NAND FLASH COMPILER USING THE SKYWATER 130NM PROCESS

By

BRANDON T. ONG

Bachelor of Science in Computer Engineering
Bachelor of Science in Electrical Engineering
Oklahoma State University
Stillwater, Oklahoma
2020

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2022

NAND FLASH COMPILER USING THE SKYWATER 130NM PROCESS

Thesis Approved:

Dr. James E. Stine, Jr.

Thesis Adviser

Dr. Bingzhe Li

Dr. Weihua Sheng

Name: BRANDON T. ONG

Date of Degree: MAY, 2022

Title of Study: NAND FLASH COMPILER USING THE SKYWATER 130NM PROCESS

Major Field: ELECTRICAL ENGINEERING

Abstract: NAND flash memory is commonly used for data storage, with applications in SSDs and flash drives. NAND flash research in academia can be limited by insufficient access to memory of varied sizes. This thesis discusses the design of a NAND flash memory compiler. This compiler provides researchers access to a customizable flash array. The array is built using the SkyWater Technology 130nm Process Design Kit (PDK) and SONOS flash technology. A detailed review of the implementation is included covering both the physical design of the flash array as well as the design of the compiler. The result of the compiler is able to be fabricated, as shown by an approved submission to Efabless' Multi-Project Wafer (MPW) shuttle program. The results consist of simulations that prove the functionality of the flash array.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION **1**

1.1 Organization 4

II. BACKGROUND **5**

2.1 Floating Gate Transistor 5

2.2 SONOS 8

2.3 NAND vs NOR flash 11

2.4 Array Structure 12

III. IMPLEMENTATION **15**

3.1 Flash Cell 15

3.2 Flash Array 16

3.3 Sense Amplifier 17

3.4 Script 21

3.5 Fabrication 25

IV. RESULTS **28**

4.1 Results 31

V. CONCLUSION AND FUTURE RESEARCH **35**

5.1 Future Research 35

REFERENCES **37**

Chapter Page

APPENDICIES **40**

APPENDIX A: FLASH SENSE AMPLIFER 2 SPICE FILE 40

APPENDIX B: NAND FLASH COMPILER SCRIPT 42

APPENDIX C: MODIFIED TRI-STATE BUFFER SPICE FILE 52

APPENDIX D: MODIFIED SONOS MODEL 53

LIST OF TABLES

Table Page

2.1 SONOS Voltage Table (Referenced from [1]) 10

3.1 Layers needed for flash cell. Referenced from [1] 16

4.1 SONOS Spice Models (Referenced from [1]) 30

4.2 Layout Sizes 34

LIST OF FIGURES

Figure Page

2.1 Cross Section Comparison 6
2.2 Fowler-Nordheim Programming 7
2.3 Fowler-Nordheim Erasing 8
2.4 Floating Gate Reading 9
2.5 SONOS Cross Section 10
2.6 Endurance of SKY130 SONOS (Referenced from [1]) 11
2.7 NOR and NAND Architecture 12
2.8 Structure of a NAND Block 13
2.9 Structure of a NAND Array 14

3.1 SONOS 2T. Referenced from [1] 15
3.2 Flash Cell 2x2 17
3.3 Flash Block 18
3.4 Flash Array 18
3.5 Sense Amplifier Schematic 20
3.6 Sense Amplifier Operation 21
3.7 Sense Amplifier Layout 22
3.8 Sense Amplifier used as reference. Adapted from Figure 6.78a in [2] . 23
3.9 Flash Array Created by Script 24
3.10 Caravel Harness [3] 26
3.11 Caravel Harness Chip 27

4.1 Modified Tri-State Buffer 29

Figure Page

4.2 FGMOS Model 30

4.3 Erasing and Programming Simulation 31

4.4 Sense Amplifier Simulation 32

4.5 Simulation of Flash Array 33

CHAPTER I

INTRODUCTION

Flash was invented in 1984 by Fujio Masuoka and Toshiba [4]. Since 1984, flash has taken the memory and storage world by storm. Flash is a type of non-volatile memory, which means it is capable or retaining the stored information when its power supply is removed. The non-volatile nature is enabled by a floating gate transistor, a transistor that is capable of storing electrons in an extra poly-silicon layer. NAND flash is a type of flash memory that is typically used for data storage. NAND flash greatly reduces the area and, therefore, the cost compared to NOR. Its compact design and sequential access makes it an ideal technology for storage. NAND flash is used in SSDs, flash drives, mobile devices, and anything else that requires compact non-volatile memory.

From 2000 to 2016, the feature size of NAND flash has reduced by a factor of $\sqrt{2}$ every other year [5]. This roughly follows Moore's Law, which states that the number of transistors on an IC will double every two years [6]. However, this trend cannot continue as fabrication costs and complexity continue to rise. The increased cost for lithography eventually outweighs the benefit of extra capacity [7]. The performance also begins to degrade when the feature size continues to decrease.

MLC (Multi Level Cell) flash [8] is a type of NAND flash that increases storage capacity without adding to the number of cells. The voltage threshold of the cell is controlled precisely to allow the storage of multiple bits per cell. MLC does not change the structure of the floating gate transistor. 3D (Three-Dimensional) NAND flash [9] is another popular technique used to improve storage capacity. In 3D NAND, the cells

are stacked vertically as well as connected horizontally. Most 3D NAND architectures are gate-stacked, meaning the gate layers are stacked and the current flows vertically. The area cost increase is minimal, but its storage ability is enormously improved. However, the fabrication of 3D NAND typically requires complicated process steps and dedicated facilities. Consequently, the physical implementation of NAND flash has been a lucrative research area. The level of area reduction and storage capacity is impressive. Despite this, the improvement rates do not always continue to improve [10].

Fortunately, there are other avenues for improvement. One such way is research into reliability of NAND flash. There may be faulty bits within a flash array or an MLC flash cell may read an incorrect value. Error correction is used to mitigate the possibility of poor storage reliability. Concatenating TCM (Trellis Coded Modulation) with BCH code [11] has been used to provide better performance of error correction codes. Temperature awareness and exploiting self-recovery [12] has also been used to improve the reliability. Self-recovery is the de-trapping of a an accidentally trapped charge.

Another element of concern for NAND flash is its endurance. The flash cell is slightly damaged each time it is programmed or erased. Given enough operation cycles, the cell becomes faulty, possibly ruining the entire flash array. Wear leveling is a solution that improves the overall endurance NAND flash. The idea is to spread the use of the flash cells among the entire flash device. Balancing page endurance [13] helps with endurance by skipping or relieving the weakest pages during programming. This technique can extend the lifetime up to 60%. Therefore, program error rate [14] is used to optimize wear leveling algorithms. The error rate is used to measure the wear of a block. Block data swapping also improves wear leveling efficiency.

These areas of research for NAND flash continue to grow. However, it can be difficult for academic researchers to produce innovations that rival with those in industry.

A large reason for this discrepancy is access to resources. In order for a researcher to develop a new wear-leveling algorithm, for example, they must have access to a NAND flash array. This provides limitations to the research. Process Design Kits (PDKs) sometimes provide NAND flash, but it will be a black-box device. That is, due to Intellectual Property (IP) issues, the underlying layout is not provided to users and instantiated at the time of fabrication. This leads to poor results for researchers to customize the flash device. In industry, companies often have memory compilers that allow them to test many different configurations. Currently, an academic researcher would need to slowly design their own memory arrays physically. The design process would be tedious and take valuable time away from their research, which is counterproductive to their original goal. This gap between industry and academia makes it difficult for academic researchers to stay up to date.

The goal of this thesis is to solve this issue by providing a NAND flash memory compiler. OpenRAM [15] is an open-source memory compiler that provides a framework to quickly design a SRAM memory array. OpenRAM generates, characterizes, and verifies the SRAM memory array. The end result is a fabricable memory design. The NAND flash compiler in this thesis is similar to OpenRAM, except with NAND flash. The compiler is designed using the SkyWater Technology [16] 130nm PDK. SkyWater Technology 130nm is an open-source PDK that was originally developed by Cypress Semiconductor. SkyWater Technology and Google collaborated to make it accessible to the public. This work discusses a flash cell that is designed using SONOS flash technology, as opposed to the conventional floating gate transistor. Theoretically, a researcher can use the flash compiler to design a flash array of any given size. The compiler also outputs a flash array that is DRC-free and fabricable. Therefore, the NAND flash compiler enables academic researchers to focus on innovations rather than the tedious design of a NAND flash memory.

1.1 Organization

The organization of this thesis is as follows: Chapter II discusses the details of flash memory, including its structure and operation. The basics of floating gate, SONOS, NAND vs NOR flash, and the structure is covered. Chapter III discusses this thesis' implementation of a NAND flash memory compiler, from the physical design of the flash cell, array, sense amplifier, and script to the fabrication of the array. Chapter IV discusses the simulation and results of the array and peripherals. Finally, Chapter V presents the conclusion to the work and any future research for this topic.

CHAPTER II

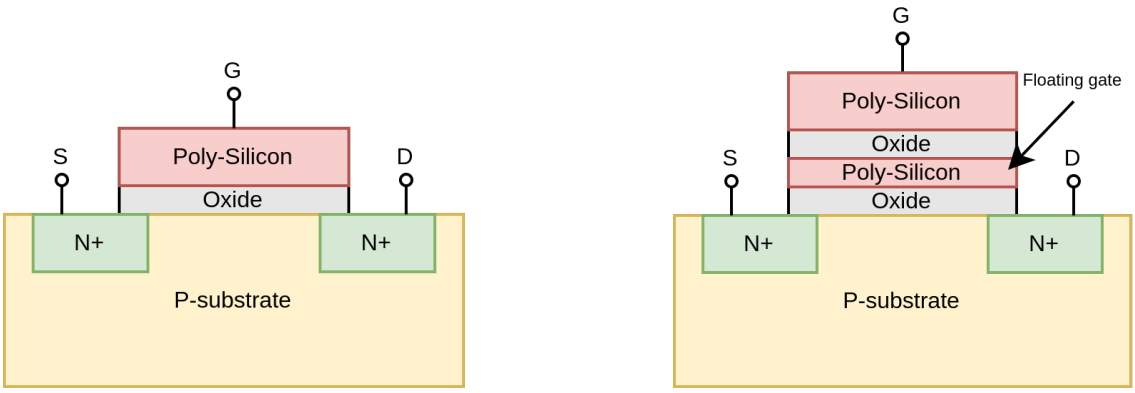
BACKGROUND

Flash memory is a non-volatile memory that is commonly used due to its low cost and compact nature. It is beneficial to use instead of DRAM since it does not require a supply voltage to store information. NAND flash is used in Solid State Drives (SSD), flash drives, and phones. Instead of SRAM, which typically uses six transistors, there is only one floating gate per bit cell. Therefore, flash has the ability to be almost six times larger in capacity.

2.1 Floating Gate Transistor

The floating gate transistor is the basic building block of flash memory. A floating gate transistor has a similar structure to a n-type MOSFET. The difference is that a floating gate transistor has an extra poly-silicon gate that is separated from the device with a dielectric oxide, hence the name "floating." The additional gate is shown in Figure 2.1b. The floating gate is able to store electrons, which in turn influence the gate voltage threshold of the transistor. This allows the floating gate to control whether current flows from source to drain during a read cycle.

In order to store electrons in the floating gate, the transistors must be programmed. There are two main methods for programming: Fowler-Nordheim (FN) tunneling and Channel Hot Electron (CHE) injection [2]. Channel Hot Electron injection is typically used by NOR Flash. In this method, the drain is set to a much higher voltage than the source and the gate is turned on. This results in a high



(a) NMOS

(b) Floating Gate Transistor

Figure 2.1: Cross Section Comparison

current flow that allows the electrons to gain enough energy to get trapped into the floating gate. CHE injection is also able to program very quickly.

Fowler-Nordheim tunneling is typically used by NAND Flash. A strong electric field is created during Fowler-Nordheim tunneling. If the oxide between the substrate and poly-silicon floating gate is thin enough, electrons will flow through the "tunnel" created. The following is a simplified Fowler-Nordheim current density equation [17].

$$J = [\tau_F^{-2} a \phi^{-1} F^2] \exp[-v_F b \phi^{3/2}/F] \tag{2.1}$$

F is the field at the surface and is also called the 'barrier field'. a and b are constants in the equation. τ_F and v_F are correction factors that help with the simplification of the Fowler-Nordheim equation. The local work function is represented by ϕ. In order to examine the relationship between the electric field and the current density, Equation 2.1 can be even further simplified.

