OpenRAM Manual

Matthew R. Guthaus - mrg@ucsc.edu
James Stine - james.stine@okstate.edu
and many others

January 19, 2018

1 License

Copyright 2018 Regents of the University of California and The Board of Regents for the Oklahoma Agricultural and Mechanical College (acting for and on behalf of Oklahoma State University)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

1 License **2**

2 Introduction **5**

2.1 Requirements 6

2.2 Environment Variables 7

2.3 Design Flow 7

2.4 Usage 7

3 Overview of the SRAM Structure **9**

3.1 Inputs/Outputs 9

3.2 Top-Level SRAM Module 10

4 Modules **10**

4.1 The Bitcell and Bitcell Array 11

4.2 Precharge Circuitry 12

4.3 Address Decoders 13

4.4 Wordline Driver 17

4.5 Column Mux 17

4.6 Sense Amplifier 20

4.7 Write Driver 21

4.8 Flip-Flop Array 22

4.9 Control Logic 22

5 Bank and SRAM **23**

6 Software Implementation **24**

6.1 Design Hierarchy 25

6.2 Creating a New Design Module 28

6.3 GDSII Files and GdsMill) 30

6.4 Technology Directory 32

6.5 DRC/LVS Interface 33

7 Custom Layout Design Functions in Software **33**

7.1 Parameterized Transistor 33
7.2 Parameterized Inverter 34
7.3 Parameterized NAND2 35
7.4 Parameterized NAND3 37
7.5 Parameterized NOR2 38
7.6 Path and Wire 38

8 Porting to a new Technologies **39**
8.1 The GDS and Spice Libraries 39
8.2 Technology Directory 40

9 Timing and Control Logic **40**
9.1 Signals 40
9.2 Timing Considerations 41
9.3 SRAM Operation 41
9.4 Zero Bus Turnaround (ZBT) 43
9.5 Control Logic 44
9.6 Replica Bitline Delay 46
9.7 Timing and Power Characterizer 46

10 Unit Tests **46**
10.1 Usage 49

11 Debug Framework **50**

2 Introduction

The OpenRAM project aims to provide a free, open-source memory compiler development framework for Random-Access Memories (RAMs). Most academic Integrated Circuit (IC) design methodologies are inhibited by the availability of memories. Many standard-cell process design kits (PDKs) are available from foundries and vendors, but these PDKs do not come with memory arrays or compilers. Some PDKs have options to request "black box" memory models, but these are not modifiable, have limited available configurations, and do not have full details available to academics. These restrictions make comparison and experimentation with real memory systems impossible. OpenRAM, however, is user-modifiable and portable through technology libraries to enable experimentation with real-world memories at a variety of performance points and costs.

The specific features of OpenRAM are:

- **Memory Array Generation**

  Currently, OpenRAM supports simple 1 read/write port synchronous memories, but it will be extended to multi-port memories, register files, and asynchronous memories in the future. The generation includes features such as automatic word-line driver sizing, efficient decoder sizing, multiple-word column support, and self-timing with replica bitlines.

- **Portability and Extensibility**

  OpenRAM is a Python program. Python enables portability to numerous platforms and enables the program to be extended by anyone. In general, it works on Linux, MacOS, and Windows platforms.

  User-readable technology files enable migration to a variety of process technologies. Currently, an implementation in a non-fabricale 45nm technology (FreePDK45) is provided and the MOSIS Scalable CMOS (SCN3ME_SUBM.30) is provided. The compiler has also been extended to several technologies. We hope to work with vendors to distribute the technology information of others commercial technologies soon.

  OpenRAM makes calls to commercial circuit simulators and DRC/LVS tools in an abstracted way for circuit simulation and verification. This enables adaptation to other design methodologies. However, it also supports a completely open-source platform for older technologies.

- **Timing and Power Characterization**

  OpenRAM provides a basic framework for analysis of timing and power. This includes both analytical estimates, un-annotated spice simulations, or back-annotated simulations. The timing and power views are provided in the Liberty open format for use with the most common logic synthesis and timing analysis tools.

- **Commercial Tool Independence and Interoperability**

  To keep OpenRAM portable and maximize its usefulness, it it independent from any specific commercial tool suite or language. OpenRAM interfaces to both open-source (e.g., NGSpice) and commercial circuit simulators through the standard Spice3 circuit format. The physical layout is directly generated in the GDSII layout stream format which can be imported into any academic or commercial layout tools. We provide a Library Exchange Format (LEF) file for interfacing with commercial Placement and Routing tools. We provide a Verilog behavioral model for simulation.

- **Silicon Verification** TBD

2.1 Requirements

Development is done on Ubuntu or MacOS systems with Python 2.7.

2.1.1 Timing Verification Tools

For peformance reasons, OpenRAM uses analytical delay models by default. If you wish to enable simulation-based timing characterization, you must enable this on the command line with the "-c" command line argument.

OpenRAM can use the following circuit simulators and possibly others if they support the Spice3 file format:

- HSpice I-2013.12-1 or later
- ngSpice 26 http://ngspice.sourceforge.net/
- CustomSim (xa) M-2017.03-SP5 or later

2.1.2 Physical Verification Tools

By default, OpenRAM will perform DRC and LVS on each level of hierarchy. To do this, you must have a valid DRC and LVS tool and the corresponding rule files for the technology. OpenRAM can, however, run without DRC and LVS verification using the "-n" command line argument. It is not recommended to use this if you make any changes, however.

DRC can be done with:

- Calibre 2012.3_15.13 or later (SCMOS or FreePDK45)
- Magic http://opencircuitdesign.com/magic/ (SCMOS only)

LVS can be done with:

- Calibre 2012.3_15.13 or later (SCMOS or FreePDK45)
- Netgen http://opencircuitdesign.com/netgen/ (SCMOS only)

2.1.3 Technology Files

To work with FreePDK45, you must install the FreePDK baseline kit from: https://www.eda.ncsu.edu/wiki/FreePDK45:Contents

We have included an example Calibre DRC deck for MOSIS SCMOS design rules, but DRC with Magic relies on its own design rules: https://www.mosis.com/files/scmos/scmos.pdf We require the format 32 or later to enable stacked vias which is included with Qflow:

```
git clone http://opencircuitdesign.com/qflow
cp tech/osu050/SCN3ME_SUBM.30.tech <your magic tech lib>
```

You can over-ride the location of the DRC and LVS rules with the DRCLVS_HOME environment variable.

2.1.4 Spice Models

FreePDK45 comes with a spice device model. Once this is installed, it is used.

SCMOS, however, does not come with a device spice model. This must be obtained from MOSIS or another vendor. We use the ON Semiconductor 0.5um device models.

You can over-ride the location of the spice models with the SPICE_MODEL_DIR environment variable.

2.2 Environment Variables

In order to make OpenRAM flexible, it uses two environment variables to make it relocatable in a variety of user scenarios. Specifically, the user may want technology directories that are separate from OpenRAM. Or, the user may want to have several versions of OpenRAM. This is done with the folowing required environment variables: specifically:

- OPENRAM_HOME defines the location of the compiler source directory.
- OPENRAM_TECH defines the location of the OpenRAM technology files. This is discussed later in Section 8.2.

Other environmental variables and additional required paths for specific technologies are dynamically added during runtime by sourcing a technology setup script. These are located in the "$OPENRAM_TECH/setup_scripts" directory. Example scripts for SCMOS and FreePDK45 are included with the distribution. These setup any things needed by the PDK.

2.3 Design Flow

2.4 Usage

The OpenRAM compiler rquires a single argument of a configuration file. The configuration file specifies, at a minimum, the memory size parameters in terms of the number of words, word size (in bits), and number of banks. By default, OpenRAM will chose the number of columns to make the memory reasonably square. Commonly, the configuration file also includes parameters for the output path, base output file name, and technology of an SRAM.

The configuration file can be used to over-ride any option in the options.py file. Many of these are controlled by the command-line, but the configuration file takes priority and allows repeatable generation of memories.

Lastly, the configuration file can over-ride any of the different circuit implementations for each module. For example, you can replace the default address decoder or bitcell with a new one by specifying a new python module that implements a new one.

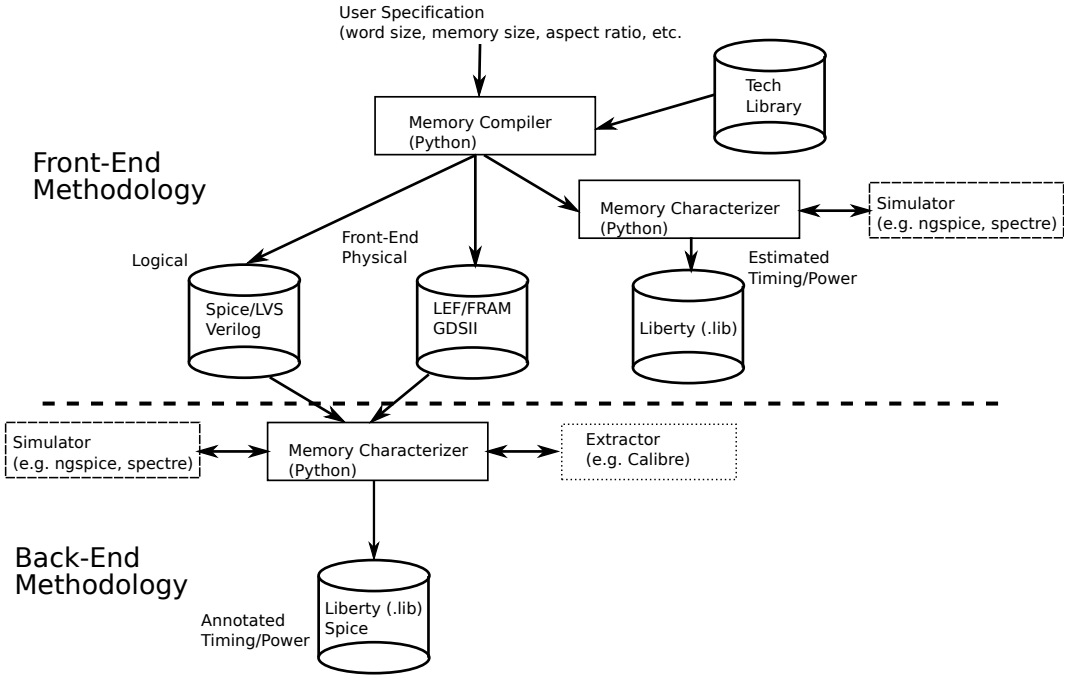

Figure 1: Overall Compilation and Characterization Methodology

An entire example configuration file looks like:

```
word_size = 16
num_words = 32
num_banks = 1

tech_name = "freepdk45"

output_path = "/tmp/outputdir"
output_name = "mysram"

bitcell = "custom_bitcell"
```

In this example, the user has specified a custom bitcell that will be used when creating the bitcell_array and other modules.

OpenRAM has many command line arguments. Other useful command line arguments are:

- -h : To get help for the command-line options
- -v : To increase the verbosity (may be used multiple times)

3 Overview of the SRAM Structure

The baseline SRAMs generated by OpenRAM have 1 read/write port as shown in Figure 2. The address is decoded (Section 4.3) into a one-hot set of word lines (WL) which are driven by word line drivers (Section 4.4) over the bit-cell array (Section 4.1). To facilitate reads, the precharge circuitry (Section 4.2) precharges the bitlines so that the column mux (Section **??**) can select the appropriate word which is then sensed by the sense amplifiers (Section 4.6). Write drivers (Section 4.7) use the bidirectional nature of the column mux to write the appropriate columns in a given memory row.

A representative layout of such a memory closely resembles the logical representation and is shown in Figure 3. The address and data flip-flops and control circuitry are not shown but are detailed in Section 9.5.

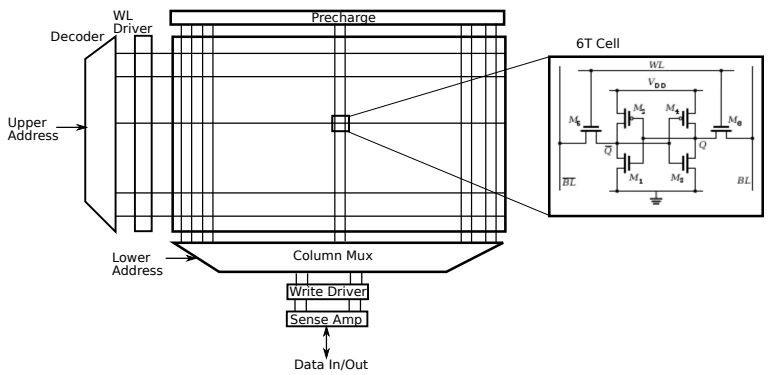

Figure 2: Single Port SRAM Architecture

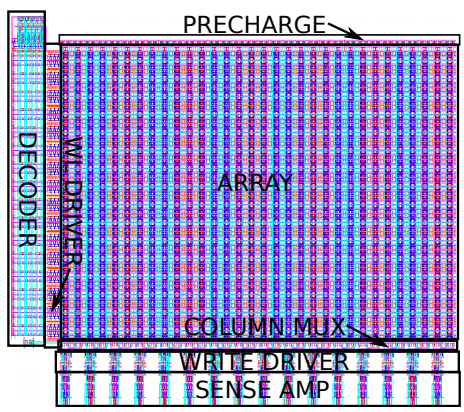

Figure 3: 1k SRAM with Two Columns and 16-bit Data

3.1 Inputs/Outputs

The inputs to the SRAM are:

- clk - External Clock
- CSb - Active-low Chip Select

- WEb - Active-low Write Enable
- OEb - Active-low Output Enable
- ADDR# - corresponds to the Address Bus input, labeled 0 to N-address bits.
- DATA# - corresponds to the bi-directional Data bus.

The outputs to the SRAM are:

- DATA# - correspond to the bi-directional Data bus.

3.2 Top-Level SRAM Module

The `sram` class in `sram.py` is the top-level SRAM module. This class handles the overall organization of the memory and the input/output signals. Based on the user inputs, the various bus and array sizes are calculated and passed to the `bank` module. All other sub-modules access the value of sizes from `bank`. The overall organization is depicted in Figure 2, discussion of the design data structure is discussed in Section 6.1 and the modules contained in the top-level SRAM are detailed in Section 4.

When the user has specified the desired size (word size, total number of words and number of banks) of the memory that is to be generated, the following parameters must be calculated. There are several constraints to be considered in this calculations:

(i) `sram` can generate 1 bank, 2 banks or 4 banks.

(ii) The area of each bank should be as square as possible which is dependent on the area of a 6T cell.

(iii) There are several options for multiplexing (column-mux): 2-way, 4-way, 8-way and none.

All of the top level routing is performed in the `sram` class.

FIXME: **More soon...**

4 Modules

This section provides an overview of the main modules that are used in an SRAM. For each module, we will provide both an architectural description and an explanation of how that design is generated and used in OpenRAM. The modules described below are provided in the first release of OpenRAM, but by no means is this an exhaustive list of the possible circuits that can be adapted into a SRAM architecture; refer to Section 6 for more information on adding different module designs to the compiler.

Each module has a corresponding python class in the `compiler` directory. These classes are used to generate both the GDSII layout and spice netlists. Each module can consist of library cells as discussed in Section 6.4, paramterized cells in Section 7 or other modules. A discussion of the design hierarchy and how to implement a module is provided in Section 6.1.

