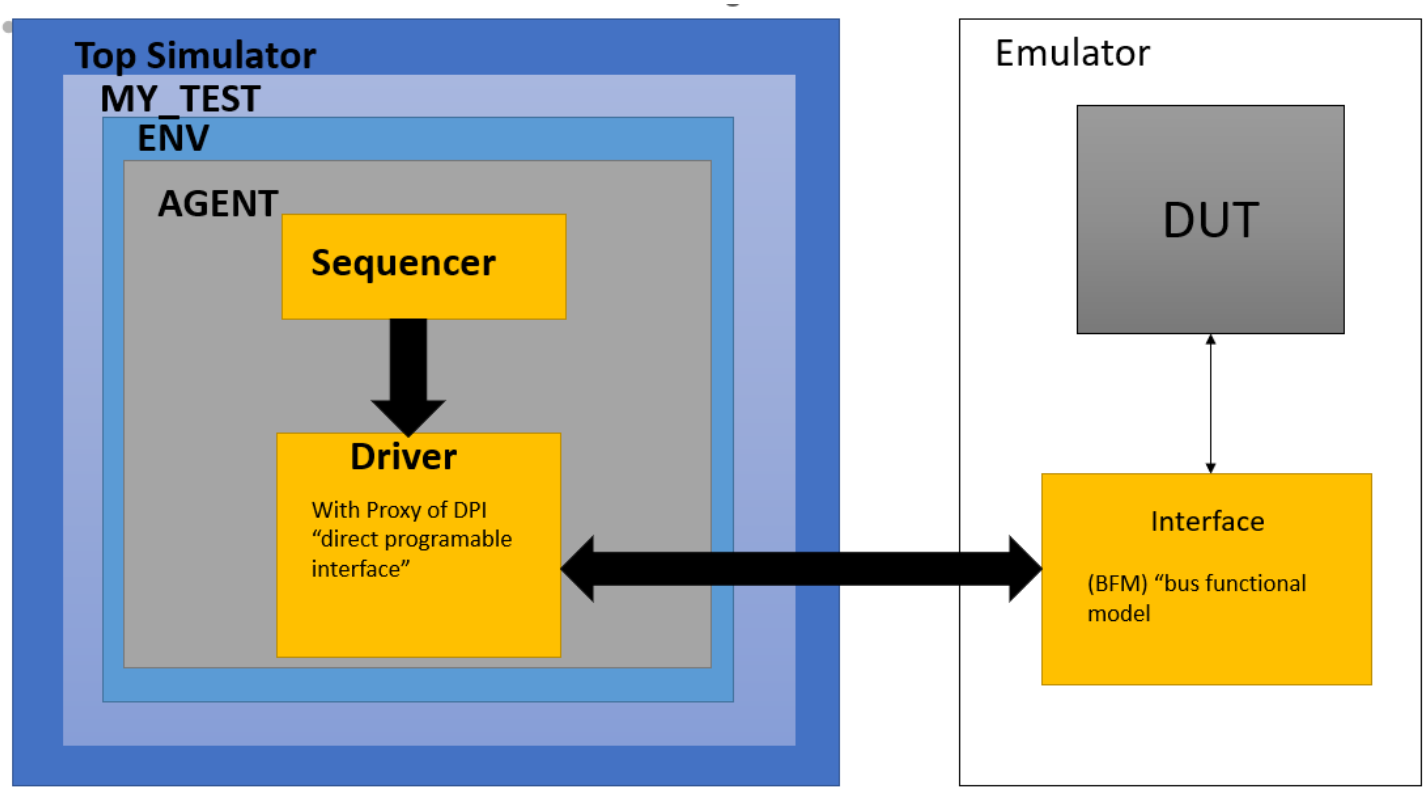
Submitted by:

Mohamed Waleed Abdelhafez

Contents

Question 1:



-Emulator is used to accelerate the UVM testbench so using emulator will saves a lot of time than using the simulation

-We need to connect the simulator with the emulator and that happens by writing a proxy of a direct programable interface "DPI" and connecting it with BFM interface in the emulator

-DPI is interface which can connect and initialize connection between System Verilog and any other language

Question 2:

The main criteria of developing the UVM with Emulator is to save time as doing the simulation on FPGA "Emulator" of course will be faster than doing it on CPU "Simulator"

So using Emulator will accelerate the UVM testbench

Also we connect the simulator with the Emulator by SCE-MI interfaces by one of the following 3 ways

1-Macro-based: where we send data from simulator to emulator by Macros

2-Function-based: use the system Verilog to call RTL functions running in the Emulator

3-Pipes-based: use of predefined System Verilog APIs to send the messages to the emulator

Question 3:

1-**System Specifications**: in this step we decide the specs that we want to achieve in the design before start coding

2-**Functional Design & Simulation**:

-in this step we start coding the design using HDL languages whether it's VHDL or Verilog, where we will define the behavioral function of the design

-Then we will start our simulation using testbench or UVM to check the functionality of the design

3- **Synthesis & Simulation**:

-In this step we turn our design to the gate-level

-Then we will make logic simulation on the design after gate-level logic

4-**Technology mapping & Simulation**:

-Here we will add the gate delays depending on the technology used and simulate to check the functionality and timing "Setup & hold violations" after adding the gate delays

5-**Place & route with Simulation**:

-Here we place the design components from the library and start routing "wiring" , adding wires delay

-Then we start simulation to check that the timing of the design is met after adding wiring delays

6- **Post-layout simulation "Verification of the design"**

-Before the fabrication we check that the system specs are met

7-**Fabrication:**

-we could fabricate the design on ASIC "Application specific integrated circuit" or programable devices like: FPGA

Question 4:

The functional verification process passes by many steps:

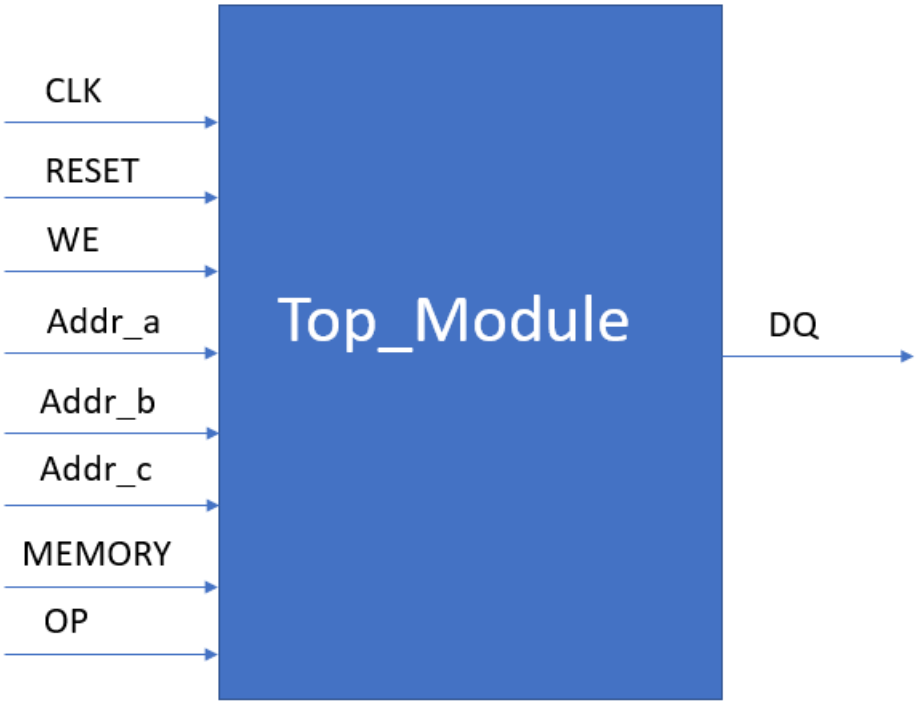
1-set DUT specs and putting verification plan

2-Creating the testbench environment

3-writing testcases to test the functionality of the design

4-creating code coverage and functional coverage modules to measure scenarios, corner cases & the system specifications as the random test and focused test are added to the coverage module then the verification will be more accurate

Top Module of the design:



Inputs & output signals of the top module:

CLK: input signal to drive the clock to the design

RESET: input signal used to reset the design when it's equal 0

WE: write enable signal used to decide whether DQ is input or output (when we=1 → DQ is input , when we=0 → DQ is output)

Addr_a : input signal where the user choose which address A he needs to make

Addr_b: input signal where the user chooses which address B he needs to make

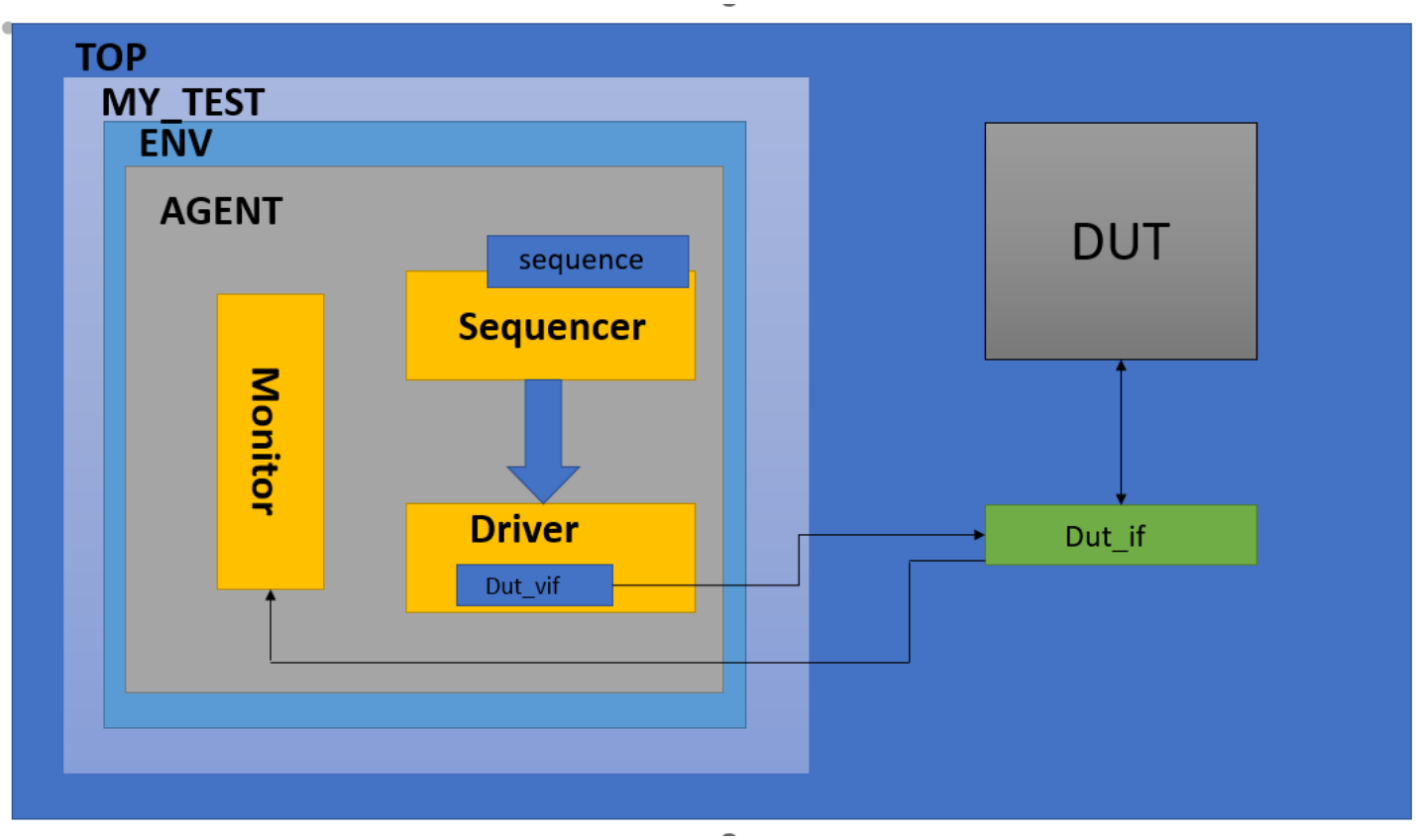
Addr_c: input signal where the user chooses which address C he needs to make

MEMORY: The memory of the design which is preloaded by a file produced through a TCL script and it contains 16 locations each one of 16 bit (we could change the size of the memory by the parameters address_width & data_width in the code)

OP: input signal used to decide which operation we are going to do while WE=0 (when OP=0 →DQ=MEMORY[address C] ,when OP=1 → MEMORY[address c]= MEMORY[address a]+ MEMORY[address b] , when OP=2 → MEMORY[address c]= MEMORY[address a]- MEMORY[address b]

DQ: is inout signal depending on the value of the WE.

