OpenROM: Design of an Open-Source Read-Only Memory Compiler for the OpenRAM Project

Sage Walker

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Engineering

in the
VLSI Design Automation Group
Baskin School of Engineering
University of California: Santa Cruz
February 2023

Abstract

This thesis documents the addition of a read-only memory (ROM) compiler, OpenROM, to the existing OpenRAM framework. Currently, OpenRAM provides a static random-access memory (SRAM) compiler; however, commercial memory compilers offer a wide range of memory types. The widespread use of read-only memory in embedded microcontrollers makes it an attractive candidate for integration into OpenRAM. Implementing a NAND-type metal-mask read-only memory bank included the design of a scalable ROM bitcell array and a dynamic NAND decoder. Development targeted the SkyWater 130nm process node while keeping the underlying compiler technology-independent. Unit tests based on an open-source DRC/LVS toolchain verified layout fabricability and software stability. Simulations using the Xyce analog simulator revealed operational frequencies of memories on SKY130 as high as 50MHz for 1kB ROM and as low as 600kHz for 8kB ROM. Investigation of worst-case read operations identified delay in the row decoder as a critical factor in the poor performance of larger memories.

Acknowledgements

First, I would like to acknowledge Prof. Matthew Guthaus for presenting me with the opportunity to work on this project. His introductory VLSI course fulfilled a dream I have had since I was in high school of learning how chips were designed.

I would like to thank Jesse Cirimelli-Low for her endless support on day-to-day technical issues. Her enthusiasm and support kept me going even when a bug seemed impossible to solve.

To Lucia Fauth I am grateful for several design reviews along the way and encouraging me to document my progress. I would also like to thank her for introducing me to the wider open-source hardware community, which has provided many valuable connections.

Travis Goodspeed graciously provided the magnified image of mask ROM included in this thesis. His documentation of delayering chips helped immensely when sanity-checking my own mask ROM design.

Last but certainly not least, I would like to thank my mother. She contributed edits, several boxes of cookies, and of course love. I wouldn't be here today without her.

Contents

List of Figures

List of Tables

Glossary

V_{dd} Voltage at Drain. Generally refers to the positive supply voltage.

V_{th} Threshhold Voltage. Minimum gate-to-source voltage required to form a conducting channel in a MOSFET.

Bitline The vertical wires which carry data from memory cells within an array to the edges of the array. Forms the columns of the bitcell array.

Diffusion An area where dopant atoms have been *diffused* into the silicon substrate. Generally used to refer to the p- or n-type source and drain of a MOSFET.

DRC Design Rule Checking. Verification tool used by used by engineers to verify a layout conforms to geometric constraints imposed by the manufacturer.

Fanout The number of inputs driven by the output of a given logic gate.

LVS Layout Versus Schematic. A process to verify that a designed layout matches the circuit described by a schematic.

Monotonic A function or signal is monotonic if it is strictly increasing or decreasing.

nMOS An n-type MOSFET.

Pitch The distance between two related elements in a layout.

pMOS A p-type MOSFET.

Polysilicon Polycrystalline silicon, or "poly." Material used to form the gate contact of a MOSFET.

SPICE Simulation Program with Integrated Circuit Emphasis. A general purpose analog circuit simulator. Used herein to refer to a family of circuit simulators sharing similar syntax.

Via A vertical connection between two horizontal layers on an integrated circuit.

Wordline Horizontal wires that activate a group of memory cells to be read from. Forms the rows of the bitcell array.

Acronyms

ASIC Application Specific Integrated Circuit.

CAM Content-Addressable Memory.

EDA Electronic Design Automation.

LSB Least Significant Bit.

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor.

MSB Most Significant Bit.

NAND Not-AND gate.

ROM Read-Only Memory.

SRAM Static Random-Access Memory.

VLSI Very Large Scale Integration.

1 Introduction

At the core of all modern integrated circuits (ICs) is the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET). Due to the ease of their fabrication and wide range of applications, they have become the most abundant device on earth. While cutting-edge fabrication techniques can produce transistors with gate widths as small as 3 nanometers, developing these process nodes has become increasingly expensive. Development of application-specific integrated circuits (ASICs) on these cutting-edge nodes comes with high costs for licensing proprietary electronic design automation (EDA) tools provided by semiconductor foundries. Restrictive licensing options for EDA tools and process design kits (PDKs) limits the ability of research groups to share their work and makes ASIC design inaccessible to start-ups and hobbyists. In order to alleviate this limitation, several fabs have recently open-sourced the PDKs for some of their process nodes. With the availability of these PDKs comes the need for a corresponding suite of open-source EDA tools. Within this open-source EDA ecosystem, the OpenRAM project aims to provide a flexible and robust open-source memory compiler.

2 Background

2.1 Memory Compilers

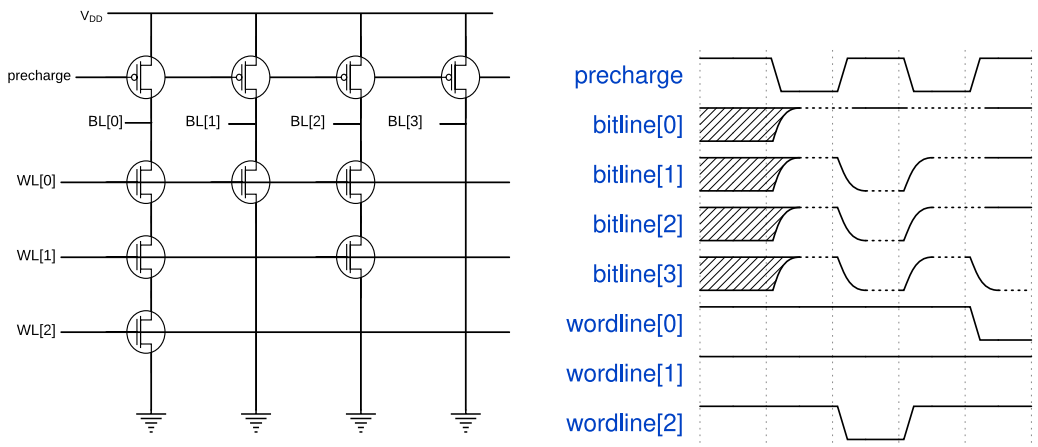
All microcontrollers require some form of computer memory. Designing computer memory, however, can be time-consuming when done manually, taking engineering resources away from developing the core elements of an integrated circuit. Memory compilers allow designers to automate this process, making it easy to generate memory based on user-defined specifications. Several leading providers of EDA tools provide their own memory compilers, including Synopsys [1], Dolphin Technologies[2], and Cadence Design Systems[3]. However, these tools all come with licensing fees and hide the internal design from end users.

