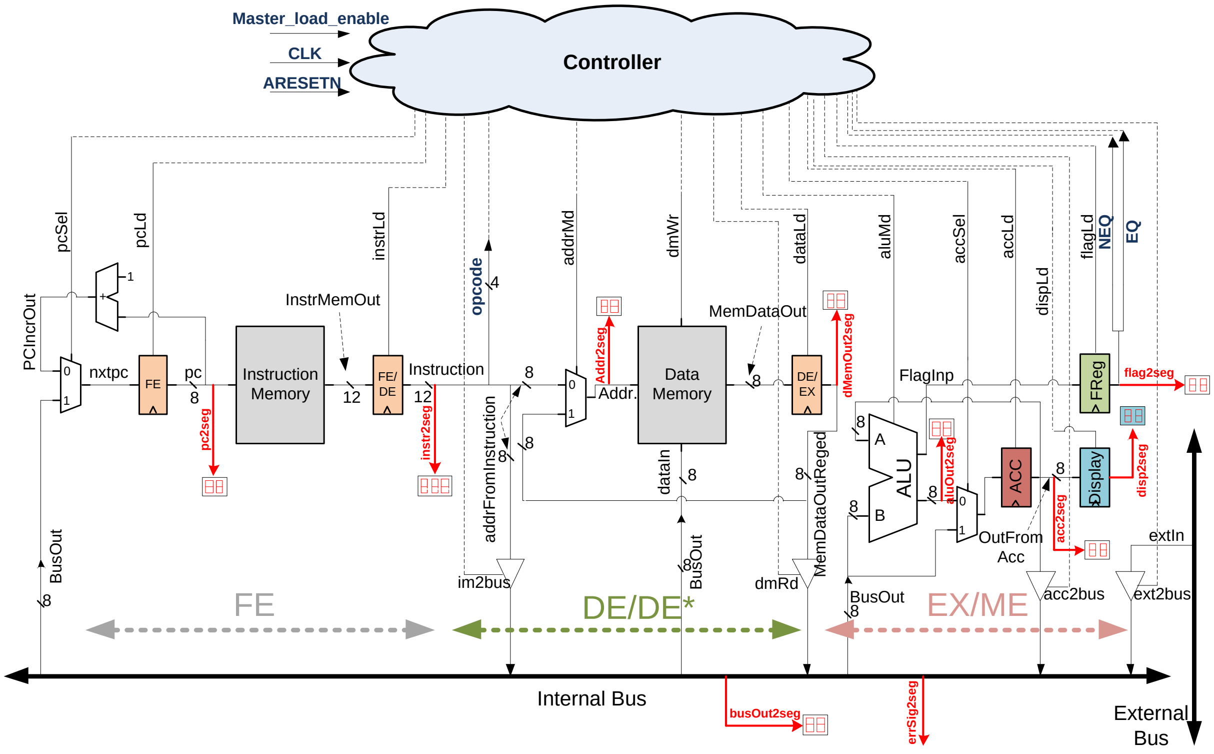

Figure 1: ChAcc processor datapath

Storage components – Task 1

Registers

The register is the simplest storage component that is used in the ChAcc processor. It contains a D flip-flop which stores the input in each positive clock edge. An extra enable signal (*loadEnable*) is used to control whether the register should save a new value or keep the current one. Therefore, a register is implemented like a mux (controlled by *loadEnable*) connected to a flip-flop, as it is also discussed in the processor's specifications document. A register is depicted in Figure 2.

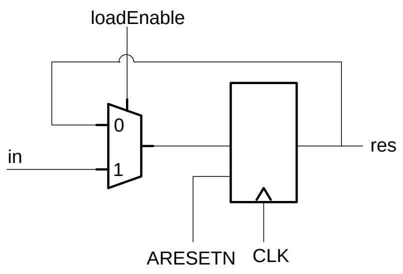

Figure 2: Register – D flip-flop with load enable

A register has **four** inputs and **one** output:

1. *in*: Data **input** of the register. May have multiple widths.
2. *CLK*(**input**): Connected to the design's clock signal.
3. *ARESETN*(**Input)**: Connected to the design's reset signal, i.e. *asynchronous* and active when it is*'0'*.
4. *loadEnable*: Control **input** signal. When '1', the register can load a new input.
5. *res*: The data **output**. It has the same width as the **datainput**.

