Hardware Verification Engineer position @SiemensEDA

Computer Storage Project Report

Mohamed Hussein Mohamed Abomakhlouf

Contact Details:

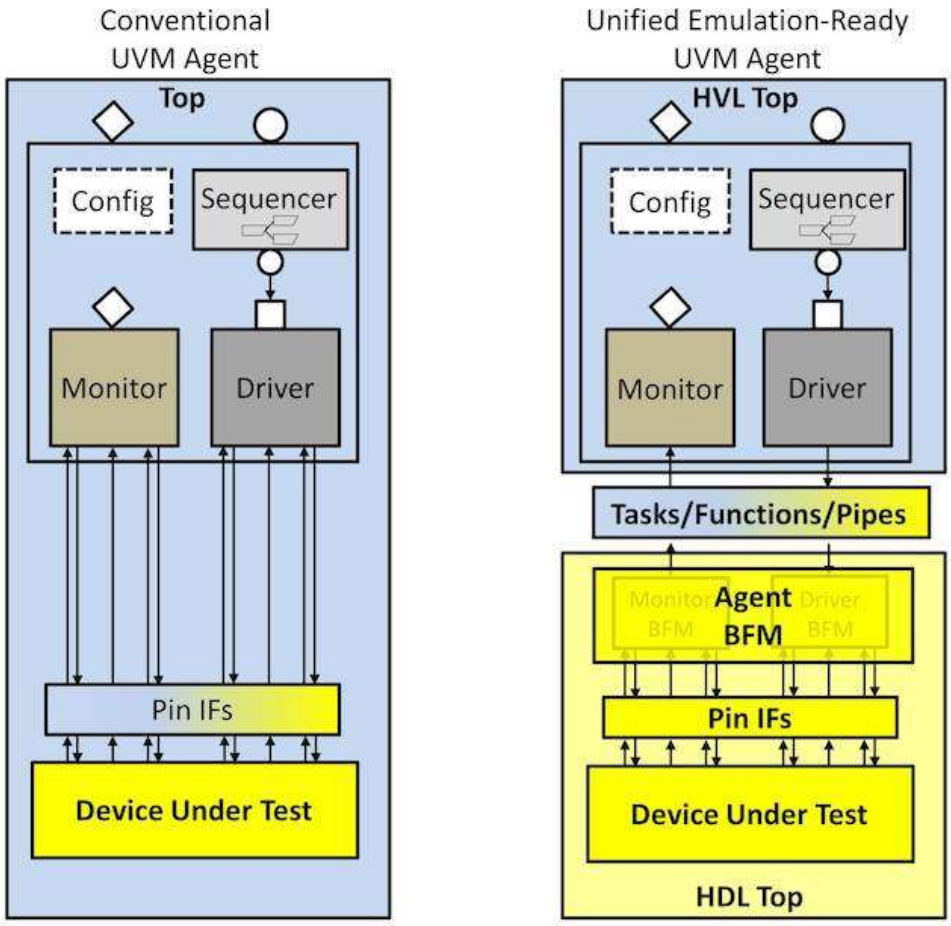
Email: Mohamed2001hussein@gmail.com

Phone: +201062367332

Contents

Theoretical Questions

Draw basic UVM Test bench Architecture diagram supporting Emulation



What are the main criteria in developing UVM TB Emulation friendly?

Avoiding Complex Randomization:

- **Emulation Limitation**: Emulators are not optimized for handling complex randomization due to limited CPU resources and focus on hardware acceleration.
- **Recommendation**: Move any complex randomization to the software side or pre-randomize data before emulation. Use simplified or deterministic randomization within the testbench during emulation runs.

Reducing the Use of Dynamic Constructs:

- **Emulation Limitation**: Dynamic constructs such as dynamic memory allocation (new), linked lists, or dynamic queues are not efficiently supported in emulation.
- **Recommendation**: Replace dynamic constructs with static arrays or pre-sized data structures. Keep memory management simple by using statically defined constructs wherever possible.

Minimizing System Verilog Assertions (SVA):

- **Emulation Limitation**: SVAs are generally not supported in hardware emulation due to their complexity and runtime overhead.
- **Recommendation**: Disable or minimize SVAs for emulation purposes. Instead, use assertions in pre-silicon simulations and rely on other verification methods, such as checkers or scoreboards, for emulation.

