VLSI Circuits and Technology 2

Homework - 2, Group 7

Daniel Garanton	dgaranton@ufl.edu
Anthony Rodriguez	anthony7131998@ufl.edu
Aashish Tattikota	a.tattikota@ufl.edu
Anirudh Jayaram Melukote	anirudh.melukote@ufl.edu

Contributions

- The entire team worked together for designing and finalizing the microarchitecture for the DRAM Circuit.
- Anthony and Daniel worked on the implementation and validation of the FSM, PISO, top level wrapper along with their testbenches and overall integration of system.
- Aashish and Anirudh worked on the implementation and validation of L2 Buffers and counters along with their testbenches. They also wrote the report in Latex.

Design Overview

1. Since inside the DRAM there is no additional command buffer, all the commands need to be sent one by one (sequentially).
i. This was mainly achieved by only having one command register defined and assuring that each state in the finite state machine only modified it once. Also, we closely analyzed the FSM testbench to confirm that commands were being sent out sequentially.
ii. There was no clear method described on how to implement this constraint, so we decided that the most re-usable option was the use of handshakes to make sure that each command was executed completely before going on to the next command. A command is considered completed when it has received a complete acknowledgement cycle (assertion and de-assertion).

2. Each command has its own execution delay. So, the controller must send the commands in such a way that all such delay constraints of DRAM can be maintained, and no command gets ignored.
i. This was done by checking for the acknowledgement bit in every state. We added a state that would wait if there was no acknowledgement received in one of the previous states. Once the acknowledgement was received it would then go back to the previous state that was waiting for the previous command to finish.

3. Periodically REFRESH command needs to be issued.
i. Added a counter to our microarchitecture that would assert a refresh flag after 125ms. We had to make some calculations to accurately translate cycles to seconds. This flag was then fed to the FSM, where we are continuously checking for it. Once the refresh flag is asserted, we pre-empt any running state and make it transition to the refresh state, which then sends the commands needed to do a complete refresh

4. Though each bank can be accessed simultaneously but at a time only one bank can transfer data to DRAM controller via DRAM bus.
i. Our group did not test if we could access banks simultaneously while only one bank was performing the data transfer. Rather, we upheld this constraint by designing a bank decoder that only selects one bank at a time no matter the case. The input to the decoder was the bank id coming from the request buffer, and the output was a decoded bank select. Examples of the output are 0000 0001, 0000 0010, 0000 0100. There will never be a case where there are two bits, or two banks asserted or selected.

Hardware Implementation

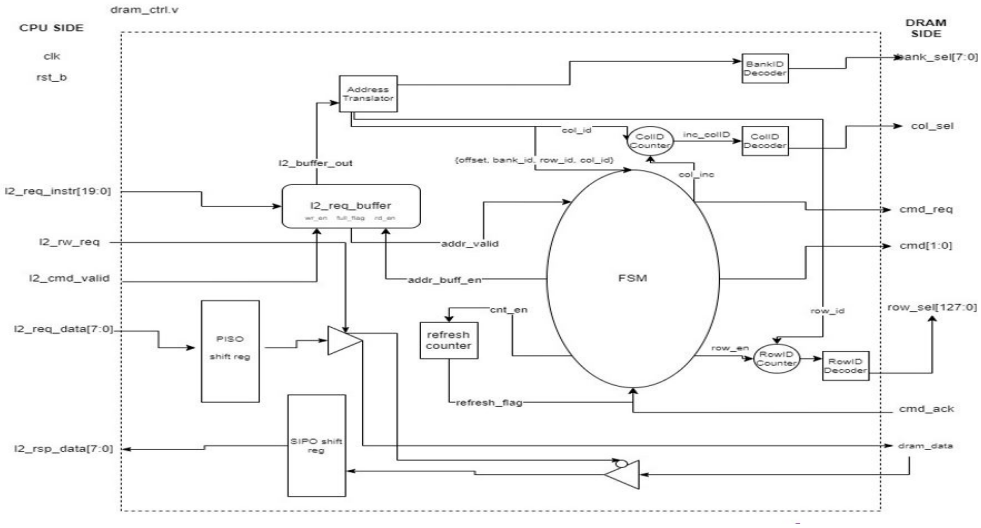

Figure 1: Microarchitecture of the DRAM Controller Circuit

Figure 1 depicts the proposed DRAM controller microarchitecture with respect to the two interface sides. Whenever there is a miss read or write within a L2 cache, a request instruction will be sent with the request data and request Read/Write enable. DRAM controller will process this request on a First Come First Serve (FCFS) basis, toggling various signals on the DRAM side. DRAM side signals directly connect to the DRAM banks and buffers as depicted in the specification (Figure 2). Table 1 details all the signals defined within the I/O interface for the DRAM controller.

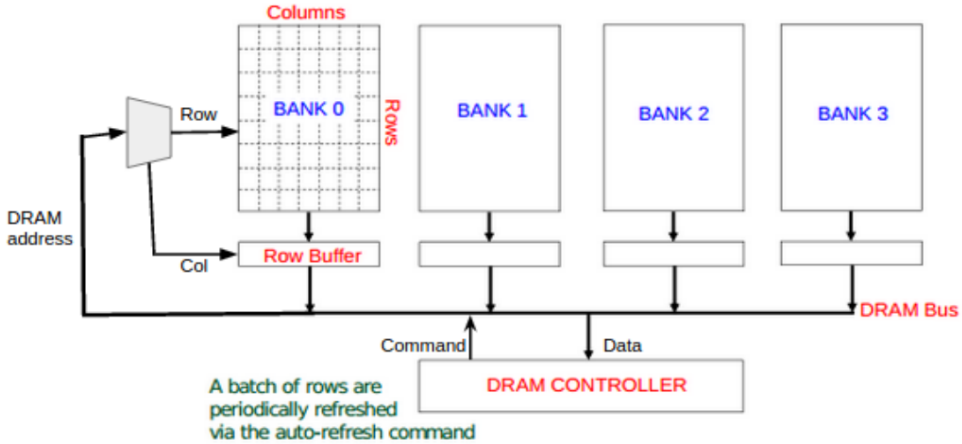

Figure 2: Provided Specification Diagram of the DRAM Controller Circuit (DRAM Side)

