Development of a Low-Power SRAM Compiler

by

Meenatchi Jagasivamani

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. Dong S. Ha, Chairman
Dr. James R. Armstrong
Dr. Joseph G. Tront

September 1, 2000
Blacksburg, Virginia

Abstract

Considerable attention has been paid to the design of low-power, high-performance SRAMs (Static Random Access Memories) since they are a critical component in both hand-held devices and high-performance processors. A key in improving the performance of the system is to use an optimum sized SRAM.

In this thesis, an SRAM compiler has been developed for the automatic layout of memory elements in the ASIC environment. The compiler generates an SRAM layout based on a given SRAM size, input by the user, with the option of choosing between fast vs. low-power SRAM. Array partitioning is used to partition the SRAM into blocks in order to reduce the total power consumption.

Experimental results show that the low-power SRAM is capable of functioning at a minimum operating voltage of 2.1 V and dissipates 17.4 mW of average power at 20 MHz. In this report, we discuss the implementation of the SRAM compiler from the basic component to the top-level SKILL code functions, as well as simulation results and discussion.

Acknowledgements

I would like to thank my committee chairman and advisor, Dr. Dong S. Ha for giving me the opportunity to work with him. His guidance and encouragement was instrumental in getting this work accomplished. I would also like to express my appreciation to Dr. James R. Armstrong and Dr. Joseph G. Tront for participating in my examiniation committee and commenting on this work.

I would like to thank all my colleagues at the VTVT (Virginia Tech VLSI for Telecommunications) lab for their support and guidance during my graduate years: Jos Sulistyo, Carrie Aust and others. I have been lucky to have had the chance to work with Jia Fei both academically and personally. Her cheerfulness always made a positive impact on all of us and her friendship has been invaluable to me during the many long nights at the lab.

Lastly, I am indebted to my family for providing me with unconditional support during difficult times at graduate school. Without their love and faith in me, I could not have found the strength and confidence to undertake this project.

Table of Contents

List of Figures and Tables

CHAPTER

1 Introduction

With the increasing use of portable consumer electronics, power consumption has become an important performance characteristic for a chip due to both limited battery life in portable systems and also due to expensive packages and heat sinks required by high power levels. Consequently, the design of low-power digital systems is becoming increasingly important. With memories typically accounting for the largest share of power consumption in a system, an emphasis has been placed on the design of low-power memories.

More than half of the transistors in today's high performance microprocessors are devoted to cache memories and this ratio is expected to increase in the foreseeable future. Typically, SRAM (Static Random Access Memory) is the choice for embedded memories as SRAM is robust to the noisy environment in such chips. As a result, considerable attention has been paid to the design of low-power, high-performance SRAMs since they are a critical component in both hand-held devices and high-performance processors.

A key in improving the performance of the system is to use an optimum sized SRAM. By incorporating an SRAM that is the correct size for the system requirements, the system can avoid using unnecessary memory cells. This leads to improvements in area, speed, and power. Therefore, depending on the application's need, an appropriate SRAM size should be used.

In this thesis, an SRAM compiler has been developed for the automatic layout of memory elements in the ASIC environment. The compiler will generate an SRAM layout based on a given SRAM size, input by the user. Also, the compiler allows the user to choose between fast vs. low-power SRAM. The SRAM memory array is partitioned into blocks in order to reduce the total power consumption. The Cadence design environment is used for this thesis. Cadence SKILL language is used to implement the compiler and Cadence Virtuoso is used for the layout-editor tool.

The organization of the thesis is as follows. In Chapter 2, the background related to the thesis and the proposed research is described. Previous work on low-power techniques for SRAM is also reviewed in this chapter. In Chapter 3, the design and

layout of the leaf-cell components are presented. In Chapter 4, the implementation of the SRAM compiler that generates an SRAM without array-partitioning is described, along with the final SRAM layout. Chapter 5 discusses the array-partitioning technique implemented for the low-power SRAM, as well as the implementation and the layout for this SRAM. In Chapter 6, experimental results for the two different types of SRAM are reported. Finally Chapter 7 concludes this thesis and presents future enhancements for the SRAM compiler. The SKILL code, along with documentation, is attached in the Appendix.

CHAPTER

2 Background

Memory elements form critical components in the implementation of CMOS circuits and are vital for most systems. They are used for a wide variety of applications with different design criterion. Though all memory elements are used to store and access data, they can be broken into three types based on how the stored information is retrieved. These three types are random access memory, serial access memory, and content access memory. Random access memory is defined as memory that has an access time independent of the physical location of the data. This can be contrasted with serial access memory where the data is retrieved sequentially with time, or content access memory, where data is retrieved based on the type of data stored. Figure 2.1 illustrates the classifications of memory elements.

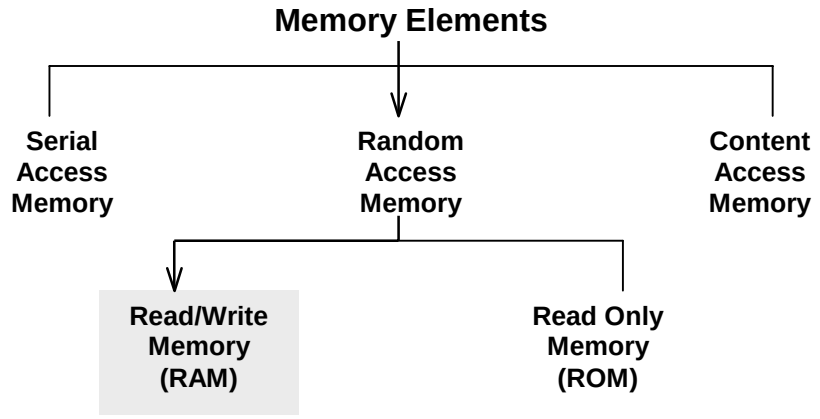


Figure 2.1 – Classification of Memory Elements

RAM can be classified into Read/Write Memory and Read Only Memory. Read Only Memory (ROM) is nonvolatile memory, where the stored data is maintained indefinitely, even without power, and writing to the memory takes considerably more time (on the order of milliseconds) than reading. Read/write memory (commonly called RAM) is data that is stored temporarily and the read and write time are approximately equal to each other.

RAM cells can be further divided into static and dynamic memory cells. Static memory (SRAM) cells use a latch composed of cross-coupled inverters to store data.

This allows the value to be maintained in a cell as long as power is available. Data storage in dynamic memory cell (DRAM) is based on the dynamic storage of charge on a capacitor. Therefore, with dynamic memory cells, periodic refreshing is necessary to maintain the value. Transistor-level schematic of a SRAM and a DRAM cell can be found in Figure 2.2. Bit-lines form the datapath to/from the cell, while word-lines select a cell to be accessed.

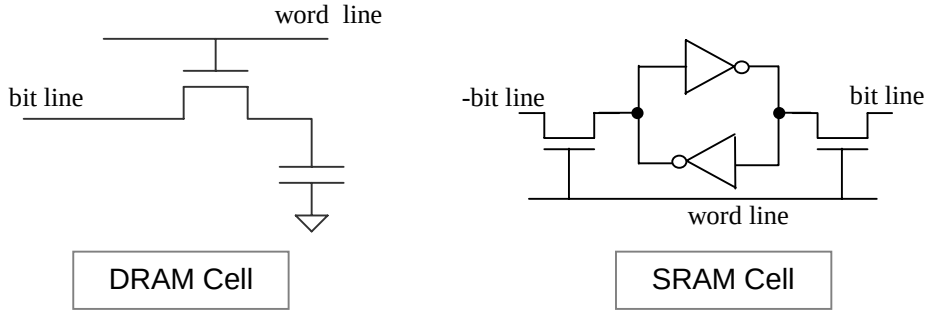


Figure 2.2 – Comparison of Static and Dynamic RAM Cells

There are many reasons to use an SRAM or a DRAM in a system design. Design tradeoffs include density, speed, volatility, cost, and features. Dynamic memory cells are smaller (since they use just a capacitor), but are slower than static memory cells. In addition, DRAMs require special processing in CMOS technology. Generally, DRAMs are custom designed for the application since there are many trade-offs to be considered with this type. The primary advantage of an SRAM over a DRAM is its speed and no need for special CMOS processing, which are compatible with random logic processing. For this project, since RAMs are to be embedded in a system, SRAMs are implemented. Also, for simplicity, an asynchronous approach is taken. In the next section, we will look at the components of the RAM architecture used for this project.

2.1 RAM Architecture

The basic architecture of a SRAM consists of an array of memory cells with support circuitry to decode addresses and implement the read and write operations. SRAM arrays are arranged in rows and columns of memory cells called wordlines and bitlines, respectively. Typically, the wordlines are made from polysilicon while the bitlines are metal. Each memory cell has a unique location or address defined by the

intersection of a row and a column. Figure 2.3 shows the generic RAM circuit for a memory chip that has just one row and one column.

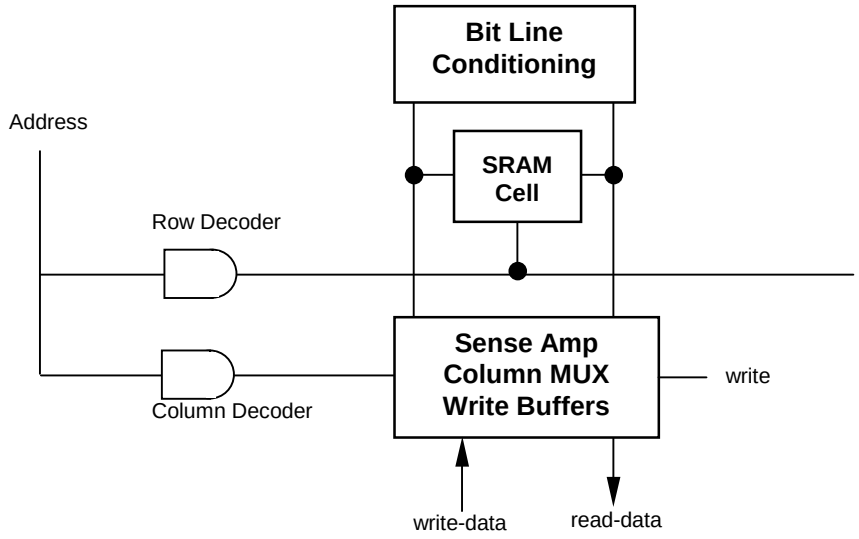


Figure 2.3 – Block Diagram of an Asynchronous SRAM Circuit

The RAM architecture consists of the following structures:

- **RAM Cell** – used to store one data bit
- **Bit Line Conditioning** – precharges bit lines to compensate for voltage drop across pass transistors
- **Column Multiplexer –** switches between read and write operation
- **Write Buffers** – buffers write-data so that it can write on RAM cells
- **Sense Amplifier** – Generate logic values from the differential input on bit-lines
- **Row & Column Decoders** – Decodes address to the correct RAM cell

In the next section, we will discuss the structure and design issues regarding these components.

2.2 RAM Cell Components

The schematic for static RAM Cell is shown in Figure 2.4. Essentially, the data is latched at the cross-coupled inverters. The bit-lines are complementary and are input to the I/O of the inverters. Thus, the value is latched during a write and maintained as long as power is available.

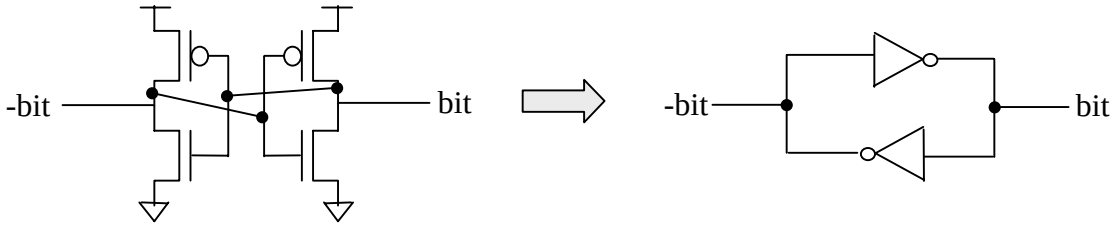


Figure 2.4 – Latched Storage for a Static RAM Cell

When the control signal "word" in Figure 2.5 is on, the RAM cell is connected to the two bit-lines. During a read operation, the two bit-lines are driven by the cell value. In contrast, the two bit-lines drive or override the cell during the write operation. Column and row decoders select a specific RAM cell by asserting proper control lines.

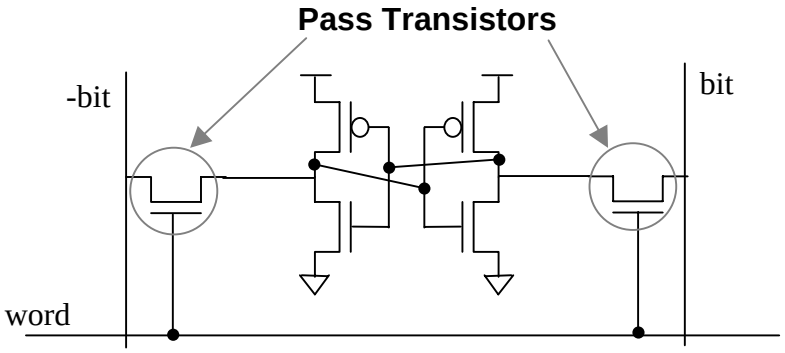


Figure 2.5 – Static RAM Cell with Select Circuit

When the word is asserted during a read operation, the bit values are available to the latch through n-type transistors. Since n-type transistors only pass a good value of '0', but not '1', it is appropriate to precharge both the bit lines to a high value and let the RAM cell pull down one of the bit lines.

Figure 2.6 shows the RAM cell with the bit-line conditioning circuit that charges the bit lines using n-type transistors. Both bit lines are charged to VDD-V_{tn}, where V_{tn} is the threshold voltage of the precharging NFET. When the word is asserted, one of the bit lines is pulled down to a '0', while the other one remains at '1'. It is also possible to use p-type transistors for the precharge transistors, and this would pull up the bit lines to VDD instead of to VDD-V_{tn}. However, it will take longer to pull down the bit lines. Thus, using n-transistors improves the speed of the RAM. Also from Figure 2.6, it can be seen that gates of the precharge transistors are tied to VDD, and hence the transistors are always turned on. This avoids generating another signal, but it requires the precharge

transistors to be weak so that they do not overcome the value driven onto the bit-lines during a read/write operation.

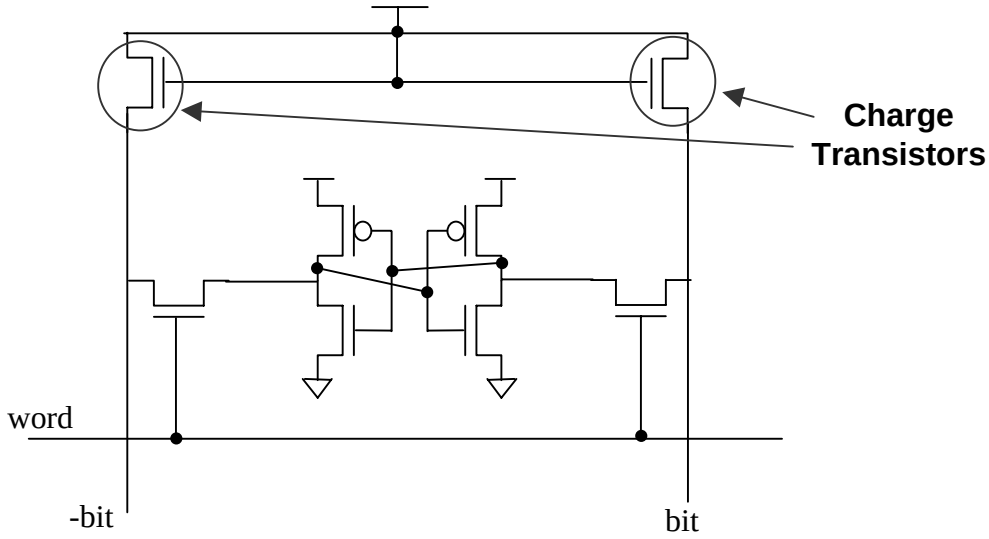


Figure 2.6 – Static pull-up RAM Cell with Bit-Line Conditioning

During the read mode, a sense-amplifier is usually used to amplify the bit-line voltage difference of the two bit-lines. The cross-coupled sense amplifier shown in Figure 2.7 was used to amplify the bit-line difference in our research. The sense amplifier is composed of a cross-coupled pair of PFETs (M_9 and M_{10}). The differential output is present at nodes sense+ and sense-.

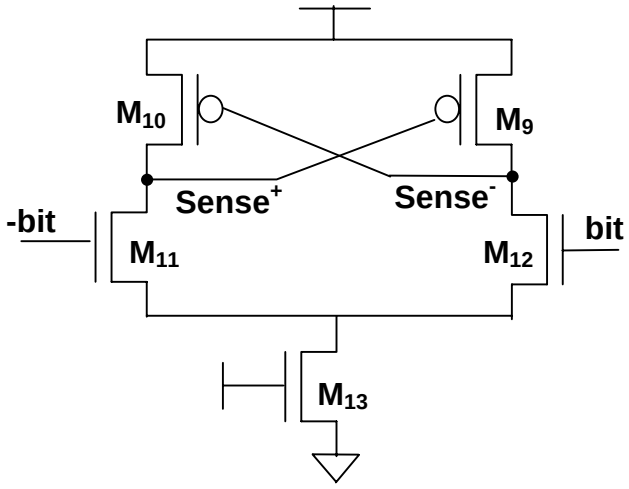


Figure 2.7 – Cross Coupled Sense Amplifier

All of the above components are the basic cells used to form an SRAM chip. The basic architecture of a SRAM includes an array of memory cells with support circuitry to decode addresses and to implement the required read and write operations.

Figure 2.8 shows a basic block diagram of an asynchronous SRAM. To perform a read/write operation, the first step is to specify the address that is being accessed. Next, the chip enable signal, CEN, and the read/write enable signals (OEN/WEN), must be enabled. When the REN control signal is enabled (read operation), the value stored at the specified cell appears at the data output port. When WEN is enabled (write operation), the value present at the "Data Inputs" is written into the specified location.

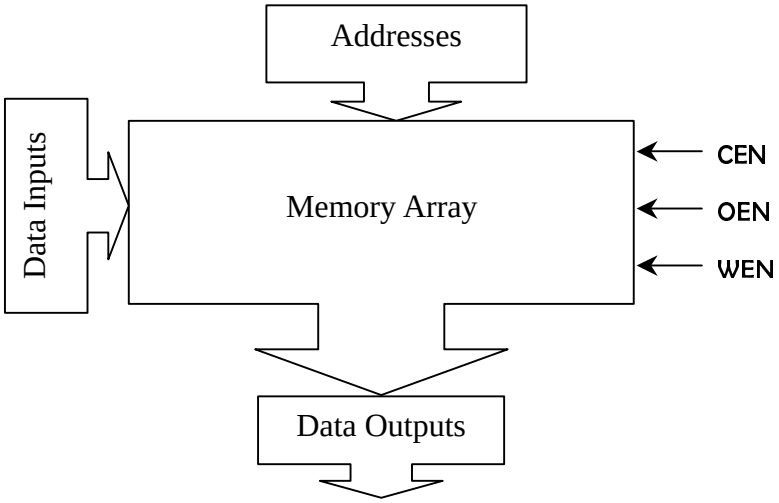


Figure 2.8 – Block Diagram of an Asynchronous SRAM

2.3 Review of Low-Power RAMs

Trends show that low power design techniques are becoming more important in the current industry. Considerable attention has been paid to the design of low-power for applications such as hand-held devices and wireless communications. There are numerous ways to reduce the power dissipation at the cost of area and/or speed, both in the cell and architectural level. In this section, we will previous works that discuss low-power SRAM techniques on the circuit and architectural level.

2.3.1 Divided and Hierarchical Bit-lines

In an SRAM, a pair of bit-lines is connected to a column of RAM cells. For large circuits, the length of the bit-lines can be considerably long, resulting in large bit-line

capacitances. The charging or discharging of bit-line capacitance causes active power dissipation, which is a major source of power dissipation. A. Karandikar and K. Parhi proposed a divided bit-line approach for reducing the active power dissipation by reducing the bit-line capacitance [1].

Active current is the current that flows when bit-lines are charging or discharging. The active current is directly proportional to the bit-line capacitance. The proposed divided bit-line approach intends to reduce bit-line capacitance, which is mainly composed of the drain capacitance of the pass transistors of the SRAM cell and the metal capacitance of bit-line.

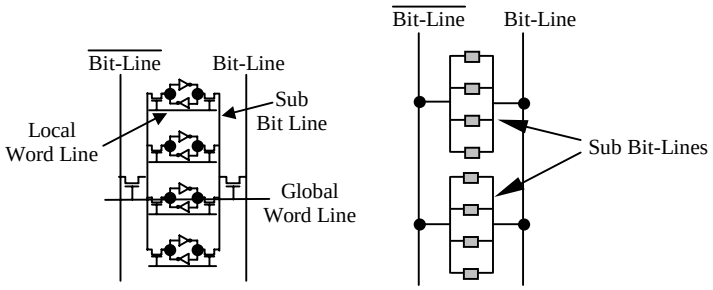


Figure 2.9 – Divided Bit-line Architecture

Figure 2.7 illustrates the concept of divided bit-line method. The bit-lines are split into sub-bit lines so that only a few bit cells share the local bit line (sub bit-line). Thus, the global bit-lines are connected to fewer pass transistors and the $C_{bitline}$ is significantly reduced. This technique can be extended to divide the bit-lines in a hierarchy for large circuits. Reducing $C_{bitline}$ not only reduces the active power, but the access time as well. The main disadvantage with this technique is the increased complexity in the basic SRAM architecture. This complexity results in a significant area overhead, as well the need for additional control signals for the global and local word-lines. Experimental results show that for a 2-kB SRAM, the power consumption is reduced by 50-60% and access time is reduced by 20-30%, with a 5% increase in the number of transistors.

2.3.2 Half-Swing Pulse-Mode

Most of the currently present techniques aim to reduce the power needed to read data from the memory. K. Mai, T. Mori, B. Amrutur, R. Ho, B. Wilburn, M. Horowitz, I.

Fukushi, T. Izawa, and Shin Mitarai aim to reduce power dissipation for write operation and for decoders using half-swing techniques [24]. In low-power embedded SRAMs with large access widths, the write-operation power can be significantly larger than the read-operation power. Since the bit lines are referenced to V_{dd}, they are discharged to GND during a write operation. Thus, decreasing the bit-line swings during writes can reduce write power.

The main problem with reduced swing signals in the past has been the need for level-conversion and/or reduced gate overdrive at the receiving gates, which causes a loss of performance. They aim to address the problem by combining positive half-swing (swinging the bit-lines from the steady state of $V_{dd}/2$ to V_{dd} and back to $V_{dd}/2$) and negative half-swing (swinging from the steady state of $V_{dd}/2$ to Gnd). Thus, all of the forward-transition driving transistors see a full gate overdrive. For example, Figure 2.8 represents a half-swing pulse-mode AND gate that uses half-swing inputs to produce a full-swing output voltage.

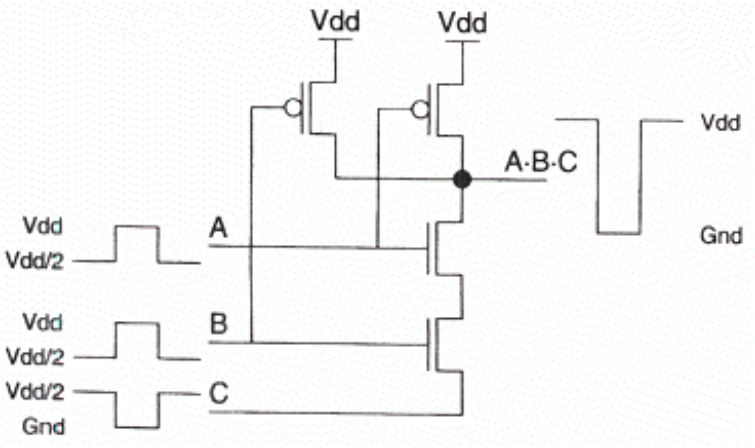


Figure 2.10 – Half-Swing Pulse-Mode AND Gate[24]

This technique requires redesign of all support circuitry so that the half-swing bit-lines can be appropriately interpreted and converted to full-swing outputs. The main disadvantage with this technique is the reduced noise margin on the bit-lines, which results in higher susceptibility to noise. Also, it requires an additional supply voltage of $V_{dd}/2$ and the routing of the rail is cumbersome. Experimental results performed on a 2-K x 16-b SRAM fabricated in a 0.25 μm dual-V_t CMOS technology show that the prototype dissipates 0.9 mW at 100 MHz using an operating voltage of 1V.

2.3.3 Sub-Blocked Array Architecture

J. Caravella proposed to reduce power dissipation by reducing both the bit-line and word-line capacitance [7]. The power dissipation for static CMOS logic circuits is given by $P = C \times V_{dd}^2 \times f$, where C is the average switched load and parasitic capacitances, V_{dd} is the supply voltage, and f is the operating frequency of the circuit. Because the power consumption increases quadratically with the supply voltage, most dramatic reduction in power can be achieved by reducing V_{dd}. However, without redesigning the circuit, reducing the supply voltage may not only slow down the circuit, but may cause the circuit to fail.

The discharge rate of the bit-lines contributes to the read access time of the SRAM, which is proportional to a time constant given by the following equation[7].

$$\tau \approx \frac{C_{bitline}}{K'\left(\frac{W}{L}\right)(V_{dd} - V_t)^2} \cdot \Delta V$$

where ΔV is the discharge voltage amount, $C_{bitline}$ is the total bit line capacitance, K' is the intrinsic transconductance of the word line pass transistor, W/L is the width to length ratio of the transistor, V_{dd} is the supply voltage, and V_t is the threshold voltage of the transistor. Therefore, if V_{dd} is reduced, then the time constant will increase, making the circuit slower. One way to maintain the time constant is to reduce the capacitance that the bit cell needs to discharge. This can be achieved by reducing the number of RAM cells sharing a given bit line.

This paper proposes to reduce the bit-line capacitance by dividing the memory array into four isolated subarrays, which would reduce both the total bit line and word line capacitance by half. The bit line capacitance is the parasitic capacitance (junction and metal) associated with the RAM cell load on the bit lines, while the word line capacitance is the parasitic capacitance (gate, fringe, and metal) associated with the RAM cell on the word lines.

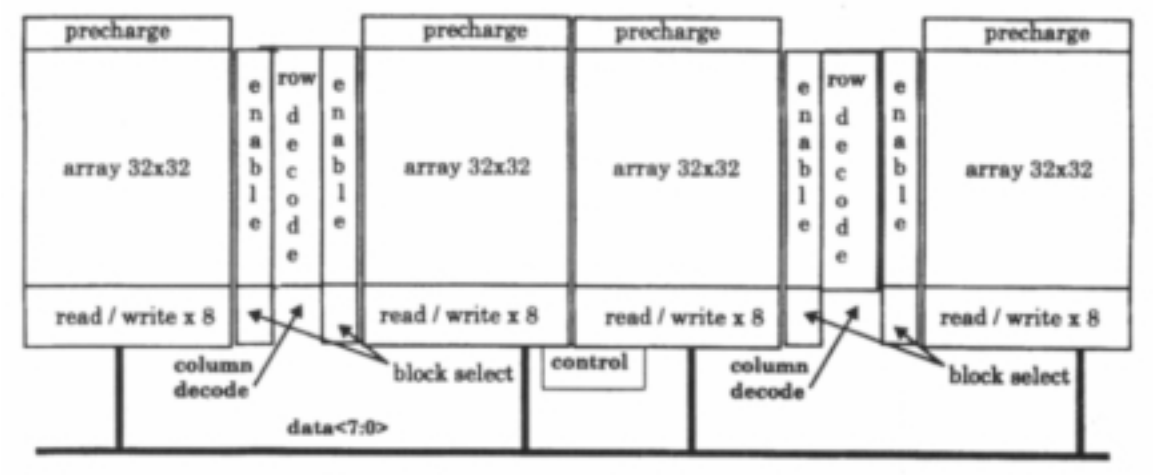


Figure 2.11 – Sub-Blocked Array Architecture

The structure used by J. Caravella for a 64 kB SRAM is shown in Figure 2.9. Dividing the array into blocks not only reduces the power dissipation, but the subarray architecture results in a faster SRAM due to the reduced capacitance. The only disadvantage with this method is the area penalty due to increased overhead of decoder logic, control logic, and routing. Experimental results for the 64 kB SRAM showed that with an area overhead of 15%, the RAM was able to operate at 50 MHz with V_{dd}=1.8 V.

Since it is relatively easy to extend a normal SRAM array to include array partitioning, this method is adopted for our RAM design. Details about the architecture are explained in section 2.5.

2.4 Introduction to SKILL

The objective of our SRAM compiler studied in this thesis is to generate a SRAM layout for a given size. The SRAM compiler must be able to instantiate the leaf cells and to layout necessary routing & connections for the circuit. The language that will be used to perform the layout automation is Cadence's SKILL. SKILL, which stands for Silicon Compiler Interface Language, has tool specific functions for several of Cadence Suites – Virtuoso (Layout Editor) and Composer (Schematic Editor), among others. These functions allow the user to use any tool-specific command, such as drawing a rectangle in a layout. Figure 2.10 gives an example of the *dbCreateRect(…)* command, used to draw a rectangle in given cellview. As shown in the figure, the user can specify exact coordinates of the rectangle as well as the layer.

dbcreateRect(compilercellview "poly1" list(x_1:y_1 x_2:y_2))

Figure 2.12 – SKILL Function to Draw Rectangle

SKILL is an interpretive script language, which means that commands are executed as they are entered. Commands are entered into the Cadence environment via the CIW (Common Interface Window). For this thesis project, we use the SKILL language to accomplish all design automation, including aspect ratio calculation, leaf cell instantiation, and routing. More details about the use of SKILL in the implementation of the SRAM compiler are discussed in Chapter 4. Table 4.1 lists some commonly used SKILL functions

Table 2.1 – Common SKILL Functions

procedure*(function_name(argument_list)* *expr1* *expr2* … *)*	Defines a function using an argument list. The body of the procedure is a list of expressions to evaluate.
for*(loopVariable initialValue finalValue* *expr1* *expr2* … *)*	Evaluates the sequence *expr1, expr2, …* for each *loopVariable* value, beginning with *intialValue* and ending with *finalValue*.
if*(condition* then *expr1* else expr2 … *)*	Evaluates *condition* and runs *expr1* if the *condition* is true. Otherwise, runs *expr2.*
x = ‘(1 2 3) or x = list(1 2 3)	Creates a list variable called x that containing the three elements.
Note: There should be no space before a ‘(‘.	

