UNIVERSITY OF CALIFORNIA

SANTA CRUZ

Timing Characterization of an SRAM Generated from OpenRAM: An Open-Source Memory Compiler

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

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Brian Chen

June 2016

The Thesis of Brian Chen
is approved:

Professor Matthew Guthaus, Chair

Professor Martine Schlag

Professor James Stine

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Timing Characterization of an SRAM Generated from

OpenRAM: An Open-Source Memory Compiler

by

Brian Chen

As process technology continues to shrink, Integrated Circuits (ICs) can hold more memories on the chip to improve overall system performance, efficiency, and cost. Most academic IC design methodologies are inhibited by the availability of memory designs and timing characterizers. Many standard-cell Process Design Kits (PDKs) are available from vendors and foundries, but these PDKs do not come with any memory compilers or characterizers, while more expensive solutions only provide memory models with limited configurations and restrictive licenses. This thesis showcases OpenRAM, a characterization methodology and the author's contributions. The author added the characterization methodology to OpenRAM to be able to generate timing and power characterizations through SPICE simulations. He also reorganized the memory modules and rewrote the low-level parameterized modules such as transistors and inverters. In addition, he ported and managed the unit and regression tests from SVN to git. Lastly, he improved and implemented multiple dynamically generated memory modules. The goal of OpenRAM is to promote academic research by providing a portable and flexible platform for the generation and verification of memory designs across various technologies, sizes, and components.

Acknowledgments

First of all, I would like to express my gratitude to my adviser, Professor Matthew Guthaus, for his guidance throughout my graduate studies. I want to thank him for initially giving me the opportunity to be apart of a Research Experience for Undergraduates (REU) during my undergraduate studies. This gave me the courage to pursue graduate school and continue my research within his graduate Very-large-scale Integration Design and Automation (VLSI-DA) lab/group.

In addition, I would also like to thank the rest of my thesis committee: Professor James Stine and Professor Martine Schlag for their time and advice. I appreciate the help of all the graduate students in the OpenRAM group: Samira Ataei and Bin Wu. I thank them for all their ideas and contributions to the project. OpenRAM is a team effort and I would not have been able to get to where I am at today without their assistance.

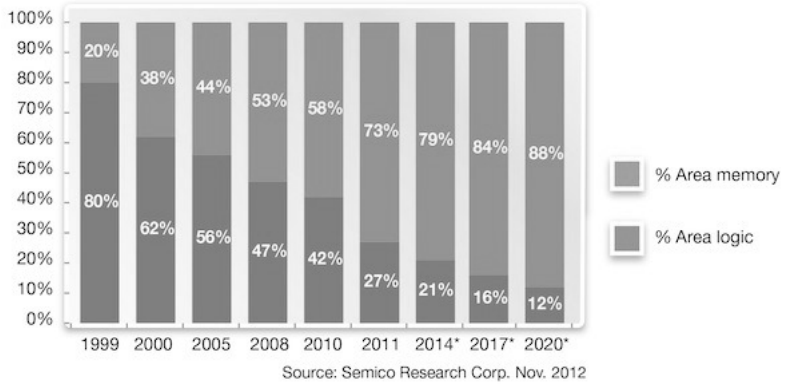
Furthermore, I want to extend my thanks to my fellow lab-mates at UCSC VLSI-DA lab: Riadul Islam, Hany Fahmy, Ping-Yao Lin, Rajsaktish Sankaranarayanan, and Rebecca Rashkin for being there whenever I am lost in my work and am in need of escape.

Last but not the least, I would like to thank my friends: Sonya Pita, Maria Lopez, Ting Gan, and Jonathan Lim for their patience, support, and encouragement throughout my academic endevours.

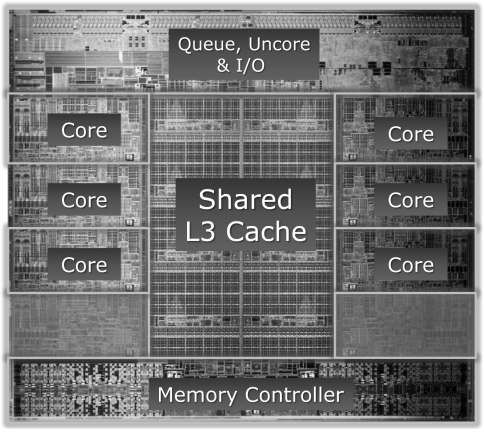
1 Introduction

Embedded Static Random-Access Memory (SRAM) has become a standard in Application-specific Integrated Circuits (ASICs), System-on-Chip (SoC), and micro-processor designs. The size and number of memories on-chip continues to increase as process technologies decrease in size. Memory designs plays a significant role in system performance and costs. Nowadays, embedded memories take up to 75% of the total chip area (Figures 1b and 1c).

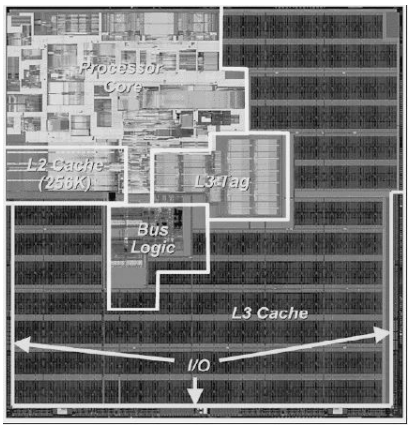
While we delve into smaller technologies, many difficult design challenges surface such as the increase in variability, instability, static power, and Static Noise Margins (SNMs). Due to these additional obstacles, SRAMs' have become one of the most challenging and time-consuming pieces to design for an IC.



(a)



(b)



(c)

Figure 1: a) Trend of memory and logic area for SOC's (Semico Research). b) Die photo of the Intel i7 Sandy Bridge [44]. c) Die photo of the Itanium 2 from Intel [41].

An SRAM has a very regular structure composed of replicated cells across a single design. This alone promotes custom design and the use of techniques such as compilers for the automatic generation of memories. Memory compilers are commonly used in the industry. There

are two flavors of memory compilers that exist in the industry. One can be obtained from vendors; the compiler is bundled with standard cell libraries. There are proprietary memory compilers but they are intellectual property owned by companies. The main disadvantage of these compilers are that they require expensive licenses and are "black boxes". They are not modifiable, have limited available configurations, and do not provide access to the source code. Some examples of industry memory compilers are Global Foundries' design kit [13], ARM's embedded IP memories [3], and Synopsys DesignWare memory compiler [49].

Commercial industry memory compilers are not a feasible solution for researchers in the academic scene. The customization limitations make comparison and experimentation with real systems impossible. Academic researchers can design a full-custom memory themselves but this can be time-consuming and may not even be apart of their research. Typically, the usage of memory is the simplest part that their research. The lack of an open-source memory compiler makes it difficult for researchers to prototype and verify circuits and methodologies. Multiple memory designs are needed for the verification of certain aspect for a system-level design, such as robustness or scalability.

The availability of an open-source memory compiler will enable further research in a variety of IC-related areas such as computer architecture with SoC design, memory circuit design, and Computer Aided Design (CAD). Computer architects and SoC designers need access to a variety of memory configurations to be able to prototype system-level implementations. Usually, researchers only have enough time to prototype a single memory instance or even just a single memory cell; this is insufficient to assess scalability and robustness of new ideas. By having access to an open-source memory compiler, many new research paths will become available in related fields for academic research. An attempt to accommodate all of these issues and needs, we showcase an open-source memory compiler.

OpenRAM is an open-source memory compiler that generates, characterizes, and verifies fabricable memory designs. This compiler can easily generate memory arrays and is portable across many technologies and operating systems. It provides reference designs in a generic 45nm technology (Freepdk45) and Scalable CMOS (SCMOS). We were able to port to several commercial technology nodes using a simple technology file. With the characterization methodology, we can generate timing/power characterization results while remaining independent of commercial tools. Most importantly, OpenRAM's source code can be modified by the user since OpenRAM is open source. The OpenRAM project is funded by the National Science Foundation (NSF) and is in collaboration between the VLSI-DA group (run

by Professor Matthew Guthaus) at the University of California - Santa Cruz [55], and the VLSI Computer Architecture Research group (run by Professor James Stine) at Oklahoma State University [45].

This thesis provides an overall summary of the OpenRAM memory compiler and an in-depth description of the characterizer. This includes its structure, implementation, features, and use. The author's contributions include: the design of basic parametrized cells, design of multiple SRAM components, technology-dependent setup scripts, modularization of regression/unit tests, and the SPICE memory characterizer. The thesis is organized as follows: Section 2 contains a brief summary on the existing memory compilers and on the previous works. Section 3 provides a background of the SRAM architecture and operation. Section 4 discusses the implementation of the OpenRAM memory compiler. Section 5 highlights the author's contributions to the project. Section 6 shows the timing and area results from the characterizer for a variety of SRAM sizes and configurations. Lastly, Section 7 summarizes the OpenRAM memory compiler and highlights some future work.

2 Background

Primitive memory compilers have been used in design flow to reduce the designing time long before our contemporary compilers [23, 7]. However, these compilers consisted of custom-designed scripts that must be changed for each technology. As design productivity began to slow down, higher levels of integration and formal research are focused on investigating memory array compiler. [36, 18, 59].

As technology continues into the Deep Sub-Micron (DSM) era, memory design became one of the most challenging parts of circuit design due to decreasing SNM, increasing variability, and increasing leakage power consumption. Memory compilers changed so that they can adapt to the ever-changing technology. Design methodologies shifted from silicon compilers to standard cell place and route. The industry started having third-party suppliers of standard cell libraries and memory compilers that allow the reuse of previous designs. These memory compilers could provide silicon-verified designs that allowed designers to focus on the quality of their work rather than time-consuming tasks such as memory generation.

While there are already existing memory compilers provided by industry, there are no robust memory compilers that are freely distributed and currently maintained. In academia, circuits and systems research is slowed because of the time-consuming tasks of memory generation. While there have been scripts that are released by various groups, these designs

are typically not silicon verified and usually only include simple structures.

2.1 Previous Work

Contemporary memory compilers have been researched widely and commercialized. Several prominent companies and foundries have provided memory compilers to their customers such as ARM Artisan, Virage Logic, and Virtual Silicon. These memory compilers usually include features that allow the customer to view simulations, timing/power values, and pin locations; these front-end features are usually only available when a paid license agreement is signed. Back-end features such as layout and SPICE netlists are normally supplied directory to the fab and are only given to the user for a licensing fee.

Specifically, Global Foundries [13] offers front-end PDKs for free, but not back-end detailed views. ARM [3] provides Intellectual Property (IP) instances of memories in TSMC and Global Foundries' technologies but not detailed views. Faraday Technologies [51] provides a "black box" design kit for UMC technologies, where we do not know the details of the internal memory design. Dolphin Technology [52] offers closed-source compilers which can create RAMs, ROMs, and CAMs for TSMC, UMC, and IBM technologies. Majority of these commercial compilers do not allow the customer to alter its based code because it is restricted by the company's rules and usually require a fee for its use.

There have been multiple attempts by academia to implement a memory compiler that is not restricted by industry standards. Examples of attempt include Institute of Microelectronics' SRAM IP Compiler [60], School of Electronic Science and Engineering at Southeast University's Memory IP Compiler [34], and Tsinghua University's Low Power SRAM Compiler [58]. Another example, NCSU's FabMem [42], is able to create small arrays, but it is dependent on the Cadence design tools and is targeted for a fixed technology. These scripts do not provide any characterization capabilities, and cannot easily integrate with commercial place and route tools. These mentioned attempts only provide methodologies and design flows for a memory compiler. They do not provide the actual memory compiler for the public to use or to view.

Synopsys [49], a leading company in the Electronic Design Automation industry, offers academia a possible solution with their recently released Generic Memory Compiler (GMC) [14]. GMC is available for universities that signs up for Synopsys University Program. Synopsys provides the the software and sample generic libraries such as Synopsys' 32/28nm and 90nm abstract technologies. This in turn can generate the whole SRAM for

these technologies. GMC provides everything that a memory compiler should have. GMC is able to generate GDSII layout data, SPICE netlists, Verilog and VHDL models, timing/power libraries, and DRC/LVS verification reports. GMC uses a user configuration file to determine what type of memory should be compiled. This compiler seemed like the perfect solution for academic researchers' needs but the main drawback this memory compiler shows is that it was designed for educational and training purposes only and is not recommended for fabrication. GMC's main purpose was to aid academic students learn about memory designs and other related designs that requires memory. GMC never intended to provide academic researchers a tool to generate fabricable memory designs for real-world designs.

With all these attempts, there are still no decent solutions for academia's research needs. We need a memory compiler that is open-source, portable, technology independent, and can generate fabricable memory designs. In addition, we also need characterization for those generated memories.

3 Architecture

Before discussing the implementation of OpenRAM, this section will give an overview of a typical SRAM architecture that is implemented using OpenRAM.

3.1 SRAM Architecture

The OpenRAM SRAM architecture is based on a bank of memory cells with peripheral circuits and a control logic circuitry as shown in Figure 2. These are further refined into eight major blocks: an array of bit-cells, the address decoder, the word-line drivers, the precharge circuitry, the column multiplexers, the sense amplifiers, the write drivers, and the control logic. Using the memory array as the main block, the address decoder is located to the left of it and uses word-line drivers to drive the word-lines of each row of the memory array. The precharge circuitry is connected at the top of each column in the memory array. The bit lines are connected through a bidirectional column multiplexer (mux) at the bottom of the memory array to the sense amplifier and the write driver. The control/timing circuitry is the core of the SRAM that ensures functionality of the SRAM. In addition, we use D-Flip-Flops (DFFs) for the address and data values to enable the synchronous operation of the SRAM.

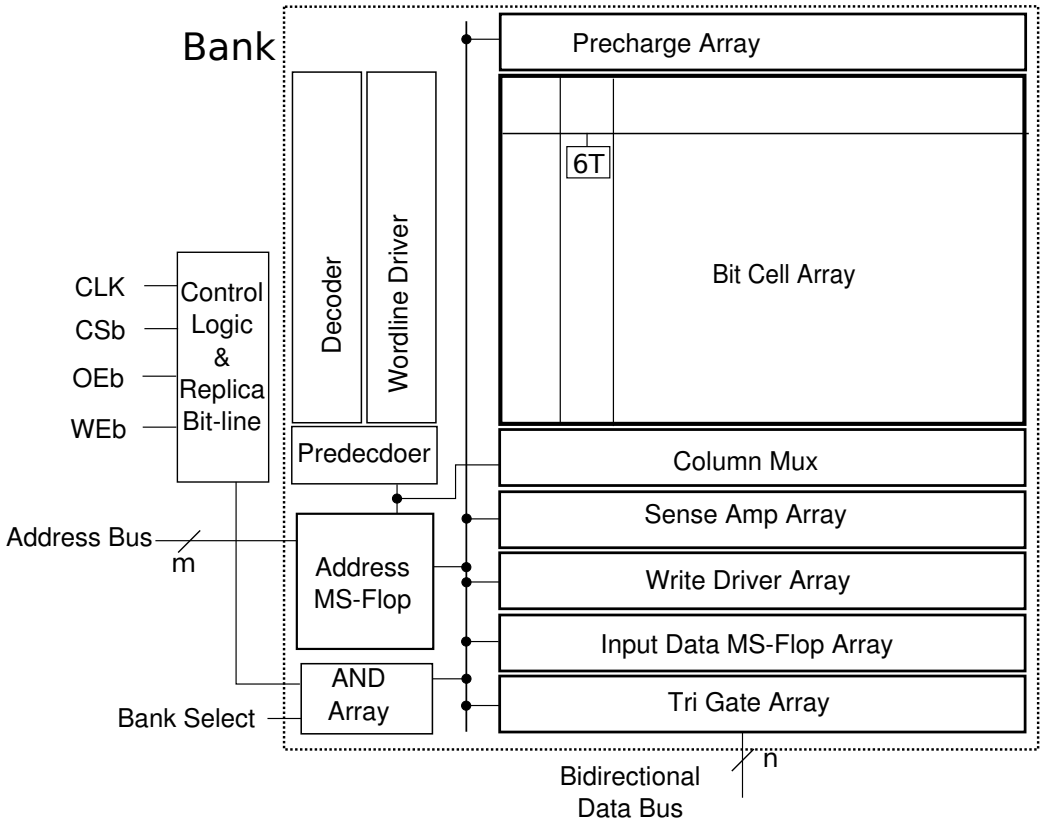


Figure 2: An OpenRAM's single port SRAM Architecture consisting of a bit-cell array along with decoder, reading/writing circuitry and control logic timed with a replica bit-line.

The following sub-sections explain the operation of each individual block within the SRAM, followed by a high-level explanation as to how these different blocks interact to function as a memory device. It is important to note that the circuits described below are the ones that are used in the first release of the OpenRAM memory compiler. By no means is this an exhaustive list of the possible circuits that can be adapted into a SRAM architecture.

3.1.1 The 6T Cell and Bit-cell Array

In the initial release of OpenRAM, the 6T cell will be used as the default memory cell because the 6T cell is the most commonly used memory cell in SRAM devices. It is named "6T cell" because it consists of six transistors: two access transistors (`M1` and `M2`) and two cross-coupled inverters as shown in Figure 3. The cross-coupled inverters hold a data bit, `X`, and its inverted value, `X_bar`. This bit value can either be written into or read from the bit-cell via its bit-lines. The access transistors are used to isolate the bit-cell from the bit-lines so that data is not corrupted while a cell is idle. In addition to the 6T cell, there

are alternative type of memory bit-cells such as 7, 8, and 10 Transistor (T) cells that are compatible.

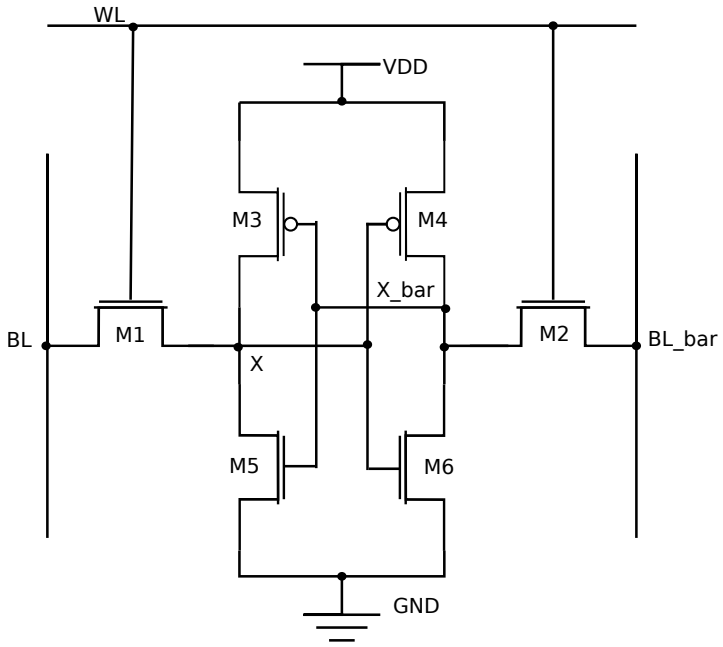


Figure 3: A schematic of a bit-cell constructed using 6 transistors.