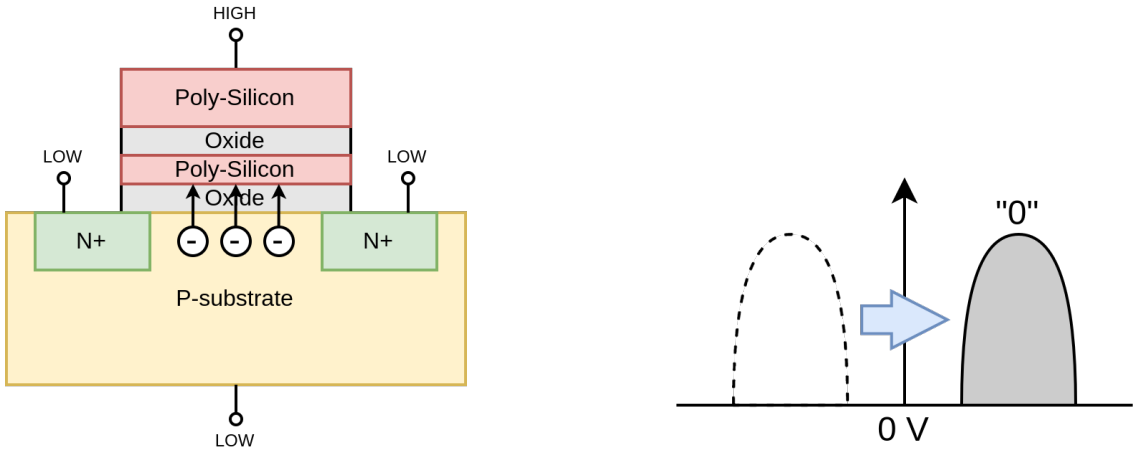
$$J = \alpha F^2 \cdot \exp[-\beta/F] \tag{2.2}$$

In Equation 2.2, $\alpha = \tau_F^{-2} a \phi^{-1}$ and $\beta = v_F b \phi^{3/2}$. As the electric field gets stronger, the current density increases at an exponential rate. Given this relationship, the choice of voltage levels becomes important. The field must be strong enough for the electrons

to move through the tunnel at a high enough rate. The stronger the field the faster the program and erase time. However, Fowler-Norhdeim tunneling causes damage to the cell every operation. Therefore, the field strength should be controlled to limit the damage.

In FN tunneling, the control gate is set to a high voltage while the source and drain are set low. The strong electric field created causes the electrons to move through the oxide layer into the floating gate. Unfortunately, the programming is slower than in Channel Hot Electron injection. FN tunneling uses little current though, which enables programming multiple bits at a time. That is, programming a floating gate causes the voltage threshold to increase. Figure 2.2 shows the Fowler-Nordheim program operation and the effect on the voltage threshold.

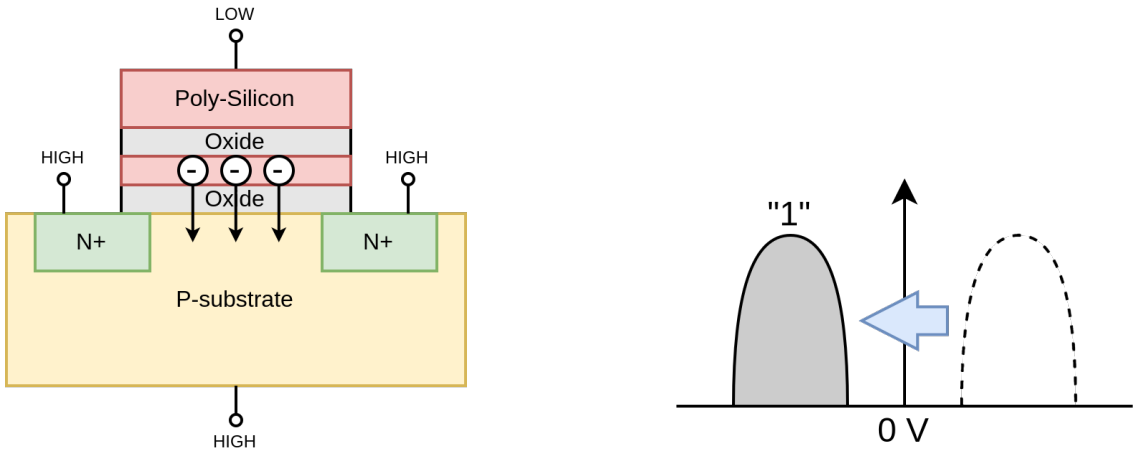
Removing the electrons from the floating gate requires erasing. Erasing of a flash cell is done using Fowler-Nordheim tunneling. Erasing using FN tunneling is the exact opposite of programming using FN tunneling. The control gate is set to a low voltage and the source and drain are set high. The negative electric field moves the electrons from the floating gate through the oxide into the source. The voltage threshold of the floating gate decreases with the removal of the electrons in the floating gate. Erasing



(a) Program

(b) Floating Gate Voltage Threshold

Figure 2.2: Fowler-Nordheim Programming



(a) Erase

(b) Floating Gate Voltage Threshold

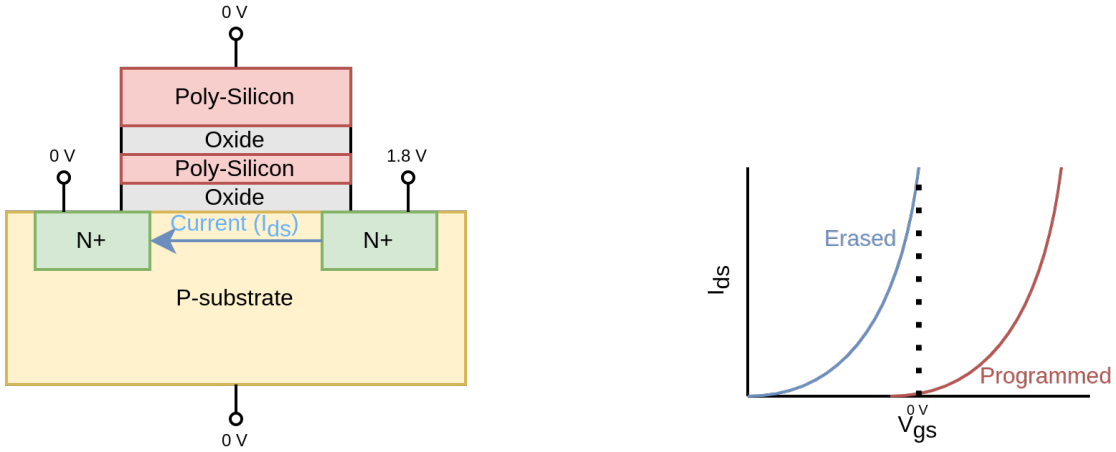
Figure 2.3: Fowler-Nordheim Erasing

with FN tunneling also requires little current, so the operation can be done to many bits at a time. Figure 2.3 shows the Fowler-Nordheim erase operation and the effect on the voltage threshold.

The state of the floating gate is determined during the reading stage. A sense amplifier determines if there is current flowing between the source and the drain. 0 V is applied to the control gate. The source is pre-charged and the drain is set to 0 V. If there is current flowing between the drain and source, then the voltage threshold of the floating gate must be negative. Consequently, the floating gate is erased and the output becomes '1'. If there is no current flow, the voltage threshold is positive and the gate is programmed. A programmed cell has an output of '0'.

2.2 SONOS

SONOS (Silicon-Oxide-Nitride-Oxide-Silicon) [18] is a flash memory technology that is an alternative to conventional floating gate technology. SONOS is part of the charge-trap memory family [19]. Charge-trapping memory is a flash memory that uses a nitride layer instead of the extra poly-silicon layer of the floating gate. The electrons are injected and ejected through a tunnel and stored in the nitride layer.



(a) Read

(b) Cell Current

Figure 2.4: Floating Gate Reading

The nitride layer is called the "Charge-Trapping Layer". A benefit of using charge-trapping technology is that it is easier to manufacture and scale than floating gate technology. Floating gate memory has issues with reducing the tunnel oxide thickness and reducing operating voltages. Floating gate transistors also require more process steps to isolate the floating gate. Additionally, in floating gate, charge can flow side to side in the poly-silicon layer. The nitride layer of charge-trap doesn't allow the incidental flow of charge, so the reliability of a cell is improved in charge-trap memory.

The SkyWater Technology 130nm process uses SONOS that was originally designed by Cypress Semiconductors. This technology was selected because of the easier manufacturing provided by using charge-trap instead of conventional floating gate. The structure of SONOS is the same as a NMOS transisotr with a ONO (oxide-nitride-oxide) dielectric stack, as can be seen in Figure 2.5. This SONOS cell is programmed and erased using Fowler-Nordheim tunneling, as described in the floating gate section.

Table 2.1 shows the voltages that must be applied to each input of the SONOS cell. The voltages required for programming and erasing are relatively small when compared to conventional floating gate operation, which can have inputs as high as 20 V. During the read phase, the bitline is pre-charged to 1.8 V. The control gate is

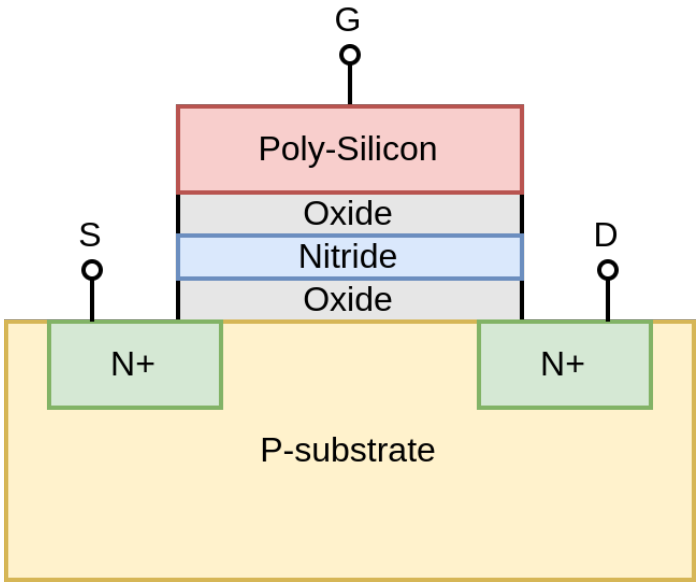


Figure 2.5: SONOS Cross Section

Operation	Drain	Gate	Source	Body
Erase	+6.7 V	−3.8 V	+6.7 V	+6.7 V
Program	−3.8 V	+6.7 V	−3.8 V	−3.8 V
Read	pre	0 V	0 V	0 V

Table 2.1: SONOS Voltage Table (Referenced from [1])

set to ground. If the cell is erased, the threshold is negative, therefore, current flows between the source and drain. If the cell was programmed, the threshold is positive and there is no current flow.

Figure 2.6 shows the endurance of SkyWater Technology's SONOS cell. Every time the cell goes through the Fowler-Nordheim tunneling operation, either programming or erasing, the oxide in between the nitride layer and the substrate is damaged. So after a specific number of operations, the integrity of the cell is compromised. However, there are 100K cycles guaranteed for each cell during its operation. The voltage threshold remains constant throughout the testing. The cell in the programmed state has a positive voltage threshold of about 1.6 V and the erased has a negative voltage

threshold of about $-1.2\,\mathrm{V}$. To preserve the cells, algorithms have been developed to ensure individual cells are not written and erased too often. Instead, it tries to spread the storage across the entire array.

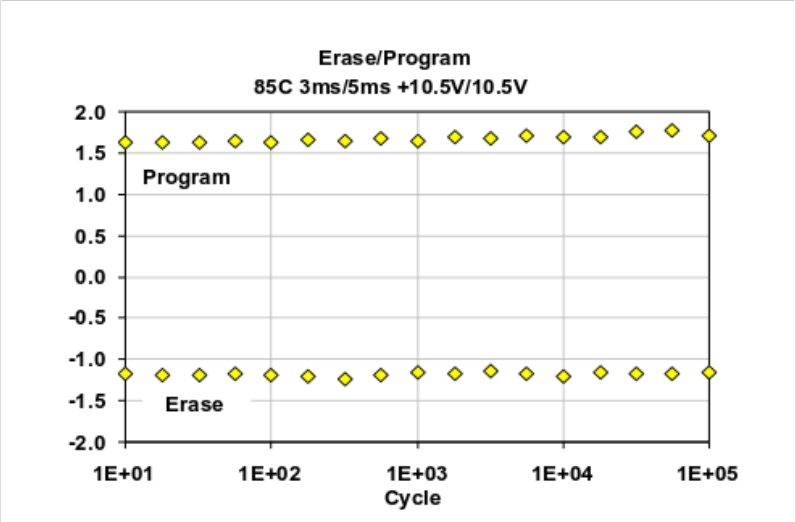


Figure 2.6: Endurance of SKY130 SONOS (Referenced from [1])

2.3 NAND vs NOR flash

NAND and NOR are the most basic types of flash memory. Both use the floating gate transistor. NOR is designed such that the floating gate transistors are connected in parallel to a bitline on their drain and a source line on their source. This design enables fast random-access reads, since each bit is accessible.It also guarantees no faulty bits, which is especially useful if executing code. For these reasons, NOR tends to be used for code and data execution. A drawback of the NOR design is slower program and erase speeds.

NAND flash is designed to have the floating gate transistors in series, which reduces the area per cell. This is the main reason that NAND flash has become so popular. Less area per cell results in cheaper production. NAND flash can efficiently store massive amounts of data. An obvious application can be seen in solid state

drives. Since NAND uses FN tunneling, it can program many cells at a time, so NAND has much faster program and erase speeds than NOR flash. Additionally, NAND flash is read a page at a time, so the read time is acceptable as long as the storage access is sequential. Figure 2.7 illustrates the difference between the architecture of NAND and NOR.

