UNIVERSITY OF CALIFORNIA

SANTA CRUZ

OpenRAM: An Open-Source Memory Compiler

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

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jeffrey T. Butera

September 2013

The Thesis of Jeffrey T. Butera
is approved:

Professor Matthew Guthaus, Chair

Professor Jose Renau

Professor Patrick E. Mantey

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

OpenRAM: An Open-Source Memory Compiler

by

Jeffrey T. Butera

In academia, many Application Specific Integrated Circuits and System-on-Chip design methodologies are limited by the availability of memories. As process technologies shrink, the size and number of memories on a chip are constantly increasing and memory designs become a more significant part of the overall system performance, efficiency, and cost. Random-Access Memories can be time consuming and tedious to custom design, and there are not many options for automating this process. Process design kits from foundries and vendors do not include memory compilers and commercial solutions require expensive licenses and are often un-modifiable and process specific. This thesis introduces OpenRAM, an open-source memory compiler and characterization methodology. The main objective of the OpenRAM compiler is to promote memory research in academia by providing a flexible and portable platform for generating and verifying memory designs across different technologies. Currently, the compiler generates GDSII layout and Spice netlists for single-port SRAM's using the FreePDK 45nm process design kit, and provides timing/power characterization through Spice simulation. Verification of OpenRAM designs in both 130nm (IBM 8RF) and 180nm (IBM 7SF) technologies are in progress.

I would like to express my gratitude to Professor Matthew Guthaus for advising me throughout my graduate studies and driving the OpenRAM project. I would also like to thank the other members of the OpenRAM group: Bin Wu, James Stine, Brian Chen, Ping-Yao Li, and Manju Subbarayappa. OpenRAM is a group effort and my work would not have been completed without their valuable contributions.

I also extend my thanks to the members of the UC Santa Cruz VLSI Design and Automation Group (Rajsaktish Sankaranarayanan, Riadul Islam, Benjamin Lacara, and Hany Famy) for being a sounding board for ideas and insight.

Lastly, a special thanks to my parents for their support and encouragement throughout my academic endevours.

1 Introduction

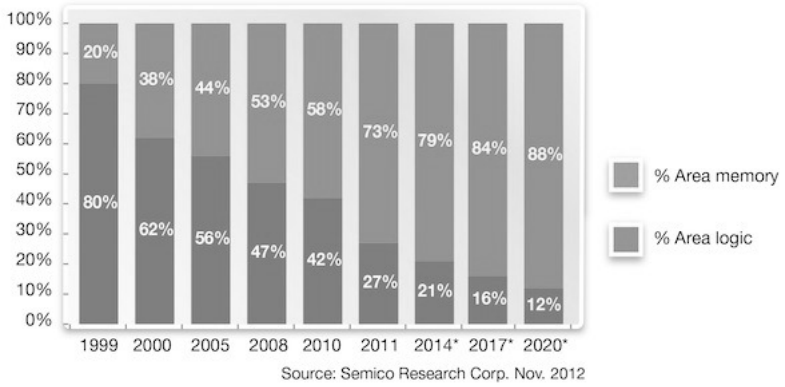
Static Random-Access Memory, or SRAM, has many applications and has become the standard for embedded memories in ASIC, SOC, and microprocessor designs. As process technologies shrink, the size and number of memories on a chip is constantly increasing and memory designs are becoming a significant part of overall system performance, efficiency, and cost. Currently, embedded memories can occupy up to 75% of the total chip area (Figures 1b and 1c). It has been reported that by 2020 the total memory area will increase closer to 90% for SOC's(Figure 1a). Consider the Itanium 2 processor with a 6MB on-die L3 cache from Intel (Figure 1c) as an example. This is a 10 year old processor, developed using a 130-nm technology, and the on-die memory dominates over 70% of the chip's area.

With current technologies pushing below the 20-nm mark, even larger and more dense memories are being placed on-chip. These newer, smaller technologies introduce very specific design challenges. The effects of variability, static power, stability, and supply noise margins are magnified as technologies shrink. Specific countermeasures must be taken to combat these effects, while still not significantly increasing the area, power, or speed of the SRAM. Consequently, SRAM's have become one of the most challenging and time consuming components to design in an integrated circuit.

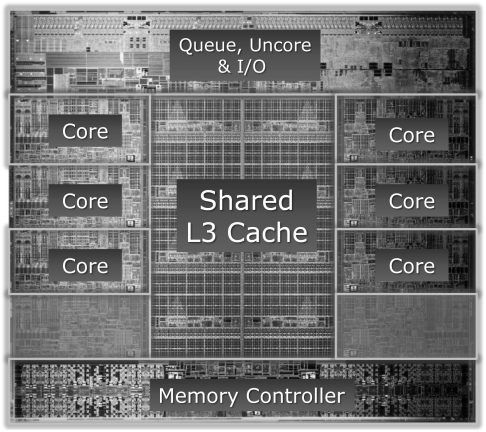
One redeeming factor of the SRAM is that it can have a very regular structure consisting of cells that can be replicated many times across a single design. This promotes full custom design and encourages the use of automation techniques such as compilers to automate the design process.

Memory compilers are commonplace in industry. Two main flavors of memory compilers exist: third party compilers released with standard cell libraries by vendors, and proprietary compilers that are considered in-house intellectual property by companies. These compilers usually have the capability of generating single and dual port RAM's as well as ROM's (read only memories) and CAM's (content addressable memories). The main drawback with commercial compilers is that they require expensive licenses and may not allow customization or changes to source code. Examples of industry memory compilers are: Global Foundries' design kit[6], ARM's embedded IP memories[2], Synopsys DesignWare memory compiler[21], Dolphin Technology's memory IP[23], Faraday Technology's library and memory compiler[22].

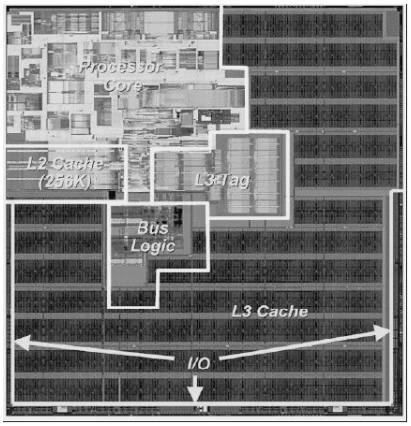
Due to funding restrictions and customization limitations, commercial memory compilers are not a feasible solution for researchers in academia. Outside of commercial solutions,



(a)



(b)



(c)

Figure 1: a)Trend of memory and logic area for SOC's (Semico Research). b)Die photo of the Intel i7 Sandy Bridge[18]. c)Die photo of the Itanium 2 from Intel[17].

the only alternative is full-custom design. This makes it difficult and time consuming for researchers to prototype and verify new circuits and methodologies above the cell level. In order to correctly verify certain aspects of a design, such as scalability and robustness, memory designs should be considered at the system level. In attempt to combat these issues and provide a common infrastructure for memory research in academia, we introduce the OpenRAM memory compiler.

OpenRAM is an open-source memory compiler and characterization methodology. The main objectives of the compiler are: easy generation of memory arrays, being portable across many technologies and platforms, providing timing/power characterization of designs, being independent of commercial tools, and providing silicon verified designs. The OpenRAM project is funded by the National Science Foundation and is a collaboration between the VLSI Design and Automation group (run by Prof. Matthew Guthaus) at the University of California, Santa Cruz, and the VLSI Computer Architecture Research group (run by Prof.

James Stine) at Oklahoma State University.

This thesis will provide an in depth discussion of the OpenRAM memory compiler, including its software implementation, features, and use. The thesis will also highlight the specific contributions of the author. These contributions include: the data structure hierarchy of the compiler, scripts to dynamically generate GDSII layouts and Spice netlists of the various blocks of the SRAM, a snap-to-grid function for the dynamic layouts, and the Spice memory characterizer.

The thesis is organized as follows. Section 2 provides a background of the SRAM architecture and operation. Section 3 discusses the implementation of the OpenRAM memory compiler and details its main features. Section 4 highlights the author's specific contributions to the project and provides preliminary results from the characterizer. Finally, Section 5 summarizes the OpenRAM memory compiler and discusses the future direction of the project.

2 Background

Before discussing the details of the OpenRAM memory compiler this section will give an overview of a typical SRAM architecture, its operation, and highlight several design challenges specifically associated with SRAM's.

2.1 SRAM Architecture

SRAM's typically consist of an array of memory cells with peripheral circuits and control logic. Figure 2 depicts the memory array (large block in the center) as well as the other main blocks: the address decoder, word line drivers, column multiplexer, precharge circuitry, write drivers, sense amplifier, and control logic. The following sub-sections explain the operation of each individual block within the SRAM, followed by a high level explanation as to how these different blocks interact to function as a memory device. It is important to note that the circuits described below are the ones that are used in the first release of the OpenRAM memory compiler. By no means is this an exhaustive list of the possible circuits that can be adapted into a SRAM architecture.

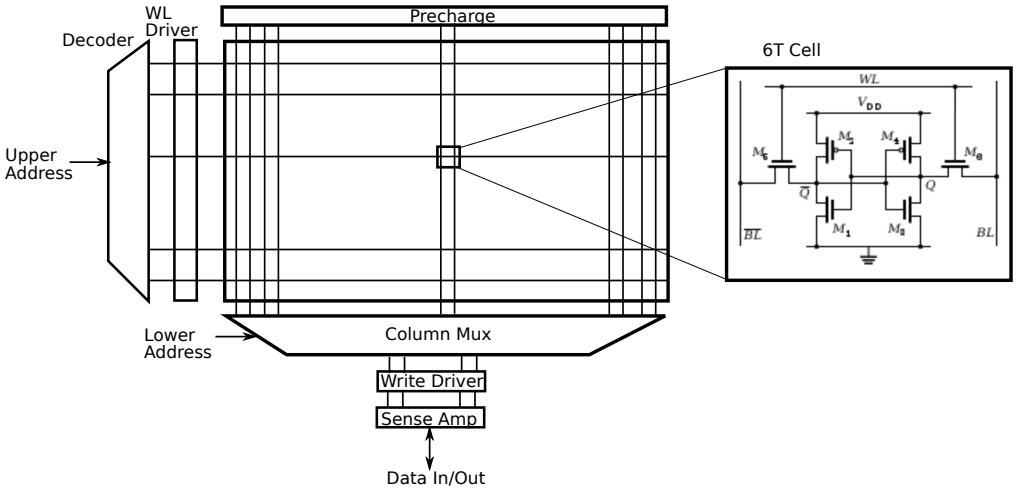


Figure 2: Single Port SRAM Architecture

2.1.1 The 6T Cell and Memory Array

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

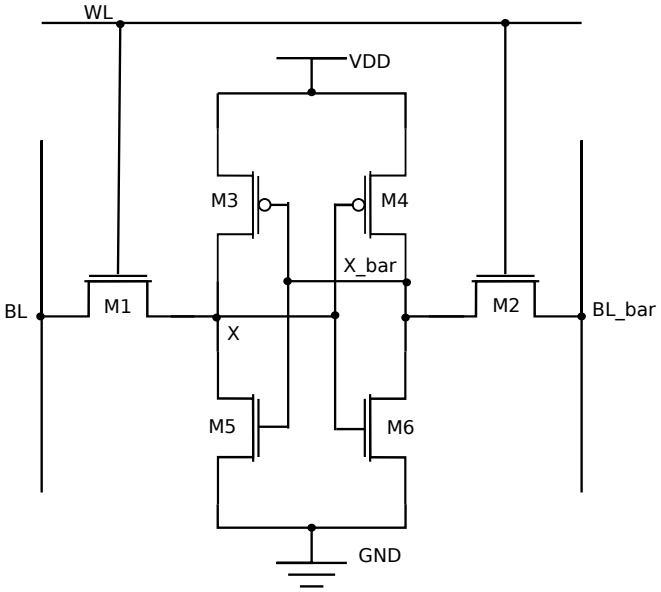


Figure 3: Schematic of 6T cell.

The 6T cell can be accessed to perform the two main operations associated with memory: reading and writing. When a read is to be performed, both bit lines are precharged to `Vdd`.

This precharging is done during the first half of the read cycle and is handled by the precharge circuitry (see Section 2.1.2). In the second half of the read cycle the word line is asserted, which enables the access transistors. If a 1 is stored in the cell then `BL_bar` is discharged to `Gnd`, and `BL` is pulled up to `Vdd`. Conversely, if a 0 is stored, then `BL` is discharged to `Gnd` and `BL_bar` is pulled up to `Vdd`. While performing a write operation, both bit lines are also precharged to `Vdd` during the first half of the write cycle. Again, the world line is asserted, and the access transistors are enabled. The value that is to be written into the cell is applied to `BL`, and its complement is applied to `BL_bar`. The drivers that are applying the signals to the bit lines must be appropriately sized so that the previous value in the cell can be overwritten (see Section 2.1.6).

The transistors in the cell must be carefully sized to ensure reliable operation. For a read operation, the `M5` and `M6` must be sized larger than access transistors `M1` and `M2`. This is necessary because when the word line is asserted, both `X` and `X_bar` are initially pulled up to the precharge value. Assuming that a 1 is stored at `X`, `X_bar` must remain 0 regardless of the voltage rise experienced when the word line is asserted. In order to prevent the value in the cell from flipping, the resistance of the access transistors must be larger than that of `M5` and `M6`. During a write operation, the value stored in the cell is being overwritten. This means that `M1` and `M2` must be strong enough to overpower the feedback inverter and must be sized larger than `M3` and `M4`[5],[13].

As previously stated, 6T cells are tiled together in both the horizontal and vertical directions to make up the memory array. This means that the memory cell should be made as small as possible so that the array can be as dense as possible. The size of the memory array is directly related to the number of words and the size of the words that will need to be stored in the RAM. For example, an 8kB memory with a word size of eight bits will consist of eight columns and 1024 rows. It is common practice to keep the aspect ratio of memory arrays as square as possible. This helps ensure that the bit lines do not become too long, which can increase the bit line capacitance, slow down the operation, and lead to increased leakage. To make the design more square, multiple words can share rows by interleaving the bits of each word. If the 8Kb memory were rearranged to allow two words per row, then the array would have 16 columns and 512 rows. Geometrically distributing the bits of each word (by allowing multiple words per row with interleaved bits) improves yield and soft-error robustness. The interleaving distance of the bits for a word can even be optimized to improve power consumption[12]. Redundant rows and columns can also be

added to improve yield and robustness[14],[7],[11]. Redundancy can add some complexity to other peripheral circuits, specifically the column multiplexer which handles selecting the appropriate column for each bit in a word (see Section 2.1.4).

Other types of memory cells, such as 7T, 8T and 10T cells, can be used as alternatives to the 6T cell. Each of these cells offer certain advantages. For example, the 8T cell provides higher read and write noise margins in comparison to the 6T cell[3]. The 10T and 12T cells provide improved soft error tolerance and operation at lower supply voltages[4],[10]. The main disadvantage of these other cells is increased area.

2.1.2 Precharge Circuitry

The precharge circuit is used to precharge both bit lines during the first phase of the clock in read and write operations and is depicted in Figure 4. It is a fairly simple circuit that consists of three PMOS transistors. The input signal to the cell, `PCLK`, enables all three transistors. `M1` and `M2` charge `BL` and `BL_bar` to `Vdd` and `M3` helps to equalize the voltages seen on the bit lines. The bit line voltages are equalized so that when the precharge phase ends and one of the bit lines experiences a voltage drop, the sense amplifier can more quickly sense the voltage difference between the two bit lines (see Section 2.1.5).

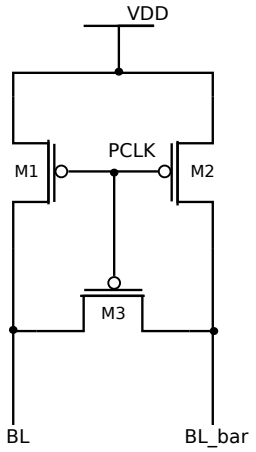


Figure 4: Schematic of a single precharge cell.

2.1.3 Address Decoder and Word line Drivers

The address decoder takes the row address bits from the address bus as inputs, and asserts the appropriate word line in the row that data is to be read from or written to. An n-bit input can control 2^n word lines. Figure 5 illustrates a 2-to-4 dynamic NAND decoder which

operates as follows: during the first phase of the clock (i.e. while clock is low), the `PCLK` signal enables the PMOS transistors which precharge all of the internal word lines (to the left of the output inverters) to `Vdd`. During the second phase of the clock, the PMOS transistors are disabled. Based on the input address, a specific internal word line is pulled down to ground[1]. The output inverters ensure that no word lines are asserted during the precharge phase and that only one is asserted during the second phase.

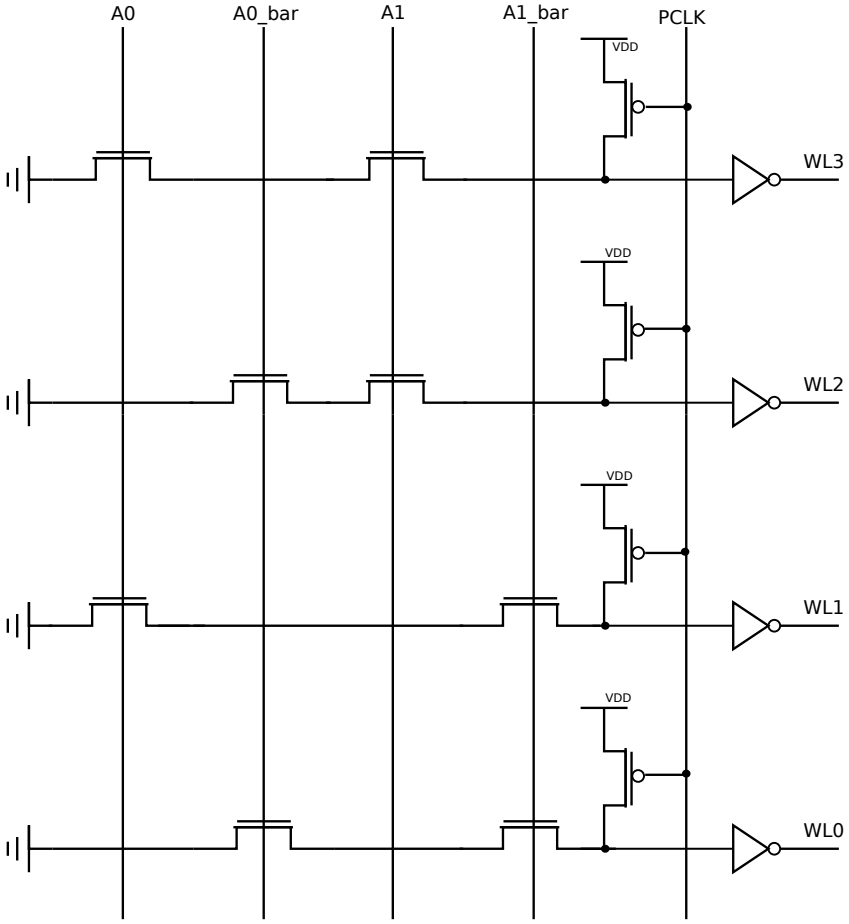


Figure 5: Schematic of 2-to-4 NAND decoder.

Asserted WL	Inp1	Inp2	Binary
WL0	A0_bar	A1_bar	00
WL1	A0	A1_bar	01
WL2	A0_bar	A1	10
WL3	A0	A1	11

Table 1: Truth table for 2-to-4 NAND decoder.

The truth table for the 2-to-4 decoder is depicted by Table 1. From the table it can be seen that the inputs are connected to the address bits in a binary reduction pattern. This

pattern can be exploited to easily scale the dynamic decoder up to handle an array with more rows. Word line drivers are inserted, as buffers, in-between the word line output of the address decoder and the input of the 6T cell. The word line drivers ensure that as the size of the memory array increases, and the word line capacitance increases, the signal is still able to turn on the access transistors in all 6T cells.

2.1.4 Column Multiplexer

The column multiplexer takes in n-bits from the address bus and can select 2^n bit line pairs associated with one word in the memory array. The schematic for a 4-to-1 tree multiplexer is shown in Figure 6. This type of tree multiplexer is bi-directional and is used for both the read and write operations; it connects the bit lines of the memory array to both the sense amplifier and the write driver.

