CPU Verification Environment Project Report (ZhenCPU)

ZHEN YU CHEN

1. Introduction

The purpose of this project was to build a complete simulation-based verification environment for a simplified CPU. Instead of creating a real hardware chip, the project used simulation to check whether each internal module—such as the register, counter, multiplexer, ALU, controller, and memory—worked correctly. Through simulation, I could observe how signals changed over time and determine whether the CPU followed the correct behavior. This allowed me to confirm that the processor executed instructions properly through the standard FETCH–DECODE–EXECUTE cycle.

To verify the CPU's correctness, I examined waveforms, simulation logs, assertion results, and functional coverage reports. These outputs provided clear evidence that the CPU behaved as expected under different conditions. The final deliverables include waveform diagrams, verification logs, coverage summaries, and documentation that demonstrate the CPU's functional accuracy. Overall, the project produced a fully working verification environment and showed how simulation-based verification is used in real hardware design to detect issues early, long before any physical chip is built.

2. Motivation and Goals

I selected this project because I wanted to obtain practical experience with verification methodologies that are widely used in the semiconductor industry. In modern chip development, pre-silicon verification plays a critical role, as errors discovered after fabrication can lead to significant financial and engineering setbacks. By working on this project, my goal was to better understand how verification engineers analyze a design, identify incorrect behaviors, and establish confidence that the hardware will operate correctly once implemented. At the beginning of the project, I also used EDAPlayground as an initial environment to develop and test my SystemVerilog code. This platform provided a simplified setting for early exploration and allowed me to gradually transition into more advanced verification tools with a stronger foundation.

My learning objectives included strengthening my proficiency in SystemVerilog, improving my ability to analyze waveforms, understanding how assertions contribute to design robustness, and gaining familiarity with functional coverage as a measure of verification completeness. From a development perspective, my goal was to create a verification environment capable of testing the full instruction flow of a CPU and ensuring that both individual modules and the integrated system behaved as intended. By the conclusion of the project, I successfully achieved these objectives. I enhanced my technical skillset, developed a clearer understanding of professional verification practices, and gained project-planning experience that aligns with workflows used in ASIC and SoC verification.

3. Tools and Technologies

Tools Used in the Project

- **SystemVerilog** – The main hardware description and verification language used to write the CPU modules and the testbench. It allowed me to describe digital logic behavior, create assertions, and define coverage models.
- **Questa**– The primary simulation tool used to compile the SystemVerilog design, run test cases, collect functional coverage, and generate waveforms. It served as the core engine of the verification workflow.
- **Cadence SimVision** – A waveform viewer used to inspect and interpret signal activity during simulation. It helped me understand timing behaviors, analyze state transitions, and debug unexpected results.
- **Cadence JasperGold (Coverage Viewer)** – Used to visualize functional coverage and verify that all instructions, states, and CPU behaviors were exercised by the testbench.
- **EDAPlayground** – An online platform used at the beginning of the project to write, compile, and test early SystemVerilog code. It provided a simple environment for initial experimentation before moving to full EDA tools.

- **Microsoft Project** – Used to plan the project timeline, organize weekly tasks, and ensure that milestones were completed on schedule.
- **Microsoft Word and Excel** – Used to create documentation, maintain the project summary, develop the risk list, and prepare all required written deliverables.
- **GitHub** – Used only to upload the final version of the project files for submission. It served as a storage location for the completed code and documentation rather than a version-control tool during development.

Technologies and Supporting Systems

- **Simulation-Based Verification** – The main verification method used to test the CPU design without physical hardware. It allowed repeated testing and detailed observation of signal behavior.
- **Assertion-Based Verification (ABV)** – Provided automatic checks for design rules, such as ensuring correct FSM encoding and correct signal behavior during each clock cycle.
- **Functional Coverage** – Used to measure how thoroughly the CPU was tested. It confirmed that different opcodes, states, and instruction flows were executed during simulation.
- **Layered Testbench Architecture** – A structured approach that separated stimulus generation, monitoring, checking, and coverage collection into different layers. This made the testbench easier to understand and maintain.