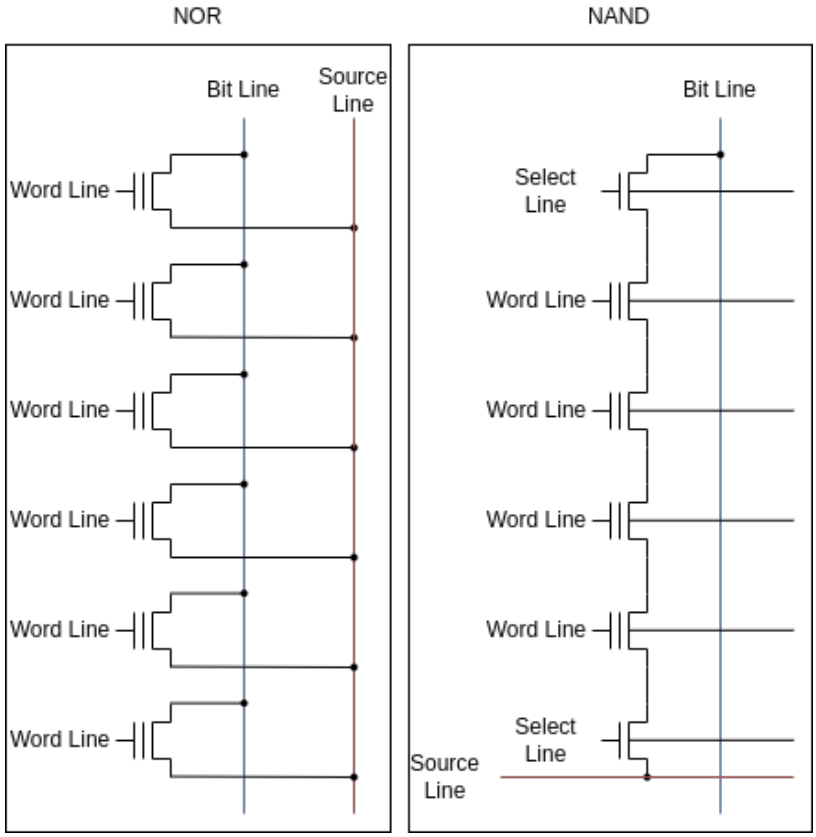


Figure 2.7: NOR and NAND Architecture

2.4 Array Structure

The structure of a NAND flash array is designed to be simple and compact. The array is made up of cells, strings, pages, and blocks. The design of a NAND block is shown in Figure 2.8, and Figure 2.9 shows how a NAND array is built out of blocks.

A NAND string is formed by NAND flash cells connected in series. A string typically contains 32 or 64 cells. High voltage transistors, Source Select Line (SSL)

and Ground Select Line (GSL), are added to either end of the string. These select lines are enabled when there is an operation within the string. They are disabled otherwise to protect the cells from overuse. The select lines must be capable of withstanding higher voltages due to the voltage ranges required for flash. The bitline is connected to the SSL transistor while the GSL transistor is connected to the source line. This differs from NOR cells, which each are connected to the bitline and source line.

A NAND block is created by connecting NAND strings to each other via their gates. Each word line goes through the entire block, connecting to the control gates of the corresponding cells. The flash cells that share a word line are called a page. This is illustrated in the horizontal red block in Figure 2.8. Unlike a NAND string, a NAND page does not have a typical size. The page is also not broken up like a string. The size of the page is the same as the number of bitlines of the flash array.

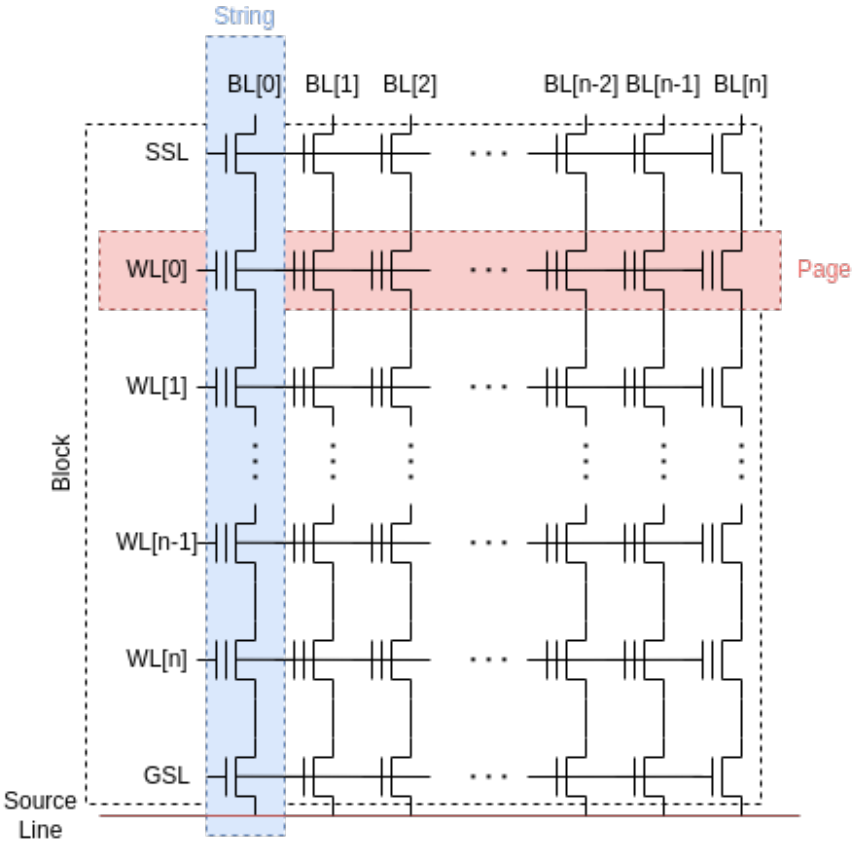


Figure 2.8: Structure of a NAND Block

Flash blocks are placed adjacent to each other with the bitline and source line contacts connecting. The blocks are mirrored over the contacts. A collection of blocks forms a flash array. The number of blocks is determined by the desired size of the flash array and the string size.

The structure of the NAND flash array has several advantages. The most obvious is the reduced area due to connecting the flash cells in series. Another is the way NAND flash is erased, programmed, and read. Erasing of NAND flash is done block by block. Programming and reading is done page by page.

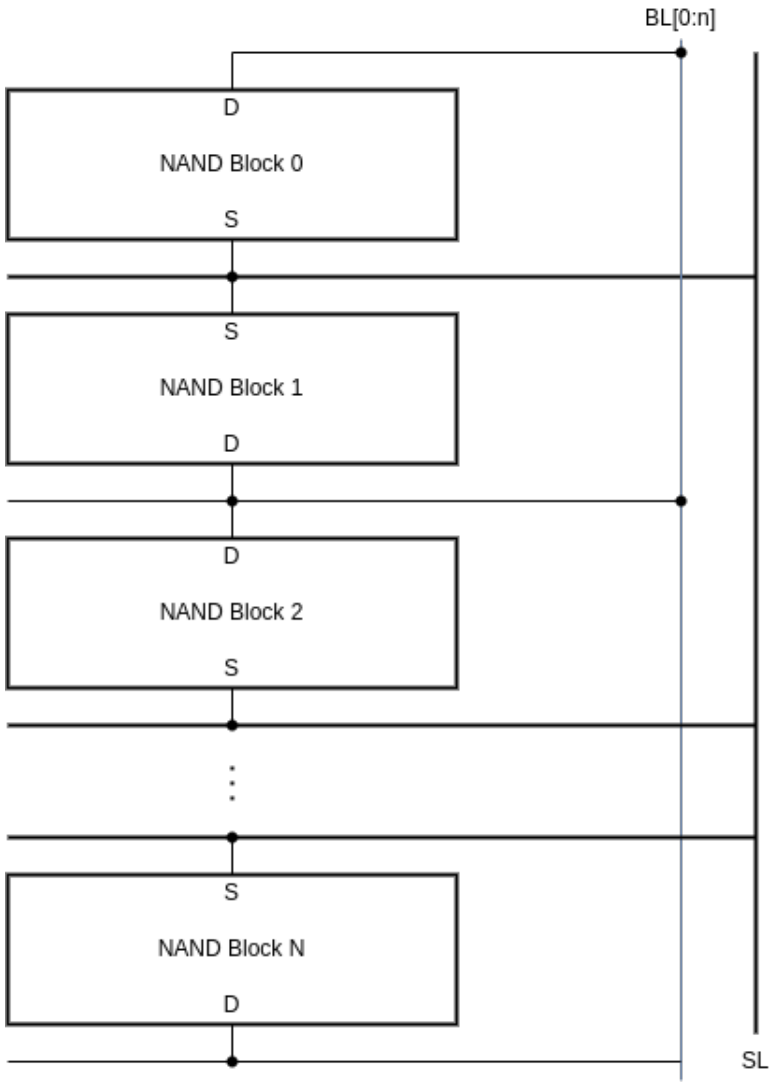


Figure 2.9: Structure of a NAND Array

CHAPTER III

IMPLEMENTATION

3.1 Flash Cell

The most basic building block of flash memory is the floating gate transistor and the single cell. The first step in this process is physical layout. The SKY130 process has access to the SONOS technology, which is why this thesis uses SONOS instead of the conventional floating gate. SONOS been used in SKY130 for a NOR Flash implementation. The NOR SONOS 2T cross section is shown in Figure 3.1. In order to design NAND flash, a new NAND flash cell was created. The cell is designed within specifications from SKY130.

The new NAND flash cell uses the same layers but omits a NPASS gate. Moreover, the area of the NAND cell is half of the area of the NOR cell. The width and length of the gate are specified by the design manual: .45/.22. SKY130 requires the following

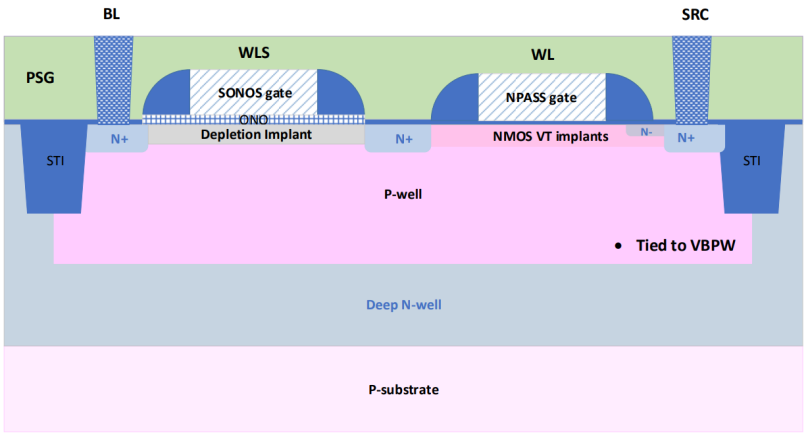


Figure 3.1: SONOS 2T. Referenced from [1]

Layer Name	LPP	Definition
diff	65:20	Active (diffusion) area (type opposite to the well/substrate underneath)
poly	66:20	Polysilicon
dnwell	64:18	Defines deep n-well regions
tunm	80:20	SONOS device tunnel implant
ldntm	11:44	N-tip implant on SONOS devices
nsdm	93:44	N+ source/drain implant
nwell	64:20	N-well regions
areaid.ce	81:2	Memory core for DRC purposes

Table 3.1: Layers needed for flash cell. Referenced from [1]

layers: diff, poly, dnwell, tunm, ldntm, nsdm, and no nwell. The presence of these layers signifies to the fabrication that the SONOS layer is present. Table 3.1 lists the layers and their definitions. There is also a memory mask, areaid.ce, that allows tighter design rules for ease in fabrication. An image of four flash cells is shown in Figure 3.2. Each intersection of the red 'poly' and green 'diffusion' is one NAND cell. The spacing between bit cells has been minimized according to the design rules within the memory mask. The total size of each single flash cell in this design is 0.72 µm x 0.43 µm. Thus, the area of a single flash cell is 0.3096 μm^2. The flash cell is made with Virtuoso, a Cadence Design Software tool, and the physical layout output uses the Graphic Design System (GDSII) format.

3.2 Flash Array

All figures in this section are screenshots from a flash array of size 8x8, with string size of 4. The purpose of this array is to illustrate the structure of the array in a way that is easy to see.

Since the structure of NAND flash is designed to be compact, turning a single cell into an array is quite simple. The array is basically structured in columns of

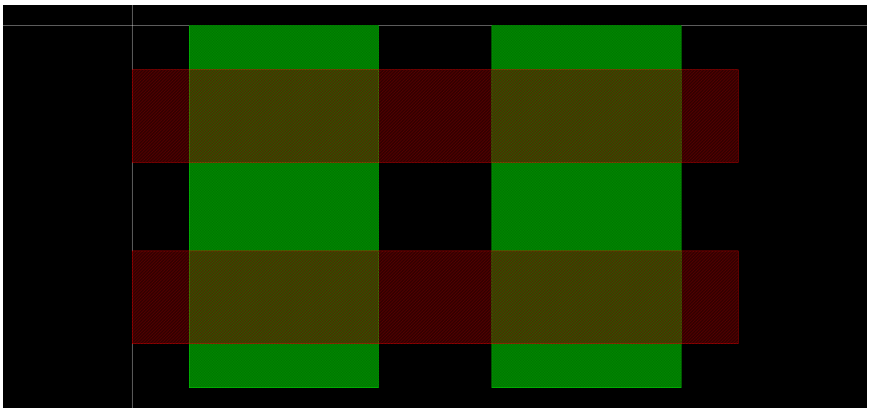


Figure 3.2: Flash Cell 2x2

bitlines and rows of word lines. The basic building piece of this design is a flash cell. Figure 3.2 shows four flash cells in a two-by-two structure. The two-by-two flash cells are copied horizontally and vertically to create a block with the string size provided by the user. The reasoning behind a two-by-two piece is to simplify copying the piece across the array, while also providing the user the maximum amount of ability to customize their design. Figure 3.3 shows the structure of the block. The block has eight bitlines and four word lines. It also has select lines at the top and bottom of the block. These are high voltage NMOS transistors. The pins for the word lines alternate between the left and right sides of the blocks. This is the conventional design. The alternative is the staggered row [2] design, which has the pins of the word lines all on the same side. The alternating pins allows for the design to be as compact as possible. The blocks are then stacked on each other to create the full array. The array is shown in Figure 3.4. It can be seen that the blocks in the example array are symmetrical along the source line.