Simplifying Functional Coverage:

- **Emulation Limitation**: Functional coverage collection can introduce significant overhead, which slows down emulation performance.
- **Recommendation**: Disable functional coverage collection during emulation runs. Use functional coverage primarily in simulation environments, and limit or entirely avoid coverage gathering in the UVM testbench when running in emulation.

Summarize digital design flow

- ***Specification***: Define system requirements and design constraints.
- ***Architectural Design*****:** Create block diagrams and system architecture.
- ***RTL Design:*** Implement the design using HDLs like Verilog or VHDL.

- ***Functional Verification:*** Verify the RTL design against the specification using simulation and testbenches.
- ***Synthesis:*** Convert RTL into a gate-level netlist.
- ***Gate-Level Simulation:*** Simulate the gate-level design to ensure correctness.
- ***Timing Analysis:*** Perform static timing analysis to meet timing constraints.
- ***Physical Design:*** Place and route the design on the chip layout.
- ***Physical Verification:*** Check the layout for manufacturability and correctness.
- ***Power Analysis:*** Analyze the design's power consumption.
- ***DFT:*** Add test features for post-silicon testing.
- ***Fabrication:*** Manufacture the design as silicon chips.
- ***Post-Silicon Validation:*** Validate the chips in real-world scenarios.
- ***Debug and Validation:*** Detect and fix any remaining issues in the silicon.
- ***Production and Deployment:*** Mass-produce and deploy the final chips.

Summarize Functional Verification flow

- ***Test Plan Creation:*** Define test cases, scenarios, and coverage goals based on the design specification.
- ***Testbench Development:*** Build the test environment, including drivers, monitors, checkers, and scoreboards.
- ***Simulation:*** Run simulations of the RTL design with stimulus from the testbench.
- ***Assertions:*** Add assertions to check for correct behavior and corner cases during simulation
- ***Functional Coverage:*** Monitor and analyze the coverage of different design features during verification.
- ***Debugging:*** Identify and fix bugs in the design or testbench.
- ***Regression Testing:*** Continuously test the design with multiple test cases to ensure no new bugs are introduced.
- ***Coverage Closure:*** Ensure all functionality and scenarios have been thoroughly tested before sign-off.

Project Documentation

Project Structure

Main module summary

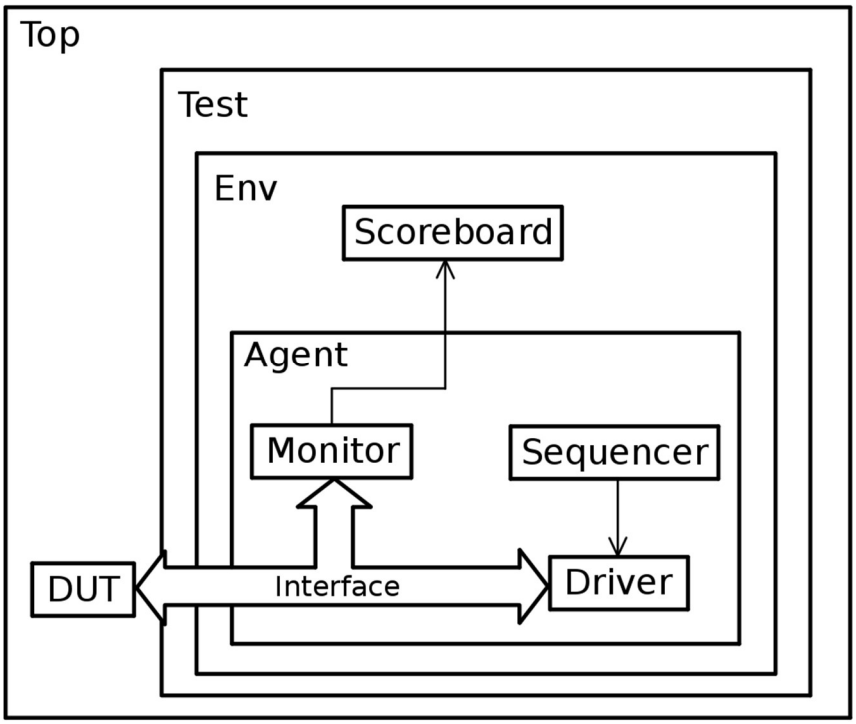
- ***Memory Declaration:*** The module contains a memory array with a width of modif.mem_width and a depth of 2**modif.mem_length.
- ***Local Parameters:*** The operations are defined using local parameters: RD_MEM_CMD (read), WR_MEM_CMD (write), ADD_CMD (addition), and SUB_CMD (subtraction).
- ***Reset Logic:*** On reset, the output modif.DQ_o is cleared, and memory is initialized from the file "mem_init.txt".
- ***Command Execution:*** Based on the command (modif.cmd), the module performs one of four operations:
- ***Read:*** Outputs data from memory[modif.addA] to modif.DQ_o.
- ***Write***: Writes modif.DQ_i to memory[modif.addC].
- ***Add***: Adds values from two addresses (modif.addA and modif.addB) and stores the result in memory[modif.addC].
- ***Subtract***: Subtracts the value at modif.addB from modif.addA and stores the result in memory[modif.addC].