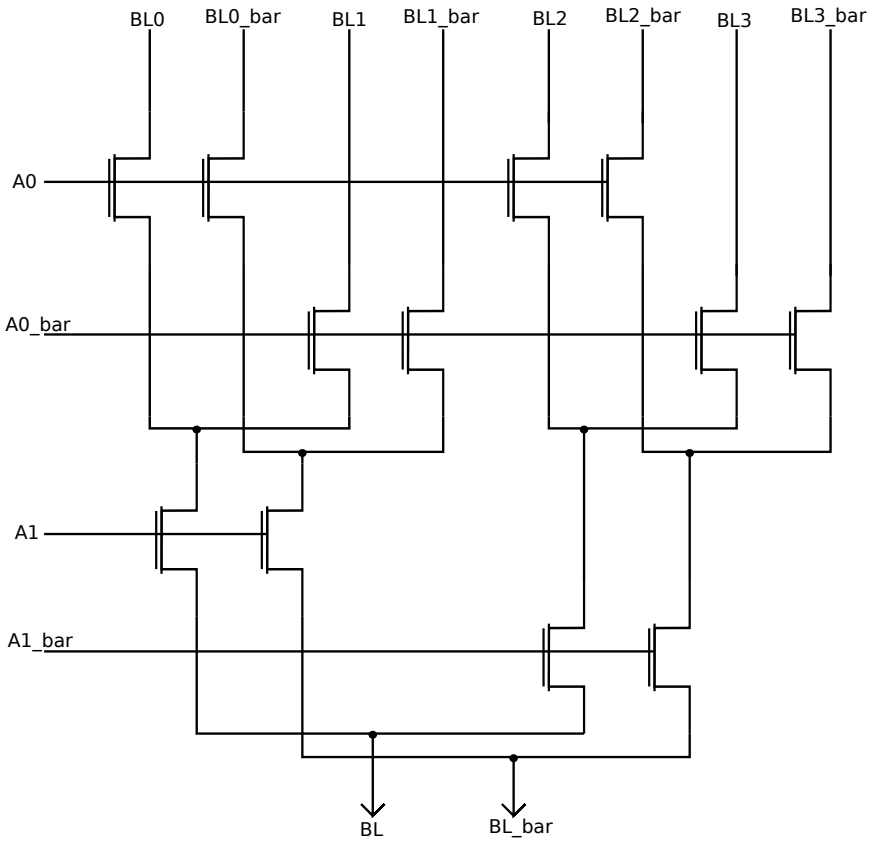


Figure 6: Schematic of 4-to-1 tree column mux that passes both of the bit lines.

As seen in Figure 6, the column mux is built of NMOS transistors in a tree-like structure. The depth of the decoder is determined based on the number of words per row in the memory array. The most basic column mux has a depth one which means that there are two words per row. If there is only one word per row in the array, then no column mux is needed. As

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

Table 2: Binary reduction pattern for 4-to-1 tree column mux.

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

Figure 6 illustrates a column mux with a depth of two. This means that there are four words per row in the memory array and two select bits from the address bus are needed to choose the bit line pairs for one of the four words. A binary reduction pattern, shown in Table 2, is used to select the appropriate bit lines. In level one, `A0`, and its complement `A0_bar`, select either the even numbered words or the odd numbered words in the row. In level two, the most significant bit `A1`, and its complement `A1_bar`, then select one of the words passed down from the previous level. Relative to other column mux designs, such as pass transistor based decoders with NOR pre-decoders, the tree mux uses significantly less devices. However, this type of design can provide poor performance if a large decoder with many levels is needed. The delay of of a tree mux quadratically increases with each level[16]. Due to this fact, other types of column decoders should be considered for larger memory arrays.

2.1.5 Sense Amplifier

The sense amplifier is used to sense the difference between the bit lines (`BL` and `BL_bar`) while a read operation is performed. A sense amplifier is necessary to recover the signals from the bit lines because they do not experience full voltage swing. As the size of the memory array grows, the capacitive load of the bit lines increase and the voltage swing is limited by the small memory cells driving this large load. A differential sense amplifier is used to sense the small voltage difference between the bit lines and accelerates the read operation[16].

The schematic for the sense amp is shown in Figure 7. The sense amplifier is enabled by the SCLK signal, which initiates the read operation. Before the sense amplifier is enabled, the bit lines are precharged to `Vdd` by the precharge unit. When the sense amp is enabled, one of the bit lines experiences a voltage drop based on the value stored in the memory cell.

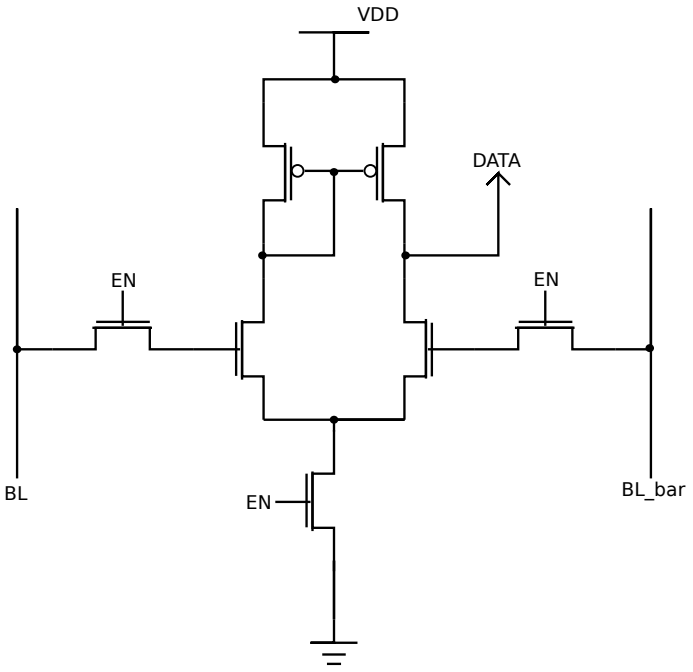


Figure 7: Schematic of a single sense amplifier cell.

If a zero is stored, the `BL` voltage drops. If a one is stored, the `BL_bar` voltage drops. The voltage difference between `BL` and `BL_bar` is sensed and the output signal is then taken to a true logic level and latched to the data bus.

2.1.6 Write Driver

The write driver is used to drive the input signal into the memory cell during a write operation. It can be seen in Figure 8 that the write driver consists of two tristate buffers, one inverting and one non-inverting. It takes in a data bit, from the data bus, and outputs that value on the bit line, and its complement on bit line bar. Both tristates are enabled by the `EN` signal. The bit lines always need to be complements to ensure that correct data is stored in the 6T cell. Also, the drivers need to be appropriately sized as the memory array grows and the bit line capacitance increases.

2.1.7 Control Logic

The control circuitry ensures that the SRAM operates as intended during a read or write cycle by enabling the necessary structures in the SRAM. Typically, the control logic takes three active low signals as inputs: chip select bar (`CSb`), output enable bar (`OEb`), and write enable bar (`WEb`). `CSb` enables the entire SRAM chip. While `CSb` is low, the appropriate

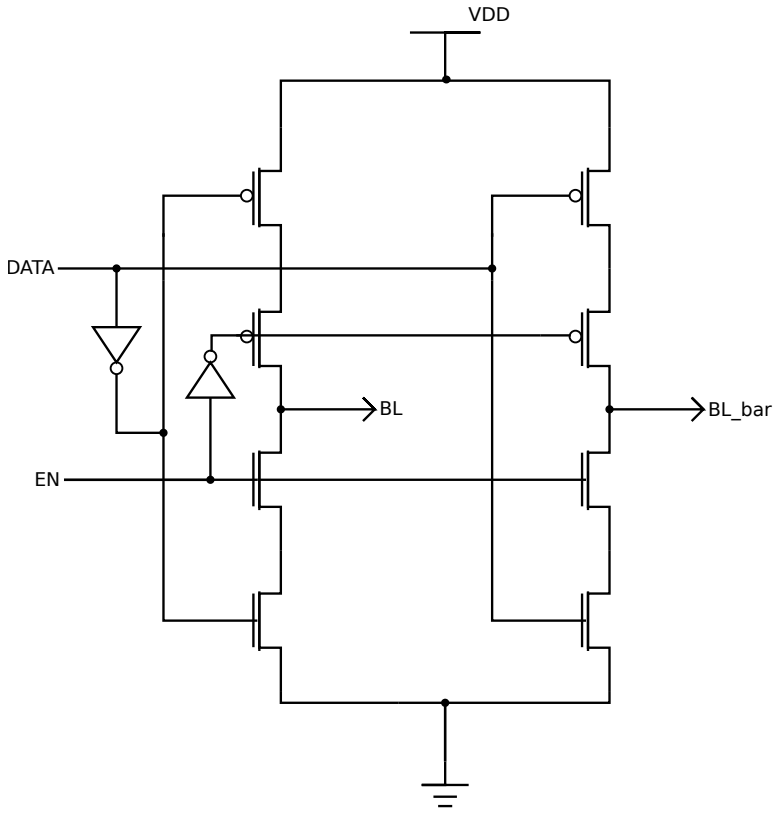


Figure 8: Schematic of a write driver cell, which consists of two tristates (non-inverting and inverting) to drive the bit lines.

control signals are generated and sent to the architecture blocks. Conversely, if `CSb` is high then no control signals are generated and various structures are turned off or disabled. The `OEb` signal signifies a read operation; while it is low the value seen on the data bus will be an output from the memory. Similarly, the `WEb` signal signifies a write operation and the data seen on the data bus is meant to be stored.

As seen in Figure 9, these three control signals need to be combined with the global clock signal to generate local signals used to enable or disable structures based on the operation. The `SCLK` signal is used to enable the sense amplifier during a read operation. The `WD_EN` signal enables the write driver during a write to the memory. Table 3 shows the truth table for the control logic. The `SCLK` signal to enable the sense amplifier is true when $\neg(CSb \vee OEb) \wedge CLK$. Similarly, the write driver enable signal `WD_EN`, is true when $\neg(CSb \vee WEb) \wedge CLK$. These signals are "anded" with the clock because the circuits should only be enabled after the precharging of the bit lines has ended.

The control logic also generates a second, delayed clock signal to help with synchronization. The delayed clock, `DCLK`, is created by feeding the normal clock through a delay chain,

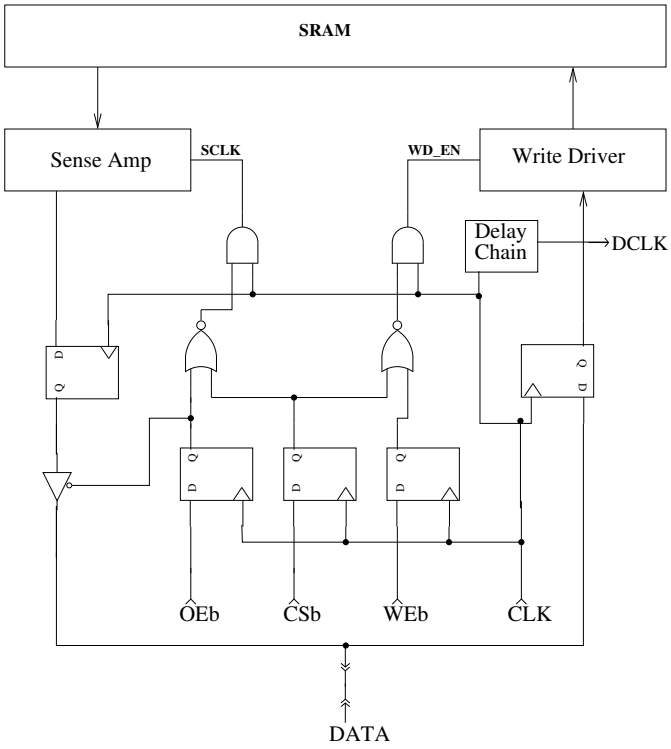


Figure 9: Control Logic Schematic

or a series of buffers. This DCLK signal is used as the precharge signal for the address decoder. The decoder needs a delayed clock signal to ensure that the precharging does not stop until the address inputs have been registered. If the precharging stops before the address inputs are ready, then the wrong word line could be temporarily asserted and data can be destroyed.

	Inputs				Outputs		
	CSb	CLK	OEb	WEb	SCLK	WD_EN	TRI_EN
Disabled	1	?	?	?	0	0	0
Precharge	0	0	?	?	0	0	0
Read	0	1	0	1	1	0	1
Write	0	1	1	0	0	1	0

Table 3: Truth table for the control signals.

In a synchronous SRAM, all of the input control signals are latched with flip-flops (see Figure 10 for master-slave flip-flop schematic). This ensures that the signals stay valid for the entire clock cycle and that no structures are disabled prematurely during operation. During a read operation, the latched `OEb` signal is inverted and used as an enable for the tristates that drive the outputs onto the data bus.

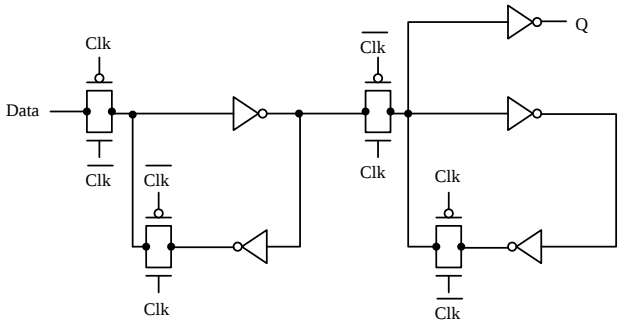


Figure 10: Schematic for simple master-slave flip-flop used in a synchronous SRAM

2.2 SRAM Operation

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

2.2.1 Signals

The typical top-level signals for a SRAM are:

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

The internally generated control signals are:

- `PCLK` - enables the precharge unit.
- `SCLK` - enables the sense amplifier during a read operation.
- `WD_EN` - enables the write driver during a write operation.
- `D_CLK` - the delayed clock used for the NAND decoder precharge.

2.2.2 Timing Considerations

The main timing considerations for an SRAM are the setup and hold times for the input signals, the memory read and write delays, and the minimum clock period. The setup and hold times are defined as the time that an input signal needs to be stable either before (setup) or after (hold) the clock edge that triggers the memory operation. The write delay is the time that it takes from the clock edge of a write operation until valid data has been driven into a memory cell. The read delay is defined as the time elapsed from the clock edge until valid data appears as an output of the sense amplifier. The minimum clock period can be calculated using Equation 1.

$$T \geq max(t_{delay}) + t_{setup} \tag{1}$$

The clock period, `T`, must be greater than or equal to the sum of the maximum delay, for either the read or write operation, and the setup time. In a synchronous SRAM, all inputs are registered using flip-flops. In this case, the setup and hold times of the memory are equal those of the flip-flops being used. If the clock period decreases below `T`, then the read or write operations may be interrupted and unable to be successfully completed.

2.2.3 Read Operation

Figure 11 displays the timing diagram for the SRAM read operation. It highlights the setup and hold times for each signal as well as the memory read time. The list below provides a step-by-step description of what is happening internally in the SRAM[9].

Read Operation:

1. Before the clock transition (low to high) that initiates the read operation:
   (a) `CS_B` set low to enable the chip (setup time).
   (b) `OE_B` set low for the control logic to generate the `SCLK` signal (setup time).
   (c) `WE_B` set high to disable the write driver (setup time).
   (d) The row and column addresses must be applied to the `ADDR` bus (setup time).
   (e) Precharge bit lines to `Vdd` and the decoder while `PCLK=CLK=0`.
2. On the rising edge of the clock:
   (a) `ADDR` and control signals are latched into flip-flops (hold time).

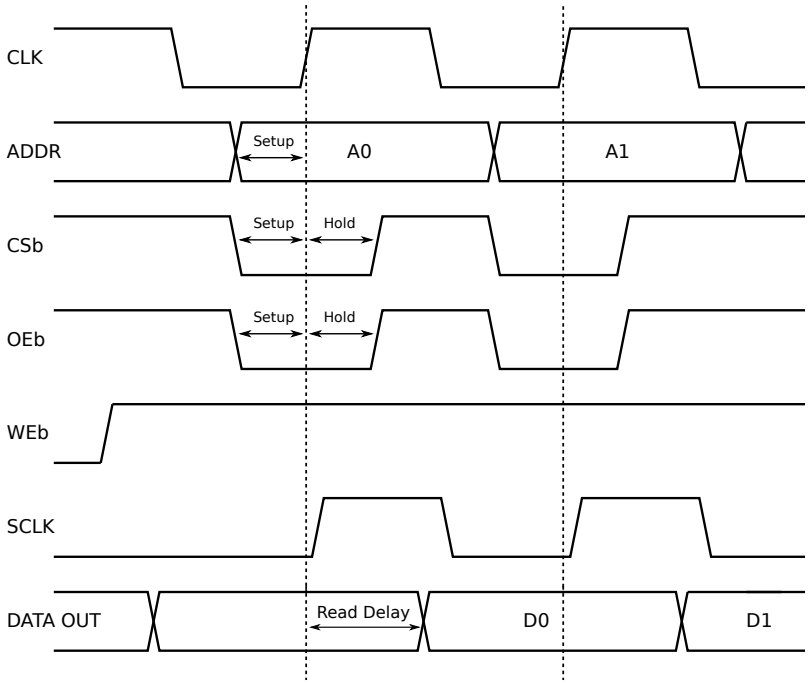


Figure 11: Timing diagram for read operation showing the setup, hold, and read times.

(b) The precharging of the bit lines and row decoder stops.

(c) The decoder and column mux select the row and columns for the word that is to be read.

(d) Once the word line has been asserted, the value stored in the memory cells pulls down one of the bit lines (`BL` if a 0 is stored, `BL_bar` if a 1 is stored).

(e) `SCLK` signal is generated by the control logic, the sense amplifier is enabled, and the output is produced (read delay).

(f) The output is then captured by flip-flops on the falling edge of the clock, the output tristate is enabled by `OEb`, and the data remains valid on the data bus for the rest of the clock cycle.

2.2.4 Write Operation

Figure 12 displays the timing diagram for the SRAM write operation. It highlights the setup and hold times for each signal as well the memory write time. The write time is illustrated by the "X" signal in the diagram, which is the internal storage node in the memory cell. The list below provides a step-by-step description of what is happening internally in the SRAM[9].

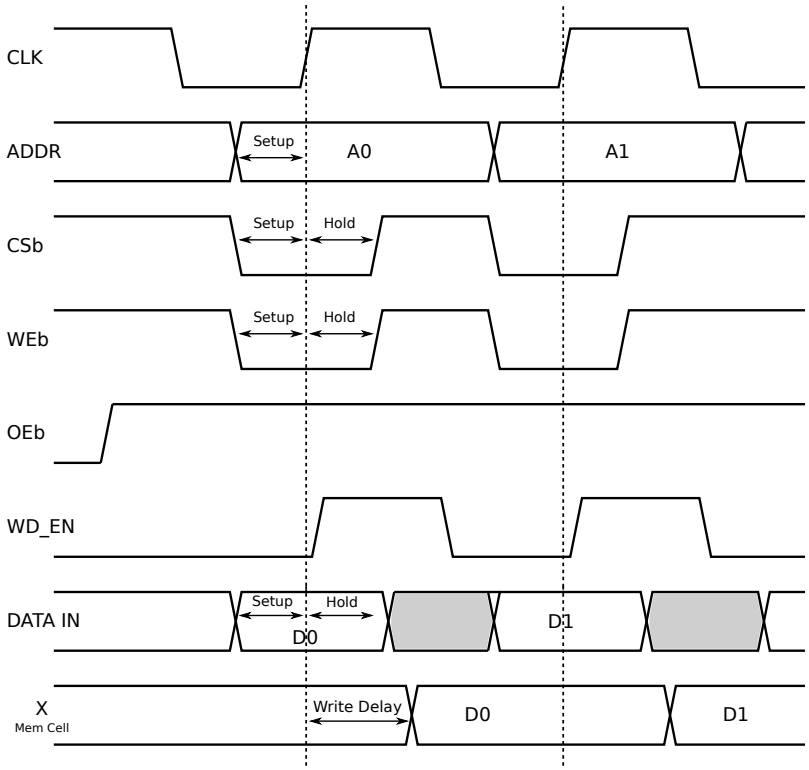


Figure 12: Timing diagram for write operation showing the setup, hold, and write times.

Write Operation:

1. Before the clock transition (low to high) that initiates the write operation:
   (a) `CS_B` set low to enable the chip (setup time).
   (b) `OE_B` set high to disable the sense amp (setup time).
   (c) `WE_B` set low to generate the write driver enable signal (setup time).
   (d) The row and column addresses must be applied to the `ADDR` bus (setup time).
   (e) The data to be written must be applied to `DATA` (setup time).
   (f) Precharge bit lines to `Vdd` and the decoder while `PCLK=CLK=0`.
2. On the rising edge of the clock (CLK):
   (a) `DATA`, `ADDR`, and control signals are latched into flip-flops (hold time).
   (b) The precharging of the bit lines and row decoder stops.
   (c) The decoder and column mux select the row and columns for the word that is to be written to.
   (d) `WD_EN` is generated from the control logic and enables the write driver.

(e) Data is driven through the column mux and into the selected memory cells (write delay).