There are two main operations associated with memory: reading and writing. We can access the 6T cell to either retrieve the saved data value or to save a new data value within the 6T bit-cell. When a read operation is performed, both bit-lines are precharged to `Vdd`. This precharging is done during the first half of the read cycle and is performed by the precharge circuitry (See Section 3.1.4). In the second half of the read cycle, the word-line is asserted, which enables the access transistors. If a logic value of "1" is stored in the bit-cell, then `BL_bar` is discharged to `Gnd`, and `BL` is charged up to `Vdd`. On the other hand, if a logic value of "0" is stored in the bit-cell, then `BL` is discharged to `Gnd` and `BL_bar` is charged to `Vdd`. While performing a write operation, the bit-lines are again precharged to `Vdd` during the first half of the write cycle. Similar to the read operation, the word-line is again asserted, and the access transistors are enabled during the second half of the write cycle. The data value that is to be written into the cell is applied to `BL`, and its complement is applied to `BL_bar`. The write drivers that are driving the data signals to the bit-lines must be appropriately sized so that the previous data value in the memory cell can be overwritten (See Section 3.1.7).

In addition to the sizing of the write drivers, the transistors in the 6T bit-cell must also be carefully sized to ensure reliable operation. For a read operation, the `M5` and `M6` must be

sized larger than access transistors M1 and M2. When the word-line is asserted, both X and X_bar are initially pulled up to the precharge value. Assuming that a logic value of "1" is stored at X, X_bar must remain a logic value of "0" regardless of the voltage rise experienced when the word-lines are asserted. The larger M5 and M6 will insure that its stored values will not change because the resistance of the access transistors will be greater. During a write operation, the data value stored in the cell is being overwritten. This means that M1 and M2 must be strong enough to overpower the feedback inverters and must be sized larger than M3 and M4 [11, 27].

The bit-cell should be designed as small as possible so that the bit-cell array can be as dense as possible. The size of the memory array is directly related to the total number of words and the size of each word. For example, an 1kB memory with a word size of eight bits will consist of 8 columns and 128 rows. It is a common practice to keep the aspect ratio of the memory array as square as possible to ensure that the bit-lines and word-lines do not become too long. This can increase the capacitance of these wires and slow down the operation. This will also increase power leakage as well. Instead of having a long memory array in the vertical directions, we would rearrange the memory array to have 32 columns and 32 rows. Each row will have four words and can correctly be selected using the column mux (See Section 3.1.5).

The 6T cells are tiled together in both the horizontal and vertical directions to make up the whole bit-cell array. Each bit-cell is connected to each other through abutment. In the vertical direction, each cell is simply tiled together such that their bit-lines are abutted together in addition to the sharing of the Vdd rail. In the horizontal directions, each cell is tiled next to each other so their word-lines are abutted. They will also be able to share the Gnd rail as well.

3.1.2 Address Decoder

The address decoder takes the row address bits from the address bus as inputs, and asserts the appropriate word-line in the row so that the correct bit-cell can be read from or written to. An n-bit row-input can control 2^n word-lines. OpenRAM provides a couple of options for address decoders, but the default is a hierarchical decoder which provides higher overall performance.

A hierarchical decoder is a type of decoder which is constructed hierarchically. A simple 2-to-4 decoder is shown in Figure 4. The decoder takes in the address bits from A0 and

A1 and will assert the correct word-line. For example, if the address input is 00, where the least significant bit being right, the output would be 0001, where the rightmost is the least significant bit. The 2-to-4 decoder uses inverters and two 2-input nand gates for its construction while the gates are sized to have equal rise and fall time. As the decoder size increase, the size of the nand gates must also increase. Table 1 shows the truth table of the inputs and possible outputs.

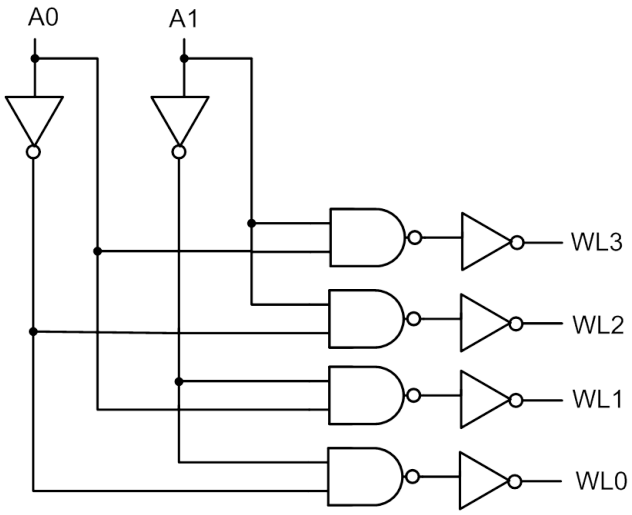


Figure 4: A schematic of a simple 2-to-4 decoder constructed with NAND gates and inverters.

A[1:0]	Selected WL
00	0
01	1
10	2
11	3

Table 1: The truth table for a 2-to-4 hierarchical decoder showing the input/output combinations.

An n-bit hierarchical decoder requires 2^n logic gates, each with n inputs. For example, with n = 6, 64 NAND6 gates are needed to drive 64 inverters to implement the decoder. It is clear that gates with more than 3 inputs create large series resistances and long timing delays. Rather than using n-input gates, it is preferable to use a cascade of gates. Typically two stages are used: a pre-decode stage and a final decode stage. The pre-decode stage generates intermediate signals that are used by multiple gates in the final decode stage.

Figure 5 shows a 4-to-16 hierarchical decoder. The structure of the decoder consists of two 2-to-4 decoders for pre-decoding and the combinations of 2-input nand gates and

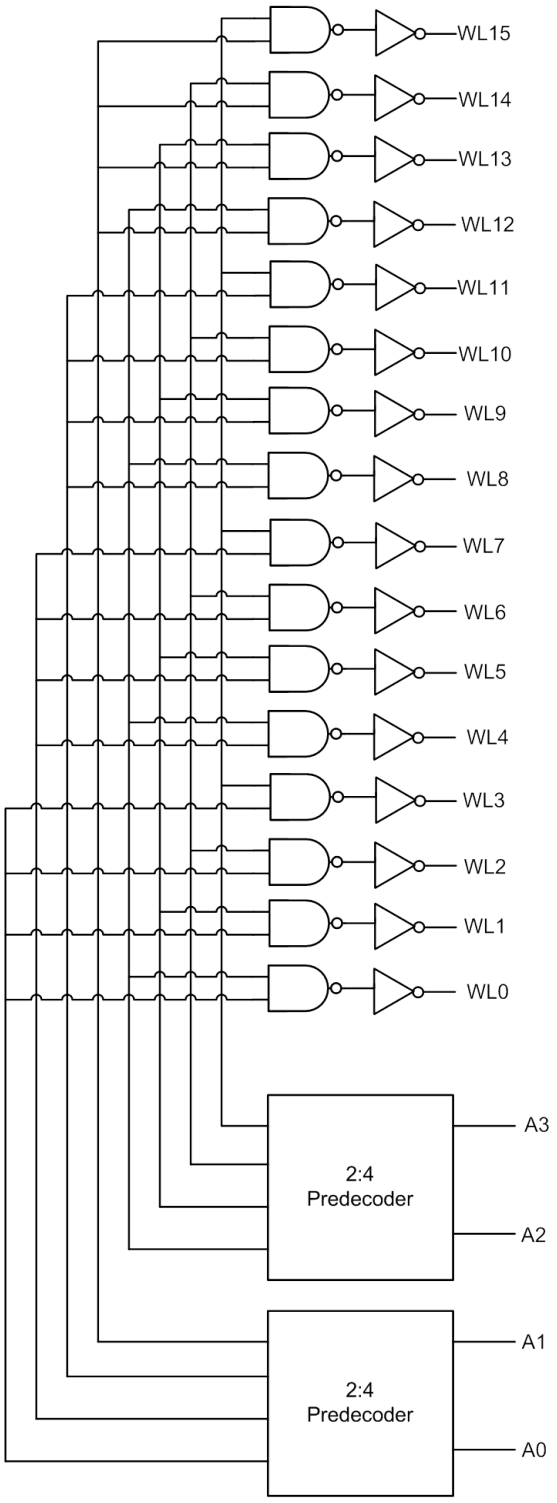


Figure 5: A schematic of a 4-to-16 hierarchical decoder that is hierarchically designed with smaller 2-to-4 decoders and basic logic gates.

inverters for the final decoding to form the 4-to-16 decoder. In the pre-decoder, a total of 8 intermediate signals are generated from the address bits and their complements. The

A[3:0]	pre-decoder1	pre-decoder2	Selected WL
0000	1000	1000	0
0001	1000	0100	1
0010	1000	0010	2
0011	1000	0001	3
0100	0100	1000	4
0101	0100	0100	5
0110	0100	0010	6
0111	0100	0001	7
1000	0010	1000	8
1001	0010	0100	9
1010	0010	0010	10
1011	0010	0001	11
1100	0001	1000	12
1101	0001	0100	13
1110	0001	0010	14
1111	0001	0001	15

Table 2: The truth table for 4-to-16 hierarchical decoder showing the input/output combinations

concept of using a pre-decoding and a final decoding stage for construction of address decoder is very productive because small decoders like 2-to-4 decoders are used for pre-decoding. The operation of a 4-to-16 hierarchical decoder can be explained with an example. If the input address is `0000`, the output of the pre-decoder1 and pre-deocder2 will be `0001` and `0001`, respectively. According to the connections in Figure 5, wordline 0 of pre-decoder1 and pre-decoder2 are connected to the first 2-input nand gate in the final decode stage representing the wordline 0 of the final decoding stage. Hence depending on the combination of the input signals, one of the wordline will be asserted. In this example, wordline 0 should will be asserted. Table 2 gives a truth table describing the possible outputs depending on the input.

As the size of the address bits increases, higher level decoder can be created using the lower level decoders. For example, for an 8-to-256 decoder, two instances of 4-to-16 becomes the pre-decoder, followed by 256 2-input nand gates and inverters forms the whole address decoder. In order to construct the 8-to-256 decoder, first 4-to-16 decoder should be constructed through using 2-to-4 deccoders. Thus the pre-decoder stage will can be composed of hierarchical components, hence why it is called a hierarchical decoder.

Another option OpenRAM provides is the NAND decoder. Figure 6 illustrates a 2-to-4 dynamic NAND decoder which operates as follows: during the first phase of the clock (i.e. while clock is low), the `PCLK` signal enables the PMOS transistors which precharge all of the

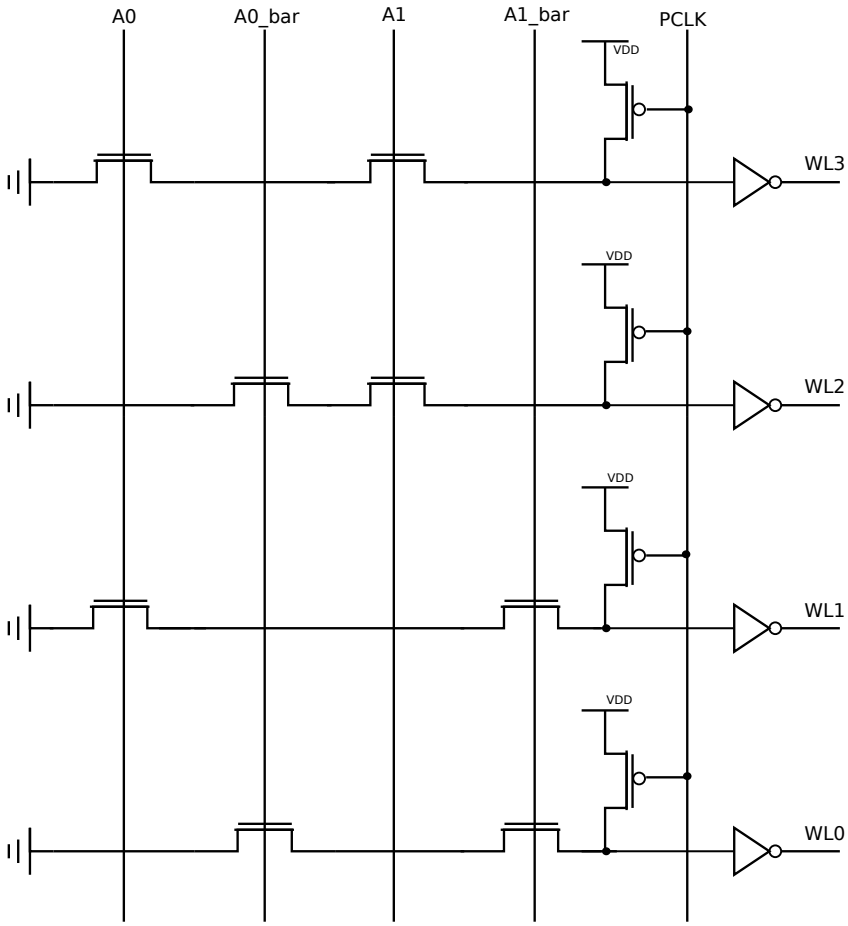


Figure 6: A transistor level look of a 2-to-4 NAND decoder constructed using NAND gates and inverters.

internal word-lines (to the left of the output inverters) to `Vdd`. During the second phase of the clock, the PMOS transistors are disabled. Based on the input address, a specific internal word-line is pulled down to ground [2]. The output inverters ensure that no word-lines are asserted during the precharge phase and that only one is asserted during the second phase.

3.1.3 Word-line Drivers

The word-line drivers are simple buffers that are used to drive a signal through the bit-cell array's word-lines. They are inserted between the address decoder and the bit-cell array. The word-line drivers' sizes are based on the width of the bit-cell array. The word-line drivers ensure that as the size of the memory array increases, and the word-line capacitance increases, the signal from the address decoder is still able to turn on the access transistors in all the bit-cells. As shown in Figure 7, the word-line driver is composed of `pinv`s and `NAND2` gates which are dynamically sized and created to appropriately drive the word-lines.

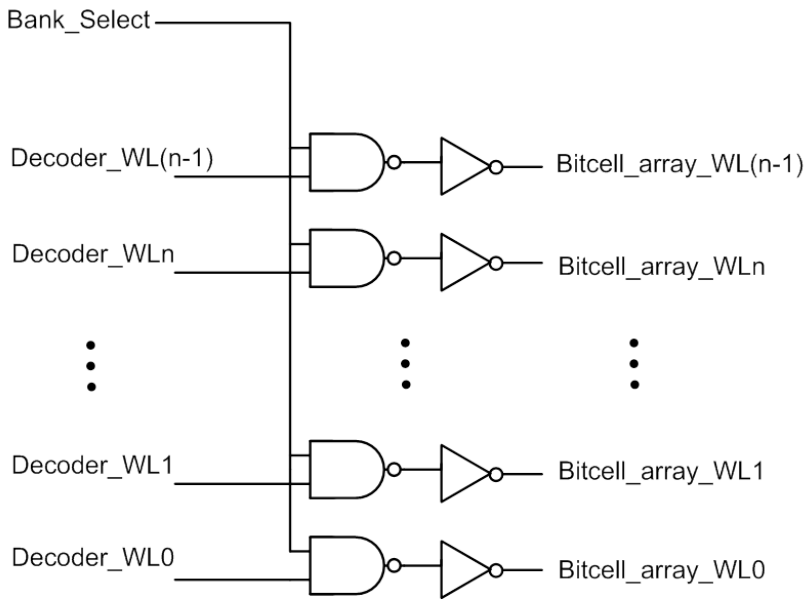


Figure 7: A schematic of a word-line driver constructed using NAND gates and inverters.

3.1.4 Precharge Circuitry

The precharge circuitry is a fairly simple circuit that consists of three P-type Metal-oxide-semiconductor (PMOS) transistors; it is shown in Figure 8. It uses two of them to control the precharging of the bit-lines during the first phase of the clock cycle during read and write operations. It has a third PMOS transistor to connect the two bit-lines together so it can help equalize the voltage between the two bit-lines.

There are two states for the precharge circuitry, active and idle. During the first phase of the clock cycle, when the `CLK` signal is high, the precharge circuitry charges the bit-lines to `Vdd` using the `PCLK`, known as the complement of `CLK`. The circuitry goes to its idle state when the `CLK` signal goes low during the second phase of the clock cycle. Once the precharge phase ends, one of the bit-lines should experience a voltage drop. For a read operation, the sense amplifier is able to sense the voltage difference between the bit-lines quicker due to the precharging of the bit-lines (See Section 3.1.6). For a write operation, the asserted memory bit-cells will be able to see the voltage difference and save the new data logic into the bit-cell.

3.1.5 Column Multiplexer

The column multiplexer (column mux) takes in `n`-bits from the address bus and can select `2^n` bit-line pairs associated with one word in the memory array. The column mux is used

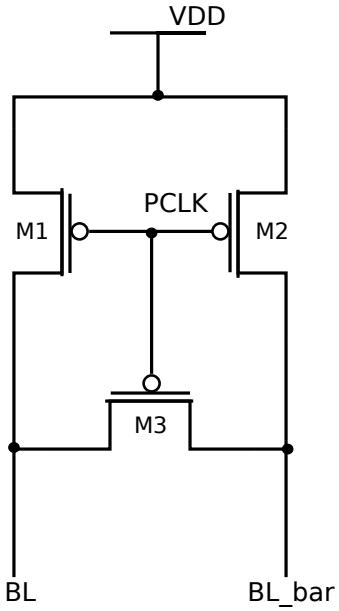


Figure 8: A schematic of a single precharge cell using PMOS transistors.

for both read and write operations; it connects the bit-lines to the bit-cell array to both the sense amplifiers and write drivers. OpenRAM provides a couple of options for the column mux, but the default is a single-level column mux which is sized for optimal speed.

The simplest design of a single level column mux is to use a single NMOS device that is driven by the address signal. Figure 9 shows a schematic of a 2-to-1 single-level column mux. For a 2-to-1 mux, 1 select signal is used to select between the two inputs. The selected transistors will connect their corresponding bit-lines to the sense amplifier and write driver. Figure 10 shows the schematic for a 4-to-1 single-level mux. In this particular column mux, two address bits are decoded using a 2-to-4 decoder. The decoder provides a one-hot set of signals for the selecting of the column mux. This ensures only one of select signals is asserted.

An alternative to the single-level column mux, is the tree column mux. The schematic for a 4-to-1 tree multiplexer is shown in Figure 11. This type of tree mux is bi-directional and is used for both the read and write operations. Just like the single-level mux, it connects the bit-lines of the memory array to both the sense amplifier and the write driver too.

As seen in Figure 11, the column mux is built of NMOS transistors in a tree-like structure. The depth of the decoder is determined based on the number of words per row in the bit-cell array. The most basic column mux has a depth one which means that there are two words per row. If there is only one word per row in the array, then no column mux is needed. As

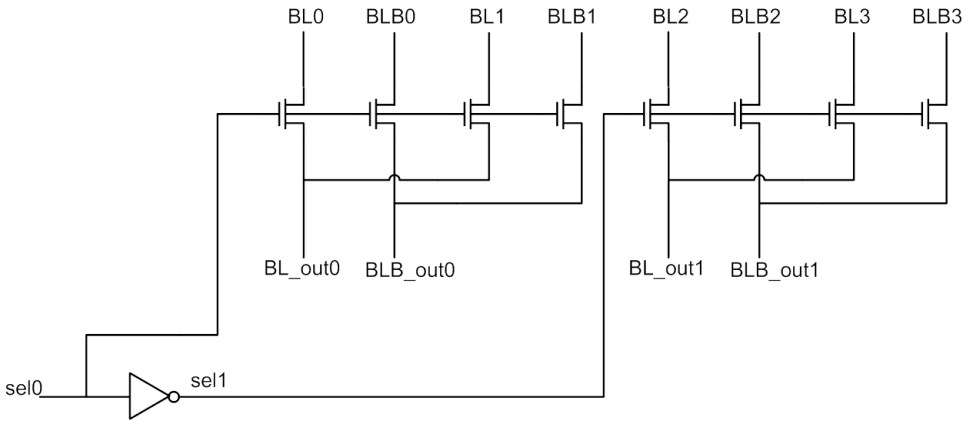


Figure 9: A schematic of a 2-to-1 single-level column mux constructed using intervers and NMOS transistors.

