IMPLEMENTATION OF HIERARCHICAL PREDECODER/DECODER STRUCTURE IN OPENRAM OPENSOURCE MEMORY COMPILER

By

MANJU KIRAN SUBBARAYAPPA

Bachelor of Engineering in Electronics and

Communication,

Sir M Visveswaraya Institute of Technology,

Bangalore, Karnataka, India

July 2011

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2013

IMPLEMENTATION OF HIERARCHICAL PREDECODER/DECODER STRUCTURE IN OPENRAM OPENSOURCE MEMORY COMPILER

Thesis Approved:

Dr. James E Stine

Thesis Adviser

Dr. Louis Johnson

Dr. Reza Abdolvand

Name: MANJU KIRAN SUBBARAYAPPA

Date of Degree: JULY, 2013

Title of Study: IMPLEMENTATION OF HIERARCHICAL REDECODER/DECODER STRUCTURE IN OPENRAM OPENSOURCE MEMORY COMPILER.

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

ABSTRACT: This thesis deals with the implementation of the decoder structure within the OPENRAM, open-source memory compiler. Memory compilers are software scripts that are used to generate memory macros according to a user's specification. Standard-Random Access Memory (SRAM) consists of an architecture that is made up of memory arrays and peripheral circuits. The peripheral circuit includes row decoder, column selection, pre-charge, write drivers and sense amplifier circuits. In this thesis, Python software is utilized to hierarchically create a decoder that is constructed efficiently given a set of arguments. In order to accomplish this task, a hierarchical decoder utilizes two levels of decoding to build its structure efficiently. This module accepts parameters for a given SRAM size, detects the pre-decoder stages required for the given address size and generates the hierarchical decoder structure accordingly. Area and Delay results are demonstrated within the FREEPDK technology for several different sizes.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

The modern day microprocessors are equipped with smaller and faster on-chip memories called caches. The caches are used for storing the commonly used data and instructions which decreases the frequency of main memory access [2]. The increase in need for the processor's performance has led to extensive use of cache memories that have significantly reduced the memory access time. Usually there are three levels of cache memories on most modern processer chips. Depending on the processor performance tradeoffs the cache sizes vary at each level. The size and associativity of cache memories plays a crucial role in processor performance. Based on the application requirement, parameters such as size and associativity of cache memories are varied to get the best performance of the Processors. In general purpose microprocessors, the size and associativity of cache memories are determined by analyzing the processors performance with different benchmark programs.

The cache memories in the microprocessors are constructed using the SRAM memory architecture due to its fast read and write access capability [2]. The time it takes for the memory layouts to be constructed manually is very large. The layouts of memory structures laid out on the silicon chip are highly regular. We take advantage of this regular structure and create software scripts which can assemble the layouts within minutes. These software scripts which are used for assembling the layouts are collectively called memory compiler.

Memory compilers are extensively used for memory layout generation for all processors and systems on chips. The hard and soft macros produced can be incorporated into the design flow there by decreasing the design to market time. The figure 1.1 shows the block diagram die image of the Intel's i7 processor adapted from [2] which has 4 cores on a single chip and has three levels of the cache memories. Level 1 and Level 2 cache memories are specific to each core and Level 3 is shared among the four cores.

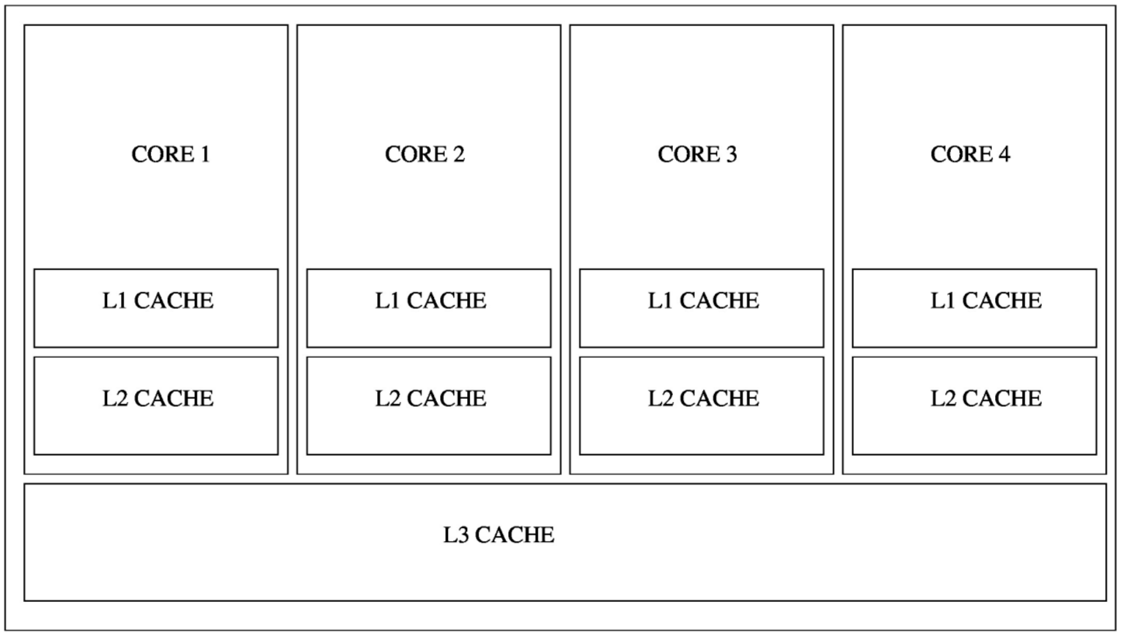


Figure1.1 Intel's I7 Quad Core Processor Die block diagram

The sizes of the cache memories on this Quad core Processor at each level are listed in Table 1.1. The small size of a cache in lower levels facilitates the processors to fetch the data and instruction in a single clock cycle. Higher level caches which are comparatively larger in size have access times ranging from 10 to 15ns. In the fetch cycle, the processor looks for the data or instruction starting from the lower level of the cache memories [2]. If the required data is found then this event is termed as a cache hit else it is termed as a cache miss. In the event of the cache miss at a lower level the processor tries to find the data or instruction at the next higher level cache. If the cache miss occurs at the highest level then the data or instruction will be fetched from the main memory. This requires 100 to 200 processor clock cycles during which the processor is idle thereby reducing the performance of the processor. The data or instruction fetched from the main memory will also be stored in the cache for later use.

CPU	Core	Intel i7	Intel i5
L1 Cache size	Specific to core	32kb	32kb
L2 Cache size	Specific to core	256kb	256kb
L3 Cache size	Shared among cores	8Mb	4Mb

Table 1.1: cache sizes in Intel i7 processors

1.2 Motivation:

The processor performance has increased extensively compared to memory performances over the years [2]. The figure 1.2 depicts the performance gap between the two and the information shown in the figure is adapted from [7].

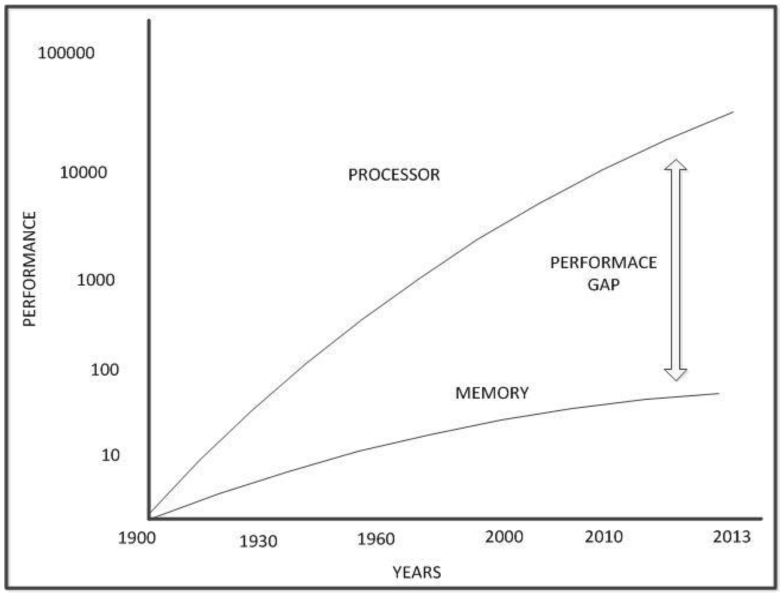


Figure 1.2 Performance gap between processor and memory

This performance gap between processor and memory calls for more research in the field of memory development. One of the leading memory compiler softwares '**Embed It'** was produced by VIRAGE Logic's corporation. The company was recently acquired by EDA giant Synopsys. All the memory compiler tools used by the physical design teams in the industries are proprietary. It cost thousands of dollars for academic institutions to obtain the software licenses to use these EDA tools in academic projects. Therefore it is not economically feasible for many of the academic institutions to use these softwares. The motivation behind this project is to provide free memory compiler for students to use in their academic projects. This memory compiler is aimed at producing a single port SRAM memory structure along with necessary components required to be incorporated in the design flow.

1.3 Thesis Organization:

The first chapter gives the brief introduction of cache memories. It also deals with the structure and importance of caches in modern processors and explains the die structure of Intel's i7 processor chip. The second chapter explains the hardware architecture of SRAM, describes each element in the SRAM and how they collectively operate to store and retrieve the binary information. The third chapter introduces the python programming language. It also presents the overview of memory implementation, GDSMILL python package, and explains the usage of the compiler. The fourth chapter explains the hierarchical decoder layout generation using the python scripts. It also explains the software implementation methodology, functional simulation of decoder and the integration of the hierarchical decoder into the memory architecture. The final chapter gives results, conclusion and future extensions of the memory compiler.

CHAPTER II

BACKGROUND

SRAM stands for static random access memory; as the name suggests it is the memory structure in which the data can be accessed randomly. The general architecture of the SRAM memory is shown in the Figure 2.1. The main components of the SRAM memory are memory array, decoders, pre-charge circuit, write driver circuit, sense amplifier circuit, column multiplexer circuit, data I/O output buffers and Control logic circuit.

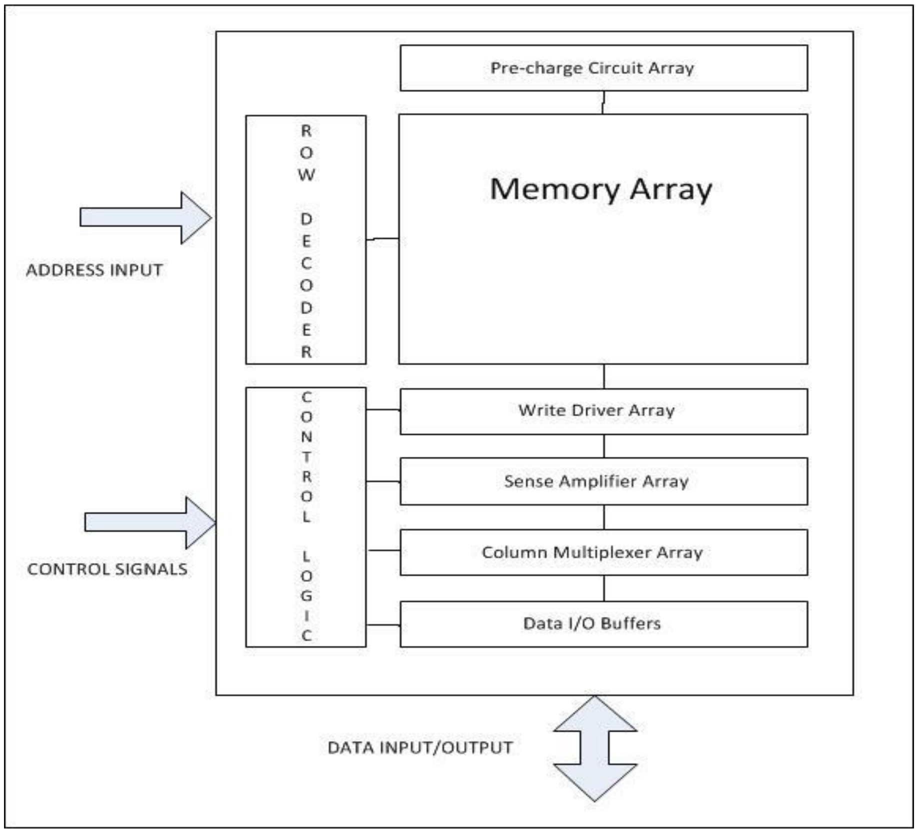


Figure 2.1: SRAM Memory Architecture

The major part of the memory architecture is composed of the memory array and rest is peripheral circuit. The 6T memory cells are arranged in rows adjacent to each other to form the memory blocks. These memory blocks are arranged one above the other to form the memory array. The peripheral circuit includes pre-charge circuit, row decoder, write drivers, sense amplifiers and control logic [1]. Each memory cell in the memory array is accessed for reading or writing by selecting the row. The memories can select 1, 4, 8, 16, 32 or 64 columns simultaneously depending on the column multiplexer incorporated in the memory.

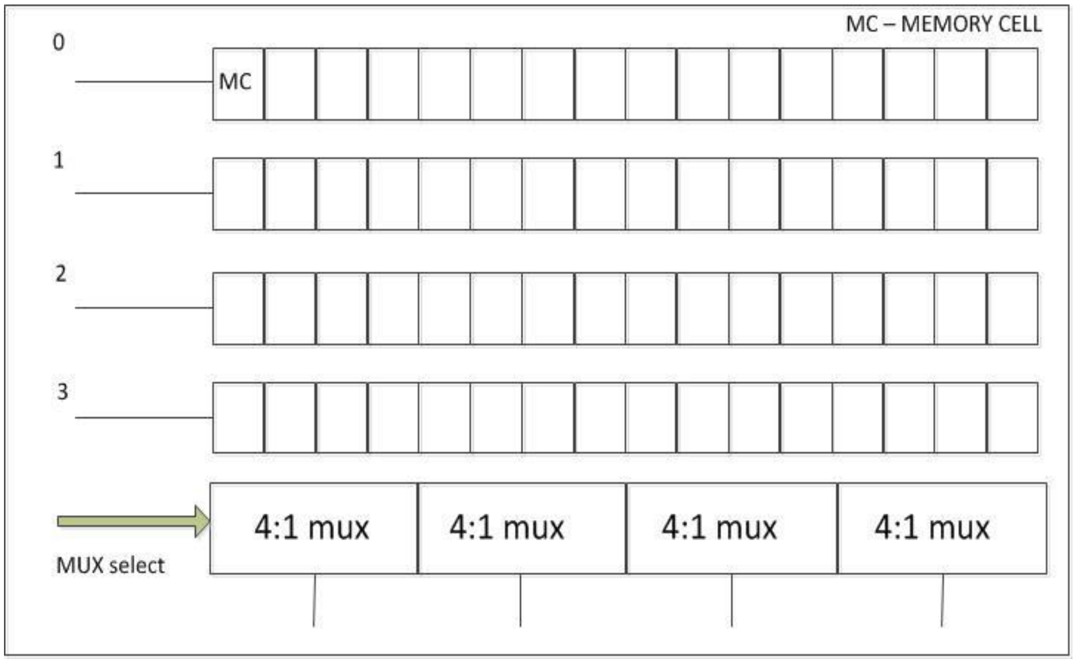


Figure 2.2 16 x 4 SRAM memory

The Figure 2.2 shows a simple 16 x 4 SRAM memory structure. The 16 memory cells in each row form one memory block and there are four such blocks. There is one 4:1 column multiplexer for four columns. Depending on the row address, one of the four rows will be enabled. In the enabled row, according to mux select code, 4 out of 16 cells are selected for reading or writing. Referring back to figure 2.1 the address signals are used to enable the required columns in the memory block. The control signals are used to perform the read or write operation on the memory block selected. Each component of memory structure is explained in detail in the following sections.

2.2 6T SRAM Memory Cell:

The 6 transistor memory cell is arranged as show in the Figure 2.3. The memory cell consists of two access transistors and a cross coupled inverter making up for the other four transistors.

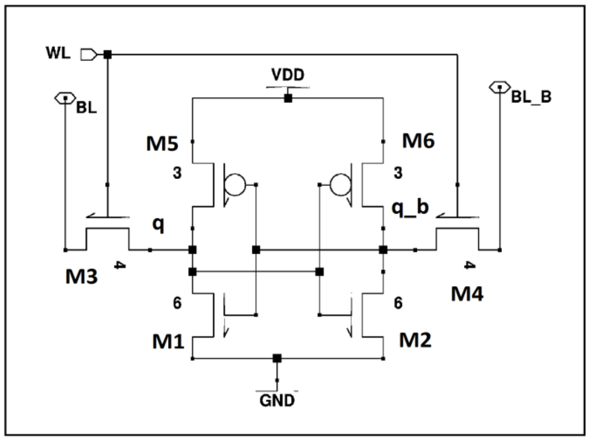


Figure 2.3 6 T SRAM Cell

The data will be transferred in and out of the memory cell using the long wires called bit lines. Each memory cell will have two bit lines holding the complementary data. In the Figure 2.3 the bit lines BL and BL_B are connected to the source terminals of the access transistors M3 and M4, respectively. Similarly the drain terminals of transistors M3 and M4 are connected to the internal nodes *q* and *q_b* of cross coupled INVERTERS, respectively. The gate terminals of the access transistors M3 and M4 are connected to the word line or WL. The data is stored in the cross coupled inverters in one of the two stable states either 0 or 1. The internal node *q* and *q_b* will always hold complementary data due to its cross coupling arrangement. The cross coupling arrangement improves the noise rejection and speed of the memory cell for memory access. The memory cell is read or written into by switching the word line ON [9].