Write the VHDL implementation of the register. You can make use of any design style: behavioral, dataflow, or structural. According to the processor's specification document, ChAcc contains registers of multiple bit-widths. Therefore, implement the register using *generic*. Note that 1-bit registers cannot be instantiated using the generic-based ones and must be additionally implemented, if needed. Verify their correct operation using ModelSim and a "do" file.
For the implementation of the register you do not need to explicitly design a Multiplexer component or include one it in the design. The Multiplexer can be inferred by your design by using a process with both the *clk* and *aresetn* in the sensitivity list (look at how flip flops are implemented in Lab Tutorial as an example).

Memory

The memory is the other storage component that is used by our processor. ChAcc has two memories, as it follows the Harvard architecture: one for the instructions and another for the data. Although there are two different memory instances, only one memory module is needed to be implemented again using *generics*. Then, during the implementation of the top-level design (Task 3), the memory module will be configured twice to implement the two different memory instances (instruction memory and data memory).

Create a vhdl file with the name "mem_array.vhdl" in ModelSim. The entity of the memory module is given below:

```
entity mem_array is
   generic (
       DATA_WIDTH: integer := 12;
       ADDR_WIDTH: integer := 8;
       INIT_FILE: string := "inst_mem.mif"
 );
 Port (ADDR : in  STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0);
DATAIN : in STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
CLK : in STD_LOGIC;
WE : in STD_LOGIC;
OUTPUT : out  STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0));
end mem_array;
```

There are **four** inputs and **one** output:

1. *ADDR:* The memory address is used as an **input** to access the memory in both read and write operations.
2. *DATAIN*: The **input** data that are saved in the memory in case of a write.
3. *WE*: The *Write Enable* signal is a control **input** and must be set when new data are written into the memory.
4. *CLK*(**input**): Connected to the design's clock signal. The CLK signal is needed because the write operation is synchronous, as is mentioned in the processor's specification document. On the other hand, the read operation is implemented in an asynchronous way but the output data are eventually saved in a register.
5. *OUTPUT*: The data that are **output** in case of a memory read.

The address and both data signals (input and output data) have particular widths (*ADDR_WIDTH* and *DATA_WIDTH* respectively) that are defined in a generic way to be parameterized later in the configuration of the memory module during instantiation (port map and generic map). Both width parameters are integers and can be initialized to any number (here they are initialized appropriately to the widths for the instruction memory) .*INIT_FILE* keeps the filename that is used for the initialization of the memory array. The memory can be initialized with zeros or a particular data set using an input file. This is explained later.

The memory is a data array and its implementation is conceptually the same to array implementations in higher level languages. The main difference is that in VHDL there is no particular memory array type but just an abstract array type. Therefore, the type must be first created. Then the memory is declared using a signal of this type. In the *declaration* region of the *Architecture Body* of the previously created file follow the next two steps:

1. Define the type, using the name *MEMORY_ARRAY*, as a two-dimensional array of data, as shown in the following text box.

```
Type MEMORY_ARRAY is ARRAY (<number of entries>) of <data type>(<data size>)
```

One dimension (rows) is the **number of memory entries** (declared as a range, e.g., "0 to N-1" for N entries). There are ways to relate the number of entries to the address width and thereby remove the need to explicitly give the number of entries. The other dimension (columns) is related to the **stored data**, thus the size and type (e.g., *std_logic_vector*) of the stored data must be explicitly determined. The type of the first dimension (rows) is integer, while the type of the stored data can be determined looking at the type of the output or input data. The fields inside the <> must be determined by you (omit the symbols '<' '>'). Remember that these fields must be *generic* so that the memory can be configured for any possible memory size (in both dimensions).

2. Create the actual memory array by declaring a signal of type *MEMORY_ARRAY*. Initialize the memory array by calling the function "init_memory_wfile" using *INIT_FILE* as argument. The function is written in the declaration region in the Architecture Body, before all signal declarations that make use of it but after the *MEMORY_ARRAY* declaration. The initialization function is given in the provided file "*mem_init_func.txt*". Study the guideline "Initializing RAM from an External File", at page 224in "xst_userguide.pdf" to understand how the function was created.