3.3 Sense Amplifier

In order to get an output from the bitlines, sense amplifiers are designed to determine the state of the flash cells. The sense amplifiers 'sense' if there is current flow between

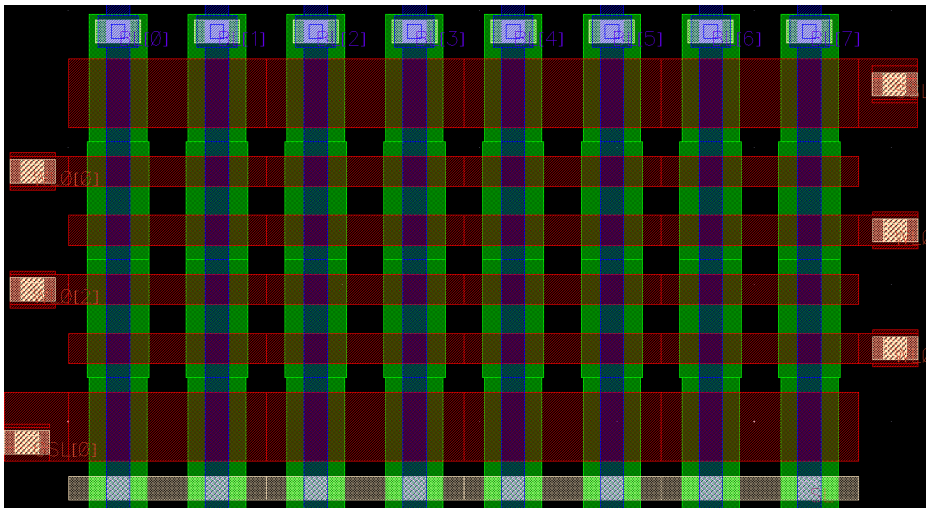


Figure 3.3: Flash Block

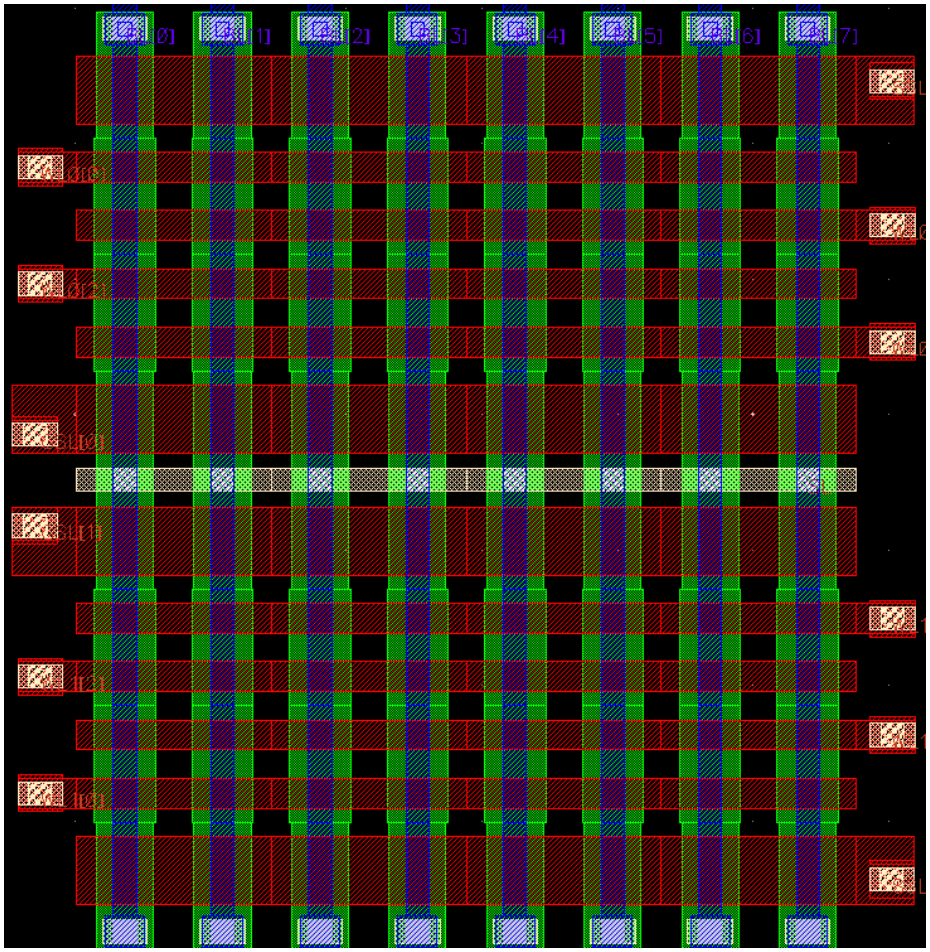


Figure 3.4: Flash Array

the drain and source of the cell. This allows differential-style voltages to be applied to a single-ended output. The two states are erased and programmed. This design uses one sense amplifier for each bitline, as opposed to other designs with one sense amplifier that reads two bitlines. This decision simplifies the sense amplifier and allows the read stage to be completed in one cycle instead of two. Since there is one sense amplifier for every bitline, the width of each sense amplifier needs to match the width a flash cell. This is challenging because the size of the flash cell has been minimized. In order to get around this, the sense amps are stacked on top of each other. This can be seen in the layout of Figure 3.7. The layout is symmetric vertically, showing the two sense amps. Stacking the sense amps enabled their width to be equal to the width of two flash cells. The extracted SPICE file of the two sense amplifiers is in Appendix A.

The sense amplifier is designed to be connected to each bitline. The bitline is treated as an inout signal. There are three inputs: *sen1*, *sen2*, and *out_en*. *sen1* and *sen2* are the control variables for the sense amplifier. These variables control all the sense amplifiers in the flash array. The other input is *out_en*, with connects the result of the sense amplifier to the outputs. There are two outputs with each sense amplifier, *out_even* and *out_odd*. These correspond to the bitlines. Figure 3.5 shows the schematic of the sense amplifier.

Figure 3.6 shows the operation of the sense amplifier. The operation starts with the pre-charging of the bitlines to 1.8 V. The pre-charging of the bitlines is done in the control logic and not in the sense amplifiers. Once the bitlines are at the correct voltage, the input *sen2* is set to VDD. Since the bitlines are set to VDD, the transistors they are controlling are turned on, resulting in the drain of the transistor controlled by *sen2* to be GND. The latch will force the drain of *sen1* to be set to VDD. *sen2* is then set to GND.

In the second stage, the bitlines are released from the pre-charge. The bitline will

either discharge to GND or remain at VDD. Once the bitlines are stable, the input *sen1* is set to VDD. If the flash cell is in an erased state, the bitline is discharged. Since the bitline is discharged, its transistor is turned off, resulting in no change within the latch. If the flash cell is in the programmed state, the bitline would remain at VDD. This would turn on the bitline's transistor, resulting in *sen1*'s drain to be set to GND, switching the latch. *sen1* is then set to GND.

The final stage is set out_en to VDD, enabling the output of the sense amplifiers. The output of an erased state cell would be '1'. The output of a programmed state cell would be '0'.

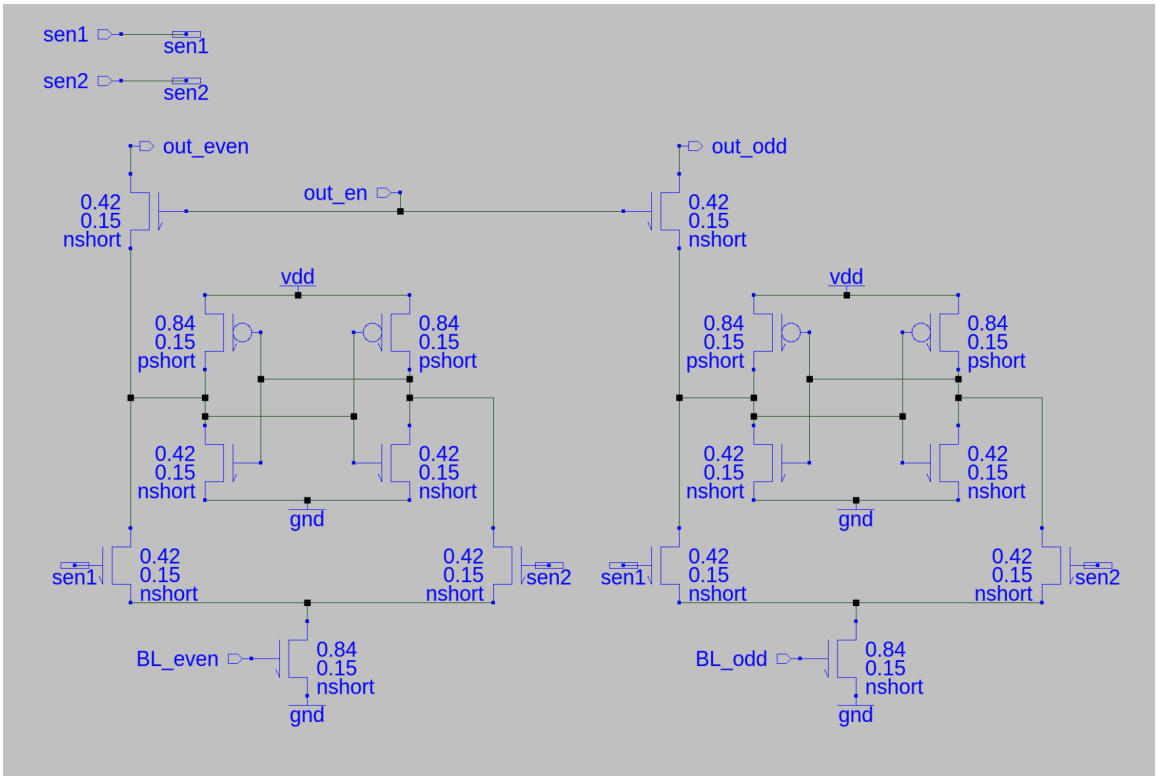


Figure 3.5: Sense Amplifier Schematic

The design of the sense amplifier is based off of the sense amplifier on page 274 of [2]. However, it has been adapted to have two sense amplifiers in the design and only have the reading function. The referenced sense amplifier is shown in Figure 3.8.

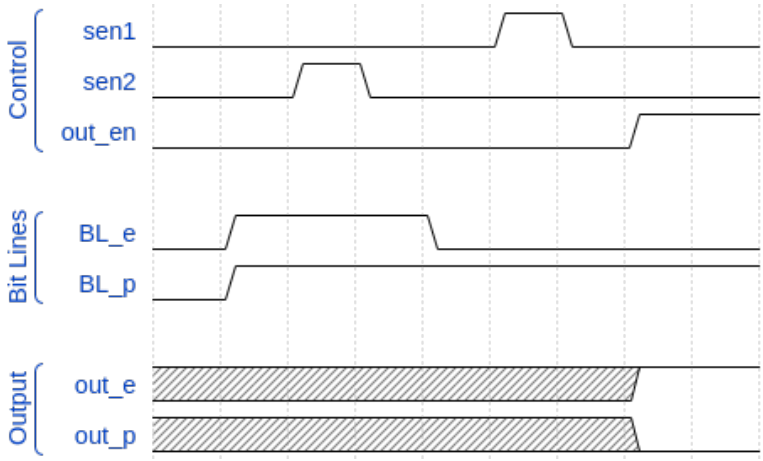


Figure 3.6: Sense Amplifier Operation

3.4 Script

With the individual components of the design created, a script is designed to compile the components into a functioning flash array. The script is written in Python and uses GDS Mill by Michael Wieckowski [20]. GDS Mill is an open source software that can read in and output binary GDSII files. It is capable of reading and placing individual GDSII files, creating a top level GDSII file. This script makes use of GDS Mill to shape the components into the array. The script is in Appendix B.

