PROJECT 2:
MEMORY DESIGN

Instructor: Tania Khanna
Class: ESE 370
Students: Celine Lee, Ransford Antwi

Table of Contents

Introduction

In this project, we build a memory queue of 32 4-bit words using SRAM memory cells, multiplexers/decoders, registers, sense amplifiers, bit conditioning circuitry, clock generation, etc. Figure 1 shows the general layout for the design.

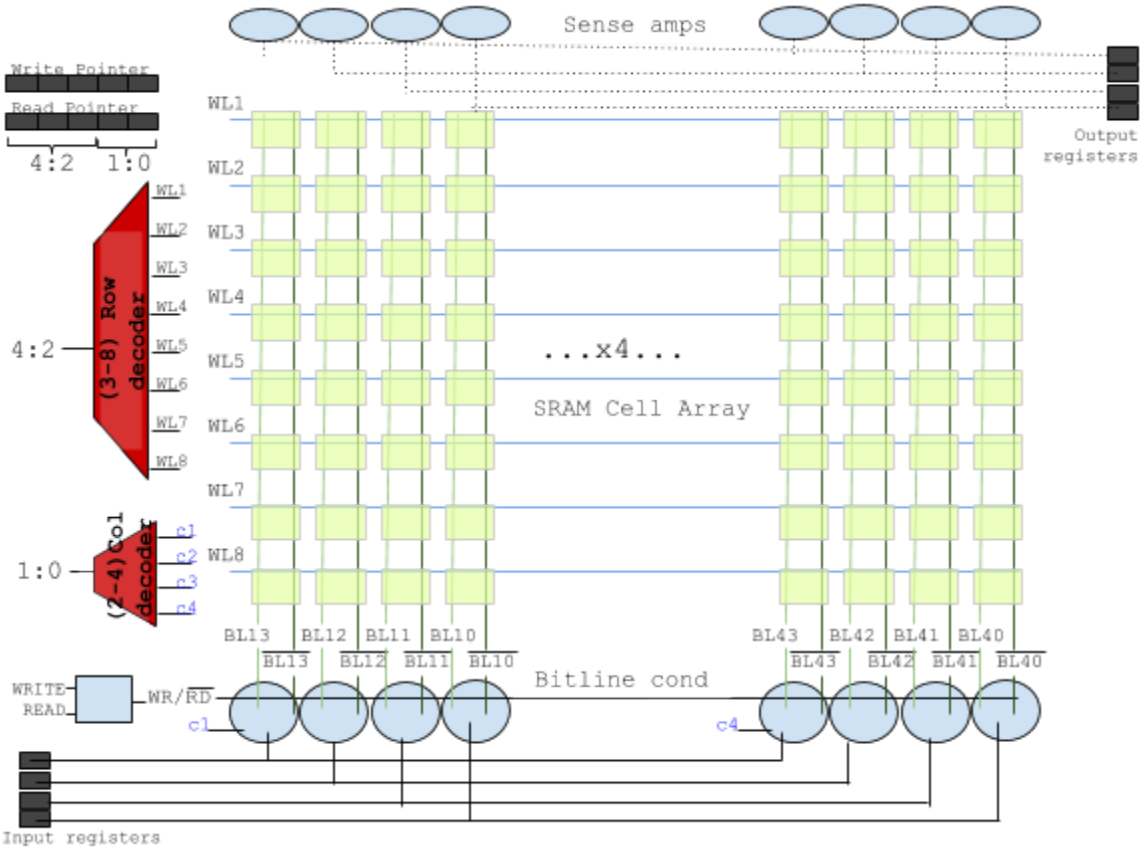


Figure. General layout of memory design.

The objective is to properly create and run the FIFO queue at 500MHz while minimizing energy usage.

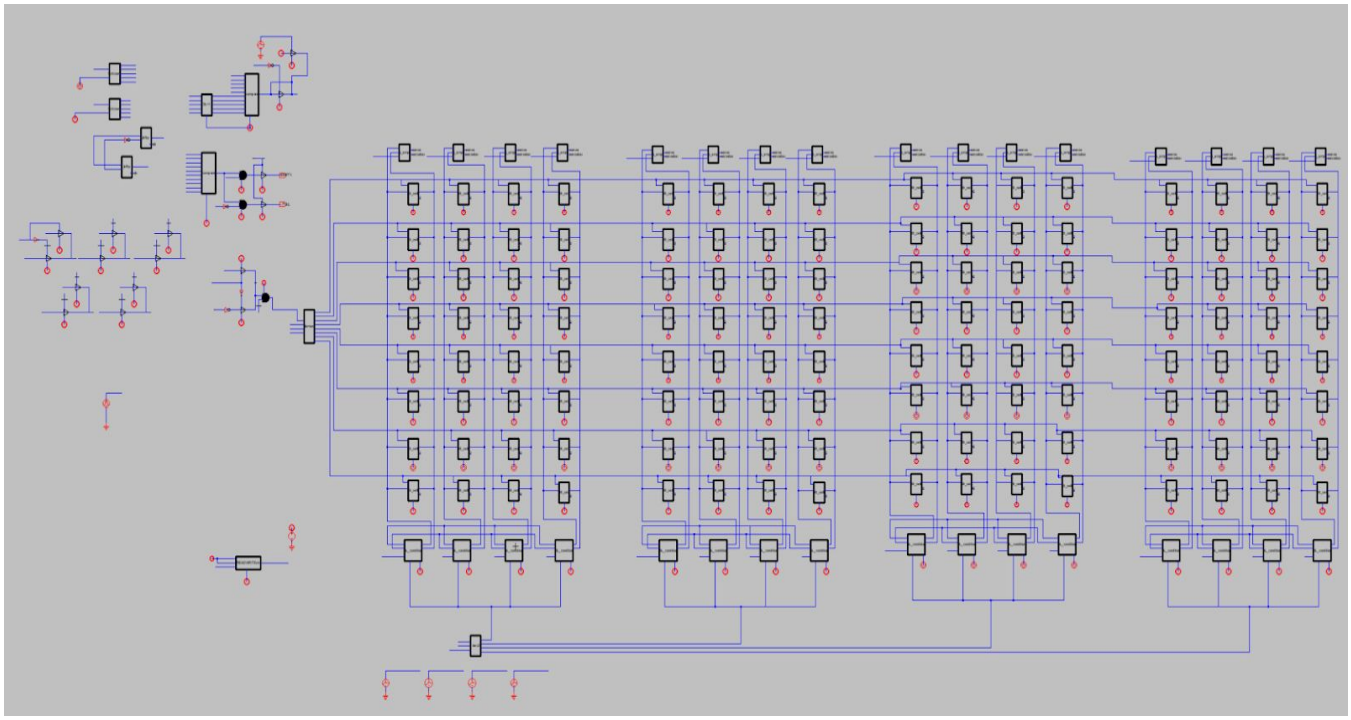


Figure. Schematic of final memory. Subcomponents are detailed throughout this report.

We opted to operate at a Vdd of 0.8V in order to reduce energy usage.

SRAM Cell Design

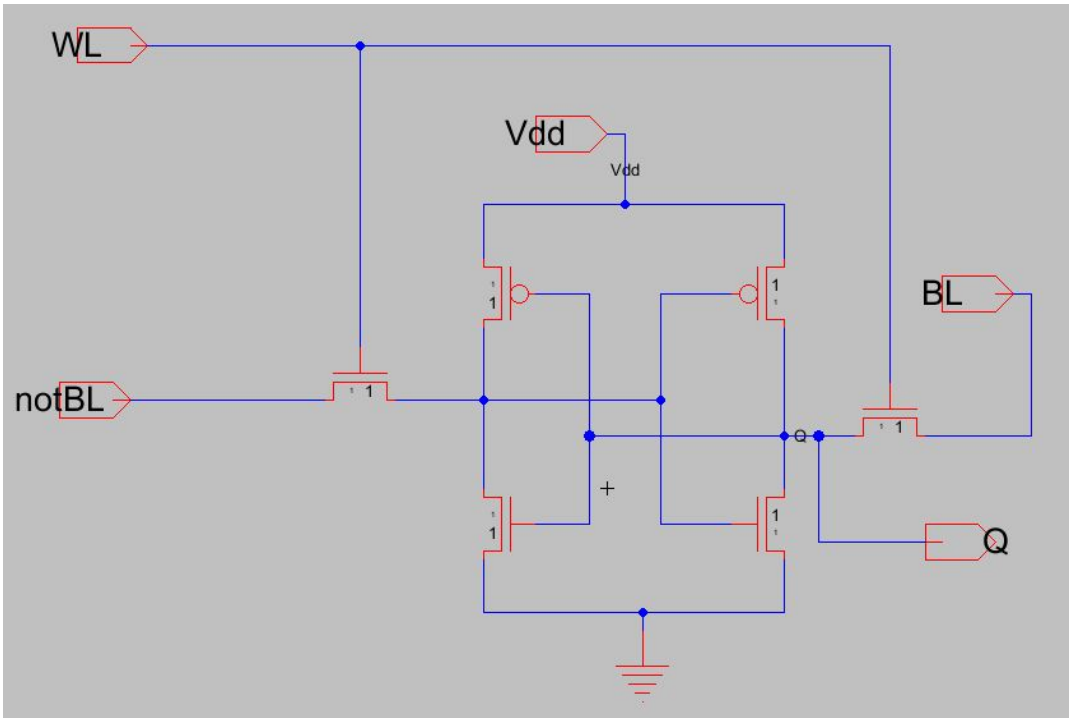


Fig. Memory cell design using 6 transistors.

To perform a read operation, initially the memory should have some value which is the value being read. This value is stored at the intermediate node q shown in Fig1. To simulate a read operation, we considered two cases q=1 and q= 0. To test read, we first precharge both bit lines to high, then turn on the word line, and read the value held at point Q. If q is low then the value read should be low and if q is high then the value read should be high. The bit-lines serve as the outputs. The first case is shown below, where the internal node q was set to high:

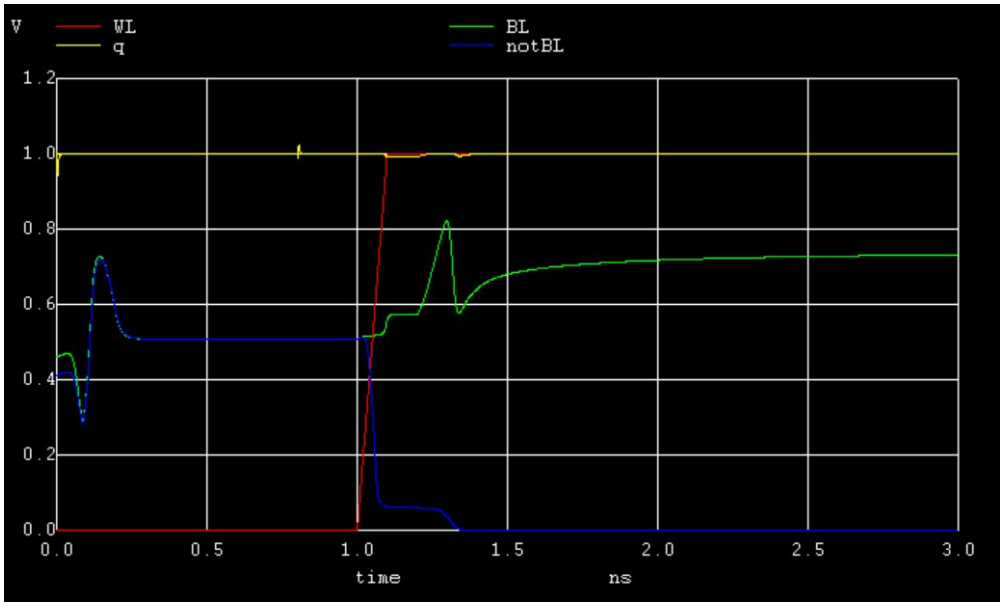


Figure. Read HIGH.

The second case shown below is when node q is low.

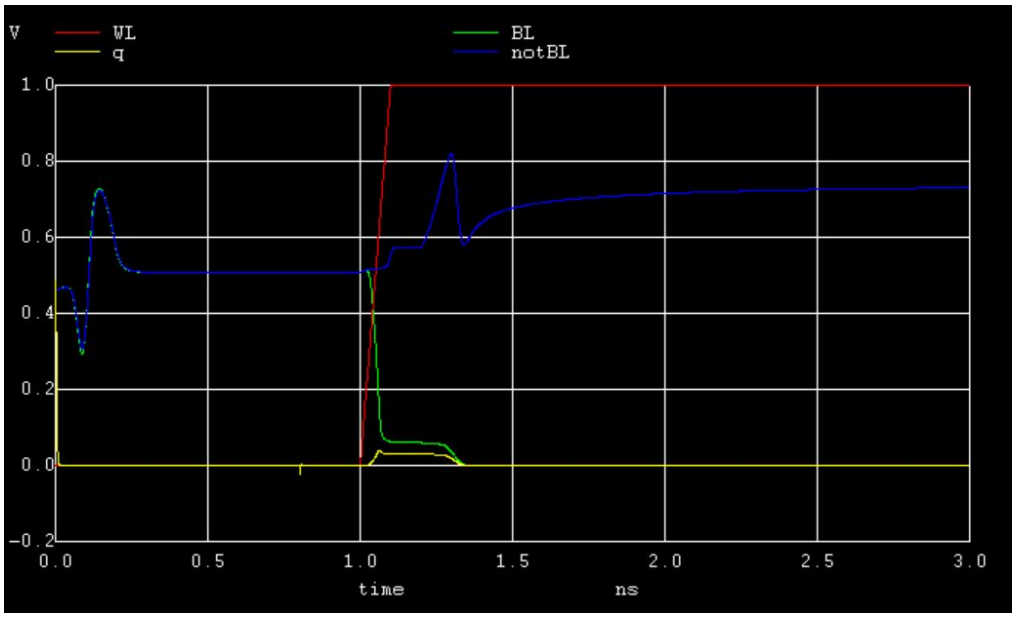


Figure. Read LOW.

For the write operation, the bit-lines serve as the inputs and the word-line is held high. To test write, we set the bit lines to their appropriate value, then send the word line high. We can keep track of the value at q, to see the success of the write operation.

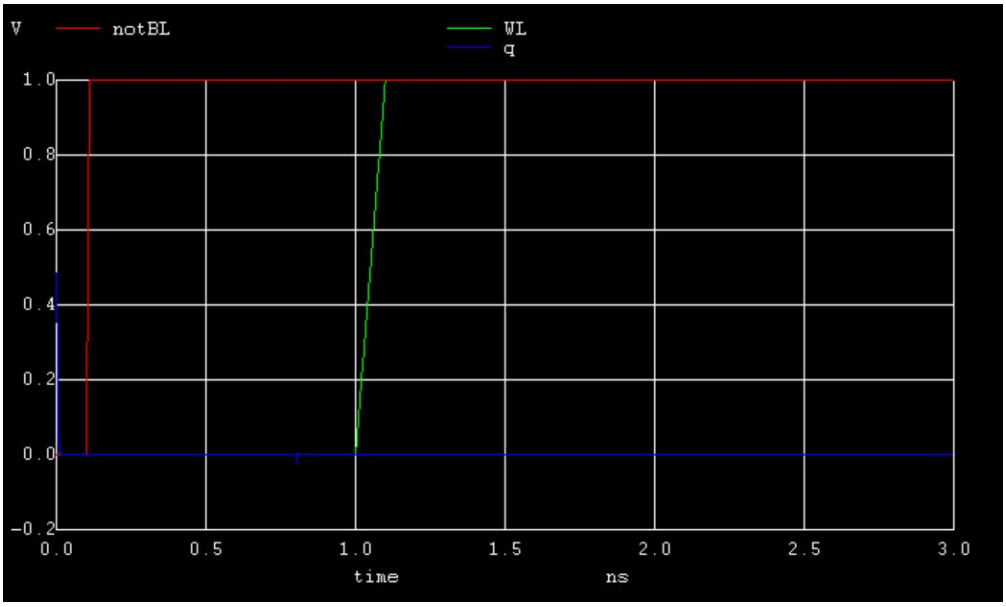


Figure. Write q (which is preset to 0) as 0.

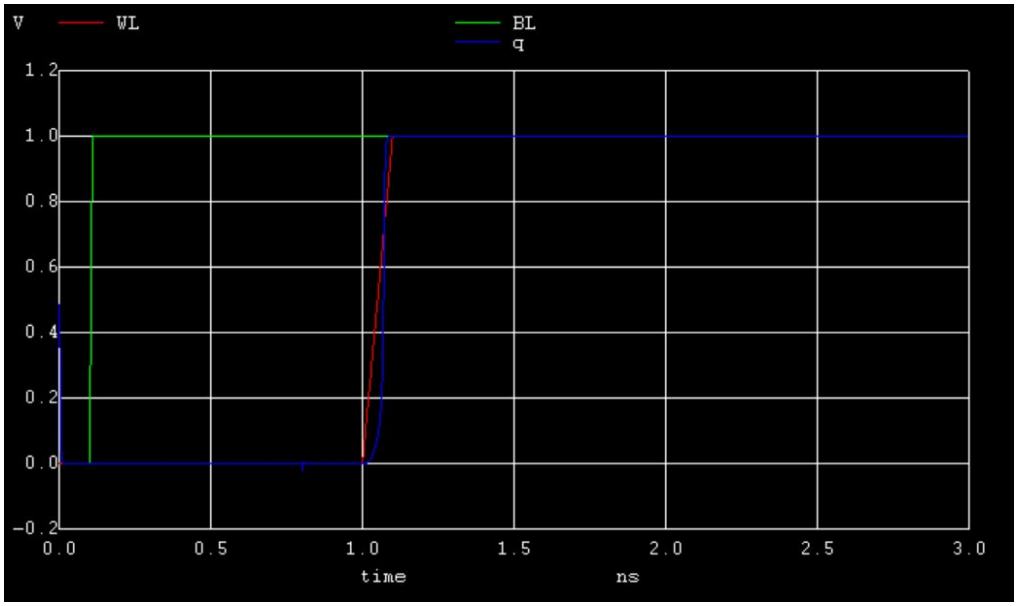


Figure. Write q (which is preset to 0) as 1.

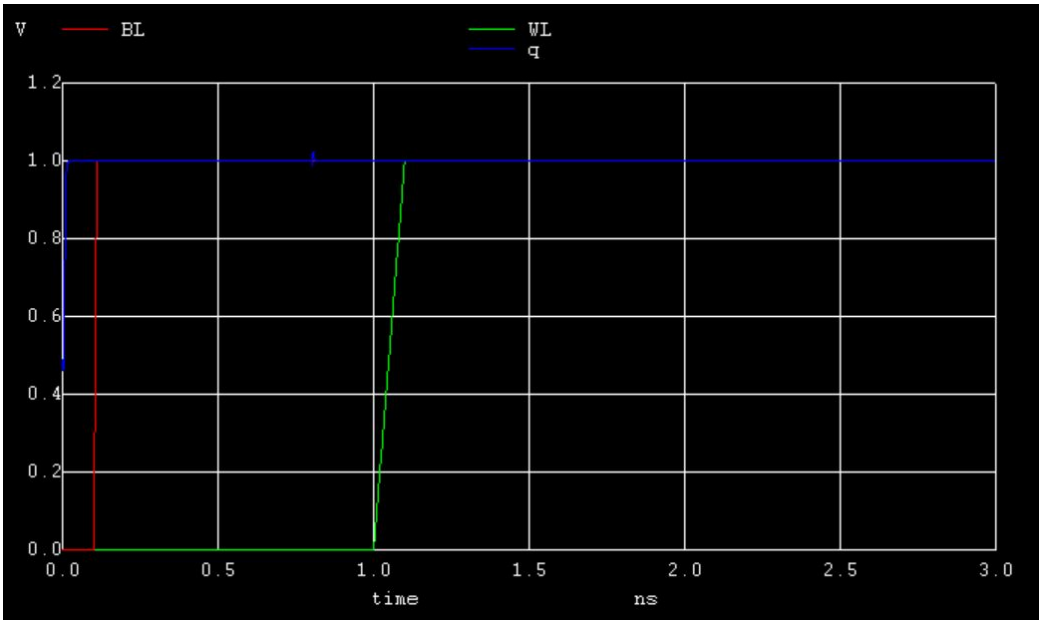


Figure. Write q (which is preset to 1) as 1.

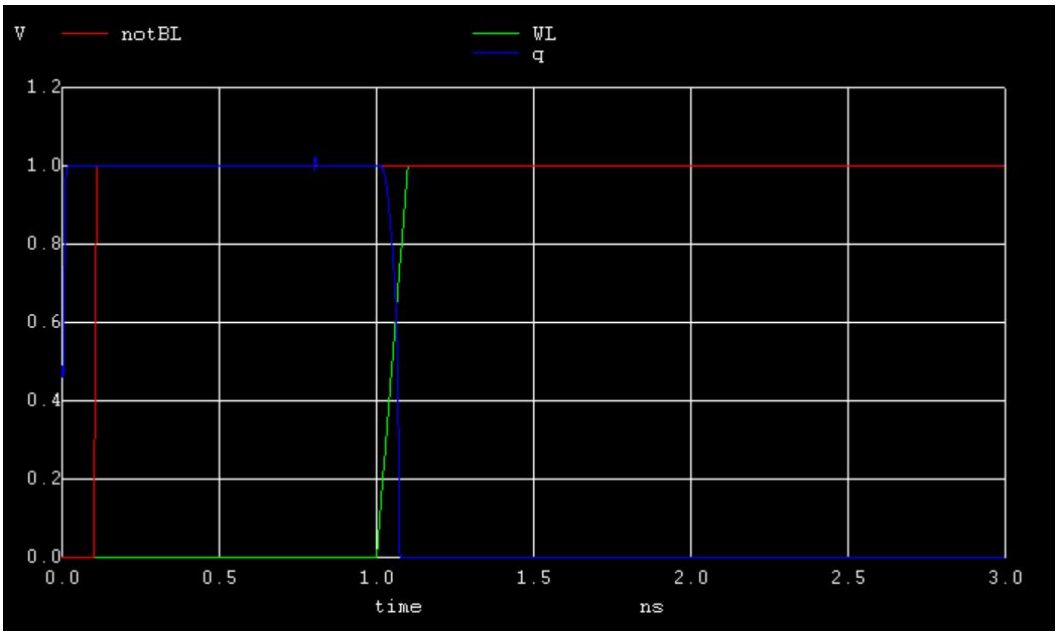


Figure. Write q (which is preset to 1) as 0.

Now we want to test a procedure of reads and writes. The test case we will run is: **(1) write (2) read (3) read.** This input will be to ensure that the cell is capable both of writing data, and of reading without causing a data upset. The schematic below shows the test bench used. The bit-lines were pre charged using the tri-state buffers. For the write operation, we wrote a 1 into the cell and bit line was charged to a 1 one whilst bit-line bar stays low. This ensures that a 1 is written into the cell, and then in the next 2 cycles of the word line we read the cell again to ensure that a 1 was still held in the cell.

Registers:

We used a gate-based latch to build the register:

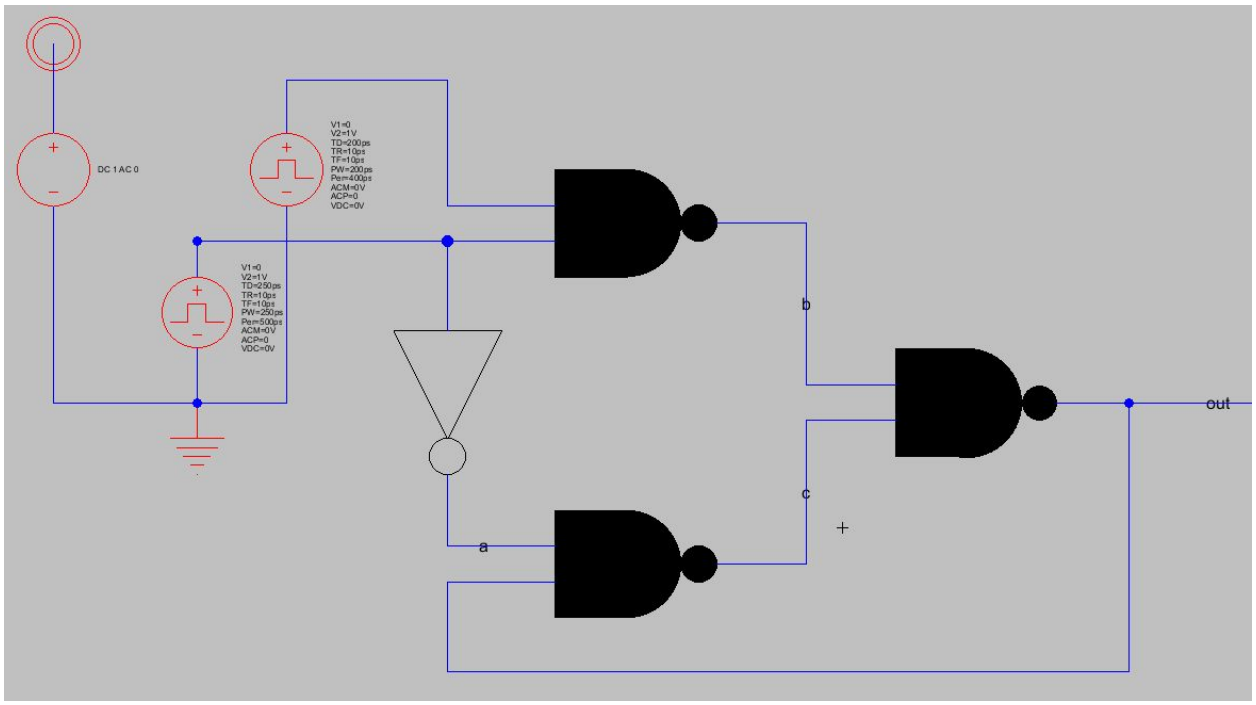


Figure. Schematic of gate-based latch.

