SRAM COMPILER FOR AUTOMATED MEMORY LAYOUT SUPPORTING MULTIPLE TRANSISTOR PROCESS TECHNOLOGIES

A Thesis
presented to
the Faculty of California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical Engineering

by
Brandon Hilgers
July 2015

COMMITTEE MEMBERSHIP

TITLE: SRAM Compiler For Automated Memory Layout
Supporting Multiple Transistor
Process Technologies

AUTHOR: Brandon Hilgers

DATE SUBMITTED: July 2015

COMMITTEE CHAIR: Tina Smilkstein, Ph.D.
Assistant Professor of Electrical Engineering

COMMITTEE MEMBER: John Oliver, Ph.D.
Associate Professor of Electrical Engineering

COMMITTEE MEMBER: Bridget Benson, Ph.D.
Assistant Professor of Electrical Engineering

ABSTRACT

SRAM Compiler For Automated Memory Layout
Supporting Multiple Transistor
Process Technologies

Brandon Hilgers

This research details the design of an SRAM compiler for quickly creating SRAM blocks for Cal Poly integrated circuit (IC) designs. The compiler generates memory for two process technologies (IBM 180nm cmrf7sf and ON Semiconductor 600nm SCMOS) and requires a minimum number of specifications from the user for ease of use, while still offering the option to customize the performance for speed or area of the generated SRAM cell. By automatically creating SRAM arrays, the compiler saves the user time from having to layout and test memory and allows for quick updates and changes to a design. Memory compilers with various features already exist, but they have several disadvantages. Most memory compilers are expensive, usually only generate memory for one process technology, and don’t allow for user-defined custom SRAM cell optimizations. This free design makes it available for students and institutions that would not be able to afford an industry-made compiler. A compiler that offers multiple process technologies allows for more freedom to design in other processes if needed or desired. An attempt was made for this design to be modular for different process technologies so new processes could be added with ease; however, different process technologies have different DRC rules, making that option very difficult to attain. A customizable SRAM cell based on transistor sizing ratios allows for optimized designs in speed, area, or power, and for academic research. Even for an experienced designer, the layout of a single SRAM cell (1 bit) can take an hour. This command-line-based tool can draw a 1Kb

SRAM block in seconds and a 1Mb SRAM block in about 15 minutes. In addition, this compiler also adds a manually laid out precharge circuit to each of the SRAM columns for an enhanced read operation by ensuring the bit lines have valid logic output values. Finally, an analysis on SRAM cell stability is done for creating a robust cell as the default design for the compiler. The default cell design is verified for stability during read and write operations, and has an area of 14.067 μm^2 for the cmrf7sf process and 246.42 μm^2 for the SCMOS process. All factors considered, this SRAM compiler design overcomes several of the drawbacks of other existing memory compilers.

ACKNOWLEDGMENTS

I would like to thank God, my parents, fiancée, and friends for all of their help, support, and encouragement through this whole process.

I would like to give a special thanks to Dr. Tina Smilkstein, as my advisor, for her technical advice, guidance with the use of the Cadence software, encouragement, and feedback on this project.

My committee members, Dr. John Oliver and Dr. Bridget Benson, have also been helpful in this process. I thank them for all of their feedback and support.

TABLE OF CONTENTS

Section Page

Appendices

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1: Introduction

SRAM is widely used in a variety of applications: CPU caches, hard drive caches, microcontrollers, and more. Its speed and (often) power-efficiency compared to DRAM makes it a great secondary memory (such as a cache). Regardless of what type of memory is used, laying out a large custom-sized memory for a chosen design by hand is a long and tedious process.

In integrated circuit (IC) design, time to market is an important factor that affects the cost to a company. Since memory is used in many of the ICs being produced today, the time needed to develop memory alone greatly intrudes on the time needed to develop the rest of a design. Due to time constraints, industry and academic IC designers will often choose to use a memory compiler to generate the memory they need. A memory compiler allows a designer to automate the process of designing memory for whatever application they need. This is done by specifying the size of memory they need and creating the memory in the fabrication process technology that matches their project design process. Different memory compilers offer various capabilities: different types of memory (6T SRAM, dual-port SRAM, DRAM, SDRAM, etc.), optimized memory (fast memory, high-density memory, low power memory, etc.), and more. Whatever the application, using a memory compiler greatly reduces the development time of a project design.

With all of the benefits that a memory compiler brings, there are several drawbacks. While using a memory compiler saves a lot of time and effort, this comes at the cost of buying an expensive compiler. A 1T-SRAM memory compiler from MoSys goes for $300,000 [1]. For academic purposes, expenses of this amount are too high for most schools, so the following option would be to apply for a discounted school license. However, The MOSIS Service [11], a fabrication service that often provides a limited number of academic-related chips for fabrication, only allows “non-academic users” to request to use their memory generators [4]. Not only that, but if one were to obtain a memory compiler, it is likely that it only functions for one or two fabrication processes, reducing the possibility for design in different transistor feature sizes or with different foundries. Lastly, when obtaining a compiler, there are bound to be very difficult bugs to fix that come when the standard libraries get updated.

The proposed solution to these issues is to design a memory compiler for academic purposes, that supports multiple transistor technologies, is easy to manage, and has minimal to no cost. In addition to these features, an option to choose custom SRAM cell transistor sizes is included for potential optimizations in speed, power, and/or area, and opportunities for users to do academic research. Since the California Polytechnic State University Electrical Engineering Department has a license to the Cadence Virtuoso Schematic and Layout, and Encounter software, the compiler will be designed for free using these tools. Specifically, this compiler will be made using Cadence's proprietary scripting language, SKILL. The transistor technologies available to the department are cmrf7sf (IBM), cmrf8sf (IBM), TSMC, and C5N-SCMOS (On Semiconductor). CMRF7SF and SCMOS are used in a course presently taught at Cal Poly and the compiler is designed to work for the ON Semiconductor 600 nm SCMOS (Scalable CMOS), and IBM 180 nm cmrf7sf process technologies giving users multiple technologies, and therefore, feature sizes to use. When using the compiler, the user specifies the technology, the names of the cell and block designs, the size of the memory block to be created, in the compiler command line function tool. A memory block is created with precharge circuitry to output valid logic values during a read operation. As an option, the user can also specify the SRAM cell transistor sizes to optimize the performance of the cell for speed, area, or power. Additional goals of this design is to make a compiler that is also well-documented and has code that is well-commented for the sake of ease of use and management as well as allowing for improvements and additions in the future. With the given materials and design choices, this memory compiler is a useful tool for academic purposes and for designs that people want to have fabricated, all at a low cost, with easy management, and with a variety of fabrication processes.

This project starts with choosing the default user transistor sizes for an SRAM cell that lead to cell stability during read and write operations. Next, the design of the compiler was made using SKILL in order to automate the process of generating an SRAM cell layout, and laying out copies of SRAM cells and manually-made precharge circuits. The compiler is designed to generate a compact SRAM cell whose layout is designed to connect to other cells without any extra routing necessary. All inputs and outputs of the SRAM cell are placed such that cells placed next to each other naturally connect and no further routing is needed when making an SRAM array. Next, there is the design of precharge circuitry for enhanced read performance. This precharge circuit is designed and laid out manually, and the compiler takes this layout and attaches copies of it to the top of the SRAM array. All of this comes together to form a functional SRAM memory compiler with multiple fabrication processes, an option for custom SRAM cell transistor sizes, and the potential to add more features in the future. This compiler design is followed by the design of an address decoder for accessing the desired cells for read and write operations. The design of the address decoder is separate from the compiler, and can be added to a layout to connect with an SRAM block generated by the compiler.

This report covers all of these topics in the following sections. Chapter 2 is a background of SRAM cells, precharge circuits, address decoders, memory compilers, and other memory peripheral circuits. Then, in Chapter 3 the Cadence software design tools and SKILL language are each discussed in terms of their significance to this SRAM compiler design. Chapter 4 discusses the design and implementation of the SRAM cell, precharge, address decoder, and SRAM compiler utilizing the SKILL language. Chapter 5 provides analysis of the testing and results of the circuits. Finally, Chapter 6 summarizes the design with concluding remarks and potential areas of future improvements and additions for the design.

Chapter 2: Background

Having a fundamental understanding of SRAM and related topics is necessary to progressing further on this topic. This section provides background information on the SRAM topology and function, a comparison of SRAM and DRAM, memory arrays, memory compilers, and other peripheral circuits that are helpful or necessary to using SRAM. For the circuits described, several topologies are observed, and only one is chosen for use in the design of the SRAM.

2.1 SRAM and DRAM: A Comparison

Two commonly used types of volatile memory are *Static* Random-Access Memory (SRAM) and *Dynamic* Random-Access Memory (DRAM). Both types have several similarities and differences. Exploring their advantages and disadvantages can help with deciding which type of memory is the optimal choice for designing a memory compiler for.

SRAM and DRAM are both volatile memory. This means that when the power supply to the memory is gone, all data is lost. This separates this kind of memory from non-volatile memories such as flash, EPROM, and hard drives where retaining large amounts of data indefinitely is more important than read and write speeds [5:554]. Another characteristic they share is their "Random-Access" feature, which allows the data they hold to be selectively read from or written to. This is in contrast to sequential types of memories, such as LIFO (last-in first-out), FIFO (first-in first-out), shift registers, and some ROMs, where the memory has to be accessed or written consecutively in order to reach the desired piece of data [5:554][2].

The differences between the two memories are what make them stand out in their respective applications. One of the clear distinctions between the two circuits is there topology. The SRAM cell (stores 1 bit of data) contains six transistors: 2 N-type MOSFETs for pass transistors, and 2 N-type MOSFETS and 2 P-type MOSFETS for the two cross-coupled inverters

(Figure 2.1). The data for the SRAM cell is stored in the cross-coupled inverters, where the feedback loop sets one output high and the other output low. With a power supply, SRAM holds its data indefinitely without needing to be refreshed, giving SRAM its *static* characteristic.

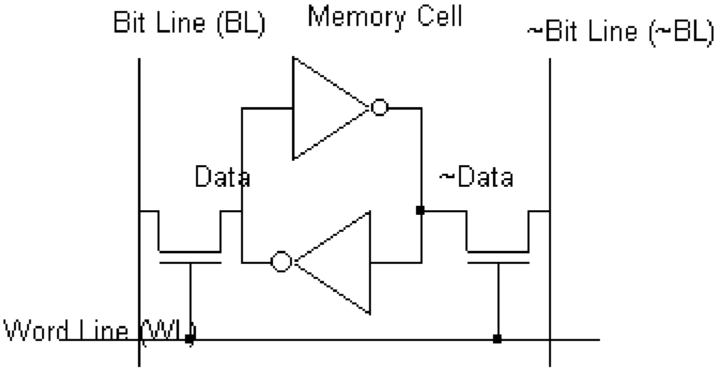


Figure 2.1: 6-Transistor SRAM Cell (showing CMOS Inverters) [3]

In contrast to the SRAM cell topology, the DRAM cell contains only one pass transistor (N-type MOSFET) and a capacitor, where the capacitor holds the data (Figure 2.2). The simple and comparatively smaller architecture of the DRAM cell allows it to form much higher density memory arrays than SRAM [9][3]. The small area of DRAM also makes it cheaper than SRAM, which is a reason it is commonly used as a system's main memory. Because capacitors have leakage current, the voltage they hold degrades over time, leading to DRAM-type memories needing to be refreshed periodically. This is done by external circuitry that amplifies the selected data, stores it, and then rewrites the cell(s) with the same data [7][8]. This refreshing describes the *dynamic* nature of DRAM.

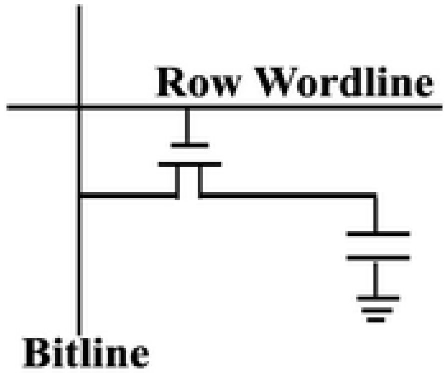


Figure 2.2: DRAM Cell [6]

DRAM's need for a refresh cycle leads to a few disadvantages that make SRAM stand out. One disadvantage is that using several processor cycles to refresh DRAM decreases the overall speed of the system, while SRAM can have access speeds that match the processor's clock frequency [9]. For this reason, SRAM is usually used as a cache to a processor where memory access times are critical [9][8]. Caches are smaller memory arrays compared to a system's main memory, so the higher cost and area of an SRAM cell is outweighed by the performance increase it gives to a system in terms of speed. Another disadvantage of the DRAM refresh cycle is the extra power consumption that doesn't exist with SRAM [8]. This is coupled with a constantly leaking DRAM cell capacitor leading to further power consumption.

In terms of designing a memory compiler across multiple technology processes, SRAM is an optimal choice between the two memory types due to how the memory compiler is designed. This memory compiler design is based on fulfilling the requirement of a Design Rule Check (DRC). The DRC is a list of rules that specify certain dimensions and connections between various layers in a layout that must be met in order for the semiconductor fabricator to successfully manufacture a design. The DRC rules depend on the fabrication service and the process technology of the design. In this case, the fabrication service is The MOSIS Service [11] and the process technologies are ON Semiconductor SCMOS and IBM cmrf7sf. The design of a functional DRAM depends on more than just passing the DRC.

One characteristic that is not determined by DRC rules is the leakage current of a capacitor. This leakage current is the reason DRAM needs to be refreshed, and the magnitude of the leakage affects the refresh rate that is needed. If the capacitances are not designed for the desired refresh rate, the DRAM will not function properly. The mere fact that DRAM needs to be refreshed according to a specific clock frequency makes it a system-dependent-based design. This adds a new layer of complexity that would prove difficult to incorporate into a compiler. DRAM complicates the design for other reasons as well. One is the device timing parameters [10]. There are at least thirteen timing parameters that need to be accounted for [12]. SRAM has timing parameters as well, but because it is static memory, it is not at risk of losing data. If the timing for DRAM is off, the whole circuit may not work, because the data will be lost, making it far more sensitive to timing parameters. Another factor is the memory-system configuration [10]. The configuration sets how wide the data buses are, whether they are 4 bits, 8 bits, 16 bits, etc., and this affects the size of each memory bank [12]. To account for every possibility that a user may want could be difficult, and not accounting for enough possibilities leads to a compiler that is limited. These few cases are enough to see that SRAM is a better choice for the purpose of this design without having to go into the several other factors that make DRAM a more complex system to design in a modular fashion.

While SRAM is simpler to design a compiler with, the parameters that define its performance can be used to the advantage of this design. In an SRAM design, the main variables that alter performance are the transistor size ratios within the cell. Changing these affects the speed, area, and power of the cell, giving the designer the freedom to optimize for their system. The plan for this memory compiler is to create a cell design optimized for stability, and then also give the user the ability to alter the transistor sizing ratios for their own optimization if they prefer. For academic purposes, this is a truly valuable feature. Users can create the best solution for their project design, or for research purposes, find how changing the sizing ratios affects the performance of the cell. All factors included, SRAM is the better of the two memory types for designing a memory compiler around.

While both types of memory each have their advantages and disadvantages, both are widely used and are important pieces of hardware needed in most processing systems. With the similarities and differences defined, the design of SRAM and its associated external hardware are now the focus in light of the end goal of creating a memory compiler.

2.2 SRAM Topology and Functionality

The topology of SRAM that is followed for this design is a standard 6-transistor (or 6T) model, which has four NMOS transistors and 2 PMOS transistors (Figure 2.3).

Having an understanding of SRAM cell functionality is important for more detailed design discussed in later sections. Figure 2.3 reveals that an SRAM cell has two bit lines, which transfer data between the cell and the external circuitry it is connected to. The bit lines – a bit line (BL) and a bit line bar (BL_BAR) – are complementary (one is a logical high, while the other is low) and are attached to two NMOS pass transistors (M5 and M6). Having two bit lines is not necessary, but having both when writing to and reading from the cell improves the noise margins in both cases [5:578]. The tradeoff is increased area for the added NMOS pass transistor attached to the complementary bit line. The cell also has a word line (WL), which enables the cell to be written to or read from, and is attached to the gates of the two pass transistors. The nodes labeled Q and Q_BAR are where 1 bit of data is stored. Q and Q_BAR are also complementary. An SRAM cell holds digitally opposite values because M1, M2, M3, and M4 form two cross-coupled CMOS inverters, as shown previously in Figure 2.1. In the case of Figure 2.3, M1 and M2 form one inverter, while M3 and M4 form the other. This inverter pair is a bistable latch and is the reason for the data being retained while power is being supplied. Its bistability simply points to the fact that the latch can hold two states: one side high and the other low, and vice versa.

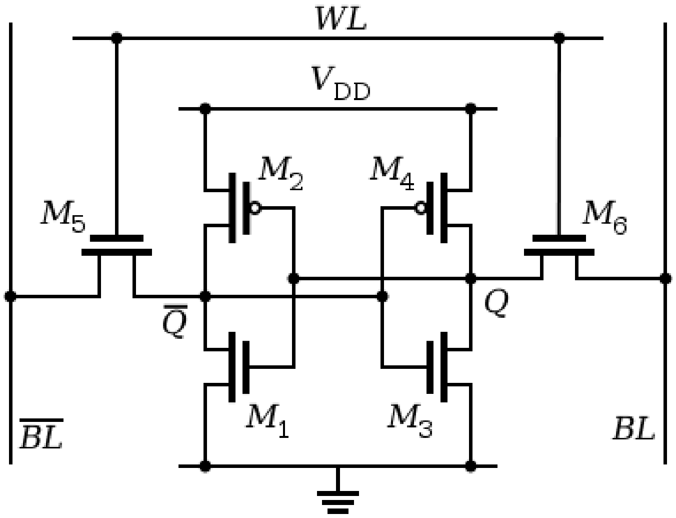


Figure 2.3: 6-Transistor CMOS SRAM Cell [2]

During a write cycle, the desired bit to be stored in the memory cell is placed on BL, with its complement placed on BL_BAR. For this example, BL is a digital 1 (1.8 V for cmrf7sf and 5 V for SCMOS), and BL_BAR is a digital 0. Then, WL is asserted high which turns on the pass transistors, M5 and M6, allowing them to conduct current. Assuming Q is 0 and Q_BAR is 1, BL tries to pull Q up to a 1 while BL_BAR tries to pull Q_BAR down to a zero. Once one side of the complementary cell flips, the feedback mechanism of the latch forces the other side to the corresponding value. The result is a value of 1 at node Q and a value of 0 at node Q_BAR. This writing ability is enhanced if the proper ratio of the PMOS (M_2 or M_4) W/L ratio to the pass NMOS (M_5 or M_6) W/L ratio is chosen. This is shown later in a more detailed SRAM cell design section. WL is then set low to 0 again, turning off the pass transistors, and Q and Q_BAR hold their values until the cell is written again or the power supply is turned off.

During a read cycle, the bit lines act as outputs instead of inputs. To read, WL is asserted high, turning on the pass transistors, and the values stored in Q and Q_BAR are placed on BL and BL_BAR. In the case where Q is initially a 0 and Q_BAR is initially a 1, when WL is set high, BL discharges down to a 0, and BL_BAR charges up to a 1. Note: with the addition of precharge circuitry (discussed in section 2.3), BL_BAR is already at a value of 1, so only BL has to discharge to a 0.

Another topology that exists is shown in Figure 2.4. This version, called the resistive load SRAM cell, has 4 NMOS transistors and 2 resistors instead of 2 PMOS transistors [15:662-664].

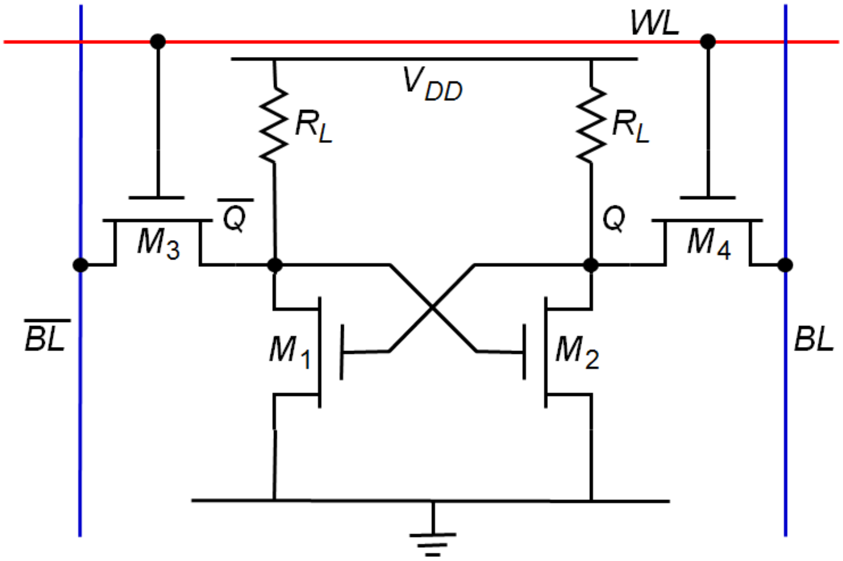


Figure 2.4: 4T2R SRAM Cell [23]

The advantage of this topology is its compact design. While the area of the cell can be reduced by about one-third, the static power dissipation increases due to the constant current through the resistors. To achieve the high resistance needed to limit the standby current through the resistors, highly resistive undoped polysilicon is used to form the resistors [15:662-664]. Due to the increased static power dissipation and the fact that the undoped polysilicon is not offered in the cmrf7sf and SCMOS process technologies used in this design, the conventional 6T SRAM cell topology is used.

2.3 Precharge Circuit for Enhanced SRAM Read

The precharge circuit is an important piece of hardware that enhances the read operation of an SRAM. By pre-charging both of the bit lines up before a read operation, less time is required during the actual read cycle. During a read operation, only one of the bit lines has to discharge down to 0. Without a precharge circuit, one bit line would have to charge up to a 1 and the other bit line would have to discharge. Also, because the SRAM cell pass transistors cannot pull a bit line up to a true logical 1, the precharge allows for this. A true logical 1 is a voltage above $V_{DD} - |Vtp|$ (Vtp is the PMOS threshold voltage). Having chosen to implement a precharge circuit, the next step is to choose a topology. There are a few different precharge circuit topologies commonly used, each with their own advantages and disadvantages. These topologies are shown to discuss what options exist, but only one is actually implemented.

One version of a precharge circuit is two pull-up NMOS transistors with their gates tied to VDD (Figure 2.5). In this topology, the NMOS transistors are always on, pulling both of the bit line up to VDD – Vtn, where VDD is the supply voltage, and V_{tn} is the NMOS threshold voltage. This circuit's inability to pull the lines to a full VDD is one disadvantage. This topology also does not have a control signal to turn the precharge circuit on or off. This is an advantage on one hand, because it works automatically and doesn't require a signal. However, it is also a disadvantage, because during a read cycle one of the pull-up transistors of the precharge circuit tries to move the bit line up to a high voltage, while the 0 stored in the cell tries to pull the bit line down to a low voltage. This results in a logical 0 that is not as close to 0 V as possible. The voltage that it drops to depends on the resistance of the NMOS pull-up which is determined by its W/L ratio. By increasing the gate channel length (L) and decreasing the channel width (W), the resistance of the pull-up transistor can be increased so that the lowest voltage on the bit line is decreased. This analysis is shown later in Chapter 4.

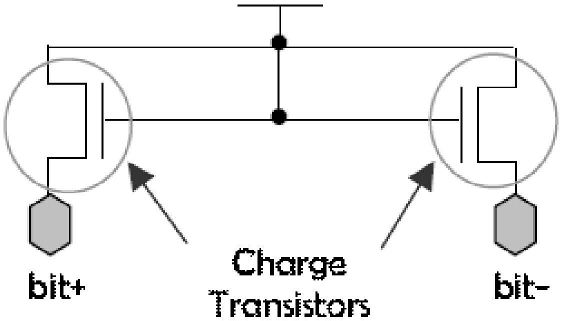


Figure 2.5: Precharge Circuit with Two NMOS Pull-Up Transistors [13]

Another commonly used precharge topology contains three PMOS transistors and an enable signal (Figure 2.6). One main difference between this version and the previous is the use of PMOS transistors instead of NMOS transistors. This advantageous change allows the bit lines to be pulled up to VDD instead of a threshold voltage lower. Another difference is the addition of a third PMOS transistor whose drain and source are connected to the bit lines. When this transistor is turned on, the bit lines are equalized during the precharge cycle. For SRAM that has a sense amplifier attached to the bit lines, this equalization is an added benefit, because once the read cycle begins, one of the bit lines will begin to discharge. With a sense amplifier the difference is more quickly detected if the bit lines are equal to begin with [14]. Since the main component of SRAM compiler design is to generate SRAM for multiple process technologies, a sense amplifier was not added to the design. Therefore, this added benefit is superfluous for this design. The last advantage of this precharge design is the clock signal, which enables the precharge circuit before the read cycle and disables it during the read cycle. This decreases power consumption since the transistors are not constantly trying to pull the bit lines up as in the previous case. It also allows one bit line to be pulled down to ground all the way during a read, because the pull-up transistors are no longer active during the read. One disadvantage of this topology is the added area of having three transistors. Having a clock signal to enable the precharge can also be considered a disadvantage, because this requires more control logic.

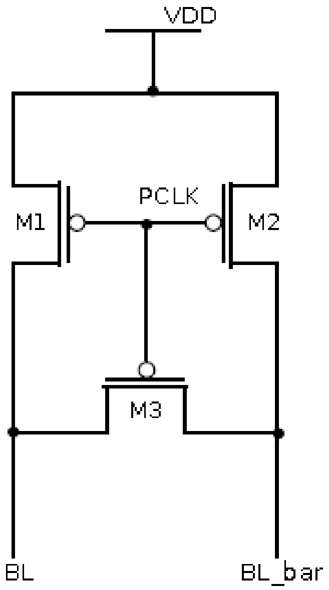


Figure 2.6: Precharge Circuit with Three PMOS Transistors and an Enable [14]

One final precharge circuit design is a hybrid of the two previous circuits in that it contains two PMOS pull-up transistors (Figure 2.7). One clear difference between this design and the previous is the reduced number of transistors. This is advantageous, because of the area reduction that occurs as a result. Since a sense amplifier is not used in this design, the third equalizing transistor is unnecessary. The two PMOS pull-ups have their gates permanently tied to ground, causing them to be on indefinitely. This reduces the number of control logic signals needed for a read operation. Also, it makes up for the lack of a sense amplifier in that it keeps one bit line close to VDD during the read cycle. The disadvantage for this circuit is similar to the NMOS-based precharge circuit where one bit line cannot be pulled all the way to 0 V; however, as mentioned before, the W/L ratios of the transistors can be altered to lower the final read voltage of the bit line. This will be discussed in further detail in Chapter 4.

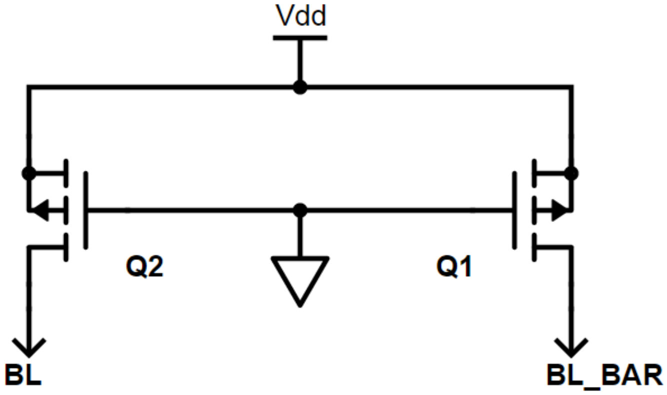


Figure 2.7: Precharge Circuit with Two PMOS Pull-Up Transistors

All factors considered, the precharge circuit of Figure 2.7 is the best option for this SRAM compiler design, and is what will be implemented. It should be noted that the precharge layout is not generated using the compiler; however, the compiler uses a manually-made precharge layout to create an SRAM with precharge. This is shown in Chapter 4. Next, how the SRAM cells fit together to form memory arrays is discussed.

2.4 Memory Arrays

Once an SRAM cell has been designed, the end goal is to use many copies of it to create a memory array so that larger amounts of data can be stored. A depiction of how an SRAM array is formed in rows and columns of many SRAM cells is shown in Figure 2.8.

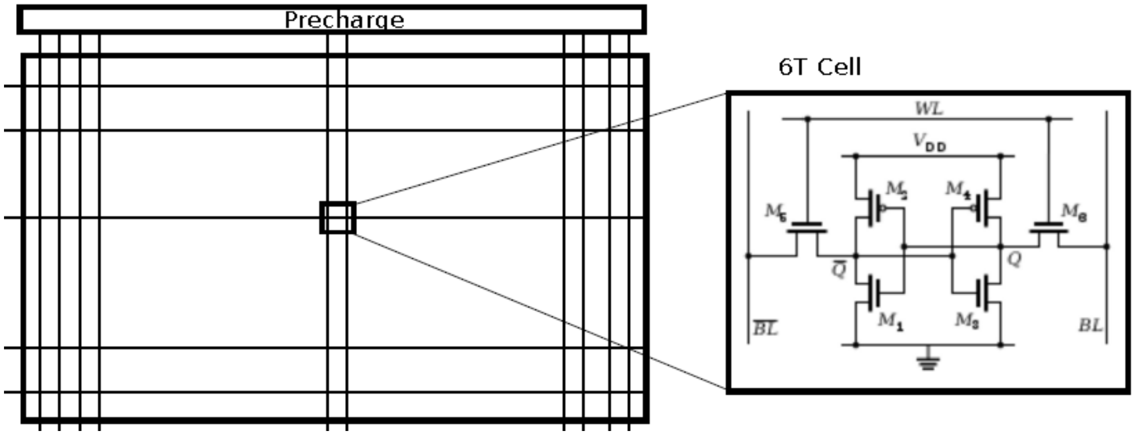


Figure 2.8: SRAM Array [14]

The way the array is structured gives SRAM its “random-access” characteristic. By selecting a row (via a word line) column (via a bit line and its complementary bit line), any SRAM cell can be accessed for reading data from or writing data to.

While the concept of an array is simple, understanding it gives light to several factors that need to be accounted for. For one, larger memories require a large number of select signals (as many rows of cells the array has) when used alone. The number of word lines can be greatly reduced with an address decoder (discussed in section 2.5). With a large memory array, power distribution across the rails leads to uneven voltages due to current through the supply rails, which have resistance. This calls for the need of power and ground rings and rails, which evenly distribute the power throughout memory arrays and digital logic circuits (discussed more in Chapter 3 and Chapter 4). As the number of cells attached to a bit line increases, the capacitance of that line increases, slowing down the charging and discharging of that line. To maintain cell stability so that the cell is not overwritten during a read cycle, proper transistor sizing within the cell is needed. This is discussed in Chapter 4. One final consideration is how cells can best fit together in layout, and how orientation across rows and columns matters. This too is shown in more detail in Chapter 4.

2.5 Address Decoder

An SRAM array can contain thousands of rows or more. Reading from or writing to an array without an address decoder would mean having the same number of select bits to turn the desired word lines on and off. An address decoder provides two main benefits despite being necessary for random-access memories: reducing the number of addressing signals to n for 2^n rows, and ensuring that only one row is being accessed at a time. The former reduces the number of bits needed to access the memory, and the latter makes certain that only the desired data is accessed.

The logic of an address decoder is described in the truth table below (Table 2.1). Each possible binary input corresponds with a separate output, giving 2^n outputs for n inputs. Table 2.1 shows that for a two-input address decoder, four outputs exist.

Table 2.1: 2-to-4 Address Decoder Truth Table

Input A_1	Input A_0	WL Address
0	0	WL0
0	1	WL1
1	0	WL2
1	1	WL3

There are multiple types of address decoders that provide the same fundamental functionality but with varying performances and drawbacks. Here, two topologies are observed, but only one is chosen to be implemented for the SRAM design. One common type of an address decoder is the dynamic logic-based decoder. One version of a dynamic-logic based decoder is the NAND decoder (Figure 2.9).

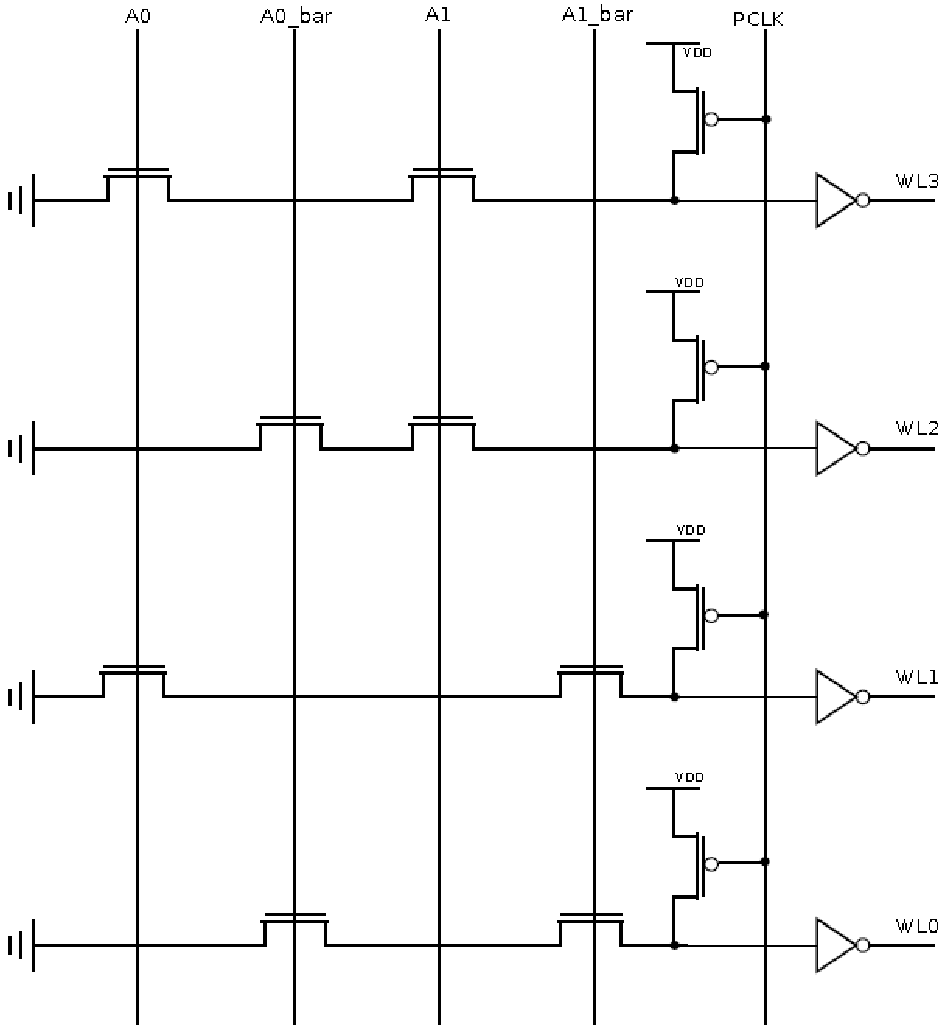


Figure 2.9: 2-to-4 Dynamic Logic NAND Address Decoder [14]

This address decoder is similar to a pseudo-NMOS logic design, but with the added control line for the pull-up PMOS transistors (making it dynamic logic), the power of this circuit is reduced [15:284-287]. One downside of this topology is the need for a control line to enable the precharge of each line, creating the need for an additional control signal. Compared to the next topology, this circuit design also requires a lot of transistors [5:591-594].

Another improved type of address decoder is static CMOS-based decoder that uses multiple stages (Figure 2.10). This multiple-stage logic provides a few benefits.

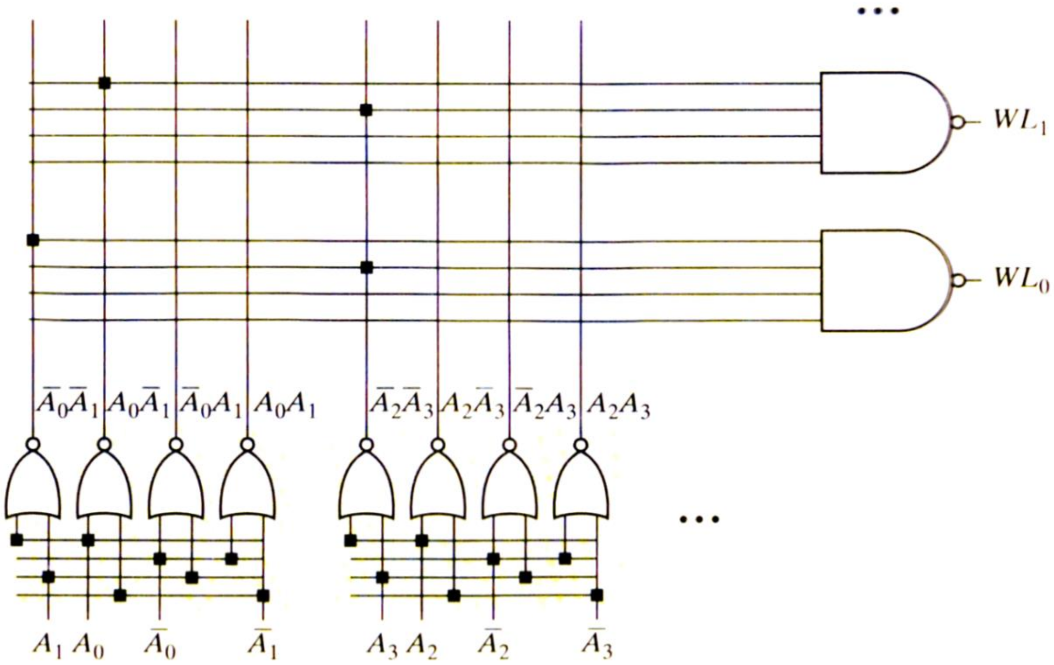


Figure 2.10: Static CMOS NAND Address Decoder Using 2-Input Predecoders [15:672-677]

One benefit that it provides over the previous address decoder is the reduction in the number of transistors. For example, a 10-input address decoder in this static CMOS topology contains only 55% of the number of transistors in the former circuit with the same number of inputs [5:591-594]. Another advantage the CMOS decoder has is a reduced propagation delay [5:591-594]. Finally, with the Cadence tools currently available for this SRAM compiler design (see Chapter 3), a circuit similar in form to the one depicted in Figure 2.10 can be generated using Verilog and standard cell libraries, removing the need to manually design the decoder by hand. The static CMOS multi-stage address decoder has been chosen as the best option for this design and is what will be implemented using the automated design flow (shown in Chapter 4). This address decoder is not generated by the compiler, but is made manually, and can be connected to an SRAM block generated by the compiler.

