NIOS2 Multiprocessor: Implementation phase

In the next 3 lab sessions, you will implement an NIOS2 non-coherent multiprocessor in VHDL. You will use the design described in this document and the annexed diagrams, and the project skeleton distributed through the course website.

Handout

The skeleton of the project is given to you in the handout distributed through the course website. The handout is organized in the following way:

- **`docs/`:** This folder has all the documentation you need for this phase of the project. There you will find this document and another pdf with only the state machines and datapath diagrams.
- **`vhdl/`:** All the code is in the `/vhdl` folder. During the implementation, you should only care about the files in `/vhdl/memory_hierarchy` and `/vhdl/testbenches` folder. There are three main components you need to implement: the bus controller, the cache controller, and the bus arbiter. You can add more modules if you want, even though there is no need for any additional module. However, you should NOT modify any of the files that are already present in this and other folders under the `vhdl` tree.
- **`modelsim/`:** This folder has a ModelSim project you can use to test your design. The project already includes all necessary `.vhd` files. If you create new files, please add them to this folder.
- **`simulator/`:** This folder contains the NIOS2 assembly simulator and some sample assembly code for you to test your design. You can use the simulator to generate tests for your design.

NIOS2 Bus-based non-coherent multiprocessor design

The design enhances the baseline design adding a bus controller, and enabling the cache controller to communicate with the bus. You are given both the state machine and the datapath descriptions of the design. There are 2 main state machines that communicate with each other: the cache and bus controller state machines.

The cache state machine implements the same functionality of the baseline state machine, but now it communicates with the bus instead of directly with the RAM. The bus controller state machine takes requests from the cache and passes them along to the RAM.

The diagram for the state machines assumes that the state machine outputs are not registered (i.e., both the input condition and the outputs should happen in the same cycle).

In the datapath diagrams, we have inputs, outputs, and several abstract blocks. The main blocks of the cache controller are:

- **`TagArray`:** It takes an address and looks it up in the tag array, and outputs in the following cycle whether the access is a hit or a miss. In case of a hit, it also outputs the id of the set; and in case of a miss it outputs the victim's set id to be evicted. The `tagArray` will only perform lookups if `tagLookupEn` is asserted. If `tagWrEn` is asserted, the `tagArray` will write a block instead. The implementation of this block is already given to you.
- **`DataArray`:** It takes an address and fetches the data stored in the cache. `DataArray` returns two blocks of data in the following cycle of the lookup. The control must choose the correct set. The implementation of this block is already given to you.
- **`RdDataTriStateBuffer`, `BusTriStateBuffer`:** These blocks are tri-state buffers. Whenever their `*OutEn` signal is not asserted, they should output (`other => 'Z'`). Wheh `*OutEn` is asserted, they must drive the bus. These blocks are necessary in order to avoid multiple devices driving the bus at the same time.
- **`CpuReqReg`, `VictimReg`:** These blocks are normal registers with `WrEn` values.

The bus main blocks are:

- **`CmdDecoder`:** This block decodes commands passed to the bus into commands that can be understood by the memory. It is a purely combinational block.
- **`Arbiter`:** This block implements a round-robin arbitration. It keeps track of the last request serviced and uses this information to adjust the priority of the requests when there is a conflict. This block outputs `'1'` to `arbiterReqValid` whenever there is a valid request pending, with the id of the cache to be serviced in `arbiterReqId`. The implementation of this block is already given to you.
- **`BusTriStateBuffer`:** This block is a tri-state buffer, same as described above.

Implementing the design

Ideally, you will only modify the `bus_controller.vhd` and `cache_controller.vhd` files. These files already contain a skeleton of the state machines described in the design.

The `vhdl/includes/mem_types.vhd` contains all the important definitions, such as address widths and word width. It also contains the type definitions used across the design.

This file also includes some helper functions for example to extract tag values, block addresses and word offsets from memory addresses. Finally, it contains definitions for the bus commands. You may change the values of the definitions, but the width and labels must remain the same, so that we will be able to test your designs.

Testing the design

You will verify whether your implementation is functionally correct using ModelSim to simulate the execution. In order to ease that task, we are distributing three testbenches that exercise most parts of the state-machines in the design. You are encouraged to come up with your own tests and modify the distributed testbenches in order to guarantee that your implementation is functionally correct. The testbenches can be found under the `/vhdl/testbenches` folder:

- **`MultiprocessorTestbench.vhd`:** This is for the end-to-end test. This testbench simulates the two cores executing different binaries. You can verify whether your design passes the tests using the waveforms, or you can enhance this testbench to enable it to automatically verify whether the tests passed or not.
    - In order to use write tests for this testbench, you need to write the assembly code that will be executed by the processors in this testbenches. You can use the `nios2sim.jar` simulator in the `/simulator` folder. To run it, use the following command: `java -jar nios2sim.jar`. Using the simulator GUI, you can assemble your code, export it to `.hex` files and simulate your code with a single core. This tool is useful to debug your assembly code before executing it with your cores.
    - You should export the `.rom` files to `vhdl/testbenches/binaries/ROM{0,1}.hex`, and the testbench will automatically load them when executing tests.
- **`Cache_controller_tb.vhd` and `Bus_controller_tb.vhd`:** They contain very basic tests for the cache controller and the bus controller respectively. These tests will exercise most parts of the cache controller and the bus controller state machines. It is recommended that you guarantee that your tests are passing in these testbenches separately before you test the integration between the cache controller and the bus controller with the `MultiprocessorTestbench.vhd` testbench. Please see the code of these testbenches in order to understand what is it doing.

Using ModelSim

In order to run a particular test, open the project in ModelSim. The project is located under `modelsim/nios2-bus.mpf`. In the ModelSim screen, you can compile all files with the `Compile All` command (Menu Compile->Compile All). Please note the code we distributed will not compile.

After compiling your project, you can start a simulation with Menu Simulate->Start simulation… . In the dialog that appears, choose your testbench by opening the *"work"* tree and selecting one of `"cache_controller_tb"`, `"bus_controller_tb"` or `"multiprocessortestbench"` by clicking on it. The name of the testbench should appear in the *"Design Unit(s)"* text box. Click on OK to see the simulation screen.

The simulation screen includes the *Sim*, *Objects*, and *Wave* panes. You can select a module in the Sim pane, and its objects will show up in the Objects pane. From there, you can add the Objects to the Wave pane, and when you start the simulation, you will see the waveforms of the objects. The wave and sim panes might be hidden in other tabs, look for them in the tabs if they don't appear immediately.

There are many buttons in the toolbar that will help you debug your project. Look for the tooltips for help. The most useful ones are:

- **Compile all:** compiles all files in the project.
- **Restart:** restarts the simulation (resets the waveforms, and reloads recently compiled files)
- **Run, ContinueRun and Run -All:** Actually runs the simulation. After the simulation is run, you can see break it with the Break button and the Wave pane should be updated with the waveforms.
- **Step {Into, Over, Out} (current thread):** You can use these buttons to step through your code, once you break simulation or put a breakpoint in your code. It is tricky to debug sequential logic this way since it is inherently parallel. However, this feature is very useful in debugging combinational logic.

Testbenches often verify correctness using assertions. These assertions will be printed out in the transcript pane, if they fail. VHDL does not have a command to end simulation, and thus, in the `cache_controller_tb` and `bus_controller_tb`, we use an assertion to break simulation in the end. That is not an error and a simulation that reaches that assertion without any errors, passed the tests.