How the latch operates: When clock (net@1) is high, **out** follows **in (net@30)**. when clock low, latch remains at whatever **in** was on clock's falling edge:

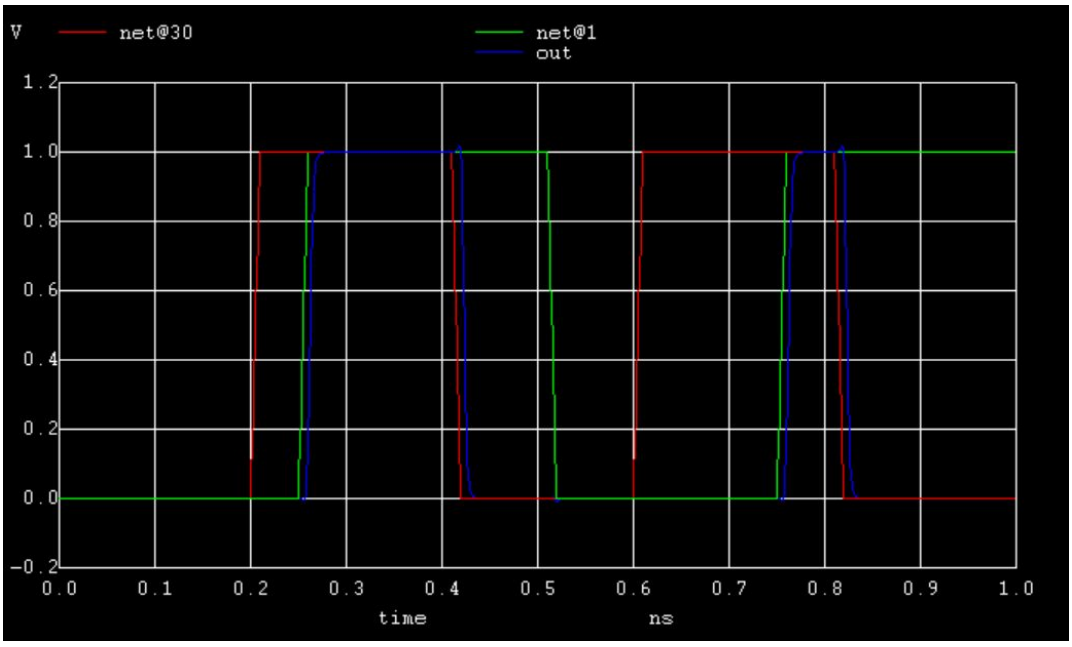


Figure. Simulation of latch operation.

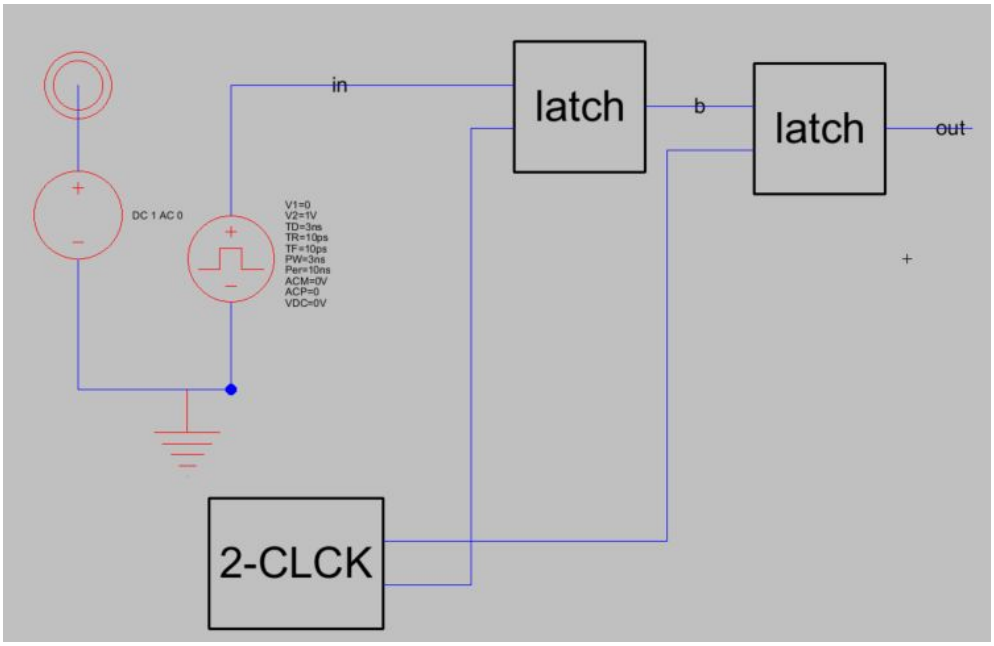


Figure. Schematic of register built from two latches and a two-phase clock generator.

When the clock put into the first latch is high, **b** follows whatever is given to **in**. When the clock put into the second latch is high (i.e. the first latch's clock is low), **out** follows what **b** was, which is locked from whatever **in** was at the time of the first latch's clock falling edge. Therefore, **out** only switches on the first latch's clock's falling edge, and it switches to whatever **in** was at the time of falling edge.

Bitline Conditioner

First, consider the tri-state buffer used throughout the circuit.

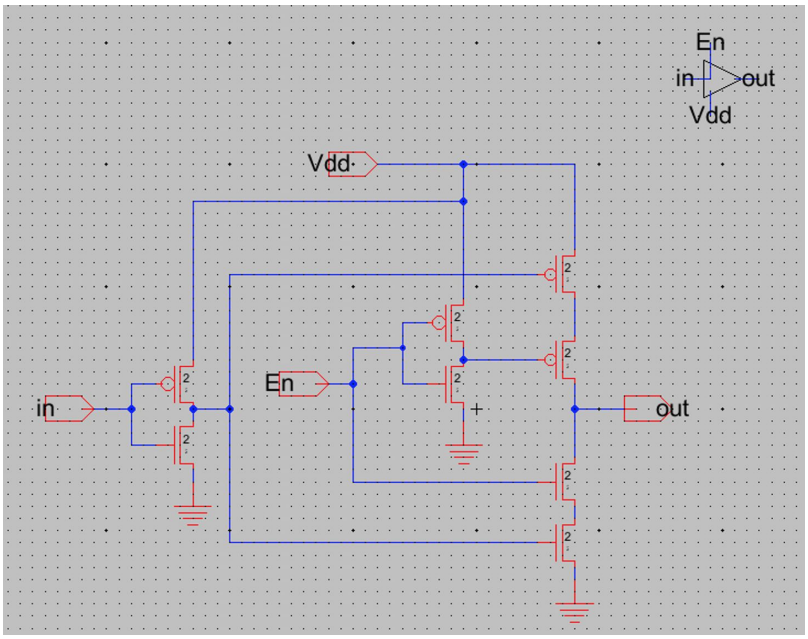


Figure. Tristate buffer

Validating logical correctness of tristate buffer:

In	En	Output
0	0	Z
1	0	Z
0	1	0
1	1	1

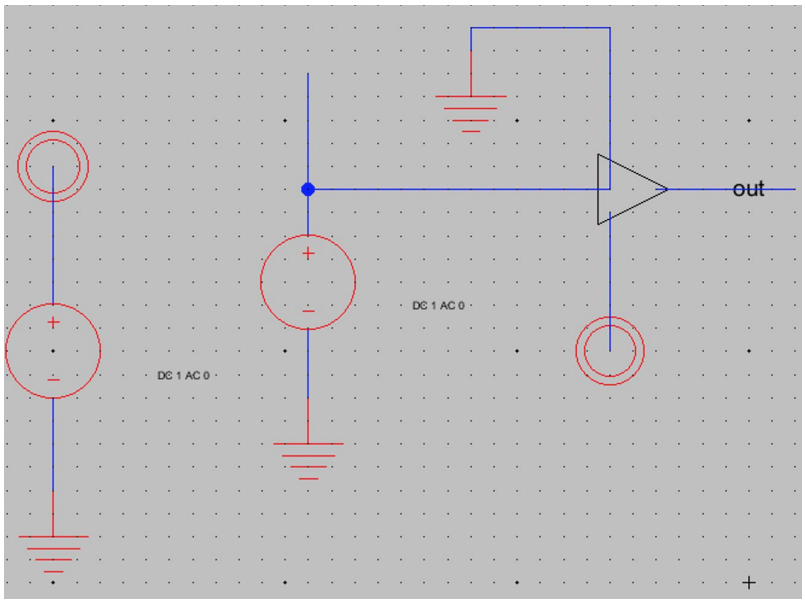


Fig. Testing the tristate buffer.

For the buffer, we only care about what happens when the write enable is high. The simulation below shows the result of simulating the above test bench. The write enable was set to high and the input was swept from high to low. The results obtained was consistent with the expected output from the truth table.

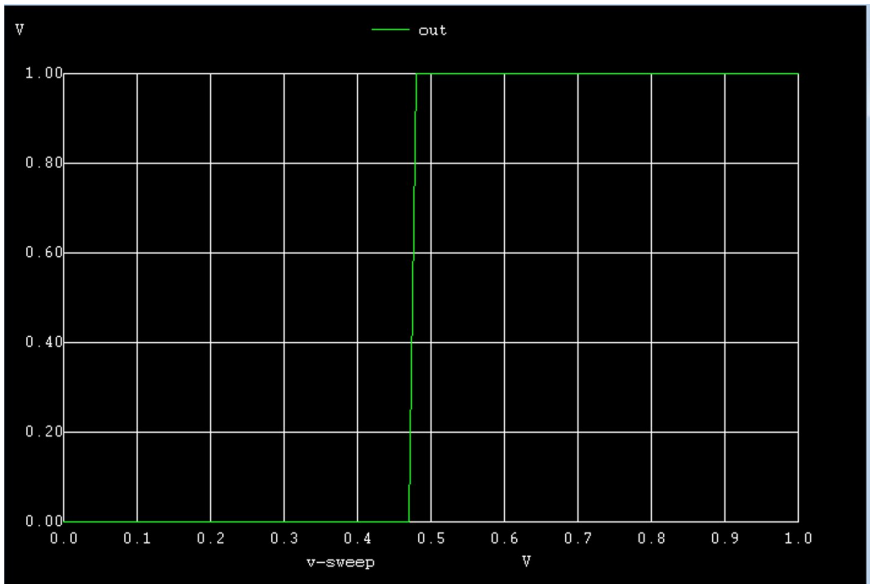


Fig. Simulation results of testing the tristate buffer.

Now consider the midpoint generator used throughout the circuit:

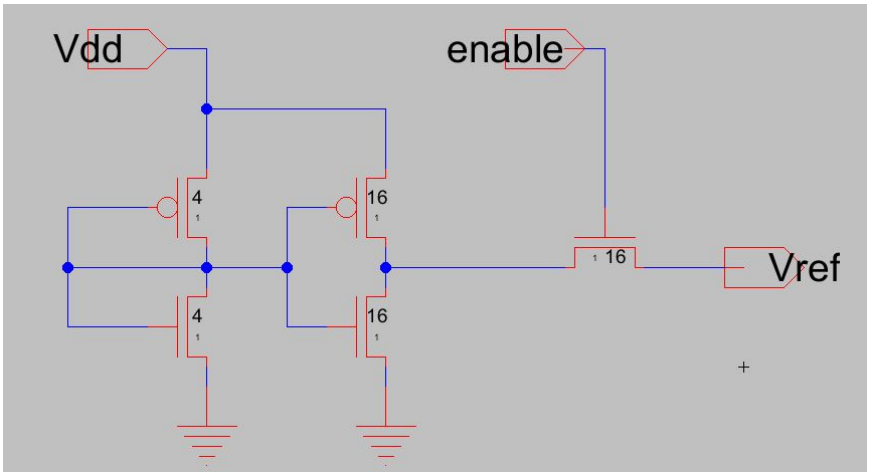


Figure. Schematic of midpoint generator. It produces a Vref ~ Vdd/2

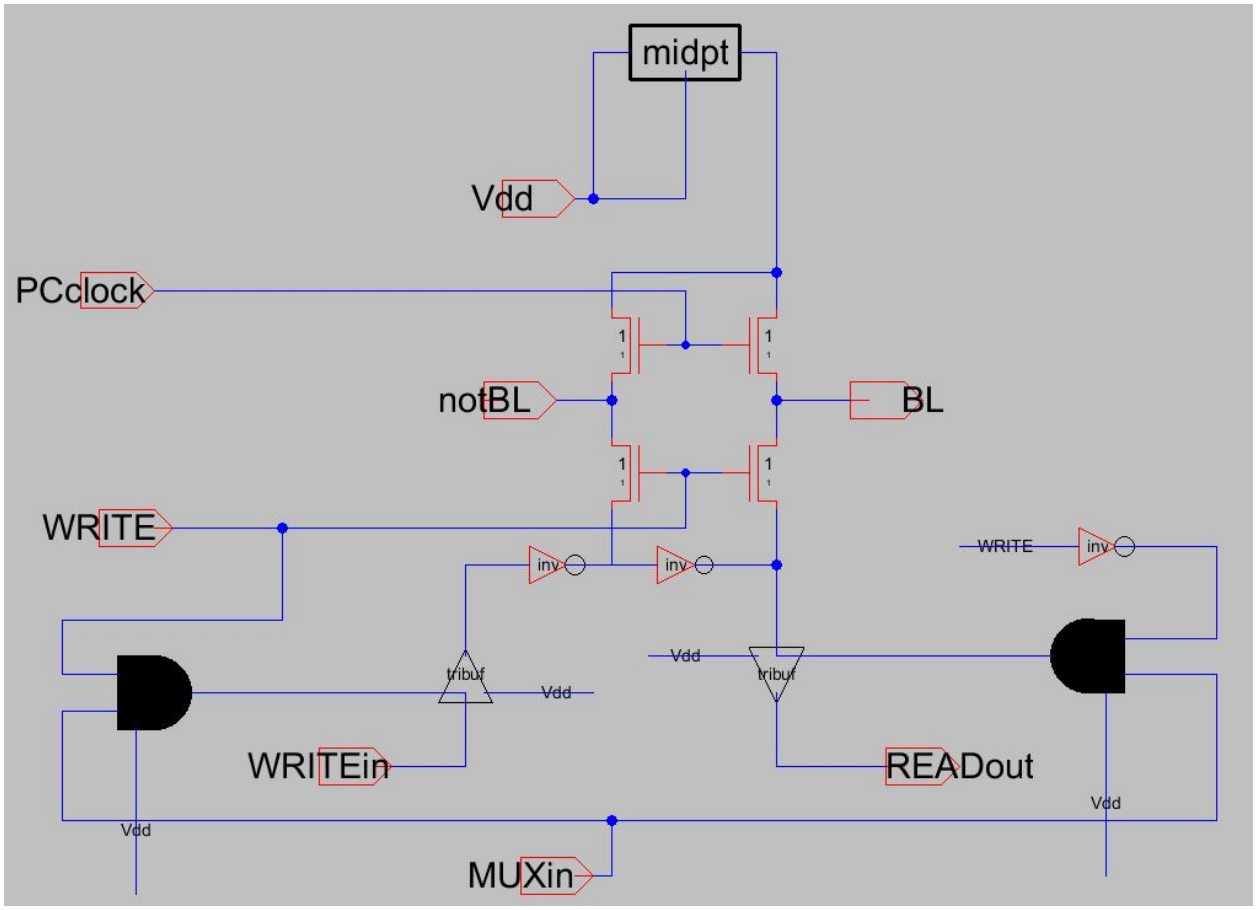


Figure. Bitline conditioner circuit.

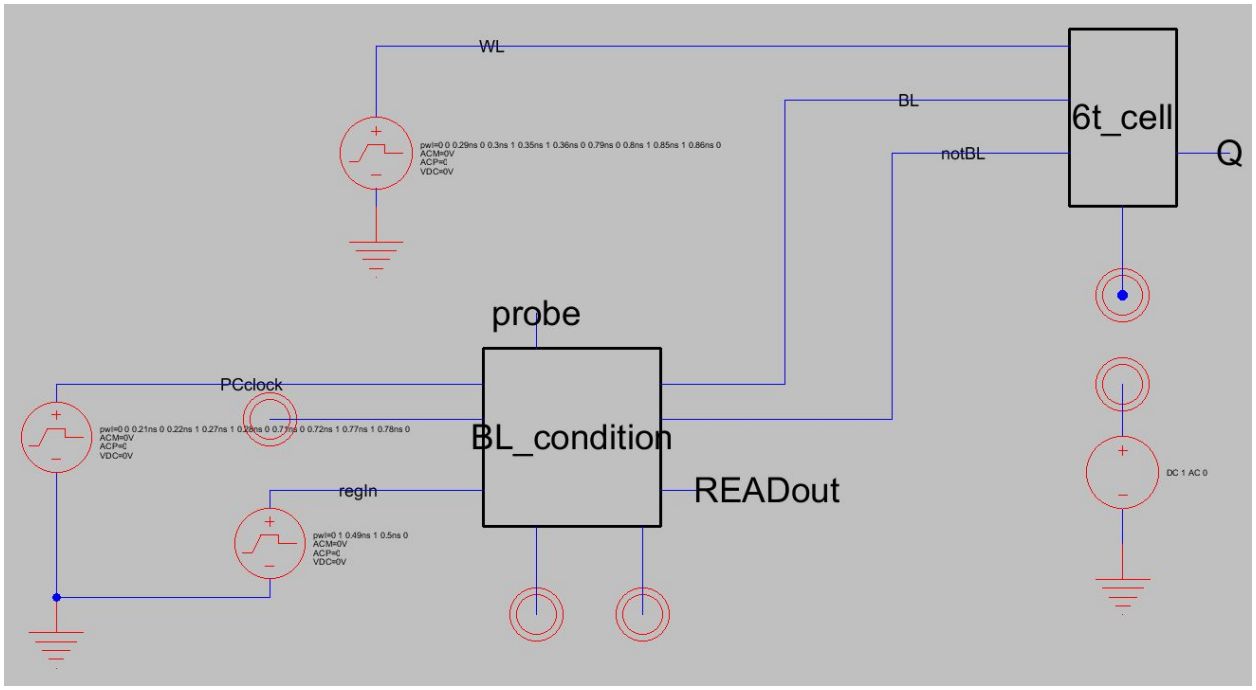
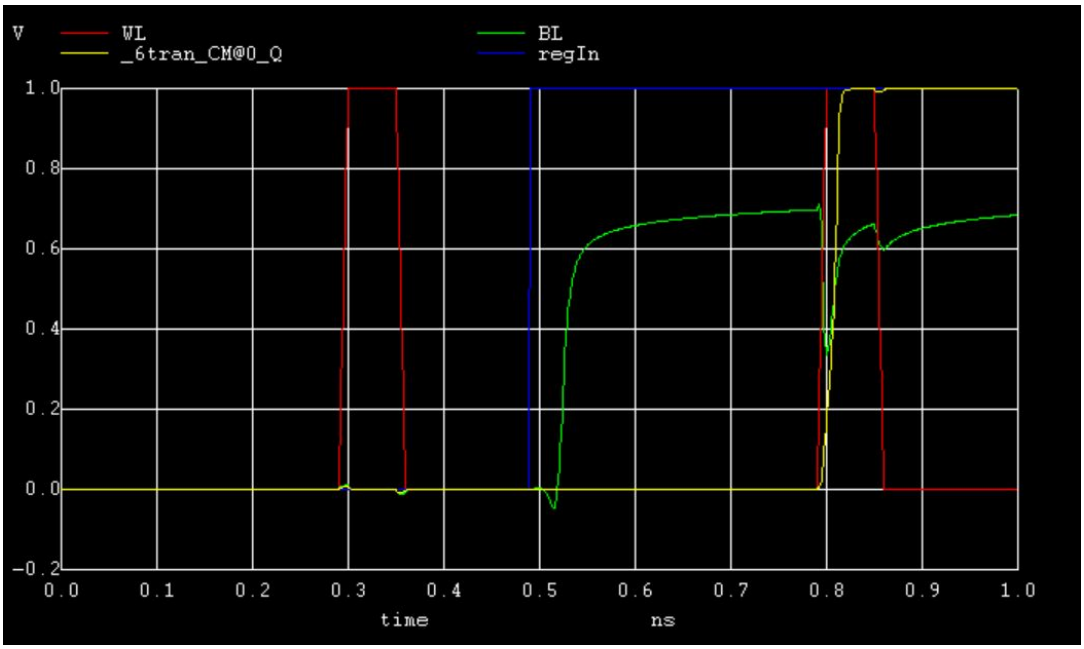


Figure. Testing circuit for bitline conditioner.

In the following cases, we want to see how effective the bitline conditioner is in getting the SRAM circuit to write one value over another. The first 0.5ns is writing the pre-value. The second 0.5ns is writing the new value over.

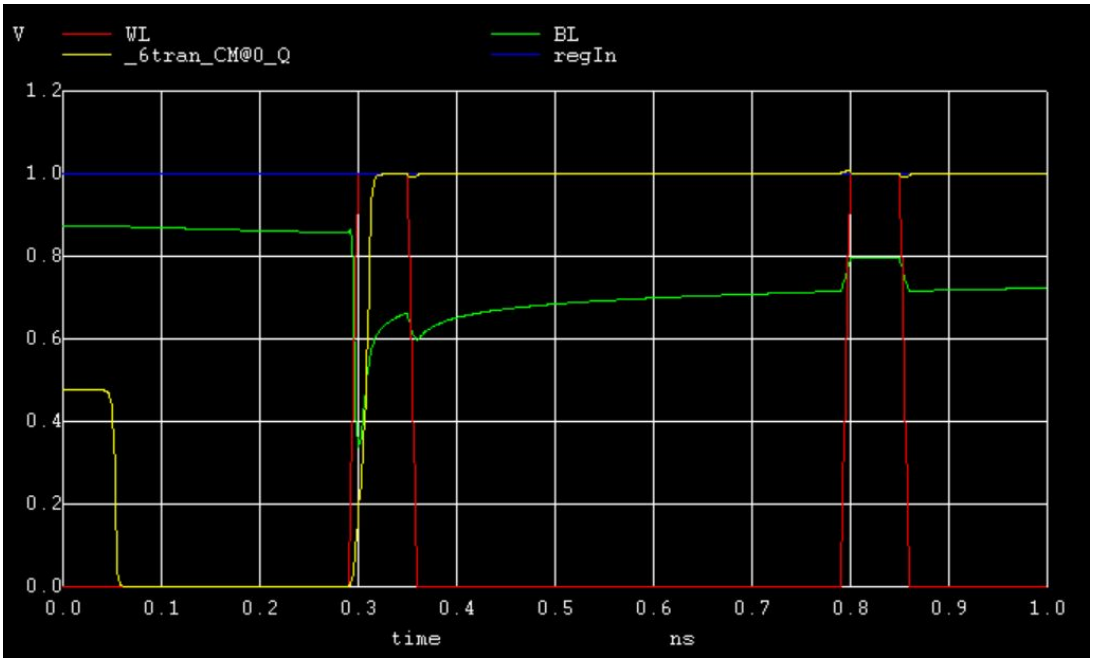
- CASE 1: (MUX = 1, WRITE 1 over 0)

```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 0 0.49ns 0 0.5ns 1
WRITE: 1
MUX: 1
```



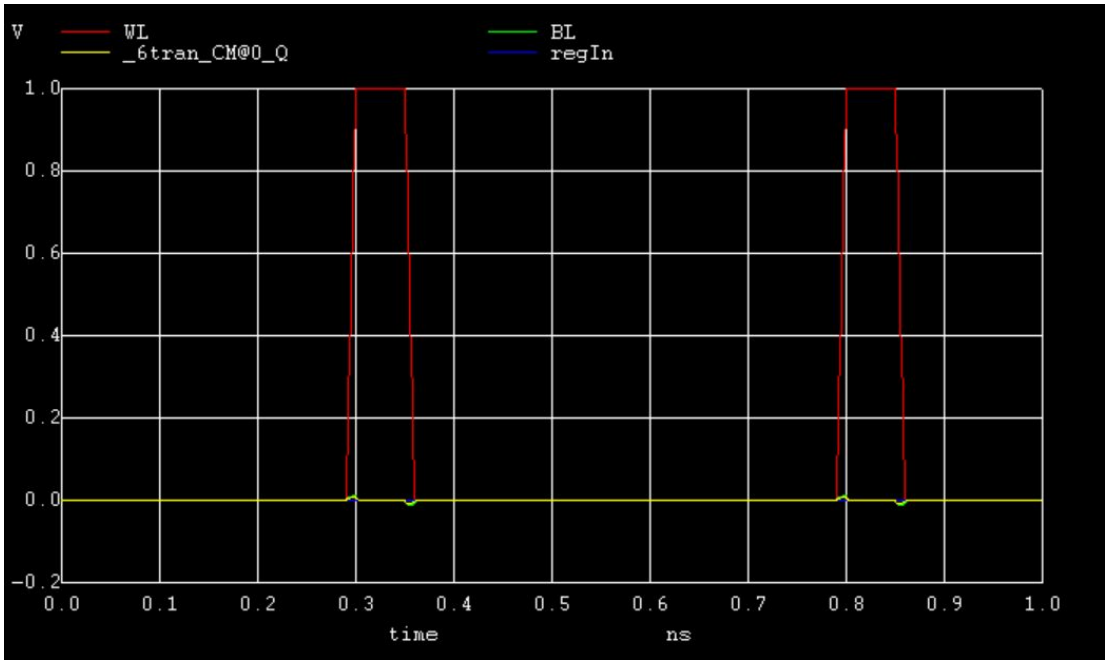
- CASE 2: (MUX = 1, WRITE 1 over 1)

```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 1 0.49ns 1 0.5ns 1
WRITE: 1
MUX: 1
```