Next, two peripheral circuits that are useful for SRAM are discussed; however, they are not implemented in this design.

2.6 Sense Amplifier

The sense amplifier is a very useful circuit for the read operation of the SRAM. One common topology of a sense amplifier is shown in Figure 2.11.

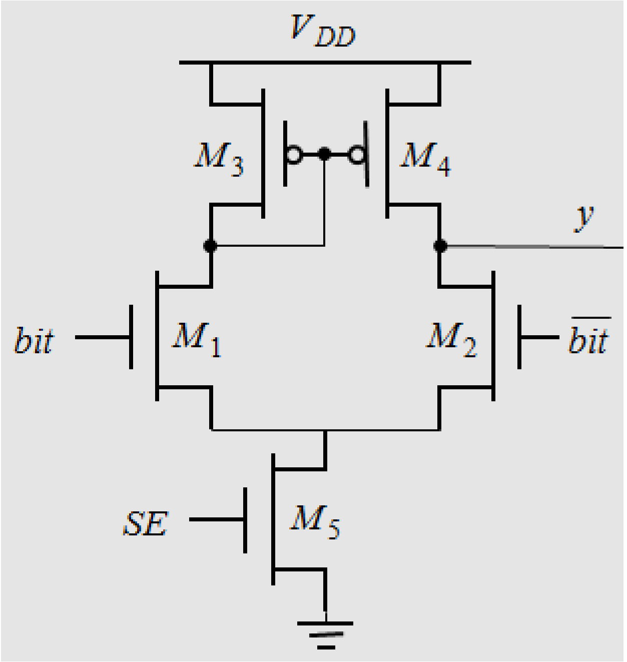


Figure 2.11: Differential Sense Amplifier [23]

The inputs of the sense amplifier connect to the two complementary bit lines, so that during a read cycle, as one bit line drops in voltage the difference that occurs between the two lines is amplified quickly by the sense amplifier, making the output value a rail-to-rail reading (V_{DD} or 0 V). Having the sense amplifier accelerates read operation by amplifying the small voltage difference, and allows for a reduction in power by minimizing the required voltage swing on the actual bit lines so they don't have to charge and discharge as much [15:679-682].

Since the primary focus of this thesis is to design an SRAM compiler for multiple process technologies and generate custom SRAM cell layouts, the sense amplifier is not included in the actual design, but should be mentioned as a useful peripheral circuit.

2.7 Column Mux

Similar to the address decoder, the column mux (or column decoder) helps select which set of bit lines (instead of word line as in the case of the address decoder) to read data from or write data to. This ultimately reduces the number of signals needed to select a pair of bit lines for read or write access, and also ensures that only one set of bit lines is accessed, removing the potential for reading from or writing to multiple cells unintentionally. One common column mux topology is shown in Figure 2.12.

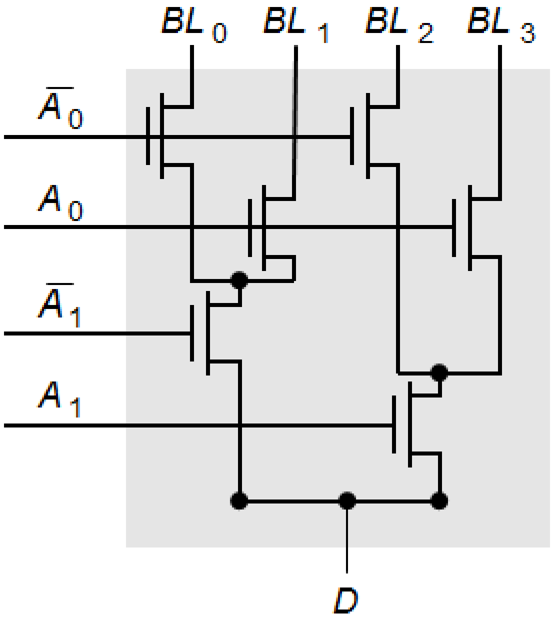


Figure 2.12: 4-to-1 Tree-based Column Mux [23]

With the column mux, an address is selected to transmit the data from the bit lines when performing a read operation, and transmit the data to the bit lines when performing a write operation. There are other topologies that affect the performance and area of the circuit, but the tree-based column mux shown in Figure 2.12 has a greatly reduced area compared to the CMOS pass-transistor multiplexer [15:677-678]. The disadvantage is that it is a much slower design [15:677-678].

The column mux is also not included in this SRAM compiler design since the focus was on designing a memory compiler for multiple process technologies, and generating custom SRAM cell layouts based on variable transistor sizes.

2.8 Memory Compiler

Before detailing the design of this compiler, it is important to understand what a memory compiler is and how it works, as well as the various features that a compiler may have. Firstly, the main purpose of a memory compiler is to reduce the time needed to design and add memory to an integrated circuit. For intermediate experience with the Cadence layout tool and having prior knowledge of what an SRAM cell layout looks like, it takes at least an hour to successfully layout an SRAM cell to completion without any DRC errors and removing any unnecessary space. Using a memory compiler can reduce the time of layout for a cell to just a matter of seconds (see the corresponding testing section of Chapter 5). Also, this doesn’t include the time it would take to generate thousands of cells in an array. This is a useful tool to have in the integrated circuit industry where development time is crucial when producing chips.

The way a memory compiler works can vary depending on the source of the compiler, but the general functionality is the same. A user will specify how large of a memory is needed, what kind of memory, and often other features such as an optimization for speed, power, or area, an option to choose the desired aspect ratio of the cell, and more. An SRAM cell layout is used as the “leaf cell” [13] which is copied numerous times to form an array (core block), and then external circuitry such as an address decoder, precharge circuits, and sense amplifiers are added to the peripherals of the core memory block.

On the cell level, this compiler design is customizable for the user’s needs. Typical memory compilers have set transistor sizes for an SRAM cell, and while this compiler offers a default optimal cell design, the option for custom sizes gives the compiler more variety for the applications it can be used for. For example, as the size of the SRAM array increases, the bit line capacitance grows decreasing the speed of the read and write operations. Having the option to increase the speed of the cell for this case is quite useful. Another special feature of this compiler is that it generates SRAM for two very different process technologies (made by different companies and significantly different feature sizes, or component and wire dimensions). There are companies that offer memory compilers for a wide variety of process technologies; however, it appears that they are offered separately, not as a whole design package [16] [17]. In total, this compiler design allows the user to specify the process technology (of two choices), the SRAM cell transistor sizes, and the size of the SRAM array.

There are a wide variety of SRAM compilers that exist. Some offer a few features and are open source [13] [14], while others offer a larger number of features and SRAM types (such as single port and dual port), but are more expensive. A summary of the reviewed available SRAM compilers is shown in Table 2.2.

Table 2.2: Summary of Reviewed Available SRAM Compilers

	Industry	Open Source (Thesis) [13][14]	My Compiler Design
Multiple Process Technologies	✓ Separate packages	✗	✓ IBM cmrf7sf and ONSEMI SCMOS
Cost	✗ Expensive	✓ Free	✓ Free
Multiple SRAM Types	✓ Single Port, Dual Port, Synchronous, Asynchronous	✗ Single Port Only	✗ Single Port Only
Customizable Cell Transistor Sizes	✗	✗	✓
Speed Optimization	✓ Often a Separate Package	✗	✓ Limited to Cell Customization
Area Optimization	✓ Often a Separate Package	✗	✓ Limited to Cell Customization
Power Optimization	✓ Often a Separate Package	✓	✓ Limited to Cell Customization

From Table 2.2, one can see that industry offers many of the possible features that could exist in a compiler, but the downside to their product is the expensive license to use the compiler, and the fact that the extra features or process technologies are often a part of a separate package that costs more money. The open source compilers found in various theses [13][14] provide valid solutions at an unbeatable price; however, they are lacking a lot of desirable features such as various optimizations and design for multiple process technologies. The open source compilers are more accessible for custom modifications than the industry-made compilers; however, neither the open source nor industry-based compilers offer custom transistor sizing for the SRAM cell. This feature, coupled with the multiple process technology design (7rf and c5n) and low cost (free) make this SRAM compiler solution a very useful tool that fills certain gaps in this area of technology.

Next, the design tools and software framework used to implement and test this SRAM compiler design are discussed.

Chapter 3: Memory Compiler Layout Tools and Supporting Software Framework (SKILL)

Designing an integrated circuit requires several software tools. These tools give the design to build the structural design (schematic), simulate the design, and finally create a physical layout of the chip. This section describes these tools and how they are used in the design of the SRAM compiler.

3.1 Cadence Software Design Tool Suite

All circuit designs for this compiler were created and tested using Cadence software tools. These tools include Virtuoso Schematic Editor, Virtuoso Layout, ADE (Analog Design Environment), RTL Compiler, and Encounter. In addition to these tools, the SKILL IDE and SKILL language were used to create the compiler.

3.1.1 Virtuoso Schematic Editor XL

Besides Verilog-based circuits, most designs begin with creating a schematic. The schematic is the first implementation step in the fully custom and semi-custom IC (integrated circuit) design flows (see section 3.2) after choosing a circuit and the needed components. The schematic is a component-level design and displays how components are connected and what the component properties are, such as resistance for resistors and gate channel width for transistors. The Cadence Virtuoso Schematic Editor utilizes a graphical user interface (GUI) to assist the user in building circuits. This is in contrast to some less appealing schematic editors that rely on the user to code net lists and SPICE parameters. Figure 3.1 shows the Virtuoso schematic GUI with an example circuit being built.

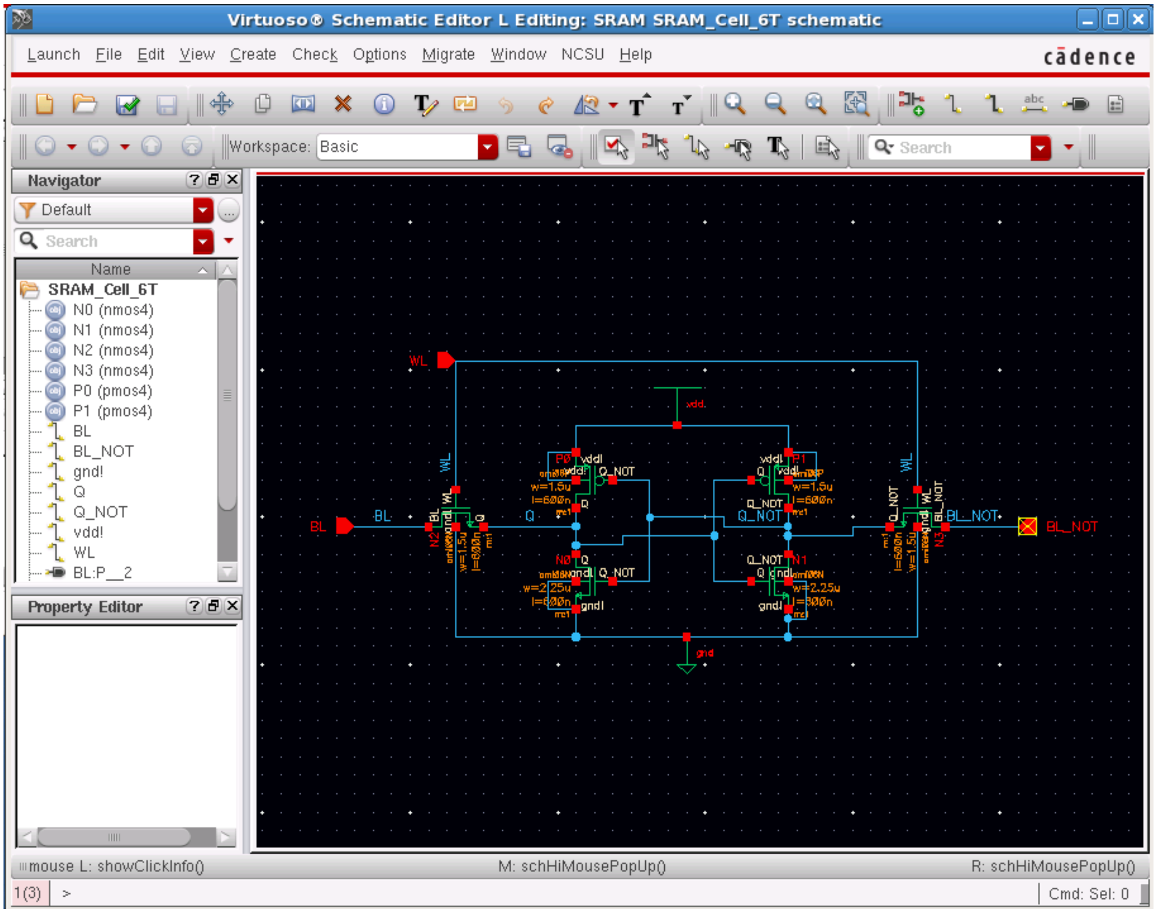


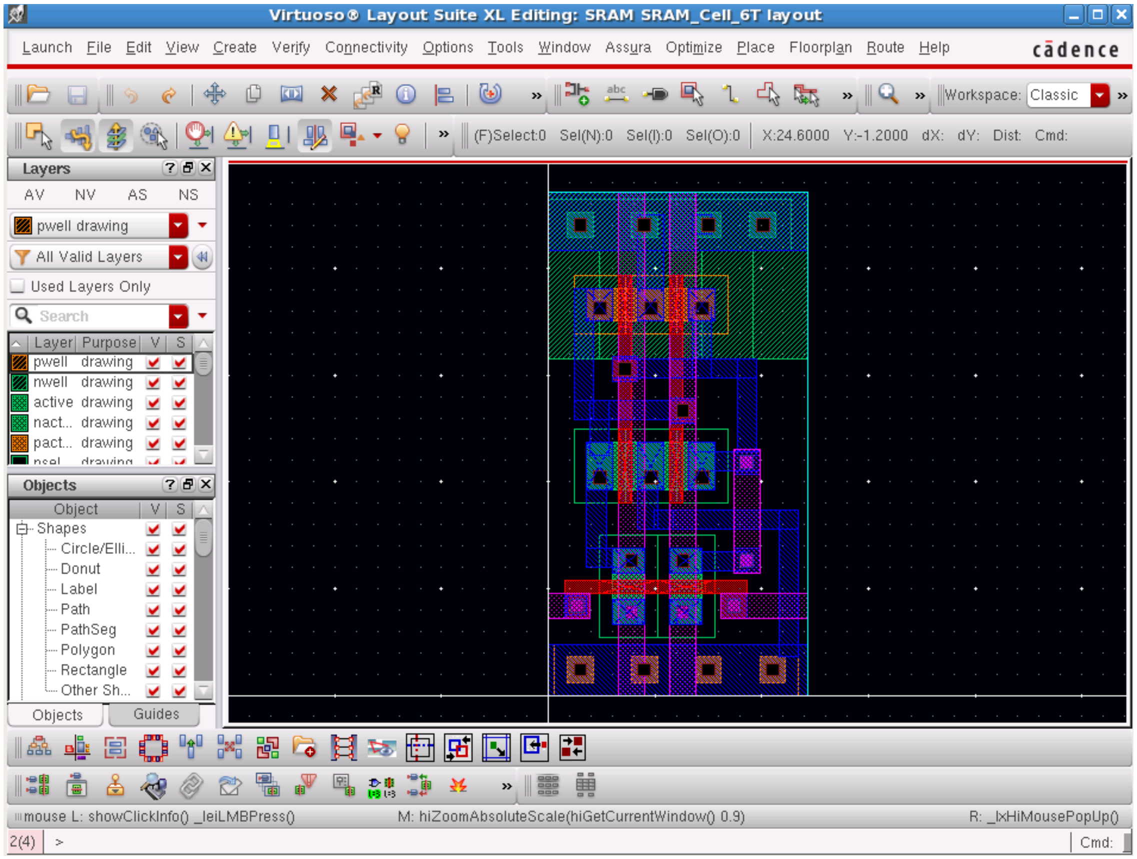
Figure 3.1: Virtuoso Schematic Editor with Example SRAM Cell Circuit

For this design, the schematic editor is used to build and simulate (using ADE) SRAM and its associated external circuitry (precharge and write select). By building schematics of these circuits, simulations (in ADE) and quick changes to component parameters (transistor sizes in the case of this design) can be made to verify the function and performance of a design.

These schematic-based circuit builds are abstract in that they don’t define physical locations or layouts of the components. What are defined in the schematic are the components used, their parameters, circuit inputs and outputs, and the connections between various nodes of the circuit (called “nets”). In order to define the physical layout of a circuit, a layout tool is used.

3.1.2 Virtuoso Layout XL

The next important tool that is used in IC design is the layout tool, which allows you to draw the physical silicon, poly (polysilicon), metal, and other layers. This tool allows the user to take the circuit they made in the schematic editor and build it in the way it will physically be fabricated on a silicon wafer. The design tool used for the layouts in this design is Virtuoso Layout XL. The GUI for this tool is similar to that of the schematic editor. Figure 3.2 shows the Virtuoso layout GUI with an example circuit.



(F)Select:0 Sel(N):0 Sel(I):0 Sel(O):0 X:24.6000 Y:-1.2000 dX: dY: Dist: Cmd:

mouse L: showClickInfo() _leiLMBPress() M: hiZoomAbsoluteScale(hiGetCurrentWindow() 0.9) R: _lxHiMousePopUp()

2(4) > Cmd:

Figure 3.2: Virtuoso Layout XL with Example SRAM Cell Circuit

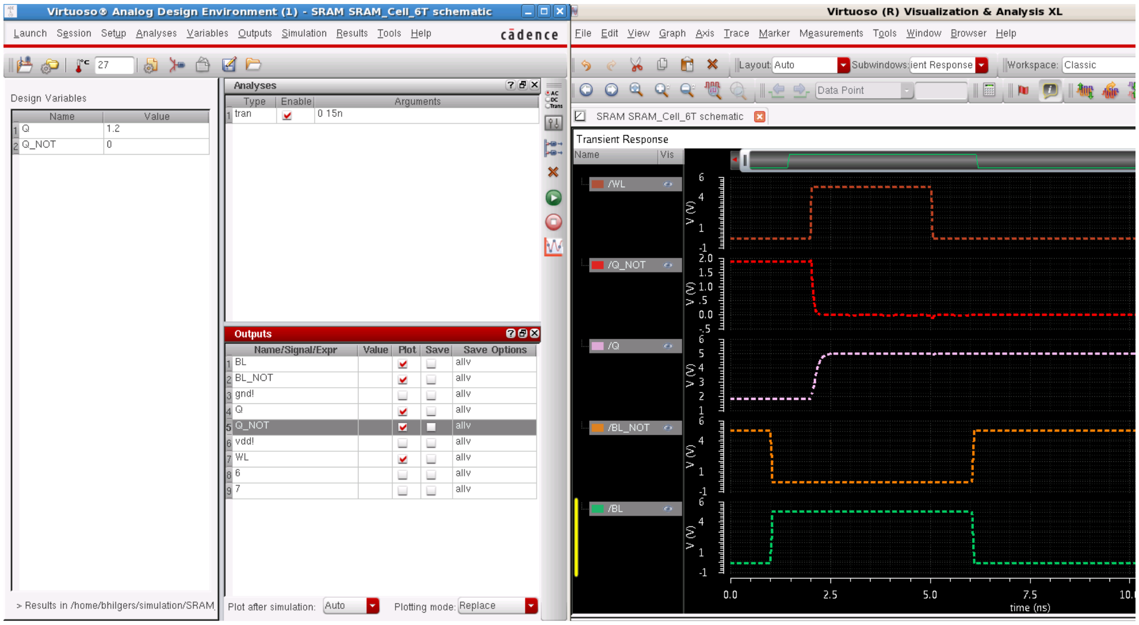
The tool offers a couple of verification steps that are important to perform. One is DRC (Design Rule Check), which reveals whether the layout meets the fabricator’s standards for proper layout dimensions and geometries. Without passing DRC, a chip cannot be properly fabricated and might not fabricate and/or operate correctly. Another verification step is LVS

(Layout vs. Schematic) which checks if the layout netlist structurally matches the netlist of the schematic that the layout was generated from. This ensures that the layout has the same structure as the intended circuit design, since it is visually more difficult to tell if all connections are made correctly in a layout compared to a schematic.

Much of the development of this SRAM compiler required the use of the layout tool. The compiler results in a layout of an SRAM cell and a layout of an SRAM array with precharge circuitry, write select circuitry, and power and ground rings.

3.1.3 ADE (Analog Design Environment)

Another important part of the IC design flow involves simulating circuits to ensure that they function properly and have the desired performance. The ADE L tool from the Cadence design tool suite is used to verify the circuits implemented in this design. More specifically, this tool allows the user to specify the input signal types (pulse, sine wave, DC, etc.) and circuit's initial voltages, and then observe the outputs and other nodes of interest after simulation. Based on the results of the simulation, the circuit design can then be changed if needed and a new simulation can be run to observe the updated circuit design performance. Figure 3.3 displays the simulation tool along with an example simulation.


Virtuoso® Analog Design Environment (1) - SRAM SRAM_Cell_6T schematic
Launch Session Setup Analyses Variables Outputs Simulation Results Tools Help
cādence
27
Analyses
AC
DC
Trans
Virtuoso (R) Visualization & Analysis XL
File Edit View Graph Axis Trace Marker Measurements Tools Window Browser Help
Layout: Auto
Subwindows: ient Response
Workspace: Classic
Data Point
SRAM SRAM_Cell_6T schematic
Transient Response
Name
Vis
/WL
/Q_NOT
/Q
/BL_NOT
/BL
V (V)
0.0
2.5
5.0
7.5
10.
time (ns)

Design Variables

	Name	Value
1	Q	1.2
2	Q_NOT	0

	Type	Enable	Arguments
1	tran	☑	0 15n

Outputs

	Name/Signal/Expr	Value	Plot	Save	Save Options
1	BL		☑	☐	allv
2	BL_NOT		☑	☐	allv
3	gnd!		☐	☐	allv
4	Q		☑	☐	allv
5	Q_NOT		☑	☐	allv
6	vdd!		☐	☐	allv
7	WL		☑	☐	allv
8	6		☐	☐	allv
9	7		☐	☐	allv

> Results in /home/bhilgers/simulation/SRAM

Plot after simulation: Auto

Plotting mode: Replace

Figure 3.3: Analog Design Environment Simulation Tool with Example Simulation

The simulation tool is a key component of this SRAM compiler design. Starting with the SRAM cell optimization, varying transistor sizes greatly affects the speed, power, and stability of the circuit. Once simulations for one cell are done, simulations for other used circuits (precharge, write select, etc.) are done, both individually and when connected with an SRAM cell or an array of SRAM cells. As examined later in Chapter 5, the performance of the SRAM varies depending on how many cells are present in an array and what other external circuits are attached.

3.1.4 Encounter RTL Compiler

Another form of IC design utilizes a register transfer language, or more specifically a hardware description language, to create a structural diagram of a circuit and does not use the schematic editor for circuit entry. The use of a hardware description language to design a circuit can greatly reduce the time required for development, layout, and testing. The RTL compiler works by taking a register transfer language, such as Verilog, and converts it to structural Verilog and creates a gate-level circuit diagram. This conversion step is called synthesis. From here, the gate-level circuit is turned into an actual layout using the Encounter Design Implementation

System tool (see section 3.1.5). The Encounter RTL Compiler tool is shown in Figure 3.4 with an example gate-level circuit diagram.

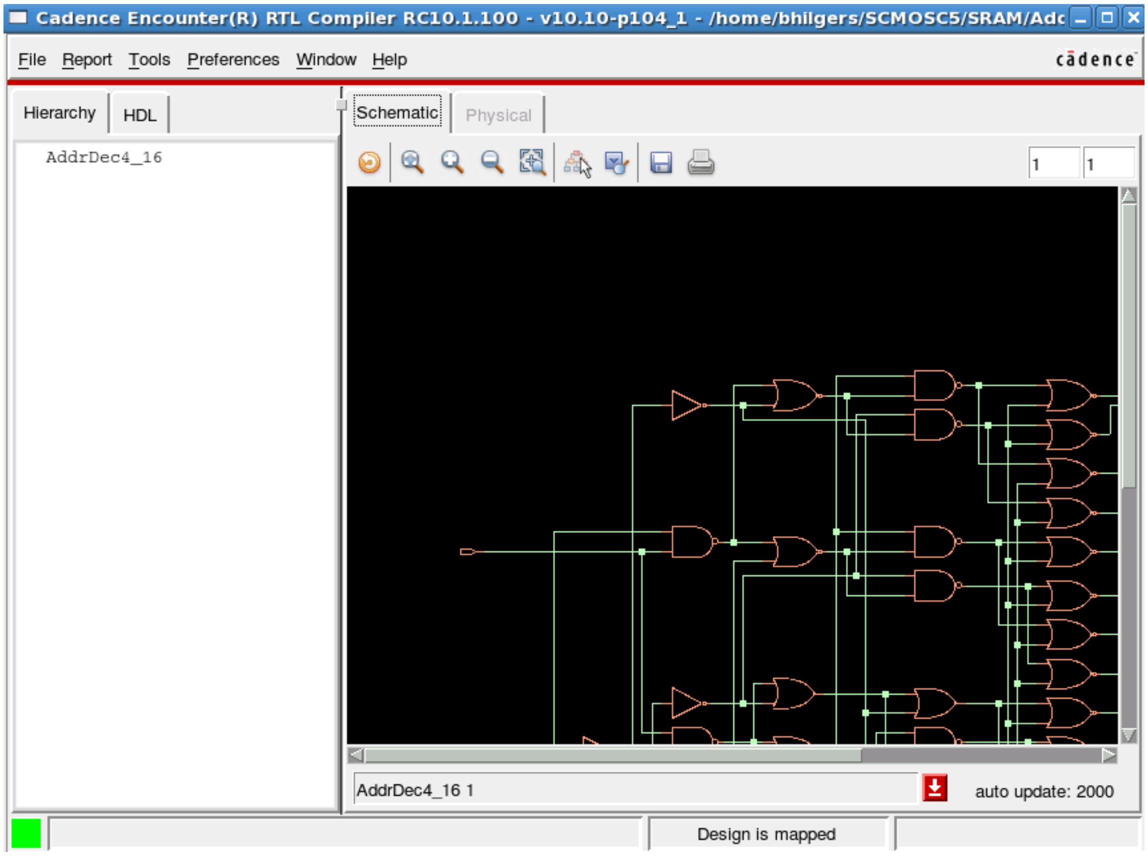


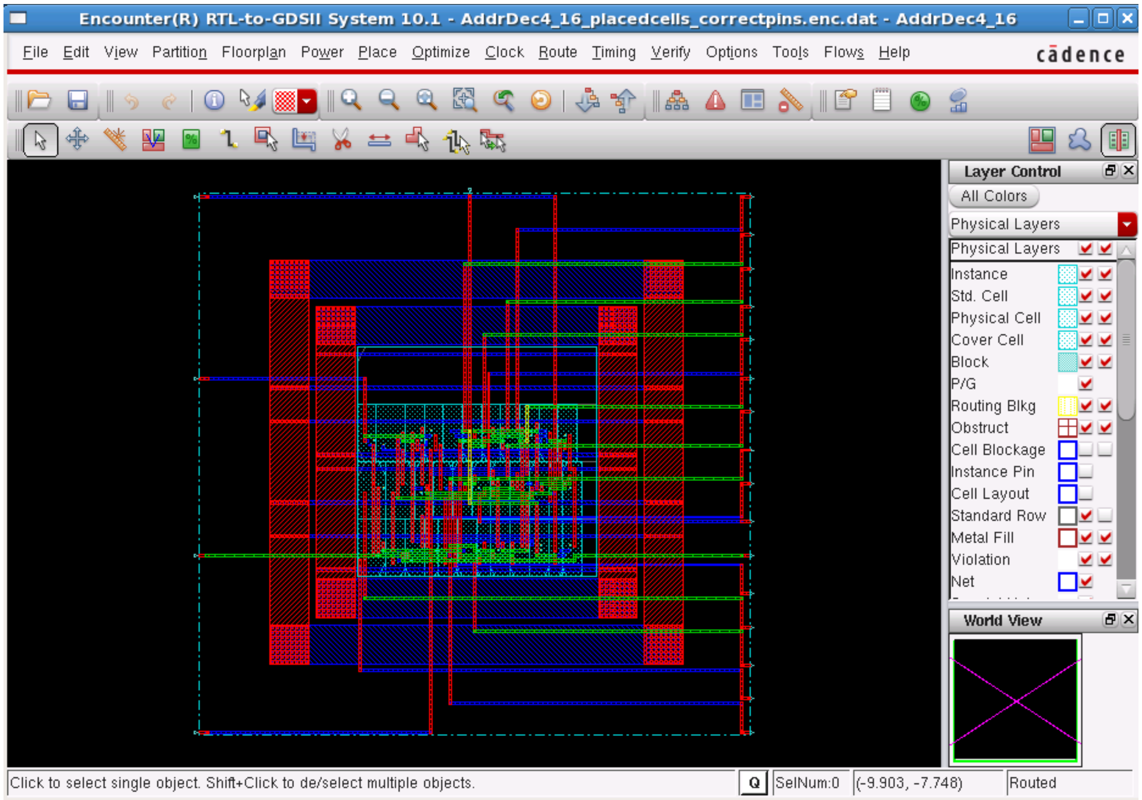
Figure 3.4: Encounter RTL Compiler with Example Circuit

The preexisting standard cell libraries and simplicity of the Verilog-to-circuit methodology make Encounter an efficient design tool option.

3.1.5 Encounter Digital Implementation System

After a structural gate-level diagram of a circuit is made using the RTL compiler, standard cells (premade logic gate layouts) used in the circuit are placed in rows and columns in a layout, and then the connections are routed together. This is all done in the Encounter Digital Implementation tool. The tool allows the user to define the final circuit dimensions and then add power and ground rings. Once this is complete, the tool uses the structural Verilog file generated from the RTL compiler and places the needed standard cells on the circuit grid accordingly.

Finally, automatic routing is done to make all of the connections between the various logic gates. Once the circuit is complete, DRC can be run to verify the layout and the circuit can be exported to a layout file that can be used in Virtuoso Layout so the circuit can be fabricated, or attached to other designs if desired. The Encounter Digital Implementation tool GUI is shown in Figure 3.5 with an example circuit layout.



Click to select single object. Shift+Click to de/select multiple objects.

Figure 3.5: Encounter Digital Implementation System Tool with Example Circuit

Having standard cells that allow for automation of a significant portion of the design process make using Encounter an appealing option for building a digital logic-gate-based circuit. Also, circuits designed in Encounter can be imported into Virtuoso Layout XL as a block and connected to other circuit layouts designed in the Virtuoso tools.

3.2 Integrated Circuit Design Flows

Having discussed the various tools used in integrated circuit design, the next step is to understand the different design flows in IC design, and what steps are taken to go from concept to a fabricated chip using these tools. There are three different design flows: Automated, Semi-Custom, and Full-Custom. These design flows are summarized in Figure 3.6.

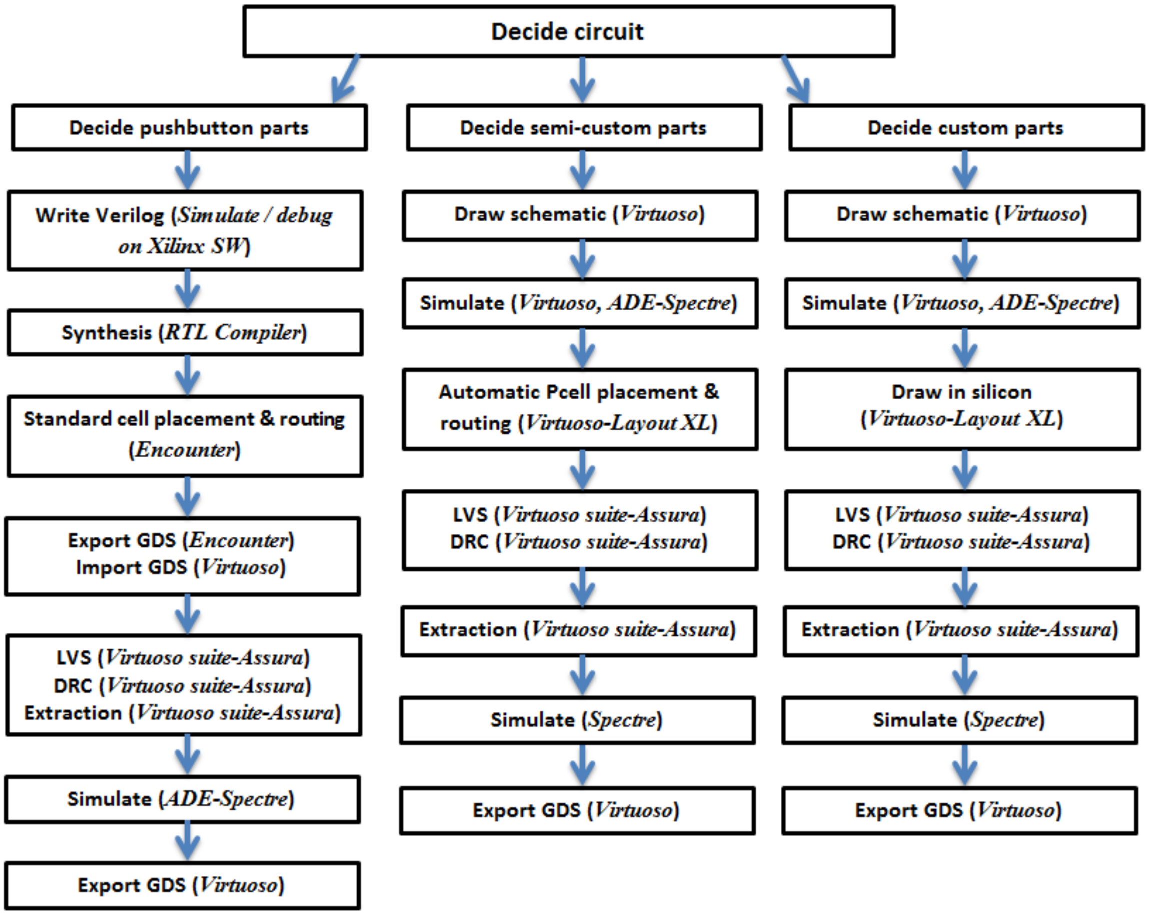

Decide circuit
Decide pushbutton parts
Write Verilog (Simulate / debug on Xilinx SW)
Synthesis (RTL Compiler)
Standard cell placement & routing (Encounter)
Export GDS (Encounter)
Import GDS (Virtuoso)
LVS (Virtuoso suite-Assura)
DRC (Virtuoso suite-Assura)
Extraction (Virtuoso suite-Assura)
Simulate (ADE-Spectre)
Export GDS (Virtuoso)
Decide semi-custom parts
Draw schematic (Virtuoso)
Simulate (Virtuoso, ADE-Spectre)
Automatic Pcell placement & routing (Virtuoso-Layout XL)
LVS (Virtuoso suite-Assura)
DRC (Virtuoso suite-Assura)
Extraction (Virtuoso suite-Assura)
Simulate (Spectre)
Export GDS (Virtuoso)
Decide custom parts
Draw schematic (Virtuoso)
Simulate (Virtuoso, ADE-Spectre)
Draw in silicon (Virtuoso-Layout XL)
LVS (Virtuoso suite-Assura)
DRC (Virtuoso suite-Assura)
Extraction (Virtuoso suite-Assura)
Simulate (Spectre)
Export GDS (Virtuoso)

Figure 3.6: IC Design Flows: Automated (Left), Semi-Custom (Center), Full-Custom (Right) [18]

3.2.1 Automated Design Flow

The left column of Figure 3.6 displays the automated design flow. The automated design flow is based on the use of premade layouts of logic gates (standard cells) which are used to create a circuit defined by a hardware description language such as Verilog. The first step begins with choosing a circuit. Once the desired circuit has been chosen, the automated design starts with writing Verilog (behavioral) code that represents the behavior of that circuit. This code can be simulated using another piece of software, such as Xilinx, and then it is ready to be synthesized. During synthesis, the RTL compiler tool takes the Verilog code and converts it to structural Verilog and creates a logic-gate-level schematic of the circuit. Much like the schematic netlist described in section 3.1.1, the structural Verilog file describes the components used – logic gates, in this case – and the connections that are made between them. Then, the Encounter Design Implementation tool is used to setup and create the physical layout. Here, the dimensions of the circuit are defined, and the circuit blocks (if there are multiple) are arranged and shaped to meet the designer's needs. This step is called floor planning. Next, in the power planning step, power and ground rings, stripes, and rails that distribute power evenly throughout the circuit are established around and along the grid of the circuit-defined area. Rings are metal wires that surround the circuit, while stripes distribute power across the circuit with vertical wires, and rails with horizontal wires. Distributing the power in this fashion creates a more even supply voltage throughout the circuit by reducing the effective resistance of the wires supplying the power. Figure 3.7 shows power rings created using Encounter.

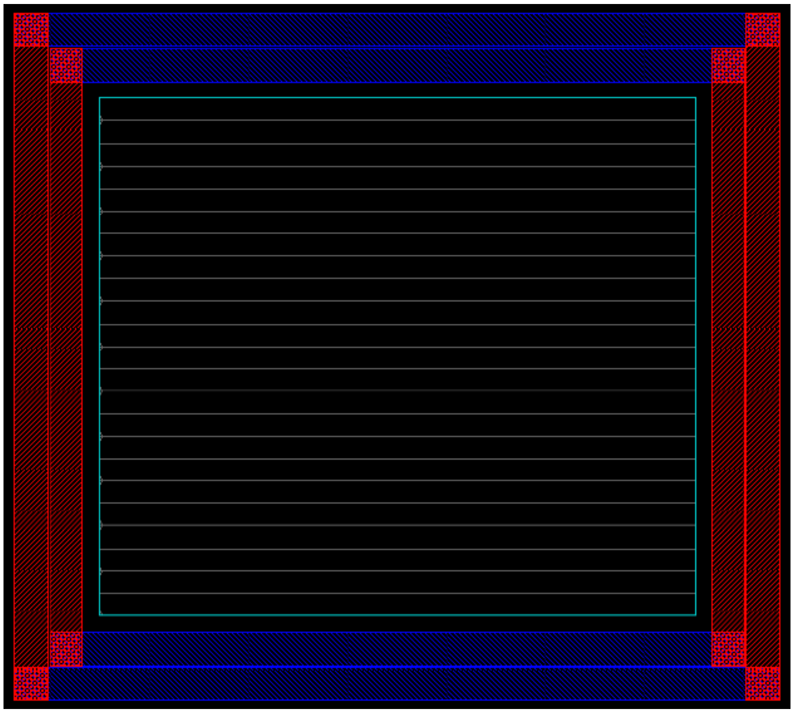


Figure 3.7: Power Rings [19]

Figure 3.8 shows power rings with stripes and rails added to the power rings. A green circle in Figure 3.8 identifies where some of the rails connect to the power ring.