In addition to the above functions, SKILL also has functions that are specific to the Layout Editor tool (Virtuoso). These functions are used to perform the actual layout of the SRAM compiler and are given in Table 2.2.

Table 2.2 – Tool-Specific Library SKILL Functions

dbOpenCellViewByType(library cellname viewname viewtype accessmode) Opens a cellview. Returns a db (database) object for the cellview.
dbCreateInst(dbcellview dbmaster InstName lpoint orientation) Places an instance of *dbmaster* onto the cellview *dbcellview*. The instance will be placed at *lpoint* with the *orientation*. Returns a db object for the instance.
dbFlattenInst(dbInst x_levels [flatten_pcells] [preservePins]) Flattens instance dbInst up through x_levels of hierarchy. Returns t/nil.
dbCreateRect(dbcellview layer list_box) Draws a rectangle onto *dbcellview* of *layer* with the coordinates given by *list_box*. Returns a db object for the rectangle.
dbCreateNet(dbcellview t_name) Create a net for a pin to attach to in *dbcellview*. The name of the pin should be *t*_name. Returns a db object for the net.
dbCreatePin(net fig t_name) Creates a pin attached to *net* for the object defined by *fig* of *t_name*. Returns a db object for the pin.
dbSave(dbcellview) Saves the results of a modified *dbcellview* that has been opened for write or append mode.

Note that all the functions are database (db) functions. All Cadence tools use the Design Framework II unified database; a binary database that stores data as "objects." There are many types of objects, including rectangles, pins, instances, and cellviews. The SKILL code structure used to implement the SRAM compiler will be discussed in Chapter 4.

2.5 Proposed Research

The SRAM compiler studied in this thesis will be used by the VTVT (Virginia Tech VLSI for Telecommunications) group for their Wireless Video Project. The project consists of transmitting wireless video using a cellular phone. One of the major

components of this project is the development of a turbo decoder, which will require SRAMs of varying sizes to store data. The maximum SRAM size that will be used by the turbo decoder is 1 kB (1024x8), with a maximum operating frequency of 20 MHz. An SRAM compiler is needed, since the turbo decoder uses various sizes of SRAMs. Furthermore, the SRAM compiler will be used for other current and future projects.

The input/output ports for a 2^k x w SRAM considered in this research is given in Figure 2.11, where 2^k is the number of word locations and w is the word size (depth). There are three active-low control signals – CEN', WEN', OEN'. CEN' is the control signal to enable the chip. When CEN' is disabled (high), all of the word-lines (active-high) are turned off (pulled low), so that no RAM cell is connected to any bit-line. When CEN' is enabled (low), the word line that is being accessed is turned on (pulled high) so that all RAM cells in the row are connected to the bit-lines. The column decode circuitry chooses the column block that is being accessed. The other two control signals, WEN' and OEN', are the active low Read/Write signals. Whenever WEN' (Write Enable') goes low, the SRAM is being written to. Similarly, whenever OEN' (Output Enable') goes low, the SRAM is being read from.

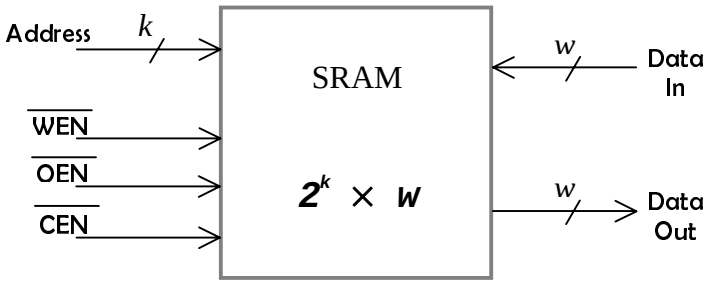


Figure 2.13 – Block Diagram of SRAM

There are k address signals for the 2^k word locations. The address signals are split between the row and column decoders. The number of rows and columns in the SRAM are chosen to make the aspect ratio close to 1 and are computed by the compiler program. Since this is an embedded SRAM (with no constraints on the number of pins), there are separate data signals for the input and output. Because of this, it is possible to read from a location while writing to that location.

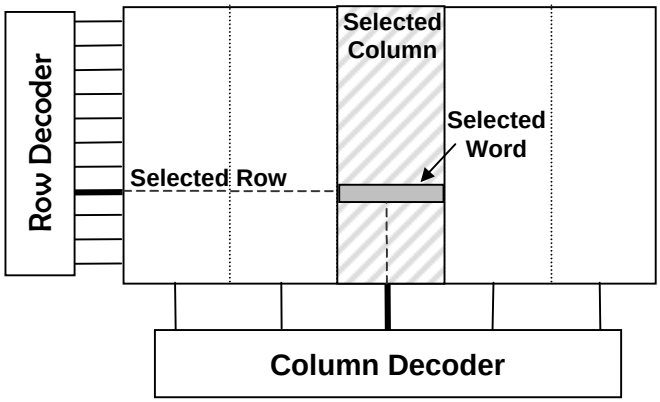


Figure 2.14 – Structural Decoder layout

Structurally, the SRAM is arranged into rows of bits and columns of blocks, as shown in Figure 2.12. The reason for this type of arrangement is to simplify column decoding for word size greater than one. Thus, the row decoder decodes for a single word line, while the column decoder decodes for a block of bit-lines. As an example, suppose that we are reading a word of 8 bits from location *(i, j)*. Then the row decoder activates word-line$_i$ and the column decoder connects all bit-lines in block$_j$ to the sense amplifiers, where block *j* consists of 8 columns. Note that the number of columns is a multiple of the word-size. The number of rows and the number of columns for our SRAMs are determined based on the word size and the aspect ratio. An overview of the development process of the SRAM compiler is as follows:

1. Design and custom layout all leaf-cells for the SRAM. *(Chapter 3)*
2. Develop SKILL code to perform design automation of all components including RAM core, decoders, and I/O buffers. *(Chapter 4)*
3. Add array partitioning to improve power dissipation. *(Chapter 5)*
4. Simulation and Verification. *(Chapter 6)*

The following chapters discuss implementation of each of the above steps.

CHAPTER

3 Leaf Cell Layout

The main responsibility of a SRAM generator is to instantiate basic components in an array, for the given size. The basic components, called *leaf cells*, are critical in determining the final performance of the generated SRAM circuit. Therefore, leaf cell design must be optimized both locally and globally for area, power, and speed. Whenever possible, the leaf cell layout must use *cell abutting.* This technique helps reduce unnecessary routing by simply placing adjacent cells close to each other.

In this chapter, we will discuss the design and layout of basic components in the RAM architecture. The basic components (as discussed in Chapter 2) are as follows:

- 6-transistor core of SRAM
- Bit-line conditioning circuit
- Sense amplifier
- Address decoder

Before we discuss each component in detail, we review the overall SRAM structure. Figure 3.1 shows a block diagram of an SRAM and connections of basic components. A RAM cell is connected to two bit lines through word-select pass transistors. Since the pass-transistors used are NFET, they are slow when pulling a line up to logic '1'. Charging the bit-lines to a high value reduces the time it takes the pass-transistors to pull-up. Though charging the bit-lines causes a speed-up in access time, it degrades the bit-line signal difference. Therefore, a sense-amplifier is needed to increase the difference and provide a good data output during a read.

Three decoders are activated or deactivated by three active-low control signals – CEN', WEN', and OEN'. The CEN' is used to indicate that the SRAM is currently being accessed and controls the word-line (row) decoders. Thus when the CEN' is off (high), none of the word-lines are on. Likewise, the WEN' signal, which specifies that the SRAM is being written to, controls the write-line decoder. Similarly, the OEN' signal indicates that the SRAM being read from and controls the read-line decoder.

Precharge circuits and sense amplifiers are the other two major components, as shown in Figure 3.1.

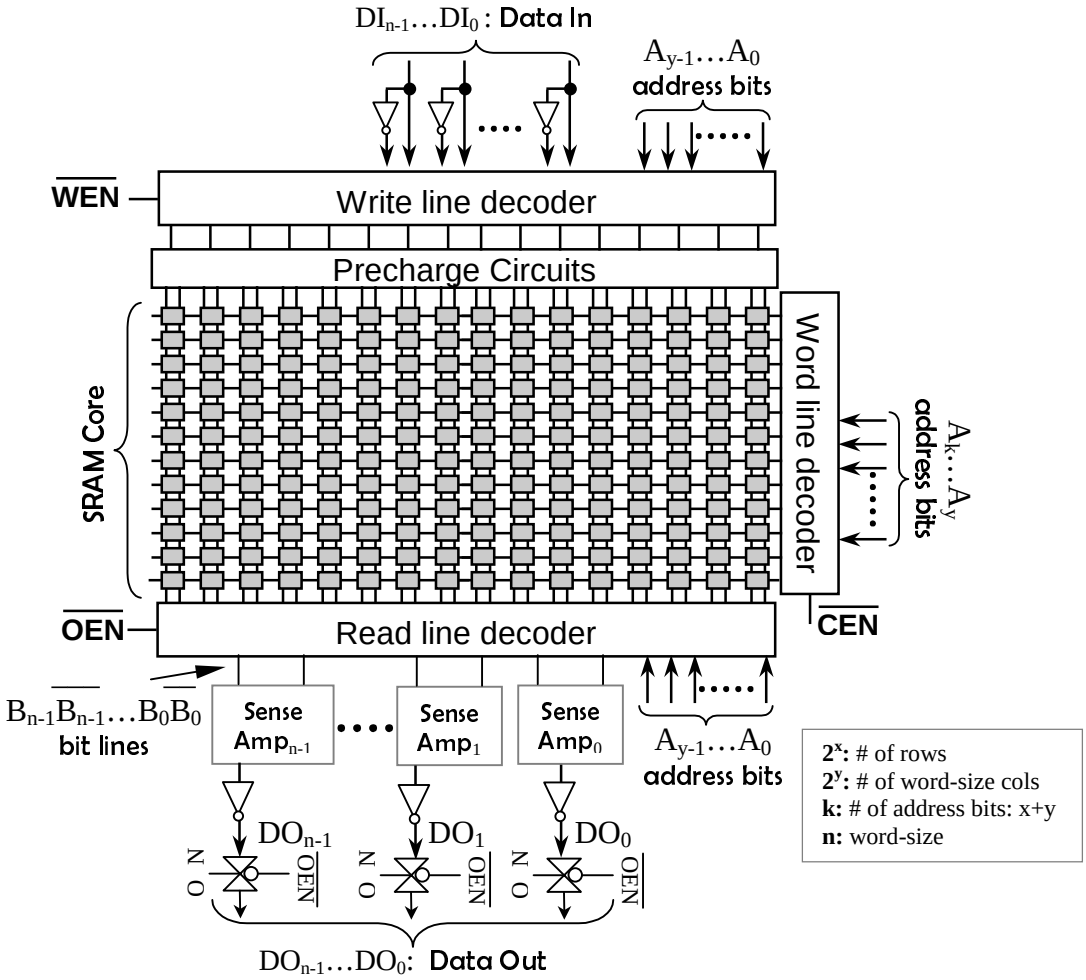


Figure 3.1 – SRAM Macro

From Figure 3.1 it can be seen that there exists two sepearte column decoders – one for read operations and one for write operations. Though a single column address decoder can be used for both read and write operations, we use two separate decoders for read and write operations. The reason for using two column decoders is that the use of two decoders reduces the delay incurred due to the routing from a single decoder. In addition, the actual area of the decoders is small, while routing area is significant.

3.1 SRAM Core

The 6-transistor (6T) SRAM core shown in Figure 3.2 stores one bit of data. It is composed of a latch and 2 pass transistors. Since the core is replicated by the number of bits, optimum design and layout of this component is critical. The size of the transistors used is the primary factor that determines the performance of the SRAM cell. We determine the optimum transistor sizes through SPICE simulation. Since the most important design criterion for us is power dissipation, we minimize the sizing as much as possible without compromising performance significantly.

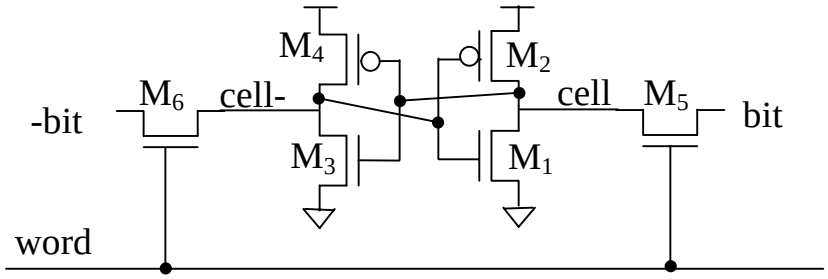


Figure 3.2 – 6-Transistor SRAM Cell

There are some issues to be considered when sizing the transistors. The latch inverters (M_1, M_2, M_3, and M_4) form a positive feedback loop, so that the stored value is maintained as long as power is available. Since the bit lines are precharged to V_{DD}-V_{tn}, the cell NFETs (M_1 and M_3) cannot be smaller than the pass NFETs (M_5 and M_6) to overcome the current value on the bit line when pulling it to a low value. Note that though a transmission gate may be used for the pass-transistors, only NFETs are used so that the area for a single SRAM cell may be small. It will be shown later that special circuitry (bit-line conditioning and sense amplifiers) is needed to recover from the performance losses due to using just NFETs.

In an array of RAM cells, a single word line is connected to an entire row of RAM cells, forming a long word-line row. Since the word line uses polysilicon (which has high resistivity), it is necessary to keep the two pass transistors (M5 and M6) small. This improves signal integrity on the word lines and reduces power dissipation. Therefore, we keep the size small.

We set all transistor lengths to the minimum, which is 2λ (= 0.35 μm) for the target 0.35 μm process. Based on simulation, we set the widths of all transistors to 0.4 μm, the minimum width for the target process.

The next step is to lay-out the leaf cell. The schematic diagram corresponding to the placement of transistors and the layout for an SRAM cell are given in Figure 3.3. The placement of the transistors is intended for cell abutting.

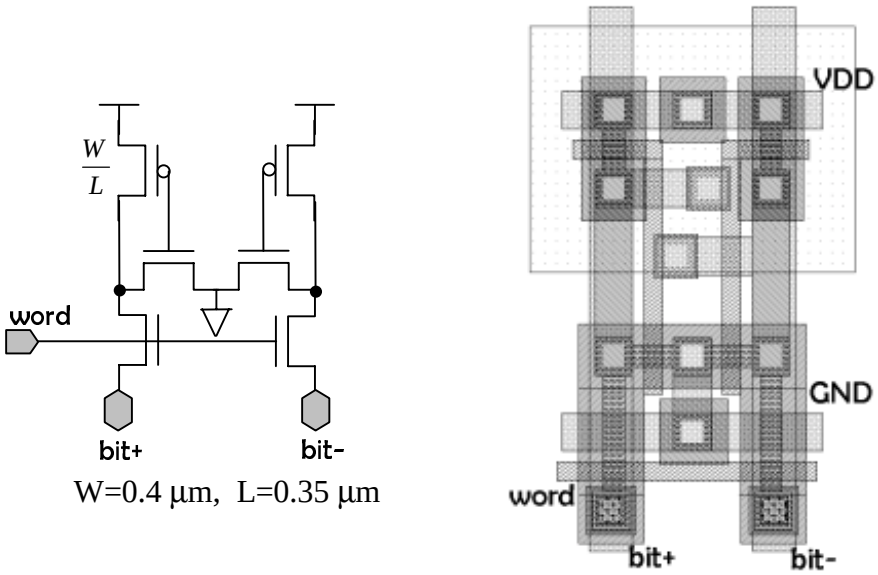


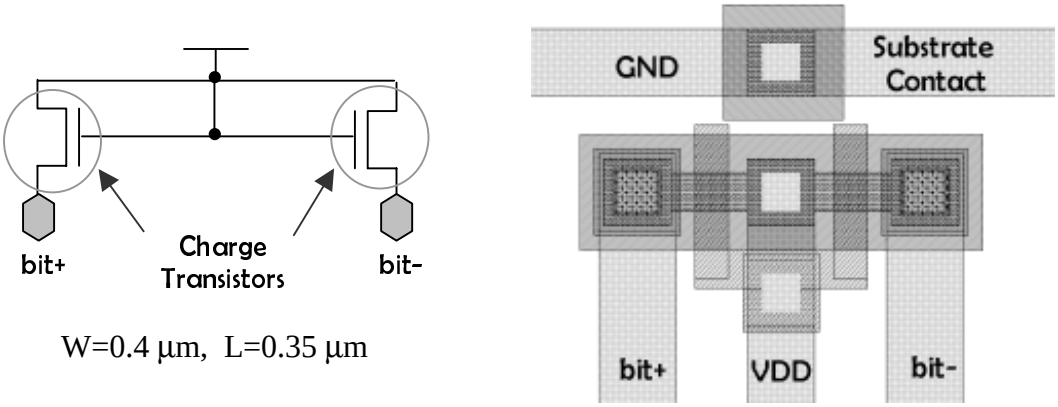
Figure 3.3 – Schematic and Layout of SRAM leaf cell

Note that all the I/O signals of the cell (word, bit+, bit-, VDD, and GND) use abutting. The layout allows both horizontal and vertical cell abutting. Vertically, the cell above this one will be flipped on the x-axis so that the n-well and VDD lines are shared. Similarly, the cell below will also be mirrored on the x-axis so that the n-diffusion and bit lines can be shared. This overlap of layers makes the layout more compact.

3.2 Bit-Line Conditioning

Figure 3.4 shows the schematic and layout of the bit-line conditioning circuit. The gates of the NFETs are tied to V_{dd}, so that the bit-line conditioning circuit is always turned on. This avoids the complexity of generating a precharging signal. It also allows the bit-lines to be equalized when the column is deselected (i.e., between two access cycles). The bit-lines get *equalized* to the charge value of V_{dd}-V_{tn} between two accesses,

when the memory array is deselected. When two RAM cells containing opposite value in the same columns are accessed subsequently, the output has to switch first to an equalized state and then to the opposite logic state. Since the capacitance on the bit lines is quite large, the time required for switching the differential from one state to the other becomes a significant portion of the overall access time. Equalization of the bit-lines between the accesses can reduce the access time. The size of the charge transistors must be as small as possible, so that they do not override the value in the latch during read and write operations. Simulation showed that the charge transistors performed optimally when W=0.4 μm and L=0.35 μm. The layout of the leaf cell allows cell abutting of the bit lines.



W=0.4 μm, L=0.35 μm

Figure 3.4 – Schematic and Layout for the Bit-Line Conditioning Circuit

3.3 Sense Amplifiers

During a read operation, the selected latch outputs the stored value onto the two bit-lines. Since the bit-lines are always precharged, the bit-line differential voltage degrades. We use sense amplifiers to improve the differential voltage from the bit-lines. The main advantage in using a differential bit-lines is common-mode rejection, which reduces noise effects and signal degradation.

In our SRAM design, a single sense amplifier is shared among multiple columns. Typically, a single amplifier is used for each column of bit-lines as shown in Figure 3.5. However, in the proposed design, a single amplifier is shared between multiple columns by inserting the column decoder pass-transistors between the bit-lines and the amplifier. This results in area savings and power reduction.

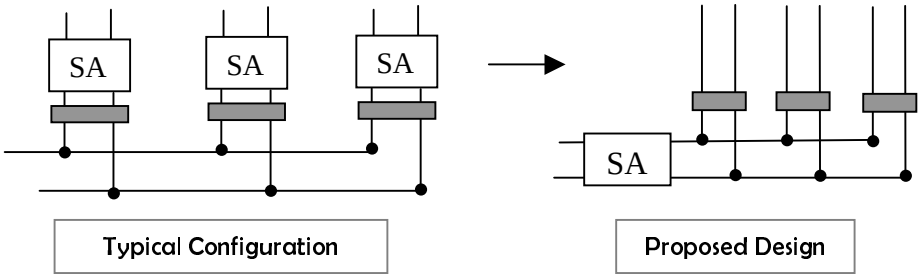
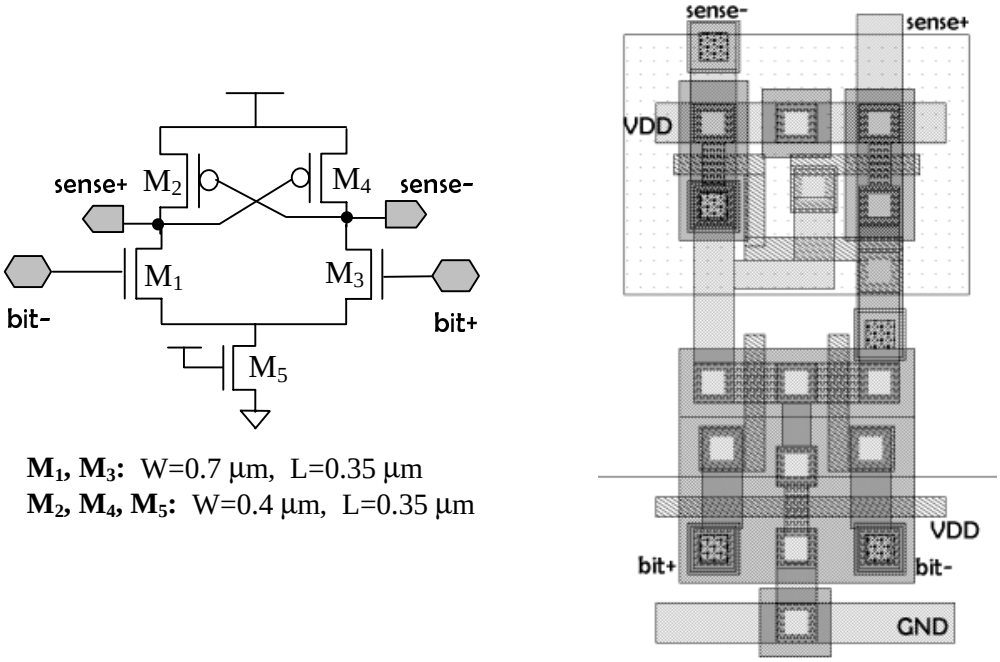


Figure 3.5 – Sense Amplifier Architecture

From simulation, it was shown that this configuration performed better than having a sense amp for each column, since it reduces the drive load of the sense amplifier. Also, this configuration allows the sense amplifier to be isolated from the bit-lines at all times except during a read operation. Because the sense amplifiers are not driven by bit-lines at all times, the switching activity is reduced on the sense outputs.



M_1, M_3: W=0.7 µm, L=0.35 µm
M_2, M_4, M_5: W=0.4 µm, L=0.35 µm

Figure 3.6 – Schematic and Layout of Sense Amplifier

Figure 3.6 shows the schematic and layout of a sense amplifier. A cross-coupled amplifier is used for the sense amp. Once a memory cell is selected for the read operation, the voltage on one of the complementary bit lines will start to drop slightly. Suppose that bit+ is higher than bit-. As a result, one of the NFETs, M_3, is turned on, causing sense$^-$ to be pulled low. Consequently, one of the PFETs, M_2, is turned on,

pulling up sense^{+} output to a high value. The positive feedback of the cross-coupled PFETs accelerates the sensing speed by reinforcing M_2's gate value (sense-) to a high through M_3.

The sense amplifier is the key component that limits the speed of read-time. Since the transistor sizing affects the speed of the sense amplifier, simulation was performed for different sizes of transistors. The fastest configuration is when the two NFETs (M_1 and M_4) are set to W=0.7 μm, L=0.4 μm and the rest were set to W=0.4 μm. The layout shown in Figure 3.5 is the fastest configuration and also uses cell abutting of V_{DD} and GND.

3.4 Leaf Cell Simulation Results

After custom layout of the leaf-cells in Cadence Virtuoso, the design rule checker (DRC) was used to verify that all leaf-cell layouts met the TSMC 0.35 μm design rules. The leaf-cells were used first to create a layout for a small test circuit to verify operation and measure preliminary performance results. The test circuit consisted of an SRAM cell core connected to the bit-line conditioning circuit through the bit-lines with a sense amplifier to amplify the read output. The sense amplifier is isolated from the bit-lines when the cell is not being read from. Following DRC verification, parasitic capacitances were extracted from the layout. From the extracted circuit, a spice netlist was generated using Analog Artist, and simulations were performed using Avanti HSPICE. The extracted netlist was simulated for the following test case.

1) *Write 0: word = 1, write=1, write_data = 1->0*
2) *Read 0: word = 1, write=0, write_data=0->1->0 (shouldn't affect contents)*
3) *Write 1: word=1, write=1, write_data = 0-> 1*
4) *Read 1: word = 1, write=0, write_data=1*
5) *Turn-off RAM Cell: word=0, write=0*

Figure 3.7 presents the simulation graph for a R/W to a single bit. The *cell* represents the value stored in the latch, while the *bit* represents the value on the bit lines. The output from the sense amplifier is labeled as *sense*. From the plot, we can see that the value in the cell node is driving the bit line. Because of the bit-line conditioning

circuitry (and also the bit-line capacitance), the bit line is not pulled to a good '0' during a read of '0'. However, the sense amplifier recovers the original value after some delay. After 20ns, the RAM cell is turned off, so the bit-line conditioning circuit drives the bit value, while the sense node floats towards an equalized value. Note that the cell still maintains the stored '1', regardless of CEN being off.

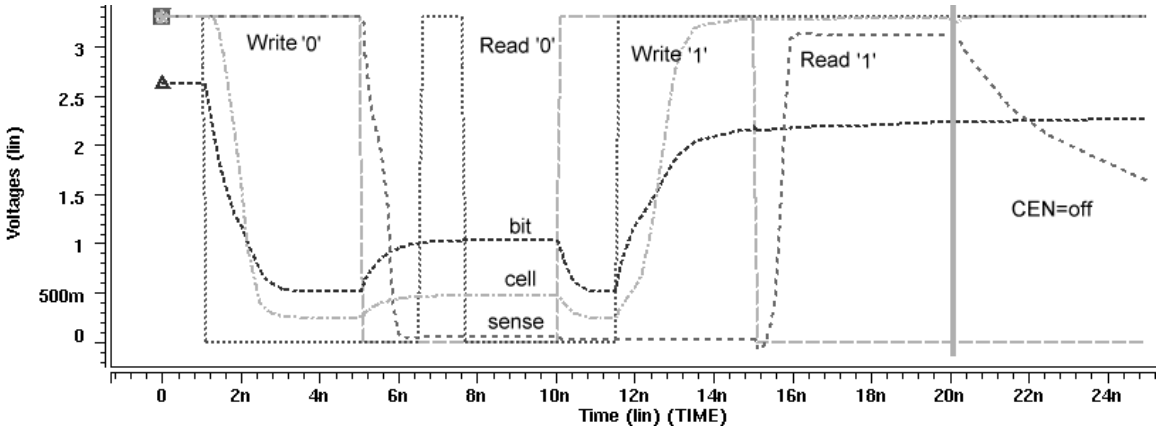


Figure 3.7 – Simulation Results for SRAM Leaf Cell

Characteristics for this cell are provided in Table 3.1. Power dissipation was obtained using HSPICE's *.measure* statement. Static power dissipated was obtained by taking the average of the two power dissipations, under the sense output at a high and the sense output at a low. Dynamic power was taken as the average of power dissipated during a change in the output due to an input change. For this example, the dynamic power dissipation included the average of dynamic power dissipation from both R/W' and write_data changes. Nodal capacitances for the cell were obtained from HSPICE by adding the *captab* (capacitance table) option to the *.option* statement.

Table 3.1 – Characteristics of a bit SRAM for V_{DD} = 3.3 V

Power Dissipation	Static = 0.45 mW Dynamic = 0.82 mW
Nodal Capacitance	Cell nodes = 8.9 fF Word lines = 8.7 fF Bit line = 11.2 fF
Area per cell	35.64 μm^2

3.5 Address Decoders

Decoders are needed to generate the word and column select signals for the SRAM. The input to the decoder is the address of the selected cell and the control signals. All decoders are implemented in a tree structure, as shown in Figure 3.8. Minimum-width (W=0.4 μm, L=0.35 μm) sized NFETs are used as pass transistors in the decoder.

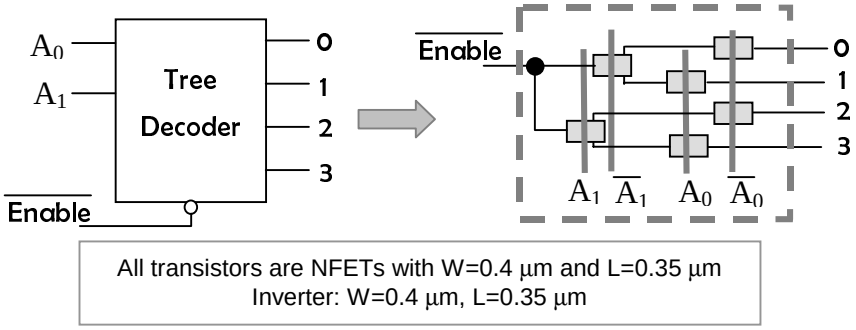


Figure 3.8 – Tree Decoder Implementation

When Enable is active (low), a selected decoded output is pulled down to a good logic '0' value due to the use of NFETs. All unselected outputs are floating. When Enable is disabled (high), the selected output is at a poor logic '1', and all unselected outputs are floating. To prevent unselect unselected outputs being floating, pull-up buffers are necessary at each output. The design of pull-up buffers is explained next.

3.5.1 Pull-up Buffers for the Decoder

In addition to pulling up unselected lines, a buffer is also needed to produce a good '1' or '0' for the select lines. A buffer is responsible for both pulling-up unselected lines and buffering the output so that the drive strength is increased. As shown in Figure 3.8, a buffer, in fact an inverter, is added at every decoded output in our design. Note that the selected output is at '1' due to the inversion. The result is that all decoder outputs are zero except for the output that is selected by the input address. It will be placed at the output of the decoder, as shown in Figure 3.9.

Two types of pull-up transistors as shown in Figure 3.10 are considered for the buffer design. Both designs require a pull-up transistor for an unselected line. To compare performance, HSPICE simulation for the two designs was performed.

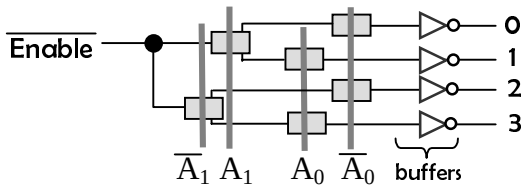


Figure 3.9 – Buffered Output for Decoder

The design in Figure 3.10 (a) uses an NFET to act as a pull-up resistor to pull-up an unselected line. However, in order not to pull-up the selected line, the driving capability of the pull-up transistor needs to be low. Therefore, the width is set to the minimum size of W=0.4 μm and the length is L= 0.75 μm.

