UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DELAY-BASED SRAM CONTROL LOGIC IN OPENRAM

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Samuel Crow

December 2023

The thesis of Samuel Crow
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Delay-Based SRAM Control Logic in OpenRAM

by

Samuel Crow

OpenRAM is a configurable SRAM compiler which can be ported to many PDKs. As such, increasing possible configurations increases the number of such PDKs that can be used. We present a new option for OpenRAM memories to use an inverter delay chain for control signal timing instead of replica bitline timing. This option increases the number of PDKs to which OpenRAM can easily be ported. This thesis presents the design and implementation of this new control logic in OpenRAM. We also present a 1KB dual-port SRAM macro with this control logic taped-out for fabrication on a multi-project wafer.

Acknowledgments

Thank you Matthew Guthaus for teaching me about chip design, SRAMs, and so much more. Thank you Tim Edwards for helping me to understand why my LVS was failing. Thank you Jesse, Bugra, Hadir, and Eren, each of you helped me quite a bit with OpenRAM. And thanks Scott Beamer and Heiner Litz for teaching me about HDLs and for reading this thesis.

Chapter 1

Introduction

OpenRAM [4] is an open-source Static Random Access Memory (SRAM) compiler implemented in Python. OpenRAM has a control logic module to send precisely timed signals to various components of the SRAM based on the inputs, in order to read or write its array of memory cells (bitcells). By default, OpenRAM uses a replica bitcell technique [2] that has a column of specially designed bitcells (replica bitcells) that output a fixed logic low value and the replica bitline (RBL) to approximate a fraction of the time required for the SRAM to complete a read or write operation. Because it uses replica bitcells, RBL control logic tracks variations in process, voltage, and temperature (PVT) better than alternatives.

While the RBL control logic functions correctly in the PDKs supported by OpenRAM out of the box, it poses a challenge to port OpenRAM to other technologies for which the PDK provided by the foundry does not include replica bitcells. In such a case, OpenRAM would need an alternate method to generate the timing of the control signals that does not rely on replica bitcells. We present a new design for OpenRAM's control logic that implements an alternate method for setting the timing of the internal control signals using glitch generation and a chain of inverters. Furthermore, OpenRAM's implementation of the RBL technique uses the first half of the clock cycle to precharge and the second half for reading or writing. Our new design does not depend directly on the clock's phases for timing.

Our contributions to OpenRAM are available publicly at:

https://github.com/VLSIDA/OpenRAM

The rest of this thesis is arranged as follows. Chapter 2 discusses the RBL-based control logic. Chapter 3 discusses the delay-based control logic. Chapter 4 discusses changes to the bitcell array. Chapter 5 discusses an SRAM with the new control logic which is currently in fabrication. Chapter 6 compares simulated characterization results for the taped-out macro to one with RBL logic. Chapter 7 discusses future work. Chapter 8 summarizes the contributions of this thesis.

Chapter 2

Background

2.1 Replica Bitline

OpenRAM's RBL-based control logic uses replica bitcells which are a modified version of the bitcell which always stores a logic low. This means that the cell's bitline (BL) will always be pulled to GND from its precharged voltage of VDD. The remainder of the bitcell including access transistors remains unchanged.

The "dummy bitcell" is another modified version of the bitcell, but whose storage element is disconnected from the bitlines. However, its access transistors provide word-line load so that it can replicate word-line delay similar to a row of regular SRAM cells. The dummy bitcells are used in a dummy row with an additional replica bitcell in the replica column to replicate wordline timing.

For each port, a column of replica bitcells abuts the array of bitcells on the same side where the control logic is placed (left or right). The column of replica cells has an additional replica cell placed above or below it (the "replica bit"), which extends past the top or bottom of the bitcell array in the dummy row. This entire column of replica cells is called the "replica column" and their collective bitline is called the "replica bitline" (RBL). This replica technique was introduced by Amrutur et al [2]. The replica bit is then followed to the right by a number of dummy cells equal to the bit-width of the bitcell array, which load the replica bit's wordline (*rblwl*) similarly to the other wordlines. An example layout for this array in a dual-port configuration is shown in Figure 2.1.

2.2 RBL-based Control Logic

The RBL-based control logic takes 3 signals as input: an active low write enable (*WEb*), an active low chip select (*CSb*), and a clock (*CLK*). Using a D Flip Flop (DFF), each of the active low signals are captured at the positive clock edge. The outputs of these DFFs are chip select (*cs*) for the *CSb* input and write enable (*we*) for the *WEb* input. AND gates and inverters are used to generate signals useful for the control logic: a gated clock (*gc*) and its inverse (*gcb*).

Using the internal signals *cs*, *we*, *gc*, *gcb*, and the inverse of *we* (*web*), the control logic produces its output signals. These signals are the wordline enable (*wlen*), the write driver enable (*wden*), the active low precharge enable (*penb*), and the sense amp enable (*sae*). *wlen* is generated by connecting *gcb* to a driver which is sized according to the number of wordlines (number of rows in the bitcell array). This means

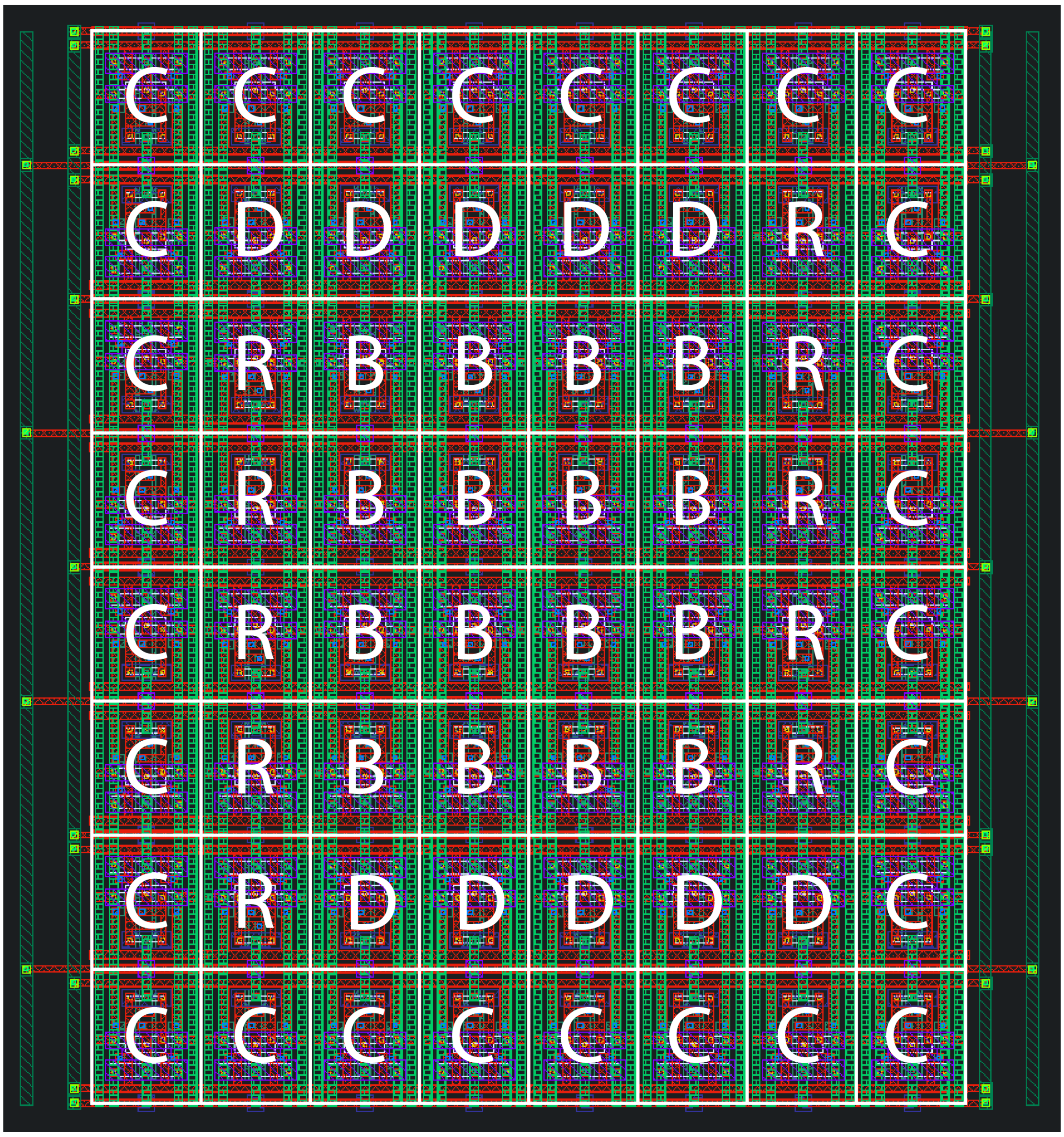

Figure 2.1: The layout of a dual-port bitcell array with replica columns. B: bitcell, C: cap cell, D: dummy bitcell, R: replica bitcell

that the wordline is always activated during the negative half of the clock cycle.