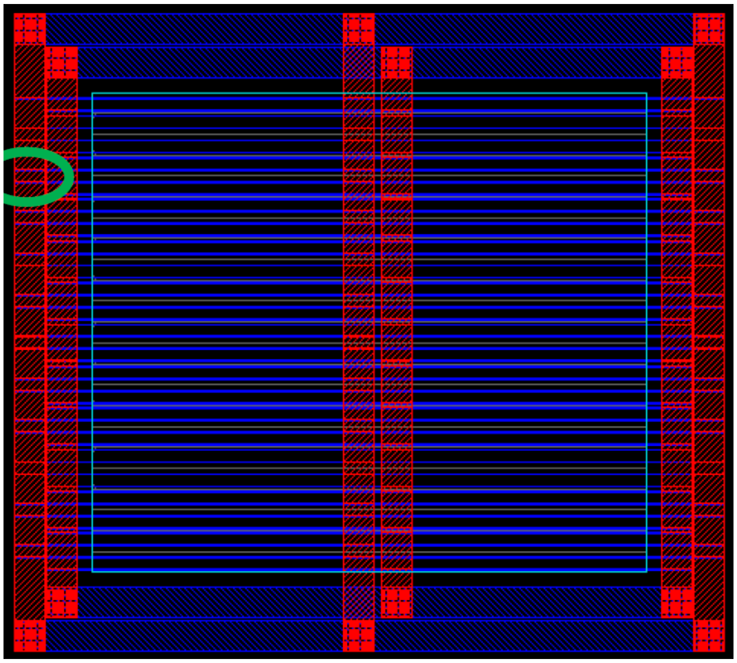


Figure 3.8: Power Rings with Stripes (Vertical) and Rails (Horizontal) [19]

After floor planning and power planning, the tool is used to place standard cells in the circuit-defined area between power and ground rails. These standard cells are circuit layouts of the logic gates which are used to build the functions described in a structural Verilog file. Figure 3.9 shows an example of a standard cell, in this case, a 2-input AND gate.

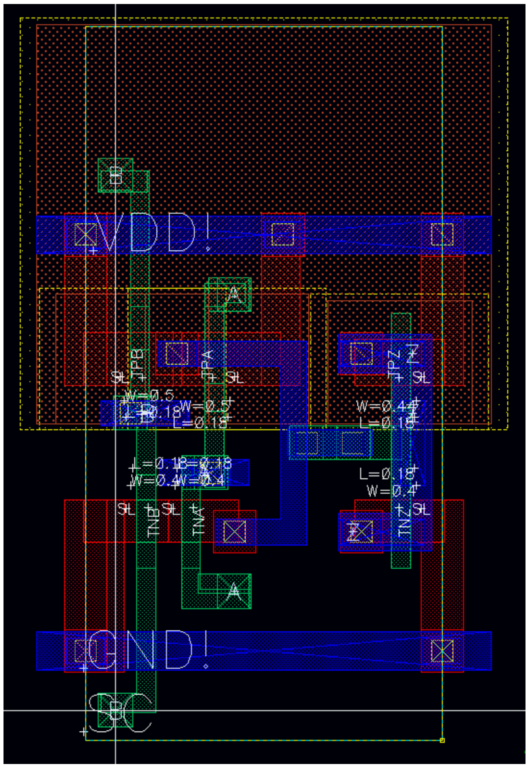


Figure 3.9: 2-Input AND Gate Standard Cell

The cells are *standardized* to have set dimensions to fit between the power and ground rails of the automated circuit design grid in Encounter, allowing them to easily be placed on this grid in compact rows and columns. This allows the physical layout of large digital circuits with complex logic to be built *automatically* and relatively quickly compared to building the circuit manually, as in the custom design flows.

Once the cells have been placed, the same Encounter tool is used to automatically route all of the connections between the logic gates for you. Besides specifying the circuit dimensions and the locations of the input and output pins, the circuit building process for this design flow is automatic. Once the circuit has been built, verification steps are done to help ensure the circuit will function and be fabricated properly. One verification step is the connectivity, which much like LVS, checks if the connections the circuit layout match the connections in the original circuit diagram. The other step is a geometry check, which is a lot like the DRC. It is not as complete as the DRC, but it is a helpful check that can save time in checking the circuit before exporting the layout over to Virtuoso.

The next step is to export the design to Virtuoso so it can be used with an additional circuit and/or verified further through DRC and LVS. From Virtuoso, the next step is extraction, which gathers information about the layout that is not shown in the schematic, such as parasitic capacitances and resistances of wires used. These characteristics of the circuit are not accounted for a schematic simulation, making it not as accurate. If extraction is done, the extracted parameters can give a more accurate representation of the physical layout's performance through a simulation done on the extracted layout. This can be done in the ADE tool. When the circuit is functioning to meet the designer's needs, the circuit layout is exported to a file that is sent off to the fabricator to physically create the integrated circuit.

The main advantage to this design methodology is that it is much faster than custom design. Since a designer is relying on premade standard cells drawn from a library, much of the time normally needed to develop these circuits is removed. One of the disadvantages to this design flow is that the design is limited to the circuits available in the library, and might not be the most efficient for a certain design, where efficiency can be defined by power, speed, or area. Another disadvantage is that this flow only creates digital circuits, not analog, since all of the standard cells used are digital logic gates. For analog circuits, and customized digital circuits, the two custom design flows are used.

3.2.2 Semi-Custom Design Flow

The custom design flows allow for more design options, but require more time to implement. The semi-custom design flow begins with building a chosen circuit design in the Virtuoso Schematic Editor. Next, the circuit is simulated to see if it functions properly. If the designer wishes to have a different performance of the circuit, changes can be made back in the schematic and it can be simulated again, repeating this process until the desired performance is achieved. With the circuit netlist created and component parameters chosen, the next step is to create a physical layout of the design using Virtuoso Layout.

The custom design flows rely more on the designer’s knowledge of component parameters that affect circuit performance. In the semi-custom flow, these parameters are chosen through options in the components in schematic, and PCELLs (parameterized cells) in layout. PCELLs are predesigned layout components, such as transistors, resistors, vias, etc. that can be used to build a physical layout. Figure 3.10 depicts a PMOS PCELL used in layout.

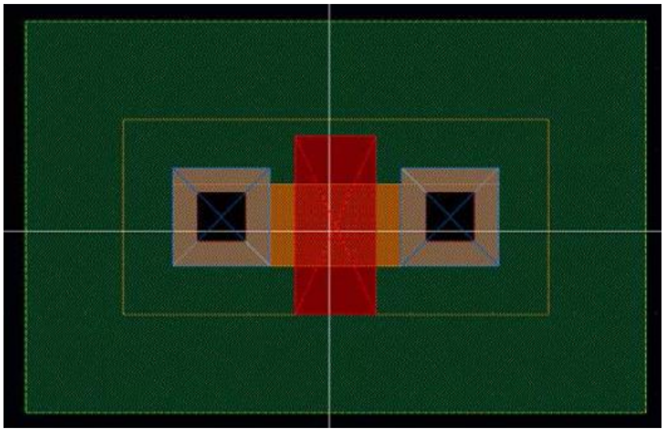


Figure 3.10: PMOS Parameterized Cell [20]

By using PCELLs, the layout time can be reduced compared to manually laying out each component (as in the full-custom design). In Virtuoso Layout, these PCELLs are generated from the schematic components. The designer needs to place the PCELLs, connect their terminals using metal layers, contacts, and vias. If the designer chooses, the software also offers the option to automatically place the PCELLs and do all of the routing; however, this can lead to a less compact design. Once the components are in place and fully routed, DRC is run to verify the circuit’s dimensions meet the fabricator’s requirements. If there are errors, adjustments are made to the design until all errors are gone. Then, LVS is run to compare the connections in the layout with the schematic. From here on, the design flow is the same as the automated flow. The layout parasitic values are extracted, and the extracted layout is simulated. Following this, the circuit is exported to a file for fabrication.

This process takes more time, but results in more customized circuits that meet the designer's needs. However, with the full-custom flow, more opportunities for customization are available.

3.2.3 Full-Custom Design Flow

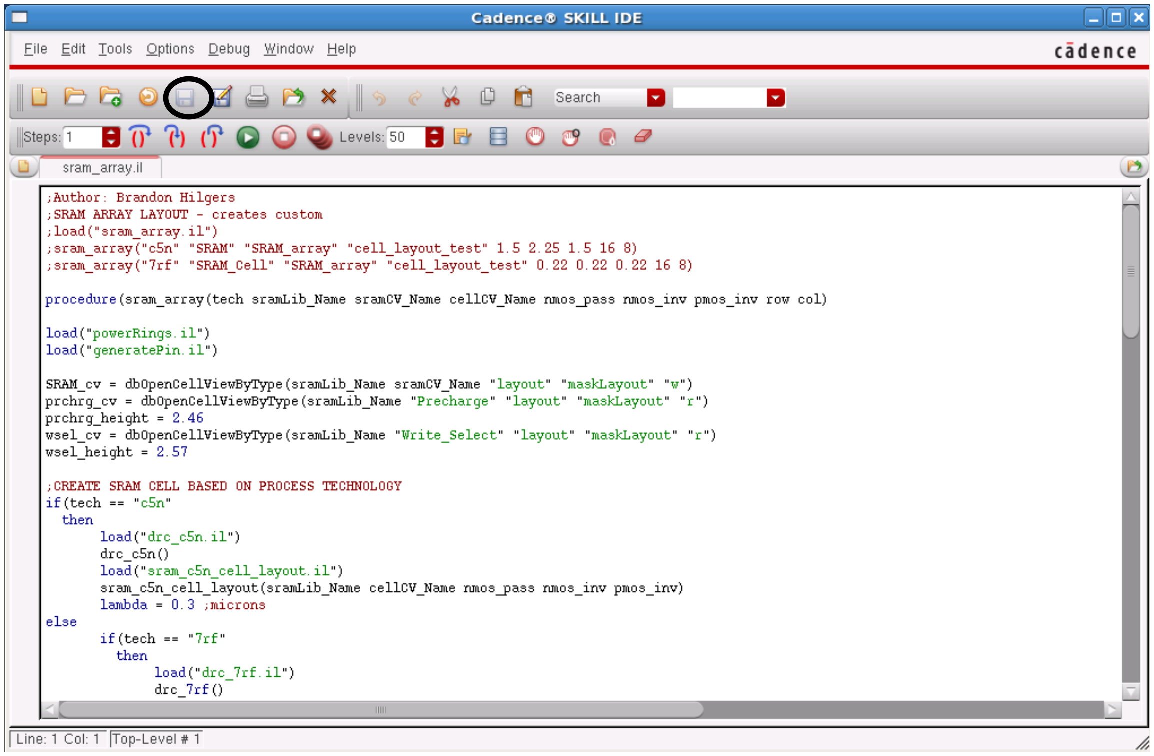
This design flow does not need much detail, because overall it is the same as the semi-custom design flow. The only difference between the full-custom and semi-custom flows is that no PCELLs are used in the full-custom flow, requiring the user to layout each component manually. This gives the designer more freedom, as long as DRC and LVS run without errors. Having a fully-customized circuit takes more time to develop, but can be more efficient in terms of speed, area, or power, and therefore can better suit the intended application.

3.3 SKILL IDE and SKILL Language

A major portion of the SRAM compiler design is centered on the SKILL IDE and SKILL programming language. Using Cadence's scripting language and the corresponding development tool was necessary to automating the SRAM cell design, SRAM array, and other peripheral circuits.

3.3.1 SKILL IDE

The development environment greatly assists in the design of SKILL functions and programs. The tool offers a GUI to write code in, a "Finder" to look up the available built-in SKILL functions, and a debugging tool. Figure 3.11 displays the SKILL IDE and some example code from this compiler design.



```
;Author: Brandon Hilgers
;SRAM ARRAY LAYOUT - creates custom
;load("sram_array.il")
;sram_array("c5n" "SRAM" "SRAM_array" "cell_layout_test" 1.5 2.25 1.5 16 8)
;sram_array("7rf" "SRAM_Cell" "SRAM_array" "cell_layout_test" 0.22 0.22 0.22 16 8)

procedure(sram_array(tech sramLib_Name sramCV_Name cellCV_Name nmos_pass nmos_inv pmos_inv row col)

load("powerRings.il")
load("generatePin.il")

SRAM_cv = dbOpenCellViewByType(sramLib_Name sramCV_Name "layout" "maskLayout" "w")
prchrg_cv = dbOpenCellViewByType(sramLib_Name "Precharge" "layout" "maskLayout" "r")
prchrg_height = 2.46
wsel_cv = dbOpenCellViewByType(sramLib_Name "Write_Select" "layout" "maskLayout" "r")
wsel_height = 2.57

;CREATE SRAM CELL BASED ON PROCESS TECHNOLOGY
if(tech == "c5n"
  then
       load("drc_c5n.il")
       drc_c5n()
       load("sram_c5n_cell_layout.il")
       sram_c5n_cell_layout(sramLib_Name cellCV_Name nmos_pass nmos_inv pmos_inv)
       lambda = 0.3 ;microns
else
       if(tech == "7rf"
         then
             load("drc_7rf.il")
             drc_7rf()
```

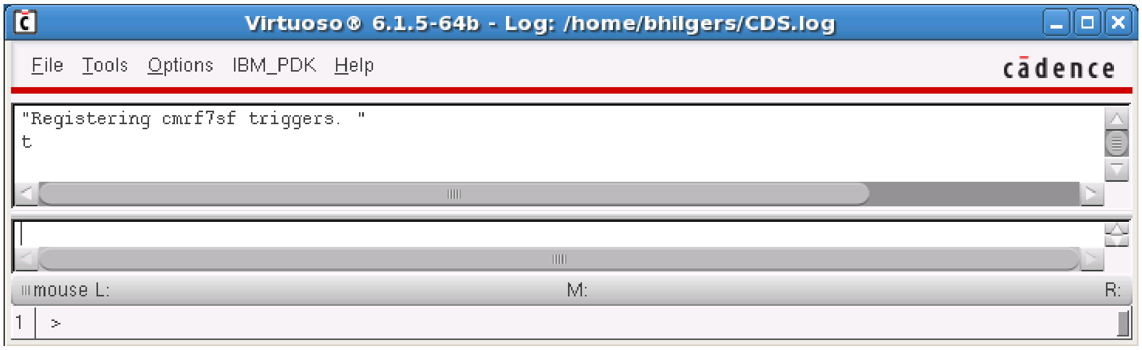
Figure 3.11: SKILL IDE with Example Code

Using VIM in a terminal or another text editor would also suffice for writing SKILL code. While not necessary, the SKILL IDE has the features previously listed, which ultimately decrease the time needed to write and verify SKILL code.

3.3.2 SKILL Language

The SKILL language is a Cadence-made scripting language based on the high-level LISP programming language [21]. The language carries characteristics of the C programming language, but also contains the higher-level functions of a scripting language catered to controlling and automating the multitude of Cadence software tools and their controls. This includes schematic and layout functions, graphics controls, and accessing, creating, and modifying database items and designs.

A summary of the primary SKILL functions pertinent to this compiler design is discussed here and in Chapter 4; however, it should be noted that the SKILL language and its long list of features as a very effective CAD development tool extend well beyond the needs of this project. Before examining SKILL functions, the SKILL file structure and how SKILL code is run should be mentioned first. Firstly, SKILL files are of the type ".il". Actions related to SKILL files are all done in the CIW (command interpreter window). This window is the first window that comes up when Virtuoso is run. The CIW serves a number of purposes including, but not limited to, controlling the design session, opening the file manager, opening the SKILL IDE, running individual SKILL commands, and running SKILL functions. The CIW is shown in Figure 3.12.



```
"Registering cmrf7sf triggers. "
t
```

Figure 3.12: Cadence Virtuoso CIW

Before a SKILL file can be run, it must be loaded. Loading the SKILL file checks each expression contained in the file and aborts if any errors occur [22]. A SKILL file can be loaded using either the load("SKILLfilename.il") command in the CIW, or by pressing the orange load button circled in Figure 3.11 while the SKILL file is open in the SKILL IDE. Once the file has been loaded without errors, the functions within the file can be called.

The functions in SKILL code are created with the procedure() function. An example code of using the procedure() function to create another function is shown below:

```
procedure( exampleFunction( x )
        x = x + 7
)
```

The created function in this case is called “exampleFunction,” which takes in one argument, ‘x’ and adds 7 to it. To run exampleFunction in the CIW with an input value of 3 (assuming it has been loaded first), the following command would be entered in the CIW:

```
exampleFunction(3)
```

This would result in x being equal to 10. Creating functions is the foundation of designing the automation portion of the compiler. Functions created for the compiler are built on other preexisting functions in the SKILL library. One function often used for the design of this compiler is the dbCreateRect() function. The dbCreateRect() function simply draws a rectangle of a chosen layer in layout. This allows for components and wires to be drawn so that the circuit can be built. This function’s arguments define which cell view (layout design in this case) that the rectangle should be drawn in, which layer to draw in (metal 1, poly, active, etc.), and the coordinates the rectangle occupies in the layout. This function and other commonly used ones will be discussed more in depth in the compiler design section of Chapter 4.

The next section, Chapter 4, discusses the details of the design of the SRAM compiler, including SRAM cell analysis, the SKILL-based compiler development, and the building of the precharge and address decoder circuits.

Chapter 4: SRAM Compiler Design and Implementation

This chapter details the design of an SRAM cell based on cell transistor sizes for stability, the layout of SRAM cells, SRAM arrays, precharge circuits, and power rings using SKILL code, and the layout of the precharge circuit and address decoder.

4.1 SRAM Cell Stability

SRAM stability for read and write operations is important to ensure that the cell won't be written to when attempting to read, and to ensure that the cell can still be written to during an intended write operation. The stability of the cell is determined by the ratios of the transistor sizes in the cell. The goal is to minimize cell area (transistor sizes) while maintaining cell stability. For this design, minimum gate channel lengths are assumed while the gate widths are varied. The following details the analyses for stable SRAM cell read and write operations in the IBM cmrf7sf and ON Semiconductor SCMOS processes.

4.1.1 Read Operation

First, the read operation is analyzed. For this analysis, a simplified SRAM cell schematic is used (Figure 4.1). With a Q value of 1 and Q_bar value of 0, the PMOS of the left inverter and the NMOS of the right inverter are turned off, and thus are removed from the diagram. Assuming that both bit lines are precharged to Vdd, it can be seen that the left bit line will decrease in voltage when the word line is high. Therefore, the analysis that is done here pertains to the left side of the schematic, and not the right. During the read operation, the M_1 and M_5 transistors discharge BL_bar down to 0. At the start of the read, however, the value on BL_bar pulls the node Q_bar up by some voltage defined by the sizes of M_5 and M_1 [15:658-661].

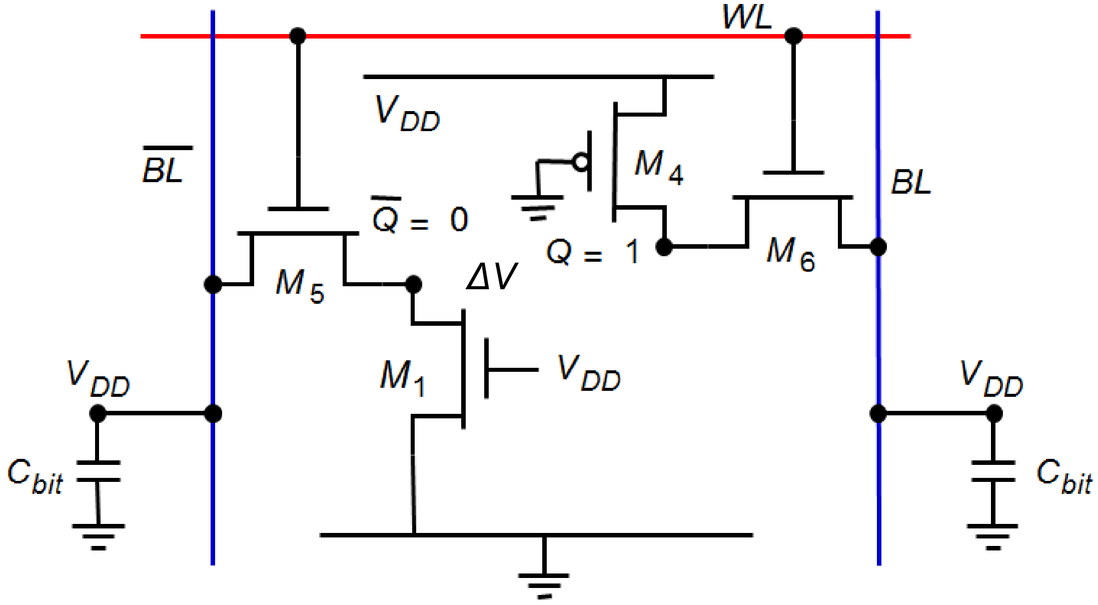


Figure 4.1: Simplified SRAM Cell for Read Stability Analysis [23]

The node ΔV (also Q_bar) can be solved for by setting the currents of M_1 and M_5 equal to each other [15:658-661]. The desired value for ΔV should be just below the NMOS threshold voltage, Vtn. This ensures that the NMOS of the right inverter won't turn on and try to pull node Q down to 0, which would start the positive feedback loop that overwrites the cell contents. With Q_bar at 0 V, M_1 is in linear mode operation, while M_5 is in saturation mode. With short channel devices such as the ones used in this analysis, velocity saturation occurs where the velocity of the carriers saturates due to collisions of the carriers [15:94-102]. This occurs at a voltage, V_{DSATn}, which is determined by a simplified equation using v_{sat}, the saturation velocity, μ_n, the electron mobility – both of which are set by the process technology – and L, the channel length:

$$V_{DSAT} = \frac{Lv_{sat}}{\mu_n} \qquad (4.1)[15:94-102]$$

Using Equation 4.1 and the current equations for M_1 and M_5, the resulting equation is:

$$k_{n,M5}\left((V_{DD}-\Delta V-V_{Tn})V_{DSATn}-\frac{V_{DSATn}^2}{2}\right)$$

$$= k_{n,M1}\left((V_{DD}-V_{Tn})\Delta V-\frac{\Delta V^2}{2}\right) \qquad (4.2)[15{:}658-661]$$

By solving Equation 4.2 for ΔV, the resulting equation below is found:

$$\Delta V = \frac{V_{DSATn}+CR(V_{DD}-V_{Tn})-\sqrt{V_{DSATn}^2(1+CR)+CR^2(V_{DD}-V_{Tn})^2}}{CR}$$

$$(4.3)[15{:}658-661]$$

Where CR (cell ratio) is the ratio of the transistor sizes of M_1 to M_5:

$$CR = \frac{W_1/L_1}{W_5/L_5} \qquad (4.4)[15{:}658-661]$$

Again, minimum channel lengths are used to decrease the overall area; therefore, Equation 4.4 effectively simplifies to the ratio of the transistor widths, W_1/W_5. Using Equations 4.3 and 4.4 with data acquired from MOSIS about process technology values for SCMOS [24], the plot shown in Figure 4.2 is made.

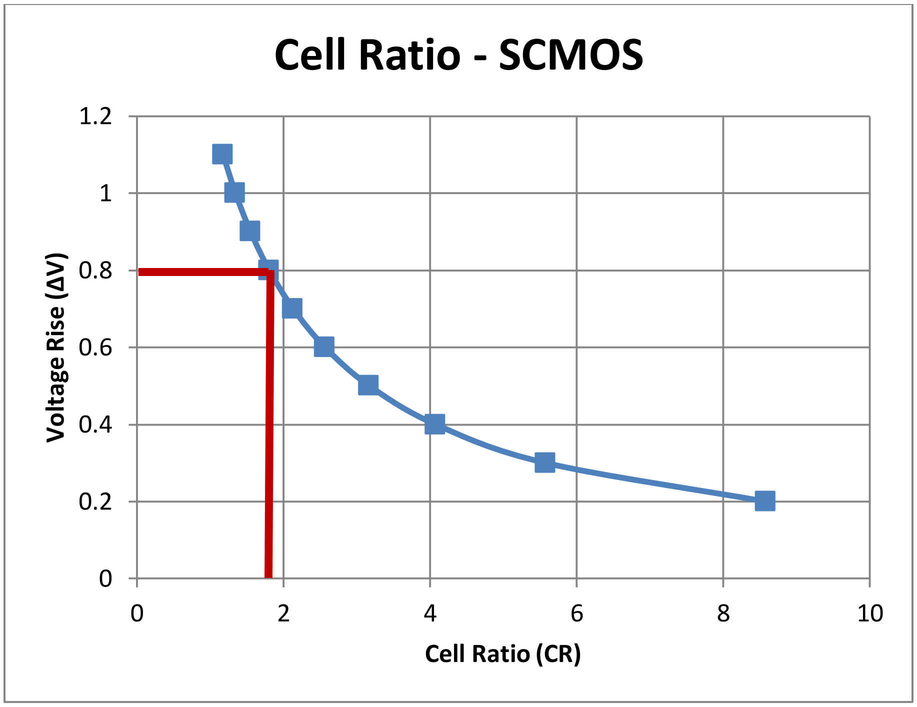


Figure 4.2: SCMOS Cell Ratios for Various Values of ΔV

Figure 4.2 shows that the cell ratio should be greater than 1.8 to ensure that ΔV never rises above the SCMOS NMOS threshold voltage of about 0.8 V. To lower ΔV further, the cell ratio can be increased; however, this leads to an increase in the size of M_1, consequently increasing the area of the cell, which is undesirable. Therefore, a cell ratio of about 1.8 is chosen. Because an SRAM cell is symmetrical, the M_1/M_5 ratio is the same as the M_3/M_6 ratio. M_1 and M_3 should match in size, and M_5 and M_6 should match in size.

Using Equations 4.3 and 4.4 with data acquired from MOSIS about process technology values for cmrf7sf [25], the plot shown in is Figure 4.3 created.

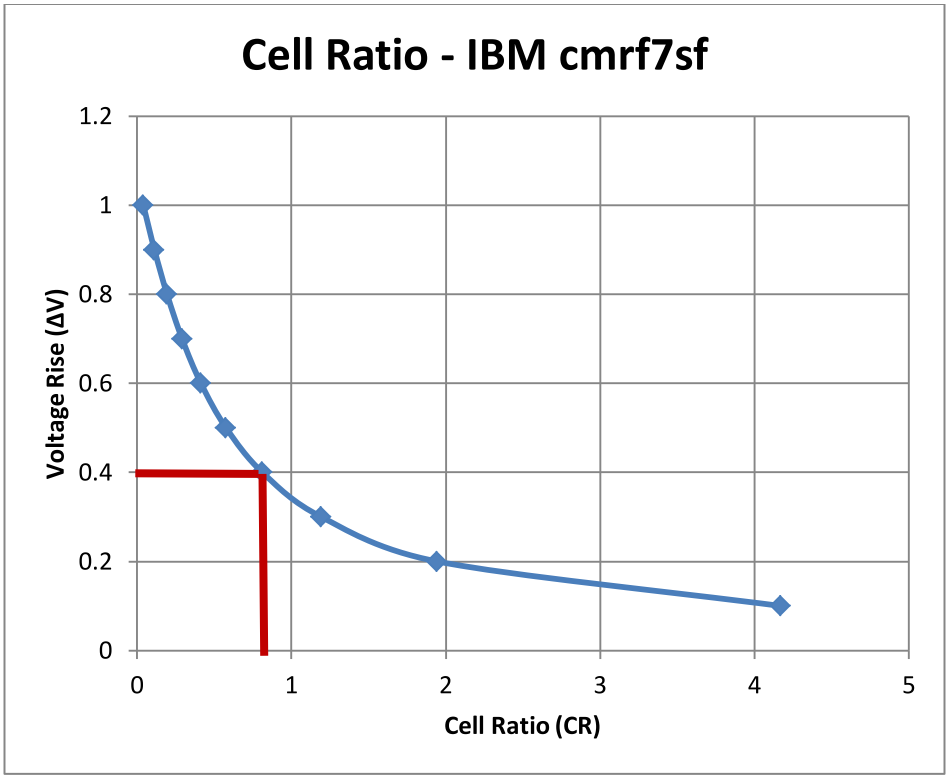


Figure 4.3: IBM cmrf7sf Cell Ratios for Various Values of ΔV

Figure 4.3 shows that the cell ratio for the IBM process should be greater than 0.8 to ensure that ΔV never rises above the cmrf7sf NMOS threshold voltage of about 0.4 V. Again, the same concept of increasing CR to lower ΔV applies here, but this does increase cell area to an extent. Due to the nature of the calculated data, this process actually allows for increasing the CR to 1 without increasing the cell area, because both M_1 and M_5 can have minimum sizes. In fact, having a CR less than 1 would most likely increase the cell area depending on the transistor sizes needed for an optimal write cycle (shown next). Therefore, a CR value of 1 is chosen.

4.1.2 Write Operation

An analysis for write stability is similar to the analysis done for the read operation. Figure 4.4 shows a simplified SRAM cell schematic during a write operation. With a value of 1 stored on node Q and a value of 0 stored on Q_bar, the NMOS of the right inverter and the PMOS of the left inverter are off, and thus are removed from the diagram. In this write operation, a value of 0 is to be written to the cell (node Q). Therefore, BL has a value of 0 and BL_bar has a value of 1. Because the M_5 and M_1 transistor sizes were chosen in the previous analysis for the read operation so that node Q_bar cannot rise above ΔV, the cell must be written to on the right side (BL) by pulling node Q down to 0 through the M_6 pass transistor. When WL is set high, the difficulty of pulling node Q down to 0 comes from the fact that the PMOS of the right inverter, M_4 is pulling node Q up to V_{DD}. While this analysis assumes that the left inverter (M_1 NMOS and M_2 PMOS) is not involved in the write operation, the positive feedback nature of the SRAM cell in reality causes this transistor pair to assist in overwriting the cell as node Q is lowered through the M_6 pass transistor.

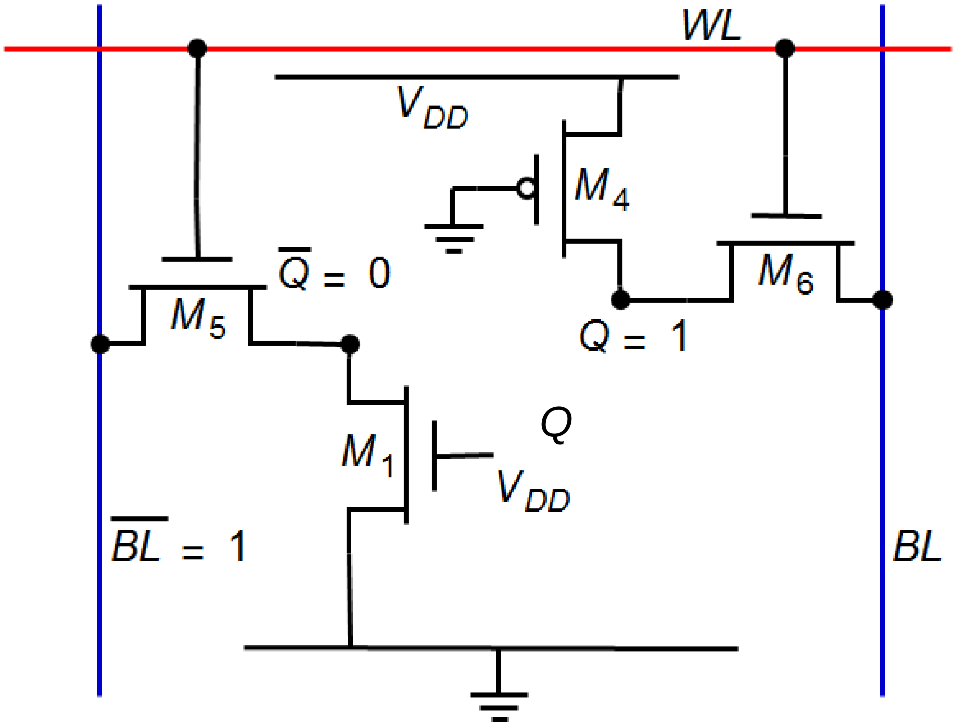


Figure 4.4: Simplified SRAM Cell for Write Stability Analysis [23]

By setting the M_4 and M_6 current equations equal to each other, these transistors can be sized so that node Q can be pulled down low enough to overwrite the cell [15:658-661].When node Q is some low voltage, V_Q, the cell can be overwritten. In this state, M_4 is in saturation mode, and M_6 is in the linear mode of operation. Again, factoring in velocity saturation, the resulting equalized current equation is shown below:

$$k_{n,M6}\left((V_{DD}-V_{Tn})V_Q-\frac{{V_Q}^2}{2}\right)$$

$$= \; k_{p,M4}\left(\left(V_{DD}-\left|V_{Tp}\right|\right)V_{DSATp}-\frac{V_{DSATp}^2}{2}\right) \quad (4.5)[15\!: 658-661]$$

By solving Equation 4.5 for V_Q, the resulting equation is shown below:

$$V_Q = \; V_{DD}-V_{Tn}-\sqrt{(V_{DD}-V_{Tn})^2-2\frac{\mu_p}{\mu_n}\mathrm{PR}\left(\left(V_{DD}-\left|V_{Tp}\right|\right)V_{DSATp}-\frac{V_{DSATp}^2}{2}\right)}$$

$$(4.6)[15\!: 658-661]$$

Where PR (pull-up ratio) is the ratio of the transistor sizes of M_4 to M_6:

$$PR = \frac{W_4/L_4}{W_6/L_6} \qquad (4.7)[15\!: 658-661]$$

By using minimum channel lengths, Equation 4.7 effectively simplifies to the ratio of the transistor widths, W_4/W_6. Using Equations 4.6 and 4.7 with data acquired from MOSIS about process technology values for SCMOS [24], the plot shown in Figure 4.5 is made.

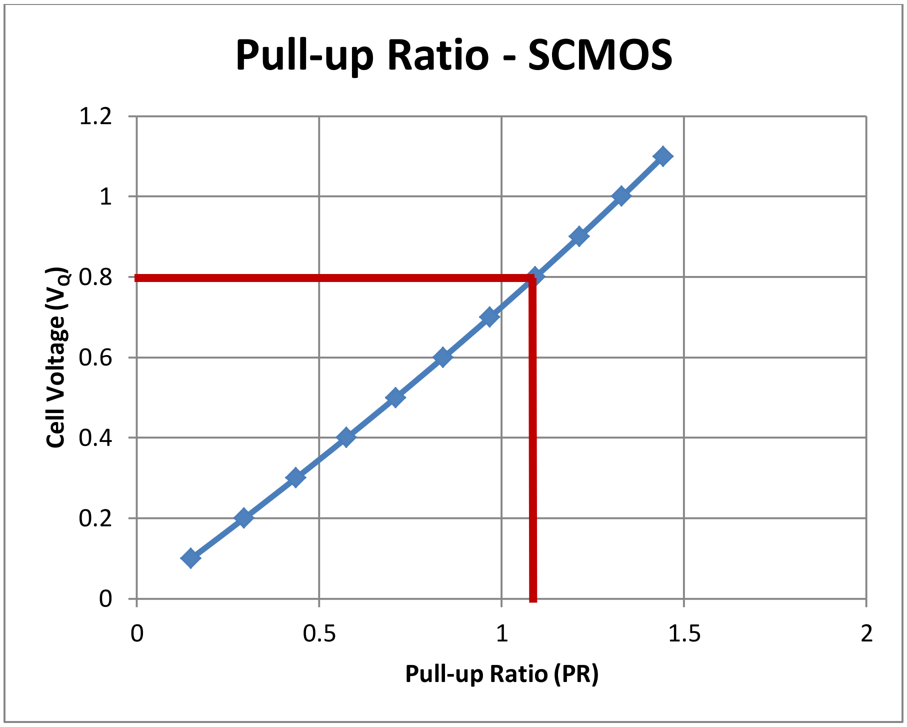


Figure 4.5: SCMOS Pull-up Ratios for Various Values of V_Q

Figure 4.5 shows that in order to properly write to the cell by lowering V_Q below the SCMOS NMOS threshold voltage of 0.8 V, a pull-up ratio of less than 1.1 is needed. Having minimally-sized M_4 and M_6 transistors, which leads to a PR value of 1, is more than sufficient to allow for writing to the cell. A stronger write operation can be had with an even lower pull-up ratio; however, the disadvantage of this is the increased cell area due to the enlarged M_6 transistor width. Therefore, a PR of 1 is chosen for this design. Again, because an SRAM cell is symmetrical, the M_4/M_6 ratio is the same as the M_2/M_5 ratio. M_2 and M_4 should match in size, and M_5 and M_6 should match in size.

Using Equations 4.6 and 4.7 with data acquired from MOSIS about process technology values for cmrf7sf [25], the plot shown in is Figure 4.6 created. It should be noted that for this process, an altered version of Equations 4.5 and 4.6 is actually used. Under the unified MOS model for manual analysis, a more accurate method is applied by using the minimum value of three parameters (V_{GT}, V_{DS}, and V_{DSAT}) in all places that V_{DSAT} is used in Equations 4.5 and 4.6

[15:94-102]. The unified model is applied specifically to this process for this operation, because V_{DSAT} is found not to be the minimum among the three previously mentioned parameters. In Figure 4.6, for values of V_Q from 0.1 V to 0.4 V, the V_{GT} parameter is used to calculate PR in place of V_{DSAT}, where V_{GT} is V_{GS}-Vtp, or the gate-source voltage of the PMOS minus the PMOS threshold voltage. For values of V_Q from 0.5 V and up, V_{SD} (source-drain voltage) of the PMOS is used.