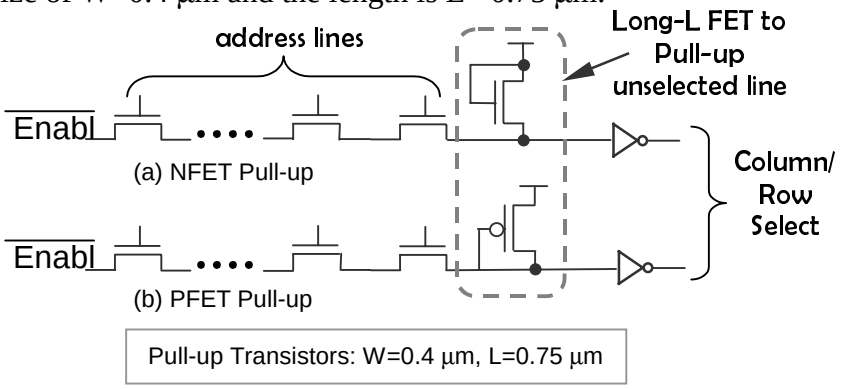


Figure 3.10 – Comparative Buffer Designs

Figure 3.11 shows the simulation results of the buffer on a 1 kB SRAM. In Figure 3.11, when the decoder is disabled, the decoder outputs, equivalently inverter outputs, are at 0V for both selected and unselected lines. However, when the decoder is enabled, the selected decoder output is at 0.4 V and fails to pull up high. This is due to the fact that the pull-up transistor is too strong to be pulled down to a sufficiently low value. Therefore, this buffer design function properly without reducing the driving capability further. It requires increasing the length (since width is already the lowest), to result in increased area, so that this configuration is not adopted in our design.

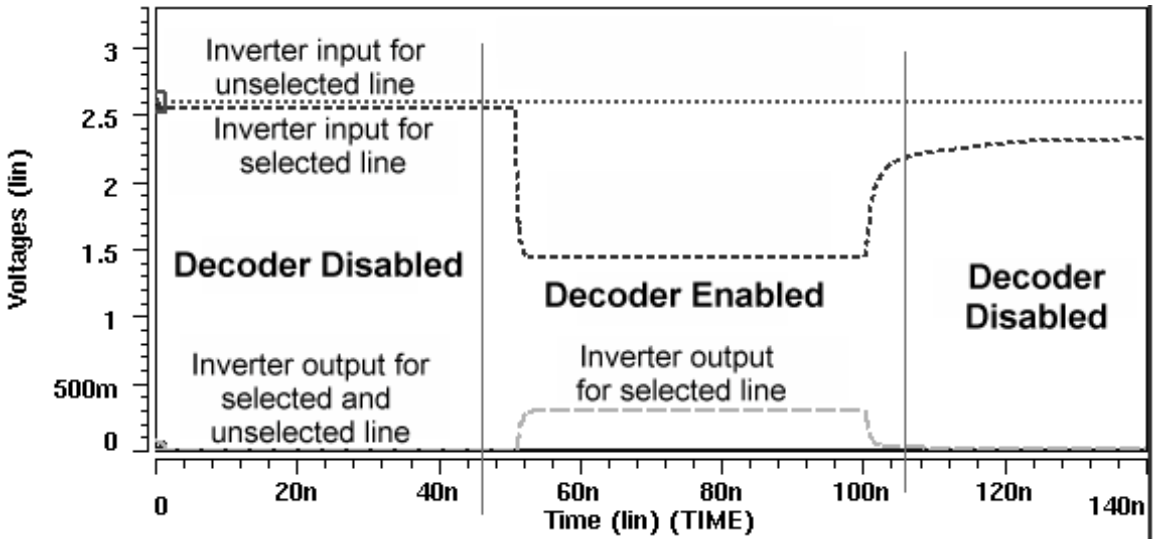


Figure 3.11 – Simulation Results for Figure 3.10 (a)

Alternatively, the design in Figure 3.9 (b) uses a PFET with W=0.4 μm and L=0.75 μm. In this case, the gate of the PFET samples the value from the line. If an unselected line is floating to '0', then it will be pulled up by the PFET. Figure 3.12 shows the simulation results for the buffer design in Figure 3.9 (b) on a 1-kB SRAM.

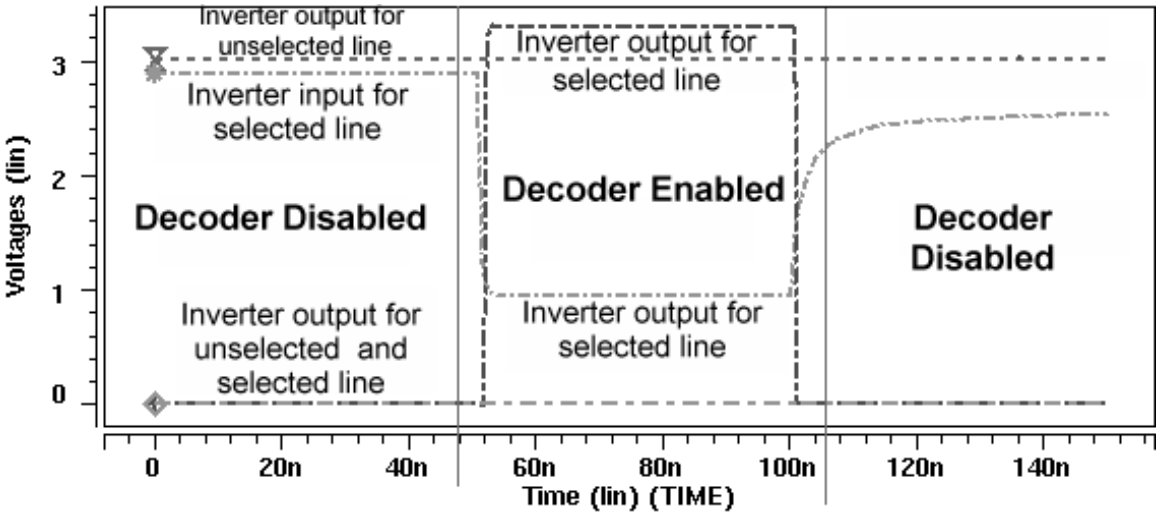


Figure 3.12 – Simulation Results for Figure 3.10 (b)

Figure 3.12 shows that when the decoder is enabled, the selected decoder (i.e. inverter) output is pulled up to V_{dd} (=3.3 V). Note that the inverter input of the selected line is sufficiently low (=0.9 V) to drive the inverter output to V_{dd}. Since this design works well, it is adopted for our final design. Figure 3.13 shows the schematic and

layout of the final decoder buffer design, which includes a pull-up PFET with L=0.75 μm, W= 0.4 μm and an inverter with L=0.35 μm, W= 0.4 μm.

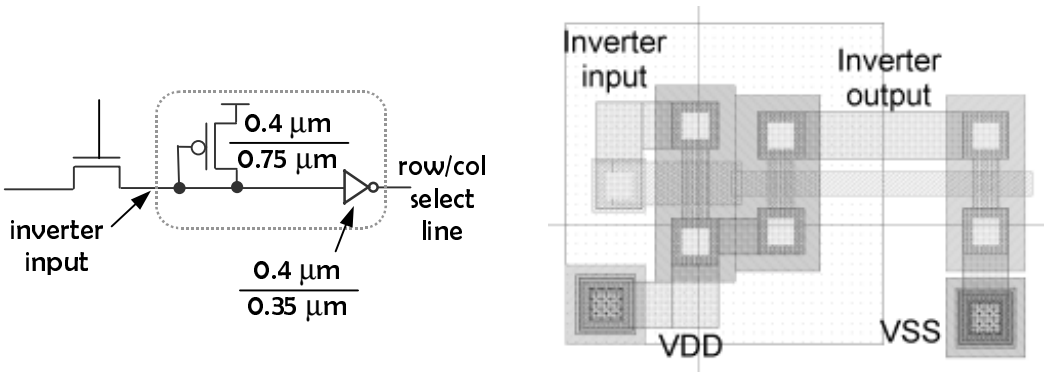


Figure 3.13 – Schematic and Layout of Buffer

3.6 Summary

Leaf cell layout is critical in to the performance of the SRAM. In this chapter we examined the leaf cell layout and design. The performance of each cell has been measured and verified through SPICE simulations.

CHAPTER

4 SRAM Compiler

In the previous chapter, we described the design of leaf-cells used to layout an SRAM core and the supporting circuitry. The next step is to develop SKILL code to perform design automation of all components including the RAM core, decoders, and I/O buffers. In this chapter, we discuss the structure of the SKILL code for our SRAM compiler.

4.1 SRAM Structure and Algorithm for SKILL Code

Our SRAM compiler should generate the layout for the SRAM core and all supporting circuits based on the input size. The entire program is broken into the modules based on the functionality. Figure 4.1 shows the organization of the program.

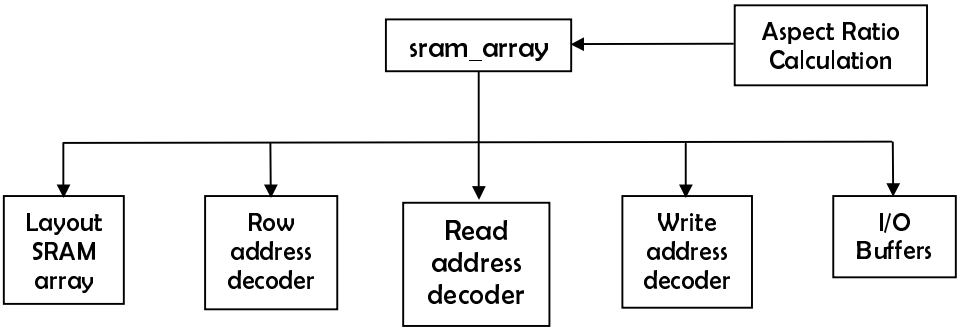


Figure 4.1 – Program Organization

The procedure *sram_array* is the top level function that calls all other modules to generate the entire circuit. We now discuss the implementation and interaction of all of the functions.

4.1.1 Aspect Ratio Calculation

The numbers of rows and of columns in an SRAM have a major impact on the final aspect ratio of the SRAM. It is undesirable for the shape of the SRAM circuit to be overly long or thin, as it incurs excessive routing area, signal delay, and capacitance. Optimally, the SRAM should have a shape close to a square. Therefore, it is important to derive a procedure to calculate the rows and columns with the aspect ratio in mind.

The first step in this procedure is to determine the aspect ratio for a single bit. Since adjacent RAM cells are flipped both horizontally and vertically to improve cell abutting (see Section 3.1), the basic tile for measuring the aspect ratio is a 2x2 cell. Figure 4.2 gives the measurements for a 2x2 cell and the derived measurement for a single cell. The aspect ratio for one cell is given by, $AR_{bit} = \frac{width}{height} = \frac{4.8}{7.4} = 0.65$.

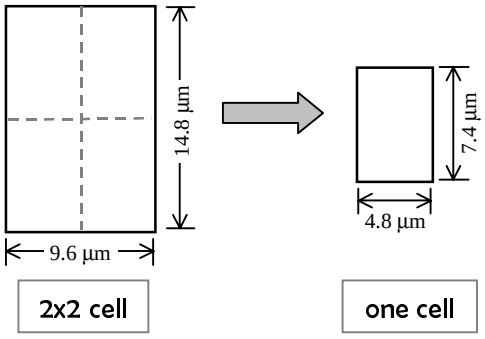


Figure 4.2 – Aspect Ratio Measurements

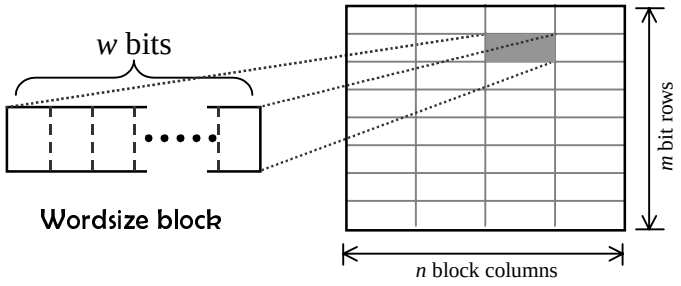


Figure 4.3 – SRAM Core and Word Blocks

Recall from Section 2.5 that the columns are arranged in word-sized blocks. A word-sized block is the smallest unit for the SRAM core as shown in Figure 4.3. The aspect ratio for one block is $AR_{block} = AR_{bit} \cdot w$, where w is the number of bits in a word. Therefore, the total aspect ratio for an SRAM can be expressed as,

$$AR_{SRAM} = AR_{block} \cdot \frac{columns}{rows} = AR_{block} \cdot \frac{n}{m} = AR_{bit} \cdot w \cdot \frac{n}{m}$$

where m is the number of rows and n is the number of word-sized columns. Note that m and n should be a power of 2 for efficient implementation of the decoders. Let $m = 2^x$ and $n = 2^y$, where x and y are integers. In order to make AR_{SRAM} close to 1,

$$1 = AR_{SRAM} = AR_{block} \cdot \frac{n}{m}$$
$$m = AR_{block} \cdot n$$
$$\log_2(m) = \log_2(AR_{block}) + \log_2(n)$$
$$\log_2(2^x) = \log_2(AR_{block}) + \log_2(2^y)$$
$$\therefore \quad x = \log_2(AR_{block}) + y$$

Assuming that the number of words of an SRAM is a power of 2, let *words* = 2^l, where *words* is the number of locations in the SRAM and l is an integer. Since $words = m \cdot n$, we obtain $2^l = 2^x \cdot 2^y$. Hence $l = x + y$. Using this relation, we can compute the value of x as:

$$x = \log_2(AR_{block}) + y = \log_2(AR_{block}) + l - x$$
$$2x = \log_2(AR_{block}) + l$$
$$\boxed{x = \frac{\log_2(AR_{block}) + l}{2}}$$

The procedure for the aspect ratio is as follows:

1. Calculate aspect ratio for one block: $AR_{block} = AR_{bit} \cdot w = 0.65 \cdot w$
2. Find l: $l = \log_2 words$
3. Find x: $x = \frac{\log_2(AR_{block}) + l}{2}$. Round down to make it an integer.
4. Find y to calculate number of rows and columns: $y = l - x$. *Hence* $m = 2^x$ and $n = 2^y$

In the above, *words* (which is the number of locations) should be a power of 2. For example, for a 256×8 SRAM,

1. AR_{block} = 0.65*8 = 5.2
2. $l = log_2 256 = 8$
3. $x = (log_2 5.2 + 8)/2 = 5.2 \rightarrow 5$
4. $y = l - x = 8 - 5 = 3$

Hence, the SRAM should have $2^x = 2^5 = 32$ rows and $2^y = 2^3 = 8$ word-sized columns for an aspect ratio close to 1. The actual aspect ratio of the SRAM core,

measured from a layout, has an aspect ratio of width/height = 306/240 = 1.275. The reason for the discrepancy is due to the constraint to impose the smallest unit to a word-sized block. In addition, the supporting circuitry, which is ignored in the aspect ratio calculation, aggravates the aspect ratio. However, the impact of the supporting circuitry decreases with the increase of SRAM size. For the 256×8 SRAM, the total aspect ratio with the support circuitry improves to width/height = 400/335 = 1.19.

A small block of code in the top-level module *sram_array* calculates *x*, *y*, and subsequently, *m* and *n* and passes them to all other functions at the lower level.

4.1.2 Layout of an SRAM Array

The next step is to layout RAM cells in *m* rows and *n***w* bit columns. This is accomplished by the *cell_layout* function. The function instantiates the leaf-cell previously created for 6T SRAM and the bit-line conditioning circuitry (Refer to Section 3.1), to create an SRAM core, bit-line conditioning, and write-select transistors (which activate a pair of selected bit-lines during the write operation). The function also places necessary I/O pins. The procedure for this function is as follows.

1. Layout $m \times n$**w* SRAM array
2. Place bit-line charging circuit and write-select transistors.
3. Route VDD & VSS lines

To layout the SRAM array, we use a nested *for-loop* to instantiate the RAM cells in an array. The pseudo-code for this function is as follows.

```
for column = 0 to (n*w)-1,
  for row = 0 to m-1,
    if (row == odd then
      Instantiate("6T_core" @ {x_offset*column, y_offset*row} flipped)
 else
      Instantiate("6T_core" @ {x_offset*column, y_offset*row} normal)
    )
  end for
end for
```

Note that "column" in the pseudo-code denotes a column of RAM cells. The cells on every other row are flipped on the y-axis to take advantage of cell abutting. This also lets the n-wells be shared between two rows, which results in a compact layout

After the array has been laid out, the next step is to add the bit-line conditioning circuitry and write-select transistors. The bit-line conditioning circuit is needed to charge the bit-lines to reduce the access time. A write-select transistor activates the selected bit-lines for writing, while isolating all other bit-lines that are not-being written to. During a write operation, the "write" signal (Refer to Figure 4.4), which is an output of the column decoder, for the selected bit-lines is '1' to activate the selected bit-lines. All other select signals should be disabled so that they don't get written to. A minimum sized (L=0.35 μm, W = 0.4 μm) NFET is used to isolate the bit-lines.

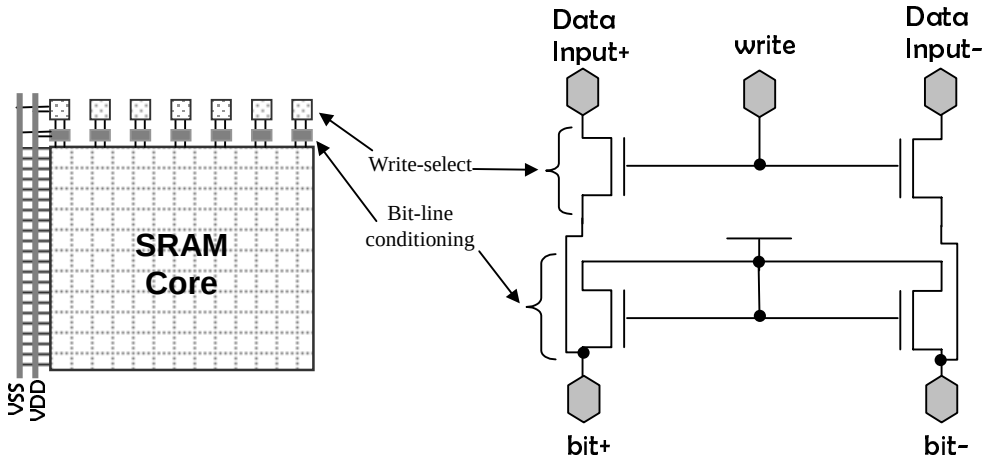


Figure 4.4 – Layout Generated by the *cell_layout* Function

The final step of the *cell_layout* function is to route the VDD and VSS lines to power the SRAM. This is done by routing two *metal-2* lines along the side of the core. Figure 4.4 shows the complete layout accomplished by the *cell_layout* function for a 256×8 SRAM. Cell-abutting is used to connect the VDD and VSS lines for the 6T-core (Refer to Section 3.1). Only one connection to the VDD and VSS bus is necessary for each row.

4.1.3 Row Address Decoder

The row address decoder is responsible for generating the word-signals for each row. As mentioned in section 3.5, a tree structure is used for the decoder. The SKILL code that is responsible for implementing the row-address decoder resides in the *word_decoder* procedure. The function consists of the following steps.

1. Layout a tree-structured row decoder using NFET.
2. Add substrate contacts.
3. Add pull-up buffers to the decoder output.

From the aspect ratio calculation given in Section 4.1.1, we obtain the number of row address bits, *x*, necessary for the decoder. As the decoder needs both non-complement and complement address lines, 2*x* address bit lines are necessary. A 3x8 tree structured row decoder with six address lines is shown in Figure 4.5.

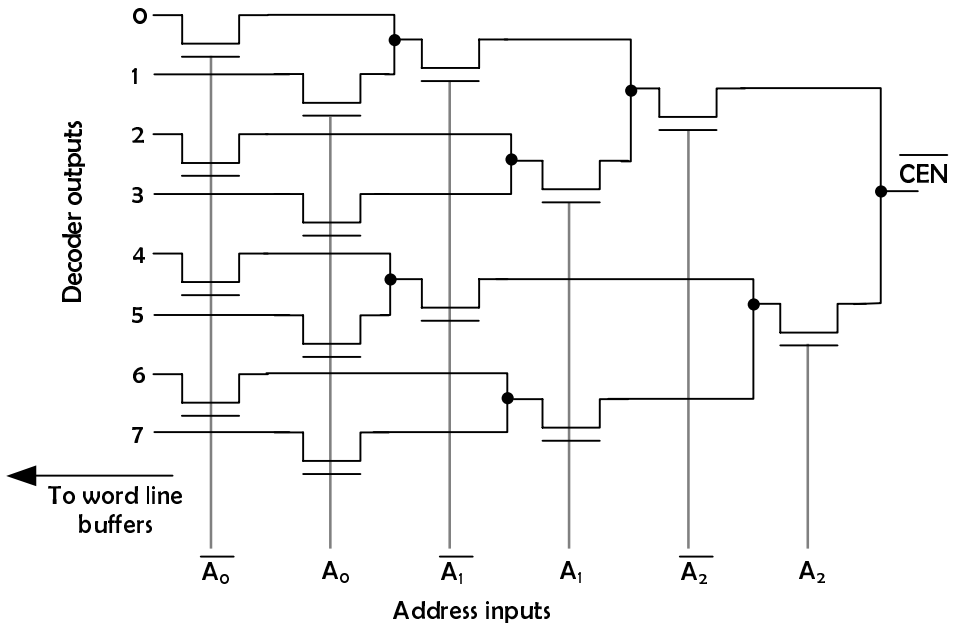


Figure 4.5 – Implementation of a Tree-Structured Row Decoder

The first step is to layout the tree-structured row decoder. 2*x* polysilicon lines are laid out to form the address lines. Next, a nested *for-loop* is used to layout NFETs at the appropriate coordinates. The pseudo-code for the layout of NFETs is as follows.

```
for addrline = 0 to x-1,
  for row = 0 to m-1,
    if (row == odd then
      if(NFET should be placed for this row then
        // Place NFET on uncomplemented address lines (ex: A0)
        Instantiate("NFET" @ {@uncomplemented address line})
        Connect to previous (lower) address line
      )
    else
```

```
      if(NFET should be placed at this row then
        // Place NFET on complemented address lines (ex: A0')
        Instantiate("NFET" @ {@complemented address line})
        Connect to previous (lower) address line
      )
    )
  end for
end for
```

The coordinates of the NFETs are calculated as a function of the current row and current address line. The source side of the two NFETs on the MSB address bit lines is connected to the enable signal, CEN'.

After the layout of the decoders, substrate contacts are placed at every 5 μm to meet DRC rules. A substrate contact is placed below and above each NFET for the LSB address lines, and subsequently every 5 μm for other address lines. All substrate contacts are connected to a VSS bus. Recall from section 3.5.1 that pull up buffers are necessary for this decoder. We use the leaf-cell for the pull-up buffers and connect them to the output of the decoder.

Figure 4.6 shows the partial layout of a 5x32 row decoder generated by the *word_decoder* function. Notice that the decoder resembles a tree-like pattern and the pull-up buffers are connected to all outputs of the decoder.

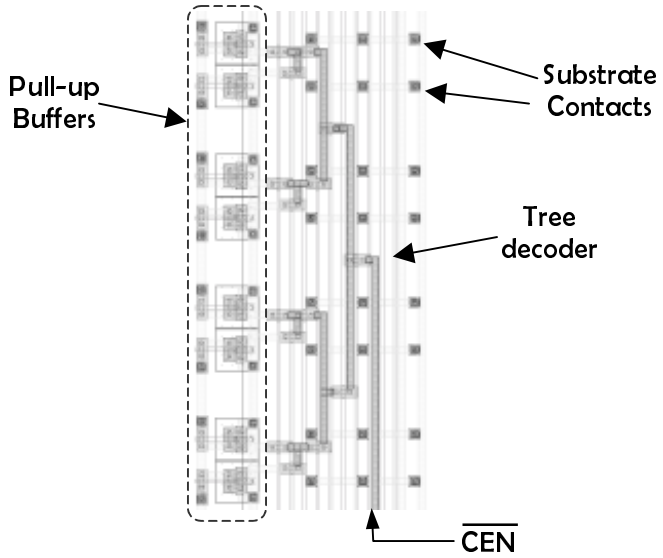


Figure 4.6 – Word-Decoder Layout

4.1.4 Read Address Decoder

The read address decoder of an SRAM activates *w* consecutive selected bit-line columns, where *w* is the wordsize of the SRAM. The layout of the two decoders is identical to that of the row decoder, and hence we describe only the control circuitry for read and write operations. The read address decoder is responsible for activating selected columns and routing the read-data to the "data out" bus.

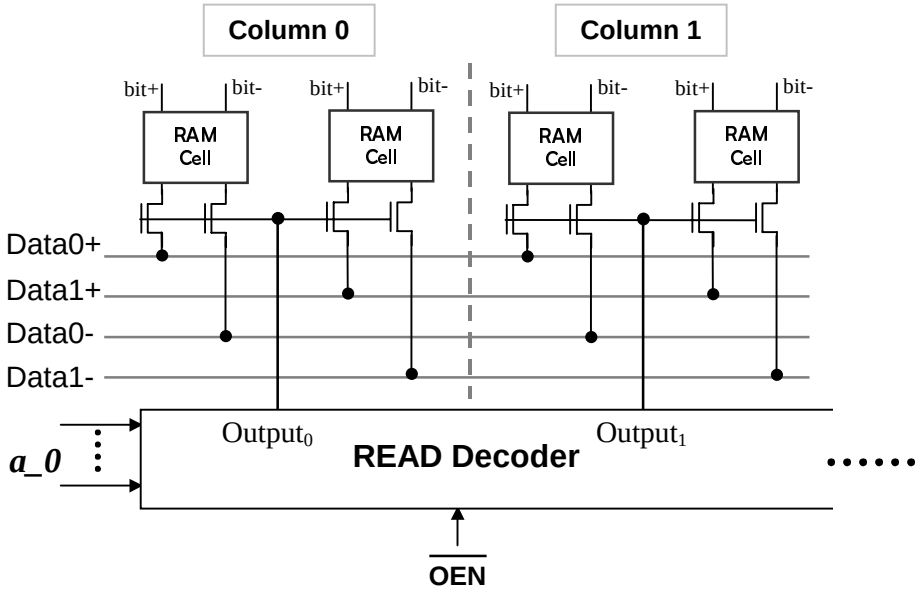


Figure 4.7 – Read decoder for Wordsize=2

The read decoder, like the row-decoder, outputs an active-high signal on it's output. This output signal is the column-select signal for a read-operation. Figure 4.1 illustrates the read-decoder and supporting circuitry for an SRAM with a wordsize of 2. As an example, suppose column 0 is selected by the read-decoder. Hence, the read decoder enables (pulls high) the $Output_0$ signal, which is gated to the read-select pass transistors. Therefore, all bit-lines in column 0 will be connected to the data bus. Meanwhile, all other bit-lines from the other columns are disconnected from the data-bus.

Routing the data-out is done in the following way. Each column is connected to a pass transistor (similar to the write-select transistor discussed in section 4.1.2), which is gated by the column-select signal output by the read-decoder. If a particular column is chosen, then the pass transistor will connect the bit-lines for that column to the data bus. The pseudo-code for this function is as follows.

```
//////////  1.  Layout Column Decoder  /////////
```
- See pseudo-code for previous section

```
//////////  2.  Connect wordsize blocks together  /////////
for col = 0 to (n*w)-1     //n*w = number of bit-columns
```
- Place write-select transistors for each columns
  - Two pass transistors for each column (for bit & bit_neg)

```
end for

////////////////  3.  Layout Data Bus   ///////////////////
for data = 0 to w-1   //w = word size
```
- Draw a horizontal bus for data+ for this bit
- Draw a horizontal bus for data- for this bit

```
end for

//////////////  4.  Connect to Decoder   /////////////////
for block = 0 to n-1  // n = number of block-columns
```
- Make column select common for columns in the same block
- Route out write-select signal (to be connected to decoder out)
- Connect write-select signal to decoder (pull-up buffer output)

```
end for
```

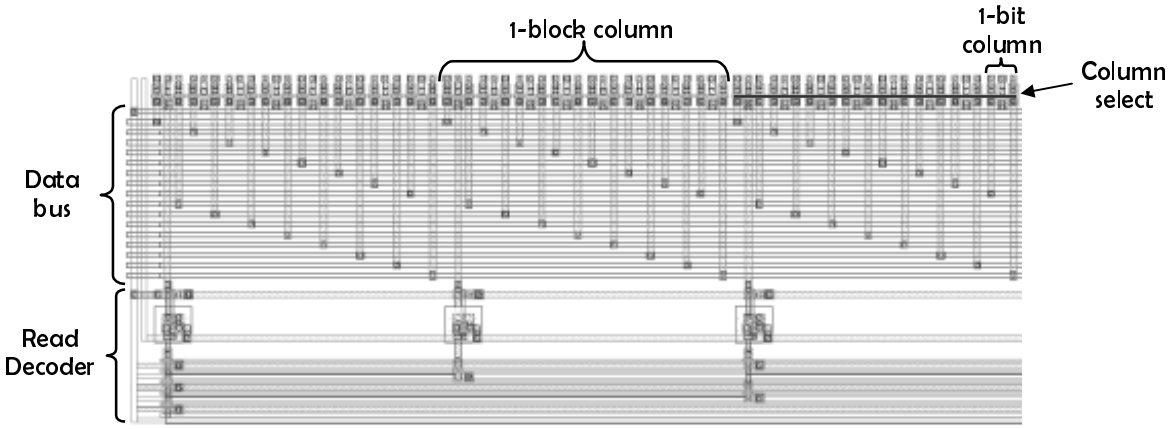


Figure 4.8 – Read Address Decoder Layout

The code to layout the read-address decoder is contained in the *read_decoder* function. The final layout for this function can be seen in Figure 4.8, which is generated for a 256×8 circuit. Note that the data output from each block is connected together so that all blocks share the same bus for a single bit. The decoder and buffer implementation is identical to the one used for the row-address decoder.

4.1.5 Write Address Decoder

The write address decoder is actually modeled after the read-address decoder. The data is routed out the same way as for the read-decoder. The code for the write-address decoder is in the *write_decoder* function. While the read address decoder is placed at the bottom of the SRAM array, the write-address decoder is placed at the top of the array.

