Taking on the task of verifying the asynchronous FIFO using a UVM testbench was a rewarding and deeply educational experience. It wasn’t just about coding up a testbench that compiles—it was about crafting a verification strategy that actually pushes the design to its limits, captures meaningful coverage, and builds confidence in the functionality under a variety of realistic and edge-case scenarios.

From the very beginning, I structured the testbench using all the key elements of a UVM environment. I created a well-defined sequence item, randomized with constraints to produce a broad range of inputs. I developed multiple sequences that simulated different traffic patterns and timing behaviors. These were driven into the DUT by a UVM driver, while a monitor observed and reported activity. The scoreboard handled checking, and the environment integrated it all. I tied it together with a test class and provided a run.do script to automate simulation and coverage reporting.

Running the simulation and seeing data flow correctly felt great, but I was most excited to review the coverage results—because that’s where the testbench proves its value.

Code coverage came back with impressive numbers. The core FIFO module achieved 100% coverage for statements, branches, and conditions. Even toggle coverage was nearly perfect, coming in at 99.5%. The interface and driver logic also performed well, with most signals toggled as expected. However, as I inspected the functional coverage report more closely, I noticed gaps that stood out—gaps that meant I couldn’t claim 100% confidence in the behavior of the design.

This is where things got introspective.

The reason I didn’t achieve full functional coverage became clear as I retraced my steps. First, I realized that my coverage model was too shallow in some areas. Sure, I covered basics like signal activations—writes, reads, flags like full and empty—but I missed key interactions. I didn’t set up cross coverage bins to capture complex combinations, like trying to write when the FIFO is full or reading from it when it’s empty. These are exactly the kinds of conditions where bugs often hide, and my testbench just didn’t trigger them.

Then came the stimulus generation. My sequences were randomized, yes—but they mostly generated valid, legal input patterns. That sounds good on the surface, but in verification, sometimes we need to throw the weird, unexpected, and even illegal inputs at the design. My testbench didn’t simulate resets during transactions, nor did it exercise high-pressure bursts that might push the FIFO into corner behavior. So naturally, the coverage report reflected that: some bins went unhit simply because those situations never occurred.

Another factor was the nature of the FIFO itself—it's asynchronous, with different clocks for read and write domains. This makes it especially tricky to trigger specific behaviors at the right moments unless explicitly targeted. For example, triggering a write just one cycle before full, or a read exactly when the FIFO becomes empty—those aren’t things that happen by accident. I didn’t build targeted sequences to force those conditions. As a result, some of the most meaningful corner-case scenarios stayed untouched.

And then there was the scoreboard. While it did its job for standard scenarios, I now recognize that it wasn’t robust enough to check certain concurrent conditions or invalid state transitions. For example, simultaneous assertions of wr\_en and full weren’t handled explicitly. Without directed tests and stronger checking, some logic branches remained idle—and some coverage bins never lit up.

Looking back, I understand now that 100% functional coverage isn’t just a metric—it’s a mindset. It requires thinking beyond the usual, imagining the unexpected, and challenging the design with scenarios that stretch its boundaries. I fell a bit short this time, but I know why, and I know what to fix.

The next steps are clear: I need to refine the coverage model to include more meaningful cross coverage. I’ll introduce directed sequences that intentionally create stress conditions. I’ll simulate reset behaviors during active operations and build more thorough checks in the scoreboard. With those improvements, I’m confident I can close the final coverage gaps.

This journey has reminded me that verification is where we battle uncertainty—and each simulation brings us one step closer to certainty. While I didn’t achieve 100% functional coverage this time, I came very close, and I gained the clarity and insight needed to get there.