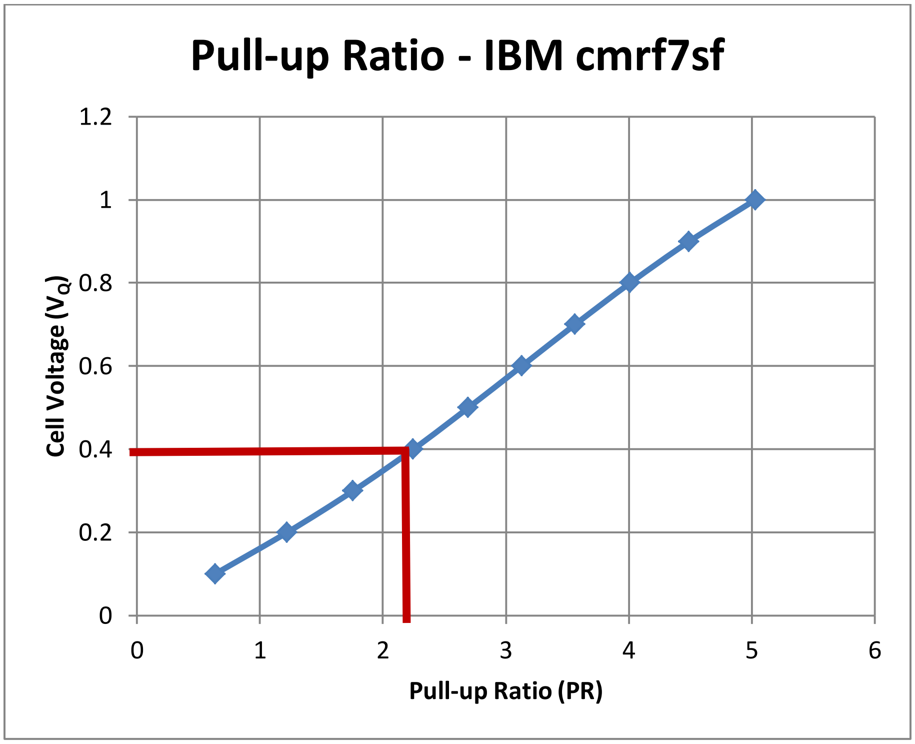


Figure 4.6: IBM cmrf7sf Pull-up Ratios for Various Values of VQ

Figure 4.6 shows that a PR of less than 2.25 should be used for an optimal write operation. Again, using minimally-sized transistors proves advantageous on two fronts, by decreasing the cell area and improving the write operation (decreased pull-up ratio). Therefore, a PR of 1 is chosen for this design.

Based on all of the previous analysis discussed, the final chosen transistor sizes are shown in Table 4.1. All transistors have minimal channel lengths (180 nm for cmrf7sf and 600 nm for SCMOS). Because the SRAM cell is symmetrical between each side of the cell, the sizes listed in Table 4.1 are for two transistors of each type. There are two pass NMOS transistors, two inverter NMOS transistors, and two inverter PMOS transistors. Based on the chosen sizes then, the two inverter NMOS transistors of the SRAM cell for the SCMOS process technology have widths of 2.7 µm and lengths of 0.6 µm.

Table 4.1: Chosen SRAM Cell Transistor Sizes for Stable Read and Write Operations

Process Technology	**IBM cmrf7sf (180nm)**		**ON Semiconductor SCMOS (600nm)**	
Transistor Parameter	Channel Width (µm)	Channel Length (µm)	Channel Width (µm)	Channel Length (µm)
Pass NMOS	0.22 (min.)	0.18 (min.)	1.5 (min.)	0.6 (min.)
Inverter NMOS	0.22	0.18	2.7	0.6
Inverter PMOS	0.22	0.18	1.5	0.6

Chapter 5 shows simulations verifying the function of these SRAM cells with their chosen transistor sizes. Next, a list of the SKILL functions used to create the SRAM compiler for automated layout is discussed, as well as the progression of the program when it is run.

4.2 SRAM Compiler Design

The major part of this SRAM compiler comes from the design of the SKILL files that generate the SRAM layouts. This section describes in more detail than previously discussed, the main SKILL functions used to develop the compiler, the design of the SRAM cell generation, SRAM array generation, placing precharge blocks, and laying out power and ground rings and rails.

4.2.1 SKILL Layout Functions

The main SKILL functions used in this design involved manipulating cell view layouts through database objects, creating patterns for various material layers, placing parameterized cells, and creating nets, pins, and terminals. These main SKILL functions are summarized in Table 4.2.

Table 4.2: Summary of Commonly Used SKILL Functions

SKILL Function	Purpose
dbOpenCellViewByType()	Open cell views (layouts, schematics, etc.) for editing or reading
dbCreateRect()	Draws a rectangle of the desired layer in a layout
dbCreateInst()	Makes a copy of a preexisting layout or component for placement in the specified layout
dbCreateNet()	Creates a net in the specified cell view
dbCreatePin()	Creates a pin that corresponds with a specified net and node within a layout
dbCreateTerm()	Creates a terminal that defines whether a net is an input, output, etc.

The first function that is discussed is the dbOpenCellViewByType() function. Layouts in the Cadence software are created in cell views that are referenced as database objects. These cell views can be created, edited, deleted, or used in other cell views using the dbOpenCellViewByType() function. Once a cell view is opened or created, different material layers can be added to draw the desired circuit. The cell views created and edited in this SRAM compiler are the layout for the generated SRAM cell, and the layout of the final SRAM array with precharge circuits and power and ground rings and rails.

The next important function is dbCreateRect(). This function is used to draw the physical layout of the circuit by manually drawing wires and component layers in specified locations. A rectangle of a layer is created by declaring which cell view to draw the rectangle in, choosing the desired layer (metal 1, metal 2, contact, polysilicon, etc.), and then listing the coordinates of the lower left and top right corners of the rectangle.

Another function used is dbCreateInst(). This function allows preexisting cell views to be placed in other cell views. By using existing layouts, the design process can be simplified. This method is applied in creating the SRAM array by using the generated layout of the SRAM cell and creating many instances of it in the final layout cell view.

The final three functions used involve the creation of input and output pins for the layout. These functions are dbCreateNet(), dbCreatePin(), and dbCreateTerm(). Pins and nets allow for a schematic symbol to be created from the layout, which can then be used to interface with other designs in a schematic, forming a larger circuit made up of circuit blocks. This schematic can then be turned into a layout where routing between the circuit blocks is automatic. Without pins and nets, there would be not automatic routing, slowing down the design process. The dbCreateNet() function is used to create a net database object. However, this net is not very useful if it is not attached to a piece of the layout. To attach the net to a node of the circuit, a rectangle of the corresponding layer must be created using the dbCreateRect() function. Next, the net needs to have an input/output status. The dbCreateTerm() function creates a terminal that specifies whether a net is an input, output, etc. Finally, the dbCreatePin() function creates a pin using the previously made rectangle and the net. The following shows a general example of how to create a net and pin.

```
pin_rect = dbCreateRect(pin_cv list(layer) list(pin_min_x:pin_min_y,
                              pin_max_x:pin_max_y))

net = dbCreateNet(pin_cv pinName)

term = dbCreateTerm(net "" termType)

pin = dbCreatePin(net pin_rect pinName)
```

All of these built-in SKILL functions for creating Cadence layouts are used extensively to create the various parts of the SRAM compiler. These pieces are the SRAM cell layout, the SRAM array, adding copies of manually-made precharge to each of the columns in the array, making power and ground rings, making power and ground rails, and creating pins and nets for the input and output nodes of the circuit. The next sections discuss the compiler design overview and the design of each of the compiler blocks.

4.2.2 SRAM Compiler Overview

Before describing the compiler steps in detail, an overview of the whole compiler is given, including how to run the compiler. The compiler has a function-based level of control, meaning that a function with the desired input arguments needs to be entered to run the compiler. While in the Virtuoso CIW (see section 3.3.2), the SRAM compiler SKILL file containing the compiler function is loaded by typing in load("sram_compiler.il"). After this, the sram_compiler() function can be entered with the chosen parameters.

There are several input arguments to the sram_compiler() function. The syntax for running the compiler is as follows:

sram_compiler(*Tech Library SRAM_name Cell_name NMOS_pass NMOS_inv PMOS_inv Words Word_size*)

The first input to the compiler defines what process technology to design the SRAM in. The options for this input are "7rf" for cmrf7sf and "c5n" for SCMOS. The next three inputs are names for the library, SRAM block layout, and SRAM cell layout, respectively. The user can choose any name, but the library must be an existing library. The next three inputs define the widths of the SRAM cell transistors in µm. These sizes can be any multiple of 0.01 µm with a minimum of 0.22 µm for cmrf7sf, and any multiple of 0.15 µm with a minimum of 1.5 µm for SCMOS. To use the default values from Table 4.1, a value of 0 must be entered for all three transistor width inputs. If only one or two of these inputs has value of 0, the compiler will not create the SRAM and an error message will be shown in the CIW. Finally, the last two inputs are the number of words (or addresses or rows) and the size of the words (number of bits or columns). These inputs to the compiler and their significance are summarized in Table 4.3.

Table 4.3: SRAM Compiler Input Arguments Summary

Input Argument	Description	Input Options
Tech	Process Technology	“7rf” or “c5n”
Library	Library name of SRAM layouts	Any desired name for an existing library, e.g., “SRAM”
SRAM_name	Name of final SRAM block layout	Any desired name, e.g., “SRAM_block”
Cell_name	Name of SRAM cell layout	Any desired name, e.g., “SRAM_cell”
NMOS_pass	Width of SRAM cell pass NMOS transistor in µm	For 7rf: Any multiple of 0.01µm, greater than or equal to 0.22µm For c5n: Any multiple of 0.15µm, greater than or equal to 1.5µm For default sizes in both technologies, use 0
NMOS_inv	Width of SRAM cell inverter NMOS transistor in µm	For 7rf: Any multiple of 0.01µm, greater than or equal to 0.22µm For c5n: Any multiple of 0.15µm, greater than or equal to 1.5µm For default sizes in both technologies, use 0
PMOS_inv	Width of SRAM cell inverter PMOS transistor in µm	For 7rf: Any multiple of 0.01µm, greater than or equal to 0.22µm For c5n: Any multiple of 0.15µm, greater than or equal to 1.5µm For default sizes in both technologies, use 0
Words	Number of words/address/rows in the SRAM array	Any positive, even integer
Word_size	Number of bits in a word (or number of columns in SRAM array)	Any positive integer

An example of the compiler function with input arguments is shown below:

```
sram_compiler(“7rf” “SRAM” “SRAM_block” “SRAM_cell” 0 0 0 256 8)
```

The output of this example is an SRAM cell layout with the default transistor sizes in the cmrf7sf process technology, and an SRAM block layout of 256x8 SRAM cells with the default transistor sizes, manually-made precharge circuits, and power and ground rings and rails.

To create these two layouts, the SRAM compiler goes through several steps. First, after the compiler function is entered, the compiler checks if the inputs are valid. For example, the transistor widths have minimum sizes and are limited to multiples of a certain dimension, as specified in Table 4.3. If any of these inputs are invalid, the compiler stops the creation of the SRAM and shows an error message in the CIW. If the inputs are valid, then an SRAM cell layout is generated in the desired process technology with the desired transistor sizes. The compiler then creates a new layout and draws power and ground rings and rails to fit the SRAM array of the chosen size (number of words and word size). Next, in the same layout, copies of the generated SRAM cell are laid out to form the array. The compiler then makes copies of a manually-made precharge circuit and places them on top of the array to form a row of precharge circuits. Finally, input and output pins for the word lines, bit lines, power, and ground are added to the layout, completing the SRAM block layout. These steps are shown in the SRAM Compiler flow diagram in Figure 4.7.

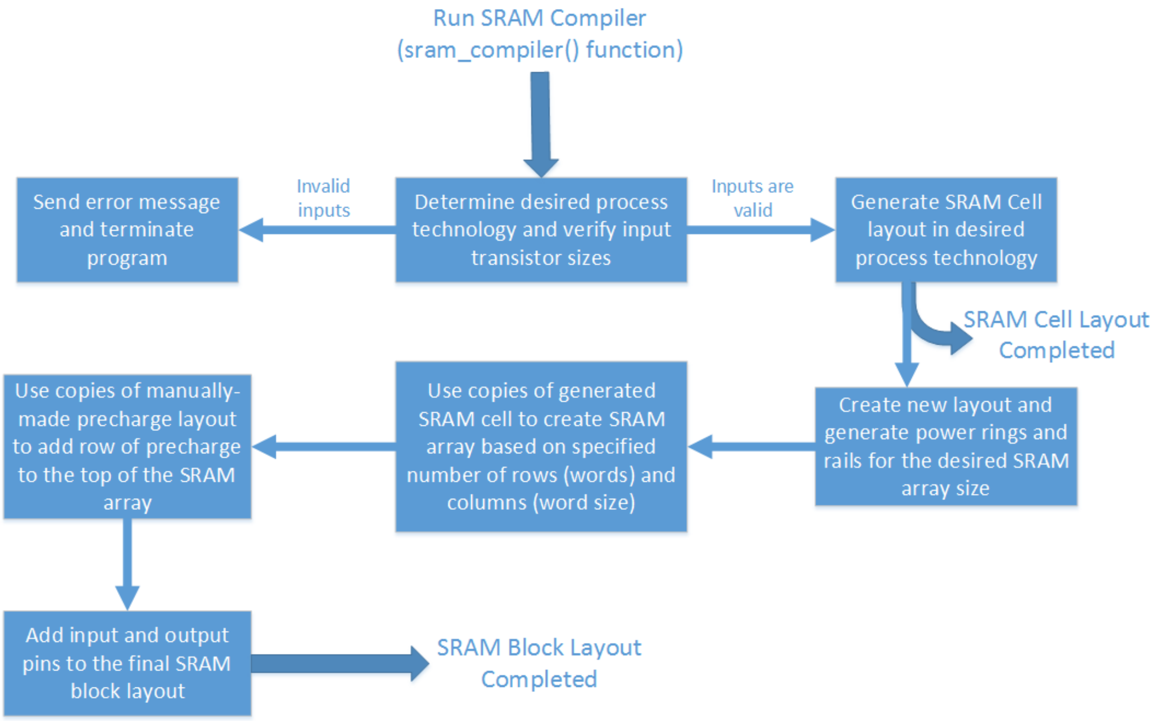

Run SRAM Compiler
(sram_compiler() function)
Send error message and terminate program
Invalid inputs
Determine desired process technology and verify input transistor sizes
Inputs are valid
Generate SRAM Cell layout in desired process technology
SRAM Cell Layout Completed
Use copies of manually-made precharge layout to add row of precharge to the top of the SRAM array
Use copies of generated SRAM cell to create SRAM array based on specified number of rows (words) and columns (word size)
Create new layout and generate power rings and rails for the desired SRAM array size
Add input and output pins to the final SRAM block layout
SRAM Block Layout Completed

Figure 4.7: SRAM Compiler Flow Diagram

These steps have several details and design considerations to be discussed. The next two sections describe these compiler steps in more detail.

4.2.3 SRAM Cell Generation

The SRAM cell generation portion of the compiler creates a layout of one SRAM cell to be used later in the SRAM array. The following describes the design process and design considerations for the generated SRAM cell layout using SKILL code. See Appendix A for all SRAM Compiler SKILL code.

There are a few goals for this SRAM cell and compiler design. One goal for this compiler is to make the SKILL code be as modular as possible so that other process technologies can be added simply by including the necessary DRC rules and the minimum transistor sizes. The cell design should also be as compact as possible. Using DRC rules allows for this compact design. Power and ground rails should be placed at the top and bottom of the cell respectively and should span the entire width of the cell for simple connections with adjacent SRAM cells in the array.

Similarly, the bit lines and word line should extend to the ends of the SRAM cell area – the bit lines should span the entire height of the cell and the word line should span the entire width of cell – ultimately allowing for automatic inter-cellular routing among adjacent cells in the array. Meeting the previous two goals should lead to the extremities of the SRAM cell forming a rectangle so that multiple cells make a rectangular array when placed together. Finally, when running the cell generation portion of the compiler, the user can specify custom transistor widths, or the default values chosen from Table 4.1 can be used.

When designing an SRAM cell for different process technologies, the main differences are the Design Rule Check (DRC) rules. By assigning the locations of wires and components based on DRC-valid distances from adjacent wires and components, the design can be made as modular as possible. Through writing the SKILL code for the SRAM cell generation and executing it for both process technologies, it is found that the design cannot be entirely modular. While much of the design transfers well between processes, there are still many places where design is unique to each process. By attempting to create a compact design, it is revealed that one constraining DRC rule (minimum distance between two layers for example) for a given process is not the same constraining DRC rule in another process. If this is ignored, then DRC will give several errors for the generated cell in one of the process technologies. To ensure that DRC is met with no errors, each process must be designed uniquely in these instances where the limiting DRC rules are not the same. Another cause for not having a completely modular design is the different layers used. While the majority of the layers are used in both processes, each process has one or two unique layers that the other does not have. This leads to additional DRC rules that need to be accounted for in one process but not the other. Because of this, this design cannot be made modular for all process technologies. Therefore, adding new process technologies in the future is a more complex procedure than just adding DRC rule values for the new process.

Because the implementation for the two processes is unique in several ways, the compiler for the cell generation is split into two functions: sram_7rf_cell_layout() and sram_c5n_cell_layout() (see Appendix A). The procedure is the same for both functions. Each starts with five input arguments: SRAM cell layout library name and cell view name, and the widths of the three SRAM cell transistor types (pass NMOS, inverter NMOS, and inverter PMOS). The functions then call their respective DRC rule functions. These DRC rule functions create the variables that the SRAM cell layout functions use to properly layout the circuit. These DRC values could be hardcoded into the cell layout functions, but in an attempt to make the compiler design modular and easy to change, variables are used instead. The functions then proceed to layout all of the components, power and ground rails, and connections starting from the bottom left of the cell and ending at the upper right. When running either of the functions, custom values can be used for the transistor widths, or if the user enters " " for all three values, then the default transistor widths from Table 4.1 are used. Running the 7rf (cmrf7sf) and c5n (SCMOS) cell layout functions with the default transistor widths result in the SRAM cell layouts shown in Figure 4.8 and Figure 4.9, respectively.

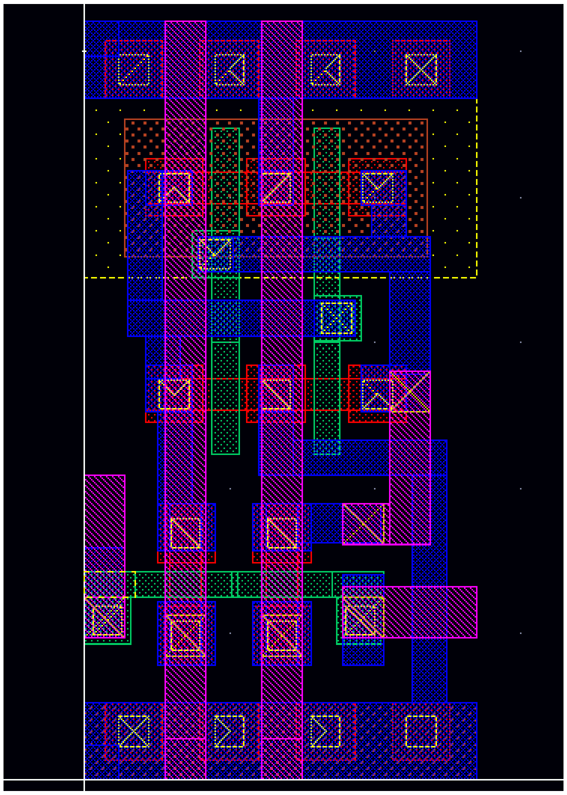


Figure 4.8: cmrf7sf (180nm) SRAM Cell Layout with Optimal Transistor Sizes

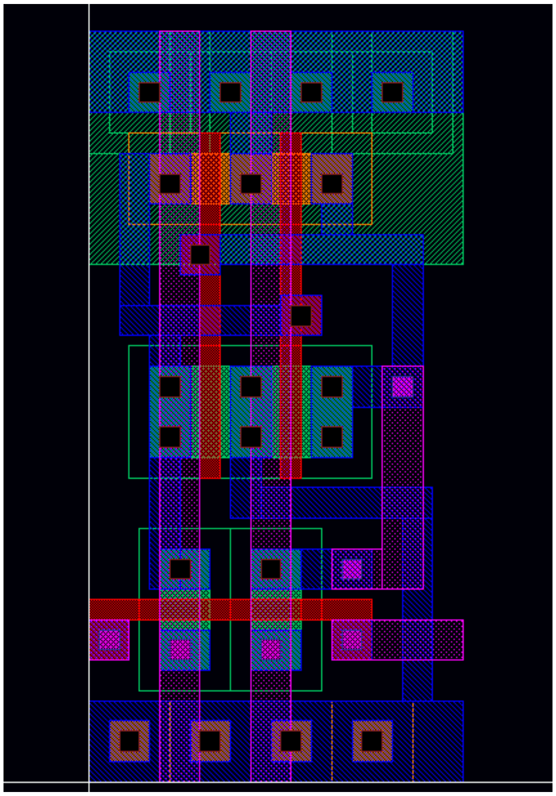


Figure 4.9: SCMOS (600nm) SRAM Cell Layout with Optimal Transistor Sizes

When looking at a layout of the SRAM cell, it is helpful to see labels of key features of the design. Figure 4.10 shows an annotated SRAM cell layout. The labeled image reveals the locations of the supply and ground rails, inverter PMOS and NMOS transistors, and the pass NMOS transistors.

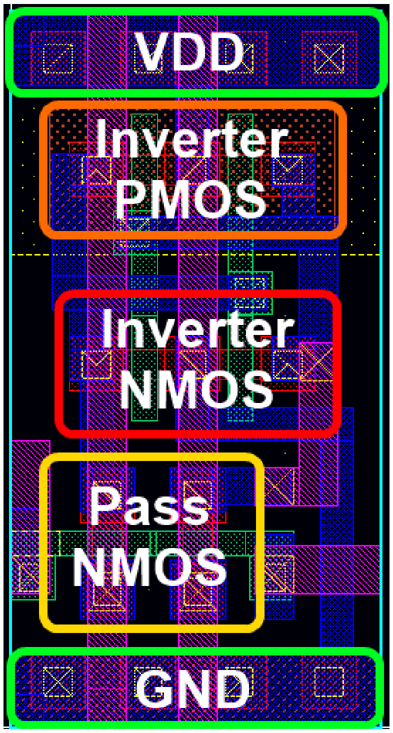


Figure 4.10: Labeled SRAM Cell Layout

One important feature to take note of is how the supply and ground rails cover the top and bottom of the cell, respectively, and span the whole width of the cell. This ensures that the supplies and grounds are locally connected to each other when multiple SRAM cells are placed next to each other to form an array. This is shown in section 4.2.4.

After the SRAM cell layout is generated, the next step is to use the cell to create an array. This is discussed in the next section where the full SRAM block is created.

4.2.4 SRAM Block Generation

The SRAM block generation includes the layout of an SRAM array, precharge circuits for each column of the SRAM array, and power and ground rings and rails for even power distribution. This section shows the design of the portion of the compiler that ties all of the pieces of the SRAM design together.

A high level function, sram_compiler() (see Appendix A), is made for the user to generate an SRAM block. This function does a number of actions including calling the SRAM cell layout function corresponding to the desired process technology, laying out the cell multiple times to create an array of the desired size, laying out multiple precharge circuits for each column of the array, calling another function to create power rings and rails, and ground rings and rails, and finally adding nets and pins to the inputs and outputs of the circuit.

The first operation the compiler executes after calling the cell layout function and generating an SRAM cell is creating an array using that cell in a new layout. The user specifies the number of rows and columns to have in the array. This array is created by opening the cell view database object for the generated SRAM cell, and then adding new instances of the cell in the new layout. The SRAM cells are laid out sequentially from left to right and bottom to top. An important design feature of the array forming portion of the SRAM compiler is that it vertically flips (about the x-axis) every other row of SRAM cells. Doing this connects the supply and ground rails of adjacent rows, and consequently avoids shorting any supply rails to ground rails. Figure 4.11 shows a 4x3 SRAM array with the middle row flipped, connecting its ground rail to the ground rail of the row above, and its supply rail to the supply rail of the row below. When more rows are added in the array, the trend of flipping every other row continues.

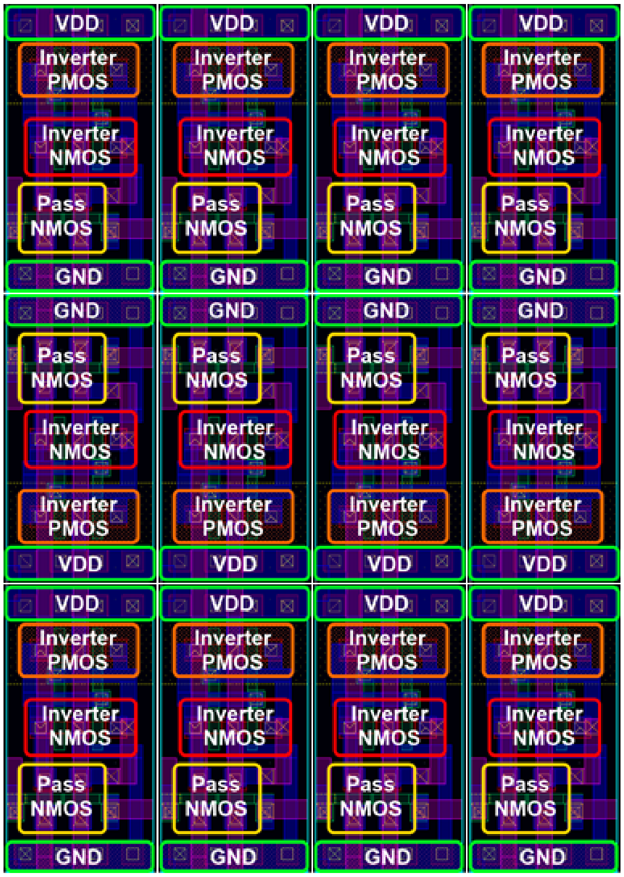


Figure 4.11: Labeled SRAM Array Showing Vertically Flipped Row

Although the power and ground rails of adjacent rows are connected, the power and ground of rows farther apart in the same array are not connected. This is resolved by creating power and ground rings, and then connecting the power and ground rails of the array to the rings. This is done in an earlier step of the sram_compiler() function.

First, the compiler lays out the power and ground rings around the SRAM array. For the rings, the vertical rectangles are made in the metal 2 layer, and the horizontal rectangles are made in the metal 1 layer. These horizontal and vertical rectangles connect through contacts, creating even distribution of power around the array. Power and ground rails (metal 1) then run horizontally across the power and ground rails of the array, connecting with the vertical metal 2 rectangles of the power and ground rings, respectively. A labeled image of this power distribution routing is shown in Figure 4.12.

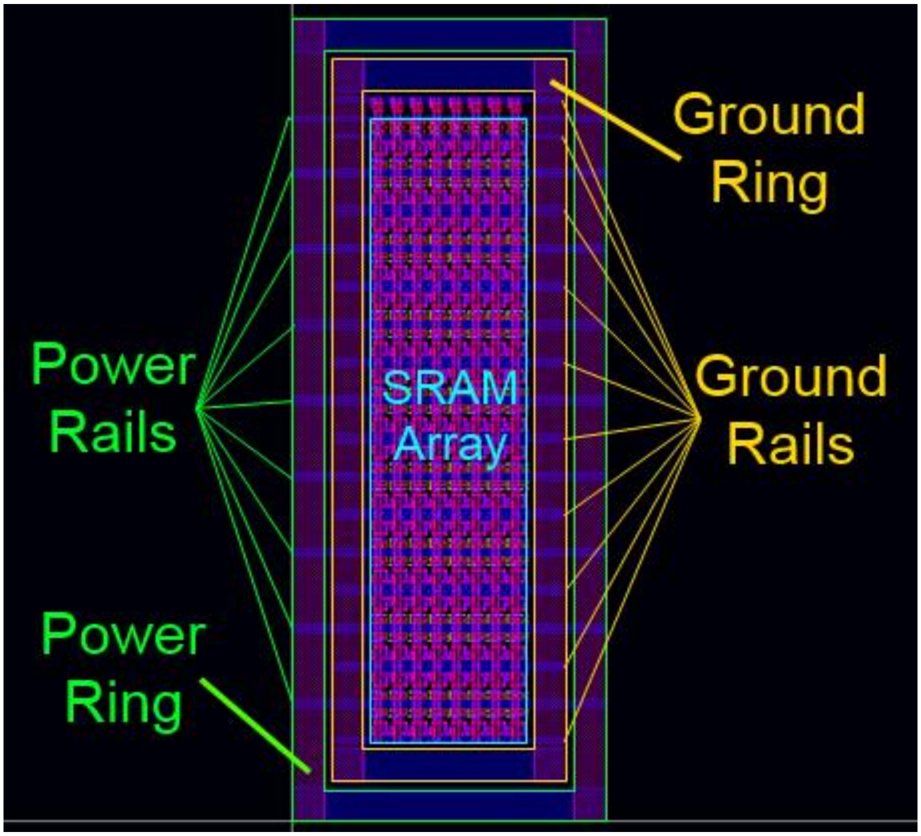


Figure 4.12: 16x8 SRAM Array with Labeled Power and Ground Rings and Rails

Figure 4.12 shows the a power ring on the outside, with a ground ring inside of it, and a 16x8 SRAM array within that. The SRAM supply and ground rails (metal 1) extend horizontally to the left and right to connect with the vertical rectangles (metal 2) of the power and ground rings.

One last operation the sram_compiler() function does, besides adding input and output nets and pins to the SRAM block, is placing precharge circuits above the top row of the SRAM array. The precharge circuits are laid out from left to right in the same manner as the SRAM cells are laid out. The cell view layout database object for the precharge is opened, and then multiple instances of the layout are added to the SRAM block cell view in the desired locations. The resulting layout of the precharge across the top of the SRAM array is shown in Figure 4.13. The design of the precharge layout and the chosen transistor sizes is shown in the next section.

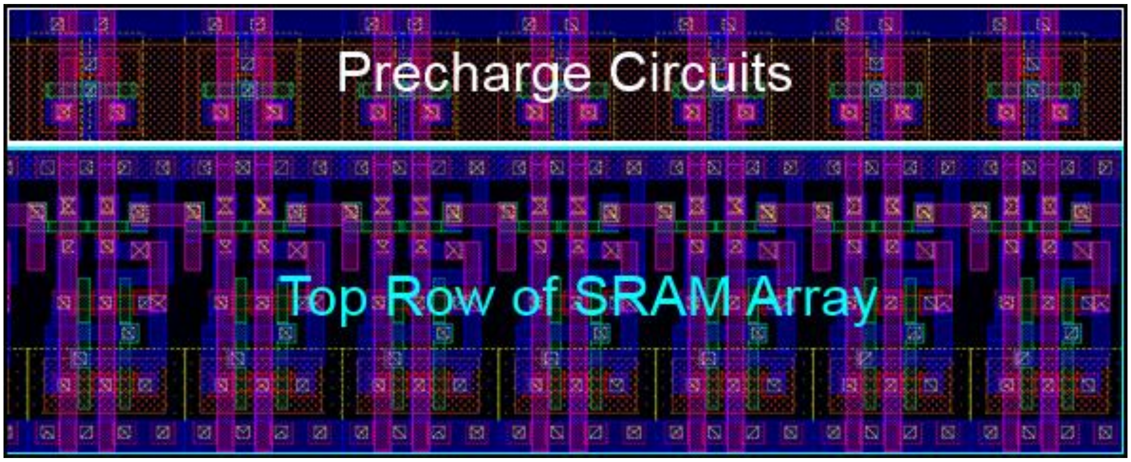


Figure 4.13: Labeled Precharge Layout Above SRAM Array

The next section discusses the details of the precharge circuit design.

4.3 Precharge Circuit Design

The precharge circuit has several design considerations in terms of transistor sizes and physical layout. Because the precharge design for the SRAM compiler is laid out manually, the design is limited to one type and is not variable based on the size of the SRAM cell or SRAM array. Therefore, this design is based on an SRAM array design that has SRAM cell transistors sizes matching those from Table 4.1.

One design consideration for the precharge circuit is the size of the PMOS pull-up transistors. The size of these transistors determines the rate at which the bit lines can be pulled up to V_{DD} and also how low one of the bit lines can go when doing a read operation. Ideally, one bit line would drop to 0 V, but because the precharge is continually on during the read operation, and there is no sense amplifier in this SRAM design, the size of the precharge PMOS transistors must be adjusted to give a low bit line voltage that is sufficient. A sufficient voltage in this case would be just below the NMOS threshold voltage. If for example, an inverter is attached as an output of the bit line, the defined low voltage will ensure that the NMOS of the inverter will be off, making the output of the inverter high. If the bit line low voltage is not set below the NMOS threshold voltage, there is a chance the inverter will see that voltage as a logical 1, and produce an output of

0, which would be incorrect. On the high bit line side, the logic is correct no matter the size of the transistors because they are PMOS transistors. Because the chosen topology from Figure 2.7 uses PMOS transistors, one of the bit lines is held high at V_{DD} during the read operation, which is a true logical 1 output. If NMOS transistors were used in the precharge circuit, the high bit line would be less than a PMOS threshold voltage below V_{DD}, making it not a true logical 1 output. Using the chosen topology fixes this issue.

The size of the precharge PMOS transistor that gives the desired low bit line voltage is determined by the voltage divider formed by the precharge PMOS, the SRAM cell inverter NMOS, and pass NMOS transistors on the low bit line side. Each transistor has some amount of resistance that is defined by its size. Increasing the width of the transistor decreases the resistance, while increasing the length increases the resistance. The voltage divider formed by the precharge PMOS transistor and the two SRAM cell NMOS transistors on the low bit line side is shown in Figure 4.14.

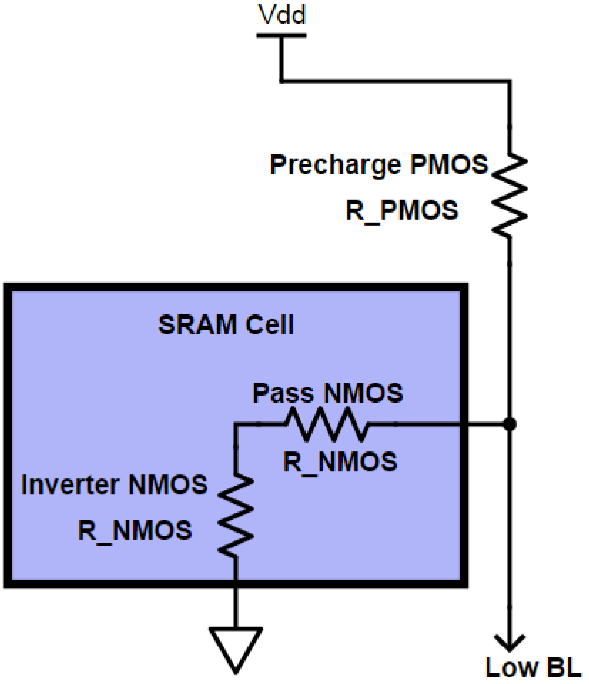


Figure 4.14: Voltage Divider Formed by Precharge Transistor and SRAM Cell NMOS Transistors on Low Bit Line Side

With the size of the NMOS pass transistors set from Table 4.1, the size of the precharge PMOS transistors are adjusted in simulation for a read operation. While the testing is only done for one side, because only one bit line goes down during a read operation, the precharge transistors are sized equally for symmetry. This accounts for either bit line dropping to a low voltage during a read operation. The size of the precharge transistors are first chosen to be minimally sized (minimum width and length) to reduce the area, and this produces a low bit line voltage that is above the NMOS threshold voltage. Therefore, the resistance of the PMOS transistor needs to be increased. Because keeping the area small is important, the resistance can be increased by keeping the channel width at a minimum and increasing the length. Through simulations (shown in Chapter 5) for the cmrf7sf and SCMOS process technologies, the chosen channel widths and lengths of the precharge PMOS transistors that produce a low bit line voltage just below the NMOS threshold voltage are found. These values are displayed in Table 4.4.

Table 4.4: Chosen Precharge PMOS Transistor Widths and Lengths

Process Technology	**IBM cmrf7sf (180nm)**		**ON Semiconductor SCMOS (600nm)**	
Transistor Parameter	Channel Width (µm)	Channel Length (µm)	Channel Width (µm)	Channel Length (µm)
Precharge PMOS	0.22 (min.)	0.26	1.5 (min.)	1.2

Increasing the lengths of the transistors would have increased the resistance further, causing the low bit line voltage to drop closer to 0 V; however, this has two negative consequences. One is the increased area that results from having the longer channels. The other consequence is a much slower pull-up network for the bit lines. As it is shown in the next chapter, increasing the number of rows in the SRAM array greatly slows the charging of the bit line. Therefore, having faster (lower resistance) PMOS transistors for the precharge is another important factor in the design. The values shown in Table 4.4 keep the precharging rate as high as possible while still meeting the constraint of allowing one of the bit lines to drop below the NMOS threshold voltage during a read operation.

The layout of the precharge circuit is made manually, not with the compiler, though it is used by the compiler. Because of that, the layout dimensions have to match the dimensions of certain features of the SRAM cell with the chosen optimal transistor sizes. The width of the precharge is designed to match the width of the SRAM cell so its supply rail and N-type well can connect to other precharge circuits adjacent to it without any extra routing. The other factor is the locations of the bit lines. The outputs of the precharge circuit should automatically connect with the bit lines of the SRAM cell when placing the precharge circuits above the top row of the SRAM array. With all of these factors considered, the resulting layouts for the crmf7sf and SCMOS process technologies are shown in Figure 4.15 and Figure 4.16, respectively.