3 Software Implementation

The compiler framework is divided into "front-end" and "back-end" methodologies as shown in Figure 13. The "front-end" consists of the compiler, which generates Spice models and GDSII layouts based on user inputs, and the characterizer, which calls a Spice simulator and produces timing/power numbers. The "back-end" uses the spice netlists and GDSII layout to generate annotated timing and power models using back-annotated characterization. This section will discuss, in detail, the implementation of the OpenRAM compiler and characterizer.

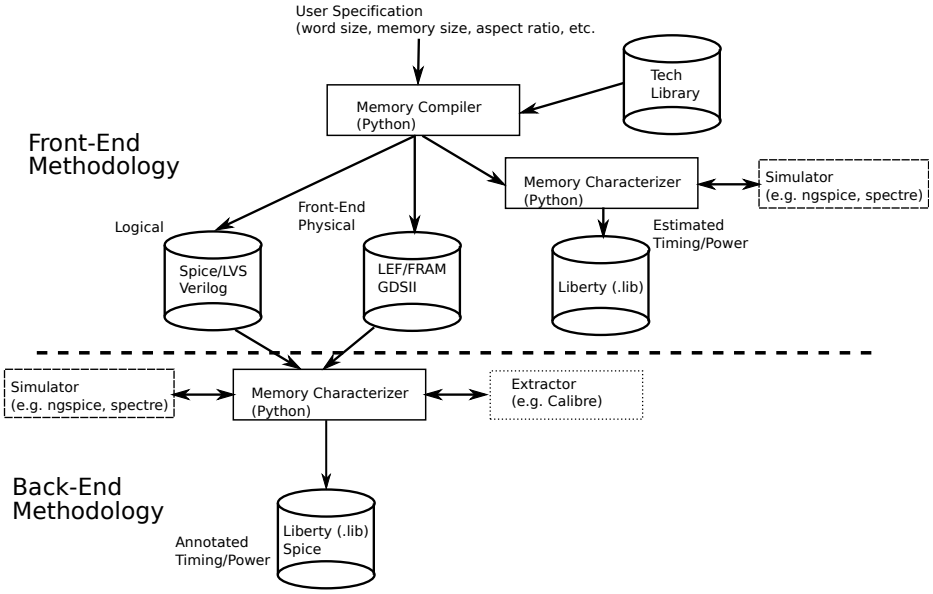


Figure 13: Overall Compilation and Characterization Methodology

3.1 Main Compiler Components

OpenRAM is implemented using object-oriented data structures in the Python programming language[15]. Python was chosen because it is a simple, yet powerful language that is easy to learn and has a syntax that is very human-readable. The open-source release also includes the FreePDK45 technology kit developed by Stine, et al., at North Carolina State University[19]. FreePDK45 is a free, 45 nm, process design kit that is used for designing, modeling, verifying, and simulating integrated circuits. OpenRAM also utilizes GdsMill, a

Python interface developed by Michael Wieckowski that is used to create and manipulate circuit layout in the GDSII format[25].

3.1.1 OpenRAM Design Hierarchy

All modules in OpenRAM are derived from the top-level `design` class. The design class is a data structure that consists of a Spice netlist, a layout, and a name. The Spice netlist capabilities are inherited from the `hierarchy_spice` class while the layout capabilities are inherited from the `hierarchy_layout` class (see Figure 14).

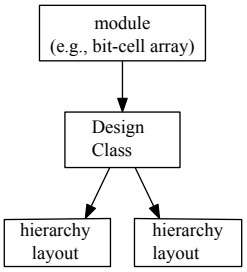


Figure 14: Class Hierarchy

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

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

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

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

  name - name of the instance.

  mod - the associated Spice module.

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

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

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

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

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

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

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

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

  text - the text for the label

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

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

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

- `gds_read():` reads in a GDSII file and creates a `vlsiLayout()` class for it (see Section 3.1.2).

- `gds_write()`: writes the entire GDS of the object to a file by GdsMill `vlsiLayout()` class and calling the `gds2writer()` (see Section 3.1.2).
- `gds_write_file()`: recursively writes all instances and objects in the layout data structure to the gds file.

Library and Dynamically Generated Cells In OpenRAM, there are two flavors of cells and designs: library and dynamically generated. The library cells are custom designed cells that have been verified and imported by the user. These cells tend to be difficult to implement dynamically or need to be custom designed based on area or performance constraints. The memory cell in the SRAM is a prime example of a library cell. This cell should be hand-designed to minimize area, because it is the most replicated cell, and to ensure proper operation. Dynamically generated designs can contain instances of library cells, parameterized cells(see Section 3.2), and GDSII shapes. These designs are created using the GdsMill interface. Section 3.2 provides further explanation on the various library and dynamically generated designs in the FreePDK45 technology.

3.1.2 GdsMill

GDSII is the standard file used in industry to store the layout information of an integrated circuit. The GDSII file is a stream file that consists of records and data types that hold the data for the various instances, shapes, and labels in the layout. In OpenRAM, we utilize a tool called GdsMill to read, write, and manipulate GDSII files. GdsMill was developed by Michael Wieckowski at the University of Michigan and has a completely unrestricted license[24].

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

- `self.rootStructureName` - name of the top-level design.
- `self.structures` -list of structures that are used in the class.

- `self.xyTree` - a list of all structure names that appear in the design.

In the compiler, dynamically generated cells and arrays each need to build a `vlsiLayout` data structure to represent the hierarchical layout. This is performed using various functions from the `vlsiLayout` class in GdsMill. To make things easier, OpenRAM has its own wrapper class called `geometry`. This wrapper class aggregates the most important and frequently used layout class methods and constructs data structures for the instances and objects that will be added to the `vlsiLayout` class. The methods `add_inst()`, `add_rect()`, `add_label()` in `hierarchy_layout`, add the structures to the `geometry` class, which is then written out to a GDSII file using `vlsiLayout` and the `gds2_writer`.

3.1.3 Technology Directory

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

3.2 OpenRAM Modules

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

Parameterized Transistors OpenRAM provided a class, called `ptx`, that generates parameterized transistors of specified type and size. The `ptx` takes the transistor width, number of fingers, and type (PMOS or NMOS) as inputs and dynamically generates the

design utilizing the various GdsMill functions described in Section 3.1.2. The parameterized transistor is the basic building block for all dynamically generated modules in the OpenRAM compiler.

Parameterized Inverter A second parameterized class, `pinv`, generates parameterized inverters. This class takes the NMOS transistor size and desired cell height as inputs, and uses instances of `ptx` transistors and GdsMill shapes to generate the inverter. If the cell height cannot accommodate a single transistor of the specified size, the transistor is split into multiple fingers. This makes the cell grow wider, but the effective drive strength of the inverter is maintained. The parameterized inverter is utilized in various dynamically generated designs and as the word line drivers for the row decoder.

Bitcell and Bitcell Array In OpenRAM, the 6T cell layout and Spice netlist are provided as library cells (see Figure 15 for layout). The memory cell is a library cell for various reasons. First, it allows the user to easily swap in different memory cell designs. Second, this cell should always be custom to minimize area and optimize performance because it is the most replicated cell in the RAM. Lastly, the transistors in the cell must be carefully sized to allow for correct read and write operations as well as provide protection against corruption.

The `bitcell` class instantiates a single memory cell. The `bitcell_array` class dynamically implements the memory array by instantiating a single memory cell and tiling it based on the number of rows and columns (see Figure 16 for layout). Two simple python "for" loops, one nested in the other, add the instances of the memory cells row by row. During the tiling process, the cells are abutted so that all bit lines and word lines are connected in the vertical and horizontal directions respectively. In order to share `Vdd` and `Gnd` rails, cells added to an odd numbered row are flipped, or mirrored over the x-axis. To avoid any extra routing, the power/ground rails, bit lines, and word lines should span the entire width/height of the cell so they automatically connect when the cells are abutted.

Precharge and Precharge Array The precharge circuitry is dynamically generated using the parameterized transistor (`ptx`) and various GdsMill functions used for drawing rectangles and labels in the layout hierarchy. The precharge class dynamically generates a single precharge cell. It adds two instances of `ptx` PMOS transistors, one for the larger PMOS and one for the smaller, equalizing PMOS. These transistor sizes are input parameters to the initialization function. Next, it connects the gate inputs and draws a metal2 rail for

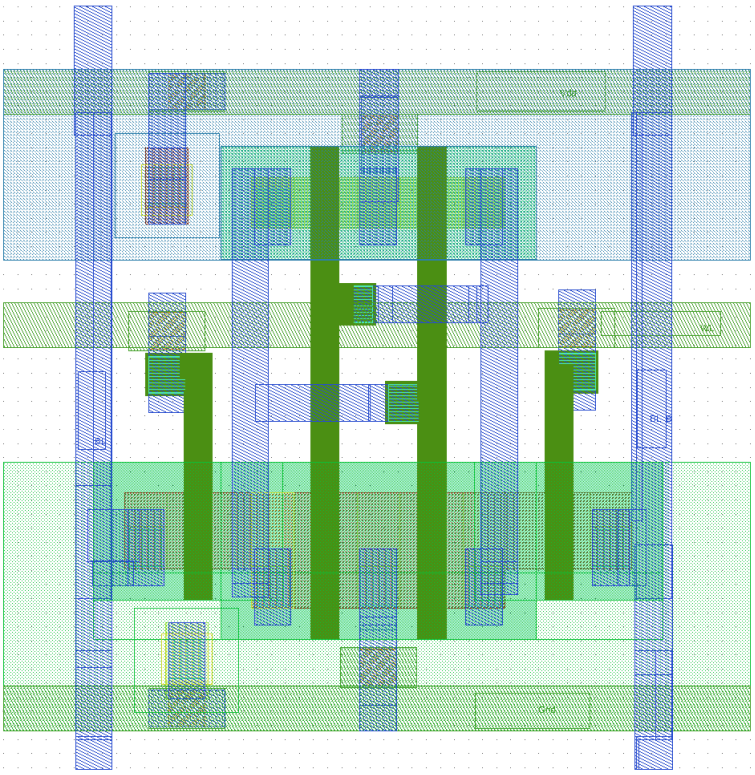


Figure 15: Layout of 6T library cell.

the `PCLK` signal. Similarly, a `Vdd` rail and the vertical bit lines are drawn in metal2 and metal1 respectively. The offsets of the bit lines and the width of the precharge cell must be equal to those of the 6T cell so that the bit lines are correctly connected down to the 6T cell. The `precharge_array` class is then used to generate a precharge array, which is a single row of `n` precharge cells, where `n` equals the number of columns in the bitcell array (see Figure 17 for layout).

NAND Decoder The NAND decoder is dynamically generated and takes the NMOS and PMOS sizes as inputs to the initialization method of the `NAND_decoder` class. The height of each row in the decoder matches the height of the 6T cell so that the power and ground rails can be abutted. First, the `Vdd` and `Gnd` rails are drawn using a "for" loop. Based on the row address bus size, instances of `ptx` NMOS transistors are then added to each row. The source of the first NMOS in each row is always connected to a ground rail. The gate inputs of the NMOS are then connected (in the binary pattern mentioned is Section 2.1.3) to the vertical address input rails. Next, the drain nets are connected to the source nets of the next PMOS in the row. A single PMOS `ptx` instance, used for precharging, is then added to each row and is connected to the `Vdd` rail and the last NMOS in the row. Finally,

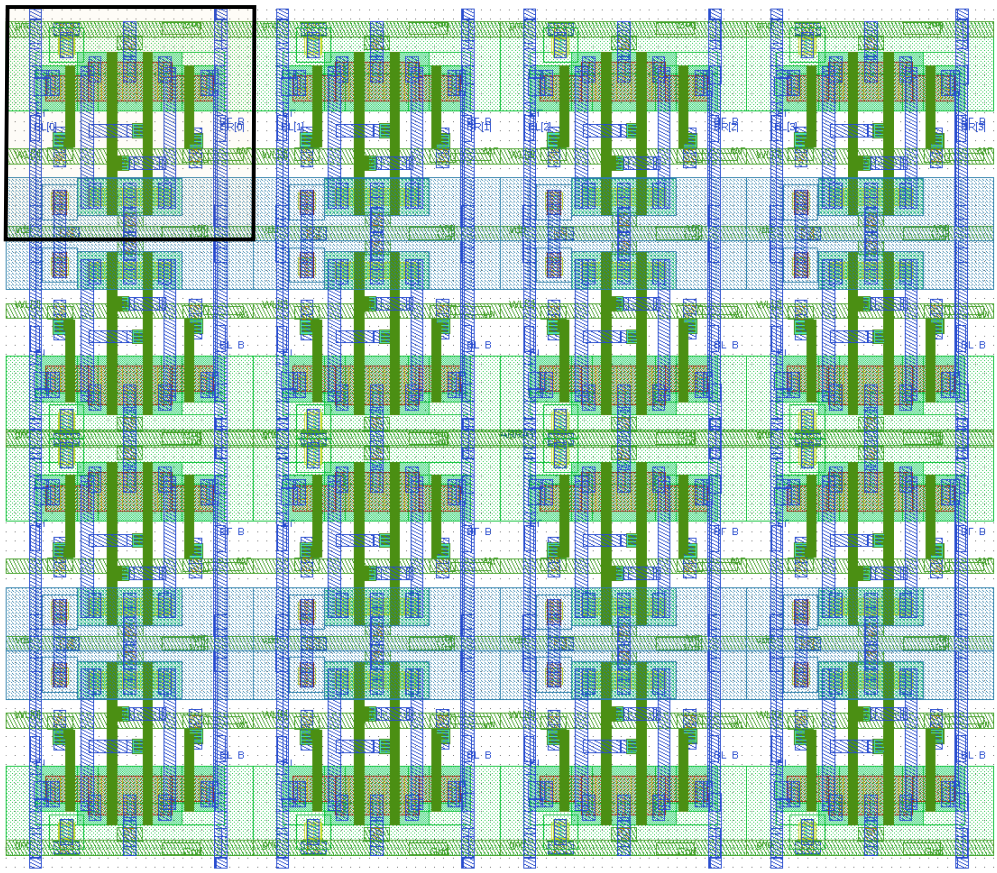


Figure 16: Layout of a 4x4 bitcell array.

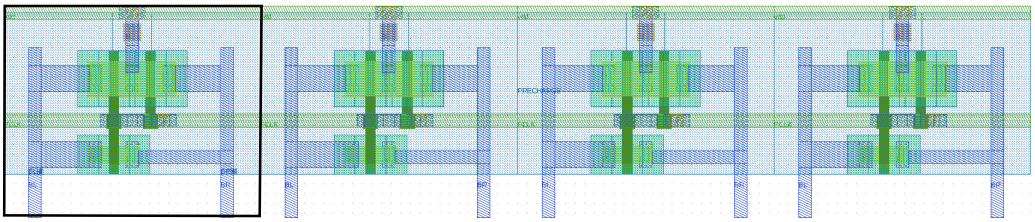


Figure 17: Layout of a four column precharge array

an inverter is placed at the end of each row using the parameterized inverter class (pinv). This inverter also acts as the word line driver and can be sized accordingly (see Figure 18 for layout).

Column Mux In OpenRAM, the column mux is a dynamically generated design. The `column_mux_array` is made up of two dynamically generated cells: `muxa` and `mux_abar`. The only difference between these cells is that the input select signal is either hooked up to `SEL` or `SEL_bar`. These cells are initialized in the the `column_muxa` and `column_muxabar` classes. Instances of `ptx` PMOS transistors are added to the design and the necessary routing is

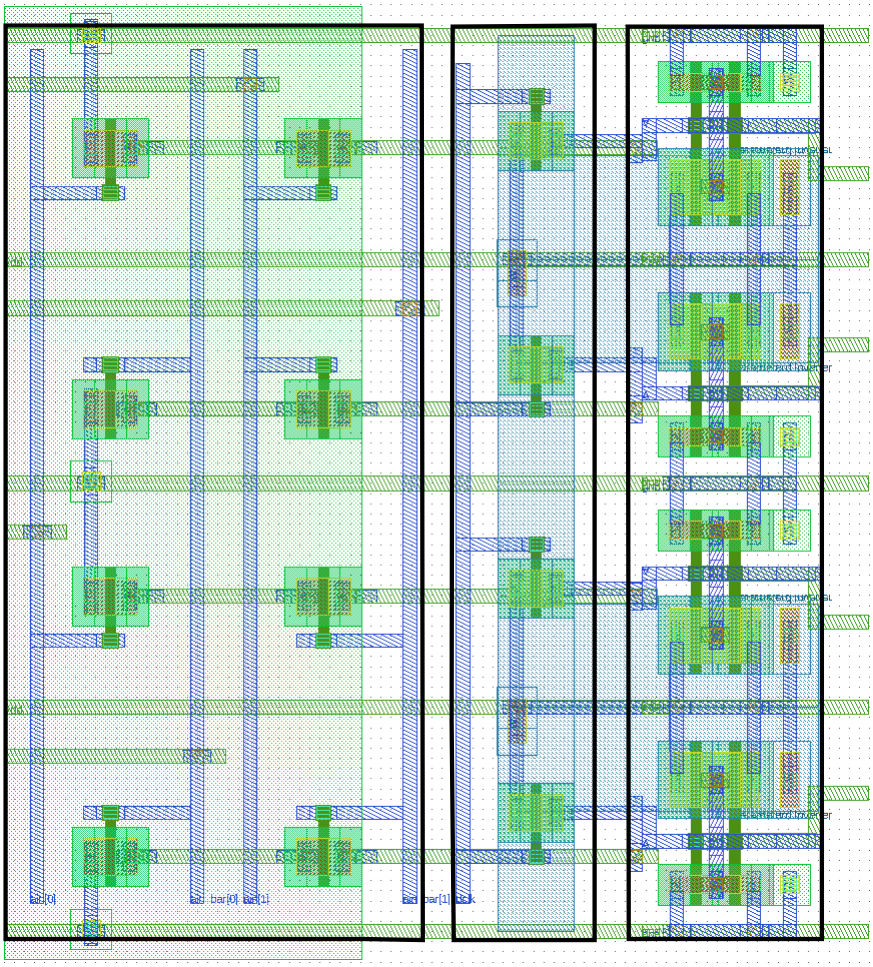


Figure 18: Layout of a four row NAND Decoder

performed using the `add_rect()` function. A horizontal rail is added in metal2 for both the `SEL` and `SEL_bar` signals. Underneath those input rails, horizontal straps are added. These straps are used to connect the `BL` and `BL_bar` outputs from muxa to the `BL` and `BL_bar` outputs of `mux_abar`. Vertical connectors in metal3 are added at the bottom of the cell so that connections can be made down to the sense amp. Vertical connectors are also added in metal1 so that the cells can connect down to other mux cells when the depth of the tree mux grows to more than one level.

The `column_mux_array` class is used to generate the tree mux. Instances of both the `muxa` and `mux_abar` cells are instantiated and are tiled row by row. The offset of the cell in a row is determined by the depth of that row in the tree mux. The pattern used to determine the offset of the mux cells is $muxa.width * (i) * (2 * row\_depth)$ where `i` is the column number. As the depth increases, the mux cells are placed further apart so that they line up with the appropriate words in the memory array. A separate "for" loop is invoked

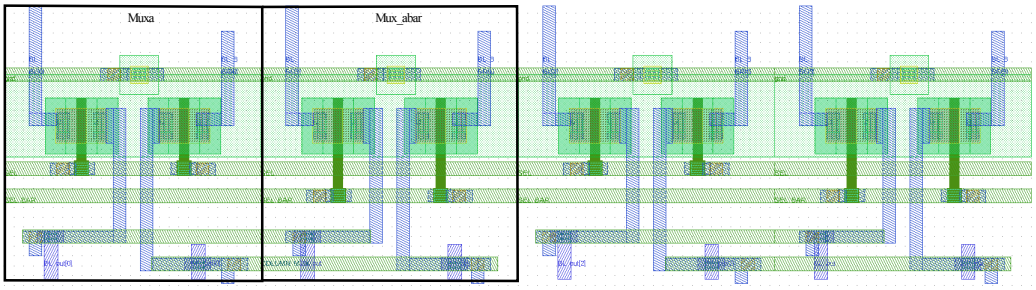


Figure 19: Layout of a four column multiplexer

if the *depth* > 1, which extends the power/ground and select rails across the entire width of the array. Similarly, if the *depth* > 1, Spice net names are created for the intermediate connection nets made at the various levels of the tree. This is necessary to ensure that a correct Spice netlist is generated and that the input/output pins of the column mux match the pins in the modules that it is connected to. Figures 19 and 20 provide the layouts for a single level column multiplexer and another with a depth of two.

