**CPEG324 Lab 3**

By Mark Betters

Team Members: Mark Betters, Elton Mwale

***Abstract:*** For this lab, my team member and I had to develop a VHDL calculator. We made a diagram of the calculator, modeling it as a single-cycle CPU. Then, we implemented the calculator in VHDL, tested each of its components separately, and then tested the whole CPU.

***Division of Labor:*** The lab was split into two phases, a diagram/presentation phase and a coding phase.

For the diagram/presentation phase, Elton and I worked on a block-and-arrows diagram collaboratively in draw.io. We sought to design a single-cycle CPU at a high level, showing the CPU’s components and their respective inputs/outputs. We also made a truth table to show how the Instruction Decoder component worked. We then presented the diagram and truth table on April 19.

The coding phase of the lab was split into components, or CPU “stages.” Our CPU stages included a Clock Generator, Instruction Decoder, Register File, Sign Extender, ALU, and Printer. Every component was implemented and tested separately. The following gives who coded which components:

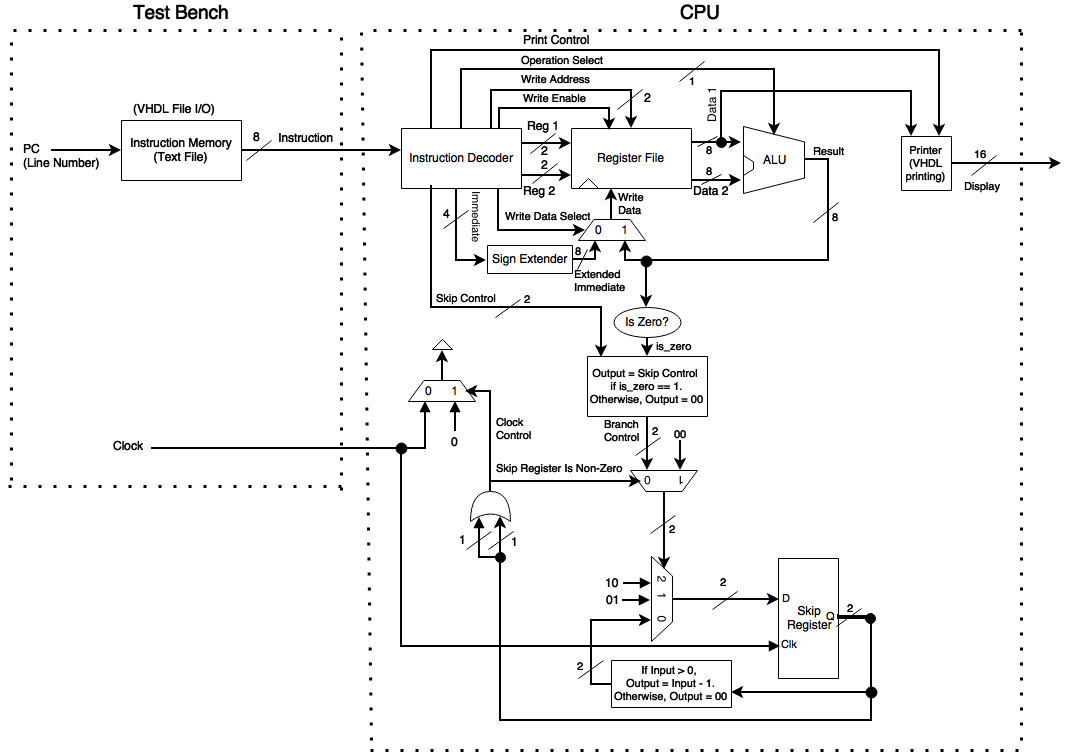
* I implemented and tested the Clock Generator, Register File, and Sign Extender.
* Elton implemented and tested the ALU, Instruction Decoder, and Printer.

***Detailed Strategy:*** Implementation of each component was done using behavioral VHDL architectures if the component was simple or structural VHDL architectures if the component could be broken down into smaller components. Every component was given its own folder, with the component VHDL, any sub-component VHDL files, and the test-bench VHDL all in the component folder. An example would be the ALU’s folder, which has alu.vhdl (the ALU component), arithmetic.vhdl (a sub-component), full\_adder.vhdl (a sub-component), and alu\_tb.vhdl (the ALU test-bench).

Component test-benches test the components on their own, ie- not attached to any other components. We implemented each component and tested them separately, until they passed individual tests.

Finally, we combined all the components into a CPU architecture (using port-maps) and tested the CPU with a test-bench that feeds instruction vectors and a clock signal into the CPU and prints out the CPU’s resulting display output signals.

***Results:*** The following is the CPU’s block diagram:



Datapaths:

- The add/subtract instructions follow a path through the Instruction Decoder, reading from the Register File, operating in the ALU, then writing back to the Register File.

- The load instruction follows a path through the Instruction Decoder, extend the immediate value using the Sign Extender, then write to the Register File.

- The compare-and-skip instruction follows a path through the Instruction Decoder, reads from the Register File, subtracts the read data in the ALU, passes the subtraction result through a Is Zero checker, which triggers a series of logical changes that results in the clock being driven to 0 if the values being compared were equal.

- The print instruction follows a path through the Instruction Decoder, reads the value to be printed from the Register File, then sends the value to the Printer, which then drives the display signals that are output from the CPU (and printed by the CPU’s test-bench)

To implement the components, many

***Conclusion:***

***Appendix I (notebooks):*** We each spent about 20 hours on this assignment.

***Appendix II (VHDL files):*** GitHub Link: …

***Appendix III (testing):*** See the testing instructions in the README of the GitHub repository linked in Appendix II. Testing methodology is also discussed there.