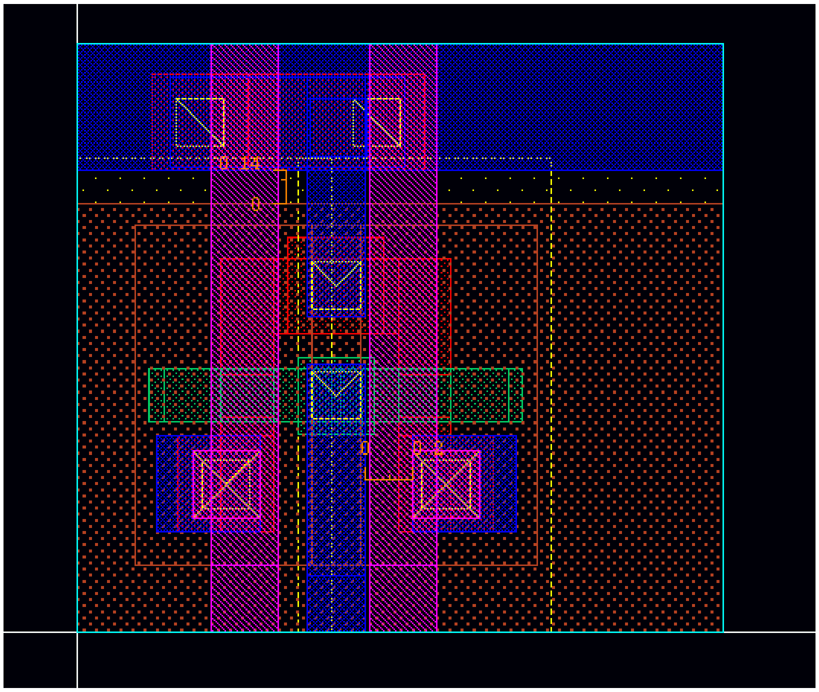


Figure 4.15: cmrf7sf Precharge Layout

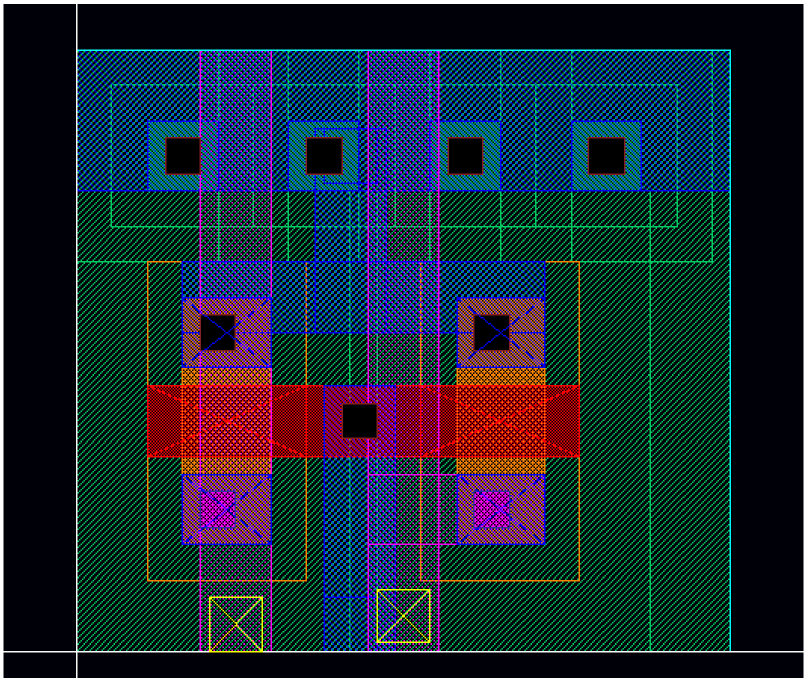


Figure 4.16: SCMOS Precharge Layout

With the precharge circuit layouts designed, the only designs remaining for this SRAM design are the address decoder and the integration of the address decoder and an SRAM block.

4.4 Address Decoder Design

This address decoder design follows the automatic design flow using the Cadence Encounter tools. One reason for this is the fact that the logic gate layouts needed to build an address decoder already exist, making the design time of the address decoder much shorter. Another reason the automatic design flow is used is that the address decoder can be better designed (less area and higher speed) using standard cells rather than some of the other topologies that exist. These ideas are discussed in more detail in section 2.5. The address decoder layout design is made separately from the compiler, since the compiler does not generate an address decoder. However, this address decoder layout can be added to an SRAM block created by the compiler. For this SRAM design, an SRAM array size of 16x8 is chosen so that the full layout can be seen with detail as a final result. With a large SRAM array, the details of the circuit layout are lost due to the limitations of the screen resolution. Since a SRAM array size of 16x8 is chosen, a 4:16 address decoder is needed.

Following the automated design flow, the Verilog code for the 4:16 address decoder (see Appendix B) is synthesized using the RTL Compiler to create structural Verilog. The gate-level diagram created by the RTL Compiler for this 4:16 decoder is shown in Figure 4.17. As mentioned in section 2.5, this address decoder is a CMOS-based decoder with a “predecoder” stage.

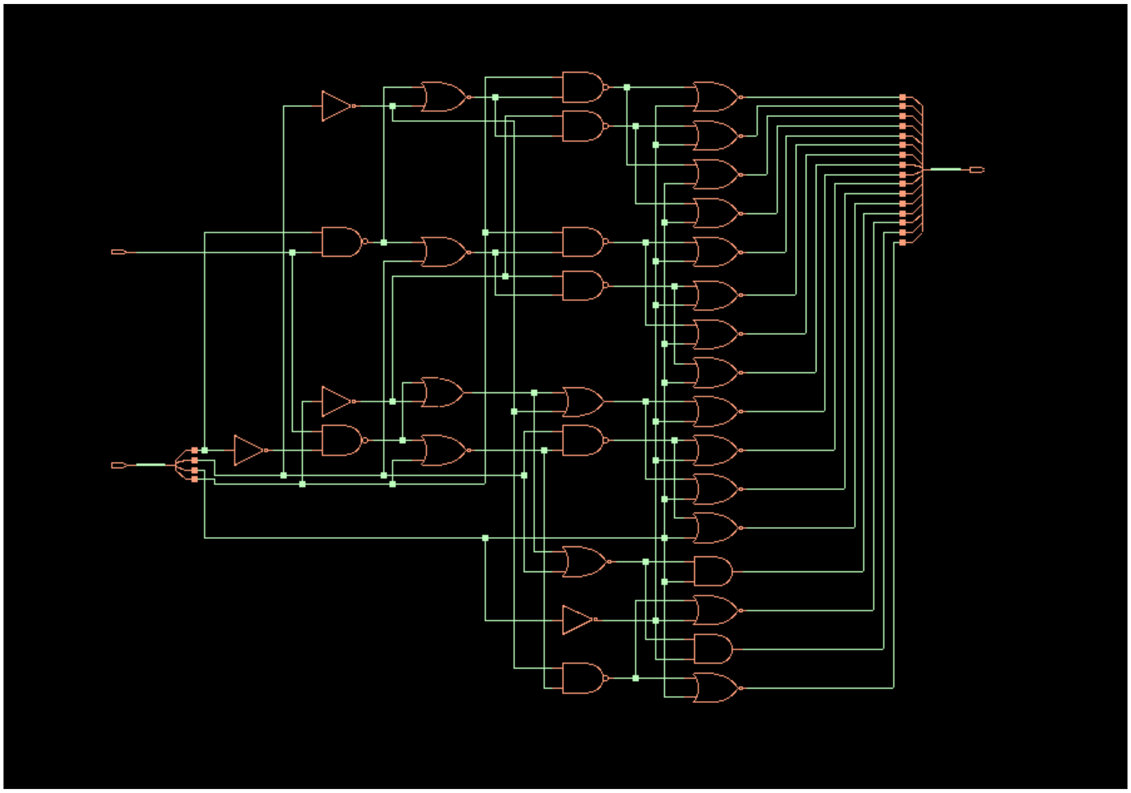


Figure 4.17: Gate-Level Diagram of a 4:16 Address Decoder

Next, the Encounter Digital Implementation tool is used to create the layout of the address decoder using the structural Verilog generated from the RTL Compiler. The resulting address decoder layout is shown in Figure 4.18.

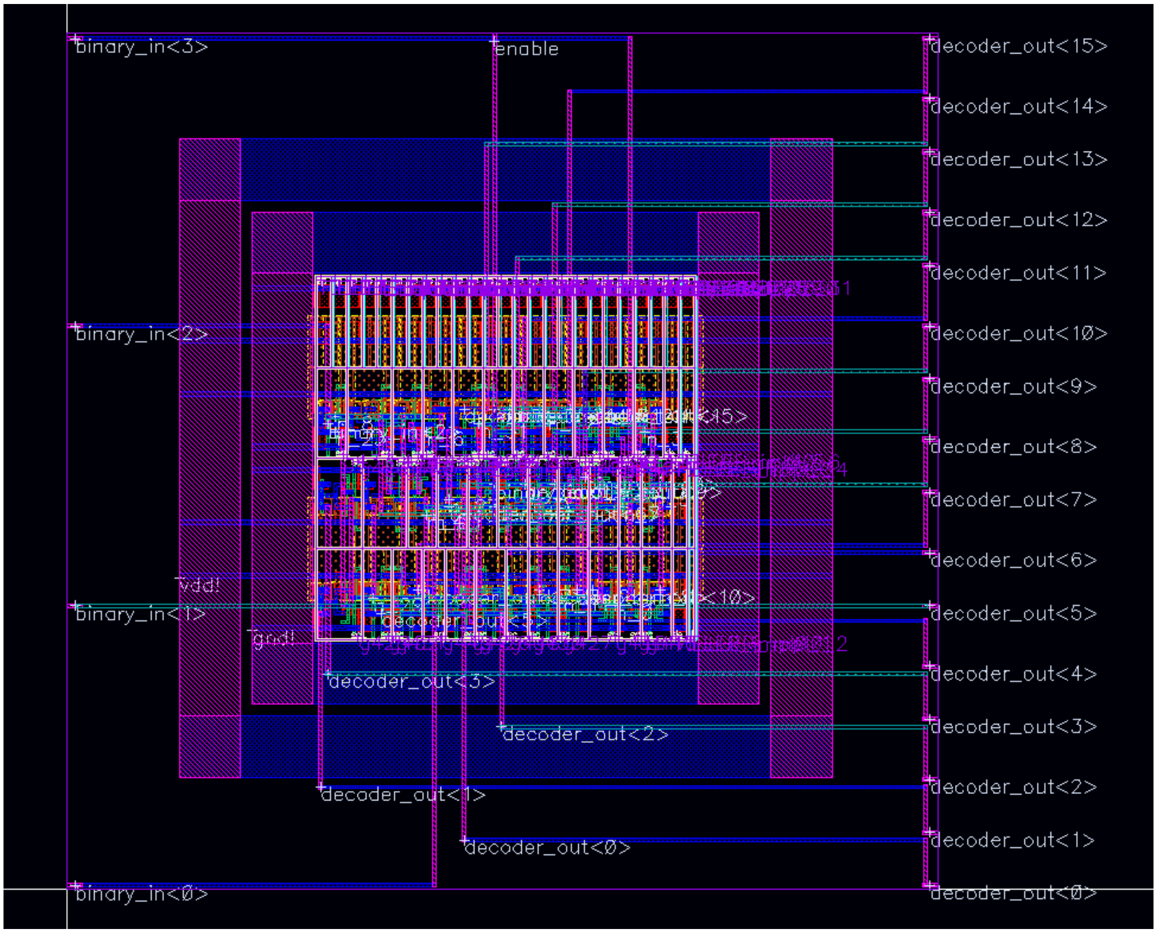


Figure 4.18: CMOS 4:16 Address Decoder Layout

The final step is to add this address decoder and the 16x8 SRAM block to a new layout and route the address decoder outputs to each of the SRAM input word lines. This final circuit is shown in the Testing and Results section.

With all of the design and implementation completed, the next step is to test the circuits through simulation and obtain the performance results of the circuit designs and the compiler. This is discussed in the next chapter.

Chapter 5: Testing and Results

Next, several simulations and measurements of voltages, dimensions, and times are displayed and analyzed. With these measurements, the performance is examined for the chosen optimal SRAM cell transistor sizes, the precharge circuits, and the SRAM compiler. Multiple simulations for both process technologies in various performance corners are completed to obtain information about settling voltage, power, and speed for the tested circuits.

For the cmrf7sf process technology, simulations of four performance corners are done. These corners represent the worst case process variations when a chip is fabricated. The four corner tests are with a slow NMOS and slow PMOS (ss), slow NMOS and fast PMOS (sf), fast NMOS and slow PMOS (fs), and fast NMOS and fast PMOS (ff). Doing all of these simulations ensures that the circuit will work even when process variations occur. The simulations also allow for the extraction of the worst case measurement of the measured parameters, such as voltage rise, power, or speed. It's important that all corners are tested since one parameter may have a worst case measurement in one corner, while another parameter has a worst case measurement in another corner. These corner simulations are done for all of the circuit simulations (SRAM cell read operation, write operation, SRAM cell with precharge, etc.) for the cmrf7sf process; however, the SCMOS technology library does not offer these process variations. Therefore, the SCMOS simulations are done with the typical NMOS and PMOS models.

5.1 Optimal SRAM Cell Transistor Sizing Simulations and Results

The first set of testing is done on the SRAM cell alone so that the performance of the chosen optimal transistor sizes can be analyzed and compared to the theory from Chapter 4.

5.1.1 Read Operation

First, a read operation simulation for the optimized SRAM cell is done for both process technologies. In the cmrf7sf simulation (see Figure 5.1), Q (pink) is initially set to a logical 1, Q_bar (red) is set to 0, and both of the bit lines are set to V_{DD} to emulate precharged bit lines. The word line (yellow) is asserted high so that the contents of the cell are placed on the bit lines (BL (green) remains high and BL_bar (orange) drops to a logical 0). Figure 5.1 shows the simulation for the read operation of the optimal SRAM cell for the cmrf7sf process technology.

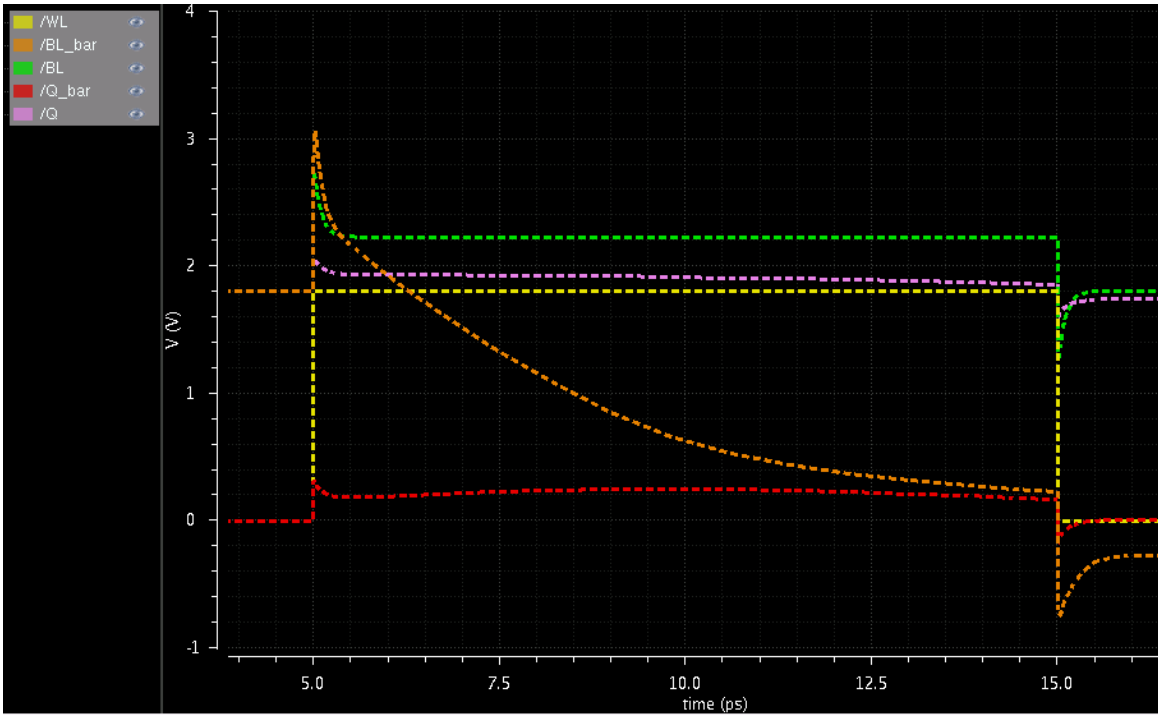


Figure 5.1: cmrf7sf Read Simulation (sf corner) for a Single SRAM Cell

The primary purpose of this simulation is to test if the read operation of the SRAM cell performs properly and if Q_bar (internal low voltage) rises above the NMOS threshold voltage (Vtn). From Figure 5.1, it is shown that the contents of the cell are successfully read, because BL stays high when the word line is asserted, and BL_bar decreases to a logical 0. Also, recall from Chapter 4 that the design of the SRAM cell is supposed to limit the internal voltage rise to less than the Vtn to avoid accidentally overwriting the cell during a read operation. The worst case corner simulation for the voltage rise is with a slow NMOS and a fast PMOS. With this simulation, the Q_bar voltage rise is 252.4 mV, which is less than the 460 mV NMOS threshold voltage. The simulation matches the theory; however, it should be noted that the number of cells attached to the bit lines will greatly influence the capacitance on the bit lines. Having bit lines with larger capacitance could cause Q_bar to rise more. The effect of the number of cells on this voltage rise is examined in section 5.2.

The other parameter measured for the read operation of the cmrf7sf process is read time. For this testing, the read time is determined by the time it takes for one of the bit lines to discharge to below the Vtn (460 mV) starting when the word line reaches V_{DD}. This read time is found to be 6.41 ps for the cmrf7sf process technology.

One other piece of information that can be noted in the simulation is the sharp peaking that occurs in Q_bar (red), Q (pink), BL (green), and BL_bar (orange) when the word line (yellow) is asserted high. This is due to the sharp increase in voltage on WL (word line) and the capacitances on the gates, drains, and sources of the NMOS pass transistors. Because capacitance resists an abrupt change in voltage, the internal nodes of the cell (Q and Q_bar) and the bit lines must sharply increase their voltage when the voltage on the gates of the pass transistors (WL) quickly increases. When these nodes change their voltage quickly with the word line, the voltage across the gates and those nodes does not increase as abruptly.

The read operation simulation is also done for the SCMOS-based SRAM cell. This simulation is shown in Figure 5.2. Q (pink) is initially set to a logical 0, Q_bar (red) is (labeled Q_NOT in the simulation image in Figure 5.2) set to 1, and the bit lines are initially set to V_{DD}. When WL (yellow) is set to a logical 1, BL_bar (orange), labeled BL_NOT in Figure 5.2, remains high, and BL (green) discharges down to a logical 0.

This simulation also seeks to verify the stability of the SRAM cell design for the SCMOS process technology. The internal voltage rise on node Q is found to be 489.9 mV when performing a read operation, and meets the requirement of being less than the Vtn of about 780 mV. Again, the voltage rise in this simulation meets the requirement by a relatively large margin of almost 300 mV, but when more cells are added to the bit lines, this margin can decrease because of the increased capacitive load of the bit lines.

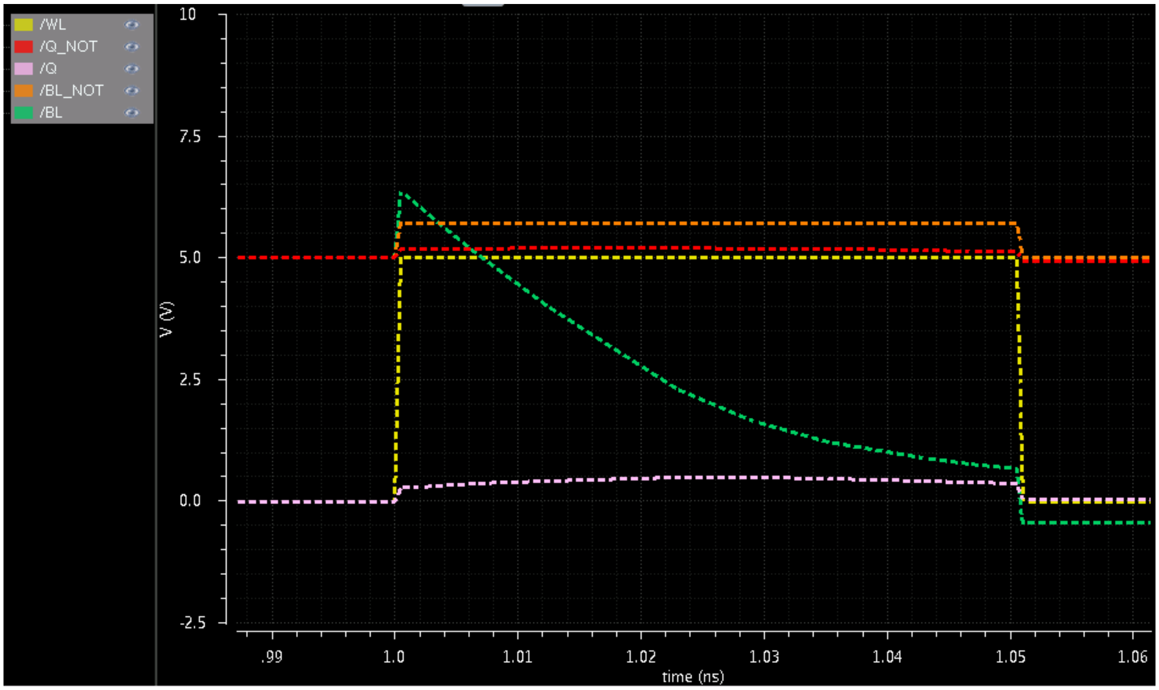


Figure 5.2: SCMOS Read Simulation for a Single SRAM Cell

As for the read time, defined in the same manner as before, the measurement is found to be 47.6 ps. This is over seven times longer of a read operation compared to the cmrf7sf process technology cell. Since SCMOS is a larger process technology, and the SRAM cell transistors in this design are larger than the cmrf7sf design, this leads to a larger capacitance on the gates of the transistors [26]. With an increase in capacitance (assuming a relatively constant resistance when scaling), an increase in the time constant occurs. This slows down the charging and discharging of the transistor’s internal capacitances [26]. These internal node voltage rise and read operation time measurements are summarized in Table 5.1 at the end of this section (section 5.1).

5.1.2 Write Operation

Next, the write operation for the single SRAM cell is simulated for functionality and read time. In this simulation for the cmrf7sf process technology (see Figure 5.3), the internal node Q (pink) is initialized to a logical 1, and Q_bar (red) is initialized to 0. Before WL (yellow) is asserted high, BL (green) is set low and BL_bar (orange) is set high. Once the word line is set high, the writing sequence begins, and Q is pulled down to 0 V, while Q_bar is pulled up to V_{DD}.

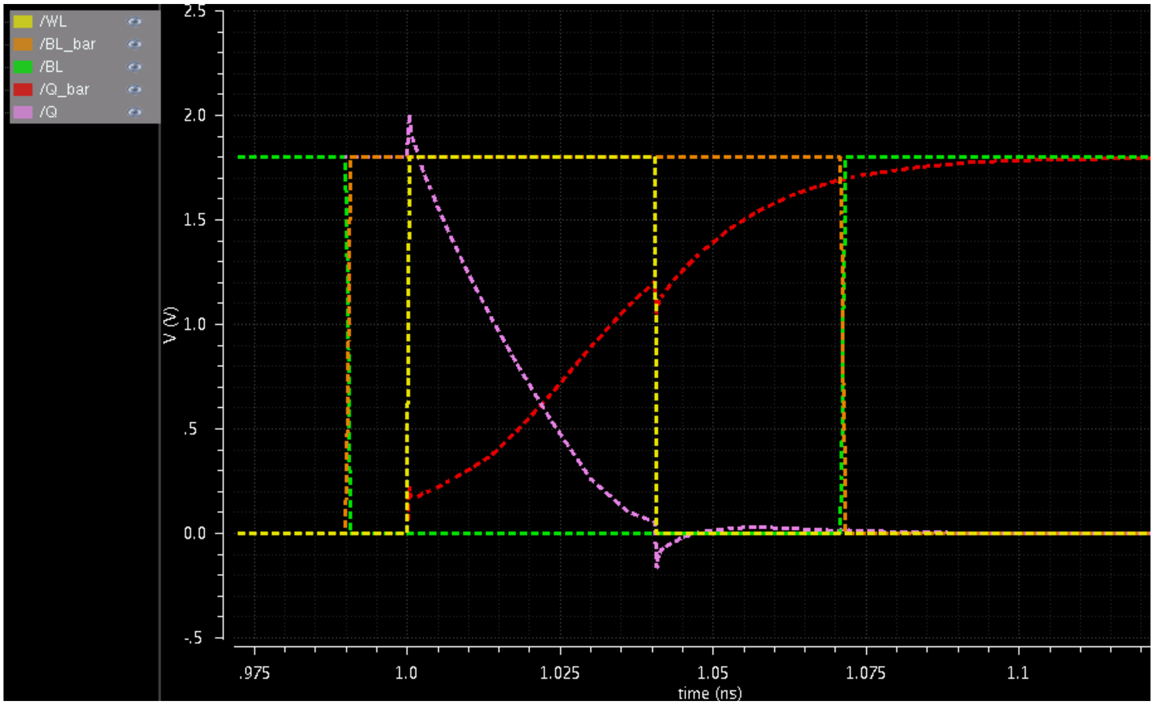


Figure 5.3: cmrf7sf Write Simulation (sf corner) for a Single SRAM Cell

From Figure 5.3, it is shown that the contents of the cell are successfully overwritten, because Q discharges to a logical 0 from a logical 1 when BL has a value of 1, while the opposite occurs for Q_bar. Again, a corner simulation with a slow NMO and fast PMOS proved to be the worst case simulation for write stability. The analysis done in Chapter 4 involved setting the ratio of the inverter PMOS transistor size to the NMOS pass transistor size so that node Q decreases below the Vtn. Because of the positive feedback loop of the cross-coupled inverters of cell, Q ultimately is pulled down to 0 V. Therefore, observing this occurrence in this simulation verifies the analysis done for the pull-up ratio in Chapter 4.

The other measured parameter for the write simulation is the write time. For this design, this write time is determined as the time from the word line reaching V_{DD} to the high internal node decreasing down to below the Vtn. For example, in this simulation, the write time is the time it takes for Q to decrease down to Vtn starting from when WL reaches V_{DD}. For the cmrf7sf process technology, this write time is measured to be 24.8 ps.

Next, the same simulation is done for the SCMOS process technology. This simulation is shown in Figure 5.4. In this simulation Q (pink) is initialized to 0 V and Q_bar (red), labeled Q_NOT, is initialized to V_{DD}. BL (green) is set to V_{DD}, BL_bar (orange), labeled BL_NOT, is set to 0 V, and then the word line (yellow) is set high to begin the write cycle. The contents of the cell are overwritten as Q rises to a logical 1 and Q_bar is pulled down to a logical 0.

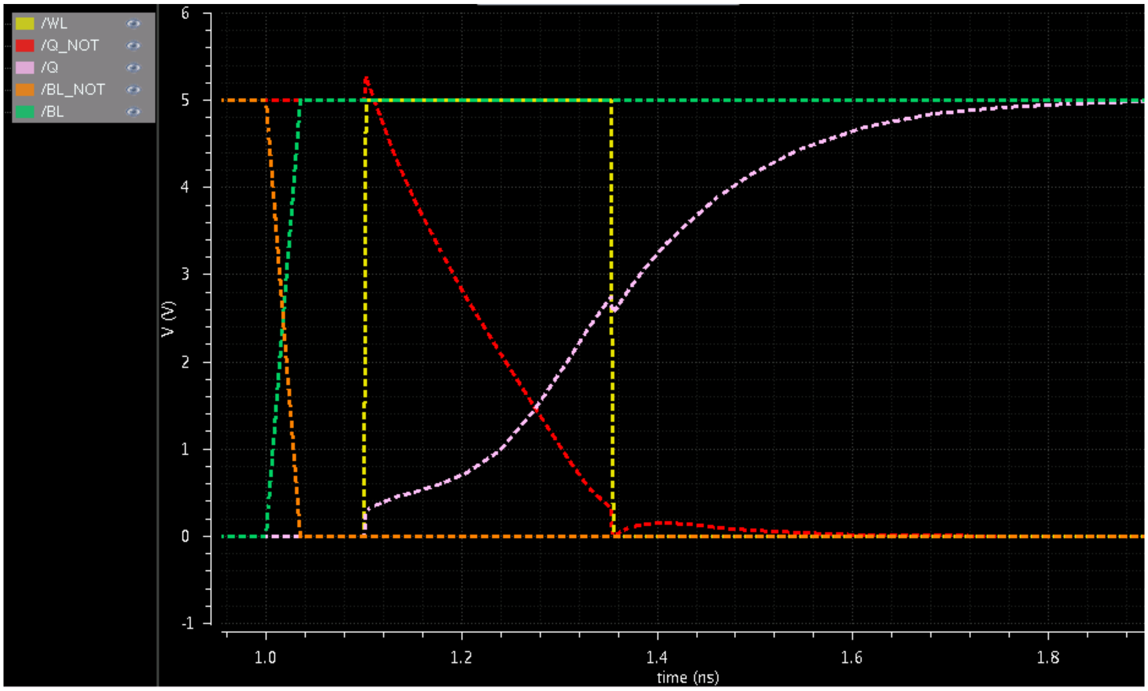


Figure 5.4: SCMOS Write Simulation for a Single SRAM Cell

As in the previous simulation, the SRAM cell for the SCMOS process technology is successfully written to, showing that the chosen transistor sizes verify the analysis done in Chapter 4. The write time of this SRAM cell is measured to be 237.8 ps. This write time is over nine times the length of the cmrf7sf-based SRAM cell write time. For the same reasons listed in the read operation testing section, the increased operation time is due to the increase in feature size (gate width and length). The increased transistor sizes increases the capacitance of the gate, leading to slower switching speeds.

In terms of simulations, the final performance parameter that is measured is the static power consumption of the single SRAM cell. The static power of the SRAM cell is due to leakage current through the MOSFET transistors when no switching is occurring. This is found by measuring the current supplied by the voltage source and multiplying it by V_{DD}. The static power for the cmrf7sf SRAM cell is 579.6 pW, and the power for the SCMOS SRAM cell is 51.5 pW. The increased power consumption for the cmrf7sf process, despite it being a smaller feature size, is due to the thinner insulative gate oxide and shorter channel length. A thinner gate oxide and shorter channel length both increase the leakage current of a transistor, ultimately leading to increased power consumption for the cell [26][27]. These values and all of the previously measured values for write and read times and cell voltage rise are summarized in Table 5.1.

Table 5.1: Single SRAM Cell Measurements with Optimal Transistor Sizes

Process Technology	IBM cmrf7sf (180nm)	ON Semiconductor SCMOS (600nm)
NMOS Threshold Voltage Vtn (mV)	460	780
Read Cell ΔV Voltage Rise (mV)	252.4	489.9
Single Cell Read Time (ps)	6.41	47.6
Single Cell Write Time (ps)	24.8	237.8
Supply Voltage (V)	1.8	5
Static Power (pW)	579.6	51.5

The next section discusses the simulations and results for SRAM cell(s) with a precharge circuit.

5.2 Precharge Simulations and Results

Next, simulations are done for a precharge circuit connected to a single SRAM cell and multiple SRAM cells for both process technologies. These simulations seek to show the performance of the precharge circuit, and analyze its effects on the voltage rise of the internal node of the SRAM cell, and observe the effects of having multiple SRAM cells attached to the same set of bit lines.

First, a read simulation is done for a single SRAM cell with a precharge circuit in the cmrf7sf process technology. In this simulation, the value of Q is initialized to V_{DD}, Q_bar is initialized to 0 V, and both of the bit lines are initialized to 0 V. The precharge is given time to pull both of the bit lines up to V_{DD} before the word line is asserted high to begin the read cycle. This simulation is shown in Figure 5.5.

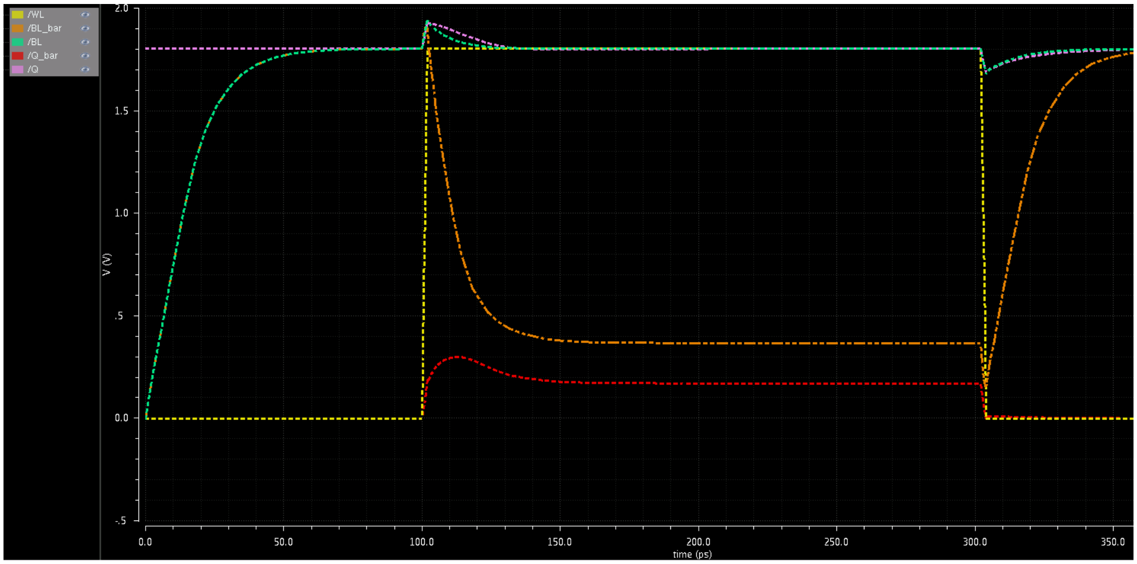


Figure 5.5: cmrf7sf Read Simulation for a Single SRAM Cell with a Precharge Circuit

This simulation reveals that the precharge operates as expected, pulling both of the bit lines up to V_{DD} with a rise time of 40.69 ps, where the rise time is defined as the time it takes the bit lines to increase from 10% to 90% of the final voltage (1.8 V for cmrf7sf). In addition to the precharge functioning correctly, the read operation also performs as expected. BL_bar is pulled down below Vtn to 365.7 mV. This verifies that the precharge PMOS transistor sizes from Table 4.4 are chosen to adequately lower the bit line voltage. The final parameter measured for this simulation is the node Q_bar voltage rise. This voltage rise is found to be 299.7 mV, which is greater than the single cell (no precharge circuit) simulation, but still is less than Vtn (460 mV).

The increase in this voltage rise from the previous simulation with no precharge circuit is due to the increased bit line capacitance from the precharge PMOS transistors.

Another identical simulation is done but instead of testing one SRAM cell with a precharge circuit, 256 SRAM cells sharing one set of bit lines and a precharge circuit are tested to observe the effects of the added bit line capacitance. This simulation proves to be successful in all respects; however, a few changes in the measured parameters occur. The bit line rise time increases from 40.69 ps to 4.28 ns. In this case, multiplying the bit line capacitance by a factor of 256 increases the rise time by over a factor of 100. This large increase in the bit line charging (and therefore discharging) time with an increase in the number of cells reveals the improvement that can be had with a sense amplifier when performing a read operation. In this simulation, BL_bar is pulled down to 365.8 mV, which is a relatively small increase of 0.1 mV from the previous measurement. This means the effect of the added bit line capacitance on the final bit line voltage is mostly negligible, while the charging and discharging time is greatly affected. Finally, the Q voltage rise increases from 299.7 mV to 366.02 mV. This relatively significant increase of 66.32 mV shows that the bit line capacitance also plays a role in the internal node voltage rise during a read operation. This voltage rise is still almost 100 mV less than Vtn, showing that the chosen transistor sizes still result in cell stability during a read operation.

Next, the same read simulation is done for a single SRAM cell with precharge circuit for the SCMOS process technology. This simulation is shown in Figure 5.6.

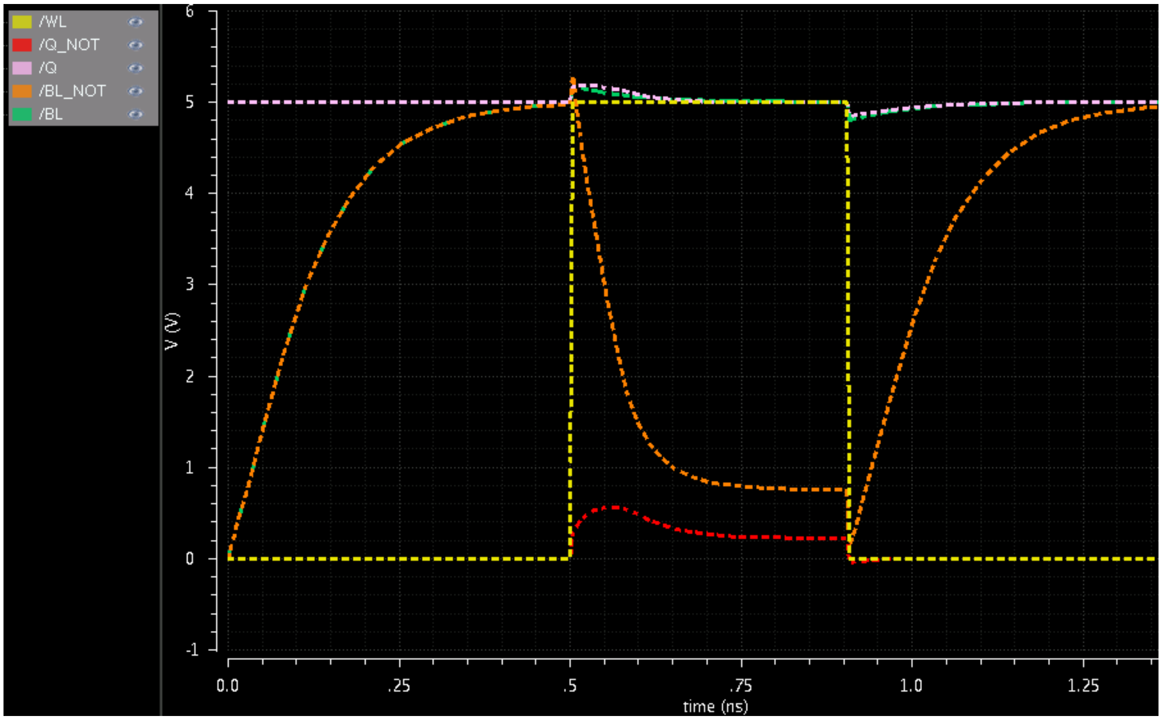


Figure 5.6: SCMOS Read Simulation for a Single SRAM Cell with a Precharge Circuit

In this simulation, the bit line rise time is 229.2 ps. BL_bar, labeled as BL_NOT in Figure 5.6, is pulled down to 761.8 mV, which is less than Vtn (780 mV), showing that the chosen precharge transistor sizes meet the requirement for the read operation. Compared to the single cell simulation with no precharge circuit, this simulation increases the internal node voltage rise from 489.9 mV to 565.6 mV. Again, this is due to the added bit line capacitance from the precharge transistors.