When combining modules at any level of hierarchy, DRC rules for minimum spacing of metals, wells, etc. must be followed and DRC and LVS are run by default after each hierarchical module's creation.

4.1 The Bitcell and Bitcell Array

The 6T cell is the most commonly used memory cell in SRAM devices. It is named a 6T cell because it consist of 6 transistors: 2 access transistors and 2 cross coupled inverters as shown in Figure 4. The cross coupled inverters hold a single data bit that can either be driven into, or read from the cell by the bitlines. The access transistors are used to isolate the cell from the bitlines so that data is not corrupted while a cell is not being accessed.

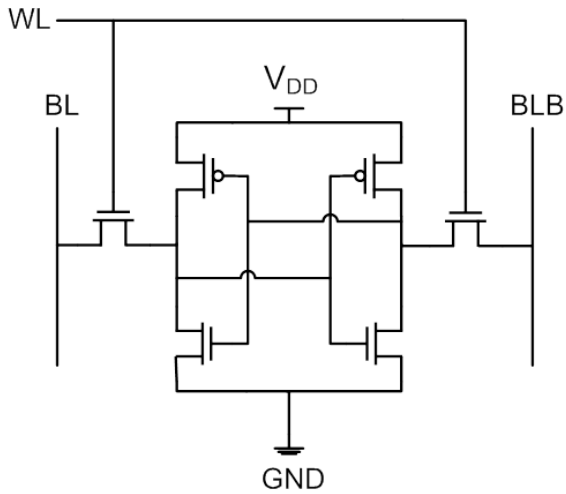

Figure 4: Schematic of 6T cell.

The 6T cell can be accessed to perform the two main operation associated with memory: reading and writing. When a read is to be performed, both bitlines are precharged to VDD. This precharging is done during the first half of the read cycle and is handled by the precharge circuitry. In the second half of the read cycle the wordline is asserted, which enable the access transistors. If a 1 is stored in the cell then BLB is discharged to Gnd and BL is pulled up to Vdd. Conversely, if the value stored is a 0, then BL is discharged to Gnd and BLB is pulled up to Vdd. While performing a write operation, both bitlines 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 BLB. The drivers that are applying the signals to the bitlines must be appropriately sized so that the previous value in the cell can be overwritten.

The 6T cells are tiled together in both the horizontal and vertical directions to make up the memory array. The size of the memory array is directly related to the numbers of words, and the size of those words, that will need to be stored in the RAM. For example, an 8kb memory with a word size of 8 bits could be implemented as 8 columns and 1024 rows.

It is common practice to keep the aspect ratio of memory array as square as possible[1]. This helps to make sure that the bitlines do not become too long, which can increase the bitline capacitance, slow down the operation and lead to more leakage. To make the design "more square", multiple words can share rows by interleaving the bits of each word. If the previous 8kb memory was rearranged to allow 2 words per row, then the array would have 16 columns and 512 rows.

In OpenRAM, we provide a library cell for the 6T cell so that users can easily swap in different

[1]Future versions will consider optimizing delay and/or power as well.

memory cell designs. The memory cell is the most important cell in the RAM and should be customized to minimize area and optimize performance. The memory cell is the most replicated cell in the RAM; minimizing its size can have a drastic effext on the overall size of the RAM. Also, the transitors in the cell must be carefully sized to allow for correct read and write operation as well as protection against corruption.

The `bitcell` class in `bitcell.py` instantiates a single memory cell and is usually a pre-made library cell. The `bitcell_array` class in `bitcell_array.py` dynamically implements the memory cell array by instantiating a single memory cell according to the number of rows and columns. During the tiling process, the cells are abutted so that all bitlines and word lines are connected in the vertical and horizontal directions respectively. In order to share supply rails, cells are flipped in alternating rows. To avoid any extra routing, the power/ground rails, bitlines, and wordlines should span the entire width/height of the cell so thay they are automatically connected when the cells are abutted.

4.2 Precharge Circuitry

The precharge circuit is depicted in Figure 5 and is implemented by three PMOS transistors. The input signal to the cell, clk, enables all three transistors during the first half of a read or write cycle (i.e. while the clock signal is low). M1 and M2 charge BL and BLB to Vdd and M3 helps to equalize the voltages seen on BL and BLB.

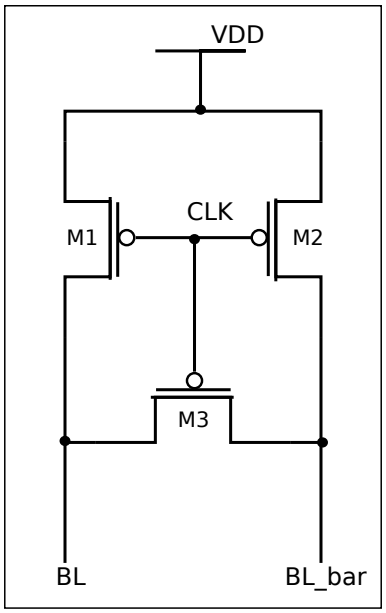

Figure 5: Schematic of a single precharge cell. FIXME: **Change PCLK to CLK.**

In OpenRAM, the precharge citcuitry is dynamically generated using the parameterized transistor class (`ptx`). The `precharge` class in `precharge.py` dynamically generates a single precharge cell.

The offsets of the bitlines and the width of the precharge cell are equal to the 6T cell so that the bitlines 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.

4.3 Address Decoders

The address decoder takes the row address bits from the address bus as inputs, and asserts the appropriate wordline in the row that data is to be read or written. A n-bit address input controls 2^n word lines.

OpenRAM provides a hierarchical address decoder as the default, but will soon have other options.

4.3.1 Hierarchical Decoder

Hierarchical decoder is a type of decoder which the constrcution takes place hierarchically. The simple 2:4 decoder is shown in the Figure 7. The operation of this decoder can be explained as follows: soon after the address signals A0 and A1 are put on the address lines, depending on the signal combination, one of the wordlines will rise after a brief amount of time. For example if the address input is A0A1=00 then the output is W0W1W2W3=1000. The 2:4 address decoder uses inverters and two input nand gates for its constrcution while the gates are sized to have equal rise and fall time. As the decoder size increases the size of the nand gates required for decoding also increases. Table 1 gives the detailed input and output siganls for the 2:4 hierarchical decoder.

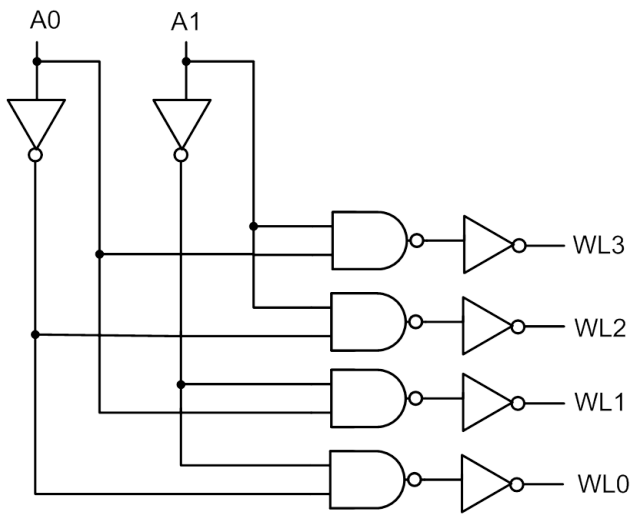

Figure 6: Schematic of 2-4 simple decoder.

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

Table 1: Truth table for 2:4 hierarchical decoder.

An n-bit decoder requires 2^n logic gates, each with n inputs. For example, with n = 6, 64 $NAND6$ gates are needed to drive 64 inverters to implement the decoder. It is clear that gates with more than 3 inputs create large series resistances and long delays. Rather than using n-input gates, it is preferable to

use a cascade of gates. Typically two stages are used: a predecode stage and a final decode stage. The predecode stage generates intermediate signals that are used by multiple gates in the final decode stage.

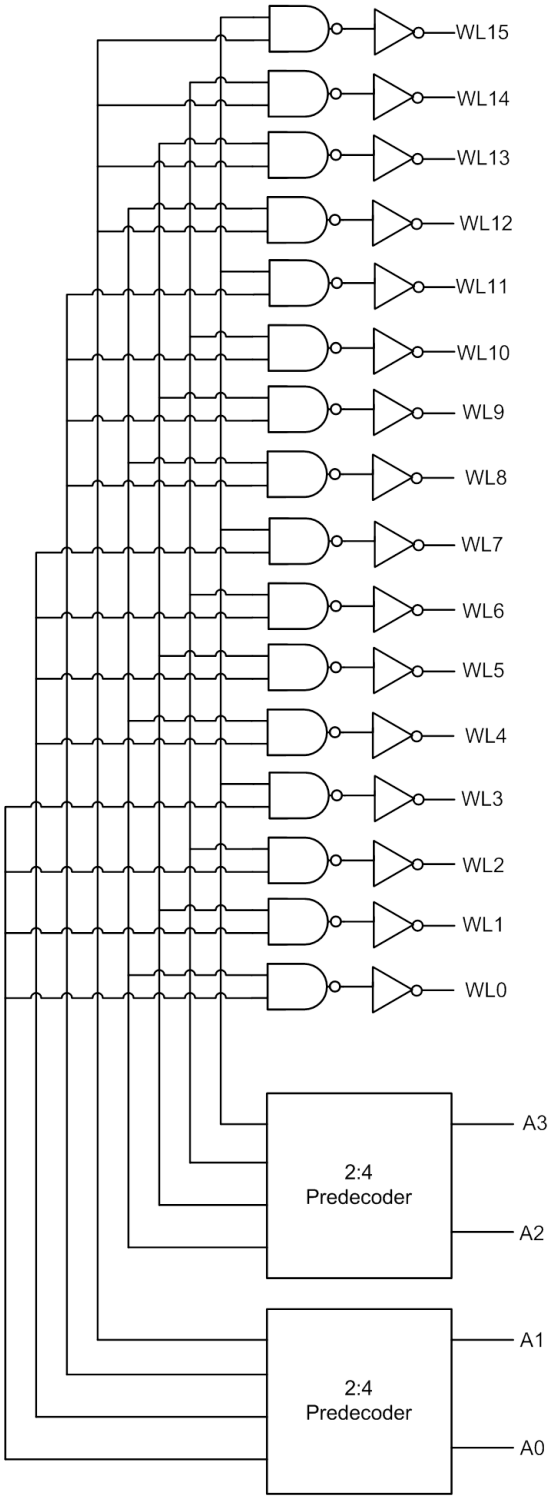

Figure 7: Schematic of 4 to 16 hierarchical decoder.

Figure 8 shows the 4 to 16 heirarchical decoder. The structure of the decoder consists of two

2:4 decoders for predecoding and 2-input nand gates and inverters for final decoding to form the 4:16 decoder. In the predecoder, a total of 8 intermediate signals are generated from the address bits and their complements. The concept of using predecoing and final decoding stage for construction of address decoder is very procutive since small decoders like 2:4 decoder is used for predecoding. The operation of 4:16 heirarchical decoder can explained with an example. If the address is A0A1A2A3=0000 the output of the predecoder1 and predeocder2 will be WL0WL1WL2WL3=1000 and WL0WL1WL2WL3=1000, respectively. According to the connections in figure 8 the wordline 0 of predecoder1 and predecoder2 are conneted to the first 2-input nand gate in the decode stage representing the wordline 0 of the final decoding stage. Hence depengin on the combination of the input signal one of the wordline will rise. In this case since the address input is A0A1A2A3=0000 the wordline 0 should go high. Table 2 gives the detailed input and output siganls for the 4:16 hierarchical decoder.

A[3:0]	predecoder1	predecoder2	Selected WL
0000	1000	1000	0
0001	1000	0100	1
0010	1000	0010	2
0011	1000	0001	3
0100	0100	1000	4
0101	0100	0100	5
0110	0100	0010	6
0111	0100	0001	7
1000	0010	1000	8
1001	0010	0100	9
1010	0010	0010	10
1011	0010	0001	11
1100	0001	1000	12
1101	0001	0100	13
1110	0001	0010	14
1111	0001	0001	15

Table 2: Truth table for 4:16 hierarchical decoder.

As the size of the address line increases higher level decoder can be created using the lower level decoders. For example for a 8:256 decoder, two instances of 4:16 followed by 256 2-input nand gates and inverters can form the decoder. In order to construct the 8:256 decoder, first 4:16 decoder should be constructed through using 2:4 deccoders. Hence the name is hierarchical decoder.

4.4 Wordline Driver

Word line drivers are inserted, in between the word line output of the address decoder and the word line input of the bitcell-array. The word line drivers ensure that as the size of the memory array increases, and the word line length and capacitance increases, the word line signal is able to turn on the access transistors in the 6T cell. Also, as the bank select signal in multi-bank structures is $ANDED$ with the word line output of decoder, bitcells turn on only when bank is selected. Figure 9 shows the diagram of word line driver and its input/output pins. In OpenRAM, word line drivers are created by using the `pinv` and `nand2` classes which takes the transistor size and cell height as inputs (so that it can abutt the 6T cell). Word line driver is added as seperate module in `compiler`.

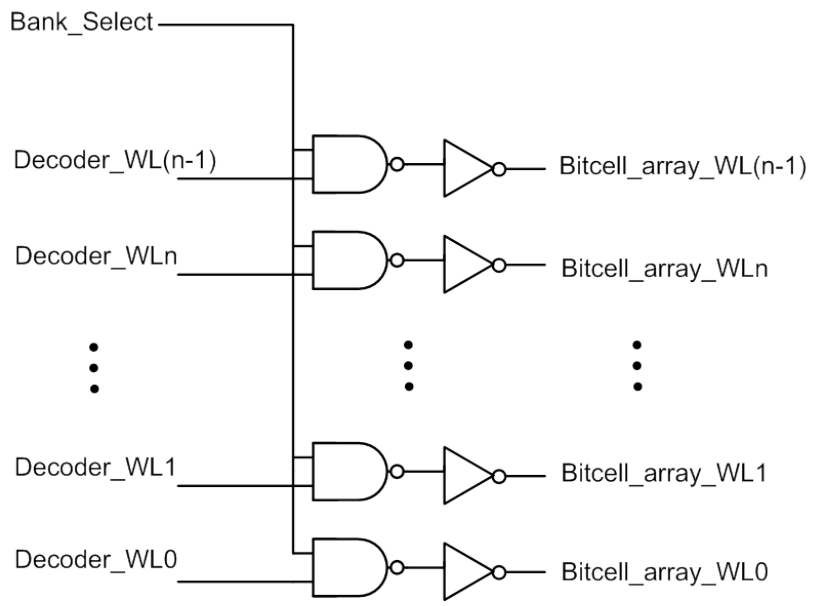

Figure 8: Diagram of word line driver.

4.5 Column Mux

The column mux takes the column address bits from the address bus selects the appropriate bitlines for the word that is to be read from or written to. It takes n-bits from the address bus and can select 2^n bitlines. The column mux is used for both the read and write operations; it connects the bitline of the memory array to both the sense ampflifier and the write driver.

OpenRAM provides several options for column mux, but the default is a single-level column mux which is sized for optimal speed.

4.5.1 Tree_Decoding Column Mux

The schematic for a 4-1 tree multiplexer is shown in Figure 10.

