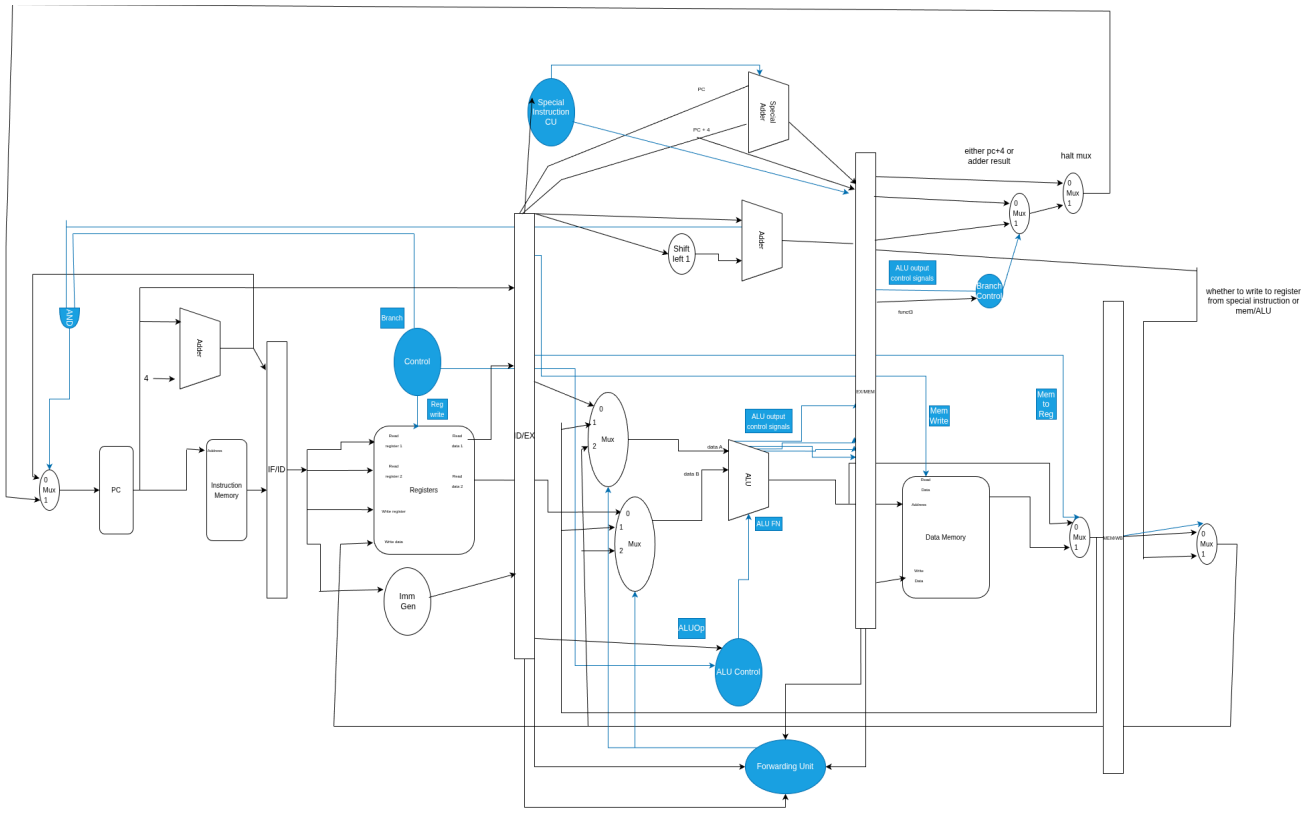
A RISC-V CPU Implementation in Verilog

Contains pipelining, forwarding, hazard detection and branch flushing

Mohammed Amer, Aly Youssef



Our Design

We designed our CPU in the usual five stage implementation with registers in between each stage. We prevented hazards by adding stalling in the control unit and added a forwarding unit. We supported instructions that depend on the Program Counter (PC) using a module that we called ‘Special Instruction’ as these instructions took the PC as an operand instead of two registers or a register and an immediate. Based on the opcode we figure out whether we want to write to registers the output of the Special Instruction part or the standard ALU/MEMORY part. According to our testing 42/42 instructions work correctly.