One final read simulation is done for the SCMOS process technology, with 256 SRAM cells attached to one set of bit lines and a precharge circuit. This increases the bit line rise time to 25.12 ns, which again is an increase in time by a factor of more than 100. The settling BL_bar voltage from the read operation remained relatively constant at 761.9 mV, verifying the chosen transistor sizes for the precharge circuit. Lastly, the internal node voltage rise increases to 627.96 mV with the added 255 SRAM cells. This voltage rise is still well under Vtn by over 150 mV, verifying that the chosen SRAM cell transistor sizes keep the cell stable during the read operation. The measured values for the SRAM cell simulations with precharge are summarized in Table 5.2.

Table 5.2: SRAM Cell(s) with Precharge Simulation Measurements

Process Technology	IBM cmrf7sf (180nm)	ON Semiconductor SCMOS (600nm)
Precharge Transistor Width (μm)	0.22 (min.)	1.5 (min.)
Precharge Transistor Length (μm)	0.26	1.2
NMOS Threshold Voltage Vtn (mV)	460	780
Single Cell Simulation		
Bit line Rise Time (ps)	40.69	229.2
Bit Line Low Voltage (mV)	365.7	761.8
Q (low) ΔV Rise (mV)	299.7	565.6
256 Cell Simulation (256 Rows)		
Bit line Rise Time (ns)	4.28	25.12
Bit Line Low Voltage (mV)	365.8	761.9
Q (low) ΔV Rise (mV)	366.02	627.96

With the SRAM cell and precharge circuits simulated and results that prove they perform as expected, the next step is to observe the performance and results of the SRAM compiler.

5.3 SRAM Compiler Automation Time Results

Since the primary advantage for using an SRAM compiler is the decrease in the amount of time needed to lay out an SRAM array, the performance metric that should be observed is the time it takes to compile various sized circuits. Along with these timing results, the areas of certain circuits are given for a comparison between the cmrf7sf and SCMOS process technologies. Finally, the layouts of the final circuits are shown.

The SRAM compiler is run to generate different sizes SRAM. These compilation times are then measured. Measurements for these times are listed in Table 5.3.

Table 5.3: Compilation Times of Various SRAM Array Sizes

Circuit Layout	Time to Generate
Single SRAM Cell	< 1s
16x8 SRAM Array	~ 1s
1024x1024 Array	~ 15 minutes

These values reveal that the SRAM compiler can generate large layouts in a relatively short amount of time. The automatic generation of a single SRAM cell takes less than a second, while manually laying out one can take at least an hour. The fact that this compiler can generate an SRAM array size of 1024x1024 in about a quarter of the time it takes to lay out one SRAM cell manually means that this tool is extremely useful in a world where time is always a factor and designs often have to be redone, further adding to the time for development.

The next measurements are the dimensions and area of a single SRAM cell and a 16x8 SRAM array. These values are shown in Table 5.4. Dimension and area measurements for SRAM layouts from other sources in the same process technologies are also displayed for comparison in Table 5.4.

Table 5.4: Dimensions and Area of an SRAM Cell and SRAM Array

Process Technology	IBM cmrf7sf (180nm)	ON Semiconductor SCMOS (600nm)
SRAM Compiler Generated Circuits		
Single Cell Dimensions (µm x µm)	5.21 x 2.7	22.2 x 11.1
Single Cell Area (µm^2)	14.067	246.42
16x8 SRAM Array Dimensions (µm x µm)	83.36 x 21.6	355.2 x 88.8
16x8 SRAM Array Area (µm^2)	1800.58	31541.76
Circuit Dimensions from Other Sources		
8x8 SRAM Array with Peripheral Circuits Dimensions (µm x µm)	N/A	244.5 x 285.3 [28]
8x8 SRAM Array with Peripheral Circuits Area (µm^2)	N/A	69755.85
Single Cell Dimensions (µm x µm)	3.5 x 4 [29]	N/A
Single Cell Area (um^2)	14	N/A

These dimensions and areas show that the process technology feature size greatly affects the area. Even though the ratio of the feature sizes (600nm/180nm) is 3.33, the ratio of the widths of a single SRAM cell is approximately 4.26, and the ratio of the heights of a single SRAM cell is approximately 4.11. This leads to a large increase in area by a factor of about 17.5 when changing from the cmrf7sf process technology to the SCMOS process technology. With SRAM being an area-hungry design, the amount of memory a designer can generate is limited for a given chip size depending on what process technology is used. Therefore, along with the speed improvements, there is a large advantage to using the cmrf7sf process technology of the SCMOS process technology because of the area reduction.

When comparing this SRAM design layout to layouts from other sources, the area measurements are comparable. One SCMOS 8x8 SRAM layout with peripheral circuits (address decoder, precharge, sense amplifier, column mux, etc.) has dimensions of 244.5 µm x 285.3 µm [28]. Accounting for the extra peripheral circuits, the height of a SCMOS SRAM cell from this source is estimated to be about the same as this thesis SCMOS SRAM cell design. However, the width of the cell from the other source is estimated to be about 2 times larger than this thesis cell design. For 7rf, another source had a cell measurement of 3.5 µm x 4 µm [29], giving a cell area of 14 µm^2. The 7rf SRAM cell design for this thesis is taller (5.21 µm) and not as wide (2.7 µm), but has a comparable area of 14.067 µm^2. This data shows that the SRAM cell layouts in this thesis have relatively compact designs compared to what exists from other sources.

When running the SRAM compiler using the sram_compiler() function, the final result is an SRAM cell layout, and an SRAM block layout of the desired number of cells (rows and columns) with precharge, power and ground rings, power and ground rails, and input and output pins for the bit lines, word lines, power, and ground. This final result of a 16x8 SRAM block for the IBM crmf7sf process technology is depicted in Figure 5.7.

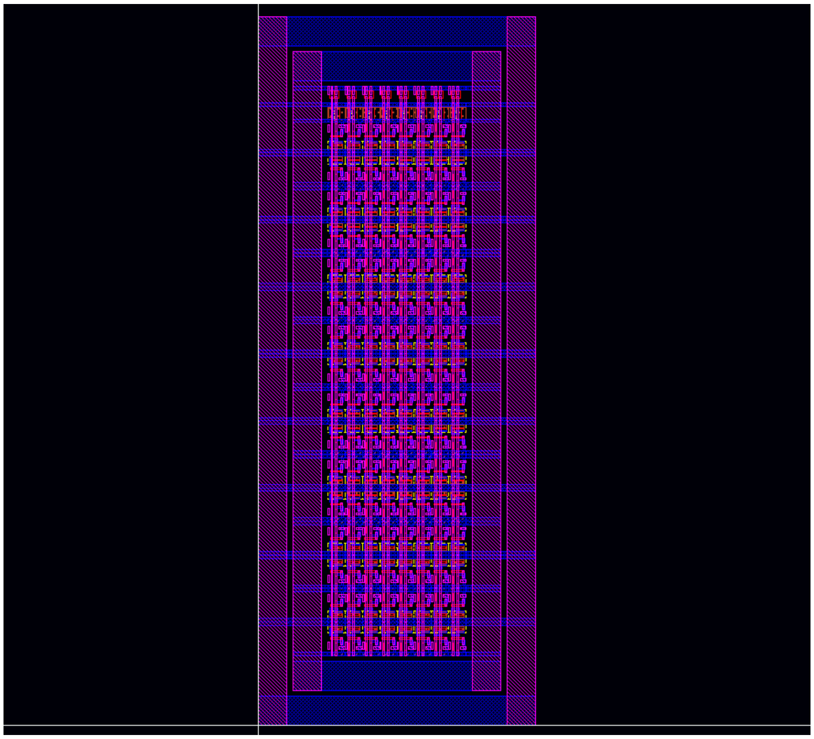


Figure 5.7: IBM cmrf7sf 16x8 SRAM Block Result From SRAM Compiler

Earlier a 4:16 address decoder was designed to connect with this SRAM block. By adding both layouts to a new layout and routing the outputs of the address decoder to the word line inputs of the SRAM, the final circuit layout is made. This layout is shown in Figure 5.8.

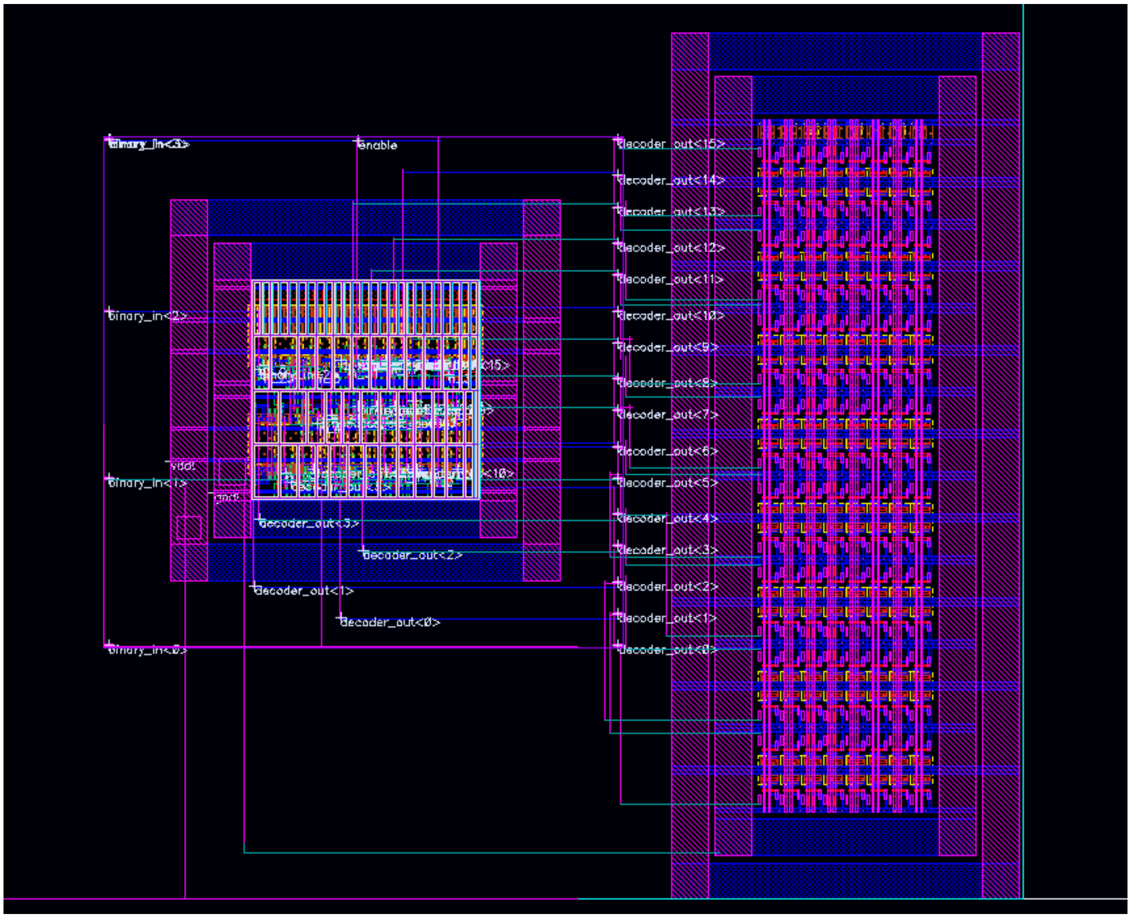


Figure 5.8: cmrf7sf 16x8 SRAM Block with a 4:16 Address Decoder

This final circuit layout completes the testing and results section. Next, this SRAM compiler design is summarized and a high-level evaluation of its significance is discussed. Areas of the design that can be improved on or added to are also discussed.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

Memory compilers are an extremely useful tool in the integrated circuit design world where designing, redesigning, and testing layouts increases the amount of time developing chips. While there are several SRAM compilers that exist today, they are lacking in many ways. This thesis design sought to offer an SRAM compiler that overcomes some of the shortcomings of other compilers.

One way this compiler is designed to stand out from other compilers is its ability to layout SRAM in multiple processes. The original hope was to design a compiler in the most modular manner possible. However, due to the differences that exist between process technologies and their corresponding Design Rule Check rules, an entirely modular design was not created. Much of the SRAM cell generation design is transferable between process technologies, but due to the large number of instances where different types of DRC rules were the limiting factor for each process, the design could not be made entirely modular. With the large number of process technologies that exist, it would be very difficult to account for every single case where the design does not match up. It is possible that the design could be made more modular for a set of process technologies with the same company, but since the two process technologies used in this thesis were made by different companies (IBM and ON Semiconductor), this has not been looked into. Therefore, the SRAM cell generation design for this compiler is made as modular as possible so that future process technologies can be added by following the same format as the current SKILL code files.

Another way this compiler is unique is that it offers the user the ability to choose the SRAM cell transistor sizes, effectively creating their own SRAM cell. This is a very useful feature especially in the academic setting, where optimizations of speed, power, or area can be made to best support their design needs. If the user does not choose their own transistor sizes, default transistor sizes are provided for both processes. These default values are chosen based on the analysis done in Chapter 4 to create cell stability when performing read and write operations, while still favoring minimal area for the SRAM cell.

The SRAM design structure has several favorable qualities. For one, the SRAM cell is made so that no extra routing is required when placing multiple cells together to form an array. Because the design is based on passing DRC rules and minimizing the area, the result is a very compact SRAM cell design, even when adjusting the transistor sizes. When creating multiple rows of SRAM cells, every other row of cells is flipped vertically. This reduces the amount of routing needed for power and ground, and minimizes area, because the SRAM cells snap together vertically. This also removes the need for additional routing that would be required if the cells were not flipped.

Simulations have verified that the circuit design matches the analysis in terms of function and performance. The SRAM cell remains stable when a read and write operation are done, and the precharge design allows for reading a logical 1 at a voltage of V_{DD} and a logical 0 at a voltage less than Vtn. In addition to verifying a properly functioning SRAM, the simulations revealed some key differences between the two process technologies used. SCMOS is a larger process that is slower and consumes less static power, while cmrf7sf is a smaller process that is faster and consumes more static power. Having these options gives the user an option for best supporting their design needs. Especially with the read operation, the simulations also showed that improvements can be made to the SRAM with added hardware (i.e. a sense amplifier). Though there are areas for improvement, this SRAM compiler is a free solution that meets the needs of Cal Poly student integrated circuit design projects.

6.2 Future Works

This SRAM compiler design is functional, provides several features that collectively make it a unique design, and it lays the foundation for improvements and additions to the design. The main improvements to this design are in the realm of adding hardware and generating layouts through the writing of SKILL code.

One really useful piece of hardware that could be added to this design is a sense amplifier. The sense amplifier would reduce the read time and would give V_{DD} for reading a logical 1 and 0 V for reading a 0 instead of some voltage less than Vtn for reading a 0. The best option would be to automate the layout of the sense amplifier so that input and output lines can properly align with the SRAM array and any other parts of the circuit if the SRAM cells have customized transistor sizes.

Another circuit to add to the compiler design is the column mux (or column decoder). Having a column mux connects the data being written to, or read from, the bit lines and ultimately reduces the number of signals to manipulate such data. There are multiple topologies that can been chosen from, but selecting one that best fits this SRAM architecture is the main goal, along with adding SKILL code to the compiler to automatically generate its layout based on the many variables already discussed (SRAM cell size, process, SRAM array size, etc.).

Automating the layouts for the precharge and address decoder circuits using SKILL is another feature that could be added to this compiler. The precharge in this design is limited to the default SRAM cell transistor sizes chosen for the compiler, and the address decoder is limited to only sixteen rows of SRAM, though it could be used more than once for SRAM arrays that have a number of addresses equal to a multiple of sixteen. Since these designs for this SRAM compiler were manually made for specific cases, making additions to the compiler to automatically generate these layouts for custom SRAM cell sizes and any number of array sizes would be a really useful improvement.

There are several different types of SRAM and SRAM architectures that can be looked into. In general, optimizations can be made for speed, power, and area, but there are a number of ways of doing this. The SRAM cell itself can be designed to be fast, or low power, or to have a small area. Another method is to manipulate the higher-level architecture of the SRAM. One low power example is to create partitions of SRAM blocks instead of one large SRAM block. Splitting up the SRAM into smaller blocks reduces the word line and/or bit line capacitances. This reduces the amount of current needed to drive the lines, effectively reducing the power. One possible design choice for increasing the speed of the SRAM is to use dual-port SRAM, which has a different cell structure and gives the ability of reading from and writing to different cells at different addresses simultaneously. The main idea is offer the user more design options so that they can create the SRAM that best fits their requirements.

While the compiler generates power rails for the SRAM array, power stripes can still be added. Because the power rings and rails have some amount of resistance and are conducting current, an IR drop can occur, creating unequal supply voltages throughout the circuit. Without power stripes, the word length (number of SRAM columns) is limited due to this fact. Adding power stripes to the compiler would resolve this limitation.

Two final features that could be added are a GUI (Graphical User Interface) and to include more process technologies. With more and more added features, a GUI will be needed to better organize the available design options for the user when they use the compiler to create the SRAM of their choice. The SKILL library has a list of GUI functions that would make this feature possible. Adding more process technologies simply gives more options for the user to design their circuit in. However, the more hardware options that are added to the existing design, the more complicated and time-consuming it would be to have all of the same options for a whole new process technology. Table 6.1 summarizes the list of possible future works for this SRAM compiler design.

Table 6.1: Summary of Potential Future Works

Future Works
Add sense amplifier to the compiler
Add column mux to the compiler
Automate the precharge layout using SKILL
Automate the address decoder layout using SKILL
Add new topologies of SRAM to compiler (dual-port, low power, etc.)
Partition SRAM into smaller blocks when using compiler
Generate power and ground stripes (vertical wires) to compiler
Create a GUI to run the compiler
Add more process technologies

From this list, it is shown that this compiler design is expandable. While the current design creates a functioning SRAM in multiple processes, with the option to choose custom cell transistor sizes, there are other areas to explore and research for adding to the existing design.

References

[1] Press Room Release, “MoSys,” [online] 1999, http://www.mosys.com/investors.php?page=pressRoomtxt&id=340762 [Accessed: 13 December 2014].

[2] Brian Dipert, “Fundamentals of volatile memory technologies,” http://www.electronicproducts.com/Digital_ICs/Memory/Fundamentals_of_volatile_memory_technologies.aspx: Hearst Electronic Products, 2011. [Accessed: 13 December 2014]

[3] “Static RAM Memory,” https://moodle.insa-toulouse.fr/pluginfile.php/2632/mod_resource/content/0/content/static_ram.html [Accessed: 13 December 2014]

[4] IBM Design Rules and Cell Libraries, “The MOSIS Service,” [online], https://www.mosis.com/vendors/view/ibm/documents [Accessed: 13 December 2014]

[5] J. Rabaey. *Digital Integrated Circuits A Design Perspective*, 1st ed. Saddle River, NJ: Prentice Hall, 1996.

[6] “Introduction to Flash Memory.” Internet: https://web2.ph.utexas.edu/~ygong/flash.html, [Accessed: May 14, 2015].

[7] International Business Machines Corp. (1996). “Understanding DRAM Operation.” [Online]. Available: https://www.ece.cmu.edu/~ece548/localcpy/dramop.pdf. [Accessed: 15 May 2015].

[8] J. Ganssle, T. Noergaard, F. Eady, L Edwards, D. Katz, R. Gentile, K. Arnold, K Hyder, and B. Perrin. (2008), pp. 106. *Embedded Hardware: Know It All.* [Online]. Available: https://books.google.com/books?id=HLpTtLjEXqcC&pg=PA106&lpg=PA106&dq=refresh+SRAM+DRAM&source=bl&ots=ANCVz6F6JV&sig=-r0FYJk3d5cbVU3epimxCCBSLzQ&hl=en&sa=X&ei=alA7UP7uO-eBiwLCjIHgBg#v=onepage&q=refresh%20SRAM%20DRAM&f=false. [Accessed: 15 May 2015].

[9] International Business Machines Corp. (1997). “Understanding Static RAM Operation.” [Online]. Available: https://www.ece.cmu.edu/~ece548/localcpy/sramop.pdf. [Accessed: 15 May 2015].

[10] University of Maryland. “DRAMSim2.” Internet: http://www.eng.umd.edu/~blj/dramsim/, [Accessed: May 15, 2015].

[11] “The MOSIS Service,” [online], https://www.mosis.com/ [Accessed: May 16, 2015].

[12] D. Wang. “MODERN DRAM MEMORY SYSTEMS: PERFORMANCE ANALYSIS AND SCHEDULING ALGORITHM” Ph.D. dissertation, University of Maryland, College Park, MD, 2005. [Online]. Available: http://www.ece.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf. [Accessed: 16 May 2015].

[13] M. Jagasivamani. “Development of a Low-Power SRAM Compiler.” M.S. thesis, Virginia Polytechnic Institute and State University, VA, 2000.

[14] J. Butera. “OpenRAM: An Open-Source Memory Compiler.” M.S. thesis, UC Santa Cruz, Santa Cruz, CA, 2013.

[15] J. Rabaey. *Digital Integrated Circuits A Design Perspective,* 2nd ed. Saddle River, NJ: Prentice Hall, 2003, pp. 284-287.

[16] “DesignWare Memory Compilers.” Internet: http://www.synopsys.com/dw/ipdir.php?ds=dwc_sram_memory_compilers, 2015 [Accessed: 25 May 2015].

[17] “SRAM.” Internet: http://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php, 2014 [Accessed: 25 May 2015].

[18] T. Smilkstein. EE 431 VLSI, Topic: “Topic0_DesignFlow.” Faculty of Electrical Engineering, California Polytechnic State University, San Luis Obispo, CA, Fall 2013.

[19] T. Smilkstein. EE 431 VLSI, Topic: “Week 4: Large digital designs – Automatic layout.” Faculty of Electrical Engineering, California Polytechnic State University, San Luis Obispo, CA, Fall 2013.

[20] T. Smilkstein. EE 431 VLSI, Topic: “Week 3: Timing analysis and PCELLs.” Faculty of Electrical Engineering, California Polytechnic State University, San Luis Obispo, CA, Fall 2013.

[21] “SKILL Language User Guide, Product Version 06.01.” Internet: http://support.ema-eda.com/search/eslfiles/default/main/sl_legacy_releaseinfo/staging/sl3/release_info/psd142/sklanguser/chap1.html, 2001 [Accessed: 5 June 2015].

[22] “SKILL Language User Guide.” Internet: https://projects.cecs.pdx.edu/attachments/download/.../sklanguser610.pdf, Aug. 2005 [Accessed: 5 June 2015].

[23] J. Rabaey. Digital Integrated Circuits 2nd Edition, Topic: “Chapter 12: Designing Memory and Array Structures.” UC Berkeley, Berkeley, CA, Jan. 20, 2003. Internet: http://icbook.eecs.berkeley.edu/resources/powerpoint-slides [Accessed: 7 June 2015].

[24] “MOSIS WAFER ELECTRICAL TESTS.” Internet: https://www.mosis.com/cgi-bin/cgiwrap/umosis/swp/params/ami-c5/v3bm-params.txt [Accessed: 7 June 2015].

[25] “MOSIS WAFER ACCEPTANCE TESTS.” Internet: https://www.mosis.com/cgi-bin/cgiwrap/umosis/swp/params/ibm-018/v08z_7rf_6lm_ml-params.txt [Accessed: 7 June 2015].

[26] “Ch. 7 MOSFET Technology Scaling, Leakage Current, and Other Topics.” Internet: http://www-inst.eecs.berkeley.edu/~ee130/sp06/chp7full.pdf [Accessed: 21 July 2015].

[27] “Leakage Current: Moore’s Law Meets Static Power.” Internet: http://www.ruf.rice.edu/~mobile/elec518/readings/DevicesAndCircuits/kim03leakage.pdf [Accessed: 7 June 2015].

[28] “SRAM IP for DSP/SoC Projects.” Internet: http://www-engr.sjsu.edu/dparent/ICGROUP/sram.pdf [Accessed: 30 July 2015].

[29] “An SRAD Image Processor as Temperature-Aware SoC Designed for Low-Power Operation.” Internet: http://www.cs.virginia.edu/~wh6p/SOCPIIReport3.doc [Accessed: 6 August 2015].

Appendices

Appendix A: SRAM Compiler SKILL Code

SRAM Compiler

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;File: sram_compiler.il
;Functions: sram_compiler(tech sramLib_Name sramCV_Name cellCV_Name nmos_pass nmos_inv
pmos_inv row col)
;Description: Creates an SRAM block layout of the desired size, generates an SRAM cell layout by
calling
;                                       the corresponding process technology SRAM cell layout function,
adds power and ground
;                                       rings and rails, and manually-made precharge circuits to the SRAM
block layout.
;Input arguments:
;          tech: process technology
;          sramLib_Name: library name
;          sramCV_Name: name of SRAM block layout
;          cellCV_Name: name of SRAM cell layout
;          nmos_pass: SRAM cell pass NMOS width
;          nmos_inv: SRAM cell inverter NMOS width
;          pmos_inv: SRAM cell inverter PMOS width
;          row: number of words (rows) in SRAM array
;          col: number of bits per word (columns) in SRAM array
;Usage example with default transitor widths:
;          In command window type:
;                    load("sram_compiler.il")
;                    sram_compiler("7rf" "SRAM" "SRAM_array" "cell_layout" 0 0 0 16 8)
;Usage example with custom transistor widths:
;          In command window type:
;                    ;load("sram_compiler.il")
;                    sram_compiler("c5n" "SRAM" "SRAM_array" "cell_layout" 1.65 3.0 1.5 16 8)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Author: Brandon Hilgers
;Date: 17 June 2015
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(sram_compiler(tech sramLib_Name sramCV_Name cellCV_Name nmos_pass nmos_inv
pmos_inv row col)

load("powerRings.il")
load("generatePin.il")

if(dbOpenCellViewByType(sramLib_Name sramCV_Name "layout" "maskLayout" "r")!=nil
  then
          error("Please delete SRAM cell view layout or choose a new cell view name")
)
SRAM_cv = dbOpenCellViewByType(sramLib_Name sramCV_Name "layout" "maskLayout" "w")
prchrg_cv = dbOpenCellViewByType(sramLib_Name "Precharge" "layout" "maskLayout" "r")

;CREATE SRAM CELL BASED ON PROCESS TECHNOLOGY
```

```
if(tech == "c5n"
  then
        load("sram_c5n_cell_layout.il")
        sram_c5n_cell_layout(sramLib_Name cellCV_Name nmos_pass nmos_inv pmos_inv)
        lambda = 0.3 ;microns
        prchrg_height = 5.0
else
        if(tech == "7rf"
          then
                load("sram_7rf_cell_layout.il")
                sram_7rf_cell_layout(sramLib_Name cellCV_Name nmos_pass nmos_inv pmos_inv)
                lambda = 0.09 ;microns
                prchrg_height = 2.46
        else
                error("Not a valid process technology.\nTerminating program...\n")
        )
)

;LAYOUT POWER RINGS AND RAILS
powerRings(SRAM_cv lambda row col cell_width cell_height gnd_height prchrg_height)

;LAYOUT SRAM ARRAY
for(i 0 row-1
        if(modulo(i 2) == 0
         then
                orientation = "R0"
                loc_y = 2*ring_wd + 2*ring_sp + cell_height*i
        else
                orientation = "MX"
                loc_y = 2*ring_wd + 2*ring_sp + cell_height*(i+1)
        )
        for(j 0 col-1
                loc_x = 2*ring_wd + 2*ring_sp + cell_width*j
                cellInst = dbCreateInst(SRAM_cv sram_cell_cv "cellInst" list(loc_x, loc_y) orientation)
                dbFlattenInst(cellInst 1 t)
                if(i == row-1
                  then
                        prchrgInst = dbCreateInst(SRAM_cv prchrg_cv "prchrgInst" list(loc_x, loc_y)
"R0")
                        dbFlattenInst(prchrgInst 1 t)
                )
                if(i == 0
                  then
                        ;BL pin
                         sprintf(bl_pinName "BL%d" j)
                        generatePin(SRAM_cv bl_pinName metal2 "output" (bl_min_x + array_min_y
+ cell_width*j) array_min_y (bl_max_x + array_min_y + cell_width*j) (array_min_y + m2wd))

                        ;BL_bar pin
                        sprintf(bl_bar_pinName "BL_bar%d" j)
                        generatePin(SRAM_cv bl_bar_pinName metal2 "output" (bl_bar_min_x +
array_min_y + cell_width*j) array_min_y (bl_bar_max_x + array_min_y + cell_width*j) (array_min_y +
m2wd))
                )
                if(j == 0
                  then
```