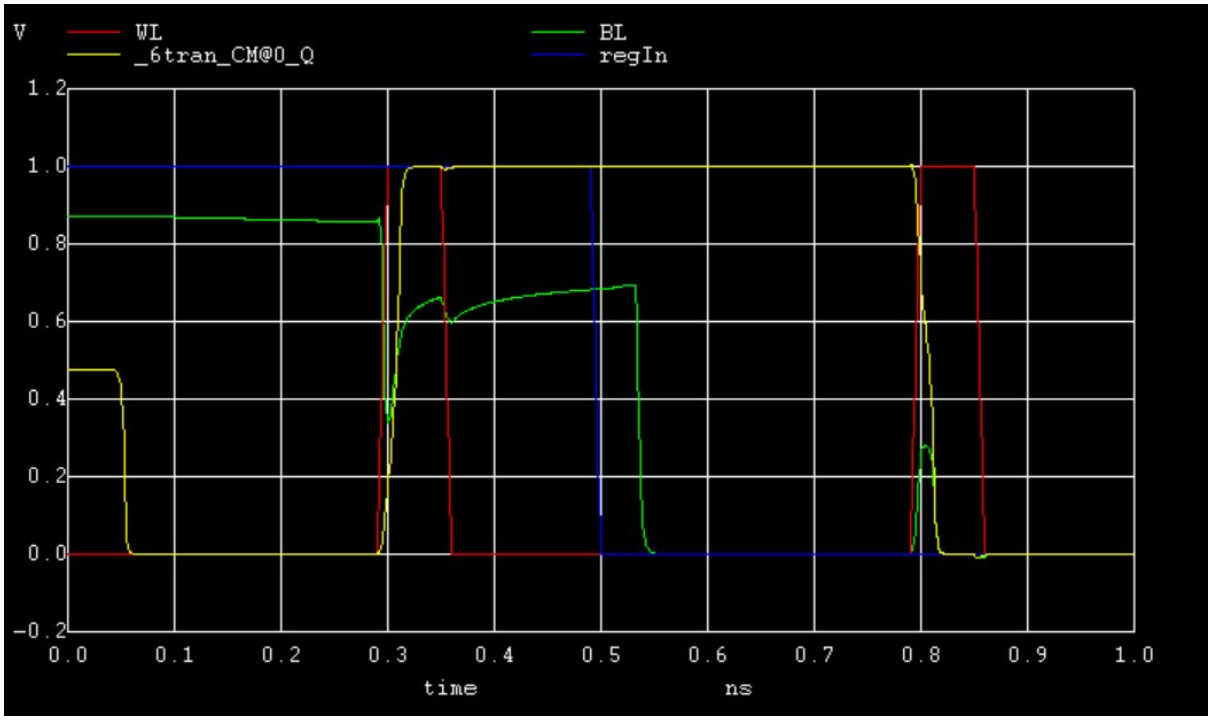
- CASE 3: (MUX = 1, WRITE 0 over 0)

```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 0 0.49ns 0 0.5ns 0
WRITE: 1
MUX: 1
```



- CASE 4: (MUX = 1, WRITE 0 over 1)

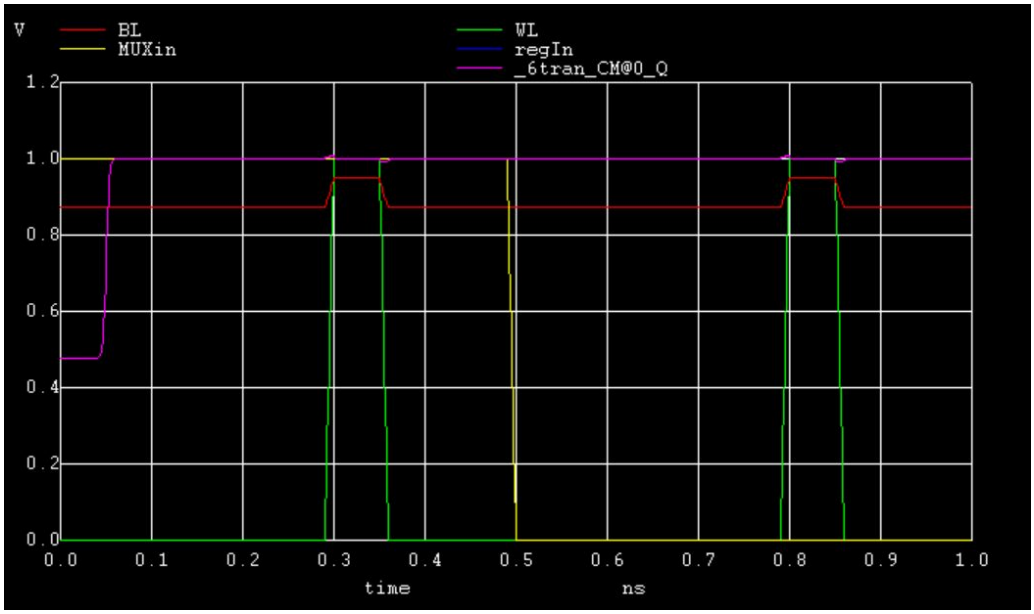
```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 1 0.49ns 1 0.5ns 0
WRITE: 1
MUX: 1
```



In the following cases, we test whether the bitline conditioner will properly disallow the cell from writing to the cell. In the first cycle, the value is set. In the second cycle, the value is set but the MUXinput is deactivated, so nothing should change in the Q value of the cell in the second cycle.

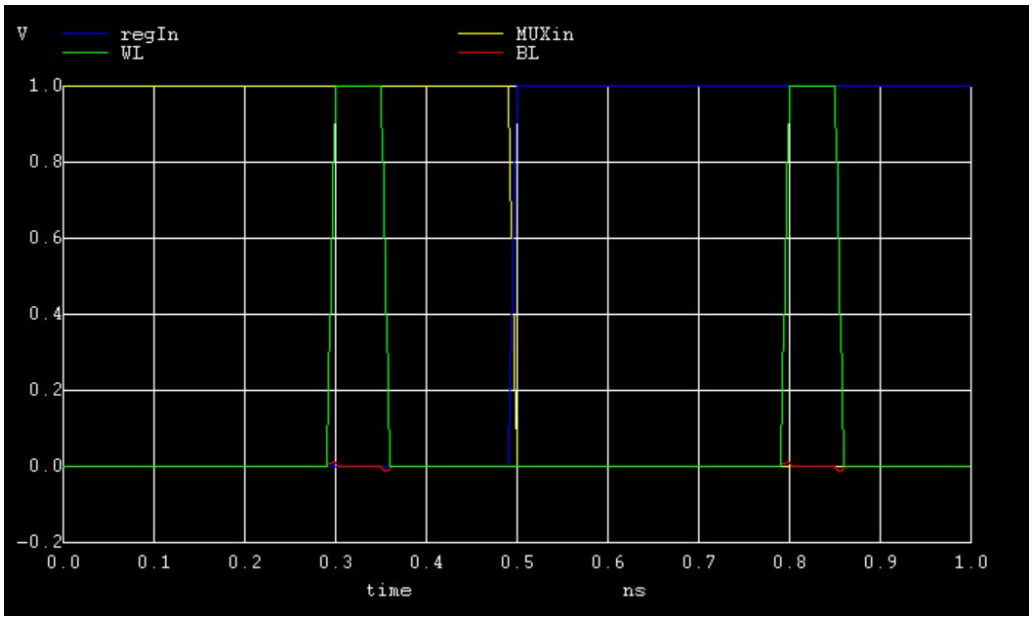
- CASE 5: (MUX = 0, WRITE 0 over 1)

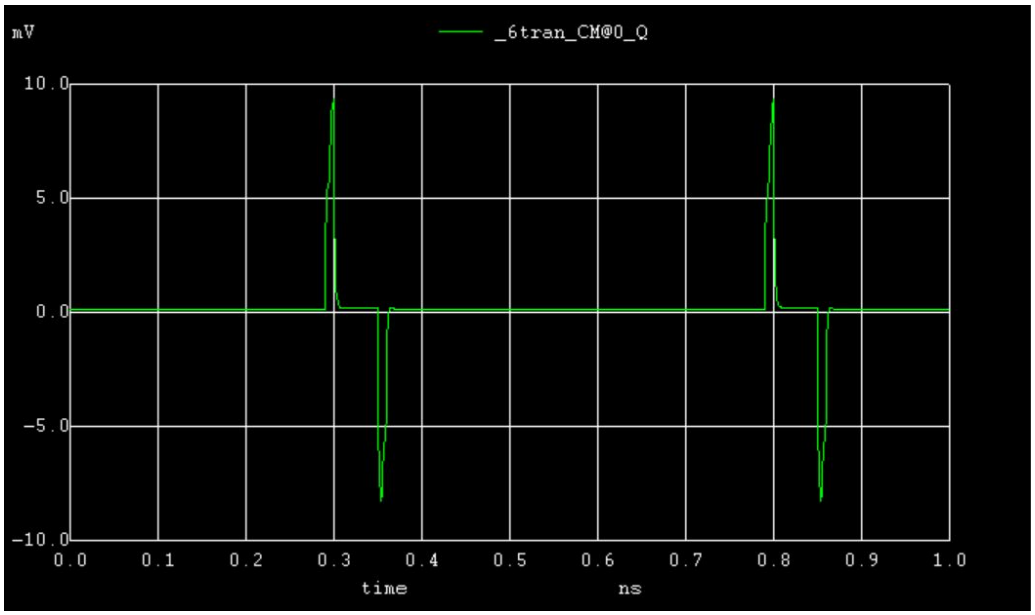
```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 1 0.49ns 1 0.5ns 0
WRITE: 1
MUX: 0 1 0.49ns 1 0.5ns 0
```



- CASE 6: (MUX = 0, WRITE 1 over 0)

```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0
PCclock: 0 0 0.21ns 0 0.22ns 1 0.27ns 1 0.28ns 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0
WriteIn: 0 0 0.49ns 0 0.5ns 1
WRITE: 1
MUX: 0 1 0.49ns 1 0.5ns 0
```



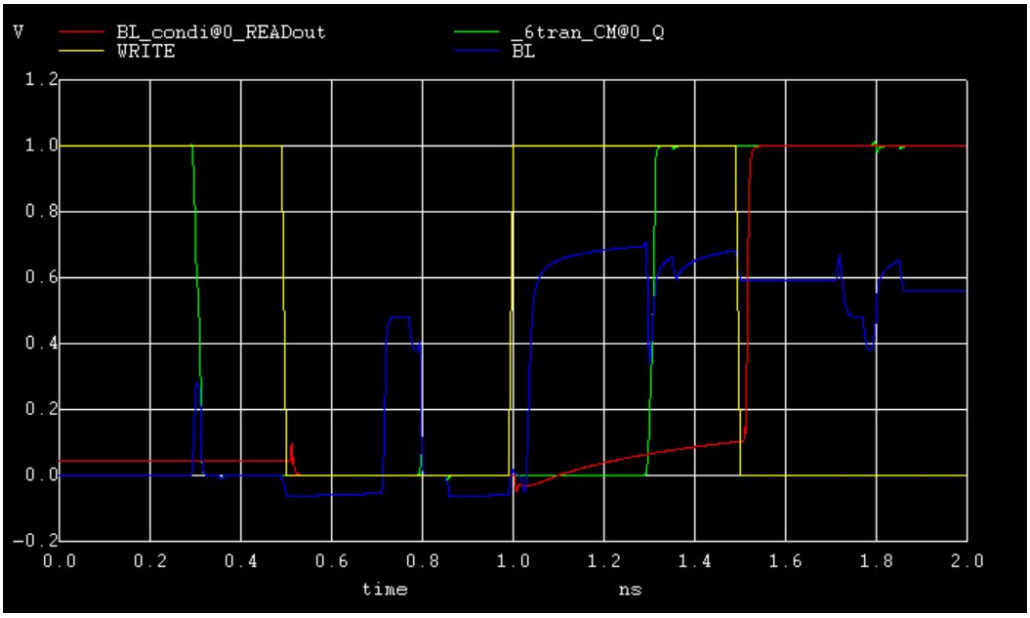


In the following cases, we test the reading functionality. We write in the precharging value and read it out, to precharge the READout as appropriate. Then we write in the desired read value and then read it out.

- CASE 7: (MUX = 1, READ 1 over 0)

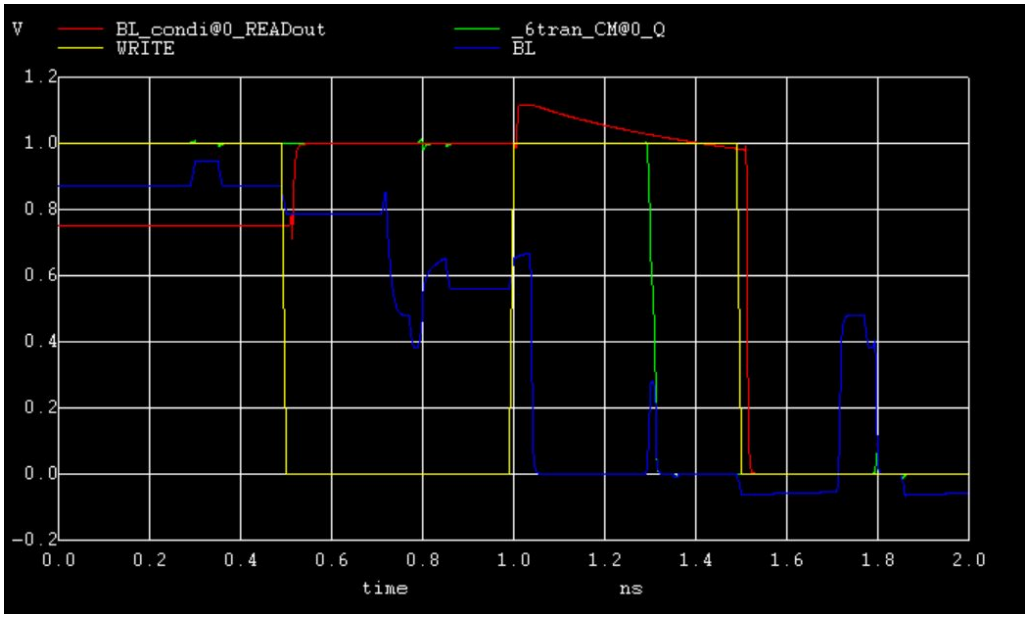
```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0 1.29ns
0 1.3ns 1 1.35ns 1 1.36ns 0 1.79ns 0 1.8ns 1 1.85ns 1 1.86ns 0
PCclock: 0 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0 1.71ns 0 1.72ns 1 1.77ns 1 1.78ns 0
WriteIn: 0 0 0.99ns 0 1ns 1
WRITE: 0 1 0.49ns 1 0.5ns 0 0.99ns 0 1ns 1 1.49ns 1 1.5ns 0
```

```
MUX: 1
```



- CASE 8: (MUX = 1, READ 0 over 1)

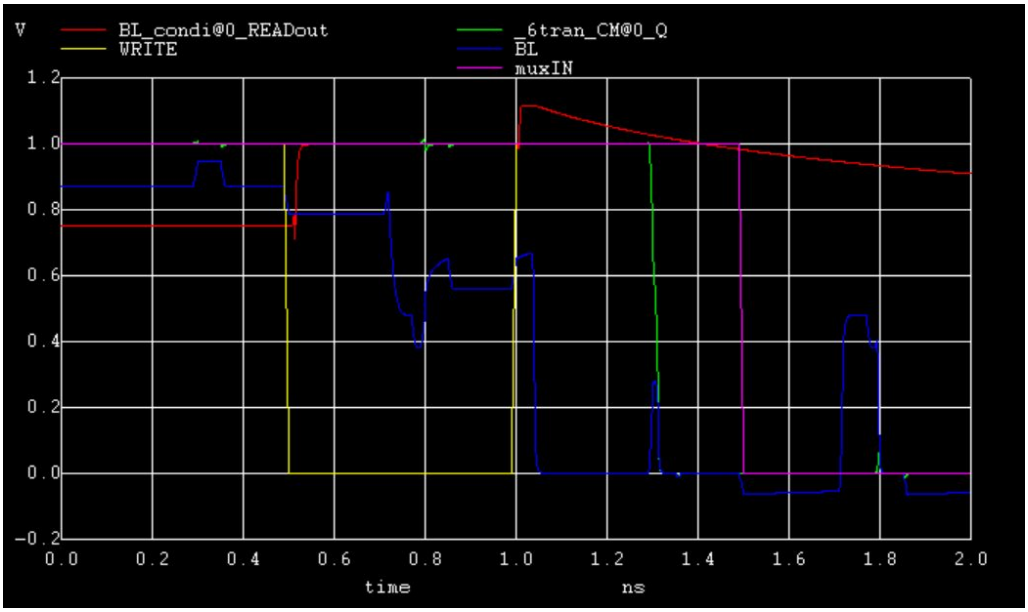
```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0 1.29ns
0 1.3ns 1 1.35ns 1 1.36ns 0 1.79ns 0 1.8ns 1 1.85ns 1 1.86ns 0
PCclock: 0 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0 1.71ns 0 1.72ns 1 1.77ns 1 1.78ns 0
WriteIn: 0 1 0.99ns 1 1ns 0
WRITE: 0 1 0.49ns 1 0.5ns 0 0.99ns 0 1ns 1 1.49ns 1 1.5ns 0
MUX: 1
```

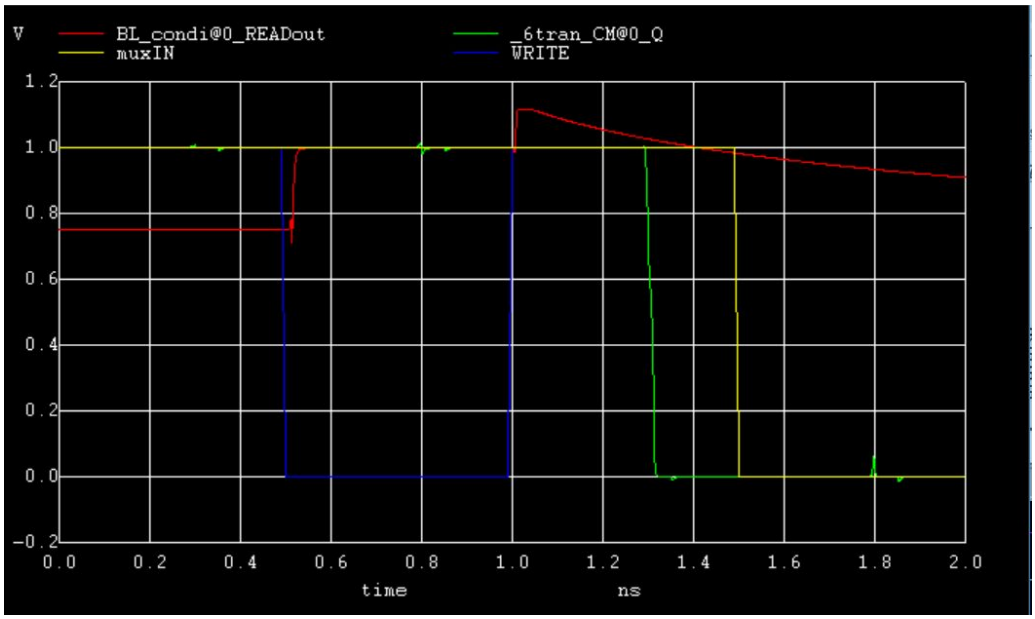


In the following cases, we want to ensure that when the MUXinput is deactivated, the read functionality does not change the READout value.

- CASE 9: (MUX = 0, READ 0 over 1)

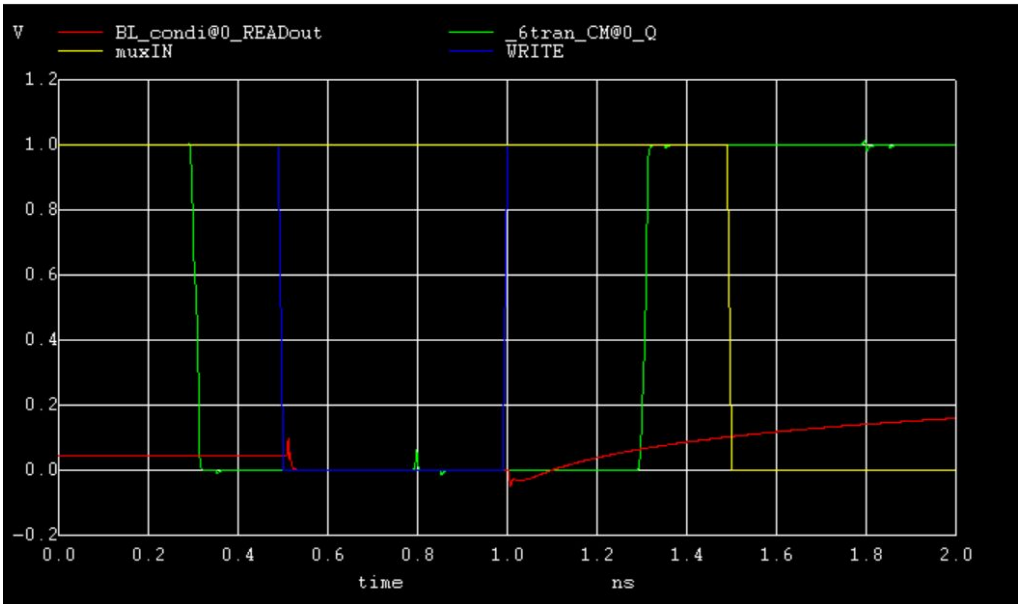
```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0 1.29ns
0 1.3ns 1 1.35ns 1 1.36ns 0 1.79ns 0 1.8ns 1 1.85ns 1 1.86ns 0
PCclock: 0 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0 1.71ns 0 1.72ns 1 1.77ns 1 1.78ns 0
WriteIn: 0 1 0.99ns 1 1ns 0
WRITE: 0 1 0.49ns 1 0.5ns 0 0.99ns 0 1ns 1 1.49ns 1 1.5ns 0
MUX: 0 1 1.49ns 1 1.5ns 0
```



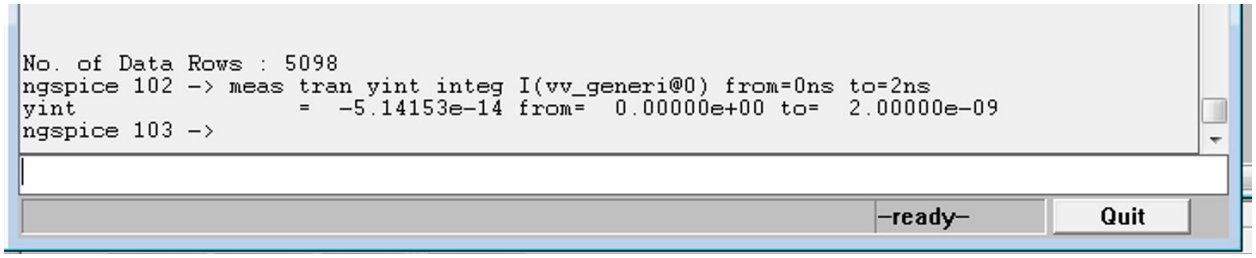


- CASE 10: (MUX = 0, READ 1 over 0)

```
WL: 0 0 0.29ns 0 0.3ns 1 0.35ns 1 0.36ns 0 0.79ns 0 0.8ns 1 0.85ns 1 0.86ns 0 1.29ns
0 1.3ns 1 1.35ns 1 1.36ns 0 1.79ns 0 1.8ns 1 1.85ns 1 1.86ns 0

PCclock: 0 0 0.71ns 0 0.72ns 1 0.77ns 1 0.78ns 0 1.71ns 0 1.72ns 1 1.77ns 1 1.78ns 0

WriteIn: 0 0 0.99ns 0 1ns 1

WRITE: 0 1 0.49ns 1 0.5ns 0 0.99ns 0 1ns 1 1.49ns 1 1.5ns 0

MUX: 0 1 1.49ns 1 1.5ns 0
```



Energy Usage



WRITE/READ Controller:

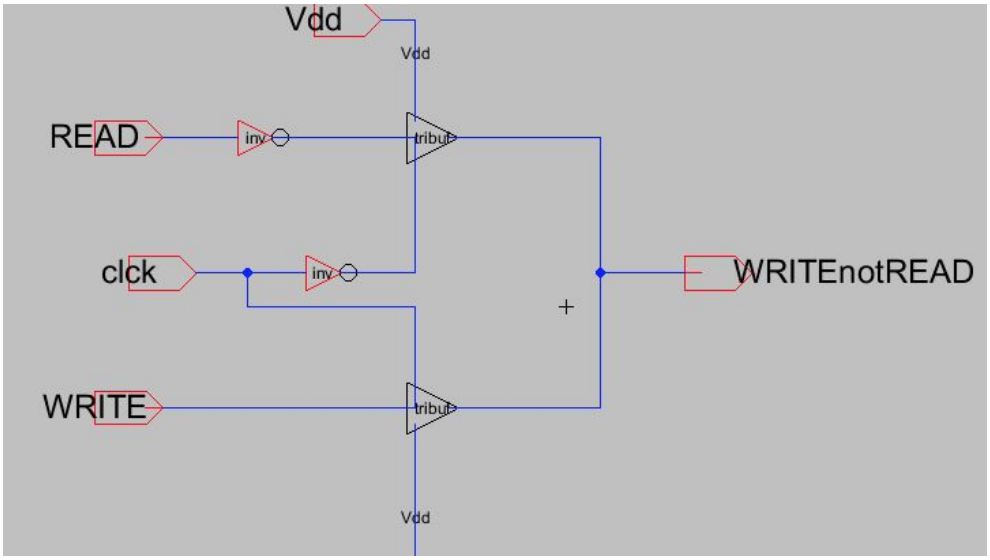


Figure. Schematic for circuitry controlling the Write/!Read line.

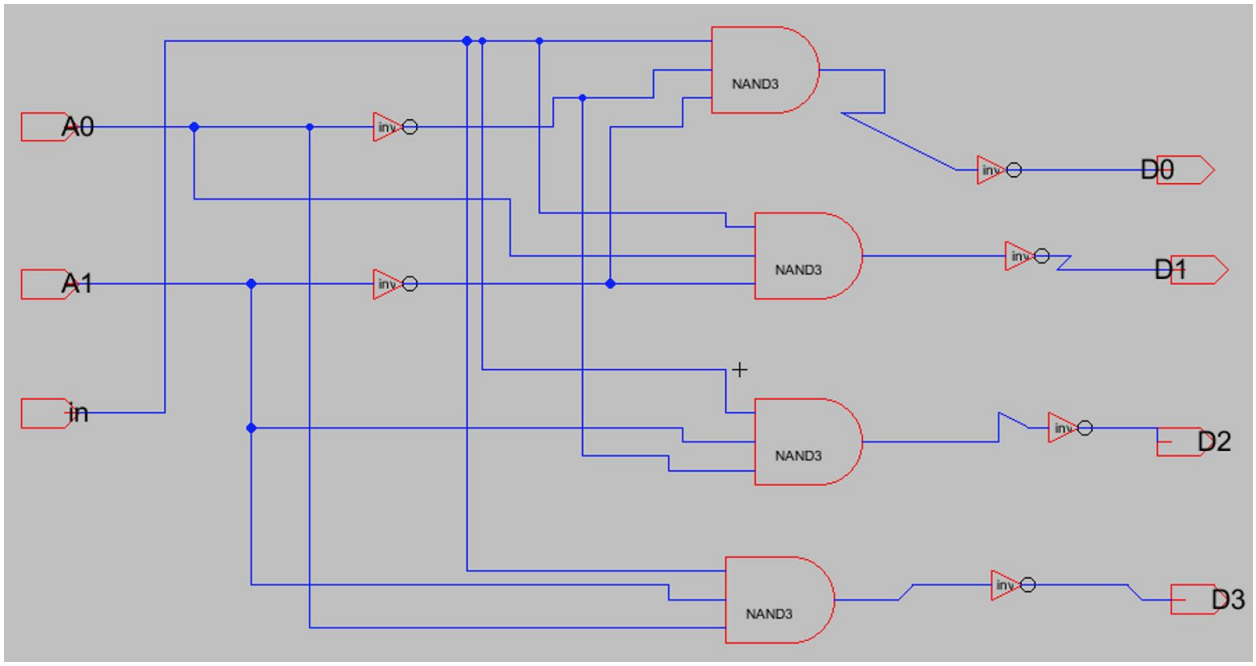
We want to make the Write/!Read line such that in the first half of the cycle, during the write portion, it is HIGH if an enqueue was requested, and during the second half of the cycle, during the read portion, it is LOW if a dequeue was requested. Consideration of FULL and EMPTY are handled in the other part of the circuit.