FIXME: **Shading/opacity is different on different platforms. Make this a box in the image. It doesn't work on OSX.**

This tree mux selects pairs of bitlines (both BL and BL_B) as inputs and outputs. This 4-1 tree mux illustrates the process of choosing the correct bitlines if there are 4 words per row in the memory array. Each bitline pair represents a single bit from each word. A binary reduction pattern, shown in Table 3, is used to select the appropriate bitlines. As 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. The 4-1 tree mux has a depth of 2. In level 1, the least significant bit from the column address bus selects either the first and second words or the third and fourth words. In level 2, the most signifant column address bit selects one of the words passed down from the previous level. Relative to other column mux designs, the tree mus uses significantly less devices. But, this type of design can provide poor performance if a large decoder with many levels are needed. The delay of

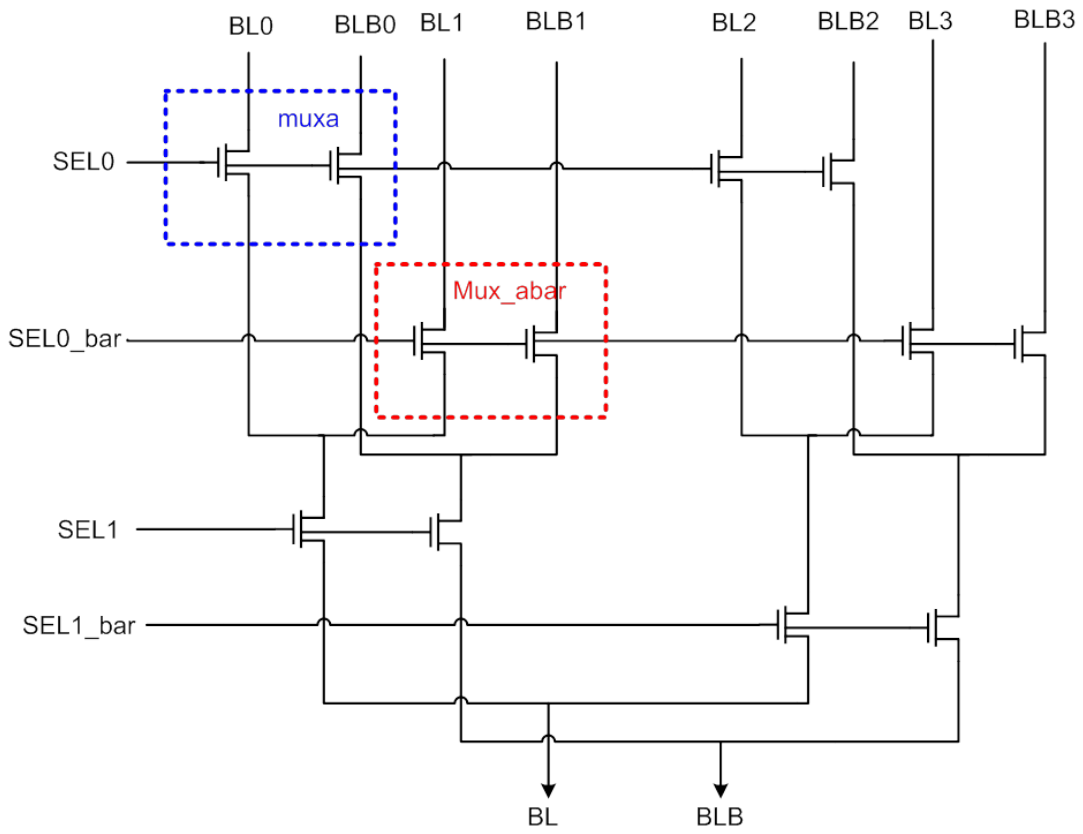

Figure 9: Schematic of 4-1 tree column mux that passes both of the bitlines.

of a tree mux quadratically increases with each level. Due to this fact, other types of column decoders should be considered for larger arrays.

Selected BL	Inp1	Inp2	Binary
BL0	SEL0_bar	SEL1_bar	00
BL1	SEL0	SEL1_bar	01
BL2	SEL0_bar	SEL1	10
BL3	SEL0	SEL1	11

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

In OpenRAM, the tree column mux is a dynamically generated design. The `tree_mux_array` is made up of two dynamically generated cells: `muxa` and `mux_abar`. The only diffference between these cells is that input select signal is either hooked up to the **SEL** or **SEL_bar** signals (see highlighted boxes in Figure 10). These cells are initialized the the `column_muxa` and `column_muxabar` classes in `columm_mux.py`. Instances of `ptx` PMOS transistors are added to the design and the necessary routing is 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_B outputs from `muxa` to the BL and BL_B outputs of `mux_abar`.

Vertical conenctors 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 is more than one level.

The `tree_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 is the column number. As the depth increases, the mux cells become further apart. A separate "for" loop is invoked 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 made at the various levels. 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.

4.5.2 Single_Level Column Mux

The optimal design for column mux uses a single NMOS device, driven by the input address or decoded input addresses. Figure 11 shows the schematic of a 2:1 single-level column mux. In this column mux one bit of address and its complementry drive the pass transistors. Selected transistors will connect their corresponding bitlines (1 set of column out of 2 set of columns) to sense-amp and write-driver circuitry for read or write operation. Figure 12 shows the schematic of a 4:1 single-level column mux. In this column mux, 2 input address are decoded using a 2:4 decoder (2:4 decoder is explain in section 4.3.2). 2:4 decoder provides a one-hot set of outputs, so only one set of columns will be selected and connected to sense-amp and write-driver (in figure 12 one set of column out of four sets of column is selected).

In OpenRAM, the `single-level_mux_array` is a dynamically generated design and it is made up of dynamically generated cell (`single-level_mux`). `single-level_mux` uses the parameterized transistor class `ptx` to generate two NMOS transistors which will connect the BL and BLB of selected columns to sense-amp and write-driver. Horizontal rails are added for *sel* signals. Vertical straps connect the BL and BLB of bitcell_array to BL and BLB of single-level column mux and also BL-out and BLB-out of single-level column mux to BL and BLB of sense-amp and write-driver.

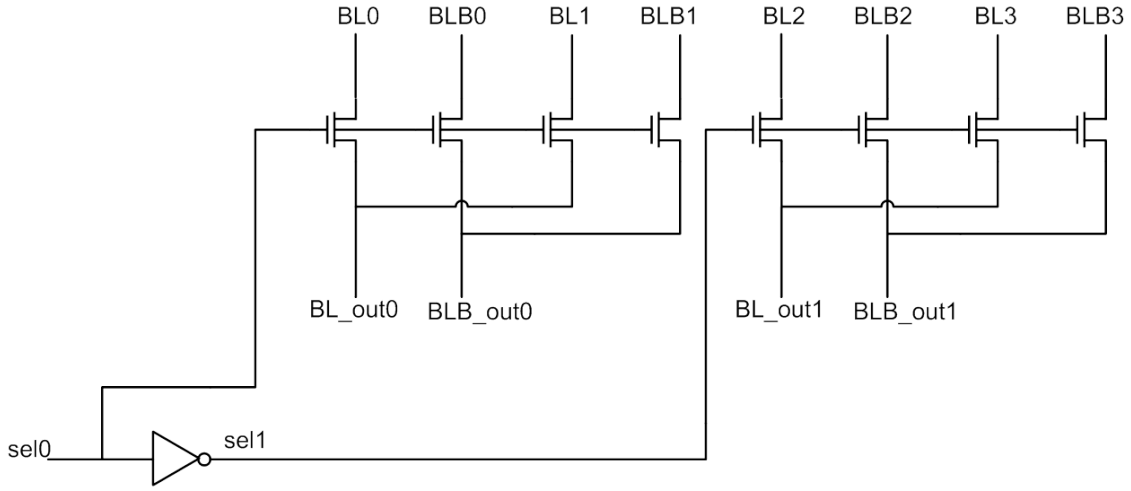

Figure 10: Schematic of a 2:1 single level column mux.

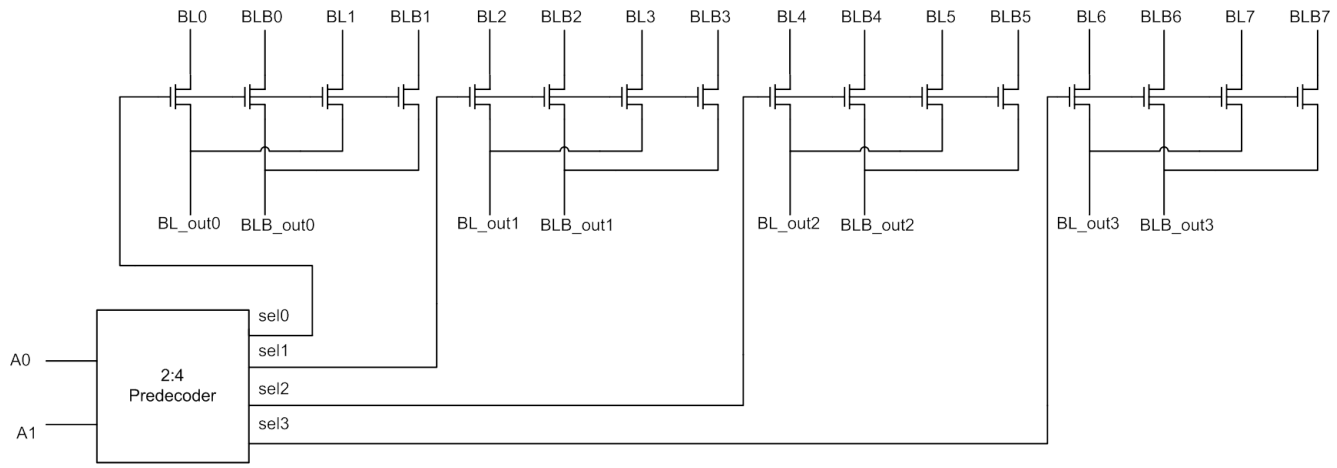

Figure 11: Schematic of a 4:1 single level column mux.

4.6 Sense Amplifier

The sense amplifier is used to sense the difference between the bitline and bitline bar while a read operation is performed. The sense amp is necessary to recover the signals from the bitlines because they do not experience full voltage swing. As the size of the memory array grows, the load of the bitlines increases and the voltage swing is limited by the small memory cell driving this large load. A differential sense amplifier is used to“sense” the small voltage difference between the bitlines.

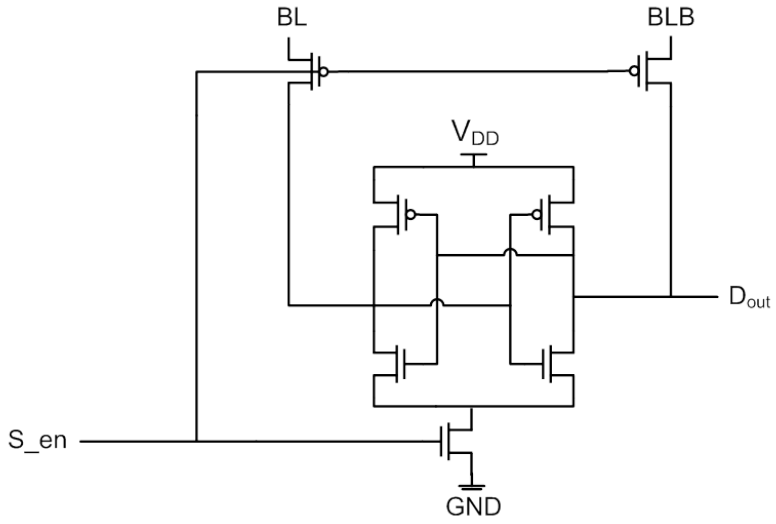

Figure 12: Schematic of a single sense amplifier cell.

The schematic for the sense amp is shown in Figure 13. The sense amplifier is enable by the SCLK signal, which initiates the read operation. Before the sense amplifier is enable, the bitlines are precharged to Vdd by the precharge unit. When the sense amp is enabled, one of the bitlines experiences a voltage drop based on the value stored in the memory cell. If a zero is stored, the bitline voltage drops. If a one is stored, the bitline bar voltage drops. The output signal is then taken to a true logic level and latched for output to the data bus.

In OpenRAM, the sense amplifier is a libray cell. The associated layout and spice netlist can be found in the `gds_lib` and `sp_lib` in the FreePDK45 directory. The `sense_amp` class in `sense_amp.py` 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 sense amp cells need to be appropriately spaced so that they can hook up to the column mux bitline 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.

We chose to leave the sense amp as a libray cell so that custom amplifier designs could be swapped into the memory as needed. The two major things that need to be considered while designing the sense amplifier cell are the size of the cell and the bitline/input pitches. Optimally, the cell should be no larger than the 6T cell so that it abuts to the column mux and no extra routing or space is needed. Also, the bitline 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 had bitlines spaning the entire height of the cell. In this case, the sense amplifier is disabled during a write operation but the bitlines still connect the write driver to the column mux without any extra routing.

4.7 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 14 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 bitline, and its complement on bitline bar. The bitlines need to be complements so that the data value can be correctly stored in the 6T cell. Both tristates are enabled by the EN signal.

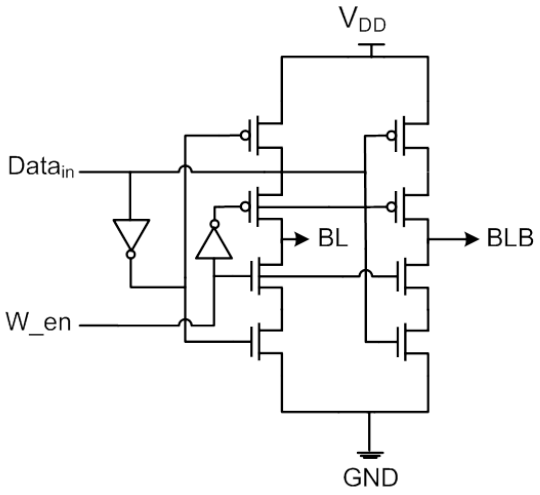

Figure 13: Schematic of a write driver cell, which consists of 2 tristates (non-inverting and inverting) to drive the bitlines.

Currently, in OpenRAM, the write driver is a library cell. The associated layout and spice netlist can be found in the `gds_lib` and `sp_lib` in the FreePDK45 directory. Similar to the `sense_amp_array`, 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. 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 bitlines. 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.

4.8 Flip-Flop Array

In a synchronous SRAM it is necessary to synchronize the inputs and outputs with a clock signal by using flip-flops. In FreePDK45 we provide a library cell for a simple master-slave flip-flop, see schematic in Figure 15. 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 in `ms_flop.py` instatitates 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` takes the number of flops and the type of array as inputs. Currently, the type of the array must be either "data_in", "data_out", "addr_row", or "addr_col" verbatim. The array type input is used to look up that associated pin names for each of the flop arrays. This was implemented very quickly and should be improved in the near future...

