Politecnico di Torino

Technical Report Preparatory Workshop STEP II **"SoC Verification & Strategies"**

Federico Fruttero

December 15, 2023

Contents

Chapter 1

P4 Adder

1.1 Introduction

In the quest to ensure the reliability and functionality of complex hardware designs, robust verification methodologies are crucial. The Universal Verification Methodology (UVM) has emerged as a standard framework that empowers engineers to construct efficient and organized testbenches. This section goes into the UVM testbench designed for the Pentium 4 adder module.

Highlighting the advantages of UVM over traditional SystemVerilog-only approaches, this section shows on how UVM's components and structured interactions contribute to a streamlined verification environment. We'll explore how UVM components, the factory mechanism, and interactions elevate the Pentium 4 adder verification process.

Subsequent sections will go deeper into more complex technical aspects of the UVM for a Register File with Windowing.

1.2 Summary of the testbench components

This summary provides an overview of the key files that constitute the Pentium 4 testbench setup, ullustrating the utilization of UVM methodologies to streamline the verification process.

p4_if.sv - This file defines the UVM interface for the Pentium 4 adder module. It encapsulates the signals involved in the communication between the testbench and the VHDL design, providing a clear abstraction for stimulus generation and response analysis.

driver.svh - The driver class, an integral component of the UVM testbench, generates transactions and drives them into the design. It interfaces with the UVM environment and the DUT, ensuring efficient and synchronized communication.

scoreboard.svh - The scoreboard class verifies the correctness of the DUT's outputs by comparing them with expected results. It plays a pivotal role in ensuring the accuracy of the verification process.

printer.svh - The printer class generates informative messages during simulation, aiding in debugging and understanding the simulation progress. This component is especially useful for verbose tests.

p4_cov.svh - This file implements functional coverage collection for the P4 adder module. It defines coverpoints and crosses to track the exercised scenarios during simulation.

tester_env.svh - The tester environment encapsulates the UVM components, creating an organized and structured testbench architecture. It promotes modularization and easy integration of various verification components.

verbose_test.svh - This class represents a UVM test using a verbose approach, where detailed logs and information about the simulation are generated. It showcases the power of UVM in providing comprehensive insights during testing.

quiet_test.svh - The quiet test class demonstrates a UVM test with minimal logs, suitable for efficiency-focused simulations. It illustrates the versatility of UVM in adapting to different simulation needs.

p4_pkg.sv - The package file includes import statements, global variables, and component inclusion macros. It provides a centralized place for defining and managing resources shared across the testbench.

top.sv - The top module instantiates the Pentium 4 interface, wraps the design, and orchestrates the test execution. It highlights the modularization and configurability of the verification environment.

run.do - The simulation script for **QuestaSim** compiles and runs the testbench, showcasing the integration of the testbench components and the execution of UVM-based tests.

This collection of files forms a comprehensive UVM-based testbench setup for the Pentium 4 adder module. It employs UVM methodologies to automate various verification tasks, from generating stimuli to analyzing coverage. Through this systematic approach, the verification process becomes structured, efficient, and adaptable to different testing scenarios.

1.3 P4 TEST

In the case of the P4, different from the register file, the testing is performed in the run_phase in the DRIVER:

```
// Run Phase: Generate stimulus for the DUT
    task run_phase(uvm_phase phase);
        int nloops = 20000; // Number of iterations

        phase.raise_objection(this); // Indicate that the agent is not
    finished yet

        repeat (nloops) begin
            @(posedge i.clk); // Wait for the rising edge of the clock

            i.CIN = $random; // Assign random value to carry input

            // Generate random values for Aif
            if ($urandom_range(0, 9) < 5) // Higher probability for values
    closer to 0
                i.Aif = $urandom_range(32'h00000000, 32'h000000ff);
            else
                i.Aif = $urandom_range(32'h7FFFFFFF, 32'hffffffff);

            // Generate random values for Bif
            if ($urandom_range(0, 9) < 5) // Higher probability for values
    closer to 0
                i.Bif = $urandom_range(32'h00000000, 32'h000000ff);
            else
```

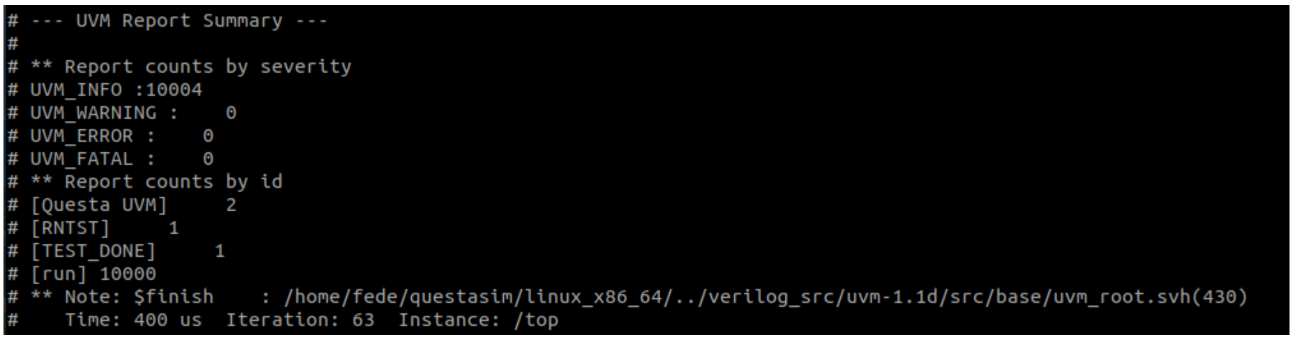
```
                i.Bif = $urandom_range(32'h7FFFFFFF, 32'hffffffff);
        end

        phase.drop_objection(this); // Indicate that the agent has finished
    endtask : run_phase
```
Listing 1.1: run_phase inside driver.svh

We can see how the testing is simply a repeat loop for which number of iterations is defined here as 20000. Inside the loop make use of the randomize() to obtain randomly generated values for the inputs, we can also see that i opted to set a higher probability to obtain values closer to zero.