4. Software Delivery Process / SDLC Activities

- **Understanding What the CPU Needed to Do**
  The project began by clearly identifying how the CPU should behave and what parts of it needed to be tested. This helped me understand the overall goal before writing any code.
- **Planning How to Test the CPU**
  After understanding the requirements, I designed the testbench. This plan explained how inputs would be sent to the CPU, how outputs would be checked, and how errors would be detected. This planning gave the project a clear structure.
- **Building and Testing One Module at a Time**
  Instead of building the whole CPU at once, I worked on one module at a time (such as the ALU or FSM). For each module, I wrote the code, ran a simulation, and checked if it behaved correctly. If it did not, I fixed it and tested again.
- **Using Continuous Simulation to Catch Mistakes Early**
  Simulations were run many times throughout the project. Each simulation helped reveal small issues before they became bigger problems. When something looked wrong, I corrected it immediately and tested again.
- **Combining Modules Slowly and Carefully**
  After each module worked correctly on its own, I started putting them together. I integrated the modules step by step, rather than all at once. This made it much easier to find and fix issues during the integration process.
- **Testing the CPU as a Complete System**
  When all modules were connected, I tested the CPU as a whole. I checked whether it could correctly fetch an instruction, decode it, and execute it—the basic steps a CPU must follow to run programs.
- **Retesting and Making Small Improvements Throughout the Project**
  Every time a new part was added or updated, I tested the CPU again to make sure previous functions still worked. This created a steady cycle of testing and improvement.

- **Checking Progress Every Week**
  I compared my weekly progress with the project plan to make sure I stayed on schedule. This kept the work organized and prevented delays.
- **Reaching a Stable Final Design**
  By following these steps—planning, building, testing, integrating, and retesting—the project moved forward smoothly. The final result was a stable CPU design that had been thoroughly tested through simulation.

5. Work Completed

All planned work was completed exactly as described in the original project plan. Every CPU module was successfully verified through simulation, and the full processor operated correctly under the complete instruction flow. The project included module-level verification, subsystem integration, instruction-level testing, assertion checking, and functional coverage analysis. Waveforms confirmed correct data movement, state transitions, and ALU operations, while coverage results indicated that all relevant instructions and CPU states were exercised.

No schedule delays occurred, no tasks were postponed, and no unplanned expansions of scope were necessary. The outcome closely matched the initial expectations, demonstrating that the combination of planning, steady execution, and incremental testing was effective at managing both the project timeline and the technical complexity.

6. Risk Management

At the start of the project, I created a formal risk list to identify possible problems, such as simulator instability, integration failures, or low functional coverage. Even though none of these risks actually occurred, planning for them early helped keep the project on track. Because I understood what might go wrong, I was able to organize my workflow in a way that reduced the chance of issues appearing later.

The main risk-mitigation action was verifying each module on its own before combining them. This step greatly reduced integration risk because any mistake inside a single module could be found and fixed early. Weekly sanity tests also played an important role. By running small, quick simulations every week, I could check the overall health of the design and make sure no unexpected problems were developing. This constant monitoring prevented minor issues from growing into larger challenges.

I also prepared simple contingency plans in case a risk became a real problem. For example, if the simulator had become unstable, I planned to switch to smaller isolated tests or use EDAPlayground to troubleshoot. If functional coverage had been too low, I was prepared to write additional tests or adjust the coverage model. If integration had failed, I could return to module-level testing, correct the behavior, and then integrate again gradually. Although these backup plans were never needed, having them ready gave me confidence throughout the project.

Overall, risk management helped create a smooth and predictable development process. By identifying risks early, using preventive strategies, and preparing contingency plans, the project avoided major obstacles and reached completion without delays.

7. Change Management

The project experienced no major changes to its scope, schedule, or objectives. All work proceeded according to the original plan. Minor adjustments, such as refining certain test behaviors, modifying waveform dump settings, or adding small coverage points, were made as part of the natural development process but did not change the direction or structure of the project. The absence of major changes indicates that the initial plan was realistic and well-constructed, and that the development process remained stable throughout the semester.

8. Project Plan Maintenance

Throughout the entire project, I made a consistent effort to keep all project planning documents—such as the project summary, risk list, schedule, and task breakdowns—updated and accurate. Each time I completed an important milestone, such as verifying a specific module or integrating the entire CPU, I immediately recorded the progress in the project summary and schedule. Keeping these documents synchronized with the actual work helped ensure that the project stayed organized and that there was always a clear picture of what had been completed and what still needed attention.