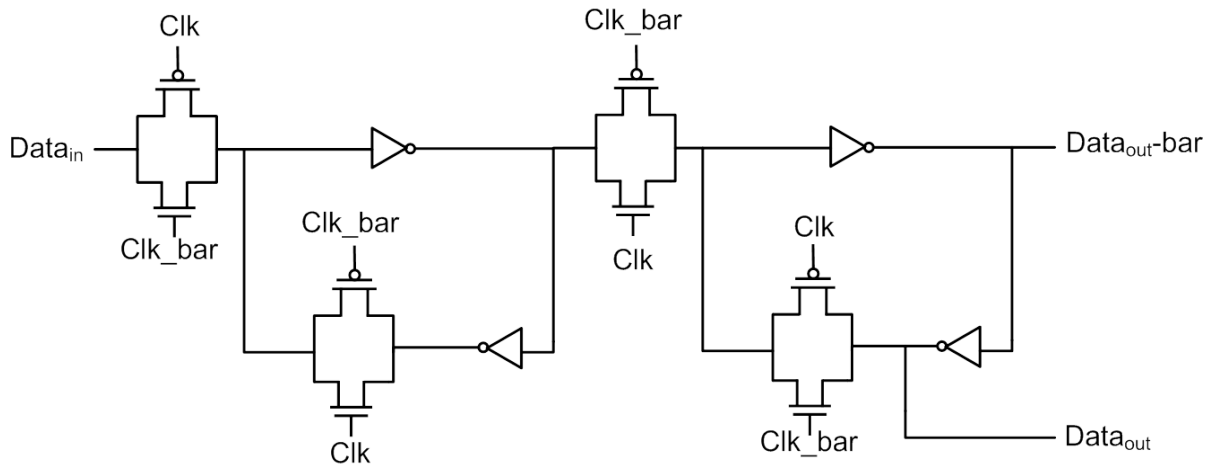

Figure 14: Schematic of a master-slave flip-flop provided in FreePDK45 library

4.9 Control Logic

The details of the control logic architecture are outlined in Section 9.5. The control logic module, `control_logic.py`, instantiates a `control_logic` class that arranges all of the flip-flops and logic associated with the control signals into a single module. Flip-flops are instantiated for each control signal input and library NAND and NOR gates are used for the logic. A delay chain, of variable length, is also generted using parameterized inverters. The associated layouts and spice netlists can be found in the `gds_lib` and `sp_lib` in the FreePDK45 directory.

5 Bank and SRAM

The overall memory architecture is shown in figure 16. As shown in this figure one Bank contains different modules including precharge-array which is positioned above the bitcell-array, column-mux-array which is located below the bitcell-array, sense-amp-array, write-driver-array, data-in-ms-flop-array to synchronize the input data with negative edge of the clock, tri-gata-array to share the bidirectional data-bus between input and output data, hierarchical decoder which is placed on the right side of the bitcell-array (predecoder + decoder), wordline-driver which drives the wordlines horizontally across the bitcell-array and address-ms-flops to synchronize the input address with positive edge of the clock.

In bitcell-array each memory cell is mirrored vertically and horizontally inorder to share VDD and GND rails with adjacent cells and form the array. Data-bus is connected to tri-gate, address-bus is connected to address-ms-flops and bank-select signal will enable the bank when it goes high. To complete the SRAM design, bank is connected to control-logic as shown in figure 16. Control-logic controls the timing of modules inside the bank. CSb, OEb, Web and clk are inputs to the control logic and output of control logic will ANDed with bank-select signal and send to the corresponding modules.

In order to reduce the delay and power, divided wordline strategy have been used in this compiler. Part of the address bits are used to define the global wordline (bank-select) and rest of address bits are connected to hierarchical decoder inside each bank to generate local wordlines that actually drive the bitcell access transistors.

As shown in figure 17 SRAM is divided to two banks which share data-bus, address-bus, control-bus and control-logic. In this case one bit of address (most significant bit) goes to an ms-flop and outputs of ms-flop (address-out and address-out-bar) are connected to banks as bank-select signals. Control logic is shared between two banks and based on which bank is selected, control signals will activate modules inside the selected bank. In this architecture, the total cell capacitance is reduced by up to a factor of two. Therefore the power will be reduced greatly and the delay among the wordlines is also reduced.

In figure 18, four banks are connected together. In this case a 2:4 decoder is added to select one of the banks using two most significant bits of input address. Control signals are connected to all banks but will turn on only the selected bank.

6 Software Implementation

OpenRAM is implemented using object-oriented data structures in the Python programming language. The top-level executable is `openram.py` which parses input arguments, creates the memory and saves the output.

6.1 Design Hierarchy

All modules in OpenRAM are derived from the `design` class in `design.py`. 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. The only additional function in design.py is `DRC_LVS()`, which performs a DRC/LVS check on the module.

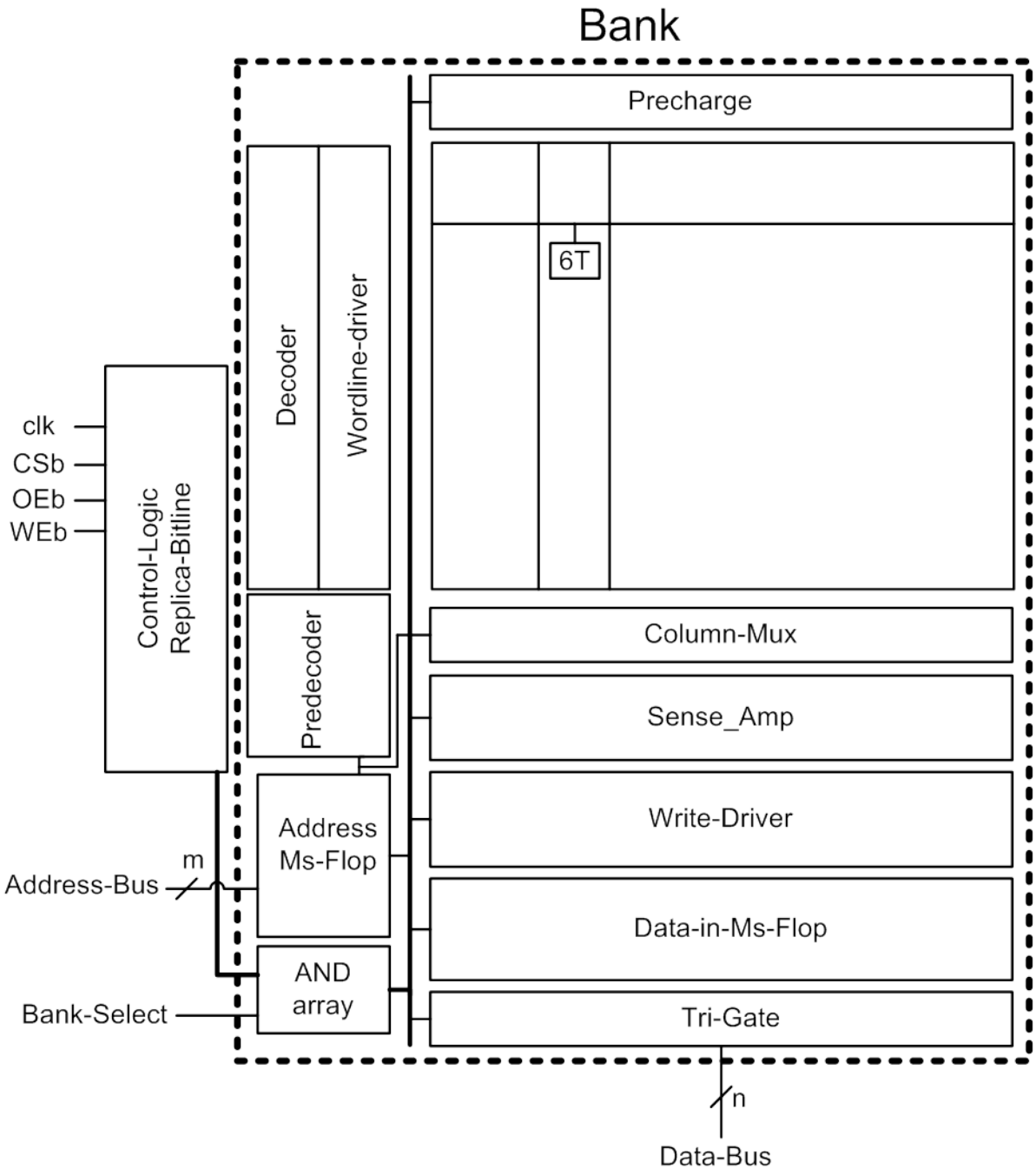

Figure 15: Overal bank and SRAM architecture.

6.1.1 Spice Hierarchy

The spice hierarchy is stored in the `spice` class in `hierarchy_spice.py`. When the design class is initialized for a module, a data structure for the spice hierarchy is created. The spice data stucture name becomes the name of the top-level subcircuit definition for the module. The list of pins for the module are added to the subcircuit definition by using the `add_pin()` function. The `add_mod()` function

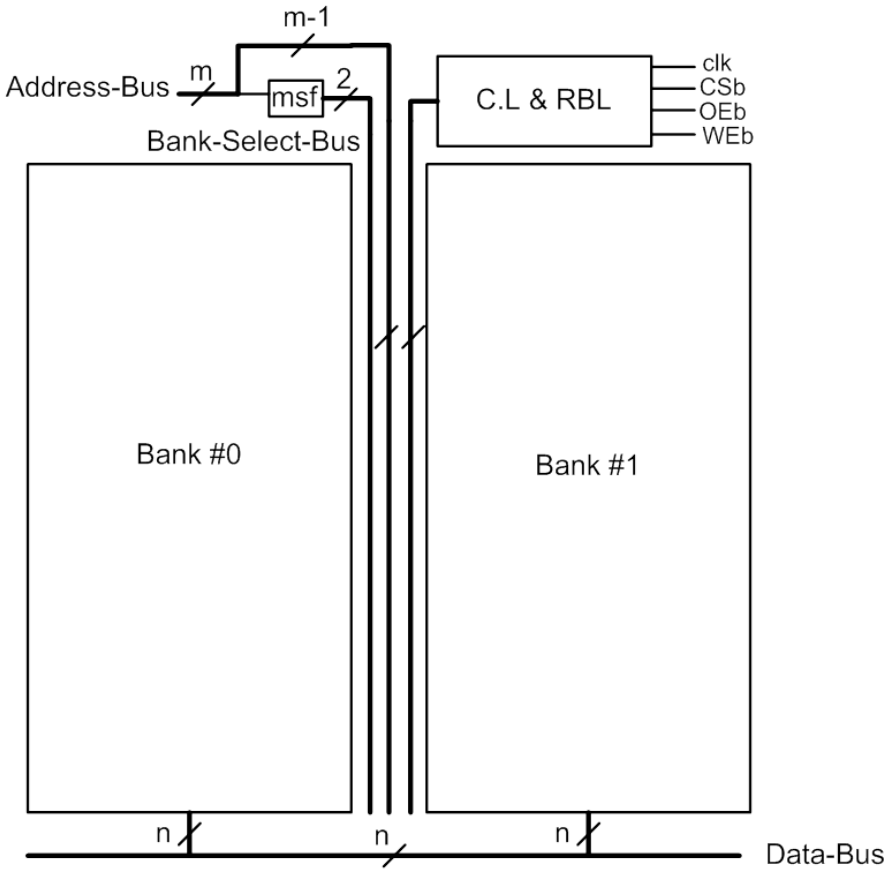

Figure 16: SRAM is divided to two banks which share the control-logic.

adds an instance of a module/library_cell/parameterized_cell as a subcircuit 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. It is important to note that the pins must be listed in the same order as they were added to the submodule. Also, an assertion error will occur if there is a mismatch in the number of net connections. The `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()`: this function creates an empty spice file in write mode and calls `sp_write_file()`.
- `sp_write_file()`: this function recursively writes the modules and sub-modules from the data structure into the spice file created by `sp_write()`.

6.1.2 Layout Hierarchy

The layout hierarchy is stroed in the `layout` class in `hierarchy_layout.py`. When the design class is initialized for a module, a data structure for the layout hierarchy is created. The layout data

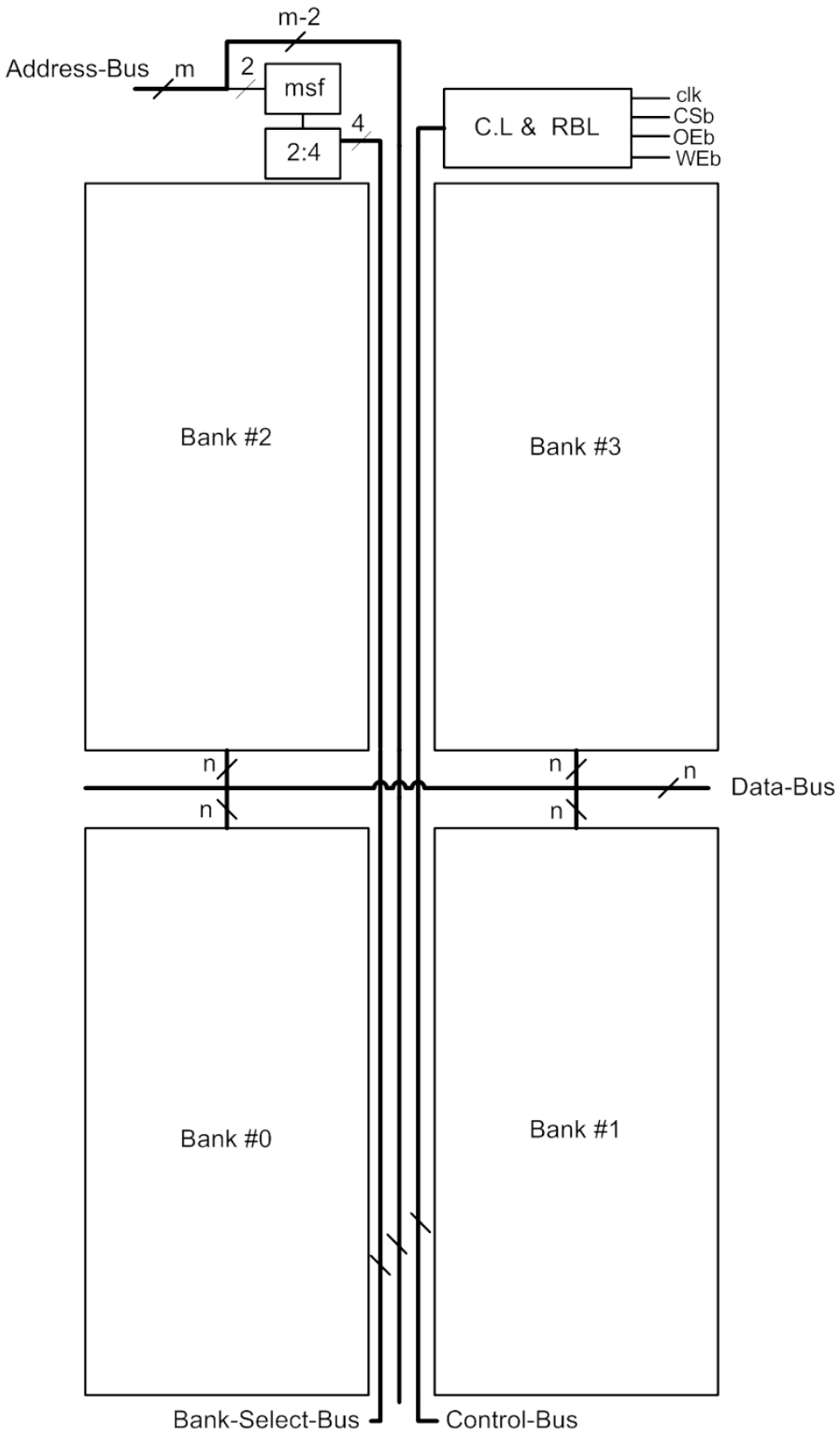

Figure 17: SRAM is divided to 4 banks wich are controlled by the control-logic and a 2:4 decoder.