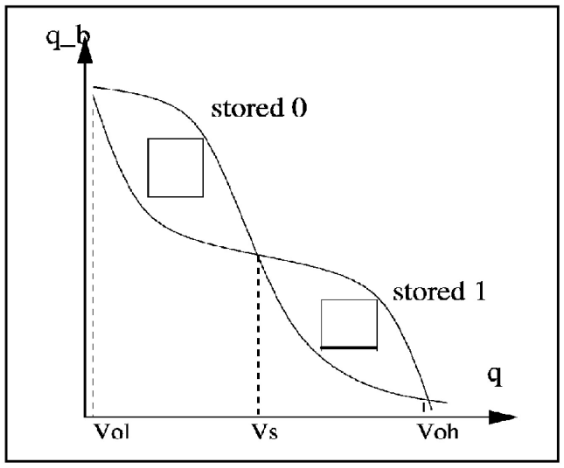


Figure 2.4 Voltage Transfer Characteristics

The VTC curve for the cross coupled inverters is shown in the Figure 2.4. It depicts the important cell design consideration for read and write operations. As stated before the internal nodes *q* and *q_b* will always hold the complementary data. The voltage transfer characteristics represents the two stable states of the cross coupled inverter. The memory cell will hold its data until one of the internal node crosses the switching threshold Vs. The write operation is performed by forcing one of the internal nodes to exceed switching voltage which flips the state of the cell and data is captured. It is crucial not to disturb the state of these internal nodes during the read operation. The VTC provides the required design margin for proper read and write operation of the memory cell. The mechanism of read and write operation is explained as follows [9].

Read Operation:

The read operation of memory cell is shown in the figure 2.5. Let us assume, 0 is stored at node *q* and 1 is stored at node *q_b*. The bit lines are pre charged to VDD before read operation is performed.

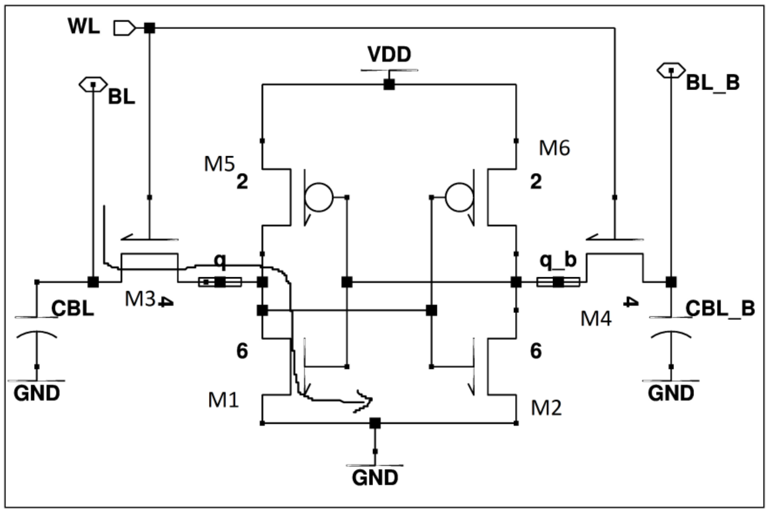


Figure 2.5 Read operation of memory cell

The word line is turned ON, and since 0 is stored at node q, M1 is ON and M2 is OFF, which opens up a path for bit line capacitance CBL to discharge to ground mean while the BL_B line will hold its value at VDD. The data stored in the memory cell will appear on the bit lines. In general bit line capacitance will be large because of multiple memory cells in a column connected to it and discharge of this capacitance will take a long time. Therefore, instead of discharging bit lines completely we use sense amplifiers. Sense amplifiers are differential amplifiers which determine the small voltage difference developed as a result of the read operation. This small voltage difference is amplified to get that data stored in the memory cell [9].

Write Operation:

The write operation is shown in the figure 2.6. As stated before the bit lines are pre charged to VDD. The bit lines are charged to obtain the correct data from the write drivers. The write driver circuit will drive one of the bit lines to VDD and other bit line to GND depending on the data input. Once the data is available on the bit lines, the word line is turned ON. Let the bit line BL be at VDD and bit line BL_B at GND. If the internal nodes *q* and *q_b* are also at the same voltages, the states of the cell will not be disturbed. The memory cell will retain the value previously present.

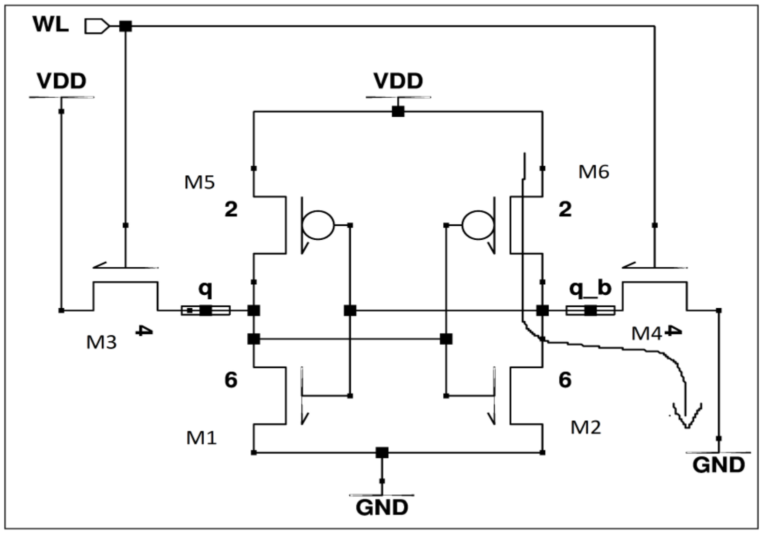


Figure 2.6 write operation of memory cell

If the internal nodes *q* and *q_b* are different from BL and BL_B, respectively then the bit line BL will rise the voltage of the node *q* to go above the switching threshold whereas the node *q_b* will be driven towards ground. The discharge of node *q_b* is shown in the Figure 2.6. This regenerative effect will cause the state of the memory cell to be flipped. The flipping of states of the internal nodes ensures that the data is written on the memory cell [9].

2.3 Address Decoder:

The circuit diagram of the 2:4 address decoders is show in the figure 2.7. The operation of the address decoder can be explained as follows. At any given point of time one of the word lines will be ON depending on the address input. For example, if the address input A0A1 is 00 then the output W0W1W2W3 would be 1000. In the case of the memory, the address decoder plays a prominent role in turning word line ON or OFF depending on the address inputs applied. From the previous section we can see that the word line is connected to the access transistor of the memory cell. Depending on the number of cells in each row, the load of the word line varies and hence the drive capability has to be changed accordingly. However, in our design we have chosen the NAND gates and INVERTERS to have equal rise and fall delays.

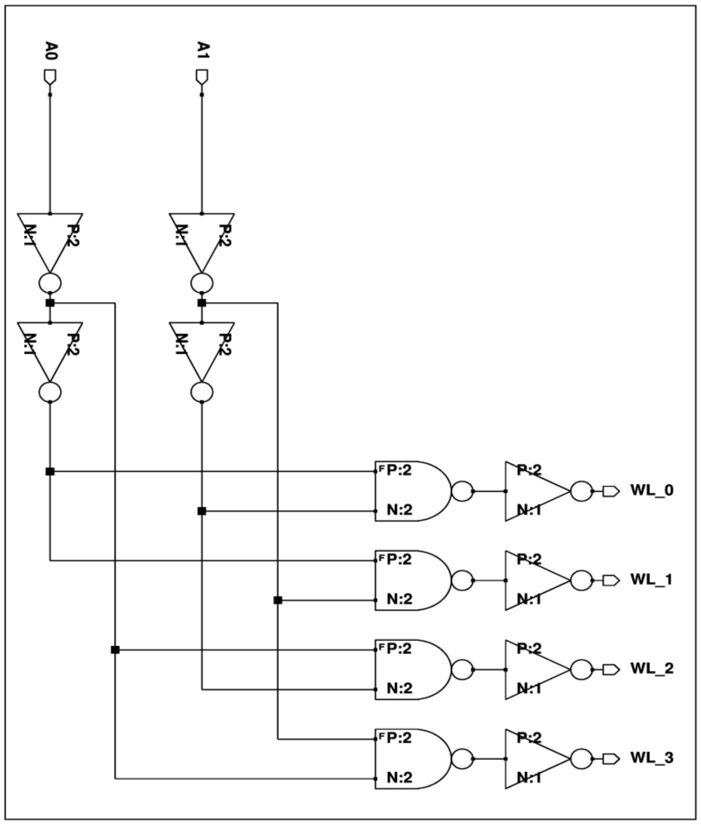


Figure 2.7 2 to 4 address decoder

As the decoder size increases, the size of the NAND gates used for driving the word line increases. In general, an n bit decoder requires the 2^n NAND gates each with n inputs [1]. In the case of 2:4 address decoder shown in the Figure 2.7, there are four, two input NAND gates. Similarly in 3:8 address decoder we need eight, three input NAND gates. As the size of the memory array increases the number of word lines increase, which in turn will increase the size of the row decoder. In order to reduce the size of the NAND gates we go for the Pre and final decoding methods for row decoding. The method of using pre and final decoding for a row decoder is explained in chapter 4. As the numbers of inputs of NAND gates exceed four , the gates become slow due to the series stack of the nmos devices. This can be solved by constructing a larger gate with smaller gates.

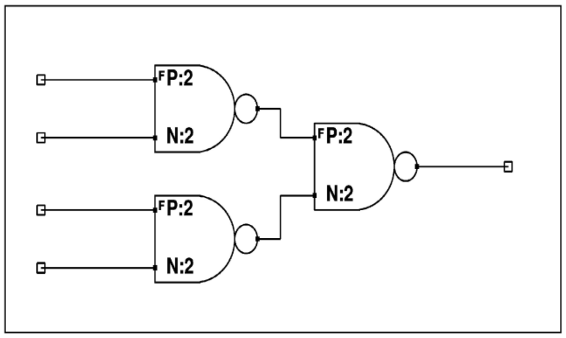


Figure 2.8 4 INPUT NAND GATE FROM 2 INPUT GATES

A 4 input NAND gate can be constructed using the three 2 input NAND gates as shown in the Figure 2.8. The address input is pre decoded before the final decode which solves the above mentioned problem. This arrangement will reduce large delays caused due to large NAND gates.

2.4 Pre-Charge Circuit:

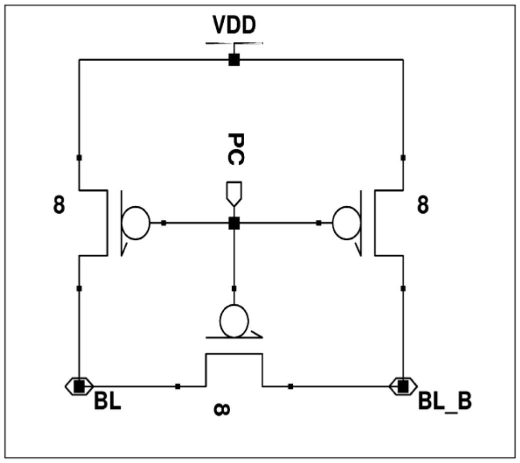


Figure 2.9 Pre – Charge Circuit

The circuit diagram of the pre charge and equalize circuit is shown in the above Figure2.9. As we know, before the read or write operation is performed the bit lines have to be pre-charged to

VDD. This is performed in order to ensure that the correct data is read from or written to the memory cells. The circuit used for the pre-charge operation is called the column pull-up circuit or pre charge circuit. It consists of three PMOS transistors in which two transistors are used for pulling up the bit lines BL and BL_B to VDD. The third transistor is used to equalize the voltage of the charged bit lines. It is necessary to equalize the bit line voltages after charging, so as to achieve the required voltage swing during the read operation. The gate terminals of all the three transistors are connected to the PC or pre charge clock. All the transistors are simultaneously activated by the PC signal before the read or write operation, which causes the bit lines to be charged to VDD.

2.5 Write Driver:

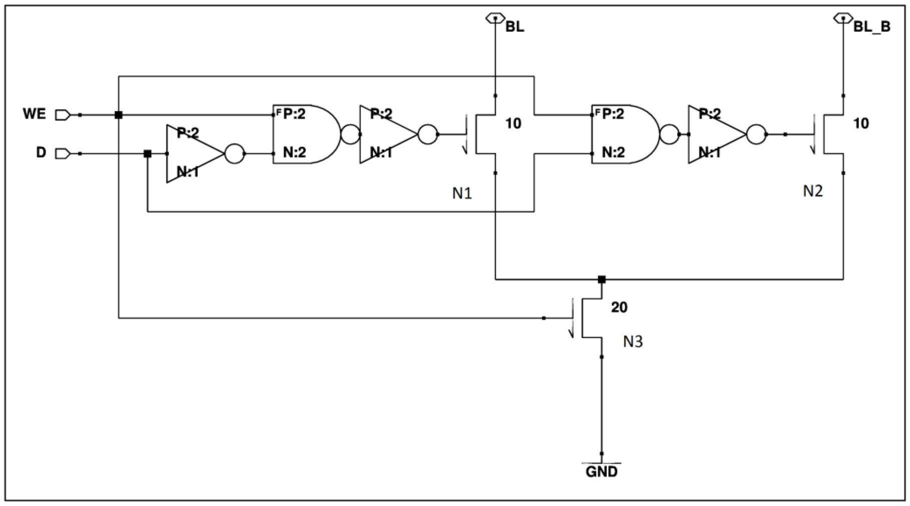


Figure 3.0 Write Driver Circuit

There are many ways to construct the write driver circuit depending on the power and speed tradeoffs. The write driver circuit used in our design is shown in the above Figure 3.0. The circuit consists of two NAND gates, three INVERTERS, and three large NMOS transistors. The operation of the write driver circuit can be explained as follows: If we assume data input is 0 and WE signal to be 1, then from the circuit diagram shown, the N1 and N3 transistor are turned ON.

The voltage of bit line BL will be discharged to the GND, while the N2 transistor is OFF and this makes the bit line BL_B to hold its value at VDD. During the read operation, the WE signal is 0 hence all the three NMOS transistors are OFF. This disables the write driver circuit during the read operation.

2.7 Sense Amplifier:

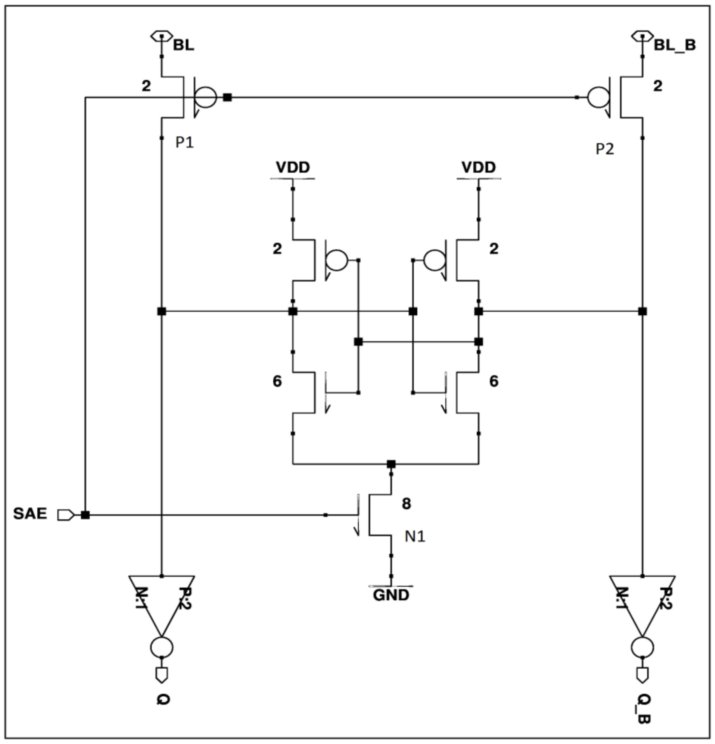


Figure 3.1 Latch Type Sense Amplifier Circuit

In memory architectures, designing a sense amplifier is a challenging task. There are two different types of sense amplifiers. One is the current mirror type and the other is latch type amplifier which is the constructed using the cross coupled inverter configuration. Basically, the sense amplifiers are nothing but differential amplifiers which detect the small change in voltage difference between the bit lines. It amplifies the detected voltage difference to produce the data output. These amplifiers are selected based on the speed and noise cancellation requirements. The current mirror sense amplifier is more susceptible to noise on the bit lines. In the current mirror type amplifier, the bit lines have to be charged at all times for performing read and write operations which consumes more power. The latch type sense amplifier is not only less susceptible to noise on the bit lines, but also consumes less power due to a self-timed mechanism which turns the sense amplifier ON only when required. Hence the power consumption of the latch type sense amplifier is much less compared to that of a current mirror type. The Figure 3.1 shows the latch type sense amplifier circuit. The operation of the sense amplifier can be explained as follows: After the word line is turned ON, a voltage difference is developed between the bit lines BL and BL_B. This voltage difference developed will store the data in the cross coupled inverter. Once the SAE signal is turned ON, the large capacitance of bit lines is isolated with the help of PMOS transistors P1 and P2. After isolation, Q and Q_B lines will produce the output according to the data stored in the cross coupled inverter. As we see the bit lines are not completely discharged, but instead only the required voltage difference determined by the sense amplifier is used to perform the read operation.