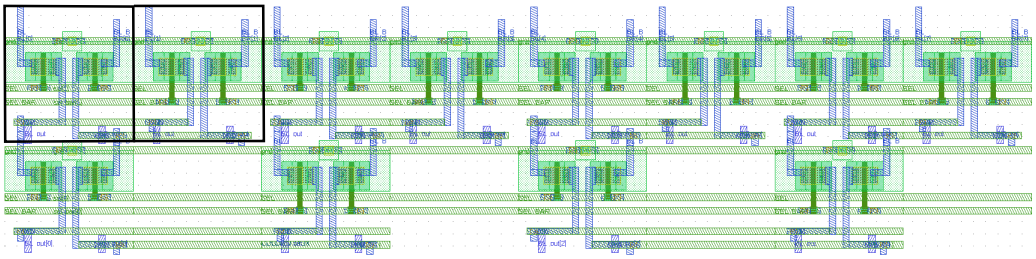


Figure 20: Layout of an eight column multiplexer with a depth of two.

Sense Amp and Sense Amp Array In OpenRAM, the sense amplifier is a library cell because it is a carefully designed analog circuit. The `sense_amp` class instantiates a single instance of the sense amp library cell. The `sense_amp_array` class handles the tiling of the sense amps cells. One sense amp cell is needed per data bit and the cells need to be appropriately spaced so that they can hook up to the column mux bit line pairs. The spacing is determined based on the number of words per row in the memory array. Instances are added and then `Vdd`, `Gnd` and `SCLK` rails that span the entire width of the array are drawn using the `add_rect()` function (see Figure 21 for layout).

We chose to leave the sense amp as a library cell so that custom amplifier designs can be swapped into the memory as needed. The two major considerations while designing the sense amplifier cell are the size of the cell and the bit line/input pitches. Optimally, the cell should be no wider than the 6T cell so that it abuts to the column mux and no extra

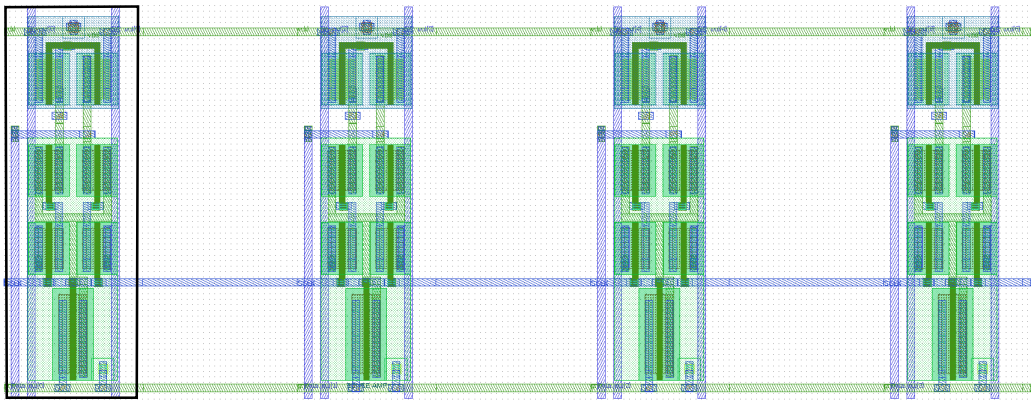


Figure 21: Layout of four sense amplifier cells.

routing or space is needed. Also, the bit line inputs of the sense amp need to line up with the outputs of the write driver. In the current version of OpenRAM, the write driver is situated under the sense amp, which has bit lines spanning the entire height of the cell. In this case, the sense amplifier is disabled during a write operation but the bit lines still connect the write driver to the column mux without any extra routing.

Write Driver and Write Driver Array Currently, in OpenRAM, the write driver is a library cell and the `write_driver_array` class tiles the write driver cells. One driver cell is needed per data bit and `Vdd`, `Gnd`, and `EN` signals must be extended to span the entire width of the cell (see Figure 22 for layout). It is not optimal to have the write driver as a library cell because the driver needs to be sized based on the capacitance of the bit lines. A large memory array needs a stronger driver to drive the data values into the memory cells. We are working on creating a parameterized tristate class, which will dynamically generate write driver cells of different sizes/strengths.

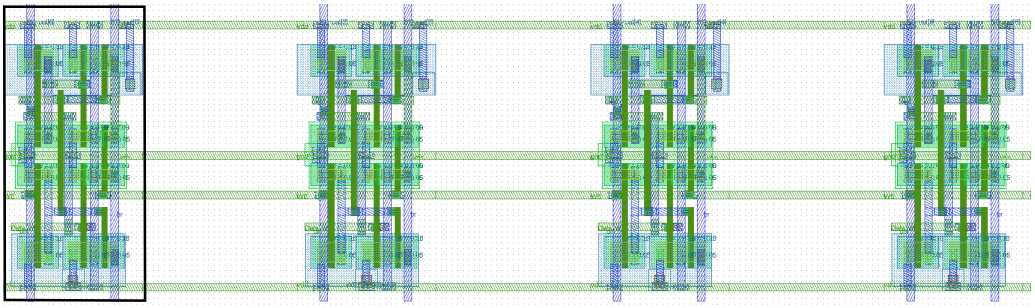


Figure 22: Layout of four write driver cells.

Flip-Flop Array In FreePDK45 we provide a library cell for a simple master-slave flip-flop, see schematic in Figure 10 and layout in Figure 23. In our library cell we provide both `Q` and `Q_bar` as outputs of the flop because inverted signals are used in various modules. The `ms_flop` class instantiates a single master-slave flop, and the `ms_flop_array` class generates an array of flip-flops. Arrays of flops are necessary for the data bus (an array for both the inputs and outputs) as well as the address bus (an array for row and column inputs). The `ms_flop_array` class takes the bus size and the type of array as inputs and dynamically tiles the flip-flops.

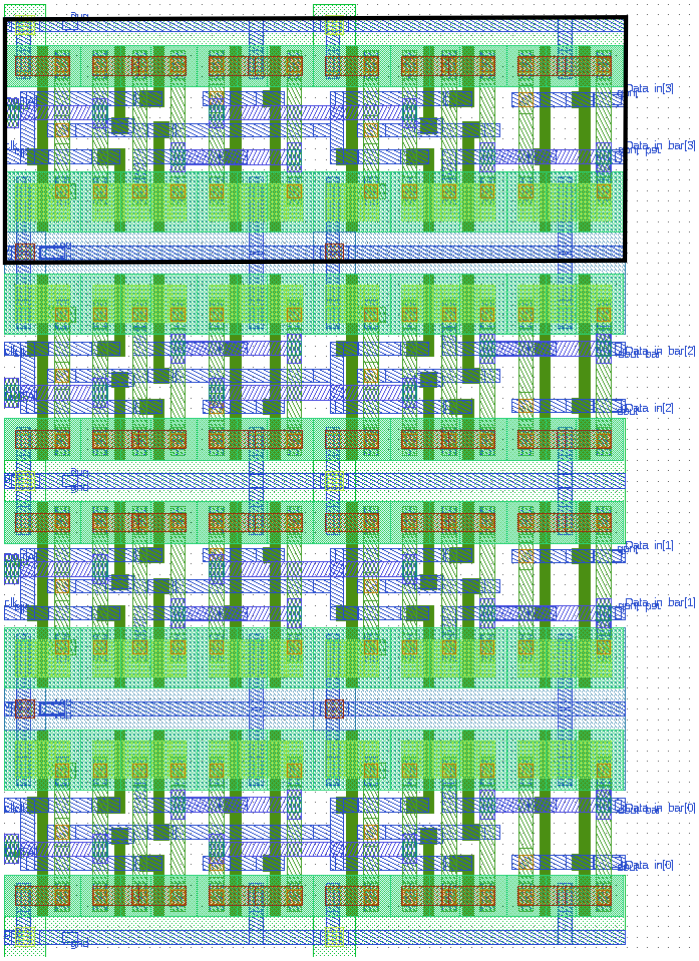


Figure 23: Layout of an array of 4 master-slave flip-flop library cells.

Tristate Array The `tristate_array` class dynamically generates an array of tristate gates used to output data onto the `DATA` bus. A single tristate is instantiated using the parameterized tristate (`ptri`) and an array with length equal the the `DATA` bus size is created by tiling the tristate cells. The parameterized tristate is still under construction, but a library

cell can be used in its place.

Control Logic The control logic module instantiates a `control_logic` class that arranges all of the flip-flops and logic associated with the control signals into a single design. Flip-flops are instantiated for each control signal input and library NAND and NOR cells are used for the logic. A delay chain, of variable length, is also generated using parameterized inverters. This delay chain is used to produce a "delayed" clock for use in precharging the address row decoder. The decoder precharge must be delayed from the normal clock signal to ensure that the address signals from the flip-flops reach the input before the precharge cycle stops. See Figures 24 and 25 for the layouts of the control logic and delay chain.

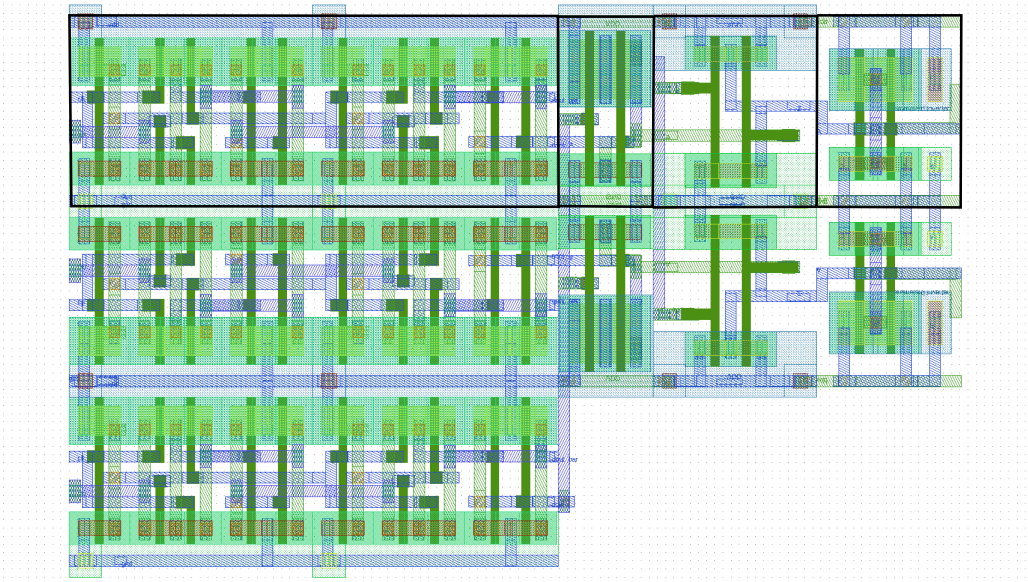


Figure 24: Layout of the control logic.

Top-Level SRAM Module The top level of the hierarchy is the SRAM module. This module handles the global organization of all sub-modules in the memory. Based on the user inputs of the word size and number of words, the parameters needed to generate the memory are calculated and passed to the sub-modules. Equations 2-9 below are used to determine the number of words per row, the aspect ratio of the memory, and the `DATA` and `ADDR` bus sizes.

$$total\_bits = word\_size * num\_words \tag{2}$$

$$words\_per\_row = sqrt(num\_words)/word\_size \tag{3}$$

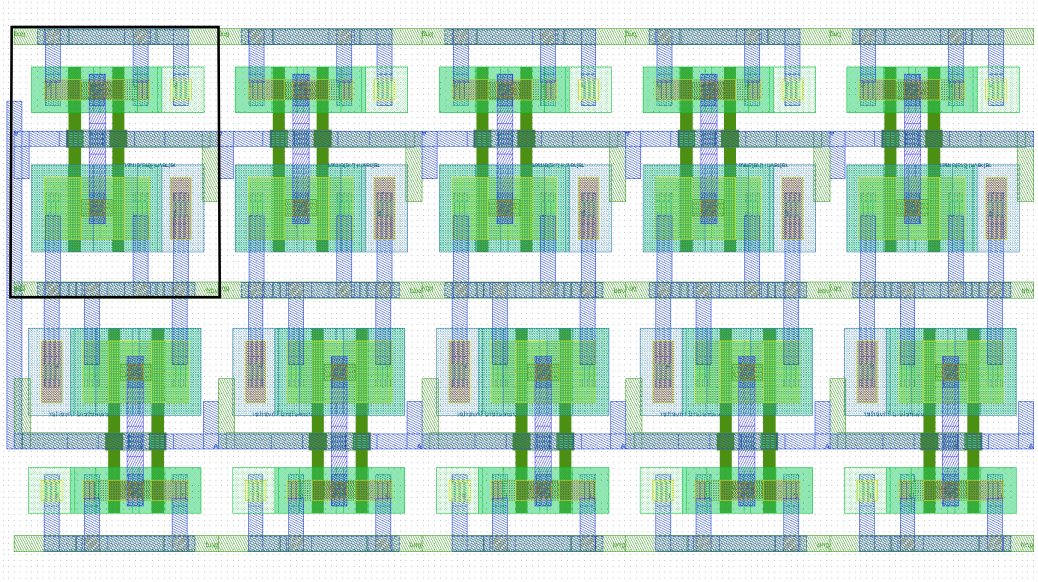


Figure 25: Layout of the delay chain.

$$num\_rows = num\_words / words\_per\_row \tag{4}$$

$$num\_cols = words\_per\_row * word\_size \tag{5}$$

$$col\_addr\_size = log(words\_per\_row, 2) \tag{6}$$

$$row\_addr\_size = log(num\_rows)/log(2) \tag{7}$$

$$total\_addr\_size = row\_addr\_size + col\_addr\_size \tag{8}$$

$$data\_size = word\_size \tag{9}$$

Once these values have been calculated, the arrays can be generated and placed in the top level of the hierarchy based on the sizes of the different modules. It is important to note that when considering the organization of the blocks in the memory, DRC rules for minimum spacing of metals, wells, and other layers must be followed. Currently, for FreePDK45, all of the major blocks that may have a conflict with p-well to n-well spacing are separated by the technology parameter `tech.drc{"pwell_extend_well"}`. This gives a cushion of 0.250μ in-between the blocks, but the I/O pins must be adjusted in the corresponding cells to handle this extra space. Figure 26 provides an example layout for a SRAM with sixteen columns and eight rows. All modules, except for the tristate array, are included in the figure. The control logic, delay chain, and flip-flops have not yet been routed. Appendix A also provides an entire OpenRAM-generated Spice netlist for a 32-byte SRAM.

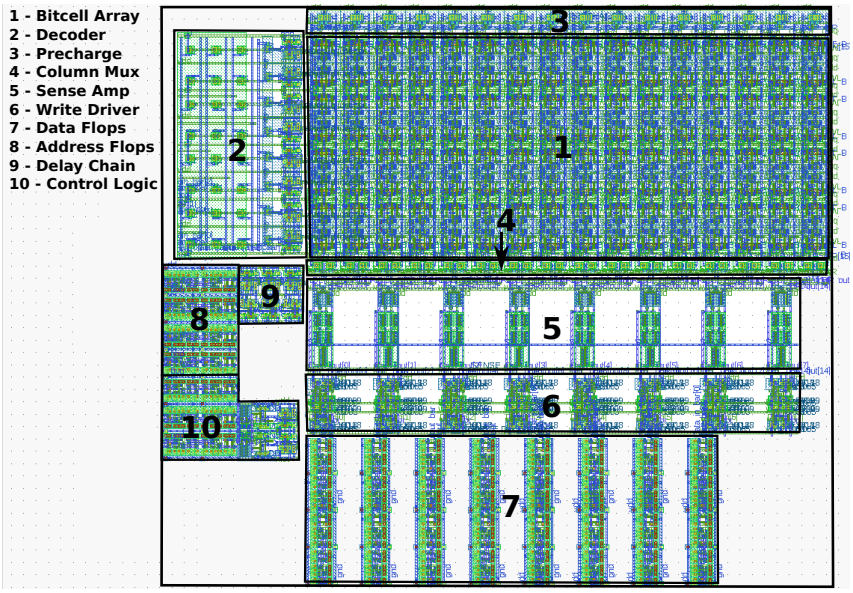


Figure 26: Layout of SRAM with 16 columns, 8 rows, and 2 words per row.

3.3 Physical Verification

OpenRAM interfaces with Calibre nmDRC and nmLVS to perform physical verification of generated designs. Calibre, a Mentor Graphics tool, has two main functions. The DRC, or design rule check, function uses pattern matching algorithms to ensure that all process design rules have been met so that the circuit can be properly fabricated. The LVS, or layout versus schematic, function provides a comparison of the physical layout to the schematic, or Spice netlist, to ensure that the number of devices and connectivity of those devices match[8]. The compiler can also interface with other physical verfification tools. There is a wrapper function that can be edited to call other verification tools.

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

3.4 Memory Characterizer

The memory characterizer is a set of Python scripts that produces the timing and power characteristics of OpenRAM-generated memories through extensive Spice simulations. The

characterizer has three main stages: generating the Spice stimulus and test structures, simulating the circuits, and parsing the simulator output. Currently the characterizer utilizes the HSPICE circuit simulator from Synopsys[20]. An in depth description and results form the characterizer are provided in Sections 4.2 and 4.3.

4 Contributions

This section provides details of the author's significant contributions to the OpenRAM memory compiler. The list below outlines the contributions that have been discussed is previous sections. Sections 4.1 and 4.2 highlight contributions not yet discussed.

1. **Design Hierarchy** - restructuring the design hierarchy into two separate sub-classes: one the Spice hierarchy, and another for the Layout hierarchy. (See Section 3.1.1)
2. **Dynamically Generated Modules** - implemented the following dynamically generated modules (See Section 3.2):
   (a) Precharge and Precharge Array.
   (b) NAND Decoder.
   (c) Tree Column Multiplexer.
   (d) Adapted the Sense Amplifier and Write Driver Arrays to handle more than one word per row in the memory array.
   (e) Control Logic.
   (f) Delay Chain.
   (g) Master-Slave flip-flop array.
3. **Snap-to-Grid Function** - See Section 4.1 below.
4. **Memory Characterizer** - See Sections 4.2 and 4.3 below.

4.1 Snap-to-Grid

The `snap_to_grid()` function ensures that every shape or instance added to the design hierarchy is placed on the manufacturing grid. The manufacturing grid refers to the reference lines used in the mask layer during the fabrication process. Each technology has can have a different size manufacturing grid. In the case of the FreePDK45 technology, the manufacturing grid is set at 0.0025μ. This means that every instance or shape must be placed at a

coordinate that is a multiple of 0.0025μ in order to avoid DRC errors. To ensure that this happens, the following function is used:

```
def snap_to_grid(offset):
    grid = tech.drc["grid"] #the manufacturing grid value
    x = offset[0]
    y = offset[1]
    xgrid = round((x/grid),0) #closest integer value
    ygrid = round((y/grid),0) #closest integer value
    xoff = xgrid*grid
    yoff = ygrid*grid
    out_offset = [xoff,yoff]
    return out_offset
```

The `snap_to_grid()` function first separates the x and y coordinates and divides them by the manufacturing grid value. This division finds the nearest integer multiple of the manufacturing grid. That integer is then multiplied by the grid value, effectively "snapping" the coordinates to the manufacturing grid. The resulting offsets are then concatenated back into an array and returned. This may seem like a very simple function, but it is extremely powerful and can save users significant time while designing modules because any coordinate given will be automatically snapped onto the manufacturing grid.

4.2 Memory Characterizer

The Memory Characterizer is a set of Python scripts that utilize a Spice circuit simulator in order to produce timing and power numbers for OpenRAM generated SRAMs. The characterizer performs four main operations: writing Spice stimulus, finding the read and write delays, finding the setup and hold times for the SRAM inputs, and calculating the average power for the read and write operations. There is also a stimulus parameters file that can be edited by the user to define certain variables such as the supply voltage and clock frequency for the simulation runs.

4.2.1 Spice Stimulus

The characterizer contains various functions that generate and write Spice stimulus and test structures for the different operations that are to be performed. The test structure refers to all devices that are needed for simulation that are not a part of the SRAM proper. For OpenRAM-generated memories with bi-directional `DATA` buses, input signal buffers and access pass gates are necessary for simulation purposes. When voltage sources are defined in Spice, they are ideal sources with infinite drive strength. To ensure accurate results, all input signals must be buffered so that a realistic signal strength is used. Also, pass gates are

used for the bi-directional `DATA` bus. This allows for the input stimulus to be cut off from the `DATA` bus while a read operation occurs. In addition to the test structures, the following functions can be used to generate stimulus:

- `inst_sram()` - adds the instance of the sram to the stimulus file with all ports.
- `gen_data_pwl()` - generates a piece-wise linear function for the `DATA` stimulus.
- `gen_addr_pwl()` - generates a piece-wise linear function for the `ADDR` stimulus.
- `gen_pulse()` - generates a pulse stimulus statement for a given signal.
- `write_supply()` - writes the `Vdd` and `Gnd` supply stimulus.
- `write_include()` - writes the include statements for the transistor models and SRAM netlist.