During a read operation, the *wlen* signal activates the wordline driver for the row being accessed (as determined by a hierarchical address decoder). This wordline driver also activates the wordline of the adjacent replica bitcell in the replica column. Similtaneously, *wlen* enables the wordline driver connected to *rblwl*. The result is that two replica bitcells in the replica column pull the RBL down to ground, roughly halving the time of the RBL's bitline swing. This RBL signal is sent back to the control logic where it is delayed by an inverter chain whose output is *rbld*. The *rbld* signal is used to generate the *penb*, *wden*, and *sae* signals such that precharging does not coincide with writing or reading. Thus, the sense amplifiers (which are on the critical path) activate with the rising edge of *rbld*, which indicates a valid differential result is visible on the bitline pairs. A schematic for the RBL-based control logic is shown in Figure 2.2. This design has been silicon verified by Cirimelli-Low et al [3].

OpenRAM's primary motivation for using RBL as the basis for the timing of the *sae* signal is to maintain performance across PVT corners. The critical path for the SRAM's read operations is from the control logic to the wordline to the bitline swing as identified by Nichols et al [6]. As such, RBL timing ensures minimum performance is lost in nominal and best case scenarios while guaranteeing correctness in worst case scenarios. This is possible because the RBL tracks variation in the bitline swing, because it approximately replicates the bitline.

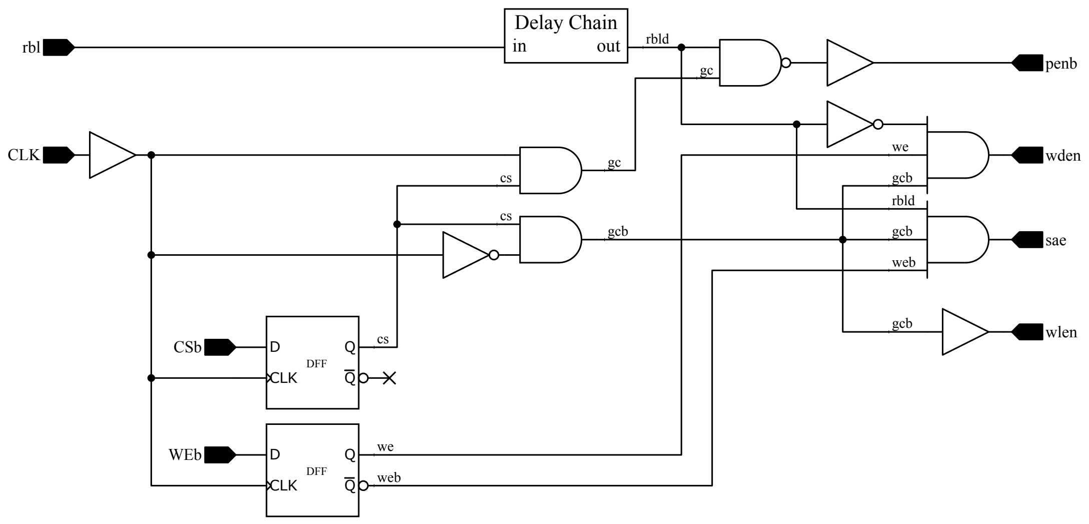

Figure 2.2: Schematic representation of RBL-based control logic

Chapter 3

Delay-Based Control Logic

3.1 Logic Design

With the existing RBL-based control logic in mind, we sought out a solution for the problem of how to time the control signals without RBL. Another consideration was the possibility of future work implementing asynchronous timing to the control logic (removing the clock). On its own, the lack of RBL would lend itself to the idea of using a chain of inverters connected to the clock to replace the RBL signal. In fact, this was the state of the art when RBL was proposed as a better way to track bitline swing across PVT corners. Because inverter chains do not track PVT variations well, control logic timings have to be designed around the worst case. Thus, inverter chains tend to exhibit slower performance than the RBL alternative. Despite its apparently lacking performance compared to RBL, inverter chain timing has an advantage related to the secondary consideration of asynchronous control timing.

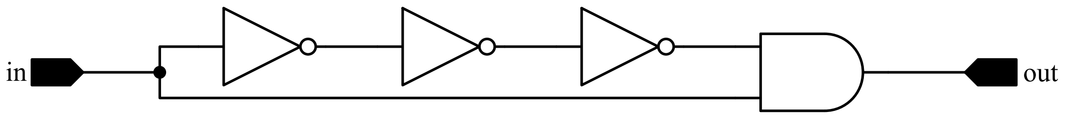

Figure 3.1: Schematic representation of ”glitch” circuit. Increasing inverter chain delay increases glitch pulse duration.

3.1.1 Glitches

Using an inverter chain and AND gates, one can create ”glitches” from a single input signal with a certain pulse width. A schematic illustration of such a circuit is shown in Figure 3.1. By varying the number of inverters in the delay chain, the pulse width of the glitch can be varied. By changing the start time and pulse width, we can use glitches to control the timing of the control signals. To control the SRAM using the delay-based control logic, we generate several glitches from a single signal (gc).

Several glitches can be generated from a single input signal. In a synchronous SRAM, the input signal is the clock (see Chapter 7 for discussion of asynchronous SRAM). With the ability to vary the pulse width of each glitch, it seems feasible to use glitches to determine the timing of all the control signals. With this general idea, we go about designing glitch-based control logic.

3.1.2 Multi-Delay Chain

Because we only use a single input signal to generate our glitches, we only need a single inverter chain. From this inverter chain, different stage inverter outputs can be used for making each glitch with differing phases. OpenRAM already implements

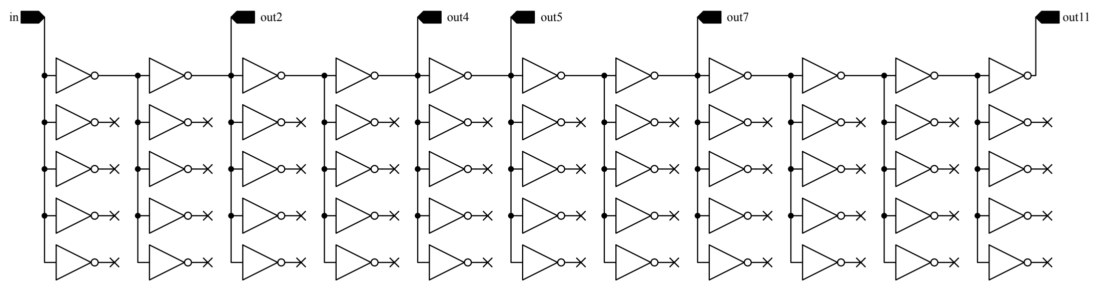

Figure 3.2: Schematic representation for the multi-delay chain circuit with outputs after 2, 4, 5, 7, and 11 delay stages. Each delay stage has a fanout of 5 inverters.

a delay chain for the RBL enable signal, so we modify this for the new control logic. We expose the output pins of some inverter stages internal to the chain, resulting in different phases of *gc* (i.e., delayed by different amounts). Using these delayed signals, we control the start and end times of each glitch. We call this modified delay chain a multi-delay chain (MDC). Figure 3.2 shows a schematic for this circuit.

3.1.3 Delay Control Circuit

In determining how to use the glitch signals in the control logic, we consider the necessary relationships between control signals. One such relationship is between *penb* and *wlen.* If the precharge drivers are actively driving VDD on the bitline pair of a bitcell when it is accessed, its stored data can be corrupted. To avoid this, we must ensure that precharging and bitcell access are mutually exclusive. Therefore, the falling edge of *wlen* must lead the falling edge of *penb* and the rising edge of *wlen* must follow the rising edge of *penb.*

Similarly, rising edges of *wden* and *sae* need to arrive some delayed time after *wlen.* The falling edges of *wden* and *sae* need to arrive before the falling edge of *penb.*

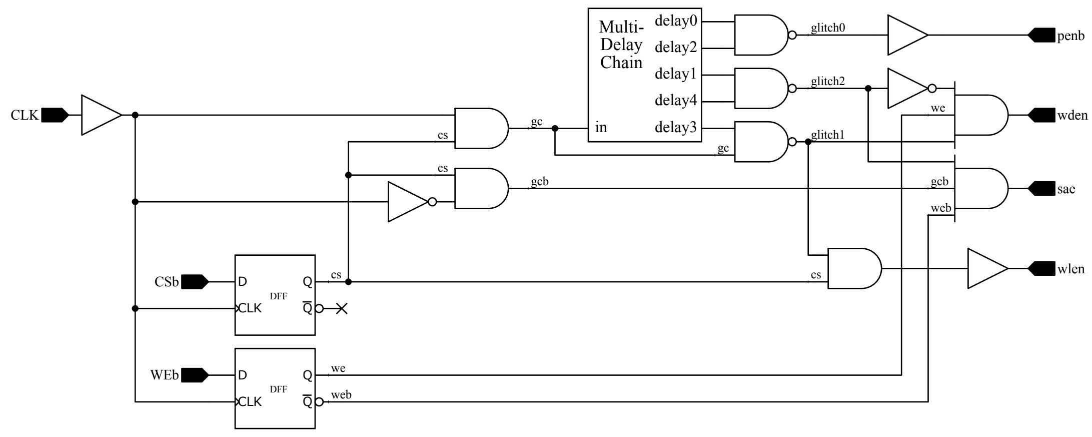

Figure 3.3: Schematic representation of delay-based control logic

Since the falling edges of all three signals besides *penb* must arrive before its falling edge, we can use the rising edge of the clock to trigger the falling edges of all signals other than *penb*. The exact amount of time these signals must arrive before/after one another is discussed in Chapter 5.