2.8 Column Selection:

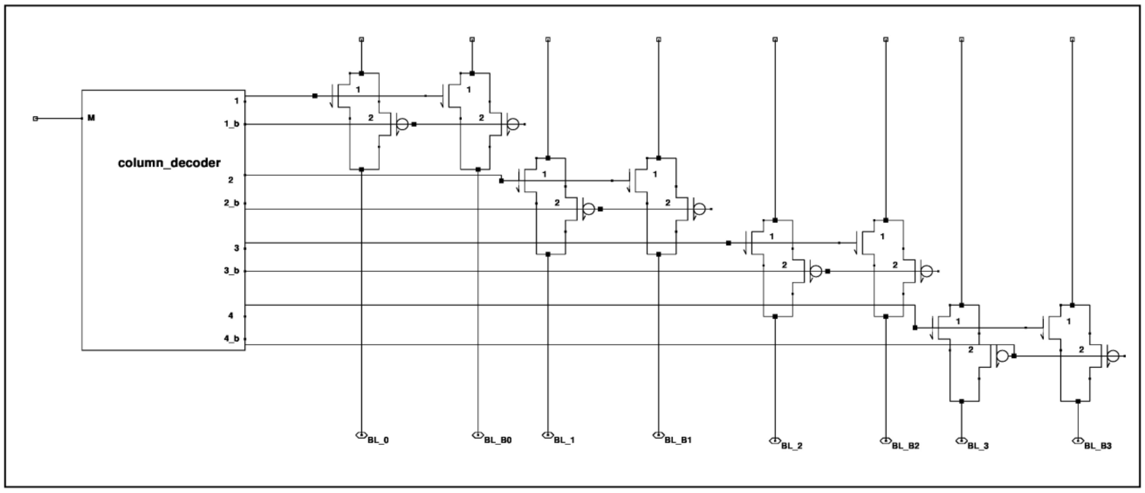


Figure 3.2 Column Selection Circuit

The column selection circuit is shown in the figure 3.2. The column selection circuit consists of a column decoder and transmission gates connecting to each of the bit lines as shown in the above figure 3.2. The column decoder decodes the M address input signals into 2^M true and complementary output signals. Each complementary pair of 2^M signals are connected to the transmission gates of bit lines BL and BL_B. The use of transmission gates ensures a good 0 and 1 data transfer without any threshold voltage degradation. Similar to the row decoder the column decoder size increases as the column multiplexer size increases. The bit lines from the transmission gates are connected to both write drivers and sense amplifiers. Depending on the address input signal, the column selection circuit selects the required columns on which read or write operations are performed. A read operation is performed by sense amplifiers and a write operation is performed by the write drivers.

2.9.1 Memory read:

The act of retrieving the data stored in the memory cells is termed as the memory read. The read operation involves the following steps:

1. The address and control signals will be introduced on their respective pins. Since it is a read operation the data inputs are not used.
2. Once the control signals are applied, irrespective of the operation, the pre-charge clock will charge the bit lines to VDD.
3. The address signal is decoded by the row decoder and the column selection circuit which activates the word line and the required columns, respectively.
4. Once the word line is enabled, all the memory cells in that row will be activated. Depending on the data stored, the memory cells will either discharge one of the bit lines to ground or they will hold the value of the other bit line at VDD.

5. As one of the bit line discharges, a voltage difference is developed between the bit lines. The sense amplifier is designed with the specific voltage difference in mind. As soon as the bit lines achieve this voltage swing, the SAE signal will be turned on.
6. As soon as the SAE signal is activated, the data that was stored in the latch type sense amplifier during the read operation is retrieved on the output data bus.
7. The contents of the data bus are transferred to the I/O buffers.

2.9.2 Memory Write:

The act of storing the data into the memory cells is termed memory write operation. The operation is explained in details as follows:

1. The address, control and data signals are applied on the respective pins.
2. The control logic will generate the PC clock which will pre charges the bit lines to VDD before the read or write operation.
3. The address signal will activate the word line and required memory columns.
4. The write driver circuit will drive one of the bit lines to VDD and other bit line to GND depending on the data input. Once the data is available on the bit lines, the word line is turned ON.
5. If the data to be written is same as the value present in the memory cell, the state of the memory cell will remain unchanged.
6. On the other hand, if the data to be stored is different, the bit line holding its value at VDD will drive the internal node of cross coupled inverter to VDD while the other node will be pulled down towards GND.
7. This regenerative effect occurring on the internal nodes of cross coupled inverter will flip the states of the memory cells thereby storing the required data.

8. When the memory is not read or being written it will be in idle state. The memory cells will retain the values stored as long as the power is ON.

CHAPTER III

METHODOLOGY

This chapter explains the methodology used for the design and implementation of the memory compiler in detail. The compiler is constructed based on an high level interpreted language called Python. Python is one of a best open source interpreted languages among many others. A python program is not compiled instead it is interpreted. The open source community of python is extensive; hence the library support obtained is enormous. Python community has also provided the PyDev plugin which makes python to be used with eclipse which is a Java IDE. It is a dynamic object oriented programming language which is used for various applications. The list of major applications in which python can be used is mentioned as follows, rapid GUI development, web development, scripts for search engine back bone, scripting for EDA tools and many more. Hence all this collectively make python a powerful language.

This is also the reason why we chose this language to implement the compiler. For our project editors available in the Linux environment were used to construct the source modules. This method is suitable because all the supporting EDA tools run on the Linux platform.

This makes it easy to drive these tools using scripts for construction, implementation, simulation and verification. Apart from the above mentioned advantages, python is a user friendly language. Since it is not strongly typed, we don’t have to declare the data types which make it less verbose. This reduces the programming time compared to other verbose languages like C++ or Java. The disadvantage of using python is its large execution time compared to other high level languages.

3.2 Overview of Memory Compiler Implementation

This section explains the compiler implementation in detail. The figure 3.3 shown below, gives a brief idea of inputs given to the compiler and outputs generated from the compiler.

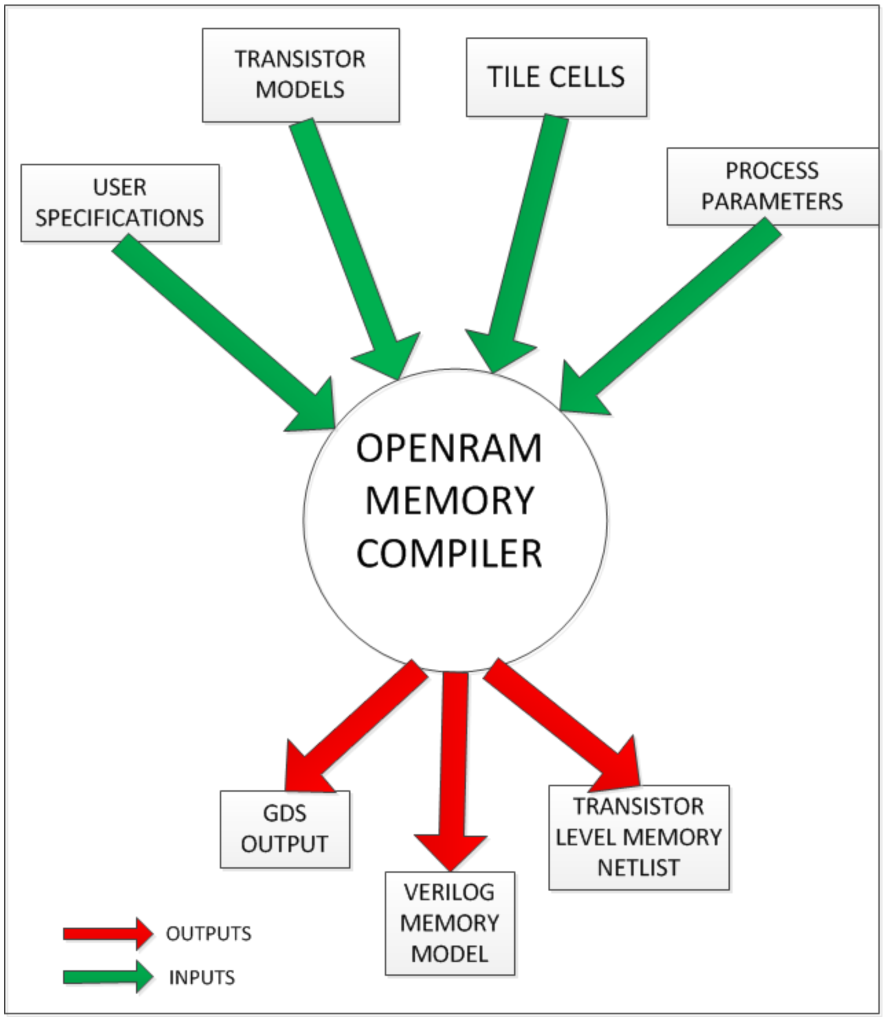


Figure 3.3 OPENRAM Memory Compilers

The green arrows in the figure 3.3 represent the inputs provided to the compiler while the red arrows represent the outputs obtained. Each of the inputs given to the compiler is described in details in the following sections.

3.2.1 User Specifications

The memory compiler is a combined group of software scripts, which generates the Memory layouts according to the specifications provided by the user. The prominent specifications in general for any memory compiler are required 'number of memory words' and the 'size of this memory word in bits'. For commercial memory compilers we have to provide the column mux option Depending on the column mux selected it will arrange the memory words to have the aspect ratio close to one. Aspect ratio is the ratio of vertical size to horizontal size of memory.

The specifications required by the OPENRAM memory compiler are listed as follows

1. Number of words
2. Word size in bits
3. Name of the output memory file

The compiler takes 'number of words' and 'word size in bits' as parameters and then does internal calculations for the arrangement of memory array. For example, if the number of words are 32 and the word size is 8, it will generate a memory array of 16 rows with each row having 2 words of size 8 bits forming 16 columns. The column mux is always selected as 2:1, i.e. one of the two columns connected to the column mux will be selected during the read or write operation. Always, the compiler will produce the near squared arrangement of memory array for the given parameters.

3.2.2 Transistor Models

Each technology has its own models for its transistors. A model is nothing but the process parameters obtained by fabricating the transistors in that technology and then analyzing for various on-chip parameters. The OPENRAM memory compiler uses the FreePDK 45 nanometer process. As we know, this process cannot be fabricated since it is constructed based on a fake technology model. The model for this process is obtained from Berkley Predictive technology

Model [BPTM]. The models of the transistors are predicted based on the analysis conducted on higher nanometer practical transistor [5].

3.3.3 Tile Cells

Tile cells are the important components which are used to form the SRAM memory structure. The tile cells used for the OPENRAM are constructed from the technology libraries provided by North Carolina State University [NCSU] for 45 Nanometer Process. The basic idea of memory compilers is to construct single tile cells for each component of the memory architecture. Once the tiles cells are constructed, a software program can be used for arraying the tile cells and combining these arrayed components to form the memory structure. In SRAM memory, significant amount of space is occupied by the arrays of 6 transistor memory cells used for storing binary data. The memory blocks are formed by arranging memory cells adjacent to each other. The memory array is formed by arranging the memory blocks one above the other. The arrangement of the memory array is carried out by the software script using the single 6T memory tile cell. Similarly, we can use programs to arrange the pre charge circuits, column multiplexers, sense amplifiers, and write drivers. Finally using a top level software script we combine these arranged layouts to from the complete memory structure. All the tiles cells used for the OPENRAM compiler were constructed in Cadence Virtuoso layout editor.

The tiles cells used in the design are listed as follows.

1. 6 transistor memory cell
2. Pre charge cell
3. Sense amplifier cell
4. Write driver cell
5. And Other Logic cells

Memory cell

The tile cells are constructed using NCSU Free Process Design Kit in Cadence Virtuoso layout editor [8]. The schematic of the 6 transistor memory cell is constructed according to the calculated sizes of transistors for good read and write stability as explained in the chapter 2. Once this is performed, the layout is constructed and verified for DRC errors. If the layout is DRC clean, the layout versus schematic (LVS) check is done using Calibre EDA physical verification tool embedded with in Cadence Virtuoso layout editor. The layout of the 6 transistor memory cell is shown in the figure 3.4. Once the cell is constructed and physically verified we simulate it for functional correctness.

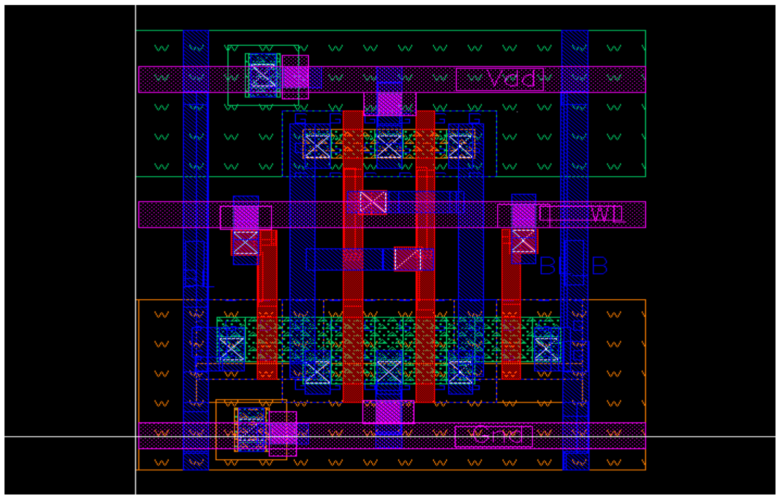


Figure 3.4 Layout of 6 transistor memory cell

Pre- charge cell

The pre-charge tile cell layout is shown in the figure 3.5. The construction is similar to that of the SRAM cells. The extracted net list was simulated using the Hspice simulation tool.

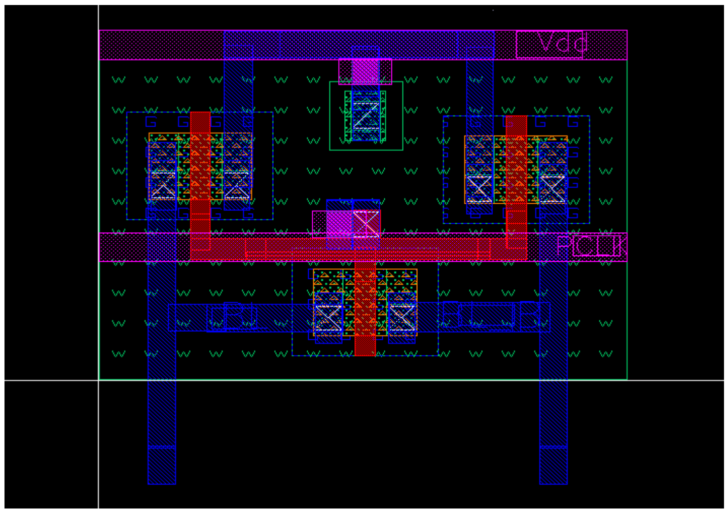


Figure 3.5 Layout of Pre Charge Circuit

Write driver cell

The layout of the write driver is shown in the figure 3.6. The cell used for writing the data to the memory cell is write driver circuit.

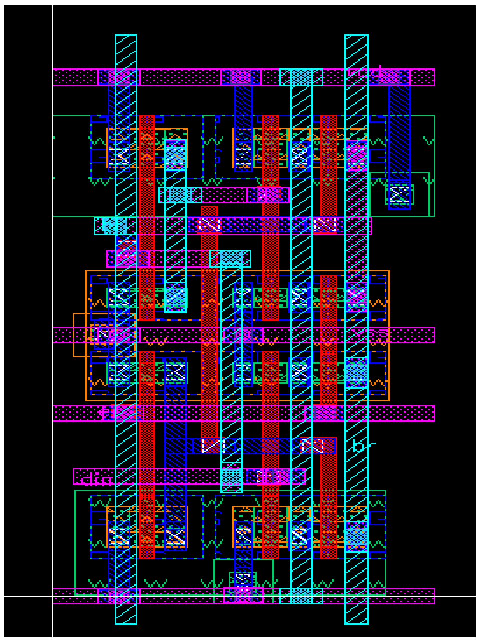


Figure 3.6 Layout Write driver cell

Sense amplifier cell

The figure 3.7 shows the layout of the latch type sense amplifier constructed using the NCSU 45 nanometer process technology libraries. Sense amplifiers are the important circuit element used for the read operation. They are used to reduce the read access time by not letting the memory cell completely discharge the bit lines.