Main module implementation

```
module main (module_if.DUT modif);
    logic [modif.mem_width-1 : 0] memory [2**modif.mem_length-1 : 0];
    localparam  RD_MEM_CMD = 2'b00,
                WR_MEM_CMD = 2'b01,
                ADD_CMD    = 2'b10,
                SUB_CMD    = 2'b11;
    always_ff @(posedge modif.clk or posedge modif.rst) begin //seq logic to
assign the result
        if(modif.rst) begin
            modif.DQ_o <= 0;
            $readmemb("mem_init.txt", memory);
        end else begin
            case (modif.cmd)
                    RD_MEM_CMD: begin
                      modif.DQ_o <= memory[modif.addA];
                    end
                    WR_MEM_CMD: begin
                      memory[modif.addC] <= modif.DQ_i;
                    end
                    ADD_CMD: begin
                      memory[modif.addC] <= memory[modif.addA] +
memory[modif.addB];
                    end
                    SUB_CMD: begin
                      memory[modif.addC] <= memory[modif.addA] -
memory[modif.addB];
                    end
            endcase
                end    end     endmodule : main
```

Main Module testing

UVM Environment structure



The UVM environment written in the code follows this block diagram, due to the length of the code, I will only be including the scoreboard here as it has the main testbench. Also, will be including the sequences used to test here too

UVM sequences

Reset Sequence

```
task body;

            seq_item = storage_seq_item::type_id::create("seq_item");
             start_item(seq_item);
                 seq_item.rst = 1;
                 seq_item.addA = 0;
                 seq_item.addB = 0;
                 seq_item.addC = 0;
                 seq_item.cmd = 0;
                 seq_item.DQ_i = 0;
                 #(CLK_PERIOD)
                 seq_item.rst = 1'b0;
                 #(CLK_PERIOD)
                 seq_item.rst = 1'b1;
             finish_item(seq_item);
         endtask : body
```

Test Sequence

```
task body;
                    repeat(10000) begin
                           seq_item =
storage_seq_item::type_id::create("seq_item");
                           start_item(seq_item);
                                  assert(seq_item.randomize());
                           finish_item(seq_item);
                    end
              endtask : body
```

UVM scoreboard

- ***Randomized Inputs:*** The scoreboard operates on sequence items that are generated by sequences. These sequences are typically randomized in UVM environments, meaning that the addresses (addA, addB, addC) and data (DQ_i) are likely generated randomly, covering different corner cases and possible scenarios.
- ***Functional Testing:*** The scoreboard tests various functional operations:
    - *Read Operation (RD_MEM_CMD):* Ensures the DUT is correctly reading memory.
    - *Write Operation (WR_MEM_CMD):* Tests if the DUT writes data correctly to memory.
    - *Addition and Subtraction (ADD_CMD, SUB_CMD):* Verifies that the DUT performs arithmetic operations on memory contents correctly.
- ***Error and Coverage Reporting:*** The scoreboard tracks the number of successful and failed operations (correct_count and error_count), which helps in verifying if the DUT behaves correctly across all the operations. It also records all memory contents at the end of the simulation for later analysis.

UVM scoreboard phases

- ***run_phase Task:***
  - Runs continuously in the background.
  - Retrieves sequence items from the scoreboard FIFO (sb_fifo).
  - Calls the reference model (ref_model) to simulate expected behavior for the sequence item.
- ***report_phase Function:***
  - Writes the contents of the reference memory (ref_mem) to a file called "memory_dump.txt".
  - Reports the total number of successful and failed reads using UVM info messages.
  - Ensures memory is dumped only if the file is successfully opened.
- ***ref_model Task:***
  - Simulates the expected behavior of the DUT based on the command in the sequence item.
  - Handles the reset condition by reloading the memory from an initialization file (mem_init.txt).
  - Depending on the command (RD_MEM_CMD, WR_MEM_CMD, ADD_CMD, SUB_CMD), the appropriate task (read_mem, write_mem, add, sub) is called to simulate memory operations.
- ***read_mem Task:***
  - Simulates a memory read operation and checks if the DUT's output matches the reference memory data.
  - Increments either the correct_count or error_count based on whether the DUT's result matches the expected value.