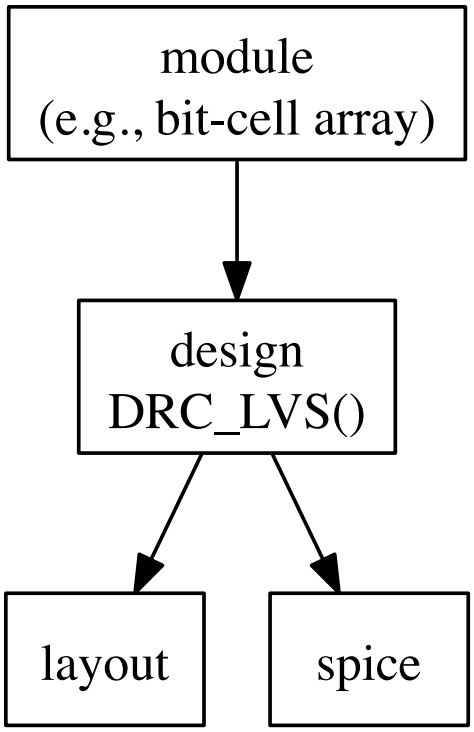

Figure 18: Class hierarchy

structure 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. The functions included in the `layout` class are:

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

  name - name for 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"` *Currently, only "R0" works.
  `"MX" or "x", "MY" or "y", "XY" or "xy"` ("xy" is equivalent to "R180")

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

- `add_path(self,layerNumber,coordinates,width)` : this function is under construction...

- `gds_read()` : reads in a GDSII file and creates a `VlsiLayout()` class for it.

- `gds_write()` : writes the entire GDS of the object to a file by gdsMill `vlsiLayout()` class and calling the `gds2writer()` (see Sections 6.3.2 and 6.3.2.

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

- `pdf_write()` : this function is under construction...

6.2 Creating a New Design Module

Each module in the SRAM is its own Python class, which contains a design class, or data structure, for the layout and spice. The `design` class (`design.py`) is initialized within the module class, subsequently creating separate data structurse to hold the layout (`hierarchy_layout`) and spice (`hierarchy_spice`) information. By having a class for each module, it is very easy to instatiate instances of the modules in any level of the hierarchy. Follow these guidelines when creating a new module:

- Derive your class from the design module:

  ```
  class bitcell_array(design.design):
  ```

- Always use the python constructor `__init__` method so that your class is initialized when an object of the module is instatiated. The module parameters should also be declared:

  ```
  def __init__(self, cols, rows):
  ```

- In the constructor, call the base class constructor with the name such as:

  ```
  design.design.__init__(self,"bitcell_array")
  ```

- Add the pins that will be used in the spice netlist for your module using the `add_pin()` function from the `hierarchy_spice` class.

  ```
  self.add_pin("vdd")
  ```

- Create an instance of the module/library_cell/parameterized cell that you want to add to your module:

  ```
  cell=bitcell.bitcell(cell_6t)
  ```

- Add the subckt/submodule instance to the spice hierarchy using the `add_mod()` function from the `hierarchy_spice` class:

  ```
  self.add_mod(cell)
  ```

- Add layout instance into your module's layout hierarchy using the `add_instance`() function, which takes a name, mod, offset, and mirror as inputs:

  ```
  self.add_inst(name=name,mod=cell,offset=[x_off,y_off],mirror=x)
  ```

- Connect the pins of the instance that was just added by using the `connect_pins` function from the `hierarchy_spice` class:

  ```
  self.connect_inst([BL[%d]%col, BR[%d]%col, WL[%d]%row, gnd, vdd]).
  ```

  The pins must be listed in the same order as they were added to the submodule. Also, an assertion error will occur if there is a mismatch in the number of net connections.

- Do whatever else needs to be done. Add rectangles for power/ground rails or routing, add labels, etc...

- Every module needs to have "self" height and width variable that can be accessed from outside of the module class. These paramaters are commonly used for placing instances modules in a layout. For library cells, the `self.width` and `self.height` variables are automatically parsed from the GDSII layout using the `cell_size()` function in `vlsi_layout`. Users must define the width and height of dynamically generated designs.

- Add a call to the `DRC_LVS()` function.

6.3 GDSII Files and GdsMill)

GDSII is the standard file used in indusrty 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, labels, etc.. in the layout. In OpenRAM, we utlize a nifty tool, called gdsMill, to read, write, and manipulate GDSII files. GdsMill was developed by Michael Wieckowski at the University of Michigan.

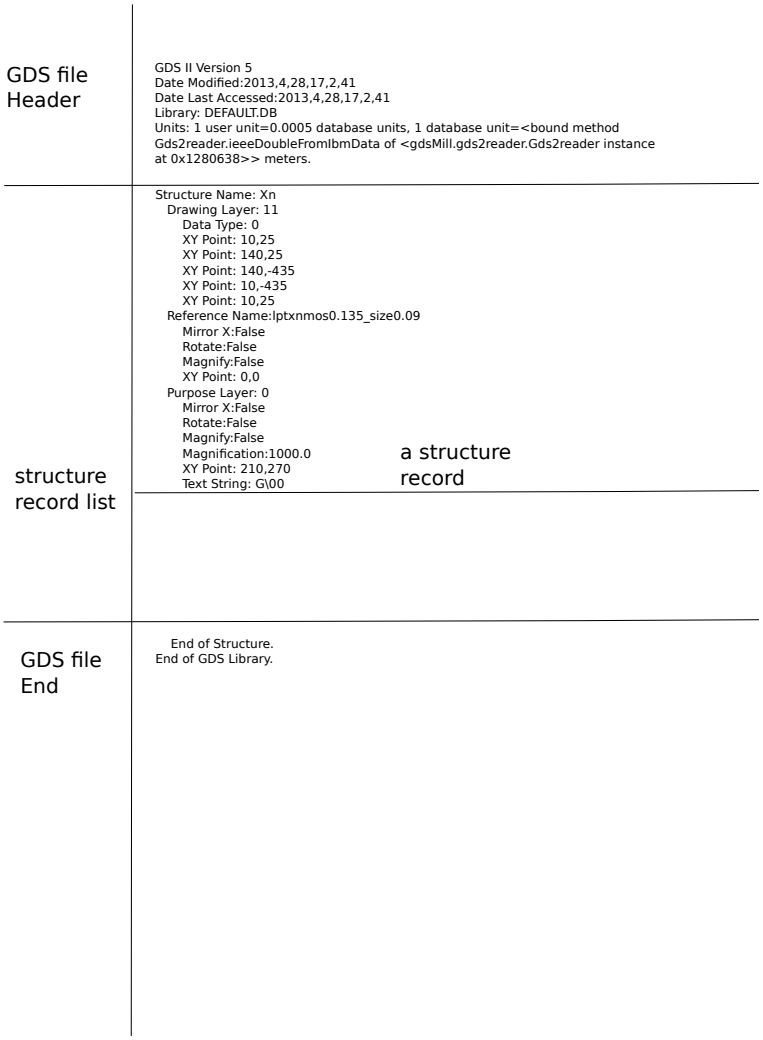

Figure 19: example of a GDSII file

6.3.1 GDSII File Format

The format of gds file contains several parts, as it could be shown in Figure 20.

The first part is the gds file header, which the contains GDSII version number, date modified, date last accessed, library, user units, and database units.

The second part is the list of structures. These structures contain geometries or references to other structures of the layout in heirarchical form. Within a structure there are several kinds of records:

- Rectangle - basic geometry unit in a design, represent one layer of material in a circuit(i.e. a metal pin). Five coordinates and layer number are stored in rectangle record.
- Structure Reference - a structure that is used in this structure. The information about this reference will be used store as a structure in the same gds file.

- Text - a text record used for labels.
- Path - used to represent a wire.
- Boundary - defines a filled polygon.
- Array Reference - specifies an array of structure instances
- Node - Electrical nets may be specified with the NODE record

The last part is the tail of the GDSII file which ends the GDS Library.

FIXME: **Provide a link to the complete GDSII specification.**

6.3.2 GdsMill

As previously stated, GdsMill is a set of scripts that can be used to read, write, and manipulate GDSII files.

The gds2_reader and gds2_writer: In GdsMill, the `gds2_reader` and `gds2_writer` classes contain the various functions used to convert data between GDSII files and the `vlsilayout` class. These classes process the data by iterating through every record in the GDS structures and check or write every data record. The record type (see Section 6.3.1),is tracked and identified using flags.

FIXME: **Do we need more information of these classes, or should we just point to the GdsMill documentation?**

The VlsiLayout Class: After the `gds2_reader` class reads in the records, the data has to be stored in a way that can be easily used by our code. Thus, the `VlsiLayout` class is made to represent the layout. `VlsiLayout` contains the same information as GDSII file but in a different way. `VlsiLayout` stores records in data structures, which are defined in `gdsPrimitives.py`. Each record type has a corresponding class defined in `gdsPrimitives`. Thus, a vlsilayout should at least contains following member data:

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

The `VlsiLayout` class also contains many functions for adding structures and records to a layout class, but the important and most useful functions have been aggregated into a wrapper file. This wrapper is called `geometry.py` and is located in the `compiler` directory.

6.3.3 OpenRAM-GdsMill Interface

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, but the GdsMill file is very large and can be difficult to understand. To make things easier, OpenRAM has its own wrapper class called `geometry` in `geometry.py`. This wrapper class initializes data structures for the instances and objects that will be added to the `VlsiLayout` class. The functions `add_inst()`, `add_rect()`, `add_label()` in `hierarchy_layout`, add the structures to the `geometry` class, which is then written out to a GDSII file using `VlsiLayout` and the `gds2_writer`.

User included library cells, which should be in gds files, can be used as dynamically generated cells by using GDSMill. Cell information such as cell size and pin location can be obtained by using built in functions in the `VlsiLayout` class.

Cell size can be finded by using the `readLayoutBorder` function of the `VlsiLayout` class. A boundary layer should be drawn in each library cell to indicate the cell area. The `readLayoutBorder` function will return the width and height of the boundary. If a boundary layer do not exist in the layout, then `measureSize` can find the physical size cell. The first method is used as primary method in `auto_Measure_libcell` the lib_utility.py, while the second method is used as a back up one. Each technolgy setup will import this utility function and read the library cell.

Pin location can be find by using the `readPin` function of the `VlsiLayout` class. The `readPin` function will return the biggest boundary which covers the label and is at the same layer as the label is.

6.4 Technology Directory

The aim of creating technology directory is to make OpenRAM portable to different technologies. This directory contains all the information related to the specific process/technology that is being used. In OpenRAM, the default technology is FreePDK45, which has it own technolony directory in the trunk. The technology-specific directory should consist of the following:

- Technology-Specific Parameters - These parameters should include layer numbers and any design rules that may be needed for generating dynamic designs (DRC rules). The parameters should be added in `/techdir/tech/tech.py` and layer map in the `/techdir`.
- Library Cells - The library cells and corresponding spice netlists should be added to the `/gds_lib` and `/sp_lib` directories.
- Portation Functions - Some of the dynamically generated cells may need helper functions to deal with technology-specific requirements. Additional, tech-specific, functions should be added to the `/techdir`.

For more information regarding the technology directory and how to set one up for a new technology, refer to Section 8

6.5 DRC/LVS Interface

Each design class contains a function `DRC_LVS()` that performs both DRC and LVS on the current design module. This enables bottom-up correct-by-construction design and easy identification of where errors occur. It does incur some run-time overhead and can be disabled on the command line. The `DRC_LVS()` function saves a GDSII file and a Spice file into a temporary directory and then calls two functions to perform DRC and LVS that are tool-dependent.

A reference implementation for the DRC and LVS functions are provided for Cadence Calibre since this is the most common DRC/LVS tool. Each of these functions generates a batch-mode "runset" file which contains the options to correctly run DRC and LVS. The functions then parse the batch mode output for any potential errors and returns the number of errors encountered.

The function `run_drc()` requires a cell name and a GDSII file. The cell name corresponds to the top level cell in the GDSII file. It also uses the layer map file for the technology to correctly import the GDSII file into the Cadence database to perform DRC. The function returns the number of DRC violations.

The function `run_lvs()` requires a cell name, a GDSII file, and a Spice file. Calibre will extract an extracted Spice netlist from the GDSII file and will then compare this netlist with the OpenRAM Spice netlist. The function returns the number of uncompared and unmatched devices/nets in the design.

For both DRC and LVS, the summary file and other report files are left in the OpenRAM temporary directory after DRC/LVS is run. These report files can be examined to further understand why errors were encountered. In addition, by increasing the debug level, the command line to re-create the DRC/LVS check can be obtained and run manually.

7 Custom Layout Design Functions in Software

OpenRAM provides classes that can be used to generated parameterized cells for the most common cells: transistors, inverters, nand2, nand3, etc... There are many advantages to having parameterized cells. The main advantage is that it makes it easier to dynamically generate designs and cuts down the necessary code to be written. We also need parameterized cells because some designs, such as the wordline drivers, need to be dynamically sized based on the size of the memory. Lastly, there may be certain physical dimension requirements that need to be met for a cell, while still maintaing the expected operation/performance. In OpenRAM we currently provide five parameterized cells: parameterized transistor (`ptx`), parameterized inverter (`pinv`), parameterized nand2 (`nand_2`), parameterized nand3 (`nand_3`) and parameterized nor2 (`nor_2`).

7.1 Parameterized Transistor

The parameterized transistor class generates a transistor of specified width and number of mults. The `ptx` is constructed as follows:

```
def __init__(self,name,width,mults,tx_type)
```

An explanation of the `ptx` parameters is shown in Table 4. A layout of ptx, generated by the following instatiation, is depicted in Figure 21.

```
fet = ptx.ptx(name = "nmos_1_finger", width = tech.drc["minwidth_tx"],
mults = 1, tx_type = "nmos").
```

Parameter	Explanation
`width`	active_height
`mults`	mult number of the transistor
`tx_type`	type of transistor,nmos and pmos

Table 4: Parameter Explanation of ptx

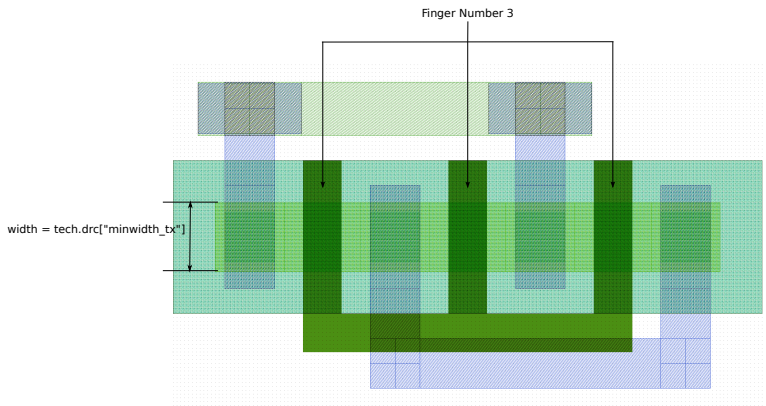

Figure 20: An example of Parameterized Transistor (ptx)

7.2 Parameterized Inverter

The parameterized inverter (`pinv`) class generated an inverter of a specified size/strength and height. The `pinv` is constructed as follows:

```
def __init__(self, cell_name, size, beta=tech.[pinv.beta],
cell_size=tech.cell[height])
```

The parameterized inverter can provide significant drive strength while adhering to physical cell size limitations. That is achieved by having many small transistors connected in parallel, thus the height of the inverter cell can be manipulated without the affecting the drive strength. The NMOS size is an input parameter, and the PMOS size will be determined by $beta * NMOS_size$, where beta is the ratio of the PMOS channel width to the NMOS channel width. The following code instatiates the `pinv` instance seen in Figure 22.

```
a=pinv.pinv(cell_name="pinv",size=tech.drc["minwidth_tx"]*8)
```

The `pinv` parameters are explained in Table 5.