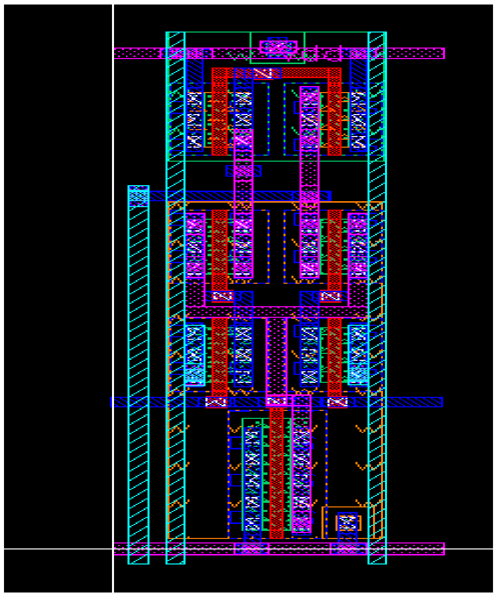


Figure 3.7 Layout of the Sense amplifier

Logic cells

The logic cells like INVERTER and NAND gates are used in the construction of the decoders and control logic of the memory circuit as shown in the figures 3.8 and 3.9, respectively. The construction of the logic cells is similar to the SRAM cell. The simulation of the logic cells were performed with the Nanosim simulation tool.

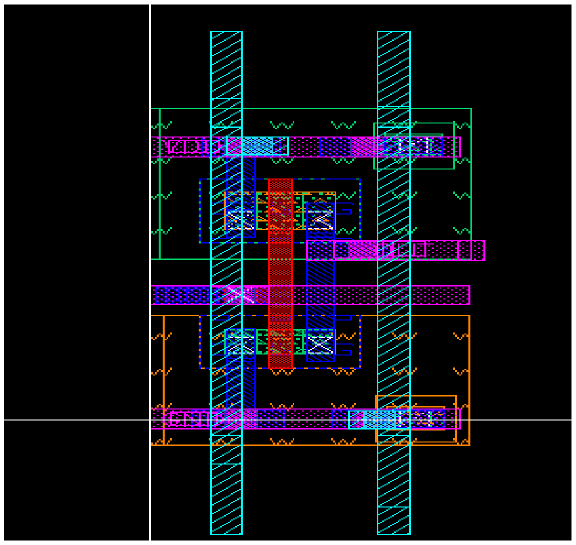


Figure 3.8 Layout of inverter

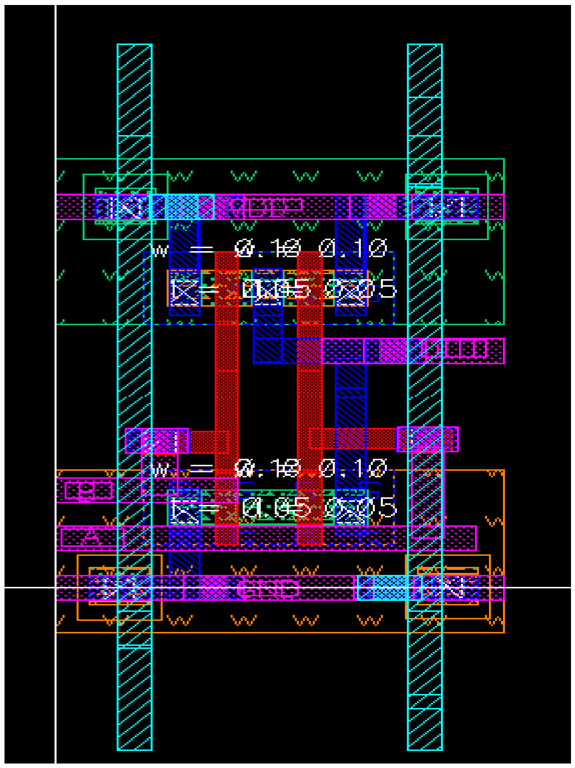


Figure 3.9 Layout of NAND gate

3.2.4 Process parameters

Each process or technology used for chip fabrication will have its own DRC rules for the construction of the layout. The DRC rules for chosen technology will be provided by the vendor. MOSIS is the fabrication facility which provides design rules for chip design industries. The DRC rules are minimum specifications for the layer used in the technology. If the designer tries to design the layout below this minimum specification, the designed information cannot be fabricated as it's a violation of the provided rule for a particular technology [4].The process parameters or DRC rule file for this project is provided by NCSU's technology library which is used in conjunction with the CALIBRE tool for physical verification. The following section explains in detail the contents of PDK provided by NCSU.

3.3 NCSU 45 Nanometer Process Design Kit

The tile required for the memory circuit construction is built in NCSU FreePDK45 process. It is an open source, variation aware Process Design Kit (PDK) based on the CMOS scalable design rules. This process is based on BSIM4 predictive technology model. BSIM is an abbreviation for Berkeley short channel IGFET (insulated gate field effect transistor) model [4]. This process gives us the physical characteristics of a metal oxide field effect transistor (MOSFET) in nanometers.

The process design kit includes the following:

1. Technology library (Transistor Models).

2. Tech files and display resources for cadence layout editor.

3. Design rules compatible with Calibre.

4. Standard cell library.

The NCSU FreePDK is a twin well process having a poly and 10 metal layers. This process is variation aware, it contains the models for devices that could be used to research or understand short channel effects in lower technologies [4]. The industrial process design kits have IP restrictions on using them for academic purposes and for classroom instruction. Hence FREEPDK resolves the problem by being open source. It is also easy to maintain with the included setup scripts. The kit can be setup easily by changing the environmental variables of the local machine. The design kit can be downloaded by typing the following command at the user's Linux Terminal:

- ⇨ *svn co* https://svn.unity.ncsu.edu/osi/freepdk45/trunk

This will download the kit in to user's current working directory. However, compiling the python code in NCSU library for obtaining Parametric transistor cells [PCELLS] is quite complex. This requires many tools to work together in the linux environment. Setting up the environmental variables for all the tools used is difficult. The installation tutorial explains the procedure in detail to obtain the PCELLS which is attached in Appendix2.

3.4 Supporting EDA tools set

This section explains the different EDA tool sets used in the flow of the memory compiler. We used different sets of EDA tools in different stages of the design. The design tools used are listed as follows

EDA tool vendor	**Name of the tool**	**Used For**
Cadence	Virtuoso Editor	Constructing tile cells
Mentor	Calibre	DRC LVS or Physical Verification

Synopsys	Hspice	Functional Verification
Synopsys	Nanosim	Functional Verification
Open Source	Layout Viewer	GDS layout Viewer

Table 2: EDA tool set

As previously stated all the tile cells required for implementation of SRAM design were constructed using Cadence Virtuoso layout editor using the NCSU FreePDK library. Once the tiles cells were created, it was verified for DRC and LVS checks using Calibre tool embedded with Virtuoso layout editor. One of the python modules was created specifically for DRC and LVS checks required during SRAM memory assembly. If tile cells designed passes these tests then, we move forward to perform simulations for functional correctness using Nanosim. The detailed procedure for performing simulations using Nanosim is explained in the next chapter.

3.5 GDSMILL Python Package

GDSMILL is a python package used for reading, writing and manipulating the GDS II files. GDS stands for Graphic Database Standard. In the field of VLSI the design information is sent to the fabrication facility in this file format. Using this package we can convert the GDS file data into software objects. The software objects created can be manipulated as any other data structures and the manipulation performed can be written back into the GDS file format. Hence this package is very useful in creating the layout structures.

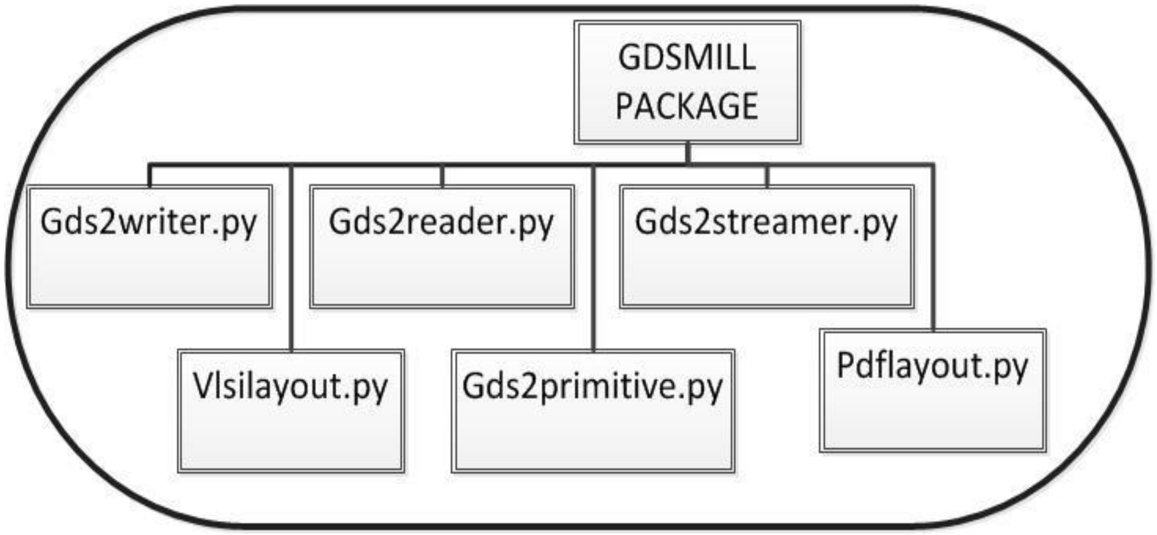


Figure 4.0 GDSMILL Python Package

GDSMILL package provides different modules for different tasks. The contents of the GDSMILL package are shown in the figure 4.0. *Gds2reader.py* and *gds2writer.py* contains the python classes with the same name and provides all the necessary functions required for reading and writing the GDS files. These classes processes the data by iterating through each record in the GDS structure and check or write every data record. The record type is tracked and identified using flags. *Vlsilayout.py* and *gds2primitives.py* are used to create the software data structures for the layouts. After the gds2reader class reads in the records, the data has to be stored in a way that can be easily used. Vlsilayout class represents the layout and it stores the records in data structures, which are defined by the *gdsPrimitive.py* module. The vlsilayout contains many functions for adding structures and recording the layouts but important functions have been wrapped in a file called *geometry.py* which acts as the interface between the GDSMILL and OPENRAM compiler. *Pdflayout.py* module is used for creating a pdf file from respective GDS file. The *Gds2stremer.py* module provides the functions necessary for streaming the GDS file in and out of the Cadence layout editor. The standard format for storing the layout data in cadence is open access. Therfore, when the layout from cadence is streamed out, we obtain the layout in GDS file format. When we stream the layout into the layout editor, the GDS data will be converted to the open access format [11].

3.6 Python Scripts Organization

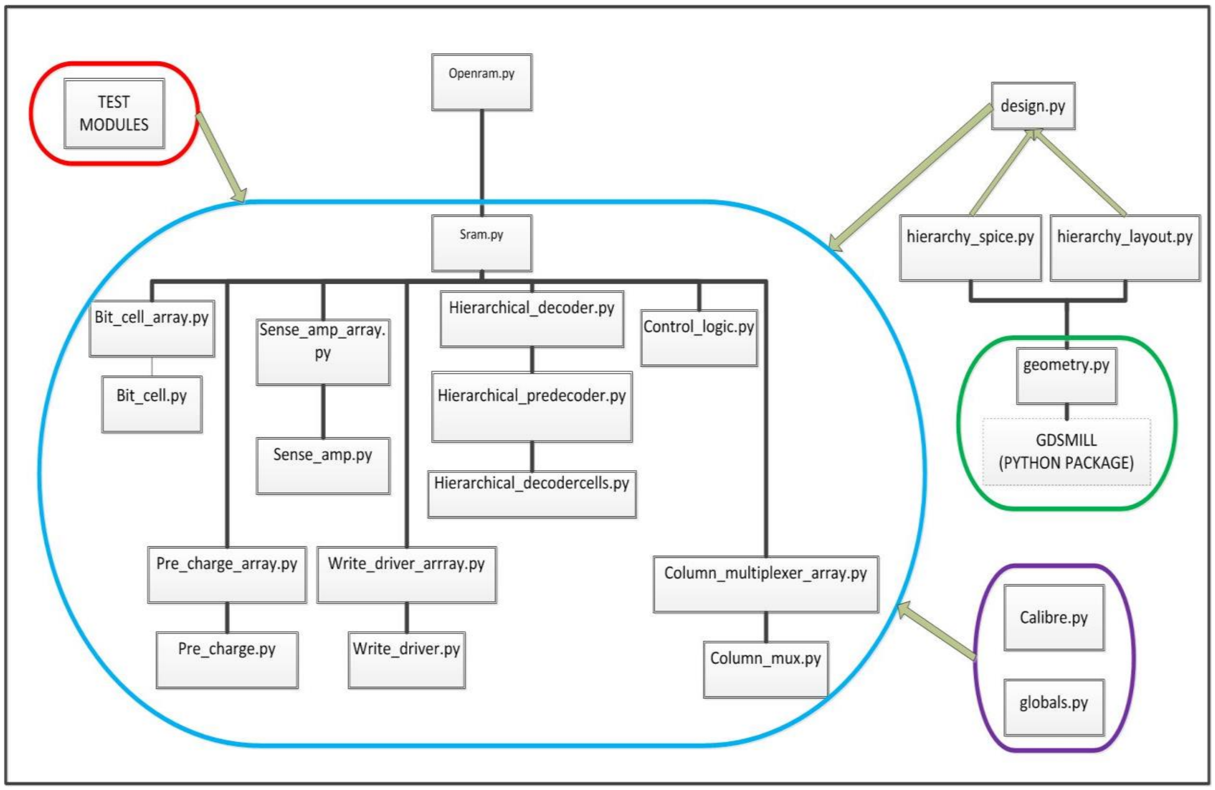


Figure 4.1 Python scripts arrangement

The organization of the python scripts in the compiler is shown in the figure 4.1. The compiler's main script starts from the module *openram.py*. The details of using the compiler script are explained in the next section. All the scripts present in the blue eclipse are used for the construction of the layout structures. The two scripts in the purple oval, calibre.py contain the necessary functions for physical verification of the layout created by the scripts in the blue oval. The *globals.py* provides the necessary global paths for the writing the temporary intermediate files generated during the compilation process respectively. *Calibre.py* modules acts as the

DRC_LVS interface and used to run the verification in batch mode. These scripts in the purple oval can be called from any modules to accomplish the task of physical verification or to access a global path. During the construction of the layout structure, these functions are called from all the modules in the blue oval for physical verification. As stated before *geometry.py* is the interface module created to communicate between the GDSMILL python package and OPENRAM. It also acts as a wrapper for using the required functions like *add_inst(), add_mod(), add_rect(), add_lable()* in *vlsilayout.py* [10]. The design hierarchy of layout and schematic or spice deck is controlled by the *design.py* module and it inherits two modules called *hierarchy_spice.py* and *hierarchy_layout.py*. The *hierarchy_spice.py* and *hierarchy_layout.py* modules control or store the data structures of the spice file and GDS files respectively. The abstraction of the python GDSMILL package is explained in software implementation section of next chapter.

3.6.1Test modules

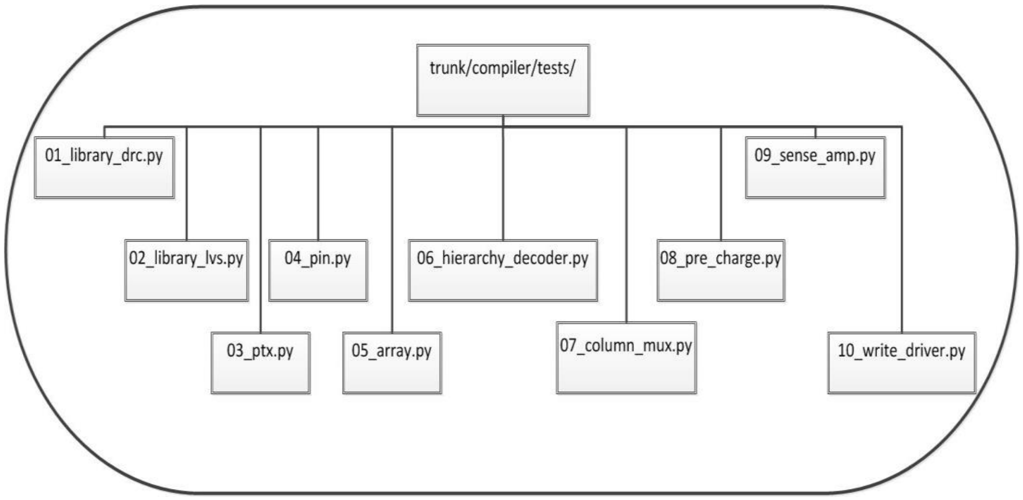


Figure 4.2 Test Modules

The red oval at the top left corner shown in the Figure 4.1 represents the test modules. Each compiler module used for generation of the layout has to be tested for the DRC and LVS. Hence we have constructed the test modules for each of the layout generating modules shown in the blue oval of the figure 4.1. Each module in the blue oval has its corresponding test module in the tests directory. For example, the *bit_cell_array.py* module generates the memory array layout and its respective spice file. In the tests directory its corresponding test is *05_array.py*. This module will run the *bit_cell_array* class as the main program, which initiates the layout and spice file generation of the memory array only. By calling the DRC_LVS check function of *calibre.py* module in *05_arrar.py,* we not only create the layout and spice for memory array but also verify them for physical correctness. This way all the modules are tested for DRC and LVS checks before they are collectively verified for full memory. This kind of ground up verification and build up will provide us a robust memory structure.