UVM Environment:



Environment Components:

Top: Contains the testbench package , DUT & interface where it connects the components of the environment to the DUT through the interface also here we preload the memory and intiate the clock

Test package: Contains the environment and here we start the process of each component

Environment: Responsible for managing all the components also it initiate the agent and the scoreboard if found

Agent: Responsible for connecting the sequencer, the driver & the monitor also it provides ports for the monitor to send the transaction to the coverage and scoreboard modules

Monitor: it monitors the transactions seen on the interface

Sequencer: Responsible for creating the transactions through different random sequences

Driver: Responsible for decoding the transactions taken from the sequencer and driving it to the DUT through the actual and virtual interfaces.

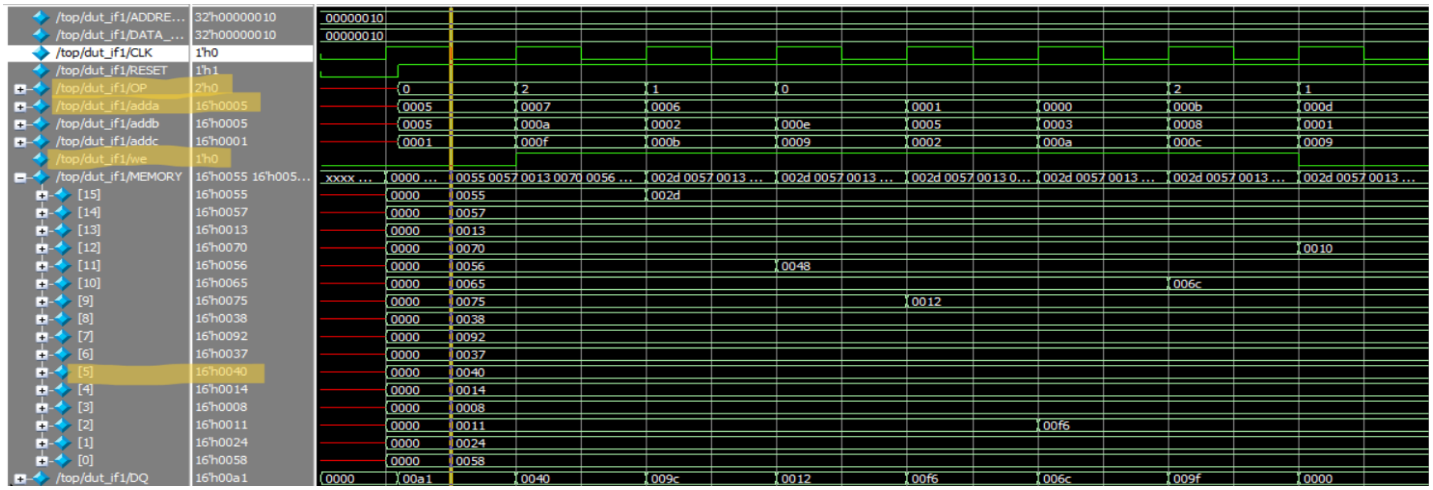
Results:

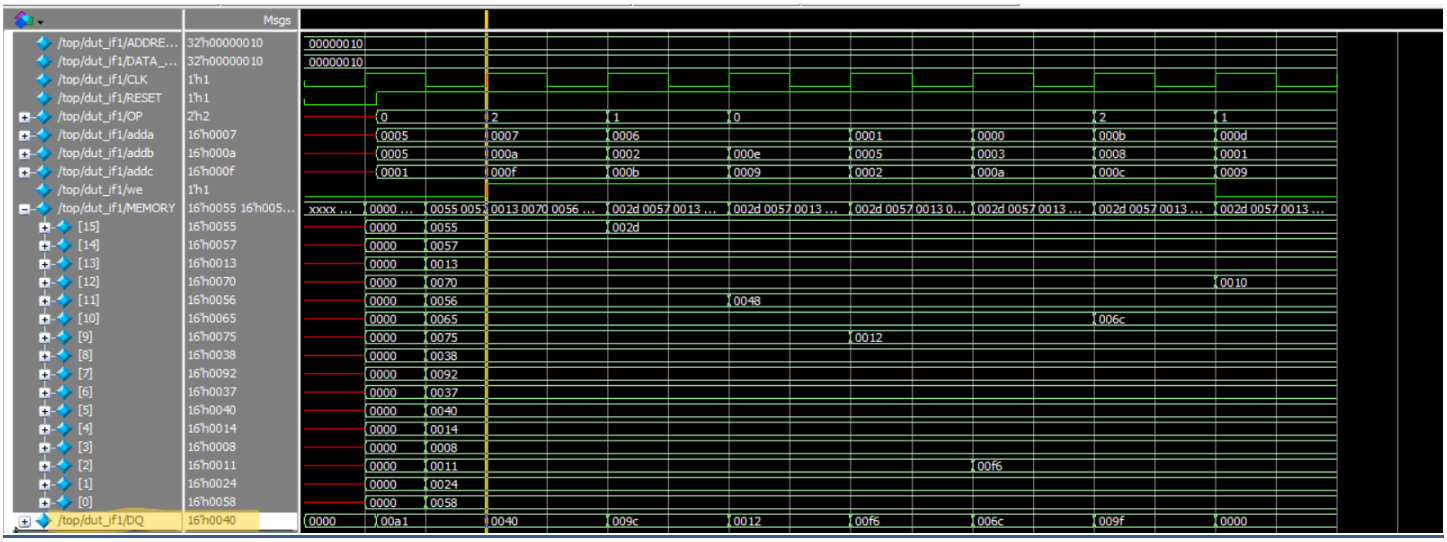
We firstly set Reset=0 at 5 ns to reset our memory then at 10 ns we start to preload our memory through the input file produced from the TCL script with 16 random values.