4.2.2 Read and Write Delays

For the read and write delay simulation, a file called `timing.sp` is written, using the stimulus functions, that contains the test structures and stimulus for all SRAM inputs. Also, Spice `.measure` statements are written so that the Spice circuit simulator can measure the delays. The Spice simulator is then called and the simulation is run. The duration of the simulation is 4 clock cycles; the first and third cycles are dummy write and read cycles, the second cycle performs the write, and the fourth cycle performs the read operation. The data is always written to and read from the top right corner cell in the bitcell array. This cell will always have the longest delay because it is the furthest from the address decoder in the x-direction and the write driver in the y-direction. The write delay measures the time delay from the clock edge until the data value has been written into the memory cell. The read delay measures the time elapsed from the clock edge until there is valid data on the sense amplifier output node. This simulation is run for both writing and reading a 1 and a 0 into the cell. After the simulations have completed, the Spice output file is then parsed and the measurement is extracted and stored.

4.2.3 Power

The average power for both the read and write operations is measured by the Spice simulator during the delay simulations. Spice `.measure` statements measure the average power for each operation using Equation 10. The power is measured over the entire clock period so

that the power from all circuits utilized during an operation are considered. After the power numbers have been parsed from the Spice output files, and the other timing simulations have completed, the Power Delay Product (PDP) is calculated. The PDP is used to classify the energy efficiency and is defined as the average energy consumed per operation (Equation 11):

$$where \quad P_{avg} = \frac{1}{T}\int_0^T p(t)\,\mathrm{d}t = \frac{V_{DD}}{T}\int_0^T i_{DD}(t)\,\mathrm{d}t \tag{10}$$

$$PDP = P_{avg}(max(t_{delay}) + max(t_{setup})) \tag{11}$$

4.2.4 Setup and Hold Time

In a synchronous SRAM, all input signals are registered using flip-flops. This means that the setup and hold times for the SRAM are equal to the setup and hold times of the flip-flop being used. In order to find the setup time of the flip-flop, the `clk-q` delay must first be measured. To measure the `clk-q` delay, a Spice stimulus file is written for the flip-flop netlist and an initial simulation is run. This `clk-q` delay measured in the initial simulation is used as the reference delay. The setup and hold times are calculated for both the low-high and high-low data transitions.

Setup Time The setup time is defined as the time that a signal must be held valid before the clock edge to ensure proper operation within an acceptable delay. In this case, an acceptable delay is defined as no more than a 10% increase of the reference delay measured in the initial simulation run. After the acceptable delay has been calculated, a bi-directional search is performed by running two simultaneous simulations while varying the time that the input switches on both sides of the clock transition. There are two reasons that both sides of the clock transitions need to be checked. First, it is possible to have a flip-flop with a negative setup time. The second reason is that we need to hone in, from both sides, on the input transition time where the delay is forced out of the acceptable delay range. This means that the right side search will move to explore the space where the input transition forces the flip-flop into a meta-stable state and eventually converge to the acceptable delay point. Below is an outline of the algorithm used:

```
def setup_time():
    #set initial left and right pointers
    lpointer = clk_period-.5
    rpointer = clk_period+.5
    #determine acceptable delay
```

```
    delay = clk_q_delay()
    setup_delay = lh_delay+.10000000*lh_delay

    def bidi_search_setup(lpointer,rpointer):
        #run simulation to find left and right delays
        ldelay = clk_q_delay(lpointer)
        rdelay = clk_q_delay(rpointer)
        #find the midpoint between the pointers
        mid_point = float(lpointer + ((rpointer-lpointer)/2))

        if(setup_delay-precision)<ldelay<(setup_delay+precision):
            parse_output(ldelay)
            setup_time = clk-lpointer
        elif (setup_delay-precision)<rdelay<(setup_delay+precision):
            parse_output(rdelay)
            setup_time = clk-rpointer
        elif rdelay==''fail'' or rdelay>(setup_delay+precision):
            bidi_search_setup(lpointer,mid_point)
        elif ldelay<(setup_delay-precision):
            bidi_search_setup(lpointer,mid_point)
        elif rdelay<(setup_delay-precision):
            bidi_search_setup(rpointer,rpointer+.5)
        else:
            assert(ldelay==''fail'' or ldelay>(setup_delay+precision)):
            bidi_search_setup(lpointer-.5,lpointer)

    bidi_search_setup(lpointer,rpointer)
```

The basic idea for the bi-directional search is that simulations are run for the left and right pointers to find the `clk-q` delay. If the delay from the left or right pointer is within the specified precision of the acceptable delay, then the setup point has been found. If the right side simulation fails or the delay is greater than the acceptable delay, the right pointer moves to the midpoint and the function is run again. If the right pointer ever passes the setup point, the left pointer moves to the right pointer position and the right pointer is pushed further right. There are also case statements to ensure that the left or right pointers do not end up on the same side of the setup point. Simulations are recursively run until the setup point has been found. The Spice output is then parsed and the setup time is calculated as the time difference between the clock edge and data transition. The precision of this function can be tuned by the user; but an increase in precision results in more simulations and a longer run-time.

Hold Time The hold time is defined as the time that an input signal must be held valid after the clock edge to ensure proper operation with an acceptable delay. The hold time of the flip-flop is calculated in the same way as the setup time; writing Spice stimulus and performing a bi-directional search. The hold time algorithm actually uses the initial delay and setup time found by the setup time function as input parameters. Using these values allows for less calls to the Spice simulator. The left pointer can be initialized to the setup point because the hold time can never be less than the setup time. The bi-directional hold

time search moves the right pointer to the midpoint if the right simulation fails or if the delay is greater than the acceptable delay. If the right delay does not fail or the delay is less than the acceptable delay point, the left pointer moves to the right pointer and the right pointer moves further to the right. Below are the details of the algorithm:

```
def hold_time(delay,setup_time):
    #determine left and right pointers
    lpointer = clk_period-setup_time
    rpointer = clk_period+.5
    #determine acceptable delay
    hold_delay = lh_delay+.10000000*lh_delay

    def bidi_search_hold(lpointer,rpointer):
        #run simulation to find left and right delays
        ldelay = clk_q_delay(lpointer)
        rdelay = clk_q_delay(rpointer)
        #find the midpoint between the pointers
        mid_point = float(lpointer + ((rpointer-lpointer)/2))

        if (hold_delay-precision)<ldelay<(hold_delay+precision):
            hold_time = data-clk
        elif (hold_delay-precision)<rdelay<(hold_delay+precision):
            hold_time = data-clk
        elif rdelay==''fail'' or rdelay>(hold_delay+precision):
            bidi_search_hold(lpointer,mid_point)
        elif rdelay!=''fail'' and rdelay<(hold_delay+sim_params.precision):
            bidi_search_hold(rpointer,rpointer+(rpointer-mid_point))

     bidi_search_hold(lpointer,rpointer)
```

Minimum Clock Period After the low-high and high-low setup, hold, and delays have been calculated, the minimum clock period that the memory can operate at can be calculated by Equation 12. The maximum clock frequency, in megahertz, can be calculated using Equation 13.

$$Min\_Clk\_Period = max(delay) + max(setup) \tag{12}$$

$$Clk\_Freq = 1/(min\_clk\_period * 10^{-3}) \tag{13}$$

4.3 Characterizer Results

As previously stated, the characterizer reports the setup and hold time of the flip-flop, the read and write delays, the average power for each operation, and the maximum operating frequency. The characterizer was run on four small memory designs generated by the OpenRAM compiler. For fair comparison of results, all memories utilize the same size write driver. Table 4 shows the results of the characterization of the master-slave flip-flop. Delay,

setup, and hold times are calculated for both the high to low and low to high data transitions. These results are somewhat optimistic because an un-loaded flip-flop is simulated; the load that the flip-flop drives will inevitably have an effect of the delay. Table 5 displays the delay, average power, and power delay product of the write operation for the different size SRAMs. Similarly, Table 6 shows the delay, average power, and power delay product of the read operation for the different size SRAMs. Finally, Table 7 reports the minimum clock period and maximum operating frequency of the SRAMs. The results obtained for the SRAM simulation are from the pre-back annotation characterizer.

Flip-Flop Characterization					
Clk-Q (ps)		Setup (ps)		Hold (ps)	
low-high	high-low	low-high	high-low	low-high	high-low
31.0099	28.1670	13.300	11.200	-12.800	-10.000

Table 4: Clk-Q delay, setup, and hold times for the master-slave flip-flop.

Write Operation						
SRAM Size	ColsxRows	Delay (ps)		Avg Power (MW)		PDP (fJ)
		Write 0	Write 1	Write 0	Write 1	
32B	16x16	167.8500	178.7582	0.13313	0.13738	24.5592
256B	32x64	258.0382	268.6272	0.33036	0.33804	90.8109
512B	64x64	375.4901	390.2605	0.67410	0.70459	274.9823
1kB	64x128	398.3297	403.8565	0.84978	0.89101	359.8524

Table 5: Delay, power, and PDP for the writing both a 0 and 1.

Read Operation						
SRAM Size	ColsxRows	Delay (ps)		Avg Power (MW)		PDP (fJ)
		Read 0	Read 1	Read 0	Read 1	
32B	16x16	194.7432	193.2575	0.32047	0.59707	116.2835
256B	32x64	318.9401	313.5573	0.80970	1.2154	387.5283
512B	64x64	425.9756	414.0976	1.8481	2.2828	972.1053
1kB	64x128	499.0511	487.1450	2.1897	2.4994	1247.1617

Table 6: Delay, power, and PDP for the reading both a 0 and 1.

Clock Period and Frequency			
SRAM Size	ColsxRows	Min. Period (ns)	Max Freq. (Ghz)
32B	16x16	0.20804	4.8067
256B	32x64	0.33224	3.0099
512B	64x64	0.43930	2.2765
1kB	64x128	0.51235	1.9518

Table 7: Minimum clock period and maximum clock frequency for each size SRAM.

Many observations can be made from examining these results. Figure 27 shows that as the size of the memory grows, the operation delay increases, as expected. It also shows that the write operation is faster than the read operation for all memory sizes. The read operation is slower due to the delay of the differential sense amplifier. It can also be seen that writing a 0 is faster than writing a 1; the write driver can discharge the bit lines much faster than it can drive a 1 into the memory cell. Conversely, it is slower to read a 0 than it is a 1. This can be explained by the fact that the bit lines only experience a small voltage swing and a sufficient differential signal needs to be established before the sense amplifier can evaluate. The aspect ratio of the SRAM also has an impact on the delay. More significant increases of delay occur when the number of columns increases, i.e. when the array becomes wider.

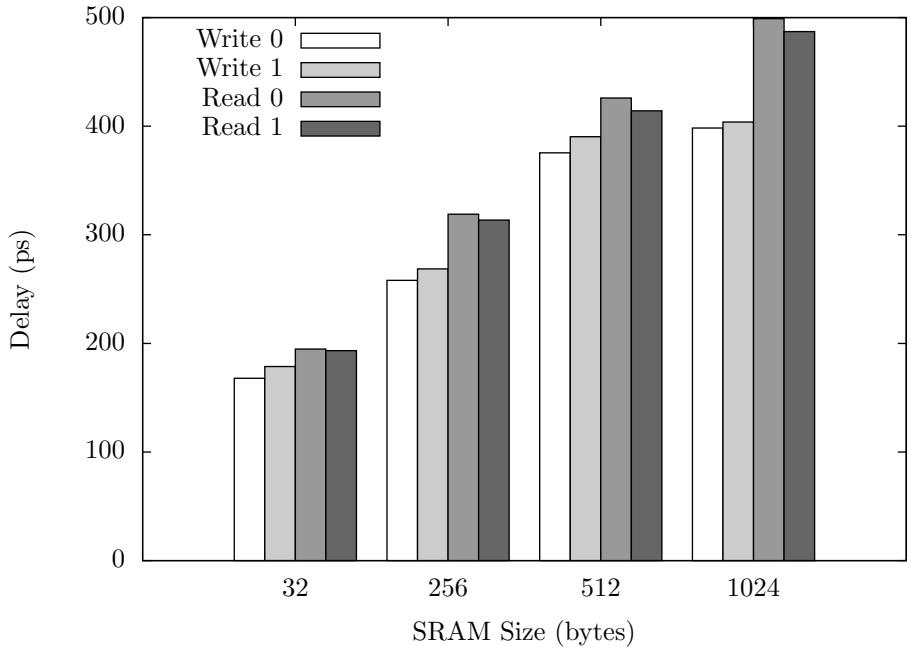


Figure 27: The delays for the read and write operations.

Most of the dynamic power in an SRAM is a by-product of the charging and discharging of the bit and word lines. During both memory operations, all bit lines are precharged to `Vdd` and one word line is always asserted. When this word line is selected, all of the cells in that row are activated and bit lines are discharged even if the column is not selected. Figure 28 highlights the average power dissipation of the various SRAMs for both read and write operations. The read operation consumes more power than a write because of the power dissipation of the sense amplifier and because extra tristate buffers are used to drive

the output onto the bi-directional data bus. It is also apparent that reading a 1 consumes more power than reading a 0. Again the sense amplifier is the cause of the power increase; the sense amplifier mirrors the current seen by the bit line. The energy-efficiency of the write operation versus the read operation is further highlighted by Figure 29. Figure 29 illustrates the power-delay-product for both operations and shows that a write can be approximately 4x more efficient than a read.

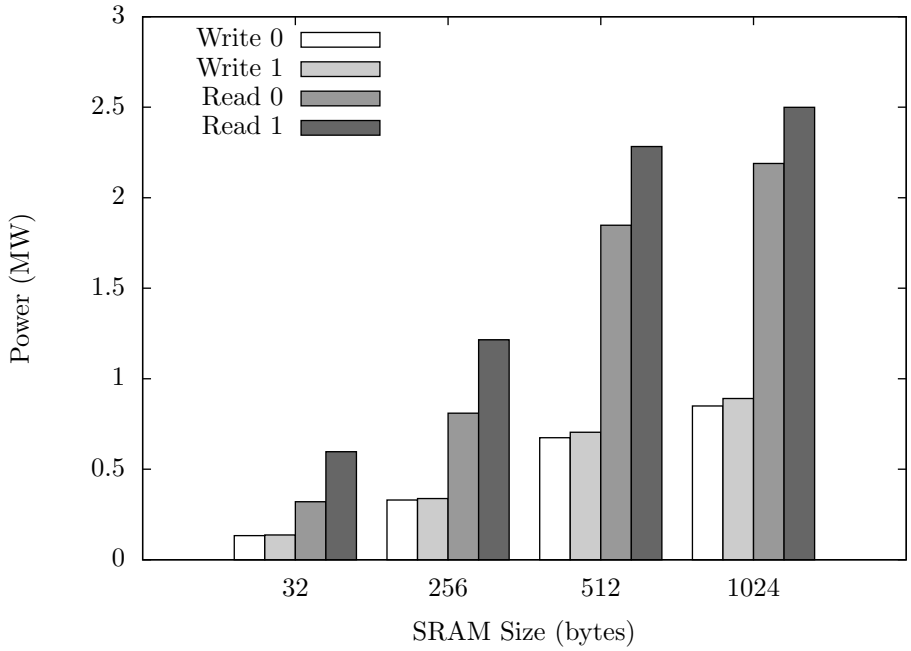


Figure 28: The average power for the read and write operations.

Lastly, Figure 30 displays the maximum operating frequencies of the different SRAM sizes as calculated by Equations 12 and 13. As expected, the operating frequency decreases as the size of the SRAM increases.

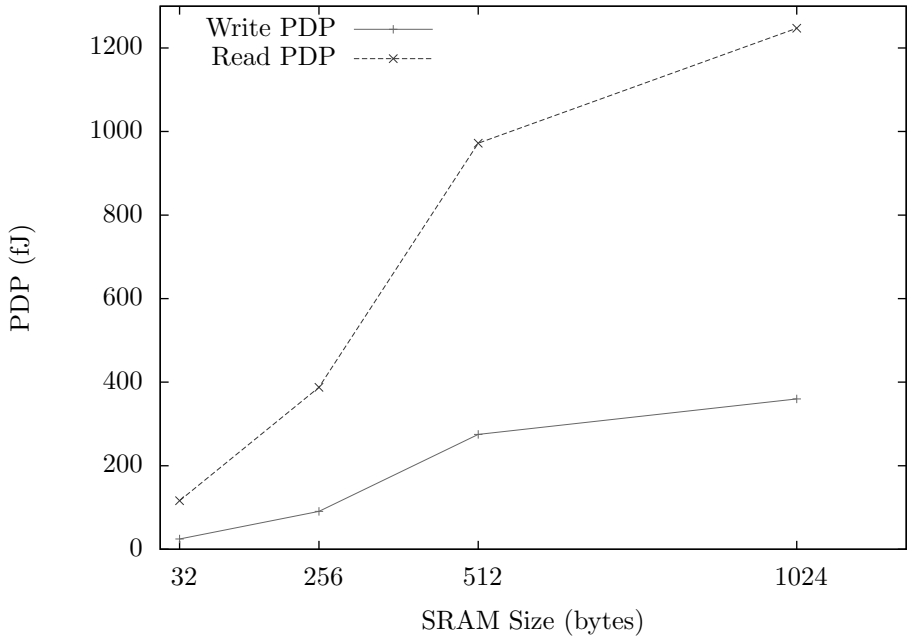


Figure 29: The power-delay products for the read and write operations.

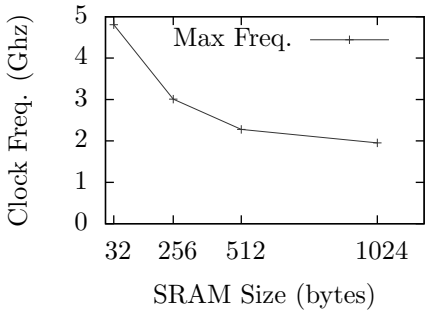


Figure 30: The maximum operating frequency for the different size SRAM's.

4.4 Area

Table 8 displays the area in mm^2 for each block in the SRAM, including the extra whitespace, as well as the total area. Figure 31 provides a graphical representation of the numbers in Table 8. It can be seen that the bitcell array dominates the total area of the SRAM for all size; proving that this cell should be optimized to reduce its size. For the larger SRAM's, the bitcell array accounts for approximately 75-80% of the total area. The peripheral circuitry occupies a significantly smaller amount of area in comparison to the memory array, but circuits such as the decoder and column multiplexer will grow much larger as the size of the memory scales up. The whitespace, or unused area, is approximately 5% of the total area for all memory sizes, but this area can be used for higher-level routing.

Area (mm^2)									
Size	Bitcell	Decoder	Mux	Precharge	SA	WD	Control	Space	Total
32B	0.3731	0.1113	0.0233	0.0233	0.0681	0.0502	0.0416	0.1597	0.8866
256B	2.9850	0.2596	0.0933	0.0933	0.2726	0.2010	0.1304	0.5540	4.5892
512B	5.9699	0.5868	0.0933	0.0933	0.2726	0.2010	0.1304	0.4111	7.7584
1kB	11.9398	0.5868	0.1866	0.1866	0.5453	0.4020	0.2357	0.8132	14.8960

Table 8: Area (mm^2) of each module and total area.

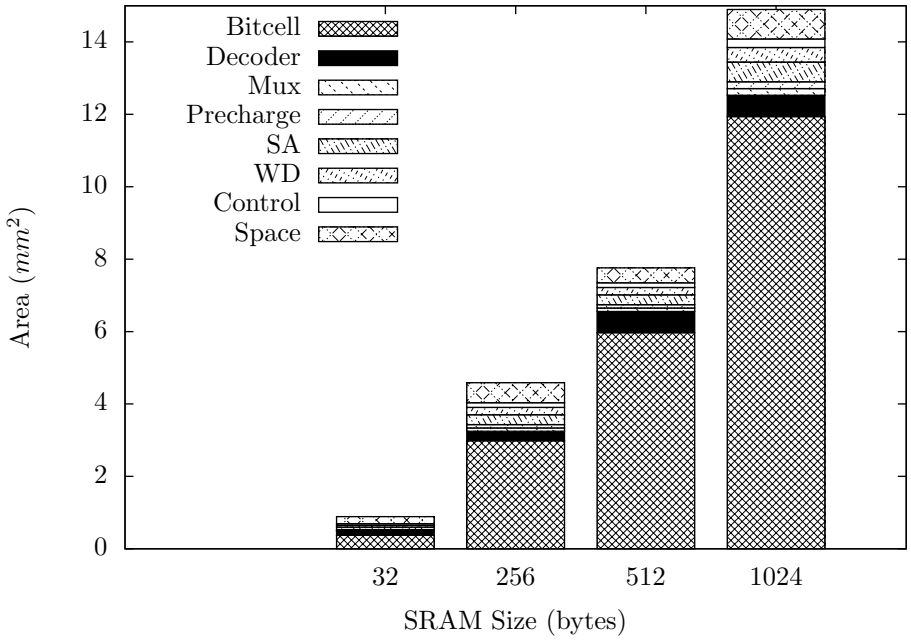


Figure 31: Total area of the different size SRAM's and the area of each module in the SRAM.