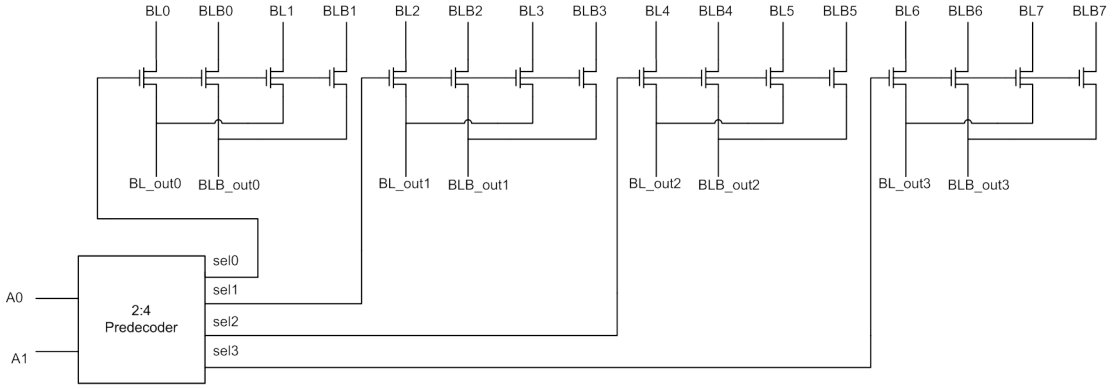


Figure 10: A schematic of a 4-to-1 single level column mux constructed using a 2-to-4 decoder and NMOS transistors.

BL Pair	Inp1	Inp2	Binary
BL0/BL0_bar	A0_bar	A1_bar	00
BL1/BL1_bar	A0	A1_bar	01
BL2/BL2_bar	A0_bar	A1	10
BL3/BL3_bar	A0	A1	11

Table 3: Binary reduction pattern for 4-to-1 tree column mux.

the number of words per row in the bit-cell array increases, the depth of the column mux grows. The depth of the column mux is equal to the number of bits in the column address bus.

A binary reduction pattern, shown in Table 3, is used to select the appropriate bit-lines. In level one, `A0`, and its complement `A0_bar`, select either the even numbered words or the odd numbered words in the row. In level two, the most significant bit `A1` and its complement `A1_bar`, select one of the words passed down from the previous level. One downside to this

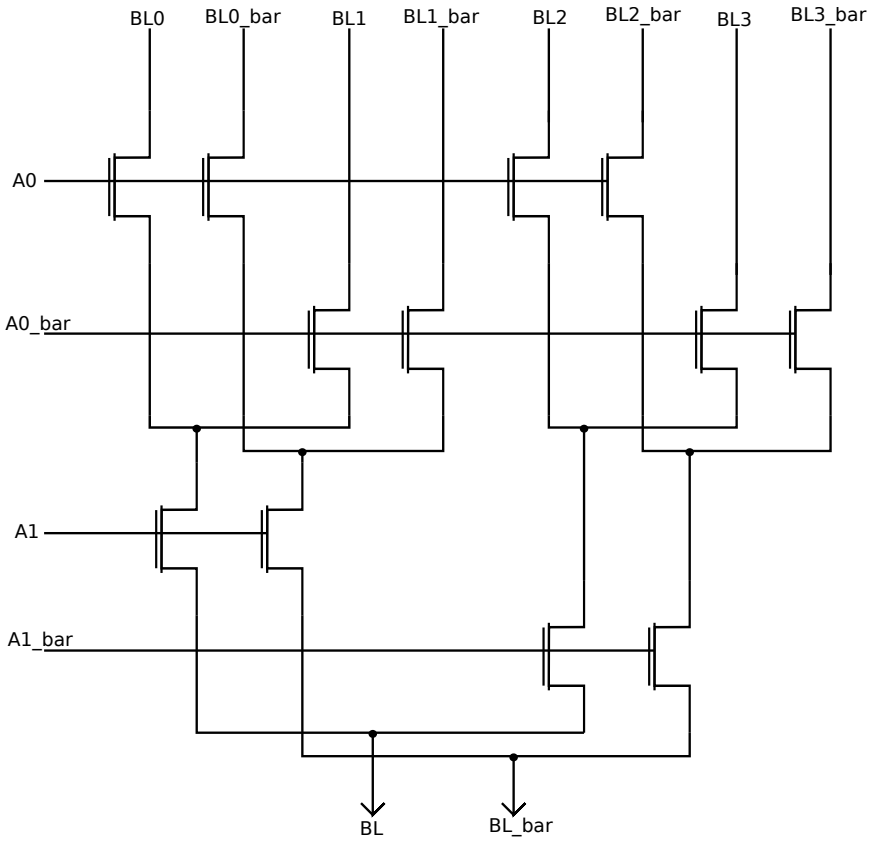


Figure 11: A schematic of 4-to-1 tree column mux that passes both of the bit-lines.

type of mux is that the delay of of a tree mux quadratically increases with each level[39].

3.1.6 Sense Amplifier

The sense amplifier (sense amp) is used to sense the voltage difference between the bit-lines (`BL` and `BL_bar`) while a read operation is performed. The sense amp is necessary to recover the signals from the bit-lines because they do not experience full voltage swing. As the size of the bit-cell array grows, the load of the bit-lines increases and thus the voltage swing is limited by the drive strength of the small bit-cell.

The schematic for the sense amp is shown in Figure 12. The sense amplifier is enabled by the `SCLK` signal, which initiates the sensing operation. Before the sense amplifier is enabled, the bit-lines are precharged to `Vdd` using the precharge circuitry (See Section 3.1.4). During the second phase of the read operation, after the precharge stage is complete, one of the bit-lines experiences a voltage drop based on the data value stored in the bit-cell. The sampling process begins and continues until the sense amp is activated some time during the second phase of the read operation. When the sense amp is activated, its PMOS isolation transistors will disconnect the sense amps from the bit-lines and begin the sensing process. This isolation

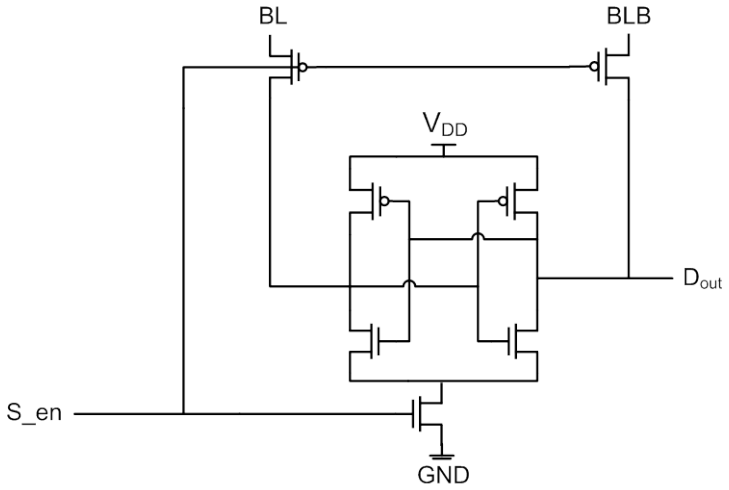


Figure 12: A schematic of a single sense amplifier cell showing its design.

technique helps speed up the response so that the bit-lines high capacitance will not affect the sensing of the voltage difference. This technique requires a timing technique to activate the sense amp at the proper time (See Section 3.1.10). The output signal is then taken to a true logic level and captured using a master-slave DFF (See Section 3.1.8).

3.1.7 Write Driver

The write driver is used to drive the input signal into the bit-cell during a write operation. It can be seen in Figure 13 that the write driver consists of two tristate buffers, one inverting and one non-inverting. It takes in a data bit, and outputs that value on to the bit-line, and its complement on the bit-line bar. Both tristates are enabled by the `EN` signal. The bit-lines always need to be complements to ensure that the correct data is stored in the bit-cell. Also, the drivers need to be appropriately sized as the memory array grows and the bit-line capacitance increases.

3.1.8 Master-slave DFF

In a synchronous SRAM, it is necessary to synchronize the inputs and outputs with a clock signal by using a synchronous flip-flop. In OpenRAM, we use a master-slave DFF for the synchronizing process. An schematic is shown in Figure 14. This ensures that the input and output signals stay valid for the entire clock-cycle.

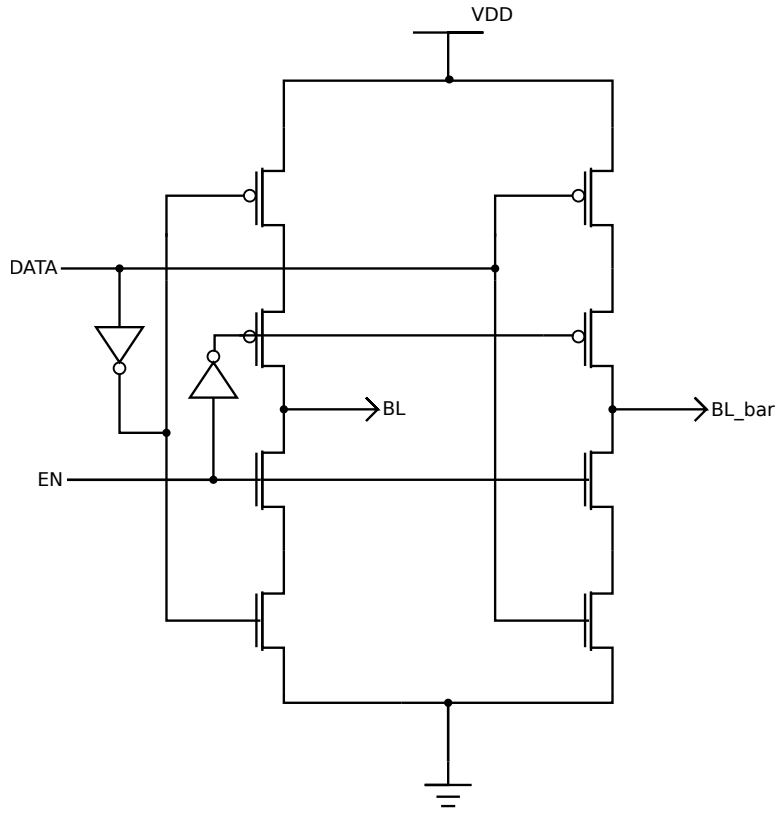


Figure 13: A schematic of a write driver cell, which consists of two tristates (non-inverting and inverting) to drive the bit-lines.

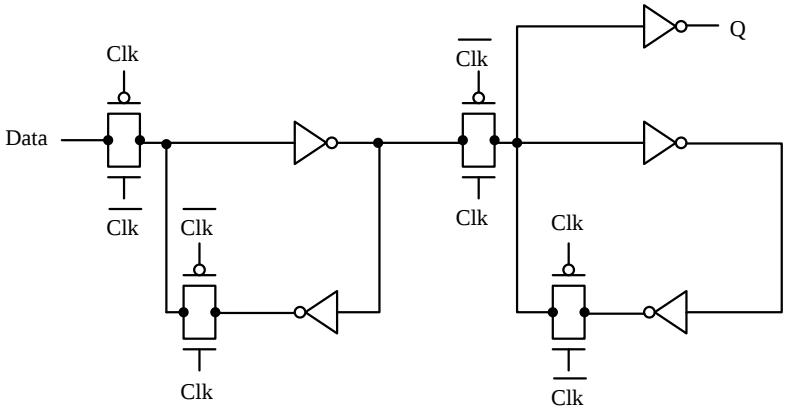


Figure 14: A schematic for simple master-slave DFF used in a synchronous SRAM.

3.1.9 Multiple Banks

As shown in Figure 15, all the previous mentioned components can be combined to form a single bank. In order to reduce timing delay and power consumption, divided word-line strategy have been used. Part of the address bits are used to define the global word-line (bank-select). The rest of the address bits are connected to the address decoders to generate

local word-line signals that drives the bit-cells' access transistors. As illustrated in Figure 16, the SRAM is divided into two banks, where both are using the same data bus, address-bus, and control logic signals. At most, only one bank will be active at a time. In this example, the word-line and bit-line delay is reduced by a factor of two; therefore, the power will be greatly reduced because the cells' capacitance is lowered. In Figure 17, an example of four banks are connected together. For this case, a 2-to-4 decoder is added to select one of the banks to be used by one-hot encoding. Only one single control logic will be used since it is connected to all four banks.

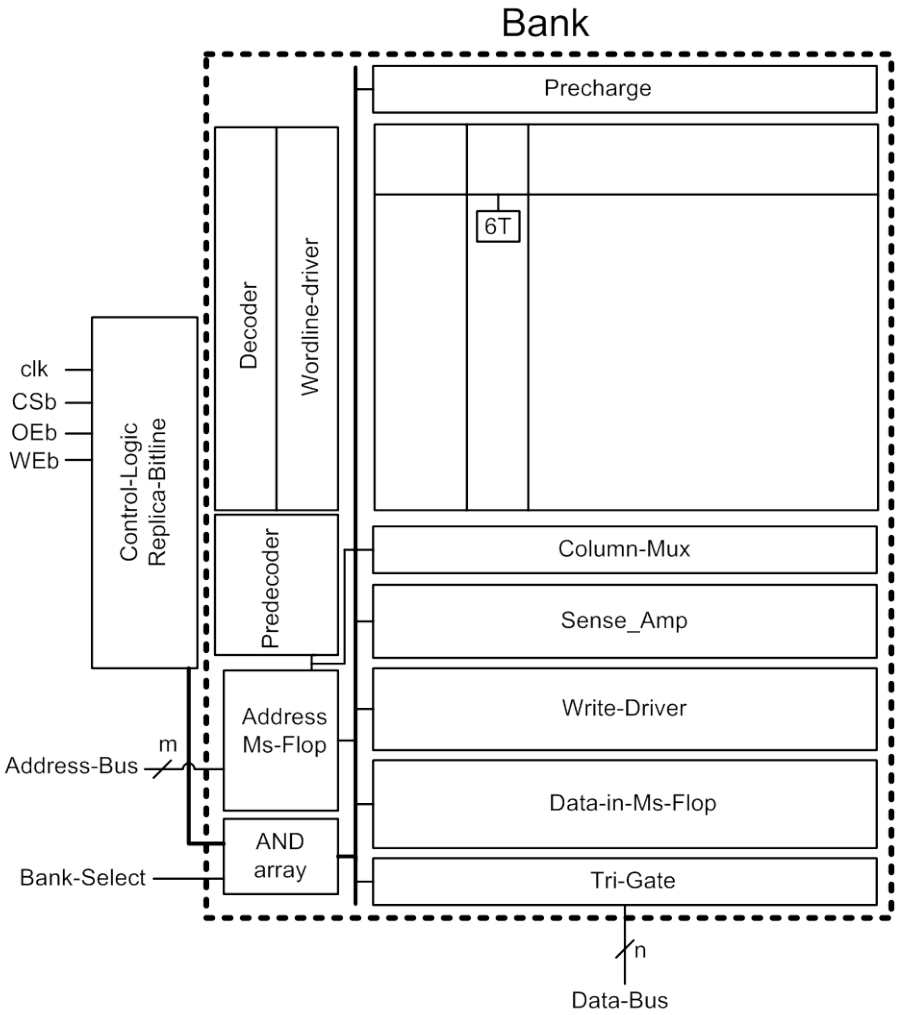


Figure 15: An overall bank showing what is included in a single bank.

3.1.10 Control Logic

The control circuitry ensures that the SRAM operates as intended during a read or write cycle by enabling the necessary structures, at the correct timing, in the SRAM. As displayed

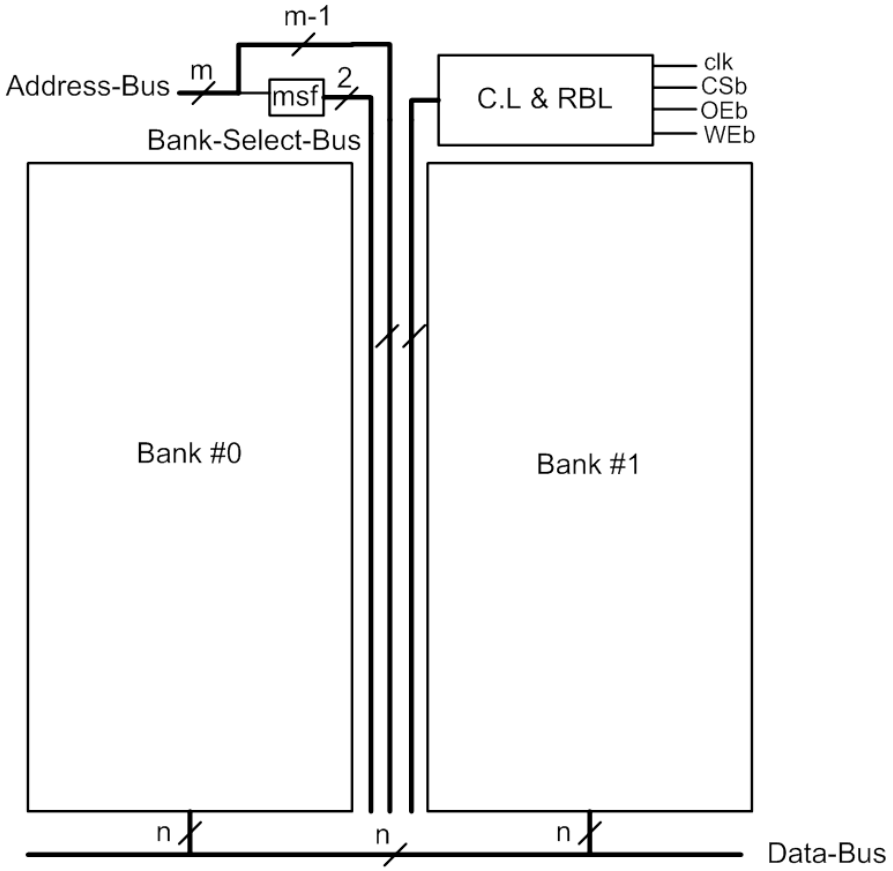


Figure 16: An SRAM is divided to two banks which share the control-logic.

in Figure 18, the control logic takes three active low signals as inputs: chip select bar (`CSb`), output enable bar (`OEb`), and write enable bar (`WEb`). `CSb` enables the entire SRAM chip. While `CSb` is low, the appropriate internal control signals are generated and are sent to the architecture blocks. Conversely, if `CSb` is high, those internal control signals will disable those structures. The `OEb` signal represents a read operation; the data values seen on the data bus will be an output from the memory. Similarly, the `WEb` signal signifies a write operation and the data seen on the data bus is meant to be stored in the SRAM.