Note: values in the simulation are in hexadecimal,we also generate 8 random sequences through the sequencer

- In each test case we have two figures where the first figure show the transaction before the positive edge clock while the second figure show the result of this transaction at the positive edge clock

1st test:

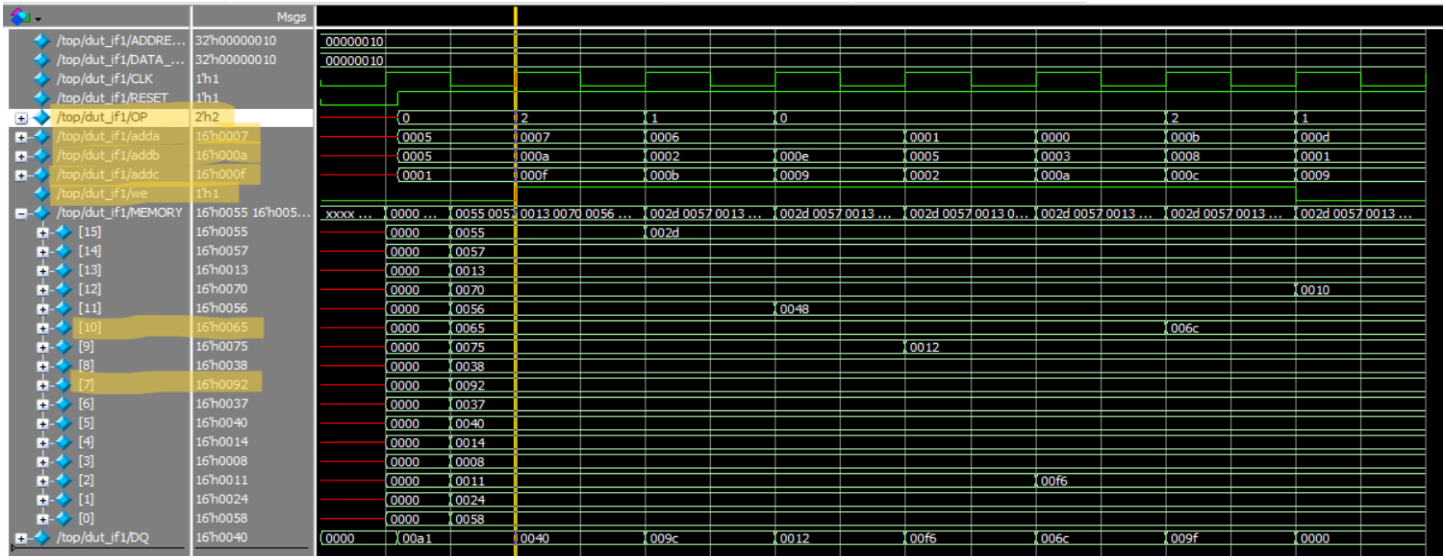


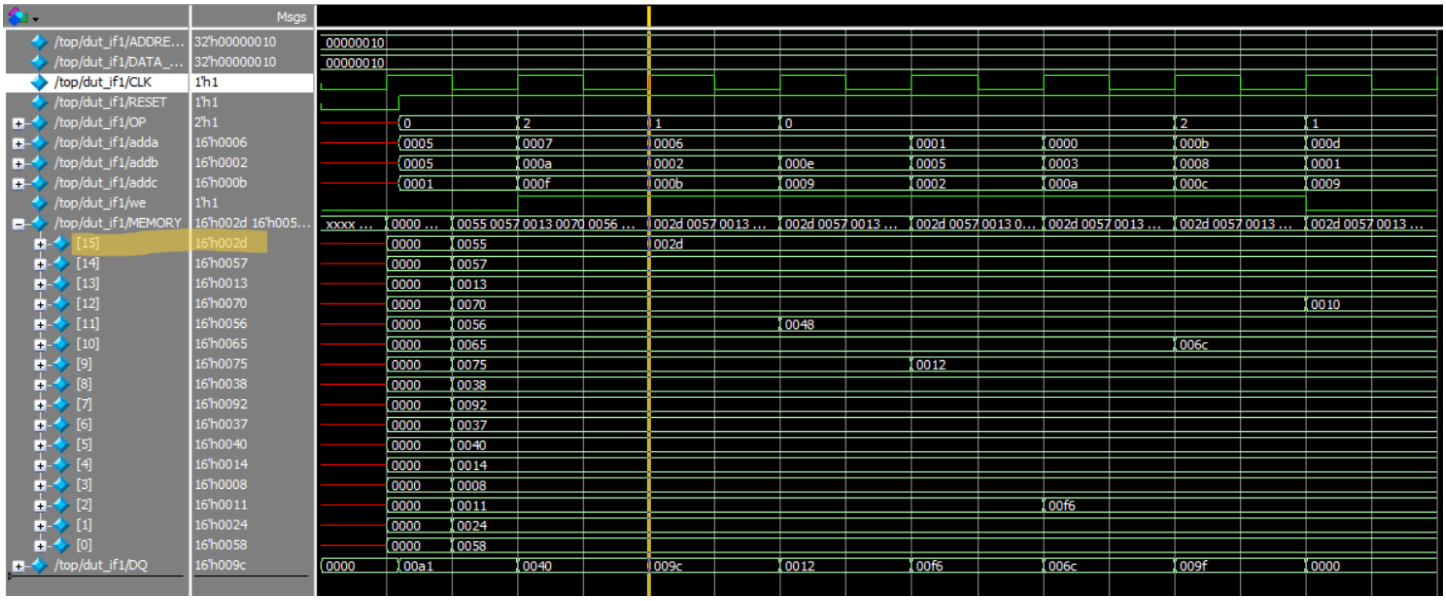


Our first test where in the transaction the WE=0 , ADDRESS A = 5 → so in the next posedge CLK we will have DQ=Memory[Address A]=16'h0040 as shown in both figures in the highlighted parts.