Column Decoder:

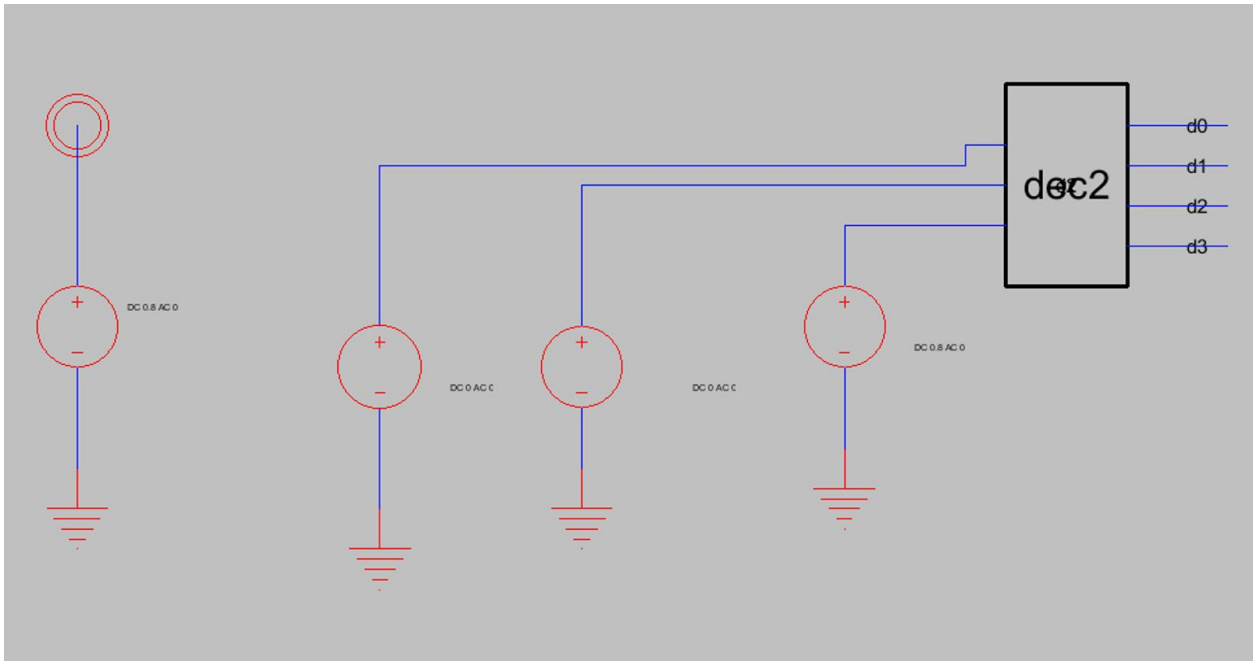
Truth Table for 2-4 decoder

In	A1	A0	D3	D2	D1	D0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

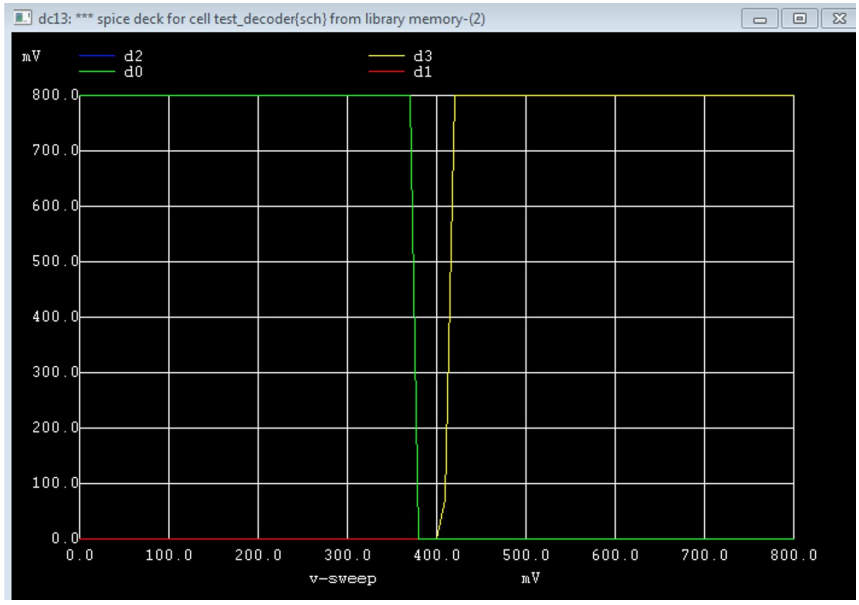
The decoder acts as a demultiplexer with one input and 2 select signals labelled A0 and A1. When the input is high then the decoder is activated and outputs according to the truth table shown above. When the input is low then all four signals outputed are low. The inputs acts as an enable signal.

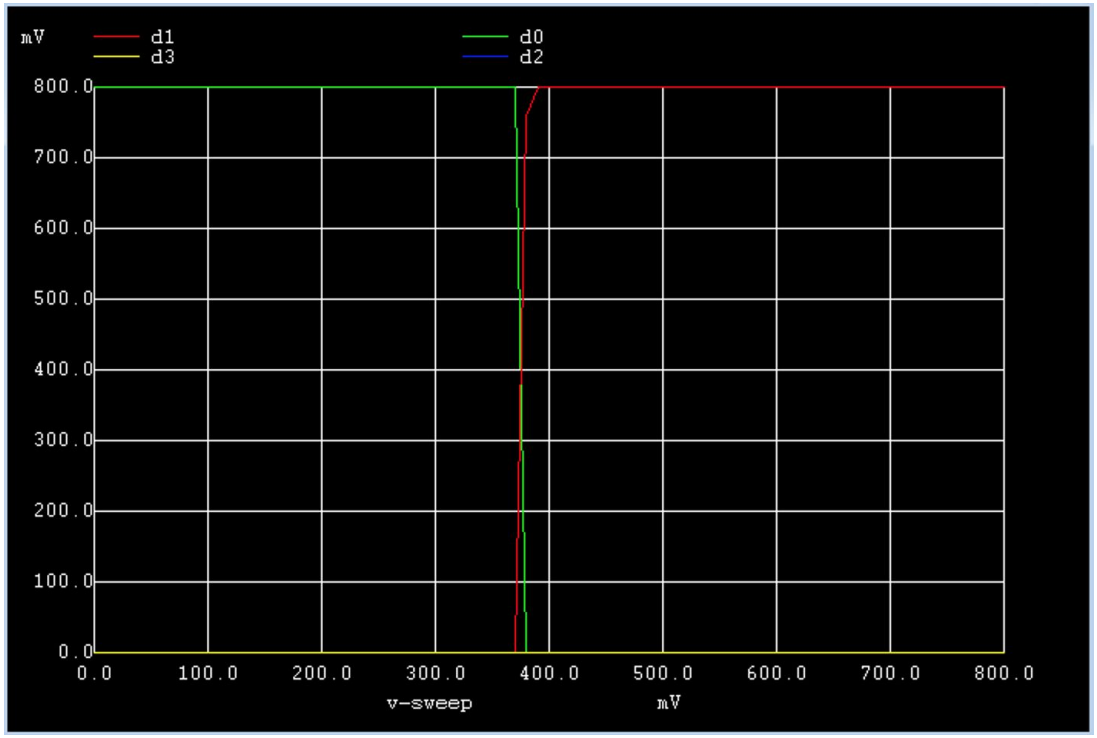


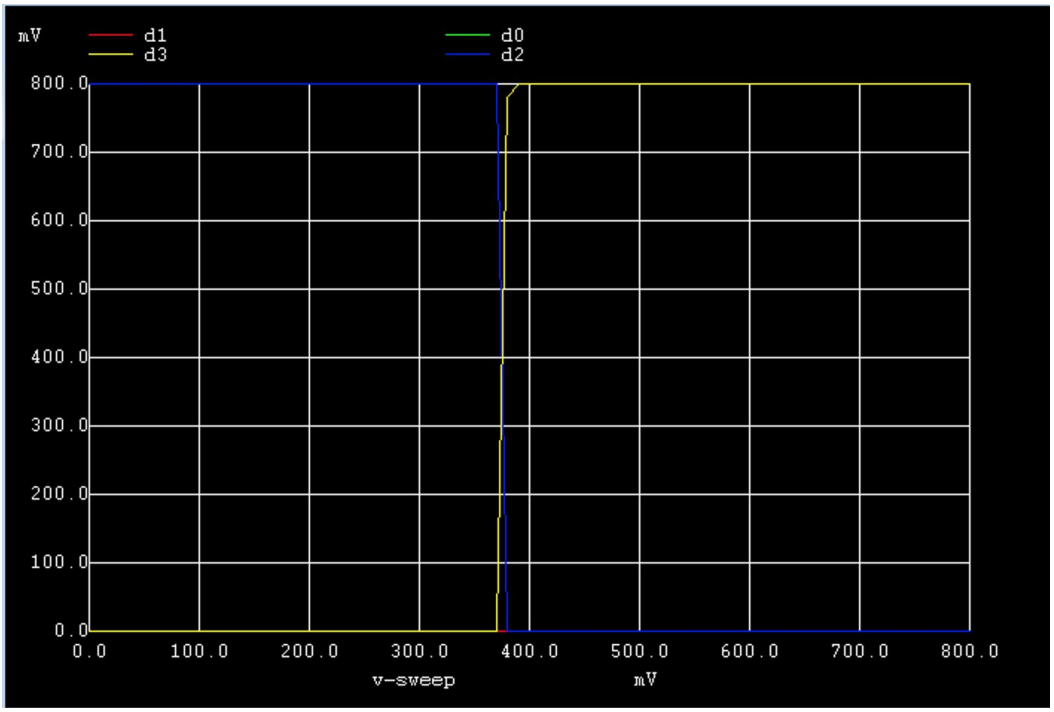
The following test bench was set-up:



A sweep was run from 0 to 0.8V for both inputs. When both A0 and A1 were low then d0 was high and the rest of the outputs were low. When both A1 and A0 were high, d3 was high and the rest of the outputs were low. The simulations match the expected results from the truth table.


dc13: *** spice deck for cell test_decoder{sch} from library memory-(2)
mV
d2
d0
d3
d1
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0.0
100.0
200.0
300.0
400.0
500.0
600.0
700.0
800.0
v-sweep
mV

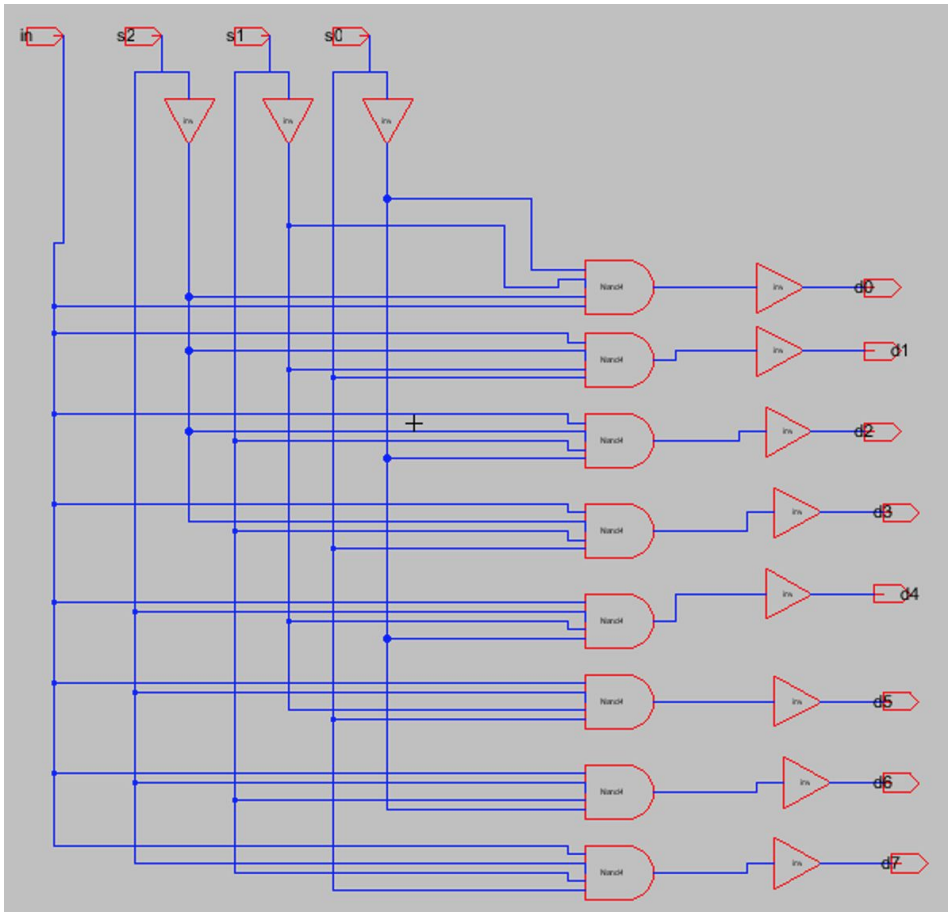

mV
d1
d3
d0
d2
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0.0
100.0
200.0
300.0
400.0
500.0
600.0
700.0
800.0
v-sweep
mV

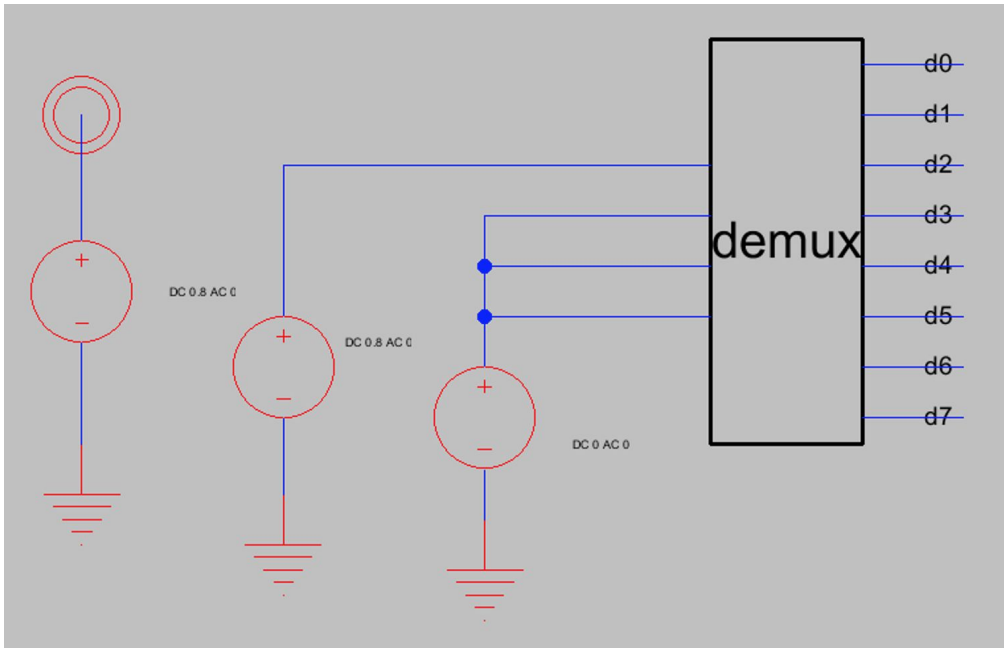


Row Decoder:

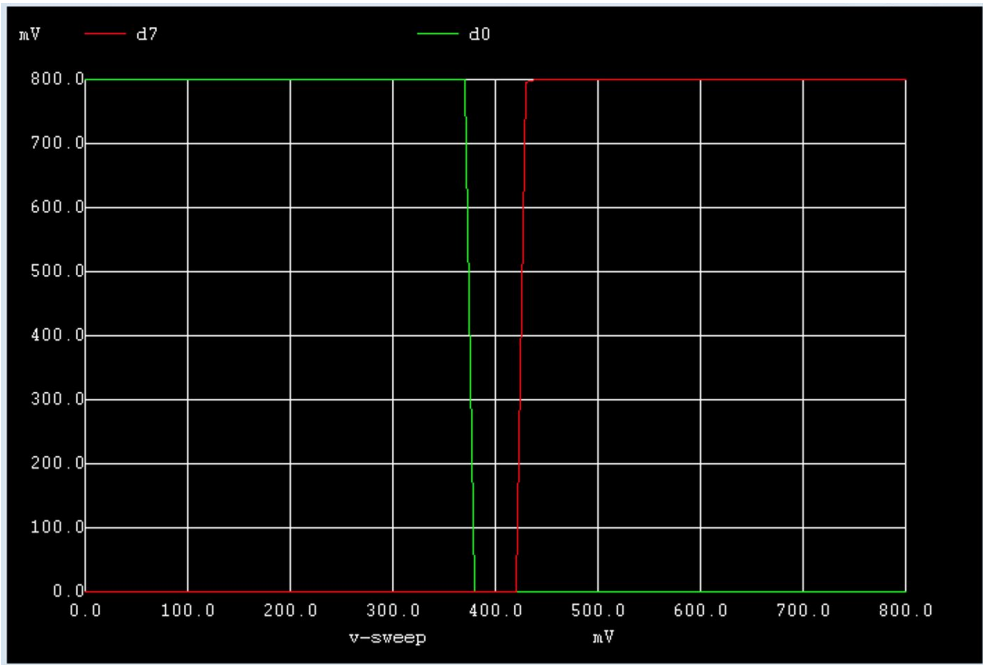
Truth Table for 3-8 decoder

A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

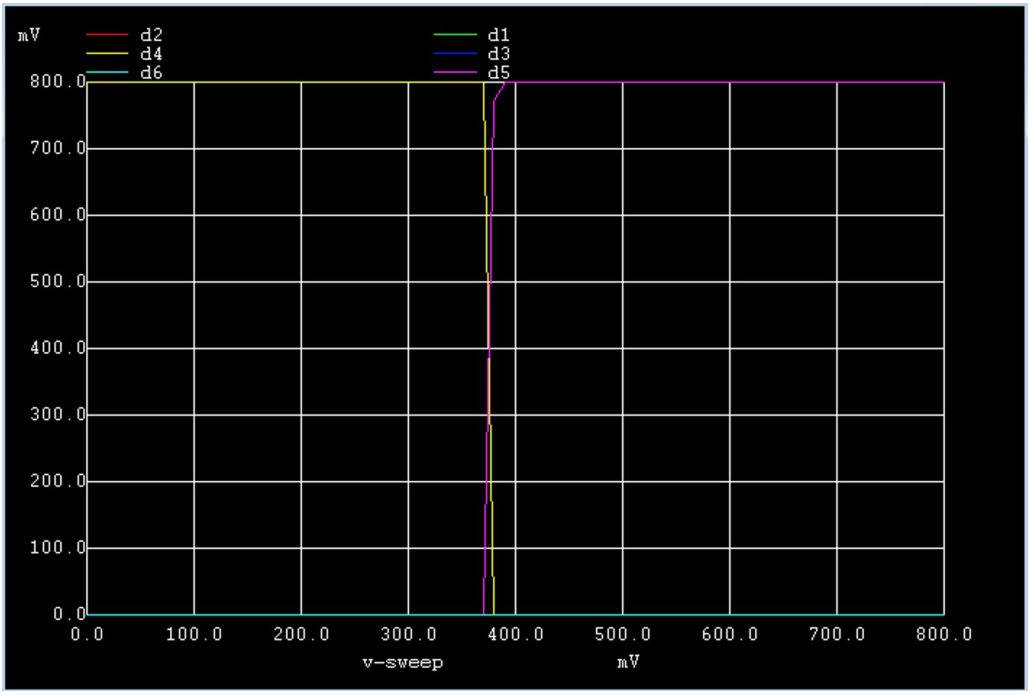

in
s2
s1
s0
d0
d1
d2
d3
d4
d5
d6
d7

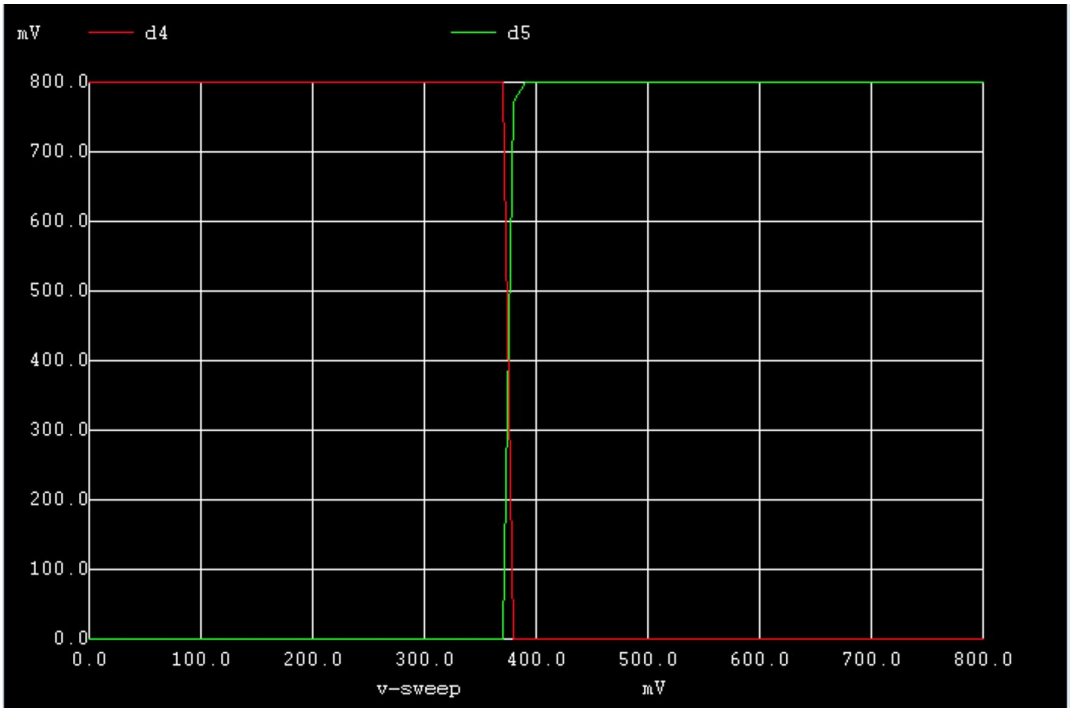

DC 0.8 AC 0
DC 0.8 AC 0
DC 0 AC 0
demux
d0
d1
d2
d3
d4
d5
d6
d7

All select signals low in the first half of the sweep and high in the second half.



Case 2: Holding s2 at vdd and s1 at ground and sweeping s3.





Clocks:

We have four different clocks that we work with: one to set FULL/EMPTY, a Word Line clock, a Pre Charge clock, and the master 500MHz clock:

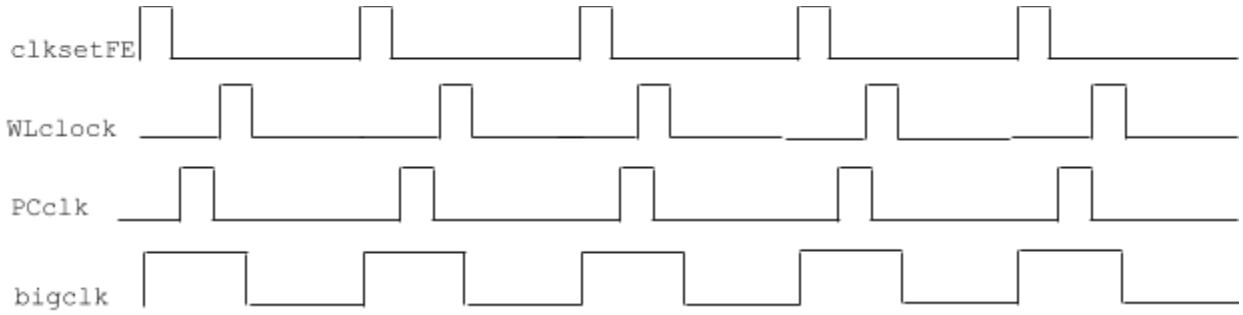


Figure. Drawing of expected clocks.