In recent years, there has been increasing interest in open-source memory compilers, designed to be more accessible and customizable than traditional proprietary software. Developing these compilers can be challenging as the software for memory compilers is typically complex and requires expertise in electronic design and software development to create. The high barrier of entry has limited the companies that produce memory compiler software to large, established players in the EDA industry. Support from industry giant Google has helped drive development of open-source EDA tooling, including memory compilers. OpenRAM is an open-source memory compiler actively developed and maintained by researchers at the VLSI Design and Automation Group at the University of California, Santa Cruz. Currently it supports single-port and dual-port SRAM in a range of process technologies, including SkyWater 130 nanometers and FreePDK 45 nanometers [4]. While it includes features like power consumption characterization and configurable design rules, it lacks the diversity of memory types that commercial tools offer. Some commercial memory compilers provide options for SRAM, ROM, CAM, and other specialized types of memories. The addition of the first non-SRAM memory to the OpenRAM is an exciting test of the extensibility of the open-source framework.

2.2 ROM

Read-only memory has historically been a key component of computer systems as it provides a reliable way to store data that is essential to the correct operation of the system. As non-volatile memory, ROM retains its data even without power, and, as the name suggests, ROM is typically incapable of being altered after its initial programming. In early computers, read-only memory was the default for distributing and storing operating systems and executable code. While most software today is stored on re-writable storage, modern computers still use ROM where un-editable data is a useful security feature. Common use cases include storing device initialization firmware or encryption keys used by the manufacturer to verify hardware authenticity. Almost every microcontroller manufactured today contains ROM to store the first-stage bootloader code that defines start-up behavior.

Read-only memory can be implemented in several ways depending on the use case and desired characteristics. While all ROMs are primarily optimized for read operations, some allow for more flexibility in initial programming or even allow for reprogramming. Mask ROM (MROM) is one of the most straightforward implementations of read-only memory. The physical layout of the memory cells during the fabrication process defines the programming. Its high density makes mask ROM one of the



(a) Schematic of a NAND ROM. (b) Timing diagram of a NAND ROM.

Figure 1: Basic operation of NAND ROM

cheapest forms of computer memory. However, any error in the program data means manufacturing a whole new chip. Programmable ROM (PROM) can be programmed once after production, typically with special programming equipment. In contrast, erasable programmable ROM (EPROM) can have its data erased and reprogrammed a limited number of times. Older EPROMs used direct die exposure to high-energy UV light to erase the memory. However, this has largely been superseded by electrically erasable programmable ROM (EEPROM), which requires only electrical signals to erase and rewrite the memory. Many modern devices take advantage of EEPROM's high density and flexibility to allow updates to firmware that would be traditionally stored in non-erasable ROM. More modern EEPROM designs are often referred to as "flash" memory as it offers much higher read/write speeds compared to other EEPROM. With improvements to the number of times EEPROM can be re-written, it has become a widespread alternative to high-capacity non-volatile memory.

While the drawbacks mentioned previously limit the applications of mask ROM, the simplicity of its implementation makes it a clear candidate for automation within the OpenRAM project. Designs of mask ROM in CMOS processes use NAND or NOR architecture. NAND ROM has the highest density of any non-volatile memory as each bit requires only a single transistor, while NOR ROM has faster access and read times. This thesis implements a NAND-style ROM as the primary proof of concept for OpenROM.

2.3 Operational Theory

The read operation of NAND ROM consists of a precharge stage and a read stage (also called the evaluation stage). An example schematic of a NAND ROM and its behavior can be found in Fig. 1. During the precharge stage, the pMOS cells at the top of the memory activate and charge the bitlines to ensure a strong signal can be read. All wordlines are held at one to ensure all internal nodes of the bitline are charged. Precharge is complete when the furthest cell from the precharge array reaches $V_{dd} - V_{th}$. If the precharge pMOS deactivate before this condition is satisfied, the charge will distribute throughout the internal nodes to a voltage below V_{th}, leading to errors in the read operation. This effect is known as "charge sharing" since the uncharged section of the bitline acts as a parasitic capacitance, causing charge to flow into it from the rest of the bitline.

During the evaluation stage, lowering a single wordline to logic zero indicates a read from that wordline; all inactive wordlines must remain at logic one during this time. This pattern is often described as a "one-cold" encoding scheme. The precharge pMOS cells are then deactivated. If the intersection of the bitline and wordline contains no transistor, then any remaining charge in the bitline will be shorted to ground, reading a zero. However, placing an nMOS in the bitcell will prevent the bitline from being pulled down, reading as a one. Using the placement of nMOS within the array, data can be stored in the physical layout of the memory.

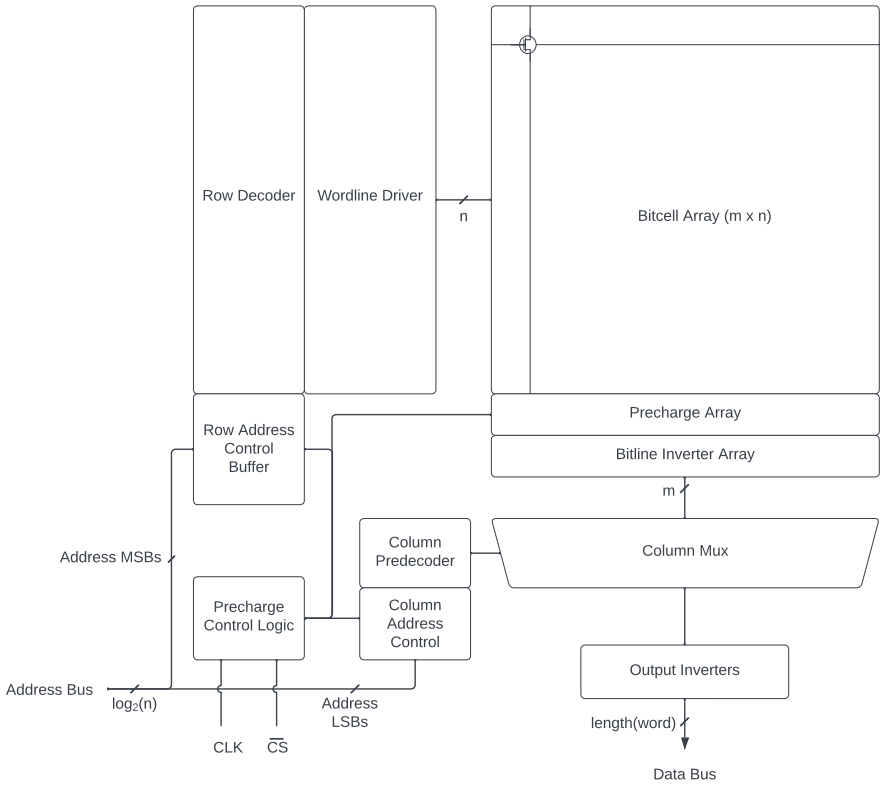


Figure 2: The ROM bank consists of the bitcell array, row and column decoder circuits, and control logic.