Issues we encountered

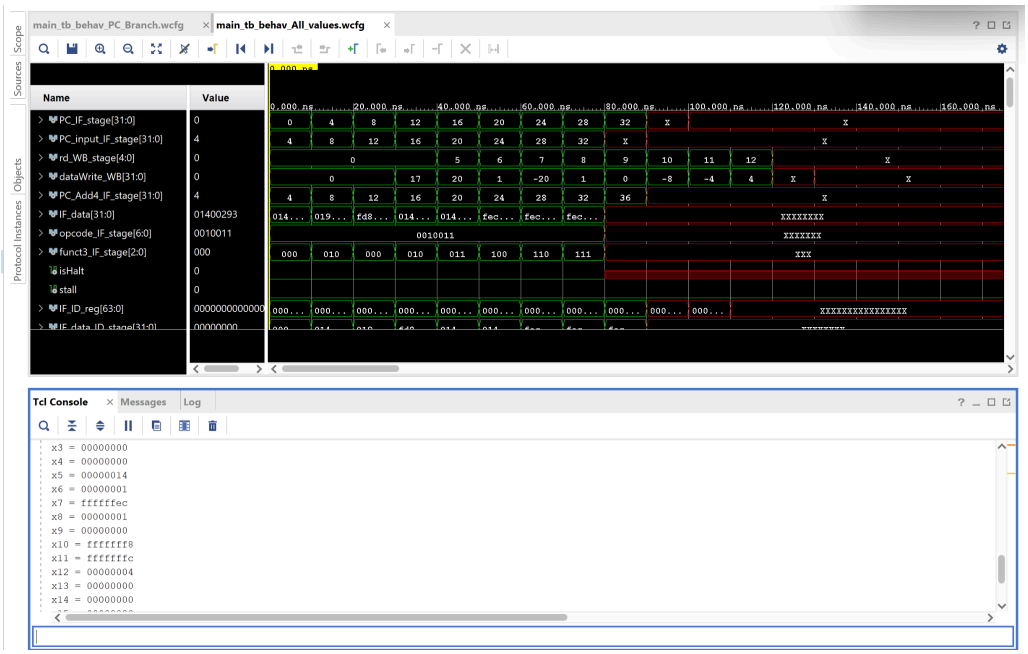
We found it somewhat difficult to figure out whether our CPU was stalling correctly and whether it was indeed loading/storing into memory. To solve this problem we wrote a task in the register file module and integrated it into the main testbench that outputs the values of all the registers at the end of program execution which we compared with the true results to make sure that our CPU was working properly.

We also found it very difficult to combine the data and instruction memory as it would need extensive design changes and many stalls to prevent reading and writing at the same time which would corrupt the data.

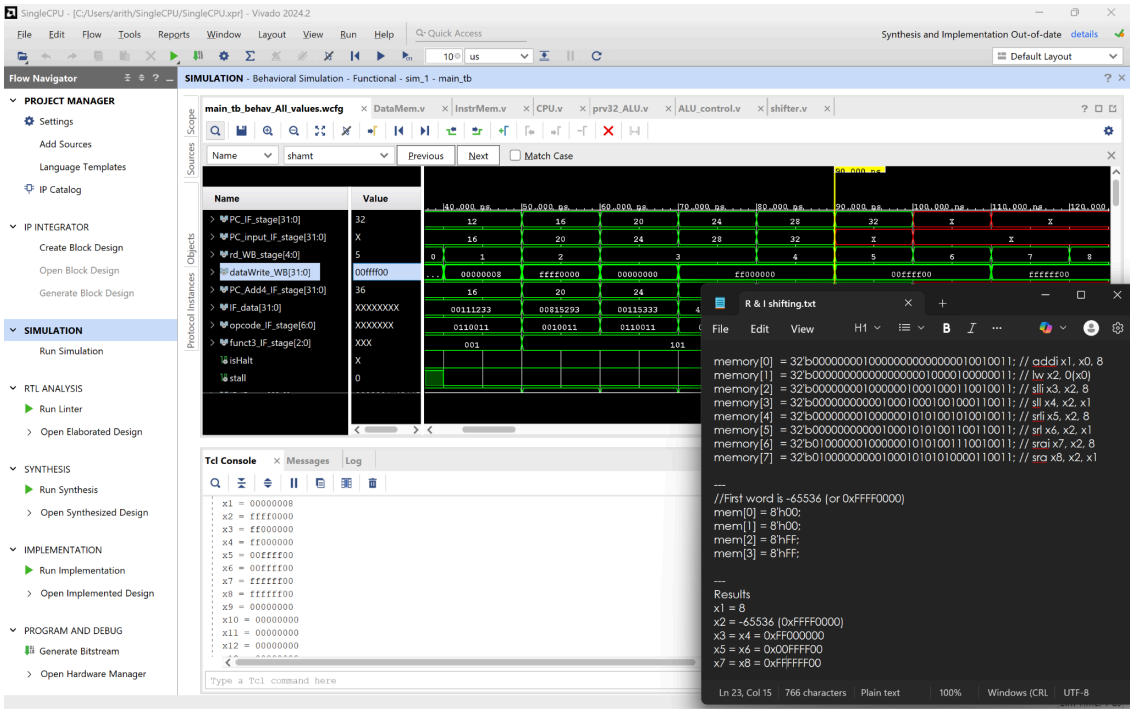
An interesting point was deciding on how to structure the verilog file that calls all the different modules. We found that the simplest way was to make a wire for every signal that’s needed at the specific stage so we had wires like ‘rd_EX_stage’, ‘rd_MEM_stage’ and ‘rd_WB_stage’ for the register destination at every stage. And every part of the verilog file is dedicated to a specific stage with the registers between each stage in the code which makes it very intuitive and similar to how the block diagram is structured.