We design the circuit shown in Figure 3.3. This circuit prevents unwanted signal overlaps and reuses some glitch signals to slightly reduce control logic power and area.

3.2 Layout Design

When designing the layout, we try to make as few changes as possible to the layout compared to the RBL-based control logic. We replace the delay chain placement with the MDC. We place two of three glitch-generating AND gates on top of the existing column of control logic gates. The third glitch gate we place on the row responsible for generating *penb*. We route signals via the control logic's internal signal bus, to which we also add wires for the MDC output and glitch signals. Finally, we remove the input pin for RBL.

3.3 Implementation

3.3.1 Refactoring Control Logic Module

OpenRAM organizes the Python code to generate the netlist and layout at different levels of hierarchy into classes called "modules". Sometimes, these modules rely on inheritance from another module. For example, the modules for bitcells, replica bitcells, and dummy bitcells all inherit from the base bitcell module. The base module contains functions and variables that are shared by all inheriting bitcell modules, to avoid code duplication. We decided the introduction of a second control logic module warranted a similar approach. Thus, we determined which methods and variables could be shared between the RBL-based and delay-based control logic modules. We moved these to a new "control logic base" module, from which each of the two different control logic modules inherit. For example, the two types of control logic share functions called *create_dffs()*, *place_dffs()*, and *route_dffs()* which are responsible for netlist, placement, and routing of the control input signal DFFs respectively.

3.3.2 Writing the Delay Control Module

With many of the functions already implemented in the control logic base module, we set about implementing those specific to the delay-based control logic in a new module. Many of the functions implemented had counterparts in the pre-existing RBL control logic, so we refer here to modifications in relation to these existing functions. We modified the *add_pins()* function to remove the RBL pin. We modified the *add_modules()* function to use the newly created multi-delay chain module instead of the original delay chain module. We modified the *setup_signal_busses()* function to add busses for the multiple delay chain outputs and the glitch signals. We modified the *place_logic_rows()* function to place the logic for *glitch1* and *glitch2* above the other logic. Of note, *glitch0* did not need its own row since it was only used by the *penb* logic, so it was placed on that same row. We modified the *route_delay()* to route each of the delayed clock signal outputs (numbered *delay0*, *delay1*, etc.) to the bus. We modified the functions to create, place, and route control logic to implement the new design. Lastly, we created new functions to create, place, and route the new glitch signals.

Chapter 4

Bitcell Array Refactor

While implementing the delay-based control logic, we believed that OpenRAM's bitcell array modules had the ability to generate bitcell arrays without replica cells given the appropriate configuration. Upon further inspection, however, we discovered these arrays modules were not designed with this purpose in mind. In fact, because the assumption when OpenRAM was first designed was that there would always be an RBL and all the related circuitry, many modules required it and would need to be modified to relax this requirement. Thus, we decided to refactor these modules.

4.1 Capped Replica Bitcell Array

In determining the best course for refactoring the array code, we decided that the *replica_bitcell_array* module could be simplified. This module was responsible for taking an array of bitcells from the *bitcell_array* module and adding to it the following: (1) the column(s) of replica bitcells (*replica_column*), (2) the row(s) of dummy bitcells (*dummy_array*), (3) the column caps (*col_cap_array*), and (4) the row caps (*row_cap_array*). We believed adding the necessary code to create arrays without both (1) and (2) would make *replica_bitcell_array* too complicated. Instead, we decided to move the responsibility for (3) and (4), the end caps, to a new module called *capped_replica_bitcell_array*. This would mean that the added complexity for supporting the new array configuration would be split between two modules, making it easier to reason about the necessary changes.

First, we removed code from *replica_column* that added column caps to the top and bottom of the column. We moved the code from *replica_bitcell_array* responsible for adding all other cap cells to *capped_replica_bitcell_array*. We moved code for grounding unused wordlines in the dummy rows to *capped_replica_bitcell_array* because this routing had to be done through the row caps. We took advantage of existing OpenRAM functions to copy supply and input/output pins from *replica_bitcell_array* to *capped_replica_bitcell_array* at runtime. We changed several getter functions for layout attributes and names to return the correct values. Finally, we changed the handling and default value of class arguments pertaining to the configuration of RBLs. After completing all of this work, we confirmed that the refactored array code produced the same layout given the same configuration by performing an XOR comparison of the *.gds* layout files generated before and after the refactor, which were equivalent.

After refactoring was complete, it was safe to begin work on adding the desired feature: allowing arrays without RBL. The first step to allowing such configurations was to remove several checks which would cause the program to error out given a configuration without RBLs. Next, we made conditional the addition of RBL pins (both bitline and bitline bar) and RBL wordlines. We also had to change some conditional statments that expected the number of ports to be equivalent to the number of RBLs, which was no longer a valid assumption. We changed these statments to instead read the number of RBLs directly from arguments supplied to the module. These were essentially all of the major changes required after the work described in previous paragraphs to simplify the structure of the modules. The layout for an array with no RBL is shown in Figure 4.1. Other work was required, however, to integrate these changes with other modules and to ensure correctness, which is discussed in Sections 4.2 and 4.3.

4.2 Other Modules

OpenRAM has configuration options for creating bitcell arrays with split "local" wordlines, to reduce the wordline capacitance and thus the time to access a row. The modules facilitating these configurations are *local_bitcell_array* and *global_bitcell_array*. *local_bitcell_array* was changed to use the new *capped_replica_bitcell_array* module for its "local" arrays. Furthermore, it needed some minor changes to code looking for RBL wordline names and adjustment to module height and width calculations. Finally, the handling of default values for the arguments passed to this module determining RBL presence and numbers was changed to match the array modules lower in the hierarchy. This same change was also made in *global_bitcell_array*. This module also was changed to conditionally copy pins and routing relevant to RBL. The *bank* module was given

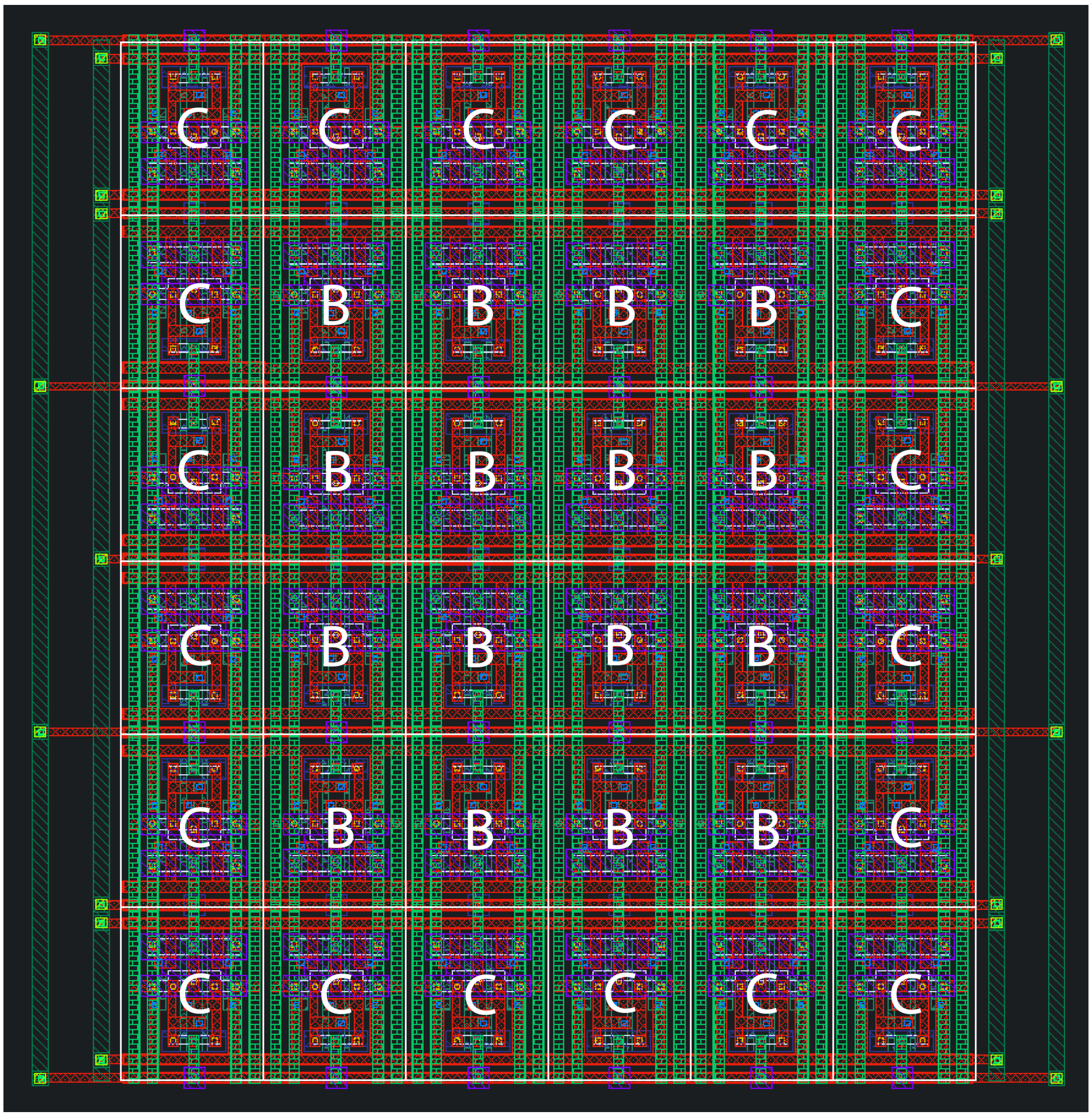

Figure 4.1: Layout of a bitcell array with no RBL. B are bitcells, C are row/column cap cells.

a new boolean variable *has_rbl* which is used to skip routing and pins for RBL as appropriate. This variable was passed as an argument to the *port_address* and *port_data* modules which use it in the same way. Specifically, *port_address* needs to route the *wlen* signal differently in the case that there is no RBL. And *port_data* needs to omit the precharge drivers for RBLs when they are absent.