```
				;WL pin
				sprintf(wl_pinName "WL%d" i)
				if(modulo(i 2) == 0
				  then
					wl_pin_min_y = wl_via_min_y + array_min_y + cell_height*i
				else
					wl_pin_min_y = array_min_y + cell_height*(i+1) - wl_via_min_y -
m2wd
				)
				wl_pin_max_y = wl_pin_min_y + m2wd
				generatePin(SRAM_cv wl_pinName metal2 "input" array_min_y
wl_pin_min_y (array_min_y + m2wd) wl_pin_max_y)
			)
		)
)
;vdd pin
generatePin(SRAM_cv "vdd!" metal1 "input" 0 0 m1wd m1wd)
;gnd pin
generatePin(SRAM_cv "gnd!" metal1 "input" gnd_min_x gnd_min_y (gnd_min_x + m1wd) (gnd_min_y
+ m1wd))
)
```

cmrf7sf SRAM Cell Layout

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;File: sram_7rf_cell_layout.il
;Functions: sram_7rf_cell_layout(cell_layout_lib cell_layout_cv nmos_passw nmos_invw pmos_invw)
;Description: Creates an SRAM cell layout for the IBM cmrf7sf process technology.
;Input arguments:
;	cell_layout_lib: library name
;	cell_layout_cv: name of SRAM cell layout
;	nmos_passw: SRAM cell pass NMOS width
;	nmos_invw: SRAM cell inverter NMOS width
;	pmos_invw: SRAM cell inverter PMOS width
;Usage example with default transitor widths:
;	In command window type:
;		load("sram_7rf_cell_layout.il")
;		sram_7rf_cell_layout("SRAM" "cell_layout" 0 0 0)
;Usage example with custom transistor widths:
;	In command window type:
;		;load("sram_7rf_cell_layout.il")
;		sram_7rf_cell_layout("SRAM" "cell_layout" 0.26 0.3 0.22)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Author: Brandon Hilgers
;Date: 17 June 2015
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(sram_7rf_cell_layout(cell_layout_lib cell_layout_cv nmos_passw nmos_invw pmos_invw)
load("drc_rules.il")
drc_7rf()

;CELL TRANSITOR Sizes
nmos_passl = 0.18 ;nmos pass transistor gate length
```

```
nmos_invl = 0.18 ;nmos inverter transistor gate length
pmos_invl = 0.18 ;pmos inverter transistor gate length

if((nmos_passw==0)&&(nmos_invw==0)&&(pmos_invw==0)
  then
        nmos_passw = 0.22
        nmos_invw = 0.22
        pmos_invw = 0.22
        printf("Using default transistor sizes\n")
else
        npassw = truncate(nmos_passw*1000)
        ninvw = truncate(nmos_invw*1000)
        pinvw = truncate(pmos_invw*1000)
        if((nmos_passw<0.22)||(mod(npassw 10)!=0)||(nmos_invw<0.22)||(mod(ninvw
10)!=0)||(pmos_invw<0.22)||(mod(pinvw 10)!=0)
         then
                error("Transistor minimum width = 0.22um, and width must be multiple of 0.01um")
        )
)

;LAYER & COMPONENT NAMES
metal1 = "M1"
metal2 = "M2"
active = "RX"
poly = "PC"
contact = "CA"
via = "V1"
nwell = "NW"
bp = "BP"

;OPEN CELL VIEWS
;SRAM cell layout cell view
if(dbOpenCellViewByType(cell_layout_lib cell_layout_cv "layout" "maskLayout" "r")!=nil
  then
        error("Please delete SRAM CELL cell view layout or choose a new cell view name")
)
sram_cell_cv = dbOpenCellViewByType(cell_layout_lib cell_layout_cv "layout" "maskLayout" "w")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;LAYOUT OF SRAM CELL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;layout begins at bottom (gnd) and makes its way up to the top (vdd)
;GND RAIL
gnd_height = 0.5*act_sp + con_size + 2*act_enc_con
;ptaps (or subcx)
ptap_act_min_x = bp_enc_actwell
ptap_act_min_y = 0.5*act_sp
ptap_act_max_x = ptap_act_min_x + act_con_size
ptap_inc = act_con_size + act_sp ;amount to increment ptap active region by in x direction
ptap_act_rect1 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x:ptap_act_min_y,
ptap_act_max_x:gnd_height))
ptap_act_rect2 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x + ptap_inc:ptap_act_min_y,
ptap_act_max_x + ptap_inc:gnd_height))
ptap_act_rect3 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x +
2*ptap_inc:ptap_act_min_y, ptap_act_max_x + 2*ptap_inc:gnd_height))
ptap_act_rect4 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x +
3*ptap_inc:ptap_act_min_y, ptap_act_max_x + 3*ptap_inc:gnd_height))
ptap_con_min_x = ptap_act_min_x + act_enc_con
ptap_con_min_y = ptap_act_min_y + act_enc_con
ptap_con_max_x = ptap_con_min_x + con_size
ptap_con_max_y = ptap_con_min_y + con_size;
ptap_con_rect1 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x:ptap_con_min_y,
ptap_con_max_x:ptap_con_max_y))
ptap_con_rect2 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
ptap_inc:ptap_con_min_y, ptap_con_max_x + ptap_inc:ptap_con_max_y))
ptap_con_rect3 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
2*ptap_inc:ptap_con_min_y, ptap_con_max_x + 2*ptap_inc:ptap_con_max_y))
ptap_con_rect4 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
3*ptap_inc:ptap_con_min_y, ptap_con_max_x + 3*ptap_inc:ptap_con_max_y))

;LEFT WORDLINE CONTACT
wl_max_y = gnd_height + max(m1_m1sp,act_sp) + act_enc_con + con_size + actcon_tgate_sp
l_wl_m1_max_x = pol_enc_con + con_size + m1_enc_con
wl_via_min_y = wl_max_y - m2wd
l_wl_poly_max_x = 2*pol_enc_con + con_size
wl_min_y = wl_max_y - l_wl_poly_max_x
if(nmos_passw < 0.46 ;calculated threshold for determining basing nmos pass location on m1 width or m2
width of  left WL contact
  then
		l_wl_max_x2 = 1.5*m2wd + m2_m2sp - float(truncate(nmos_passw*50))/100 - pol_olap_act
else
		l_wl_max_x2 = l_wl_m1_max_x + m1_m1sp - pol_olap_act
)
l_wl_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(0:wl_min_y, l_wl_poly_max_x:wl_max_y))
l_wl_con_rect = dbCreateRect(sram_cell_cv list(contact) list(pol_enc_con:wl_min_y + pol_enc_con,
l_wl_poly_max_x - pol_enc_con:wl_max_y - pol_enc_con))
l_wl_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:wl_via_min_y,
l_wl_m1_max_x:wl_via_min_y + m1wd28_area))
l_wl_via_rect = dbCreateRect(sram_cell_cv list(via) list(0:wl_via_min_y, via_size:wl_max_y))
l_wl_m2_max_y = wl_via_min_y + m2wd_area
l_wl_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(0:wl_via_min_y, m2wd:l_wl_m2_max_y))
l_wl_poly2_rect = dbCreateRect(sram_cell_cv list(poly) list(0:wl_max_y, l_wl_max_x2:wl_max_y +
nmos_passl))

;NMOS PASS TRANSISTORS - 'l' means left transitor, 'b' means bottom piece, 't' top
;nmos_pass_inst0 = dbCreateInst(sram_cell_cv nmos_cv "nmos_pass_inst0" list(3.45,5.1) "R90")
;LEFT PASS TRANSISTOR
l_pass_act_min_x = l_wl_max_x2 + pol_olap_act
pass_act_min_y = gnd_height + max(m1_m1sp,act_sp)
l_pass_act_max_x = l_pass_act_min_x + nmos_passw
pass_act_max_y = wl_max_y + nmos_passl + actcon_tgate_sp + con_size + act_enc_con
l_pass_act_rect = dbCreateRect(sram_cell_cv list(active) list(l_pass_act_min_x:pass_act_min_y,
l_pass_act_max_x:pass_act_max_y))
l_pass_poly_max_x = l_wl_max_x2 + 2*pol_olap_act + nmos_passw
l_pass_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(l_wl_max_x2:wl_max_y,
l_pass_poly_max_x:wl_max_y + nmos_passl))
l_pass_con_min_x = l_pass_act_min_x + float(truncate(nmos_passw*50))/100 - 0.5*con_size
l_pass_con_max_x = l_pass_con_min_x + con_size
b_pass_con_min_y = pass_act_min_y + act_enc_con
t_pass_con_min_y = wl_max_y + nmos_passl + actcon_tgate_sp
l_b_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(l_pass_con_min_x:b_pass_con_min_y,
l_pass_con_max_x:b_pass_con_min_y + con_size))
```

```
l_t_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(l_pass_con_min_x:t_pass_con_min_y,
l_pass_con_max_x:t_pass_con_min_y + con_size))
l_b_pass_m1_max_x = l_pass_act_min_x + nmos_passw
b_pass_m1_max_y = pass_act_min_y + m1wd40_area
l_t_pass_m1_min_x = l_pass_act_max_x - nmos_passw
t_pass_m1_min_y = pass_act_max_y - act_enc_con - con_size - m1_enc_con
l_b_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(l_pass_act_min_x:pass_act_min_y,
l_b_pass_m1_max_x:b_pass_m1_max_y))
l_t_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(l_t_pass_m1_min_x:t_pass_m1_min_y,
l_pass_act_max_x:pass_act_max_y))
if(nmos_passw < (2*act_enc_con + con_size) ;adding contacts with sufficient area
  then
            l_pass_act_con_min_x = l_pass_con_min_x - act_enc_con
            l_pass_act_con_max_x = l_pass_con_max_x + act_enc_con
            b_pass_act_con_max_y = pass_act_min_y + 2*act_enc_con + con_size
            t_pass_act_con_min_y = pass_act_max_y - 2*act_enc_con - con_size
            l_b_pass_act_con_rect = dbCreateRect(sram_cell_cv list(active)
list(l_pass_act_con_min_x:pass_act_min_y, l_pass_act_con_max_x:b_pass_act_con_max_y))
            l_t_pass_act_con_rect = dbCreateRect(sram_cell_cv list(active)
list(l_pass_act_con_min_x:t_pass_act_con_min_y, l_pass_act_con_max_x:pass_act_max_y))
            l_b_pass_m1_area_rect = dbCreateRect(sram_cell_cv list(metal1)
list(l_pass_act_con_min_x:pass_act_min_y, l_pass_act_con_max_x:b_pass_m1_max_y))
            l_b_pass_m1_area_rect = dbCreateRect(sram_cell_cv list(metal1)
list(l_pass_act_con_min_x:t_pass_m1_min_y, l_pass_act_con_max_x:pass_act_max_y))
)
;RIGHT PASS TRANSITOR
if(nmos_passw < (2*act_enc_con + con_size)
  then
            r_pass_act_min_x = l_pass_act_con_max_x + act_sp + act_enc_con + 0.5*con_size -
float(truncate(nmos_passw*50))/100
else
            r_pass_act_min_x = l_pass_act_max_x + act_sp
)
r_pass_act_max_x = r_pass_act_min_x + nmos_passw
r_pass_act_rect = dbCreateRect(sram_cell_cv list(active) list(r_pass_act_min_x:pass_act_min_y,
r_pass_act_max_x:pass_act_max_y))
r_pass_poly_min_x = r_pass_act_min_x - pol_olap_act
r_pass_poly_max_x = r_pass_act_max_x + pol_olap_act
r_pass_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(r_pass_poly_min_x:wl_max_y,
r_pass_poly_max_x:wl_max_y + nmos_passl))
r_pass_con_min_x = r_pass_act_min_x + float(truncate(nmos_passw*50))/100 - 0.5*con_size
r_pass_con_max_x = r_pass_con_min_x + con_size
r_b_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_pass_con_min_x:b_pass_con_min_y,
r_pass_con_max_x:b_pass_con_min_y + con_size))
r_t_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_pass_con_min_x:t_pass_con_min_y,
r_pass_con_max_x:t_pass_con_min_y + con_size))
r_b_pass_m1_max_x = r_pass_act_min_x + nmos_passw
r_t_pass_m1_min_x = r_pass_act_max_x - nmos_passw
r_b_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_pass_act_min_x:pass_act_min_y,
r_b_pass_m1_max_x:b_pass_m1_max_y))
r_t_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_t_pass_m1_min_x:t_pass_m1_min_y,
r_pass_act_max_x:pass_act_max_y))
if(nmos_passw < (2*act_enc_con + con_size) ;adding contacts with sufficient area
  then
            r_pass_act_con_min_x = r_pass_con_min_x - act_enc_con
            r_pass_act_con_max_x = r_pass_con_max_x + act_enc_con
            r_b_pass_act_con_rect = dbCreateRect(sram_cell_cv list(active)
list(r_pass_act_con_min_x:pass_act_min_y, r_pass_act_con_max_x:b_pass_act_con_max_y))
            r_t_pass_act_con_rect = dbCreateRect(sram_cell_cv list(active)
list(r_pass_act_con_min_x:t_pass_act_con_min_y, r_pass_act_con_max_x:pass_act_max_y))
            r_b_pass_m1_area_rect = dbCreateRect(sram_cell_cv list(metal1)
list(r_pass_act_con_min_x:pass_act_min_y, r_pass_act_con_max_x:b_pass_m1_max_y))
            r_b_pass_m1_area_rect = dbCreateRect(sram_cell_cv list(metal1)
list(r_pass_act_con_min_x:t_pass_m1_min_y, r_pass_act_con_max_x:pass_act_max_y))
)

bl_min_x = l_pass_act_min_x + float(truncate(nmos_passw*50))/100 - 0.5*m2wd
bl_max_x = bl_min_x + m2wd
bl_bar_min_x = r_pass_act_min_x + float(truncate(nmos_passw*50))/100 - 0.5*m2wd
bl_bar_max_x = bl_bar_min_x + m2wd
bl_via_min_y = b_pass_con_min_y - 0.5*(via_size - con_size)

;RIGHT WORDLINE CONTACT
if(nmos_passw < (2*act_enc_con + con_size)
  then
            r_wl_m1_min_x = max(r_pass_act_con_max_x + m1_m1sp,bl_bar_max_x + m2_m2sp)
else
            r_wl_m1_min_x = max(r_pass_act_max_x + m1_m1sp,bl_bar_max_x + m2_m2sp)
)
r_wl_m1_max_x = r_wl_m1_min_x + via_size
m1wd28_area = 0.62 ;min m1 area width of 0.62 when other dimension is 0.28
r_wl_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_wl_m1_min_x:pass_act_min_y,
r_wl_m1_max_x:pass_act_min_y + m1wd28_area))
r_wl_poly_min_x = r_wl_m1_min_x + m1_enc_con - pol_enc_con
r_wl_poly_max_x = r_wl_poly_min_x + 2*pol_enc_con + con_size
r_wl_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(r_wl_poly_min_x:wl_min_y,
r_wl_poly_max_x:wl_max_y))
r_wl_con_min_x = r_wl_poly_min_x + pol_enc_con
r_wl_con_max_x = r_wl_con_min_x + con_size
r_wl_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_wl_con_min_x:wl_min_y + pol_enc_con,
r_wl_con_max_x:wl_max_y - pol_enc_con))
r_wl_via_rect = dbCreateRect(sram_cell_cv list(via) list(r_wl_m1_min_x:wl_via_min_y,
r_wl_m1_min_x + via_size:wl_max_y))
cell_width = r_wl_m1_min_x + via_size + 2*m1_m1sp + m1wd
r_wl_m2_max_y = wl_via_min_y + 0.35 ;adjustment value to maintain minimum m2 area of 0.3136
r_wl_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(r_wl_m1_min_x:wl_via_min_y,
cell_width:r_wl_m2_max_y))
r_wl_poly2_rect = dbCreateRect(sram_cell_cv list(poly) list(r_pass_poly_max_x:wl_max_y,
r_wl_poly_max_x:wl_max_y + nmos_passl))

;Finish GND Rail
gnd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:0, cell_width:gnd_height))
ptap_bp_rect = dbCreateRect(sram_cell_cv list(bp) list(0:0, cell_width:gnd_height))

;INVERTER NMOS TRANSITORS
inv_act_min_x = act_welledg_sp
if(nmos_invw < (2*act_enc_con + con_size)
  then
            inv_act_min_y = pass_act_max_y + 2*m1_m1sp + m1wd + m1_enc_con + 0.5*con_size -
float(truncate(nmos_invw*50))/100
else
            inv_act_min_y = pass_act_max_y + 2*m1_m1sp + m1wd
```

```
)
inv_act_max_x = inv_act_min_x + 2*act_enc_con + 3*con_size + 4*actcon_tgate_sp + 2*nmos_invl
inv_act_max_y = inv_act_min_y + nmos_invw
inv_act_rect = dbCreateRect(sram_cell_cv list(active) list(inv_act_min_x:inv_act_min_y,
inv_act_max_x:inv_act_max_y))
inv_con_min_y = inv_act_min_y + float(truncate(nmos_invw*50))/100 - 0.5*con_size
inv_con_max_y = inv_con_min_y + con_size
inv_con_min_x1 = inv_act_min_x + act_enc_con
inv_con_min_x2 = inv_con_min_x1 + con_size + 2*actcon_tgate_sp + nmos_invl
inv_con_min_x3 = inv_act_max_x - act_enc_con - con_size
inv_con_rect1 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x1:inv_con_min_y,
inv_con_min_x1 + con_size:inv_con_max_y))
inv_con_rect2 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x2:inv_con_min_y,
inv_con_min_x2 + con_size:inv_con_max_y))
inv_con_rect3 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x3:inv_con_min_y,
inv_con_min_x3 + con_size:inv_con_max_y))
inv_m1_max_x1 = inv_act_min_x + act_enc_con + con_size + m1_enc_con
inv_m1_min_x2 = inv_con_min_x2 - m1_enc_con
inv_m1_max_x2 = inv_con_min_x2 + con_size + m1_enc_con
inv_m1_min_x3 = inv_act_max_x - act_enc_con - con_size - m1_enc_con
if(nmos_invw < (2*act_enc_con + con_size)
  then
        inv_m1_min_y = inv_con_min_y - m1_enc_con
        inv_m1_max_y = inv_con_max_y + act_enc_con
else
        inv_m1_min_y = inv_act_min_y
        inv_m1_max_y = inv_act_max_y
)

inv_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_act_min_x:inv_m1_min_y,
inv_m1_max_x1:inv_m1_max_y))
inv_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:inv_m1_min_y,
inv_m1_max_x2:inv_m1_max_y))
inv_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x3:inv_m1_min_y,
inv_act_max_x:inv_m1_max_y))
inv_poly_min_x1 = inv_con_min_x1 + con_size + actcon_tgate_sp
inv_poly_max_x1 = inv_poly_min_x1 + nmos_invl
inv_poly_min_x2 = inv_con_min_x2 + con_size + actcon_tgate_sp
inv_poly_max_x2 = inv_poly_min_x2 + nmos_invl
if(nmos_invw < (2*act_enc_con + con_size) ;different poly overlaps depending on if active corner nearby
  then
        inv_poly_min_y = inv_act_min_y - pol_olap_act_cor
else
        inv_poly_min_y = inv_act_min_y - pol_olap_act
)
inv_poly_max_y = inv_act_max_y + pol_olap_act
inv_poly_rect1 = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x1:inv_poly_min_y,
inv_poly_max_x1:inv_poly_max_y))
inv_poly_rect2 = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_poly_min_y,
inv_poly_max_x2:inv_poly_max_y))
if(nmos_invw < (2*act_enc_con + con_size) ;adding contacts with sufficient area
  then
        inv_act_con_min_y = inv_con_min_y - act_enc_con
        inv_act_con_max_y = inv_con_max_y + act_enc_con
        l_inv_act_con_max_x = inv_act_min_x + 2*act_enc_con + con_size
        r_inv_act_con_min_x = inv_act_max_x - 2*act_enc_con - con_size
```

```
            inv_act_con_min_x2 = inv_con_min_x2 - act_enc_con
            inv_act_con_max_x2 = inv_con_min_x2 + con_size + act_enc_con
            inv_act_con_rect1 = dbCreateRect(sram_cell_cv list(active)
list(inv_act_min_x:inv_act_con_min_y, l_inv_act_con_max_x:inv_act_con_max_y))
            inv_act_con_rect2 = dbCreateRect(sram_cell_cv list(active)
list(inv_act_con_min_x2:inv_act_con_min_y, inv_act_con_max_x2:inv_act_con_max_y))
            inv_act_con_rect3 = dbCreateRect(sram_cell_cv list(active)
list(r_inv_act_con_min_x:inv_act_con_min_y, inv_act_max_x:inv_act_con_max_y))
)

;PMOS INVERTER TRANSITORS
;active
inv_pact_min_x = act_welledg_sp
;if(nmos_invw < (2*act_enc_con + con_size)
  ;then
            ;inv_pact_min_y = inv_act_con_max_y + 2*m1_m1sp + m1wd + m1_enc_con + con_size +
polcon_act_sp
;else
            ;inv_pact_min_y = inv_act_max_y + 2*m1_m1sp + m1wd + m1_enc_con + con_size +
polcon_act_sp
;)
inv_pact_min_y = inv_m1_max_y + 2*m1_m1sp + m1wd + m1_enc_con + con_size + polcon_act_sp
if(pmos_invw < (2*act_enc_con + con_size)
  then
            inv_pact_min_y = inv_pact_min_y + (0.5*act_con_size - float(truncate(pmos_invw*50))/100)
)
inv_pact_max_x = inv_pact_min_x + 2*act_enc_con + 3*con_size + 4*actcon_tgate_sp + 2*pmos_invl
inv_pact_max_y = inv_pact_min_y + pmos_invw
inv_pact_rect = dbCreateRect(sram_cell_cv list(active) list(inv_pact_min_x:inv_pact_min_y,
inv_pact_max_x:inv_pact_max_y))
;active contacts
invp_con_min_y = inv_pact_min_y + float(truncate(pmos_invw*50))/100 - 0.5*con_size
invp_con_max_y = invp_con_min_y + con_size
invp_con_min_x1 = inv_pact_min_x + act_enc_con
invp_con_min_x2 = invp_con_min_x1 + con_size + 2*actcon_tgate_sp + pmos_invl
invp_con_min_x3 = inv_pact_max_x - act_enc_con - con_size
invp_con_rect1 = dbCreateRect(sram_cell_cv list(contact) list(invp_con_min_x1:invp_con_min_y,
invp_con_min_x1 + con_size:invp_con_max_y))
invp_con_rect2 = dbCreateRect(sram_cell_cv list(contact) list(invp_con_min_x2:invp_con_min_y,
invp_con_min_x2 + con_size:invp_con_max_y))
invp_con_rect3 = dbCreateRect(sram_cell_cv list(contact) list(invp_con_min_x3:invp_con_min_y,
invp_con_min_x3 + con_size:invp_con_max_y))
;metal 1
invp_m1_max_x1 = inv_pact_min_x + act_enc_con + con_size + m1_enc_con
invp_m1_min_x2 = invp_con_min_x2 - m1_enc_con
invp_m1_max_x2 = invp_con_min_x2 + con_size + m1_enc_con
invp_m1_min_x3 = inv_pact_max_x - act_enc_con - con_size - m1_enc_con
invp_m1_min_y = invp_con_min_y - m1_enc_con
invp_m1_max_y = invp_con_max_y + m1_enc_con
invp_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_pact_min_x:invp_m1_min_y,
invp_m1_max_x1:invp_m1_max_y))
invp_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(invp_m1_min_x2:invp_m1_min_y,
invp_m1_max_x2:invp_m1_max_y))
invp_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(invp_m1_min_x3:invp_m1_min_y,
inv_pact_max_x:invp_m1_max_y))
;poly gates
```

```
invp_poly_min_x1 = invp_con_min_x1 + con_size + actcon_tgate_sp
invp_poly_max_x1 = invp_poly_min_x1 + nmos_invl
invp_poly_min_x2 = invp_con_min_x2 + con_size + actcon_tgate_sp
invp_poly_max_x2 = invp_poly_min_x2 + nmos_invl
invp_poly_min_y = inv_pact_min_y - pol_olap_act
if(pmos_invw < (2*act_enc_con + con_size) ;different poly overlaps depending on if active corner nearby
  then
        invp_poly_max_y = inv_pact_max_y + pol_olap_act_cor
else
        invp_poly_max_y = inv_pact_max_y + pol_olap_act
)
invp_poly_rect1 = dbCreateRect(sram_cell_cv list(poly) list(invp_poly_min_x1:invp_poly_min_y,
invp_poly_max_x1:invp_poly_max_y))
invp_poly_rect2 = dbCreateRect(sram_cell_cv list(poly) list(invp_poly_min_x2:invp_poly_min_y,
invp_poly_max_x2:invp_poly_max_y))
if(pmos_invw < (2*act_enc_con + con_size) ;adding contacts with sufficient area
  then
        invp_act_con_min_y = invp_con_min_y - act_enc_con
        invp_act_con_max_y = invp_con_max_y + act_enc_con
        l_invp_act_con_max_x = inv_pact_min_x + 2*act_enc_con + con_size
        r_invp_act_con_min_x = inv_pact_max_x - 2*act_enc_con - con_size
        invp_act_con_min_x2 = invp_con_min_x2 - act_enc_con
        invp_act_con_max_x2 = invp_con_min_x2 + con_size + act_enc_con
        invp_act_con_rect1 = dbCreateRect(sram_cell_cv list(active)
list(inv_pact_min_x:invp_act_con_min_y, l_invp_act_con_max_x:invp_act_con_max_y))
        invp_act_con_rect2 = dbCreateRect(sram_cell_cv list(active)
list(invp_act_con_min_x2:invp_act_con_min_y, invp_act_con_max_x2:invp_act_con_max_y))
        invp_act_con_rect3 = dbCreateRect(sram_cell_cv list(active)
list(r_invp_act_con_min_x:invp_act_con_min_y, inv_pact_max_x:invp_act_con_max_y))
        invp_nwell_min_y = invp_act_con_min_y - act_welledg_sp
else
        invp_nwell_min_y = inv_pact_min_y - act_welledg_sp
)
;bp
invp_bp_min_x = inv_pact_min_x - bp_enc_actwell
invp_bp_max_x = inv_pact_max_x + bp_enc_actwell
invp_bp_min_y = inv_pact_min_y - bp_enc_actpol
invp_bp_max_y = inv_pact_max_y + bp_enc_actpol
invp_bp_rect = dbCreateRect(sram_cell_cv list(bp) list(invp_bp_min_x:invp_bp_min_y,
invp_bp_max_x:invp_bp_max_y))
;nwell
inv_nwell_min_y = inv_pact_min_y - act_welledg_sp
cell_height = invp_bp_max_y + actnw_bpsp + 2*act_enc_con + con_size + 0.5*act_sp
inv_nwell_rect = dbCreateRect(sram_cell_cv list(nwell) list(0:invp_nwell_min_y,
cell_width:cell_height))

;;;;;DISPLAY CELL WIDTH AND HEIGHT;;;;;;;;;;;;;;
printf("Cell width: %2f um\nCell height: %2f\n", cell_width, cell_height)

;pin_rect = dbCreatePin(sram_cell_cv list("pin") list(0:0,cell_width:cell_height))

;VDD RAIL (TOP)
;ntaps
ntap_act_min_y = invp_bp_max_y + actnw_bpsp
ntap_act_max_y = ntap_act_min_y + 2*act_enc_con + con_size
ntap_act_rect1 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x:ntap_act_min_y,
ptap_act_max_x:ntap_act_max_y))
ntap_act_rect2 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x + ptap_inc:ntap_act_min_y,
ptap_act_max_x + ptap_inc:ntap_act_max_y))
ntap_act_rect3 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x +
2*ptap_inc:ntap_act_min_y, ptap_act_max_x + 2*ptap_inc:ntap_act_max_y))
ntap_act_rect4 = dbCreateRect(sram_cell_cv list(active) list(ptap_act_min_x +
3*ptap_inc:ntap_act_min_y, ptap_act_max_x + 3*ptap_inc:ntap_act_max_y))
ntap_con_min_y = ntap_act_min_y + act_enc_con
ntap_con_max_y = ntap_con_min_y + con_size
ntap_con_rect1 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x:ntap_con_min_y,
ptap_con_max_x:ntap_con_max_y))
ntap_con_rect2 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
ptap_inc:ntap_con_min_y, ptap_con_max_x + ptap_inc:ntap_con_max_y))
ntap_con_rect3 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
2*ptap_inc:ntap_con_min_y, ptap_con_max_x + 2*ptap_inc:ntap_con_max_y))
ntap_con_rect4 = dbCreateRect(sram_cell_cv list(contact) list(ptap_con_min_x +
3*ptap_inc:ntap_con_min_y, ptap_con_max_x + 3*ptap_inc:ntap_con_max_y))
vdd_m1_min_y = cell_height - gnd_height
vdd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:vdd_m1_min_y, cell_width:cell_height))

;BITLINES
BL_via_rect = dbCreateRect(sram_cell_cv list(via) list(bl_min_x:bl_via_min_y, bl_max_x:bl_via_min_y
+ via_size))
BL_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_min_x:0, bl_max_x:cell_height))
BL_bar_via_rect = dbCreateRect(sram_cell_cv list(via) list(bl_bar_min_x:bl_via_min_y,
bl_bar_max_x:bl_via_min_y + via_size))
BL_bar_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_bar_min_x:0,
bl_bar_max_x:cell_height))

;ROUTING
;routing begins at bottom and makes its way up to the top
;nmos pass to nmos inv drain/source connection 1
pass_inv_ds1_m1_min_x = inv_m1_max_x1 - m1wd
pass_inv_ds1_m1_rect = dbCreateRect(sram_cell_cv list(metal1)
list(pass_inv_ds1_m1_min_x:pass_act_max_y, inv_m1_max_x1:inv_m1_min_y))

;nmos pass to nmos inv drain/source connection 2
pass_inv_ds2_m1_max_x1 = bl_bar_max_x + m2_m2sp + m1_enc_via + via_size
pass_inv_ds2_m1_min_y1 = pass_act_max_y - via_size - m1_enc_via
pass_inv_ds2_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1)
list(r_pass_act_max_x:pass_inv_ds2_m1_min_y1, pass_inv_ds2_m1_max_x1:pass_act_max_y))
pass_inv_ds2_via_min_x1 = bl_bar_max_x + m2_m2sp
pass_inv_ds2_via_max_x1 = pass_inv_ds2_via_min_x1 + via_size
pass_inv_ds2_via_min_y1 = pass_act_max_y - via_size
pass_inv_ds2_via_rect1 = dbCreateRect(sram_cell_cv list(via)
list(pass_inv_ds2_via_min_x1:pass_inv_ds2_via_min_y1, pass_inv_ds2_via_max_x1:pass_act_max_y))
pass_inv_ds2_m2_min_x1 = bl_bar_max_x + m2_m2sp
if(r_pass_act_max_x < inv_act_max_x
  then
		pass_inv_ds2_m2_max_x1 = inv_m1_min_x3 + m1_m1sp
		pass_inv_ds2_m2_rect1 = dbCreateRect(sram_cell_cv list(metal2)
list(pass_inv_ds2_m2_min_x1:pass_inv_ds2_via_min_y1, pass_inv_ds2_m2_max_x1:pass_act_max_y))
else
		pass_inv_ds2_m2_max_x1 = pass_inv_ds2_m2_min_x1
)
pass_inv_ds2_m2_max_x2 = pass_inv_ds2_m2_max_x1 + via_size
```

```
pass_inv_ds2_m2_max_y2 = inv_m1_min_y + via_size
pass_inv_ds2_m2_rect2 = dbCreateRect(sram_cell_cv list(metal2)
list(pass_inv_ds2_m2_max_x1:pass_inv_ds2_via_min_y1,
pass_inv_ds2_m2_max_x2:pass_inv_ds2_m2_max_y2))
pass_inv_ds2_via_min_x2 = pass_inv_ds2_m2_max_x2 - via_size
pass_inv_ds2_via_max_x2 = pass_inv_ds2_m2_max_x2
pass_inv_ds2_via_min_y2 = inv_m1_min_y
pass_inv_ds2_via_max_y2 = pass_inv_ds2_via_min_y2 + via_size
pass_inv_ds2_via_rect2 = dbCreateRect(sram_cell_cv list(via)
list(pass_inv_ds2_via_min_x2:pass_inv_ds2_via_min_y2,
pass_inv_ds2_via_max_x2:pass_inv_ds2_via_max_y2))
if((pass_inv_ds2_via_min_x2 > inv_act_max_x)
  then
            pass_inv_ds2_m1_min_x2 = inv_act_max_x
else
            pass_inv_ds2_m1_min_x2 = pass_inv_ds2_via_min_x2
)
pass_inv_ds2_m1_max_y2 = pass_inv_ds2_via_max_y2 + m1_enc_via
pass_inv_ds2_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(pass_inv_ds2_m1_min_x2:inv_m1_min_y, pass_inv_ds2_m2_max_x2:pass_inv_ds2_m1_max_y2))


;nmos inv sources to gnd
inv_sgd_m1_max_x1 = inv_m1_min_x2 + m1wd
inv_sgd_m1_min_y1 = inv_m1_min_y - m1_m1sp - m1wd
inv_sgd_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:inv_sgd_m1_min_y1,
inv_sgd_m1_max_x1:inv_m1_min_y))
inv_sgd_m1_max_x2 = max(pass_inv_ds2_m1_max_x1,r_wl_m1_max_x) + m1_m1sp + m1wd
inv_sgd_m1_max_y2 = inv_sgd_m1_min_y1 + m1wd
inv_sgd_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_sgd_m1_max_x1:inv_sgd_m1_min_y1, inv_sgd_m1_max_x2:inv_sgd_m1_max_y2))
inv_sgd_m1_min_x3 = inv_sgd_m1_max_x2 - m1wd
inv_sgd_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(inv_sgd_m1_min_x3:gnd_height,
inv_sgd_m1_max_x2:inv_sgd_m1_min_y1))

;inverter pmos and nmos gate connections
;inv_gate_ln = max(pmos_invl, nmos_invl) - might not need this
inv_gate1_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x1:inv_poly_max_y,
inv_poly_max_x1:invp_poly_min_y))
inv_gate2_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_poly_max_y,
inv_poly_max_x2:invp_poly_min_y))

;inv input to output 1 (left input to right output)
inv_io1_poly_min_x = inv_poly_max_x1 - 2*pol_enc_con - con_size
inv_io1_poly_min_y = inv_m1_max_y + 2*m1_m1sp + m1wd + m1_enc_con - pol_enc_con
inv_io1_poly_max_y = inv_io1_poly_min_y + 2*pol_enc_con + con_size
inv_io1_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_io1_poly_min_x:inv_io1_poly_min_y,
inv_poly_max_x1:inv_io1_poly_max_y))
inv_io1_m1_min_x = inv_io1_poly_min_x + pol_enc_con - m1_enc_con
inv_io1_m1_min_y = inv_io1_poly_min_y + pol_enc_con - m1_enc_con
inv_io1_m1_max_x = inv_io1_m1_min_x + 2*m1_enc_con + con_size
inv_io1_m1_max_y = inv_io1_m1_min_y + 2*m1_enc_con + con_size
inv_io1_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(inv_io1_m1_min_x:inv_io1_m1_min_y,
inv_io1_m1_max_x:inv_io1_m1_max_y))
inv_io1_con_min_x = inv_poly_max_x1 - pol_enc_con - con_size
inv_io1_con_max_x = inv_poly_max_x1 - pol_enc_con
```

```
inv_io1_con_min_y = inv_io1_poly_min_y + pol_enc_con
inv_io1_con_max_y = inv_io1_con_min_y + con_size
inv_io1_con_rect = dbCreateRect(sram_cell_cv list(contact) list(inv_io1_con_min_x:inv_io1_con_min_y,
inv_io1_con_max_x:inv_io1_con_max_y))
inv_io1_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_io1_m1_max_x:inv_io1_m1_min_y,
pass_inv_ds2_m2_max_x2:inv_io1_m1_max_y))
inv_io1_m1_min_x2 = inv_pact_max_x - m1wd
inv_io1_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io1_m1_min_x2:inv_io1_m1_max_y, inv_pact_max_x:invp_m1_min_y))
inv_io1_m1_min_x3 = pass_inv_ds2_via_min_x2
inv_io1_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io1_m1_min_x3:pass_inv_ds2_m1_max_y2, pass_inv_ds2_m2_max_x2:inv_io1_m1_min_y))

;inv input to output 2 (right input to left output)
inv_io2_poly_max_x = inv_poly_min_x2 + 2*pol_enc_con + con_size
inv_io2_poly_min_y = inv_m1_max_y + m1_m1sp + m1_enc_con - pol_enc_con
inv_io2_poly_max_y = inv_io2_poly_min_y + 2*pol_enc_con + con_size
inv_io2_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_io2_poly_min_y,
inv_io2_poly_max_x:inv_io2_poly_max_y))
inv_io2_m1_min_x = inv_poly_min_x2 + pol_enc_con - m1_enc_con
inv_io2_m1_min_y = inv_io2_poly_min_y + pol_enc_con - m1_enc_con
inv_io2_m1_max_x = inv_io2_m1_min_x + 2*m1_enc_con + con_size
inv_io2_m1_max_y = inv_io2_m1_min_y + 2*m1_enc_con + con_size
inv_io2_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(inv_poly_min_x2:inv_io2_m1_min_y,
inv_io2_m1_max_x:inv_io2_m1_max_y))
inv_io2_con_min_x = inv_poly_min_x2 + pol_enc_con
inv_io2_con_max_x = inv_poly_min_x2 + pol_enc_con + con_size
inv_io2_con_min_y = inv_io2_poly_min_y + pol_enc_con
inv_io2_con_max_y = inv_io2_con_min_y + con_size
inv_io2_con_rect = dbCreateRect(sram_cell_cv list(contact) list(inv_io2_con_min_x:inv_io2_con_min_y,
inv_io2_con_max_x:inv_io2_con_max_y))
inv_io2_m1_min_x1 = inv_io1_poly_min_x - m1_m1sp - m1wd
inv_io2_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io2_m1_min_x1:inv_io2_m1_min_y, inv_poly_min_x2:inv_io2_m1_max_y))
inv_io2_m1_max_x2 = inv_act_min_x + m1wd
inv_io2_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(inv_act_min_x:inv_act_max_y,
inv_io2_m1_max_x2:inv_io2_m1_min_y))
inv_io2_m1_max_x3 = inv_io2_m1_min_x1 + m1wd
inv_io2_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io2_m1_min_x1:inv_io2_m1_max_y, inv_io2_m1_max_x3:invp_m1_max_y))
if(inv_io2_m1_max_x3 < inv_pact_min_x
  then
		inv_io2_m1_rect4 = dbCreateRect(sram_cell_cv list(metal1)
list(invp_m1_min_x2:inv_pact_max_y, inv_pact_min_x:inv_pact_max_y))
)

;inv pmos source to vdd rail
inv_svdd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:invp_m1_max_y,
inv_m1_max_x2:vdd_m1_min_y))

;;;;;;PINS;;;;;;;;;;;;;;;;;;;;
vdd_pin_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:cell_height - m1wd, m1wd:cell_height))
vdd_net = dbCreateNet(sram_cell_cv "vdd!")
vdd_term = dbCreateTerm(vdd_net "" "powerInput")
vdd_pin = dbCreatePin(vdd_net vdd_pin_rect "vdd!")
```

```
gnd_pin_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:0, m1wd:m1wd))
gnd_net = dbCreateNet(sram_cell_cv "gnd!")
gnd_term = dbCreateTerm(gnd_net "" "powerInput")
gnd_pin = dbCreatePin(gnd_net gnd_pin_rect "gnd!")

BL_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_min_x:0, bl_max_x:m2wd))
BL_net = dbCreateNet(sram_cell_cv "BL")
BL_term = dbCreateTerm(BL_net "" "output")
BL_pin = dbCreatePin(BL_net BL_pin_rect "BL")

BL_bar_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_bar_min_x:0, bl_bar_max_x:m2wd))
BL_bar_net = dbCreateNet(sram_cell_cv "BL_bar")
BL_bar_term = dbCreateTerm(BL_bar_net "" "output")
BL_bar_pin = dbCreatePin(BL_bar_net BL_bar_pin_rect "BL_bar")

WL_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(0:wl_via_min_y, m2wd:wl_via_min_y +
m2wd))
WL_net = dbCreateNet(sram_cell_cv "WL")
WL_term = dbCreateTerm(WL_net "" "input")
WL_pin = dbCreatePin(WL_net WL_pin_rect "WL")
)
```

SCMOS SRAM Cell Layout

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;File: sram_c5n_cell_layout.il
;Functions: sram_c5n_cell_layout(cell_layout_lib cell_layout_cv nmos_passw nmos_invw pmos_invw)
;Description: Creates an SRAM cell layout for the ON Semiconductor SCMOS process technology.
;Input arguments:
;       cell_layout_lib: library name
;       cell_layout_cv: name of SRAM cell layout
;       nmos_passw: SRAM cell pass NMOS width
;       nmos_invw: SRAM cell inverter NMOS width
;       pmos_invw: SRAM cell inverter PMOS width
;Usage example with default transitor widths:
;       In command window type:
;               load("sram_c5n_cell_layout.il")
;               sram_c5n_cell_layout("SRAM" "cell_layout" 0 0 0)
;Usage example with custom transistor widths:
;       In command window type:
;               ;load("sram_c5n_cell_layout.il")
;               sram_c5n_cell_layout("SRAM" "cell_layout" 1.65 3.0 1.5)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Author: Brandon Hilgers
;Date: 17 June 2015
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(sram_c5n_cell_layout(cell_layout_lib cell_layout_cv nmos_passw nmos_invw pmos_invw)

load("drc_rules.il")
drc_c5n()

;CELL TRANSITOR PARAMETERS
```

```
nmos_passl = 0.6 ;nmos pass transistor gate length
nmos_invl = 0.6 ;nmos inverter transistor gate length
pmos_invl = 0.6 ;pmos inverter transistor gate length

if((nmos_passw==0)&&(nmos_invw==0)&&(pmos_invw==0)
  then
        nmos_passw = 1.5
        nmos_invw = 2.7
        pmos_invw = 1.5
        printf("Using default transistor sizes\n")
else
        npassw = truncate(nmos_passw*1000)
        ninvw = truncate(nmos_invw*1000)
        pinvw = truncate(pmos_invw*1000)
        if((nmos_passw<1.5)||(mod(npassw 150)!=0)||(nmos_invw<1.5)||(mod(ninvw
150)!=0)||(pmos_invw<1.5)||(mod(pinvw 150)!=0)
          then
                error("Transistor minimum width = 1.5um, and width must be multiple of 0.15um")
        )
)

;LAYER & COMPONENT NAMES
metal1 = "metal1"
metal2 = "metal2"
nactive = "nactive"
pactive = "pactive"
poly = "poly"
contact = "cc"
via = "via"

;CALCULATE CELL WIDTH AND LENGTH
cell_width = 2*con_size + 4*m1_enc_con + 4*m1_m1sp + 2*nmos_passw + actcon_act_sp -
act_enc_con + m1wd
cell_height = 2*psel_olap_act + 4*con_size + 2*act_enc_con + 6*m1_m1sp + 4*m1_enc_con +
2*actcon_tgate_sp + nmos_passl + 2*m1wd
;extend cell_height calculation to new line because previous line got too long
cell_height = cell_height + nmos_invw + pmos_invw + wellwd
printf("Cell width: %2f um\nCell height: %2f\n", cell_width, cell_height)

;OPEN CELL VIEWS
;SRAM cell layout cell view
if(dbOpenCellViewByType(cell_layout_lib cell_layout_cv "layout" "maskLayout" "r")!=nil
  then
        error("Please delete SRAM CELL cell view layout or choose a new cell view name")
)
sram_cell_cv = dbOpenCellViewByType(cell_layout_lib cell_layout_cv "layout" "maskLayout" "w")

;tap cell views
ptap_cv = dbOpenCellViewByType("NCSU_TechLib_ami06" "ptap" "layout" "" "r")
ntap_cv = dbOpenCellViewByType("NCSU_TechLib_ami06" "ntap" "layout" "" "r")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;LAYOUT OF SRAM CELL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;layout begins at bottom (gnd) and makes its way up to the top (vdd)
;GND RAIL
```

```
gnd_height = con_size + 2*act_enc_con + 2*psel_olap_act
gnd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:0, cell_width:gnd_height))
rail_cen_y = gnd_height/2
ptap_y = rail_cen_y - con_size/2
tap_sp = con_size + act_enc_con + actcon_act_sp
ptap_inst0 = dbCreateInst(sram_cell_cv ptap_cv "ptap_inst0" list(ptap_y,ptap_y) "R0")
ptap_inst1 = dbCreateInst(sram_cell_cv ptap_cv "ptap_inst1" list(ptap_y + tap_sp,ptap_y) "R0")
ptap_inst2 = dbCreateInst(sram_cell_cv ptap_cv "ptap_inst2" list(ptap_y + 2*tap_sp,ptap_y) "R0")
ptap_inst3 = dbCreateInst(sram_cell_cv ptap_cv "ptap_inst3" list(ptap_y + 3*tap_sp,ptap_y) "R0")

;LEFT WORDLINE CONTACT
wl_max_y = gnd_height + m1_m1sp + act_enc_con + con_size + actcon_tgate_sp
l_wl_max_x = 2*m1_enc_con + con_size
wl_min_y = wl_max_y - l_wl_max_x
wl_via_min_y = wl_min_y ;used in pin placement for word line in SRAM array
l_wl_max_x2 = l_wl_max_x + m1_m1sp - nsel_olap_act
l_wl_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(0:wl_min_y, l_wl_max_x:wl_max_y))
l_wl_con_rect = dbCreateRect(sram_cell_cv list(contact) list(m1_enc_con:wl_min_y + m1_enc_con,
l_wl_max_x - m1_enc_con:wl_max_y - m1_enc_con))
l_wl_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:wl_min_y, l_wl_max_x:wl_max_y))
l_wl_via_rect = dbCreateRect(sram_cell_cv list(via) list(m1_enc_con:wl_min_y + m1_enc_con,
l_wl_max_x - m1_enc_con:wl_max_y - m1_enc_con))
l_wl_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(0:wl_min_y, l_wl_max_x:wl_max_y))
l_wl_poly2_rect = dbCreateRect(sram_cell_cv list(poly) list(0:wl_max_y, l_wl_max_x2:wl_max_y +
nmos_passl))