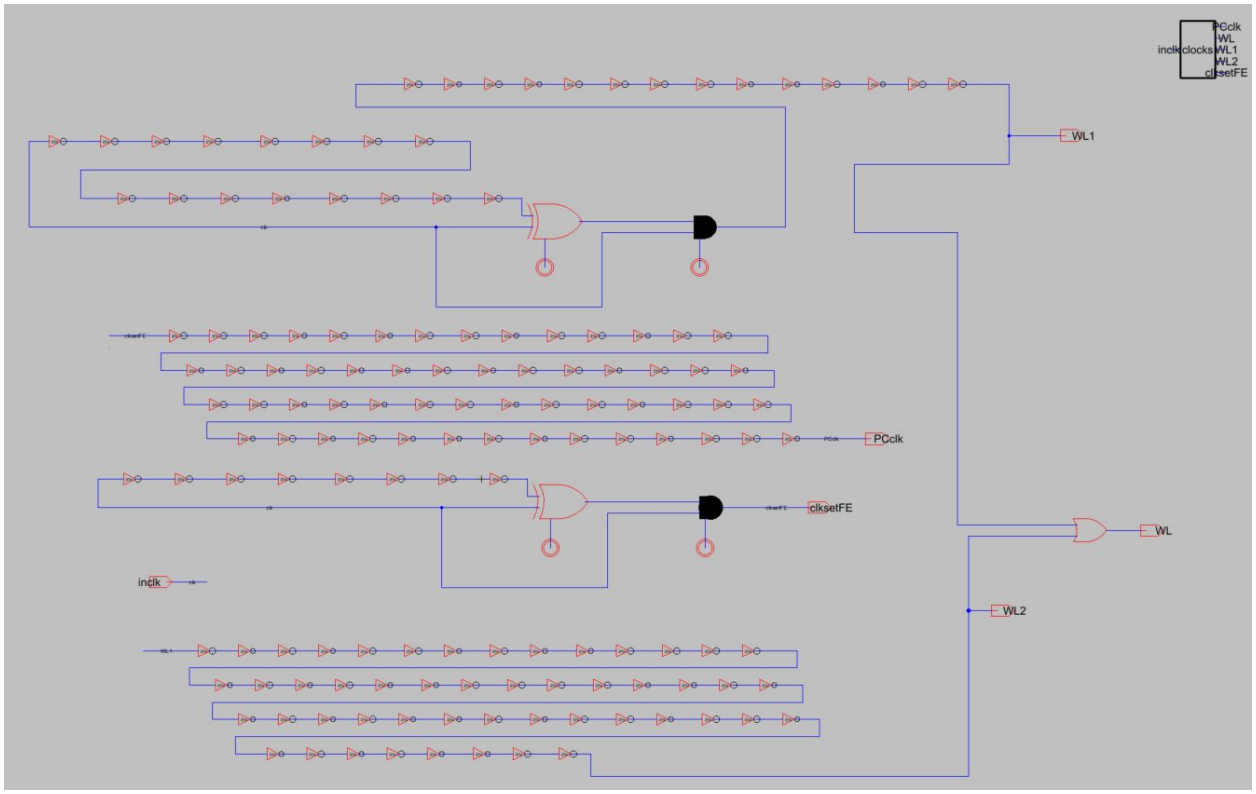


Figure. Schematic to generate clocks.

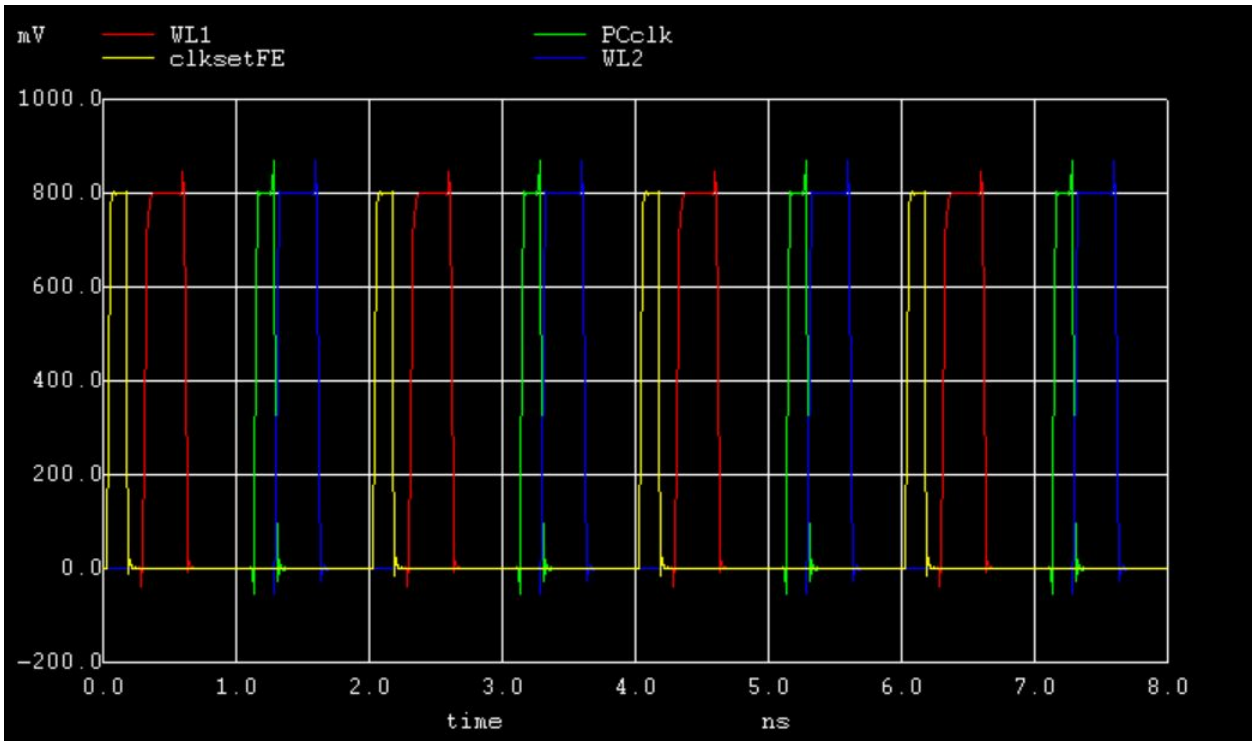


Figure. Simulation results of generated clocks.

Energy of clock generation:

```
ngspice 434 -> meas tran yint integ I(vv_generi@0) from=0ns to=2ns
yint                =  -2.56025e-14 from=  0.00000e+00 to=  2.00000e-09
```

Sense Amplifiers:

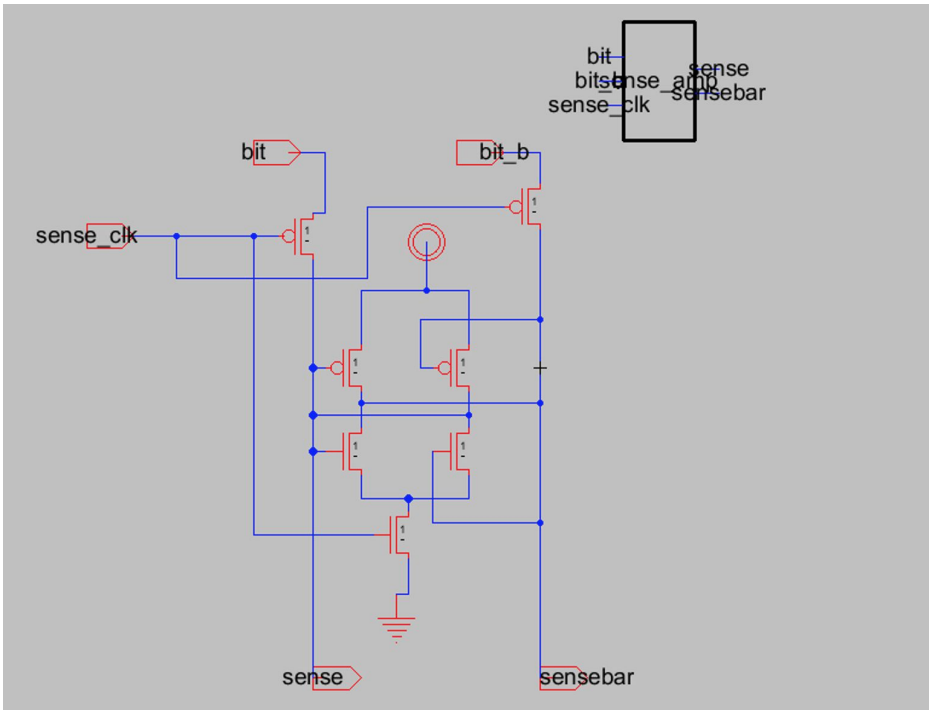


Figure. Schematic of clocked sense amplifier.

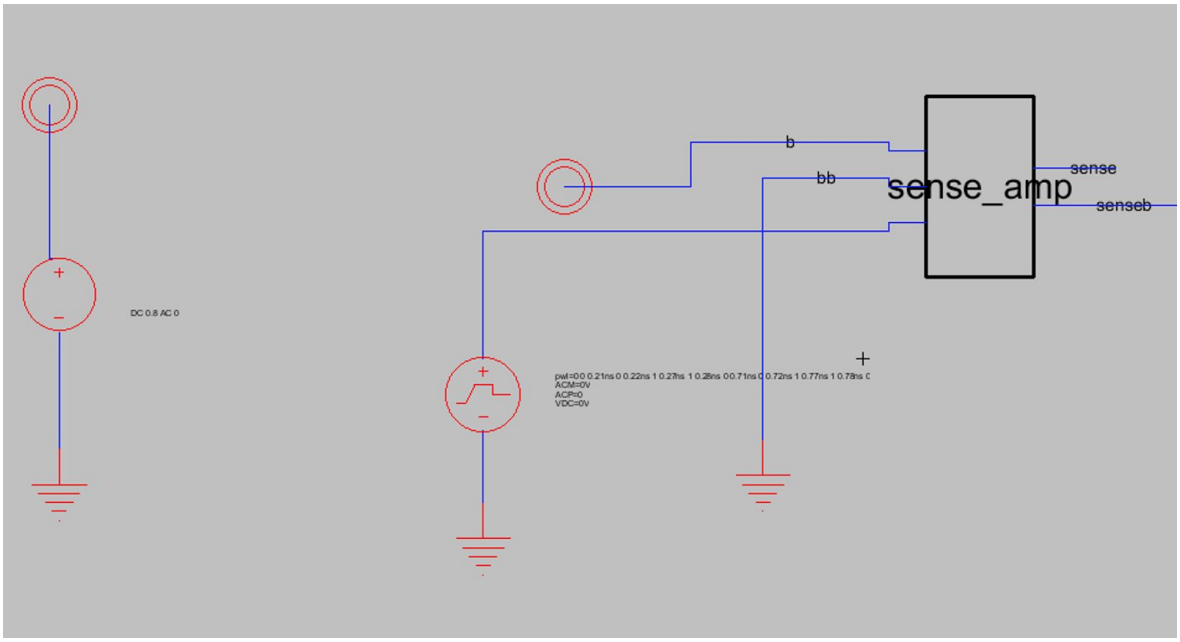


Figure. Test bench for sense amp.

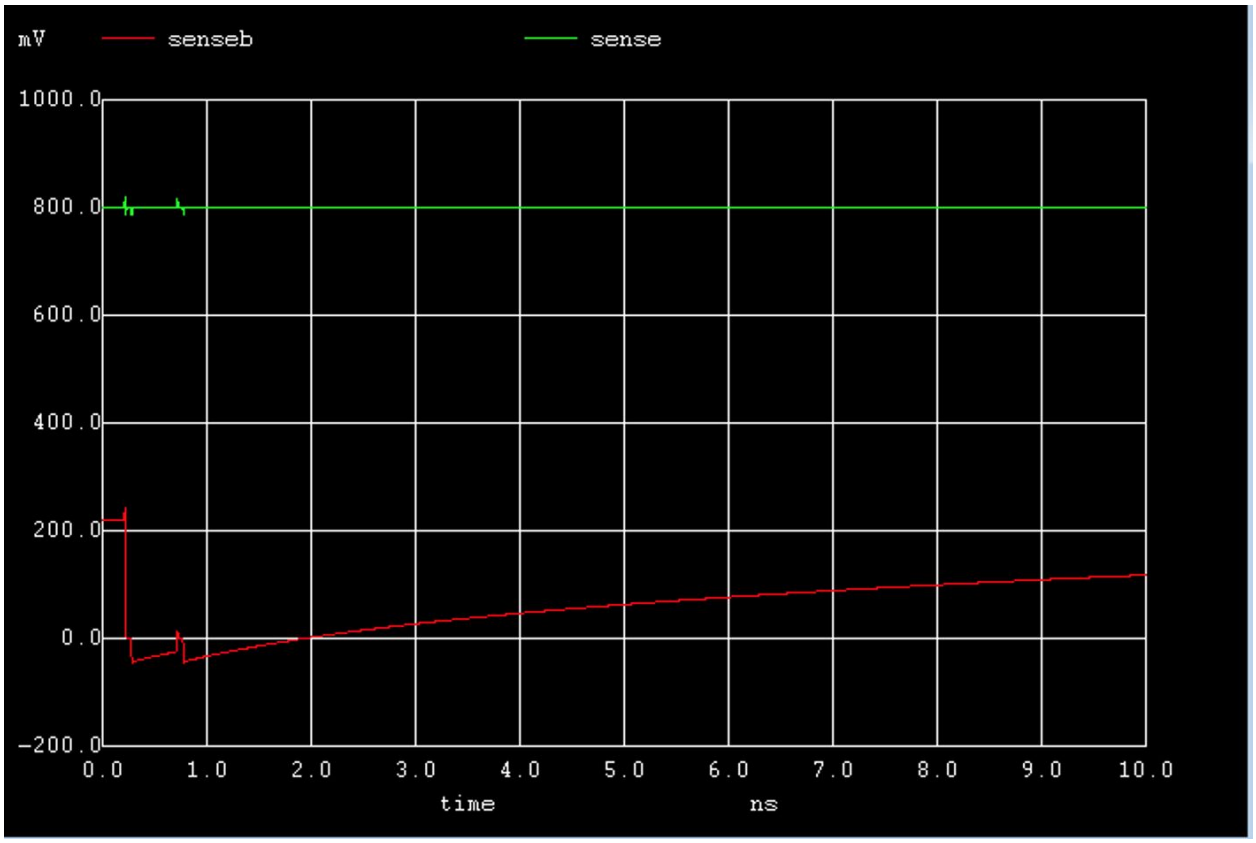


Figure. Test case 1: BL is Vdd, BLbar is GND.

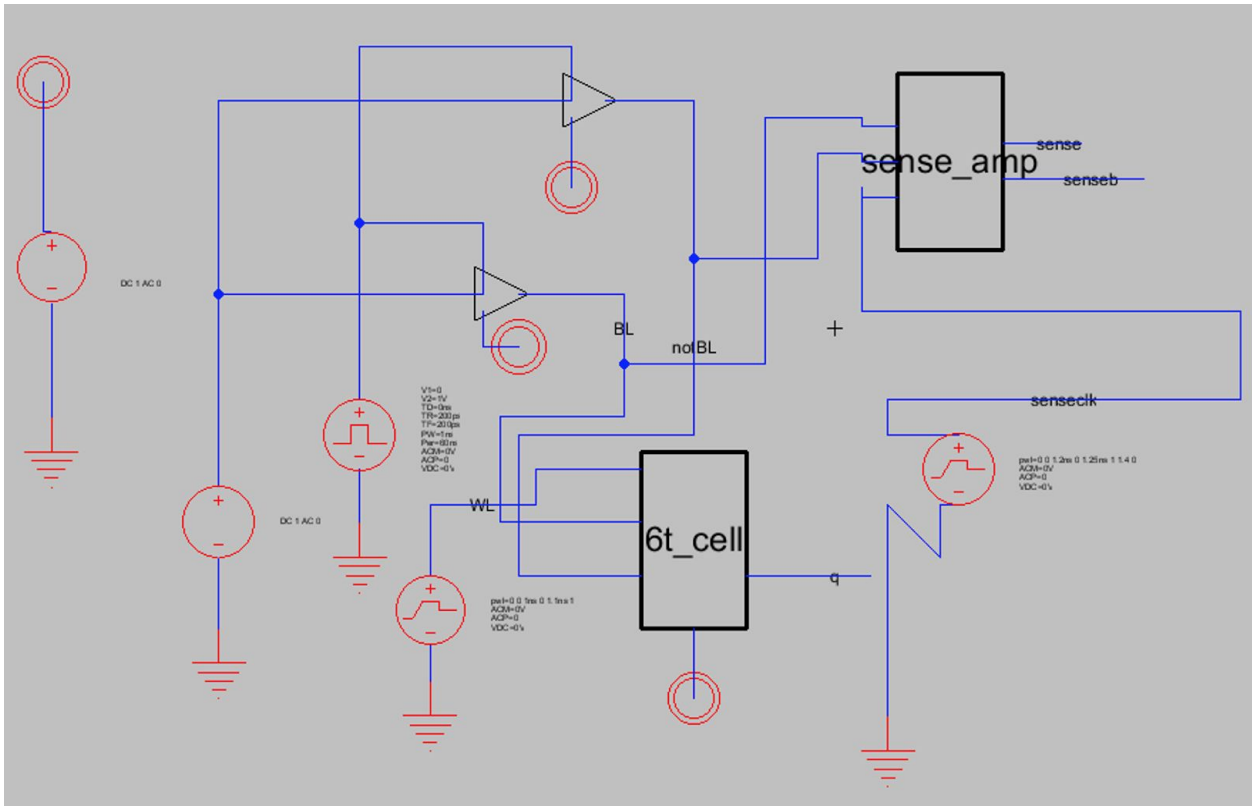
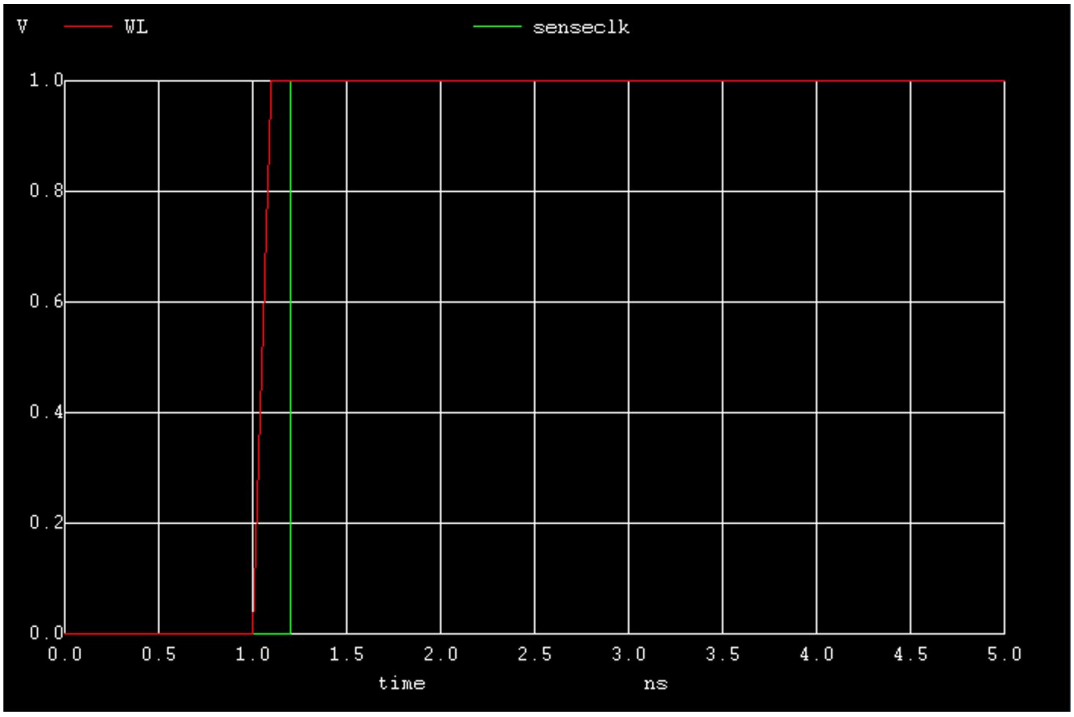


Figure. Test bench for sense amp with memory cell.

Before starting the simulation, the bitlines are precharged and a logic low is stored in the 6t_cell. First, the wordline (WL) is brought high, enabling the SRAM cell. This allows the bitline voltage (labeled BL) to begin decreasing. Once a sufficient voltage difference has developed between bitline and bitline (notBL), the sense amplifier is enabled by bringing the sense clock signal high. When the clock is low, the sources of the top PMOSes are charged to bit line and bit line bar respectively.

In the simulation below, the sense clock was made to stay high after turning on.



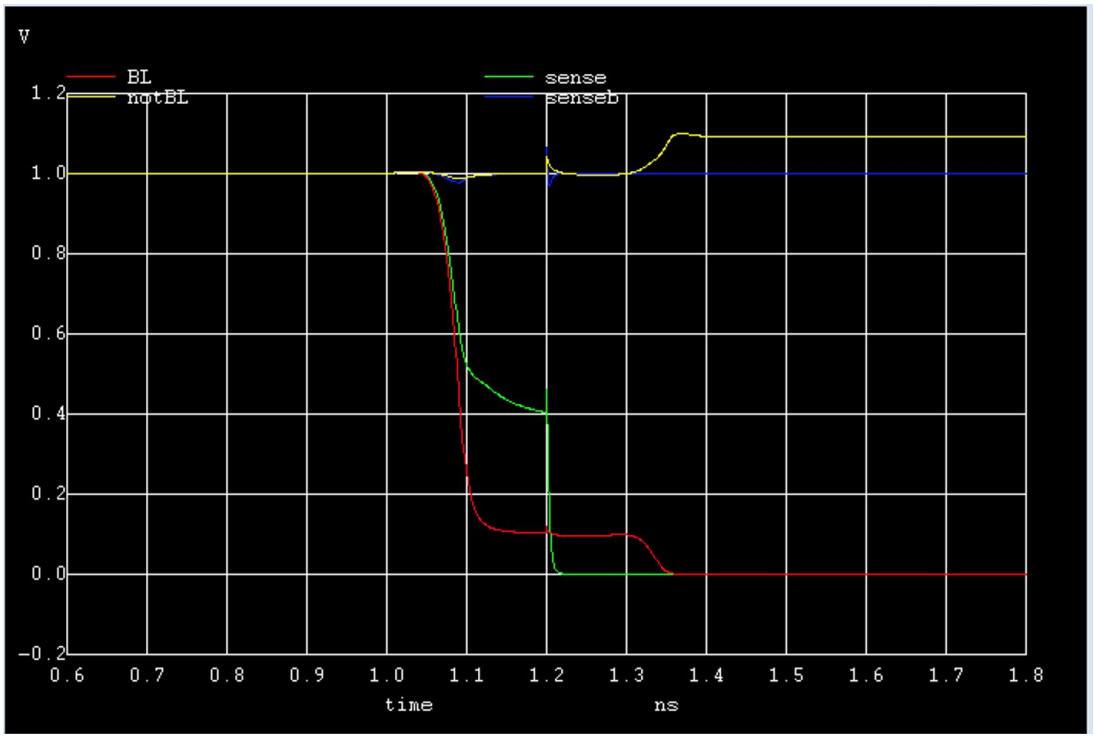


Figure. Sense clock stays high:

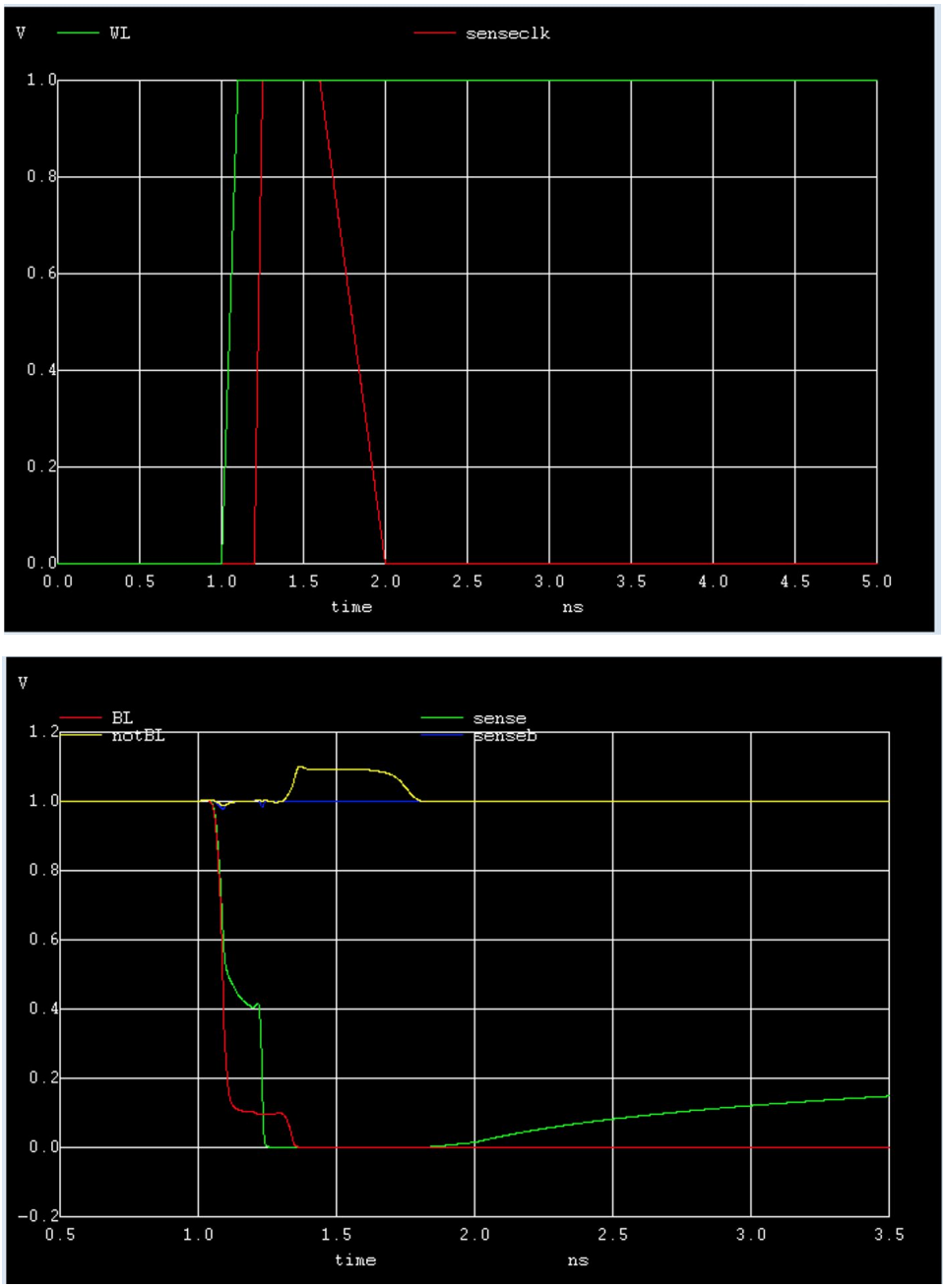


Figure. Sense clock stays low:

Energy Usage:

```
ngspice 92 -> meas tran yint integ I(vv_generi@1) from=0ns to=2ns
yint                =  -2.15201e-15 from=  0.00000e+00 to=  2.00000e-09
ngspice 93 ->
```

-ready- Quit

Operations

The sequence of events within each clock cycle are as follows:

1. Compare: `c := READPTR == WRITEPTR`
2. If `c == 1`, set `EMPTY` and `FULL` based on `flag` value.
3. Set `flag := READPTR+1 == WRITEPTR`
4. Decode `WRITEPTR`
5. Condition `BL` and `notBL`
6. Send appropriate `WL` up if `ENQUEUE && !FULL`. Bring it back down.
7. Decode `READPTR`
8. Precharge `BL` and `notBL`
9. Send appropriate `WL` up if `DEQUEUE && !EMPTY`. Bring it back down.
10. Sense amplifiers read `BL` out to the output registers
11. Set new pointer addresses as appropriate.