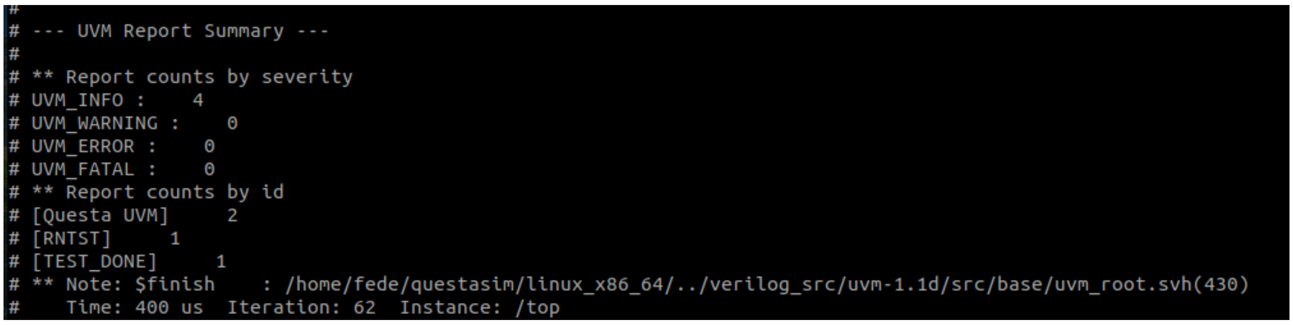
1.4 Simulation

Tu run the simulation, we must run the **simulateP4.sh** file(which its only line inside is vsim -c -do run.do). If we take a look inside the run.do, we can see that we are first running a verbose test followed by a quiet test. If we see the console we can check that 0 errors are detected:



```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :10004
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [run] 10000
# ** Note: $finish    : /home/fede/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 400 us  Iteration: 63  Instance: /top
```

Figure 1.1: Output of Verbose Test



```
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    4
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# ** Note: $finish    : /home/fede/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 400 us  Iteration: 62  Instance: /top
```

Figure 1.2: Output of Quiet Test

1.5 Coverage

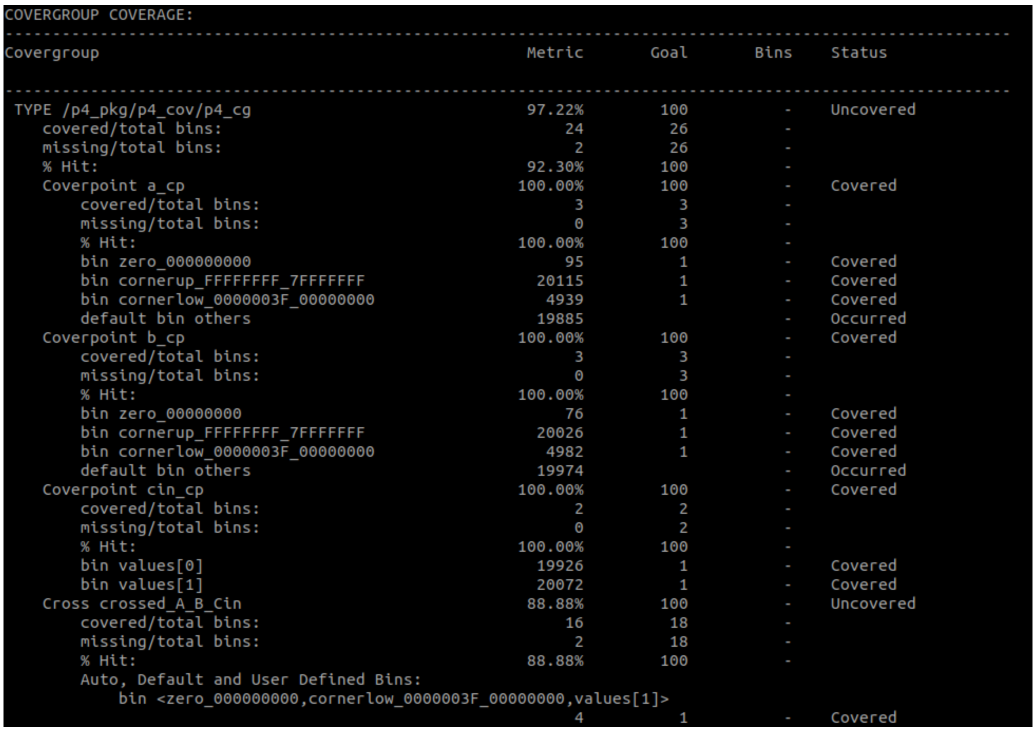
The effectiveness of any verification process lies not only in the thoroughness of tests but also in the extent to which different scenarios are explored. Functional coverage is a critical metric in ensuring that the design under test (DUT) is subjected to a comprehensive range of stimuli. In the Pentium 4 adder testbench, achieving this high level of functional coverage is made possible through the implementation of the **p4_cov class.** This section highlights the purpose and functionality of the coverage class, showcasing its contribution to the verification process. The primary objective of the p4_cov class is to capture a wide spectrum of input operand values and the carry input (CIN) signals that the Pentium 4 adder module is subjected to during simulation. By comprehensively evaluating these operands and signals, the coverage class ensures that various combinations and scenarios are exercised, contributing to the overall coverage goal.

1.5.1 Coverpoints and Crosses

Within the covergroup, two coverpoints—a_cp and b_cp—are defined to track the input operands A and B, respectively. These coverpoints have bins that capture specific ranges of operand values, including zeroes, upper boundaries, and various other ranges. A third coverpoint—cin_cp—captures the values of the carry input (CIN) signal, registering both possible binary states (0 and 1). Additionally, a cross called crossed_A_B_Cin is established to capture interactions between the coverpoints. This cross helps identify scenarios where specific operand values and CIN signals converge, enabling better understanding of coverage interactions. During the run phase, the covergroup is sampled in a continuous loop, triggered by the falling edge of the clock signal. This constant monitoring and sampling contribute to the accumulation of functional coverage data.

1.5.2 Results

Both the results of the coverage analysis of the performed tests are merged in the file **P4_ADDER.ucdb** and then printed through the console. A fragment of the coverage output obtained is shown in the following image:



```
COVERGROUP COVERAGE:
-----------------------------------------------------------------------------------------------------------
Covergroup                                              Metric       Goal       Bins    Status

-----------------------------------------------------------------------------------------------------------
 TYPE /p4_pkg/p4_cov/p4_cg                              97.22%        100          -    Uncovered
    covered/total bins:                                     24         26          -
    missing/total bins:                                      2         26          -
    % Hit:                                              92.30%        100          -
    Coverpoint a_cp                                    100.00%        100          -    Covered
        covered/total bins:                                  3          3          -
        missing/total bins:                                  0          3          -
        % Hit:                                         100.00%        100          -
        bin zero_000000000                                  95          1          -    Covered
        bin cornerup_FFFFFFFF_7FFFFFFF                   20115          1          -    Covered
        bin cornerlow_0000003F_00000000                   4939          1          -    Covered
        default bin others                               19885                     -    Occurred
    Coverpoint b_cp                                    100.00%        100          -    Covered
        covered/total bins:                                  3          3          -
        missing/total bins:                                  0          3          -
        % Hit:                                         100.00%        100          -
        bin zero_00000000                                   76          1          -    Covered
        bin cornerup_FFFFFFFF_7FFFFFFF                   20026          1          -    Covered
        bin cornerlow_0000003F_00000000                   4982          1          -    Covered
        default bin others                               19974                     -    Occurred
    Coverpoint cin_cp                                  100.00%        100          -    Covered
        covered/total bins:                                  2          2          -
        missing/total bins:                                  0          2          -
        % Hit:                                         100.00%        100          -
        bin values[0]                                    19926          1          -    Covered
        bin values[1]                                    20072          1          -    Covered
    Cross crossed_A_B_Cin                               88.88%        100          -    Uncovered
        covered/total bins:                                 16         18          -
        missing/total bins:                                  2         18          -
        % Hit:                                          88.88%        100          -
        Auto, Default and User Defined Bins:
            bin <zero_000000000,cornerlow_0000003F_00000000,values[1]>
                                                             4          1          -    Covered
```