- ***write_mem Task:***
    - Simulates writing data to the reference memory at the specified address.
- ***add Task:***
    - Simulates an addition operation, where the values from two memory addresses (addrA and addrB) are added and stored at the specified memory address (addrC).
- ***sub Task:***
    - Simulates a subtraction operation, where the value at addrB is subtracted from the value at addrA and stored at addrC.

UVM Scoreboard implementation

```
package storage_scoreboard_pkg;
    import uvm_pkg::*;
    `include "uvm_macros.svh"
    import storage_seq_item_pkg::*;
    parameter mem_width = 16; //bits
    parameter mem_length = 8; //bits
    localparam  RD_MEM_CMD = 2'b00,
                WR_MEM_CMD = 2'b01,
                ADD_CMD    = 2'b10,
                SUB_CMD    = 2'b11;
    class storage_scoreboard extends uvm_scoreboard;
        `uvm_component_utils(storage_scoreboard);
        uvm_analysis_export #(storage_seq_item) sb_export;
        uvm_tlm_analysis_fifo #(storage_seq_item) sb_fifo;
        storage_seq_item seq_item_sb;
        logic [mem_width-1 : 0] ref_mem [2**mem_length-1 : 0];
        logic [mem_width-1 : 0] ref_res;
        int error_count = 0;
        int correct_count = 0;
        integer file;  // File handler
        virtual module_if modvif;

        function new(string name = "storage_scoreboard", uvm_component parent =
null);
            super.new(name, parent);
        endfunction : new

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            sb_export = new("sb_export", this);
```

```
        sb_fifo = new("sb_fifo", this);
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        sb_export.connect(sb_fifo.analysis_export);
    endfunction : connect_phase

    task run_phase(uvm_phase phase);
        super.run_phase(phase);
        forever begin
            sb_fifo.get(seq_item_sb);
            ref_model(seq_item_sb);
        end
    endtask : run_phase

    function void report_phase(uvm_phase phase);
        super.report_phase(phase);
        file = $fopen("memory_dump.txt", "w");
        // Check if the file is successfully opened
        if (file) begin
            // Write the contents of the memory array to the file
            for (int i = 0; i < 2**mem_length; i = i + 1)
                $fwrite(file, "%0h\n", ref_mem[i]);
        end
        // Close the file
        $fclose(file);
        `uvm_info("report_phase", $sformatf("Total succesful reads: %0d",
correct_count), UVM_MEDIUM);
```

```
            `uvm_info("report_phase", $sformatf("Total failed reads: %0d",
error_count), UVM_MEDIUM);
        endfunction : report_phase

        task ref_model(storage_seq_item seq_item_chk);
            if (seq_item_chk.rst) begin
                $readmemb("mem_init.txt", ref_mem);
            end
            else begin
                case (seq_item_chk.cmd)
                        RD_MEM_CMD: begin
                            read_mem(seq_item_chk.addA);
                        end
                        WR_MEM_CMD: begin
                            write_mem(seq_item_chk.addC, seq_item_chk.DQ_i);

                        end
                        ADD_CMD: begin
                            add(seq_item_chk.addA, seq_item_chk.addB,
seq_item_chk.addC);
                        end
                        SUB_CMD: begin
                            sub(seq_item_chk.addA, seq_item_chk.addB,
seq_item_chk.addC);
                        end
                endcase
            end
        endtask : ref_model

        task automatic read_mem(input [mem_length-1 : 0] addr);
```

```
            begin
                ref_res = ref_mem[addr];
                @(negedge modvif.clk);
                if (ref_res == seq_item_sb.DQ_o) begin
                    correct_count = correct_count + 1;
                end
                else error_count = error_count + 1;
            end
        endtask
        task automatic write_mem(input [mem_length-1 : 0] addr, input [mem_width-
1 : 0] data);
            begin
                ref_mem[addr] = data;
            end
        endtask
        task automatic add(input [mem_length-1 : 0] addrA, input [mem_length-1 :
0] addrB, input [mem_length-1 : 0] addrC);
            begin
                ref_mem[addrC] = ref_mem[addrA] + ref_mem[addrB];
            end
        endtask
        task automatic sub(input [mem_length-1 : 0] addrA, input [mem_length-1 :
0] addrB, input [mem_length-1 : 0] addrC);
            begin
                ref_mem[addrC] = ref_mem[addrA] - ref_mem[addrB];
            end
        endtask

    endclass : storage_scoreboard
endpackage : storage_scoreboard_pkg
```