3.7 Using the Compiler

The first step before using the compiler is to set the required environmental variable on the system. The list of environmental variables is as follows OPENRAM_HOME

1. OPENRAM_TECH
2. DRCLVS_HOME
3. PYTHONPATH
4. CALIBRE

The setup script used by OSU computer architecture group is represented in the Appendix1. The OPENRAM_HOME environmental variable sets the path for the compilers trunk directory where all the scripts for the compiler are present. OPENRAM_TECH will point to the trunk/freepdk45/ directory which contains all the library cells, tech files, and layer map. DRCLVS_HOME will set the path for calibre design rules required for 45 nanometer process. GDSMILL is the python package used and hence its directory path has to be attached to the existing PYTHONPATH. CALIBRE variable of calibre physical verification tool will be set to a point, which will be called in batch mode for verification during compilation. We will define all the above environmental variables in a file called *pycali.cshrc* which also includes the environmental variable for python interpreter. Finally we source this file by typing the following command.

⇨ *source pycali.cshrc*

This will set all the environmental variable's required by the compiler. In the same directory where this file was sourced we can run the compiler by typing the command.

⇨ *python openram.py* –n 32 8 -o sram32x8

openram.py is the top level module which accepts the inputs and parses it to calculate the sizes of different components and produces the output. After typing this command the compiler will produce the banner of OPENRAM and finally when the compilation is complete it produces two files *sram32x8.gds* and *sram32x8.sp* which represents the layout and spice file of the required memory macro respectively.

CHAPTER IV

HIERARCHICAL PRE DECODER/DECODER IMPLEMENTATION IN OPENRAM MEMORY COMPILER

Hierarchical decoder is a type of decoder in which the construction is carried out hierarchically. The simple 2:4 decoder is shown in the figure 4.3. The operation of the address decoder can be explained as follows. After the address signals A0 and A1 are applied on the address lines, one of the word lines will go high after a brief amount of time [3]. The word line which will go high depends on the signal combination of the address input. For example, if the address input A0A1 is 00, the output W0W1W2W3 is 1000. The table 4.1 gives the output of the address decoder for different combinations of the input. The address decoder uses INVERTER gates and two input NAND gates for its construction. The INVERTER and NAND gates used are sized to have equal rise and fall times. As the decoder size increases, the size of the NAND gates required for decoding also increases.

Asserted ADDRESS Signal A[1:0](bin)	Word line OUTPUT WL[3:0](bin)
00	0001
01	0010
10	0100
11	1000

Table 4.1 2:4 ADDRESS INPUT OUTPUT

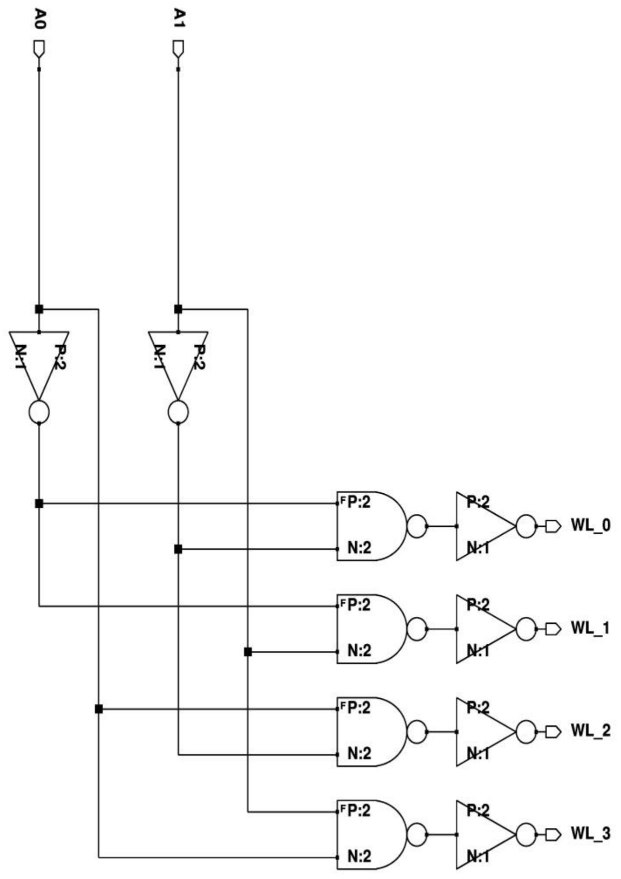


Figure 4.3 2:4 Address decoder

The layout of the 2:4 address decoder equivalent to the above shown schematic is shown in the figure 4.4. We require 3 input NAND gates for decoding 3 address lines . As the number of address lines increase, the input size of the NAND gates also increases. The NAND gates with more than three inputs will have large pull down delay because of the series stack of nmos transistors in pull down circuit. Hence we cannot use the NAND gates beyond three input since it takes vast amount of time to activate the wordline which increases the access time for reading and writing [3]. We can solve this problem by adopting the heirarchical decoder structure. The Figure 4.5 shows the schematic of 4 to 16 heirarchical decoder.The structure of the decoder consists of two 2:4 decoders for predecoding the address lines and 16 two input NAND gates and INVERTERS for final decoding completing the 4:16 decoder structure.

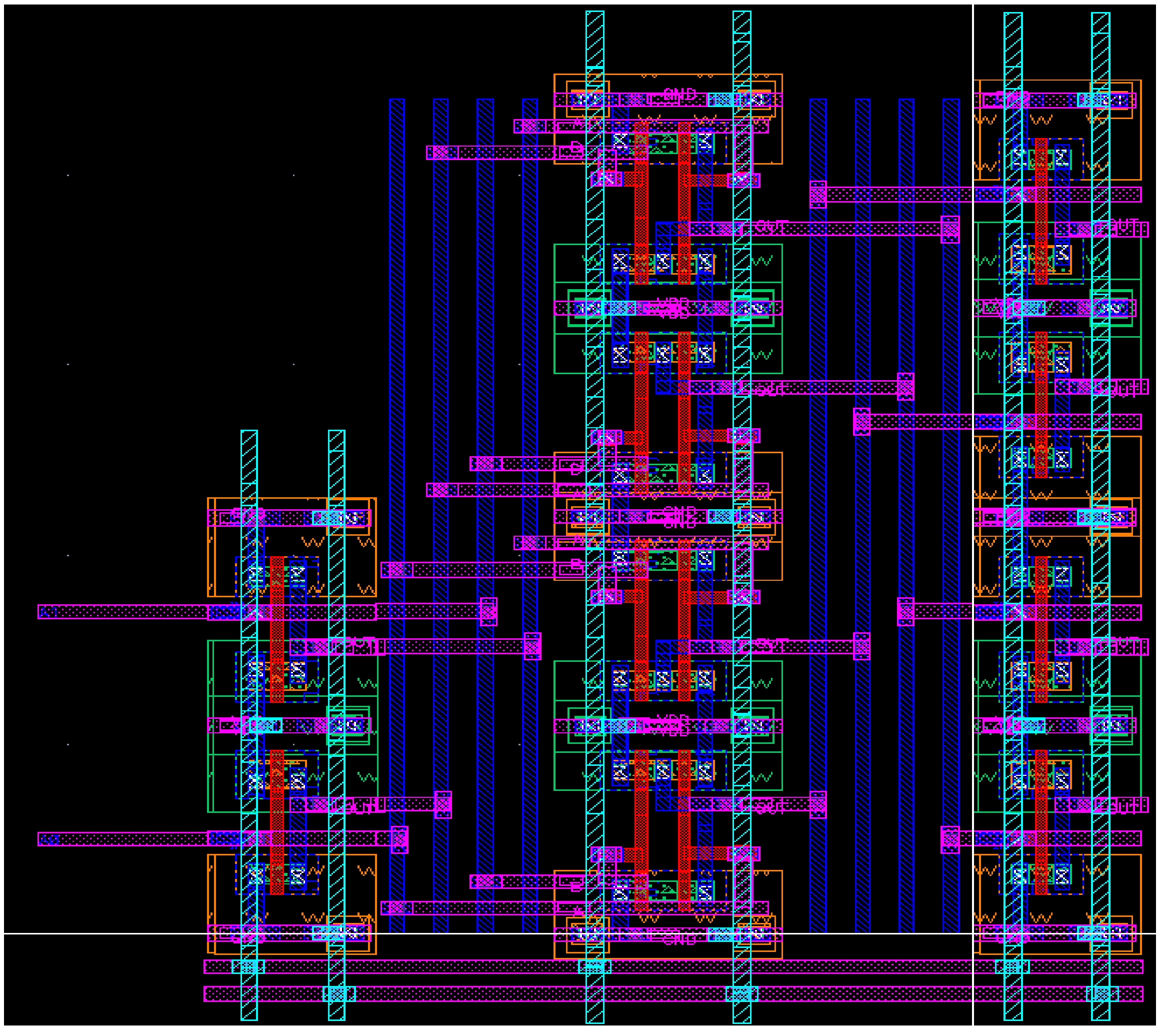


Figure 4.4 2:4 address decoder layout

In the figure 4.5 it can be seen that the heirarchical decoder uses the concept of predecoding and final decoding for its implementation. The construction of the heirarchical address decoder is very productive because of the small 2:4 decoders used for predecoding. The signals predecoded are given to the final decoding stage consisting of two input NAND and INVERTER gates. This will eliminate the need for constructing decoders with large NAND gates, which is a slower design. The operation of heirarchical decoder can be explained as follows. Let us consider the

Figure 4.5. If the address signal A0A1A2A3 is 0000 the output of the predecoder1 WL0WL1WL2WL3 is 1000 and predecoder2 WL0WL1WL2WL3 will be 1000 respectively. As we see from the figure 4.5 the zeroth wordline of predecoder1 and predecoder2 (PR1)WL0 ,(PR2)WL0 are both 1. These wordlines are conneted to the two inputs of the first NAND gate which in turn is connected to INVERTER. The output of the zeroth wordline of the final decoding stage will be 1 and rest of the wordlines will be 0.

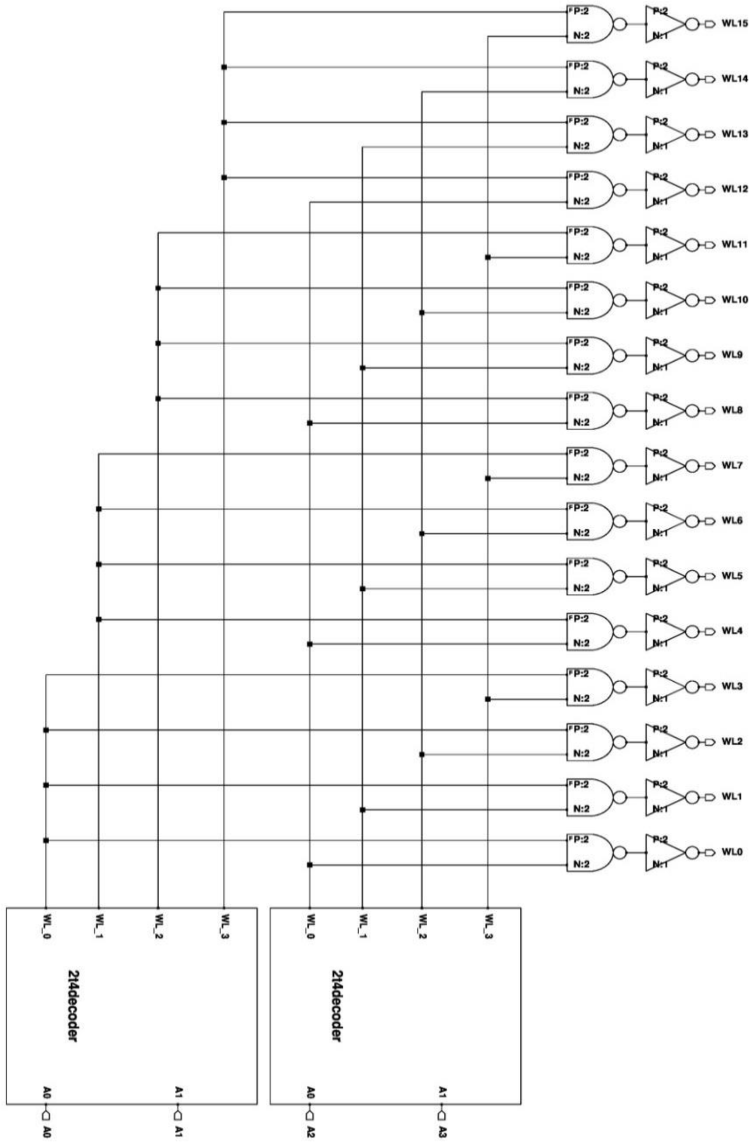


Figure 4.5 4:16 Hierarchical Address decoder

Depening on the combination of the input signal, one of the wordlines will be 1 at any time. For example, if the address input A0A1A2A3 is 0000, then the final output WL0 to WL15 is 16'h8000. The following Table 4.1 illustartes the detailed input and output siganls for the hierarchical decoder. As the size of the address line increases we can create the higher level decoder using the lower level decoders. For example, if we need a 8:256 decoder we can use two instances of 4:16(which is actually constructed based on 2:4 decoders as seen in figure 4.6) and two input NAND and INVERTER to form the final decoding stage.

Asserted ADDRESS SIGNAL(bin) A[3:0]	Predecoder stage 1 (bin)	Predecoder stage 2 (bin)	Wordline output WL[16:0] (hex)
0000	1000	0001	0001
0001	1000	0010	0002
0010	1000	0100	0004
0011	1000	1000	0008
0100	0100	0001	0010
0101	0100	0010	0020
0110	0100	0100	0040
0111	0100	1000	0080
1000	0010	0001	0100
1001	0010	0010	0200
1010	0010	0100	0400
1011	0010	1000	0800
1100	0001	0001	1000
1101	0001	0010	2000
1110	0001	0100	4000
1111	0001	1000	8000

Table 4.1 4:16 Address decoder input and output

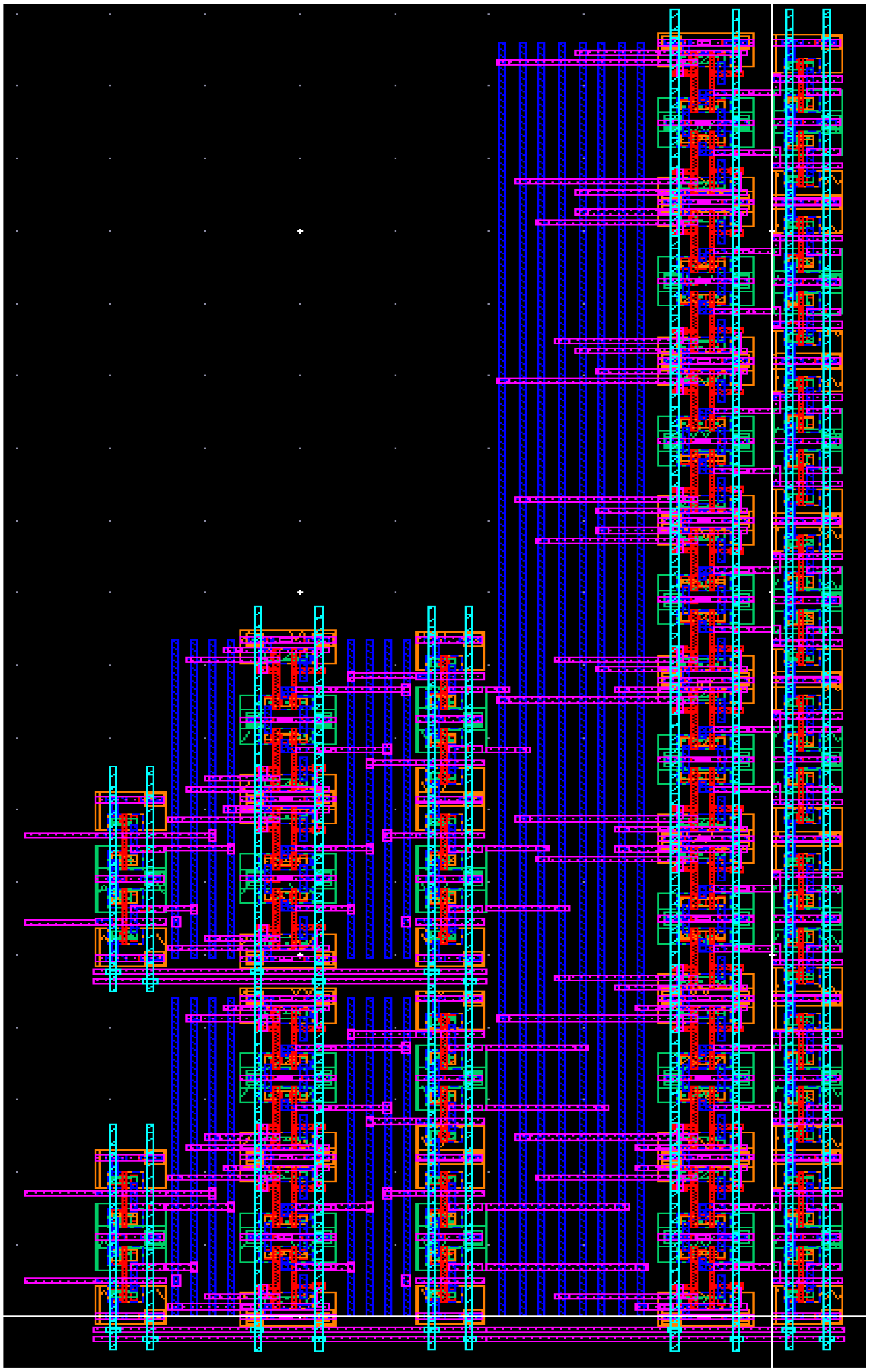


Figure 4.6 4:16 hierarchical decoder layout

As we know in order to construct the 8:256 decoder we need to construct the 4:16 using 2:4 deccoder heirarchically, hence the name hierarchical decoder. The Figure 4.6 shows the layout of 4:16 heirarchical decoder constructed using the software script. It consists of two instances of 2:4 decoders and a final decoding stage comprising of NAND and INVERTER gates.