As shown in Figure 18, these three control signals are captured with master-slave DFFs to ensure that the these signals stay valid for the entire operation cycle. The captured signals are combined with the global clock signal to generate internal control signals used to enable or disable the SRAM structures based on its current operation. The generated internal `s_en` signal is used to enable the sense amplifier during a read operation. The `w_en` signal, generated using a Replica Bit-line `(RBL)` (See subsection in 3.1.10), enables the write driver during a write operation. To control the usage of the bi-directional data bus

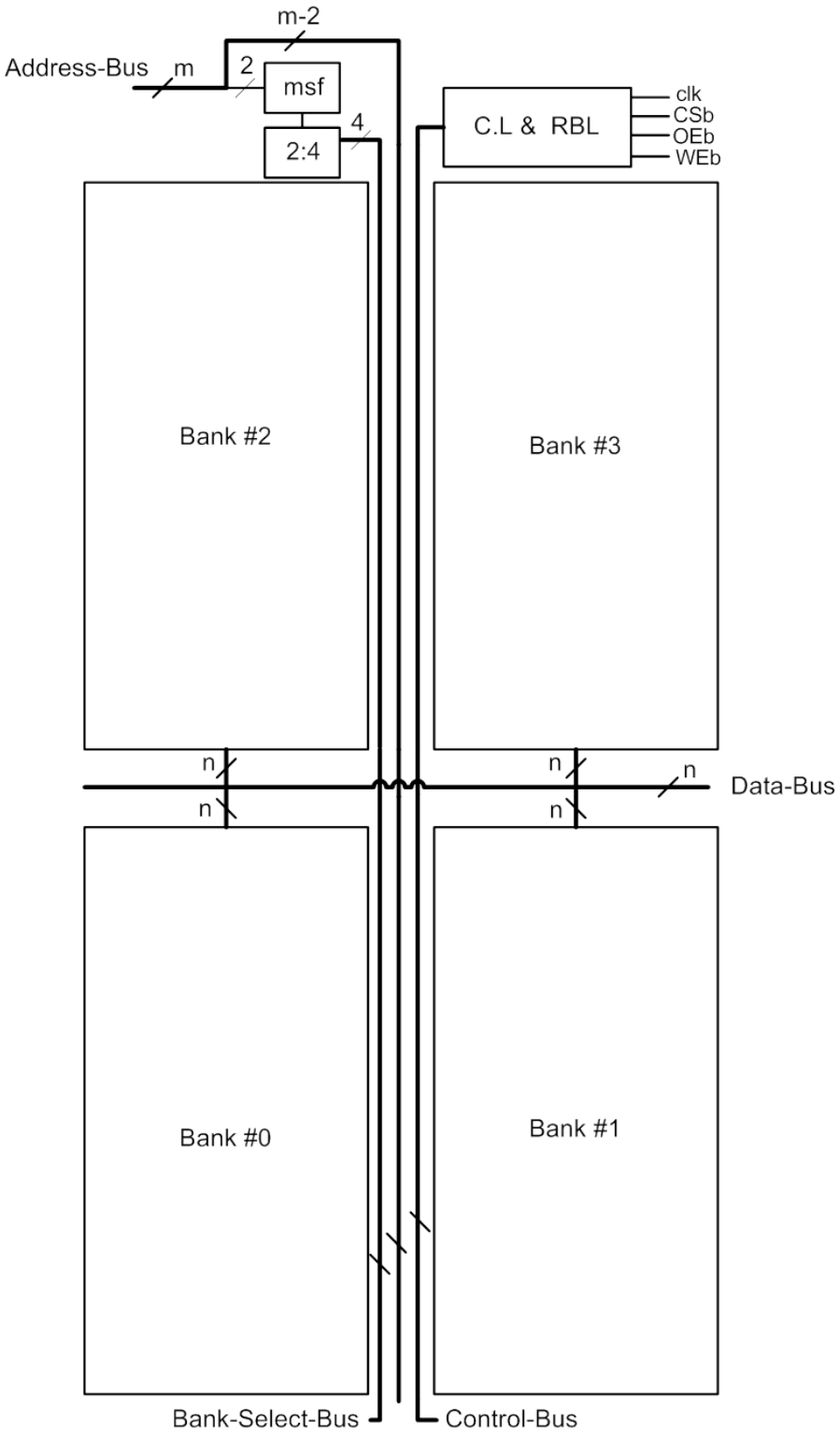


Figure 17: An SRAM is divided to 4 banks which are controlled by the control-logic and a 2-to-4 decoder.

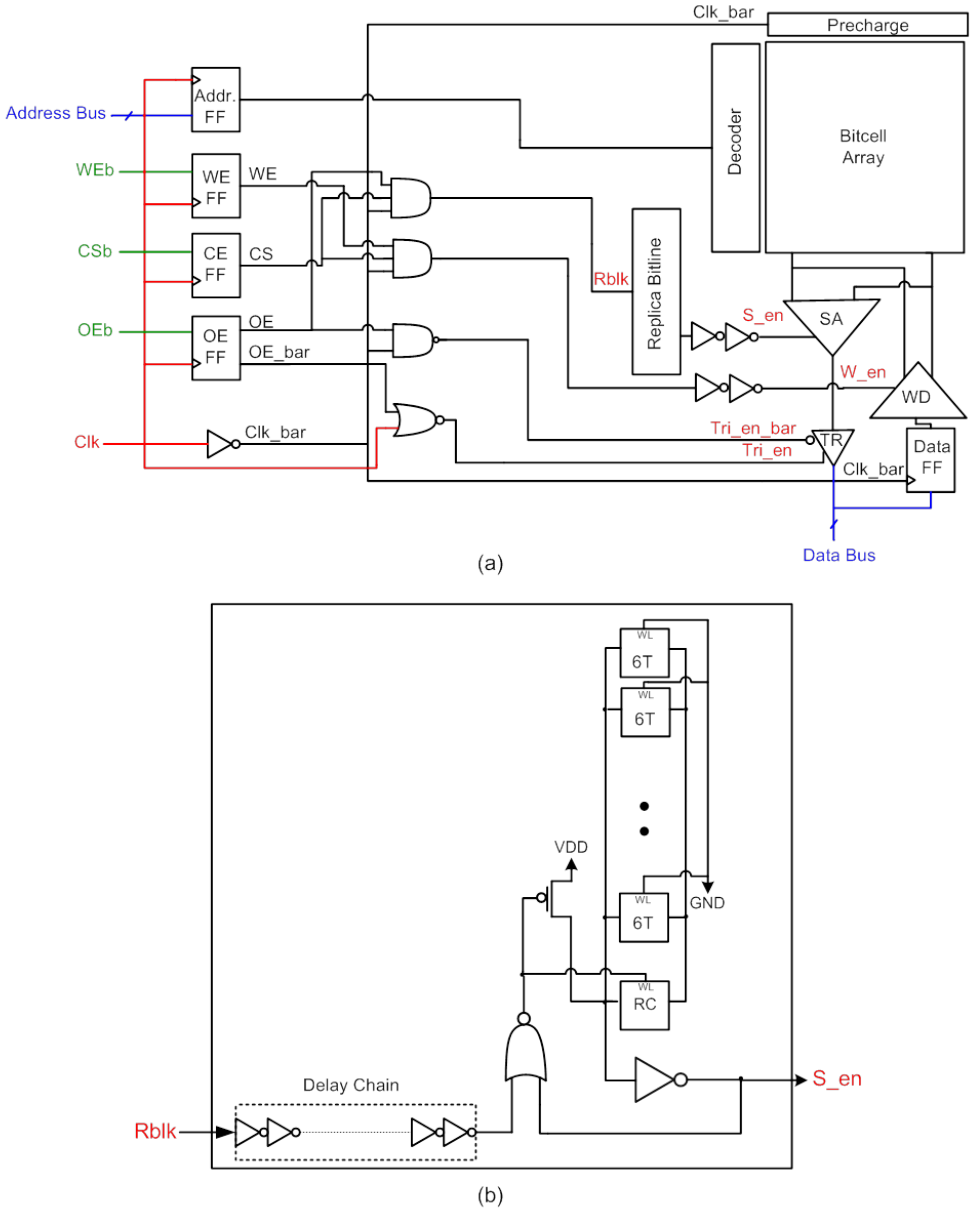


Figure 18: (a) The Control Logic diagram displaying the timing circuitry and (b) The Replica Bit-line schematic that uses a delay-chain that is sized depending on the number of bit-cells vertically.

for both a read and a write operation, the control logic generates `tri_en` and `tri_en_bar` signals for the tri-stating of the data bus. Table 4 shows the truth table for the control logic. The `s_en` signal is active when $\neg(CSb \vee OEb) \wedge CLK$. Similarly, `w_en` signal is active when $\neg(CSb \vee WEb) \wedge CLK$. Lastly, the `tri_en` and `tri_en_bar` are active when $\neg(OEb_bar \vee clk)$ and $\neg(OEb \wedge clk_bar)$ are true, respectively. All of these signals are "ANDED" with the clock because these circuits should only be enabled after the precharging of the bit-lines has ended.

Operation	Inputs			Outputs		
	CSb	OEb	WEb	s_en	w_en	tri_en
READ	0	0	1	1	0	1
WRITE	0	1	0	0	1	0

Table 4: The truth table of the control signals combinations.

Replica Bit-line Delay In an SRAMs' read operations, discharging the bit-line is the most time-consuming portion of the read operation. Generally, sense-amp increases the small voltage difference seen on the bit-lines at the proper sense timing, to realize high-speed operations. Therefore, the timing of the activation of sense amps is crucial for high-speed and low-power SRAMs. If the control signal, `s_en`, arrives too early before the bit-line voltage difference reaches the sense amps, a read functional failure may occur. Additionally, if the `s_en` signal arrives too late after the voltage difference reached the sense amps to isolate the signal from the bit-lines, unnecessary time will be wasted thereby consuming more power.

To generate the correct control signal, `s_en` to enable optimal operation, OpenRAM utilities the Replica Bit-line (RBL) technique for the generation of `s_en` [6]. As shown in Figure 19, The RBL circuit consists of a column of dummy SRAM bit-cells, which track the random process variation in the array. Since the RBL is constructed of the same bit-cells as in our bit-cell array, we can generate the `s_en` signal to match the timing of the data signal reaching the sense amps. The delay shift of the control path according to the Process, Voltage, and Temperature (PVT) variation is the same ratio as of the read path. By using replica circuits, the variation on the delay of the sense amp activation and bit-line swing is minimized.

Each Replica Cell (RC) drives a short bit-line signal. This bit-lines' capacitance is set to be a fraction of the main bit-line's capacitance. This fraction is determined by the required bit-line swing for proper sensing. The capacitance ratio is therefore based off the ratio of the geometric lengths of the main bit-lines. The RC is hard wired to store a zero value such that it will discharge the RBL once it is accessed. Since this cell is similar to the actual bit-cell in terms of design and fabrication, the delay of RBL tracks the delay of the real bit-lines and therefore the delays will be roughly equal.

The RBL technique generates a signal, `s_en`, to activate the sense amps, as follows. At the beginning of the clock cycle, both the real bit-lines and RBL are precharged to `Vdd`. During the second half of the clock cycle, the selected bit-cells are activated for reading. The normal bit-lines are discharged through the accessed bit-cell. At the same time, the

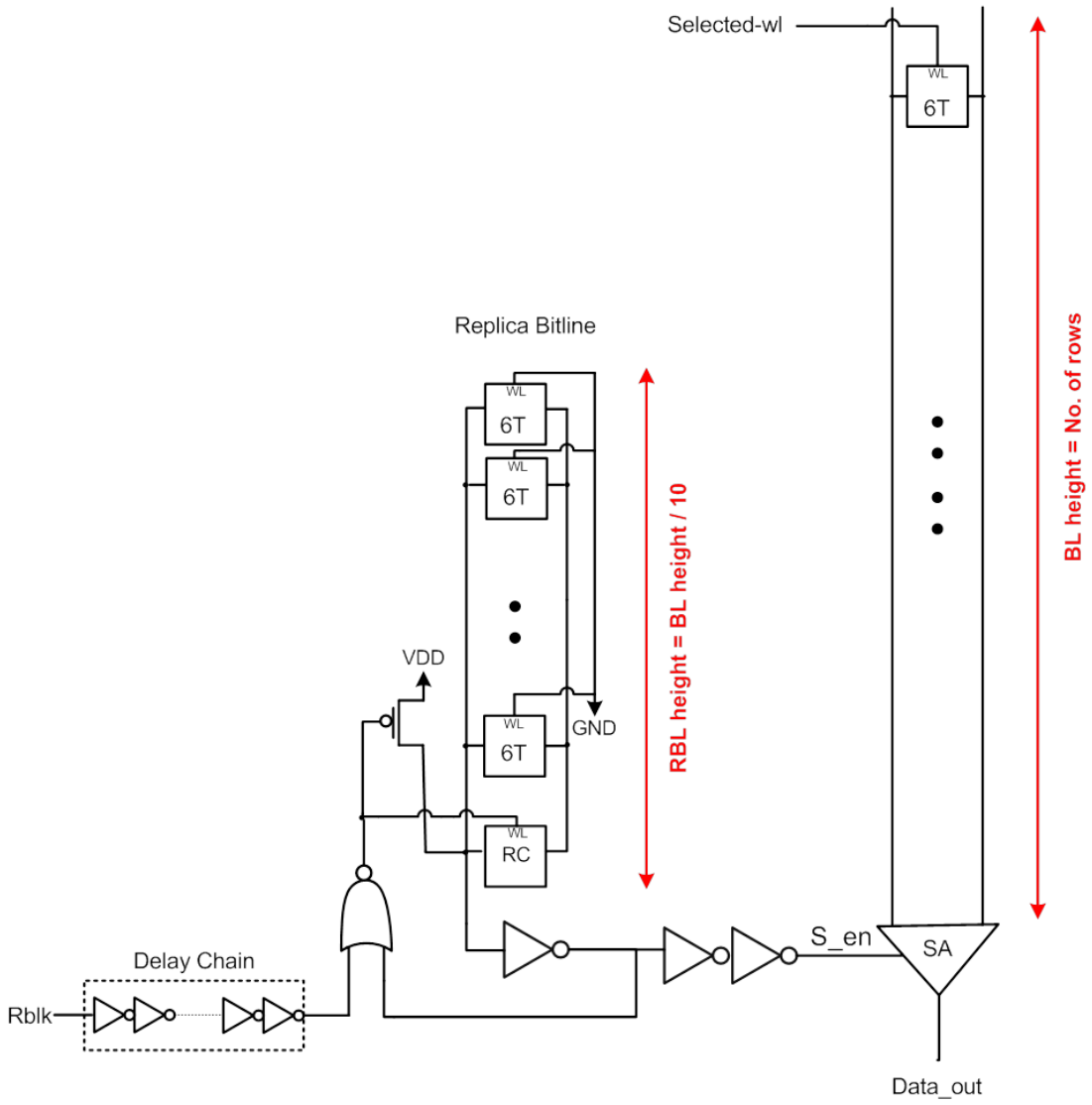


Figure 19: The Replica Bit-line schematic showing that the RBL is the same height as the bit-cell array.

RC draws current from the RBL to simulate the operation of the discharging bit-lines. The discharged voltage swing on the RBL is inverted and buffered to generate the `s_en` signal to enable the sense amps.

3.2 SRAM Operation

In order to explain the read and write operations of a SRAM, it is necessary to summarize the internal and external signals as well as the important timing considerations.

Top Level Signals

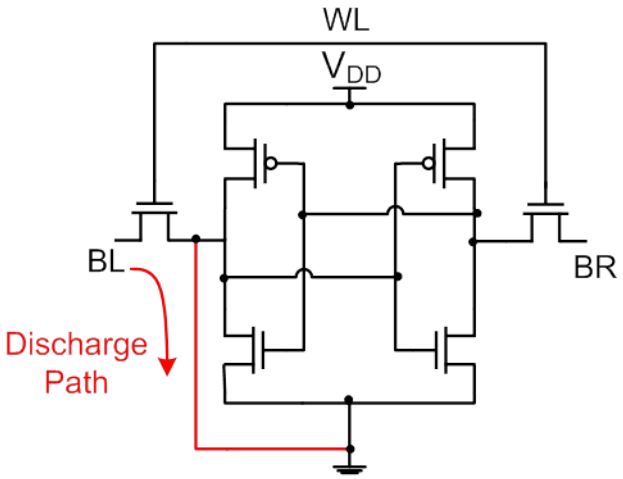


Figure 20: A schematic of the Replica Bit-line cell.

- `ADDR` - the address bus.
- `DATA` - the bi-directional data bus.
- `CLK` - the global clock.
- `OEb` - the output enable signal (active low).
- `CSb` - the chip select signal (active low).
- `WEb` - the write enable signal (active low).

Internal Control Signals

- `clk_bar` - enables the precharge circuitry.
- `s_en` - enables the sense amplifier during a read operation.
- `w_en` - enables the write driver during a write operation.
- `tri_en` and `tri_en_bar` - enables the data tri-gates during a read operation.

Timing Considerations

- Setup Time - time an input needs to be stable before the positive/negative clock edge.
- Hold Time - time an input needs to stay valid after the positive/negative clock edge.
- Minimum Cycle Time - time in-between subsequent memory operations.
- Memory Read Time - time from positive clock edge until valid data appears on the data bus.

- Memory Write Time - time from negative clock edge until data has been driven into a memory cell.

3.2.1 Read Operation

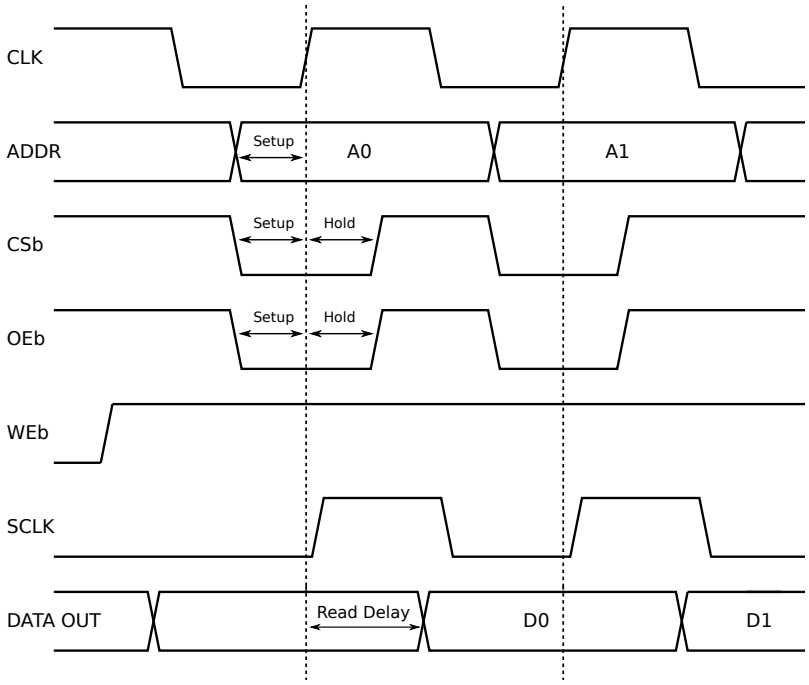


Figure 21: Timing diagram for read operation showing the setup, hold, and read times.

Figure 21 displays the timing diagram for the SRAM read operation. It highlights the setup and hold times for each signal as well as the memory read time. The list below provides a step-by-step description of what is happening internally in the SRAM[19].

Read Operation

1. Before the read cycle begins, (before the `CLK` signal transitioning from low to high): (These signals should be set with respect to the setup time)
   (a) The chip must be selected therefore `CSb` should be set low.
   (b) The `WEb` signal must be set high to disable the write drivers.
   (c) The `OEb` signal must be set low to signify a read operation.
   (d) The address signals should be valid on the ADDR bus for capturing.
2. On the rising edge of the clock (`CLK`):
   (a) The control signals and address are captured into DFFs and the read cycle begins.