Needed sequences to be covered & Corner cases

Sequences to be Covered

Read Memory Command (RD_MEM_CMD):

- Scenario:
    - Input: Address addA.
    - Expected Output: The value at Mem[addA] is driven on the DQ line.
- Sequences:
    - **Valid Read**: Perform a read operation from a valid address and verify that the correct data is output.
    - **Repeated Read**: Perform repeated reads from the same address and verify consistent output.

Write Memory Command (WR_MEM_CMD):

- Scenario:
    - Input: Address addC, Data DQ.
    - Expected Output: The value of DQ is written to Mem[addC].
- Sequences:
    - **Valid Write**: Perform a write to a valid memory address and verify that the data is correctly stored.
    - **Write and Readback**: Write a value to memory and immediately read it back to verify the write operation.

Addition Command (ADD_CMD):

- Scenario:
    - Input: Addresses addA, addB, and addC.
    - Expected Output: The sum of Mem[addA] and Mem[addB] is written to Mem[addC].
- Sequences:
    - **Valid Addition**: Perform addition on valid addresses and verify that the result is stored correctly.
    - **Zero Addition**: Add memory contents where one or both operands are zero (Mem[addA] = 0 or Mem[addB] = 0).
    -

Subtraction Command (SUB_CMD):

- Scenario:
    - Input: Addresses addA, addB, and addC.
    - Expected Output: The difference between Mem[addA] and Mem[addB] is written to Mem[addC].
- Sequences:
    - **Valid Subtraction**: Perform subtraction on valid addresses and verify that the result is stored correctly.
    - **Zero Subtraction**: Subtract memory contents where one or both operands are zero (Mem[addA] = 0 or Mem[addB] = 0).

Corner Cases to be Considered

Address Boundaries:

- Ensure that read, write, addition, and subtraction operations handle the memory's boundary addresses correctly. For example, ensure operations at address 0 and the maximum address (2^mem_length - 1) behave as expected.

Data Width Overflow:

- Check for overflow and underflow conditions in addition and subtraction operations. For instance, when adding two maximum values (max_value + max_value), ensure proper handling of overflow.

Uninitialized Memory Access:

- Perform read operations on memory locations that have not been initialized to verify that the design handles uninitialized memory access safely.

Multiple Operations:

- Test back-to-back operations of different types (e.g., write followed by read, add followed by subtract) to ensure that the DUT correctly handles these transitions.
- Perform simultaneous operations on different memory addresses to check for potential conflicts.

Reset Behavior:

- Check how the DUT behaves after a reset. Ensure that memory values are retained or correctly initialized depending on the reset behavior specified.

Performance under Stress:

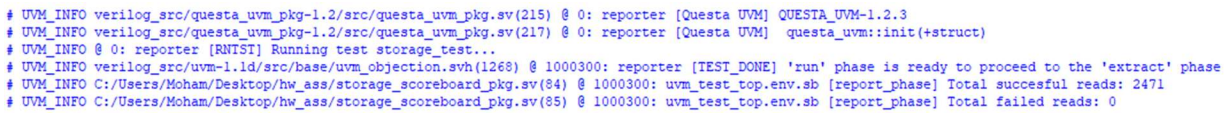
- Perform stress testing by issuing a large number of operations back-to-back and verify the system's performance

Results and testing script

Here we have extracted 2 types of results

1. Doing Read commands and comparing the reads with the output of the module:

   using a conditional if in the report_phase and using an error count and a correct count, as we can see in the screenshot below, we have 2471 successful reads and zero failed read attempts