4.2 Software implementation of layout

Software implementation of the layout can be explained as follows. OPENRAM is implemented using object-oriented data structures in the Python language. The top-level module is openram.py which parses input arguments, creates the memory and produces the output.

4.2.1 Design Hierarchy

The deisgn hierarchy of OPENRAM compiler can be explained as follows. All modules in shown in the blue oval of Figure 4.1 are derived from the design class in design.py. The design class is a data structure that consists of a name, a layout and a spice netlist. 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 [10].

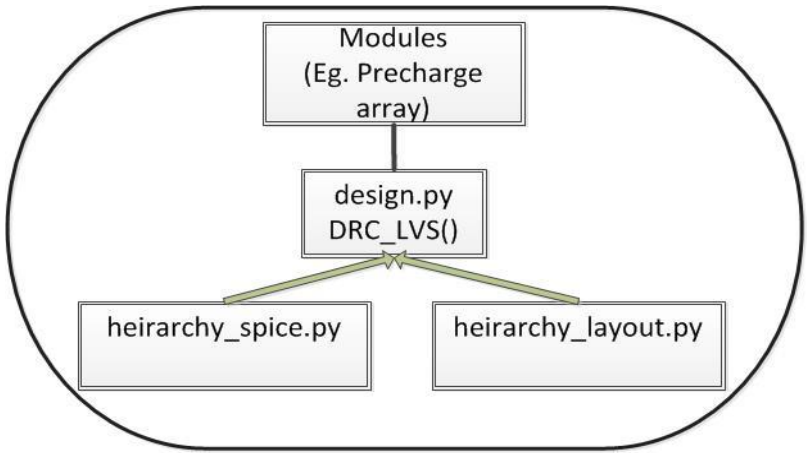


Figure 4.7 Design Hierarchy

4.2.2 Spice Hierarchy

The spice hierarchy of the design is stored in the hierarchy_spice.py. Initialization of the design class will create the spice data structure of the design. 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. Similarly the addition of an instance of a module/library cell/parameterized cell as a subcircuit to the top-level structure is carried out by add_mod() function. The connections between the sub –modules pins added to the spice hierarchy is performed using connect_pins() function . It is important to maintain the addition order of pins same as the sub- module. If there is a mismatch in the number of net connections, an assertion error will occur. The spice class contains the following functions for reading or writing spice files [10] :

- *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
- *sp_write_file()*: this function recursively writes the modules and sub-modules from the data structure into the spice file created by sp_write().

4.2.3 Layout Hierarchy

The layout hierarchy is stored in the layout class in hierarchy_layout.py. Similar to spice hierarchy, whenever the design class is initialized for a module, a data structure for the layout hierarchy is created. The layout data 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'slayout. 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 adds the path of the layer .

- **layernumber -** the layer that the label is to be drawn in .
- **Coordinates -** the x-y coordinates, in microns, where the label will be placed in the layout.
- **Width** – the width of the path to be added

• *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().

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

• *pdf_write():* this function takes the gds object and writes equivalent pdf image [10].

4.2.4Creating a New Design Module

All modules shown in the blue oval of Figure 4.1 is its own Python class, which inherits the design class. As stated before, design class inherits two other classes hierarchy_spice and hierarchy_layout. When the design class (design.py) is initialized within the module class, it subsequently creates separate data structures to hold the layout (hierarchy_layout) and spice (hierarchy_spice) information. Hence by inheriting the design class for each module, it is very easy to instantiate instances of the modules. These are the guidelines to be followed when creating a new module:

1. First we make required class from the design module:

```
Class precharge_array(design.design):
```

2. We will use the python constructor __init__ method so that required class class is initialized when an object of the module is instantiated. The module parameters should also be declared:

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

3. In the constructor odf the class , we should call the base class constructor with the name present in the library cell such as:

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

4. We add the pins that will be used in the spice net list for the required module using the add_pin() function from the hierarchy_spice class.

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

5. We create an instance of the module/library cell/parameterized cell that we want to add to the required module:

pre= precharge.precharge(precharge)

6. We add the subckt/submodule instance to the spice hierarchy using the add_mod() function from the hierarchy_spice class:

self.add_mod(pre)

7. We add the layout instance into required 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)

8. Finally we connect the pins of the instance that was just added by using the connect_pins function from the hierarchy_spice class:

self.connect_inst([PCLK[%d]%col,BL[%d]%col, BR[%d]%col, vdd]).

9. According to the required connections we can use add_rect() functions to add the metal layers. For example, adding rectangles for power/ground rails or routing, add labels, etc.

10. Every module needs to have "self" height and width variable that can be accessed from outside of the module class. These parameters are commonly used for placing instances modules in a layout. For library cells, the self.width and self.height variables are obtained from the GDSII layout using the cell_size() function in vlsi_layout.

11. We call to the DRC_LVS () function for physical verification of the modules [10].

4.3 Hierarchical decoder implementation

One of the demanding tasks in building the memory circuit is implementation of the decoder structure. All the components of the memory circuit can be arrayed side by side without any hassle except the address decoder structure. As they are arranged one beside the other the connection between them can be easily established, because of the abutment of layout instances. However, the address decoder uses logic gates like INVERTER and NAND gates in its construction. The number of gates in each stage of the address decoder is different. In order to explain the problem let us consider a 2:4 address decoder shown in the figure 4.2. First the address signals A0 and A1 are connected to the inputs of the INVERTER gates to obtain the true and complementary outputs of the address inputs. These true and complementary signals gained from the INVERTER is then linked to the NAND gates and in turn output from each NAND gate is connected to respective INVERTERS. The NAND gate and INVERTER together form an AND gate. The connections are made in such a way that, at any point of time for given inputs A0 and A1 only one output will go high. The layout of 2:4 address decoders is shown in the figure 4.4. This layout is equivalent to the schematic shown in the Figure 4.3. The routing of the connections between the INVERTERS and the NAND gates was tricky because we have to connect the true and complementary signals to the NAND gates. As we can see from the figure 4.4 the connection between INVERTER in the beginning and NAND gates were carried out using two metal layers M1 and M2 drawn horizontally and vertically respectively [6]. The implementation of the decoder using the software scripts is shown in the Figure 4.6. Similar to the software implementation explained before we follow the steps below to create a hierdecoder_predecoder module :

creating a hierdecoder_predecoder module:

1. Derived hierdecoder_predecoder from the design module:

    Class hierdecoder_predecoder (design.design):

2. The python constructor __init__ method is initialized. The module parameters for hierdecoder_predecoder is the address size it can be either 2 or 4 :

    def __init__(self, rowse):

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

    design.design.__init__(self,"address_decoder_%d_%d"%(addr_size,self.row_size))

4. Added all the pins that will be used in the spice net list for your module using the add_pin() function from the hierarchy_spice class.

self.add_pin("vdd")

5. Three gates were used inverter, nand2, and nand3 they were instantiated as follows in the module depending on the address size the program will add the required instance Since all the required class were in the same module we did not have to use file name during instantiation:

invobj= hier_decoder_inverter(“hier_decoder_inverter”)

nand2obj= hier_decoder_nand2(“hier_decoder_nand2”)

nand3obj= hier_decoder_nand3(“hier_decoder_nand3”)

6. Added the subckt/submodule instance to the spice hierarchy using the add_mod() function from the hierarchy_spice class for all the three instances:

self.add_mod(invobj)

7. Added layout instance into your module’s layout hierarchy using the add_instance() function, which takes a name, mod, offset, and mirror as inputs and also used function for generating names called as NameGenerator () located in the same file as decoder classes:

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

8. Connected the pins of the instance that was just added by using the connect_pins function from the hierarchy_spice class this was done simultaneously while adding the layout instance:

self.connect_inst([PCLK[%d]%col,BL[%d]%col, BR[%d]%col, vdd]).

9. After adding the instance the instances were connected using the by adding the metal and vias using the function :

self.add_rect(offset,width,hieght).

10. After all the connection were done self.width and self.hieght variable were calculated depending on the instances added and there width and lengths were calculated and assigned to these global vairables.

11. And finally DRC_LVS () function was called to make sure the address decoder generated is DRC and LVS clean [10].

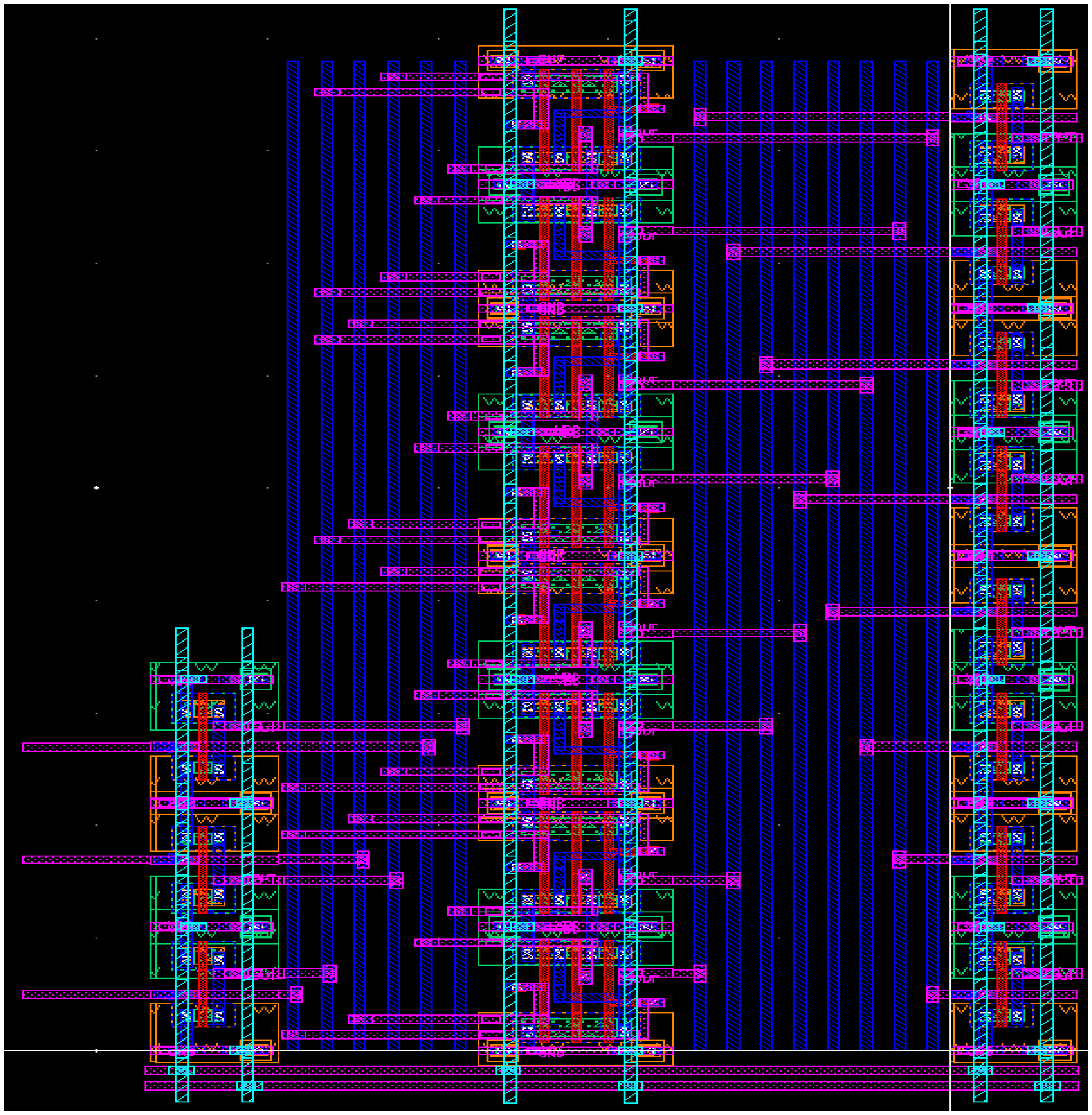


Figure 4.8 Layout of 3:8 Address Decoder

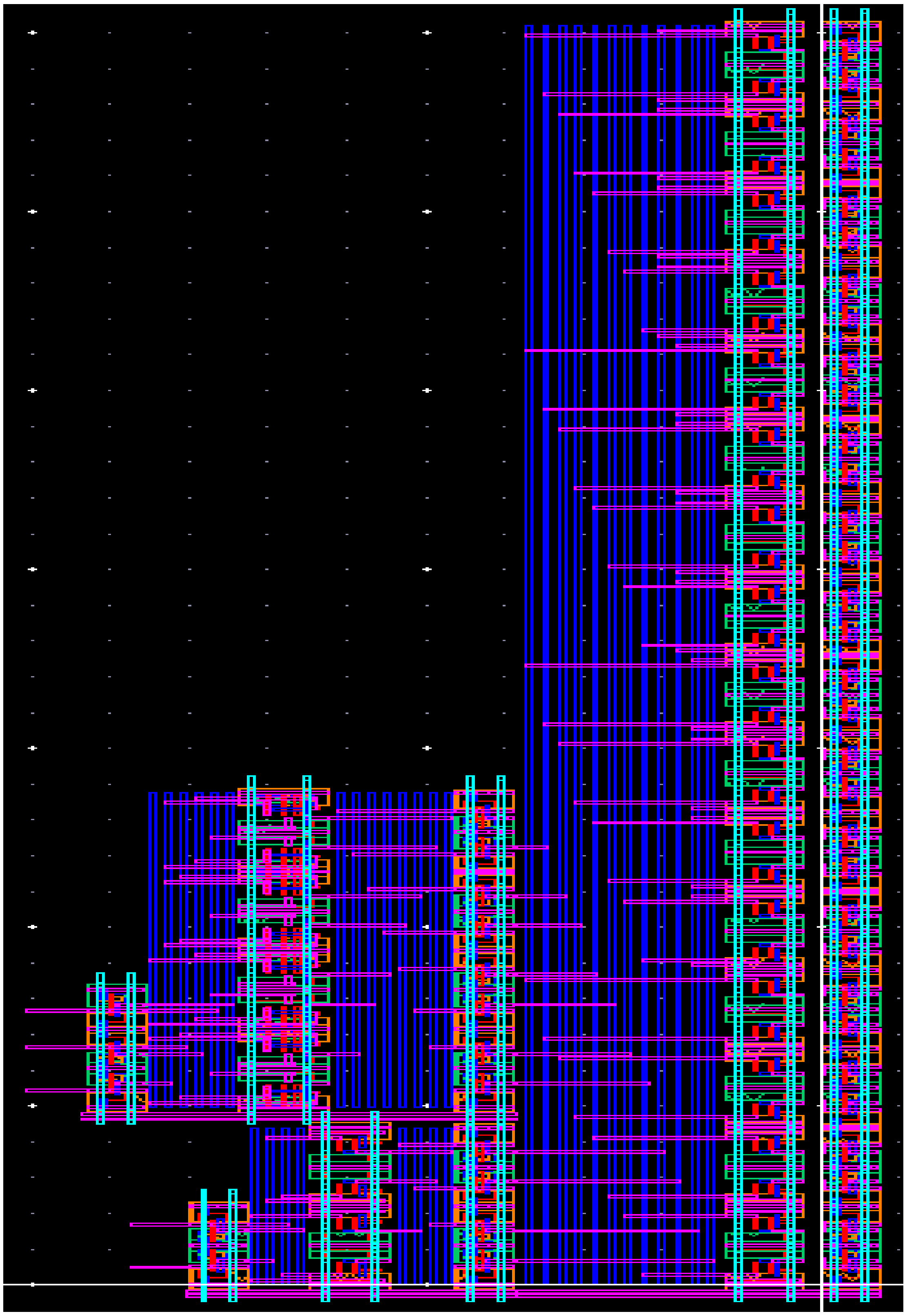


Figure 4.9 5:32 hierarchical address decoder

The Figure 4.7 shows the 3:8 address decoder generated using the software script. The higher order decoder structure greater than 3:8 is constructed using the class called hierdecoder. This class is also located in the same module as hierdecoder_predecoder. The address size accepted by the module is always 2 to power of X, where the X is whole number. It can be split into two equal halves. If we want to generate the 4:16 decoder, the hierdecoder class accepts the inputs for 4:16 decoder generation and it calculates and splits the predecoding stage into two 2:4 decoders and uses NAND and INVERTER instances for final decoding.