Here we will discard the value of OP as the write enable is zero so in this case we will have DQ as an output port and we will not write in the memory

2nd test:

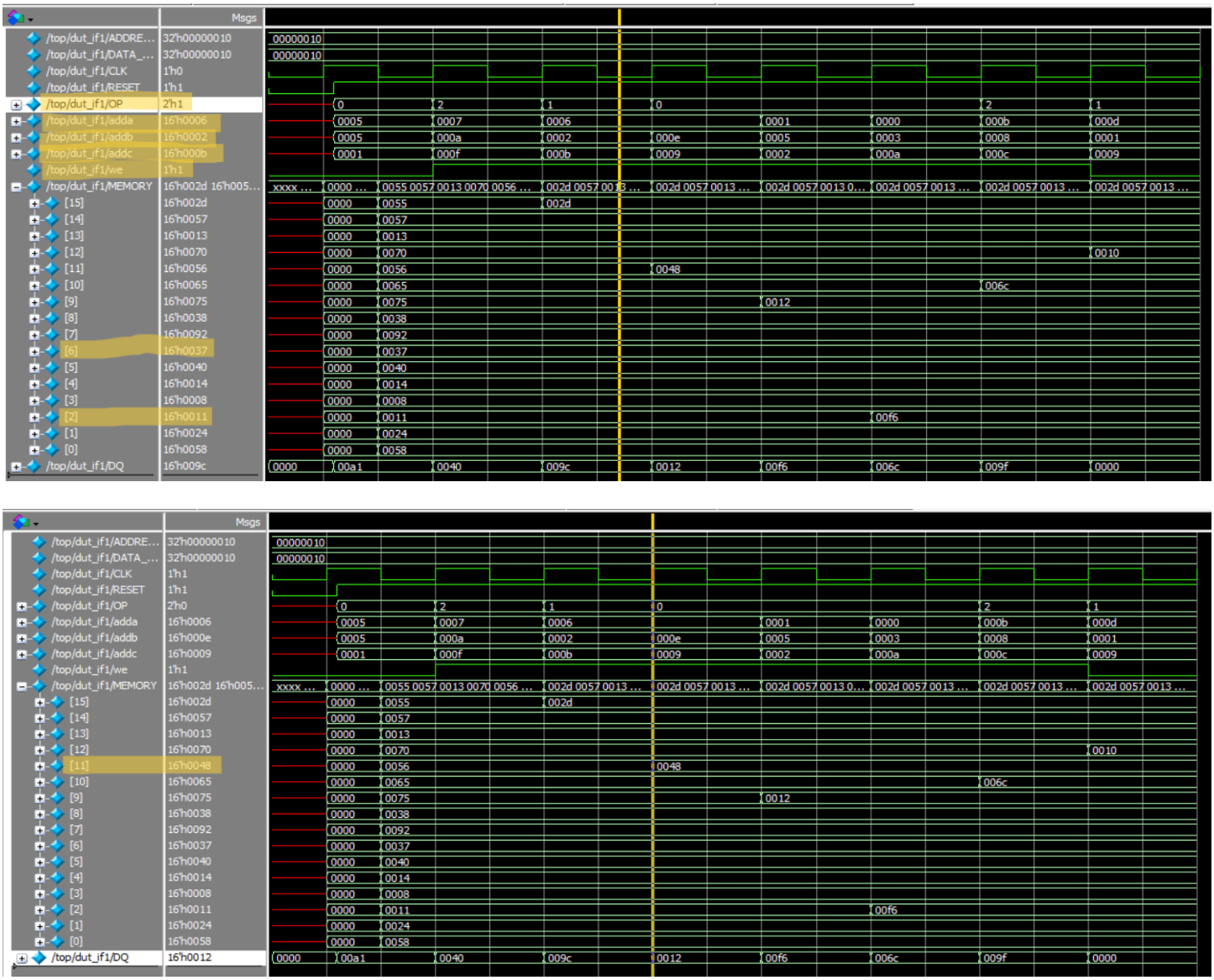




Our second test where in the transaction the WE=1 , OP=2 ,ADDRESS A=7, ADDRESS B = 10, ADRESS C= 15 → so in the next posedge CLK we will have

MEMORY[Address C]=Memory[Address A]-Memory[Address B]=16'h0092-16'h0065=16'h002d as shown in both figures in the highlighted parts

3rd Test:

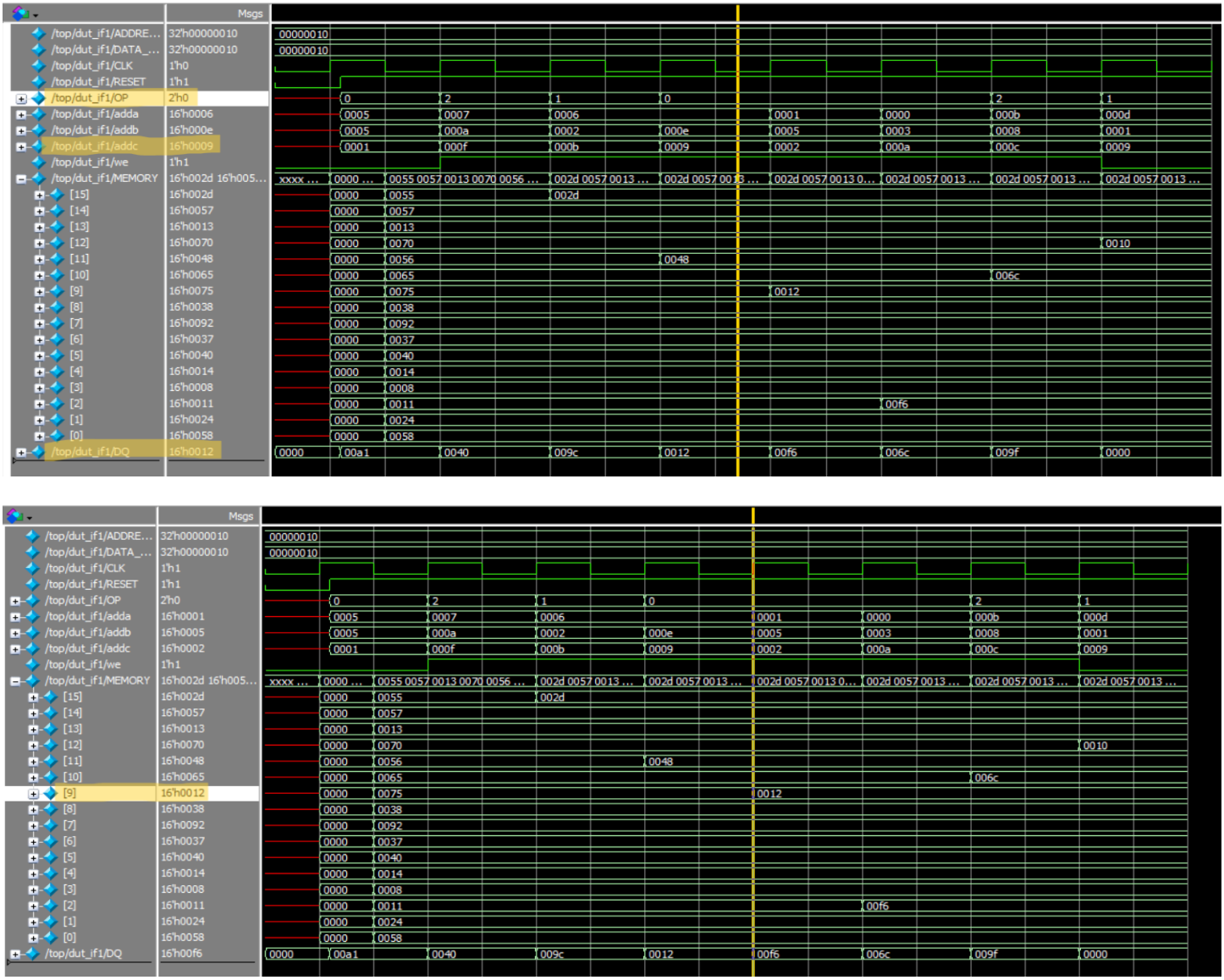


Our third test where in the transaction the WE=1 , OP=1 ,ADDRESS A=6, ADDRESS B = 2, ADRESS C= 11 → so in the next posedge CLK we will have

MEMORY[Address C]=Memory[Address A]+Memory[Address B]=16'h0037

+16'h0011= 16'h0048 as shown in both figures in the highlighted parts

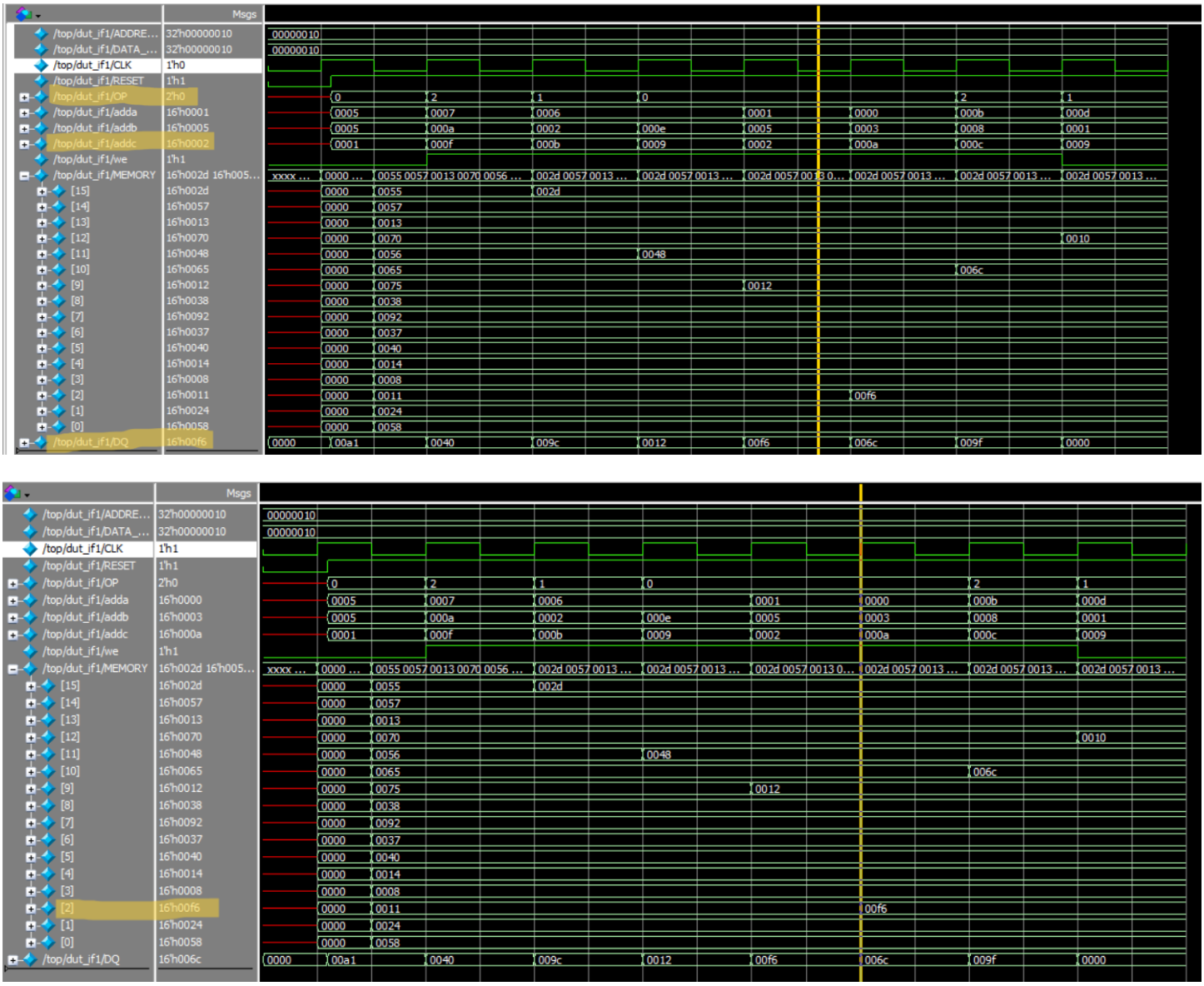
4th test :



