Digital Design Lab (EDA322)

Lab 2

Examiner: Prof. Ioannis Sourdis
TAs: Ahsen Ejaz, Ghaith Abou Dan, Fredrik Jansson, Konstantinos Sotiropoulos, Magnus Östgren, Neethu Bal Mallya, Panagiotis Strikos

Last Edited: 2024-01-23
Deadline: Your respective lab session during Study Week 4

1 Introduction

The goal of this lab is to implement the top-level design of the ChAcc processor. Before starting the lab, please prepare as described below.

1.1 Preparation

1. Complete Lab 1.
2. Listen to the corresponding coding tutorials.
3. Study ***Section 3 (Datapath)*** in the provided processor specification document (*processor.pdf*). This lab manual references Figures and Tables in *processor.pdf* document. So keep the document handy.
4. Study the lecture material up to the previous study week.

1.2 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.

2 Tasks

Figure 1 (processor.pdf) shows the datapath of the ChAcc processor. Towards completion of the top-level design of the ChAcc processor, this lab **requires** you to do the following three tasks:

1. Implement the storage components in the datapath, which includes the registers and memory using VHDL, and verify its functionality.
2. Implement the internal bus using VHDL, and verify its functionality.
3. Connect all the modules of the top-level design of the ChAcc processor, assuming the controller is a black box. For this purpose, use the mock controller provided. Due to this assumption, you will not be able to verify the correct operation of the whole datapath until the next lab.

2.1 Task 1: Storage components

2.1.1 Registers

Write the VHDL implementation of the register in `reg.vhd`, which has the interface described in *Table 5 (processor.pdf)*. You can make use of any design style: behavioral, dataflow, or structural. As given in *Table 6 (processor.pdf)*, ChAcc contains registers of multiple bit-widths. Therefore, implement the register using ***generics***. Verify their correct operation using QuestaSim and a do file.

Tips: If you want to configure a 1-bit register, it will still be a `std_logic_vector` i.e., `std_logic_vector(0 downto 0)`, and this is not the same as a `std_logic`. To connect a signal of type `std_logic` to such a `std_logic_vector(0 downto 0)` port, you can do a `port map` by name and include an index:

```
foo0:  foo port map (v(0) => s);
```

where `v` is a `std_logic_vector(0 downto 0)` in the component `foo` and `s` is a local `std_logic` signal.

2.1.2 Memory modules

Check the entity of the memory module in `memory.vhd`. The inputs/outputs for instruction and data memories are described in *Table 3 (processor.pdf)* and *Table 4 (processor.pdf)*. The structure of both memories are quite similar, with the main differences being:

- The Instruction Memory has no write port
- The Instruction Memory is 12 bits wide whereas the Data Memory is 8 bits wide

You should build only one memory module with generic width, and both read and write enable inputs. For the Instruction Memory, you have to set the *writeEn* to constant 0, and *dataIn* must be connected to something, for example, all zeros.

1. The memory is a data array and its implementation is conceptually the same as 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. Define the type, `MEMORY_ARRAY` as a two-dimensional array of data as below:

   ```
   type MEMORY_ARRAY is ARRAY (<number_of_entries>) of <data_type>(<data_size>);
   ```

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

2. Now, create the memory array by declaring a signal of type `MEMORY_ARRAY`.
3. Initialize memory array by calling the function `init_memory_wfile` using `INIT_FILE` as the argument.

   - `INIT_FILE` keeps the filename used for the initialization of the memory array. The memory can be initialized with zeros or a particular data set using the input file.
   - The initialization function is given in the provided file ***mem_init_func.txt***. Study the guideline *"Initializing RAM from an External File"*, on page 224 in *xst_userguide.pdf* to understand how the function was created.
   - The function should be included in the declaration region within the *Architecture Body*; before all signal declarations that use it but after the `MEMORY_ARRAY` declaration.

4. Describe the working of the memory by implementing the read and write operations using behavioral style in the main region of the *Architecture Body*.

   - The write and read operations are synchronous (make use of the `CLK` input)
   - Note that typecasting is required to access the memory array, as the memory entry is of type integer, but the address is of type `STD_LOGIC`. Typecasting is performed using a particular built-in function as given below:

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

- Since we use data types, such as integer, unsigned, string, etc., not included in the basic library (`IEEE.STD_LOGIC_1164.ALL`), 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 files for any possible errors and verify their correct operation. You need to copy the provided `i_memory_lab2.mif` and `d_memory_lab2.mif` file (Available in Canvas: *Files* > *Labs* > *Lab 2* > *lab2_files*) into the project directory, which will initialize the memory. Run simulations using "do" files to test the design; e.g., write to a memory location and then read from it.