3 Design

The architecture of ROM centers around an array of memory cells, with auxiliary circuits surrounding it (see Fig. 2). The structure can be broken down into circuits for the bitcell array, row and column decoders, column mux, and control logic.

3.1 Bitcell

The bitcell of NAND ROM uses the presence or absence of a single nMOS transistor at the intersection of bitlines and wordlines to encode a zero or a one. The presence or absence of a transistor can be achieved by placing transistors in some bitcells and not others. This approach requires changing features across multiple process layers, increasing the cost of manufacturing a new ROM. To limit the data encoding to a single layer, it is most effective to place a transistor in every bitcell, regardless of its value. The transistor gate is then shorted for zero cells, effectively removing the transistor. This limits the programming of the array to a single metal mask layer and is appropriately termed "metal mask" ROM. Due to each transistor requiring vias to the metal layer, metal mask ROM is limited by the via-to-gate pitch. Some processes achieve a similar effect by adding a threshold-lowering implant to the bitcell transistor gate. This technique is often called diffusion ROM and avoids using metal vias on every transistor, allowing it to have an overall tighter pitch than metal mask ROM.

This work chose to use a metal mask bitcell due to the simplicity of its implementation. The flexible nature of OpenRAM's hierarchical structure makes diffusion ROM a possible future feature. Despite the loss of area due to via-to-gate spacing, the bitcell presented in this thesis is still significantly smaller than OpenRAM's SRAM cells. The smallest SRAM cell in OpenRAM uses six transistors, compared to the single transistor of NAND ROM.

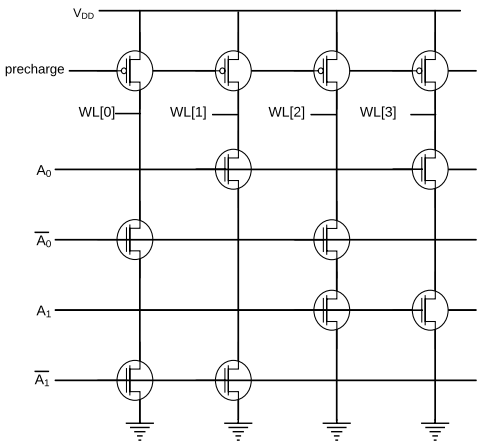


Figure 3: Placement of bits in a NAND ROM to create a NAND decoder.

3.2 Bitcell Array

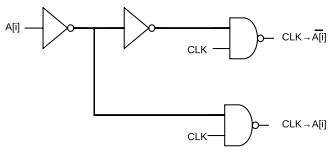
This thesis aimed to balance memory density and read delay when designing a bitcell array. While the nMOS-to-nMOS pitch of a technology is the limiting factor on the maximum density of NAND ROM, attempting to minimize delay leads to further reductions in density. NAND ROM delay is constrained by three factors: wordline delay, bitcell delay, and precharge delay. Wordline delay is the time it takes for the read signal from the row decoder to propagate through the entire wordline. This delay is impacted by the capacitance of bitcell gates connected to the wordline and the parasitic capacitance of the wordline wire. For long wordlines, the wire's capacitance dominates. An initial design used abutting polysilicon bitcell gates as wordlines. However, polysilicon's resistance is approximately three orders of magnitude larger than metal layers, making this design unable to scale up to larger memories. An additional wordline routed on a metal layer accounts for this issue. Regularly spaced "poly taps" reconnecting the polysilicon and metal wordlines make sure the signal quickly propagates across the entire array. In order to comply with poly-to-diffusion design rules, an offset must be added between bitcells with a poly tap between them. However, this is a small trade-off in density compared to the reduction in wordline delay.

The internal capacitance and resistance of bitcells determine bitcell and precharge delay. *Precharge delay* is defined as the amount of time it takes to charge the end of the bitline to $V_{dd} - V_{th}$. Charging the entire bitline is affected by the bitcell capacitance and resistance, metal bitline resistance, and the maximum precharge current. The bitline will charge only to $V_{dd} - V_{th}$ due to the fundamental behavior of MOSFETs. There will also be an additional voltage drop based on the diffusion and channel resistance of the nMOS transistors. *Bitcell delay* is the inverse of precharge delay, defined as the time to discharge the bitline during a read. Increasing the width of nMOS used in the bitcell acts to decrease the resistance of the gate, leading to decreased bitcell delay. However, it also increases the drain/source capacitance, increasing the precharge delay. The bitcell in this design uses only minimum width nMOS. Density is valued over fast read speeds since it is common for microcontrollers to read ROM into a faster volatile memory bank like SRAM before making use of the data. Exploring the trade-offs of larger bitcell sizes is a potential topic of future research.

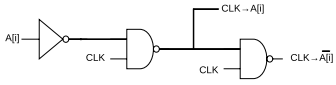
The array includes precharge pMOS transistors at the top of each bitline and a row of nMOS "foot" transistors added to the end of each bitline to prevent it from shorting to ground during the precharge cycle. These nMOS help decrease the impact of charge sharing on bitlines during read cycles by allowing the entire length of the bitlines to precharge.

3.3 Row Decoder

The row decoder converts the most significant bits (MSB) of the incoming read address into activation of a single wordline. Row decoders are present in nearly all computer memory, and as such, there are several strategies for implementing them[5]. At its core, a decoder is a de-multiplexer that converts an N-bit input signal into 2^N output bits using one-hot/one-cold encoding. A circuit with 2^N AND



(a) Parallel IMPLY gates



(b) Series IMPLY gates

Figure 4: The series realization requires one less inverter and is more easily implemented in layout in a way that can match the pitch of the decoder inputs.