Then in the main region of the *Architecture Body*, describe the working of the memory by implementing the read/write operations using behavioral style. Note that type casting is required to access the memory array, as the memory entry is of type *integer* but the address is of type *std_logic*. Type casting is performed using particular built-in functions:

```
<mem_array_name>(to_integer(unsigned(ADDR)))
```

Due to this and because other data types, such as integer, unsigned, string, etc., are used and are not included in the basic *IEEE.STD_LOGIC_1164.ALL* library, the following two libraries must be added to the top of the vhdl file where the libraries are declared: *IEEE.NUMERIC_STD.ALL* and *std.textio.ALL*.

Compile the vhdl file for any possible errors and then verify the correct operation of it using ModelSim. You need to copy the two provided "mif" files into the working directory. The memory is already configured as an instruction memory if you followed the guidelines above and initialized it using "*inst_mem.mif*". Simulate a read from a memory location using a "do" file. Then configure the memory as a data memory and initialize it using "data_mem.mif". Run a simulation using a "do" file, where you try to write to a memory location and then read from it. You can use the provided "*memory.do*" file as a starting point.

Bus – Task 2

The goal of this task is to implement the Bus. The bus is depicted at the bottom of Figure 1 while its interface (inputs/outputs) is presented in detail in the processor's specification document. In Figure 1, it is designed using tri-states buffers (described in lecture 3). However, the tri-state buffer-based implementation suffers from the drawback that the bus output takes an undefined value if more than one control signal is enabled.

For this reason, it is strongly recommended to implement the bus using a multiplexor (mux), instead. The mux has four data inputs. Every input is multiplexed to the output according to control signals that are exactly the same to the ones that drive the tri-state buffers. However, the number of control signals is 4 but the mux's control signal can be only two bits. Therefore, extra logic is needed to convert the 4 control signals into a two-bit signal. What logic is needed? In alternative, the 4 control signals can be combined into one vector, which is used to make the selection without any conversion to two bits. Note that in the mux-based bus implementation, one input must always be multiplexed to the output, no matter if any control input is disabled. However, this doesn't affect processor's correct operation unless the bus output is used.

Write the VHDL (any design style) code to implement the bus using one of the above design alternatives (mux or tri-states). The entity declaration is given in the following box. The signal *ERR* must be set when two or more control signals are set at the same time. The ERR signal is driven to a led. At the end, verify the correct operation of the bus by running a simulation at ModelSim. Use a "do" file.

```
entity procBus is
    Port ( INSTRUCTION : in  STD_LOGIC_VECTOR (7 downto 0);
           DATA : in  STD_LOGIC_VECTOR (7 downto 0);
           ACC : in  STD_LOGIC_VECTOR (7 downto 0);
           EXTDATA : in  STD_LOGIC_VECTOR (7 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR (7 downto 0);
           ERR : out  STD_LOGIC;
           instrSEL : in  STD_LOGIC;
           dataSEL : in  STD_LOGIC;
           accSEL : in  STD_LOGIC;
           extdataSEL : in  STD_LOGIC);
end procBus;
```

Top-level design of the datapath – Task 3

In the previous two tasks, you implemented the storage modules and the bus. The rest of the modules are ready from the previous labs. The adder (PC=PC+1) on the top left corner of the datapath can be implemented either by using a ripple carry adder or by expressing it in behavioral code. Verify that you have all the modules by looking at Figure 1. The final task is to implement the top-level design of the ChAcc processor datapath by connecting the modules, as is exactly depicted in Figure 1, using structural VHDL (components and port maps).

In ModelSim, create a new vhdl file and name it ***EDA322_processor.vhdl***. The *Entity Declaration* is given below in the text box. In the declaration region of the *Architecture Body* declare all the components and add the respective files to the project (if missing). Note that depending on which writing style you are using to create the port maps, you may not need to declare the components, but the files must be added to the project in either case.