Our fourth test where in the transaction the WE=1 , OP=0 , ADRESS C= 9, DQ=16'h0012 → so in the next posedge CLK we will have

MEMORY[Address C]=DQ =16'h0012 as shown in both figures in the highlighted parts

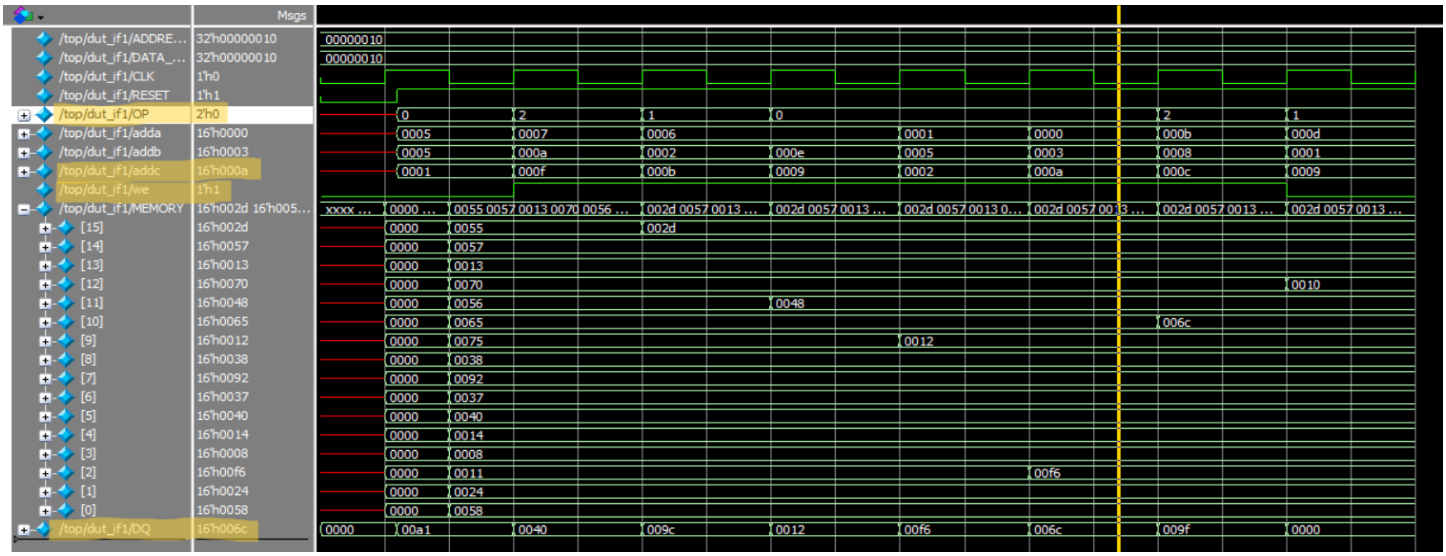
5th test:

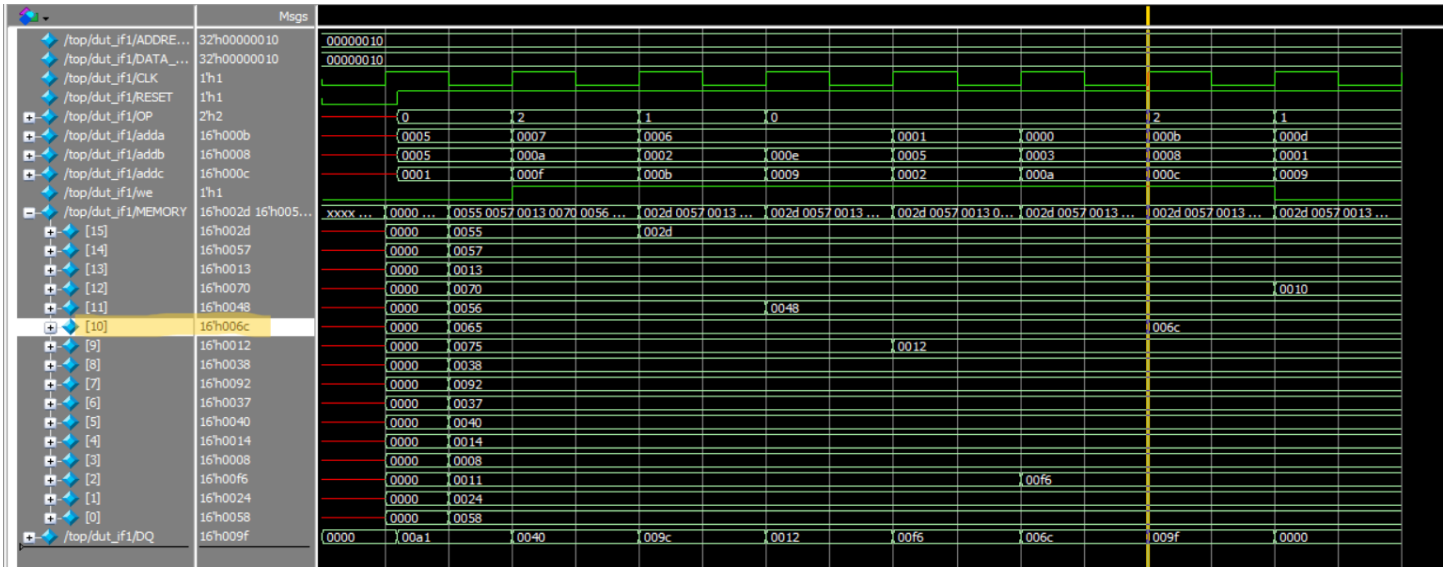


Our fifth test where in the transaction the WE=1 , OP=0 , ADRESS C= 2 , DQ=16'h00f6 → so in the next posedge CLK we will have