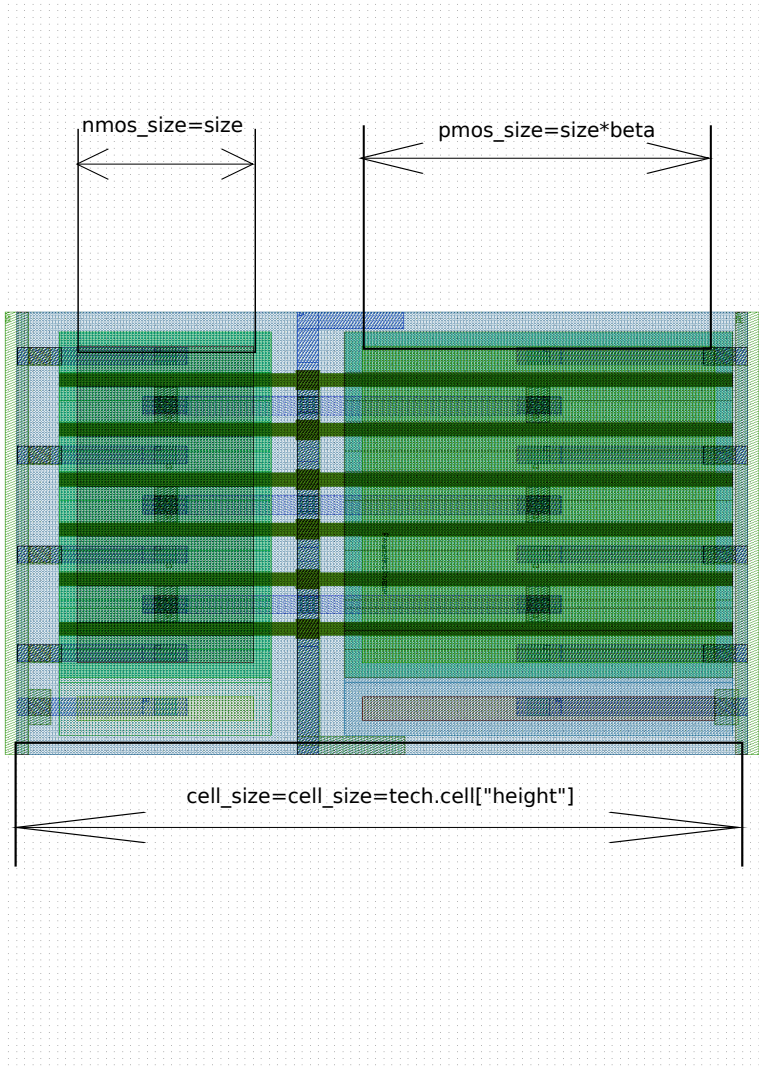

Figure 21: An example of Parameterized Inverter(pinv)

Parameter	Explanation
`size`	The logic size of the transistor of the nmos in the pinv
`beta` = tech.[pinv.beta]	Ratio of pmos channel width to nmos channel width.
`cell_size` = tech.cell[height]	physical dimension of cell height.

Table 5: Parameter Explanation of pinv

7.3 Parameterized NAND2

The parameterized nand2 (`nand_2`) class generated a 2-input nand gate of a specified size/strength and height. The `nand_2` is constructed as follows:

```
def __init__(self, name, nmos_width, height=tech.cell_6t[height])
```

The NMOS size is an input parameter, and the PMOS size will be equal to NMOS to have the equal rising and falling for output. The following code instatiates the `nand_2` instance seen in Figure 23.

```
a=nand_2.nand_2(name="nand2", nmos_width=2*tech.drc["minwidth_tx"],
height=tech.cell_6t["height"])
```

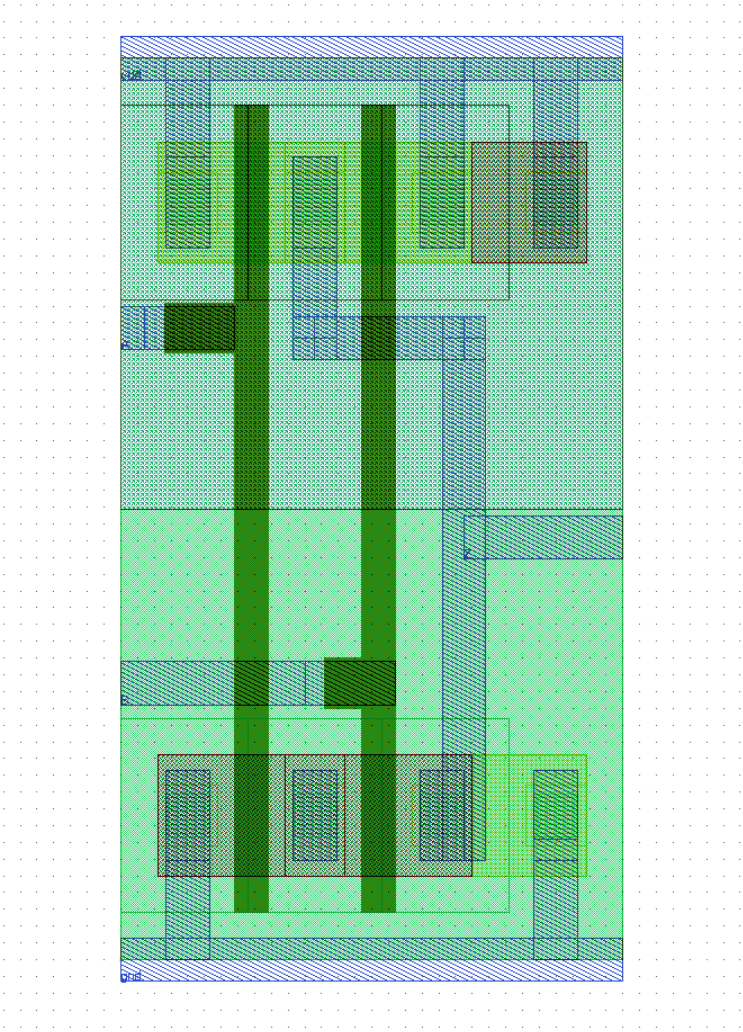

Figure 22: An example of Parameterized NAND2(nand_2)

The `nand_2` parameters are explained in Table 6.

Parameter	Explanation
`nmos_width`	The logic size of the transistor of the nmos in the nand2
`height` = tech.cell_6t[height]	physical dimension of cell height.

Table 6: Parameter Explanation of nand2

7.4 Parameterized NAND3

The parameterized nand3 (`nand_3`) class generated a 3-input nand gate of a specified size/strength and height. The `nand_3` is constructed as follows:

```
def __init__(self, name, nmos_width, height=tech.cell_6t[height])
```

The NMOS size is an input parameter, and the PMOS size will be equal to $2/3$ NMOS size to have the equal rising and falling for output. The following code instatiates the `nand_3` instance seen in Figure 24.

```
a=nand_3.nand_3(name="nand3", nmos_width=3*tech.drc["minwidth_tx"],
height=tech.cell_6t["height"])
```

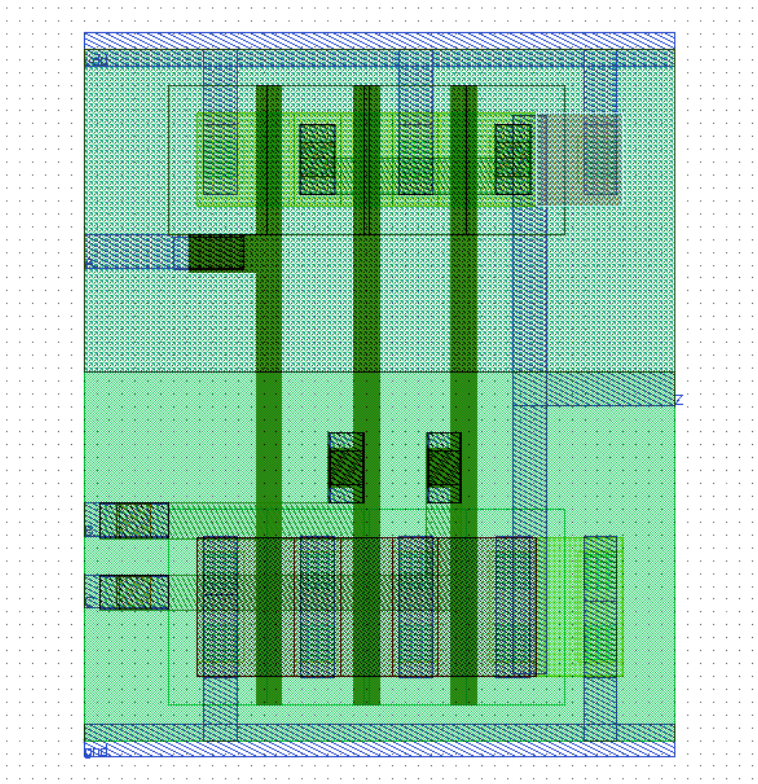

Figure 23: An example of Parameterized NAND3(nand_3)

The `nand_3` parameters are explained in Table 7.

Parameter	Explanation
`nmos_width`	The logic size of the transistor of the nmos in the nand3
`height` = tech.cell_6t[height]	physical dimension of cell height.

Table 7: Parameter Explanation of nand3

7.5 Parameterized NOR2

The parameterized nor2 (`nor_2`) class generated a 2-input nor gate of a specified size/strength and height. The `nor_2` is constructed as follows:

```
def __init__(self, name, nmos_width, height=tech.cell_6t[height])
```

The NMOS size is an input parameter, and the PMOS size will be equal to 2 NMOS size to have the equal rising and falling for output. The following code instatiates the `nor_2` instance seen in Figure 25.

```
a=nor_2.nor_2(name="nor2", nmos_width=2*tech.drc["minwidth_tx"],
height=tech.cell_6t["height"])
```

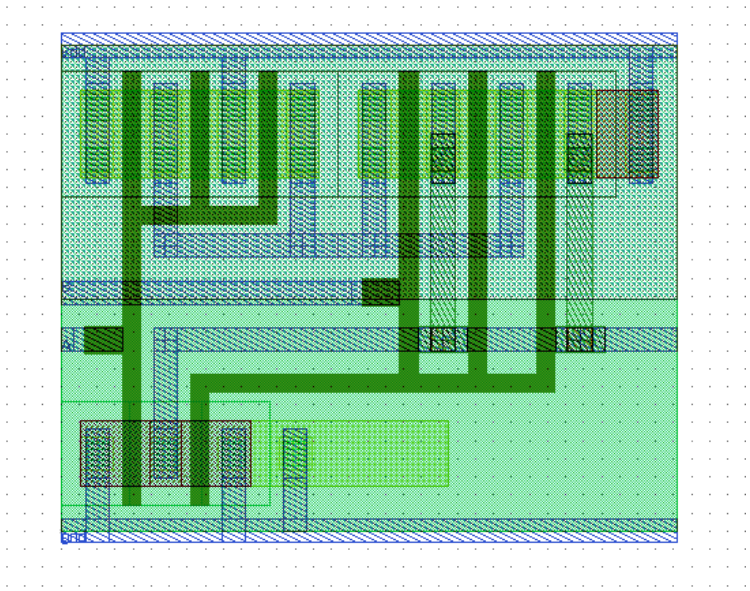

Figure 24: An example of Parameterized NOR2(nor_2)

The `nor_2` parameters are explained in Table 8.

Parameter	Explanation
`nmos_width`	The logic size of the transistor of the nmos in the nor2
`height` = tech.cell_6t[height]	physical dimension of cell height.

Table 8: Parameter Explanation of nor2

7.6 Path and Wire

OpenRam provides two routing classes in custom layout design. Both Path and wire class will take a set of coordinates connect those points with rectilinear metal connection.

The difference is that path only use the same layers for both vertical and horizontal connection while wire will use two different adjacent metal layers. The this example will construct a metal1 layer path

```
layer_stack = ("metal1")
position_list = [(0,0), (0,3), (1,3), (1,1), (4,3)]
w=path.path(layer_stack,position_list)
```

and This exmaple will construct a wire using metal1 for vertical connection and metal2 for horizontal connection:

```
layer_stack = ("metal1","via1","metal2")
position_list = [(0,0), (0,3), (1,3), (1,1), (4,3)]
w=wire.wire(layer_stack,position_list)
```

8 Porting to a new Technologies

The folllowing sub-directories and files should be added to your new technology directory:

- `/sp_lib` - spice netlists for library cells
- `/gds_lib` - GDSII files for the library cell
- `layers.map` - layer/purpose pair map from the technology
- `/tech` - contains tech parameters, layers, and portation functions.

8.1 The GDS and Spice Libraries

The GDS and Spice libraries , `\gds_lib` and `\sp_lib`, should contain the GDSII layouts and spice netlists for each of the library cells in your SRAM design. For the FreePDK45 technology, library cells for the 6T Cell, Sense Amp, Write Driver, Flip-Flops, and Control Logic are provided. To reiterate: all layouts must be exported in the GDSII file format. The following commands can be used to stream GDSII files into or out of Cadence Virtuoso:

```
To stream out of Cadence:

  strmout -layerMap ../sram_lib/layers.map
    -library sram -topCell $i -view layout
      -strmFile ../sram_lib/$i.gds

To stream a layout back into Cadence:

  strmin -layerMap ../sram_lib/layers.map
    -attachTechFileOfLib NCSU_TechLib_FreePDK45
       -library sram_4_32 -strmFile sram_4_32.gds
```

When you import a gds file, make sure to attach the correct tech lib or you will get incorrect layers in the resulting library.

8.2 Technology Directory

Inside of the `/tech` directory should be the Python classes for `tech.py`, `ptx_port.py`, and any other portation functions. The `tech.py` file is very important and should contain the following:

- Layer Number/Name - GDSII files only contain layer numbers and it can be difficult to keep track of which layer corresponds to what number. In OpenRAM code, layers are referred to by name and `tech.py` maps the layer names that we use to the layer numbers in the `layer.map` This will associate the layer name used in OpenRAM program with the number used in the layer.map, thus the code in complier wont need to be changed for each technology.
- Tech Parameters - important rules from the DRC rule deck(such as layer spacing and minimum sizes) should be included here. Please refer to the rules that are included in `tech.py` to get a better idea as to what is important.
- Cell Sizes and Pin Offsets - The `cell_size()` and `pin_finder()` functions should be used to populate this class with the various cell sizes and pin locations in your library cells. These functions are relatively slow because they must traverse the every shape in the entire hierarchy of a design. Due to this fact, these function are not invoked each time the compiler is run, it should be run one time or if any changes have been made to library cells. This sizes and pin locations gathered are needed to generate the dynamic cells and perform routing at the various levels of the hierarchy. It is suggested that boundary boxes on a specific layer should be added to define the cell size.

9 Timing and Control Logic

This section outlines the necessary signals, timing considerations, and control circuitry for a synchronous SRAM.

