ESE 370: CIRCUIT-LEVEL OPTIMIZATION FOR DIGITAL SYSTEMS

Project 2 Milestone: FIFO Queue

Mauricio Mutai, Jack Harkins

Instructor: Dr. Tania Khanna

TA: Martin Deng

Date: 11/26/16

November 27, 2016

Bitline Capacitance

Each SRAM cell loads each bitline with a capacitance of $4\gamma C_0$. This can be seen in Figure 2 below, which shows that BL and BL$'$ are each connected to the drain/source terminal of a transistor of width 4.

Next, each bitline is loaded with at most 16 SRAM cells. This number could be reduced by arranging the words in a 4x4 pattern, for example, but we will consider the worst case capacitance, which arises with 16 cells. Thus, each bitline is loaded with $64\gamma C_0$ from the SRAM cells.

In addition, each bitline is loaded with $16\gamma C_0$ from the driver circuit, and $16\gamma C_0$ from the tri-state buffer or inverter. Therefore, the total load on the bitline is $96\gamma C_0$.

Circuit Schematics

Memory Column Driver Schematic

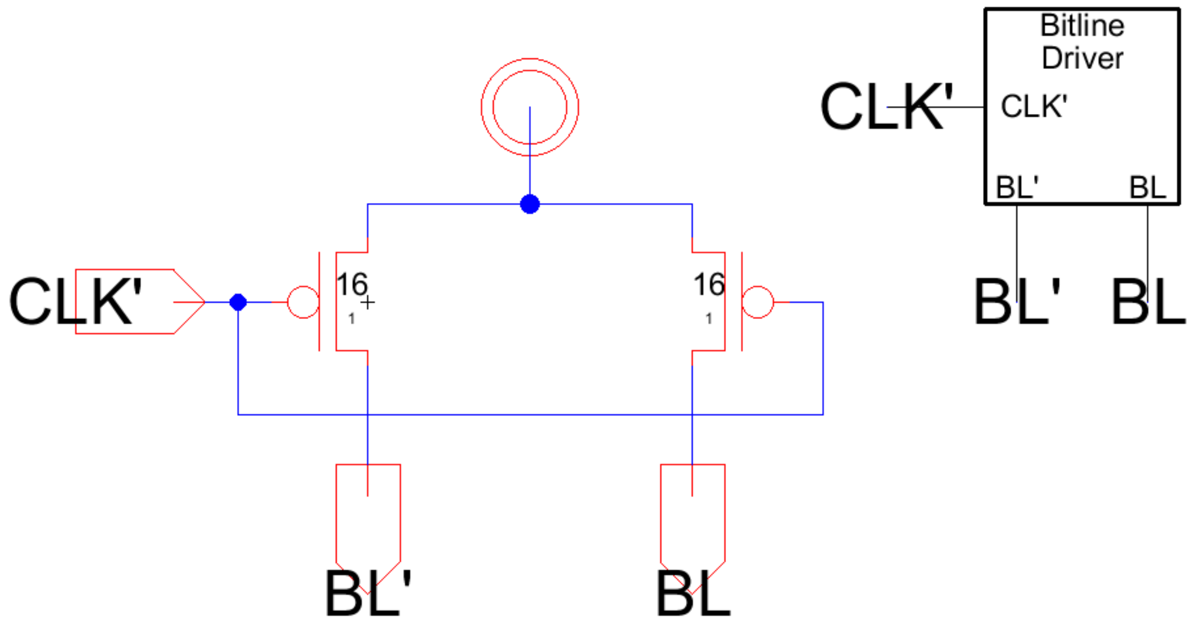

Figure 1: Bitline driver circuit

Sized Memory Cell Schematic

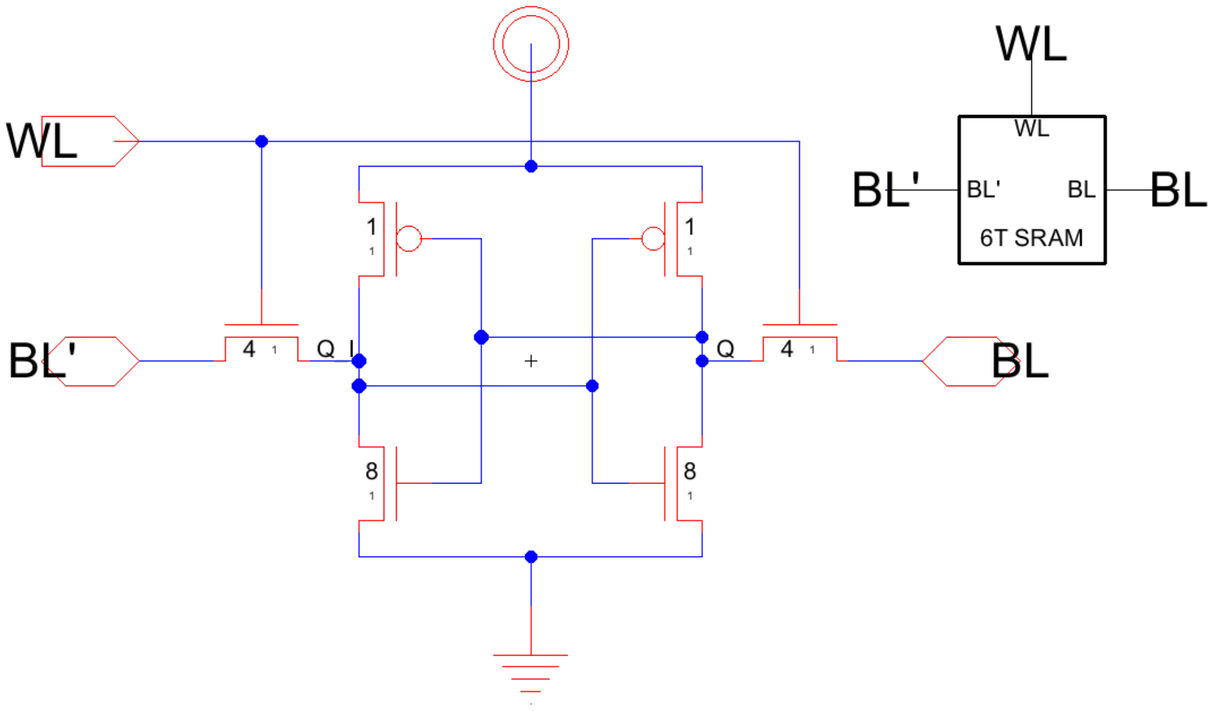

Figure 2: Sized SRAM memory cell circuit

Tri-State Buffer Schematic

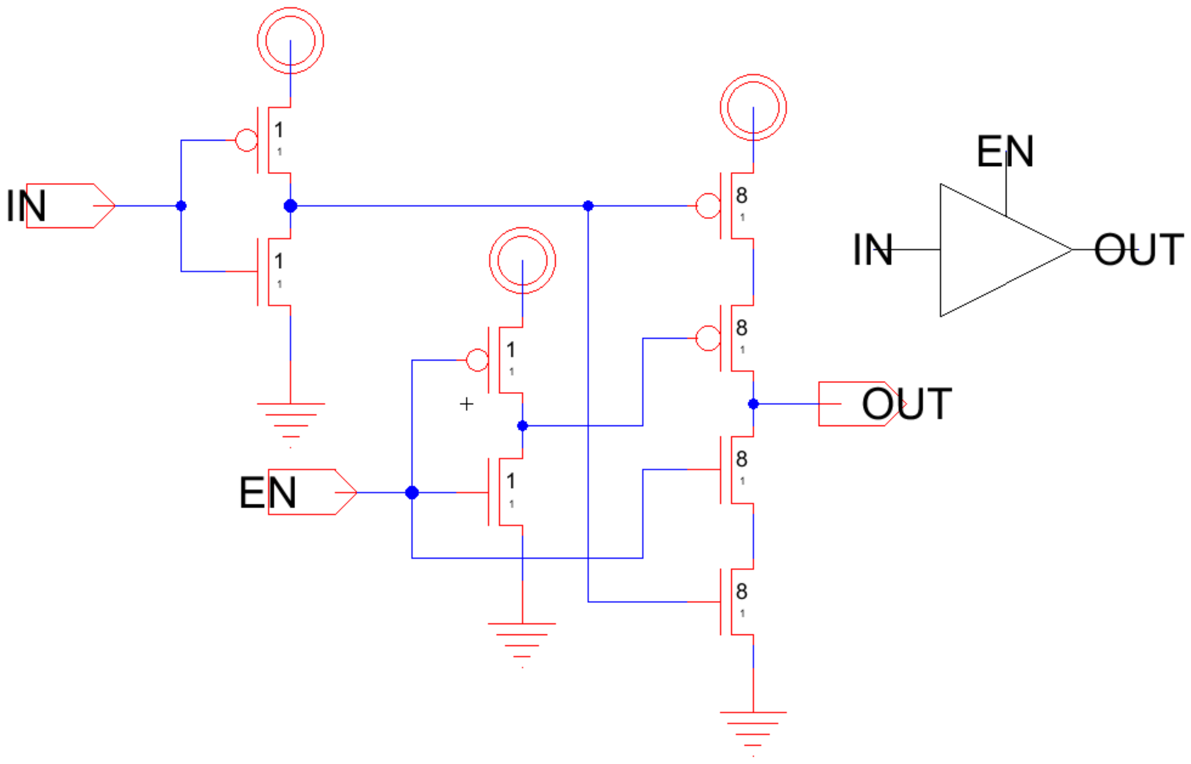

Figure 3: Tri-state buffer circuit

Tri-State Inverter Schematic

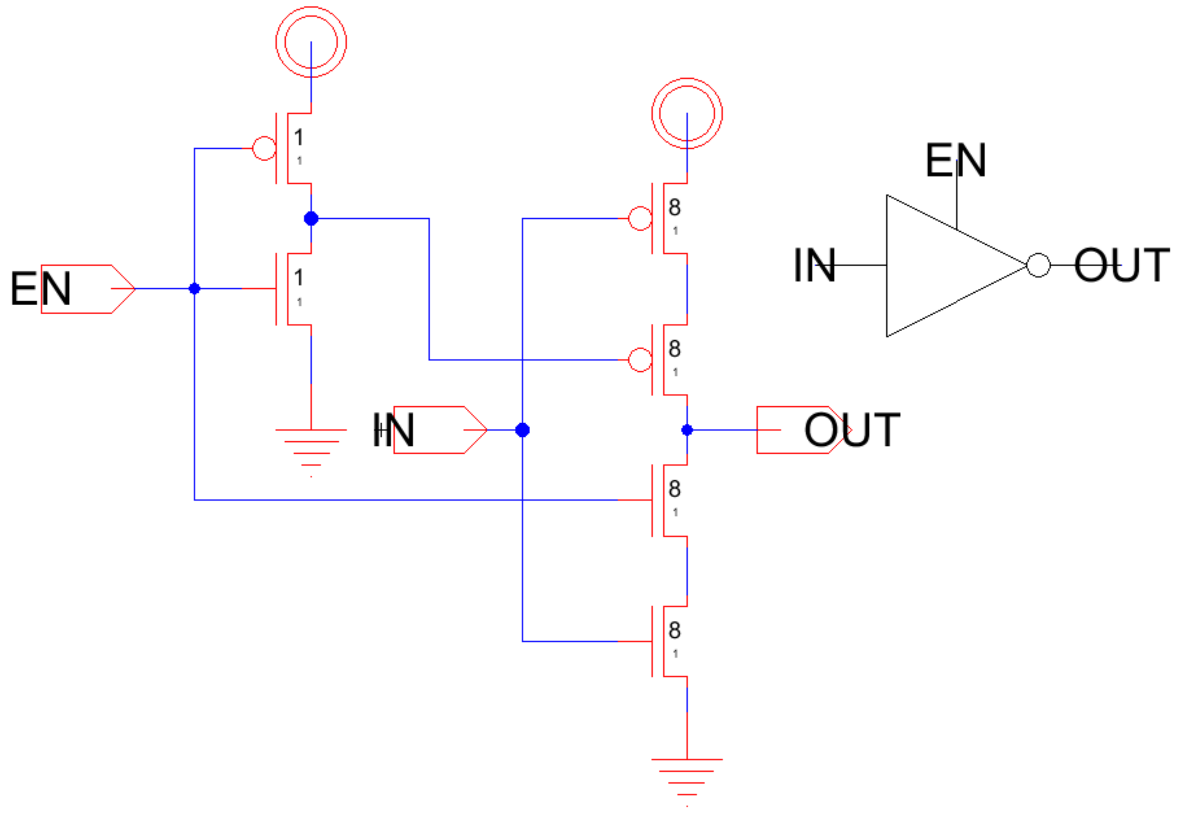

Figure 4: Tri-state inverter circuit

Reading and Writing (Cell)

Test Cases to Consider

For writing to the cell, we considered four cases:

1. Writing a 0 to a cell with 0 (Figure 11). We first write a 0 by driving BL low and BL′ high by driving $write\_enable$ high, and driving WL to high. We then write a 0 again using the same process. The Q and Q′ signals inside the cell were verified to ensure that writes were being completed properly, even after $write_enable$ was set low.

2. Writing a 1 to a cell with 0 (Figure 10). This uses the same process for writing a 1 as for writing a 0, but BL is driven high and BL′ is driven low.

3. Writing a 0 to a cell with 1 (Figure 8). Since the cell already starts in a state in which Q is high and Q′ is low, we only need to write a 1 to verify this case.

4. Writing a 1 to a cell with 1 (Figure 9). This is the same case as (3) in that the cell already starts with $Q = 1$, but we instead write a 1.

Since there are two present states ($Q_t = 0$, $Q_t = 1$) and two next states ($Q_{t+1} = 0$, $Q_{t+1} = 1$), these four cases completely test writing to the cell.

For reading from the cell, we considered two cases:

1. Reading a 0 from a cell that was written to 0 (Figure 6).
2. Reading a 1 from a cell that was written to 1 (Figure 7).

In each reading case, we made sure that the reading process did not cause enough of a read upset to overwrite Q. These read upsets are not visible in Figures 6 or 7 and were on the order of tens of millivolts, so we are confident that reading from these cells is not destructive.