Figure 1.3: Coverage Output P4_ADDER

The p4_cg covergroup demonstrates substantial verification progress, achieving an overall coverage of 97.22%. This impressive figure showcases the thoroughness of the testing process.

Then three coverpoints, namely a_cp, b_cp, and cin_cp, exhibit full coverage with percentages of 100%. This indicates that all possible scenarios involving input operands A, B, and the carry input (CIN) have been explored.

Cross Coverage: The cross crossed_A_B_Cin has a coverage rate of 88.88%. While it signifies substantial coverage of interaction scenarios between A, B, and CIN, two bins remain unexplored. These untested interactions represent opportunities for enhancing verification coverage.

1.5.3 Coverage Conclusion

To conclude, the coverage analysis output provides a comprehensive insight into the verification status of the Pentium 4 adder. It underscores areas of coverage success, such as achieving full coverage for individual coverpoints—namely A, B, and CIN (with 100% coverage). Simultaneously, it identifies opportunities for further investigation, as observed through the relatively small number of unexplored bins within the cross coverage.

Chapter 2

Register File with Windowing

2.1 Introduction & Obstacles Encountered

In this second project of this step of the workshop, I embarked on a more intricate journey - the creation of a Register File Windowing testbench. This particular attempt proved to be the most challenging and time-intensive of all. As I delved into the complex working of the Register File and the associated windowing mechanisms, I found myself diving deeper into the core concepts of analysis FIFOs, TLM FIFOs, and the seamless interaction through ports like i mentioned at the beggining of the report.

At points, I faced confusion and occasional frustration. The complexity of analysis FIFOs aggregating data for comprehensive analysis, and TLM FIFOs facilitating seamless data exchange between components, presented intricate hurdles. Meanwhile, shaping the predictor into a functional model proved to be an equally formidable process that consumed considerable time.

Nonetheless, persisting through those intense periods bore fruitful results in terms of good insights and skills development. This phase was pivotal in grasping the point of UVM's efficacy. As I meticulously and slowly assembled the Register File Windowing testbench, I succeeded in harmonizing analysis FIFOs, TLM FIFOs, ports, agents,components,sequencers, among others, into a coherent ensemble.

Indeed, my approach may have leaned towards complexity, but the sense of accomplishment it brought was much like cracking a complex puzzle. Through the challenges, this journey taught me resilience and determination.

The Register File Windowing testbench I designed stands as proof of my dedication, learning important skills, and the joy of unraveling complex challenges to build a strong and, in my point of view, effective solution.

2.2 Summary of the testbench components

rf_if: This file defines a UVM interface for connecting to the Register File (RF). The interface includes input and output signals for communication with the RF module.

rf_pkg: This package includes all the necessary components, transactions, and configurations to build and run the RF testbench. It includes enums for operation types, a global virtual interface to the P4_if, and references to the UVM components.

generic_pkg: This package defines the local parameters of the DUT, which are, NBITS, NREGISTERS, F(Number of windows), N(Numbers of registers in each block) and M(number of global registers) that are used by almost all the modules. **rf_data:** his file defines a UVM transaction class for the Register File (RF) data. It includes data fields for address, data, and port. The class provides functions for string conversion, deep copying, comparison, and loading data into a transaction.

rf_req: This file defines a UVM transaction class for the Register File (RF) request. It extends the rf_data transaction class by adding an additional operation field. The class provides functions for string conversion, deep copying, comparison, and loading data into a transaction.

interface_base: If we go into the monitor and the driver, both of them need to use an interface to talk to the memory. Rather than each of them having their own build_phase, we are going to create a common build phase by means of the interface_base file.

responder: The responder generates read response transactions for the memory reads and sends them to the monitor. From there, the response transactions flow through the predictor and finally to the comparator, where they are compared with the predicted responses. This entire flow helps in verifying whether the DUT's behavior matches the expected behavior.

driver: This file defines a UVM driver class for sending transactions to the memory interface (memory_if) of the DUT. It takes transaction items from the sequence, translates them into appropriate signal-level actions, and sends these actions to the DUT through the memory interface signals. Responses from the DUT are collected, and the appropriate transaction is constructed and sent back to the sequence.

monitor: This file defines a UVM monitor class for observing the signals on the memory interface (memory_if) of the DUT. It watches for changes in these signals, such as address updates, read/write operations, and data updates. When a change occurs, the monitor constructs corresponding transactions (requests and responses) based on the observed signals. If it detects a valid read or write operation, it clones the transaction and sends it through analysis ports to be captured by downstream components like the predictor and comparator. This separation of signal-level monitoring and transaction-level analysis helps maintain the modularity and scalability of the testbench architecture.

predictor: This file defines a UVM agent class for predicting memory operations in the Register File (RF) simulation. The predictor processes incoming requests and simulates memory operations based on the request types. It maintains a simulated register file and memory model and generates responses for read requests, sending them to the comparator for validation.

comparator: This file defines a UVM agent class for comparing actual and predicted memory operations in the Register File (RF) simulation. The comparator receives actual responses from the DUT and predicted responses from the predictor. It compares these responses and reports any discrepancies.

printer: This file defines a UVM agent class for printing out transaction data from an analysis FIFO. The printer agent receives transactions from the FIFO and reports their contents.

tester_env: This file defines the environment for the testbench. It includes the setup and connections of various components like the test sequence, driver, monitor, predictor, comparator, and printers.

test_seq: This file defines the test sequence that will be executed.

coverage: This class defines the coverage agent for the testbench. It includes covergroups to capture different aspects of the design's behavior.

verbose_test: This file defines the verbose test class for the testbench.

top: This is the top module for the RF testbench. It instantiates the RF interface and wraps it with the RF wrapper. The test is started using the run_test() function.

run.do: This **QuestaSim** script compiles and runs the RF testbench. It generates coverage reports and saves them in UCDB format.