Address	Precharge	Decoder input
0	0	1
1	0	1
0	1	0
1	1	1

Table 1: Truth values of decoder address logic.

gates, each with N inputs can accomplish this functionality. In practice, this is not space efficient and introduces a large amount of delay. OpenRAM mitigates delay using a hierarchical decoder in its SRAM. Adding pre-decode stages reduces delay; however, multi-stage decoding comes at the cost of a more complex layout and increased pitch between wordlines. This increased pitch is unfavorable for ROM as it significantly impacts the size conservation due to the small bitcell.

Dynamic decoders offer a convenient alternative. Rather than use traditional static CMOS logic, dynamic decoders use dynamic logic gates that efficiently implement many-input functions. While a static CMOS NAND gate takes 2N transistors to implement an N inputs, a dynamic NAND takes N+1 transistors. A dynamic decoder is also appealing as it can be designed using the same structure as the bitcell array as seen in Fig. 3. Rather than programming the array with arbitrary data, each bitline contains the encoding of the wordline index in binary. Inputs to the decoder are the address bits and their complements. The output is 2^N wordlines with a one-cold encoding of the active wordline.

In order to mitigate charge-sharing effects during read cycles, all decoder inputs must remain at one during the precharge cycle and must be monotonic during the read. The address inputs to the decoder must then behave according to the truth values found in Table 1.

This behavior corresponds to the logical conditional operator, known in hardware as the IMPLY gate. Our decoder requires both the address and its complement as inputs and therefore needs two IMPLY gates for each bit of the read address.

Additionally, incoming address signals must go through at least one stage of CMOS buffering before they are used in logic to guarantee signal integrity. Considering these facts, implementing two IMPLY gates in series removes the need for an additional inverter if the gates were in parallel (see Fig. 4

In order to conserve space and maintain the tight pitch of the decoder inputs, the control circuits were implemented with non-standard cell gates (see Fig. 5). The same requirement holds for the decoder outputs, which must remain at one during the precharge cycle and monotonic during the read cycle. The dynamic decoder satisfies this requirement without any further modification, another one of its advantages.

The outputs of the row decoder are passed through a static CMOS buffer to compensate for the large capacitance of the wordlines. Sizing the wordline buffer to balance speed and drive strength is vital for proper memory function. As such, initial sizing was estimated based on simple logical effort modeling. The design described in this thesis was targeted for the SKY130 process node and assumed $\beta = 3$.

The input capacitance of a bitcell is:

$$C_{bitcell} = C_{nmos} = \frac{C_{inv}}{\beta + 1} = \frac{1}{4} C_{inv}$$

Therefore the capacitance of a single wordline with width m, is: $\frac{1}{4} m C_{inv}$ (ignoring wire capacitance). To roughly account for wire parasitics, wordline effort was estimated to be $\frac{1}{2} m$.

The existing decoder buffer module (`pbuf_dec.py`) in OpenRAM attempts to have at least a stage effort of 4 between its two stages. In keeping with this number, the final inverter in the wordline buffer

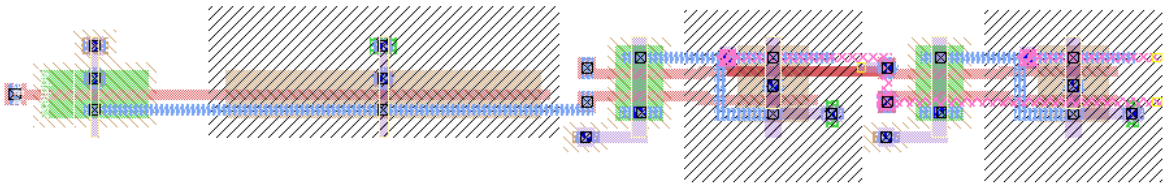


Figure 5: Layout of small-pitch IMPLY gates. Features custom thin NAND gate layout [6]

was sized:

$$\frac{(\frac{1}{2}m)}{4} = \frac{1}{8}m.$$

3.4 Column Decoder

While the row decoder handles the MSBs of the incoming address, the column decoder must use the LSBs to select a single word from the multiple words stored in each row. Unlike the row decoder, the column decoder is not a true decoder but an n-output multiplexer, where n is the word size. Inputs to the column decoder are the bitlines of the bitcell array, and the output is a single data word. The LSBs of the read address effectively serve as select lines for the mux. The most straightforward design solution is a static CMOS multiplexer implemented using AND, OR, and NOT gates. While this is an effective approach for small-scale 2:1 or 4:1 multiplexing, it fails to scale efficiently with the large number of bitlines in computer memory. An alternative design uses nMOS pass transistors, controlled by the select lines and their complements, to multiplex the signal. While very fast due to the lack of pMOS devices in the data path, this approach leads to signal degradation as the words per row increase due to voltage drop across the gate of the nMOS devices.

A practical compromise between pass transistors and static CMOS multiplexers is the combination of a column pre-decoder and a pass transistor mux. The column predecoder operates as a true decoder, taking the LSBs and outputting a one-hot encoded select signal. The select signal activates a single nMOS pass transistor per bitline, minimizing the impact of nMOS in the data path. The single pass transistor approach is particularly attractive in NAND ROM due to the tight pitch of the bitlines relative to other memories. Having only a single nMOS required for selecting each bitline means that the pitch of the pass transistor array can match the tight pitch of our bitcell. Reusing the row decoder architecture reduces the complexity of the design and effectively satisfies our requirements for speed and area.

To simplify the layout of the column mux, the select lines run horizontally, branching off at regular intervals to their respective pass nMOS through vertical wires. To avoid complicated routing from the pass transistors to the outputs of the column mux, bits must be grouped by their position in the stored word. For example, in an array with W words per row, the first bit of each word is stored in columns 0 to $W-1$, the second bit in columns W to $2W-1$, etc. Due to this, stored data cannot be simply broken into rows and programmed into the array. Data first must be "scrambled" to properly fit the configuration of the column mux.

It is also important to note that an array of inverters was placed between the output of the bitcell array and the column decoder. Since each bitline is effectively the output of a dynamic NAND gate, its output signal is limited to the capacitive charge stored in the bitline. Without buffering from a static CMOS gate, this charge would dissipate through the column mux and fail to read the correct data. Instead of using a non-inverting buffer in every bitline, the data signal is inverted after the column mux to save space.