9.1 Signals

Top-Level Signals:

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

Internal Signals:

- clk_bar - enables the precharge unit.
- s_en - enables the sense amp during a read operation.

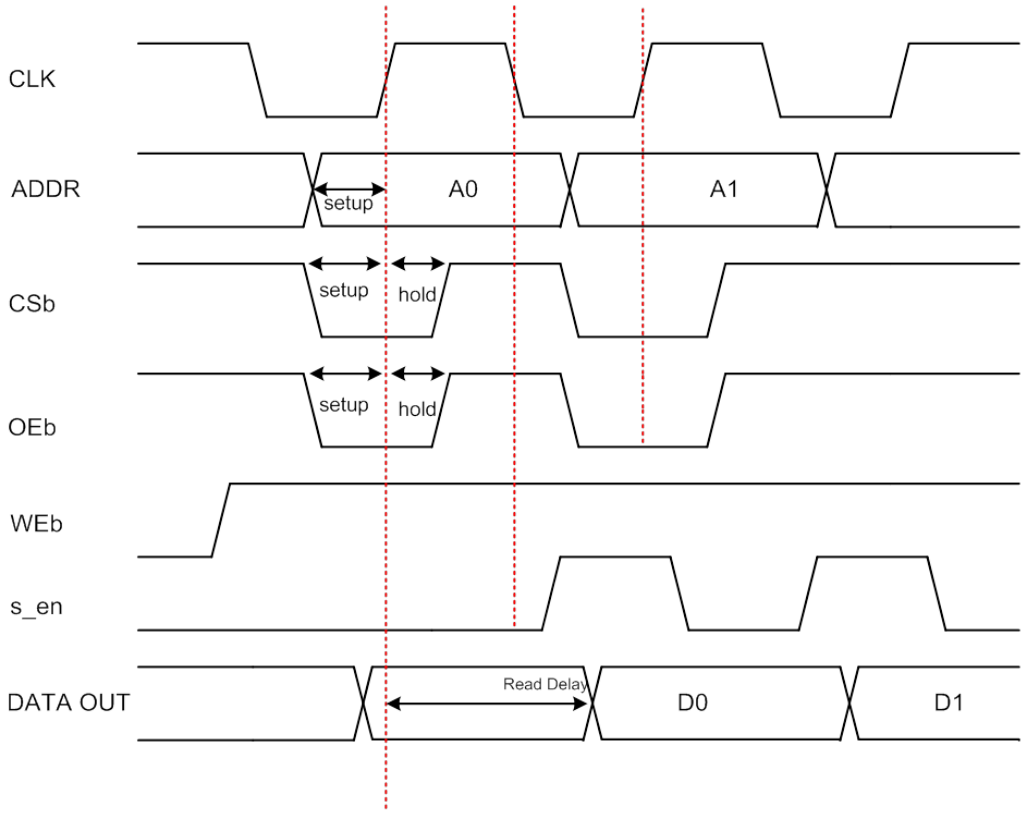

Figure 25: Timing diagram for read operation showing the setup, hold, and read times.

- w_en - enable the write driver during a write operation.
- tri_en and tri_en_bar - enable the data input tri-gate during a read operation.

9.2 Timing Considerations

The main timing considerations for an SRAM are:

- Setup Time - time an input needs to be stable before the positive/negative clock edge.
- Hold Time - time an input needs to stay valid after the positive/negative clock edge.
- Minimun Cycle Time - time inbetween subsequent memory operations.
- Memory Read Time - time from positive clock edge until valid data appears on the data bus.
- Memory Write Time - time from negative clock edge until data has been driven into a memory cell.

9.3 SRAM Operation

Read Operation:

1. Before the clock transition (low to high) that initiates the read operation:

(a) The chip must be selected (CSb low).
(b) The WEb must be high (read).
(c) The row and column addresses must be applied to the address input pins (ADDR).
(d) OEb should be selected (OEb low).

2. On the rising edge of the clock (CLK):
   (a) The control signals and address are latched into flip-flops and the read cycle begins.
   (b) The precharging of the bit lines starts.
   (c) The address bits become available for the decoder and column mux, which select the row and columns that we want to read from.

3. On the falling edge of the clock (CLK):
   (a) Word line has been asserted, the value stored in the memory cells pulls down one of the bitlines (BL if a 0 is stored, BL_bar if a 1 is stored).
   (b) s_en enables the sense amplifier which senses the voltage difference of the bit lines, produces the output and keeps the value in its latch circuitry.
   (c) Tri-gate enables and put the output data on data bus. Data remains valid on the data bus for a complete clock cycle.

Write Operation:

1. Before the clock transition (low to high) that initiates the write operation:
   (a) The chip must be selected (CSb low).
   (b) The WEb must be low to enable the data input tristates.
   (c) The row and column addresses must be applied to the address input pins (ADDR).
   (d) OEb must be high (no output is available and sense amp disabled)

2. On the rising edge of the clock (CLK):
   (a) OEb stays high (no output is available and sense amp disabled)
   (b) The inputs addresses are latched into flip-flops, precharging starts, and the write operation begins.
   (c) The address bits become available for the decoder and column mux, which select the row and columns that we want to write to.

3. On the falling edge of the clock (CLK):
   (a) The data to be written must be applied to DATA and latched into flip-flops.
   (b) w_en enables the write driver, which drives the data input through the column mux and into the selected memory cells. The write delay is the time from the negative clock edge until the data value is stored in the memory cell on node X.

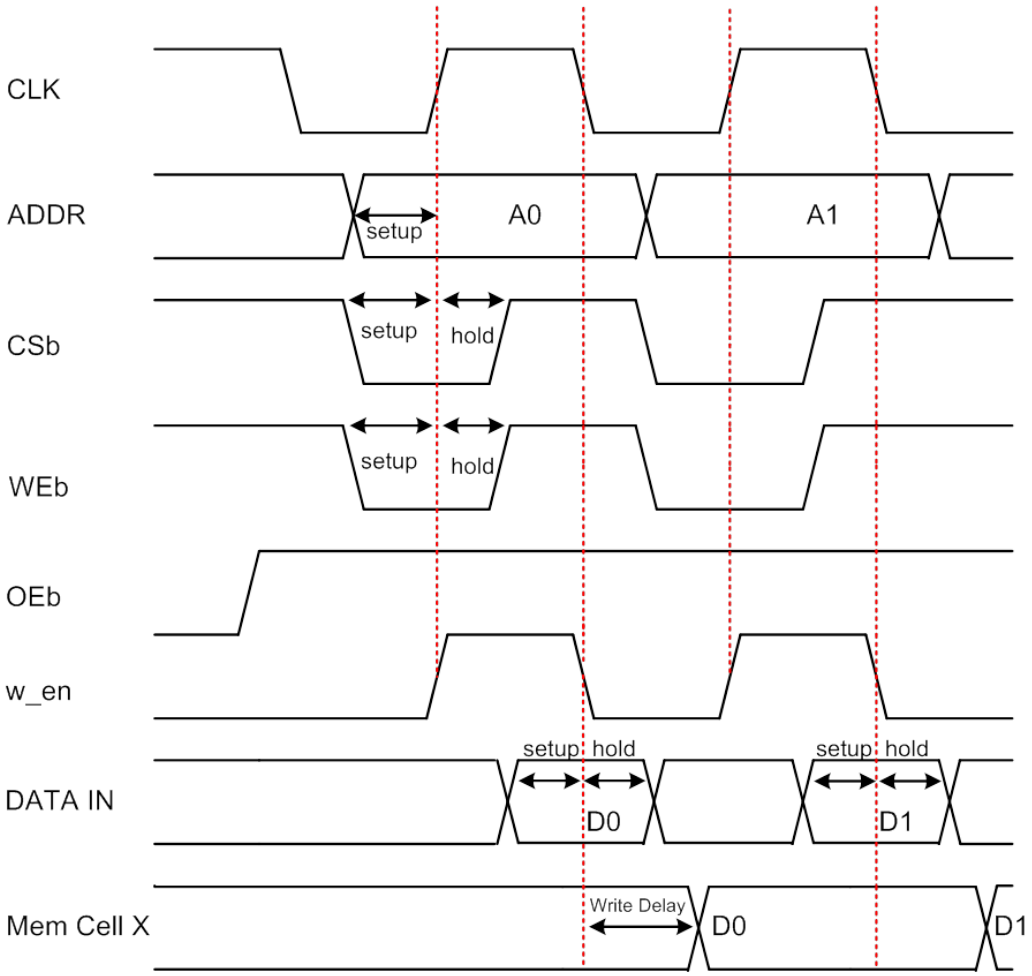

Figure 26: Timing diagram for write operation showing the setup, hold, and write times.

9.4 Zero Bus Turnaround (ZBT)

In timing of SRAM, during a read operation, data should be available after the clock edge while during a write, data should be set up before the clock edge. Due to this issue a wait state (dead cycle) is neccessary when SRAM switches from read mode to write mode. To avoide dead cycles in SRAM timing which slow down the operation and degrade the performance of SRAM, Zero Bus turnaround (ZBT) technique is used. Using ZBT, during a write, data is set up after positive clock edge and before negative clock edge and input data is latched in negative edge flip-flops. Using ZBT, we will get a higher memory throughput and there is no waite states. Figure 27 shows the correct timing for input signals during the write opertion to avoide the wait states. Figure 28 shows how a write cycle is followed by a read cycle with no wait state through using ZBT. Input address bits should be ready before positive edge to be loaded to positive edge flip-flops. Output data is ready to be loaded to data-bus during seconde half of cycle (after negative edge of clock) and input data should be ready before negative edge of clock to be loaded in negative edge flip-flops.

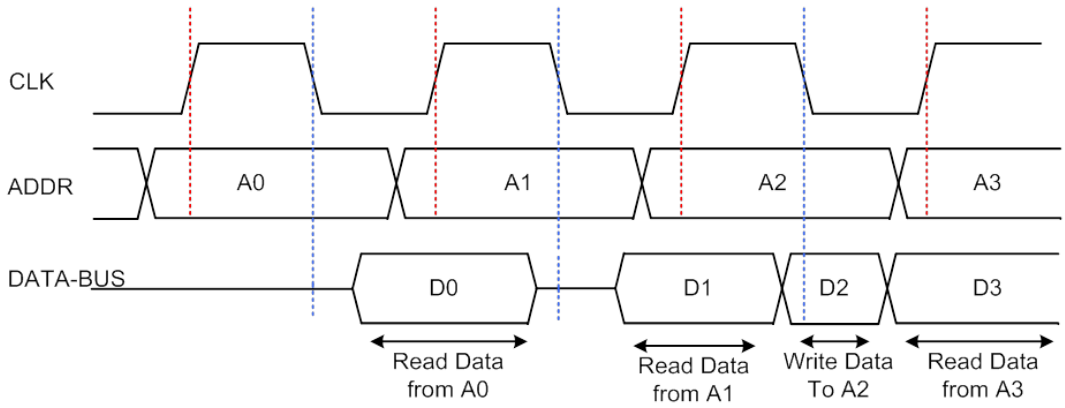

Figure 27: (a) Zero Bus Turnaround timing.

9.5 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. As shown in Figure 29, 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. When CSb is low, the appropriate control signals are generated and sent to the architecture blocks. Conversely, if CSb is high then no control signals are generated and SRAM is 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. All of the input control signals are latched with master-slave flip-flops, ensuring that the control signal stays valid for the entire operation cycle. The control signal flip-flops use the normal clock to generate local signals used to enable or disable structures based on the operation. Address flip-flops are combined with global clock as well. In a standard write SRAM, switching from a read to a write operation results in a dead cycle. To avoid this dead cycle, Data flip-flops are latched with clk_bar in order to have a Zero Bus Turnaround (ZBT) memory. More details on ZBT timing are outlined in Section 9.4. After all control signals are latched, they are ANDED with the clk_bar because the read/write circuitries should only be enabled after the precharging of the bitlines had ended on the negative edge of the clock. The w_en signal enables the write driver during a write to the memory .The s_en signal is generated using a Replica Bitline (RBL) to enable the sense amplifier during a read operation. Details on RBL architecture are outlined in section 9.6. tri_en and tri_en_bar enable the tristates during read in order to drive the outputs onto the data bus. Table 9 shows the truth table for the control logic. The s_en signal to enable the sense amplifier is true when $(CS.OE.Clk_bar)$ is true. Similarly, write driver enable signal, w_en, is true when $(CS.WE.clk_bar)$ is true. tri_en and tri_en_bar are true when $\neg(OEb_bar|clk)$ and $\neg(OEb.clk_bar)$ are true, respectively.

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

Table 9: Generation of control signals.

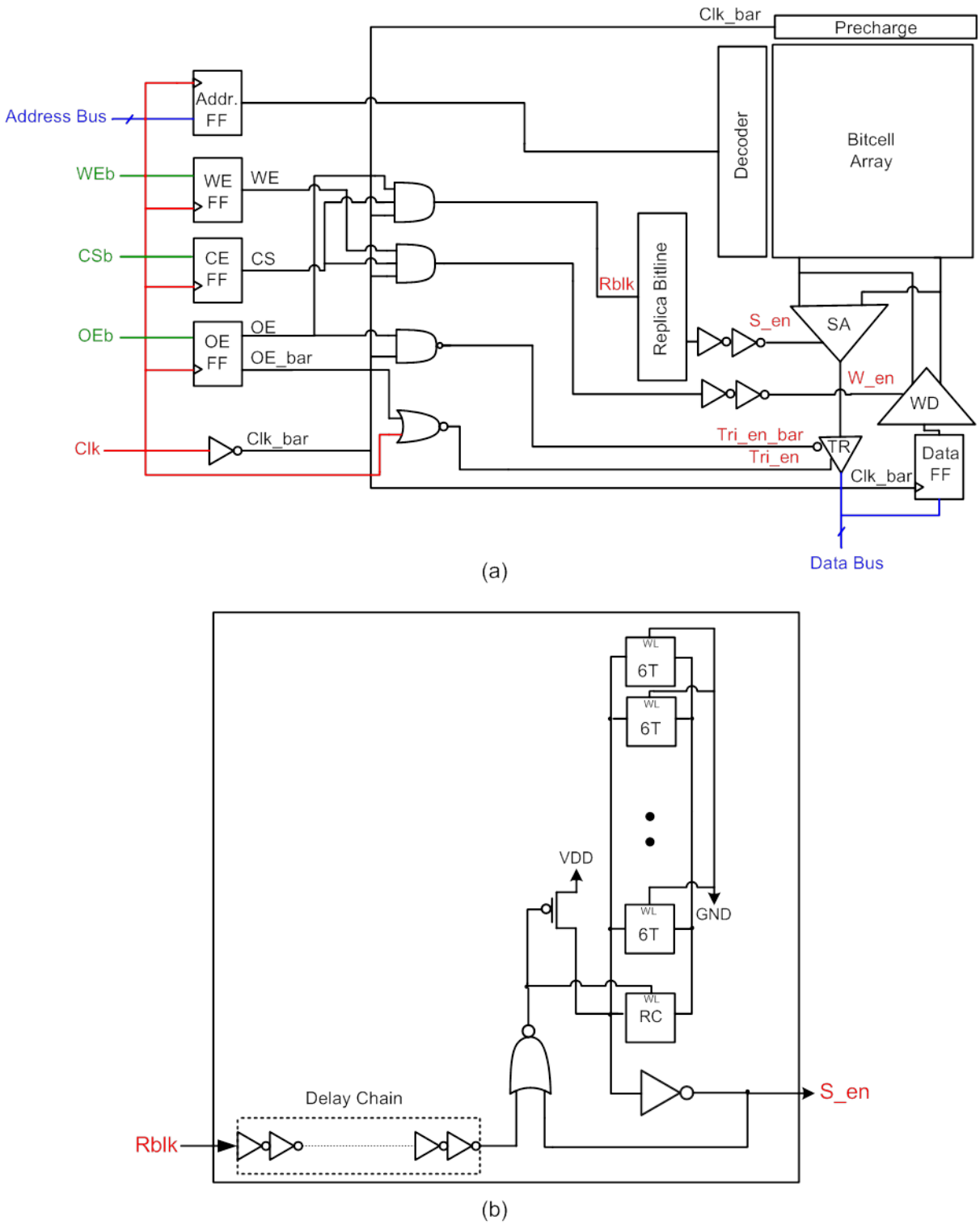

Figure 28: (a) Control Logic diagram and (b) Replica Bitline Schematic.