Table 1: Input and Output Signals for the DRAM Controller Circuit

Signal Name	Signal Direction	Bit length	Purpose
clk	Input	1	To synchronize with a clock signal of 150 MHz
rst_b	Input	1	Active-Low Reset for Sequential blocks
l2_rw_req	Input	1	Indicates if Instruction is a Read or Write Access
cmd_ack	Input	1	Command acknowledgement
l2_req_data	Input	8	The data to be written into a particular DRAM bank row
l2_req_instr	Input	20	Read-and-write Instruction that concatenates access count and address: **<offset, address>**
dram_data	Inout	1	Bi-directional data bus
cmd_req	Output	1	Command Request Signal to indicate an issued command to DRAM banks
cmd	Output	2	Indicates to DRAM which command needs to be executed
bank_sel	Output	8	Selects which bank will take control of DRAM Data bus
row_sel	Output	8	Indicates a particular row within a bank to write
col_sel	Output	127	Indicates a particular column within a bank to write to a bit cell
L2_rsp_data	Output	8	Accessed data from row within DRAM Bank

L2 Request Buffer

The L2 Request Buffer consists of a depth size equal to the number of rows (128) and a bit width of equal to the L2 Instruction width (20). L2 request instructions will fill up buffer whenever there is a valid command sent to the DRAM controller. This is determined by the l2_cmd_valid signal. Within the module, there are two pointers that dictate which register position: rd_addr and wr_addr. FSM will assert a addr_buff_en whenever it wishes to process the next instruction.

FIFO is an acronym for First In First Out, which describes how data is managed relative to time or priority. In this case, the first data that arrives will also be the first data to leave from a group of data. A FIFO Buffer is a read/write memory array that keeps track of the order in which data enters into the module and reads the data out in the same order. It is often implemented as a circular queue, and has two pointers:

I. Read Address Pointer
II. Write Address Pointer

Read and write addresses are initially both at the first memory location, say zero. We used a signal called “count to determine the difference between the read address and

write address of the FIFO. Initially the FIFO Counter is set to 0. When this difference is equal to the size of the memory array, then the FIFO queue is Full. The FIFO is empty when the FIFO counter is zero.

We designed a synchronous FIFO where writes to the buffer and reads from FIFO Buffer are conducted in same clock domain. The FIFO is filled with data only when the FIFO Counter is not full. The FIFO is read when it is not empty. This ensures that we do not write into an already full buffer or do not read from it when there's no actual data present in it.

Our implementation counts the number of writes to FIFO and reads from the FIFO buffer to increment on FIFO write but no read, decrement on FIFO read but no write or hold (no writes and reads, or simultaneous write and read operation) the counter value of the FIFO buffer.

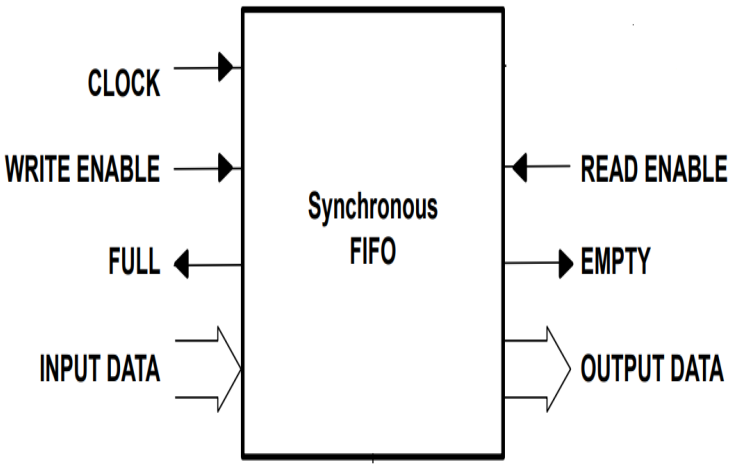

Figure 3: Block Diagram of FIFO

Refresh Counter

Refresh rate was defined in the given specification to be every 125 ms. To achieve this functionality, a synchronous counter module was designed. Assuming a clock frequency of 150 MHz for our circuit, we know that the period of circuit is 6.77 ns (1/150M). For 125 ms, we have 17857142 cycles. Hence, the counter counts from 0 to 17857142.

The counter is always enabled. When the enable signal is 1, our counter starts counting from 0. Once it reaches 17857142, refresh flag becomes 1 and the counter is reset to 0. The refresh flag is used by our DRAM Controller (FSM) to generate the Refresh Command and enter the Refresh State of the FSM.

Key component of the proposed DRAM controller consists of a six state FSM that handles the following:
• First Come First Serve (FCFS) scheduling of incoming L2 cache requests.
• Waits for a particular command to complete through a REQ/ACK handshake scheme.
• Assertions of Command bits sent to the DRAM banks and buffers
• Mealy and Moore Outputs to Control Read/Write of DRAM Banks and internal modules.
Since this design is intended for an Advanced SoC, grey encoding of our cmd and state bits were chosen to optimize power consumption during state transitions. Additionally, state and next_state signals were given a bit-width of three, particularly since the number of states could be represented in three bits. These states were stored as local parameters within the FSM module. The following section details each state and appropriate state transition logic.