The components are categorized into three types: cells, ends, and middle. *flash_cell_2x2*, *flash_left_2*, and *flash_right_2* are the cell type. They include the actual SONOS gates. *flash_cell_2x2* is four SONOS gates organized in a 2x2 manner. The left and right components include the pins of the word lines as well as the tap that surrounds the array. The left and right components for each type have a similar role. The ends consist of *flash_end_2*, *flash_end_left*, and *flash_end_right*. These are placed at the top and bottom of the array. There are also *flash_bot_2*, *flash_bot_left*, and *flash_bot_right*. These are at the bottom of the array and include the sense amplifiers connected to the ends. Finally, there is the middle type: *flash_midSL_2*, *flash_midSL_left*, f*lash_midSL_right*, *flash_midBL_2*, f*lash_midBL_left*, and *flash_midBL_right*. The middle type links the

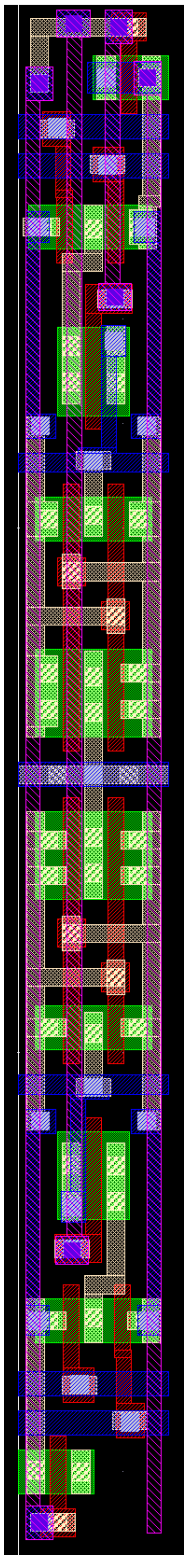


Figure 3.7: Sense Amplifier Layout

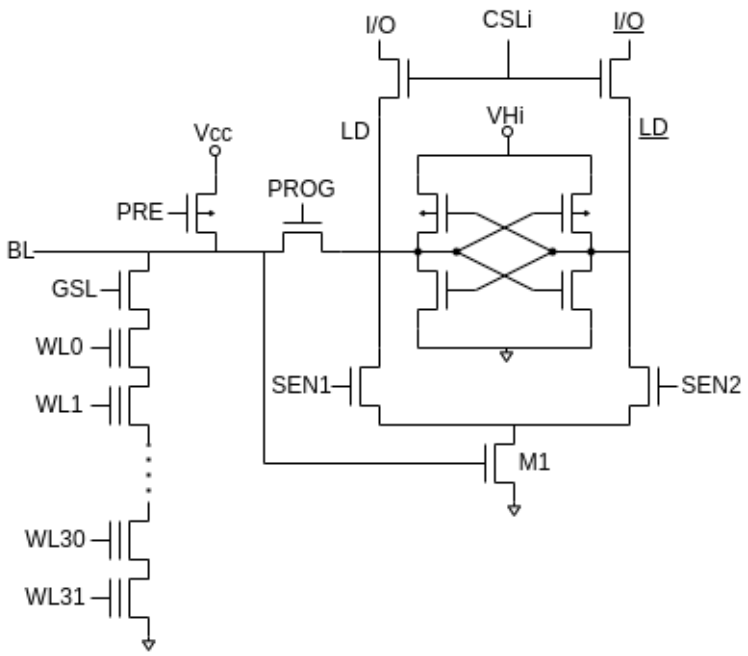


Figure 3.8: Sense Amplifier used as reference. Adapted from Figure 6.78a in [2]

flash blocks together. The components with 'midSL' connect the blocks that share the source line. The components with 'midBL' connect the blocks that have their bitlines connected directly.

The script takes three inputs from the command line. First is the number of columns or bitlines. This is also referred to as the page size. The user can make the page any size. This allows for the user to create a flash array that includes error correction bits. The second input is the number of rows or word lines. The final input is block size or string size. With the way the script is designed, the string size must be divisible by two. The number of rows should also be a multiple of Block Size X 2, due to the symmetric nature of the design. Figure 3.9 illustrates a 8x8 flash array with the sub cells shown. The number of columns is eight, number of rows is eight, and the block size is four. The result of the script is an array of the size specified. The output is a GDSII file that can be fabricated.

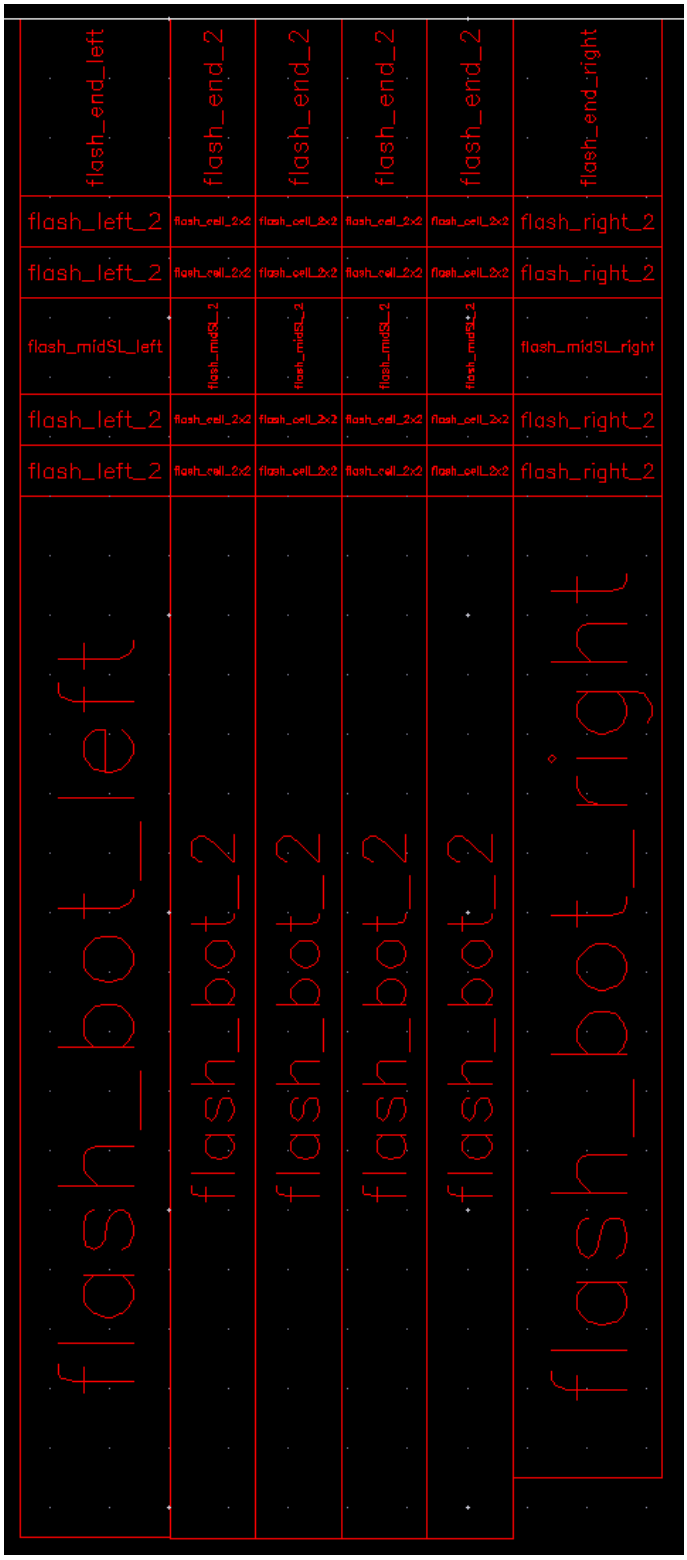


Figure 3.9: Flash Array Created by Script

3.5 Fabrication

With the script designed, it is easy to create a flash array of any size. Simulation is possible and will be discussed in the next section. However, the information gained from simulation is limited. We rely completely on the models provided by SKY13O. These models, although it would be beneficial to have a physically implemented array that can be tested using a probe card. This would enable the testing of the flash array to obtain accurate specifications for both the timing as well as the optimal operating voltages of this newly-designed NAND flash array.

The 8x8 flash array has been sent to fabrication on the Efabless Open MPW Shuttle Program [21]. This program is a collaboration of Google and Efabless to provide opportunities for designers to fabricate their research without having to worry about the cost associated. The costs are covered by Google as long as the design is completely open source. This program also uses SKY130. Caravel [3] is a SoC used for the Efabless Open MPW shuttle. It is shown in Figure 3.10. The flash array will be placed within the 'User Project Wrapper' block and connected to the rest of the Caravel Harness Chip. Below are the detailed steps required for this process.

In order to create the caravel harness, Efabless requires users make a public git repository that contains the user project wrapper. This repository can be found here: [22]. A GDSII and LEF of the flash array are placed into the repository. Verilog files are also required to connect the design to the wrapper. These files are referenced in the appendix. The flash array will be treated like a macro. Only the pins are relevant, the inside of the design is unknown to the project wrapper. The pins of the macro are connected to the pins of the user project wrapper by OpenLane [23], a place and route program that hardens the cells. When OpenLane is finished, the design is ready to be placed into the caravel harness.

A pre-check is completed to ensure the design meets all requirements and is free

of errors. Finally, the design was submitted for fabrication. Figure 3.11 shows the submitted design. The successful submission of the design proves that the array output is fabricable and DRC-free.

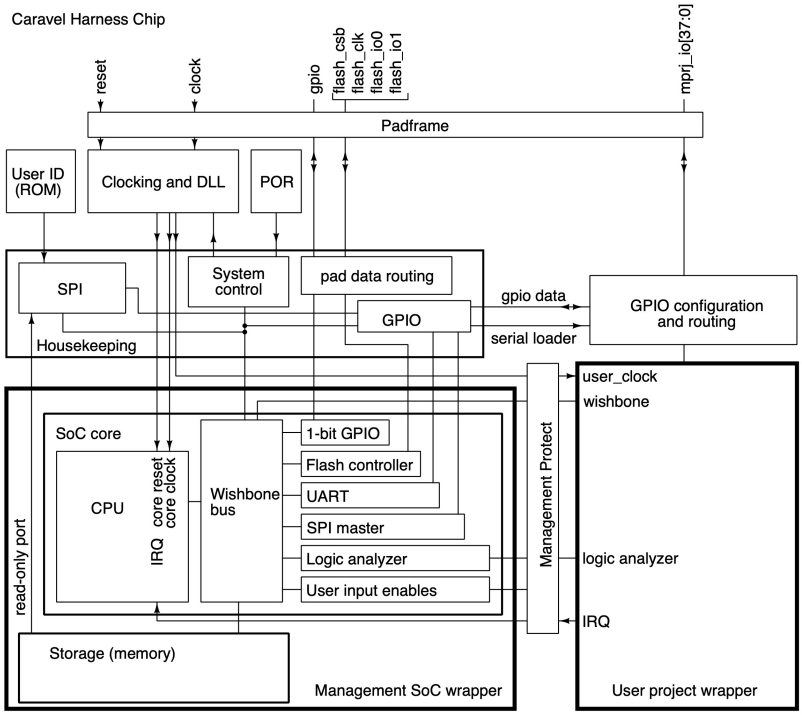


Figure 3.10: Caravel Harness [3]

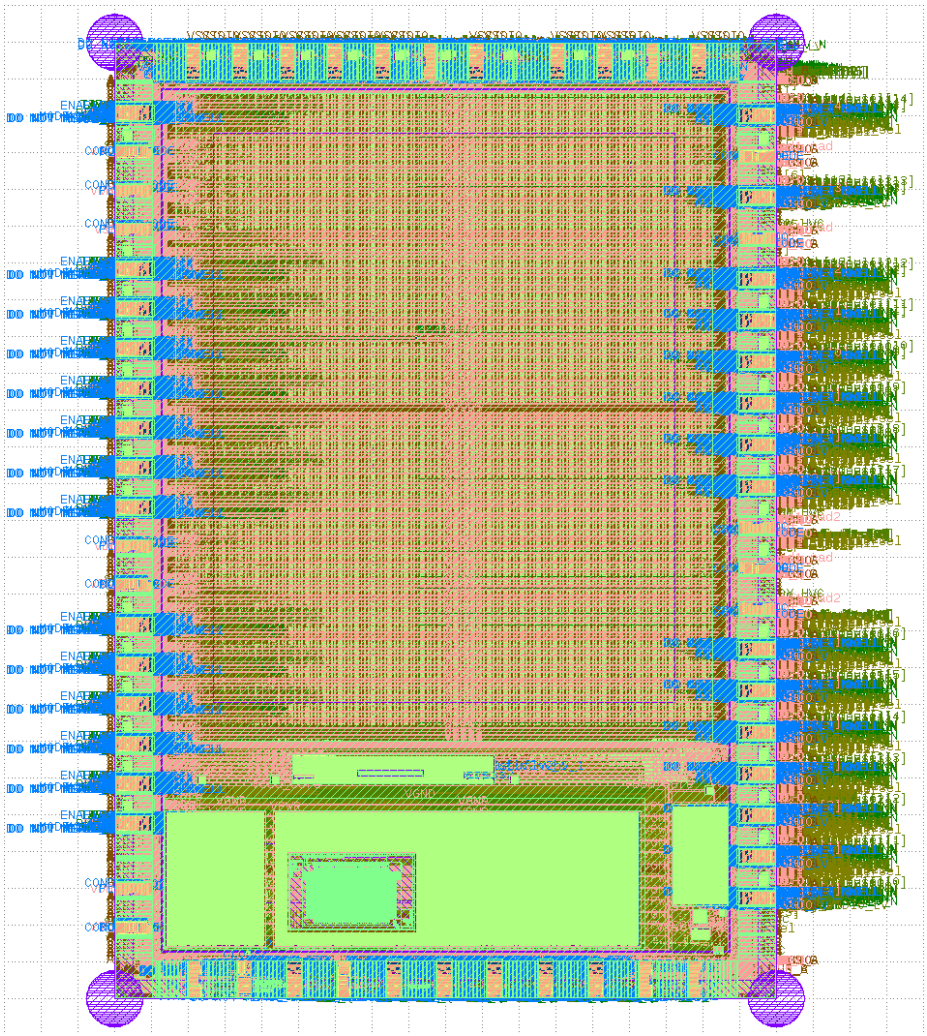


Figure 3.11: Caravel Harness Chip

CHAPTER IV

RESULTS

Since the only results achievable for this thesis are reliant on the models provided by SkyWater, simulation is the only way forward. The models used for simulation are SPECTRE models. SPECTRE is a simulator designed by Cadence Design System. It utilizes the SPICE engine found in all spice simulators and can use SPICE files to run the simulation.

In SPICE, inputs can be controlled and set to a specific voltage. This is helpful for simulating flash, for example, because flash requires a large range of voltages. However, after setting an input to a specified voltage, it is impossible to stop controlling the input. In other words, SPICE can have inputs and outputs, but no inout. This is no problem for the control gates and select gates of the flash array, but the bitlines require inout pins to function. During the program and erase stages, the bitlines can be controlled completely with no issues. The reading stage requires the bitlines to be uncontrolled, allowing them to either discharge or remain unchanged. The obvious solution is the tri-state buffer.