4.3 Unit Tests

OpenRAM has a number of unit tests for its many components, especially the modules. Each module test verifies that OpenRAM is able to generate that module with a given configuration, in all supported technologies. For the newly created *capped_replica_bitcell_array* module, we wrote tests for many configurations, including tests for the cases with no replica bitcells. Furthermore, we standardized the naming for this series of tests to make it more apparent from the file name what configurations are tested. Finally, we updated tests for modules whose arguments were adjusted (by the addition of the *has_rbl* argument) to match the new variable. In addition, new tests were added to confirm that both values for this boolean variable worked as intended in these modules. The unit tests added or modified for this work are listed in Appendix A. These unit tests were useful during the development of these new and refactored modules to be able to quickly verify new code worked as intended.

4.4 Sky130 Dual-Port Array LVS Failures

After we completed the refactoring work on the bitcell arrays, we could not understand why dual-port bitcell arrays made with the SkyWater 130nm PDK [1] were failing LVS. Only array configurations which included dummy cells were able to pass LVS. Arrays using the ”norbl” configuration, as we called it, were failing LVS. We investigated whether the issue was with the newly added code for the norbl configuration, but comparison of the *.gds* showed that the arrays of regular bitcells (originating from the *bitcell_array* module) were the same in all configurations. Following this investigation, we discovered that the LVS reported a mismatch of the bitcell extracted from the *.gds* by Magic [7]. We performed several rounds of trying different versions of Magic (the tool used for extraction) and Netgen [8] (the tool used for LVS) to eliminate them as a sources of error.

It was during these trials that we discovered the Sky130 dual port bitcell had been failing LVS for as long as we had historical LVS reports to consult (going back over two years). Many of these reports came from chips that had been taped-out successfully, so we did not have reason to believe that the problem was with the tools, as this would have likely led to other problems in the silicon for previous OpenRAM chips. Failing to determine the root cause of the LVS failure, we reached out to Tim Edwards, the author of both Magic and Netgen. After looking at the sample layouts we sent him, he determined that the LVS models provided with the Sky130 PDK for the bitcell, replica bitcell, and dummy bitcell all contained the same error. Each has two dangling

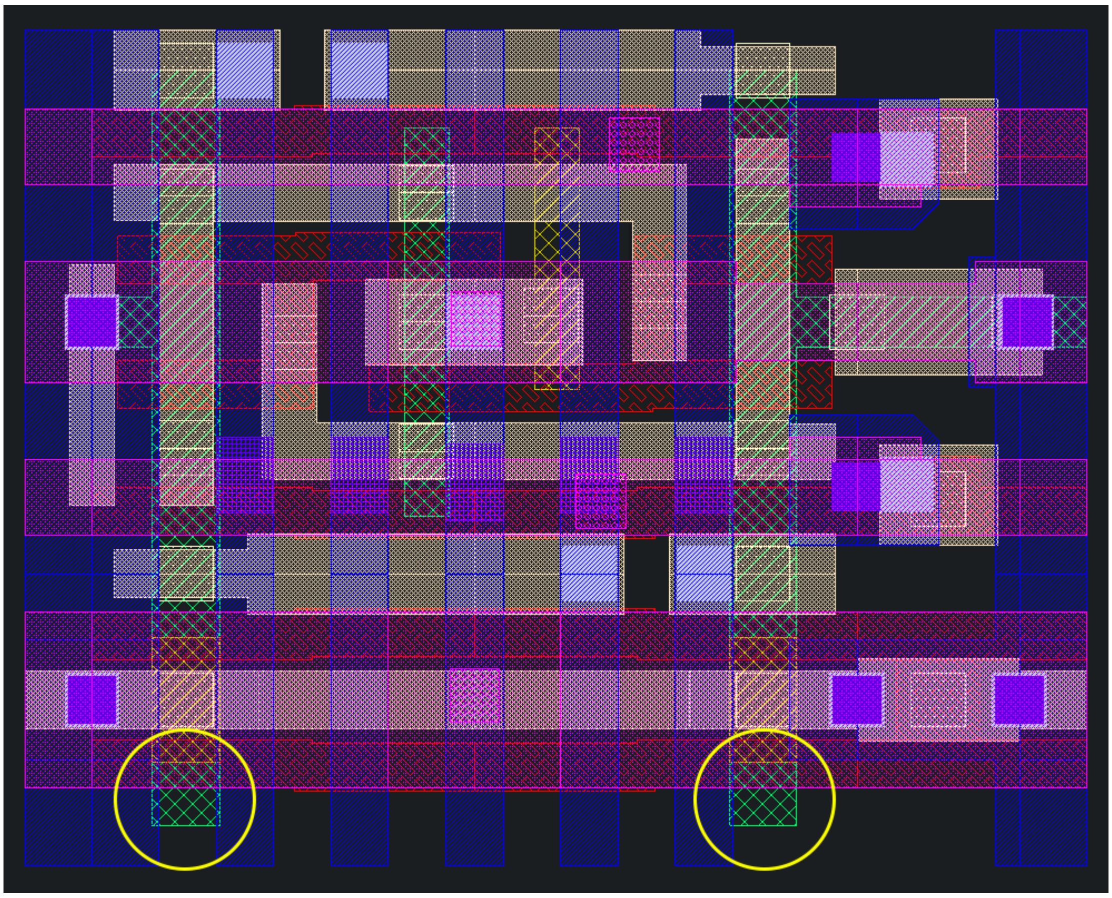

Figure 4.2: Layout of a sky130 dual-port bitcell showing the two disconnected devices.

pieces of diffusion at the bottom of the cell which partially overlaps polysilicon. The devices formed are disconnected from metal layers, as shown in Figure 4.2. However, the netlist provided by the foundary indicates that these devices connect to the bitlines as shown in Figure 4.3. When these cells are tiled together, these disconnected devices do eventually connect to the bitlines. But at the top and/or bottom of the array, they stay disconnected and cause a matching failure during LVS. Unfortunately, designing a valid solution to this problem was out of scope for this thesis.

```
...

* tap under poly

X10 GND GND BL1 GND sky130_fd_pr__special_nfet_latch w=0.21u l=0.08u m=1

X11 GND GND BR1 GND sky130_fd_pr__special_nfet_latch w=0.21u l=0.08u m=1

...
```

Figure 4.3: Excerpt from SPICE netlist for dual-port bitcell from sky130A PDK. These devices should not show connections to BL1 and BR1.

Chapter 5

Tapeout

We decided to tape out two dual-port (1r1rw) 1KB SRAM macros for comparison. The first uses the RBL-based control logic and the second the delay-based control logic. We submitted this design as part of the Efabless CI 2309 ChipIgnite Shuttle program, using the open-source SkyWater 130nm PDK [1].

Because of the unresolved LVS failures in the bitcell array without replica cells described in Section 4.4, we made temporary modifications for the tapeout to always add dummy bitcells to the top and bottom of the bitcell array. We grounded the wordlines of these dummy rows when replica columns were absent. This array configuration was not meant to be supported by OpenRAM in general, so the code for this was not commited to the main OpenRAM repository.

Since we did not implement a mechanism for OpenRAM to automatically select the sizing and output stages of the multi-delay chain, we had to estimate appropriate values for these parameters by hand. We used simulations of an identically sized SRAM with RBL-based control logic as a reference for the timing of control signals and their propogation delays.

Based on this reference, we selected conservative durations for each control signal and set the output pins for the multi-delay chain accordingly. Simulation waveforms for a read operation for the reference control signals and the delay-based variant are shown in Figures 5.1 and 5.2 respectively. Waveforms for a write operation for the delay-based variant are also shown in Figure 5.3. Based on the internal delay of the delay chain we estimated the delay for a single stage of the multi-delay chain would be 0.16ns ignoring load on the chain's output pins. Using this, we set the durations for control signals based on the reference as 8 stages for *penb*, 2 stages from falling edge of *wlen* to falling edge of *penb*, 7 stages from rising edge of *penb* to rising edge of *wlen*, and 13 stages from the beginning of *wlen* to that of *sae*.