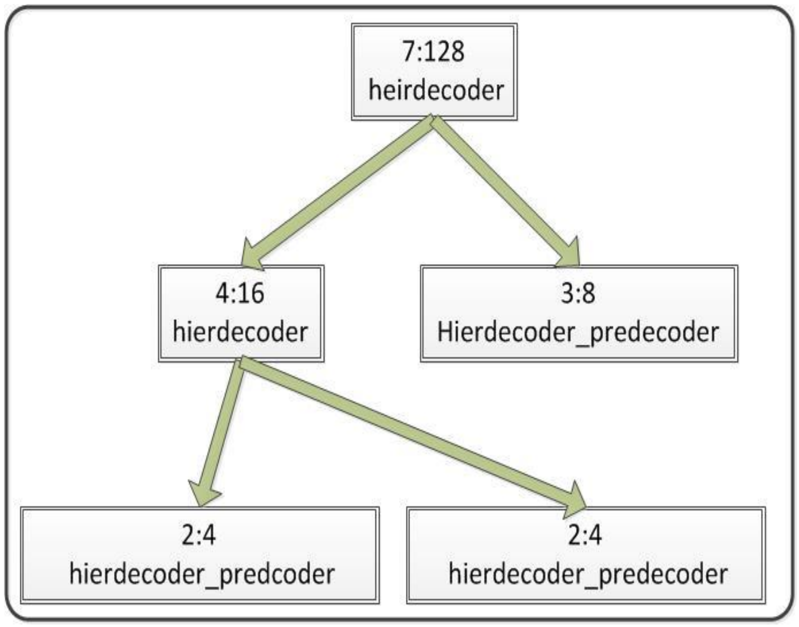


Figure 5.0 Program executions for 7:128 address decoder generation

As the decoder size increases, the hierdecoder splits the given address size to half. In the Figure 5.0 the address size is 7 and the program divides 7 by 2. The quotient is 3.5 and it rounds this value to the next nearest integer which in this case is 4. This value 4 is subtracted from the given address line 7 to get 3. As calculated, the programs now knows it has to generate 4:16 decoder and 3:8 decoder for predecoding. The program stops recursion when it reaches the primitive decoder stage like 2:4 or 3:8. Right branch of the tree in the figure has reached 3:8 hence it reached its end whereas the left branch of the tree is 4 due to which the recursion takes place again to split 4: 16 decoder into two 2:4 address decoders for predecoding. As we can see from the figure 5.0 whenever the calculation reaches the base case it uses the hierdecoder_predecoder class. Hence the hierdecoder module takes top level address size and adds the NAND and INVERTERS gates of that stage. This iteration of splits and the addition of gates takes place until the primitive decoder stage is reached. The Figure 4.9 shows the layout of the 5:32 address decoder constructed using the script.

4.4 Simulation for functional verification of decoder using Nanosim

Simulations were conducted using Nanosim Synopsys simulation tool in order to verify the functionality of the decoder. Nanosim is a similar to HSPICE but we have several advantages using Nanosim. One of them is we can provide the input vectors in the binary format in the text file called vector file. Nanosim will generate the pulses according to the binary vectors provided in the vector file which is very advantageous when we are simulating any logic design structure. The files required for the Nanosim simulation are given in the following table.

Name	Type	Definition
Netlist.sp	Spice	Extracted Netlist from layout editor.(can be any spice netlist)
Vector.inc	Spice	45 nanometer transistor models
Run	Executable	Command to execute the Nanosim simulation
Netlist.cfg	Configuration	Consists of Nanosim configurations and options. For examples signal format to wave viewing
Netlist.vec	Vector input file	Consists of the input vectors and expected output values

Table 4.3 List of Nanosim Simulation files

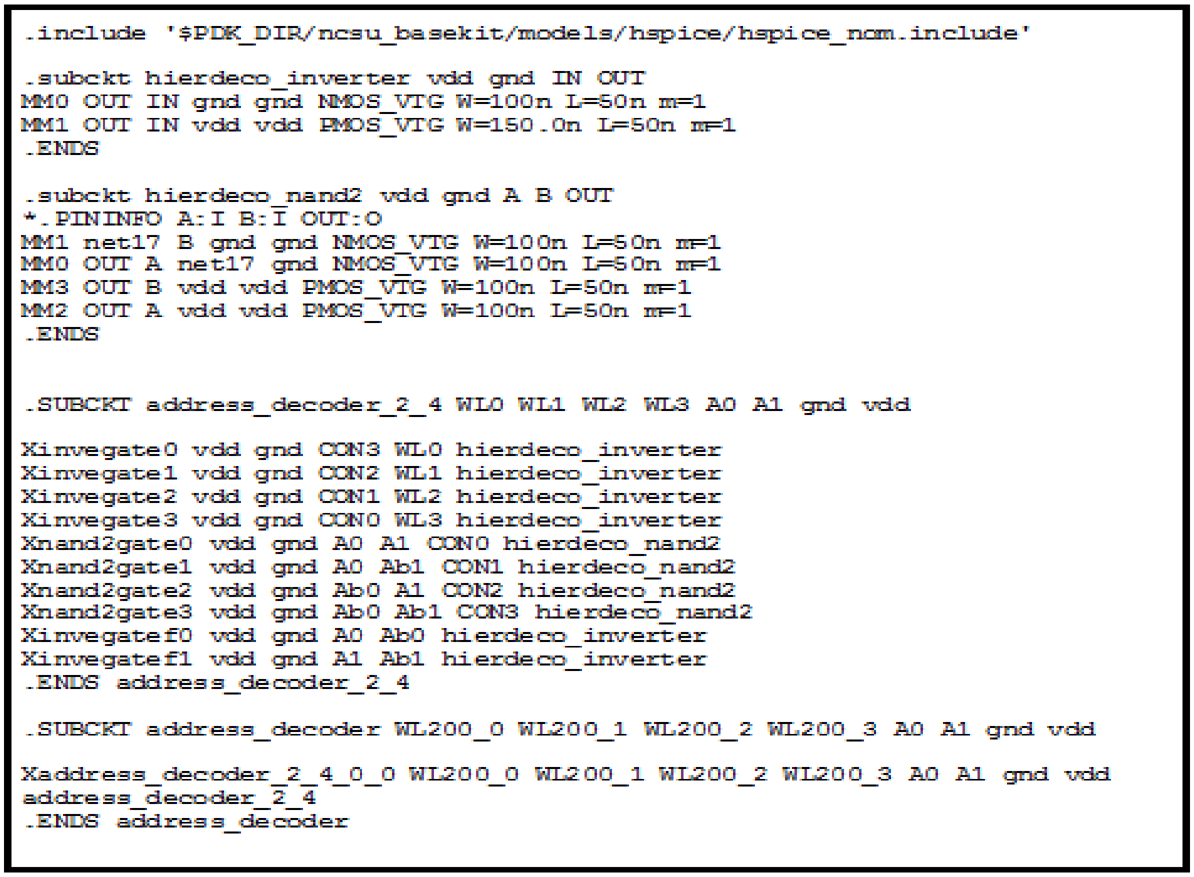
Nanosim applies the given input vector at the specified simulation time and asserts the output signals at the specified simulation for errors. Nanosim generates a list of files when the simulation is complete depending on the options selected during the simulation run. The files generated will use the name provided in the run file and use this name as the suffix for the file generated. Some of the files generated are as follows.

File Extensions	**Information represented**
User_specified.fsdb	Contains the information for the wave plots
User_specified.err	Specifies the comparison errors occurred during simulation
User_specified.fcap	Contains the information of the node capacitance when simulated with extracted spice file
User_specified.log	Specifies the commands used by the simulator

Table 4.4 List of Nanosim output files

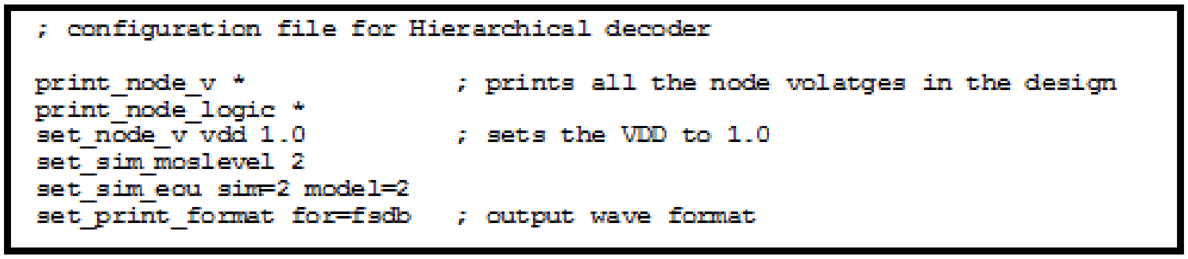
Before running the simulation it is necessary to set the environmental variable required by the nanosim tool on to the system's path. At OSU we have standard script *synopsys.chsrc* which has the path for the entire Synopsys tool environmental variable. Hence by sourcing this script we can run all the Synopsys tools. The sample files used for the simulation of the hierarchical decoder are shown and contents are briefly described as follows.

The Figure 5.1 shows the sample spice net list of the 2:4 address decoders. Usually the sub circuits like hierdeco_inverter and hierdeco_nand2 present in the net list will be in a separate file which includes all the sub circuits used for construction of different blocks. The configuration file used for the nanosim is shown in the figure 5.2. This file defines the print definition required for printing the node voltages. As well as defining the source voltage required for simulation usually the tool gets this information from the technology file but we can override it by providing the command specified in the configuration file. According the requirement, additional commands can be used for simulation.



```
.include '$PDK_DIR/ncsu_basekit/models/hspice/hspice_nom.include'

.subckt hierdeco_inverter vdd gnd IN OUT
MM0 OUT IN gnd gnd NMOS_VTG W=100n L=50n m=1
MM1 OUT IN vdd vdd PMOS_VTG W=150.0n L=50n m=1
.ENDS

.subckt hierdeco_nand2 vdd gnd A B OUT
*.PININFO A:I B:I OUT:O
MM1 net17 B gnd gnd NMOS_VTG W=100n L=50n m=1
MM0 OUT A net17 gnd NMOS_VTG W=100n L=50n m=1
MM3 OUT B vdd vdd PMOS_VTG W=100n L=50n m=1
MM2 OUT A vdd vdd PMOS_VTG W=100n L=50n m=1
.ENDS


.SUBCKT address_decoder_2_4 WL0 WL1 WL2 WL3 A0 A1 gnd vdd

Xinvegate0 vdd gnd CON3 WL0 hierdeco_inverter
Xinvegate1 vdd gnd CON2 WL1 hierdeco_inverter
Xinvegate2 vdd gnd CON1 WL2 hierdeco_inverter
Xinvegate3 vdd gnd CON0 WL3 hierdeco_inverter
Xnand2gate0 vdd gnd A0 A1 CON0 hierdeco_nand2
Xnand2gate1 vdd gnd A0 Ab1 CON1 hierdeco_nand2
Xnand2gate2 vdd gnd Ab0 A1 CON2 hierdeco_nand2
Xnand2gate3 vdd gnd Ab0 Ab1 CON3 hierdeco_nand2
Xinvegatef0 vdd gnd A0 Ab0 hierdeco_inverter
Xinvegatef1 vdd gnd A1 Ab1 hierdeco_inverter
.ENDS address_decoder_2_4

.SUBCKT address_decoder WL200_0 WL200_1 WL200_2 WL200_3 A0 A1 gnd vdd

Xaddress_decoder_2_4_0_0 WL200_0 WL200_1 WL200_2 WL200_3 A0 A1 gnd vdd
address_decoder_2_4
.ENDS address_decoder
```

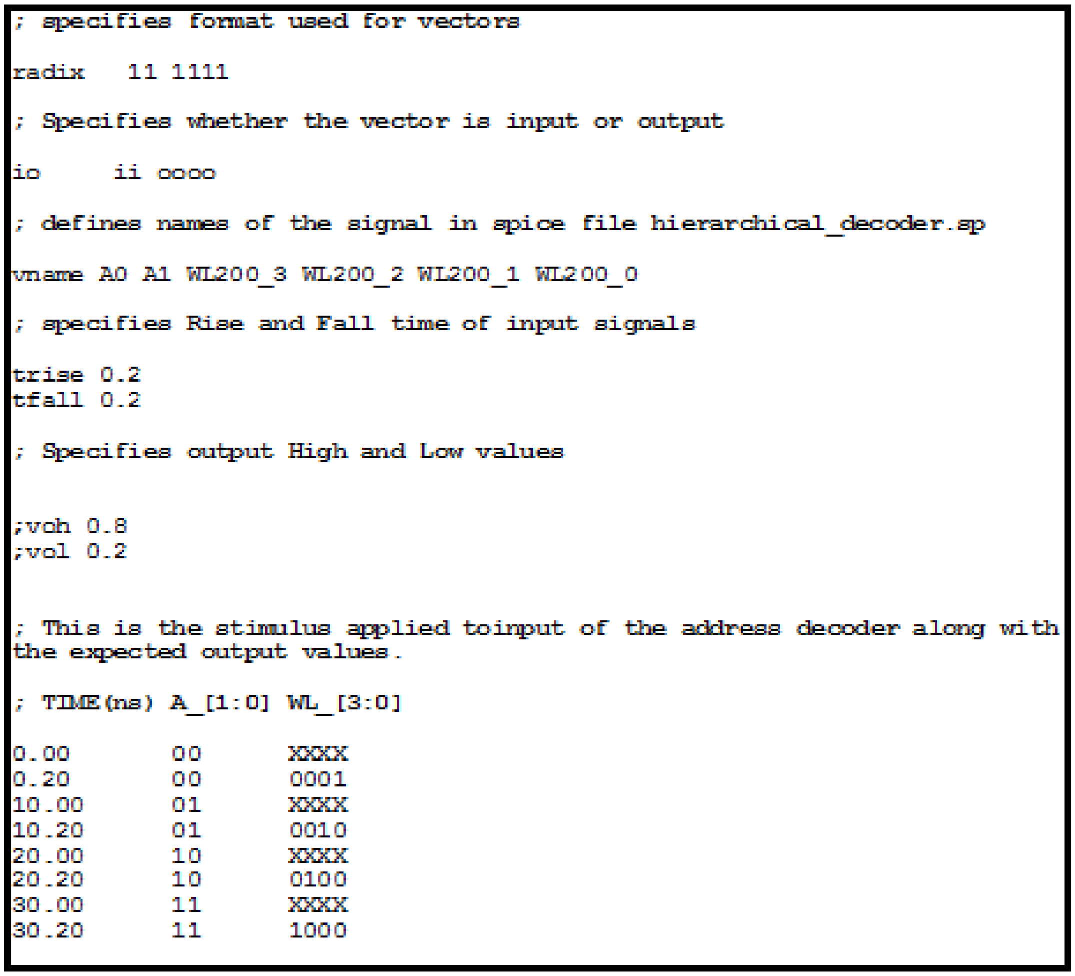
Figure 5.1 2:4 address decoder net list



```
; configuration file for Hierarchical decoder

print_node_v *              ; prints all the node volatges in the design
print_node_logic *
set_node_v vdd 1.0          ; sets the VDD to 1.0
set_sim_moslevel 2
set_sim_eou sim=2 model=2
set_print_format for=fsdb   ; output wave format
```

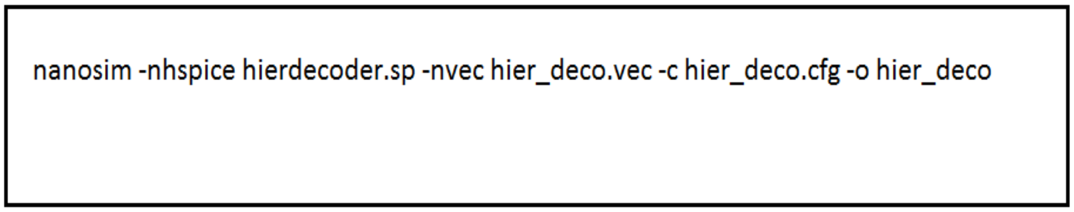
Figure 5.2 address_decoder.cfg

The vector file in which the inputs are defined is shown in the figure 5.3. The radix defines the bit size of the input variable were as the io represent the direction of the pins. Vname represents the input output signals names defined in the spice file being simulated.



```
; specifies format used for vectors

radix   11 1111

; Specifies whether the vector is input or output

io      ii oooo

; defines names of the signal in spice file hierarchical_decoder.sp

vname A0 A1 WL200_3 WL200_2 WL200_1 WL200_0

; specifies Rise and Fall time of input signals

trise 0.2
tfall 0.2

; Specifies output High and Low values


;voh 0.8
;vol 0.2


; This is the stimulus applied toinput of the address decoder along with
the expected output values.

; TIME(ns) A_[1:0] WL_[3:0]

0.00        00      XXXX
0.20        00      0001
10.00       01      XXXX
10.20       01      0010
20.00       10      XXXX
20.20       10      0100
30.00       11      XXXX
30.20       11      1000
```

Figure 5.3 address_decoder.vec



```
nanosim -nhspice hierdecoder.sp -nvec hier_deco.vec -c hier_deco.cfg -o hier_deco
```