The buffer has two inputs: enable and in. When enable is '1', out is connected to in, so the output equals the input. When enable is '0', the output is in a state of high impedance. The input has no affect on the output. The issue with the tri-state buffer is it only works with digital voltages, GND and VDD. Flash requires a greater range of inputs. The solution is a modified tri-state buffer that can handle a large range of inputs. This subcircuit can be seen in Figure 4.1. The SPICE file of this can be found in Appendix C. The inputs and outputs are similar to that of a tri-state

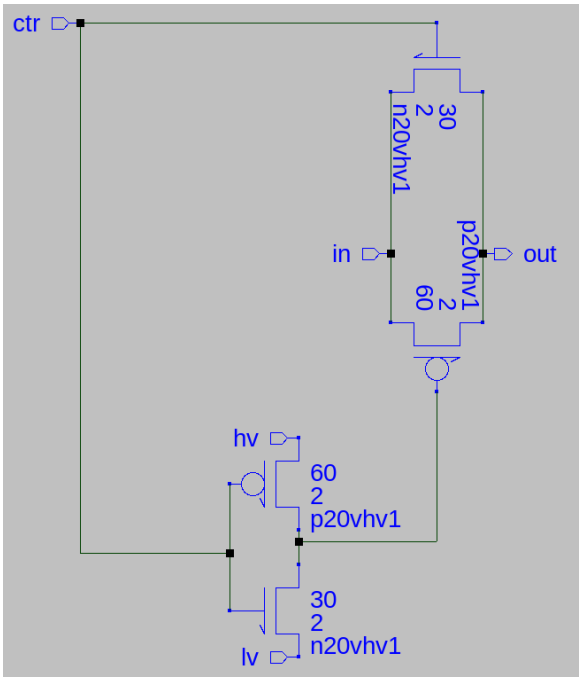


Figure 4.1: Modified Tri-State Buffer

buffer. *hv* and *lv* replace VDD and GND respectively. *ctr* is similar to enable and in/out is the same as in the tri-state buffer. When *ctr* is equal to *hv*, out is equal to in. When *ctr* is equal to *lv*, out is in a state of high impedance. Normal CMOS transistors are replaced by n20vhv1 and p20vhv1, which can handle the higher range of voltages. This comes at a cost of much greater area. Since the modified tri-state buffer is used only in simulations, the increased area not important.

As mentioned before, simulation is limited by the models given in SKY130. SkyWater only includes SONOS models that are in the erased state or the programmed state. Simulation of the reading phase is possible, but programming and erasing is not. The names of the spice models are in Table 4.1.

A pseudo-simulation can be done to ensure the flash array is functional, however it is not accurate. A floating gate simulation model is designed based off of Steven Joseph Rapp's simulation model [24]. A large resistor, capacitor and a couple current sources are used to roughly approximate the programming and erasing of a floating

Stage	Programmed	Erased
Beginning of Life	sonos_bol_p	sonos_bol_e
End of Life	sonos_eol_p	sonos_eol_e

Table 4.1: SONOS Spice Models (Referenced from [1])

gate transistor. The difference between an erased and programmed floating gate is basically the threshold of the control gate. The threshold of a programmed gate is positive, while an erased gate is negative. The new simulation model includes a SONOS model in the programmed state attached to the extra circuitry. When the voltages required for erasing are set, the erase current is turned on. This current raises the voltage applied to the control gate of the SONOS model. That makes it easier to overcome the gate threshold. This is equivalent to reducing the gate threshold. The model now acts as a SONOS model in the erased state. The simulation can be programmed back to the programmed state. This simulation model allows us to ensure the correct voltages are being applied throughout the flash array. The new SPICE model is in Appendix D.

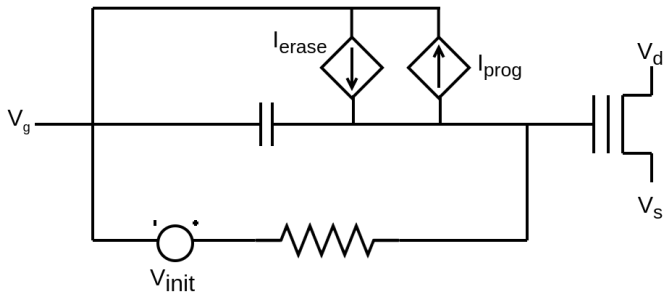


Figure 4.2: FGMOS Model

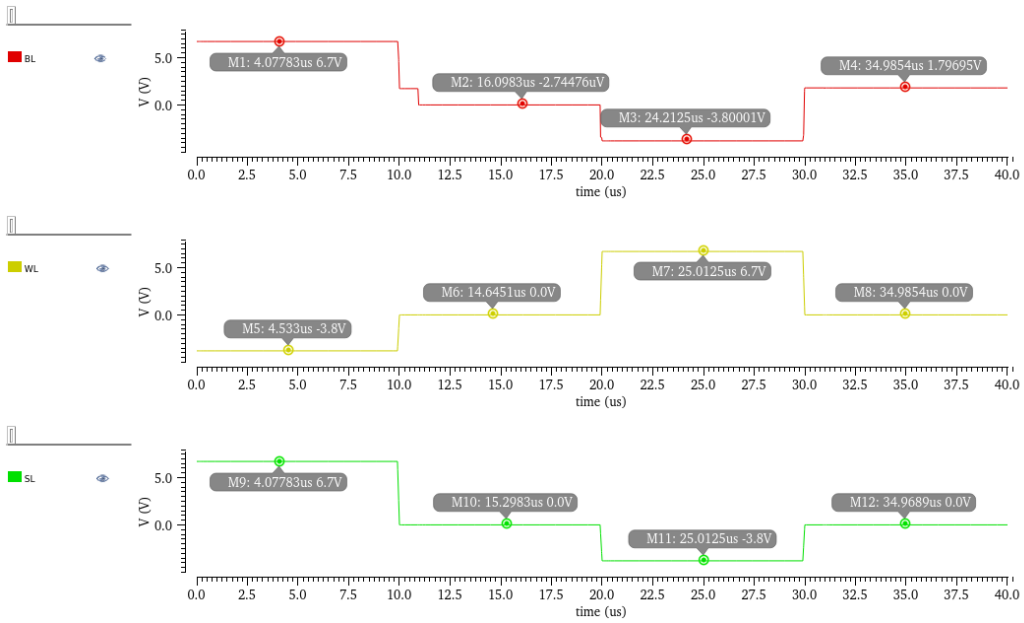


Figure 4.3: Erasing and Programming Simulation

4.1 Results

Figure 4.3 shows the simulation of the erasing and programming of the new SONOS model. The simulation has four phases: erase, read, program, read. Each phase is 10 µs long. In the first, 0 to 10 µs, the model is being erased. The bitline and source line are set to 6.7 V while the word line or gate is set to −3.8 V. The models threshold voltage is shifted to negative. In the second phase, the bitline is pre-charged and then disconnected. The word line and source line are both set to 0 V. Since the model is erased, the bitline discharges to 0 V. The model is programmed in the third phase. The bitline and source line are set to −3.8 V. The word line is set to 6.7 V. The model's threshold voltage increases to positive. The forth phase is the final reading. The inputs are set to the same as in phase two. This time, since the model is programmed, the bitline remains at 1.8 V. This simulation behaves as expected and proves that the floating gate model is valid.

Figure 4.4 shows the simulation of the sense amplifier. The simulation is of the reading stage of NAND flash. The operation of the sense amplifier can be seen in

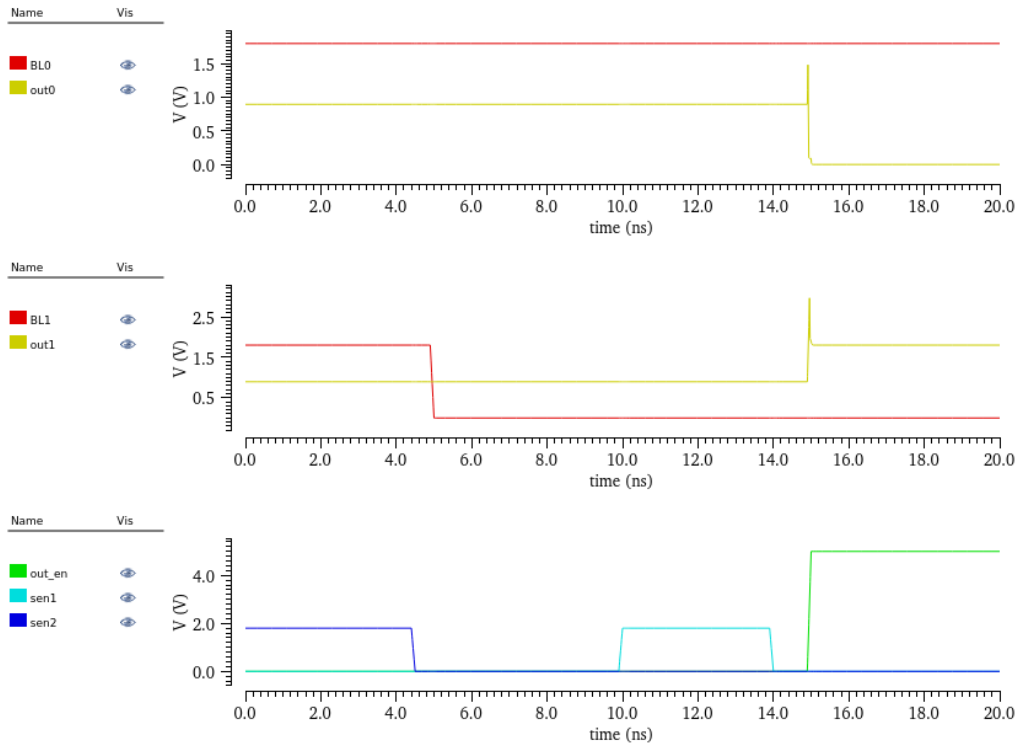


Figure 4.4: Sense Amplifier Simulation

Figure 3.6. The top section is a programmed flash cell. The middle section is an erased flash cell. The bottom section is the controls of the sense amplifier. The red lines indicate the bitlines and the yellow indicates the output. There are four phases: 0 to 5 ns, 5 to 10 ns, 10 to 15 ns, and 15 to 20 ns. In the first phase, the bitlines are pre-charged and *sen2* is turned on. In phase two, the bitlines are disconnected and *sen2* is turned off. The programmed bitline remains the same while the erased bitline discharges to GND. During phase three, *sen1* is turned on and the latch in the sense amplifier is being prepared for the final output. *sen1* is then turned off. In the final phase, *out_en* is turned on and the output is enabled. The programmed flash cell has an output of '0' and the erase flash cell has an output of '1'.

The last simulation is a simulation of the flash array. A 32x32 flash array with string size of 16 is built and the SPICE file is extracted from the GDSII. This simulation is performed to ensure the flash array is working correctly. To start, all of

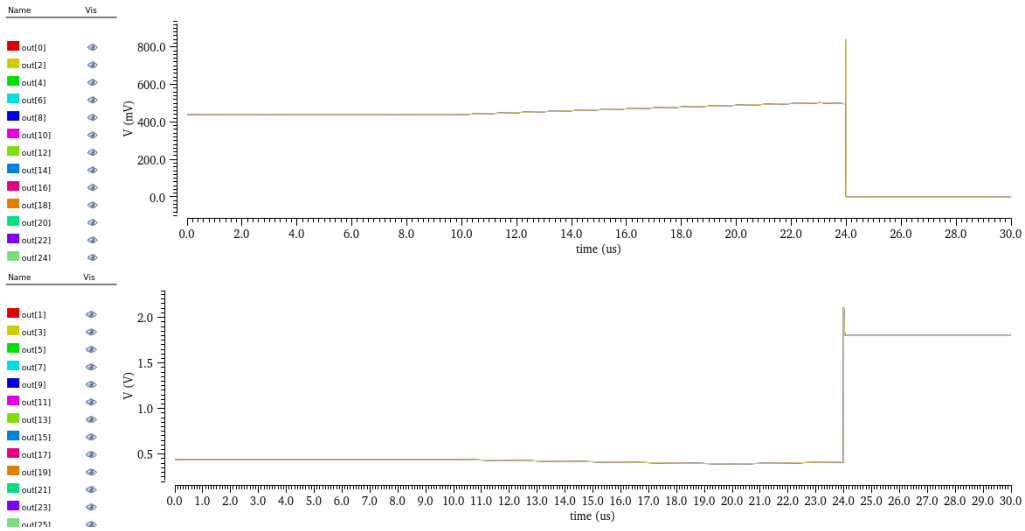


Figure 4.5: Simulation of Flash Array

block 0 is erased. Then, page 1 of block 0 is programmed. The bitlines are alternately programmed and program-inhibited. The even bits are programmed while the odd bits are program-inhibited. Program-inhibited means that the cells are left in the erased state. Finally, when the page is read, the output is 0xAAAAAAAA. The simulation shows that erasing by block, programming by page, and reading by page all function correctly. It also proves that blocks and pages that are not part of the operation are left alone. Figure 4.5 shows the outputs of the simulation. The even final outputs are '0' and the odd final outputs are '1'. This results in a final 32-bit output of 0xAAAAAAAA.

Table 4.2 shows the sizes of several flash arrays as well as a single flash cell and sense amplifier. The NAND flash cell is half the size of a NOR flash cell. When the size of the array is smaller, the sense amplifiers and outside of the array have an impact on the area. But as the array grows, the sense amplifier becomes insignificant.