2.3 Design Decisions

The architecture of the testbench involves several critical design decisions to ensure seamless communication between its components. The **driver** plays a pivotal role in transmitting requests to the Device Under Test (DUT) by toggling specific signals that define operations, addresses, and data. It adeptly orchestrates the flow of transactions by sending out requests and efficiently managing the wait for responses. On the other end, the **monitor** acts as a vigilant observer, meticulously monitoring signals related to address updates, read/write actions, and data modifications. When any change occurs, the monitor takes action, generating appropriate transactions – both requests and responses.

At the heart of this orchestration, the **responder** takes on the role of generating read response transactions for read operations. These responses are then seamlessly sent to the monitor, completing the feedback loop. The magic unfolds as the monitor captures these transactions and shares them through analysis ports. This data then flows downstream, where components like the **predictor** and **comparator** come into play. The comparator carries out a critical role in this process, meticulously comparing the expected values generated by the predictor's model with the actual responses from the register file. These complex design decisions culminate in a synchronized dance of components that ensure the testbench's accuracy and effectiveness.

2.4 Register File Sequence & Test

This section takes a deep dive into the intricate workings of the test sequence, a pivotal component in comprehensively evaluating the performance of the register file windowing system. This sequence encapsulates a series of operations that simulate real-world scenarios encountered by the register file.

Before embarking on an exploration of the sequence's inner workings, it's essential to highlight three fundamental tasks that serve as the cornerstones of its functionality:

```
// Task to perform a call, return, or reset request task
   task call_return_reset(rf_op operation);
        $display("");
        // Create the request object for the specified operation
        req = new();
        req.op = operation;
        // Start and finish the transaction
        start_item(req);
        finish_item(req);
        // Get the response
        get_response(rsp);
        $display("");
```

```
    endtask
```
Listing 2.1: Return/Reset/Call task

This task has the responsibility of transmitting a request to the Device Under Test (DUT) to execute the specified operation, as indicated by the argument it receives. This task proves to be versatile, serving both as a way for calling and returning from subroutines, as well as for executing the reset operation. The following task used is:

```
// Task to perform read and write operations in a window
    task read_write_window(rf_op operation);
        i = 0;
        repeat(14) begin // Read all positions in the window
            req = new();
            assert(req.randomize());
            assert(req.randomize());
            // Randomize the data (for simplicity, I only want to randomize
 the data)
            req.op   = operation;
            req.addr = i;
            req.port = 0;
            // Start and finish the transaction
            start_item(req);
            `uvm_info("test_seq", {"Sending transaction ", req.convert2string
 ()}, UVM_HIGH);
            finish_item(req);
            // Get the response
            get_response(rsp);
            // Print the response if the operation was a read
            if (req.op == read)
                `uvm_info("test_seq", {"Got back: ", rsp.convert2string()},
 UVM_HIGH);
            i++;
        end
        i = 0;
        #1; // Allow the initial read to complete before printing the
 following uvm_info
        $display("");
    endtask
```
Listing 2.2: Read Write request task

This task serves to execute both **READ** and **WRITE** operations across all registers within the window, including the GLOBALS (a total of 5 registers). A notable aspect of this process is the utilization of the randomize() function to derive randomized 64-bit values that are written into the registers. Finally, the last task is in charge or performing a number of random operations, the number is passed as an argument, this is used in the last part of the sequence:

```
// Task to perform a given number of random operations without verbosity
    task random_operations(int number);
        // Turn off verbosity for the test's duration
        uvm_top.set_report_verbosity_level_hier(UVM_NONE);
        // Perform random operations
        repeat(number) begin
            req = new();
            // Randomize the request
            assert(req.randomize() with {
                addr <= 'hd; // Max value for the address
                // Distribution for the 'op' field
                op dist {
                    read   := 45,
                    write  := 45,
                    [call:ret]  :/ 5,
                    reset  := 1
                };
            });
            // Start and finish the transaction
            start_item(req);
            `uvm_info("test_seq", {"Sending transaction ", req.convert2string()}, UVM_HIGH);
            finish_item(req);
            // Get the response
            get_response(rsp);
            // Print the response if the operation was a read
            if (req.op == read)
                `uvm_info("test_seq", {"Got back: ", rsp.convert2string()}, UVM_HIGH);
        end
        // Restore verbosity level
        uvm_top.set_report_verbosity_level_hier(UVM_MEDIUM);
    endtask
```

Listing 2.3: Random operation request task

Now, let's dive into the initial steps of the test. We begin by initiating a "**RESET**" to provide the system with a clean slate. Following this, we proceed with a sequence where various registers receive a series of random values. Subsequently, we move through a sequence of subroutine calls and register writes, ensuring a comprehensive assessment of the system's functionality. It's like a methodical check to confirm that all the essential actions are in harmony.

```
    // Print information about initializing the registers in all windows
    `uvm_info("run", "We are about to initialize the registers in all the windows", UVM_MEDIUM);

    read_write_window(write);            // Write all registers in Main window
    call_return_reset(call);             //FIRST SUBROUTINE CALL
    read_write_window(write);            // Write all registers in SUB1
    call_return_reset(call);             //SECOND SUBROUTINE CALL
    read_write_window(write);            // Write all registers in SUB2
    call_return_reset(call);             //THIRD SUBROUTINE CALL
    read_write_window(write);            // Write all registers in SUB3
    read_write_window(read);             //Read all registers in SUB3
    call_return_reset(ret);              //RETURN TO SUB2
    read_write_window(read);             //Read all registers in SUB2
    call_return_reset(ret);              //RETURN TO SUB1
    read_write_window(read);             //Read all registers in SUB1
    call_return_reset(ret);              //RETURN TO MAIN PROGRAM
    read_write_window(read);             //Read all registers in MAIN PROGRAM
     `uvm_info("run", "Now we are going to call our first subroutine and read all the registers in the window", UVM_MEDIUM);
    call_return_reset(call);
    read_write_window(read);
    `uvm_info("run", "Returning from the subroutine, SIGRETURN is set", UVM_MEDIUM);
    call_return_reset(ret);
```

```
        `uvm_info("run", "Now we are going to read all the registers in the
    current window", UVM_MEDIUM);
        read_write_window(read);
```