After running the configuration through OpenRAM, we finally had our completed macro design, including netlist and physical views (LEF/GDS). A simplified view of this entire SRAM macro is shown in Figure 5.4.

We ran functional simulations on netlist and confirmed that it behaved as expected with a 7ns clock period. While characterizing the delay and power of the new macro, we discovered the binary search responsible for determining the minimum feasible period was unable to converge. This search began simulating at a clock period of 10ns by default, but doubled the period and started a new simulation after each failure. With the RBL-based control logic, this search was always able to converge. But with the delay-based control logic, sufficiently long clock periods caused the circuit to

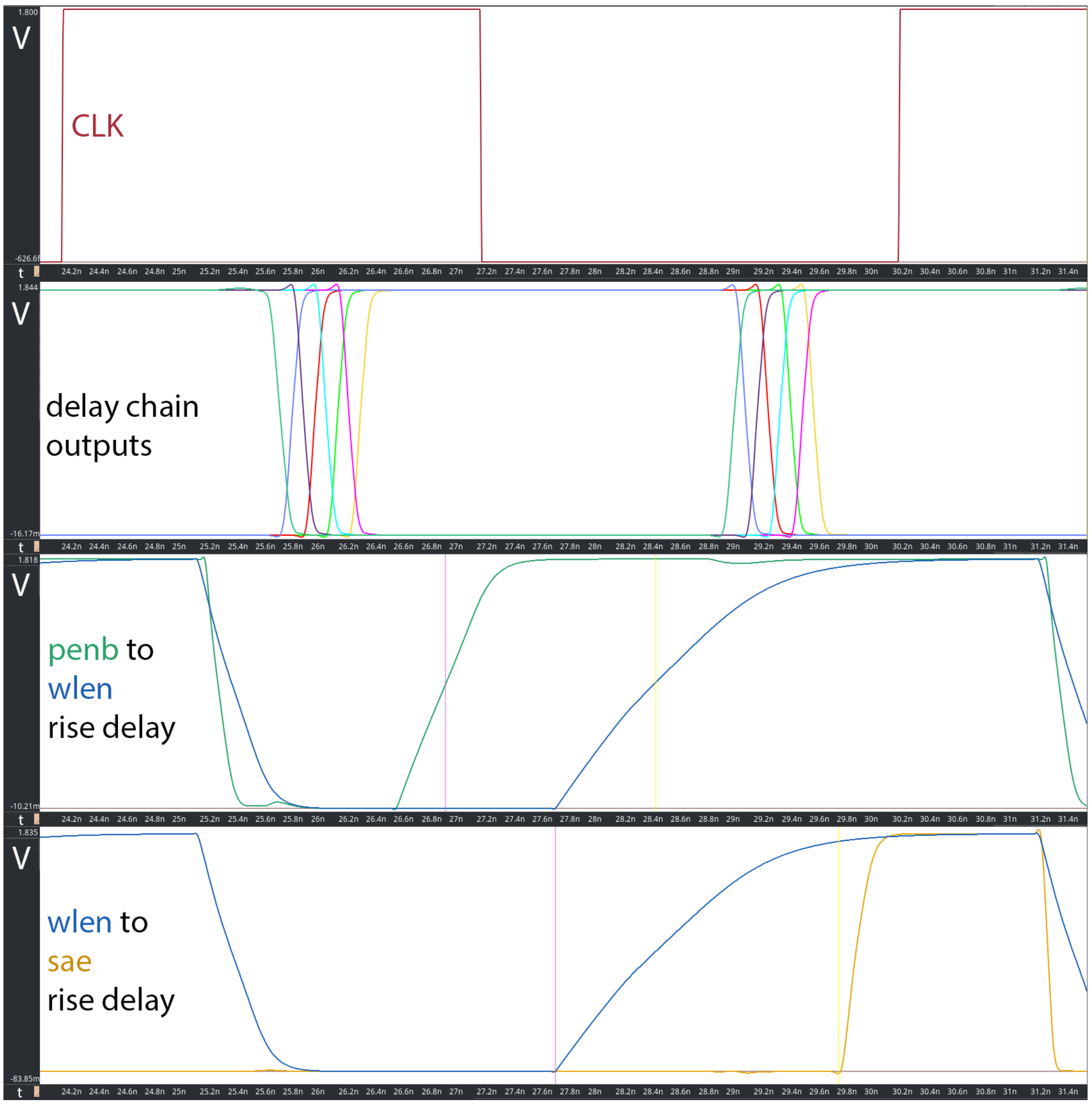

Figure 5.1: Waveforms for the RBL-based control logic read operation with 6.045ns period

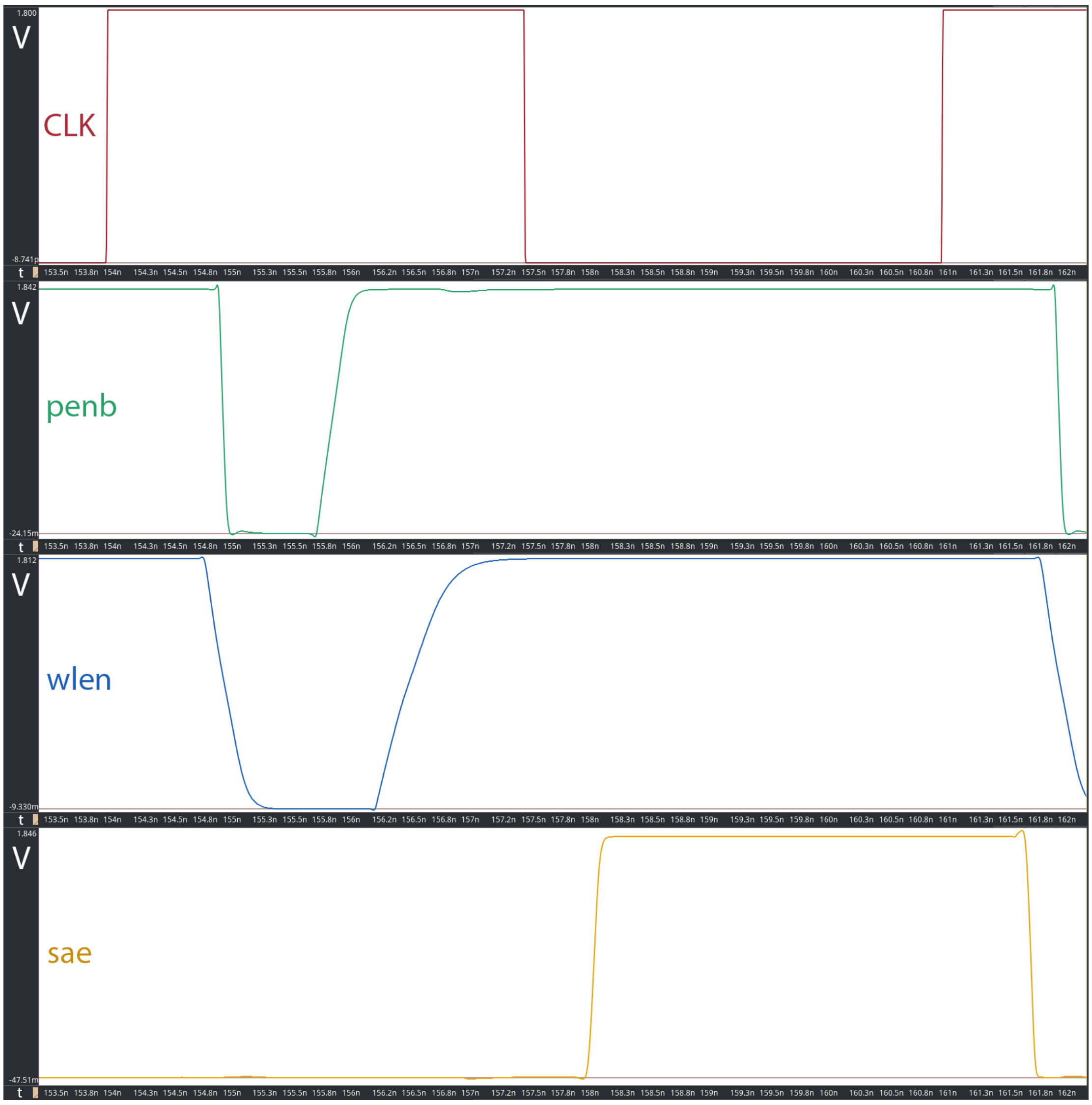

Figure 5.2: Waveforms for the delay-based control logic read operation with 7ns period