2. Comparing the final memory contents using $fopen and $fwrite system functions

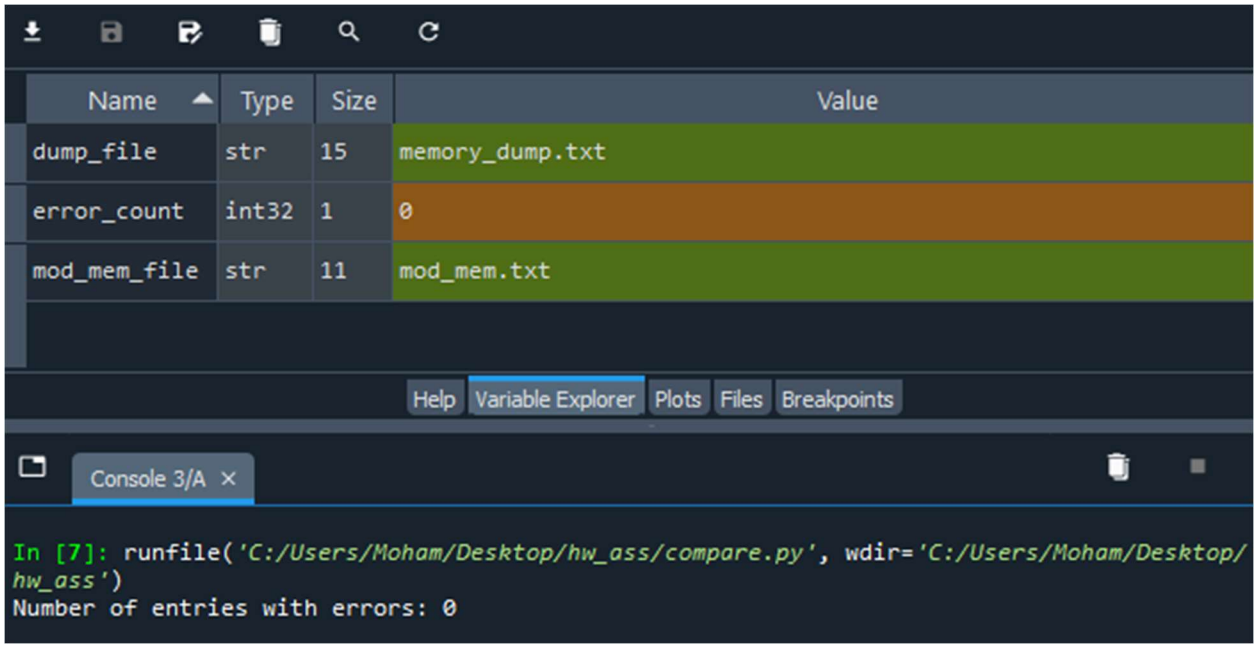
```
logic [mem_width-1 : 0] ref_mem [2**mem_length-1 : 0];
file = $fopen("memory_dump.txt", "w");
            // Check if the file is successfully opened
            if (file) begin
                // Write the contents of the memory array to the file
                for (int i = 0; i < 2**mem_length; i = i + 1)
                    $fwrite(file, "%0h\n", ref_mem[i]);
            end
            // Close the file
            $fclose(file);
```

Here we extracted the memory file in the testbench in a text file called `memory_dump.txt` and for the module itself I included a do file to automate the generation of the memory contents of the module itself, the do file is as follows:

```
vsim -gui -voptargs=+acc work.uvm_top
run -all
mem save -o mod_mem.txt -f mti -noaddress -data hex -addr hex -startaddress 0
-endaddress 255 -wordsperline 1 /uvm_top/m0/memory
```

To compare the contents of both memories, I wrote a pthon script that loops through the memory contents and outputs how many errors or unmatches were found here is the python script used and thr result is included below

```
import numpy as np

def compare_memory_files(mod_mem_file, dump_file):

    with open(dump_file, 'r') as f:
        dump_mem = np.array(f.read().splitlines())

    with open(mod_mem_file, 'r') as f:
        mod_mem = np.array(f.read().splitlines()[3:])

    if len(mod_mem) != len(dump_mem):
        raise ValueError("Files have different lengths. Ensure both memory files
contain the same number of entries.")

    mod_mem_normalized = np.char.zfill(mod_mem, 4)
    dump_mem_normalized = np.char.zfill(dump_mem, 4)
    mismatches = np.sum(mod_mem_normalized != dump_mem_normalized)
    return mismatches

if __name__ == "__main__":
    mod_mem_file = 'mod_mem.txt'
    dump_file = 'memory_dump.txt'

    error_count = compare_memory_files(mod_mem_file, dump_file)
    print(f"Number of entries with errors: {error_count}")
```


Name
Type
Size
Value
dump_file
str
15
memory_dump.txt
error_count
int32
1
0
mod_mem_file
str
11
mod_mem.txt
Help
Variable Explorer
Plots
Files
Breakpoints
Console 3/A
In [7]: runfile('C:/Users/Moham/Desktop/hw_ass/compare.py', wdir='C:/Users/Moham/Desktop/hw_ass')
Number of entries with errors: 0