Listing 2.4: First Sequence

Reaching the end of our sequence file, we have a very important sequence, dedicated to test the **SPILL** & **FILL** operations. Once we call the 4th subroutine, because of the fact of not having any free windows, a SPILL will be performed of the window0 (main program) into the memory to make it free for the new subroutine to perform read and write on its registers. The subsequent phase involves the step-by-step return from the subroutines. Once we reach Subroutine 1 and we make a return, a FILL operation comes into play, retrieving the preserved register values from memory and seamlessly reinstating them to their state prior to the invocation of the 4th subroutine.

```
`uvm_info("run", "Now the moment of truth, we are going to spill", UVM_HIGH)
    ;
        $display("");
        `uvm_info("run", "Subroutine 1", UVM_MEDIUM);
        call_return_reset(call);
        `uvm_info("run", "Subroutine 2", UVM_MEDIUM);
        call_return_reset(call);
        `uvm_info("run", "Subroutine 3", UVM_MEDIUM);
        call_return_reset(call);
        // Print information about Subroutine 4 (SPILL):
        `uvm_info("run", "Subroutine 4", UVM_MEDIUM); // SPILL
        call_return_reset(call);
        read_write_window(write);
        read_write_window(read);
        call_return_reset(ret); // Return to Sub3
        read_write_window(read);
        call_return_reset(ret); // Return to Sub2
        read_write_window(read);
        call_return_reset(ret); // Return to Sub1
        read_write_window(read);
        // Call a subroutine for return operation (Return to main, a FILL
    will be performed)
        call_return_reset(ret);
        `uvm_info("run", "Returning to main program", UVM_MEDIUM)

        // Perform some NOP operations to give time for filling
        repeat(10) begin
            req = new();
            req.op = nop;
            start_item(req);
            finish_item(req);
            get_response(rsp);
        end
        // Read FILLED registers from memory
        read_write_window(read)
```

Listing 2.5: SPILL and FILL test sequence

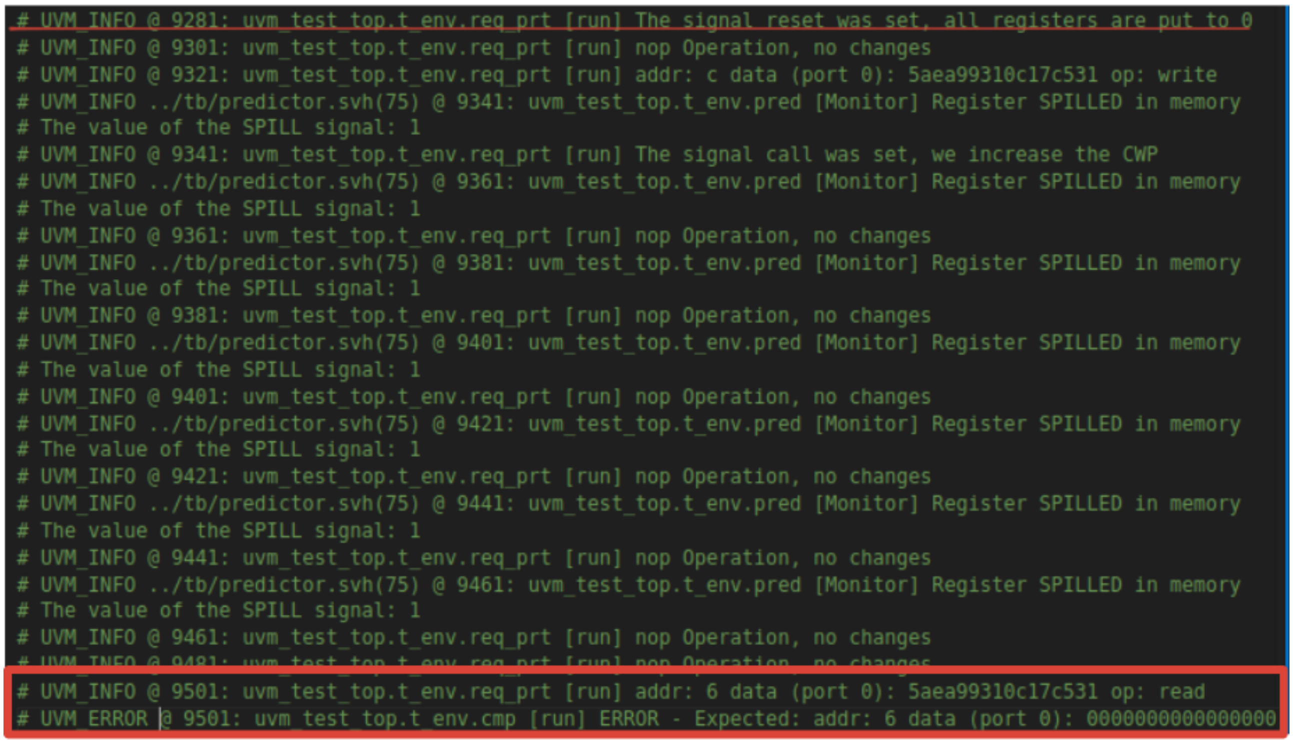
To conclude, we perform 10000 random operations with the use of another task **random_operations (int number)**, this way we can test in a more extensive way the correct behavior of the DUT and check for errors. An interesting strategy I employed was to manage verbosity effectively. With the potential of generating a significant 10,000 transactions, I devised an intelligent solution. By tuning the verbosity level to **"UVM_NONE,"** I adeptly avoided inundating outputs.

```
    // Print information about performing random operations without
verbosity
    `uvm_info("run", "NOW I WANT TO PERFORM 10000 RANDOM OPERATIONS
WITHOUT VERBOSITY::::::::::", UVM_MEDIUM);
    $display("");
    // Perform random operations without verbosity
    call_return_reset(reset);
    random_operations(10000);
```

Listing 2.6: Final 10000 random operations to expand Coverage (No verbosity)

2.5 Results

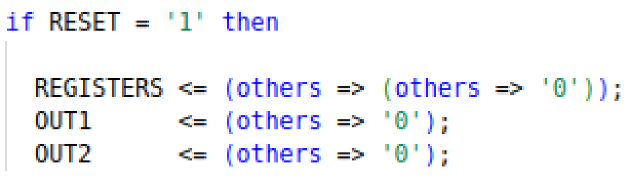
After simulation the previously commented sequence, i found a bug in the DUT:



```
# UVM_INFO @ 9281: uvm_test_top.t_env.req_prt [run] The signal reset was set, all registers are put to 0
# UVM_INFO @ 9301: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO @ 9321: uvm_test_top.t_env.req_prt [run] addr: c data (port 0): 5aea99310c17c531 op: write
# UVM_INFO ../tb/predictor.svh(75) @ 9341: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9341: uvm_test_top.t_env.req_prt [run] The signal call was set, we increase the CWP
# UVM_INFO ../tb/predictor.svh(75) @ 9361: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9361: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO ../tb/predictor.svh(75) @ 9381: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9381: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO ../tb/predictor.svh(75) @ 9401: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9401: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO ../tb/predictor.svh(75) @ 9421: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9421: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO ../tb/predictor.svh(75) @ 9441: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9441: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO ../tb/predictor.svh(75) @ 9461: uvm_test_top.t_env.pred [Monitor] Register SPILLED in memory
# The value of the SPILL signal: 1
# UVM_INFO @ 9461: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO @ 9481: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO @ 9501: uvm_test_top.t_env.req_prt [run] addr: 6 data (port 0): 5aea99310c17c531 op: read
# UVM_ERROR @ 9501: uvm_test_top.t_env.cmp [run] ERROR - Expected: addr: 6 data (port 0): 0000000000000000
```