9.6 Replica Bitline Delay

In SRAM read operation, discharging the bitline is the most time consuming procedure. Generally, sense amplifier amplifies the small voltage difference on the bitlines at the proper sense timing, to realize high-speed operation. Therefore, the timing for sense amplifier (s_en) is extremely important for the high speed and low power SRAM. If the s_en arrives early before the bitline difference reaches the sense amplifier input transistors offset voltage, a read functional failure may occur. Contrarily, a late-arrived s_en would consume more unnecessary time, thereby wasting the power. The conventional way of generating s_en signal is to use a replica bitline (RBL). RBL as shown in 30 consists of a column of SRAM cells (dummy cells), which track the random process variation in array. RBL is presented for matching the delay of the activation of the sense amplifier with the delay of the propagation of the required voltage swing at the bitlines. In RBL technique, delay driven memory cell in control path is same as read path. Therefore the delay shift of control path according to the Process, Voltage and Temperature (PVT) variation is same ratio as that of read path. The RBL technique attains self-timed tracking with optimal s_en timing according to PVT variation. Using replica circuits, the variation on the delay of the sense amp activation and bitline swing is minimized.

RBL technique uses a Replica Cell (RC) driving a short bitline signal. The short bitlineś capacitance is set to be a fraction of the main bitline capacitance (e.g. one tenth). This fraction is determined by the required bitline swing (bitline voltages larger than offset voltage at input transistors of sense amplifier) for proper sensing. So in SRAM, an extra column block is converted into the replica column whose capacitance is the desired fraction of the main bitline. Therefore, its capacitance ratio to the main bitlines is set purely by the ratio of the geometric lengths (e.g. one tenth). The RC is hard wired to store a zero such that it will discharge the RBL once it is accessed. Because of its similarity with the actual memory cell (in terms of design and fabrication) the delay of RBL tracks the delay of real bitlines very well and can be made roughly equal. Figure 31 shows the schematic of the 6T replica cell. The timing for s_en is generated as follows. At first, the RBL and the normal bitlines are precharged to VDD. Next, selected memory cells and RC are activated. RC draws the current from the RBL and normal bitlines are also discharged through the accessed cell. Discharged swing on RBL is inverted and then buffered to generate the signal to enable the sense amplifier.

9.7 Timing and Power Characterizer

The section will provide an explanantion of the characterizer that will generete spice stimuli for the top-level SRAM and perform spice timing simulations to determine the memory setup&hold times, the write delay, and read delay. It will also provide a spice power estimate.

10 Unit Tests

OpenRAM comes with a unit testing framework based on the Python unittest framework. Since OpenRAM is technology independent, these unit tests can be run in any technology to verify that the technology is properly ported. By default, FreePDK45 is supported.

The unit tests consist of the following tests that test each module/sub-block of OpenRAM:

- `00_code_format_check__test.py` - Checks the format of the codes. returns error if finds TAB in codes.

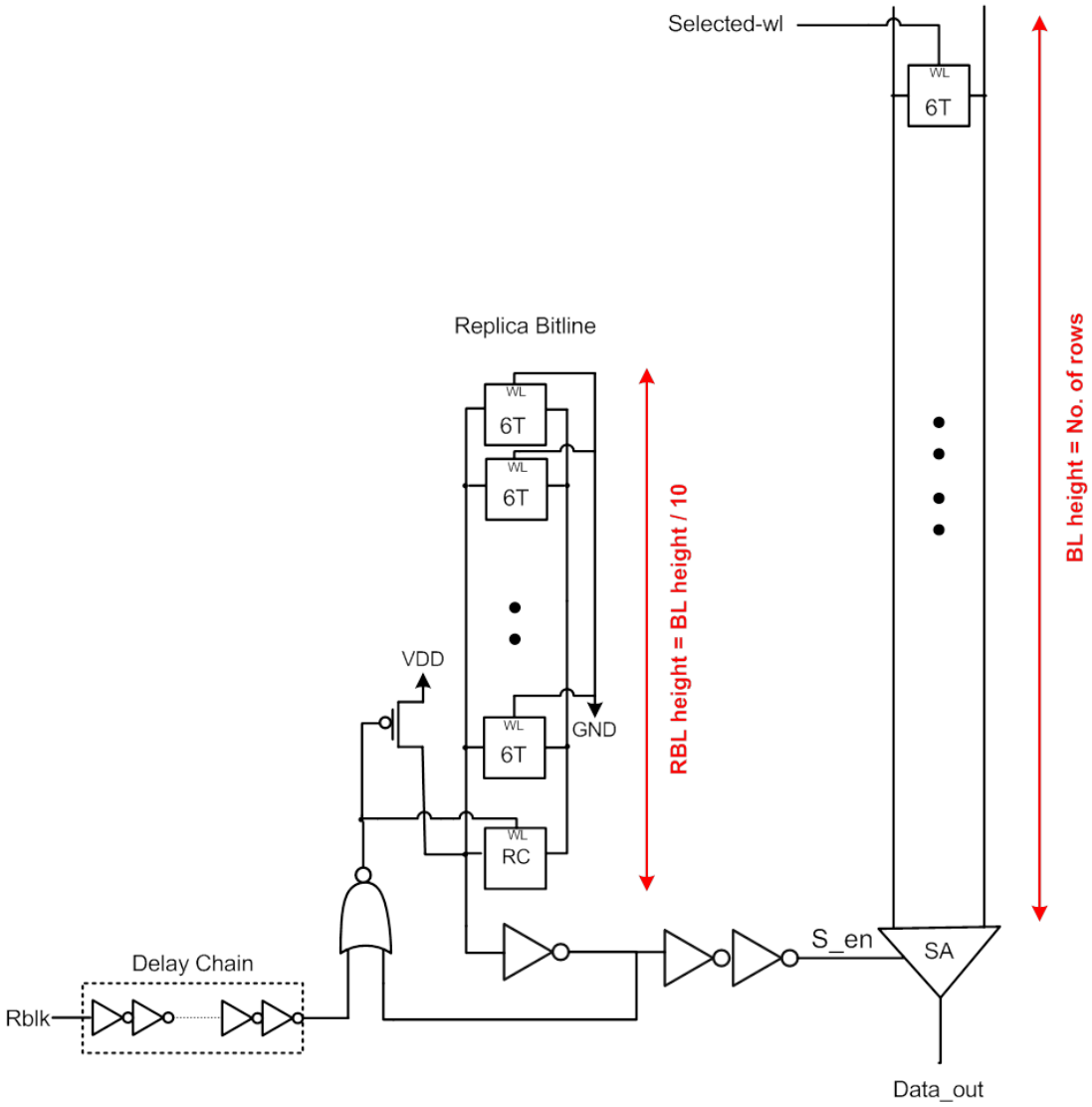

Figure 29: Replica Bitline Schematic

- `01_library_drc_test.py` - DRC of library cells in technology `gds_lib`
- `02_library_lvs_test.py` - LVS of library cells in technology `gds_lib` and `sp_lib`
- `03_contact_test.py` - Test contacts/vias of different layers
- `03_path_test.py` - Test different types of paths based off of the wire module
- `03_ptx_test.py` - Test various sizes/fingers of PMOS and NMOS parameterized transistors
- `03_wire_test.py` - Test different types of wires with different layers
- `04_pinv_test.py` - Test various sizes of parameterized inverter
- `04_nand_2_test.py` - Test various sizes of parameterized nand2

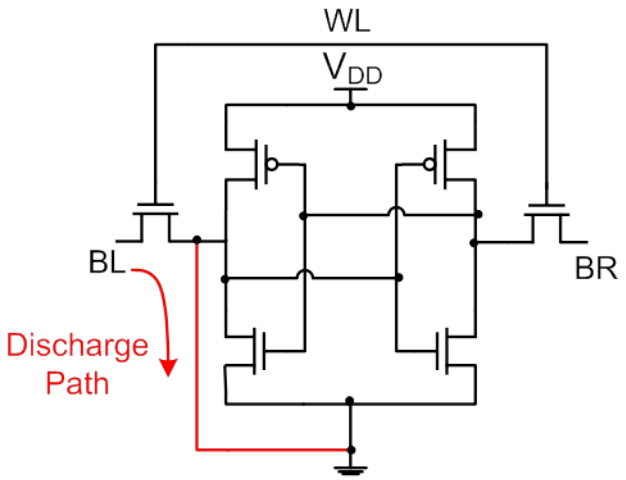

Figure 30: Replica Bitline Schematic

- `04_nand_3_test.py` - Test various sizes of parameterized nand3
- `04_nor_2_test.py` - Test various sizes of parameterized nor2
- `04_wordline_driver_test.py` - Test a wordline_driver array.
- `05_array_test.py` - Test a small bit-cell array
- `06_nand_decoder_test.py` - Test a dynamic NAND address decoder
- `06_hierarchical_decoder_test.py` - Test a dynamic hierarchical address decoder
- `07_tree_column_mux_test.py` - Test a small tree column mux.
- `07_single_level_column_mux_test.py` - Test a small single level column mux.
- `08_precharge_test.py` - Test a dynamically generated precharge array
- `09_sense_amp_test.py` - Test a sense amplifier array
- `10_write_driver_test.py` - Test a write driver array
- `11_ms_flop_array_test.py` - Test a MS_FF array
- `13_control_logic_test.py` - Test the control logic module
- `14_delay_chain_test.py` - Test a delay chain array
- `15_tri_gate_array_test.py` - Test a tri-gate array
- `16_replica_bitline_test.py` - Test a replica bitline
- `19_bank_test.py` - Test a bank
- `20_sram_test.py` - Test a complete small SRAM

- `21_timing_sram_test.py` - Test timing of SRAM
- `22_sram_func_test.py` - Test functionality of SRAM

Each unit test instantiates a small component and performs DRC/LVS. Automatic DRC/LVS inside OpenRAM is disabled so that Python unittest assertions can be used to track failures, errors, and successful tests as follows:

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

Each of these assertions will trigger a test failure. If there are problems with interpreting modified code due to syntax errors, the unit test framework will not capture this and it will result in an Error.

10.1 Usage

A regression script is provided to check all of the unit tests by running:

```
python tests/regress.py
```

from the compiler directory located at: ”OpenRAM/trunk/compiler/”. Each individual test can be run by running:

```
python tests/{unit-test file}
e.g. python tests/05_array_test.py
```

from the compiler directory located at: ”openram/trunk/compiler/”. As an example, the unit tests all complete and provide the following output except for the final `20_sram_test` which has 2 DRC violations:

```
[trunk/compiler]$ python tests/regress.py
runTest (01_library_drc_test.library_drc_test) ... ok
runTest (02_library_lvs_test.library_lvs_test) ... ok
runTest (03_contact_test.contact_test) ... ok
runTest (03_path_test.path_test) ... ok
runTest (03_ptx_test.ptx_test) ... ok
runTest (03_wire_test.wire_test) ... ok
runTest (04_pinv_test.pinv_test) ... ok
runTest (04_nand_2_test.nand_2_test) ... ok
runTest (04_nand_3_test.nand_3_test) ... ok
runTest (04_nor_2_test.nor_2_test) ... ok
runTest (04_wordline_driver_test.wordline_driver_test) ... ok
runTest (05_array_test.array_test) ... ok
runTest (06_hierdecoder_test.hierdecoder_test) ... ok
runTest (07_single_level_column_mux_test.single_level_column_mux_test) ... ok
runTest (08_precharge_test.precharge_test) ... ok
```

```
runTest (09_sense_amp_test.sense_amp_test) ... ok
runTest (10_write_driver_test.write_driver_test) ... ok
runTest (11_ms_flop_array_test.ms_flop_test) ... ok
runTest (13_control_logic_test.control_logic_test) ... ok
runTest (14_delay_chain_test.delay_chain_test) ... ok
runTest (15_tri_gate_array_test.tri_gate_array_test) ... ok
runTest (19_bank_test.bank_test) ... ok
runTest (20_sram_test.sram_test) ... ok
```

If there are any DRC/LVS violations during the test, all the summary,output,and error files will be generated in the technology directory's "openram_temp" folder. One would view those files to determine the cause of the DRC/LVS violations.

More information on the Python unittest framework is available at

http://docs.python.org/2/library/unittest.html.

11 Debug Framework

All output in OpenRAM should use the shared debug framework. This is still under development but is in a usable state. It is going to be replaced with the Python Logging framework which is quite simple.

All of the debug framework is contained in debug.py and is based around the concept of a "debug level" which is a single global variable in this file. This level is, by default, 0 which will output normal minimal output. The general guidelines for debug output are:

- 0 Normal output
- 1 Verbose output
- 2 Detailed output
- 3+ Excessively detailed output

The debug level can be adjusted on the command line when arguments are parsed using the "-v" flag. Adding more "-v" flags will increase the debug level as in the following examples:

```
python tests/01_library_drc_test.py -vv
python openram.py 4 16 -v -v
```

which each put the program in debug level 2 (detailed output).

Since every module may output a lot of information in the higher debug levels, the output format is standardized to allow easy searching via grep or other command-line tools. The standard output formatting is used through three interface functions:

- debug.info(int, msg)

- debug.warning(msg)
- debug.error(msg)

The msg string in each case can be any string format including data or other useful debug information. The string should also contain information to make it human understandable. **It should not just be a number!** The warning and error messages are independent of debug levels while the info message will only print the message if the current debug level is above the parameter value.

The output format of the debug info messages are:

```
[ module ]:  msg
```

where module is the calling module name and msg is the string provided. This enables a grep command to get the relevant lines. The warning and error messages include the file name and line number of the warning/error.

GDSMill

OpenRAM uses gdsMill, a GDS library written by Michael Wieckowski at the University of Michigan. Michael gave us complete permission to use the code. Since then, we have made several bug and performance enhancements to gdsMill. In addition, gdsMill is no longer available on the web, so we distribute it along with OpenRAM.

```
From: Michael Wieckowski <wieckows@umich.edu>
Date: Thu, Oct 14, 2010 at 12:49 PM
Subject: Re: GDS Mill
To: Matthew Guthaus <mrg@soe.ucsc.edu>

Hi Matt,

Feel free to use / modify / distribute the code as you like.

-Mike

On Oct 14, 2010, at 3:07 PM, Matthew Guthaus wrote:
> Hi Michael (& Dennis),
>
> A student and I were looking at your GDS tools, but
> we noticed that there is no license. What is the license?
>
> Thanks,
>
> Matt
```