3.5 Control Logic

Without the need to handle write operations, the control signals for ROM consist of an active-low chip select and clock. The control logic is primarily responsible for disabling the precharge signal when chip select is disabled, as well as buffering the clock signal for distribution to various subcircuits in the memory bank. The control logic contains two buffers in series: the first drives the precharge signal for the row decoder, while the second drives the array and column decoder precharge.

Variable	Equation
Data Size (bytes)	`len(bin_data)/8`
# of Words	`data_size / word_size`
Bytes per Row	`sqrt(num_words)`
Words per Row	`ceil(bytes_per_row / (2 * word_size))`
# of Columns	`words_per_row * word_size * 8`
# of Rows	`num_words / words_per_row`

Table 2: Important array parameters and how OpenROM calculates them.

4 Software Implementation

Implementing ROM compiler functionality took advantage of the existing low-level python framework for VLSI provided by OpenRAM [7]. OpenRAM provides a robust API for technology-agnostic layout and SPICE schematic generation. The object-oriented structure of OpenRAM made it easy to extend its abstract base classes to implement a ROM bitcell array. The parameterized gate (`pgate.py`) classes also made implementing simple logic functions straightforward. Layout constants such as device spacing or sizing were parameterized where possible to use the built-in design rule constants. This dynamic sizing allows OpenROM to remain technology-agnostic, although only one node was targeted for testing (as detailed in section 5).

The ROM-oriented classes were optimized to reuse as much existing code from OpenRAM while adding ROM-specific features. Each submodule of the ROM was implemented in its own class, following the hierarchical design approach of the larger library. Each class inherits from the `design.py` class, which keeps track of the module's SPICE netlist and layout. The design class further inherits from the lower level `hierarchy_spice.py` and `hierarchy_layout.py`. The spice class provides functions for manipulating the SPICE netlist, such as top-level pins and subcircuits the module contains. The layout class provides functions for placing components within the module's physical layout, routing wires, and placing other simple shapes or labels.

New modules are created by calling the `factory.create()` method, which accepts parameters about the module to be generated, such as name, type, and any values required for its initialization. For example, calling:

```
factory.create(module_type="rom_base_array", cols=10, rows=10, bitmap=self.data)
```

Might create a 10x10 ROM bitcell array using the bitmap passed in self.data. During initialization, the newly created module's constructor is called with the parameters passed to `factory.create()`. Modules represent the abstract features of a component with the given parameters; they do not yet have a placement in layout or connections to other components in the top-level module. An "instance" is created from the module that stores information about the netlist connectivity and placement within the layout. The class for a component generally follows the same format: A constructor initializes relevant variables like the name, routing layer, or component size. Then a call to `self.create_netlist()` instantiates any submodules used, defines the netlist connections for their respective instances, and sets the top-level pinout for the module. `self.create_layout()` handles layout generation, including placing the instances, routing wires, and labeling relevant layout pins.

4.1 Top-Level Class

The top-level class for ROM creation, `rom.py`, provides a constructor that accepts configuration parameters such as a hexadecimal data file to program memory, the word size of the data, endianness, and user-defined features of the layout, such as spacing between polysilicon taps. Data is read from the input file and extracted into a 2D list of binary values. During the input processing the data is properly scrambled as described in section 3.4. Other parameters of the array are dynamically generated based on the size of data words and the size of the input data (Table 2). The top level class is made accessible through the `rom_compiler.py` script. It provides a command-line interface for ROM generation with options for specifying the destination of output files and user-created configuration files.

4.2 Module Classes

Every module of the ROM bank has a respective class consisting of subcomponents programmatically generated to parameterized sizes. At the lowest level lie base cell modules such as the bitcell (`rom_base_cell.py`), precharge cell (`rom_precharge_cell.py`), column mux (`rom_column_mux.py`), and address control logic (`rom_address_control_buf.py`).

These cells are tiled by their respective "array" classes (`rom_base_array.py`, `rom_precharge_array.py`, `rom_column_mux_array.py`). One of the advantages of the object-oriented structure of OpenRAM is that base cell and array classes can easily be reused or inherited from other classes. Since the ROM bitcell is simply an nMOS transistor, changing the transistor type to a pMOS allows the precharge module to match the pitch of the bitcell without any additional alignment.

While the bitcell only contains a single transistor, the class for the bitcell also handles placing the transistor within the cell so that it can be tiled correctly by the array class. By carefully placing the nMOS and setting the dimensions of the bitcell, the cell can tile perfectly just by offsetting each cell by its row and column index in the array. This abstraction allows for future changes to the size of the bitcell nMOS without needing to change the array class.

The ROM bitcell array class is designed to be flexible for multiple use cases and inherits from the abstract `bitcell_base_array.py` class used for other bitcell arrays in OpenRAM. It accepts parameters for the programming bitmap, layers on which to run wordlines and bitlines, and spacing and orientation of well and polysilicon taps. These customization parameters allow for reusing the bitcell array as a data storage array and a hardware-programmable NAND array for use in the decoders. The decoder class needs only to generate a bitmap encoding the correct logic behavior and program a ROM array module with it.

5 Tests and Simulation

All testing targeted the SkyWater Open Source PDK, an open source version of SkyWater Technology Foundry's 130nm technology node (SKY130)[8]. Developed in collaboration with Google, SKY130 aims to provide a comprehensive free PDK for use by researchers and educators. While the 130nm node was cutting edge around 2001-2002, it is still a viable process for the development of microcontrollers and other low-power embedded devices.

OpenRAM can generate layouts for several open-source PDKs; however, the scope of this thesis is limited to testing on SKY130.

5.1 Unit Tests

Unit tests for design rule checking (DRC) and layout versus schematic (LVS) were implemented using the OpenRAM unit test framework in python. Magic VLSI [9] was used for DRC and netlist extraction, and Netgen [10], [11] was used for LVS comparisons.

Each active module received a corresponding unit test to verify that the layout meets the basic standards for functionality. These basic tests check if the module cannot be correctly instantiated or if DRC/LVS fails. While these tests are sufficient for a first line of defense against simple runtime, layout, or design rule errors, these tests fail to catch functional errors. A common issue that was encountered during the development of OpenROM is the case where the LVS test passes, but the netlist does not implement the intended function. LVS can also pass while ignoring pin label mismatches by assuming symmetries between functionally different pins. In both cases, tests for higher-level modules sometimes catch the error, but in the worst case, the error does not surface until simulation.