MEMORY[Address C]=DQ =16'h00f6 as shown in both figures in the highlighted parts

6th test:





Our sixth test where in the transaction the WE=1 , OP=0 , ADRESS C= 10 , DQ=16'h006c → so in the next posedge CLK we will have

MEMORY[Address C]=DQ =16'h006c as shown in both figures in the highlighted parts

7th test:

Our third test where in the transaction the WE=1 , OP=2 ,ADDRESS A=11, ADDRESS B = 8, ADRESS C= 12 → so in the next posedge CLK we will have

MEMORY[Address C]=Memory[Address A]-Memory[Address B]=16'h0048

-16'h0038= 16'h0010 as shown in both figures in the highlighted parts

Display of the monitor every 10 ns

```
# UVM_INFO monitor.sv(43) @ 5: uvm_test_top.env.agent.m0 [MON] Saw address A = 0  Saw address B = 0  Saw address C = 0
# UVM_INFO monitor.sv(44) @ 5: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 0  Saw write_enable = 0  Saw DQ = 0
# UVM_WARNING D:/uvm_examples/my_testbench_pkg.sv(86) @ 10: uvm_test_top [] Hello World!
# UVM_INFO monitor.sv(43) @ 15: uvm_test_top.env.agent.m0 [MON] Saw address A = 5  Saw address B = 5  Saw address C = 1
# UVM_INFO monitor.sv(44) @ 15: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 0  Saw write_enable = 0  Saw DQ = 161
# UVM_INFO monitor.sv(43) @ 25: uvm_test_top.env.agent.m0 [MON] Saw address A = 7  Saw address B = 10  Saw address C = 15
# UVM_INFO monitor.sv(44) @ 25: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 2  Saw write_enable = 1  Saw DQ = 64
# UVM_INFO monitor.sv(43) @ 35: uvm_test_top.env.agent.m0 [MON] Saw address A = 6  Saw address B = 2  Saw address C = 11
# UVM_INFO monitor.sv(44) @ 35: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 1  Saw write_enable = 1  Saw DQ = 156
# UVM_INFO monitor.sv(43) @ 45: uvm_test_top.env.agent.m0 [MON] Saw address A = 6  Saw address B = 14  Saw address C = 9
# UVM_INFO monitor.sv(44) @ 45: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 0  Saw write_enable = 1  Saw DQ = 18
# UVM_INFO monitor.sv(43) @ 55: uvm_test_top.env.agent.m0 [MON] Saw address A = 1  Saw address B = 5  Saw address C = 2
# UVM_INFO monitor.sv(44) @ 55: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 0  Saw write_enable = 1  Saw DQ = 246
# UVM_INFO monitor.sv(43) @ 65: uvm_test_top.env.agent.m0 [MON] Saw address A = 0  Saw address B = 3  Saw address C = 10
# UVM_INFO monitor.sv(44) @ 65: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 0  Saw write_enable = 1  Saw DQ = 108
# UVM_INFO monitor.sv(43) @ 75: uvm_test_top.env.agent.m0 [MON] Saw address A = 11  Saw address B = 8  Saw address C = 12
# UVM_INFO monitor.sv(44) @ 75: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 2  Saw write_enable = 1  Saw DQ = 159
# UVM_INFO monitor.sv(43) @ 85: uvm_test_top.env.agent.m0 [MON] Saw address A = 13  Saw address B = 1  Saw address C = 9
# UVM_INFO monitor.sv(44) @ 85: uvm_test_top.env.agent.m0 [MON] Saw OPERATION = 1  Saw write_enable = 0  Saw DQ = 0
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 85: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   22
# UVM_WARNING :    1
# UVM_ERROR :    0
# UVM_FATAL :    0
```

Here the monitor display every signal every 10 ns by uvm_info command also we could display the memory here

Tcl script used to preload memory

puts "*************** Scripting file using TCL to Preload Memory of 16 random values***************"

set fh [open input_file w+]

for {set x 0} {$x<16} {incr x} {

set MEMORY($x) [exp int ([exp rand()*100])]

puts $fh $MEMORY($x)

}

close $fh

- We also print output file of the last memory values by $fdisplay command in the testbench code

TCL script used to compare final memory values with the expected values and save whether they are identical or not in report file:

```
puts "****Comparing 2 files****"

set fh_0 [open expectedoutput_file.txt r]

set fh_1 [open output_file r]

set fh_2 [open report.txt w+]


set file_data_0 [read $fh_0]

set file_data_1 [read $fh_1]


set x [string compare $file_data_0  $file_data_1]


if {$x==0} {

set str "Both outputs are identical"

puts "Both outputs are identical"} else {

set str "Both outputs are different"

puts "Both outputs are different"}


puts $fh_2 $str


close $fh_0

close $fh_1

close $fh_2
```

Notes on the Project & Process:

-The process in this position was a very good learning opportunity as I was interested in studying the verification methodologies because it was a missing part in the digital design flow that I haven't learn it yet.

-So that was my first UVM project that was very challenging and it also expands my mind to invest my time in completing studying the UVM and its connectivity with the emulation.

-All of the report and the code of the project are due to self-education & searching over the internet without the help of anyone in writing the code or explaining commands.

-Future work on this project: Adding the coverage module and scoreboard module, but in this project, we have covered 8 random different cases including the 4-corner case also we have done a comparison between the final output of the memory and the expected final memory using the TCL Scripting