(b) The precharging of the bit-lines begins for the first half of the clock cycle.

(c) The address bits become available for the address decoder and column mux, which select the row and columns of the bit-cell array that we want to read from.

3. On the falling edge of the clock (`CLK`):

(a) A word line is asserted, the value stored in the selected bit-cells pulls down one of the bit-lines (`BL` if a 0 is stored, `BL_bar` if a 1 is stored).

(b) `s_en` enables the sense amplifier which senses the voltage difference of the bit-lines, produces the output and keeps the value in its latch circuitry.

(c) The Tri-gate enables and puts the output data on data bus. Data remains valid on the data bus for the remainder of the clock cycle.

3.2.2 Write Operation

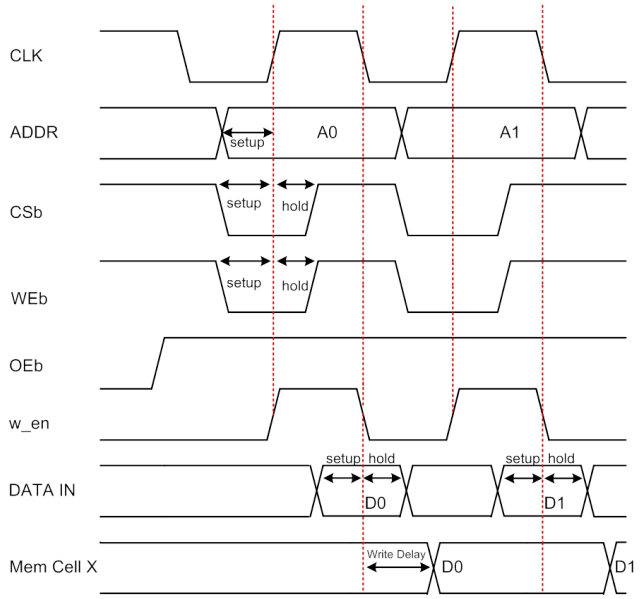


Figure 22: Timing diagram for write operation showing the setup, hold, and write times.

Figure 22 displays the timing diagram for the SRAM write operation. It highlights the setup and hold times for each signal as well the memory write time. The write time is illustrated by the "X" signal in the diagram, which is the internal storage node in the bit-cell. The list below provides a step-by-step description of what is happening internally in the SRAM[19].

Write Operation

1. Before the write cycle begins, (before the `CLK` signal transitioning from low to high): (These signals should be set with respect to the setup time)
   (a) The chip must be selected therefore `CSb` should be set low.
   (b) The `WEb` signal must be set low to enable the write drivers and the data-input tri-state gates.
   (c) The `OEb` signal must be set high to disable the sense amplifiers to disable any read operations.
   (d) The address signals should be valid on the ADDR bus for latching.
2. On the rising edge of the clock (`CLK`):
   (a) The control signals and address are captured into DFFs and the write cycle begins.
   (b) The precharging of the bit-lines begins for the first half of the clock cycle.
   (c) The address bits become available for the address decoder and column mux, which select the row and columns of the bit-cell array that we want to write to.
3. On the falling edge of the clock (`CLK`):
   (a) The data to be written must be applied to DATA bus and captured into DFFs on the falling edge (See subsection in 3.2.2).
   (b) The `w_en` signal enables the write driver, which drives the data input through the column mux and into the selected bit-cells.
   (c) The write delay is the time from the negative clock edge until the data value is stored in the memory cell on node X.

Zero Bus Turnaround (ZBT) In the timing of SRAMs, for a read operation, data should be available after the clock edge; but for a write operation, data should be set up before the clock edge. Due to this nature, a wait state (a dead cycle) is necessary when SRAM switches from a read operation to a write operation. This issue ultimately slows down the read and write operations and degrades the performance of SRAMs. In OpenRAM, we avoided this problem by using the Zero Bus Turnaround (ZBT) technique [20]. With ZBT, data to be written should be set up after the positive clock edge and before the negative clock edge so that the data input is captured on the falling edge of the clock instead of the rising edge. In

the scenario when we go from a read operation to a write operation, we do not have to wait for the next rising edge and instead take advantage of the next negative edge to complete our write operation. By using ZBT, we will get a higher memory throughput and avoid the use of wait states. Figure 23 shows how the aforementioned scenario where we do not have to use a wait state in-between a read and write operations.

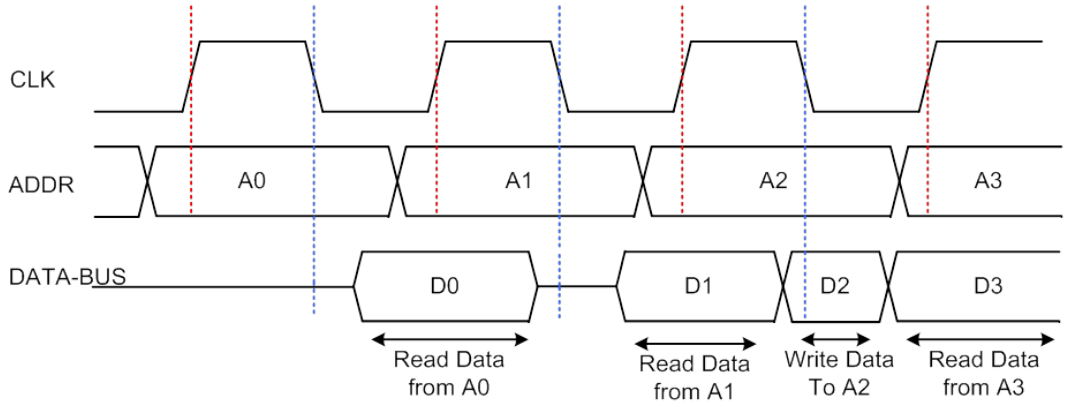


Figure 23: Zero Bus Turnaround timing

4 Software Implementation

OpenRAM, the compiler framework is divided into "front-end" and "back-end" methodologies as shown in Figure 24. The "front-end" consists of the compiler, which generates Spice models and GDSII layouts based on user inputs, and the characterizer, which calls a Spice simulator and produces timing/power numbers. The "back-end" uses the generated spice netlists and GDSII layouts to generate annotated timing and power models using back-annotated characterization. This section will discuss, in detail, the implementation of the OpenRAM compiler and the characterizer.

4.1 Main Compiler Components

OpenRAM is implemented using object-oriented data structures in the Python programming language [38]. Python was chosen because it is a simple, yet powerful language that is easy to learn and has a syntax that is very human-readable. The open-source release also includes the FreePDK45 technology kit developed by Stine, et al., at North Carolina State University [46]. FreePDK45 is a free, 45 nm, process design kit that is used for designing, modeling, and simulating integrated circuits. OpenRAM also utilizes GdsMill, a Python

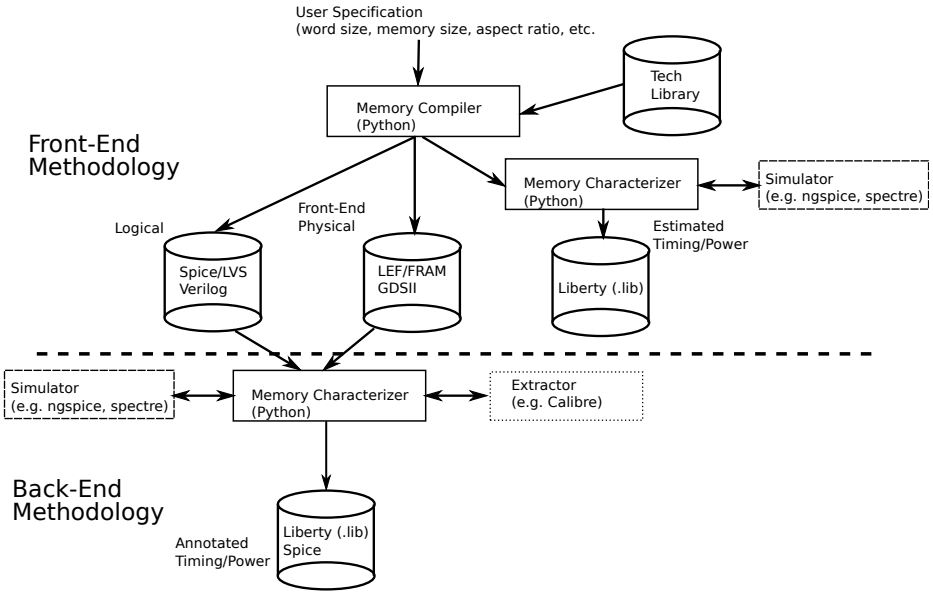


Figure 24: Overall Compilation and Characterization Methodology

interface developed by Michael Wieckowski that is used to create and manipulate circuit layout in the GDSII format [57].

4.1.1 OpenRAM Design Hierarchy

All modules in OpenRAM are derived from the top-level `design` class. The design class is a data structure that consists of a Spice netlist, a layout, and a name. The Spice netlist capabilities are inherited from the `hierarchy_spice` class while the layout capabilities are inherited from the `hierarchy_layout` class (See Figure 25). There is an additional function in the `design` class called `DRC_LVS()`. This function will perform a DRC/LVS check on the module.

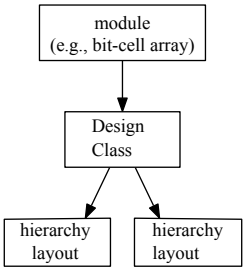


Figure 25: Class Hierarchy

Spice Hierarchy When the design class is initialized for a module, a data structure for the Spice hierarchy is created. This `hierarchy_spice` class will hold the spice structures

for the modules, the pins of the modules, and the pin connections as well as useful functions to add and connect instances in the hierarchy. The list of pins for the module are added to the sub-circuit definition by using the `add_pin()` function. The `add_mod()` function adds an instance of a module, library cell, or parameterized cell as a sub-circuit to the top-level or current structure. Each time a sub-module has been added to the hierarchy, the pins of the sub-module must be connected using the `connect_pins()` function. One limitation of the pin data structure is that the pins must be listed in the same order as they were added to the sub-module. Also, the other limitation is that the number of net connections must match that of the sub-circuit definition. The `hierarchy_spice` class also contains utility functions for reading or writing Spice files:

- `sp_read()`: this function is used to read in a Spice netlist file and parse the inputs defined by the "subckt" definition.
- `sp_write_file()`: this function recursively writes the modules and sub-modules from the Spice data structure into a Spice (.sp) file.

Layout Hierarchy A data structure for the layout hierarchy is also created when an instance of a design is initialized. The `hierarchy_layout` class has two main components: a structure for the instances of sub-modules contained in the layout, and a structure for the objects (such as shapes, labels, etc...) contained in the layout. Utility functions are also provided to add these instances and shapes to the layout hierarchy:

- `add_inst(self,name,mod,offset,mirror)` function adds an instance of a physical layout, (library cell, module, or parameterized cell), to the module. The parameters are:

  name - name of the instance.

  mod - the associated Spice module.

  offset - the x-y coordinates, in microns, where the instance should be placed in the layout.

  mirror - mirror or rotate the instance before it is added to the layout. Accepted values for mirror are: `"R0", "R90", "R180", "R270"`
  `"MX" or "x", "MY" or "y", "XY" or "xy"`

- `add_rect(self,layerNumber,offset,width,height)` function adds a rectangle shape to the module's layout. The parameters are:

layerNumber - the layer that the rectangle is to be drawn in.

offset - the x-y coordinates, in microns, where the rectangle's origin will be placed in the layout.

width - the width of the rectangle, can be positive or negative value.

height - the height of the rectangle, can be positive or negative value.

- `add_label(self,text,layerNumber,offset,zoom)` function adds a label to the layout. The parameters are:

  text - the text for the label

  layerNumber - the layer that the label is to be drawn in.

  offset - the x-y coordinates, in microns, where the label will be placed in the layout.

  zoom - magnification of the label (ex: "1e9").

- `gds_read()` function reads in a GDSII file and creates a `vlsiLayout()` class for it (See Section 4.1.2).

- `gds_write()` function writes the entire GDS of the object to a file by GdsMill `vlsiLayout()` class and calling the `gds2writer()` (See Section 4.1.2).

- `gds_write_file()` function recursively writes all instances and objects in the layout data structure to the gds file.

Library and Dynamically Generated Cells In OpenRAM, there are two options when it comes to modules: library and dynamically generated. The library cells are custom-designed-cells that have been verified and imported by the user. These cells tend to be difficult to implement dynamically or need to be custom designed based on area or performance constraints. The memory cell in the SRAM is a prime example of a library cell. This cell should be hand-designed to minimize area, because it is the most replicated cell. Dynamically generated designs can contain instances (sub-modules) of library cells, parameterized cells(See Section 4.2), and GDSII shapes. These designs are created using the GdsMill interface.

4.1.2 GdsMill

GDSII is the standard file used in the industry to store the layout information of an integrated circuit. The GDSII file is a stream file that consists of records and data types that hold the

data for the various instances, shapes, and labels in the layout. In OpenRAM, we utilize a tool called GdsMill to read, write, and manipulate GDSII files. As mentioned before, GdsMill was developed by Michael Wieckowski at the University of Michigan and has a completely unrestricted license [56].

In GdsMill, the `vlsilayout` class contains all data relating to the structures and records in a GDSII layout. The `gds2_reader` and `gds2_writer` classes contain the various functions used to convert data between GDSII files and the `vlsilayout` class. The `gds2_reader` and `gds2_writer` classes also process the GDSII data by iterating through every record and structure in the file. GDSII records are stored in the `vlsilayout` data structure so that they can be easily utilized and/or manipulated by Python code. Each record type has a corresponding class defined in the `gdsPrimitives` class. Thus, a `vlsilayout` should contain the following member data:

- `self.rootStructureName` - name of the top-level design.
- `self.structures` -list of structures that are used in the class.
- `self.xyTree` - a list of all structure names that appear in the design.

In the compiler, dynamically generated cells and arrays each need to build a `vlsiLayout` data structure to represent the hierarchical layout. This is performed using various functions from the `vlsiLayout` class in GdsMill. To simplify our usage, OpenRAM has its own wrapper class called `geometry`. This wrapper class aggregates the most important and frequently used layout class functions and constructs data structures for the instances and objects that will be added to the `vlsiLayout` class. The functions `add_inst()`, `add_rect()`, `add_label()` in `hierarchy_layout`, add the structures to the `geometry` class, which is then written out to a GDSII file using GdsMill.

Additionally, GdsMill can read in a user-designed library cell. The cell file should be in GDSII file format. The cell's information such as cell size and pin locations can be obtained by using existing functions in the `VlsiLayout` class. The cell size can be found by using `readLayoutBorder()`. To ensure proper functionality, a boundary layer should be drawn in the library cell to indicate the cell area. The `readLayoutBorder()` function will return the width and height of the boundary. If the boundary layer was not drawn in the layout, then `measureSize()` function is used as the backup to find the physical size of the cell. The pin location can be found by using the `readPin()` function. It will return the largest boundary which covers the label with the same layer as the label.

4.1.3 Technology Directory

The open-source release of OpenRAM includes a fully implemented SRAM and supporting technology directory for the default technology, FreePDK45. All process-specific information and library cells are contained within the FreePDK45 technology directory. Technology specific parameters, such as DRC rules and the GDS layer map, must be added to this directory to ensure that dynamically generated designs are conforming to the technology's rules. Similarly, modules that utilize library cells check the GDS and Spice libraries in this directory for custom-designed-cells to be added to the design hierarchy. Lastly, the technology directory should include any necessary helper functions to port the compiler to a new technology. Some technologies may have very specific design requirements that may not be natively supported by OpenRAM. Any helper functions should be added to the technology directory so that the main compiler can remain free of dependencies to specific technologies.

4.2 OpenRAM Modules

OpenRAM provides design modules for the various blocks of a SRAM. Each module has a corresponding Python class that instantiates a design data structure to hold the layout and Spice hierarchies. The modules consist of library cells and/or dynamically generated designs that are added as instances to the module's design hierarchy. Below is brief description of each module included in OpenRAM:

Parameterized Transistors OpenRAM provided a class, called `ptx`, that generates parameterized transistors of specified type and size. The `ptx` takes the transistor width, number of fingers, and type (PMOS or NMOS) as inputs and dynamically generates the design utilizing the various GdsMill functions described in Section 4.1.2. The parameterized transistor is the basic building block for all dynamically generated modules in the OpenRAM compiler. An example is shown in Figure 26.

Parameterized Inverter A second parameterized class, `pinv`, generates parameterized inverters. This class takes the NMOS transistor size, beta value, and desired cell height as inputs, and uses instances of `ptx` transistors and GdsMill shapes to dynamically generate the inverter. If the cell height cannot accommodate a single transistor of the specified size, the transistor is split into multiple fingers. This makes the cell grow wider, but the effective

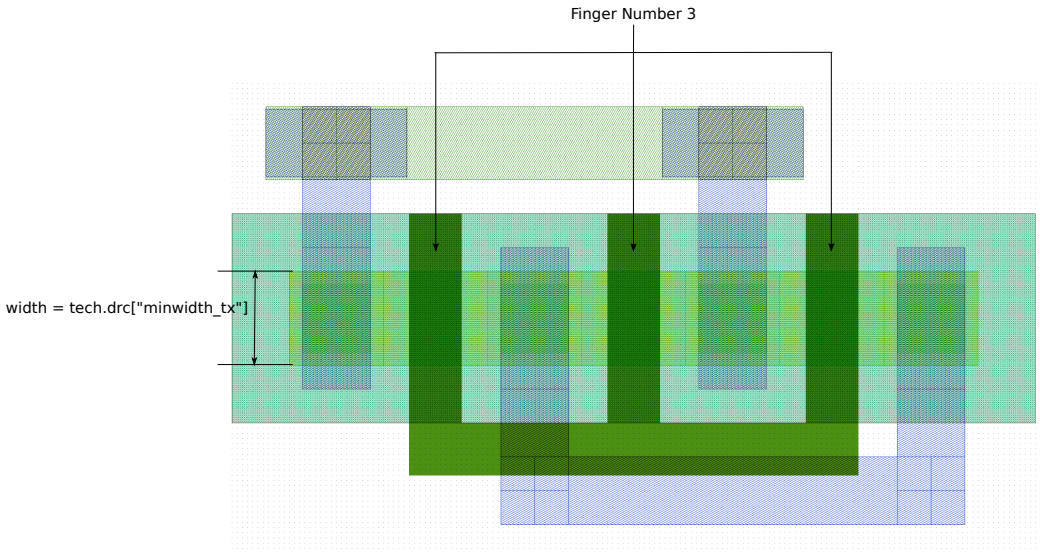


Figure 26: An example of Parameterized Transistor (ptx)

drive strength of the inverter is maintained. The parameterized inverter is utilized in various dynamically generated designs. An example is shown in Figure 27.

Parameterized NAND Gates A third type of parameterized class, `nand_2` and `nand_3`, generates parameterized NAND gates. This class takes the NMOS transistor size and the desired cell height as input. The PMOS size will be sized accordingly to have equal rising and falling transitions for the output. This class will use the parameterized instances of ptx to construct the NAND gates. Examples are shown in Figure 28 and Figure 29.