5 Conclusion

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

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

In addition to the compiler, a memory characterization methodology has been introduced. The memory characterizer uses the OpenRAM generated Spice netlist and a Spice simulator to produce the timing and power characteristics of the synchronous memory. The characterizer writes Spice stimulus files and invokes a Spice simulator to measure the read and write delays and average power. The library cell flip-flop is also characterized and a bi-directional search and exhaustive Spice simulations are used to determine the setup and hold times for the input signals as well as the maximum operating frequency. Lastly, the characterizer was run on several OpenRAM generated memories and the results were reported.

It is our hope that the OpenRAM compiler and characterization methodology will become a widely adopted option for embedded memory design generation. OpenRAM provides a simple way of generating memories that can be utilized in any SOC, ASIC, or microprocessor. It also provides a platform to implement and test new memory cells and sub-circuits without considerable overhead. Currently, the compiler is being ported over to the IBM 7SF (180nm) and 8RF (130nm) technologies for fabrication.

5.1 Future Work

The OpenRAM memory compiler and characterizer are still in the early phases of development. Many improvements and additions are currently under construction or in the implementation queue:

1. Dynamic generation and sizing of write drivers based on the memory array size.
2. Adding helper functions to perform routing using GdsMill.
3. Expanding the types of OpenRAM library and dynamically generated cells.
4. Adding capability to generate multi-port RAM's and register files.
5. Option to generate and characterize asynchronous SRAM's.
6. Porting the compiler over to IBM 7SF and 8RF technologies for tape-out and fabrication.
7. Adding back-annotation and extracted parasitic characterization.
8. Writing .liberty files for the timing and power outputs.

References

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

[2] ARM. Embedded memory IP. `http://www.arm.com/products/physical-ip/embedded-memory-ip/index.php`, 2013.

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

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

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

[6] Global Foundries. Memory IP. `http://www.globalfoundries.com/design/memory_ip.aspx`, 2013.

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

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

[9] IBM. Understanding static ram operation. Technical report, Mar 1997.

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

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

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

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

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

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

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

[17] S. Rusu, J. Stinson, S. Tam, J. Leung, H. Muljono, and B. Cherkauer. A 1.5-ghz 130-nm itanium reg; 2 processor with 6-mb on-die l3 cache. *Solid-State Circuits, IEEE Journal of*, 38(11):1887–1895, 2003.

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

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

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

[21] Synopsys. Designware memory compilers. http://www.synopsys.com/dw/ipdir.php?ds=dwc_sram_memory_compilers, 2013.

[22] Faraday Technologies. Memory compiler architecture. http://www.faraday-tech.com/html/Product/IPProduct/LibraryMemoryCompiler/index.htm, 2013.

[23] Dolphin Technology. Memory products. http://www.dolphin-ic.com/memory-products.html, 2011.