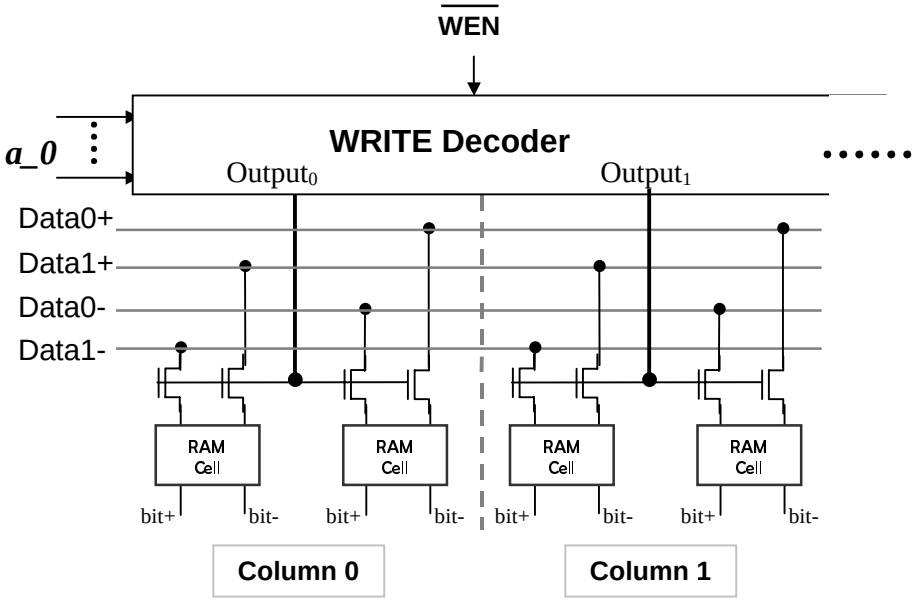


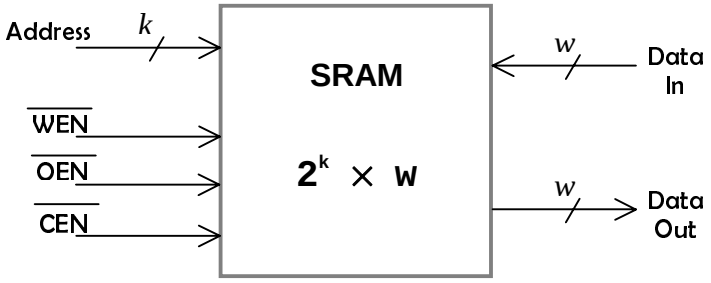
Figure 4.9 – Write-Decoder for Wordsize of 2

Figure 4.9 illustrates the write-decoder for a word-size of 2. The architecture is similar to the read-decoder architecture given in Figure 4.7. The operation of the write-decoder is also similar to the read-decoder operation. From the address given, a_0 … a_x, the write-decoder outputs an active-high for the selected column output. This connects the selected column block with the data-bus. The relative position of all components for the SRAM will be discussed in the next section.

4.1.6 I/O Buffers and Packaging

The final step is to add the I/O buffers for the SRAM circuit. All I/O signals need to be routed out to the outside so that they are easily accessible by a router. We used only *metal1* and *metal2* layers for our SRAM, so other metal layers maybe used by an auto-router, if needed. Finally, each signal line is labeled for identification. The following I/O

signals and their labels are shown in Figure 4.10. In this section we will discuss the final packaging for each I/O signal group.



I/O Signals	Labels
Address	A_0 … A_k-1
Data-In	DI_0 … DI_w-1
Data-Out	DO_0 … DO_w-1
Control Signals (active-low)	WEN OEN CEN

Figure 4.10 – I/O Pins of an SRAM

As mentioned in Section 4.1.3, each decoder generates the complemented address signals necessary for the decoder. Therefore, routing of complemented address signals is unnecessary. The three active-low control signals, WEN', OEN', and CEN', form the enable signals for the three decoders. Since they are direct input, they are routed out to the top right corner of the SRAM. Figure 4.11 shows the placement of I/O signals and the major components of an SRAM.

Two data signal groups are Data In (DI) and Data Out (DO). Both signals are output from their respective column decoders. Recall from the previous section that, a data bus is present for the column decoders. A data bus routes $2 \cdot w$ data signals, the non-complemented and complemented bit signals for each bit, for the word size, w. A pair of data-output signals, non-complemented and complemented signals, is fed into a sense amplifier, which generates a logic value read from the cell. The sense output is buffered via an inverter. The inverter drives the DO outputs through transmission gates only when OEN is enabled.

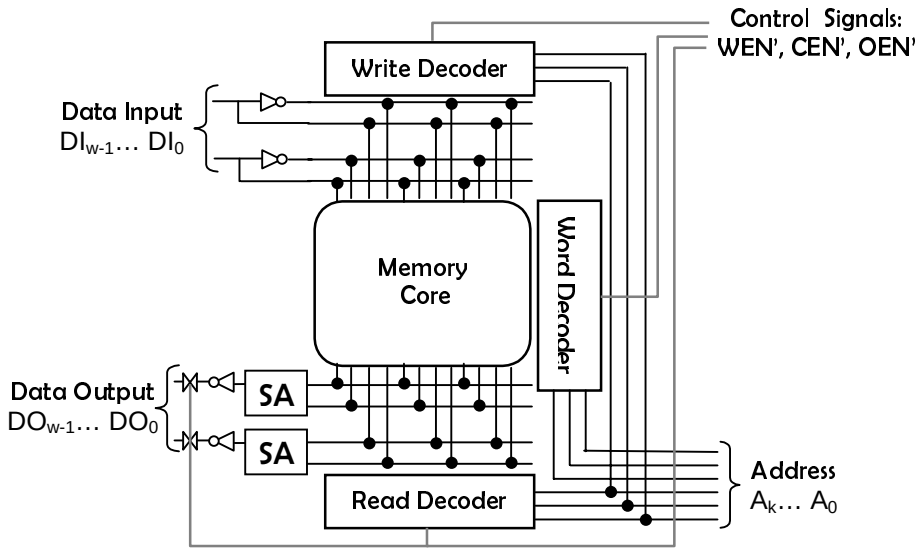


Figure 4.11 – Placement of I/O Signals

The SKILL code for this section is contained in the *package* function. This function will layout I/O buffers and sense amplifiers, route all I/O signals, and create pins.

4.2 SRAM Macro Layout

Figure 4.12 shows the layout of a 256x8 SRAM generated by our RAM compiler. This circuit is 390 μm wide by 340 μm high, with the aspect ratio being 390/340 = 1.2. This circuit contains 13,019 transistors.

The following page also contains the SRAM circuit generated for a 1-kB SRAM (Figure 4.13). This SRAM circuit is 700 μm wide by 580 μm high, with the aspect ratio being 700/580 = 1.2. This circuit contains 50,513 transistors.

Note from the figure that the SRAM array occupies most of the area and the overhead from the support circuitry is very little. Simulation results for the power dissipation and delay characteristics are discussed in Chapter 6.

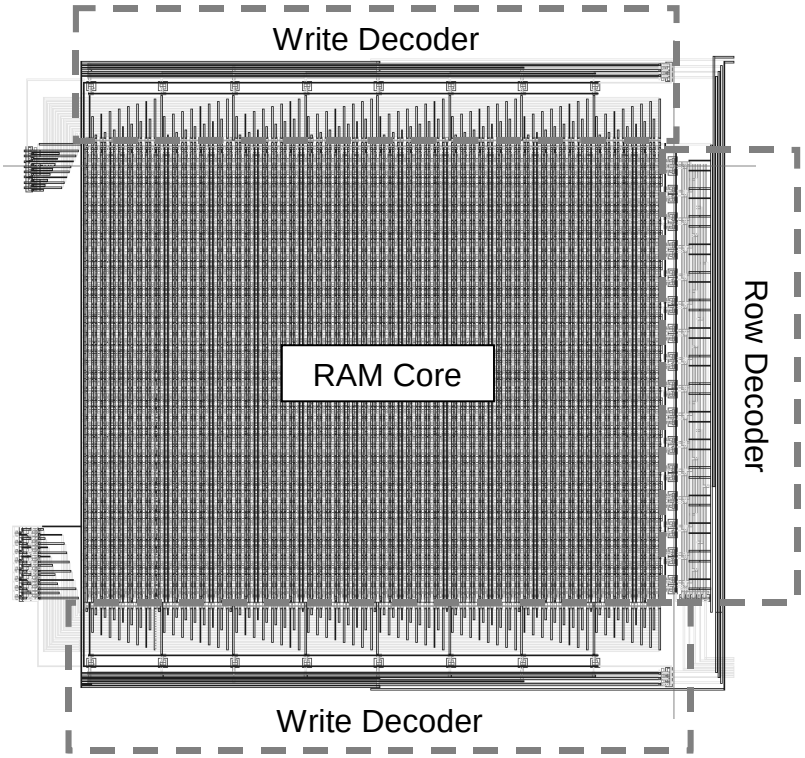


Figure 4.12 – Layout for a 256x8 SRAM

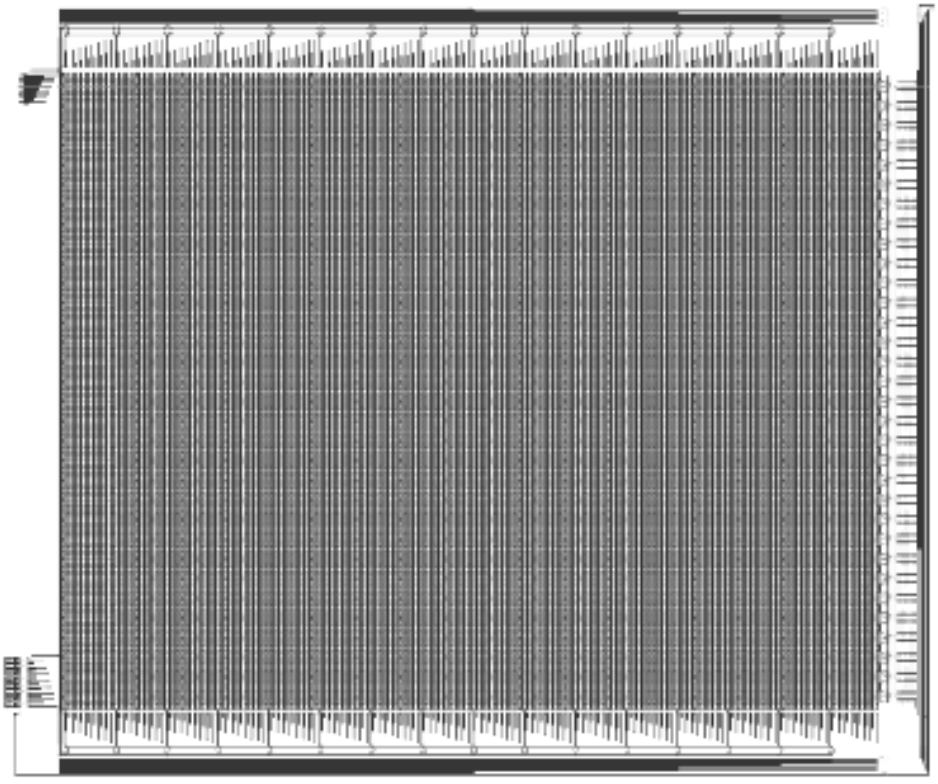


Figure 4.13 – Layout for a 1-kB SRAM

CHAPTER

5 Array Partitioning

With the increased use of portable consumer electronic products, power consumption becomes a critical design criterion. This requires engineers to optimize their design not only for speed and area, but also for power. In order to reduce the power dissipation, we incorporate the array partitioning technique proposed by J. Caravella, as mentioned in Chapter 2. The technique is applied to the architecture and modified our SKILL code to generate a partitioned SRAM. In this chapter, we discuss the structure and SKILL code for array-partitioned SRAMs.

5.1 Preliminary

The total power dissipated in a circuit is the sum of static and dynamic power dissipation. The dominant term is the dynamic power dissipation for capacitor charging and discharging. Thus, power consumption for static CMOS logic can be approximated as $P = \alpha \times CV^2 f$, where α is the average signal activity, C is the load and parasitic capacitance, V is the supply voltage, and f is the operating frequency of the circuit. For the case of the SRAM, a major portion of dynamic power dissipation is due to the load and parasitic capacitances, the bit-lines and the word-lines of the SRAM. These lines tend to be long and are switch most often.

The array partitioning technique aims to reduce the power dissipation by reducing the bit-line and word-line capacitances, which are charged/discharged whenever a cell is accessed. As mentioned in section 2.3.3, the technique partitions the memory array into blocks so that only one block is activated at any time. The array partition requires extra circuitry, hence it is slower compared with non-partitioned SRAM array.

For the ease of incorporating this technique into our existing SKILL code, we adopt array partitioning into our final design. The following sections discuss the details of the array partitioning regarding the structure and the implementation.

5.2 Design for Array Partitioning

We partition our array into four blocks, which produces a symmetrical design for easy implementation. Each of the blocks constitutes a separate SRAM circuit that is one-fourth the total size. A 2x4 decoder is used to select one block. The outputs of the block-selector has twelve control signals – the three control signals, OEN, WEN, and CEN, for each of the four blocks. The structure of the decoders is as follows.

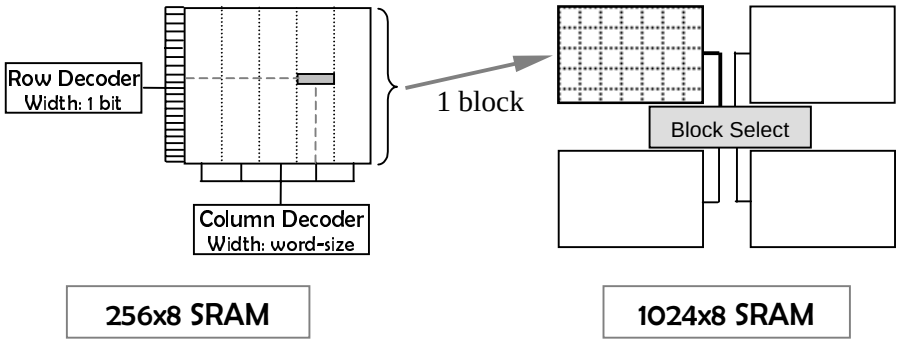


Figure 5.1 – Array partitioned Architecture

The block-selector is implemented the same way as the decoder implementation for the rows and the columns. The transistor level schematic for the block-selector is given in Figure 5.2.

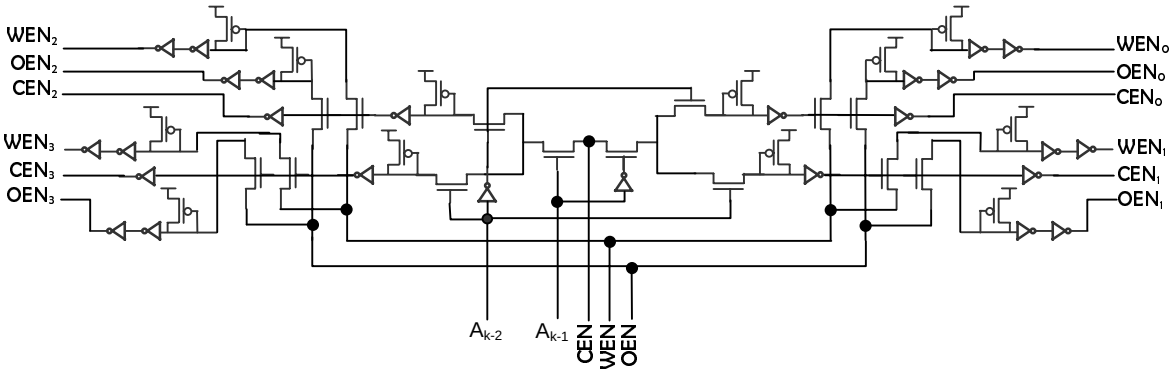


Figure 5.2 – Schematic of Block Select

The block-selector decodes the CEN signal based on the two most significant address bits. Thus, if the chip is being accessed, one of the four CEN signals, CEN_i (where i is the block being accessed), is enabled. As is the case for the row and column

decoders (Section 3.5), pull-up buffers are needed at the decoder outputs so that all unselected lines are disabled.

When the CEN signal is disabled for a block (which is not selected), it is desirable to disable OEN and WEN signals of the block to save power. Hence, we use the decoded CEN_i signal to enable/disable the OEN and WEN signals at the output of the block-selector. Note that, since the CEN is an active-low signal, the output is inverted before being used to switch pass transistors, in Figure 5.2.

Suppose that we read a data from block 0. The two MSB address bits are both 0, CEN and OEN signals for block 0 are enabled (pulled low), while WEN is disabled (pulled high). The 2x4 decoder connects the CEN signal to block 0. Since the CEN_0 is enabled, the two pass transistors associated with CEN_0 connects OEN and WEN signals to OEN_0 and WEN_0 signals. For all the other three blocks, the pull-up buffers pull up the control signals to be disabled.

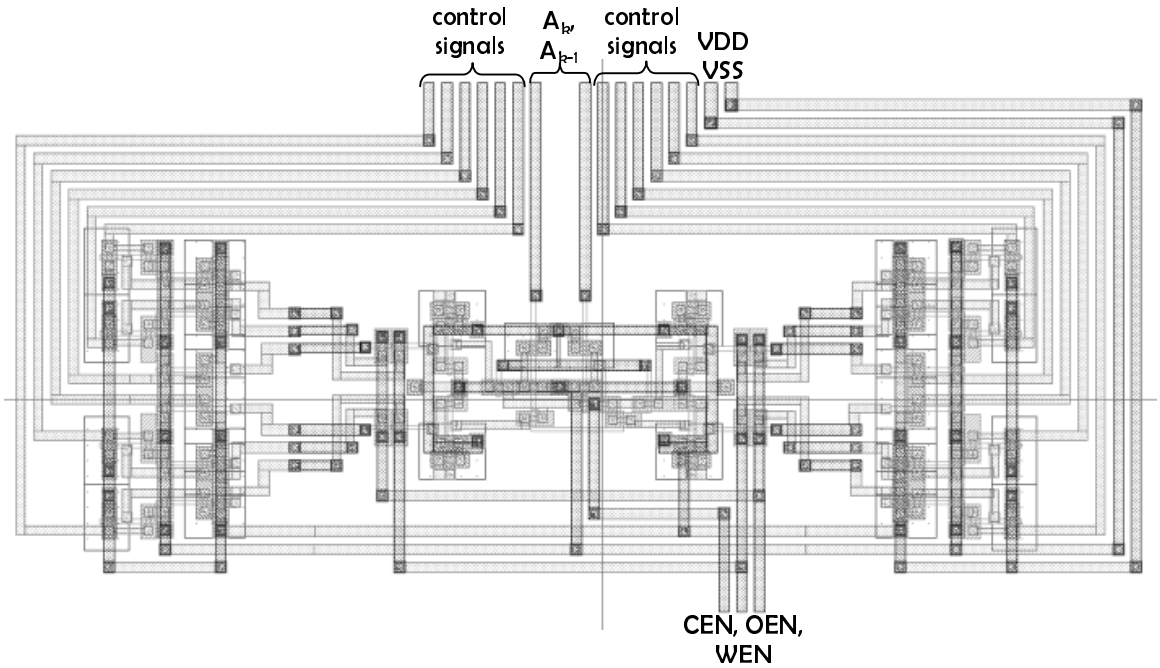


Figure 5.3 – Block Select Layout

The block-selector circuit is laid out to create a leaf-cell to be used by the SKILL code. The block-selector layout is shown in Figure 5.3. The three control signals are fed at the bottom of the block-selector. The two address signals are routed to the top of the SRAM circuit, so that they are bundled with the other address bits. The block select

outputs the twelve control signals, which are routed to their respective SRAM blocks. The dimension of the block-selector is 89 μm x 37 μm. In order to make the layout compact, the block select is placed at the bottom of an SRAM. The following section discusses the skill code implementation.

5.3 SKILL Code for Array Partition

The skill code for the array partition makes use of the *sram_array* function, which generates an unpartitioned SRAM array (Refer to Section 4.1). The structure of the modified SRAM compiler is shown in Figure 5.4

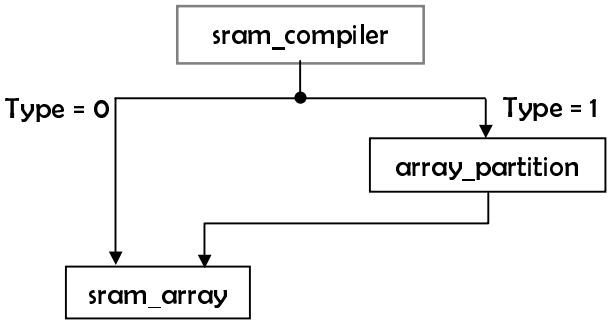


Figure 5.4 – Overall Structure of Sram_Compiler

The SRAM compiler allows users to choose between the two types of SRAMs– single array SRAM or array-partitioned SRAM. As indicated in Figure 5.4, if the user specifies type 0 (or type 1), a single-array SRAM (or array-partitioned SRAM) is generated. The pseudo-code for the top-most function, *sram_compiler*, is as follows. The function is responsible for differentiating between the two types.

- Load all necessary functions

```
// Check which type of SRAM the user wants to generate
if (Type == 0  // simple SRAM array
 // Generate simple SRAM array
 sram_array(library cellview words wordsize)
else if (Type == 1   // array-partitioned circuit
 // Generate circuit for 1 block
 sram_array(library temp_cellview words/4 wordsize)
 //call function to layout routing and blocks
 array_partition(library cellview words wordsize)
 dbDeleteObj(ddGetObj(library temp_cellview)) //delete temp layout
)
```

Thus, for type 1 SRAM, a temporary cellview for an SRAM generates one-fourth the size first. Next, the *array_partition* function is called to place blocks and necessary routings. The pseudo-code for the *array-partition* function is given below.

```
////////////// Instantiate 4 blocks  //////////////
Instantiate("temp_cellview" @ {x_offset, y_offset})
Instantiate("temp_cellview" @ {x_offset, -y_offset})
Instantiate("temp_cellview" @ {-x_offset, y_offset})
Instantiate("temp_cellview" @ {-x_offset, -y_offset})

/////////////// Route Data Lines  /////////////////
```

- Connect Data-In lines of all blocks together
- Connect Data-Out lines of all blocks together
- Route out Data signals to the top of the circuit
- Place a Pin for the PI Data-In and Data-Out signals

```
////////////// Route Control Signals //////////////
Instantiate("Block-Select" @ {bottom of the circuit})
```

- Route PI control signals (CEN', WEN', OEN') to the block select
- Route control signal from Block select to respective blocks
- Place a Pin for the PI control signals

```
////////////// Route Address Signals //////////////
```

- Connect Address lines of all blocks together
- Route out Address signals to the top of the circuit
- Route the two-most-significant address bits from the PI to the block-select
- Place a Pin for all Address signals

```
///////////  Make VDD & GND connections  ///////////
```

5.4 Final Layout

The final layout for a partitioned array of 1 kB (1024x8) SRAM is given in Figure 5.5. The SRAM is 860 μm wide by 730 μm high, with the aspect ratio being 860/730 = 1.2. The RAM contains 52,157 transistors. Though this is a 35% increase in silicon area when compared with a single array SRAM (Refer to Section 4.2), there is only a 3.15% increase in the number of transistors. This discrepancy is due to the overhead of routing

associated with the block-selector. In the next chapter, we discuss the simulation results for the power and delay characteristics of the two types of SRAMs.

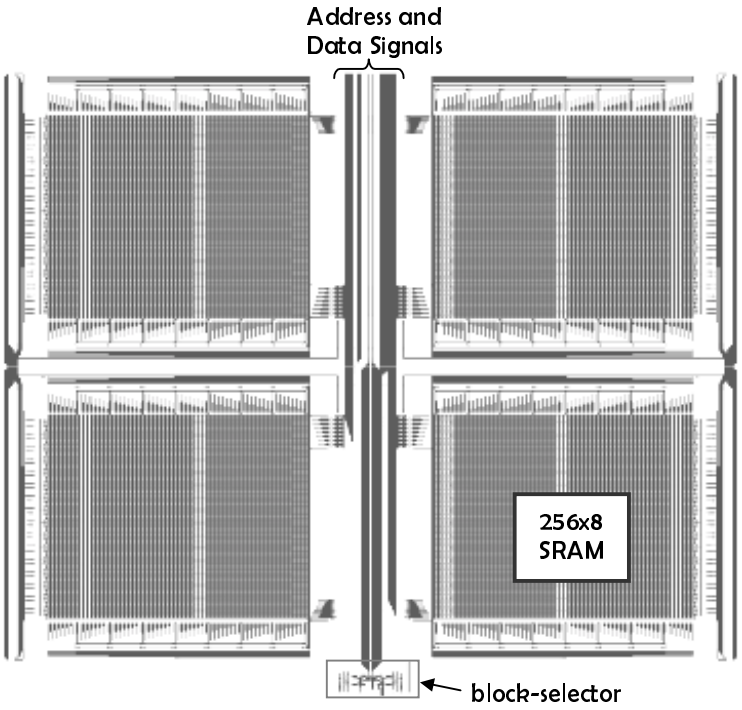


Figure 5.5 – Array-Partitioned 1 kB SRAM

CHAPTER

6 Simulation Results

So far, we discussed the implementation of our SRAM compiler and the basic components. The SRAM compiler enables a user to choose between two types of SRAMs – a fast vs. a low power version. In this chapter, we present the simulation results on the performance of the two types of SRAMs for three different sizes.

6.1 Simulation Environment

In addition to verifying the correct operation of SRAMs generated by our compiler, we measured the performance of SRAMs for different sizes, 256x8, 512x8, and (1024x8) 1 kB SRAM. It should be noted that 1 kB is the largest SRAM size required for the project. We measured the performance in:

- **Area:** Silicon Area, Transistor Count
- **Time:** Cycle, Access, Setup, Hold
- **Power:** Static, Dynamic, Average

After the layout was generated, Cadence's Analog Artist was used to extract the spice netlist. Input stimuli were manually added to simulate the circuit for different test cases and measure parameters. As was done for leaf-cells, Avanti HSPICE was used for SPICE simulation.

We performed two writes followed by two reads on two locations. Two farthest cells from the address pins were selected as the propagation delay and the dynamic power dissipation would be the worst on those cells. The data background (Data) used in the simulation for the 8-bit word SRAMs is 00110011 (x33), with the complemented data background (Data') being 11001100 (xCC). This allows for the most number of data changes. The timing of the simulation is shown in Figure 6.1. In the figure, locations 000 (hex) and 3FFF (hex) denote the addresses of first and the last cells, respectively.

The simulation was performed for 120 ns which includes 20 ns for two consequetive write operations, another 40 ns for two read operations, and 20 ns standby mode at the end of simulation. The period of an operation is set to 20 ns (50 MHz) in the simulation, which is based on the slowest SRAM, 1 kB partitioned-array RAM. A load

capacitance, C_{load} of 20fF is attached at each data output for the simulation. The following sections present the results obtained and discuss the trends for each of the three design parameters.

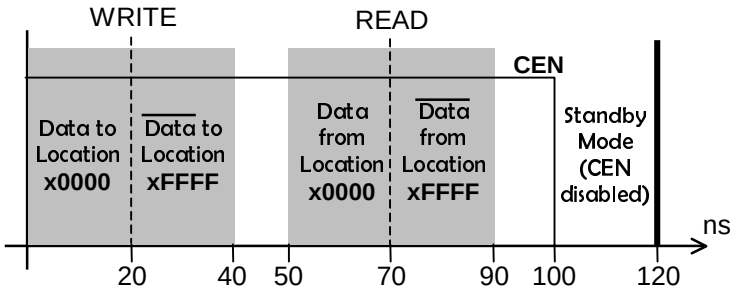


Figure 6.1 – Input Stimuli for Characterization

The waveform in Figure 6.2 shows the write and read operation for a 1kB partitioned-array SRAM. The DO in the waveform represents a data-output bit. During the writes, the voltages on the two bit-lines are affected by the input-data. During a read, voltages on the two bit-lines are pulled up/down by the data contained in the RAM cell. Note that, although the voltages on the bit-lines are not at a sufficiently high or low value, the data output, DO, is pulled to a good value by the sense amplifier. After the two reads, the CEN signal is disabled. It disables the SRAM and the voltages of the two bit-lines converge at this point.

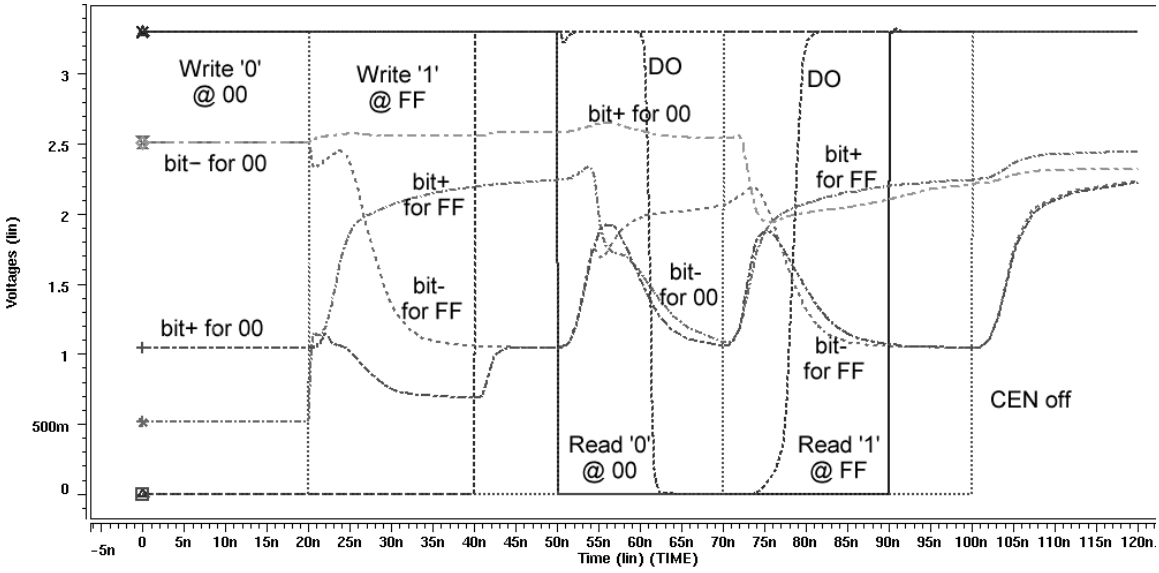


Figure 6.2 – Simulation Waveform for 1-kB SRAM

6.2 Area Measurement

We measured both the silicon area based on the layout and the total number of transistors for both type of SRAMs for the three different sizes. Table 6.1 gives the results on the area. In the table, the column headings "single" and "partition" denote single-array RAMs and partitioned-array SRAMs, respectively. The "Ratio" specified in the table is the area or transistor count of a partitioned-array SRAM to that of a single-array SRAM.