Figure 2.1: Error due to a reset malfunction

We can see that in **UVM_INFO @ 9281** the signal reset is set, but then, when the first subroutine is called, a spill is performed and we get an error at 9561. We have found a fault in the DUT. If we go to the source file of the register file, in the part belonging to the reset, we find this:

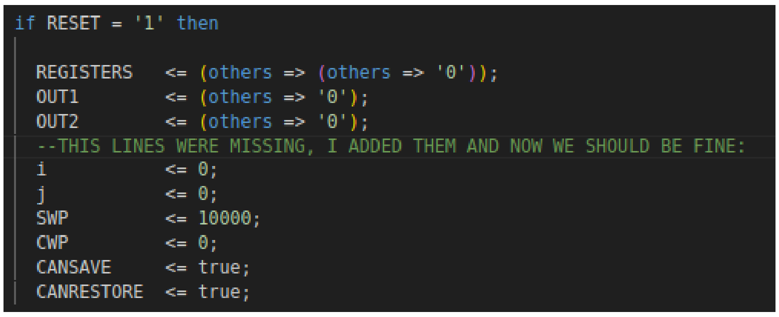


```
if RESET = '1' then

  REGISTERS <= (others => (others => '0'));
  OUT1      <= (others => '0');
  OUT2      <= (others => '0');
```

Figure 2.2: Reset VHDL description

As we may see, we are missing something here, we are not resetting the values of the Current and Save Window Pointer (CWP and SWP), together with variables used during the simulation like "i","j","CANSAVE","CANRESTORE". After changing the source code, we finally get:

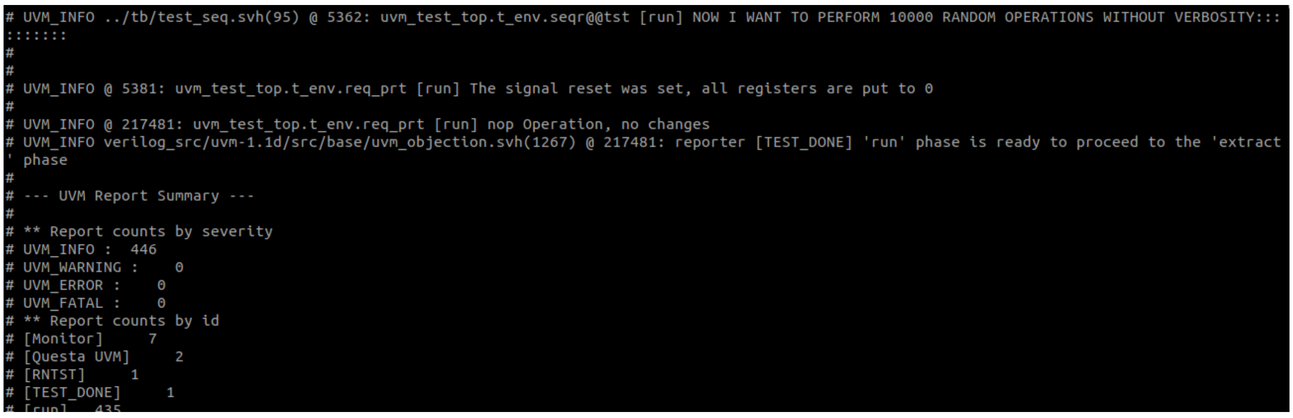


```
if RESET = '1' then

  REGISTERS   <= (others => (others => '0'));
  OUT1        <= (others => '0');
  OUT2        <= (others => '0');
  --THIS LINES WERE MISSING, I ADDED THEM AND NOW WE SHOULD BE FINE:
  i           <= 0;
  j           <= 0;
  SWP         <= 10000;
  CWP         <= 0;
  CANSAVE     <= true;
  CANRESTORE  <= true;
```

Figure 2.3: NEW Reset VHDL description

Now, if we run again the simulation the error is gone and there are no $UVM_ERRORS:



```
# UVM_INFO ../tb/test_seq.svh(95) @ 5362: uvm_test_top.t_env.seqr@@tst [run] NOW I WANT TO PERFORM 10000 RANDOM OPERATIONS WITHOUT VERBOSITY:::
:::::::
#
#
# UVM_INFO @ 5381: uvm_test_top.t_env.req_prt [run] The signal reset was set, all registers are put to 0
#
# UVM_INFO @ 217481: uvm_test_top.t_env.req_prt [run] nop Operation, no changes
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 217481: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract
' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  446
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Monitor]     7
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [run]   435
```