Finite State Machine States and Transition Logic

Table 2: Input and output signals for the FSM

Signal Name	Signal Direction	Bit length	Purpose
Clk and rst_b	Input	1	To synchronize with a clock signal
refresh_flag	Input	1	When refresh counter reaches 17857142, refresh flag is asserted
cmd_ack	Input	1	Command acknowledgement
bank_id	Input	clog2(NUMBER_OF_BANKS)	for 8 banks, size = 3
row_id	Input	clog2(NUMBER_OF_ROWS)	for 128 rows, size = 8
col_id	Input	clog2(NUMBER_OF_COLUMNS)	for 8 columns, size = 3
offset	Input	clog2(NUMBER_OF_ROWS)	for 128 rows, size = 8
count_en	Output	1	This enables the counter
row_inc	Output	1	To increment the row id and go to next row
col_inc	Output	1	To increment the column id and go to next column
cmd_req	Output	1	Command Request Signal to indicate an issued command to DRAM banks
cmd	Output	2	Indicates to DRAM which command needs to be executed
row_en	Output	1	Row enable to store inc_row_id with the value from the address buffer
col_en	Output	1	Column enable to store inc_col_id with the value from the address buffer
load_data	Output	1	Enables the l
bank_en	Output	1	Enable for bank selection
address_buff_en	Output	1	Enable for address buffer selection
read_data_en	Output	1	Enable for reading the data

1. **IDLE_STATE (3'b000)**
This is the initial state of our DRAM and it does nothing in this stage. When address valid bit is high, it goes to the next state which is Bank and Row Access State.

2. **BNR_STATE(3'b001)**
This the Bank and Row Access state. In this state row buffer is filled with data corresponding to all cells of the selected bank and selected row. Here, if cmd_ack signal is not 0, we enable the load data bit, reset cmd to zero and check for access count. If access count is zero, we set next_access_count to the offset and enable the address buffer and row. We then go to the next state which is WAIT_ACK STATE.

3. **COL_STATE (3'b010)**
This is the Column Access state. The aim is to select appropriate column from selected bank and selected row. If cmd_ack is not 0, we set cmd to 1. If column counter reaches its value of 8, we go the next row and also set column counter back to 0. If column counter is not 8, we increment the column counter by 1 to read/write data from the next column.

4. **PRECHARGE_STATE (3'b011)**
If access count is zero, we set next_access_count to the offset and enable the address buffer. Else, we set cmd = 11 and decrement access_count by 1.

5. **REFRESH_STATE (3'b100)**
Whenever refresh flag is asserted, our FSM goes to the refresh state. In this state, cmd is set to 10 and refresh counter is disabled. Since we wanted a design where our DRAM is refreshed every 125 ms, we assert the refresh flag whenever the counter reaches 17857142 (assuming a clock frequency of 150 MHz). if command acknowledge bit is high, FSM goes back to the previous state.

6. **WAIT_ACK (3'b101)**
In this state we check for the command acknowledge bit. If command acknowledge bit is high, our FSM goes from previous state to the next state. So if the previous state is BNR_STATE, FSM goes to the COL_STATE. If the previous state is COL_STATE, FSM goes to the PRECHARGE_STATE. If the previous state is PRECHARGE_STATE, FSM goes back to the BNR_STATE.

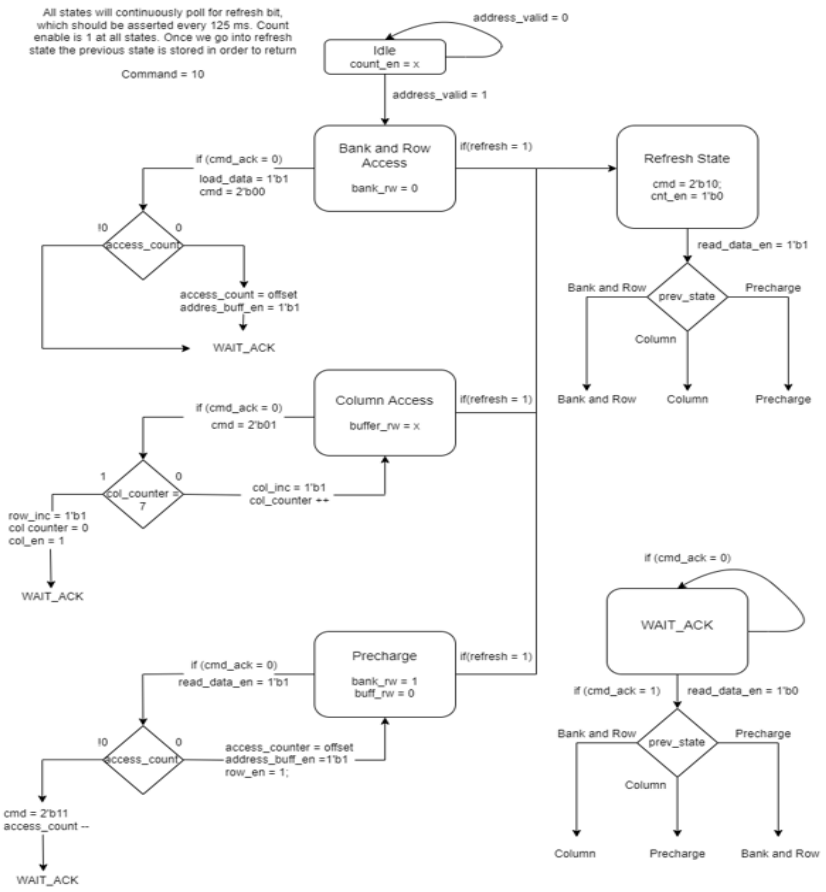

Figure 4: FSM State Transition Diagram

Parallel In Serial Out (PISO) Shift Register

A parallel-in serial out shift register module was designed to feed the dram_req_data (data that is going to be used in writes) to the dram memory module. We deemed it necessary because the input to the controller is an 8-bit value while the output from the controller to the memory cells needed to be single bit values. Therefore, a parallel-in serial out module was used to read out 8-bit data values as 1-bit data entries to the memory cell. As a result, 8 cycles were needed to completely feed the 8-bit data entry to memory.