Table 6.1 – Area Characteristics

	256x8		512x8		1024x8	
	Single	Partition	Single	Partition	Single	Partition
Area (μm^2)	134	259	222	396	406	606
Ratio	1.93		1.78		1.49	
Transistor #	13019	14021	25469	26637	50513	52157
Ratio	1.08		1.04		1.03	
Aspect ratio	1.2	1.2	0.7	0.8	1.2	1.2
Ratio	1		1.1		1	

As the size of the SRAM is doubled from 256 byte to 512 byte and finally to 1 kB, the area approximately (*1.7) doubles. Likewise, doubling the SRAM size also approximately (*1.8) doubles the transistor count. This is true for both types of SRAMs. This result is expected as doubling the SRAM size implies that there will be twice as much RAM cells. Since RAM cells dominate both the area and the transistor count, the increase in size is proportional to the RAM cell increase. As the overhead of supporting circuitry will decrease with increasing size, this trend is expected to continue so that the increase in both the transistor count and the area will be further closer to two for larger SRAMs.

The overhead of the additional circuitry for the array partitioned SRAM results in increased area over the single-partition SRAM. Note that, for the 1 kB SRAM, though the transistor count only increases by 1.03 for the 1 kB SRAM, there is a 1.49 increase for the overall area for the layout. The small increase in transistor count results in a large

increase in layout. The reason for such a difference between layout-area and transistor count is increased routing to and from the four blocks. This illustrates the impact of routing in the final design.

Also included in the table is the aspect ratio (width/height) of the layout. Note that the aspect ratio decreases to 0.7 for the 512x8 SRAM. This change in aspect ratio is due to the method in which the aspect ratio is calculated. In the aspect ratio calculation, the number of rows and columns are calculated using the aspect ratio for one block. Also, the limitation of the number of rows and columns having to be a power of 2 limits the accuracy of the aspect ratio calculation. This results in less accurate aspect ratio because of the block size.

6.3 Time Measurement

The speed of SRAM cells and the propagation delay to access a certain cell attributes the access time for read or write operations. First, we measured the speed of a 6T SRAM cell core, with sense amplifiers and write-select, described in Section 3.4 for read and write operations. Table 6.2 presents the results for these operations.

Table 6.2 – Speed of a Single RAM Cell

Operation	Speed (ns)
Write 1	2.2
Read 1	0.93
Write 0	2.1
Read 0	0.7

Observations from the table show that the write-operation takes longer than the read operation. This is because for a write, the data has to first be inverted to provide both the complement and uncomplemented value that are fed to the bit-lines. Whereas for the read, as soon as the bit-lines start to be pulled by the RAM cell, the fast sense amp amplifies the difference, allowing the output to appear quickly. The same trend can be found below (see Table 6.4) for the 1 kB SRAMs. Another point to note is that it takes longer to read or write the logical value ‘1’, rather than ‘0’. The reason for this is because all pass-transistors use NFETs rather than PFETs, and since NFETs cannot

transmit a good ‘1’, it takes them longer to pullup a line. Though, the bit-lines are conditioned to alleviate this problem, there is still a slight bias towards ‘0’.

Let us now analyze detailed timing parameters that are helpful to understand the speed of SRAMs. The timing diagram of a read operation is given in Figure 6.3. The parameter read-cycle time, t_{RC}, indicates the minimum time that the address has to be valid in order for a valid data to be output sometime in the future. The address access time, t_{AA}, is the time from the start of a valid address to when valid-data is available at the output. This time includes both latency (the overhead of preparing to access it) and transfer time. Note that the read cycle time indicates a minimum, while the address access time is a maximum. For this reason, t_{RC} is usually less than t_{AA}. The output enable time, t_{OE}, represents the time that it takes for the data to appear on the output after the OEN signal is enabled.

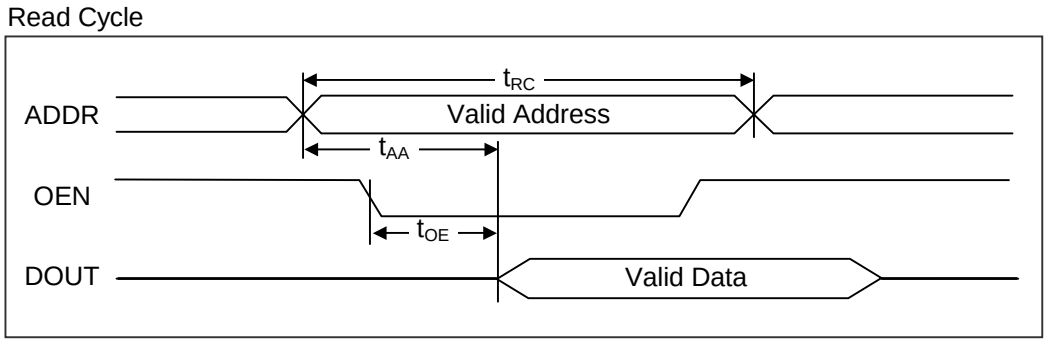


Figure 6.3 – Timing Parameters of a Read Cycle

Timing parameters related to write operations is shown in Figure 6.4. The write cycle time represents the minimal time from the start of an access to the time when the next access can be started. The write enable access time, t_{WA}, is the time it takes the data to be written to the RAM cell after the address has been setup. Likewise, the data-in access time, t_{DA}, represents the time it takes to write the data after a change in the input data. The address setup time, t_{AS}, gives the time that a valid write address must be present before WEN is enabled. The address hold time, t_{AH}, represents the time that the current address should be valid after WEN is disabled. Similarly, the data-in setup time, t_{DW}, specifies the time that a valid data must be available prior to disabling WEN, while the data-in hold time, t_{DH}, specifies the time for which the current data is held even after WEN is disabled.

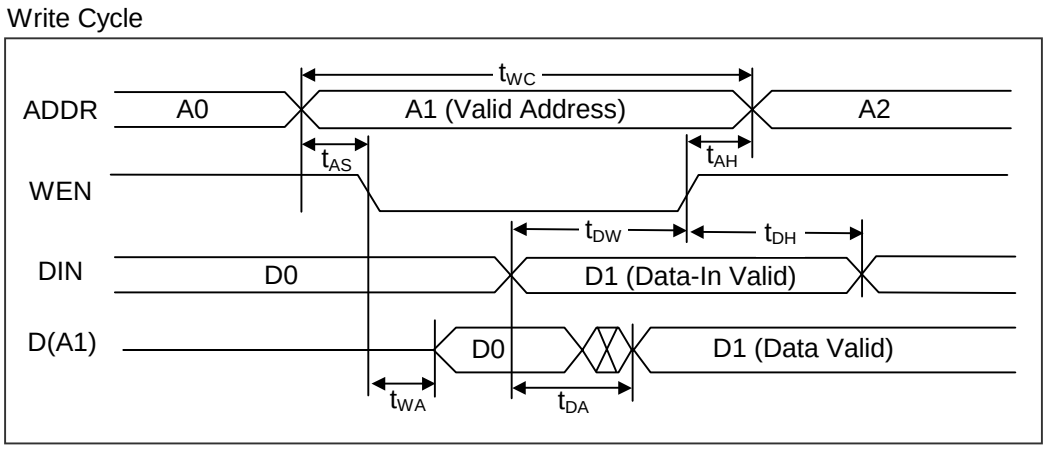


Figure 6.4 – Timing Parameters for a Write Cycle

The most critical timing parameter is the read access time during a read-operation, which determines the clock speed of an SRAM. Table 6.3 contains the access time for both types of SRAMs for the different sizes and types. This access time is the time it takes for the data to be output once the address is setup.

Table 6.3 – Comparison of Address-Access Times (ns)

	256x8		512x8		1024x8	
	Single	**Partition**	**Single**	**Partition**	**Single**	**Partition**
Access time (ns)	6.6	14.7	8.9	17.5	15.0	21.8
Ratio	2.22		1.96		1.45	

As expected, the results from the above table indicate that partitioned-array SRAM is slower than the single-array SRAM. However, the ratio for the different sizes indicates that as the SRAM size is increased, the speed interval between the two SRAM types decrease. The reduced speed for the array-partitioned SRAM is due mainly to the overhead of the supporting circuitry such as the block-selector and routing to & from the four blocks. For example, during a read, the sense amplifier has to drive a longer data-bus, with increased line capacitance, causing the data to appear slower on the output. However, since the overhead of supporting circuitry decreases with increased size, the ratio decreases as the size is increased.

Now we look at the timing characteristics for the 1-kB SRAM. Table 6.4 gives the results obtained for the 1-kB SRAM from the worst-case simulations, mentioned in section 6.1.

Table 6.4 – Timing Parameters for 1-kB SRAM

Symbol	Parameter	tpd (ns)		% Increase
		Type = 0	Type = 1	
Read Cycle				
t_{RC}	Read Cycle Time	12.7	18.9	1.49
t_{AA}	Address Access Time	15.0	21.8	1.45
t_{OE}	Output Enable Time	9.4	19.7	2.09
Write Cycle				
t_{WC}	Write Cycle Time	8.3	12.5	1.51
t_{WA}	Write Enable Access Time	4.7	6.3	1.34
t_{DA}	Data-In Access Time	2.1	4.4	2.09
t_{AS}	Address Setup Time	2.6	6.2	2.38
t_{AH}	Address Hold Time	0.3	0.4	1.33
t_{DW}	Data-In Setup Time	4.3	9.1	2.12
t_{DH}	Data-In Hold Time	0.1	0.1	1

As can be seen from the table results, the array-partitioned SRAM is about 1.5 times slower than a single-array SRAM in read time. All timing parameters are measured as the time it takes for the output to reach 90% of its final value. Though the operating period is 20 ns, we are able to obtain the value for the address access time of 21.8 ns for the array-partitioned SRAM because the data-output is held on the data bus for some time even after the enable signal is turned off, due to the line capacitance. The t_{AA} parameter determines the speed of a SRAM.

If we compare the percentage increase of the setup time over the hold time, it can be seen that the low-power SRAM takes much more time to setup over the normal SRAM. This is because the setup time includes the time it takes for both the address decoder and the block-select to decode the new address. In addition to this, there is also the time it takes for the control signals to reach the blocks. Since these lines tend to be long, the line capacitance can be large, leading to the slower time.

6.4 Power Measurement

Dynamic power dissipation occurs during a R/W access. Static power dissipation is the power dissipated when there are no read or write operations and all nodes are at the

steady state value. The average power dissipation is the power dissipated during the entire simulation, which includes the standby mode of 20 ns at the end of simulation.

Table 6.5 shows the power dissipation for three different sizes of SRAMs.

Table 6.5 – Power Characteristics

	256x8		512x8		1024x8	
	Single	**Partition**	**Single**	**Partition**	**Single**	**Partition**
Dynamic (mW)	31.11	25.86	61.15	30.00	79.21	41.54
Ratio	0.83		0.49		0.52	
Static (mW)	0.65	3.46	0.73	3.62	0.96	3.57
Ratio	5.32		4.95		3.72	
Average (mW)	24.68	21.39	48.08	24.16	66.59	36.83
Ratio	0.87		0.50		0.55	

As expected, the circuit with array partitioning reduces both dynamic and average power dissipated. For the 1 kB SRAM, the array-partitioned SRAM dissipates 45% less average power dissipation. The dynamic power dissipation reduces due to the reduced bit-line and word-line capacitances and consequently the average power dissipation is reduced, since it is dominated by the dynamic power dissipated.

Though both dynamic and average power dissipation is reduced, note that the static power dissipation actually increases with 3.72 times for the 1024x8 SRAM. This is because the static power dissipated is determined by the overhead of the support circuitry, especially the ones that contain a resistive load. Though both types of circuits have the same number of RAM cells, the partitioned SRAM has 4 times the support circuitry for the four different blocks. For example, for a 1 kB circuit, there are eight sense-amplifiers (one for each data bit) for the type 0 circuit. On the other hand, for the partitioned SRAM, since there are four independent SRAM blocks, there are 32 sense amplifiers. Therefore, the overhead is the cause of the increased static power dissipation. However, from Table 6.1, it can be seen that there is a decreasing trend with the percentage increase, so that the effect of the overhead will decrease with increased SRAM size. Also, note that the average power dissipation is dominated by the dynamic power, allowing us to ignore the effect of static power.

There is yet another interesting trend to be noted. We may expect that the average power savings will increase linearly as the SRAM size is increased. However, notice that there is a non-linearity for the 512x8 SRAM size where there is actually more power savings at the 512x8 SRAM than the 1 kB SRAM. The reason for this can be seen in the shape of the SRAM shown in Figure 6.5.

Notice that the blocks of the 512-size circuit are more elongated than the 1 kB SRAM. This means that for the 512x8, there are more rows than columns, whereas in the 1-kB SRAM, there are more columns than rows. This leads to the word-length being proportionally much longer in the 1 kB when compared to the bit-line length. Since the word-lines use *polysilicon*, while the bit-lines use a lower-resistive *metal1* layer, this puts the 512x8 circuit at an advantage, leading to the slightly higher power-savings. It should be noted that this trend is repeated for every quadrupled-SRAM (0.5 kB, 2 kB, 8 kB,…) due to the aspect ratio calculation. Therefore, the fault lies in the aspect ratio calculation where we assumed that an SRAM that has close to equal rows and columns is most desirable.

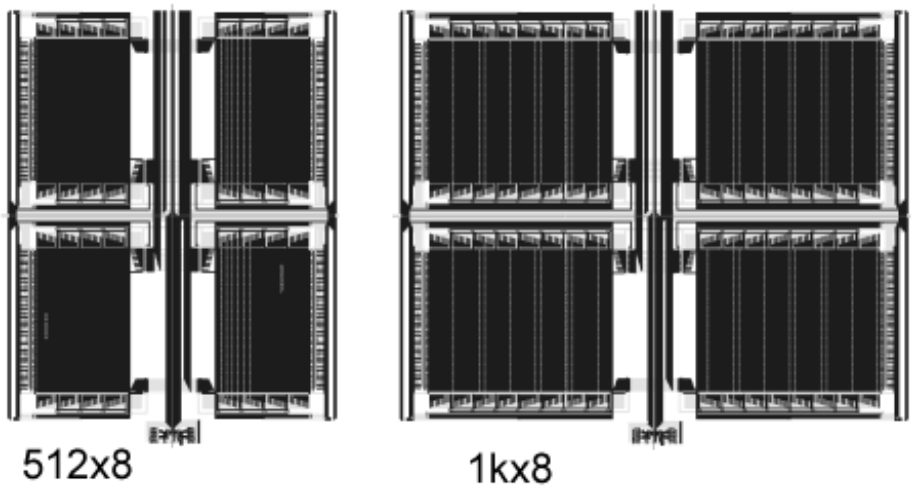


Figure 6.5 – Aspect Ratio Comparison for Array-Partitioned SRAM

Nevertheless, the static power dissipated is not affected by the SRAM and the results indicate a linear change. This is because the overhead of the support circuitry is not affected by the length of the bit or the word-lines. This allows for the linear trend in percentage savings. Given more time, the optimum bit-line to word-line length ratio should be determined and the aspect ratio calculation should be improved to take advantage of this phenomenon.

6.5 Minimum Operating Voltage

For low-power purposes, it is desirable to operate a circuit at the minimum possible operating voltage without exceeding system requirements. With the 20 MHz timing requirement for the SRAM, the minimum operating voltage is 1.9 V for the single-array SRAM, and 2.1 V for the partitioned-array SRAM. Table 6.6 specifies the performance of the SRAMs at the minimum operating voltage.

Table 6.6 – Performance at Min Operating Voltage

	Single-Array	**Partition-Array**	**Ratio**
Min. Operating Voltage	1.9 V	2.1 V	1.1
Address Access, t_{AA}	44.3 ns	48.1 ns	1.1
Average Power	22.65 mW	17.39 mW	0.7
Power Savings by reduced voltage	0.37	0.47	1.27

The reason for the difference in the minimum voltage between the two SRAMs is due to the restriction of speed. We want the SRAM to be operational for a frequrency of 20 MHz. However, note from the last section that the single-array SRAM is faster than the partitioned-array. This allows the single-array SRAM to have a reduced V_{dd} without reducing speed as much as the partitioned-array SRAM.

The power savings resulting from reducing V_{dd} comes at a cost of reduced speed. The equation for the delay,

$$\tau \approx \frac{C_{bitline}}{K'\left(\frac{W}{L}\right)(V_{dd} - V_t)^2} \cdot \Delta V ,$$

shows that reducing the V_{dd} slows down the circuit quadratically. There is a reduction in speed by 2.5 times due to reducing the operating voltage. However, the SRAM is capable of functioning within the 20 MHz required by the project.

By reducing the operating voltage, the speed was compromised by 60%. However, the power savings was close to 50% for the partition-array SRAM making the tradeoff reasonable. An important trend to note from the results is that at 3.3 V, the ratio of average power dissipated between the two types was 0.55. When the operating voltage

was reduced to minimum V_{dd}, the ratio increased to 0.7. This means that the difference in power dissipated between the two types reduces with decreased supply voltage. The reason for this can be found in the basic equation for power dissipation: $P = \alpha \times CV^2 f$. At 3.3 V, the main difference between the two types of SRAMs was the reduction in the capacitance, *C*. However, when the supply voltage was reduced to 2.1 V, the most dramatic change in the power dissipated is *V*, since its an quadratic term. Therefore, the 1.2 V drop in supply voltage dominates the total power dissipated. However, since *C* is still less for the array-partitioned SRAM, there is still a 30% power savings by using the array-partitioned SRAM.

6.6 Conclusion

The array-partitioned circuit proved to save power over the normal SRAM. However, this savings comes at the price of speed and area. For the 1-kB SRAM, the type 1 SRAM is proven to save 48% dynamic power and 45% overall power dissipation. However, the access time for the low-power circuit reduces to 21.8 ns – 31% slower than the 15 ns type 0 SRAM. Also there is an increased area of 33% and an increase of 3% in transistor count. By reducing the supply voltage to 2.1 V, the partitioned array was able to lower average power dissipation to 17.39 mW at a cost of reducing the speed to 20 MHz. These results give an account of the design tradeoffs involved with low-power circuits.

CHAPTER

7 Conclusion

An embedded SRAM compiler has been successfully developed with low-power capabilities. The compiler allows the user to choose between two types of SRAMs – one that is low power and one that is fast. This gives the user the ability to decide on the most critical design criteria for the application.

The low-power SRAM uses the array partitioning technique to reduce power dissipation. By dividing the entire memory array into four blocks, we are able to reduce the bit-line and word-line capacitance by half. Thus, the partitioned memory arrays reduce the total capacitance that is switched per access. Reducing these capacitances reduces the dynamic power dissipated and consequently, the total power that is dissipated.

Simulation results for the 1kB SRAM show that the low-power SRAM dissipates 45% less power than the normal SRAM, with the low-power SRAM dissipating 36.83 mW of average power. The area overhead due to array partitioning is 33%, with a 3% increase in the number of transistors. For a size of 1kB, both types of SRAM are shown to be capable of operating at a frequency of 50 MHz, well within the 20 MHz requirements for this thesis. At the minimum operating voltage of 2.1 V, the array-partitioned SRAM dissipated 20 mW of average power, operating at a speed of 20 MHz.

Finally, a test circuit has been prepared which will be fabricated. The layout for the test circuit is shown in Figure 7.1. The layout shows the 1 kB array-partioned SRAM (type 1) with I/O pads connected. The test circuit will be used to physically verify the operation and get actual measurements of the SRAM. After this verification, the SRAM will be embedded in the Wireless Video Project mentioned in Section 2.5. For the test circuit, the SRAM is not embedded and requires a pin for each I/O pins. Due to the high cost of I/O pads, the data-input and data-output signals are connected together to reduce pin number. Because the data-output signal is isolated from the bus by transmission gates, there is no reason for a bus contention.

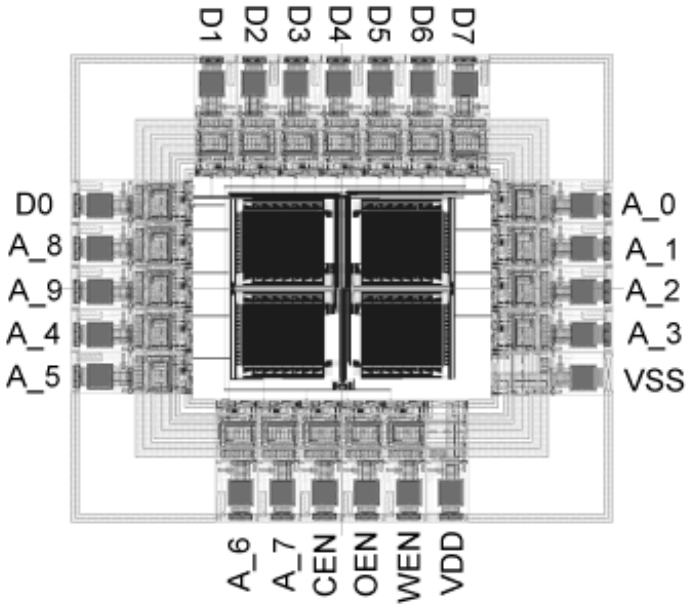


Figure 7.1 – Test Circuit for 1 kB Array-Partitioned SRAM

The area for the test circuit is 2080 μm x 1795 μm = 3.73 mm^2, which is approximately six times larger than just the SRAM. As can be seen in the figure, the I/O pads occupy a large part of the total area, even after reducing the pins by sharing the data-in and data-out signals. However, since the SRAM will be embedded in the final circuit, the effect of the I/O pads on the area will not be as dramatic.

As a conclusion to this thesis, we describe a possible improvement to the design. As mentioned in the previous chapter, the aspect ratio of the SRAM plays an important factor in the final design. Though the aspect ratio calculation for the compiler was designed in order to accommodate equal rows and columns, the results indicate that this may not be the best choice. Because the word-lines, which are *polysilicon*, are more resistive than the bit-lines, which are *metal1*, it is preferable to make the word-lines shorter than the bit-lines. This implies that there should be more rows than columns. Then, the aspect ratio should be calculated not to have equal rows and columns, but an optimum row-column ratio. Therefore, the optimum aspect ratio should be determined through experimental results and the compiler should be modified to generate the layout for this aspect ratio.

APPENDIX

A Program Execution

The following tables lists the functions used in our SRAM compiler.

Table A.1 – Functions in the SRAM Compiler

sram_compiler(library cellview Words Wordsize Type) procedure sram_compiler will generate an embedded SRAM layout Possible Types: • Type = 0 -- Simple SRAM array without Array Partitioning • Type = 1 -- Array Partitioned SRAM array with the Block Select at bottom
array_partition(library cellview Words Wordsize) procedure array_partition will partition the memory array into 4 blocks for low-power
sram_array(library cellview words wordsize) procedure sram_array is the top-level function to layout an SRAM circuit
cell_layout(library cellview number_of_rows number_of_cols) procedure cell_layout layouts an array of sram cells with m rows and n columns
word_decoder(library cellview x y number_of_rows number_of_cols wordsize)
read_decoder(library cellview x y number_of_rows number_of_cols wordsize)
write_decoder(library cellview x y number_of_rows number_of_cols wordsize) These procedures layouts the decoders for the SRAM
package(library cellview x y number_of_rows number_of_cols wordsize) procedure package makes the circuit fit the final package → add all I/O pins and route signals to meet package criteria

A.1 Compiler Setup

Before generating the circuit, the compiler must first be setup in the CADENCE environment in the following way.

1. After starting CADENCE icfb, follow the procedure to setup the TSMC 0.35 um process from the following page:

   http://www.ee.vt.edu/ha/cadtools/cadence/unix_env.html

2. In your working directory, copy all .il files that are present in the present directory.

3. Copy the sramleaf/ directory onto your working directory.

4. Add the sramleaf/ directory as a library in your CADENCE design environment using the procedure from the following page:

http://www.ee.vt.edu/ha/cadtools/cadence/gate.html

A.2 Layout Generation

Each procedure is contained in a separate file whose filename is the name of the procedure. The compiler is executed in the following way.

1. Load skill code: **load("sram_compiler.il")**

   → Loads the contents of the file *sram_compiler.il*

   → This file also contains the commands to load all other functions that will be used by the *sram_compiler* procedure.

2. Call top-level function: **sram_compiler(library cellview words wordsize type)**

   → Generates an SRAM layout for the specified size of the specified type.

   → Example: *sram_compiler("ram" "sram_1k_8" 1024 8 1)* generates a layout for an 1 kB (1024×8) array-partitioned SRAM.

Note that the above commands should be typed in the CIW. Also, the load command assumes that the file is in the cadence working directory. If this is not the case, the correct path of the file should be entered. All SKILL code files are enclosed in Appendix B.

APPENDIX

B SKILL Code

The SRAM compiler consists of 8 SKILL code files, all of which have a *il* extension. The files are stored in the *VISC* workstations at the following location: /project/asic/SRAM_Compiler. The directory listing for this location is shown in Table B.1.

Table B.1 – Directory Listing of /project/asic/SRAM_Compiler

Filename	**Contents**
array_partition.il	Function *array_partition*
cell_layout.il	Function *cell_layout*
package.il	Function *package*
read_decoder.il	Function *read_decoder*
README_compiler	README for SRAM Compiler with instructions for compiler setup and execution
sram_array.il	Function *sram_array*
sram_compiler.il	Function *sram_compiler*
word_decoder.il	Function *word_decoder*
write_decoder.il	Function *write_decoder*
spice/	Directory of SRAM HSPICE files
sramleaf/	Directory of leaf-cell layouts
testcircuit_1kx8/	Directory containing 1 kB test-circuit cellview

As can be seen from the directory listing, each function is contained in a separate file whose filename is the name of the function. The following pages contain the SKILL code files in alphabetical order.

B.1 array_partition.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FileName: array_partition.il
;; Author:  Meenatchi Jagasivamani, April 2000
;;
;; procedure array_partition will partition the memory array into 4 blocks for low
;;         power purposes and place the Block Select at the bottom (compact)
;;
;; Usage In CIW:
;;        array_partition(library cellview Words Wordsize)
;;
;; Ex:  array_partition("sram" "block_256_8" 256 8)
;;   --> to create array of sram cells with 256 words with a wordsize of 8 bits
;;       array will be partitioned into 4 blocks with the BS at the bottom
;;   --> Layout will be stored in cellview "block_256_8" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(array_partition(clib cname words w)

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;; Declare leaf cells to be used ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  library = "sramleaf"
  viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")
  blkselcv = dbOpenCellViewByType(library "BlockSelect " "layout" "" "r")
  ccv  =  dbOpenCellViewByType(clib  cname  "layout"  "maskLayout"  "a")  ;  final  circuit
cellview

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;  Define Routing Variables ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  block = int(words/4)    ; block size
  AR_1bit = 0.5

  ;;;;;;Calculate row and col for equal aspect ratio
  ar = log(w* AR_1bit)/log(2)        ;aspect ratio for 1 wordsize block
  k = int(log(block)/log(2))         ;number of address lines
  y = floor((k-ar)/2)                ;x+y = k
  x = k-y                            ;x = ar+y

  m = int(2**x)
  n = (2**y)*w

  startx=4.8*n+2.5*w+19.25+2.85 ;address side
  endx=9.25+2.9*x+1.4*y ;data side

  ;;;; split address lines for symmetrical routing
  ; let right side have less address lines (because of VDD & GND routing)
  halfadr = ceiling(k/2)    ;MSB

  ;xwidth = (DI+DO)+Decoder Width+VDD&GND+address
   xwidth = (1.4*w)+1.4*(k-halfadr)+10.75+4.6

  ;yheight = address+3 control signals on both sides
 ; yheight = 1.4*(halfadr+2)+3.9-1.8
 yheight = 1.4*(k-halfadr)+4.7

  ;;;; Offset required to put the sram blocks on the axis
  xspacing = startx+5.55+0.85+xwidth
  yspacing = 10.5+14.8*(m/2)+2.7*w+2.9*y+yheight

  ;;; coordinates for one block
  top = yspacing+30.55+2.7*w+2.9*y-2.8
  bottom = yspacing-10.9-14.8*(m/2)-2.7*w-2.9*y

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
  ;;;;;;;;;;;;;generate circuit for 1 block = words/4 ;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ; create block in a temporary cellview called : "temp_" + cname
  blkcvname = buildString(list("temp" pcExprToString(cname)) "_")

  ;open temporary block cell view
  blockcv = dbOpenCellViewByType(clib blkcvname "layout" "" "r")

  Instpoint1 = xspacing:-yspacing
  blockInst = dbCreateInst(ccv blockcv "blockInst1" Instpoint1 "MX")
  dbFlattenInst(blockInst 1 t)

  ;instantiate blocks onto final cellview
  Instpoint2 = -xspacing:-yspacing
  blockInst = dbCreateInst(ccv blockcv "blockInst2" Instpoint2 "R180")
  dbFlattenInst(blockInst 1 t)

  Instpoint3 = xspacing:yspacing
  blockInst = dbCreateInst(ccv blockcv "blockInst3" Instpoint3 "R0")
  dbFlattenInst(blockInst 1 t)

  Instpoint4 = -xspacing:yspacing
  blockInst = dbCreateInst(ccv blockcv "blockInst4" Instpoint4 "MY")
  dbFlattenInst(blockInst 1 t)

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;  Route DATA Lines ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  for(bit 0 w-1
    ;connect write-data lines together
    dbCreateRect(ccv     "metal1"     list(xspacing-startx+2.85:yspacing+7.55-2.8*bit     -
xspacing+startx-2.85:yspacing+8.35-2.8*bit))
    dbCreateRect(ccv     "metal1"     list(xspacing-startx+2.85:-yspacing-7.55+2.8*bit     -
xspacing+startx-2.85:-yspacing-8.35+2.8*bit))

    ;connect read-data lines together
    dbCreateRect(ccv "metal1" list(xspacing-startx-7.2:yspacing+8.55-14.8*(m/2)+4.8*bit -
xspacing+startx+7.2:yspacing+9.35-14.8*(m/2)+4.8*bit))
    dbCreateRect(ccv  "metal1"  list(xspacing-startx-7.2:-yspacing-8.55+14.8*(m/2)-4.8*bit
-xspacing+startx+7.2:-yspacing-9.35+14.8*(m/2)-4.8*bit))

    ;route write-data out vertically
    ;place via for vertical routing at top
    viapt = xspacing-startx-8.2-1.4*bit:yspacing+8-2.8*bit
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
    ;place via for vertical routing at bottom
    viapt = xspacing-startx-8.2-1.4*bit:-yspacing-8+2.8*bit
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
    dbCreateRect(ccv    "metal2"    list(xspacing-startx-7.8-1.4*bit:-yspacing-7.6+2.8*bit
xspacing-startx-8.6-1.4*bit:top))