The top-level module has multiple unit tests to verify correct scaling based on array size. A "small" bank test consisting of only 64 bytes of data was used primarily during the development of the top-level module. While this is not a realistic size for data storage, testing with a small bank allowed for faster iteration, as larger arrays take much longer to generate. Tests with 1kB, 2kB, 4kB, and 8kB data verified that the array and peripheral circuits scaled properly. The 1kB and 2kB tests used a word size of 8 bits (1B), and the 4kB and 8kB tests used a 16-bit (2B) word size.

5.2 Simulation Setup

Characterization and functional testing of the completed ROM banks were done with analog circuit simulation. The Xyce ([12]) parallel electronic simulator was used to run simulations. Xyce was chosen due to its capability to effectively utilize available HPC resources and its compatibility with standard SPICE stimulus files. Simulations were run without extracted wire parasitics since the goal of testing was primarily to verify the hardware's functionality rather than the simulation's accuracy. Further research will require more comprehensive simulations accounting for these extracted parameters.

Simulations used netlists generated by 1kB, 2kB, 4kB, and 8kB unit tests and transistor models from the SKY130 ngspice library. Functional testing was the primary intent, so nominal environmental parameters were assumed. All tests used a typical-typical process corner at a temperature of 27° C and a supply voltage of 1.8V. Characterization used the same environmental parameters along with the worst-case data layout.

5.3 Functional Tests

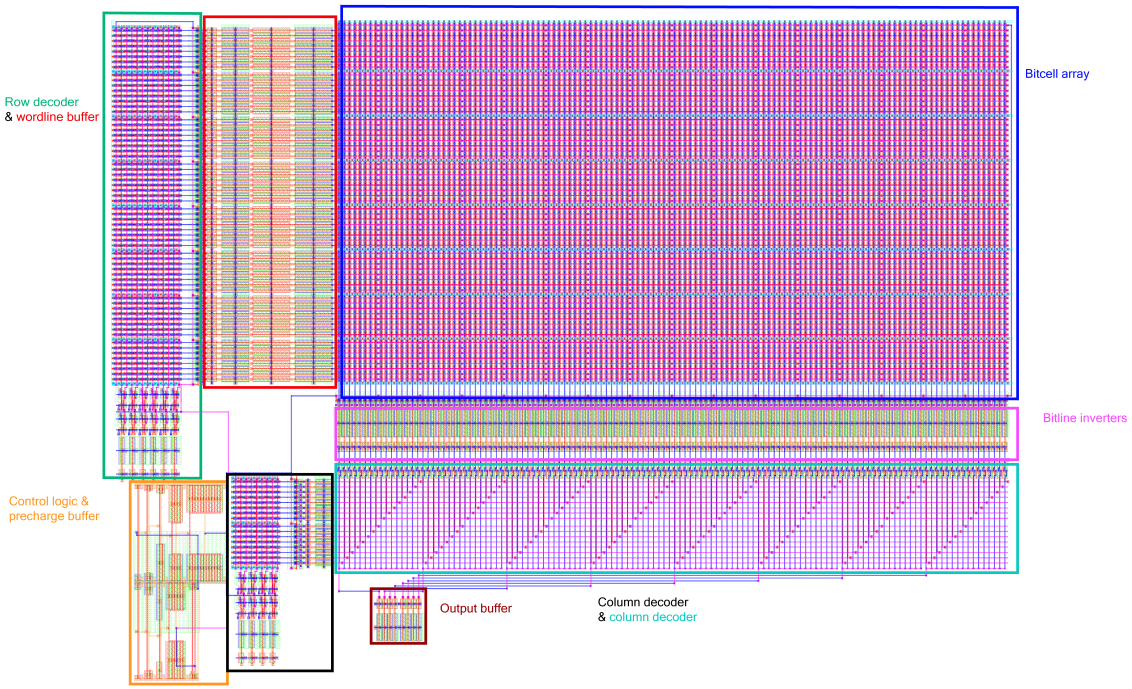
The functionality of a ROM was confirmed by initializing it with random data, then reading a specific word and verifying the correctness of the data output. The researcher wrote all SPICE stimuli files by hand. A test passed when visual inspection of the output waveform agreed with the data at the given address. A pulsed voltage source provided the precharge signal. Tests gave a sufficiently long delay before reading to guarantee enough time to precharge the entire array. This delay was estimated based on data sheets of worst-case read times from consumer MROM chips from the 1990s to the early 2000s. Initial tests set this delay to 100ns, as many commercial MROM chips from the period had maximum read delays between 100-200ns [13], [14]. This timing proved suitable for most of the test cases.

5.4 Characterization Tests

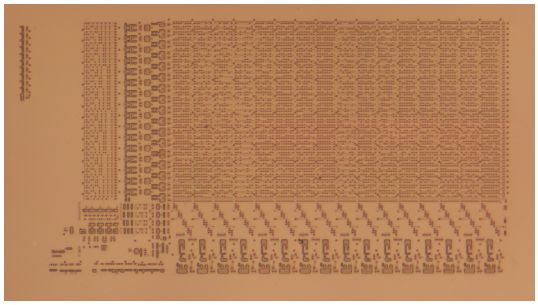
Characterization tests were designed to understand the worst-case read and precharge delays in the memory under test. The speed at which computer memory reads data is usually independent of stored data. However, in MROM, the speed of a read operation depends on the number of programmed bitcells in a bitline. Reading a bit from the upper right corner of a bitline containing nearly all bitcells programmed to a one is the worst-case. This case maximizes the delay of the bitline, as a bitcell containing a one has higher resistance than the metal wire in a zero cell. For similar reasons, this configuration maximizes the precharge delay as well. Finally, reading from the upper-right corner maximizes the wordline delay since the signal must propagate across the entire array. However, since the extracted netlists were without wire parasitics, these tests only consider the delay due to gate capacitance on the wordline.

Reading data in a specific order also impacts the read delay. When a one is read from a bitline immediately following the read of a zero, the bitline is entirely discharged and must wait the maximum amount of precharge time before it will read a stable one. The inverse case, that a bitline reading a zero must discharge entirely after reading a one, is the worst-case condition for a zero read. To test both cases, the characterization tests read alternating zero and one bits from the upper right corner of an array populated otherwise entirely by ones. Test accomplished this by toggling the LSB of the row address to switch between the last two wordlines in the array. Tests started with a clock of $T = 100ns$ and progressed if reads produced the correct data. Parameters tested for included zero access time(t_{za}), one access time (t_{oa}), row decoder delay (t_{rd}), and maximum viable precharge clock frequency (f_{max} or $\frac{1}{T_{min}}$). All test clocks had a duty cycle of 50%.