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

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

Appendix A: Spice Netlist

```
#OpenRAM Generated 16x16 (32B) SRAM

.SUBCKT cell_6t BL BR WL vdd gnd
mPL x x_bar vdd vdd pmos_vtg w=90.000000n l=50.000000n
mNL x x_bar gnd gnd nmos_vtg w=205.000000n l=50.000000n
mPR x_bar x vdd vdd pmos_vtg w=90.000000n l=50.000000n
mNR x_bar x gnd gnd nmos_vtg w=205.000000n l=50.000000n
mAL BL WL x gnd nmos_vtg w=135.000000n l=50.000000n
mAR BR WL x_bar gnd nmos_vtg w=135.000000n l=50.000000n
.ends cell_6t

.SUBCKT bitcell_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4] BR
    [4] BL[5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10] BR
    [10] BL[11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR[15]
     WL[0] WL[1] WL[2] WL[3] WL[4] WL[5] WL[6] WL[7] WL[8] WL[9] WL[10] WL[11]
     WL[12] WL[13] WL[14] WL[15] vdd gnd
Xbit_r0_c0 BL[0] BR[0] WL[0] vdd gnd cell_6t
Xbit_r0_c1 BL[1] BR[1] WL[0] vdd gnd cell_6t
Xbit_r0_c2 BL[2] BR[2] WL[0] vdd gnd cell_6t
Xbit_r0_c3 BL[3] BR[3] WL[0] vdd gnd cell_6t
Xbit_r0_c4 BL[4] BR[4] WL[0] vdd gnd cell_6t
Xbit_r0_c5 BL[5] BR[5] WL[0] vdd gnd cell_6t
Xbit_r0_c6 BL[6] BR[6] WL[0] vdd gnd cell_6t
Xbit_r0_c7 BL[7] BR[7] WL[0] vdd gnd cell_6t
Xbit_r0_c8 BL[8] BR[8] WL[0] vdd gnd cell_6t
Xbit_r0_c9 BL[9] BR[9] WL[0] vdd gnd cell_6t
Xbit_r0_c10 BL[10] BR[10] WL[0] vdd gnd cell_6t
Xbit_r0_c11 BL[11] BR[11] WL[0] vdd gnd cell_6t
Xbit_r0_c12 BL[12] BR[12] WL[0] vdd gnd cell_6t
Xbit_r0_c13 BL[13] BR[13] WL[0] vdd gnd cell_6t
Xbit_r0_c14 BL[14] BR[14] WL[0] vdd gnd cell_6t
Xbit_r0_c15 BL[15] BR[15] WL[0] vdd gnd cell_6t
Xbit_r1_c0 BL[0] BR[0] WL[1] vdd gnd cell_6t
Xbit_r1_c1 BL[1] BR[1] WL[1] vdd gnd cell_6t
Xbit_r1_c2 BL[2] BR[2] WL[1] vdd gnd cell_6t
Xbit_r1_c3 BL[3] BR[3] WL[1] vdd gnd cell_6t
Xbit_r1_c4 BL[4] BR[4] WL[1] vdd gnd cell_6t
Xbit_r1_c5 BL[5] BR[5] WL[1] vdd gnd cell_6t
Xbit_r1_c6 BL[6] BR[6] WL[1] vdd gnd cell_6t
Xbit_r1_c7 BL[7] BR[7] WL[1] vdd gnd cell_6t
Xbit_r1_c8 BL[8] BR[8] WL[1] vdd gnd cell_6t
Xbit_r1_c9 BL[9] BR[9] WL[1] vdd gnd cell_6t
Xbit_r1_c10 BL[10] BR[10] WL[1] vdd gnd cell_6t
Xbit_r1_c11 BL[11] BR[11] WL[1] vdd gnd cell_6t
Xbit_r1_c12 BL[12] BR[12] WL[1] vdd gnd cell_6t
Xbit_r1_c13 BL[13] BR[13] WL[1] vdd gnd cell_6t
Xbit_r1_c14 BL[14] BR[14] WL[1] vdd gnd cell_6t
Xbit_r1_c15 BL[15] BR[15] WL[1] vdd gnd cell_6t
Xbit_r2_c0 BL[0] BR[0] WL[2] vdd gnd cell_6t
Xbit_r2_c1 BL[1] BR[1] WL[2] vdd gnd cell_6t
Xbit_r2_c2 BL[2] BR[2] WL[2] vdd gnd cell_6t
Xbit_r2_c3 BL[3] BR[3] WL[2] vdd gnd cell_6t
Xbit_r2_c4 BL[4] BR[4] WL[2] vdd gnd cell_6t
Xbit_r2_c5 BL[5] BR[5] WL[2] vdd gnd cell_6t
Xbit_r2_c6 BL[6] BR[6] WL[2] vdd gnd cell_6t
Xbit_r2_c7 BL[7] BR[7] WL[2] vdd gnd cell_6t
Xbit_r2_c8 BL[8] BR[8] WL[2] vdd gnd cell_6t
Xbit_r2_c9 BL[9] BR[9] WL[2] vdd gnd cell_6t
Xbit_r2_c10 BL[10] BR[10] WL[2] vdd gnd cell_6t
Xbit_r2_c11 BL[11] BR[11] WL[2] vdd gnd cell_6t
Xbit_r2_c12 BL[12] BR[12] WL[2] vdd gnd cell_6t
Xbit_r2_c13 BL[13] BR[13] WL[2] vdd gnd cell_6t
Xbit_r2_c14 BL[14] BR[14] WL[2] vdd gnd cell_6t
Xbit_r2_c15 BL[15] BR[15] WL[2] vdd gnd cell_6t
Xbit_r3_c0 BL[0] BR[0] WL[3] vdd gnd cell_6t
Xbit_r3_c1 BL[1] BR[1] WL[3] vdd gnd cell_6t
Xbit_r3_c2 BL[2] BR[2] WL[3] vdd gnd cell_6t
```

```
Xbit_r3_c3 BL[3] BR[3] WL[3] vdd gnd cell_6t
Xbit_r3_c4 BL[4] BR[4] WL[3] vdd gnd cell_6t
Xbit_r3_c5 BL[5] BR[5] WL[3] vdd gnd cell_6t
Xbit_r3_c6 BL[6] BR[6] WL[3] vdd gnd cell_6t
Xbit_r3_c7 BL[7] BR[7] WL[3] vdd gnd cell_6t
Xbit_r3_c8 BL[8] BR[8] WL[3] vdd gnd cell_6t
Xbit_r3_c9 BL[9] BR[9] WL[3] vdd gnd cell_6t
Xbit_r3_c10 BL[10] BR[10] WL[3] vdd gnd cell_6t
Xbit_r3_c11 BL[11] BR[11] WL[3] vdd gnd cell_6t
Xbit_r3_c12 BL[12] BR[12] WL[3] vdd gnd cell_6t
Xbit_r3_c13 BL[13] BR[13] WL[3] vdd gnd cell_6t
Xbit_r3_c14 BL[14] BR[14] WL[3] vdd gnd cell_6t
Xbit_r3_c15 BL[15] BR[15] WL[3] vdd gnd cell_6t
Xbit_r4_c0 BL[0] BR[0] WL[4] vdd gnd cell_6t
Xbit_r4_c1 BL[1] BR[1] WL[4] vdd gnd cell_6t
Xbit_r4_c2 BL[2] BR[2] WL[4] vdd gnd cell_6t
Xbit_r4_c3 BL[3] BR[3] WL[4] vdd gnd cell_6t
Xbit_r4_c4 BL[4] BR[4] WL[4] vdd gnd cell_6t
Xbit_r4_c5 BL[5] BR[5] WL[4] vdd gnd cell_6t
Xbit_r4_c6 BL[6] BR[6] WL[4] vdd gnd cell_6t
Xbit_r4_c7 BL[7] BR[7] WL[4] vdd gnd cell_6t
Xbit_r4_c8 BL[8] BR[8] WL[4] vdd gnd cell_6t
Xbit_r4_c9 BL[9] BR[9] WL[4] vdd gnd cell_6t
Xbit_r4_c10 BL[10] BR[10] WL[4] vdd gnd cell_6t
Xbit_r4_c11 BL[11] BR[11] WL[4] vdd gnd cell_6t
Xbit_r4_c12 BL[12] BR[12] WL[4] vdd gnd cell_6t
Xbit_r4_c13 BL[13] BR[13] WL[4] vdd gnd cell_6t
Xbit_r4_c14 BL[14] BR[14] WL[4] vdd gnd cell_6t
Xbit_r4_c15 BL[15] BR[15] WL[4] vdd gnd cell_6t
Xbit_r5_c0 BL[0] BR[0] WL[5] vdd gnd cell_6t
Xbit_r5_c1 BL[1] BR[1] WL[5] vdd gnd cell_6t
Xbit_r5_c2 BL[2] BR[2] WL[5] vdd gnd cell_6t
Xbit_r5_c3 BL[3] BR[3] WL[5] vdd gnd cell_6t
Xbit_r5_c4 BL[4] BR[4] WL[5] vdd gnd cell_6t
Xbit_r5_c5 BL[5] BR[5] WL[5] vdd gnd cell_6t
Xbit_r5_c6 BL[6] BR[6] WL[5] vdd gnd cell_6t
Xbit_r5_c7 BL[7] BR[7] WL[5] vdd gnd cell_6t
Xbit_r5_c8 BL[8] BR[8] WL[5] vdd gnd cell_6t
Xbit_r5_c9 BL[9] BR[9] WL[5] vdd gnd cell_6t
Xbit_r5_c10 BL[10] BR[10] WL[5] vdd gnd cell_6t
Xbit_r5_c11 BL[11] BR[11] WL[5] vdd gnd cell_6t
Xbit_r5_c12 BL[12] BR[12] WL[5] vdd gnd cell_6t
Xbit_r5_c13 BL[13] BR[13] WL[5] vdd gnd cell_6t
Xbit_r5_c14 BL[14] BR[14] WL[5] vdd gnd cell_6t
Xbit_r5_c15 BL[15] BR[15] WL[5] vdd gnd cell_6t
Xbit_r6_c0 BL[0] BR[0] WL[6] vdd gnd cell_6t
Xbit_r6_c1 BL[1] BR[1] WL[6] vdd gnd cell_6t
Xbit_r6_c2 BL[2] BR[2] WL[6] vdd gnd cell_6t
Xbit_r6_c3 BL[3] BR[3] WL[6] vdd gnd cell_6t
Xbit_r6_c4 BL[4] BR[4] WL[6] vdd gnd cell_6t
Xbit_r6_c5 BL[5] BR[5] WL[6] vdd gnd cell_6t
Xbit_r6_c6 BL[6] BR[6] WL[6] vdd gnd cell_6t
Xbit_r6_c7 BL[7] BR[7] WL[6] vdd gnd cell_6t
Xbit_r6_c8 BL[8] BR[8] WL[6] vdd gnd cell_6t
Xbit_r6_c9 BL[9] BR[9] WL[6] vdd gnd cell_6t
Xbit_r6_c10 BL[10] BR[10] WL[6] vdd gnd cell_6t
Xbit_r6_c11 BL[11] BR[11] WL[6] vdd gnd cell_6t
Xbit_r6_c12 BL[12] BR[12] WL[6] vdd gnd cell_6t
Xbit_r6_c13 BL[13] BR[13] WL[6] vdd gnd cell_6t
Xbit_r6_c14 BL[14] BR[14] WL[6] vdd gnd cell_6t
Xbit_r6_c15 BL[15] BR[15] WL[6] vdd gnd cell_6t
Xbit_r7_c0 BL[0] BR[0] WL[7] vdd gnd cell_6t
Xbit_r7_c1 BL[1] BR[1] WL[7] vdd gnd cell_6t
Xbit_r7_c2 BL[2] BR[2] WL[7] vdd gnd cell_6t
Xbit_r7_c3 BL[3] BR[3] WL[7] vdd gnd cell_6t
Xbit_r7_c4 BL[4] BR[4] WL[7] vdd gnd cell_6t
Xbit_r7_c5 BL[5] BR[5] WL[7] vdd gnd cell_6t
Xbit_r7_c6 BL[6] BR[6] WL[7] vdd gnd cell_6t
Xbit_r7_c7 BL[7] BR[7] WL[7] vdd gnd cell_6t
```

```
Xbit_r7_c8 BL[8] BR[8] WL[7] vdd gnd cell_6t
Xbit_r7_c9 BL[9] BR[9] WL[7] vdd gnd cell_6t
Xbit_r7_c10 BL[10] BR[10] WL[7] vdd gnd cell_6t
Xbit_r7_c11 BL[11] BR[11] WL[7] vdd gnd cell_6t
Xbit_r7_c12 BL[12] BR[12] WL[7] vdd gnd cell_6t
Xbit_r7_c13 BL[13] BR[13] WL[7] vdd gnd cell_6t
Xbit_r7_c14 BL[14] BR[14] WL[7] vdd gnd cell_6t
Xbit_r7_c15 BL[15] BR[15] WL[7] vdd gnd cell_6t
Xbit_r8_c0 BL[0] BR[0] WL[8] vdd gnd cell_6t
Xbit_r8_c1 BL[1] BR[1] WL[8] vdd gnd cell_6t
Xbit_r8_c2 BL[2] BR[2] WL[8] vdd gnd cell_6t
Xbit_r8_c3 BL[3] BR[3] WL[8] vdd gnd cell_6t
Xbit_r8_c4 BL[4] BR[4] WL[8] vdd gnd cell_6t
Xbit_r8_c5 BL[5] BR[5] WL[8] vdd gnd cell_6t
Xbit_r8_c6 BL[6] BR[6] WL[8] vdd gnd cell_6t
Xbit_r8_c7 BL[7] BR[7] WL[8] vdd gnd cell_6t
Xbit_r8_c8 BL[8] BR[8] WL[8] vdd gnd cell_6t
Xbit_r8_c9 BL[9] BR[9] WL[8] vdd gnd cell_6t
Xbit_r8_c10 BL[10] BR[10] WL[8] vdd gnd cell_6t
Xbit_r8_c11 BL[11] BR[11] WL[8] vdd gnd cell_6t
Xbit_r8_c12 BL[12] BR[12] WL[8] vdd gnd cell_6t
Xbit_r8_c13 BL[13] BR[13] WL[8] vdd gnd cell_6t
Xbit_r8_c14 BL[14] BR[14] WL[8] vdd gnd cell_6t
Xbit_r8_c15 BL[15] BR[15] WL[8] vdd gnd cell_6t
Xbit_r9_c0 BL[0] BR[0] WL[9] vdd gnd cell_6t
Xbit_r9_c1 BL[1] BR[1] WL[9] vdd gnd cell_6t
Xbit_r9_c2 BL[2] BR[2] WL[9] vdd gnd cell_6t
Xbit_r9_c3 BL[3] BR[3] WL[9] vdd gnd cell_6t
Xbit_r9_c4 BL[4] BR[4] WL[9] vdd gnd cell_6t
Xbit_r9_c5 BL[5] BR[5] WL[9] vdd gnd cell_6t
Xbit_r9_c6 BL[6] BR[6] WL[9] vdd gnd cell_6t
Xbit_r9_c7 BL[7] BR[7] WL[9] vdd gnd cell_6t
Xbit_r9_c8 BL[8] BR[8] WL[9] vdd gnd cell_6t
Xbit_r9_c9 BL[9] BR[9] WL[9] vdd gnd cell_6t
Xbit_r9_c10 BL[10] BR[10] WL[9] vdd gnd cell_6t
Xbit_r9_c11 BL[11] BR[11] WL[9] vdd gnd cell_6t
Xbit_r9_c12 BL[12] BR[12] WL[9] vdd gnd cell_6t
Xbit_r9_c13 BL[13] BR[13] WL[9] vdd gnd cell_6t
Xbit_r9_c14 BL[14] BR[14] WL[9] vdd gnd cell_6t
Xbit_r9_c15 BL[15] BR[15] WL[9] vdd gnd cell_6t
Xbit_r10_c0 BL[0] BR[0] WL[10] vdd gnd cell_6t
Xbit_r10_c1 BL[1] BR[1] WL[10] vdd gnd cell_6t
Xbit_r10_c2 BL[2] BR[2] WL[10] vdd gnd cell_6t
Xbit_r10_c3 BL[3] BR[3] WL[10] vdd gnd cell_6t
Xbit_r10_c4 BL[4] BR[4] WL[10] vdd gnd cell_6t
Xbit_r10_c5 BL[5] BR[5] WL[10] vdd gnd cell_6t
Xbit_r10_c6 BL[6] BR[6] WL[10] vdd gnd cell_6t
Xbit_r10_c7 BL[7] BR[7] WL[10] vdd gnd cell_6t
Xbit_r10_c8 BL[8] BR[8] WL[10] vdd gnd cell_6t
Xbit_r10_c9 BL[9] BR[9] WL[10] vdd gnd cell_6t
Xbit_r10_c10 BL[10] BR[10] WL[10] vdd gnd cell_6t
Xbit_r10_c11 BL[11] BR[11] WL[10] vdd gnd cell_6t
Xbit_r10_c12 BL[12] BR[12] WL[10] vdd gnd cell_6t
Xbit_r10_c13 BL[13] BR[13] WL[10] vdd gnd cell_6t
Xbit_r10_c14 BL[14] BR[14] WL[10] vdd gnd cell_6t
Xbit_r10_c15 BL[15] BR[15] WL[10] vdd gnd cell_6t
Xbit_r11_c0 BL[0] BR[0] WL[11] vdd gnd cell_6t
Xbit_r11_c1 BL[1] BR[1] WL[11] vdd gnd cell_6t
Xbit_r11_c2 BL[2] BR[2] WL[11] vdd gnd cell_6t
Xbit_r11_c3 BL[3] BR[3] WL[11] vdd gnd cell_6t
Xbit_r11_c4 BL[4] BR[4] WL[11] vdd gnd cell_6t
Xbit_r11_c5 BL[5] BR[5] WL[11] vdd gnd cell_6t
Xbit_r11_c6 BL[6] BR[6] WL[11] vdd gnd cell_6t
Xbit_r11_c7 BL[7] BR[7] WL[11] vdd gnd cell_6t
Xbit_r11_c8 BL[8] BR[8] WL[11] vdd gnd cell_6t
Xbit_r11_c9 BL[9] BR[9] WL[11] vdd gnd cell_6t
Xbit_r11_c10 BL[10] BR[10] WL[11] vdd gnd cell_6t
Xbit_r11_c11 BL[11] BR[11] WL[11] vdd gnd cell_6t
Xbit_r11_c12 BL[12] BR[12] WL[11] vdd gnd cell_6t
```

```
Xbit_r11_c13 BL[13] BR[13] WL[11] vdd gnd cell_6t
Xbit_r11_c14 BL[14] BR[14] WL[11] vdd gnd cell_6t
Xbit_r11_c15 BL[15] BR[15] WL[11] vdd gnd cell_6t
Xbit_r12_c0 BL[0] BR[0] WL[12] vdd gnd cell_6t
Xbit_r12_c1 BL[1] BR[1] WL[12] vdd gnd cell_6t
Xbit_r12_c2 BL[2] BR[2] WL[12] vdd gnd cell_6t
Xbit_r12_c3 BL[3] BR[3] WL[12] vdd gnd cell_6t
Xbit_r12_c4 BL[4] BR[4] WL[12] vdd gnd cell_6t
Xbit_r12_c5 BL[5] BR[5] WL[12] vdd gnd cell_6t
Xbit_r12_c6 BL[6] BR[6] WL[12] vdd gnd cell_6t
Xbit_r12_c7 BL[7] BR[7] WL[12] vdd gnd cell_6t
Xbit_r12_c8 BL[8] BR[8] WL[12] vdd gnd cell_6t
Xbit_r12_c9 BL[9] BR[9] WL[12] vdd gnd cell_6t
Xbit_r12_c10 BL[10] BR[10] WL[12] vdd gnd cell_6t
Xbit_r12_c11 BL[11] BR[11] WL[12] vdd gnd cell_6t
Xbit_r12_c12 BL[12] BR[12] WL[12] vdd gnd cell_6t
Xbit_r12_c13 BL[13] BR[13] WL[12] vdd gnd cell_6t
Xbit_r12_c14 BL[14] BR[14] WL[12] vdd gnd cell_6t
Xbit_r12_c15 BL[15] BR[15] WL[12] vdd gnd cell_6t
Xbit_r13_c0 BL[0] BR[0] WL[13] vdd gnd cell_6t
Xbit_r13_c1 BL[1] BR[1] WL[13] vdd gnd cell_6t
Xbit_r13_c2 BL[2] BR[2] WL[13] vdd gnd cell_6t
Xbit_r13_c3 BL[3] BR[3] WL[13] vdd gnd cell_6t
Xbit_r13_c4 BL[4] BR[4] WL[13] vdd gnd cell_6t
Xbit_r13_c5 BL[5] BR[5] WL[13] vdd gnd cell_6t
Xbit_r13_c6 BL[6] BR[6] WL[13] vdd gnd cell_6t
Xbit_r13_c7 BL[7] BR[7] WL[13] vdd gnd cell_6t
Xbit_r13_c8 BL[8] BR[8] WL[13] vdd gnd cell_6t
Xbit_r13_c9 BL[9] BR[9] WL[13] vdd gnd cell_6t
Xbit_r13_c10 BL[10] BR[10] WL[13] vdd gnd cell_6t
Xbit_r13_c11 BL[11] BR[11] WL[13] vdd gnd cell_6t
Xbit_r13_c12 BL[12] BR[12] WL[13] vdd gnd cell_6t
Xbit_r13_c13 BL[13] BR[13] WL[13] vdd gnd cell_6t
Xbit_r13_c14 BL[14] BR[14] WL[13] vdd gnd cell_6t
Xbit_r13_c15 BL[15] BR[15] WL[13] vdd gnd cell_6t
Xbit_r14_c0 BL[0] BR[0] WL[14] vdd gnd cell_6t
Xbit_r14_c1 BL[1] BR[1] WL[14] vdd gnd cell_6t
Xbit_r14_c2 BL[2] BR[2] WL[14] vdd gnd cell_6t
Xbit_r14_c3 BL[3] BR[3] WL[14] vdd gnd cell_6t
Xbit_r14_c4 BL[4] BR[4] WL[14] vdd gnd cell_6t
Xbit_r14_c5 BL[5] BR[5] WL[14] vdd gnd cell_6t
Xbit_r14_c6 BL[6] BR[6] WL[14] vdd gnd cell_6t
Xbit_r14_c7 BL[7] BR[7] WL[14] vdd gnd cell_6t
Xbit_r14_c8 BL[8] BR[8] WL[14] vdd gnd cell_6t
Xbit_r14_c9 BL[9] BR[9] WL[14] vdd gnd cell_6t
Xbit_r14_c10 BL[10] BR[10] WL[14] vdd gnd cell_6t
Xbit_r14_c11 BL[11] BR[11] WL[14] vdd gnd cell_6t
Xbit_r14_c12 BL[12] BR[12] WL[14] vdd gnd cell_6t
Xbit_r14_c13 BL[13] BR[13] WL[14] vdd gnd cell_6t
Xbit_r14_c14 BL[14] BR[14] WL[14] vdd gnd cell_6t
Xbit_r14_c15 BL[15] BR[15] WL[14] vdd gnd cell_6t
Xbit_r15_c0 BL[0] BR[0] WL[15] vdd gnd cell_6t
Xbit_r15_c1 BL[1] BR[1] WL[15] vdd gnd cell_6t
Xbit_r15_c2 BL[2] BR[2] WL[15] vdd gnd cell_6t
Xbit_r15_c3 BL[3] BR[3] WL[15] vdd gnd cell_6t
Xbit_r15_c4 BL[4] BR[4] WL[15] vdd gnd cell_6t
Xbit_r15_c5 BL[5] BR[5] WL[15] vdd gnd cell_6t
Xbit_r15_c6 BL[6] BR[6] WL[15] vdd gnd cell_6t
Xbit_r15_c7 BL[7] BR[7] WL[15] vdd gnd cell_6t
Xbit_r15_c8 BL[8] BR[8] WL[15] vdd gnd cell_6t
Xbit_r15_c9 BL[9] BR[9] WL[15] vdd gnd cell_6t
Xbit_r15_c10 BL[10] BR[10] WL[15] vdd gnd cell_6t
Xbit_r15_c11 BL[11] BR[11] WL[15] vdd gnd cell_6t
Xbit_r15_c12 BL[12] BR[12] WL[15] vdd gnd cell_6t
Xbit_r15_c13 BL[13] BR[13] WL[15] vdd gnd cell_6t
Xbit_r15_c14 BL[14] BR[14] WL[15] vdd gnd cell_6t
Xbit_r15_c15 BL[15] BR[15] WL[15] vdd gnd cell_6t
.ENDS bitcell_array
```

```
.subckt pmos3 D G S B
Mpmos D G S B pmos_vtg m=1 w=0.090000u l=0.050000u
.ends pmos3

.subckt pmos1 D G S B
Mpmos D G S B pmos_vtg m=1 w=0.180000u l=0.050000u
.ends pmos1

.SUBCKT precharge BL BR clk vdd
xpmos1 BL clk vdd vdd pmos1
xpmos2 BR clk vdd vdd pmos1
xpmos3 BL clk BR vdd pmos3
xpcont  prechargecontact
.ENDS precharge

.SUBCKT precharge_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4]
    BR[4] BL[5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10]
    BR[10] BL[11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR
    [15] clk vdd
Xpre_c0 BL[0] BR[0] clk vdd precharge
Xpre_c1 BL[1] BR[1] clk vdd precharge
Xpre_c2 BL[2] BR[2] clk vdd precharge
Xpre_c3 BL[3] BR[3] clk vdd precharge
Xpre_c4 BL[4] BR[4] clk vdd precharge
Xpre_c5 BL[5] BR[5] clk vdd precharge
Xpre_c6 BL[6] BR[6] clk vdd precharge
Xpre_c7 BL[7] BR[7] clk vdd precharge
Xpre_c8 BL[8] BR[8] clk vdd precharge
Xpre_c9 BL[9] BR[9] clk vdd precharge
Xpre_c10 BL[10] BR[10] clk vdd precharge
Xpre_c11 BL[11] BR[11] clk vdd precharge
Xpre_c12 BL[12] BR[12] clk vdd precharge
Xpre_c13 BL[13] BR[13] clk vdd precharge
Xpre_c14 BL[14] BR[14] clk vdd precharge
Xpre_c15 BL[15] BR[15] clk vdd precharge
.ENDS precharge_array

.subckt a_nmos1 D G S B
Mnmos D G S B nmos_vtg m=1 w=0.180000u l=0.050000u
.ends a_nmos1

.SUBCKT mux_a BL BLB BL_O BLB_O sel gnd
xa_nmos1 BL sel BL_O gnd a_nmos1
xa_nmos2 BLB sel BLB_O gnd a_nmos1
.ENDS mux_a

.subckt a_nmos2 D G S B
Mnmos D G S B nmos_vtg m=1 w=0.180000u l=0.050000u
.ends a_nmos2

.SUBCKT mux_abar BL BLB BL_O BLB_O sel_bar gnd
xa_nmos3 BL sel_bar BL_O gnd a_nmos2
xa_nmos4 BLB sel_bar BLB_O gnd a_nmos2
.ENDS mux_abar

.SUBCKT column_mux_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4]
     BR[4] BL[5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10]
    BR[10] BL[11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR
    [15] BL_out[0] BR_out[0] BL_out[2] BR_out[2] BL_out[4] BR_out[4] BL_out[6]
     BR_out[6] BL_out[8] BR_out[8] BL_out[10] BR_out[10] BL_out[12] BR_out[12]
     BL_out[14] BR_out[14] sel[0] sel_bar[0] gnd
Xmux_a00 BL[0] BR[0] BL_out[0] BR_out[0] sel[0] gnd mux_a
Xmux_abar01 BL[1] BR[1] BL_out[0] BR_out[0] sel_bar[0] gnd mux_abar
Xmux_a02 BL[2] BR[2] BL_out[2] BR_out[2] sel[0] gnd mux_a
Xmux_abar03 BL[3] BR[3] BL_out[2] BR_out[2] sel_bar[0] gnd mux_abar
Xmux_a04 BL[4] BR[4] BL_out[4] BR_out[4] sel[0] gnd mux_a
Xmux_abar05 BL[5] BR[5] BL_out[4] BR_out[4] sel_bar[0] gnd mux_abar
Xmux_a06 BL[6] BR[6] BL_out[6] BR_out[6] sel[0] gnd mux_a
Xmux_abar07 BL[7] BR[7] BL_out[6] BR_out[6] sel_bar[0] gnd mux_abar
```

```
Xmux_a08 BL[8] BR[8] BL_out[8] BR_out[8] sel[0] gnd mux_a
Xmux_abar09 BL[9] BR[9] BL_out[8] BR_out[8] sel_bar[0] gnd mux_abar
Xmux_a010 BL[10] BR[10] BL_out[10] BR_out[10] sel[0] gnd mux_a
Xmux_abar011 BL[11] BR[11] BL_out[10] BR_out[10] sel_bar[0] gnd mux_abar
Xmux_a012 BL[12] BR[12] BL_out[12] BR_out[12] sel[0] gnd mux_a
Xmux_abar013 BL[13] BR[13] BL_out[12] BR_out[12] sel_bar[0] gnd mux_abar
Xmux_a014 BL[14] BR[14] BL_out[14] BR_out[14] sel[0] gnd mux_a
Xmux_abar015 BL[15] BR[15] BL_out[14] BR_out[14] sel_bar[0] gnd mux_abar
.ENDS column_mux_array

.subckt sense_amp BL BR sclk D vdd gnd
mnl BL sclk BL_int gnd nmos_vtg w=360.000000n l=50.000000n
mnr BR sclk BR_int gnd nmos_vtg w=360.000000n l=50.000000n
mP1 D_bar D_bar vdd vdd pmos_vtg w=360.000000n l=50.000000n
mP2 D D_bar vdd vdd pmos_vtg w=360.000000n l=50.000000n
mN1 D_bar BL_int vgnd gnd nmos_vtg w=360.000000n l=50.000000n
mN2 D BR_int vgnd gnd nmos_vtg w=360.000000n l=50.000000n
ms vgnd sclk gnd gnd nmos_vtg w=720.000000n l=50.000000n
.ends sense_amp_cell

.SUBCKT sense_amp_array Data_out[0] BL_out[0] BR_out[0] Data_out[1] BL_out[2]
    BR_out[2] Data_out[2] BL_out[4] BR_out[4] Data_out[3] BL_out[6] BR_out[6]
    Data_out[4] BL_out[8] BR_out[8] Data_out[5] BL_out[10] BR_out[10] Data_out
    [6] BL_out[12] BR_out[12] Data_out[7] BL_out[14] BR_out[14] SCLK vdd gnd