    ;Place a pin for Write Data signals
    fig  =  dbCreateRect(ccv  "metal2"  list(xspacing-startx-8.5-1.4*bit:top-0.1  xspacing-
startx-7.9-1.4*bit:top-0.7))
    pinname = buildString(list("DI" pcExprToString(bit)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)

    ;route read-data in vertically
    ;place via for vertical routing at top
    viapt = -xspacing+startx+8.2+1.4*bit:yspacing+8.95-14.8*(m/2)+4.8*bit
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
    ;place via for vertical routing at bottom
    viapt = -xspacing+startx+8.2+1.4*bit:-yspacing-8.95+14.8*(m/2)-4.8*bit
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
```

```
    dbCreateRect(ccv      "metal2"      list(-xspacing+startx+7.8+1.4*bit:-yspacing-7.75-0.8+14.8*(m/2)-4.8*bit -xspacing+startx+8.6+1.4*bit:top))

    ;Place a pin for Read Data signals
    fig   =   dbCreateRect(ccv   "metal2"   list(-xspacing+startx+8.5+1.4*bit:top-0.1   -xspacing+startx+7.9+1.4*bit:top-0.7))
    pinname = buildString(list("DO" pcExprToString(bit)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)
   )

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;  Route Control Signals ;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;place block select signal
 blkselpt = 2.1-1.05+2.25:-top-25
 blkselInst = dbCreateInst(ccv blkselcv "blkselInst" blkselpt "R0")
 dbFlattenInst(blkselInst 1 t)

 for(sig 0 2

   ;;;;;; Route control signals from PI to the decoder
   ;;; Order from left to right: CEN, OEN, WEN
   ;place pin at end
   fig = dbCreateRect(ccv  "metal2"  list(12.6+1.4*sig:-top-25-16.6+0.1  13.2+1.4*sig:-top-16.6-25+0.7))
   if(sig == 0
    then pinname="CEN"
   else if(sig== 1
    then pinname = "OEN"
    else pinname = "WEN"
    )
   )
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "input")
   pin = dbCreatePin(net fig pinname)

   ;;;;;; For Block 1 (x, y)
   ;route control signal from block select
   dbCreateRect(ccv "metal2" list(2.95+1.4*sig:-top 3.75+1.4*sig:3.9-1.4*sig))
   viapt = 3.35+1.4*sig:3.5-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)

   dbCreateRect(ccv                 "metal1"                 list(6.55-1.4*sig:0.3+1.4*sig xspacing+endx+1.4*k+1.4*sig:1.1+1.4*sig))
   ;place a via at the ends
   viapt = xspacing+endx+0.4+1.4*k+1.4*sig:0.7+1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv         "metal2"         list(xspacing+endx+1.4*k+1.4*sig:1.1+1.4*sig xspacing+endx+0.8+1.4*k+1.4*sig:top-1.4*sig))
   ;add via at the end and connect to line
   viapt = xspacing+endx+0.4+1.4*k+1.4*sig:top-0.4-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv         "metal1"         list(xspacing+endx+1.4*k+1.4*sig:top-1.4*sig xspacing+endx+3.9:top-0.8-1.4*sig))


   ;;;;;; For Block 2 (x, -y)
   ;route control signal from block select
   dbCreateRect(ccv "metal2" list(7.15+1.4*sig:-top 7.95+1.4*sig:-0.3-1.4*sig))
   viapt = 7.55+1.4*sig:-0.7-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)

    dbCreateRect(ccv                "metal1"                list(7.15+1.4*sig:-0.3-1.4*sig xspacing+endx+1.4*k+1.4*sig:-1.1-1.4*sig))
   ;place a via at the ends
```

```
   viapt = xspacing+endx+0.4+1.4*k+1.4*sig:-0.7-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv         "metal2"         list(xspacing+endx+1.4*k+1.4*sig:-1.1-1.4*sig
xspacing+endx+0.8+1.4*k+1.4*sig:-top+1.4*sig))
   ;add via at the end and connect to line
   viapt = xspacing+endx+0.4+1.4*k+1.4*sig:-top+0.4+1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv         "metal1"         list(xspacing+endx+1.4*k+1.4*sig:-top+1.4*sig
xspacing+endx+3.9:-top+0.8+1.4*sig))

   ;;;;;; For Block 3 (-x, y)
   ;route control signal from block select
   dbCreateRect(ccv "metal2" list(-2.95-1.4*sig:-top -3.75-1.4*sig:3.9-1.4*sig))
   viapt = -3.35-1.4*sig:3.5-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)

   dbCreateRect(ccv    "metal1"    list(-6.55+1.4*sig:0.3+1.4*sig    -xspacing-endx-1.4*k-
1.4*sig:1.1+1.4*sig))
   ;place a via at the ends
   viapt = -xspacing-endx-0.4-1.4*k-1.4*sig:0.7+1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv   "metal2"   list(-xspacing-endx-1.4*k-1.4*sig:1.1+1.4*sig   -xspacing-
endx-0.8-1.4*k-1.4*sig:top-1.4*sig))
   ;add via at the end and connect to line
   viapt = -xspacing-endx-0.4-1.4*k-1.4*sig:top-0.4-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv   "metal1"   list(-xspacing-endx-1.4*k-1.4*sig:top-1.4*sig   -xspacing-
endx-3.9:top-0.8-1.4*sig))

   ;;;;;; For Block 4 (-x, -y)
   ;route control signal from block select
   dbCreateRect(ccv "metal2" list(-7.15-1.4*sig:-top -7.95-1.4*sig:-0.3-1.4*sig))
   viapt = -7.55-1.4*sig:-0.7-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)

    dbCreateRect(ccv    "metal1"    list(-7.15-1.4*sig:-0.3-1.4*sig    -xspacing-endx-1.4*k-
1.4*sig:-1.1-1.4*sig))
   ;place a via at the ends
   viapt = -xspacing-endx-0.4-1.4*k-1.4*sig:-0.7-1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv   "metal2"   list(-xspacing-endx-1.4*k-1.4*sig:-1.1-1.4*sig   -xspacing-
endx-0.8-1.4*k-1.4*sig:-top+1.4*sig))
   ;add via at the end and connect to line
   viapt = -xspacing-endx-0.4-1.4*k-1.4*sig:-top+0.4+1.4*sig
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
   dbCreateRect(ccv   "metal1"   list(-xspacing-endx-1.4*k-1.4*sig:-top+1.4*sig   -xspacing-
endx-3.9:-top+0.8+1.4*sig))

 )

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;  Route Address Lines ;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  ;;;;;;;;;;;;connect col-address lines together
  for(addrline 0 y-1
    ;connect col lines together
    dbCreateRect(ccv    "metal2"    list(-xspacing-endx-1.4*addrline:bottom+3.45+2.9*(y-1)-
1.4*addrline -xspacing-endx-0.8-1.4*addrline:-bottom-3.45-2.9*(y-1)+1.4*addrline))
    dbCreateRect(ccv     "metal2"     list(xspacing+endx+1.4*addrline:-bottom-3.45-2.9*(y-
1)+1.4*addrline xspacing+endx+0.8+1.4*addrline:bottom+3.45+2.9*(y-1)-1.4*addrline))

    ;place a via at the ends
```

```
    viapt = -xspacing-endx-0.4-1.4*addrline:bottom+3.85+2.9*(y-1)-1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;place a via at the ends
    viapt = -xspacing-endx-0.4-1.4*addrline:-bottom-3.85-2.9*(y-1)+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t);

    ;place a via at the ends
    viapt = xspacing+endx+0.4+1.4*addrline:-bottom-3.85-2.9*(y-1)+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;place a via at the ends
    viapt = xspacing+endx+0.4+1.4*addrline:bottom+3.85+2.9*(y-1)-1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
  )

  ;;;;;;;;;;;;connect row-address lines together
  for(addrline 0 x-1
    ;connect row lines together
    dbCreateRect(ccv  "metal2"  list(-xspacing-endx-1.4*y-1.4*addrline:bottom+4.9+2.9*(y-
1)+1.4*addrline              -xspacing-endx-1.4*y-0.8-1.4*addrline:-bottom-4.9-2.9*(y-1)-
1.4*addrline))
    dbCreateRect(ccv  "metal2"  list(xspacing+endx+1.4*y+1.4*addrline:-bottom-4.9-2.9*(y-
1)-1.4*addrline xspacing+endx+1.4*y+0.8+1.4*addrline:bottom+4.9+2.9*(y-1)+1.4*addrline))

    ;place a via at the ends
    viapt = -xspacing-endx-0.4-1.4*addrline-1.4*y:bottom+5.3+2.9*(y-1)+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

   ;place a via at the ends
   viapt = -xspacing-endx-0.4-1.4*addrline-1.4*y:-bottom-5.3-2.9*(y-1)-1.4*addrline
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)

   ;place a via at the ends
   viapt = xspacing+endx+0.4+1.4*addrline+1.4*y:bottom+5.3+2.9*(y-1)+1.4*addrline
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t);

   ;place a via at the ends
   viapt = xspacing+endx+0.4+1.4*addrline+1.4*y:-bottom-5.3-2.9*(y-1)-1.4*addrline
   viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
   dbFlattenInst(viaInst 1 t)
 )

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 for(addrline 0 k-1
   if(addrline < halfadr
   then
     dbCreateRect(ccv           "metal1"           list(-xspacing-endx-0.4-1.4*addrline:-
bottom+1.4+1.4*addrline xspacing+endx+0.4+1.4*addrline:-bottom+0.6+1.4*addrline))

    ;place a via at the ends
    viapt = -xspacing-endx-0.4-1.4*addrline:-bottom+1+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;place a via at the ends
    viapt = xspacing+endx+0.4+1.4*addrline:-bottom+1+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;route to PI vertically
    viapt = 14.95+1.4*addrline:-bottom+1+1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
```

```
    dbCreateRect(ccv        "metal2"        list(14.55+1.4*addrline:-bottom+1+1.4*addrline
15.35+1.4*addrline:top))

    ;Place a pin for Address signals
    fig      =      dbCreateRect(ccv       "metal2"       list(14.65+1.4*addrline:top-0.1
15.25+1.4*addrline:top-0.7))
    pinname = buildString(list("A" pcExprToString(addrline)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)

   else
     dbCreateRect(ccv        "metal1"        list(-xspacing-endx-0.4-1.4*addrline:bottom-
1.4+1.4*halfadr-1.4*addrline        xspacing+endx+0.4+1.4*addrline:bottom-0.6+1.4*halfadr-
1.4*addrline))

    ;place a via at the ends
    viapt = -xspacing-endx-0.4-1.4*addrline:bottom-1+1.4*halfadr-1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;place a via at the ends
    viapt = xspacing+endx+0.4+1.4*addrline:bottom-1+1.4*halfadr-1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)

    ;route to PI vertically
    viapt = -11.75-1.4*(addrline-halfadr):bottom-1+1.4*halfadr-1.4*addrline
    viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
    dbFlattenInst(viaInst 1 t)
    dbCreateRect(ccv   "metal2"   list(-11.35-1.4*(addrline-halfadr):bottom-1+1.4*halfadr-
1.4*addrline -12.15-1.4*(addrline-halfadr):top))

    ;Place a pin for Address signals
    fig  =  dbCreateRect(ccv  "metal2"  list(-11.45-1.4*(addrline-halfadr):top-0.1  -12.05-
1.4*(addrline-halfadr):top-0.7))
    pinname = buildString(list("A" pcExprToString(addrline)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)
   )
 )


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;  Place pin for MSB 2 address bits ;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;; route out  address lines from block select
 dbCreateRect(ccv "metal2" list(1.55:-top 2.35:top))
 dbCreateRect(ccv "metal2" list(-2.35:-top -1.55:top))

 ;Place a pin for Ak
 fig = dbCreateRect(ccv "metal2" list(1.65:top-0.1 2.25:top-0.7))
 pinname = buildString(list("A" pcExprToString(k)) "_")
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)

 ;Place a pin for Ak+1 == MSB address bit
 fig = dbCreateRect(ccv "metal2" list(-1.65:top-0.1 -2.25:top-0.7))
 pinname = buildString(list("A" pcExprToString(k+1)) "_")
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;  Route VDD & GND Signals ;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 ;;;;;;;Route VDD & GND from PI
```

```
 ; VSS
 dbCreateRect(ccv "metal2" list(24-12.65:-top 25-12.65:top))
 ;create pin
 fig = dbCreateRect(ccv "metal2" list(24.2-12.65:top-0.2 24.8-12.65:top-0.8))
 pinname = "VSS!"
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)

 ; VDD
 dbCreateRect(ccv "metal2" list(25.6-12.65:-top 26.6-12.65:top))
 ;create pin
 fig = dbCreateRect(ccv "metal2" list(25.8-12.65:top-0.2 26.4-12.65:top-0.8))
 pinname = "VDD!"
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)


 ;route VDD lines together for blocks
 dbCreateRect(ccv    "metal1"    list(-xspacing+startx-9.9:-bottom-3.4-2.9*y    xspacing-
startx+9.9:-bottom-4.2-2.9*y))
 dbCreateRect(ccv    "metal1"     list(-xspacing+startx-9.9:bottom+3.4+2.9*y    xspacing-
startx+9.9:bottom+4.2+2.9*y))
 ;place via to connect to VDD PI
 viapt = 26.1-12.65:-bottom-3.8-2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)
 ;place via to connect to VDD PI
 viapt = 26.1-12.65:bottom+3.8+2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)

 ;route VSS lines together for blocks
 dbCreateRect(ccv   "metal1"   list(-xspacing+startx-13.1-2.5*w:-bottom-2-2.9*y   xspacing-
startx+13.1+2.5*w:-bottom-2.8-2.9*y))
 ;place via to connect to VSS line of block
 viapt = -xspacing+startx-13.1-2.5*w:-bottom-2.4-2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)
 ;place via to connect to VSS line of block
 viapt = xspacing-startx+13.1+2.5*w:-bottom-2.4-2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)

 dbCreateRect(ccv   "metal1"   list(-xspacing+startx-13.1-2.5*w:bottom+2+2.9*y   xspacing-
startx+13.1+2.5*w:bottom+2.8+2.9*y))
 ;place via to connect to VSS line of block
 viapt = -xspacing+startx-13.1-2.5*w:bottom+2.4+2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)
 ;place via to connect to VSS line of block
 viapt = xspacing-startx+13.1+2.5*w:bottom+2.4+2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)

 ;place via to connect to VSS PI
 viapt = 24.5-12.65:-bottom-2.4-2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)
 ;place via to connect to VSS PI
 viapt = 24.5-12.65:bottom+2.4+2.9*y
 viaInst = dbCreateInst(ccv viacv "viaInst" viapt "R0")
 dbFlattenInst(viaInst 1 t)

)
```

B.2 cell_layout.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FileName: cell_layout.il
;; Author: Meenatchi Jagasivamani, April 2000
;;
;; procedure cell_layout will layout an array of sram cells m rows and n columns
;;
;; Usage In CIW:
;;     cell_layout(library cellview number_of_rows number_of_cols)
;;
;;
;;  Ex:    cell_layout("sram" "sram_32x64" 32 64)
;;           --> to create array of sram cells with 32 rows and 64 columns
;;           --> Layout will be stored in cellview "sram_32x64" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

procedure(cell_layout(clib cname m n)

 ; Leaf-Cell library
 library = "sramleaf"

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;; Open necessary leaf-cells ;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 scv = dbOpenCellViewByType(library "sram_6t " "layout" "" "r")
 pcv = dbOpenCellViewByType(library "precharge" "layout" "" "r")
 buffcv = dbOpenCellViewByType(library "write_select" "layout" "" "r")
 viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;; Routing variables ;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 startx = -6.25       ;right side
 endx = (-4.8*n)-8.1
 starty = 12.15              ;top
 endy = (-((m/2)-1)*16.7)-22-((2-(m/2))*1.9)

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;  Create an array of mxn sram cells ;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 for(col 0 n-1

   coladjustment = col*-4.8

   ;;  Place Precharge at the top
   PreInst = coladjustment-16.5:3.05
   PInst = dbCreateInst(ccv pcv "PInst" PreInst "R0")
   dbFlattenInst(PInst 1 t)

   ;;  Place Write Buffers at the top
   BuffInst = coladjustment-10.35:13.2
   BInst = dbCreateInst(ccv buffcv "BuffInst" BuffInst "R0")
   dbFlattenInst(BInst 1 t)

   ;Place a pin for bit signals
   fig    =    dbCreateRect(ccv    "metal2"    list(coladjustment-9.4-1.45:starty+1.65-1.3
coladjustment-8.8-1.45:starty+2.25-1.3))
   pinname = buildString(list("Cbit" pcExprToString(col)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "inputOutput")
   pin = dbCreatePin(net fig pinname)

   ;Place a pin for bit_neg signals
   fig    =    dbCreateRect(ccv    "metal2"    list(coladjustment-9.4+1.45:starty+1.65-1.3
coladjustment-8.8+1.45:starty+2.25-1.3))
```

```
   pinname = buildString(list("CbitNeg" pcExprToString(col)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "inputOutput")
   pin = dbCreatePin(net fig pinname)

   yadjustment = 0

   ;; layout one row
   for(row 0 m-1
    if(modulo(row 2) == 1
     then
       ;instances sram_cellview
       Instpoint = coladjustment:yadjustment
       SInst = dbCreateInst(ccv scv "SInst" Instpoint "MX")
       dbFlattenInst(SInst 1 t)

       ; connect vdd
       via = dbCreateInst(ccv  viacv "via" endx+0.65:yadjustment-7.1 "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv              "metal1"              list(endx+1.05:yadjustment-6.75
endx+1.05+(0.3*n):yadjustment-7.45))

       ; connect vss to precharge
       via = dbCreateInst(ccv  viacv "via" endx-0.6:yadjustment+1.75 "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv               "metal1"               list(endx-0.2:yadjustment+2.1
endx+1.05+(0.3*n):yadjustment+1.4))

       ; connect poly contact for word signal
       dbCreateRect(ccv "poly1" list(startx:yadjustment-0.8 startx+0.8:yadjustment))
       dbCreateRect(ccv  "metal1"  list(startx+0.05:yadjustment-0.75  startx+2:yadjustment-
0.05))
       dbCreateRect(ccv "contact"  list(startx+0.2:yadjustment-0.6  startx+0.6:yadjustment-
0.2))
       dbCreateRect(ccv "poly1" list(startx:yadjustment-0.6 startx-0.5:yadjustment-0.25))
       dbCreateRect(ccv  "metal1"  list(startx-0.9:yadjustment-1.75  startx-0.4:yadjustment-
1.5))

       if(col == 0  then
         ;Place a pin for word signals
         fig     =     dbCreateRect(ccv     "metal1"     list(startx+0.1:yadjustment-0.7
startx+0.7:yadjustment-0.1))
        pinname = buildString(list("word" pcExprToString(row)) "_")
        net = dbCreateNet(ccv pinname)
        trm = dbCreateTerm(net pinname "input")
        pin = dbCreatePin(net fig pinname)
       )
       yadjustment = yadjustment-16.1

     else

       if(row > 0 then        yadjustment = yadjustment+1.9)
       ;instances sram_cellview
       Instpoint = coladjustment:yadjustment
       SInst = dbCreateInst(ccv scv "SInst" Instpoint "R0")
       dbFlattenInst(SInst 1 t)

       ; connect vdd
       via = dbCreateInst(ccv  viacv "via" endx+0.65:yadjustment+7.1 "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv              "metal1"              list(endx+1.05:yadjustment+7.45
endx+1.05+(0.3*n):yadjustment+6.75))

       ; connect vss to precharge
       via = dbCreateInst(ccv  viacv "via" endx-0.6:yadjustment-1.75 "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv               "metal1"               list(endx-0.2:yadjustment-1.4
endx+1.05+(0.3*n):yadjustment-2.1))

       ; connect a poly contact for word signal
       dbCreateRect(ccv "poly1" list(startx:yadjustment+.05 startx+0.8:yadjustment+0.85))
```

```
       dbCreateRect(ccv             "metal1"             list(startx+0.05:yadjustment+0.1
startx+0.75:yadjustment+0.8))
       dbCreateRect(ccv            "contact"             list(startx+0.2:yadjustment+0.25
startx+0.6:yadjustment+0.65))
       dbCreateRect(ccv "poly1" list(startx:yadjustment+0.25 startx-0.5:yadjustment+0.6))
       dbCreateRect(ccv             "metal1"             list(startx+0.35:yadjustment+0.1
startx+1.05:yadjustment+1.2))
       dbCreateRect(ccv             "metal1"             list(startx+0.35:yadjustment+1.2
startx+1.3:yadjustment+1.9))

       if(col == 0  then
         ;Place a pin for word signals
         fig     =     dbCreateRect(ccv     "metal1"     list(startx+0.1:yadjustment+0.15
startx+0.7:yadjustment+0.75))
         pinname = buildString(list("word" pcExprToString(row)) "_")
         net = dbCreateNet(ccv pinname)
         trm = dbCreateTerm(net pinname "input")
         pin = dbCreatePin(net fig pinname)
       )
       yadjustment = yadjustment-0.6
    )

   );end for column
 );end for row

;;;;;;;;;;;;;;;;;;;;;;;;  VDD  Connections  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;VDD route
 dbCreateRect(ccv "metal2" list(endx+0.35:starty endx+0.95:endy+1.3))

 ;Place a pin for VDD
 fig = dbCreateRect(ccv "metal2" list(endx+0.35:starty-2.55 endx+0.95:starty-3.15))
 net = dbCreateNet(ccv "vdd!")
 trm = dbCreateTerm(net "vdd!" "input")
 pin = dbCreatePin(net fig "vdd!")

;;;;;;;;;;;;;;;;;;;;;;;;  VSS  Connections  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;VSS route
 dbCreateRect(ccv "metal2" list(endx-0.9:starty endx-0.3:endy+1.3))

 ;Place a pin for VSS
 fig = dbCreateRect(ccv "metal2" list(endx-0.9:starty-0.9 endx-0.3:starty-0.3))
 net = dbCreateNet(ccv "vss!")
 trm = dbCreateTerm(net "vss!" "input")
 pin = dbCreatePin(net fig "vss!")

 ; connect vss to precharge
 via = dbCreateInst(ccv  viacv "via" endx-0.6:starty-0.6 "R0")
 dbFlattenInst(via 1 t)
 dbCreateRect(ccv "metal1" list(endx-0.2:starty-0.25 endx+1.05+(0.3*n):starty-0.95))

);end procedure cell_layout
```

B.3 package.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;; FileName: package.il
;; Author: Meenatchi Jagasivamani, April 2000
;;
;; procedure package makes the circuit fit the final package
;;
;; Usage In CIW:
;;  package(library  cellview  row_address  col_address  number_of_rows  number_of_cols
wordsize)
;;
;;  Ex:    package("library" "sram_32_4" 5 3 32 64 8)
;;      --> to add all I/O pins and route signals to meet package criteria
;;      --> Layout will be stored in cellview "sram_32_4" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;

procedure(package(clib cname x y m n w)

 ;Leaf-Cell library
 library = "sramleaf"

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;; Open necessary leaf-cells ;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")
 polyxcv = dbOpenCellViewByType(library "poly_M1" "layout" "" "r")
 sensecv = dbOpenCellViewByType(library "read_buffer" "layout" "" "r")
 invcv = dbOpenCellViewByType(library "wdata_inverter" "layout" "" "r")
 oeninvcv = dbOpenCellViewByType(library "oen_inverter" "layout" "" "r")

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;; Routing variables ;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 startx = -6.25       ;right side
 endx = (-4.8*n)-8.1
 starty = 12.15              ;top
 endy = (-((m/2)-1)*16.7)-((2-(m/2))*1.9)-13.35

 Changey = starty+endy+0.4+3.35

 deltax=1.25
 deltay=1.35

 for(bit 0 w-1

   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;;;;;;;;;;;;;;;;;  READ DATA ROUTING ;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;keep another variable to make calculations easier
   reversebit = w-1-bit

   ;place a sense amp for each bit
   sensept = endx-2.5*w-4.4:endy+4.8*bit+13.35
   sense = dbCreateInst(ccv  sensecv "sense" sensept "R90")
   dbFlattenInst(sense 1 t)

   ;place sensepos and senseneg pins
    fig  =  dbCreateRect(ccv  "metal2"  list(endx-2.5*w-11.65:endy+4.8*bit+7.75  endx-2.5*w-
12.05:endy+4.8*bit+8.15))
    pinname = buildString(list("senseneg" pcExprToString(bit)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)
```

```
   ;place pin for read-data
   fig  =  dbCreateRect(ccv  "metal1"  list(endx-2.5*w-14.8:endy+4.8*bit+7.7  endx-2.5*w-
15.3:endy+4.8*bit+8.2))
   pinname = buildString(list("DO" pcExprToString(bit)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "input")
   pin = dbCreatePin(net fig pinname)


   ;;;;;;;;;;;;;;;;;;route negative read data line
   dbCreateRect(ccv "metal1" list(endx-1.5:endy-deltay*bit endx-1.55-deltax*bit:endy+0.6-
deltay*bit))
   dbCreateRect(ccv   "metal1"   list(endx-1.55-deltax*bit:endy-deltay*bit      endx-2.15-
deltax*bit:endy-deltay*bit+1.35*bit+7.55+4.8*bit))

   ;place a via amp for each bit
   viapt = endx-1.85-deltax*bit:endy-deltay*bit+1.35*bit+7.95+4.8*bit
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;connect to senseamp
   dbCreateRect(ccv                "metal2"               list(endx-2.25-deltax*bit:endy-
deltay*bit+8.35+1.35*bit+4.8*bit endx-2.5*w-2.75:endy-deltay*bit+1.35*bit+7.55+4.8*bit))


   ;;;;;;;;;;;;;;;route positive read data line
   dbCreateRect(ccv   "metal1"   list(endx-1.5:endy-1.35*w-deltay*bit   endx-1.25*w-1.55-
deltax*bit:endy-1.35*w+0.6-deltay*bit))
   dbCreateRect(ccv    "metal1"    list(endx-1.25*w-1.55-deltax*bit:endy-1.35*w-deltay*bit
endx-1.25*w-2.15-deltax*bit:endy-deltay*bit+1.35*bit+10.45+4.8*bit))

   ;place a via amp for each bit
   viapt = endx-1.25*w-1.85-deltax*bit:endy-deltay*bit+1.35*bit+7.95+2.9+4.8*bit
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;connect to senseamp
   dbCreateRect(ccv            "metal2"            list(endx-1.25*w-2.25-deltax*bit:endy-
deltay*bit+1.35*bit+8.35+2.9+4.8*bit                                  endx-2.5*w-2.75:endy-
deltay*bit+1.35*bit+7.55+2.9+4.8*bit))

   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;;;;;;;;;;;;;;;;; WRITE DATA ROUTING ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   ;;;;;;;;;;;;;;;;;route positive read data line
   dbCreateRect(ccv     "metal1"     list(endx-1.5:Changey-(endy-deltay*bit)     endx-1.55-
deltax*bit:Changey-(endy+0.6-deltay*bit)))
   dbCreateRect(ccv  "metal1"  list(endx-1.55-deltax*bit:Changey-(endy-deltay*bit)   endx-
2.15-deltax*bit:Changey-(endy+7.55+(4.15-deltay)*bit)))

   ;place a via amp for each bit
   viapt = endx-1.85-deltax*bit:Changey-(endy+(4.15-deltay)*bit+7.95)
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;connect to senseamp
   dbCreateRect(ccv        "metal2"        list(endx-2.25-deltax*bit:Changey-(endy+(4.15-
deltay)*bit+8.35) endx-2.5*w-2.75:Changey-(endy+(4.15-deltay)*bit+7.55)))


   ;;;;;;;;;;;;;;;route negative read data line
   dbCreateRect(ccv  "metal1"  list(endx-1.5:Changey-(endy-1.35*w-deltay*bit)  endx-1.25*w-
1.55-deltax*bit:Changey-(endy-1.35*w+0.6-deltay*bit)))
   dbCreateRect(ccv     "metal1"     list(endx-1.25*w-1.55-deltax*bit:Changey-(endy-1.35*w-
deltay*bit) endx-1.25*w-2.15-deltax*bit:Changey+1.5-(endy+(4.15-deltay)*bit+10.45)))
   ;place a via amp for each bit
   viapt = endx-1.25*w-1.85-deltax*bit:Changey+1.5-(endy+(4.15-deltay)*bit+10.85)
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)
```

```
   ;connect to senseamp
   dbCreateRect(ccv   "metal2"   list(endx-1.25*w-2.25-deltax*bit:Changey+1.5-(endy+(4.15-
deltay)*bit+11.25) endx-2.5*w-2.75:Changey+1.5-(endy+(4.15-deltay)*bit+10.45)))

   ;invert write-data input to provide both positive and negative data lines
   ;place inverter for each bit
   invpt = endx-2.5*w-2.75+5.8:Changey+1.5+5.25-(endy+(4.15-deltay)*bit+10.45)
   inv = dbCreateInst(ccv  invcv "inv" invpt "MYR90")
   dbFlattenInst(inv 1 t)

   ;place pin for write-data
   fig    =    dbCreateRect(ccv    "metal2"    list(endx-2.5*w-11.05:Changey-(endy+(4.15-
deltay)*bit+8.25) endx-2.5*w-10.45:Changey-(endy+(4.15-deltay)*bit+7.65)))
   pinname = buildString(list("DI" pcExprToString(bit)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "input")
   pin = dbCreatePin(net fig pinname)

 )

 ;;;;;;;;;; connect TG of Read-buffers to OEN & OEN-neg
 oeninvpt = endx-2.5*w-13.7:endy+5.95
 oeninv = dbCreateInst(ccv  oeninvcv "oeninv" oeninvpt "MXR90")
 dbFlattenInst(oeninv 1 t)


 ;make power and ground connections for senseamp
 ;vss connection:
 ;place a via amp for each bit
 viapt = endx-2.5*w-1.85:endy+4.8*w+7.5
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)

 dbCreateRect(ccv "metal2" list(endx-2.5*w-1.45:endy+4.8*w+7.2 endx-0.9:endy+4.8*w+7.8))

 ;vss connection for senseamp inverter:
 dbCreateRect(ccv    "metal1"    list(    endx-2.5*w-1.45:endy+4.8*w+7.2       endx-2.5*w-
17.05:endy+4.8*w+7.8))

 ;vss connection for writedata  inverter
 viapt =  endx-2.5*w-3.4:Changey-endy-4.35
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)
 dbCreateRect(ccv   "metal2"   list(endx-2.5*w-3:Changey-endy-3.95   endx-0.9:Changey-endy-
4.75))