Parameterized NOR Gates Similar to the parameterized NAND gates, we have another class called `nor_2`, which generates a parameterized 2-input NOR gate. Like the NAND gate's implementation, this class also takes in the NMOS transistor size as well as the desired cell height as input. To ensure that the gate have equal rising and falling transition times, we need to use 2x the size of the NMOS transistor for the PMOS transistor. An example is shown in Figure 30.

Bit-cell and Bit-cell Array In OpenRAM, the 6T cell layout and Spice netlist are provided as library cells (See Figure 3 for layout). The memory cell is a library cell for various reasons. First, it allows the user to easily swap in different memory cell designs. Second, this cell should always be custom to minimize area and optimize performance because it is the most replicated cell in the SRAM. Lastly, the transistors in the cell must be carefully sized to allow for correct read and write operations as well as provide protection against

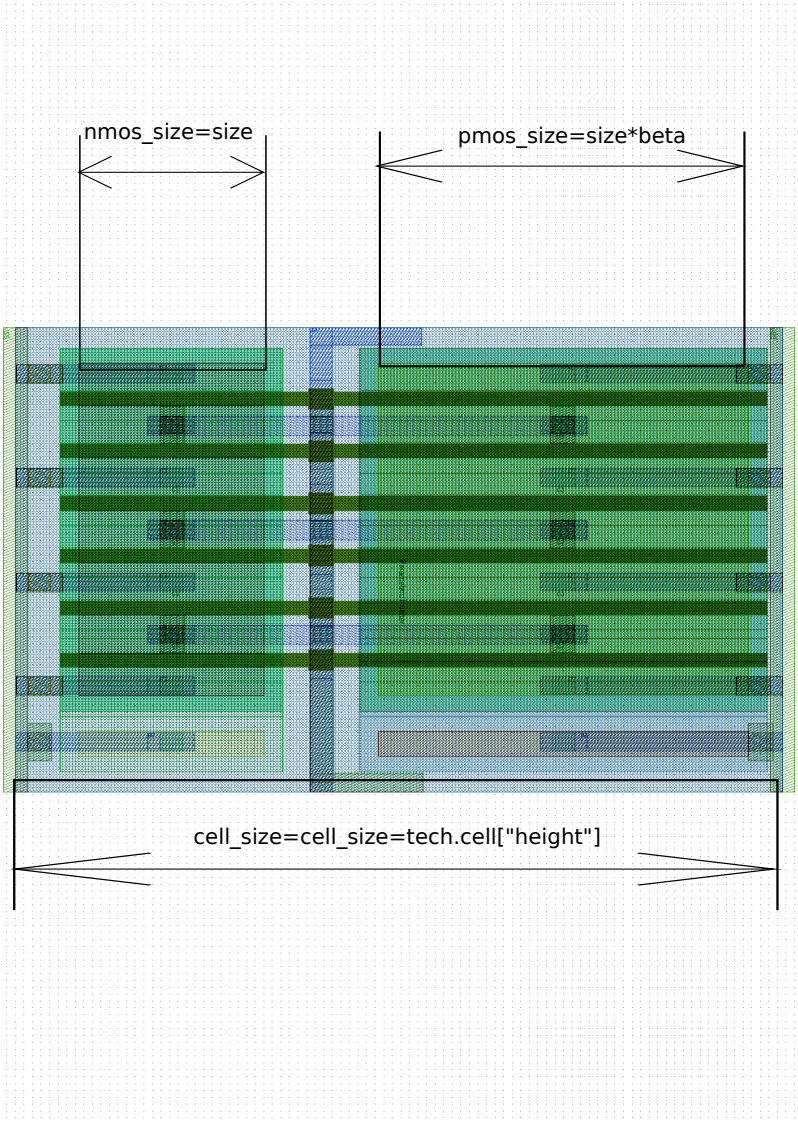


Figure 27: An example of Parameterized Inverter (pinv)

corruption.The `bit-cell` class instantiates a single memory cell. The `bit_cell_array` class dynamically implements the memory array by instantiating a single memory cell and tiling it based on the number of rows and columns. Two simple python "for" loops, one nested in the other, add the instances of the memory cells row by row. During the tiling process, the cells are abutted so that all bit lines and word lines are connected in the vertical and horizontal directions respectively.

Precharge and Precharge Array The precharge circuitry is dynamically generated using the parameterized transistor (`ptx`) and various GdsMill functions used for drawing rectangles and labels in the layout hierarchy. The precharge class dynamically generates

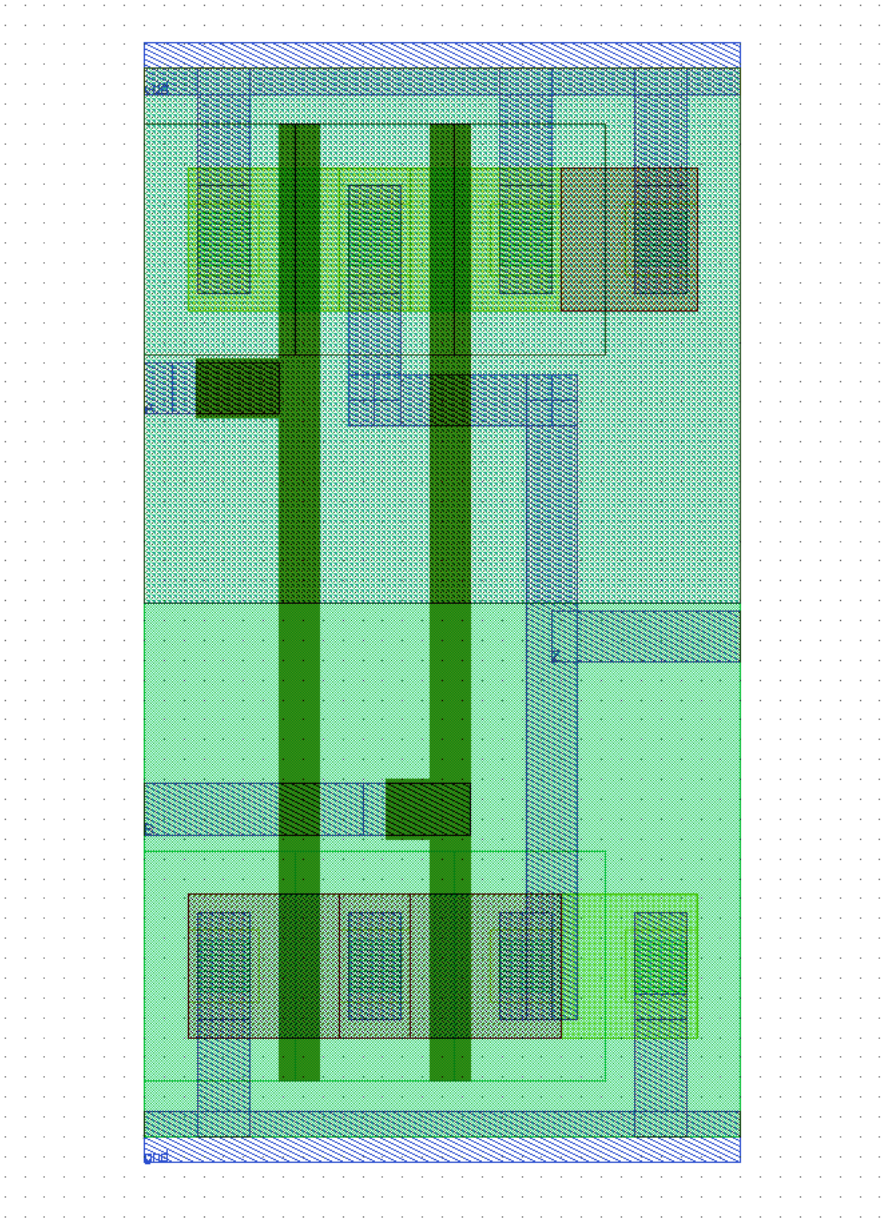


Figure 28: An example of Parameterized NAND2 (nand_2)

a single precharge cell. It adds two instances of `ptx` PMOS transistors, one for the larger PMOS and one for the smaller, equalizing PMOS. These transistor sizes are input parameters to the initialization function. The offsets of the bit lines and the width of the precharge cell must be equal to those of the 6T cell so that the bit lines are correctly connected down to the 6T cell. The `precharge_array` class is then used to generate a precharge array, which is a single row of `n` precharge cells, where `n` equals the number of columns in the bit-cell array.

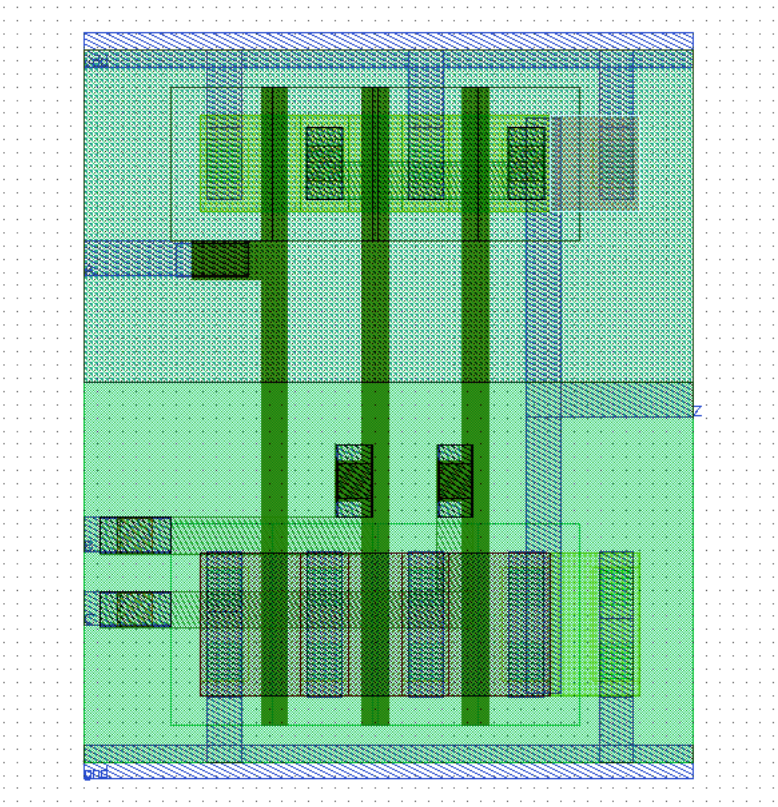


Figure 29: An example of Parameterized NAND3 (nand_3)

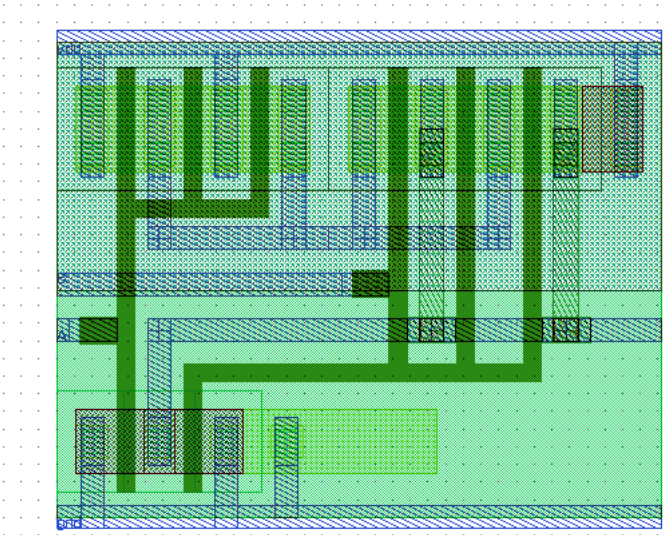


Figure 30: An example of Parameterized NOR2 (nor_2)

Address Decoder and Word-line Drivers The address decoder is dynamically generated by taking in the user's input. For the hierarchical decoder, it takes in the size of the NMOS for the nand2 gate and nand3 gate. In addition, it also accepts the number of rows

required for the decoder. The `hierarchical_decoder` class uses those input parameters to generate the parameterized nand2 and nand3 gates for the construction of the hierarchical decoder. First, it creates the an initial stage for the decoder. Depending on the number of rows, it may use multiple stages of the initial decoder. This is why it is called hierarchical. To ensure proper functionality and drive strength of the signal, word-line drivers are added between the hierarchical decoder and the bit-cell array. There is a `wordline_driver` class that uses parameterized nand2 gates and inverters to generate the column of word-line drivers.

OpenRAM also has the option to use an NAND decoder. It is dynamically generated and takes the NMOS and PMOS sizes as inputs to the initialization method of the `nand_decoder` class. Based on the row address bus size, instances of `ptx` NMOS transistors are then used for each row. The source of the first NMOS in each row is always connected to a ground rail. The gate inputs of the NMOS are then connected (in the binary pattern) to the vertical address input rails. Next, the drain nets are connected to the source nets of the next PMOS in the row. A single PMOS `ptx` instance, used for precharging, is then added to each row and is connected to the `Vdd` rail and the last NMOS in the row. Finally, an inverter is placed at the end of each row using the parameterized inverter class (pinv). This inverter also acts as the word line driver and can be sized accordingly.

Column Mux and Column Mux Array In OpenRAM, the column mux is a dynamically generated design. There are two different flavors that we provide, the tree column mux and the single-level column mux. The `tree_mux` class generates a single cell that mux between two inputs to a single output. It uses parameterized NMOS transistors to control control which input to pass to the output. The `tree_mux_array` class creates instances of the `tree_mux` cell and place them accordingly to hierarchically create the tree column mux depending on the type of mux the memory requires.

Another option is the single-level column mux. The `single_level_column_mux` class generates a simple two to one mux cell, similar to the one generated by the `tree_mux` class. It uses the parameterized `ptx` class to generate two NMOS transistors which will connect the BL and BLB of the selected columns to the sense-amp and write-drivers. Horizontal rails are added for the `sel` signals. The `single_level_column_mux_array` class uses the dynamically generated cell to create the required number of muxes depending on the word size.

Sense Amp and Sense Amp Array In OpenRAM, the sense amplifier is a library cell because it is a carefully designed analog circuit. The `sense_amp` class instantiates a single instance of the sense amp library cell. The `sense_amp_array` class handles the tiling of the sense amps cells. One sense amp cell is needed per data bit and the cells need to be appropriately spaced so that they can hook up to the column mux bit line pairs. The spacing is determined based on the word size and the number of words per row in the bit-cell array.

Write Driver and Write Driver Array Currently, in OpenRAM, the write driver is a library cell and the `write_driver_array` class tiles the write driver cells. Similar to the sense amp, one driver cell is needed per data bit. It is not optimal to have the write driver as a library cell because the driver needs to be sized based on the capacitance of the bit lines. A large memory array needs a stronger driver to drive the data values into the memory cells.

Flip-Flop Array In FreePDK45, we provide a library cell for a simple master-slave d flip-flop. In our library cell we provide both `Q` and `Q_bar` as outputs of the dffs because inverted signals are used in various modules. The `ms_flop` class instantiates a single master-slave flop cell, and the `ms_flop_array` class generates an array of flip-flops. Arrays of dffs are necessary for the incoming data as well as the address bus because we need these values to stay valid for the entire clock cycle. The `ms_flop_array` class takes the bus size and the type of array as inputs and dynamically tiles the flip-flops.

Tristate Array The `tristate_array` class dynamically generates an array of tristate gates used to output data onto the `DATA` bus. A single tristate is instantiated using a user-designed library cell. The `tri_gate_array` class uses that cell to generate an array with length equal to the word size by tiling the tristate cells.

Control Logic The control logic module instantiates a `control_logic` class that arranges all of the flip-flops and logic associated with the control signals into a single design. dffs are instantiated for each control signal input. In addition, parameterized NAND, NOR, and inverter cells are used for the control logic. We use a replica bit-line to help generate the control signal for the activation of the sense amps. The RBL is constructed of dummy bit-cells to mimic the capacitance of the bit-lines in the bit-cell array.

Top-Level SRAM Module The top level of the hierarchy is the SRAM module. This module handles the global organization of all sub-modules in the memory. Based on the user

inputs of the word size and number of words, the parameters needed to generate the memory are calculated and passed to the sub-modules. Equations 1 below are used to determine the number of words per row, the aspect ratio of the memory, and `ADDR` bus sizes. Once these values have been calculated, the arrays can be generated and placed in the top level of the hierarchy based on the sizes of the different modules. It is important to note that when considering the organization of the blocks in the memory, DRC rules for minimum spacing of metals, wells, and other layers must be followed.

$$\textbf{num\_words\_per\_bank} = num\_words/num\_banks \tag{1}$$

$$\textbf{num\_bits} = word\_size * num\_words\_per\_bank \tag{2}$$

$$\textbf{area} = cell\_6t\_width * num\_bits \tag{3}$$

$$\textbf{tentative\_num\_cols} = |\sqrt{area}/cell\_6t\_width| \tag{4}$$

if (tentative_num_cols < 1.5∗word_size):

$$\textbf{words\_per\_row} = 1words\_per\_row \tag{5}$$

if (tentative_num_cols > 3∗word_size):

$$\textbf{words\_per\_row} = 4words\_per\_row \tag{6}$$

Otherwise:

$$\textbf{words\_per\_row} = 2words\_per\_row \tag{7}$$

$$\textbf{tentative\_num\_rows} = |num\_bits/(words\_per\_row * word\_size)| \tag{8}$$

$$\textbf{num\_rows} = num\_words\_per\_bank/words\_per\_row \tag{9}$$

$$\textbf{num\_cols} = words\_per\_row * word\_size \tag{10}$$

$$\textbf{col\_addr\_size} = log_2 words\_per\_row \tag{11}$$

$$\textbf{row\_addr\_size} = log_2 num\_rows \tag{12}$$

$$\textbf{bank\_addr\_size} = col\_addr\_size + row\_addr\_size \tag{13}$$

$$\textbf{addr\_size} = bank\_addr\_size + |log_2 num\_banks| \tag{14}$$

4.3 Physical Verification

OpenRAM interfaces with Calibre nmDRC and nmLVS to perform physical verification of generated designs. Calibre, a Mentor Graphics tool, has two main functions. The DRC, or design rule check, function uses pattern matching algorithms to ensure that all process design rules have been met so that the circuit can be properly fabricated. The LVS, or layout versus schematic, function provides a comparison of the physical layout to the schematic, or Spice netlist, to ensure that the number of devices and connectivity of those devices match [16]. The compiler can also interface with other physical verification tools. There is a wrapper function that can be edited to call other verification tools.

In OpenRAM, DRC and LVS can be performed at any level of the design hierarchy: cell level, module level, and the top-level SRAM design. The compiler has a built-in `DRC_LVS()` function that uses the design hierarchy functions, `sp_write()` and `gds_write()`, to generate the Spice netlist and GDSII layout. The `DRC_LVS()` function then calls the `run_drc()` and `run_lvs()` functions, which prepare the Calibre runset files and perform the physical verification checks in batch mode. When the checks have been completed, the output files are then parsed to determine if there are any errors.

4.4 Memory Characterizer

The memory characterizer is a set of Python scripts that produces the timing and power characteristics of OpenRAM-generated memories through extensive Spice simulations. The characterizer has three main stages: generating the Spice stimulus and test structures, simulating the circuits, and parsing the simulator output. The characterizer can utilizes the industry's HSPICE circuit simulator from Synopsys [48]. In addition, OpenRAM will also utilize the free version, NGSPICE circuit simulator as well. An in depth description and results form the characterizer are provided in Section 5.2.

5 Contributions

This section provides details of the author's significant contributions to the OpenRAM project. The list below outlines the contributions that have been discussed in previous sections.