The risk list was also reviewed regularly, even though no major risks occurred. Whenever I passed a development stage that could have triggered a risk—such as module integration—I checked the list to confirm whether any potential issues needed to be updated or removed. This habit kept the risk list meaningful and helped maintain awareness about possible challenges, even if they never materialized. In addition, reviewing the risks helped reinforce confidence that the project was staying on track.

The schedule, created at the beginning of the project, was updated weekly to reflect completed tasks and upcoming goals. If a task took more or less time than expected, I adjusted the timeline so that the remaining work could be planned realistically. This prevented the schedule from becoming outdated and ensured that it continued to serve as a useful guide rather than just an initial plan. Even though the project proceeded smoothly without unexpected delays, these updates helped maintain accuracy and accountability.

By consistently updating all planning documents, I developed strong project-management habits. These documents were not treated as static items but as living tools that supported the project from start to finish. Keeping everything up to date made it easier to evaluate progress, maintain clarity about next steps, and communicate the overall status of the project. In the end, this regular documentation played an important role in helping the work progress smoothly and predictably.

9. Retrospective

This project gave me my first complete experience with a full hardware-verification cycle, from initial planning to module testing, system integration, debugging, and final verification closure. One of the most important lessons I learned was the value of modular verification. By confirming that each component worked correctly before combining them, I avoided the confusion that often happens when too many untested pieces are integrated at once. This step-by-step approach made the integration phase much smoother than I expected and showed me how powerful disciplined and organized engineering practices can be.

Another major insight came from early and frequent waveform analysis. Waveforms revealed details that were not visible in simulation logs and helped me understand the exact timing behavior of the CPU. Many issues that were confusing at first—especially in timing-dependent modules like counters, controllers, and the FSM—became clear once I visualized the signals. Waveforms also helped me confirm when a problem was not in the design but in the test environment. Several times, the CPU appeared to behave incorrectly, but the waveform showed that the design itself was correct. The real issues came from the testbench, such as incorrectly written assertions or missing clock connections. Without careful waveform analysis, these mistakes would have been much harder to diagnose. This experience taught me how essential waveforms are not only for understanding CPU behavior but also for distinguishing between design problems and testbench problems.

Functional coverage also changed the way I think about testing. Instead of only checking whether the CPU could run, coverage results showed whether all important instruction types, branches, and states were actually exercised. This shifted my mindset from simply “Did the simulation finish?” to “Did we truly test all meaningful behaviors?” This difference—between completion and completeness—is something I will carry into future verification work.

Looking at the overall workflow, I realized that strong initial planning, steady weekly milestones, and continuous integration were key factors in the project’s success. The organized schedule kept the work moving at a reliable pace, and integrating components as soon as they were ready made the system more stable over time. If I were to improve the process, I would define the coverage model earlier to guide test creation more

effectively and add more automation to regression testing so repeated tasks could run faster and more consistently.

Overall, this project helped me grow both technically and professionally. I now have a much clearer understanding of what verification engineers do in real chip-development environments: how they debug complex interactions, how they use coverage to measure verification progress, and how they rely on disciplined planning to manage large systems. Completing this verification environment has increased my confidence in pursuing future roles in ASIC or SoC verification and silicon validation.

10. Conclusion

The ZhenCPU Verification Environment achieved its goal of thoroughly validating a simplified CPU using a structured simulation-based approach. The project confirmed that each module operated correctly on its own and that the full processor functioned reliably when executing instructions through its complete cycle. The verification artifacts—such as waveforms, assertion checks, and coverage results—demonstrated that the CPU behaved consistently under all tested conditions and met its intended functional requirements. These outcomes show that the verification environment was effective and that the CPU design was successfully evaluated without needing physical hardware.

In addition to confirming the technical correctness of the CPU, the project resulted in a well-organized verification framework supported by clear documentation, consistent testing procedures, and disciplined project management. The smooth progress of the project, with no delays or major issues, highlights the effectiveness of the development approach and the stability of the verification workflow. Overall, the completed work represents a solid and reliable verification environment, along with a comprehensive demonstration of how simulation can be used to validate digital designs efficiently and accurately.