Figure 2.4: New Test output

2.6 Coverage

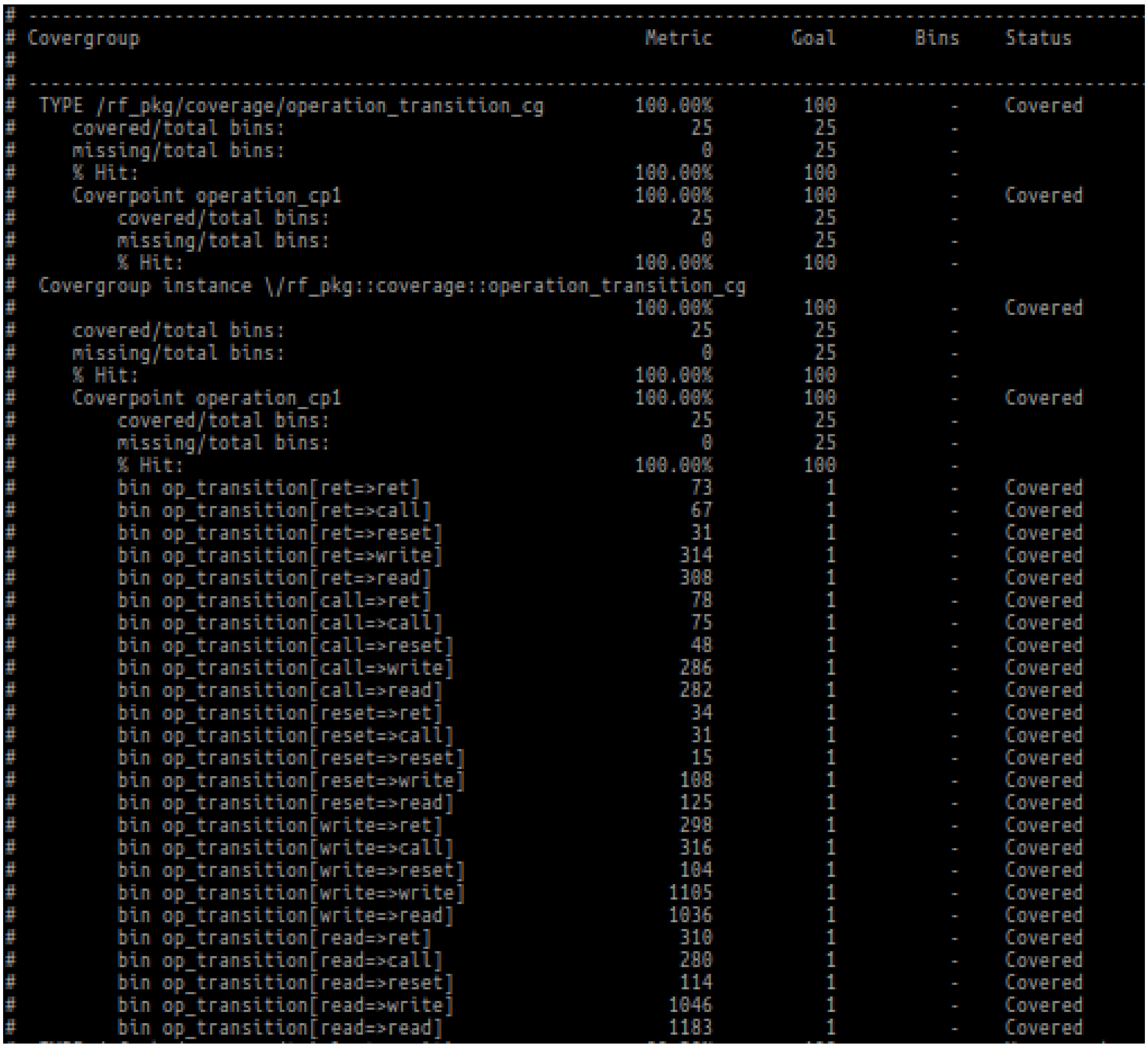
The coverage class is designed to comprehensively capture key behaviors of the register file module within the testbench. In the context of this coverage analysis, a notable challenge arose repect to the coverage of operation transitions, particularly involving consecutive call, ret, and reset operations. This challenge was due to to the insertions of NOP operations due to delays in the test sequence between for example, two consecutive CALL operations. This NOP operation was not permiting that the transitions CALL –> CALL, CALL –> RET, RET –> RESET, among others, were covered.

To address this issue, a clever solution was implemented. The coverage agent now avoids sampling coverage data when a NOP operation is encountered. By excluding nop operations from coverage sampling, the impact of delayed nop insertions on operation transitions was mitigated. As a result, accurate and consistent operation transition coverage is now achieved, providing a more faithful representation of the register file's behavior during simulation.

2.6.1 Results

Analyzing the coverage output provided, the simulation reports a comprehensive coverage report consisting of three distinctive covergroups, each considering different aspects of the design's behavior.

The first covergroup, named **operation_transition_cg**, meticulously captures singular operation transitions, exemplified by shifts from call –> reset, call –> read, read –> write, and all other possible transitions (this was the one of the issue mentioned before). The coverage of these transitions stands at an impressive 100%, indicating a comprehensive exploration of these operation dynamics.