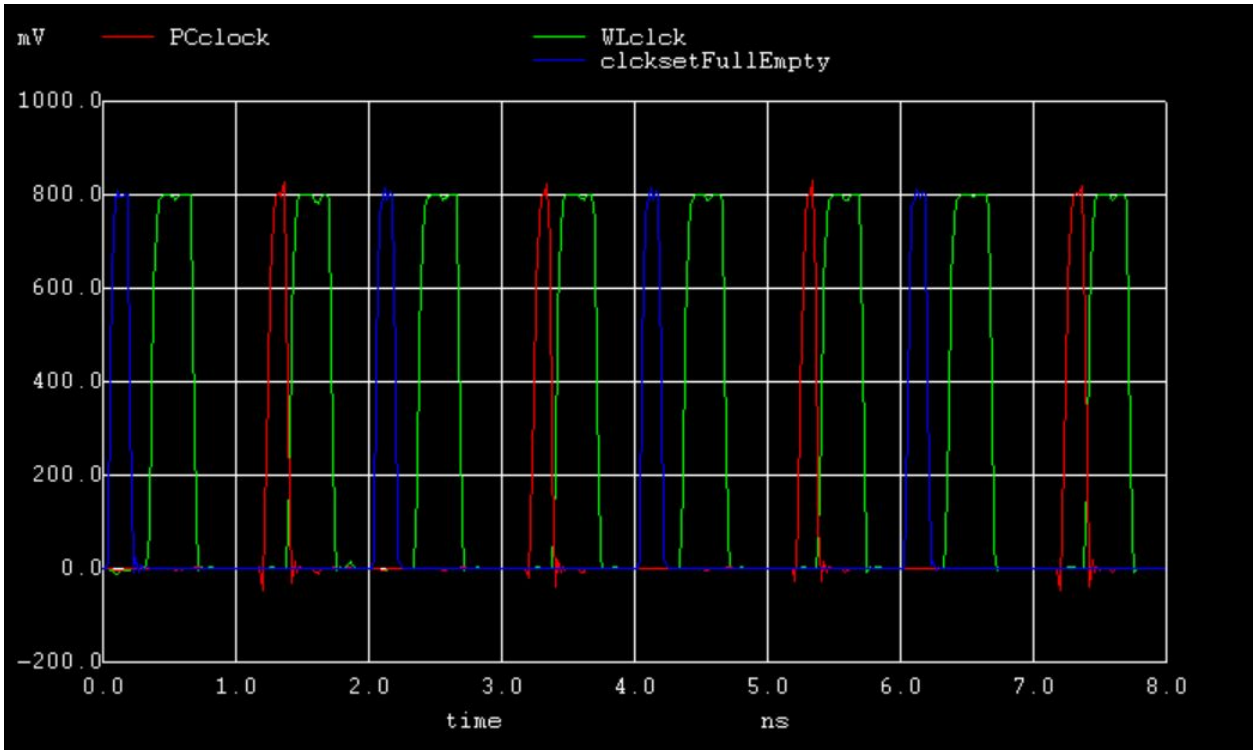


Figure. Simulation showing our generated clocks.

To demonstrate the efficacy of the circuit, we will demonstrate the following sequence:

1. WRITE 0110
2. WRITE 1001, READ

Line 2 should output 0110 in the output.

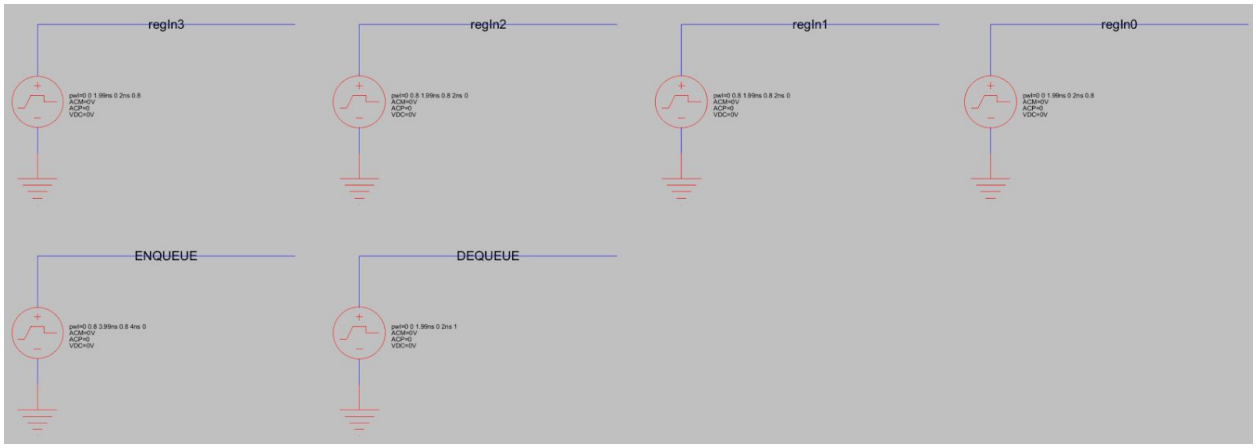


Figure. Inputs for demonstration of circuit.

We had trouble with getting our pointers to both start at [00000]. Instead, the read pointer started at [00001] and the write pointer at [00000] despite our efforts to precharge them both to the 0 starting value. As a result, the reads are not from the places we expect but we can demonstrate that the write and read operations do perform correctly (disregarding the pointers):

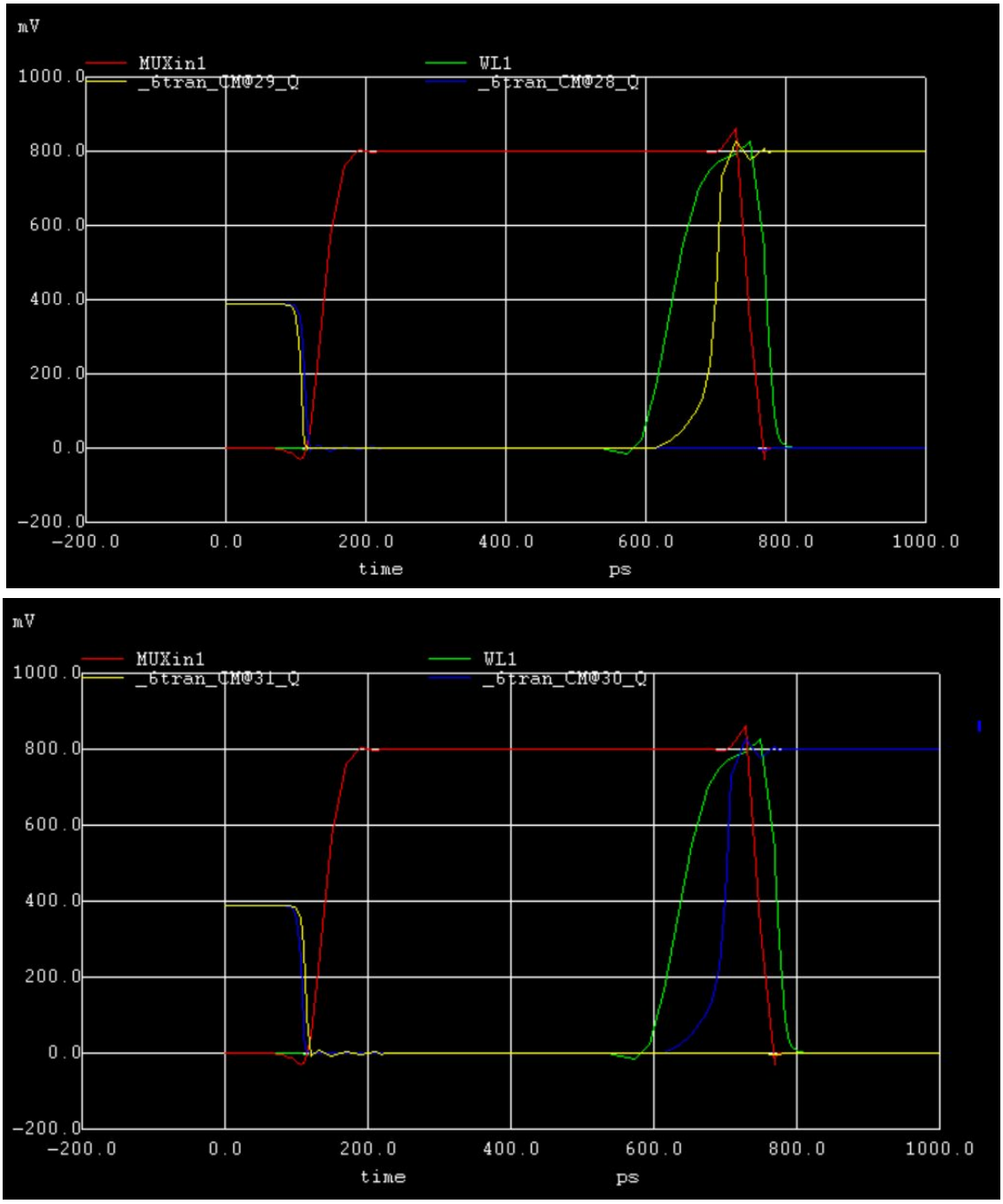


Figure. Line 1: Simulation results of writing 0110 to the 00000 address spot.

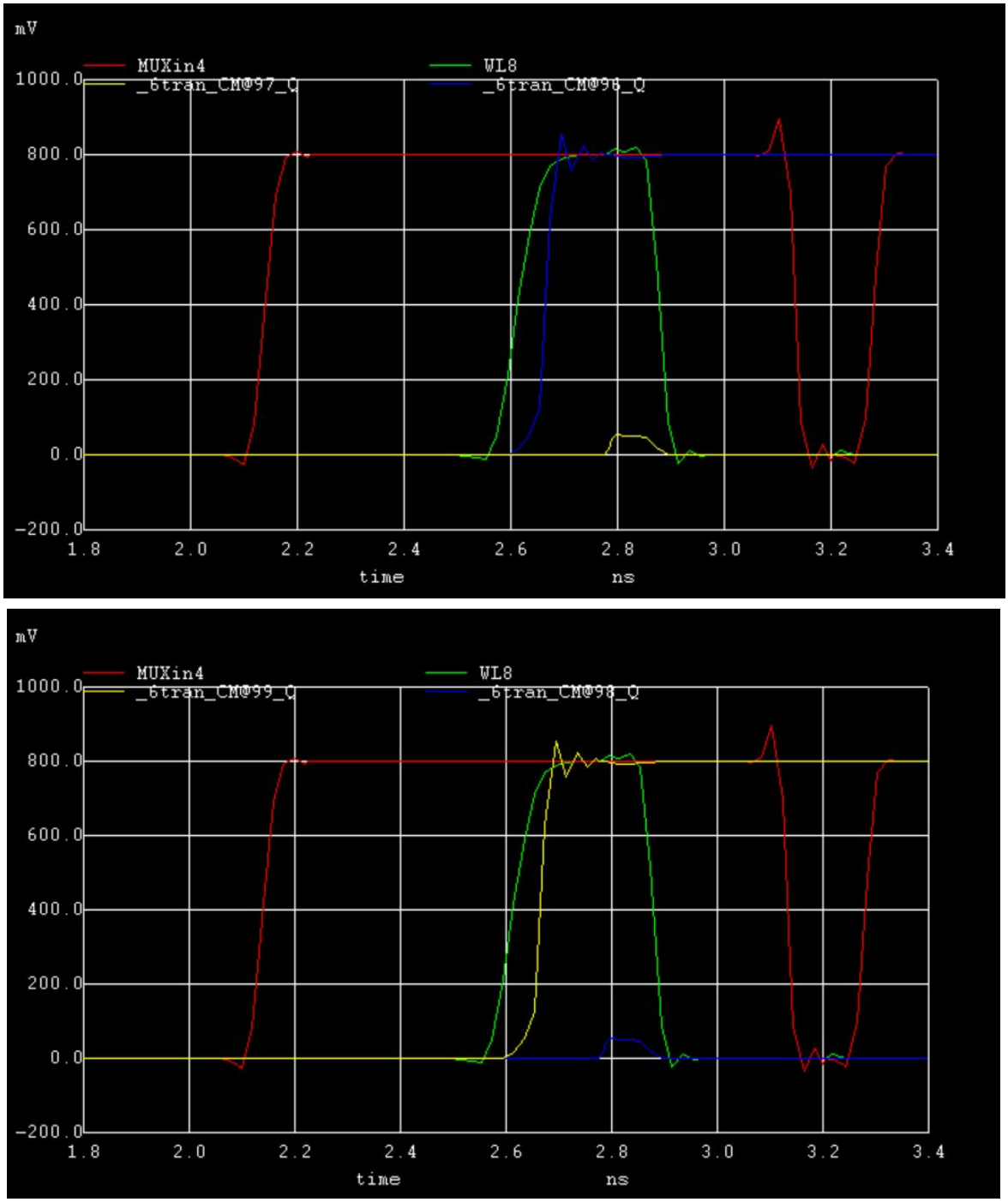


Figure. Line 2: Writing 1001.

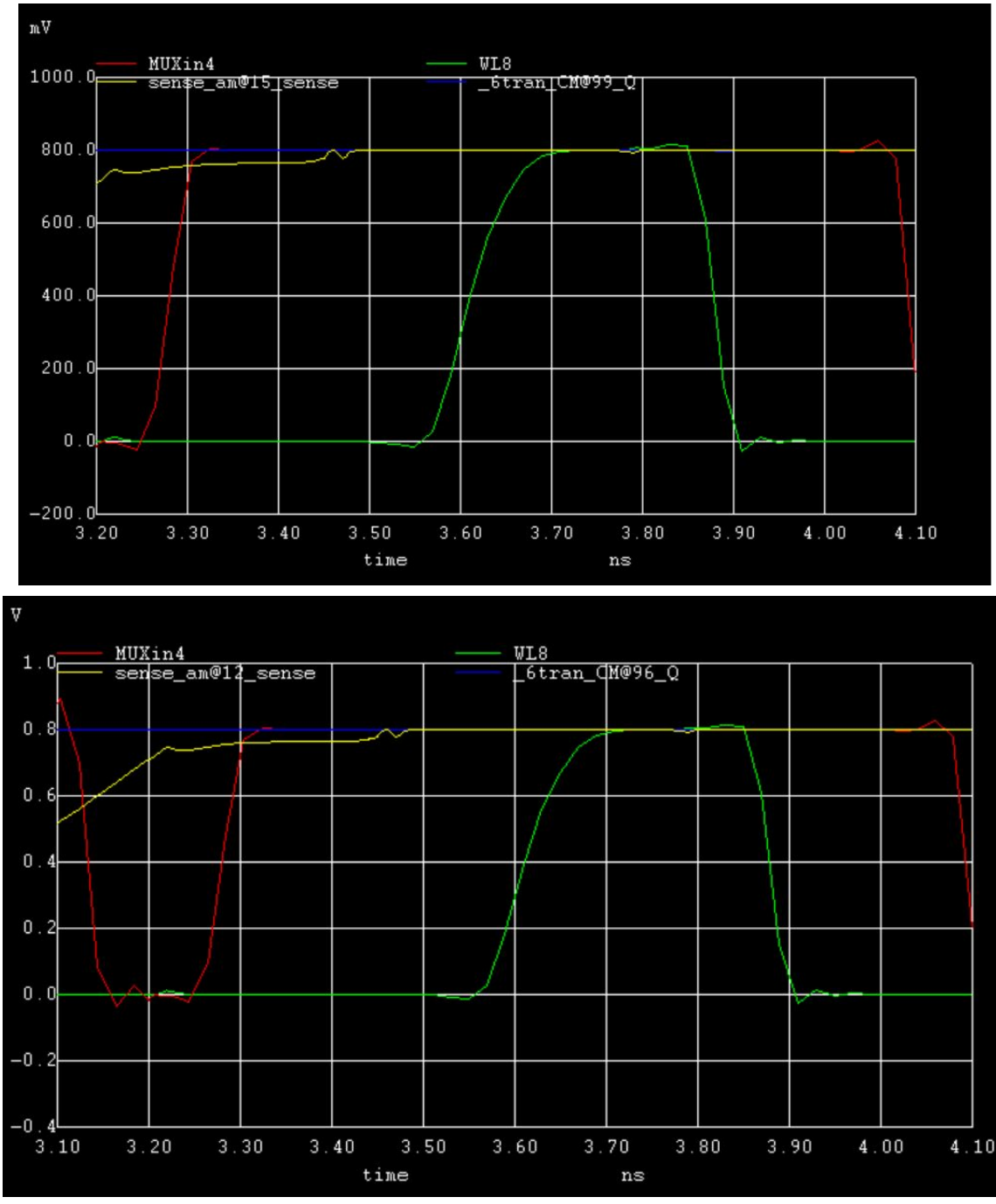


Figure. Line 2: Reading 1 from the memory cell.

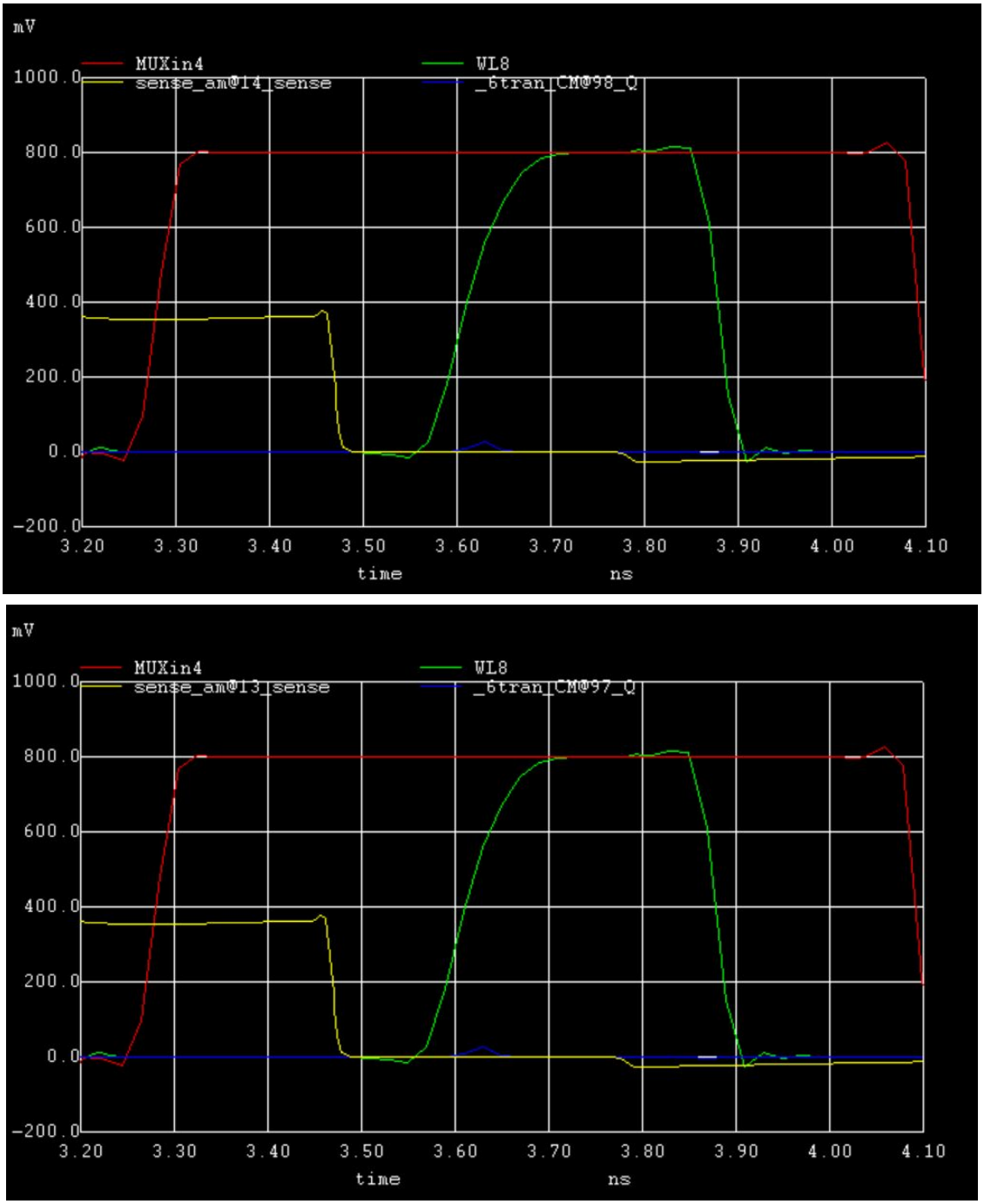


Figure. Line 2: Reading 0 from the memory cell.

Delay of critical path:

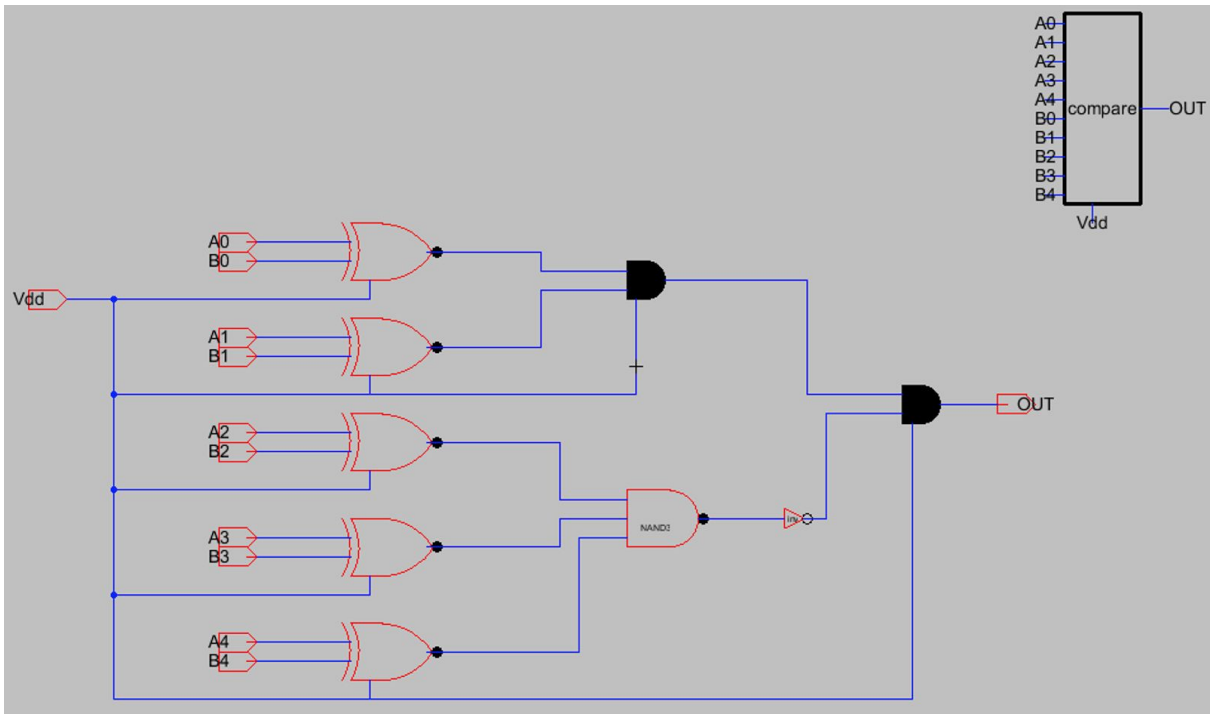
The main bottleneck in the WL. As noted in the timing section of this report, we needed WLclock to have a fairly long high in order to make sure that the cells would have enough time to charge up in the write component. We found that the sense amplifiers were practically instant, so that provided the least problem. PCclock and the clock to calculate pointer FULL/EMPTY took the second-longest amount of

time, as both required waiting for values to charge up and compare. Decoding the pointers would have taken longer if it was not performing in parallel with other longer calculations.

Memory operation and design choices:

We opted to keep everything minimum size in order to reduce capacitive load and thus energy usage. Also, we kept a Vdd of 0.8V to likewise reduce energy usage. Since components still worked within the time constraint with minimum size, we kept them that way. We used many simple gates in order to make components function as desired. For example, we used sequences of inverters in order to push needed clocks into desired timings.