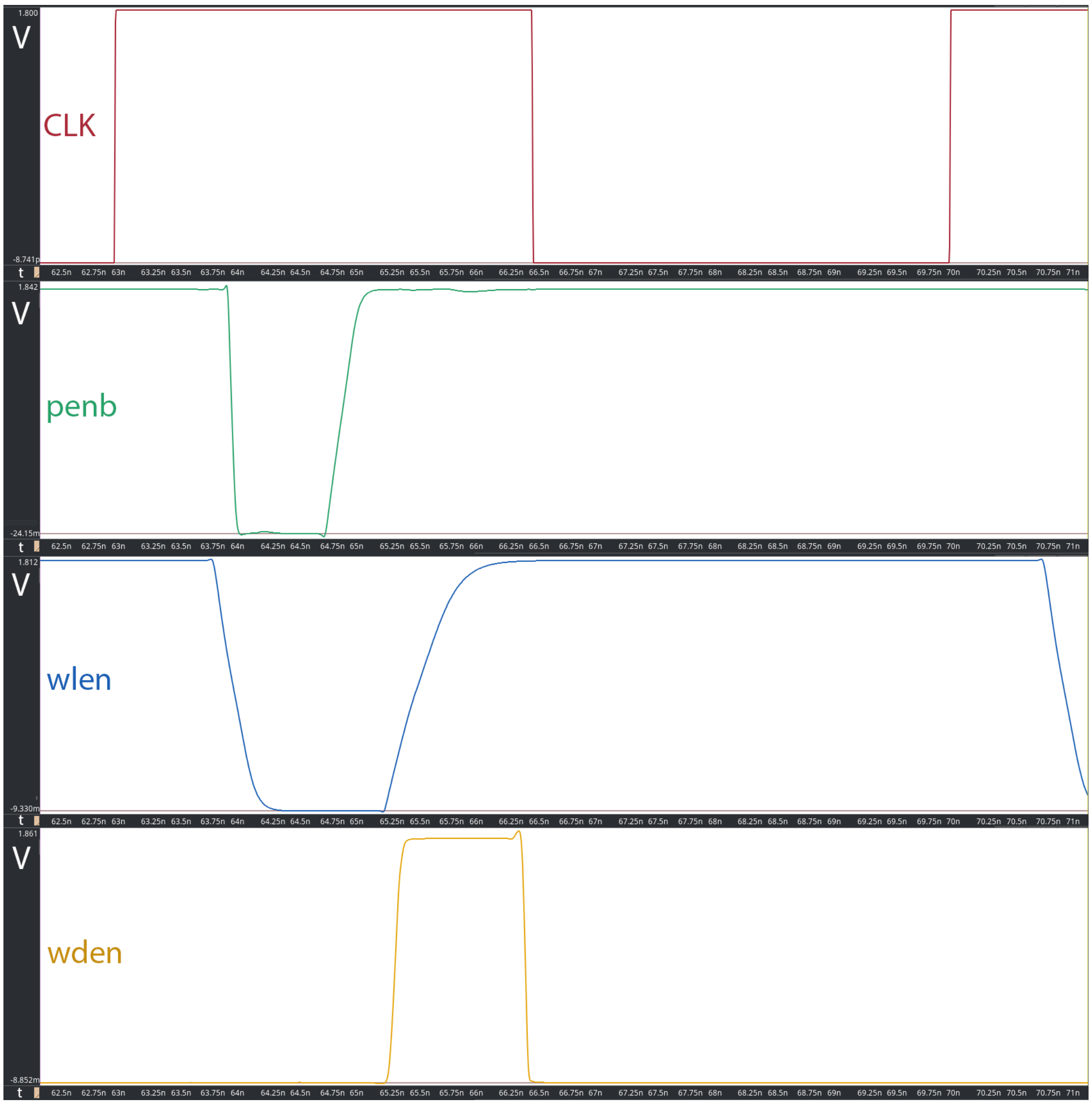

Figure 5.3: Waveforms for the delay-based control logic write operation with 7ns period

not pass functional simulations.

Recognizing this limitation, we restarted the binary search with a period of 7ns, which we knew from prior functional simulations would pass. This set an upper bound on the search space after succeeding, and the characterizer was able to converge on a final minimum period of 6.125ns. Solving this binary search problem or the underlying flaw in the new control logic that allows long clock periods to cause functional failure of the SRAM is left as future work, discussed in Chapter 7.

After successful layout, simulation, and characterization of the design, we delivered the design files to Efabless for silicon fabrication. DRC checks performed by Efabless found a single metal-2 minimum spacing error in the design which was uncaught by OpenRAM's DRC script. See Figure 5.5 for a depiction of this error. We fixed this error by hand and verified that the design still passed LVS, then sent the design back to Efabless.

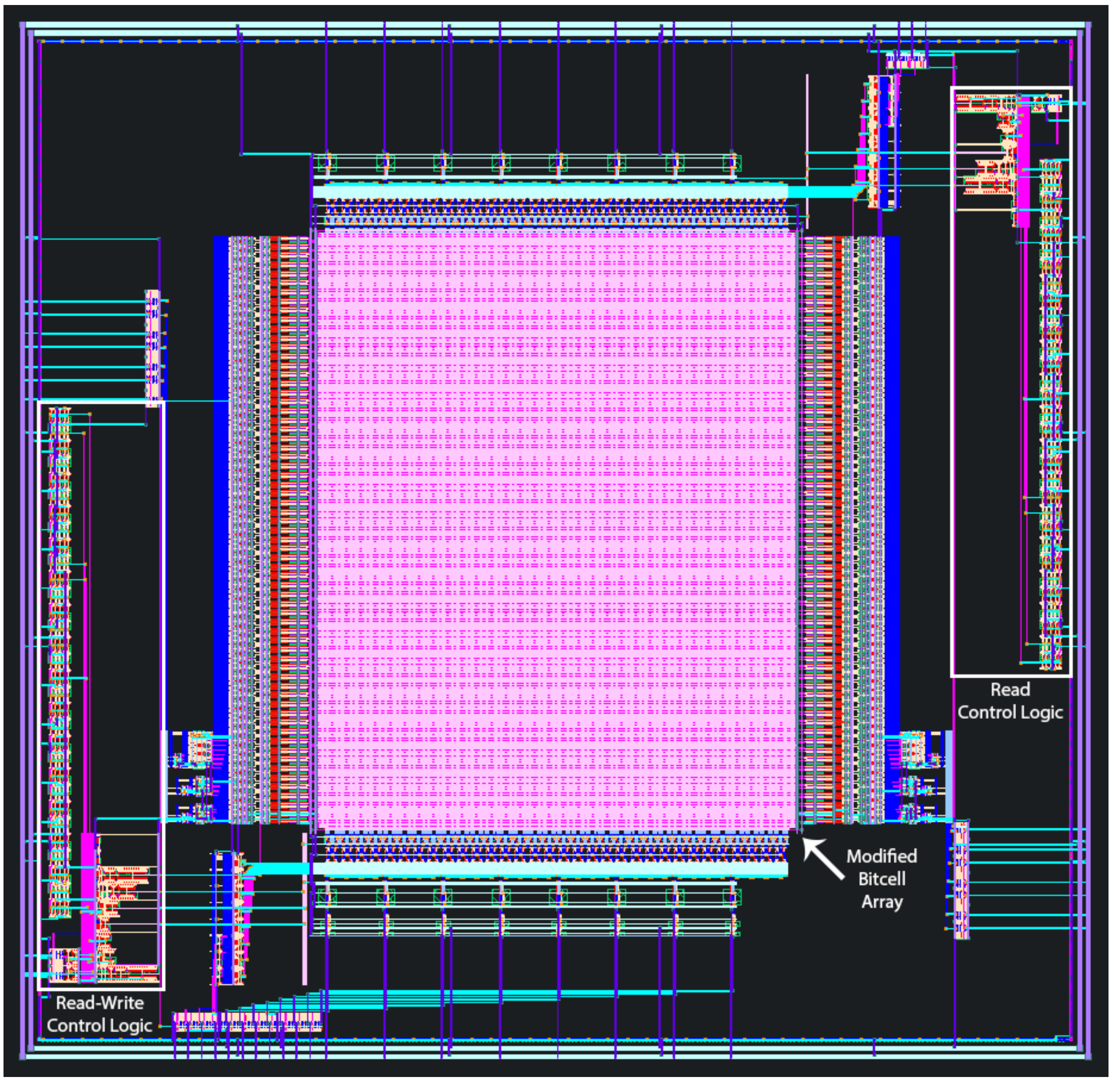

Figure 5.4: Layout of the 1KB dual-port macro with delay-based control logic.