Tips: Note that we have provided default values for `INIT_FILE`, `DATA_WIDTH` and `ADDR_WIDTH` in `memory.vhd`. All the default values can be changed when the *memory* component is instantiated in another design. However, when you create a "do" file to test your *memory* design, make sure that the default value matches the name of an existing file in the project directory (e.g., `i_memory_lab2.mif` or `d_memory_lab2.mif` we provided for this lab). Similarly, before running the respective "do" file, the default value of `DATA_WIDTH` should be **12** for Instruction Memory and **8** for Data Memory.
Tips: 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 downto 0). To modify this, you should use (0 to X). However, this should not affect the functionality of your memory unit.

2.2 Task 2: Bus

This task aims to implement the internal bus of the ChAcc processor. The bus is depicted at the bottom of *Figure 1 (processor.pdf)*. The bus can be implemented with a 2:4 decoder and four tri-state buffers as presented in detail in the processor specification document (Section 3.7, *processor.pdf*). An alternative implementation is to use a 4:1 multiplexer. If you go with a multiplexer implementation then keep in mind that *Section 3.7 (processor.pdf)* outlines a tri-state buffer version specifically, and some design choices (a default state/value of high impedance for example) don't make any sense for a multiplexer based bus.

Write the VHDL (any design style) code in `proc_bus.vhd` to implement the bus using one of the above design alternatives. The inputs/outputs are described in *Table 10 (processor.pdf)*. Run the simulation using a `proc_bus.do` file, and verify the correctness of your design.

Tips: To keep your code clean and readable, we recommend not using raw bit vectors for the bus states, but instead named constants. These are already provided for you and available in the `chacc_pkg` package(in `chacc_pkg.vhd`).

2.3 Task 3: Top-level design of the datapath

The final task is to implement the top-level design of the ChAcc processor datapath by connecting the modules, as depicted in *Figure 1 (processor.pdf)*, using structural VHDL (components and port maps).

- The two adders in the datapath can be implemented either by using a ripple carry adder (from Lab 1) or by expressing it in behavioral code. The adder with the +/- sign is the one you'll have to use for the instructions JE, and JNE, JZ. Based on the offset (i.e., busOut[7]), you've to either add or subtract the offset from (PC+1). Please read the footnote in Table 1 (processor.pdf). If you are using the RCA, take care of the 2's complement for the second operand as you did for the ALU (Lab 1).
- Verify that you have all the modules by looking at *Figure 1 (processor.pdf)*.
- The entity declaration for the top-level design is provided in your template files. Please make sure you have the same entity definition as in `EDA322_processor.vhd`

- In the declaration region of the *Architecture Body*, declare all the components and also add the individual 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 is assumed to be a black box (you will implement it in Lab 3). Use the mock controller provided (Canvas: *Files* > *Labs* > *Lab 2* > *lab2_files* > `mock_controller.vhd`) 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 (processor.pdf)*. In cases where a signal name is not given, give a name yourself but try to use descriptive names to make it easier when debugging (e.g., *jumpOffsetToAdder*).
- Finally, in *Figure 1 (processor.pdf)*, note that some signals (marked red) are connected to the 7-segment displays. These signals must be connected to the respective output signals.

Compile your file in Questa and show the VHDL code to the instructor during the demonstration. You cannot simulate the design since the real controller is still missing. However, 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.

Tips: **Design bounding check**: Running the simulation for Task 3 is not possible yet due to the absence of a functional controller.
Tips: **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.

3 Demonstration and Evaluation

The lab will be evaluated according to the checked aspects in the table below. To demonstrate your successful completion of Lab 2, keep all the essential files and simulation results ready to be presented to a TA.

Task#	Files	Coding Style	Simulation
1	`reg.vhd`, reg.do, `memory.vhd`, memory.do	✓	✓(Using .do files and the given .mif files)
2	`proc_bus.vhd`, proc_bus.do	✓	✓(Using .do file)
3	`EDA322_processor.vhd`	✓	Only compilation

– **The demo must be completed during your registered lab session**.
– **Note that there is no code upload required for this lab.**