Subcircuits not previously explained:



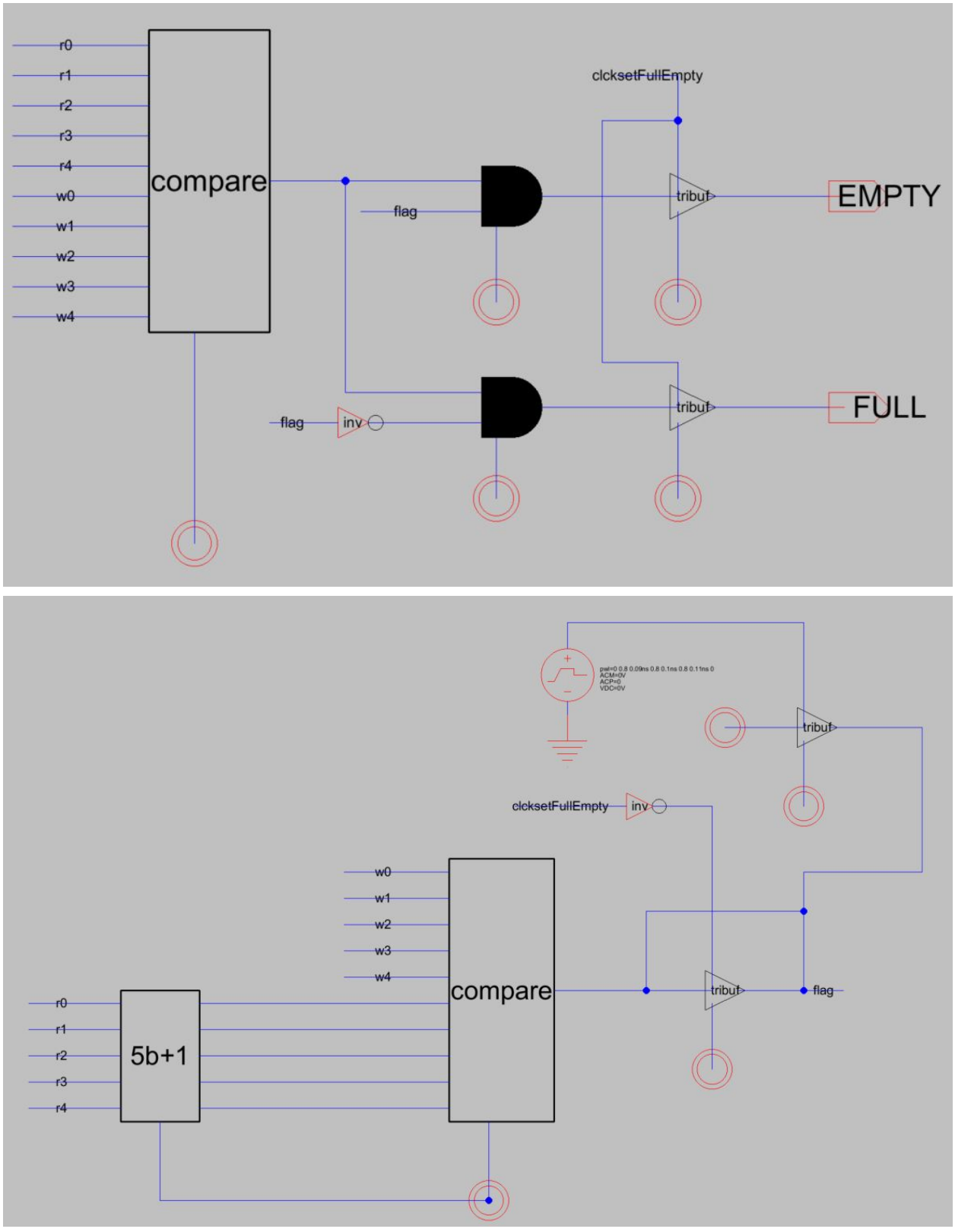


Figure. Comparator and flags used to determine EMPTY/FULL.

The flag is set in every cycle to point out whether or not the read pointer is about to catch up to the write pointer. Then once the pointers are the same, the circuit can determine whether it is FULL or EMPTY.

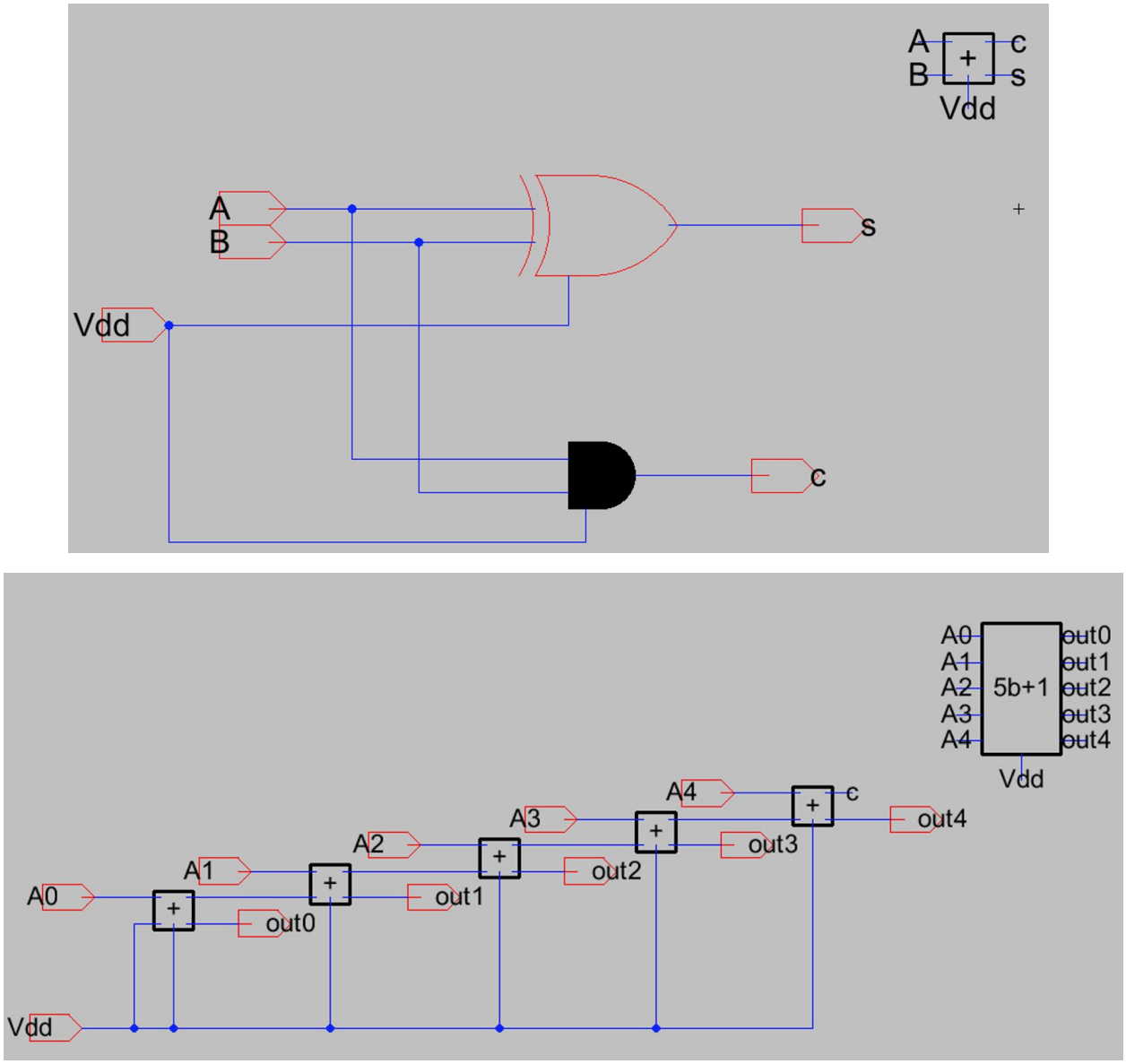


Figure. Adder circuitry for 5-bit number.

The adder is used in the comparator function. We use half-adders to create the overall 5-bit adder, so that pointers move around in a “circular” fashion.

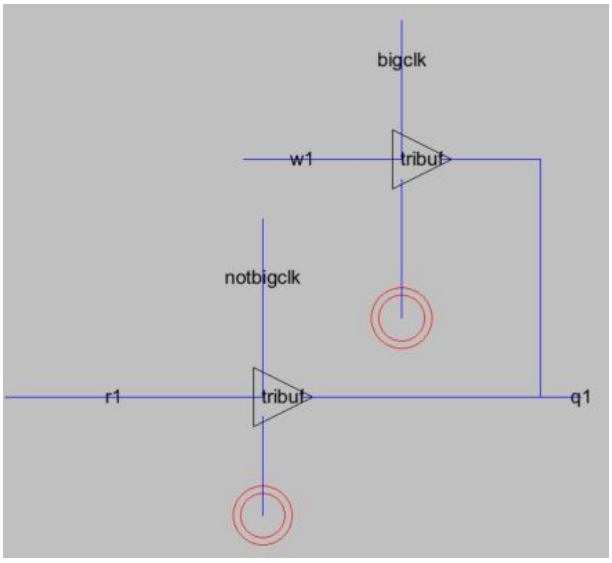


Figure. Circuitry used to determine which address to put into the decoder at certain parts of the cycle.

In the first half of the cycle, we want the decoders to decode the write address. In the second half, the read address. So that we write to where the writer is pointed (if we can/want to write) and read from where the reader is pointed (if we can/want to read).

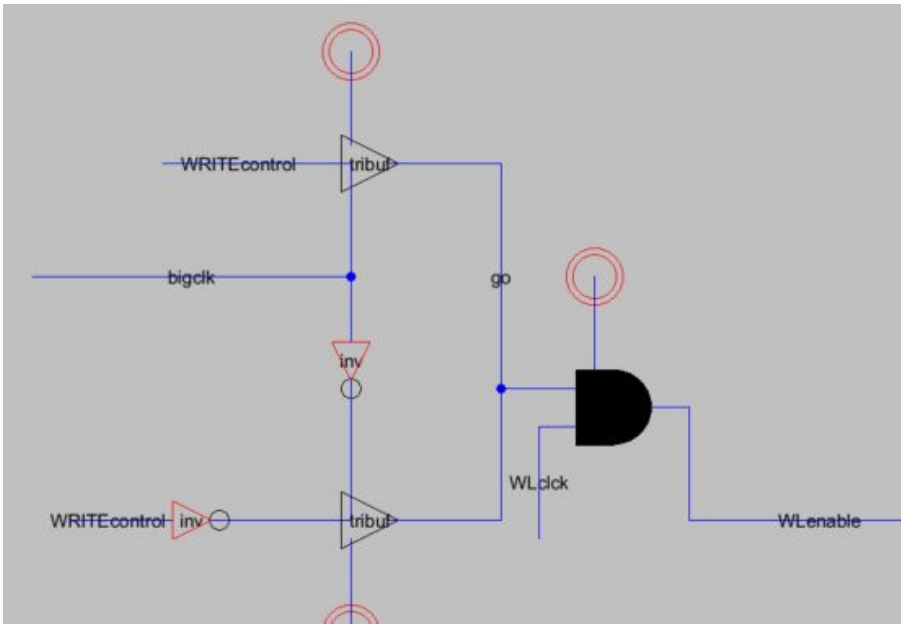


Figure. Circuitry used to enable decoder.

We want to enable the decoder (and thus the corresponding word line) only during the WLclock time, and only if we can/want to write/read.

For the WL control:

We find that we needed to provide enough time for the WL to reach its full potential. Through testing, we find that we want to provide at least ~0.3ns for the WL to reach its value.

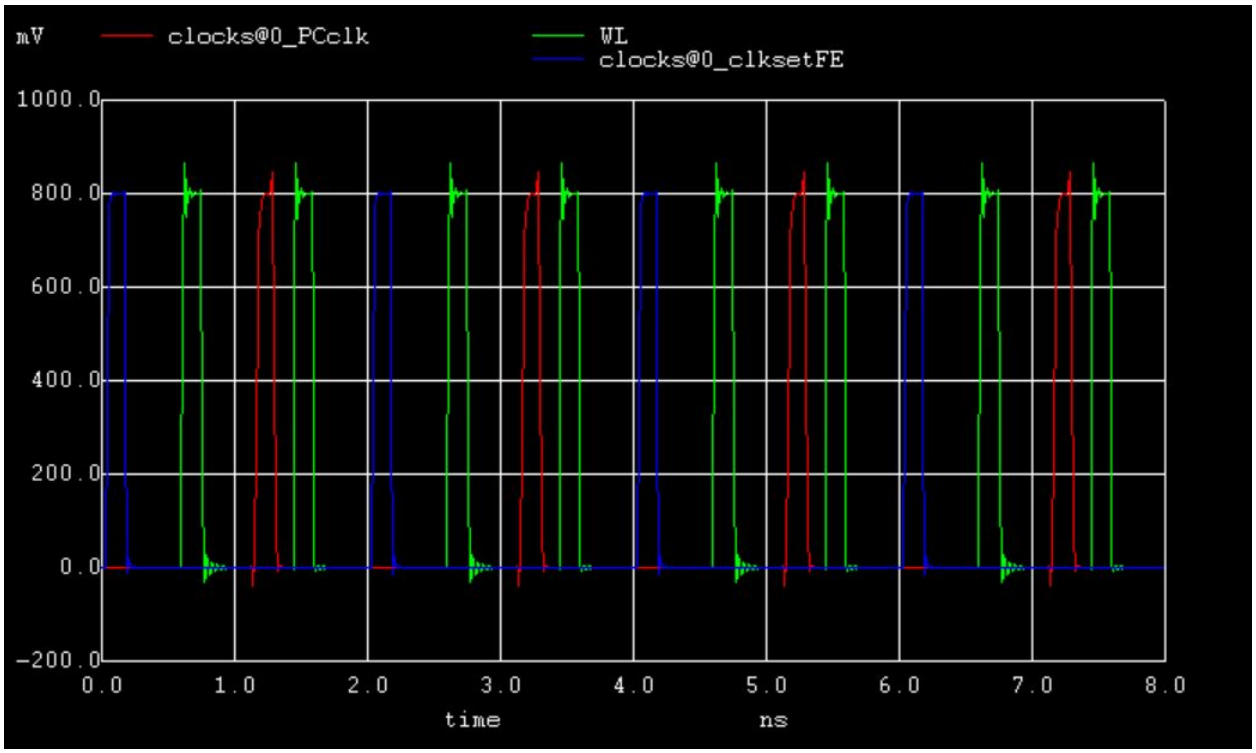


Figure. Initially, there was not enough time given to drive WL.

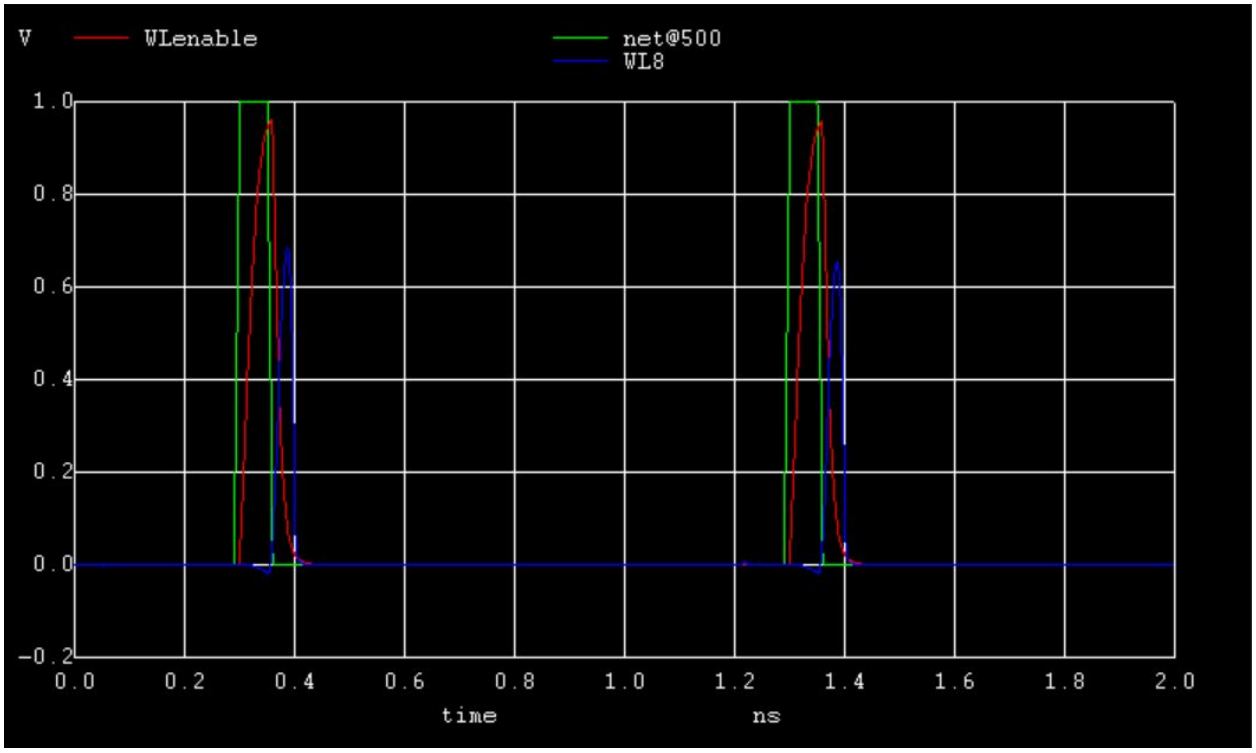


Figure. With ~0.05ns, the WL fails to reach its full value, and the WL is not driven enough.

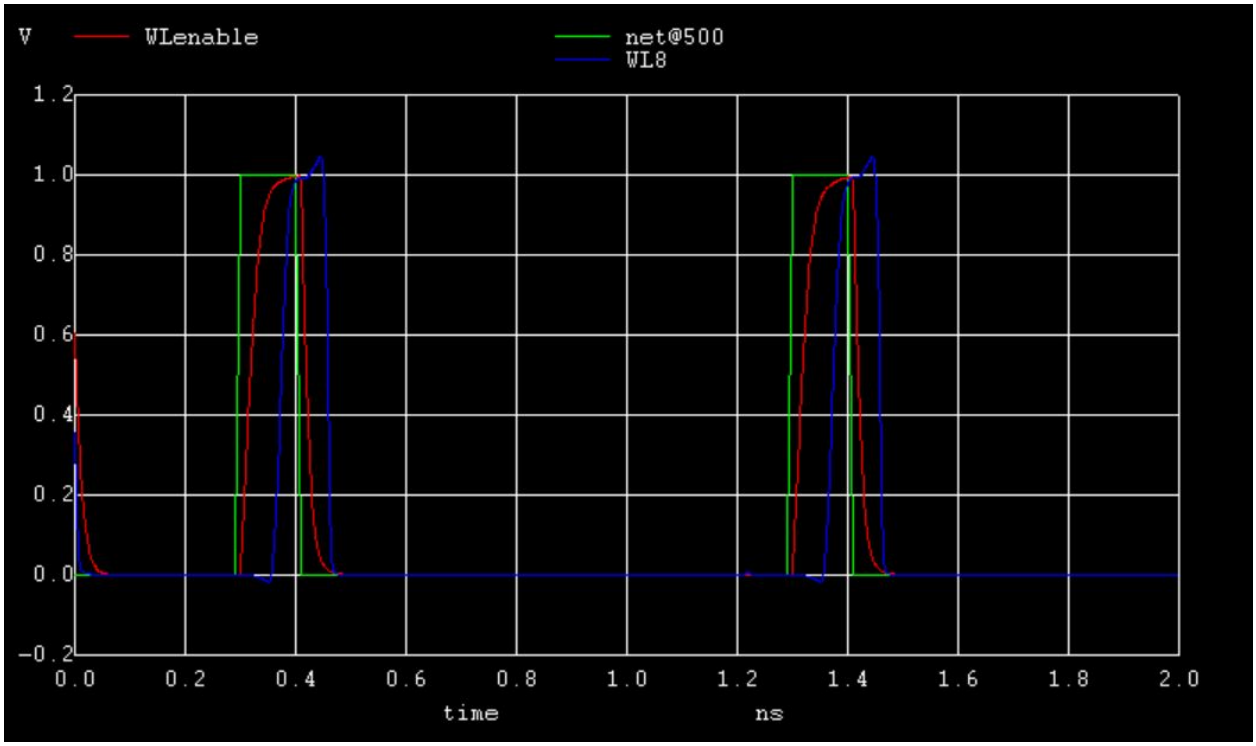


Figure. With ~0.1ns, the WL reaches its full value.

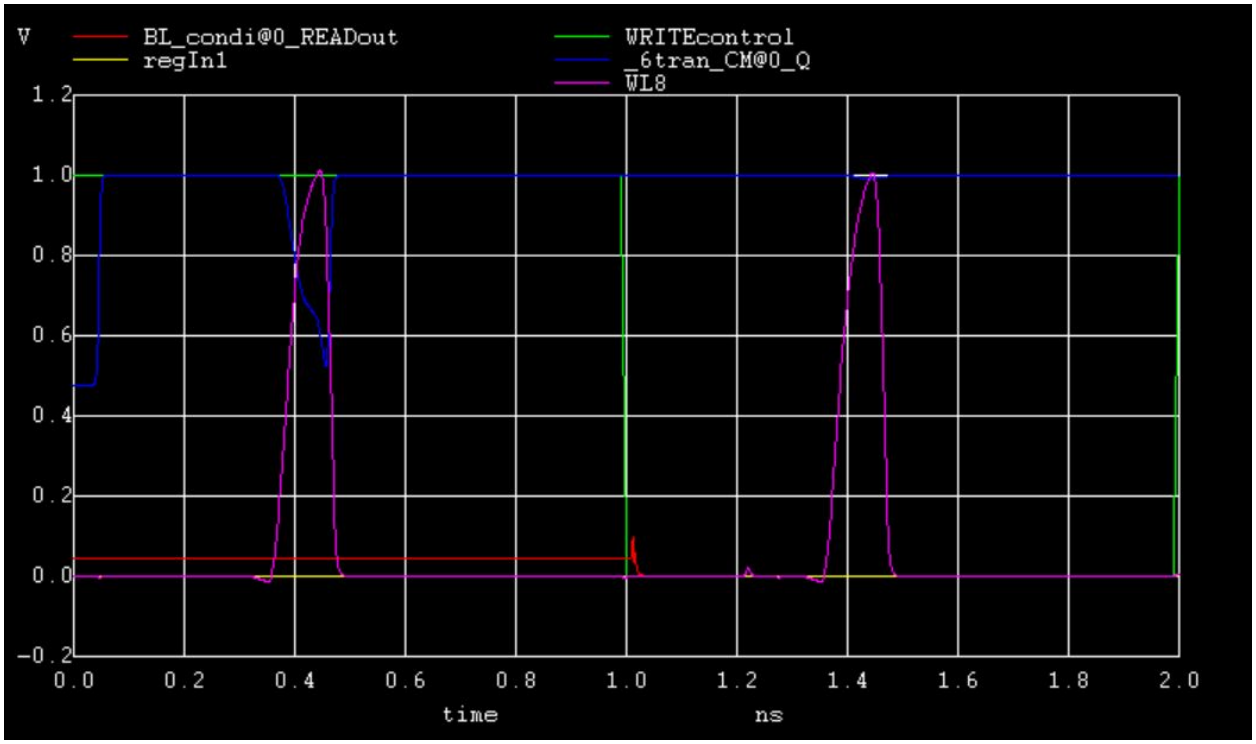


Figure. After all 32 words are put in, ~0.1ns WL is not enough time for proper writing full value.

5-bit counter:

The pointers store the address of the current word being read or written to. We implemented the pointers as a 5-bit synchronous binary counter using master-slave positive edge triggered JK Flip-Flops. The master and slave were controlled using the 2-phase non-overlapping clock generator.

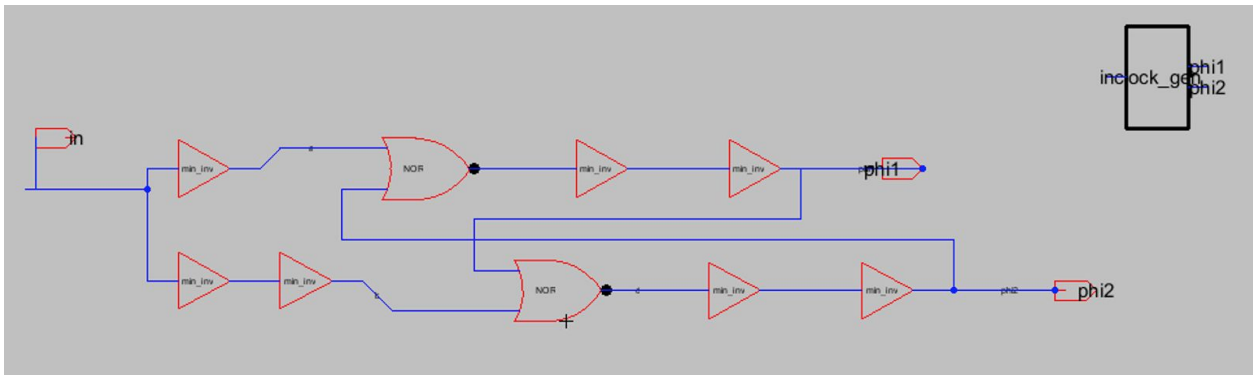


Figure. Clock generator.