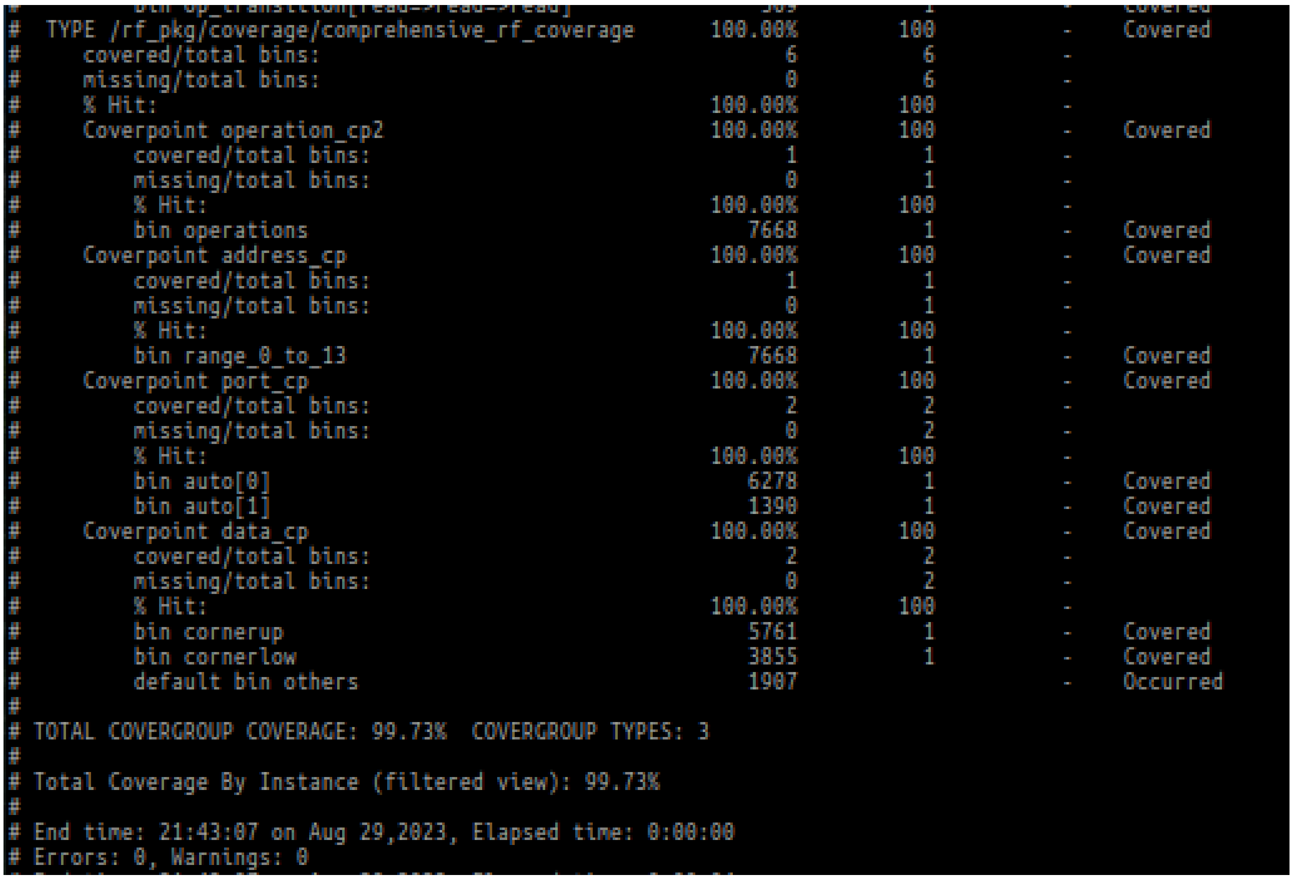


```
# -----------------------------------------------------------------------------------------------
# Covergroup                                          Metric       Goal       Bins    Status
#
# -----------------------------------------------------------------------------------------------
#  TYPE /rf_pkg/coverage/operation_transition_cg        100.00%        100          -    Covered
#     covered/total bins:                                   25          25          -
#     missing/total bins:                                    0          25          -
#     % Hit:                                            100.00%        100          -
#     Coverpoint operation_cp1                          100.00%        100          -    Covered
#         covered/total bins:                               25          25          -
#         missing/total bins:                                0          25          -
#         % Hit:                                        100.00%        100          -
#  Covergroup instance \/rf_pkg::coverage::operation_transition_cg
#                                                       100.00%        100          -    Covered
#     covered/total bins:                                   25          25          -
#     missing/total bins:                                    0          25          -
#     % Hit:                                            100.00%        100          -
#     Coverpoint operation_cp1                          100.00%        100          -    Covered
#         covered/total bins:                               25          25          -
#         missing/total bins:                                0          25          -
#         % Hit:                                        100.00%        100          -
#         bin op_transition[ret=>ret]                       73           1          -    Covered
#         bin op_transition[ret=>call]                      67           1          -    Covered
#         bin op_transition[ret=>reset]                     31           1          -    Covered
#         bin op_transition[ret=>write]                    314           1          -    Covered
#         bin op_transition[ret=>read]                     308           1          -    Covered
#         bin op_transition[call=>ret]                      78           1          -    Covered
#         bin op_transition[call=>call]                     75           1          -    Covered
#         bin op_transition[call=>reset]                    48           1          -    Covered
#         bin op_transition[call=>write]                   286           1          -    Covered
#         bin op_transition[call=>read]                    282           1          -    Covered
#         bin op_transition[reset=>ret]                     34           1          -    Covered
#         bin op_transition[reset=>call]                    31           1          -    Covered
#         bin op_transition[reset=>reset]                   15           1          -    Covered
#         bin op_transition[reset=>write]                  108           1          -    Covered
#         bin op_transition[reset=>read]                   125           1          -    Covered
#         bin op_transition[write=>ret]                    298           1          -    Covered
#         bin op_transition[write=>call]                   316           1          -    Covered
#         bin op_transition[write=>reset]                  104           1          -    Covered
#         bin op_transition[write=>write]                 1105           1          -    Covered
#         bin op_transition[write=>read]                  1036           1          -    Covered
#         bin op_transition[read=>ret]                     310           1          -    Covered
#         bin op_transition[read=>call]                    280           1          -    Covered
#         bin op_transition[read=>reset]                   114           1          -    Covered
#         bin op_transition[read=>write]                  1046           1          -    Covered
#         bin op_transition[read=>read]                   1183           1          -    Covered
```

Figure 2.5: Covergroup Operation Transition

Moving forward, the second covergroup, denoted as **triple_transition_cg**, takes on a more complex role, going into triple transitions. This covergroup exhibits a high coverage rate of 99.20%, leaving only one triple transition uncovered reset $\rightarrow$ reset $\rightarrow$ reset. The exploration of these triple transitions, comprising not the most commonly seen sequences such as ret –> call –> reset and ret –> reset –> write, contributes to a comprehensive understanding of the system behavior. For a matter of space, the picture is not displayed in this case.

Lastly, the **comprehensive_rf_coverage** covergroup achieves full coverage of 100%, covering all the possible values the address, port and operation can take, and covering the two bins of the possible data values.



```
#  TYPE /rf_pkg/coverage/comprehensive_rf_coverage      100.00%        100          -    Covered
#     covered/total bins:                                     6          6          -
#     missing/total bins:                                     0          6          -
#     % Hit:                                            100.00%        100          -
#     Coverpoint operation_cp2                          100.00%        100          -    Covered
#         covered/total bins:                                 1          1          -
#         missing/total bins:                                 0          1          -
#         % Hit:                                        100.00%        100          -
#         bin operations                                   7668          1          -    Covered
#     Coverpoint address_cp                             100.00%        100          -    Covered
#         covered/total bins:                                 1          1          -
#         missing/total bins:                                 0          1          -
#         % Hit:                                        100.00%        100          -
#         bin range_0_to_13                                7668          1          -    Covered
#     Coverpoint port_cp                                100.00%        100          -    Covered
#         covered/total bins:                                 2          2          -
#         missing/total bins:                                 0          2          -
#         % Hit:                                        100.00%        100          -
#         bin auto[0]                                      6278          1          -    Covered
#         bin auto[1]                                      1390          1          -    Covered
#     Coverpoint data_cp                                100.00%        100          -    Covered
#         covered/total bins:                                 2          2          -
#         missing/total bins:                                 0          2          -
#         % Hit:                                        100.00%        100          -
#         bin cornerup                                     5761          1          -    Covered
#         bin cornerlow                                    3855          1          -    Covered
#         default bin others                               1907                     -    Occurred
#
# TOTAL COVERGROUP COVERAGE: 99.73%  COVERGROUP TYPES: 3
#
# Total Coverage By Instance (filtered view): 99.73%
#
# End time: 21:43:07 on Aug 29,2023, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

Figure 2.6: Covergroup comprehensive_rf_coverage

Chapter 3

Conclusion

In conclusion, this report has provided a comprehensive overview of verification methodologies, with a focus on UVM testbenches for the **Pentium 4 adder** and the **Register File** with a Windowing system. While my implementations may not be flawless, they have greatly contributed to my understanding of UVM components like agents, tlm_fifos, analysis fifos, ports, among others.

These hands-on experiences have been valuable learning opportunities, allowing me to delve deep into UVM's complexities. While not perfect, they signify significant progress in making use of UVM's capabilities for hardware verification.

Ultimately, this step of the Workshop highlights, for me, the journey of learning and growth, rather than just the final results. They mark a significant step forward and set the stage for continuous refinement and exploration within the empire of UVM-powered verification.