Serial-In Parallel Out (SIPO) Shift Register

A serial in parallel out shift register module was designed to read from memory and output this data in the correct format. It was necessary because dram reads are done in a bit-by-bit fashion and the response data is output 8 bits at a time. Therefore, a serial in parallel out module was used to accumulate eight 1-bit reads from the dram memory and store them into a single 8-bit value. This 8-bit value was then used as the output for the controller when a read is performed from a certain row in a bank.

Validation

We tested our system at an individual module level and then also tested the system at the top-level/wrapper level. The individual components tested were the following: FSM, Decoders, Buffers, Shift Registers and Refresh Counter for refresh flag. This bottom-up testing scheme allowed for systematic validation and seamless easy integration time to generate top-level module.

Since this design communicates with various DRAM banks and buffers, a DRAM Behavioral Functional Model (BFM) was developed and instantiated on the wrapper testbench. This model behaves the same as the specification's bank and buffers. Additionally, it made testing for read and write coherence simpler.

Quick Testbench Explanation

To begin the testbench we instantiate the dram controller and the dram behavioral functional model. Then, we connect these modules to simulate a complete dram design. After connecting all the signals, we start the test bench by running the clock and making sure to assert/reassert the reset signal to begin the testing with a clean design. After the reset, we drive make begin writing to the memory via the bfm by asserting the l2_rw_req to write and driving l2_req_intr signal with a random signal to make sure we are getting random column, row, and bank ids. After we write to these random locations, we make sure to read from these locations by driving l2_rw_req to a read state and waiting for the controller to get back the values from the memory. After we complete the writes and reads to check for correct functionality, we wait until the refresh flag is asserted at 125 ms to make sure that this behavior is functioning correctly as well. After the refresh flag is asserted, we give some time for the module to run and see what happens on a refresh. Finally we deactivate the clock. Throughout this process we are constantly printing display functions on the terminal that show what data we are writing or reading. It also prints what cell (via bank, column, and row id) is being written to or read from.

Testing the Buffers

We created a test bench to test the functionality of our FIFO buffer. We decided upon some test inputs – 00000000, 00000001, 00000010, 00000011 00000100 to be given as stimulus and verify if they were appearing on the outputs of our FIFO. Our FIFO model worked for the stimulus provided and we could observe that all our inputs were written into the FIFO when write enable is on (logic 1). This data then appeared at the output when read enable is on (logic 1). The model was simulated using ModelSim and the waveform for the same is attached below:

Testing the Counter

We tested our counter using a test bench and the simulation screenshot is attached be-

low:

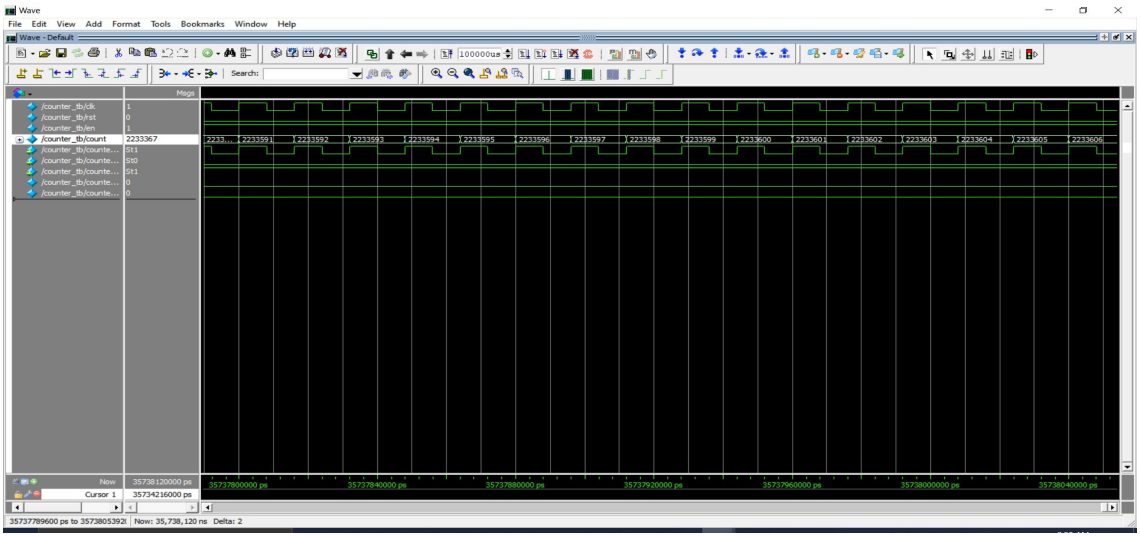

Figure 5: Counter Simulation

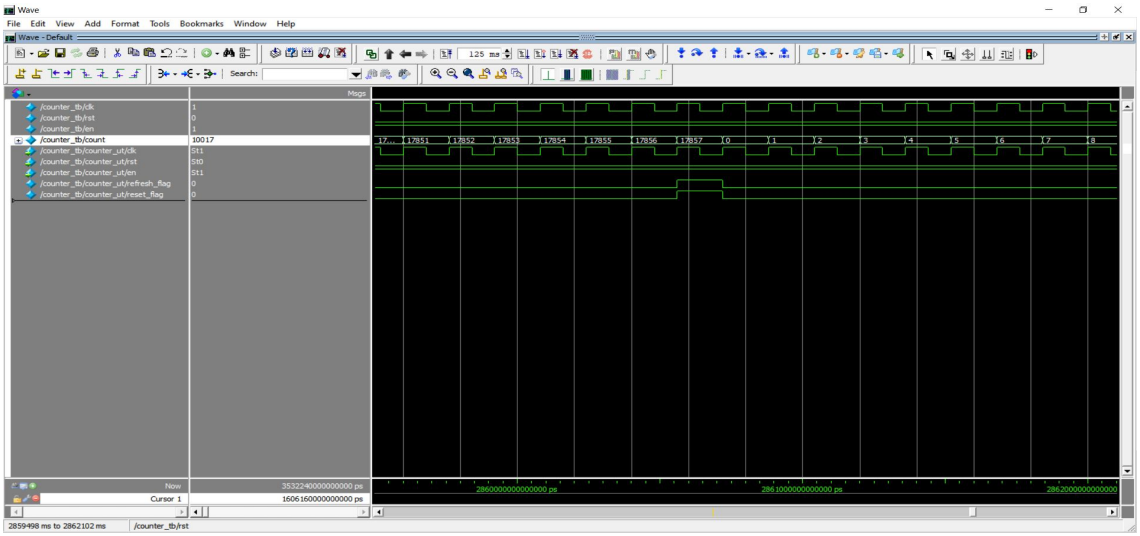

Figure 6: Counter resets when it reaches value of 17857 (for sake of testing)

```
# time=0.698595000ms l2_rsp_data=0 bank_id=5 row_id=6 col_id=0
# time=0.711995000ms l2_rsp_data=ec bank_id=5 row_id=4 col_id=0
# time=0.712195000ms l2_rsp_data=12 bank_id=5 row_id=5 col_id=0
# time=0.712395000ms l2_rsp_data=0 bank_id=5 row_id=6 col_id=0
# time=0.724595000ms l2_rsp_data=ec bank_id=5 row_id=4 col_id=0
# time=0.724795000ms l2_rsp_data=12 bank_id=5 row_id=5 col_id=0
# time=0.724995000ms l2_rsp_data=0 bank_id=5 row_id=6 col_id=0
# time=0.729795000ms l2_rsp_data=88 bank_id=5 row_id=1e col_id=0
# time=0.729995000ms l2_rsp_data=64 bank_id=5 row_id=1f col_id=0
# time=0.730195000ms l2_rsp_data=0 bank_id=5 row_id=20 col_id=0
# time=0.731195000ms l2_rsp_data=c6 bank_id=4 row_id=60 col_id=0
# time=0.731395000ms l2_rsp_data=a2 bank_id=4 row_id=61 col_id=0
# time=0.736195000ms l2_rsp_data=0 bank_id=4 row_id=5f col_id=0
# time=0.736395000ms l2_rsp_data=c6 bank_id=4 row_id=60 col_id=0
# time=0.736595000ms l2_rsp_data=a2 bank_id=4 row_id=61 col_id=0
# time=0.746595000ms l2_rsp_data=0 bank_id=0 row_id=6b col_id=0
# time=0.760995000ms l2_rsp_data=d4 bank_id=0 row_id=33 col_id=0
# time=0.761195000ms l2_rsp_data=17 bank_id=0 row_id=34 col_id=0
# time=0.761595000ms l2_rsp_data=78 bank_id=0 row_id=36 col_id=0
# time=0.761795000ms l2_rsp_data=4 bank_id=0 row_id=37 col_id=0
# time=0.765595000ms l2_rsp_data=0 bank_id=0 row_id=6b col_id=0
# time=0.779995000ms l2_rsp_data=d4 bank_id=0 row_id=33 col_id=0
# time=0.780195000ms l2_rsp_data=17 bank_id=0 row_id=34 col_id=0
# time=0.780595000ms l2_rsp_data=78 bank_id=0 row_id=36 col_id=0
# time=0.780795000ms l2_rsp_data=4 bank_id=0 row_id=37 col_id=0
# time=0.784595000ms l2_rsp_data=0 bank_id=0 row_id=2 col_id=0
# time=0.794395000ms l2_rsp_data=d4 bank_id=0 row_id=33 col_id=0
# time=0.794595000ms l2_rsp_data=17 bank_id=0 row_id=34 col_id=0
# time=0.794995000ms l2_rsp_data=78 bank_id=0 row_id=36 col_id=0
# time=0.795195000ms l2_rsp_data=4 bank_id=0 row_id=37 col_id=0
# time=0.800195000ms l2_rsp_data=0 bank_id=0 row_id=50 col_id=0
# time=0.825655000ms l2_rsp_data=a3 bank_id=4 row_id=2 col_id=0
# time=0.825855000ms l2_rsp_data=a2 bank_id=4 row_id=3 col_id=0
# time=0.826255000ms l2_rsp_data=1 bank_id=4 row_id=5 col_id=0
# time=0.826455000ms l2_rsp_data=0 bank_id=4 row_id=6 col_id=0
# time=0.836055000ms l2_rsp_data=4 bank_id=4 row_id=36 col_id=0
# time=0.836255000ms l2_rsp_data=d2 bank_id=4 row_id=37 col_id=0
# time=0.840455000ms l2_rsp_data=0 bank_id=4 row_id=2e col_id=0
# time=125.000025000ms REFRESH DRAM
```