In the main *Architecture Body*, write all the needed port maps. Some modules, e.g., the register, will require generic maps. The controller, which is implemented in the next lab, is assumed to be a black box. A mock controller is given (**procController.vhd**) to be able to connect the control signals (output from the controller) to the datapath components. Internal signals are also needed when connecting components to each other. These signals must be declared in the declaration region of the *Architecture Body*. Use the same names, as used in Figure 1. In cases a signal name is not given, give a name yourself. Use descriptive names (e.g., InstrToInstrReg), as they become very useful when debugging.

Finally, in Figure 1, note that some signals are connected to 7-segment displays, except for the bus error signal (1-bit) which is connected to a led. These signals must be connected to the respective output signals that are mentioned in the entity below. The other input signals of the entity (*CLK*, *ARESET* and *Master_load_enable*) are explained in the document processor.pdf.

```
entity EDA322_processor is
    Port (   externalIn : in  STD_LOGIC_VECTOR (7 downto 0); -- “extIn” in Figure 1
             CLK : in STD_LOGIC;
             master_load_enable: in STD_LOGIC;
             ARESETN : in STD_LOGIC;
       pc2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- PC
       instr2seg : out  STD_LOGIC_VECTOR (11 downto 0); -- Instruction register
       Addr2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Address register
       dMemOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Data memory output
       aluOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- ALU output
       acc2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Accumulator
       flag2seg : out  STD_LOGIC_VECTOR (3 downto 0); -- Flags
       busOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Value on the bus
             disp2seg: out STD_LOGIC_VECTOR(7 downto 0); --Display register
             errSig2seg : out STD_LOGIC; -- Bus Error signal
             ovf : out STD_LOGIC; -- Overflow
             zero : out STD_LOGIC); -- Zero
end EDA322_processor;
```

Demonstration

Tasks to be done for successfully completing this lab:

1. Write a VHDL module that implements the register and another that implements the memory. Verify their correct operation using ModelSim and "do" files. The memory module must be tested twice: once as instruction memory and another as data memory initializing it using the respective file among the provided mif files. Show the simulations to the instructor.

2. Write a VHDL module that implements the bus. Make the implementation using either a mux or tri-state buffers. Simulate the design using ModelSim. Show your results to the instructor.

3. Write a VHDL module that implements the top-level design of the ChAcc datapath. Connect all the modules using Figure 1. Assume that the controller is a black box. Compile your file in ModelSim and show the VHDL code to the instructor. You cannot simulate the design since the real controller is still missing.

Evaluation:

The instructor will check for the following:

Task#	Coding style	Simulation
1	X	X
2	X	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:

- Describe how you implemented each of the two storage elements: register and memory.
- Describe how you implemented the bus using the mux and any extra logic or the tri-state buffers.
- Show (a) snapshot(s) of the memory simulation waveforms, to demonstrate that both the write and read operations work as intended.
- Justify any design decisions you had to make (e.g., in which manner did you implement the bus and why) and mention any challenges you faced and how you overcame them (e.g., how did you use the given parameters to make the memory design generic).

Learning outcome:

After completing this lab, you should be able to:

- Implement storage components like registers and memory arrays.
- Implement VHDL modules using *generics.*
- Know how to initialize memory arrays using initialization files.
- Know how to connect many VHDL components using **structural** VHDL.

Hints and Tips

Using "init_memory_wfile" function

The function "init_memory_wfile" should be copy-pasted **without any changes** in the declaration region in the Architecture Body, **before** all signal declarations that make use of it but **after** the *MEMORY_ARRAY* declaration. However, *MEMORY_ARRAY* should be **initialized after** "init_memory_wfile" function is declared.

Design bounding check

At the end of this lab, due to absence of a functional controller, running the simulation is not possible yet. However, in order to check if all components are created and connected correctly, i.e. types and sizes of ports are consistent, you should attempt to start the simulation for `EDA322_processor` and check if the tool shows any error when the design is loaded.

7 is out of bound of 12

A very common error message is: "7 is out of bound of 12"! This is directly related to the dimensions of the MEMORY_ARRAY. Double check the range.

The values are stored in the memory starting from the last position

This case will happen if in the MEMORY_ARRAY declaration you chose to write (X donwto 0). In order to modify this, you should use (0 to X). However, this should not affect the functionality of your memory unit.