 ;vdd connection
 ;for gate:
 polyxpt = endx-2.5*w-3.85:endy+6.65
 polyx = dbCreateInst(ccv  polyxcv "polyx" polyxpt "R0")
 dbFlattenInst(polyx 1 t)
 ;connect gate and subx together
 dbCreateRect(ccv "metal1" list(endx-2.5*w-10.95:endy+6.35  endx-2.5*w-3.5:endy+6.95))
 ;route to vdd line
 dbCreateRect(ccv   "metal1"   list(endx-2.5*w-4.1:endy+6.35     endx-2.5*w-3.5:endy-2.7*w-
6.95))
 dbCreateRect(ccv   "metal1"   list(endx-2.5*w-3.5:endy-2.7*w-6.95   endx+0.35:endy-2.7*w-
6.35))

 ;for substrate contact
 viapt = endx+0.65:endy-2.7*w-6.65
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)

 ;vdd connection for writedata  inverter
 dbCreateRect(ccv "metal1" list(endx-2.5*w-9.15:Changey-3.9-endy endx-2.5*w-9.85:Changey-
4.15-(endy-3.3-1.35*2*w-7)))
 dbCreateRect(ccv     "metal1"     list(endx-2.5*w-9.85:Changey-4.15-(endy-3.3-1.35*2*w-7)
endx+1:Changey-4.15+0.8-(endy-3.3-1.35*2*w-7)))
 viapt = endx+1:Changey-3.75-(endy-3.3-1.35*2*w-7)
```

```
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)

 ;;;;;;;;;;;;;;;;;  Address Routing   ;;;;;;;;;;;;;;;;;;;;;
 addressbits = x+y
 for(addrline 0 addressbits-1
   if(addrline < y then
    ;add address pins to row (from word) address lines
    fig   =   dbCreateRect(ccv   "metal1"   list(14.05+(2.9*x)+1.4*(y-1):endy-9.8-(2.7*w)-
(1.4*addrline) 14.55+(2.9*x)+1.4*(y-1):endy-9.3-(2.7*w)-(1.4*addrline)))
    pinname = buildString(list("A" pcExprToString(addrline)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)
   else
    ;add address pins to  column (from read) address lines
    fig     =     dbCreateRect(ccv     "metal1"     list(14.05+(2.9*x)+1.4*(y-1):endy-9.8-
(2.7*w)+(1.4*(addrline-y+1))      14.55+(2.9*x)+1.4*(y-1):endy-9.3-(2.7*w)+(1.4*(addrline-
y+1))))
    pinname = buildString(list("A" pcExprToString(addrline)) "_")
    net = dbCreateNet(ccv pinname)
    trm = dbCreateTerm(net pinname "input")
    pin = dbCreatePin(net fig pinname)
   )
 )

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;  Route Control Signals ;;;;;;;;;;;;;;;;;;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;place via at the end of cen
  viapt = 14.15+(2.9*x)+1.4*(y-1):Changey+9.25-(endy-2.2-(2.7*w)-(2.9*y))
  via = dbCreateInst(ccv  viacv "via" viapt "R0")
  dbFlattenInst(via 1 t)

  ;;;;;;;;;;;;; route wen signal
  dbCreateRect(ccv   "metal1"   list(endx+4.8*(n/2):Changey+8.25-(endy-1.4-(2.7*w)-(2.9*y))
14.55+(2.9*x)+1.4*(y-1):Changey+8.25-(endy-2.2-(2.7*w)-(2.9*y))))

  ;;;;;;;;;;;;; route oen signal
  dbCreateRect(ccv       "metal2"       list(endx-2.5*w-13.15:endy-8.25-1.4-(2.7*w)-(2.9*y)
8.65+(2.9*x)+1.4*y:endy-8.25-2.2-(2.7*w)-(2.9*y)))
  dbCreateRect(ccv      "metal2"      list(8.65+(2.9*x)+1.4*y:endy-8.25-2.2-(2.7*w)-(2.9*y)
7.85+(2.9*x)+1.4*y:Changey+6.85-(endy-2.2-(2.7*w)-(2.9*y))))
  dbCreateRect(ccv     "metal2"     list(7.85+(2.9*x)+1.4*y:Changey+6.85-(endy-2.2-(2.7*w)-
(2.9*y)) 14.55+(2.9*x)+1.4*(y-1):Changey+6.05-(endy-2.2-(2.7*w)-(2.9*y))))

  ;connect oen to TG
  dbCreateRect(ccv   "metal2"   list(endx-2.5*w-13.15:endy-8.25-2.2-(2.7*w)-(2.9*y)   endx-
2.5*w-13.95:endy+2.85))

  ;place via at the end of oen
  viapt = 14.15+(2.9*x)+1.4*(y-1):Changey+6.45-(endy-2.2-(2.7*w)-(2.9*y))
  via = dbCreateInst(ccv  viacv "via" viapt "R0")
  dbFlattenInst(via 1 t)

)
```

B.4 read_decoder.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;
;; FileName: read_decoder.il
;; Author: Meenatchi Jagasivamani, April 2000
;;
;; procedure read_decoder will layout the column decoder for sense output
;;
;; Usage In CIW:
;;  read_decoder(library  cellview  row_address  col_address  number_of_rows  number_of_cols
wordsize)
;;
;;
;;  Ex:    read_decoder("sram" "sram_32_4" 5 3 32 64 8)
;;  --> to create decoder for sram cells with 32 rows and 64 columns & wordsize of 8
;;  --> Layout will stored in cellview "sram_32_4" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;

procedure(read_decoder(clib cname x y m n w)

 ;Leaf-Cell library
 library = "sramleaf"

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;; Open necessary leaf-cells ;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 nfetcv = dbOpenCellViewByType(library "nfet" "layout" "" "r")
 subcv = dbOpenCellViewByType(library "substrate_contact" "layout" "" "r")
 viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")
 polyxcv = dbOpenCellViewByType(library "poly_M1" "layout" "" "r")
 buffcv = dbOpenCellViewByType(library "buffer" "layout" "" "r")
 invcv = dbOpenCellViewByType(library "ColAdr_inverter" "layout" "" "r")


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;; Routing variables ;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 startx = -6.25       ;right side
 endx = (-4.8*n)-8.1
 starty = 12.15               ;top
 endy = (-((m/2)-1)*16.7)-22-((2-(m/2))*1.9)


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;; Layout column decoders (for write_neg -- during read operations) ;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;; initialize column variables
 ;;  2**y = # of blocks -> y=#of address lines  ->  # of columns = n = 2**total_cols
 deltay = 1.35*w   ; to adjust for sense data line routing

 ;;;;;;;;;;;;;;  Connect wordsize blocks together ;;;;;;;;;;;;;;;
 for(col 0 n-1
   ;place a M1_M2 via at every sense_neg port
   viapt = endx+2.35+(4.8*col):endy+1.35+13.35
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;place a M1_M2 via at output of sense_neg switch
   viapt = endx+2.35+(4.8*col):endy-1.7+13.35
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;place a M1_M2 via at every write_neg port
   viapt = endx+3.8+(4.8*col):endy+1.35+13.35
```

```
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;place a M1_M2 via at every sense port
   viapt = endx+5.25+(4.8*col):endy+1.35+13.35
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;place a M1_M2 via at output of sense switch
   viapt = endx+5.25+(4.8*col):endy-1.7+13.35
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;connect sense_neg vias together
   dbCreateRect(ccv              "metal1"              list(endx+2+4.8*col:endy+13.35-0.05
endx+2.7+4.8*col:13.35+endy+0.9))

   ;connect write_neg vias together
   dbCreateRect(ccv             "metal1"             list(endx+3.45+4.8*col:13.35+endy-0.05
endx+4.15+4.8*col:13.35+endy+0.9))

   ;connect sense vias together
   dbCreateRect(ccv              "metal1"              list(endx+5.6+4.8*col:13.35+endy-0.05
endx+4.9+4.8*col:13.35+endy+0.9))

   ;place a poly contact to connect to switches
   polyxpt = endx+3.8+(4.8*col):13.35+endy-0.4
   polyx = dbCreateInst(ccv  polyxcv "polyx" polyxpt "R0")
   dbFlattenInst(polyx 1 t)

   ;;; add switches for sense outputs
   ;place a switch at every sense+ port
   nfetInst = endx+2.4+4.8*col:13.35+endy-1.7
   nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
   dbFlattenInst(nfInst 1 t)

   ;place a switch at every sense+ port
   nfetInst = endx+5.3+4.8*col:13.35+endy-1.7
   nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
   dbFlattenInst(nfInst 1 t)

   ;;; layout access grid for sense data lines
   ;;;define sense grid variables
   placex = endx+2.35+col*4.8
   placey = endy-1.35-3.05-1.35*modulo(col w)+13.35

   ;;; layout connections for sense- nodes
   ;connect current sense- to appropriate address line
   dbCreateRect(ccv "metal2" list(placex-0.4:13.35+endy-2.1 placex+0.4:placey+0.4))

   ;place a M1_M2 via at intersection
   viapt = placex:placey
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;;; layout connections for sense- nodes
   ;redefine grid variables for sense-
   placex = endx+5.25+col*4.8
   placey = endy-3.05-1.35*modulo(col w)-deltay-1.35+13.35

   ;connect current sense+ to appropriate address line
   dbCreateRect(ccv "metal2" list(placex-0.4:13.35+endy-2.1 placex+0.4:placey+0.4))

   ;place a M1_M2 via at intersection
   viapt = placex:placey
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;instantiate a sub contact below sense output switches
   subInst = endx+3.9+(4.8*col):13.35+endy-2.15
   scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
```

```
   dbFlattenInst(scInst 1 t)

   ; connect sub contact to VSS line
   dbCreateRect(ccv    "metal1"    list(endx+3.9-0.45+(4.8*col):13.35+endy-2.3    endx+3.9-
0.45+0.7+(4.8*col):13.35+endy-0.45-2.3))
 )

 for(databit 0 w-1
   caddr_startx=endx+4.2+(databit*4.8*w)

   ;; layout the horizontal line for the current sense- bit lines
   dbCreateRect(ccv      "metal1"      list(endx+2.8-4.35:13.35+endy-3.35-1.35-databit*1.35
endx+4.9+4.8*(n-1):13.35+endy-1.35-databit*1.35-2.75))
   ;Place a pin for sense- signals
   fig   =   dbCreateRect(ccv   "metal1"   list(endx+2.8:13.35+endy-1.35-3.35-databit*1.35
endx+3.4:13.35+endy-2.75-1.35-databit*1.35))
   pinname = buildString(list("BlkSnse" pcExprToString(databit)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "output")
   pin = dbCreatePin(net fig pinname)

   ;; layout the horizontal line for the current sense+ bit lines
   dbCreateRect(ccv  "metal1"  list(endx+2.8-4.35:13.35+endy-1.35-3.35-deltay-databit*1.35
endx+4.9+4.8*(n-1):13.35+endy-1.35-deltay-databit*1.35-2.75))
   ;Place a pin for sense+ signals
   fig = dbCreateRect(ccv "metal1" list(endx+2.8:13.35+endy-1.35-3.35-deltay-databit*1.35
endx+3.4:13.35+endy-1.35-deltay-2.75-databit*1.35))
   pinname = buildString(list("BlkSnseNeg" pcExprToString(databit)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "output")
   pin = dbCreatePin(net fig pinname)
 )

 for(block 0 (2**y)-1
    caddr_startx=endx+4.2+(block*4.8*w)
   ;join word-size blocks together for write_neg signal
   dbCreateRect(ccv    "poly1"    list(caddr_startx-1.15:13.35+endy-1.15    caddr_startx-
1.15+1.5+4.8*(w-1):13.35+endy-0.8))

   ;route write_neg signal out of read-data access lines
   placex = endx+3.8+block*w*4.8
   placey = 13.35+endy-3.3-1.35-1.35*2*w
   dbCreateRect(ccv "metal2" list(placex-0.4:13.35+endy+0.95 placex+0.4:placey))

   ;Place a pin for write_neg signals
   fig = dbCreateRect(ccv "metal2" list(placex-0.3:placey+0.7 placex+0.3:placey+0.1))
   pinname = buildString(list("BlkRead" pcExprToString((2**y)-1-block)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "input")
   pin = dbCreatePin(net fig pinname)

   ;place a M1_M2 via at the end
   viapt = placex:placey+0.4
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

  ;add buffer at end
  buffpt = placex+1.35:placey-5.25
  buff = dbCreateInst(ccv  buffcv "buff" buffpt "MY")
  dbFlattenInst(buff 1 t)

  ;connect m2 to buffer
  dbCreateRect(ccv "metal1" list(placex-0.35:placey placex+0.35:placey-0.6))

  ;connect buffer to decoder
  if(modulo(block 2) == 1
  then
    dbCreateRect(ccv "metal1" list(placex-0.35:placey-6.8 placex+0.35:placey-9.25))
  else
    dbCreateRect(ccv "metal1" list(placex-0.35:placey-6.8 placex+0.35:placey-8.45))
  )
```

```
)

;; Connect buffer to VDD & VSS
; connect vdd to vdd bus
dbCreateRect(ccv "metal2" list(endx+0.95:placey-6.3 placex+2.35:placey-7))

;connect vss together
dbCreateRect(ccv "metal2" list(endx+2.95:placey-0.55 placex+2.35:placey-1.25))

; add via
viapt = endx+2.55:placey-0.9
via = dbCreateInst(ccv  viacv "via" viapt "R0")
dbFlattenInst(via 1 t)

; connect vss to vss bus
dbCreateRect(ccv "metal1" list(endx+2.1:placey-0.55 endx-0.15:placey-1.25))

; add via
viapt = endx-0.6:placey-0.9
via = dbCreateInst(ccv  viacv "via" viapt "R0")
dbFlattenInst(via 1 t)

;;;;;;;;;;;;;;;  Layout column decoder for read signal ;;;;;;;;;;;;;;;;;;;;;
for(addrline 0 y-1
  placex = endx+3.85
  placey = endy-1.35-4.35-(2.7*w)-(2.9*addrline)-9.2+13.35
  for(col 0 (2**y)-1

    ;;;;;;;;check if a nfet should be place at this col
    if(modulo((col-(2**addrline)) (2**(addrline+1))) == 0
    then       ;;  layout negative address lines
      ;;  Place nfet decoders for this col
      nfetInst = placex+(4.8*w)*col:placey-1.45
      nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
      dbFlattenInst(nfInst 1 t)
      if(addrline == 0
      then
       dbCreateRect(ccv         "metal1"         list(placex-0.4+(4.8*w)*col:placey+0.35
placex+0.3+(4.8*w)*col:placey+0.35+0.65))
      else
       dbCreateRect(ccv         "metal1"         list(placex-0.4+(4.8*w)*col:placey+0.35
placex+0.3+(4.8*w)*col:placey+0.35+2.2))
      )
      ;;;;;;;;put substrate contact at every other address lines
      ;instantiate a sub contact next to nfet
      subInst = placex+(4.8*w)*col+1.5:placey-1.45-0.2
      scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
      dbFlattenInst(scInst 1 t)
      ; draw m1 to connect to next ** address line **
      dbCreateRect(ccv     "metal1"     list(placex-0.4+(4.8*w)*col:placey-1.8     placex-
0.4+(4.8*w)*col-(4.8*w*(2**addrline)):placey-1.2))

      ;;;connect decoder's sub contact to VSS line
      ;connect to M2
      viapt =  placex+(4.8*w)*col+1.5-0.1:placey-1.45
      via = dbCreateInst(ccv  viacv "via" viapt "R0")
      dbFlattenInst(via 1 t)
      dbCreateRect(ccv "metal2" list(placex+(4.8*w)*col+1:placey-1.45-0.3 endx+3.9-0.45-
0.65-3.1:placey-1.45-0.2+0.5))

    else if(modulo(col (2**(addrline+1))) == 0
    then      ;; layout positive address lines
      ; place instance
      nfetInst = placex+(4.8*w)*col:placey
      nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
      dbFlattenInst(nfInst 1 t)
      ;connect to next nfet for this col
      dbCreateRect(ccv        "metal1"        list(placex-0.4+(4.8*w)*col:placey+1.7-2.9
placex+0.3+(4.8*w)*col:placey+1.8+0.75-2.9))

      ;;;;;;;;put substrate contact at every other address lines
```

```
       ;instantiate a sub contact next to nfet
       subInst = placex+(4.8*w)*col+1.5:placey-0.2
       scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
       dbFlattenInst(scInst 1 t)
       ;;;connect decoder's substrate contact to VSS line
       ;let sub contact connect to M2
       viapt =  placex+(4.8*w)*col+1.4:placey
       via = dbCreateInst(ccv  viacv "via" viapt "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv "metal2" list(placex+(4.8*w)*col+1:placey-0.3 endx+3.9-0.45-0.65-
3.1:placey-0.2+0.5))


    );; if positive address line
     )
   );end for col

   ;route poly for positive and negative address lines
   dbCreateRect(ccv     "poly1"     list(startx-0.95:placey+0.9     endx+2.8:placey+0.55))
;positive
   dbCreateRect(ccv  "poly1"  list(startx-0.95:placey-1.45+0.9  endx+2.8:placey-1.45+0.55))
;negative

   ;place inverters to get both negative and positive address
   invpt =  startx-7.85:placey+4.95
   inv = dbCreateInst(ccv  invcv "inv" invpt "R270")  ; connect vdd
   dbFlattenInst(inv 1 t)
   ;connect sub contact to ground
   dbCreateRect(ccv "metal2" list(startx-1.8:placey-0.35 endx-0.3:placey+0.35))

   ;route write-address lines to read-address lines
   dbCreateRect(ccv                  "metal1"                list(startx+5.65:placey+1.05
7.85+(2.9*x)+1.4*addrline:0.35+placey))
   ;place via
   viapt =  8.25+(2.9*x)+1.4*addrline:0.7+placey
   via = dbCreateInst(ccv  viacv "via" viapt "R270")  ; connect vdd
   dbFlattenInst(via 1 t)
   dbCreateRect(ccv           "metal2"           list(7.85+(2.9*x)+1.4*addrline:placey+0.35
8.65+(2.9*x)+1.4*addrline:-7.4*m))
   dbCreateRect(ccv           "metal1"           list(7.85+(2.9*x)+1.4*addrline:placey+1.05
8.65+(2.9*x)+1.4*addrline:endy-0.45-(2.7*w)-(1.4*addrline)))
   dbCreateRect(ccv       "metal1"       list(8.65+(2.9*x)+1.4*addrline:endy-0.45-(2.7*w)-
(1.4*addrline) 14.65+1.4*(x+y)+(2.9*x)+1.4*(y-1):endy-1.25-(2.7*w)-(1.4*addrline)))

 )

 ;;;;;;;;Route VSS line for read decoder Sub x
 ;extend VSS route
 dbCreateRect(ccv    "metal2"    list(endx-0.9:13.35+endy+1.3    endx-0.3:13.35+endy-13.75-
(2.7*w)-(2.9*y)))
 ;extend VDD route
 dbCreateRect(ccv   "metal2"   list(endx+0.35:13.35+endy+1.3   endx+0.95:13.35+endy-11.65-
(2.7*w)))

 ;connect VSS to first line (connected to subx)
 dbCreateRect(ccv  "metal1"  list(endx+0.6-0.45-0.65:13.35+endy-0.45-2.3  endx+4.9+4.8*(n-
1):13.35+endy-0.45-2.3-0.6))
 ;place a M1_M2 via at VSS line
 viapt = endx+3.9-0.45-0.65-3.1-0.3:13.35+endy-0.45-2.3-0.6+0.3
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)

 ;connect well contact of inverter to vdd
 dbCreateRect(ccv    "metal1"    list(startx+5.05:endy+2.45-(2.7*w)    startx+0.05-4.8*(w-
1):endy+1.7-(2.7*w)))

 ;place pin for OEN (read enable) signal
 dbCreateRect(ccv          "metal1"          list(endx+4.8*(n/2):endy-0.45-(2.7*w)-(2.9*y)
endx+0.8+4.8*(n/2):endy-0.95-(2.7*w)-(2.9*y)))
 ;place a M1_M2 via at VSS line
 viapt = endx+0.4+4.8*(n/2):13.35+endy-14.75-(2.7*w)-(2.9*y)
```

```
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)
 fig   =   dbCreateRect(ccv   "metal2"   list(endx+0.15+4.8*(n/2):13.35+endy-14.75-(2.7*w)-
(2.9*y) endx+0.65+4.8*(n/2):13.35+endy-14.35-(2.7*w)-(2.9*y)))
 pinname = "OEN"
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)

) ;; end procedure read_decoder.il
```

B.5 sram_array.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FileName: sram_array.il
;; procedure sram_array will layout an array of sram cells
;;
;; Usage In CIW:
;; procedure sram_array is the top-level function to layout an SRAM circuit
;;
;; Ex:  load("sram_array.il")  sram_array("sram" "sram_32_4" 256 8)
;;   --> to create array of sram cells with 256 words with a wordsize of 8 bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  Load all other necessary files ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
load("cell_layout.il")
load("word_decoder.il")
load("read_decoder.il")
load("write_decoder.il")
load("package.il")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;  procedure sram_array  ;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
procedure(sram_array(clib cname words wordsize)

  ;approximate aspect ratio for 1 bit = 2^-2
  AR_1bit = 0.65

  ;;;;;;Calculate row and col for equal aspect ratio
  ar = floor(log(wordsize* AR_1bit)/log(2))         ;aspect ratio for 1 wordsize block
  k = int(log(words)/log(2))         ;number of address lines
  y = floor((k-ar)/2)                ;x+y = k
  x = k-y                            ;x = ar+y

  total_rows = int(2**x)
  total_cols = (2**y)*wordsize

  ;; layout sram cells
  cell_layout(clib cname total_rows total_cols)

  ;; layout row & column address decoder
  word_decoder(clib cname x y total_rows total_cols wordsize)
  read_decoder(clib cname x y total_rows total_cols wordsize)
  write_decoder(clib cname x y total_rows total_cols wordsize)
  package(clib cname x y total_rows total_cols wordsize)

)
```

B.6 sram_compiler.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FileName: sram_compiler.il
;; Author:  Meenatchi Jagasivamani, April 2000
;;
;; procedure sram_compiler will generate an embedded SRAM layout
;;
;; Usage In CIW:
;;   sram_compiler(library cellview Words Wordsize Type)
;;
;;   --> Possible Types:
;;        Type = 0  -- Simple SRAM array without Array Partitioning
;;        Type = 1  -- Array Partitioned SRAM array with the Block Select at bottom
;;
;; Ex:  sram_compiler("sram" "block_1024_8" 256 8 1)
;;        --> Create a 1024x8 size SRAM that is partitioned into 4 blocks for low-power
;;        --> Layout will be stored in cellview "block_1024_8" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  Load all other necessary files ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
load("sram_array.il")
load("BS_center.il")
load("BS_bottom.il")


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;; procedure to layout SRAM ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
procedure(sram_compiler(clib cname words w type)

  if(type == 0 then
    ;;;;; generate a simple SRAM array
    sram_array(clib cname words w)
  else

    ;;;;; generate circuit for 1 block = words/4

    ; create block in a temporary cellview called : "temp_" + cname
    block = int(words/4)    ; block size
    blkcvname = buildString(list("temp" pcExprToString(cname)) "_")
    sram_array(clib blkcvname block w)

    ;call array-partitioning function
    array_partition(clib cname words w)

    ;delete temporary  block cellview
    ddDeleteObj(ddGetObj(clib blkcvname))
  )

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;; Save Cellview before quitting ;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 dbSave(ccv)    ;save cellview
 dbClose(ccv)   ;close cellview
)
```

B.7 word_decoder.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;
;; FileName: word_decoder.il
;; Author:  Meenatchi Jagasivamani, April 2000
;;
;; procedure word_decoder will layout the row decoder for word sram cells
;;
;; Usage In CIW:
;;  word_decoder(library  cellview  row_address  col_address  number_of_rows  number_of_cols
wordsize)
;;
;;  Ex:   word_decoder("sram" "sram_32_4" 5 3 32 64 8)
;;  --> to create decoder for sram cells with 32 rows and 64 columns & wordsize of 8
;;  --> Layout will stored in cellview "sram_32_4" under library "sram"
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;

procedure(word_decoder(clib cname x y m n w)

 ;Leaf-Cell library
 library = "sramleaf"

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;; Open necessary leaf-cells ;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 nfetcv = dbOpenCellViewByType(library "nfet" "layout" "" "r")
 subcv = dbOpenCellViewByType(library "substrate_contact" "layout" "" "r")
 viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")
 polyxcv = dbOpenCellViewByType(library "poly_M1" "layout" "" "r")
 buffcv = dbOpenCellViewByType(library "buffer" "layout" "" "r")
 invcv = dbOpenCellViewByType(library "RowAdr_inverter" "layout" "" "r")


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;; Routing variables ;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 startx = -6.25       ;right side
 endx = (-4.8*n)-8.1
 starty = 12.15              ;top
 endy = (-((m/2)-1)*16.7)-22-((2-(m/2))*1.9)

 deltax = 8.85


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;; Layout buffers for row address decoder (for word signal) ;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 for(row 0 m-1
   if(modulo(row 2) == 1
     then  ;; odd row numbers
       ;instantiate a buff contact below nfet
       buffInst = startx+6:-3.05-7.4*(row-1)
       scInst = dbCreateInst(ccv buffcv "buffInst" buffInst "R90")
       dbFlattenInst(scInst 1 t)

       ;connect buffer to decoder output
       dbCreateRect(ccv  "metal1"  list(startx+6.85:-1.35-7.4*(row-1)  startx+10.65:-0.65-
7.4*(row-1)))
     else
       ;instantiate a buf contact below nfet
       buffInst = startx+6:-4.5-7.4*(row-1)
       scInst = dbCreateInst(ccv buffcv "buffInst" buffInst "MYR90")
       dbFlattenInst(scInst 1 t)
```

```
       ;connect buffer to decoder output
       dbCreateRect(ccv   "metal1"   list(startx+7.55:-6.35-7.4*(row-1)   startx+9.2:-5.65-
7.4*(row-1)))
   )
 )

 ;;;;;;;;  Make VDD & VSS Connections for buffer
 ;;;;; for VSS
 dbCreateRect(ccv "metal2" list(startx+1.25:starty-0.25 startx+2.05:endy+19.85))
 viapt = startx+1.65:starty-0.6
 via = dbCreateInst(ccv  viacv "via" viapt "R0")  ; connect vdd
 dbFlattenInst(via 1 t)

 ;;;;; for VDD
 dbCreateRect(ccv "metal2" list(startx+7:starty-4.7 startx+7.8:endy+19.85))
 viapt = startx+7.4:starty-5.05
 via = dbCreateInst(ccv  viacv "via" viapt "R0")  ; connect vdd
 dbFlattenInst(via 1 t)


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;; Layout decoders for row address decoder (for word signal) ;;;;;;
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 for(raddress 0 x-1
  ;initialize row counters
  neg_row = 0
  pos_row = 0

  for(row 0 m-1
   if(modulo(row 2) == 1
   then  ;; odd row numbers

     ;determine if a nfet should be placed for the current row
     if(modulo((row-(2**raddress)) (2**(raddress+1))) == 0
     then
       ;;  layout negative address lines for LSB  -- A0_neg  only
       ;;  Place nfet decoders for this row
       nfetInst = -4.1+deltax:-1.05-7.4*(row-1)
       nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R0")
       dbFlattenInst(nfInst 1 t)

       ; connect to next address line
       dbCreateRect(ccv "metal1" list(-3+deltax:8.45-7.4*row -2.3+deltax:6.75-7.4*row))

       ;;;;;;;;;;;  connect last address line to VDD for odd rows  ;;;;;;;;;;;;;;
       if(raddress == x-1  ;; the last address line --> means raddress = x-1 = 0 & row =
1
         then
           currentx = -4.1+(2.9*raddress)
           currenty = 1.35

           ;Make connection for VDD -- to power the address decoder
           dbCreateRect(ccv    "metal1"    list(currentx+1.8+deltax:currenty-0.3        -
3.75+(2.9*x)+deltax:currenty+0.4))
           viapt = 0.45-3.75+(2.9*x)+deltax:currenty+0.05
           via = dbCreateInst(ccv  viacv "via" viapt "R0")  ; connect vdd
           dbFlattenInst(via 1 t)
       )
     )

     ;;;;;;;;  Substrate Contacts ;;;;;;;;;;;;;;;;;;;;;;
     ;put substrate contact every other row and at every other address lines
     ;instantiate a sub contact below nfet
     subInst = -1.1+(5.8*floor(raddress/2))+deltax:-2.65-7.4*(row-1)
     scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
     dbFlattenInst(scInst 1 t)

     ;instantiate a sub contact above nfet
     subInst = -1.1+(5.8*floor(raddress/2))+deltax:2.65-7.4*(row-1)
     scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
     dbFlattenInst(scInst 1 t)
```

```
     ;instantiate a via contact below nfet
     viapoint = -1.2+(5.8*floor(raddress/2))+deltax:-3.15+0.7-7.4*(row-1)
     viaInst = dbCreateInst(ccv viacv "viaInst" viapoint "R0")
     dbFlattenInst(viaInst 1 t)

     ;instantiate a via contact above nfet
     viapoint = -1.2+(5.8*floor(raddress/2))+deltax:2.85-7.4*(row-1)
     viaInst = dbCreateInst(ccv viacv "viaInst" viapoint "R0")
     dbFlattenInst(viaInst 1 t)

     ;; connect substrates to VSS line
     dbCreateRect(ccv  "metal2"  list(-1.05+0.25+(5.8*floor(raddress/2))+deltax:2.65+0.55-
7.4*(row-1) -3.65+(2.9*x)+deltax:1.95+0.55-7.4*(row-1)))
     dbCreateRect(ccv  "metal2"  list(-1.05+0.25+(5.8*floor(raddress/2))+deltax:-2.65+0.55-
7.4*(row-1) -3.65+(2.9*x)+deltax:-3.35+0.55-7.4*(row-1)))

   else  ;; even row numbers (first one)
     ;determine if a nfet should be placed for the current row
     if(modulo(row    (2**(raddress+1)))    ==    0    ||    (modulo((row-(2**raddress))
(2**(raddress+1))) == 0)
     then
       ;;  layout even rows -- both pos & neg  address lines
       if(raddress == 0
       then
         ;;  Place nfet decoders for this row
         nfetInst = -5.55+deltax:1.35-7.4*row
         nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R0")
         dbFlattenInst(nfInst 1 t)