Xsa_d0 BL_out[0] BR_out[0] SCLK Data_out[0] vdd gnd sense_amp
Xsa_d1 BL_out[2] BR_out[2] SCLK Data_out[1] vdd gnd sense_amp
Xsa_d2 BL_out[4] BR_out[4] SCLK Data_out[2] vdd gnd sense_amp
Xsa_d3 BL_out[6] BR_out[6] SCLK Data_out[3] vdd gnd sense_amp
Xsa_d4 BL_out[8] BR_out[8] SCLK Data_out[4] vdd gnd sense_amp
Xsa_d5 BL_out[10] BR_out[10] SCLK Data_out[5] vdd gnd sense_amp
Xsa_d6 BL_out[12] BR_out[12] SCLK Data_out[6] vdd gnd sense_amp
Xsa_d7 BL_out[14] BR_out[14] SCLK Data_out[7] vdd gnd sense_amp
.ENDS sense_amp_array

.subckt write_driver din BL BR en vdd gnd
#inverters for enable and data input
minP BL_bar din vdd vdd pmos_vtg w=450.000000n l=50.000000n
minN BL_bar din gnd gnd nmos_vtg w=225.000000n l=50.000000n
moutP en_bar en vdd vdd pmos_vtg w=450.000000n l=50.000000n
moutN en_bar en gnd gnd nmos_vtg w=225.000000n l=50.000000n
#tristate for BL
mout0P int1 BL_bar vdd vdd pmos_vtg w=450.000000n l=50.000000n
mout0P2 BL en_bar int1 vdd pmos_vtg w=450.000000n l=50.000000n
mout0N BL en int2 gnd nmos_vtg w=225.000000n l=50.000000n
mout0N2 int2 BL_bar gnd gnd nmos_vtg w=225.000000n l=50.000000n
#tristate for BR
mout1P int3 din vdd vdd pmos_vtg w=450.000000n l=50.000000n
mout1P2 BR en_bar int3 vdd pmos_vtg w=450.000000n l=50.000000n
mout1N BR en int4 gnd nmos_vtg w=225.000000n l=50.000000n
mout1N2 int4 din gnd gnd nmos_vtg w=225.000000n l=50.000000n
.ends write_driver

.SUBCKT write_driver_array BL_out[0] BR_out[0] Data_in[0] BL_out[2] BR_out[2]
    Data_in[1] BL_out[4] BR_out[4] Data_in[2] BL_out[6] BR_out[6] Data_in[3]
    BL_out[8] BR_out[8] Data_in[4] BL_out[10] BR_out[10] Data_in[5] BL_out[12]
     BR_out[12] Data_in[6] BL_out[14] BR_out[14] Data_in[7] EN vdd gnd
Xwrite_d0 Data_in[0] BL_out[0] BR_out[0] EN vdd gnd write_driver
Xwrite_d1 Data_in[1] BL_out[2] BR_out[2] EN vdd gnd write_driver
Xwrite_d2 Data_in[2] BL_out[4] BR_out[4] EN vdd gnd write_driver
Xwrite_d3 Data_in[3] BL_out[6] BR_out[6] EN vdd gnd write_driver
Xwrite_d4 Data_in[4] BL_out[8] BR_out[8] EN vdd gnd write_driver
Xwrite_d5 Data_in[5] BL_out[10] BR_out[10] EN vdd gnd write_driver
Xwrite_d6 Data_in[6] BL_out[12] BR_out[12] EN vdd gnd write_driver
Xwrite_d7 Data_in[7] BL_out[14] BR_out[14] EN vdd gnd write_driver
.ENDS write_driver_array

.subckt tx_dec D G S B
Mnmos D G S B nmos_vtg m=1 w=0.180000u l=0.050000u
.ends tx_dec
```

```
.subckt Xnwl_driver D G S B
Mnmos D G S B nmos_vtg m=2 w=0.090000u l=0.050000u
.ends Xnwl_driver

.subckt Xpwl_driver D G S B
Mpmos D G S B pmos_vtg m=2 w=0.270000u l=0.050000u
.ends Xpwl_driver

.SUBCKT wl_driver A Z vdd gnd
Xnwl_driver Z A gnd gnd Xnwl_driver
Xpwl_driver Z A vdd vdd Xpwl_driver
.ENDS wl_driver

.subckt pre_tx D G S B
Mpmos D G S B pmos_vtg m=1 w=0.180000u l=0.050000u
.ends pre_tx

.SUBCKT address_decoder ain[0] ain_bar[0] ain[1] ain_bar[1] ain[2] ain_bar[2]
    ain[3] ain_bar[3] WL[0] WL[1] WL[2] WL[3] WL[4] WL[5] WL[6] WL[7] WL[8] WL
    [9] WL[10] WL[11] WL[12] WL[13] WL[14] WL[15] dclk vdd gnd
Xinv0 WL_bar[0] WL[0] vdd gnd wl_driver
Xinv1 WL_bar[1] WL[1] vdd gnd wl_driver
Xinv2 WL_bar[2] WL[2] vdd gnd wl_driver
Xinv3 WL_bar[3] WL[3] vdd gnd wl_driver
Xinv4 WL_bar[4] WL[4] vdd gnd wl_driver
Xinv5 WL_bar[5] WL[5] vdd gnd wl_driver
Xinv6 WL_bar[6] WL[6] vdd gnd wl_driver
Xinv7 WL_bar[7] WL[7] vdd gnd wl_driver
Xinv8 WL_bar[8] WL[8] vdd gnd wl_driver
Xinv9 WL_bar[9] WL[9] vdd gnd wl_driver
Xinv10 WL_bar[10] WL[10] vdd gnd wl_driver
Xinv11 WL_bar[11] WL[11] vdd gnd wl_driver
Xinv12 WL_bar[12] WL[12] vdd gnd wl_driver
Xinv13 WL_bar[13] WL[13] vdd gnd wl_driver
Xinv14 WL_bar[14] WL[14] vdd gnd wl_driver
Xinv15 WL_bar[15] WL[15] vdd gnd wl_driver
Xpre0 vdd dclk WL_bar[0] vdd pre_tx
Xpre1 vdd dclk WL_bar[1] vdd pre_tx
Xpre2 vdd dclk WL_bar[2] vdd pre_tx
Xpre3 vdd dclk WL_bar[3] vdd pre_tx
Xpre4 vdd dclk WL_bar[4] vdd pre_tx
Xpre5 vdd dclk WL_bar[5] vdd pre_tx
Xpre6 vdd dclk WL_bar[6] vdd pre_tx
Xpre7 vdd dclk WL_bar[7] vdd pre_tx
Xpre8 vdd dclk WL_bar[8] vdd pre_tx
Xpre9 vdd dclk WL_bar[9] vdd pre_tx
Xpre10 vdd dclk WL_bar[10] vdd pre_tx
Xpre11 vdd dclk WL_bar[11] vdd pre_tx
Xpre12 vdd dclk WL_bar[12] vdd pre_tx
Xpre13 vdd dclk WL_bar[13] vdd pre_tx
Xpre14 vdd dclk WL_bar[14] vdd pre_tx
Xpre15 vdd dclk WL_bar[15] vdd pre_tx
xtx_WL0_A0 gnd ain_bar[0] i0_0 gnd tx_dec
xtx_WL0_A1 i0_0 ain_bar[1] i0_1 gnd tx_dec
xtx_WL0_A2 i0_1 ain_bar[2] i0_2 gnd tx_dec
xtx_WL0_A3 i0_2 ain_bar[3] WL_bar[0] gnd tx_dec
xtx_WL1_A0 gnd ain[0] i1_0 gnd tx_dec
xtx_WL1_A1 i1_0 ain_bar[1] i1_1 gnd tx_dec
xtx_WL1_A2 i1_1 ain_bar[2] i1_2 gnd tx_dec
xtx_WL1_A3 i1_2 ain_bar[3] WL_bar[1] gnd tx_dec
xtx_WL2_A0 gnd ain_bar[0] i2_0 gnd tx_dec
xtx_WL2_A1 i2_0 ain[1] i2_1 gnd tx_dec
xtx_WL2_A2 i2_1 ain_bar[2] i2_2 gnd tx_dec
xtx_WL2_A3 i2_2 ain_bar[3] WL_bar[2] gnd tx_dec
xtx_WL3_A0 gnd ain[0] i3_0 gnd tx_dec
xtx_WL3_A1 i3_0 ain[1] i3_1 gnd tx_dec
xtx_WL3_A2 i3_1 ain_bar[2] i3_2 gnd tx_dec
xtx_WL3_A3 i3_2 ain_bar[3] WL_bar[3] gnd tx_dec
```

```
xtx_WL4_A0 gnd ain_bar[0] i4_0 gnd tx_dec
xtx_WL4_A1 i4_0 ain_bar[1] i4_1 gnd tx_dec
xtx_WL4_A2 i4_1 ain[2] i4_2 gnd tx_dec
xtx_WL4_A3 i4_2 ain_bar[3] WL_bar[4] gnd tx_dec
xtx_WL5_A0 gnd ain[0] i5_0 gnd tx_dec
xtx_WL5_A1 i5_0 ain_bar[1] i5_1 gnd tx_dec
xtx_WL5_A2 i5_1 ain[2] i5_2 gnd tx_dec
xtx_WL5_A3 i5_2 ain_bar[3] WL_bar[5] gnd tx_dec
xtx_WL6_A0 gnd ain_bar[0] i6_0 gnd tx_dec
xtx_WL6_A1 i6_0 ain[1] i6_1 gnd tx_dec
xtx_WL6_A2 i6_1 ain[2] i6_2 gnd tx_dec
xtx_WL6_A3 i6_2 ain_bar[3] WL_bar[6] gnd tx_dec
xtx_WL7_A0 gnd ain[0] i7_0 gnd tx_dec
xtx_WL7_A1 i7_0 ain[1] i7_1 gnd tx_dec
xtx_WL7_A2 i7_1 ain[2] i7_2 gnd tx_dec
xtx_WL7_A3 i7_2 ain_bar[3] WL_bar[7] gnd tx_dec
xtx_WL8_A0 gnd ain_bar[0] i8_0 gnd tx_dec
xtx_WL8_A1 i8_0 ain_bar[1] i8_1 gnd tx_dec
xtx_WL8_A2 i8_1 ain_bar[2] i8_2 gnd tx_dec
xtx_WL8_A3 i8_2 ain[3] WL_bar[8] gnd tx_dec
xtx_WL9_A0 gnd ain[0] i9_0 gnd tx_dec
xtx_WL9_A1 i9_0 ain_bar[1] i9_1 gnd tx_dec
xtx_WL9_A2 i9_1 ain_bar[2] i9_2 gnd tx_dec
xtx_WL9_A3 i9_2 ain[3] WL_bar[9] gnd tx_dec
xtx_WL10_A0 gnd ain_bar[0] i10_0 gnd tx_dec
xtx_WL10_A1 i10_0 ain[1] i10_1 gnd tx_dec
xtx_WL10_A2 i10_1 ain_bar[2] i10_2 gnd tx_dec
xtx_WL10_A3 i10_2 ain[3] WL_bar[10] gnd tx_dec
xtx_WL11_A0 gnd ain[0] i11_0 gnd tx_dec
xtx_WL11_A1 i11_0 ain[1] i11_1 gnd tx_dec
xtx_WL11_A2 i11_1 ain_bar[2] i11_2 gnd tx_dec
xtx_WL11_A3 i11_2 ain[3] WL_bar[11] gnd tx_dec
xtx_WL12_A0 gnd ain_bar[0] i12_0 gnd tx_dec
xtx_WL12_A1 i12_0 ain_bar[1] i12_1 gnd tx_dec
xtx_WL12_A2 i12_1 ain[2] i12_2 gnd tx_dec
xtx_WL12_A3 i12_2 ain[3] WL_bar[12] gnd tx_dec
xtx_WL13_A0 gnd ain[0] i13_0 gnd tx_dec
xtx_WL13_A1 i13_0 ain_bar[1] i13_1 gnd tx_dec
xtx_WL13_A2 i13_1 ain[2] i13_2 gnd tx_dec
xtx_WL13_A3 i13_2 ain[3] WL_bar[13] gnd tx_dec
xtx_WL14_A0 gnd ain_bar[0] i14_0 gnd tx_dec
xtx_WL14_A1 i14_0 ain[1] i14_1 gnd tx_dec
xtx_WL14_A2 i14_1 ain[2] i14_2 gnd tx_dec
xtx_WL14_A3 i14_2 ain[3] WL_bar[14] gnd tx_dec
xtx_WL15_A0 gnd ain[0] i15_0 gnd tx_dec
xtx_WL15_A1 i15_0 ain[1] i15_1 gnd tx_dec
xtx_WL15_A2 i15_1 ain[2] i15_2 gnd tx_dec
xtx_WL15_A3 i15_2 ain[3] WL_bar[15] gnd tx_dec
.ENDS address_decoder

.subckt control_nor_nand a CSb clk out vdd gnd
xcntrl_nor a CSb int vdd gnd control_nor2
xcntrl_nand int clk out vdd gnd control_nand2
.ends control_nor_nand

.subckt control_nor2 a b z vdd gnd
mP1 int1 a vdd vdd pmos_vtg w=360.000000n l=50.000000n
mP2 z b int1 vdd pmos_vtg w=360.000000n l=50.000000n
mN1 z b gnd gnd nmos_vtg w=90.000000n l=50.000000n
mN2 z a gnd gnd nmos_vtg w=90.000000n l=50.000000n
.ends control_nor2

.subckt control_nand2 a b z vdd gnd
m1 z a net1 gnd NMOS_VTG  w=90.000000n l=50.000000n
m0 net1 b gnd gnd NMOS_VTG  w=90.000000n l=50.000000n
m3 z b vdd vdd PMOS_VTG  w=90.000000n l=50.000000n
m2 z a vdd vdd PMOS_VTG  w=90.000000n l=50.000000n
.ENDS control_nand2
```

```
.subckt Xncntrl_inv D G S B
Mnmos D G S B nmos_vtg m=2 w=0.090000u l=0.050000u
.ends Xncntrl_inv

.subckt Xpcntrl_inv D G S B
Mpmos D G S B pmos_vtg m=2 w=0.270000u l=0.050000u
.ends Xpcntrl_inv

.SUBCKT cntrl_inv A Z vdd gnd
Xncntrl_inv Z A gnd gnd Xncntrl_inv
Xpcntrl_inv Z A vdd vdd Xpcntrl_inv
.ENDS cntrl_inv

.subckt ms_flop din dout dout_bar clk vdd gnd
xmaster din mout mout_bar clk clk_bar vdd gnd dlatch
xslave mout_bar dout_bar dout clk_bar clk_nn vdd gnd dlatch
.ends flop

.subckt dlatch din dout dout_bar clk clk_bar vdd gnd
#clk inverter
mPff1 clk_bar clk vdd vdd PMOS_VTG W=180.0n L=50n m=1
mNff1 clk_bar clk gnd gnd NMOS_VTG W=90n L=50n m=1
#transmission gate 1
mtmP1 din clk int1 vdd PMOS_VTG W=180.0n L=50n m=1
mtmN1 din clk_bar int1 gnd NMOS_VTG W=90n L=50n m=1
#foward inverter
mPff3 dout_bar int1 vdd vdd PMOS_VTG W=180.0n L=50n m=1
mNff3 dout_bar int1 gnd gnd NMOS_VTG W=90n L=50n m=1
#backward inverter
mPff4 dout dout_bar vdd vdd PMOS_VTG W=180.0n L=50n m=1
mNf4 dout dout_bar gnd gnd NMOS_VTG W=90n L=50n m=1
#transmission gate 2
mtmP2 int1 clk_bar dout vdd PMOS_VTG W=180.0n L=50n m=1
mtmN2 int1 clk dout gnd NMOS_VTG W=90n L=50n m=1
.ends dlatch

.SUBCKT control_logic CSb WEb OEb clk OE EN SCLK vdd gnd
XflopCSb CSb CSbar CS clk vdd gnd ms_flop
XflopOEb OEb OEbar OE clk vdd gnd ms_flop
XcontrlOEb OEbar CSbar clk SCLK_bar vdd gnd control_nor_nand
XinvOEb SCLK_bar SCLK vdd gnd cntrl_inv
XflopWEb WEb WEbar WE clk vdd gnd ms_flop
XcontrlWEb WEbar CSbar clk EN_bar vdd gnd control_nor_nand
XinvWEb EN_bar EN vdd gnd cntrl_inv
.ENDS control_logic

.SUBCKT addr_row_flop_array ADDR[0] ain[0] ain_bar[0] ADDR[1] ain[1] ain_bar
    [1] ADDR[2] ain[2] ain_bar[2] ADDR[3] ain[3] ain_bar[3] clk vdd gnd
Xflopaddr_row0 ADDR[0] ain[0] ain_bar[0] clk vdd gnd ms_flop
Xflopaddr_row1 ADDR[1] ain[1] ain_bar[1] clk vdd gnd ms_flop
Xflopaddr_row2 ADDR[2] ain[2] ain_bar[2] clk vdd gnd ms_flop
Xflopaddr_row3 ADDR[3] ain[3] ain_bar[3] clk vdd gnd ms_flop
.ENDS addr_row_flop_array

.SUBCKT addr_col_flop_array ADDR[4] sel_bar[0] sel[0] clk vdd gnd
Xflopaddr_col0 ADDR[4] sel_bar[0] sel[0] clk vdd gnd ms_flop
.ENDS addr_col_flop_array

.SUBCKT data_in_flop_array DATA[0] Data_in[0] Data_in_bar[0] DATA[1] Data_in
    [1] Data_in_bar[1] DATA[2] Data_in[2] Data_in_bar[2] DATA[3] Data_in[3]
    Data_in_bar[3] DATA[4] Data_in[4] Data_in_bar[4] DATA[5] Data_in[5]
    Data_in_bar[5] DATA[6] Data_in[6] Data_in_bar[6] DATA[7] Data_in[7]
    Data_in_bar[7] clk vdd gnd
Xflopdata_in0 DATA[0] Data_in[0] Data_in_bar[0] clk vdd gnd ms_flop
Xflopdata_in1 DATA[1] Data_in[1] Data_in_bar[1] clk vdd gnd ms_flop
Xflopdata_in2 DATA[2] Data_in[2] Data_in_bar[2] clk vdd gnd ms_flop
Xflopdata_in3 DATA[3] Data_in[3] Data_in_bar[3] clk vdd gnd ms_flop
Xflopdata_in4 DATA[4] Data_in[4] Data_in_bar[4] clk vdd gnd ms_flop
Xflopdata_in5 DATA[5] Data_in[5] Data_in_bar[5] clk vdd gnd ms_flop
```

```
Xflopdata_in6 DATA[6] Data_in[6] Data_in_bar[6] clk vdd gnd ms_flop
Xflopdata_in7 DATA[7] Data_in[7] Data_in_bar[7] clk vdd gnd ms_flop
.ENDS data_in_flop_array

.SUBCKT data_out_flop_array Data_out[0] fdata[0] fdata_bar[0] Data_out[1]
    fdata[1] fdata_bar[1] Data_out[2] fdata[2] fdata_bar[2] Data_out[3] fdata
    [3] fdata_bar[3] Data_out[4] fdata[4] fdata_bar[4] Data_out[5] fdata[5]
    fdata_bar[5] Data_out[6] fdata[6] fdata_bar[6] Data_out[7] fdata[7]
    fdata_bar[7] clk_bar vdd gnd
Xflopdata_out0 Data_out[0] fdata[0] fdata_bar[0] clk_bar vdd gnd ms_flop
Xflopdata_out1 Data_out[1] fdata[1] fdata_bar[1] clk_bar vdd gnd ms_flop
Xflopdata_out2 Data_out[2] fdata[2] fdata_bar[2] clk_bar vdd gnd ms_flop
Xflopdata_out3 Data_out[3] fdata[3] fdata_bar[3] clk_bar vdd gnd ms_flop
Xflopdata_out4 Data_out[4] fdata[4] fdata_bar[4] clk_bar vdd gnd ms_flop
Xflopdata_out5 Data_out[5] fdata[5] fdata_bar[5] clk_bar vdd gnd ms_flop
Xflopdata_out6 Data_out[6] fdata[6] fdata_bar[6] clk_bar vdd gnd ms_flop
Xflopdata_out7 Data_out[7] fdata[7] fdata_bar[7] clk_bar vdd gnd ms_flop
.ENDS data_out_flop_array

.subckt Xnchain_inv D G S B
Mnmos D G S B nmos_vtg m=2 w=0.090000u l=0.050000u
.ends Xnchain_inv

.subckt Xpchain_inv D G S B
Mpmos D G S B pmos_vtg m=2 w=0.270000u l=0.050000u
.ends Xpchain_inv

.SUBCKT chain_inv A Z vdd gnd
Xnchain_inv Z A gnd gnd Xnchain_inv
Xpchain_inv Z A vdd vdd Xpchain_inv
.ENDS chain_inv

.SUBCKT delay_chain clk clk_bar dclk vdd gnd
Xinv1_chain0 clk[4] clk[5] vdd gnd chain_inv
Xinv2_chain0 clk clk_bar vdd gnd chain_inv
Xinv1_chain1 clk[5] clk[6] vdd gnd chain_inv
Xinv2_chain1 clk_bar clk[1] vdd gnd chain_inv
Xinv1_chain2 clk[6] clk[7] vdd gnd chain_inv
Xinv2_chain2 clk[1] clk[2] vdd gnd chain_inv
Xinv1_chain3 clk[7] clk[8] vdd gnd chain_inv
Xinv2_chain3 clk[2] clk[3] vdd gnd chain_inv
Xinv1_chain4 clk[8] dclk vdd gnd chain_inv
Xinv2_chain4 clk[3] clk[4] vdd gnd chain_inv
.ENDS delay_chain

.SUBCKT tristate in out en vdd gnd
minP in_bar in vdd vdd pmos_vtg w=180.000000n l=50.000000n
minN in_bar in gnd gnd nmos_vtg w=90.000000n l=50.000000n
moutP en_bar en vdd vdd pmos_vtg w=180.000000n l=50.000000n
moutN en_bar en gnd gnd nmos_vtg w=90.000000n l=50.000000n
#tristate for BL
moutP1 int1 in_bar vdd vdd pmos_vtg w=180.000000n l=50.000000n
moutP2 out en_bar int1 vdd pmos_vtg w=180.000000n l=50.000000n
moutN1 out en int2 gnd nmos_vtg w=90.000000n l=50.000000n
moutN2 int2 in_bar gnd gnd nmos_vtg w=90.000000n l=50.000000n
.ends tristate

.SUBCKT tristate_array fdata[0] fdata[1] fdata[2] fdata[3] fdata[4] fdata[5]
    fdata[6] fdata[7] DATA[0] DATA[1] DATA[2] DATA[3] DATA[4] DATA[5] DATA[6]
    DATA[7] OE vdd gnd
Xtri_out0 fdata[0] DATA[0] OE vdd gnd tristate
Xtri_out1 fdata[1] DATA[1] OE vdd gnd tristate
Xtri_out2 fdata[2] DATA[2] OE vdd gnd tristate
Xtri_out3 fdata[3] DATA[3] OE vdd gnd tristate
Xtri_out4 fdata[4] DATA[4] OE vdd gnd tristate
Xtri_out5 fdata[5] DATA[5] OE vdd gnd tristate
Xtri_out6 fdata[6] DATA[6] OE vdd gnd tristate
Xtri_out7 fdata[7] DATA[7] OE vdd gnd tristate
.ENDS tristate_array
```

```
.SUBCKT test_sram DATA[0] DATA[1] DATA[2] DATA[3] DATA[4] DATA[5] DATA[6] DATA
    [7] ADDR[0] ADDR[1] ADDR[2] ADDR[3] ADDR[4] CSb WEb OEb clk vdd gnd
Xbitcell_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4] BR[4] BL
    [5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10] BR[10] BL
    [11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR[15] WL[0]
    WL[1] WL[2] WL[3] WL[4] WL[5] WL[6] WL[7] WL[8] WL[9] WL[10] WL[11] WL[12]
     WL[13] WL[14] WL[15] vdd gnd bitcell_array
Xprecharge_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4] BR[4]
    BL[5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10] BR[10]
    BL[11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR[15] clk
    vdd precharge_array
Xcolumn_mux_array BL[0] BR[0] BL[1] BR[1] BL[2] BR[2] BL[3] BR[3] BL[4] BR[4]
    BL[5] BR[5] BL[6] BR[6] BL[7] BR[7] BL[8] BR[8] BL[9] BR[9] BL[10] BR[10]
    BL[11] BR[11] BL[12] BR[12] BL[13] BR[13] BL[14] BR[14] BL[15] BR[15]
    BL_out[0] BR_out[0] BL_out[2] BR_out[2] BL_out[4] BR_out[4] BL_out[6]
    BR_out[6] BL_out[8] BR_out[8] BL_out[10] BR_out[10] BL_out[12] BR_out[12]
    BL_out[14] BR_out[14] sel[0] sel_bar[0] gnd column_mux_array
Xsense_amp_array Data_out[0] BL_out[0] BR_out[0] Data_out[1] BL_out[2] BR_out
    [2] Data_out[2] BL_out[4] BR_out[4] Data_out[3] BL_out[6] BR_out[6]
    Data_out[4] BL_out[8] BR_out[8] Data_out[5] BL_out[10] BR_out[10] Data_out
    [6] BL_out[12] BR_out[12] Data_out[7] BL_out[14] BR_out[14] SCLK vdd gnd
    sense_amp_array
Xwrite_driver_array BL_out[0] BR_out[0] Data_in[0] BL_out[2] BR_out[2] Data_in
    [1] BL_out[4] BR_out[4] Data_in[2] BL_out[6] BR_out[6] Data_in[3] BL_out
    [8] BR_out[8] Data_in[4] BL_out[10] BR_out[10] Data_in[5] BL_out[12]
    BR_out[12] Data_in[6] BL_out[14] BR_out[14] Data_in[7] EN vdd gnd
    write_driver_array
Xaddress_decoder ain[0] ain_bar[0] ain[1] ain_bar[1] ain[2] ain_bar[2] ain[3]
    ain_bar[3] WL[0] WL[1] WL[2] WL[3] WL[4] WL[5] WL[6] WL[7] WL[8] WL[9] WL
    [10] WL[11] WL[12] WL[13] WL[14] WL[15] dclk vdd gnd address_decoder
Xcntrl CSb WEb OEb clk OE EN SCLK vdd gnd control_logic
Xarow_flop_array ADDR[0] ain[0] ain_bar[0] ADDR[1] ain[1] ain_bar[1] ADDR[2]
    ain[2] ain_bar[2] ADDR[3] ain[3] ain_bar[3] clk vdd gnd
    addr_row_flop_array
Xacol_flop_array ADDR[4] sel_bar[0] sel[0] clk vdd gnd addr_col_flop_array
Xdatain_flop_array DATA[0] Data_in[0] Data_in_bar[0] DATA[1] Data_in[1]
    Data_in_bar[1] DATA[2] Data_in[2] Data_in_bar[2] DATA[3] Data_in[3]
    Data_in_bar[3] DATA[4] Data_in[4] Data_in_bar[4] DATA[5] Data_in[5]
    Data_in_bar[5] DATA[6] Data_in[6] Data_in_bar[6] DATA[7] Data_in[7]
    Data_in_bar[7] clk vdd gnd data_in_flop_array
Xdataout_flop_array Data_out[0] fdata[0] fdata_bar[0] Data_out[1] fdata[1]
    fdata_bar[1] Data_out[2] fdata[2] fdata_bar[2] Data_out[3] fdata[3]
    fdata_bar[3] Data_out[4] fdata[4] fdata_bar[4] Data_out[5] fdata[5]
    fdata_bar[5] Data_out[6] fdata[6] fdata_bar[6] Data_out[7] fdata[7]
    fdata_bar[7] clk_bar vdd gnd data_out_flop_array
Xdchain clk clk_bar dclk vdd gnd delay_chain
Xtri fdata[0] fdata[1] fdata[2] fdata[3] fdata[4] fdata[5] fdata[6] fdata[7]
    DATA[0] DATA[1] DATA[2] DATA[3] DATA[4] DATA[5] DATA[6] DATA[7] OE vdd gnd
     tristate_array
.ENDS test_sram
```