Figure 7: Transcript of Top Level Simulation

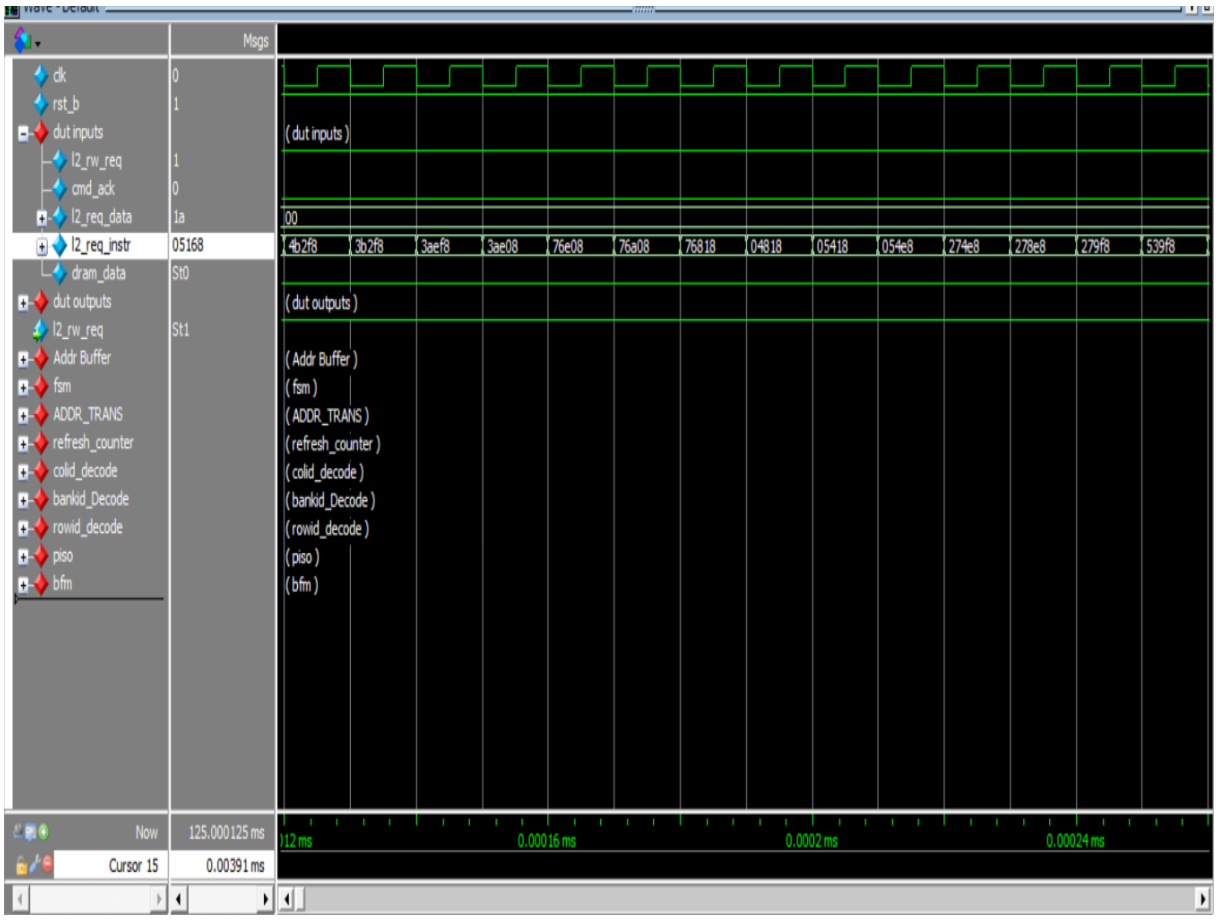

Figure 8: Top Level Simulation- Generating Random Unsigned Numbers for the L2 Write Instructions

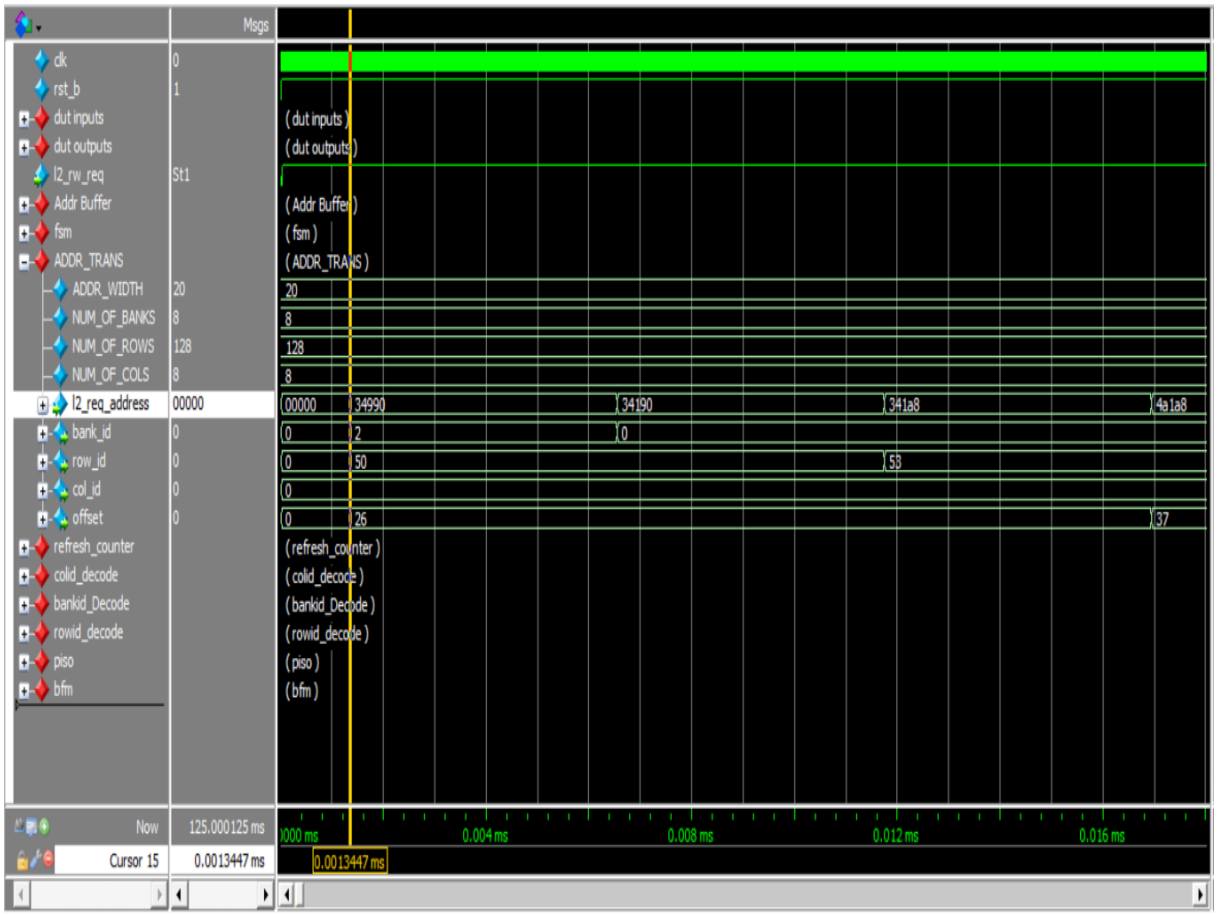

Figure 9: Top Level Simulation- Address Translation of L2 Request Instructions Producing BankID, RowID, ColID