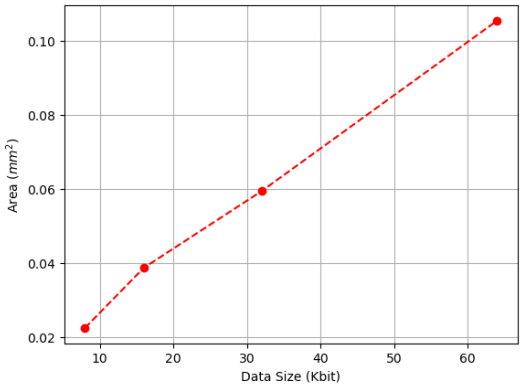
Notably, most consumer MROM chips are asynchronous, meaning that no precharge signal is required. The user needs only to change the read address and wait a given amount of time for the output to be valid. A clocked precharge signal was better suited for these tests since this thesis focused on exploring the underlying timing characteristics of the memory. A clocked precharge signal is not inherent to this memory architecture, so adding asynchronous operation is possible for future development.



(a) A 1kB MROM bank generated by an OpenROM unit test.

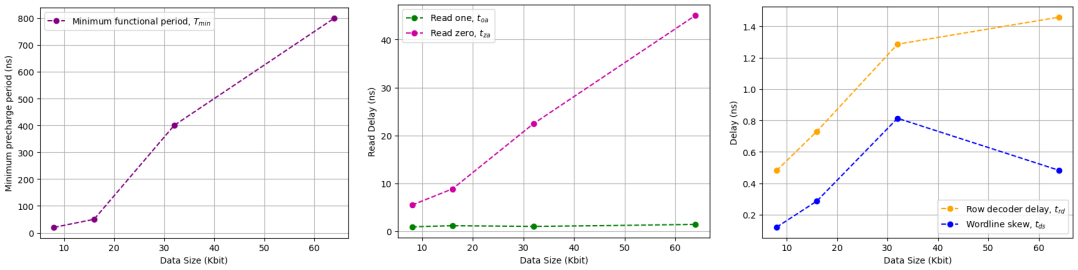


(b) TI MSP430F149 MROM courtesy of Travis Goodspeed.



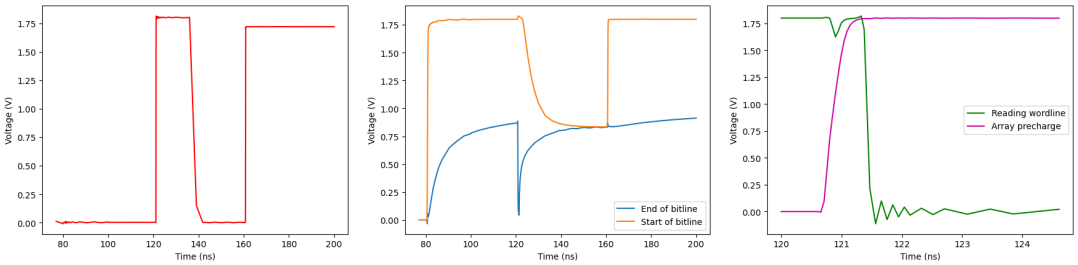
(c) Area of ROM banks generated in SKY130.

Figure 6: Results of layout generation. Includes layout of an MROM found on a 130nm consumer microcontroller for comparison. Note the similarity of features in the row and column decoder. The MSP430F149 ROM also has a slightly more compact layout in the lower left corner.



(a) Minimum precharge period (b) Output delay (c) Decoder delay

Figure 7: Measured delay results.



(a) Data output signal. (b) Start and end of the bitline (c) Wordline being read

Figure 8: Relevant signals from a "decode skew" read failure. Read begins at $t = 120ns$, ends at $t = 160ns$. The output data in (a) reads a one at first before falling back to zero about halfway through the read. The charge sharing effect can be seen in (b), and (c) shows the delay between the rising edge of the precharge signal and the falling edge of the wordline signal.

6 Results

6.1 Layout

Layouts generated by unit tests scaled unremarkably as OpenROM does not implement multi-bank support. Fig. 6a shows a typical ROM generated for SKY130. The data array takes up the majority of the layout, with the row decoder on the left being the second largest feature. Area increased approximately linearly with size as shown in Fig. 6c.

A comparison between the generated layout and images of MROM from de-capped 130nm processors reveals possible future improvements. (Fig. 6b). The obvious area to save space in the existing layout is the lower left corner. Currently, the space formed by the row and column decoder in the bottom left contains the control logic and buffers to drive the precharge signal around the rest of the layout. Clever placement of the control logic gates could minimize the remaining. While this is a possible future improvement, it is a low priority since MROM is already significantly smaller than most other types of memory and since bitcell array area dominates as data size increases.

6.2 Simulations and Performance

Preliminary functional tests of the 64-byte array revealed an issue with the design that was quickly corrected. The capacitive charge of the bitline was insufficient to drive a signal through the column mux and led to significant charge sharing between the bitline and output drivers, leading to failed reads. Placing a stage of inverters between the end of the bitlines and the column mux resolved this, acting as a buffer and providing a stronger drive through the mux.

Testing of the 1kB, 2kB, and 4kB array identified a more complex issue that severely degraded the performance of the larger memories. In the 1kB array, functional testing with a precharge clock of $T_{pre} = 80ns$ failed, as shown in Fig. 8. While the output signal shown in Fig. 8a should remain at one for the duration of the read cycle, instead it drops to zero about halfway through. Investigation of the voltage on internal nodes of the bitline showed that the end of the bitline had reached a suitable

voltage for a read operation (Fig. 8b). However, it discharged significantly at the rising clock edge, suggesting that the terminating nMOS activated before the wordline signal could deactivate the row. This timing difference resulted in enough loss of charge in the bitline to corrupt the read. Inspecting the wordline signal (Fig. 8c) compared to the buffered precharge clock showed that the precharge signal activated the nMOS before the row decoder activated the wordline.