All testing was done with the test circuit in Figure 5. Note that this test circuit includes only one SRAM cell to keep the image visible. Actual testing was done on one SRAM cell with 15 other SRAM cells loading the bitlines in parallel to simulate how the memory circuit would behave in the actual 16-word FIFO queue.

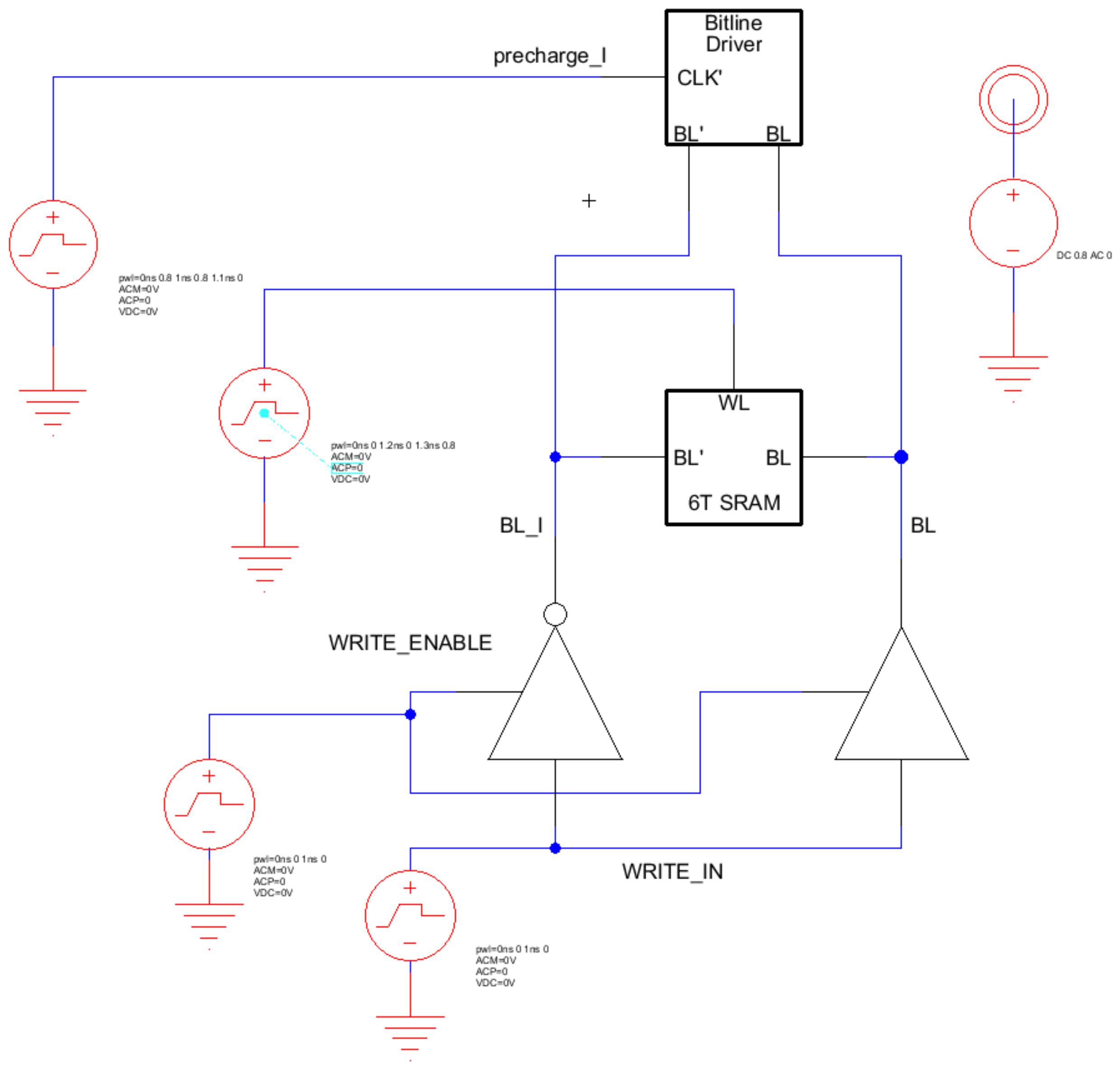

Figure 5: Memory cell test circuit

SRAM Cell Test Results

Note: in the figures below, `net@16` is the word line (*WL*) signal.