1. **Component Attributes** - Reorganized all the memory modules' attributes into a dictionary for a cleaner and easier method to access a component's attribute.

2. **Parameterized Transistor** - Rewrote the design `ptx.py` file to generate a more efficient parameterized transistor.

3. **Parameterized Inverter** - Rewrote the design `pinv.py` file to generate a more efficient parameterized inverter.

4. **Unit and Regression Tests** - Ported to git and managed the unit and regression tests for OpenRAM.

5. **Geometry Rotation Issue** - Fixed geometry rotation issue in GDSMill.

6. **Setup-scripts for Paths** - Wrote setup scripts to be dynamically-imported for easier path dependencies.

7. **Dynamically Generated Modules** - Improved/implemented the following dynamically generated modules:

   (a) Contact and Via.

   (b) Wires.

   (c) Precharge and Precharge Array.

   (d) NAND Decoder Row and NAND Decoder Array.

   (e) Tree Column-mux and Column-mux Array.

   (f) Sense Amplifier Array.

   (g) Write Driver Array.

   (h) Master-Slave flip-flop Array.

5.1 Unit/Regression Tests

Another feature of OpenRAM is the set of unit/regression tests. These tests has been ported to be compatible with GIT. They are run on a schedule that is controlled via a cron job. The cron job invokes a Python script to sync all the files from GIT to a temp directory on our server. Using the synced OpenRAM files, the script will run all the tests files listed in the tests directory. The results of the test runs will be emailed to the users. This set of tests is implemented with the Python unit test framework and allow users to add features and custom code without worrying about the breaking of functionality. These set of tests also guide users when porting to new technologies. Every sub-module has its own regression test in addition to the top-level regression test for the whole memory. Lastly, there are

also tests for the library cell verification, timing verification, and technology verification. Each of these tests will initialize their respectively modules and verify the designs with DRC and LVS. These unit tests can be run in any technology because OpenRAM is technology independent.

Currently, we have numerous tests setup that is running on an interval. These include the testing of whole SRAM integration functionality, the testing of each component's functionality, the validation of library cells, and the testing of the timing values for our SRAMS. Each unit test uses assertions to track failures, errors, and successful runs by using:

```
self.assertFalse(calibre.run_drc(a.cell_name,tempgds))
self.assertFalse(calibre.run_lvs(a.cell_name,tempgds,tempspice))
```

For the timing regression tests, we use this assertion to track its failures:

```
self.assertTrue(timing.timing_main(params, probe_address, probe_data))
```

The timing scripts will return a non-None object if it was successful in finding a functional minimum clock period, otherwise a None object will be returned and trigger the assertion to fail. Each of the tests assertion will trigger a test failure when there is a problem with the module. If there are problems due to syntax errors, the tests will fail and result in an error. These unit tests can be run individually or they can all be run by using `regress.py`. This script file will run all the listed unit tests. If there are any failures due to DRC or LVS violations, the tests will generate the summary, output, and error files. These files will be saved to the technology directory's "openram_temp" folder by default unless an output directory is specified upon running the tests. Users would view these files to determine the cause of the failures.

5.2 Memory Characterizer

OpenRAM includes a Memory Characterizer that measures the timing and power characteristics through Spice simulations. It is a set of Python scripts that utilize a Spice circuit simulator in order to produce timing and power numbers for OpenRAM generated memory. The characterizer performs four main stages for delay and power: creating the Spice stimulus, running the circuit simulations, parsing the simulator's output, and producing the characteristics in a Liberty (.lib) file. In addition to finding the delay and power characteristics, the characterizer also calculates the setup and hold times for the memory's inputs. The characterizer will only use and access the interface of the memory (e.g., bi-directional

data bus, address bus, and control signals) to perform "black box" timing measurements. Users can modify the technology's parameters file `tech.py` to provide user-defined variables such as supply voltage, rise and fall times.

5.2.1 Spice Stimulus

The stimulus is written in standard Spice format and can be used with any simulator that support this [1]. Various methods are used to create the Spice stimulus file. These methods, within `stimuli.py`, include the creation of buffers, transmission gates, voltage sources, etc. For OpenRAM-generated memories with bi-directional `DATA` buses, input signal buffers and transmission gates are necessary for simulation purposes. When voltage sources are defined in Spice, they are ideal sources with infinite drive strength. To ensure accurate results, all input signals must be buffered so that a realistic signal strength is used. Also, transmission gates are used for the bi-directional `DATA` bus. This allows for the input stimulus to be disconnected from the `DATA` bus while a read operation is active. The following methods can be used to generate stimulus:

- `inst_sram()` - adds the instance of the sram to the stimulus file with all its ports.
- `inst_model()` - adds the instance of a component and its ports; such as ms-flip-flop.
- `create_inverter()` - creates a min-sized inverter.
- `create_buffer()` - creates a buffer(2 inverters) given a size.
- `add_inverter()` - adds inverters to the stimulus given a list of signals.
- `add_buffer()` - adds buffers to the stimulus given a list of signals.
- `add_accesstx()` - adds transmission gates to the stimulus given a list of data-signals.
- `gen_clk_pwl()` - generates a piece-wise linear function for the `CLK` for the min-period Algorithm.
- `gen_data_pwl()` - generates a piece-wise linear function for the `DATA` stimulus.
- `gen_addr_pwl()` - generates a piece-wise linear function for the `ADDR` stimulus.
- `gen_csb_pwl()` - generates a piece-wise linear function for the `CSb` control stimulus.
- `gen_web_pwl()` - generates a piece-wise linear function for the `WEb` control stimulus.

- `gen_oeb_pwl()` - generates a piece-wise linear function for the `OEb` control stimulus.
- `gen_constant()` - generates a constant voltage source.
- `gen_meas_delay()` - generates a statement to measure delay for a specific `DATA` port.
- `gen_meas_power()` - generates a statement to measure average power for a specified time period.
- `write_include()` - writes include statements for transistor models and any components needed.
- `write_supply()` - writes the `Vdd` and `Gnd` supply stimulus.
- `obtain_key_times()` - returns all the positive edge times for the generated `CLK`.

5.2.2 Binary Search Delay Algorithm

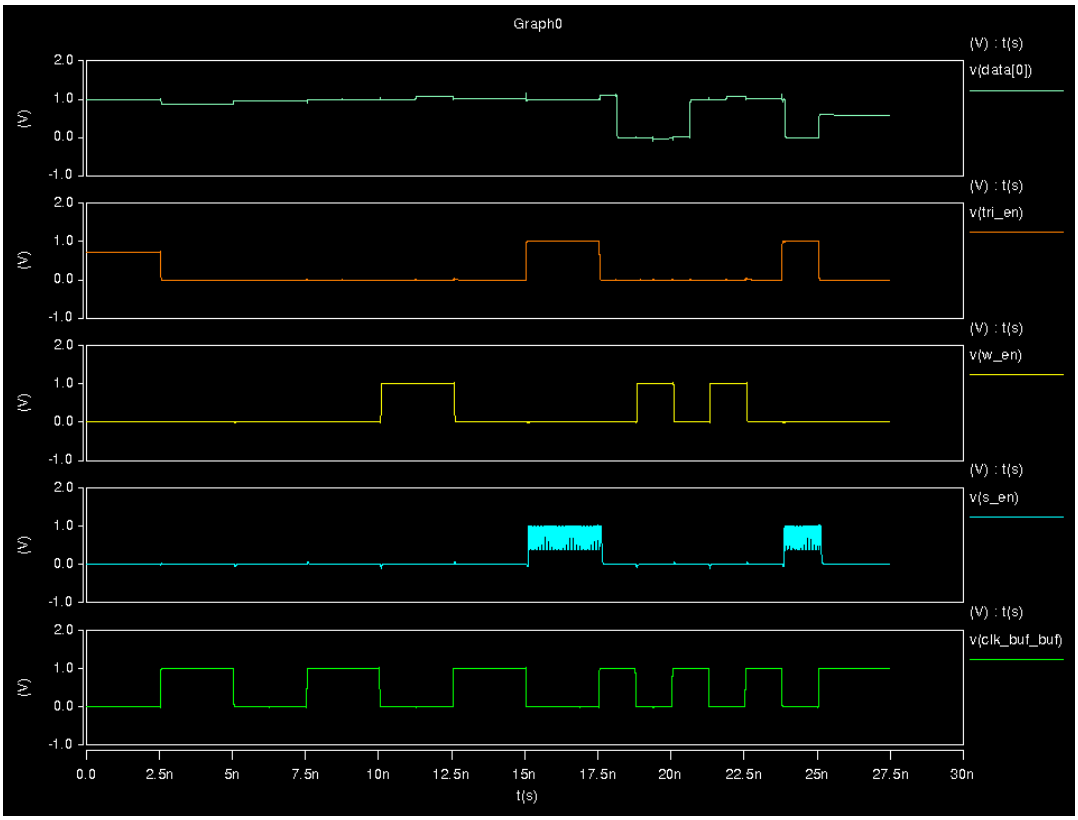


Figure 31: A timing diagram showing the 6 clock cycle scheme to finding min-delay (max frequency).

For the read and write delay simulations, a file called `stim.sp` was written; the file contains the test structures and stimulus for all SRAM inputs. Spice's `.measure` statements

were used to allow the Spice circuit simulator to measure the timing values. The Spice simulator is then called and the simulation is run. A binary search algorithm is used to find the minimum clock period (max frequency). The algorithm is as follows:

- Given a reference clock period in the `tech.py` file, the algorithm checks if the listed value is a feasible clock period. This is done by running a simulation for both low-to-high and high-to-low transitions. If it is not feasible, the algorithm continues to double the value until a feasible reference clock period is found (with a time-out of 8 iterations).
- With a working reference clock period, the binary search algorithm will begin its iterations to find a target clock period that has its delay within 2% of the previous feasible clock period's delay.
- Once the clock period value is within our error tolerance, the algorithm found the minimum clock period and the read delay.

As shown in Figure 31, the duration of each simulation is 6 clock cycles; the first three clock cycles are based on an already verified clock speed and the next three clock cycles are used to verify an unverified clock speed. The first three clock cycles are also used to set up for the verification of the next three clock cycles. The first clock cycle is a dummy cycle to let every component stabilize before performing any operations. During the second clock cycle, a write operation is done to drive a `data_value` into a specified bit-cell, determined by the address provided. During the third clock cycle, a read operation is performed on the same address to verified that the stored `data_value` is correct.

Since the stored `data_value` in the memory is now known, the next three clock cycles are used to verified the target clock speed. During the fourth clock cycle, the opposite `data_value` is written into the address specific bit-cell. Since OpenRAM uses a Bi-directional `DATA` bus, during the fifth clock cycle, the `data_value` on the `DATA` bus will still be the same `data_value` as the one during the fourth clock cycle. So the fifth clock cycle is used to clear the `data_value` on the `DATA` bus by using a write operation of the opposite `data_value`. The other difference is that the address is now inverted to avoid overwriting our fourth clock cycle's write operation. Last but not least, the sixth clock cycle is used for a read operation from the initial address to verified the `data_value` that was written during the fourth clock cycle. If the write or read operation of the unverified clock speed fails, the

clock speed is deem unusable for this specific design. These simulations will continue to run until two back-to-back successful simulations are found.

5.2.3 Delay

The delay measures the time elapsed from the positive clock edge until there is valid data on the `DATA` bus. The delay is calculated for both high-to-low and low-to-high transitions. After the simulations have completed, the Spice output file is then parsed and the measurement is extracted and formatted into a Liberty (.lib) file.

5.2.4 Setup and Hold Time

The setup time is defined as the time that a signal must be held valid before the clock edge to ensure proper operation. The hold time is defined as the time that an input signal must be held valid after the clock edge to ensure proper operation. In a synchronous SRAM, all input signals are registered using DFFs. This means that the setup and hold times for the SRAM are the same as the setup and hold times of the DFFs. In order to find the setup and hold time of the d-flip-flop, a bisection method was utilized. The bisection method uses a binary search algorithm to optimize a variable that is dependent on a goal value and a error tolerance. The bisection method will require a reference point to begin. The same reference clock period provided in the `tech.py` file will be used. The setup and hold times are calculated for both the low-high and high-low data transitions. The Bisection method is as follows:

- Similar to the binary search delay algorithm, this algorithm starts by verifying if the given clock period is feasible. If it is not, the algorithm tries to find a feasible clock period by doubling the value (with a time-out of 8 iterations).
- With a working reference clock period, the binary search algorithm begins to find a target clock period that is within 0.01% percent error of its previous clock period.
- A clock period is valid if the output voltage swing is within 10% of the target `data_value`. For example, if the `data_value` is 1 Volt, then a valid output is between 0.9 Volt and higher; vice-versa for the opposite `data_value`.
- Once the voltage value is within the error tolerance and their `data_value` is valid, the algorithm calculates the setup and hold time between the clock edge and when the data value changes.

5.2.5 Power

The average power for both the read and write operations is measured by the Spice simulator during the delay simulations. Spice `.measure` statements measure the average power. The power is measured over the entire clock period so that the power from all circuits utilized during an operation are considered. These values are extracted from the Spice output files and formatted into a Liberty (.lib) file.

6 Results

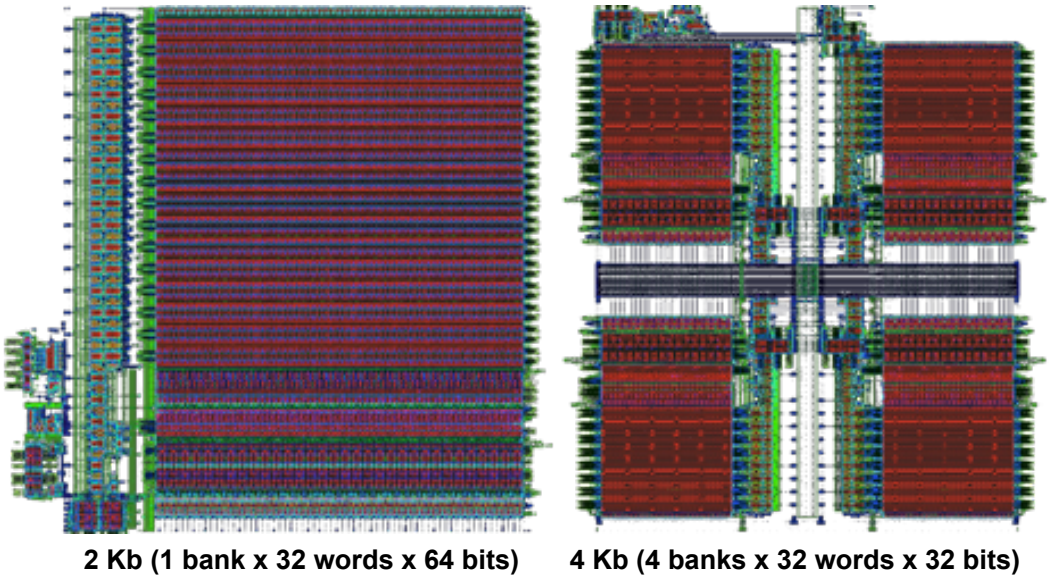
Figure 32 shows several different SRAM layouts generated by OpenRAM in FreePDK45. OpenRAM can generate both single and multi bank SRAM arrays. Banks are symmetrically placed around the control logic so that they all have the same delay for data, address and control signals.

Figure 33 and Figure 34 shows the memory area of different total size and data word width in FreePDK45 and SCMOS, respectively. As expected, the smaller process technology (45nm) has lower total area overall but the trends are similar in both technologies.

Figure 35 and Figure 36 shows the access time of different size and data word width in FreePDK45 and SCMOS, respectively. Increasing the memory size generally increases the access time; long bit-line and word-line increase the access time by adding more parasitic capacitance and resistance, therefore having shorter bit-line and word-line helps to speed up the read operation. Since OpenRAM uses multiple banks and column muxing, it is possible to have a smaller access time for larger memory designs, but this will sacrifice density.

Table 5 compares the bit-density of OpenRAM against published designs using similar technology nodes. The results show the benefit of technology scaling and that OpenRAM has very good density in both our generic FreePDK45 and SCMOS technologies.

Table 6 shows multiple technologies with their access time and power consumption. Comparison of read access time and power consumption are a bit more complicated to make a conclusion because there are many trade-offs. Power and performance are highly dependent on circuit style (CMOS, ECL, etc.), memory organization (more banks reduces delay but sacrifices density), and the optimization goals: low-power or high-performance. In general, OpenRAM has reasonable trade-off between the two and can be customized by using an alternate sense amplifier or structure of the memory. As a comparison, a 75ns SRAM consumes 3.9mW [43] while SCMOS, generated by OpenRAM, is much faster at



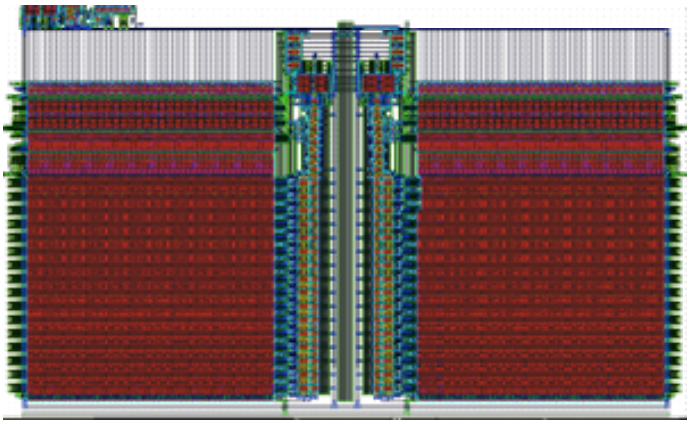


Figure 32: Single and multi bank SRAMs with symmetrical placement of bank to equalize the signal delays.

44.9ns but consumes 115mW for the same size.

As previously stated, the characterizer also reports the setup and hold time of the d-flip-flop. Table 7 shows the results of the characterization of the master-slave d-flip-flop. These results are somewhat optimistic because an un-loaded flip-flop is simulated; the load that the d-flip-flop drives will inevitably have an effect of the delay. The results obtained for the SRAM simulation are from the pre-back annotation characterizer.