Varying parameters in the design, such as the size of the wordline buffers and the size of the precharge signal buffer, showed that proper function was primarily dependent on the timing delay between the wordline and precharge signals, the "decoder skew" (t_{ds}). The delay in the wordline signal is primarily due to the delay of the row decoder, t_{rd}. When this delay becomes large, it results in the failure mode described above. Two minor changes in the design partially resolved this. First, two signals with slightly varying delays were used rather than having the entire memory bank use the same buffered precharge signal. The first is the decoder precharge signal which has a smaller delay by only fanning out to the row decoder. This decoder precharge is then routed through a larger, secondary buffer to drive the high fanout of the bitcell array precharge. This change introduces a delay between when the decoder activates and when the array reads, essentially allowing the decoder to get a head start before the array begins reading. The second design change was correcting the sizing of the wordline buffers to reduce the delay through them. Initial designs set the size of the wordline buffer to the number of bitcells in a wordline. However, properly sizing as described in section 3.3 helped reduce the total decoder delay.

While these changes successfully reduced t_{ds}, they do not solve the underlying problem. The row decoder delay is proportional to the row address size squared (or approximately $(\log_2(data\_size))^2$). If not adequately compensated for, this quickly reaches a considerable delay, enough to interfere with the operation of the memory. In the 1kB and 2kB designs, $t_{rd}$was small enough to be compensated for by the changes described and by increasing T_{pre} to offset the charge leakage. However, T_{pre} had to be significantly increased in the larger arrays to get the array functioning.

Another realization from simulation is that the performance of zero reads is notably slower than that of ones. As shown in Fig. 7b, the output delay of a zero is longer in all tests. This delay is due primarily to the bitline delay, t_{cell}, as longer bitlines take longer to discharge entirely. This delay is proportional to the square of bitline length [5], so potential future improvements include a better algorithm to prioritize lower-length bitlines over longer wordlines.

7 Conclusions and Future Work

This thesis aimed to design a proof of concept for an open-source ROM compiler. OpenROM successfully demonstrates a memory compiler integrated into the existing OpenRAM library capable of producing functioning mask ROM. The flexible and modular nature of the system allows for easy modification and extension to resolve some of the roadblocks mentioned in section 6.

Possible future work includes a more advanced precharge timing logic that allows asynchronous memory operation. This improvement would require solving the decoder timing problem and extending the existing control logic. Since NAND ROM operates by many of the same principles as dynamic logic, looking at strategies used to mitigate dynamic charge leakage could prove helpful for future work.

Introducing a multi-bank-based generation mode would allow for larger memory sizes while preventing read times from growing out of control. Many commercial MROM chips offer a "paged read" mode in which they read several addresses in quick succession, a feature only possible with multi-bank memory[13]. Further integration into the existing test framework for OpenRAM is also needed. The researcher conducted all timing characterization tests manually, while SRAM characterization is done automatically in OpenRAM. Modifying the existing simulation API would require minimal changes to support ROM read characterization. Power consumption characteristics are also missing, a standard in characterizing other memories in OpenRAM.

In the long term, this work could serve as the basis for adding other types of memory to OpenRAM. While NAND MROM is limited in its use cases, many modern devices feature NAND flash memory, which shares a similar architecture. However, instead of bitcells encoding data in the presence/absence of a transistor, flash memories program data into a floating-gate transistor. Diffusion MROM and fuse-based PROM designs could also be explored.

In the upcoming year, the designs shown in this thesis will have the opportunity for fabrication as part of the SkyWater/Efabless multi-project-wafer program. This presents an exciting opportunity

to evaluate the impact of relevant parameters described in this work in a real-world environment. Physical verification of the design paves the way for integration of OpenROM memories into future open-source ICs.

All source code will soon be available at:

https://github.com/VLSIDA/OpenRAM

And all other files used for simulation are available for review at:

https://github.com/polymerizedsage/OpenROM_resources

References

[1] Synopsys, *Synopsys memory compilers*, Last accessed 26 February 2023, 2022. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=dwc_sram_memory_compilers.

[2] Dolphin Technology, *Memory products*, Last accessed 26 February 2023, 2021. [Online]. Available: https://www.dolphin-ic.com/memory-products.html.

[3] Cadence Design Systems, *Cadence announces Legato memory solution*, Last accessed 26 February 2023, Sep. 2017. [Online]. Available: https://www.cadence.com/en_US/home/company/newsroom/press-releases/pr/2017/cadence-announces-legato-memory-solution--industrys-first-integr.html.

[4] UC Santa Cruz VLSI Design and Automation Research Lab, *OpenRAM*, Last accessed 26 February 2023, 2023. [Online]. Available: https://github.com/VLSIDA/OpenRAM.

[5] K. Abbas, *Handbook of Digital CMOS Technology, Circuits, and Systems*. Springer, 2020, pp. 479–518.

[6] J. Cirimelli-Low, M. H. Khan, S. Crow, *et al.*, "SRAM design with OpenRAM in SkyWater 130nm," in *Proceedings of 2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, Monterey, CA, 2023, pp. 1–4.

[7] 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*, 2016, pp. 1–6. DOI: 10.1145/2966986.2980098.

[8] Google and SkyWater Technology Foundry, *SkyWater Open Source PDK*, Last accessed 26 February 2023, 2020. [Online]. Available: https://github.com/google/skywater-pdk.

[9] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G. Taylor, "Magic: a VLSI layout system," in *21st Design Automation Conference Proceedings*, 1984. DOI: 10.1109/dac.1984.1585789.

[10] R. T. Edwards, *Netgen 1.5*, Last accessed 26 February 2023, Apr. 2017. [Online]. Available: http://opencircuitdesign.com/netgen/.

[11] M. A. Sivilotti, "Wiring considerations in analog VLSI Systems, with application to field-programmable networks," Ph.D. dissertation, California Institute of Technology, USA, Jan. 1991. [Online]. Available: https://dl.acm.org/doi/10.5555/145827.

[12] E. R. Keiter, H. K. Thornquist, R. J. Hoekstra, T. V. Russo, R. L. Schiek, and E. L. Rankin, "Parallel transistor-level circuit simulation," *Simulation and Verification of Electronic and Biological Systems*, pp. 1–21, 2011. DOI: 10.1007/978-94-007-0149-6_1.

[13] Hitachi America Ltd., *HN62321E Datasheet*, Last accessed 6 March 2023, 1990. [Online]. Available: https://datasheet.datasheetarchive.com/originals/distributors/Datasheets-111/DSAP0022889.pdf.

[14] Macronix International Co. Ltd., *MX23L3211 Datasheet*, Last accessed 6 March 2023, 2006. [Online]. Available: https://www.datasheetbank.com/MX23L3211-Datasheet-PDF-Macronix.