;NMOS PASS TRANSISTORS - 'l' means left transitor, 'b' means bottom piece, 't' top
;nmos_pass_inst0 = dbCreateInst(sram_cell_cv nmos_cv "nmos_pass_inst0" list(3.45,5.1) "R90")
;LEFT PASS TRANSISTOR
pass_nsel_min_y = gnd_height + m1_m1sp - nsel_olap_act
pass_nsel_max_y = wl_max_y + nmos_passl + actcon_tgate_sp + con_size + act_enc_con +
nsel_olap_act
nmos_pass_nselw = nmos_passw + 2*nsel_olap_act
l_pass_nsel_rect = dbCreateRect(sram_cell_cv list("nselect") list(l_wl_max_x2:pass_nsel_min_y,
l_wl_max_x2 + nmos_pass_nselw:pass_nsel_max_y))
l_pass_act_min_x = l_wl_max_x2 + nsel_olap_act
pass_act_min_y = pass_nsel_min_y + nsel_olap_act
l_pass_act_max_x = l_wl_max_x2 + nmos_pass_nselw - nsel_olap_act
pass_act_max_y = pass_nsel_max_y - nsel_olap_act
l_pass_act_rect = dbCreateRect(sram_cell_cv list(nactive) list(l_pass_act_min_x:pass_act_min_y,
l_pass_act_max_x:pass_act_max_y))
l_pass_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(l_wl_max_x2:wl_max_y, l_wl_max_x2 +
nmos_pass_nselw:wl_max_y + nmos_passl))
l_pass_con_min_x = l_pass_act_min_x + act_enc_con
l_pass_con_max_x = l_pass_con_min_x + con_size
b_pass_con_min_y = pass_act_min_y + act_enc_con
t_pass_con_min_y = wl_max_y + nmos_passl + actcon_tgate_sp
pass_con_num = truncate((nmos_passw - 2*act_enc_con + con_consp)/(con_size + con_consp))
con_incr = con_size + con_consp
for(ac 0 pass_con_num-1 ;increment ac (active contact)
        l_b_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(l_pass_con_min_x +
ac*con_incr:b_pass_con_min_y, l_pass_con_max_x + ac*con_incr:b_pass_con_min_y + con_size))
        l_t_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(l_pass_con_min_x +
ac*con_incr:t_pass_con_min_y, l_pass_con_max_x + ac*con_incr:t_pass_con_min_y + con_size))
)
l_b_pass_m1_max_x = l_pass_act_min_x + nmos_passw
```

```
b_pass_m1_max_y = pass_act_min_y + 2*m1_enc_con + con_size
l_t_pass_m1_min_x = l_pass_act_max_x - nmos_passw
t_pass_m1_min_y = pass_act_max_y - 2*m1_enc_con - con_size
l_b_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(l_pass_act_min_x:pass_act_min_y,
l_b_pass_m1_max_x:b_pass_m1_max_y))
l_t_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(l_t_pass_m1_min_x:t_pass_m1_min_y,
l_pass_act_max_x:pass_act_max_y))

;RIGHT PASS TRANSITOR
r_pass_nsel_rect = dbCreateRect(sram_cell_cv list("nselect") list(l_wl_max_x2:pass_nsel_min_y,
l_wl_max_x2 + 2*nmos_pass_nselw:pass_nsel_max_y))
r_pass_act_min_x = l_pass_act_max_x + 2*nsel_olap_act
r_pass_act_max_x = l_wl_max_x2 + 2*nmos_pass_nselw - nsel_olap_act
r_pass_act_rect = dbCreateRect(sram_cell_cv list(nactive) list(r_pass_act_min_x:pass_act_min_y,
r_pass_act_max_x:pass_act_max_y))
r_pass_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(l_wl_max_x2 +
nmos_pass_nselw:wl_max_y, l_wl_max_x2 + 2*nmos_pass_nselw:wl_max_y + nmos_passl))
r_pass_con_min_x = r_pass_act_min_x + act_enc_con
r_pass_con_max_x = r_pass_con_min_x + con_size
for(ac 0 pass_con_num-1 ;increment ac (active contact)
        r_b_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_pass_con_min_x +
ac*con_incr:b_pass_con_min_y, r_pass_con_max_x + ac*con_incr:b_pass_con_min_y + con_size))
        r_t_pass_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_pass_con_min_x +
ac*con_incr:t_pass_con_min_y, r_pass_con_max_x + ac*con_incr:t_pass_con_min_y + con_size))
)
r_b_pass_m1_max_x = r_pass_act_min_x + nmos_passw
r_t_pass_m1_min_x = r_pass_act_max_x - nmos_passw
r_b_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_pass_act_min_x:pass_act_min_y,
r_b_pass_m1_max_x:b_pass_m1_max_y))
r_t_pass_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_t_pass_m1_min_x:t_pass_m1_min_y,
r_pass_act_max_x:pass_act_max_y))

;BITLINES
bl_min_x = l_pass_con_min_x - m2_enc_via
bl_max_x = l_pass_con_max_x + m2_enc_via
bl_bar_min_x = r_pass_con_min_x - m2_enc_via
bl_bar_max_x = r_pass_con_max_x + m2_enc_via
BL_via_rect = dbCreateRect(sram_cell_cv list(via) list(l_pass_con_min_x:b_pass_con_min_y,
l_pass_con_max_x:b_pass_con_min_y + con_size))
BL_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_min_x:0, bl_max_x:cell_height))
BL_bar_via_rect = dbCreateRect(sram_cell_cv list(via) list(r_pass_con_min_x:b_pass_con_min_y,
r_pass_con_max_x:b_pass_con_min_y + con_size))
BL_bar_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_bar_min_x:0,
bl_bar_max_x:cell_height))

;RIGHT WORDLINE CONTACT
r_wl_min_x = r_pass_act_max_x + m1_m1sp
r_wl_max_x = r_wl_min_x + 2*m1_enc_con + con_size
r_wl_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(r_wl_min_x:wl_min_y,
r_wl_max_x:wl_max_y))
r_wl_con_rect = dbCreateRect(sram_cell_cv list(contact) list(r_wl_min_x + m1_enc_con:wl_min_y +
m1_enc_con, r_wl_max_x - m1_enc_con:wl_max_y - m1_enc_con))
r_wl_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(r_wl_min_x:wl_min_y,
r_wl_max_x:wl_max_y))
r_wl_via_rect = dbCreateRect(sram_cell_cv list(via) list(r_wl_min_x + m1_enc_con:wl_min_y +
m1_enc_con, r_wl_max_x - m1_enc_con:wl_max_y - m1_enc_con))
```

```
r_wl_m2_rect = dbCreateRect(sram_cell_cv list(metal2) list(r_wl_min_x:wl_min_y,
cell_width:wl_max_y))
r_wl_poly2_rect = dbCreateRect(sram_cell_cv list(poly) list(r_pass_act_max_x:wl_max_y,
r_wl_max_x:wl_max_y + nmos_passl))

;INVERTER NMOS TRANSITORS
inv_nsel_min_x = act_welledg_sp + 0.5*pmos_invl - 0.5*nmos_invl - nsel_olap_act
inv_nsel_min_y = pass_act_max_y + 2*m1_m1sp + m1wd - nsel_olap_act
inv_nsel_max_x = inv_nsel_min_x + 2*nsel_olap_act + 2*act_enc_con + 3*con_size +
4*actcon_tgate_sp + 2*nmos_invl
inv_nsel_max_y = inv_nsel_min_y + 2*nsel_olap_act + nmos_invw
inv_nsel_rect = dbCreateRect(sram_cell_cv list("nselect") list(inv_nsel_min_x:inv_nsel_min_y,
inv_nsel_max_x:inv_nsel_max_y))
inv_act_min_x = inv_nsel_min_x + nsel_olap_act
inv_act_min_y = inv_nsel_min_y + nsel_olap_act
inv_act_max_x = inv_nsel_max_x - nsel_olap_act
inv_act_max_y = inv_nsel_max_y - nsel_olap_act
inv_act_rect = dbCreateRect(sram_cell_cv list(nactive) list(inv_act_min_x:inv_act_min_y,
inv_act_max_x:inv_act_max_y))
inv_con_min_y = inv_act_min_y + act_enc_con
inv_con_max_y = inv_con_min_y + con_size
inv_con_min_x1 = inv_act_min_x + act_enc_con
inv_con_min_x2 = 0.5*inv_nsel_max_x + 0.5*inv_nsel_min_x - 0.5*con_size
inv_con_min_x3 = inv_act_max_x - act_enc_con - con_size
inv_con_num = truncate((nmos_invw - 2*act_enc_con + con_consp)/(con_size + con_consp))
for(ac 0 inv_con_num-1 ;increment ac (active contact)
        inv_con_rect1 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x1:inv_con_min_y +
ac*con_incr, inv_con_min_x1 + con_size:inv_con_max_y + ac*con_incr))
        inv_con_rect2 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x2:inv_con_min_y +
ac*con_incr, inv_con_min_x2 + con_size:inv_con_max_y + ac*con_incr))
        inv_con_rect3 = dbCreateRect(sram_cell_cv list(contact) list(inv_con_min_x3:inv_con_min_y +
ac*con_incr, inv_con_min_x3 + con_size:inv_con_max_y + ac*con_incr))
)
inv_m1_max_x1 = inv_act_min_x + 2*m1_enc_con + con_size
inv_m1_min_x2 = inv_con_min_x2 - m1_enc_con
inv_m1_max_x2 = inv_con_min_x2 + con_size + m1_enc_con
inv_m1_min_x3 = inv_act_max_x - 2*m1_enc_con - con_size
inv_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_act_min_x:inv_act_min_y,
inv_m1_max_x1:inv_act_max_y))
inv_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:inv_act_min_y,
inv_m1_max_x2:inv_act_max_y))
inv_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x3:inv_act_min_y,
inv_act_max_x:inv_act_max_y))
inv_poly_min_x1 = inv_con_min_x1 + con_size + actcon_tgate_sp
inv_poly_max_x1 = inv_poly_min_x1 + nmos_invl
inv_poly_min_x2 = inv_con_min_x2 + con_size + actcon_tgate_sp
inv_poly_max_x2 = inv_poly_min_x2 + nmos_invl
inv_poly_rect1 = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x1:inv_nsel_min_y,
inv_poly_max_x1:inv_nsel_max_y))
inv_poly_rect2 = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_nsel_min_y,
inv_poly_max_x2:inv_nsel_max_y))

;PMOS INVERTER TRANSITORS
;nwell
inv_nwell_min_y = inv_act_max_y + 3*m1_m1sp + m1wd + 2*m1_enc_con + con_size - act_welledg_sp
inv_nwell_rect = dbCreateRect(sram_cell_cv list("nwell") list(0:inv_nwell_min_y,
cell_width:cell_height))
;pselect
inv_psel_min_x = act_welledg_sp - psel_olap_act
inv_psel_min_y = inv_nwell_min_y + act_welledg_sp - psel_olap_act
inv_psel_max_x = inv_psel_min_x + 2*psel_olap_act + 2*act_enc_con + 3*con_size +
4*actcon_tgate_sp + 2*pmos_invl
inv_psel_max_y = inv_psel_min_y + 2*psel_olap_act + pmos_invw
inv_psel_rect = dbCreateRect(sram_cell_cv list("pselect") list(inv_psel_min_x:inv_psel_min_y,
inv_psel_max_x:inv_psel_max_y))
;pactive
inv_pact_min_x = inv_psel_min_x + psel_olap_act
inv_pact_min_y = inv_psel_min_y + psel_olap_act
inv_pact_max_x = inv_psel_max_x - psel_olap_act
inv_pact_max_y = inv_psel_max_y - psel_olap_act
inv_pact_rect = dbCreateRect(sram_cell_cv list(pactive) list(inv_pact_min_x:inv_pact_min_y,
inv_pact_max_x:inv_pact_max_y))
;active contacts
invp_con_min_y = inv_pact_min_y + act_enc_con
invp_con_max_y = invp_con_min_y + con_size
invp_con_min_x1 = inv_pact_min_x + act_enc_con
invp_con_min_x2 = 0.5*inv_psel_max_x + 0.5*inv_psel_min_x - 0.5*con_size
invp_con_min_x3 = inv_pact_max_x - act_enc_con - con_size
invp_con_num = truncate((pmos_invw - 2*act_enc_con + con_consp)/(con_size + con_consp))
for(ac 0 invp_con_num-1 ;increment ac (active contact)
        invp_con_rect1 = dbCreateRect(sram_cell_cv list(contact)
list(invp_con_min_x1:invp_con_min_y + ac*con_incr, invp_con_min_x1 + con_size:invp_con_max_y +
ac*con_incr))
        invp_con_rect2 = dbCreateRect(sram_cell_cv list(contact)
list(invp_con_min_x2:invp_con_min_y + ac*con_incr, invp_con_min_x2 + con_size:invp_con_max_y +
ac*con_incr))
        invp_con_rect3 = dbCreateRect(sram_cell_cv list(contact)
list(invp_con_min_x3:invp_con_min_y + ac*con_incr, invp_con_min_x3 + con_size:invp_con_max_y +
ac*con_incr))
)
;metal 1
invp_m1_max_x1 = inv_pact_min_x + 2*m1_enc_con + con_size
invp_m1_min_x2 = invp_con_min_x2 - m1_enc_con
invp_m1_max_x2 = invp_con_min_x2 + con_size + m1_enc_con
invp_m1_min_x3 = inv_pact_max_x - 2*m1_enc_con - con_size
invp_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_pact_min_x:inv_pact_min_y,
invp_m1_max_x1:inv_pact_max_y))
invp_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(invp_m1_min_x2:inv_pact_min_y,
invp_m1_max_x2:inv_pact_max_y))
invp_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(invp_m1_min_x3:inv_pact_min_y,
inv_pact_max_x:inv_pact_max_y))
;poly gates
invp_poly_min_x1 = invp_con_min_x1 + con_size + actcon_tgate_sp
invp_poly_max_x1 = invp_poly_min_x1 + pmos_invl
invp_poly_min_x2 = invp_con_min_x2 + con_size + actcon_tgate_sp
invp_poly_max_x2 = invp_poly_min_x2 + pmos_invl
invp_poly_rect1 = dbCreateRect(sram_cell_cv list(poly) list(invp_poly_min_x1:inv_psel_min_y,
invp_poly_max_x1:inv_psel_max_y))
invp_poly_rect2 = dbCreateRect(sram_cell_cv list(poly) list(invp_poly_min_x2:inv_psel_min_y,
invp_poly_max_x2:inv_psel_max_y))

;VDD RAIL (TOP)
```

```
ntap_y = cell_height - 0.5*(con_size + wellwd)
ntap_x = 0.5*(wellwd - con_size)
ntap_inst0 = dbCreateInst(sram_cell_cv ntap_cv "ntap_inst0" list(ntap_x,ntap_y) "R0")
ntap_inst1 = dbCreateInst(sram_cell_cv ntap_cv "ntap_inst1" list(ntap_x + tap_sp,ntap_y) "R0")
ntap_inst2 = dbCreateInst(sram_cell_cv ntap_cv "ntap_inst2" list(ntap_x + 2*tap_sp,ntap_y) "R0")
ntap_inst3 = dbCreateInst(sram_cell_cv ntap_cv "ntap_inst3" list(ntap_x + 3*tap_sp,ntap_y) "R0")
vdd_m1_min_y = cell_height - gnd_height
vdd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:vdd_m1_min_y, cell_width:cell_height))

;ROUTING
;routing begins at bottom and makes its way up to the top
;nmos pass to nmos inv drain/source connection 1
pass_inv_ds1_m1_max_x = inv_act_min_x + m1wd
pass_inv_ds1_m1_min_y = pass_act_max_y - 2*m1_enc_con - con_size
pass_inv_ds1_m1_rect = dbCreateRect(sram_cell_cv list(metal1)
list(inv_act_min_x:pass_inv_ds1_m1_min_y, pass_inv_ds1_m1_max_x:inv_act_min_y))

;nmos pass to nmos inv drain/source connection 2
pass_inv_ds2_m1_max_x1 = r_pass_act_max_x + m1_m1sp + 2*m1_enc_con + via_size
pass_inv_ds2_m1_min_y1 = pass_act_max_y - 2*m1_enc_con - con_size
pass_inv_ds2_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1)
list(r_pass_act_max_x:pass_inv_ds2_m1_min_y1, pass_inv_ds2_m1_max_x1:pass_act_max_y))
pass_inv_ds2_via_min_x1 = pass_inv_ds2_m1_max_x1 - m1_enc_via - via_size
pass_inv_ds2_via_max_x1 = pass_inv_ds2_m1_max_x1 - m1_enc_via
pass_inv_ds2_via_min_y1 = pass_act_max_y - m1_enc_con - con_size
pass_inv_ds2_via_max_y1 = pass_act_max_y - m1_enc_con
pass_inv_ds2_via_rect1 = dbCreateRect(sram_cell_cv list(via)
list(pass_inv_ds2_via_min_x1:pass_inv_ds2_via_min_y1,
pass_inv_ds2_via_max_x1:pass_inv_ds2_via_max_y1))
pass_inv_ds2_m2_min_x1 = pass_inv_ds2_m1_max_x1 - 2*m2_enc_via - via_size
if(r_pass_act_max_x < inv_act_max_x
  then
		pass_inv_ds2_m2_max_x1 = inv_act_max_x + m1_m1sp
		pass_inv_ds2_m2_rect1 = dbCreateRect(sram_cell_cv list(metal2)
list(pass_inv_ds2_m2_min_x1:pass_inv_ds2_m1_min_y1, pass_inv_ds2_m2_max_x1:pass_act_max_y))
else
		pass_inv_ds2_m2_max_x1 = r_pass_act_max_x + m1_m1sp
)
pass_inv_ds2_m2_max_x2 = pass_inv_ds2_m2_max_x1 + 2*m2_enc_via + via_size
pass_inv_ds2_m2_rect2 = dbCreateRect(sram_cell_cv list(metal2)
list(pass_inv_ds2_m2_max_x1:pass_inv_ds2_m1_min_y1, pass_inv_ds2_m2_max_x2:inv_act_max_y))
pass_inv_ds2_via_min_x2 = pass_inv_ds2_m2_max_x1 + m1_enc_via
pass_inv_ds2_via_max_x2 = pass_inv_ds2_via_min_x2 + via_size
pass_inv_ds2_via_min_y2 = inv_act_max_y - m1_enc_via - via_size
pass_inv_ds2_via_max_y2 = inv_act_max_y - m1_enc_via
pass_inv_ds2_via_rect2 = dbCreateRect(sram_cell_cv list(via)
list(pass_inv_ds2_via_min_x2:pass_inv_ds2_via_min_y2,
pass_inv_ds2_via_max_x2:pass_inv_ds2_via_max_y2))
pass_inv_ds2_m1_min_y2 = inv_act_max_y - 2*m1_enc_via - via_size
pass_inv_ds2_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_act_max_x:pass_inv_ds2_m1_min_y2, pass_inv_ds2_m2_max_x2:inv_act_max_y))



;nmos inv sources to gnd
inv_sgd_m1_max_x1 = inv_m1_min_x2 + m1wd
inv_sgd_m1_min_y1 = inv_act_min_y - m1_m1sp - m1wd
```

```
inv_sgd_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:inv_sgd_m1_min_y1,
inv_sgd_m1_max_x1:inv_act_min_y))
inv_sgd_m1_max_x2 = r_pass_act_max_x + 2*m1_m1sp + 2*m1_enc_con + con_size + m1wd
inv_sgd_m1_max_y2 = inv_sgd_m1_min_y1 + m1wd
inv_sgd_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_sgd_m1_max_x1:inv_sgd_m1_min_y1, inv_sgd_m1_max_x2:inv_sgd_m1_max_y2))
inv_sgd_m1_min_x3 = inv_sgd_m1_max_x2 - m1wd
inv_sgd_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(inv_sgd_m1_min_x3:gnd_height,
inv_sgd_m1_max_x2:inv_sgd_m1_min_y1))

;inverter pmos and nmos gate connections
;inv_gate_ln = max(pmos_invl, nmos_invl) - might not need this
inv_gate1_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x1:inv_nsel_max_y,
inv_poly_max_x1:inv_psel_min_y))
inv_gate2_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_nsel_max_y,
inv_poly_max_x2:inv_psel_min_y))

;inv input to output 1
inv_io1_poly_min_x = inv_poly_max_x1 - 2*pol_enc_con - con_size
inv_io1_poly_min_y = inv_act_max_y + 2*m1_m1sp + m1wd
inv_io1_poly_max_y = inv_io1_poly_min_y + 2*pol_enc_con + con_size
inv_io1_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_io1_poly_min_x:inv_io1_poly_min_y,
inv_poly_max_x1:inv_io1_poly_max_y))
inv_io1_m1_rect = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io1_poly_min_x:inv_io1_poly_min_y, inv_poly_max_x1:inv_io1_poly_max_y))
inv_io1_con_min_x = inv_poly_max_x1 - pol_enc_con - con_size
inv_io1_con_max_x = inv_poly_max_x1 - pol_enc_con
inv_io1_con_min_y = inv_io1_poly_min_y + pol_enc_con
inv_io1_con_max_y = inv_io1_con_min_y + con_size
inv_io1_con_rect = dbCreateRect(sram_cell_cv list(contact) list(inv_io1_con_min_x:inv_io1_con_min_y,
inv_io1_con_max_x:inv_io1_con_max_y))
inv_io1_m1_min_y1 = inv_io1_poly_max_y - m1wd
inv_io1_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1) list(inv_poly_max_x1:inv_io1_m1_min_y1,
pass_inv_ds2_m2_max_x2:inv_io1_poly_max_y))
inv_io1_m1_min_x2 = inv_pact_max_x - m1wd
inv_io1_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io1_m1_min_x2:inv_io1_poly_max_y, inv_pact_max_x:inv_pact_min_y))
inv_io1_m1_min_x3 = pass_inv_ds2_m2_max_x2 - m1wd
inv_io1_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1) list(inv_io1_m1_min_x3:inv_act_max_y,
pass_inv_ds2_m2_max_x2:inv_io1_m1_min_y1))

;inv input to output 2
inv_io2_poly_max_x = inv_poly_min_x2 + 2*pol_enc_con + con_size
inv_io2_poly_min_y = inv_act_max_y + m1_m1sp
inv_io2_poly_max_y = inv_io2_poly_min_y + 2*pol_enc_con + con_size
inv_io2_poly_rect = dbCreateRect(sram_cell_cv list(poly) list(inv_poly_min_x2:inv_io2_poly_min_y,
inv_io2_poly_max_x:inv_io2_poly_max_y))
inv_io2_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(inv_poly_min_x2:inv_io2_poly_min_y,
inv_io2_poly_max_x:inv_io2_poly_max_y))
inv_io2_con_min_x = inv_poly_min_x2 + pol_enc_con
inv_io2_con_max_x = inv_poly_min_x2 + pol_enc_con + con_size
inv_io2_con_min_y = inv_io2_poly_min_y + pol_enc_con
inv_io2_con_max_y = inv_io2_con_min_y + con_size
inv_io2_con_rect = dbCreateRect(sram_cell_cv list(contact) list(inv_io2_con_min_x:inv_io2_con_min_y,
inv_io2_con_max_x:inv_io2_con_max_y))
inv_io2_m1_min_x1 = inv_io1_poly_min_x - m1_m1sp - m1wd
```

```
inv_io2_m1_max_y1 = inv_io2_poly_min_y + m1wd
inv_io2_m1_rect1 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io2_m1_min_x1:inv_io2_poly_min_y, inv_poly_min_x2:inv_io2_m1_max_y1))
inv_io2_m1_max_x2 = inv_act_min_x + m1wd
inv_io2_m1_rect2 = dbCreateRect(sram_cell_cv list(metal1) list(inv_act_min_x:inv_act_max_y,
inv_io2_m1_max_x2:inv_io2_poly_min_y))
inv_io2_m1_max_x3 = inv_io2_m1_min_x1 + m1wd
inv_io2_m1_rect3 = dbCreateRect(sram_cell_cv list(metal1)
list(inv_io2_m1_min_x1:inv_io2_m1_max_y1, inv_io2_m1_max_x3:inv_pact_max_y))
if(inv_io2_m1_max_x3 < inv_pact_min_x
  then
            inv_io2_m1_rect4 = dbCreateRect(sram_cell_cv list(metal1)
list(invp_m1_min_x2:inv_pact_max_y, inv_pact_min_x:inv_pact_max_y))
)

;inv pmos source to vdd rail
inv_svdd_m1_rect = dbCreateRect(sram_cell_cv list(metal1) list(inv_m1_min_x2:inv_pact_max_y,
inv_m1_max_x2:vdd_m1_min_y))

;;;;;;PINS;;;;;;;;;;;;;;;;;;;
vdd_pin_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:cell_height - m1wd, m1wd:cell_height))
vdd_net = dbCreateNet(sram_cell_cv "vdd!")
vdd_term = dbCreateTerm(vdd_net "" "powerInput")
vdd_pin = dbCreatePin(vdd_net vdd_pin_rect "vdd!")

gnd_pin_rect = dbCreateRect(sram_cell_cv list(metal1) list(0:0, m1wd:m1wd))
gnd_net = dbCreateNet(sram_cell_cv "gnd!")
gnd_term = dbCreateTerm(gnd_net "" "powerInput")
gnd_pin = dbCreatePin(gnd_net gnd_pin_rect "gnd!")

BL_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_min_x:0, bl_max_x:m2wd))
BL_net = dbCreateNet(sram_cell_cv "BL")
BL_term = dbCreateTerm(BL_net "" "output")
BL_pin = dbCreatePin(BL_net BL_pin_rect "BL")

BL_bar_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(bl_bar_min_x:0, bl_bar_max_x:m2wd))
BL_bar_net = dbCreateNet(sram_cell_cv "BL_bar")
BL_bar_term = dbCreateTerm(BL_bar_net "" "output")
BL_bar_pin = dbCreatePin(BL_bar_net BL_bar_pin_rect "BL_bar")

WL_pin_rect = dbCreateRect(sram_cell_cv list(metal2) list(0:wl_via_min_y, m2wd:wl_via_min_y +
m2wd))
WL_net = dbCreateNet(sram_cell_cv "WL")
WL_term = dbCreateTerm(WL_net "" "input")
WL_pin = dbCreatePin(WL_net WL_pin_rect "WL")
)
```

DRC Rules File

*******FILE NOT INCLUDED DUE TO NDA PROTECTED CONTENT*********

Power Rings Layout

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;File: powerRings.il
;Functions: powerRings(powRing_cv tech_lambda rows cols cell_wd cell_ht gnd_ht prchrg_ht)
;Description: Generates power and ground rings, and power and ground rails for an SRAM array. This
;                       function is used by the sram_compiler() function.
;Input arguments:
;           powRing_cv: cell view layout to put power ring in
;           tech_lambda: lambda value for the chosen process technology
;           rows: number of SRAM rows
;           cols: number of SRAM columns
;           cell_wd: SRAM cell width
;           cell_ht: SRAM cell height
;           gnd_ht: height of ground rail. Used for laying out rails
;           prchrg_ht: height of precharge
;Usage example:
;           In command window type:
;                       load("powerRings.il")
;                       powerRings("SRAM" 0.3 16 8 11.1 22.2 2.4 6.9)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Author: Brandon Hilgers
;Date: 17 June 2015
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(powerRings(powRing_cv tech_lambda rows cols cell_wd cell_ht gnd_ht prchrg_ht)

ring_wd = 50*tech_lambda
ring_sp = 10*tech_lambda

;;;;;;;;LAYOUT RINGS;;;;;;;
;VDD RING
;bottom vdd
vdd_max_x = 4*ring_wd + 4*ring_sp + cols*cell_wd
bot_vdd = dbCreateRect(powRing_cv list(metal1) list(0:0, vdd_max_x:ring_wd))
;top vdd
vdd_max_y = 4*ring_wd + 4*ring_sp + rows*cell_ht + prchrg_ht
vdd_top_min_y = vdd_max_y - ring_wd
top_vdd = dbCreateRect(powRing_cv list(metal1) list(0:vdd_top_min_y, vdd_max_x:vdd_max_y))
;left vdd
left_vdd = dbCreateRect(powRing_cv list(metal2) list(0:0, ring_wd:vdd_max_y))
;right vdd
vdd_r_min_x = vdd_max_x - ring_wd
right_vdd = dbCreateRect(powRing_cv list(metal2) list(vdd_r_min_x:0, vdd_max_x:vdd_max_y))

;GND RING
;bottom gnd
gnd_min_x = ring_wd + ring_sp
gnd_min_y = gnd_min_x
gnd_max_x = 3*ring_wd + 3*ring_sp + cols*cell_wd
bot_gnd_max_y = gnd_min_y + ring_wd
bot_gnd = dbCreateRect(powRing_cv list(metal1) list(gnd_min_x:gnd_min_y,
gnd_max_x:bot_gnd_max_y))
;top vdd
gnd_max_y = vdd_max_y - ring_wd - ring_sp
gnd_top_min_y = gnd_max_y - ring_wd
```

```
top_vdd = dbCreateRect(powRing_cv list(metal1) list(gnd_min_x:gnd_top_min_y,
gnd_max_x:gnd_max_y))
;left vdd
gnd_l_max_x = gnd_min_x + ring_wd
left_vdd = dbCreateRect(powRing_cv list(metal2) list(gnd_min_x:gnd_min_y,
gnd_l_max_x:gnd_max_y))
;right vdd
gnd_r_min_x = gnd_max_x - ring_wd
right_vdd = dbCreateRect(powRing_cv list(metal2) list(gnd_r_min_x:gnd_min_y,
gnd_max_x:gnd_max_y))

;;;;;;;;;LAYOUT RING VIAS;;;;;;;;;;;;;;;;;;;;
via_num = truncate(ring_wd/(via_size + via_sp)) ;determine number of contacts that fit in width of ring
via_incr = via_size + via_sp ;distance to start drawing next via
;vdd ring via starting coordinates
l_vdd_via_min_x = 0.5*via_sp
r_vdd_via_min_x = vdd_r_min_x + 0.5*via_sp
b_vdd_via_min_y = 0.5*via_sp
t_vdd_via_min_y = vdd_top_min_y + 0.5*via_sp
;gnd ring via starting coordinates
l_gnd_via_min_x = gnd_min_x + 0.5*via_sp
r_gnd_via_min_x = gnd_r_min_x + 0.5*via_sp
b_gnd_via_min_y = gnd_min_y + 0.5*via_sp
t_gnd_via_min_y = gnd_top_min_y + 0.5*via_sp
for(c 0 via_num-1 ;incrementing contacts in x direction
        for(d 0 via_num-1 ;incremnting contacts in y direction
                b_l_vdd_ring_via = dbCreateRect(powRing_cv list(via) list(l_vdd_via_min_x +
c*via_incr:b_vdd_via_min_y + d*via_incr, l_vdd_via_min_x+via_size +
c*via_incr:b_vdd_via_min_y+via_size + d*via_incr))
                b_r_vdd_ring_via = dbCreateRect(powRing_cv list(via) list(r_vdd_via_min_x +
c*via_incr:b_vdd_via_min_y + d*via_incr, r_vdd_via_min_x+via_size +
c*via_incr:b_vdd_via_min_y+via_size + d*via_incr))
                t_l_vdd_ring_via = dbCreateRect(powRing_cv list(via) list(l_vdd_via_min_x +
c*via_incr:t_vdd_via_min_y + d*via_incr, l_vdd_via_min_x+via_size +
c*via_incr:t_vdd_via_min_y+via_size + d*via_incr))
                t_r_vdd_ring_via = dbCreateRect(powRing_cv list(via) list(r_vdd_via_min_x +
c*via_incr:t_vdd_via_min_y + d*via_incr, r_vdd_via_min_x+via_size +
c*via_incr:t_vdd_via_min_y+via_size + d*via_incr))
                b_l_gnd_ring_via = dbCreateRect(powRing_cv list(via) list(l_gnd_via_min_x +
c*via_incr:b_gnd_via_min_y + d*via_incr, l_gnd_via_min_x+via_size +
c*via_incr:b_gnd_via_min_y+via_size + d*via_incr))
                b_r_gnd_ring_via = dbCreateRect(powRing_cv list(via) list(r_gnd_via_min_x +
c*via_incr:b_gnd_via_min_y + d*via_incr, r_gnd_via_min_x+via_size +
c*via_incr:b_gnd_via_min_y+via_size + d*via_incr))
                t_l_gnd_ring_via = dbCreateRect(powRing_cv list(via) list(l_gnd_via_min_x +
c*via_incr:t_gnd_via_min_y + d*via_incr, l_gnd_via_min_x+via_size +
c*via_incr:t_gnd_via_min_y+via_size + d*via_incr))
                t_r_gnd_ring_via = dbCreateRect(powRing_cv list(via) list(r_gnd_via_min_x +
c*via_incr:t_gnd_via_min_y + d*via_incr, r_gnd_via_min_x+via_size +
c*via_incr:t_gnd_via_min_y+via_size + d*via_incr))
        )
)
;;;;;;;;;;;LAYOUT RAILS AND RAIL VIAS;;;;;;;;
array_min_y = 2*ring_wd + 2*ring_sp
for(n 0 row-1
        if(modulo(n 2) == 0
        then
                gnd_rail_min_y = array_min_y + cell_ht*n
                vdd_rail_min_y = array_min_y + cell_ht*(n+1) - gnd_ht
                gnd_rail_via_min_y = gnd_rail_min_y + 0.5*via_sp
                vdd_rail_via_min_y = vdd_rail_min_y + gnd_ht - 0.5*via_sp - via_size
        else
                gnd_rail_min_y = array_min_y + cell_ht*(n+1) - gnd_ht
                vdd_rail_min_y = array_min_y + cell_ht*n
                gnd_rail_via_min_y = gnd_rail_min_y + gnd_ht - 0.5*via_sp - via_size
                vdd_rail_via_min_y = vdd_rail_min_y + 0.5*via_sp
        )
        gnd_rail_max_y = gnd_rail_min_y + gnd_ht
        gnd_rail = dbCreateRect(powRing_cv list(metal1) list(gnd_min_x:gnd_rail_min_y,
gnd_max_x:gnd_rail_max_y))
        vdd_rail_max_y = vdd_rail_min_y + gnd_ht
        vdd_rail = dbCreateRect(powRing_cv list(metal1) list(0:vdd_rail_min_y,
vdd_max_x:vdd_rail_max_y))
        ;layout vias
        for(c 0 via_num-1 ;incrementing contacts in x direction
                l_gnd_rail_via = dbCreateRect(powRing_cv list(via) list(l_gnd_via_min_x +
c*via_incr:gnd_rail_via_min_y, l_gnd_via_min_x+via_size + c*via_incr:gnd_rail_via_min_y+via_size))
                r_gnd_rail_via = dbCreateRect(powRing_cv list(via) list(r_gnd_via_min_x +
c*via_incr:gnd_rail_via_min_y, r_gnd_via_min_x+via_size + c*via_incr:gnd_rail_via_min_y+via_size))
                l_vdd_rail_via = dbCreateRect(powRing_cv list(via) list(l_vdd_via_min_x +
c*via_incr:vdd_rail_via_min_y, l_vdd_via_min_x+via_size + c*via_incr:vdd_rail_via_min_y+via_size))
                r_vdd_rail_via = dbCreateRect(powRing_cv list(via) list(r_vdd_via_min_x +
c*via_incr:vdd_rail_via_min_y, r_vdd_via_min_x+via_size + c*via_incr:vdd_rail_via_min_y+via_size))
        )
)

;precharge vdd rail
prchrg_vdd_rail_min_y = array_min_y + row*cell_ht + prchrg_ht - gnd_ht
prchrg_vdd_rail_max_y = prchrg_vdd_rail_min_y + gnd_ht
prchrg_vdd_rail = dbCreateRect(powRing_cv list(metal1) list(0:prchrg_vdd_rail_min_y,
vdd_max_x:prchrg_vdd_rail_max_y))
;precharge vdd rail vias
prchrg_vdd_rail_via_min_y = prchrg_vdd_rail_min_y + 0.5*via_sp
for(c 0 via_num-1 ;incrementing contacts in x direction
        l_prch_vdd_rail_via = dbCreateRect(powRing_cv list(via) list(l_vdd_via_min_x +
c*via_incr:prchrg_vdd_rail_via_min_y, l_vdd_via_min_x+via_size +
c*via_incr:prchrg_vdd_rail_via_min_y+via_size))
        r_prch_vdd_rail_via = dbCreateRect(powRing_cv list(via) list(r_vdd_via_min_x +
c*via_incr:prchrg_vdd_rail_via_min_y, r_vdd_via_min_x+via_size +
c*via_incr:prchrg_vdd_rail_via_min_y+via_size))
)
)
```

Generate Pins in Layout

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;File: generatePin.il
;Functions: generatePin(pin_cv pinName layer termType pin_min_x pin_min_y pin_max_x pin_max_y)
;Description: Generates a pin of the desired type (input/output) and layer, and in the desired location.
```

```
;Input arguments:
;	pin_cv: cell view layout to put pin in
;	pinName: name of the pin
;	layer: layer to draw the pin in
;	termType: terminal type (input,output,etc.)
;	pin_min_x: pin minimum x-coordinate
;	pin_min_y: pin minimum y-coordinate
;	pin_max_x: pin maximum x-coordinate
;	pin_max_y: pin maximum y-coordinate
;Usage example:
;	In command window type:
;		load("generatePin.il")
;		generatePin(sram_cv "Pin1" "metal1" "input" 0.25 0.3 0.5 0.55)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Author: Brandon Hilgers
;Date: 17 June 2015
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
procedure(generatePin(pin_cv pinName layer termType pin_min_x pin_min_y pin_max_x pin_max_y)

pin_rect = dbCreateRect(pin_cv list(layer) list(pin_min_x:pin_min_y, pin_max_x:pin_max_y))
net = dbCreateNet(pin_cv pinName)
term = dbCreateTerm(net "" termType)
pin = dbCreatePin(net pin_rect pinName)
)
```

Appendix B: Address Decoder Verilog

```
//4:16 Address Decoder
module AddrDec4_16 (
binary_in   , //  4 bit binary input
decoder_out , //  16-bit  out
enable        //  Enable for the decoder
);
//Specify inputs and outputs
input [3:0] binary_in  ;
input  enable ;
output [15:0] decoder_out ;

reg [15:0] decoder_out ;

//Define 4:16 decoder behavior
always @ (enable or binary_in)
begin
  decoder_out = 0;
  if (enable) begin
    case (binary_in)
      4'h0 : decoder_out = 16'h0001;
      4'h1 : decoder_out = 16'h0002;
      4'h2 : decoder_out = 16'h0004;
      4'h3 : decoder_out = 16'h0008;
      4'h4 : decoder_out = 16'h0010;
      4'h5 : decoder_out = 16'h0020;
      4'h6 : decoder_out = 16'h0040;
      4'h7 : decoder_out = 16'h0080;
      4'h8 : decoder_out = 16'h0100;
      4'h9 : decoder_out = 16'h0200;
      4'hA : decoder_out = 16'h0400;
      4'hB : decoder_out = 16'h0800;
      4'hC : decoder_out = 16'h1000;
      4'hD : decoder_out = 16'h2000;
      4'hE : decoder_out = 16'h4000;
      4'hF : decoder_out = 16'h8000;
    endcase
  end
end

endmodule
```