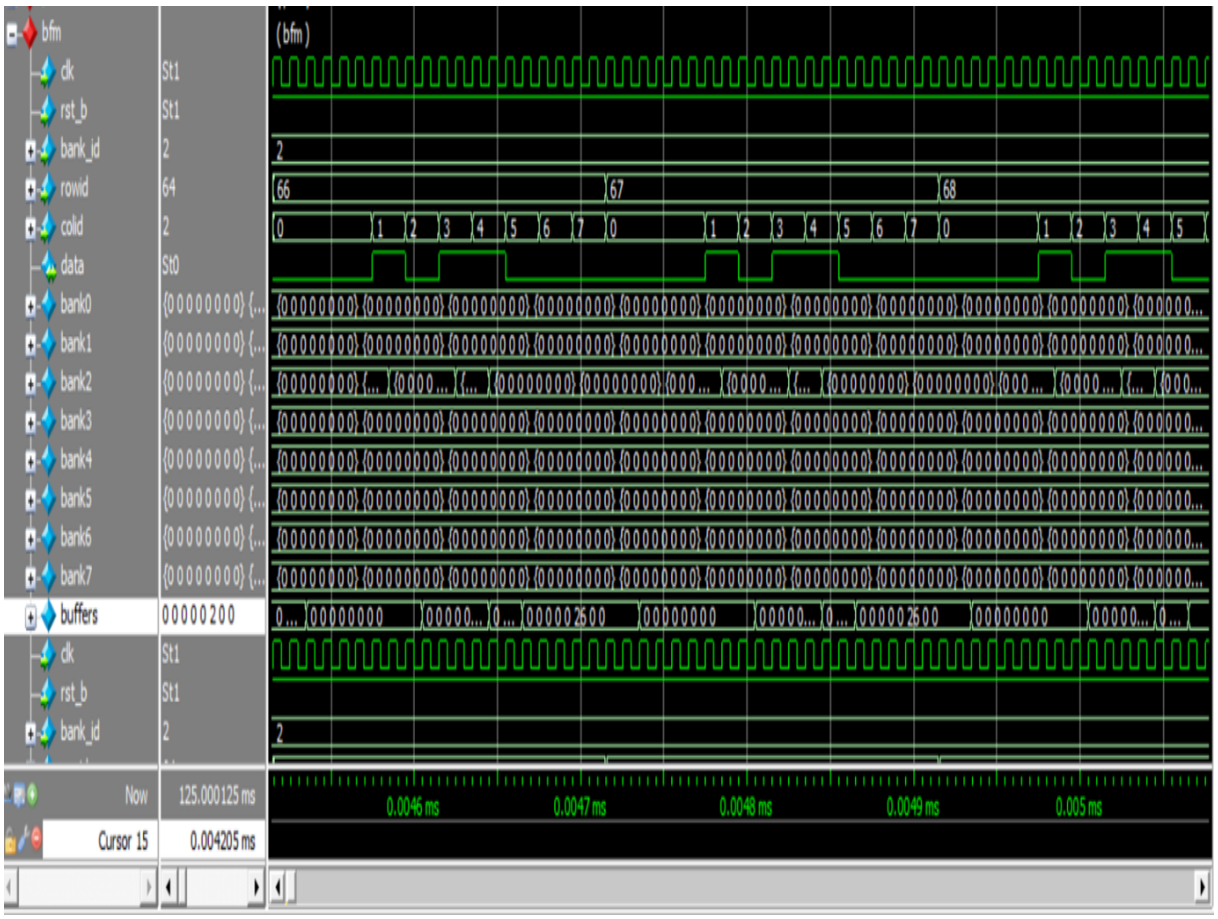

Figure 10: Top Level Simulation- Random Generated Unsigned Data Values Written to DRAM BFM