         ; draw m1 to connect to next address line
         dbCreateRect(ccv   "metal1"   list(-3.75+deltax:1.75-7.4*row   -2.3+deltax:1.05-
7.4*row))

       else ;; not the first address line
         currentx = -5.55+(2.9*raddress)
         currenty = 3.85-(7.4*((2*row)+(2**raddress)-1)/2)

         if(raddress == 1
         then
           currenty = 1.35-7.4*row
         )

         if(modulo((row-(2**raddress)) (2**(raddress+1))) == 0
         then  ;; for negative addresslines
           currentx = -4.1+(2.9*raddress)
           neg_row = neg_row+1

           ; draw m1 to connect to next address line
           dbCreateRect(ccv                    "metal1"                    list(-
3+(2.9*raddress)+deltax:currenty+0.4+(14.8*(2**(raddress-1)))                    -
2.3+(2.9*raddress)+deltax:currenty+0.4))

           ; negative address lines need to connect to previous line
           dbCreateRect(ccv   "metal1"   list(currentx-0.35+deltax:currenty+0.4   currentx-
1.8+deltax:currenty-0.3))

           if(raddress == x-1
           then
             ;Make connection for CEN -- to power the address decoder
             dbCreateRect(ccv        "metal1"        list(currentx+1.8+deltax:currenty-0.3
currentx+3.3+deltax+1.25:currenty+0.4))
             viapt = currentx+5+deltax:currenty+0.05
             via = dbCreateInst(ccv  viacv "via" viapt "R0")  ; connect vdd
             dbFlattenInst(via 1 t)

             ;place a pin for CEN (chip enable) signal
             fig   =   dbCreateRect(ccv   "metal2"   list(currentx+4.8+deltax:currenty-0.15
currentx+5.2+deltax:currenty+0.25))
             pinname = "CEN"
             net = dbCreateNet(ccv pinname)
```

```
            trm = dbCreateTerm(net pinname "input")
            pin = dbCreatePin(net fig pinname)

            ;route out CEN
            dbCreateRect(ccv       "metal2"       list(currentx+4.6+deltax:currenty+0.05
currentx+5.4+deltax:starty+15.6+(2.7*w)+(2.9*y)))
            dbCreateRect(ccv                                                    "metal2"
list(currentx+5.4+deltax:starty+14.8+(2.7*w)+(2.9*y)               14.55+(2.9*x)+1.4*(y-
1):starty+15.6+(2.7*w)+(2.9*y)))
          )

        else
          pos_row = pos_row+1
          ; draw m1 to connect to next address line
           dbCreateRect(ccv   "metal1"   list(-3.75+(2.9*raddress)+deltax:currenty-0.3   -
3+(2.9*raddress)+deltax:currenty+0.7-0.3))
        )

        ; place instance
         nfetInst = currentx+deltax:currenty
         nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R0")
         dbFlattenInst(nfInst 1 t)
       )
     )
   )
 );end for row

 ;;;connect nfet together for this address line
 ;connect negative address lines together
 dbCreateRect(ccv           "poly1"           list(-5+(2.9*raddress)+deltax:0.7          -
5.55+(2.9*raddress)+0.9+deltax:-7.4*(m-1)))
 ;connect positive address lines together
 dbCreateRect(ccv          "poly1"          list(-3.55+(2.9*raddress)+deltax:0.7         -
4.1+(2.9*raddress)+0.9+deltax:-7.4*(m-1)))

 ;place inverters to get both negative and positive address
 invpt =  -0.05+(2.9*raddress)+0.9+deltax:11.4-7.4*(m-1)
 inv = dbCreateInst(ccv  invcv "inv" invpt "R180")  ; connect vdd
 dbFlattenInst(inv 1 t)

; ;;;;place pin for positive address line -- 0.6x0.6  dx = 1.45
;  fig  =  dbCreateRect(ccv  "metal1"  list(-3.7+(2.9*raddress)+deltax:-0.7+5.5-7.4*m  -
3.1+(2.9*raddress)+deltax:-0.1+5.5-7.4*m))
; pinname = buildString(list("wordAdr" pcExprToString(raddress)) "_")
; net = dbCreateNet(ccv pinname)
; trm = dbCreateTerm(net pinname "input")
; pin = dbCreatePin(net fig pinname)

 ;; route word address lines
 dbCreateRect(ccv       "metal1"       list(-3.1+(2.9*raddress)+deltax:5.4-7.4*m       -
3.8+(2.9*raddress)+deltax:endy+0.2+(1.4*raddress)-(2.7*w)))
 dbCreateRect(ccv    "metal1"    list(-3.8+(2.9*raddress)+deltax:endy+0.2+(1.4*raddress)-
(2.7*w) 14.65+(2.9*x)+1.4*(x+y)+1.4*(y-1):endy+1+(1.4*raddress)-(2.7*w)))

) ;;;;;;;;;;;;end for raddress

 ;Make connection for VSS -- for the substrate contacts
 dbCreateRect(ccv "metal1" list(-6.65:11.2 -3.75+(2.9*x)+deltax:11.9))
 viapt = -3.3+(2.9*x)+deltax:11.55
 via = dbCreateInst(ccv  viacv "via" viapt "R0")  ; connect vdd
 dbFlattenInst(via 1 t)
 dbCreateRect(ccv "metal2" list(-3.7+(2.9*x)+deltax:11.2 -2.9+(2.9*x)+deltax:-7.4*m))

 ;Make connection for VDD -- to power the row address decoder
 dbCreateRect(ccv "metal1" list(-6.65:6.75 startx+7:7.45))

 ;connect inverter to VDD
 dbCreateRect(ccv   "metal1"   list(startx+7.05:11.4-12.2-7.4*(m-1)   startx+7.05+0.7:15.7-
12.25-7.4*(m-1)))

)
```

B.8 write_decoder.il

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FileName: write_decoder.il
;; procedure write_decoder will layout the column decoder for write data
;; given:  number of rows, number of cols, wordsize
;;
;; Usage In CIW:
;;     write_decoder(row_address col_address number_of_rows number_of_cols wordsize)
;;
;;
;;  Ex:    write_decoder(5 3 32 64 8)
;;  --> to create decoder for sram cells with 32 rows and 64 columns & wordsize of 8
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  load("write_decoder.il")    write_decoder(2 3 4 8 1)
;; for 32x4: load("write_decoder.il")  write_decoder("sram" "sram_32_4" 3 2 8 16 4)
;;;;;;;;;;;;;;;;;;;;;;;;;;
;; for 64x8: load("write_decoder.il")  write_decoder("sram" "sram_32_4" 4 2 16 32 8)
;; for 32x8: load("write_decoder.il")  write_decoder(3 2 8 32 8)
;;;;;;;;;;;;;;;
procedure(write_decoder(clib cname x y m n w)
 library = "sramleaf"

 ;create db variable for compiler
 ccv = dbOpenCellViewByType(clib cname "layout" "maskLayout" "a")
 nfetcv = dbOpenCellViewByType(library "nfet" "layout" "" "r")
 subcv = dbOpenCellViewByType(library "substrate_contact" "layout" "" "r")
 viacv = dbOpenCellViewByType(library "M1_M2" "layout" "" "r")
 polyxcv = dbOpenCellViewByType(library "poly_M1" "layout" "" "r")
 buffcv = dbOpenCellViewByType(library "buffer" "layout" "" "r")
 invcv = dbOpenCellViewByType(library "ColAdr_inverter" "layout" "" "r")

 Instpoint = 0:0

 ; Routing variables
 startx = -6.25       ;right side
 endx = (-4.8*n)-8.1
 starty = 12.15              ;top
 endy = (-((m/2)-1)*16.7)-22-((2-(m/2))*1.9)

 Changey = starty+endy+0.4

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;Layout column decoders (for write -- during write operations)
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;; initialize column variables
 ;; total_cols = y + int(log(w)/log(2))
 ;;  2**y = # of blocks -> y=#of address lines  ->  # of columns = n = 2**total_cols
 deltay = 1.35*w   ; to adjust for write_data data line routing

 ;;;;;;;;;;;;;  Connect wordsize blocks together ;;;;;;;;;;;;;;
 for(col 0 n-1
   ;;; layout access grid for write_data data lines
   ;;;define write_data grid variables
   placex = endx+2.35+col*4.8
   placey = endy-3.05-1.35*modulo(col w)

   ;;; layout connections for write_data- nodes
   ;connect current write_data- to appropriate address line
   dbCreateRect(ccv   "metal2"   list(placex-0.4:Changey-(endy-2.1)   placex+0.4:Changey-
(placey+0.4)))

   ;place a M1_M2 via at intersection
   viapt = placex:Changey-placey
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;; layout connections for write_data- nodes
```

```
   ;redefine grid variables for write_data-
   placex = endx+5.25+col*4.8
   placey = endy-3.05-1.35*modulo(col w)-deltay

   ;connect current write_data+ to appropriate address line
   dbCreateRect(ccv   "metal2"   list(placex-0.4:Changey-(endy-2.1)   placex+0.4:Changey-
(placey+0.4)))

   ;place a M1_M2 via at intersection
   viapt = placex:Changey-placey
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)
 )

 for(databit 0 w-1
   caddr_startx=endx+4.2+(databit*4.8*w)
   ;; layout the horizontal line for the current write_data- bit lines
   dbCreateRect(ccv      "metal1"      list(endx+2.8-4.35:Changey-(endy-3.35-databit*1.35)
endx+4.9+4.8*(n-1):Changey-(endy-databit*1.35-2.75)))
;   ;Place a pin for write_data- signals
;      fig  =  dbCreateRect(ccv  "metal1"  list(endx+2.8:Changey-(endy-3.35-databit*1.35)
endx+3.4:Changey-(endy-2.75-databit*1.35)))
;   pinname = buildString(list("BlkWData" pcExprToString(databit)) "_")
;   net = dbCreateNet(ccv pinname)
;   trm = dbCreateTerm(net pinname "output")
;   pin = dbCreatePin(net fig pinname)

   ;; layout the horizontal line for the current write_data+ bit lines
   dbCreateRect(ccv   "metal1"   list(endx+2.8-4.35:Changey-(endy-3.35-deltay-databit*1.35)
endx+4.9+4.8*(n-1):Changey-(endy-deltay-databit*1.35-2.75)))
   ;Place a pin for write_data+ signals
   fig  =  dbCreateRect(ccv  "metal1"  list(endx+2.8:Changey-(endy-3.35-deltay-databit*1.35)
endx+3.4:Changey-(endy-deltay-2.75-databit*1.35)))
   pinname = buildString(list("BlkWDataNg" pcExprToString(databit)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "output")
   pin = dbCreatePin(net fig pinname)
 )

 for(block 0 (2**y)-1
   caddr_startx=endx+4.2+(block*4.8*w)

   ;join word-size blocks together for write signal
   dbCreateRect(ccv   "poly1"   list(caddr_startx-1.15:Changey-(endy-1.15)   caddr_startx-
1.15+1.5+4.8*(w-1):Changey-(endy-0.8)))

   ;route write signal out of read-data access lines
   placex = endx+3.8+block*w*4.8
   placey = endy-3.3-1.35*2*w
   dbCreateRect(ccv  "metal2"  list(placex-0.4:Changey-(endy+0.95)+2.1  placex+0.4:Changey-
placey))
   ;Place a pin for write signals
   fig      =      dbCreateRect(ccv      "metal2"      list(placex-0.3:Changey-(placey+0.7)
placex+0.3:Changey-(placey+0.1)))
   pinname = buildString(list("BlkWrite" pcExprToString((2**y)-1-block)) "_")
   net = dbCreateNet(ccv pinname)
   trm = dbCreateTerm(net pinname "input")
   pin = dbCreatePin(net fig pinname)


   ;place a M1_M2 via at start of the write signal **
   viapt = placex:Changey-(placey+0.4)-2.7*(w-1)-4.05
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

   ;place a M1_M2 via at the end
   viapt = placex:Changey-(placey+0.4)
   via = dbCreateInst(ccv  viacv "via" viapt "R0")
   dbFlattenInst(via 1 t)

  ;add buffer at end
```

```
  buffpt = placex+1.35:Changey-(placey-5.25)
  buff = dbCreateInst(ccv  buffcv "buff" buffpt "R180")
  dbFlattenInst(buff 1 t)

  ;connect m2 to buffer
  dbCreateRect(ccv  "metal1"  list(placex-0.35:Changey-placey  placex+0.35:Changey-(placey-
0.6)))

  ;connect buffer to decoder
  if(modulo(block 2) == 1
  then
    dbCreateRect(ccv  "metal1"  list(placex-0.35:Changey-(placey-6.8)  placex+0.35:Changey-
(placey-9.25)))
  else
    dbCreateRect(ccv  "metal1"  list(placex-0.35:Changey-(placey-6.8)  placex+0.35:Changey-
(placey-8.45)))
  )
 )
 ;; Connect buffer to VDD & VSS
 ; connect vdd to vdd bus
 dbCreateRect(ccv   "metal2"   list(endx+0.95:Changey-(placey-6.3)   placex+2.35:Changey-
(placey-7)))

 ;connect vss together
 dbCreateRect(ccv   "metal2"   list(endx+2.95:Changey-(placey-0.55)   placex+2.35:Changey-
(placey-1.25)))
 ; add via
 viapt = endx+2.55:Changey-(placey-0.9)
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)
 ; connect vss to vss bus
 dbCreateRect(ccv "metal1"  list(endx+2.1:Changey-(placey-0.55) endx-0.15:Changey-(placey-
1.25)))
 ; add via
 viapt = endx-0.6:Changey-(placey-0.9)
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)
 ; connect vss to vss bus


 ;;;;;;;;;;;;;;  Layout column decoder for read signal ;;;;;;;;;;;;;;;;;;;;;
 for(addrline 0 y-1
   placex = endx+3.85
   placey = endy-4.35-(2.7*w)-(2.9*addrline)+1.45-9.2
   for(col 0 (2**y)-1
     ;check if a nfet should be place at this col
     if(modulo((col-(2**addrline)) (2**(addrline+1))) == 0
     then       ;  layout negative address lines
       ;  Place nfet decoders for this col
       nfetInst = placex+(4.8*w)*col:Changey-(placey-1.45)
       nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
       dbFlattenInst(nfInst 1 t)
       ; connect to next nfet in this column
       if(addrline == 0
       then
         dbCreateRect(ccv  "metal1"  list(placex-0.4+(4.8*w)*col:Changey-(placey+0.35-1.45)
placex+0.3+(4.8*w)*col:Changey-(placey+0.35+0.65-1.45)))
       else
         dbCreateRect(ccv  "metal1"  list(placex-0.4+(4.8*w)*col:Changey-(placey+0.35-1.45)
placex+0.3+(4.8*w)*col:Changey-(placey+0.35+2.2-1.45)))
       )
       ;;;;;;;;put substrate contact at every other address lines
       ;instantiate a sub contact next to nfet
       subInst = placex+(4.8*w)*col+1.5:Changey-(placey-1.45-0.2)+1.05
       scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
       dbFlattenInst(scInst 1 t)
       ; draw m1 to connect to next ** address line **
       dbCreateRect(ccv   "metal1"   list(placex-0.4+(4.8*w)*col:Changey-(placey-1.8-1.45)
placex-0.4+(4.8*w)*col-(4.8*w*(2**addrline)):Changey-(placey-1.2-1.45)))

       ;;;connect decoder's sub contact to VSS line
```

```
       ;connect to M2
       viapt =  placex+(4.8*w)*col+1.5-0.1:Changey-(placey-0.1)+2.8
       via = dbCreateInst(ccv  viacv "via" viapt "R0")
       dbFlattenInst(via 1 t)
       dbCreateRect(ccv    "metal2"    list(placex+(4.8*w)*col+1:Changey-(placey-0.4)+2.8
endx+3.9-0.45-0.65-3.1:Changey+2.8-(placey-0.3+0.5)))


     else if(modulo(col (2**(addrline+1))) == 0
     then      ; layout positive address lines
       ; place instance
       nfetInst = placex+(4.8*w)*col:Changey-placey
       nfInst = dbCreateInst(ccv nfetcv "nfetInst" nfetInst "R90")
       dbFlattenInst(nfInst 1 t)
       ;connect to next nfet for this col
       dbCreateRect(ccv    "metal1"    list(placex-0.4+(4.8*w)*col:Changey-(placey+1.7-2.9-
1.45) placex+0.3+(4.8*w)*col:Changey-(placey+1.8+0.75-2.9-1.45)))
       ;;;;;;;;put substrate contact at every other address lines
         ;instantiate a sub contact next to nfet
         subInst = placex+(4.8*w)*col+1.5:Changey-(placey-0.2)+1.05
         scInst = dbCreateInst(ccv subcv "subInst" subInst "R0")
         dbFlattenInst(scInst 1 t)
         ;;;connect decoder's substrate contact to VSS line
         ;let sub contact connect to M2
         viapt =  placex+(4.8*w)*col+1.4:Changey-placey+1.45
         via = dbCreateInst(ccv  viacv "via" viapt "R0")
         dbFlattenInst(via 1 t)
         dbCreateRect(ccv   "metal2"   list(placex+(4.8*w)*col+1:Changey-(placey-1.45-0.3)
endx+3.9-0.45-0.65-3.1:Changey-(placey-1.45-0.2+0.5)))
       ); if positive address line
     )
   );end for col

   ; for each address line route poly and place pins
   placey = placey-1.45

   ;route poly for positive and negative address lines
   dbCreateRect(ccv    "poly1"    list(startx-0.95:Changey-(placey+0.9)    endx+2.8:Changey-
(placey+0.55))) ;positive
   dbCreateRect(ccv   "poly1"   list(startx-0.95:Changey-(placey-0.55)   endx+2.8:Changey-
(placey-1.45+0.55))) ;negative

   ;place inverters to get both negative and positive address
   invpt =  startx-7.85:Changey-(placey+4.95)
   inv = dbCreateInst(ccv  invcv "inv" invpt "MXR90")  ; connect vdd
   dbFlattenInst(inv 1 t)
   ;connect sub contact to ground
   dbCreateRect(ccv   "metal2"   list(startx-1.8:Changey-(placey-0.35)   endx-0.3:Changey-
(placey+0.35)))

;   ;place pin for address signal
;          fig    =    dbCreateRect(ccv    "metal1"    list(startx+5.65:Changey-(placey+0.5)
startx+6.05:Changey-(placey+0.9)))
;   pinname = buildString(list("WriteAdr" pcExprToString(addrline)) "_")
;   net = dbCreateNet(ccv pinname)
;   trm = dbCreateTerm(net pinname "input")
;   pin = dbCreatePin(net fig pinname)

   ;route write-address lines to read-address lines
   dbCreateRect(ccv           "metal1"           list(startx+6.2:Changey-1.4-(placey-0.35)
7.85+(2.9*x)+1.4*addrline:Changey-0.7-(placey-0.35)))
   ;place via
   viapt =  8.25+(2.9*x)+1.4*addrline:Changey-1.05-(placey-0.35)
   via = dbCreateInst(ccv  viacv "via" viapt "MXR90")  ; connect vdd
   dbFlattenInst(via 1 t)
   dbCreateRect(ccv    "metal2"    list(7.85+(2.9*x)+1.4*addrline:Changey-0.7-(placey-0.35)
8.65+(2.9*x)+1.4*addrline:-7.4*m))

)
 ;;;;;;;;Route VSS line for write decoder Sub x
 ;extend VSS route
```

```
 dbCreateRect(ccv  "metal2"  list(endx-0.9:starty  endx-0.3:Changey+9.2-(endy-3.2-(2.7*w)-
(2.9*y))))
 ;extend VDD route
 dbCreateRect(ccv   "metal2"   list(endx+0.35:Changey-(endy+1.3)   endx+0.95:Changey-1.35-
(endy-11.65-(2.7*w))))

 ;connect well contact of inverter to vdd
 dbCreateRect(ccv  "metal1"  list(startx+5.05:Changey+12-(endy+2.45-(2.7*w))  startx+0.05-
4.8*(w-1):Changey+12-(endy+1.7-(2.7*w))))

 ;place pin for WEN (read enable) signal
 dbCreateRect(ccv   "metal1"   list(endx+4.8*(n/2):Changey+12-(endy-0.45-(2.7*w)-(2.9*y))
endx+0.8+4.8*(n/2):Changey+12-(endy-1-(2.7*w)-(2.9*y))))
 ;place a M1_M2 via at VSS line
 viapt = endx+0.4+4.8*(n/2):Changey+12-(13.35+endy-14.75-(2.7*w)-(2.9*y))
 via = dbCreateInst(ccv  viacv "via" viapt "R0")
 dbFlattenInst(via 1 t)
 fig  =  dbCreateRect(ccv  "metal1"  list(endx+0.15+4.8*(n/2):Changey+12-(13.35+endy-14.75-
(2.7*w)-(2.9*y)) endx+0.65+4.8*(n/2):Changey+12-(13.35+endy-14.35-(2.7*w)-(2.9*y))))
 pinname = "WEN"
 net = dbCreateNet(ccv pinname)
 trm = dbCreateTerm(net pinname "input")
 pin = dbCreatePin(net fig pinname)

) ;; end procedure
```

APPENDIX

C Bibliography

1. A. Karandikar and K. Parhi, "Low Power SRAM Design using Hierarchial Divided Bit-Line Approach," Proceedings of the International Conference on Computer Design, pp. 82-88, Oct. 1998.

2. B. Bhaumik, P. Pradhan, G. Visweswaran, R. Varambally, and A. Hardi, "A Low Power 256 KB SRAM Design," Proceedings of the IEEE International Conference on VLSI Design, pp. 67-70, 1999.

3. F. Vargas and M. Nicolaidis, "SEU-Tolerant SRAM Design Based on Current Monitoring," International Symposium on Fault Tolerant Computing, pp.106-115, 1994.

4. H. Nambu, K. Kanetani, Y. Idei, T. Masuda, K. Higeta, M. Ohayashi, M. Usami, K. Yamaguchi, T. Kikuchi, T. Ikeda, K. Ohhata, T. Kusunoki, and N. Homma, "A 0.65-ns, 72-kb ECL-CMOS RAM Macro for a 1-Mb SRAM," *IEEE Journal of Solid-State Circuits*, Vol. 30, No. 4, Apr. 1995.

5. H. Tran, "Demonstration of 5T SRAM and 6T Dual-Port RAM Cell Arrays," 1996 Symposium on VLSI Circuits, pp 68-69, June 1996.

6. J. Caravella, "A 0.9V, 4K SRAM For Embedded Applications," IEEE 1996 Custom Integrated Circuits Conference, pp. 119-122, 1996.

7. J. Caravella, "A Low Voltage SRAM For Embedded Applications," *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 3, pp. 428-432, Mar. 1997.

8. J. Tsaur, C. Jih, H. Tsaur, and J. Kuo, "Scaling Consideration of BiCMOS SRAMs," 1991 IEEE International Symposium on Circuits and Systems, 1991, Vol. 4, pp. 2116-2119, 1991.

9. J. Wang and H. Lee, "A New Current-Mode Sense Amplifier for Low-Voltage Low-Power SRAM Design," Proceedings of the Annual IEEE ASIC Conference and Exhibit, pp. 163-167, 1998.

10. J. Wang, P. Yang, and W. Tseng, "Low-Power Embedded SRAM Macros with Current-Mode Read/Write Operations," Proceedings of the International Symposium on Low Power Electronics and Design, Digest of Technical Papers, pp. 282-287, 1998.

11. K. Itoh, A. Fridi, A. Bellaouar, M. Elmasry, “A Deep Sub-V, Single Power-Supply SRAM Cell with Multi-V_T, Boosted Storage Node and Dynamic Load,” 1996 Symposium on VLSI Circuits, pp. 132-133, 1996.

12. K. Koyama, O. Ikenaga, T. Takigawa, Y. Kobayashi, S. Sakamoto, and S. Watanabe, “Shape Data Operations for VSB EB Data Conversion Using CAD Tools,” *Japanese Journal of Applied Physics*, Part 1, Vol. 28, No. 11, pp. 2329-2332, 1989.

13. K. Kumagai, T. Yamada, H. Iwaki, H. Nakamura, H. Onishi, Y. Matsubara, K. Imai, and S. Kurosawa, “A New SRAM Cell Design Using 0.35 μm CMOS/SIMOX Technology,” Proceedings 1997 IEEE International SOI Conference, pp. 174-175, Oct. 1997.

14. K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano, “A Low Power SRAM using Auto-Backgate-Controlled MT-CMOS,” Proceedings of the International Symposium on Low Power Electronics and Design, Digest of Technical Papers, pp. 293-298, 1998.

15. K. Toh, C. Chuang, S. Wiedmann, and K. Chin, “A 1.9ns/6.3W/256Kb Bipolar SRAM Design” IEEE 1990 Bipolar Circuits and Technology Meeting, pp.71-74, 1990.

16. L. Jacunski, S. Doyle, D. Jallice, N. Haddad, and T. Scott, “SEU Immunity: The Effects of Scaling on the Peripheral Circuits of SRAMs,” *IEEE Transactions on Nuclear Science*, Vol. 41, No. 6, pp. 2272-2276, Dec. 1994.

17. P. Fung, H. Tran, and D. Scott, “Impact of BiCMOS Technology on SRAM Circuit Design,” 1989 BiCMOS Technology on SRAM Circuit Design, pp. 310-313, 1989.

18. P. Gee and J. Tou, “A Diffused CMOS SRAM Compiler for Gate-Arrays,” Proceedings of the 34th Midwest Symposium on Circuits and Systems, Vol. 2, pp. 807-810, 1992.

19. S. Flannagan, P. Pelley, N. Herr, B. Engles, T. Feng, S. Nogle, J. Eagan, R. Dunnigan, L. Day, and R. Kung, “8-ns CMOS 64K X 4 and 256K X 1 SRAM’s,” *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, Oct. 1990.

20. T. Sakural, “High-Speed Circuit Design with Scaled-Down MOSFET’s and Low Supply Voltage,” 1993 IEEE International Symposium on Circuits and Systems, Vol. 3, pp. 1487-1490, May 1993.

21. W. Herndon, “Trends in BIPOLAR Static Random Access Memory (SRAM) Design,” Proceedings of the 1989 Bipolar Circuits and Technology Meeting, pp. 203-208, 1989.

22. H. Nambu, K. Kanetani, K. Yamasaki, K. Higeta, M. Usami, Y. Fujimura, K. Ando, T. Kusunoki, K. Yamaguchi, and N. Homma, “A 1.8-ns Access, 550-MHz, 4.5-Mb

CMOS SRAM," IEEE Journal of Solid-State Circuits, pp. 1650-1656, Vol. 33, No. 11, November 1998.

23. H. Lee, "Design of Ultra Low Power Pseudo-Asynchronous SRAM," ASIC Conference, September 1999.

24. K. Mai, T. Mori, B. Amrutur, R. Ho, B. Wilburn, M. Horowitz, I. Fukushi, T. Izawa, and Shin Mitarai, "Low-Power SRAM Design Using Half-Swing Pulse-Mode Techniques," IEEE Journal of Solid-State Circuits, pp. 1659-1669, Vol. 33, No. 11, November 1998.

25. H. Morimura, S. Shigematsu, and S. Konaka, "A Shared-Bitline SRAM Cell Architecture for 1-V Ultra Low-Power Word-Bit Configurable Macrocells," International Symposium on Low-Power Design & Electronics, pp. 12-17, 1999.

26. H. Yamaguchi, T. Iwata, H. Akamatsu, and A. Matsuzawa, "A 0.5V/100MHz Over-Vcc Grounded Data Storage (OVGS) SRAM Cell Architecture with Boosted Bit-line and Offset Source Over-Driving Schemes," International Symposium on Low-Power Design & Electronics, pp. 49-54, 1996.

27. N. Tzartzanis and W. Athas, "Energy Recovery for the Design of High-Speed, Low-Power Static RAMs," International Symposium on Low-Power Design & Electronics, pp. 55-60, 1996.

28. H. Morimura and N. Shibata, "A 1-V 1-Mb SRAM for Portable Equipment," International Symposium on Low-Power Design & Electronics, pp. 61-66, 1996.

29. J. Alowersson and P. Andersson, "SRAM Cells for Low-Power Write in Buffer Memories," Procedings of the 1995 Symposium on Low Power Electronics, San Jose, CA, October 9-11, 1995.

30. M. Izumikawa, H. Igura, K. Furuta, H. Ito, H. Wakabayashi, K. Nakajima, T. Mogami, T. Horiuchi, and M. Yamashina, "A 0.25 μm CMOS 0.9 V 100-MHz DSP Core," IEEE Journal of Solid-State Circuits, vol. 32, pp. 52-61, Jan. 1997.

31. H. Nambu, K. Kanetani, K. Yamasaki, K. Higeta, M. Usami, Y. Fujimura, K. Ando, T. Kusunoki, K. Yamaguchi, and N. Homma, "A 1.8-ns Access, 550-MHz, 4.5-Mb CMOS SRAM," IEEE Journal of Solid-State Circuits, vol. 33, pp. 1650-1656, no. 11, November 1998.

32. T. Chappell, B. Chappell, S. Schuster, J. Allan, S. Klepner, R. Joshi, and R. Franch, "A 2-ns cycle, 3.8-ns access 512 kb CMOS ECL SRAM with a fully pipelined architecture," IEEE Journal of Solid-State Circuits, vol. 26, pp. 1577-1584, Nov. 1991.

33. T. Mori, B. Amrutur, K. Mai, M. Horowitz, I. Fukushi, T. Izawa, and S. Mitarai, "A 1 V 0.9 mW at 100 MHz 2Kx16b SRAM utilizing a half-swing pulsed-decoder and

write-bus architecture in 0.25 mm dual-V_t CMOS," IEEE Journal of Solid-State Circuits, pp. 354-355, Feb. 1998.

34. S. Kang, Y. Leblebici, *CMOS Digital Integrated Circuits*, New York, McGraw-Hill, 1999.

35. J. Rabaey, *Digital Integrated Circuits*, New York, Prentice Hall, 1996.

36. R. Baker, H. Li, and D. Boyce, *CMOS: Circuit Design, Layout, and Simulation*, New York, IEEE Press, 1999.

37. J. Kuo and J. Lou, *Low-Voltage CMOS VLSI Circuits*, New York, 1999.

38. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, New York, Addison-Wesley, 1993.

39. D. Johns and K. Martin, *Analog Integrated Circuit Design*, New York, Wiley, 1997.

Vita

Meenatchi Jagasivamani was born on July 31, 1979 in Madras, India. In May 1998, she earned the degrees of Bachelor of Science in Computer Engineering and Bachelor of Science in Electrical Engineering from Virginia Polytechnic Institute and State University. She joined the Electrical and Computer Engineering at Virginia Polytechnic Institute and State University in August 1998. After graduation, she began employment with Intel at Chandler, Arizona as a technology engineer.