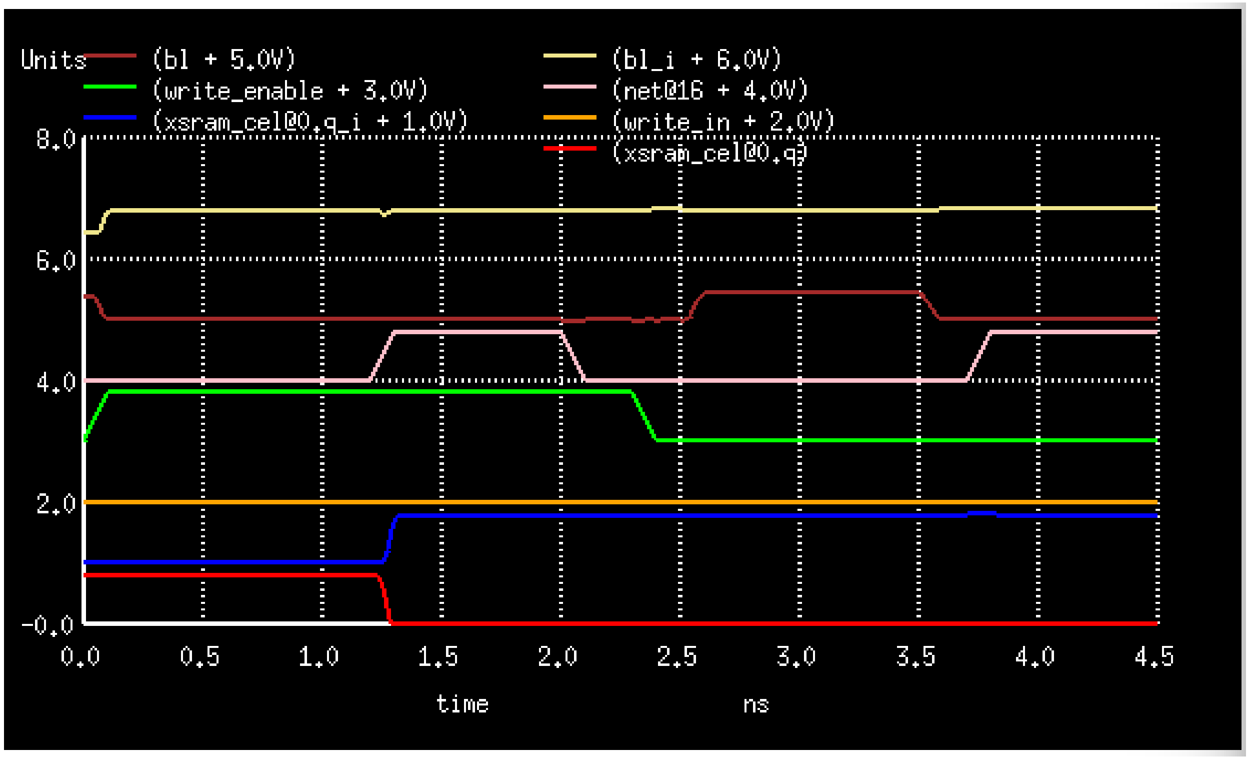

Figure 6: Writing 0 and then reading 0

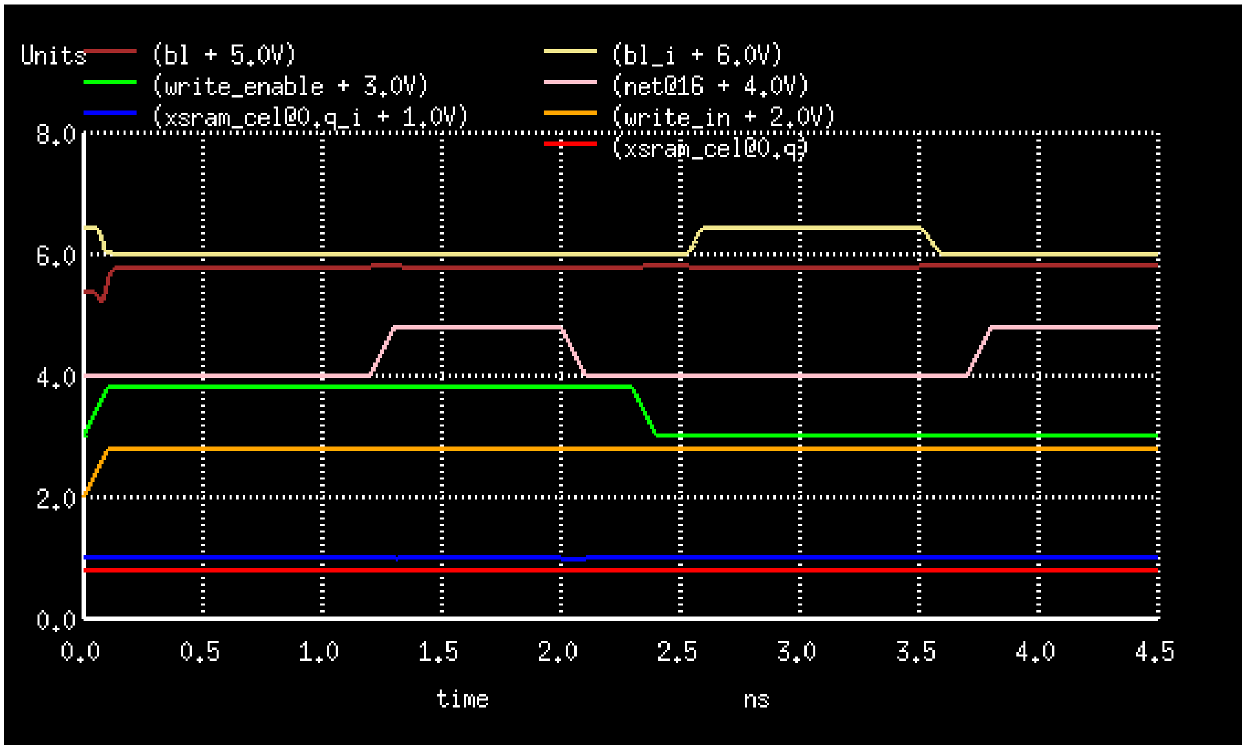

Figure 7: Writing 1 and then reading 1

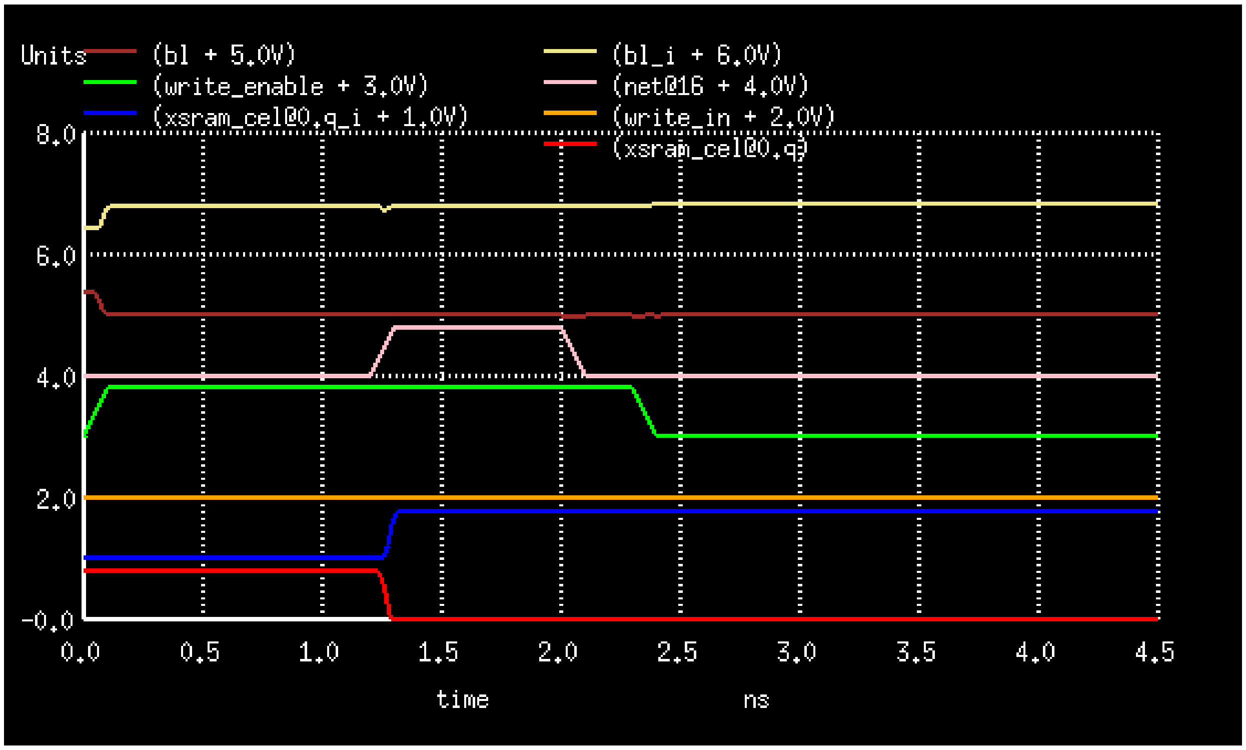

Figure 8: Writing 0

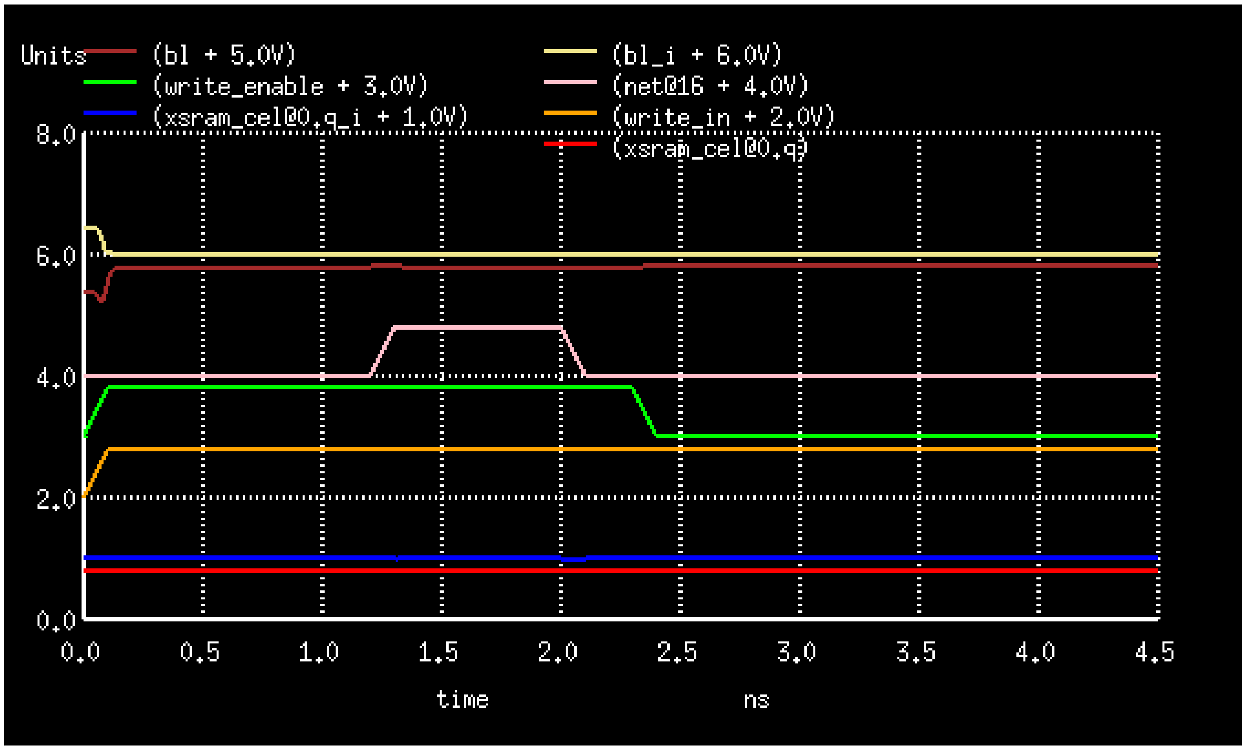

Figure 9: Writing 1

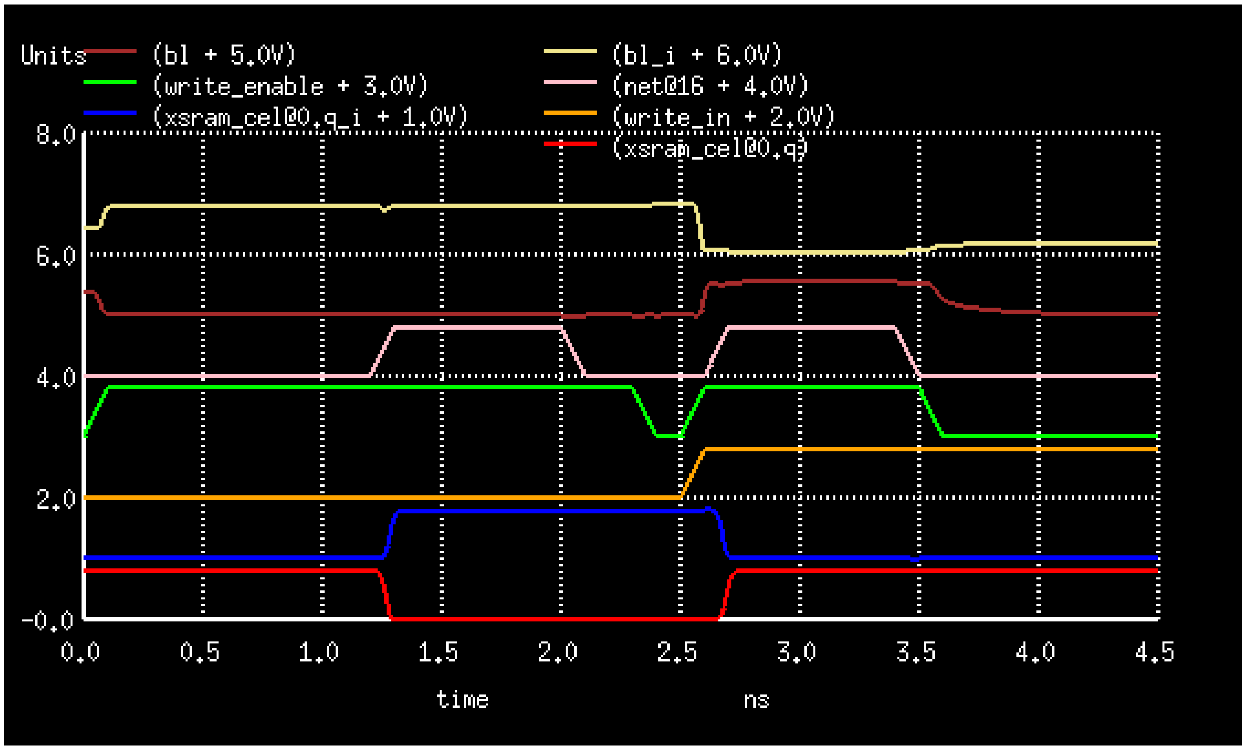

Figure 10: Writing 0, then writing 1

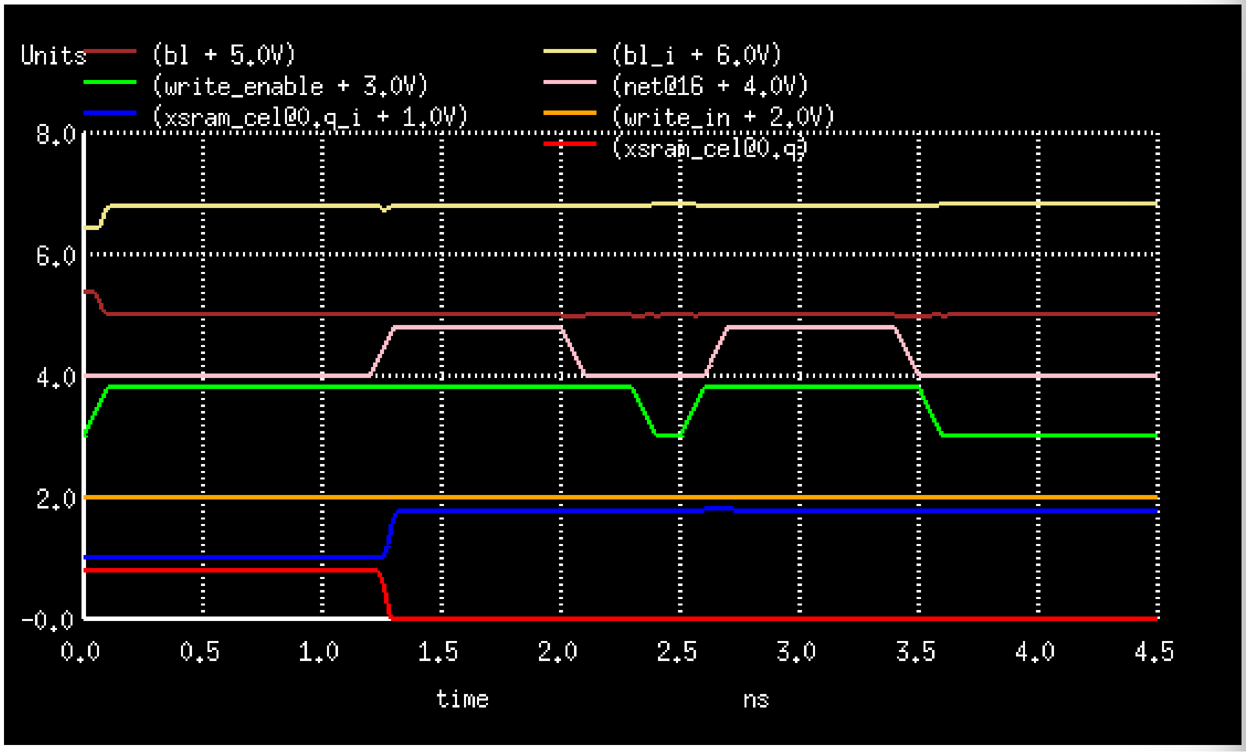

Figure 11: Writing 0, then writing 0

Constraints on Write Timing, Full FIFO Design

One important constraint for write timing is if we are simultaneously enqueuing and dequeuing from the same word at the same time. To ensure that the dequeue operation returns the correct value to the output, and because a cell can only be written to or read in a single clock cycle, we will want to make sure that we dequeue the old value first before enqueueing the new value. To test this, we will first enqueue a single element (suppose it's ”0011”), and then we'll issue both an enqueue (say, of ”1100”) and a dequeue at the same time. The value returned by dequeue should be ”0011”.

Another edge case we must consider is if we are simultaneously enqueuing and dequeuing when the queue is full. Assuming that we dequeue first before enqueueing, we will need to make sure that we save the fact that the queue was full into a latch before we proceed with the enqueue operation. This will be done to guarantee that simultaneous enqueues and dequeues on a full queue will ensure that the queue has one fewer element afterward. To verify this, we can examine the values of the read and write pointers (counters) to ensure that the queue has exactly 15 elements and not 16.