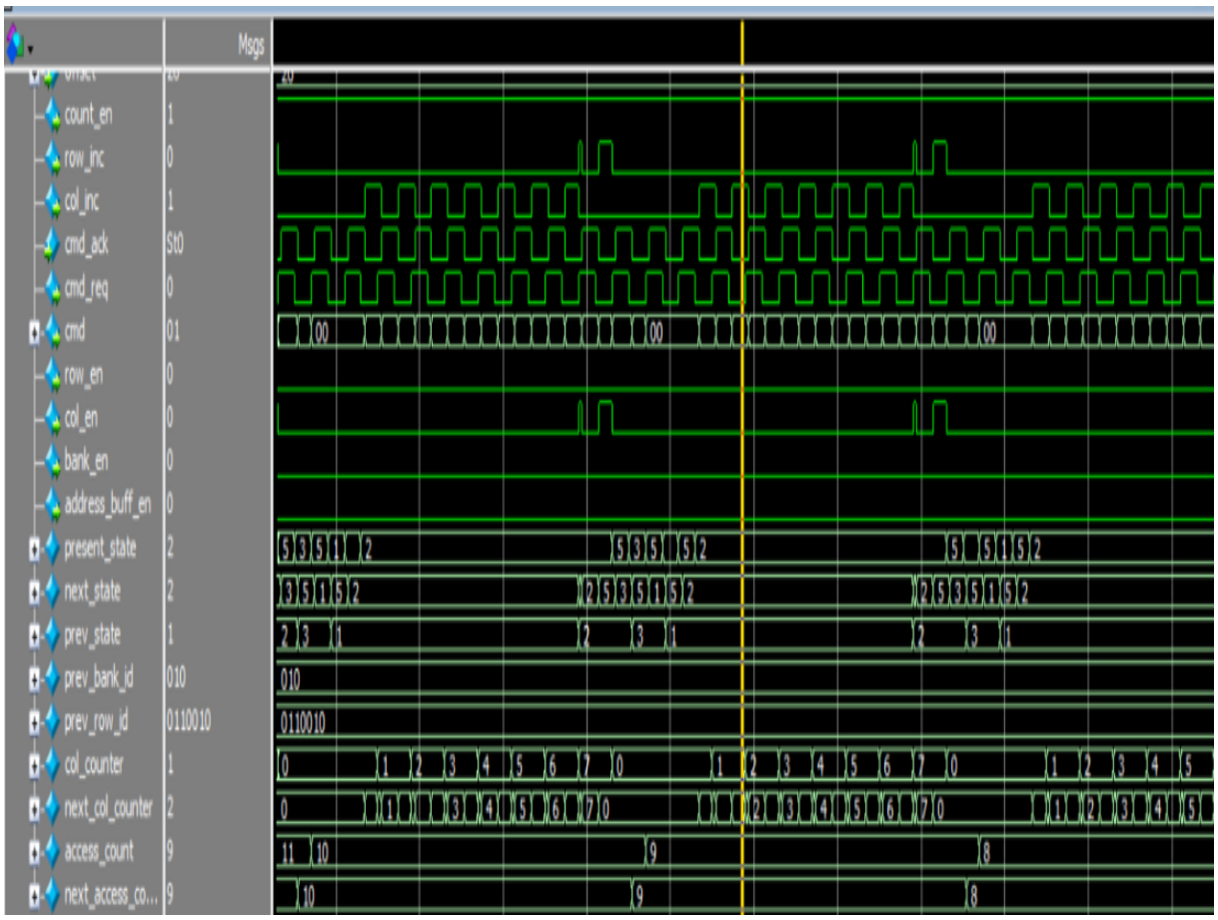

Figure 11: Top Level Simulation – FSM State Transitions during a Single Instruction

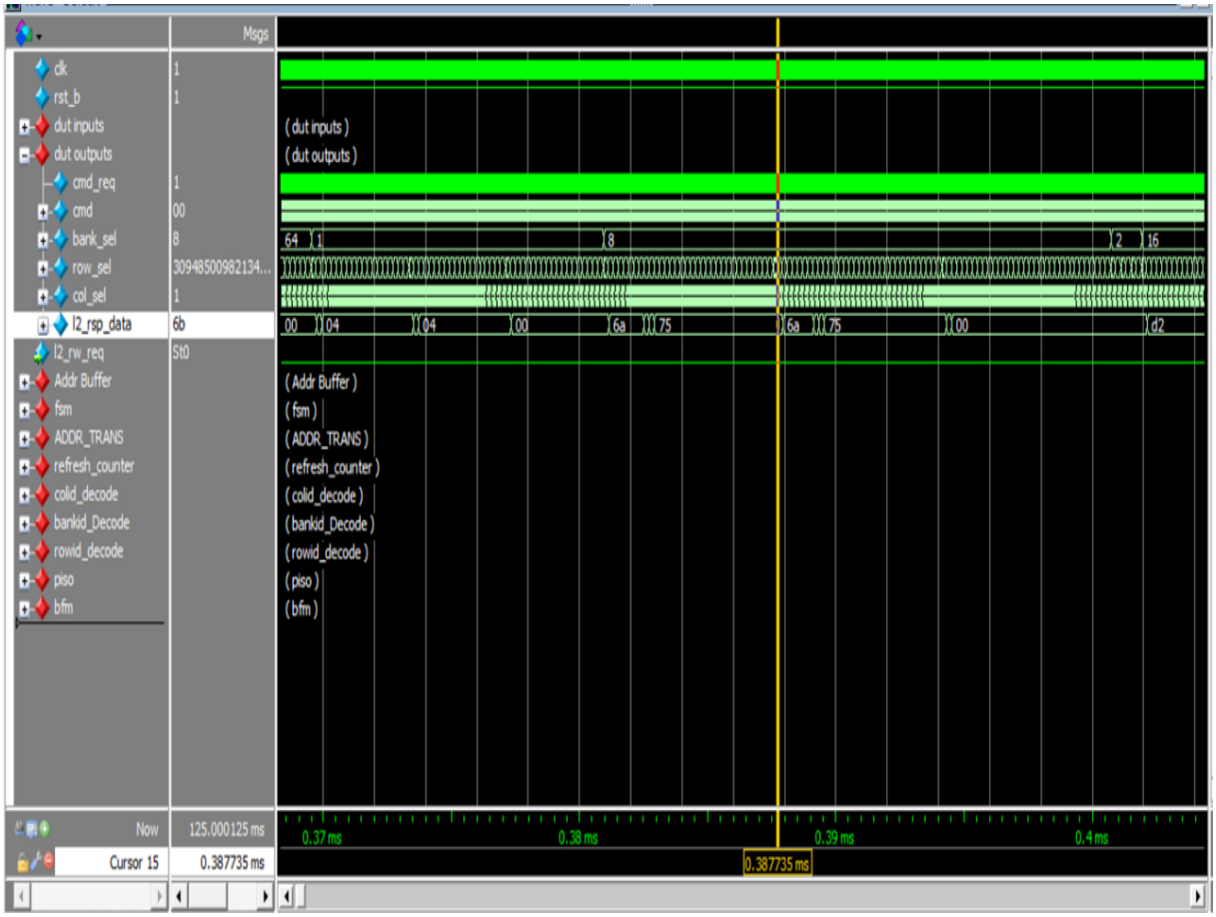

Figure 12: Top Level Simulation-Reading data from DRAM and Populating CPU Bus