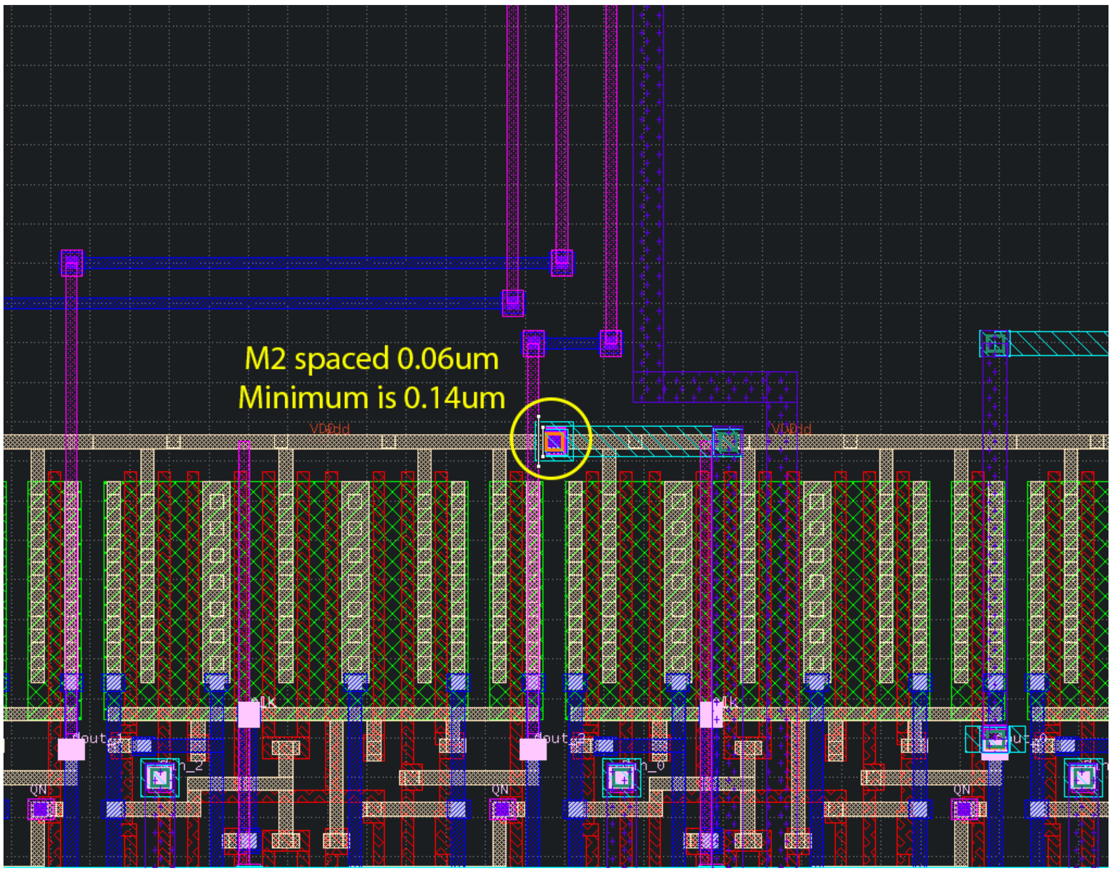

Figure 5.5: M2 minimum spacing drc error on VDD rail of data input DFF array

Chapter 6

Results

6.1 Layouts

Example layouts for the RBL-based control logic and delay-based control logic are shown in Figure 6.1 and Figure 6.2 respectively. These are representative of the general structure for the control logic layouts although they are not the layouts used in the taped-out macros.

6.2 PPA Comparison

Using the taped-out macros, we compared the power, performance, and area of the two control logic options. The difference in area is trivial, simply related to the removal of the replica columns (207840 and 206022 μm^2 for RBL-based and delay-based control logic respectively).

We ran each netlist through OpenRAM's SPICE simulation-based character-

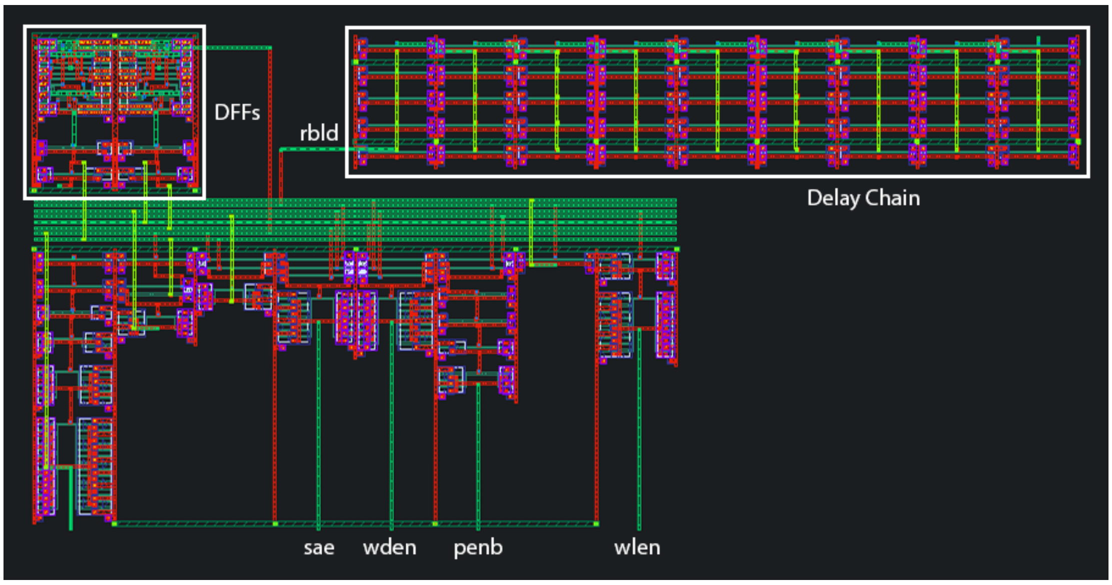

Figure 6.1: Layout of RBL-based read-write control logic

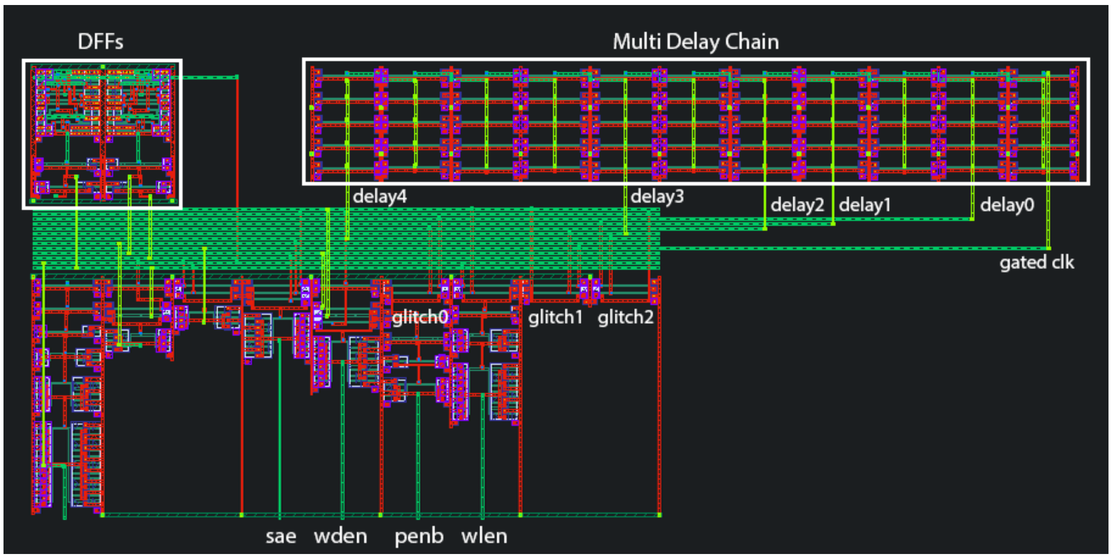

Figure 6.2: Layout of delay-based read-write control logic

ization (using Xyce [5] for simulation). We performed characterization in the SS, TT, and FF corners, using temperature of 25° and voltage of 1.8V. While characterizing the SS process corner, we found that the delay-based logic's *sae* signal arrived too early, so we modified the netlist to extend the multi-delay chain by an additional 4 stages (and used this same netlist when characterizing the other corners).

We used input slews of 0.005ns for all input signals. This is the minimum slew defined by OpenRAM's sky130 technology file, given as "rise_time" and "fall_time". The output capacitance used for measuring output delay was 6.89 fF, which is the DFF output capacitance from the technology file. Output delay was measured as the time from the rising edge of the clock to the rising edge of data out. The rising edge was considered to have been reached when a signal crossed 50% of VDD. We measured dynamic power for the duration of a clock cycle in which the SRAM performed a read, write, or no operation ($CSb = 1$).

6.3 PPA Analysis

Compiled results for the SS corner are shown in Table 6.1, for the TT corner in Table 6.2, and for the FF corner in Table 6.3. From these results, we see that the delay-based control logic has a smaller range of minimum period accross process corners. This may mean the circuit is optimized for the worst case, the SS corner. Hence, in the SS corner the delay-based logic is faster but in the TT and FF corners it is not. In each corner, power is greater for the faster circuit, partly because it is performing more operations.

In each corner, the delay-based logic has at least 30% lower latency than the RBL-based logic. This follows from the architecture, since the RBL-based logic uses the negative clock phase to activate the wordline drivers, while the delay-based logic can activate them earlier during the positive clock phase. It should be noted that this is not a shortcoming of RBL-based control logic, but rather in OpenRAM's implementation. This suggests the RBL-based logic could improve by decreasing precharge duration in favor of earlier wordline activation in slower process corners. As for the delay-based logic, the lacking performance in average and best case process corners is to be expected since it makes no attempt to track these variations in process. We expect voltage and temperature variations would produce similar results as process variations, although further simulations are required. Future work is also required to prove the efficacy of OpenRAM's characterization algorithm in comparing these different control logic circuits.