Screenshots of our Waveforms

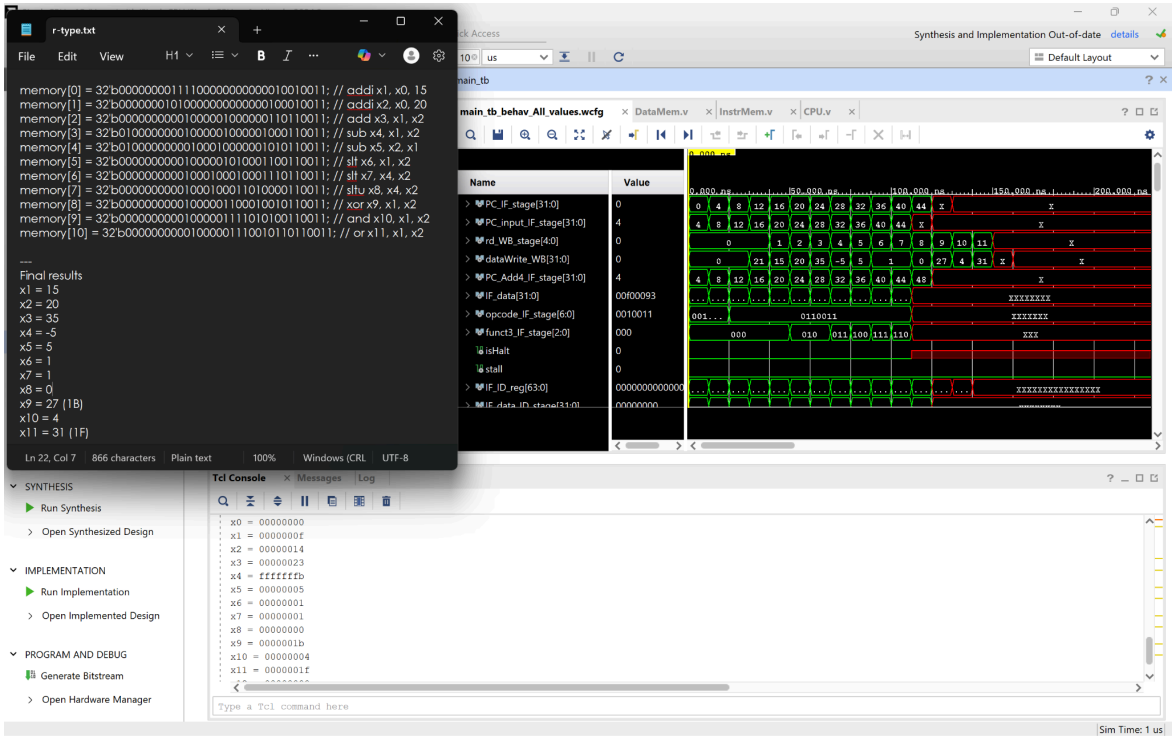
I-type Instructions



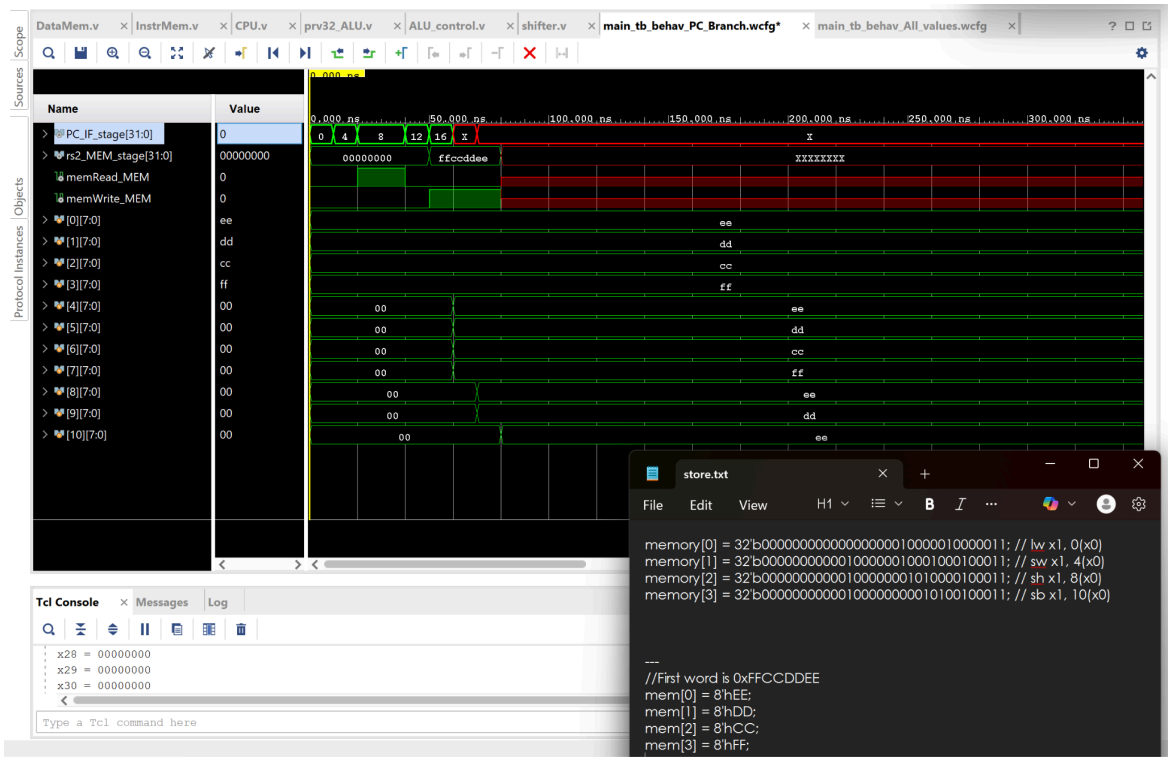
Shifting Instructions



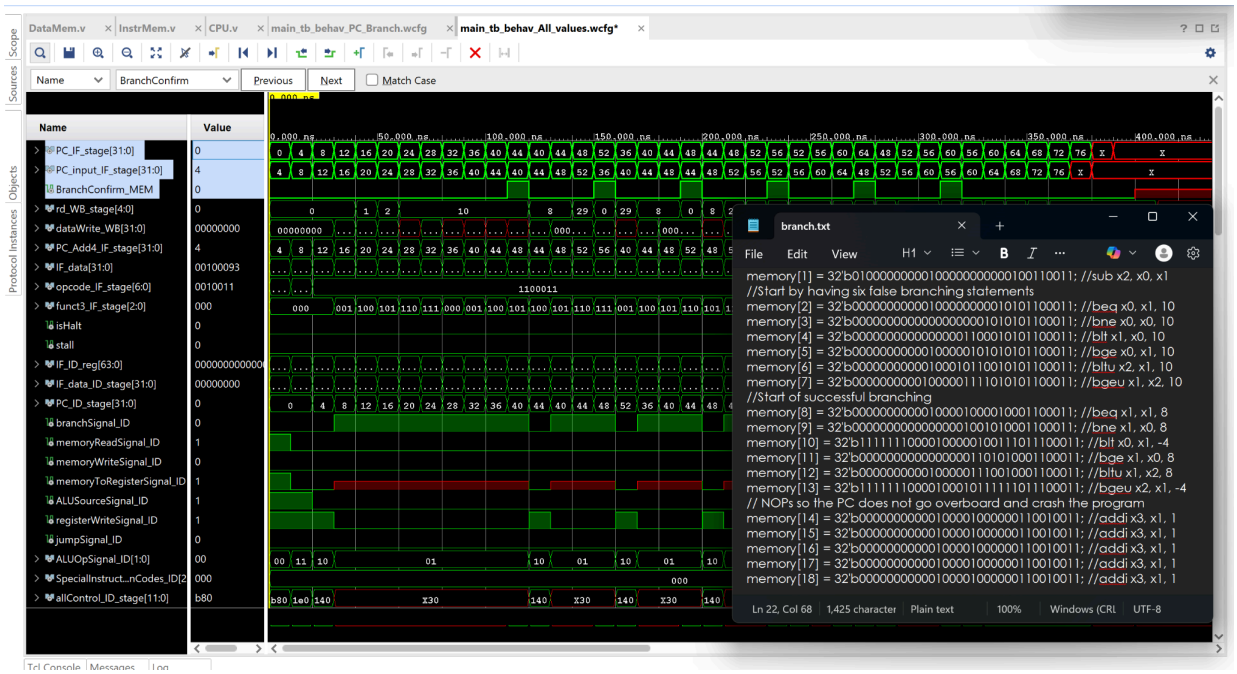
R-type instructions



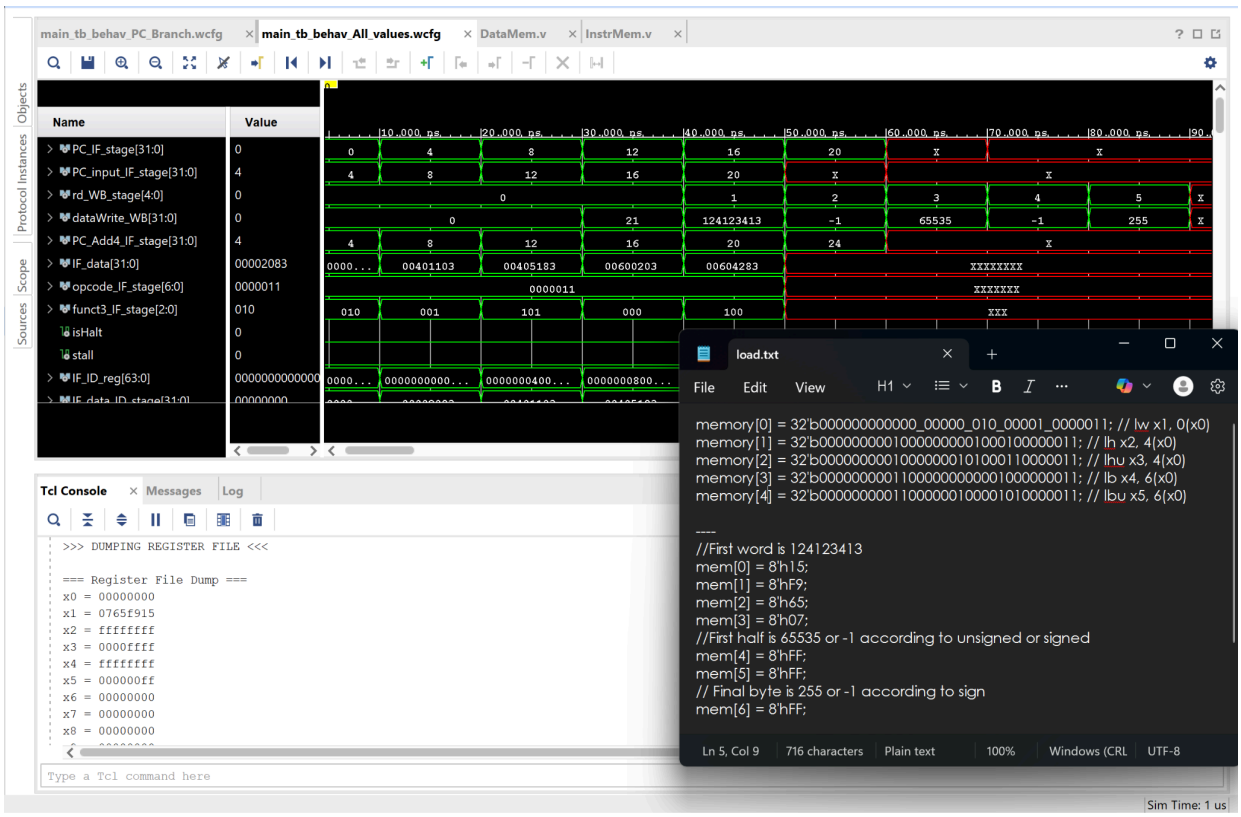
Store Instructions



Branch Instructions



Load Instructions



Further Improvements

This CPU implementation doesn't unfortunately take advantage of the different types of Caches which might increase speed. It also doesn't have access to any DRAM or Secondary storage, it only relies on the predefined tiny Data memories and Instruction memories. It also doesn't offer a way to view or use the results of its computation in a useful way so it is unlikely to be used in real-world applications.

Another potential improvement is to implement this CPU on a PCB to allow it to run natively instead of emulating it on an FPGA, doing this would allow us to add many custom features like video output or input from devices like mice or keyboards.

CONCLUSION

This project was incredibly fun, interesting and an incredible learning experience for us. We learned the basics and somewhat intermediate parts of how CPU's work and how to design one. Our implementation is simple, intuitive and relatively easy to add further improvements to as we intended to go further with this project than we were able to. Thank you for leading us every step of the way until we wrote a functional central processing unit ourselves and we hope to use this knowledge to benefit others.