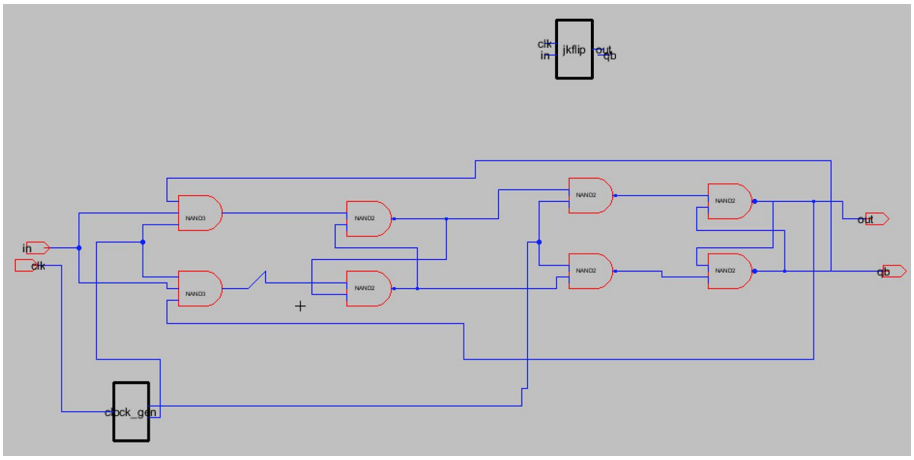


Figure. Flip-flop

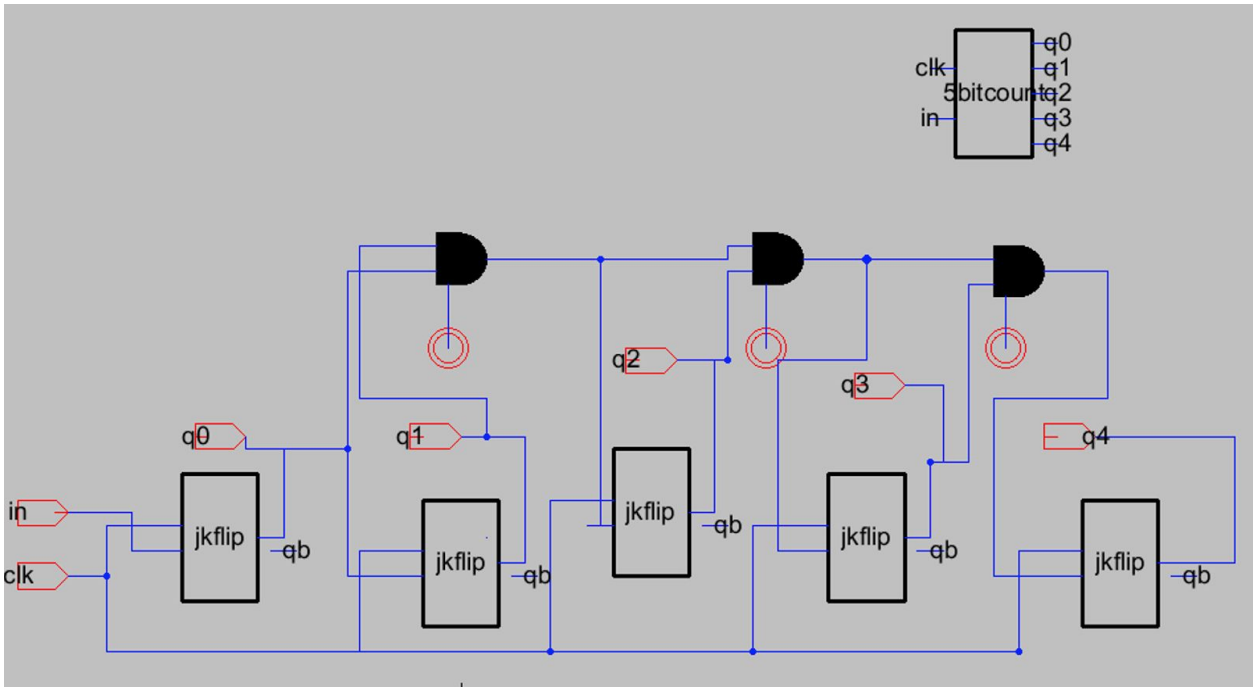
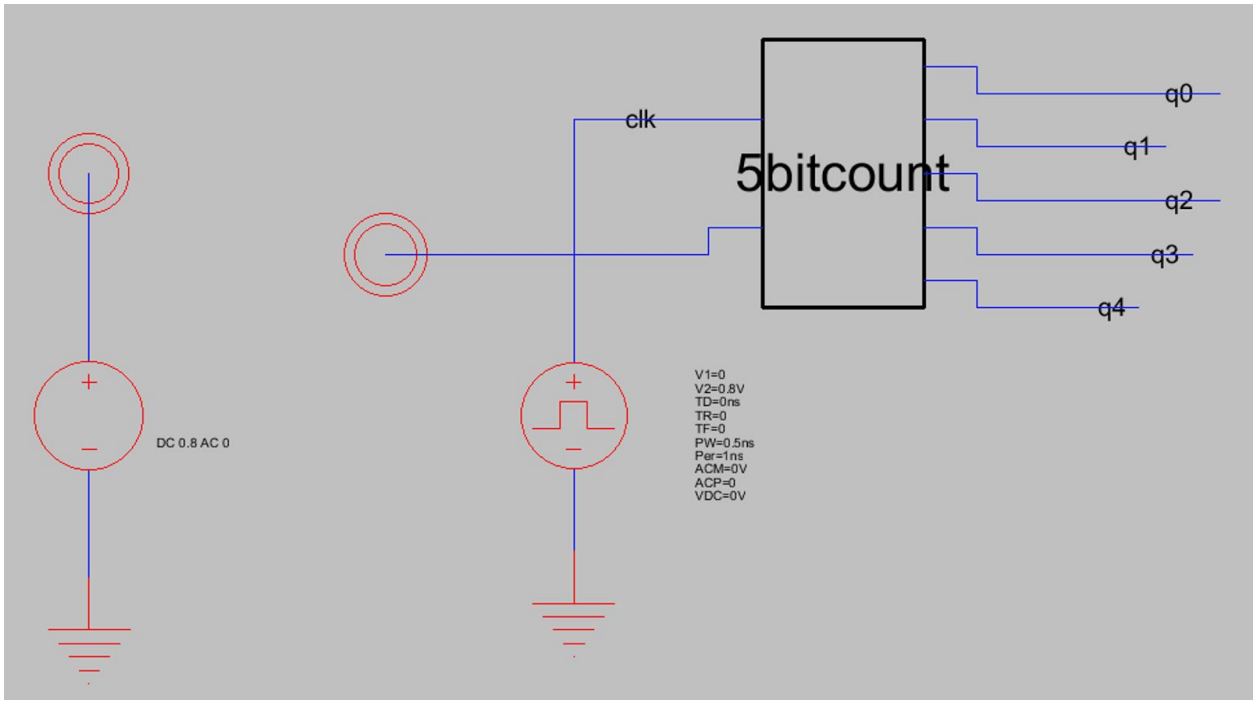
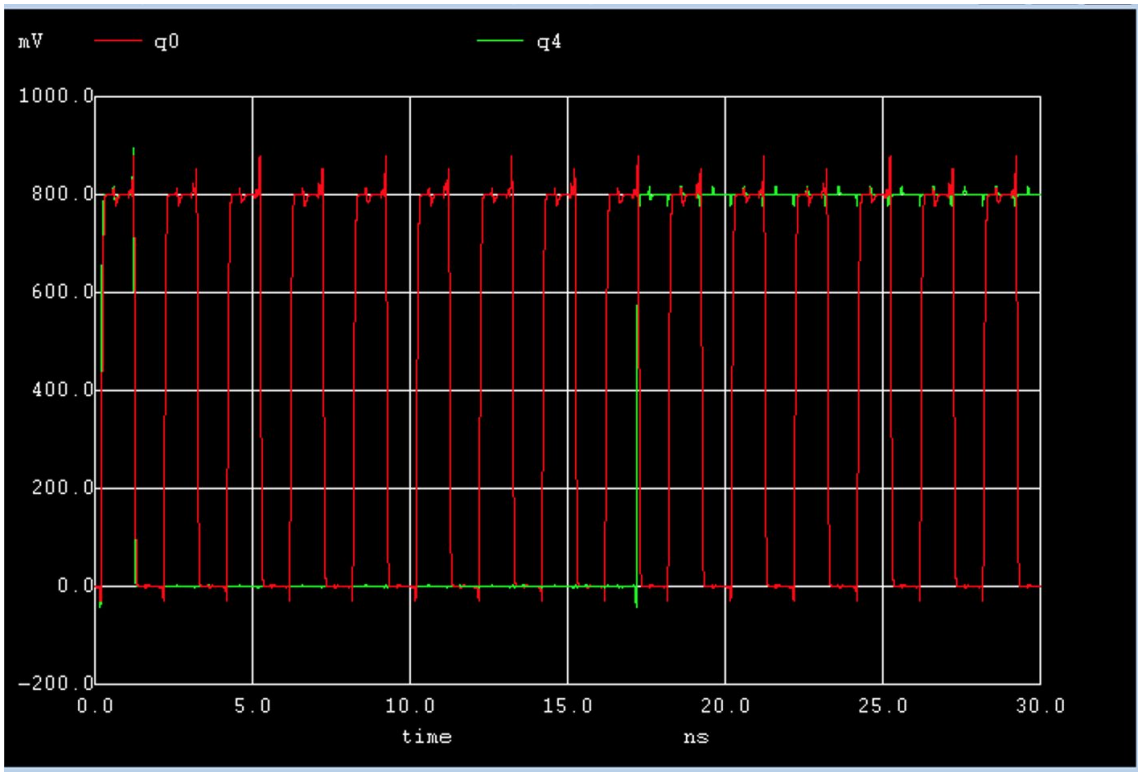


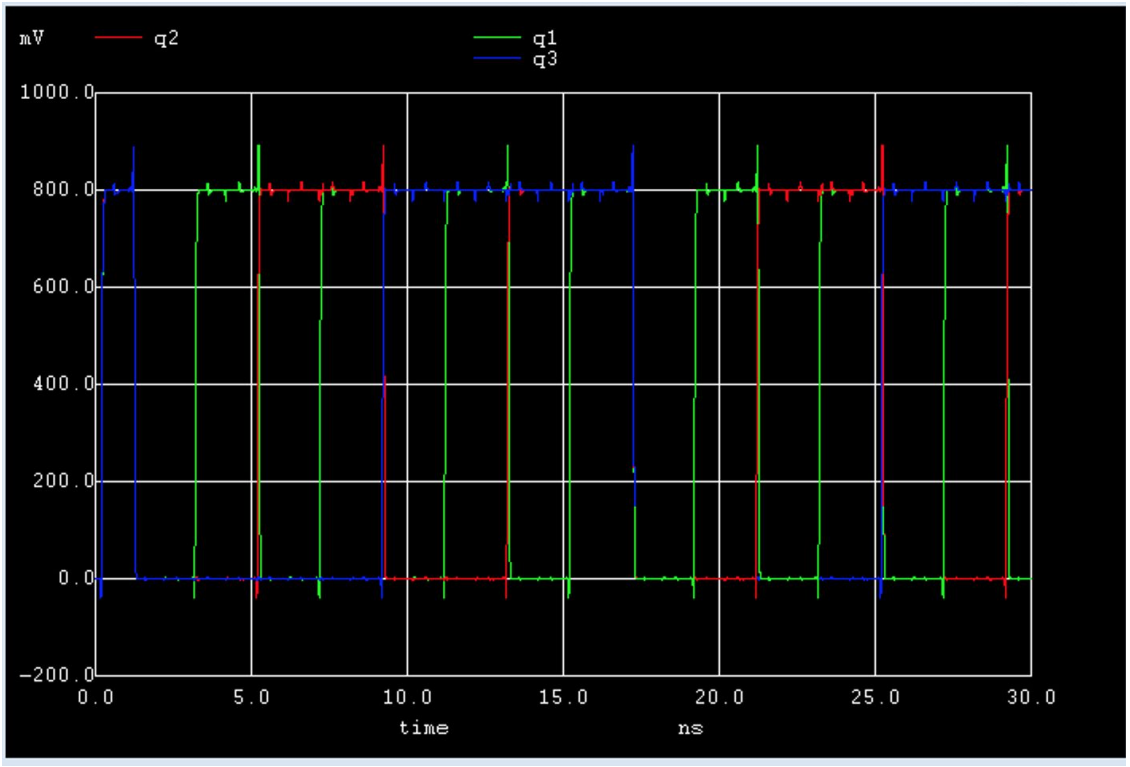
Figure. Total 5-bit counter.

The following test bench was set up:



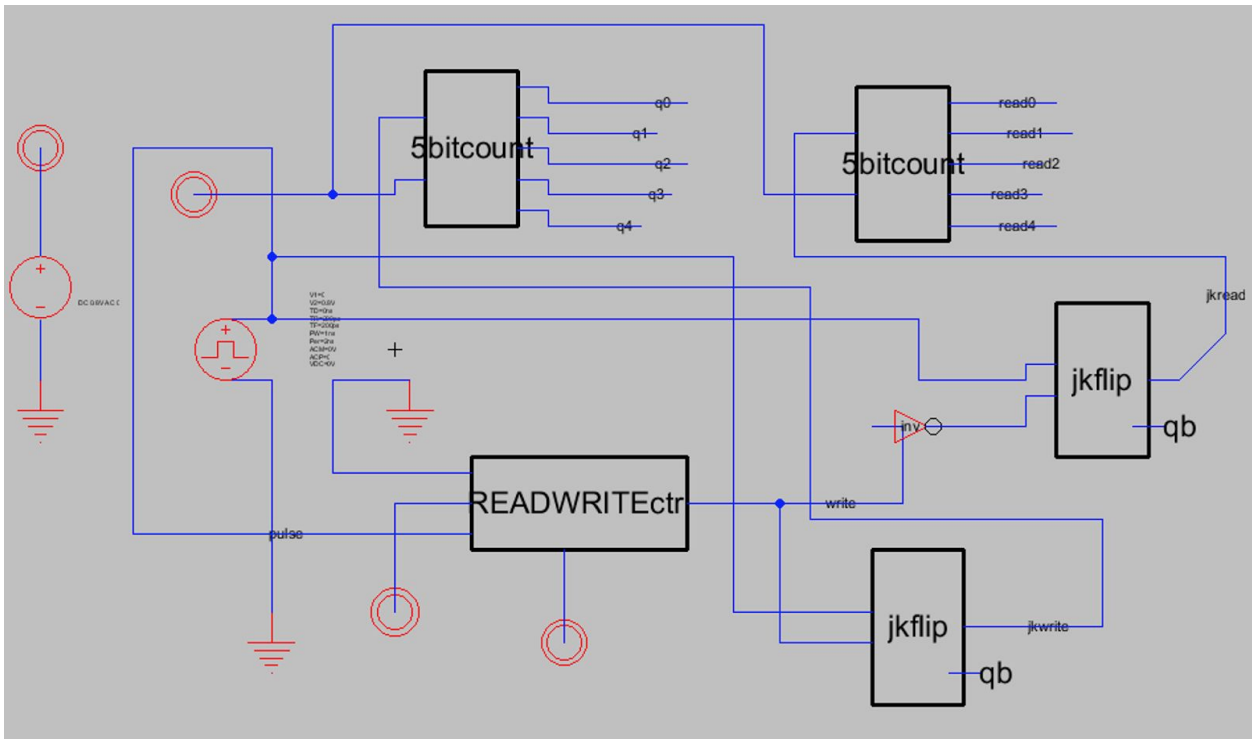
A logical high (vdd) is connected at the input of the counter and the values stored increment on each clock cycle. Q0 is the least significant bit and q4 is the MSB. The binary counter starts at 1.2ns which is when all outputs are 0, i.e. 00000. It then increments on each clock cycle i.e. 00001 then 00010...until 11111 at which point it resets and starts again. The results of the simulation shown for 30 clock periods is shown below





[explain rationale for memory cell sizing. Provide any supporting simulations]

Read and Write Pointers



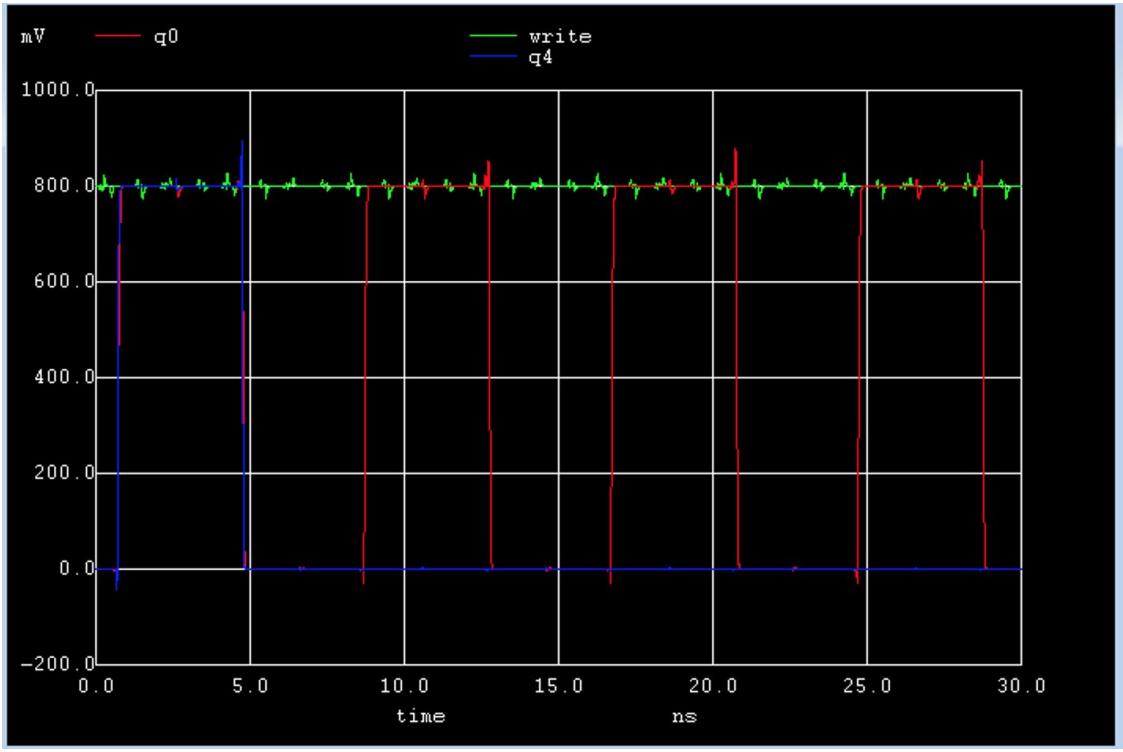
The read and write signals are generated by the READWRITEctr block. When the output(write) is high then a high input is fed into the write flip flop and a low input is fed into the read flip flop. When a high input is fed into the flip flop, it toggles and acts a clock input to the counter and increments on each cycle. When a low input is fed, the flip flop latches onto the previous output and hence does not toggle and the counter it's connected to does not increment. The flip flops essentially act as clock inputs for the synchronous 5-bit counter. The left counter represents the write pointer and it stores the current memory address of the write pointer and the counter on the right corresponds to the read pointer. The read pointer is incremented on a read cycle, which is when the output of the READWRITEctr is low and hence the inverted input into the read flip-flop will be high, enabling the toggling operation and incrementing the counter on each cycle.

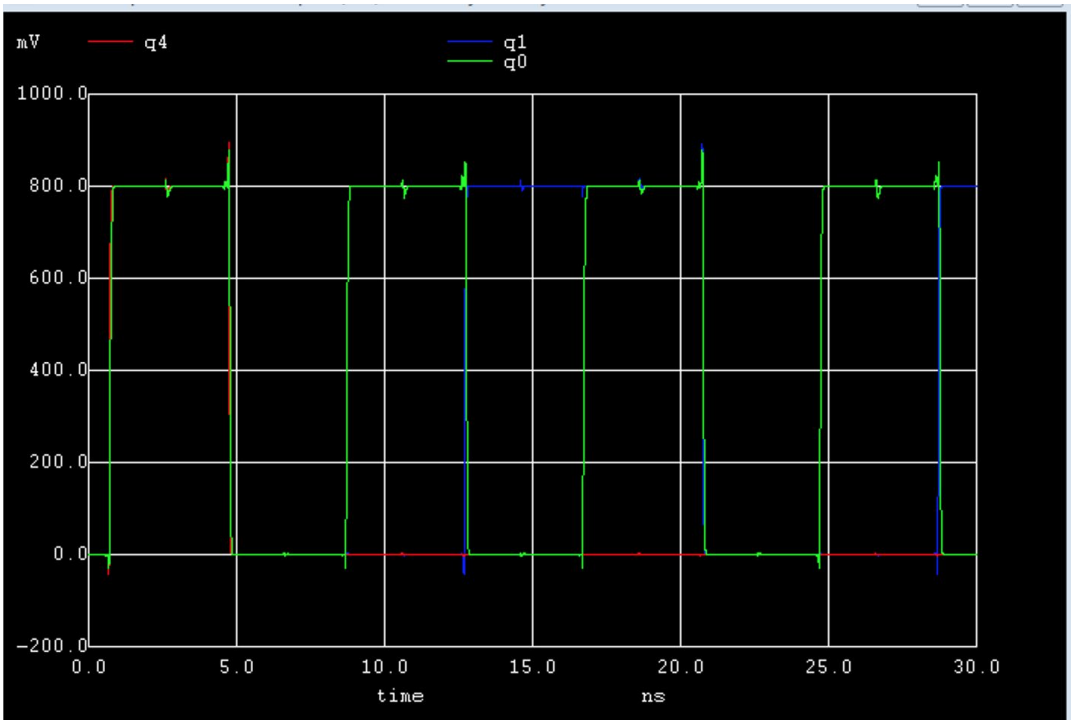
The results of the simulation are shown below:

Case 1: Write is high and Read is low

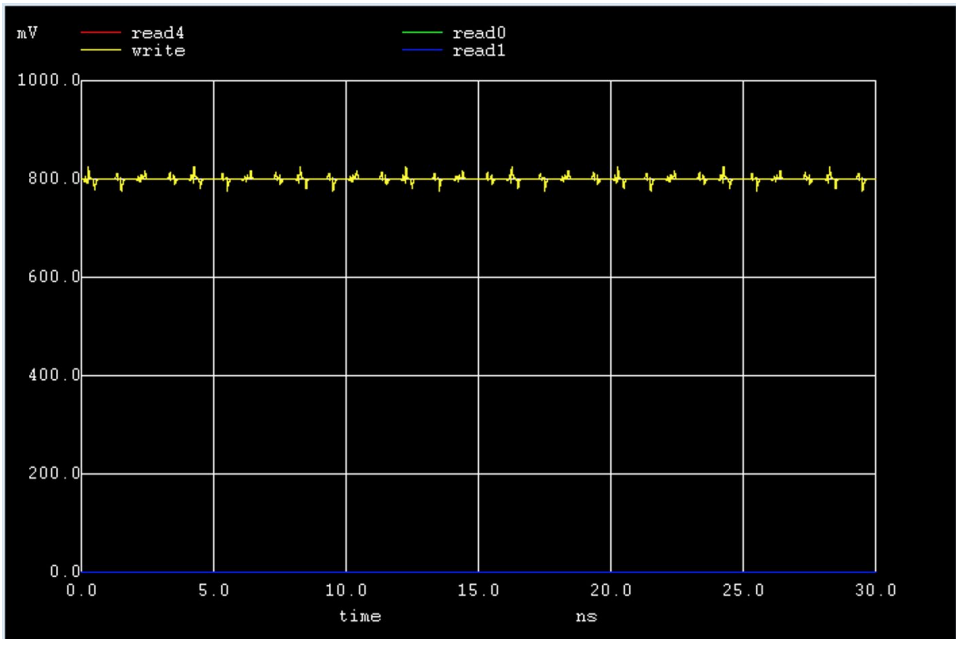
Write pointer/counter:

This correctly increments the write pointer.g



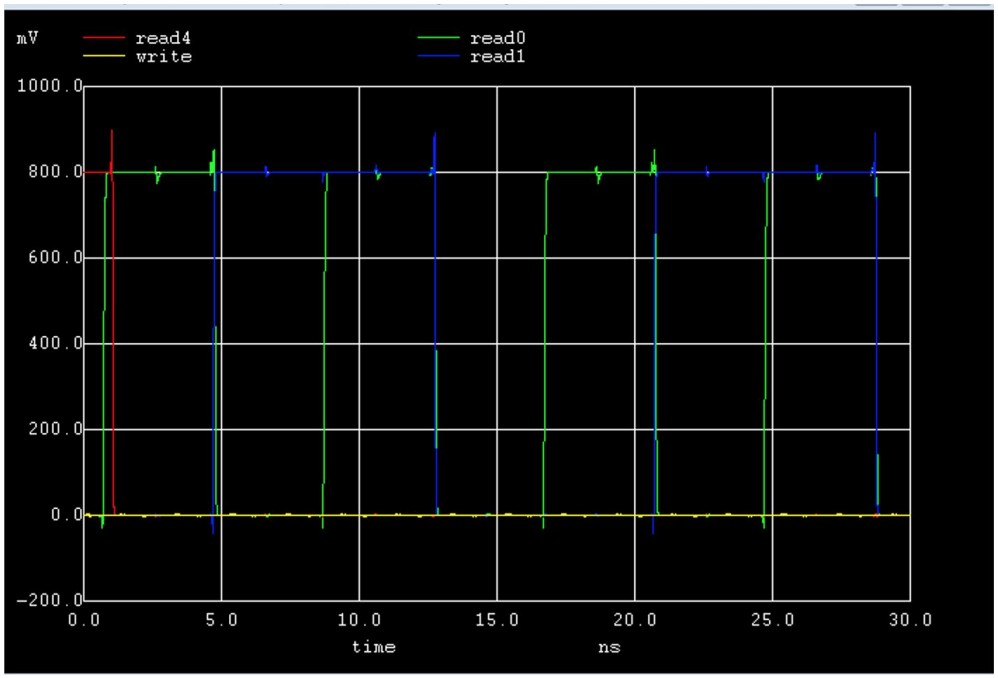


Read pointer/counter:



Case 2: Write is low and Read is high

This correctly reads and increments the read pointer.




```
ngspice 85 -> meas tran yint integ I(vv_generi@0) from=0ns to=2ns
yint                =  -1.29090e-14 from=  0.00000e+00 to=  2.00000e-09
ngspice 86 ->
```

-ready- Quit

The energy used in incrementing the pointers is 12.9J * 0.8 = **10.32fJ**

Energy

The energy was calculated by measuring the integral over the clock period and multiplying by vdd (0.8V). For standby energy, both enqueue and dequeue operations are off i.e. nothing is happening.

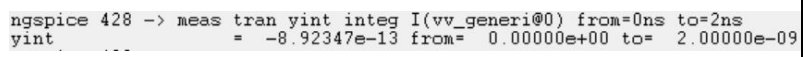
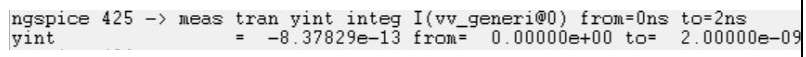
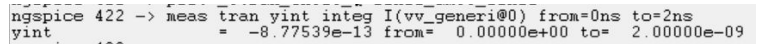
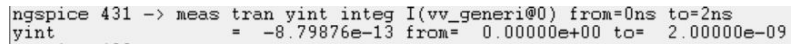
Summary of Energy Contributions

Component/Function	Energy
Incrementing and storing in pointer	10.32fJ
Comparator	0.092fJ
Sense Amplifier	1.72fJ

Bit-line Conditioning	41.12fJ
Single SRAM - write read read	0.556fJ
Energy of clock generation	0.026pC*0.8V = 0.0208pJ

Summary of design metrics:

Since we are using the 22nm technology, 1 width of 1 corresponds to 22nm. The total transistor width in a memory cell is 6 and for a total of 128 cells this gives 768 for the cell width, which equals 768*22 = 16.9um.

Area	132um
Memory Cell Area	16.9um
Enqueue Energy	 0.892pC*0.8V = 0.714pJ
Dequeue Energy	 0.838pC * 0.8V = 0.67pJ
Enqueue/Dequeue Energy	 0.878pC * 0.8V = 0.702pJ
Standby Energy	 0.889pC*0.8V = 0.711pJ
Average Energy	15% enqueue/dequeue, 10% enqueue, 10% dequeue, 65% standby operation = 0.706pJ

Statement of Academic Integrity

I, Celine Lee, certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this project.

I, Ransford Antwi, certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this project.