Name	Width (μm)	Height (μm)	Area (μm^2)
NOR Flash Cell	0.72	0.86	0.62
NAND Flash Cell	0.72	0.43	0.31
Sense Amp x 2	1.44	17.48	25.17
Flash Array 8 x 8 (8 B)	10.79	25.52	275.36
Flash Array 32 x 32 (128 B)	28.07	35.84	1,006.03
Flash Array 256 x 256 (8 MB)	189.35	141.82	26,853.62
Flash Array 1024 x 2048 (256 MB)	742.31	1,002.54	744,195.47

Table 4.2: Layout Sizes

CHAPTER V

CONCLUSION AND FUTURE RESEARCH

This thesis accomplishes several goals. First is to design a NAND flash cell using SONOS technology from SKY130. Second is to create a compiler capable of taking size inputs from the user and outputting a complete GDSII file of the corresponding flash array. The final goal is to physically implement the design by fabricating the flash array.

NOR flash using SKY130 has been implemented by a separate research group. In order to implement NAND flash, a new NAND flash cell is created using the SONOS technology. This cell is more compact and follows the design specifications required by SKY130. Peripherals are also created to surround the cell and make a valid array.

The compiler is accomplished using a Python script. The user inputs the size of the array, and the script creates a GDSII file of the flash array from the individual parts that are hand drawn. The output is able to be fabricated and DRC-free.

Since the design is not yet physical, the testing and simulations done are limited. A 8x8 flash array has been submitted to the Efabless MPW-5 shuttle to be fabricated. It was submitted on March 15, 2022.

5.1 Future Research

Future work consists of three parts: receiving and testing the fabricated flash array, making the compiler more customizable, and building the logic and peripherals around the flash array that are required for fully functional NAND flash.

According to Efabless, the fabricated flash array will be shipped on August 2, 2022. Once the flash array is received, testing can begin. Through Caravel, there is access to each pin. Precise timing for the array and each cell can be found. Different voltages can be tested to find the optimal combination.

The compiler can be improved to support multiple planes or different orientations. More options available to academic researchers only helps with their research and innovation.

The end goal of this project is to compile fully functional NAND flash. The output should include the final GDSII, timing, and test benches. The user should be able to run the script, then easily submit the design for fabrication. The final design will not only include the flash array, but also all of the peripherals. These peripherals include control logic, row decoder, column decoder, charge pumps, voltage regulators, and high voltage switches. Eventually the design should have an input of the address and operation.

REFERENCES

[1] SkyWater, "SkyWater SKY130 PDK." `https://skywater-pdk.readthedocs.io/en/main/index.html`.

[2] J. Brewer and M. Gill, *Nonvolatile memory technologies with emphasis on Flash : a comprehensive guide to understanding and using NVM devices*. IEEE Press series on microelectronic systems, Hoboken, N.J: Wiley, 2008.

[3] Efabless, "Caravel Harness." `https://github.com/efabless/caravel`.

[4] F. Masuoka and H. Iizuka, "Semiconductor memory device and method for manufacturing the same," Jul 1985.

[5] C. Monzio Compagnoni, A. Goda, A. S. Spinelli, P. Feeley, A. L. Lacaita, and A. Visconti, "Reviewing the Evolution of the NAND Flash Technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1609–1633, 2017.

[6] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[7] S. K. Park, "Technology Scaling Challenge and Future Prospects of DRAM and NAND Flash Memory," in *2015 IEEE International Memory Workshop (IMW)*, pp. 1–4, 2015.

[8] K. Takeuchi, T. Tanaka, and T. Tanzawa, "A multipage cell architecture for high-speed programming multilevel NAND flash memories," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 8, pp. 1228–1238, 1998.

[9] G. H. Lee, S. Hwang, J. Yu, and H. Kim, "Architecture and Process Integration Overview of 3D NAND Flash Technologies," *Applied Sciences*, vol. 11, no. 15, 2021.

[10] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *FAST*, vol. 7, pp. 10–2, 2012.

[11] S. Li and T. Zhang, "Improving multi-level NAND flash memory storage reliability using concatenated BCH-TCM coding," *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 18, no. 10, pp. 1412–1420, 2009.

[12] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 504–517, IEEE, 2018.

[13] X. Jimenez, D. Novo, and P. Ienne, "Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance," in *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 47–59, 2014.

[14] X. Shi, F. Wu, S. Wang, C. Xie, and Z. Lu, "Program error rate-based wear leveling for NAND flash memory," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1241–1246, IEEE, 2018.

[15] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An Open-Source Memory Compiler," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[16] "SkyWater Technology." `https://www.skywatertechnology.com/`.

[17] R. G. Forbes and J. H. B. Deane, "Reformulation of the standard theory of Fowler-Nordheim tunnelling and cold field electron emission," *Proceedings: Mathematical, Physical and Engineering Sciences*, vol. 463, no. 2087, pp. 2907–2927, 2007.

[18] K. Ramkumar, "Cypress SONOS - A Scalable Embedded Flash Technology." `https://www.chipestimate.com/Cypress-SONOS-A-Scalable-Embedded-Flash-Technology/Cypress-Semiconductor/Technical-Article/2008/10/21`, 2008.

[19] P. Dimitrakis, *Charge-Trapping Non-Volatile Memories Volume 1 – Basic and Advanced Devices*. Springer International Publishing, 1st ed., 2015.

[20] M. Wieckowski, "GDS Mill." `http://michaelwieckowski.com/software/`, 2022.

[21] "Welcome to the efabless open MPW shuttle program." `https://platform.efabless.com/open_shuttle_program/5`.

[22] B. T. Ong, "Caravel User Project." `https://github.com/BrandonOng22/caravel_user_project`, 2022.

[23] "OpenLane." `https://github.com/The-OpenROAD-Project/OpenLane`.

[24] S. J. Rapp, "A comprehensive simulation model for floating gate transistors," Master's thesis, West Virginia University, Morgantown, West Virginia, 2010.

[25] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash memories*. Springer, 2010.

[26] S. Jeloka, J. Lee, Z. Li, J. Shah, Q. Dong, K. Yang, D. Sylvester, and D. Blaauw, "An ultra-wide program, 122pj/bit flash memory using charge recycling," in *2017 Symposium on VLSI Circuits*, pp. C196–C197, 2017.

APPENDICES

APPENDIX A: FLASH SENSE AMPLIFER 2 SPICE FILE

```
* SPICE NETLIST
**************************************

.SUBCKT Dpar d0 d1
.ENDS
**************************************
.SUBCKT flash_senseamp2 gnd vdd sen2 out_en sen1 BL_even
   BL_odd out_odd out_even
** N=17 EP=11 IP=0 FDC=17
M0 out_even out_en 11 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a
   =0.063 p=1.14 mult=1 $X=310 $Y=-13965 $D=9
M1 14 sen2 12 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=355 $Y=-2220 $D=9
M2 17 sen1 11 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=430 $Y=-12655 $D=9
M3 gnd 6 11 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=430 $Y=-9865 $D=9
M4 gnd 7 12 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=430 $Y=-5010 $D=9
M5 17 BL_even gnd gnd nshort L=0.15 W=0.84 m=1 r=5.6 a
   =0.126 p=1.98 mult=1 $X=645 $Y=-11485 $D=9
M6 gnd BL_odd 14 gnd nshort L=0.15 W=0.84 m=1 r=5.6 a
   =0.126 p=1.98 mult=1 $X=645 $Y=-3810 $D=9
M7 6 11 gnd gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=860 $Y=-9865 $D=9
M8 7 12 gnd gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=860 $Y=-5010 $D=9
M9 7 sen1 14 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=860 $Y=-2220 $D=9
M10 6 sen2 17 gnd nshort L=0.15 W=0.42 m=1 r=2.8 a=0.063 p
   =1.14 mult=1 $X=935 $Y=-12655 $D=9
M11 7 out_en out_odd gnd nshort L=0.15 W=0.42 m=1 r=2.8 a
   =0.063 p=1.14 mult=1 $X=980 $Y=-910 $D=9
M12 vdd 6 11 vdd pshort L=0.15 W=0.84 m=1 r=5.6 a=0.126 p
   =1.98 mult=1 $X=430 $Y=-8430 $D=79
M13 vdd 7 12 vdd pshort L=0.15 W=0.84 m=1 r=5.6 a=0.126 p
   =1.98 mult=1 $X=430 $Y=-6880 $D=79
M14 6 11 vdd vdd pshort L=0.15 W=0.84 m=1 r=5.6 a=0.126 p
   =1.98 mult=1 $X=860 $Y=-8430 $D=79
M15 7 12 vdd vdd pshort L=0.15 W=0.84 m=1 r=5.6 a=0.126 p
   =1.98 mult=1 $X=860 $Y=-6880 $D=79
X16 gnd vdd Dpar a=3.96 p=8.38 m=1 $[nwdiode] $X=0 $Y
```

```
   =-8610 $D=185
*.CALIBRE WARNING OPEN Open circuit(s) detected by
   extraction in this cell. See extraction report for
   details.
.ENDS
****************************************
```