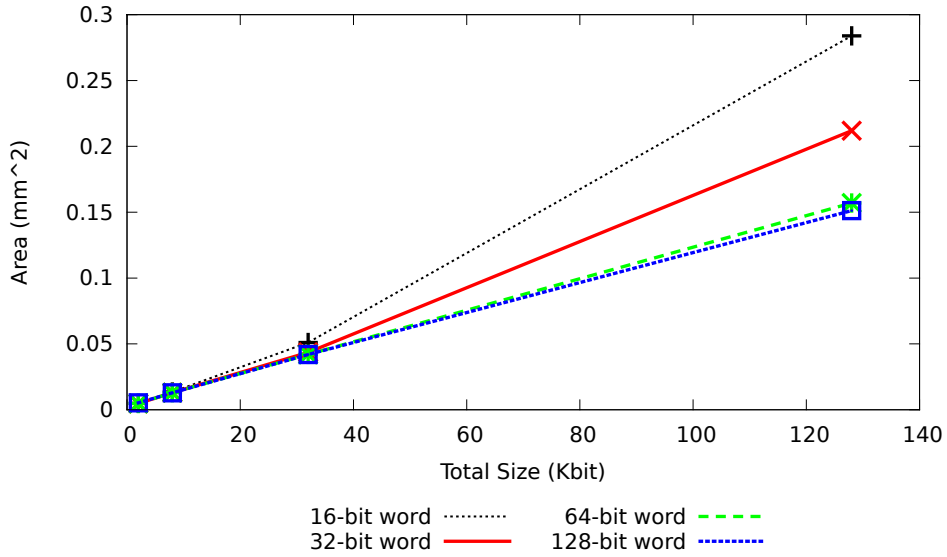


Figure 33: OpenRAM generated FreePDK45's memory area

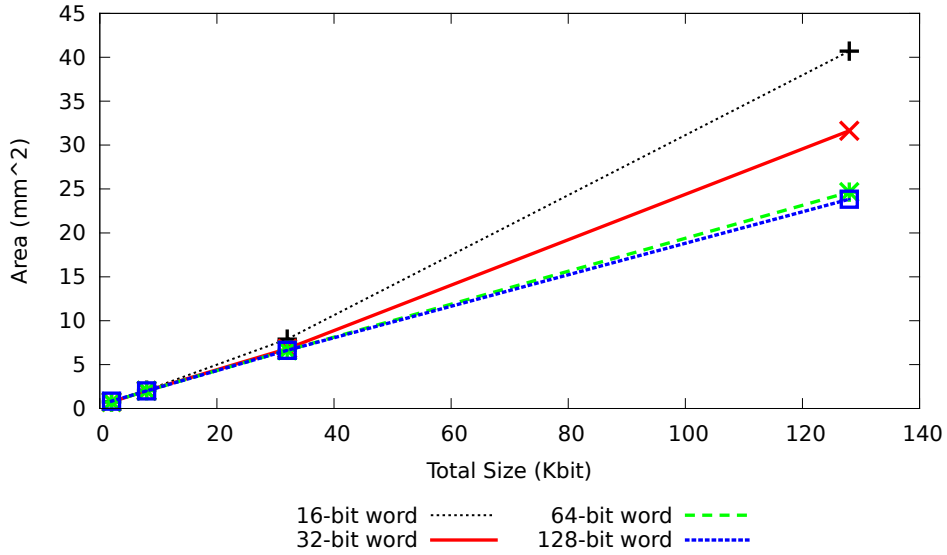


Figure 34: OpenRAM generated SCMOS's memory area

Ref.	Feature Size	Tech.	Density [Mb/mm^2]
[31]	65 nm	CMOS	0.7700
[53]	45 nm	CMOS	0.3300
[35]	40 nm	CMOS	0.9400
OpenRAM	45 nm	FreePDK45	0.8260
[62]	0.5 um	CMOS	0.0036
[50]	0.5 um	BiCMOS	0.0020
[43]	0.5 um	CMOS	0.0050
OpenRAM	0.5 um	SCMOS	0.0050

Table 5: OpenRAM has high density compared to other published memories in similar technologies.

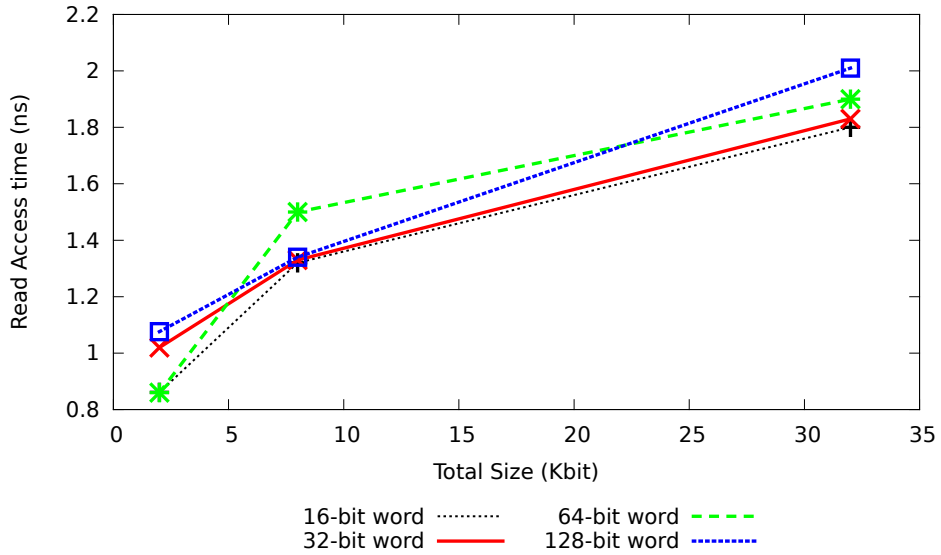


Figure 35: OpenRAM generated FreePDK45's access time

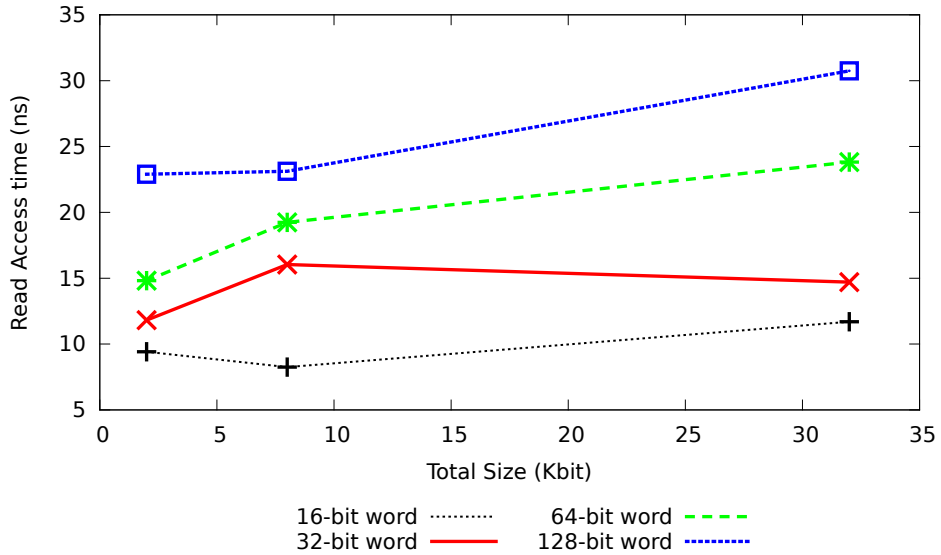


Figure 36: OpenRAM generated SCMOS's access time

Reference	Technology	Access time (ns)	Power consumption
[30]	$65nmCMOS$	28	22 uW/MHz
[35]	$40nmCMOS$	45	13.8 $pJ/access/Mbit$
OpenRAM	$45nmFreePDK45$	9.86	13.14 mW
[61]	$0.5umCMOS$	1.5	6 W
[50]	$0.5umBiCMOS$	1.5	35 W
[43]	$0.5umCMOS$	75	3.9 mW
OpenRAM	$0.5umSCMOS$	44.9	115 mW

Table 6: OpenRAM has fast access time and low power consumption compared to other published memories in similar technologies.

D-Flip-Flop Characterization			
Setup (ps)		Hold (ps)	
low-high	high-low	low-high	high-low
13.300	11.200	-12.800	-10.000

Table 7: The setup and hold times for the master-slave DFF for our FreePDK45 technology.

7 Conclusion

Embedded memories, specifically SRAM's, account for a significant portion of a chip's total performance, power, and area. This trend, and the importance of the SRAM design will continue to be of high interest over time. Due to this, it is necessary to have an easy way to test and prototype memory designs. Since SRAMs have a very regular structure, it can be exploited by design automation tools. Memory compilers are not a new concept; many exist as commercial products and intellectual property but few are open-source and modifiable. In this thesis, an open-source memory compiler, OpenRAM, has been introduced to aid in the memory design process. The main motivation behind the OpenRAM project is to promote and simplify memory research in academia. OpenRAM is meant to be an open-source, flexible, and portable tool that can be used to generate the circuit, functional model, and layout of variable-sized SRAMs across many technologies. In addition, a memory characterizer provides synthesis timing and power models.

OpenRAM is implemented in Python, using the object-oriented paradigm. It utilizes a Python-GDSII interface, called GdsMill, and a set of data structures to construct a hierarchical representation of a GDSII layout and a corresponding Spice netlist. Modules, or blocks of the SRAM, are dynamically generated by code or through the use of user designed library cells. They are added to the design hierarchy. Once the design hierarchy has been populated, a GDSII file and Spice netlist of the SRAM are written as outputs.

In addition to the compiler, a memory characterization methodology is provided. The memory characterizer uses the OpenRAM generated Spice netlist and a Spice simulator to produce the timing and power characteristics of the SRAM. The characterizer writes Spice stimulus files and invokes a Spice simulator to measure the read and write delays, average power, as well as the setup/hold times. Lastly, the characterizer was run on several OpenRAM generated memories and the results were reported.

OpenRAM provides a simple way of generating memories quickly which can be utilized in any SOC, ASIC, or microprocessor design. It also provides a platform to implement and test new memory cells and sub-circuits without considerable overhead. Designs are currently being fabricated to test OpenRAM's generated memory designs in SCMOS. We are also continuously introducing new features, such as non-6T bit-cells, variability characterization, word-line segmenting, characterization speed-up, a Graphical User Interface (GUI), and overall code efficiency. There are still many work to be done in the future for this project. These future-work include the dynamic generation and sizing of write drivers, the addition

of multi-port RAMs and register files, the modification to include asynchronous SRAMs, and the completion of the "back-end". With the "back-end", OpenRAM would be able to use back-annotation and extracted parasitics for detailed characterization of the generated memory. Lastly, when the code is released to the public as open-source, we hope to engage an active community for the future development and improvement of OpenRAM.

References

[1] Process assessment. ISO/IEC 15504, 2004.

[2] B.S. Amrutur and M.A. Horowitz. Fast low-power decoders for RAMs. *Solid-State Circuits, IEEE Journal on*, 36(10):1506–1515, Oct 2001.

[3] ARM. Embedded memory IP. `http://www.arm.com/products/physical-ip/embedded-memory-ip/index.php`, 2015.

[4] P. Athe and S. Dasgupta. A comparative study of 6t, 8t and 9t decanano SRAM cell. In *Industrial Electronics Applications, 2009. ISIEA 2009. IEEE Symposium on*, volume 2, pages 889–894, 2009.

[5] S. Baeg, S. Wen, and R. Wong. SRAM interleaving distance selection with a soft error failure model. *Nuclear Science, IEEE Transactions on*, 56(4):2111–2118, Aug. 2009.

[6] Mark A. Horowitz Bharadwaj S. Amrutur. A replica technique for wordline and sense control in low-power SRAM's. *JSSC*, 33(8):1208–1219, Aug 1998.

[7] R. Broderson. *Anatomy of a Silicon Compiler*. Springer, 1992.

[8] J. Butera. OpenRAM: An open-source memory compiler. Master's thesis, University of California - Santa Cruz, 2013.

[9] AC Cabe, Z Qi, W Huang, Y Zhang, MR Stan, and GS Rose. A flexible, technology adaptive memory generation tool. *Cadence CDNLive*, 2006.

[10] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *Nuclear Science, IEEE Transactions on*, 43(6):2874–2878, 1996.

[11] A. Chandrakasan, W.J. Bowhill, and F. Fox. *Design of High Performance Microprocessor Circuits*. IEEE Press, 2001.

[12] International Technology Roadmap for Semiconductors. 2012 ITRS report: System drivers. 2012.

[13] Global Foundries. Memory IP. `http://www.globalfoundries.com/design/memory_ip.aspx`, 2015.

[14] R. Goldman, K. Bartleson, T. Wood, V. Melikyan, and E. Babayan. Synopsys' educational generic memory compiler. In *Microelectronics Education (EWME), 10th European Workshop on*, pages 89–92, May 2014.

[15] M. Goudarzi and T. Ishihara. SRAM leakage reduction by row/column redundancy under random within-die delay variation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(12):1660–1671, 2010.

[16] Mentor Graphics. Calibre nmdrc and nmlvs. `http://www.mentor.com/products/ic_nanometer_design/verification-signoff/physical-verification/`, 2013.

[17] S. Hanson, M. Seok, D. Sylvester, and D. Blaauw. Nanometer device scaling in subthreshold logic and SRAM. *IEEE Transactions on Electron Devices*, 55(1):175–185, 2008.

[18] T.-H. Huang, C.-M. Liu, and C.-W. Jen. A high-level synthesizer for VLSI array architectures dedicated to digital signal processing. In *International Conference on Acoustics, Speech and Signal Processing*, pages 1221–1224, 1991.

[19] IBM. Understanding static RAM operation. Technical report, Mar 1997.

[20] IDT. A comparison of zero bus turn-around (ZBT) SRAMS and late write SRAMS. Technical report, 1996.

[21] K. Itoh. Trends in megabit DRAM circuit design. *Solid-State Circuits,IEEE Journal on*, 25(3):778–798, Jun 1990.

[22] K. Itoh. *VLSI Memory Chip Design*. Springer-Verlag, 2001.

[23] D. Jahannsen. Bristle blocks: A silicon compiler. In *Design Automation Conference (DAC)*, pages 195–198, 1979.

[24] I. Jung, Y. Kim, and F. Lombardi. A novel sort error hardened 10t SRAM cells for low voltage operation. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 714–717, 2012.

[25] S. Kim and M. Guthaus. Leakage-aware redundancy for reliable sub-threshold memories. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 435–440, 2011.

[26] S. Kim and M. Guthaus. Low-power multiple-bit upset tolerant memory optimization. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 577–581, 2011.

[27] S. Kim and M. Guthaus. SNM-aware power reduction and reliability improvement in 45nm SRAMs. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 204–207, 2011.

[28] S. Kim and M. Guthaus. Dynamic voltage scaling for SEU-tolerance in low-power memories. In *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, pages 207–212, 2012.

[29] F.J. Kurdahi, A.M. Eltawil, Y.H. Park, R.N Kanj, and S.R. Nassif. System-level SRAM yield enhancement. *Quality Electronic Design, 7th International Symposium on*, Mar 2006.

[30] K. Kushida et al. A 0.7 V single-supply SRAM with 0.495 um^2 cell in 65 nm technology utilizing self-write-back sense amplifier and cascaded bit line scheme. 44(4):1192–1198, Apr 2009.

[31] K. Kushida, A. Suzuki, G. Fukano, A. Kawasumi, O. Hirabayashi, Y. Takeyama, T. Sasaki, A. Katayama, Y. Fujimura, and T. Yabe. A 0.7v single-supply SRAM with 0.495 um^2 cell in 65nm technology utilizing self-write-back sense amplifier and cascaded bit line scheme. In *IEEE Symposium on VLSI Circuits*, pages 46–47, June 2008.

[32] Virage Logic. SiWare memory. `http://www.viragelogic.com`, 2015.

[33] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, Jan 1979.

[34] Chen Ming and Bai Na. An efficient and flexible embedded memory ip compiler. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, pages 268–273, Oct 2012.

[35] Sh. Miyano et al. Highly energy-efficient SRAM with hierarchical bit line charge-sharing method using non-selected bit line charges. 48(4):924–931, Apr 2013.

[36] P. Poechmueller, G. K. Sharma, and M. Glesner. A CAD tool for designing large, fault-tolerant VLSI arrays. In *Great Lakes Symposium on VLSI (GLSVLSI)*, 1991.

[37] R.P. Preston. *Register Files and Caches.* In [11], 2001.

[38] Python. The python programming language. `http://www.python.org`, 2013.

[39] J. Rabaey, A. Chandrakasan, and B. Nikoli. *Digital Integrated Circuits: A Design Perspective*. Pearson Education, Inc., 2nd edition, 2003.

[40] W.M. Regitz and J. Karp. Three-transistor-cell 1024-bit 500-ns MOS RAM. *Solid-State Circuits, IEEE Journal of*, 5(5):181–186, 1970.

[41] S. Rusu, J. Stinson, S. Tam, J. Leung, H. Muljono, and B. Cherkauer. A 1.5-GHz 130-nm itanium reg; 2 processor with 6-MB on-die L3 cache. *Solid-State Circuits, IEEE Journal of*, 38(11):1887–1895, 2003.

[42] T. Shah. Fabmem: A multiportedRAM and CAM compiler for superscalar design space exploration. Master's thesis, North Carolina State University, 2010.

[43] N. Shibata, H. Morimura, and M. Watanabe. A 1-V, 10-MHz, 3.5-mw, 1-Mb MTCMOS SRAM with charge-recycling input/output buffers. 34(6):866–877, Jun 1999.

[44] A. Shimpi. Intel core i7 3960x (sandy bridge) review: Keeping the high-end alive. `http://www.anandtech.com/show/5091/intel-core-i7-3960x-sandy-bridge-e-review-keeping-the-high-end-alive`, Nov. 2011.

[45] OK State. VLSI computer architecture research group at OK state. `http://vlsiarch.ecen.okstate.edu/`, 2016.

[46] J. Stine, I. Castellanos, M. Wood, J. Henson, and F. Love. FreePDK: An open-source variation-aware design kit. *International Conference on Computer-Aided Design*, Jan 2007.

[47] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, Julie Oh, and R. Jenkal. FreePDK: An open-source variation-aware design kit. In *IEEE International Conference on Microelectronic Systems Education*, pages 173–174, June 2007.

[48] Synopsis. Hspice. `http://www.synopsys.com/tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Pages/default.aspx`, 2013.

[49] Synopsys. Designware memory compilers. `http://www.synopsys.com/dw/ipdir.php?ds=dwc_sram_memory_compilers`, 2015.

[50] N. Tamba et al. A 1.5-ns 256-kb BiCMOS SRAM with 60-ps 11-k logic gates. 48(11):1344–1352, Nov 1994.

[51] Faraday Technologies. Memory compiler architecture. http://www.faraday-tech.com/html/Product/IPProduct/LibraryMemoryCompiler/index.htm, 2015.

[52] Dolphin Technology. Memory products. http://www.dolphin-ic.com/memory-products.html, 2015.

[53] S. O. Toh, Zh. Guo, T. K. Liu, and B. Nikolic. Characterization of dynamic SRAM stability in 45 nm CMOS. 46(11):2702–2712, Nov 2011.

[54] Y. Tosaka, S. Satoh, T. Itakura, K. Suzuki, T. Sugii, H. Ehara, and G.A. Woffinden. Cosmic ray neutron-induced soft errors in sub-half micron CMOS circuits. *Electron Device Letters, IEEE Transactions on*, 18(3):99–101, 1997.

[55] UCSC. VLSI-DA group at UCSC. https://vlsida.soe.ucsc.edu/, 2016.

[56] M. Wieckowski. *GDS Mill User Manual*, 2010.

[57] Michael Wieckowski. Gds mill. http://michaelwieckowski.com/?page_id=190, 2010.

[58] Sheng Wu, Xiang Zheng, Zhiqiang Gao, and Xiangqing He. A 65nm embedded low power SRAM compiler. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 123–124, April 2010.

[59] Y Xu, Z Gao, and X He. A flexible embedded SRAM IP compiler. *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3756–3759, 2007.

[60] Yi Xu, Zhiqiang Gao, and Xiangqing He. A flexible embedded SRAM IP compiler. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3756–3759, May 2007.

[61] K. Yamaguchi et al. A 1.5-ns access time, 78- um^2 memory-cell size, 64-kb ECL-CMOS SRAM. 27(2):167–174, Feb 1992.

[62] K. Yamaguchi, H. Nambu, K. Kanetani, Y. Idei, N. Homma, T. Hiramoto, N. Tamba, K. Watanabe, Masanori Odaka, T. Ikeda, K. Ohhata, and Y. Sakurai. A 1.5-ns access time, 78 um^2 memory-cell size, 64-kb ECL-CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 27(2):167–174, Feb 1992.