Table 6.1: Power, Performance, and Area comparison: SS corner

	RBL-based Control Logic	Delay-based Control Logic
Minimum Period (ns)	11.81	8.00
Read Power (mW)	0.88	1.29
Write Power (mW)	0.77	1.21
No-op Power (mW)	0.34	0.51
Data Setup (ns)	1.32	0.64
Data Hold (ns)	1.33	0.66
Data Out Delay (ns)	10.08	6.01

Table 6.2: Power, Performance, and Area comparison: TT corner

	RBL-based Control Logic	Delay-based Control Logic
Minimum Period (ns)	6.04	7.50
Read Power (mW)	1.75	1.39
Write Power (mW)	1.56	1.35
No-op Power (mW)	0.66	0.48
Data Setup (ns)	0.51	0.28
Data Hold (ns)	0.53	0.30
Data Out Delay (ns)	5.79	4.01

Table 6.3: Power, Performance, and Area comparison: FF corner

	RBL-based Control Logic	Delay-based Control Logic
Minimum Period (ns)	4.33	5.04
Read Power (mW)	2.58	2.14
Write Power (mW)	2.33	2.09
No-op Power (mW)	1.00	0.71
Data Setup (ns)	0.57	0.28
Data Hold (ns)	0.58	0.30
Data Out Delay (ns)	4.01	2.65

Chapter 7

Future Work

As described in Section 4.4, future work is required to get arrays of exclusively bitcells without replica or dummy cells to pass LVS. Dual-port SRAM configurations are an important part of OpenRAM and as such any new control logic should seek to support them in all supported technologies. As described in Chapter 5, future work is required to address the challenge of characterizing delay-based control logic. Since only one complete configuration has been characterized using this new logic, it remains to be seen if there are other issues that the characterizer and other components of OpenRAM may face when generating designs with the new control logic. Currently, only the simulation-based characterizer is able to characterize such designs. Future work is required on the analytical (Elmore Delay) characterizer to allow the critical path delay for the delay-based control logic to be estimated. For the time being, it displays an error due to the two valid timing paths from the clock to the sense-enable signal. This could possibly be fixed by manually pruning out one of these paths, but such debugging was considered out of scope for this thesis.

As discussed in Chapter 5, automated configuration of the multi-delay chain was not implemented. While some exploratory work was performed to this end, we decided that it would be better to wait for improvements to the Elmore Delay analytical models to leverage these for the estimation of the various delays in the relevant control signal paths. Specifically, models for the precharge drivers and bitlines would allow estimation of the minimum precharge duration. And models for the wordline drivers and wordline delays would allow estimation of the bitcell access time and therefore the sense-enable delay. Taken together, these delays form the critical path and are therefore necessary to estimate for the purpose of automatically determining the start and end times for control signals in the delay-based control logic. For now, we have made manual configuration of the multi-delay chain a requirement for the user configuration file when generating an SRAM with delay-based control logic.

The above future work items are essential to the full implementation of delay-based control logic in OpenRAM. However, as stated in Section 3.1, one of the considerations when choosing the particular type of delay-based control logic to design and implement was the possibility of adding support for asynchronous control logic. With the delay-based control logic as a starting point, we believe this goal can be achieved with minimal additions. Asynchronous control logic has the potential to reduce the power draw of the SRAM by eliminating the clock. It may also decrease the latency of memory access. Most importantly, providing an asyncronous interface to SRAM facilitates the development of larger asyncronous computing systems.

Asynchronous control logic exchanges the clock used by its synchronous counterpart for (1) an ”enable” signal input which begins the read or write operation and (2) a valid or complete signal output which indicates to external circuitry that the operation is complete. We believe that (1) can be replaced for the clock directly in this design, assuming it is also able to tell the DFFs to capture the input. Alternately, the DFFs could be removed and the user of the asynchronous SRAM would be required to hold input signals steady until the completion signal. For (2) a conservative estimate for the worst case duration of an operation could be used to add additional delay stages to the multi-delay chain and an additional glitch using these stages could generate the completion signal in the same way as the internal control signals. While this description certainly simplifies the complexity one could expect to encounter while implementing this feature in OpenRAM, doing so seems feasible nonetheless.

Chapter 8

Conclusion

In this thesis, we presented a new control logic design for OpenRAM-generated SRAMs with inverter chain delay-based signal timing. We designed and implemented the netlist and layout for this control logic, and integrated it into OpenRAM. We also refactored the bitcell array code to add configurations needed for use with this control logic. Finally, we taped-out a 1KB dual-port SRAM macro using delay-based control logic for fabrication on an Efabless chip shuttle alongside other designs.

Bibliography

[1] Skywater open source PDK. `https://github.com/google/skywater-pdk`.

[2] B.S. Amrutur and M.A. Horowitz. A replica technique for wordline and sense control in low-power sram's. *IEEE Journal of Solid-State Circuits*, 33(8):1208–1219, 1998.

[3] Jesse Cirimelli-Low, Muhammad Hadir Khan, Samuel Crow, Amogh Lonkar, Bugra Onal, Andrew D. Zonenberg, and Matthew R. Guthaus. Sram design with openram in skywater 130nm. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2023.

[4] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, Nov 2016.

[5] Eric R. Keiter, Thomas V. Russo, Richard L. Schiek, Heidi K. Thornquist, Ting Mei, Jason C. Verley, Peter E. Sholander, Karthik V. Aadithya, and Joshua D. Schickling. Xyce™ parallel electronic simulator reference guide (v.7.4). 10 2021.

[6] Hunter Nichols, Michael Grimes, Jennifer Sowash, Jesse Cirimelli Low, and M. R.

Guthaus. Automated Synthesis of Multi-Port Memories and Control. In *2019 IEEE/International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 1–6, Oct 2019.

[7] Magic Development Team. Magic VLSI Layout Tool. `http://opencircuitdesign.com/magic/`, 2023.

[8] Netgen Development Team. Netgen Netlist Comparison (LVS) and Format Manipulation Tool. `http://opencircuitdesign.com/netgen/`, 2023.

Appendix A

Unit Tests Added or Modified

A.1 Unit Tests Added

```
14_capped_replica_bitcell_array_bothrbl_1rw_1r_test
14_capped_replica_bitcell_array_dummies_1rw_1r_test
14_capped_replica_bitcell_array_dummies_1rw_test
14_capped_replica_bitcell_array_leftrbl_1rw_1r_test
14_capped_replica_bitcell_array_leftrbl_1rw_test
14_capped_replica_bitcell_array_norbl_1rw_1r_test
14_capped_replica_bitcell_array_norbl_1rw_test
14_capped_replica_bitcell_array_rightrbl_1rw_1r_test
14_replica_bitcell_array_dummies_1rw_1r_test
14_replica_bitcell_array_dummies_1rw_test
14_replica_bitcell_array_leftrbl_1rw_test
14_replica_bitcell_array_norbl_1rw_test
14_replica_bitcell_array_rightrbl_1rw_1r_test
14_replica_column_1rw_test
15_global_bitcell_array_norbl_1rw_1r_test
15_global_bitcell_array_norbl_1rw_test
15_global_bitcell_array_rbl_1rw_1r_test
15_global_bitcell_array_rbl_1rw_test
15_local_bitcell_array_bothrbl_1rw_1r_test
15_local_bitcell_array_dummies_1rw_1r_test
15_local_bitcell_array_dummies_1rw_test
15_local_bitcell_array_leftrbl_1rw_1r_test
15_local_bitcell_array_leftrbl_1rw_test
15_local_bitcell_array_norbl_1rw_1r_test
15_local_bitcell_array_norbl_1rw_test
15_local_bitcell_array_rightrbl_1rw_1r_test
16_control_logic_delay_multiport_test
16_control_logic_delay_r_test
16_control_logic_delay_rw_test
16_control_logic_delay_w_test
19_single_bank_nomux_norbl_1rw_1r_test
19_single_bank_nomux_norbl_test
20_sram_1bank_nomux_norbl_1rw_1r_test
20_sram_1bank_nomux_norbl_test
```

A.2 Unit Tests Modified

```
14_replica_bitcell_array_bothrbl_1rw_1r_test
14_replica_bitcell_array_leftrbl_1rw_1r_test
```

```
14_replica_bitcell_array_norbl_1rw_1r_test
14_replica_column_1rw_1r_test
16_control_logic_multiport_test
18_port_address_16rows_1rw_1r_test
18_port_address_16rows_test
18_port_address_256rows_1rw_1r_test
18_port_address_512rows_test
18_port_data_16mux_1rw_1r_test
18_port_data_16mux_test
18_port_data_2mux_1rw_1r_test
18_port_data_2mux_test
18_port_data_4mux_1rw_1r_test
18_port_data_4mux_test
18_port_data_8mux_1rw_1r_test
18_port_data_8mux_test
18_port_data_nomux_1rw_1r_test
18_port_data_nomux_test
18_port_data_spare_cols_test
18_port_data_wmask_1rw_1r_test
18_port_data_wmask_test
```