APPENDIX B: NAND FLASH COMPILER SCRIPT

```
#!/usr/bin/env python
import sys
import gdsMill

input_path = "../../virtuoso/gds/"
output_path = "../gds/"

if len(sys.argv) > 1:
    numCols = int(sys.argv[1])
else:
    numCols = 1024
if len(sys.argv) > 2:
    numRows = int(sys.argv[2])
else:
    numRows = 1024
if len(sys.argv) > 3:
    blockSize = int(sys.argv[3])
else: blockSize = 32

out_name = "flash_array_" + str(numCols) + "x" + str(
   numRows)

# Size of cells
SIDE_WIDTH = 2.515
MID_WIDTH = 1.44
END_HEIGHT = 2.96
CELL_HEIGHT = 0.86
MID_HEIGHT = 1.61

# Location of labels
BL_end_x = 2.875
BL_BL_x = .72
WL_x = 0.26

y_offset = 0

cellMidLayout = gdsMill.VlsiLayout()
cellLeftLayout = gdsMill.VlsiLayout()
cellRightLayout = gdsMill.VlsiLayout()

endMidLayout = gdsMill.VlsiLayout()
endLeftLayout = gdsMill.VlsiLayout()
endRightLayout = gdsMill.VlsiLayout()

botMidLayout = gdsMill.VlsiLayout()
```

```
botLeftLayout = gdsMill.VlsiLayout()
botRightLayout = gdsMill.VlsiLayout()

BLMidLayout = gdsMill.VlsiLayout()
BLLeftLayout = gdsMill.VlsiLayout()
BLRightLayout = gdsMill.VlsiLayout()

SLMidLayout = gdsMill.VlsiLayout()
SLLeftLayout = gdsMill.VlsiLayout()
SLRightLayout = gdsMill.VlsiLayout()

reader = gdsMill.Gds2reader(cellMidLayout)
reader.loadFromFile(input_path + "flash_cell_2x2.gds")
reader = gdsMill.Gds2reader(cellLeftLayout)
reader.loadFromFile(input_path + "flash_left_2.gds")
reader = gdsMill.Gds2reader(cellRightLayout)
reader.loadFromFile(input_path + "flash_right_2.gds")

reader = gdsMill.Gds2reader(endMidLayout)
reader.loadFromFile(input_path + "flash_end_2.gds")
reader = gdsMill.Gds2reader(endLeftLayout)
reader.loadFromFile(input_path + "flash_end_left.gds")
reader = gdsMill.Gds2reader(endRightLayout)
reader.loadFromFile(input_path + "flash_end_right.gds")

reader = gdsMill.Gds2reader(botMidLayout)
reader.loadFromFile(input_path + "flash_bot_2.gds")
reader = gdsMill.Gds2reader(botLeftLayout)
reader.loadFromFile(input_path + "flash_bot_left.gds")
reader = gdsMill.Gds2reader(botRightLayout)
reader.loadFromFile(input_path + "flash_bot_right.gds")

reader = gdsMill.Gds2reader(BLMidLayout)
reader.loadFromFile(input_path + "flash_midBL_2.gds")
reader = gdsMill.Gds2reader(BLLeftLayout)
reader.loadFromFile(input_path + "flash_midBL_left.gds")
reader = gdsMill.Gds2reader(BLRightLayout)
reader.loadFromFile(input_path + "flash_midBL_right.gds")

reader = gdsMill.Gds2reader(SLMidLayout)
reader.loadFromFile(input_path + "flash_midSL_2.gds")
reader = gdsMill.Gds2reader(SLLeftLayout)
reader.loadFromFile(input_path + "flash_midSL_left.gds")
reader = gdsMill.Gds2reader(SLRightLayout)
reader.loadFromFile(input_path + "flash_midSL_right.gds")

newLayout = gdsMill.VlsiLayout(name=out_name)

def placeTopRow(block):
    x_offset = 0
    global y_offset
```

```
mirror = None
BL_y = 2.155
fnpass_y = 0.42

psub_x = 1.765
# psub_y = 1.645
psub_y = 9.325

newLayout.addInstance(endLeftLayout,
                      offsetInMicrons=(x_offset,
                         y_offset),
                      mirror = mirror)
newLayout.addBox(layerNumber=68,
                 dataType=20,
                 offsetInMicrons=(x_offset + psub_x,
                    y_offset-psub_y),
                 width=.1,
                 height=.1,
                 center=True)
newLayout.addText(text = "VBPW",
                  layerNumber=68,
                  purposeNumber=16,
                  offsetInMicrons=(x_offset + psub_x,
                     y_offset - psub_y))
x_offset += SIDE_WIDTH
for i in range(numCols/2):
    newLayout.addInstance(endMidLayout,
                          offsetInMicrons=(x_offset,
                             y_offset),
                          mirror=mirror)
    newLayout.addBox(layerNumber=68,
                     dataType=20,
                     offsetInMicrons=(x_offset +
                        BL_BL_x/2,y_offset - BL_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "BL[" + str(2*i) + "]",
                      layerNumber=68,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset +
                         BL_BL_x/2,y_offset - BL_y))
    x_offset += MID_WIDTH
    newLayout.addBox(layerNumber=68,
                     dataType=20,
                     offsetInMicrons=(x_offset -
                        BL_BL_x/2,y_offset - BL_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "BL[" + str(2*i+1) + "]",
                              layerNumber=68,
                              purposeNumber=16,
```

```
                                        offsetInMicrons=(
                                           x_offset - BL_BL_x/2,
                                           y_offset - BL_y))

    newLayout.addInstance(endRightLayout,
                          offsetInMicrons= (x_offset,
                             y_offset),
                          mirror = mirror)

    y_offset-=END_HEIGHT
    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset + WL_x,
                        y_offset + fnpass_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "SSL[" + str(block) + "]",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset + WL_x,
                         y_offset + fnpass_y))

def placeBotRow(block):
    x_offset = 0
    global y_offset

    mirror = None
    fnpass_y = 0.42
    m1_x = 0.2
    gnd1 = 7.202
    gnd2 = 13.142
    vddy = 10.18
    sen1_1 = 4.37
    sen1_2 = 15.995
    sen2_1 = 4.00
    sen2_2 = 16.365

    out_enx = 0.2
    out_eny = 17.32
    out_evx = 0.6
    out_evy = 16.835
    out_oddx = 1.3
    out_oddy = 17.35

    newLayout.addInstance(botLeftLayout,
                          offsetInMicrons=(x_offset,
                             y_offset),
                          mirror = mirror)
    x_offset += SIDE_WIDTH
    for i in range(numCols/2):
        ### Label output pins
        newLayout.addInstance(botMidLayout,
```

```
                            offsetInMicrons=(x_offset,
                               y_offset),
                            mirror=mirror)
        newLayout.addBox(layerNumber=69,
                         dataType=20,
                         offsetInMicrons=(x_offset +
                            out_enx,y_offset - out_eny),
                         width = .1,
                         height=.1,
                         center=True)
        newLayout.addText(text = "out_en[" + str(i) + "]",
                         layerNumber=69,
                         purposeNumber=16,
                         offsetInMicrons=(x_offset +
                            out_enx,y_offset - out_eny))

        newLayout.addBox(layerNumber=69,
                         dataType=20,
                         offsetInMicrons=(x_offset +
                            out_oddx,y_offset - out_oddy),
                         width = .1,
                         height=.1,
                         center=True)
        newLayout.addText(text = "out[" + str(2*i + 1) + "
           ]",
                         layerNumber=69,
                         purposeNumber=16,
                         offsetInMicrons=(x_offset +
                            out_oddx,y_offset - out_oddy))

        newLayout.addBox(layerNumber=67,
                         dataType=20,
                         offsetInMicrons=(x_offset +
                            out_evx,y_offset - out_evy),
                         width = .1,
                         height=.1,
                         center=True)
        newLayout.addText(text = "out[" + str(2*i) + "]",
                         layerNumber=67,
                         purposeNumber=16,
                         offsetInMicrons=(x_offset +
                            out_evx,y_offset - out_evy))

        x_offset += MID_WIDTH

    newLayout.addInstance(botRightLayout,
                          offsetInMicrons= (x_offset,
                             y_offset),
                          mirror = mirror)

    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset + WL_x,
```

```
                    y_offset - fnpass_y),
                  width = .1,
                  height=.1,
                  center=True)
newLayout.addText(text = "SSL[" + str(block) + "]",
                   layerNumber=67,
                   purposeNumber=16,
                   offsetInMicrons=(x_offset + WL_x,
                      y_offset - fnpass_y))

### Label power lines
newLayout.addBox(layerNumber=68,
                  dataType=20,
                  offsetInMicrons=(x_offset - m1_x,
                     y_offset - gnd1),
                  width = .1,
                  height=.1,
                  center=True)
newLayout.addText(text = "GND",
                   layerNumber=68,
                   purposeNumber=16,
                   offsetInMicrons=(x_offset - m1_x,
                      y_offset - gnd1))

newLayout.addBox(layerNumber=68,
                  dataType=20,
                  offsetInMicrons=(x_offset - m1_x,
                     y_offset - vddy),
                  width = .1,
                  height=.1,
                  center=True)
newLayout.addText(text = "VDD",
                   layerNumber=68,
                   purposeNumber=16,
                   offsetInMicrons=(x_offset - m1_x,
                      y_offset - vddy))

### Label sen control lines
newLayout.addBox(layerNumber=68,
                  dataType=20,
                  offsetInMicrons=(x_offset - m1_x,
                     y_offset - sen1_1),
                  width = .1,
                  height=.1,
                  center=True)
newLayout.addText(text = "sen1",
                   layerNumber=68,
                   purposeNumber=16,
                   offsetInMicrons=(x_offset - m1_x,
                      y_offset - sen1_1))
newLayout.addBox(layerNumber=68,
                  dataType=20,
                  offsetInMicrons=(x_offset - m1_x,
```

```
                        y_offset - sen2_1),
                    width = .1,
                    height=.1,
                    center=True)
    newLayout.addText(text = "sen2",
                     layerNumber=68,
                     purposeNumber=16,
                     offsetInMicrons=(x_offset - m1_x,
                        y_offset - sen2_1))

def placeBLRow(block):
    x_offset = 0
    global y_offset

    fnpass_y = 0.47

    newLayout.addInstance(BLLeftLayout,
                          offsetInMicrons=(x_offset,
                             y_offset))
    x_offset += SIDE_WIDTH
    for i in range(numCols/2):
        newLayout.addInstance(BLMidLayout,
                              offsetInMicrons=(x_offset,
                                 y_offset))
        x_offset += MID_WIDTH

    newLayout.addInstance(BLRightLayout,
                          offsetInMicrons= (x_offset,
                             y_offset))
    newLayout.addBox(layerNumber=67,
                    dataType=20,
                    offsetInMicrons=(x_offset + WL_x,
                       y_offset - fnpass_y),
                    width = .1,
                    height=.1,
                    center=True)
    newLayout.addText(text = "SSL[" + str(block-1) + "]",
                     layerNumber=67,
                     purposeNumber=16,
                     offsetInMicrons=(x_offset + WL_x,
                        y_offset - fnpass_y))
    y_offset-=MID_HEIGHT
    newLayout.addBox(layerNumber=67,
                    dataType=20,
                    offsetInMicrons=(x_offset + WL_x,
                       y_offset + fnpass_y),
                    width = .1,
                    height=.1,
                    center=True)
    newLayout.addText(text = "SSL[" + str(block) + "]",
                     layerNumber=67,
                     purposeNumber=16,
                     offsetInMicrons=(x_offset + WL_x,
```

```
                        y_offset + fnpass_y))

def placeSLRow(block):
    x_offset = 0
    global y_offset

    fnpass_y = 0.47

    newLayout.addInstance(SLLeftLayout,
                          offsetInMicrons=(x_offset,
                             y_offset))
    x_offset += SIDE_WIDTH
    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset - WL_x,
                        y_offset - fnpass_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "GSL[" + str(block-1) + "]",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset - WL_x,
                         y_offset - fnpass_y))
    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset - WL_x,
                        y_offset + fnpass_y - MID_HEIGHT),
                     width =.1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "GSL[" + str(block) + "]",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset - WL_x,
                         y_offset + fnpass_y - MID_HEIGHT)
                         )
    for i in range(numCols/2):
        newLayout.addInstance(SLMidLayout,
                              offsetInMicrons=(x_offset,
                                 y_offset))
        x_offset += MID_WIDTH

    newLayout.addInstance(SLRightLayout,
                          offsetInMicrons= (x_offset,
                             y_offset))

    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset - BL_BL_x
                        /2,y_offset - MID_HEIGHT/2),
                     width =.1,
                     height=.1,
```

```
                    center=True)
    newLayout.addText(text = "SL",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset - BL_BL_x
                         /2,y_offset - MID_HEIGHT/2))

    y_offset-=MID_HEIGHT



def placeCellRow(block,row,orientation):
    x_offset = 0
    global y_offset

    if orientation:
        mirror = None
        WL0_y = -0.215
        WL1_y = 0.215 - CELL_HEIGHT
    else:
        mirror  = "x"
        WL0_y = 0.215
        WL1_y = CELL_HEIGHT - 0.215
        y_offset-=CELL_HEIGHT
        row = blockSize/2 - row - 1

    newLayout.addInstance(cellLeftLayout,
                          offsetInMicrons=(x_offset,
                             y_offset),
                          mirror = mirror)
    x_offset += SIDE_WIDTH
    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset - WL_x,
                        y_offset + WL0_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "WL" + str(block) + "[" + str
       (2*row) + "]",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset - WL_x,
                         y_offset + WL0_y))
    for i in range(numCols/2):
        newLayout.addInstance(cellMidLayout,
                              offsetInMicrons=(x_offset,
                                 y_offset),
                              mirror=mirror)
        x_offset += MID_WIDTH

    newLayout.addInstance(cellRightLayout,
                          offsetInMicrons= (x_offset,
```

```
                                y_offset),
                            mirror = mirror)

    newLayout.addBox(layerNumber=67,
                     dataType=20,
                     offsetInMicrons=(x_offset + WL_x,
                        y_offset + WL1_y),
                     width = .1,
                     height=.1,
                     center=True)
    newLayout.addText(text = "WL" + str(block) + "[" + str
       (2*row+1) + "]",
                      layerNumber=67,
                      purposeNumber=16,
                      offsetInMicrons=(x_offset + WL_x,
                         y_offset + WL1_y))
    if orientation:
        y_offset-=CELL_HEIGHT


def placeCellBlock(block,orientation,size=(0,0)):
    global y_offset
    for row in range(size[1]/2):
        placeCellRow(block,row,orientation)

for block in range(numRows/blockSize):
    if (block % 2) == 0:
        if block: #if block is not 0
            placeBLRow(block)
        else:
            placeTopRow(block)
        placeCellBlock(block,True, size=(numCols,blockSize
           ))
    else:
        placeSLRow(block)
        placeCellBlock(block,False, size=(numCols,
           blockSize))
placeBotRow(numRows/blockSize - 1)

writer = gdsMill.Gds2writer(newLayout)
writer.writeToFile(output_path + out_name + ".gds")
```

APPENDIX C: MODIFIED TRI-STATE BUFFER SPICE FILE

```
* FILE: pass_trans.sp

.SUBCKT pass_trans in out ctr hv lv
M_1 out ctr_ in hv p20vhv1 W='60' L='2'
M_2 out ctr in lv n20vhv1 W='30' L='2'

M_3 ctr_ ctr hv hv p20vhv1 W='60' L='2'
M_4 ctr_ ctr lv lv n20vhv1 W='30' L='2'
.ENDS    $ pass_trans
```

APPENDIX D: MODIFIED SONOS MODEL

```
.global gnd
.param POS_PUMP = 6V
.param NEG_PUMP = -3V
.param MIN = -2V
.param MAX = 4V
.subckt FGMOS d g s b
Ginj float g CUR='((abs(v(d,gnd)-NEG_PUMP) - (v(d,gnd)-
   NEG_PUMP)) / (abs(v(d,gnd)-NEG_PUMP)+1e-9) * \
                    (abs(v(g,gnd)-POS_PUMP) + (v(g,gnd)-
                       POS_PUMP)) / (abs(v(g,gnd)-
                       POS_PUMP)+1e-9) * \
                    (abs(v(s,gnd)-NEG_PUMP) - (v(s,gnd)-
                       NEG_PUMP)) / (abs(v(s,gnd)-
                       NEG_PUMP)+1e-9) * \
                    (abs(v(float,gnd)-MIN)  + (v(float,
                       gnd)-MIN))  / (abs(v(float,gnd)-
                       MIN) +1e-9)) * 2e-3'


Gtun g float CUR='((abs(v(d,gnd)-POS_PUMP) + (v(d,gnd)-
   POS_PUMP)) / (abs(v(d,gnd)-POS_PUMP)+1e-9) * \
                    (abs(v(g,gnd)-NEG_PUMP) - (v(g,gnd)-
                       NEG_PUMP)) / (abs(v(g,gnd)-
                       NEG_PUMP)+1e-9) * \
                    (abs(v(s,gnd)-POS_PUMP) + (v(s,gnd)-
                       POS_PUMP)) / (abs(v(s,gnd)-
                       POS_PUMP)+1e-9) * \
                    (abs(v(float,gnd)-MAX)  - (v(float,
                       gnd)-MAX))  / (abs(v(float,gnd)-
                       MAX) +1e-9)) * 2e-3'
R1 float vr 1e14
C1 float g 1n ic=0V
Vinit vr g DC 0V
Mfg d float s b sonos_p w=0.35 l=0.15 m=1 r=2.8 a=0.063 p
   =1.14 mult=1
.probe i(Ginj)
.probe i(Gtun)
.ends FGMOS
```

VITA

Brandon T. Ong

Candidate for the Degree of

Master of Science

Thesis: NAND FLASH COMPILER USING THE SKYWATER 130NM PROCESS

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2022.

Completed the requirements for the Bachelor of Science in Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in 2020.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in 2020.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
June 2020 - May 2022