Figure 5.4 address_decoder.run

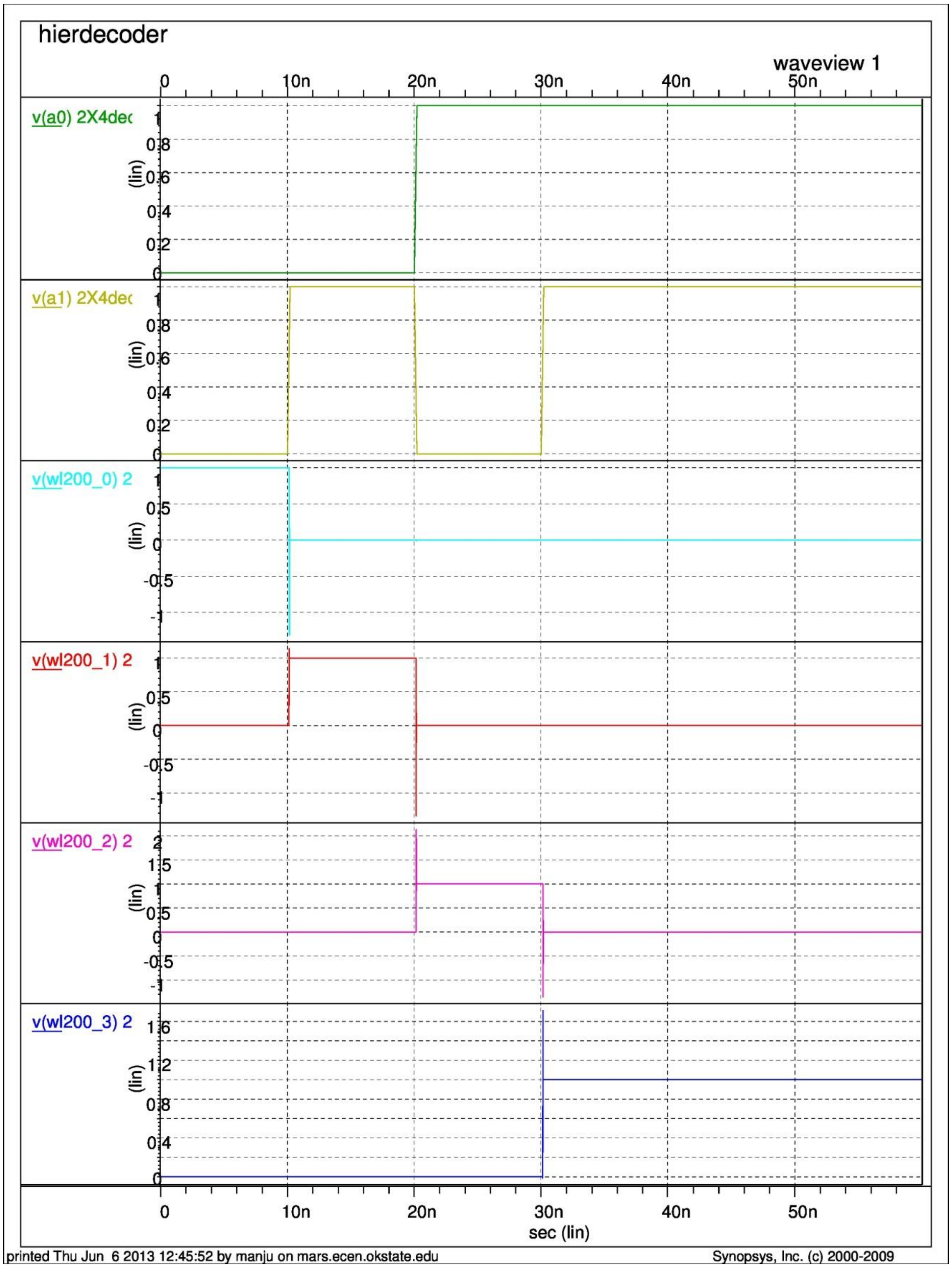


Figure 5.5 Wave form of the simulation

The Figure 5.4 shows the contents of the address_decoder.run where this file is given permission to be executed. When this command is run onto the system we obtained the list of output files shown in the Table 4.2. The Figure 5.5 shows the wave form of the simulation of the 2:4 address decoder. This waveform data was represented in the form fsdb data which was then used with Waveform viewer to see the waveforms of the simulation.

4.5 Timing Analysis for different decoder sizes

As the decoder sizes increase the fan out of the NAND gates in each stage increases. Due to this the access time or the delay from address signal to the word line activation increases drastically. But the hierarchical address decoder performs well for smaller size from 2:4 to 6:64. In fact the address decoders larger than this size will not be used, because the memory structures uses architecture of the global word line and local word line concept. In this architecture the memory blocks are divided into fixed blocks of small sizes like 32x8 and each of these blocks will have its own small size decoder. This local decoder will be driven by global decoders which will only face the load of local decoders. The local decoders in turn will drive a large load of memory array with less time delay.

ADDRESS decoder size	Delay Form Address to word line rise (ps)	Area of the decoder (um2)
2:4	40.7	4.87 X 5.125
3:8	72.8	6.175 X 9.40
4:16	76.8	8.625 X 18.25
5:32	80.2	10.75 X 36.25
6:64	88.1	11.60 X 70.50

Table 4.3 Address decoder size delay comparison

By choosing the hierarchical address decoder structure for memory implementation, we will eliminate the unnecessary delays caused by a large size row decoder during read and write operations. The delay is measured by simulating the extracted layout net list; the delay measurement is shown in the figure 5.6.

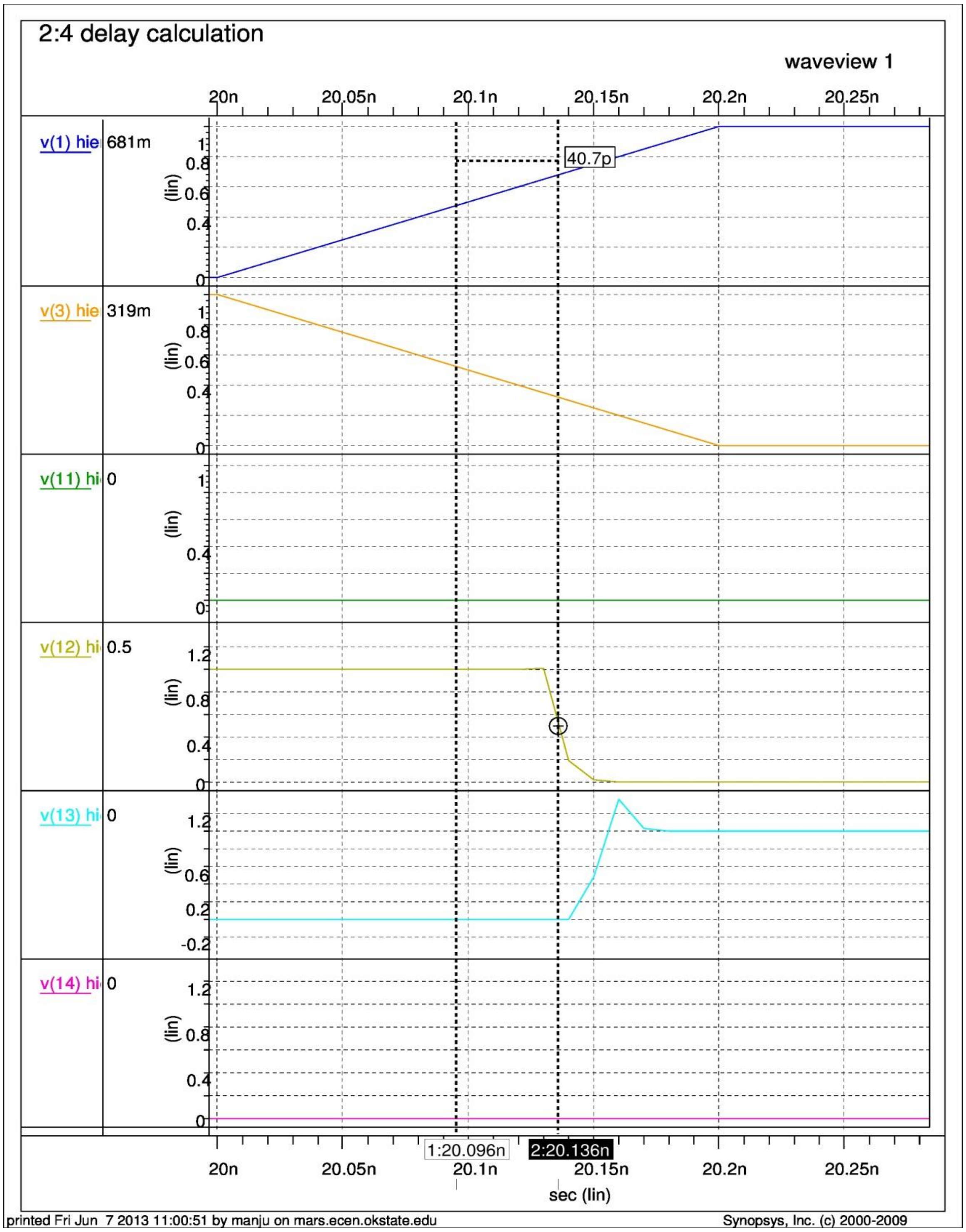


Figure 5.6 2:4 address decoder delay calculation

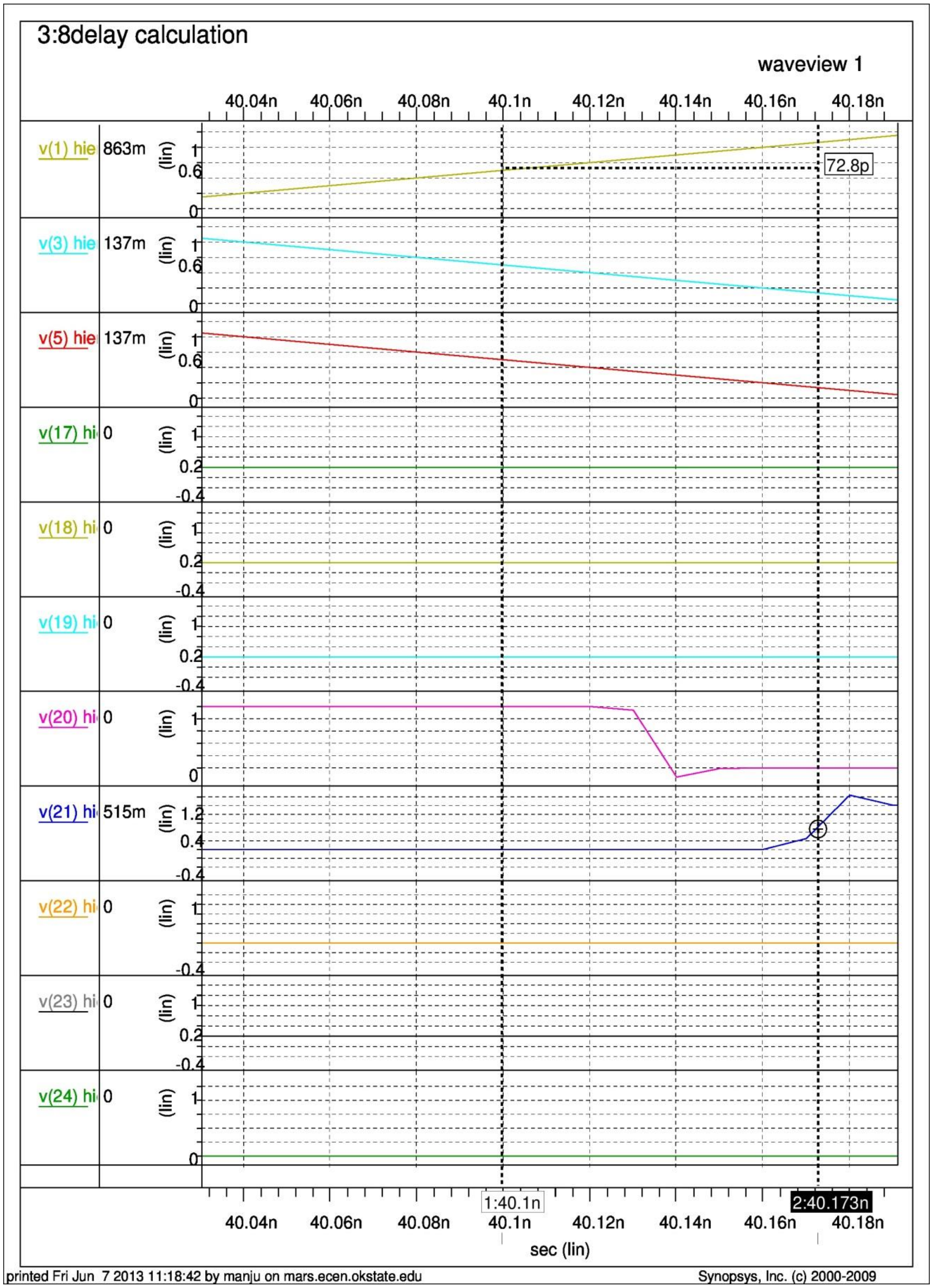


Figure 5.7 3:8 address decoder delay calculation

The extracted net list was simulated using Nanosim and we used Wave viewer to calculate the propagation delay between the input and output signals. In digital designs where we use the standard cells, they are previously characterized for timing. The EDA tool will obtain the pre characterized timing data from the liberty model to achieve the design constraints applied during synthesis. However, in physical synthesis we do consider the parasitic of the metal wires. In this case the delay calculation will be more accurate since the extracted net list of the complete decoder is simulated with parasitic of the metal wire connection included which gives us the approximately correct timing values.

4.6 Memory layout generation using the software scripts

The implementation of the memory structure is similar to the construction of the hierarchical address decoder explained in the section 4.2.1. Let us consider the figure 5.8.

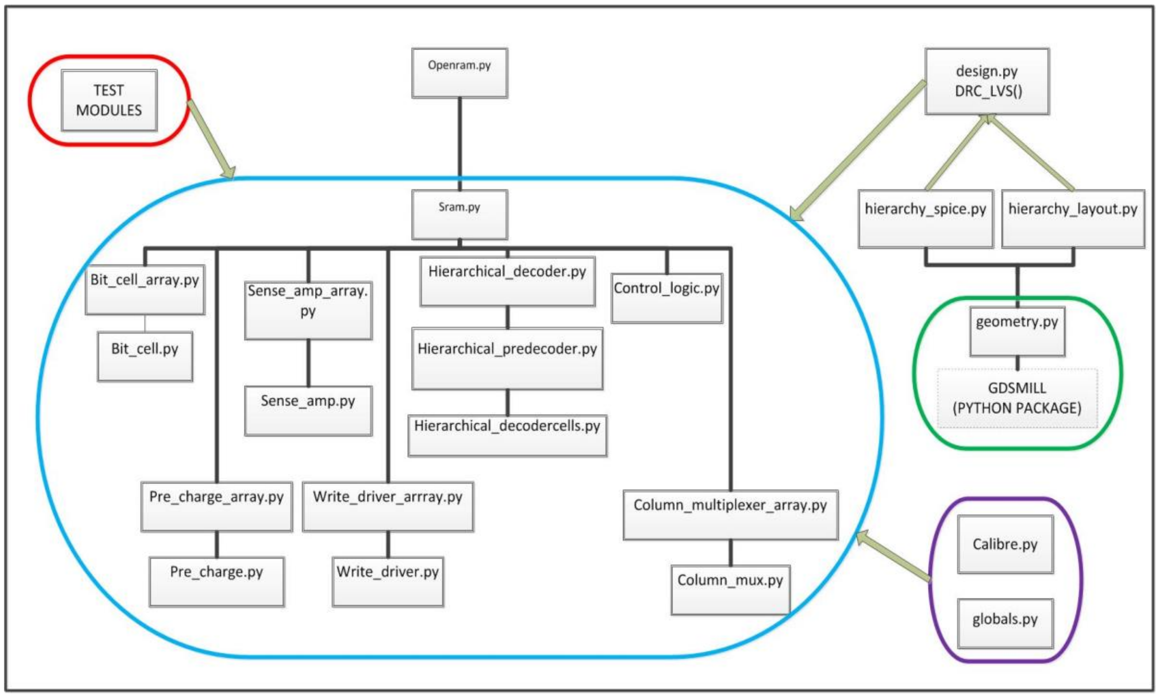


Figure 5.8 OPENRAM Compiler

This figure 5.8 serves two purpose one is that it shows the arrangement of the python scripts in the compiler and also depicts the design implementation of the OPENRAM compiler. Similar to the construction of the hierarchical decoder explained in the section 4.3. First the class for all the structure like pre charge array, write driver array, sense amplifier array, control logic and memory array are derived from the design class. As we can see from figure 5.8 the design class inherits the layout and spice capabilities from hierarchy_layout and hierarchy_spice classes. The geometry class is used as the interface between the OPENRAM compiler and GDSMILL python package which is necessary for GDS file manipulation. The design class also provides the DRC_LVS function which is obtained from *calibre.py* module and these functions can be called in any modules just by importing the *calibre.py* module. The construction of the memory layout starts with *sram.py* module which pass the required parameters for the rest of the layout components construction at the end of the module all the layout structures are assembled and DRC_LVS functions is called for the final verification. Even during the construction of each component DRC_LVS function is called individually to verify the generated layouts. Due to this ground up verification buildup of the memory structure obtained out of this compiler will be robust.

4.6.1 Memory timing

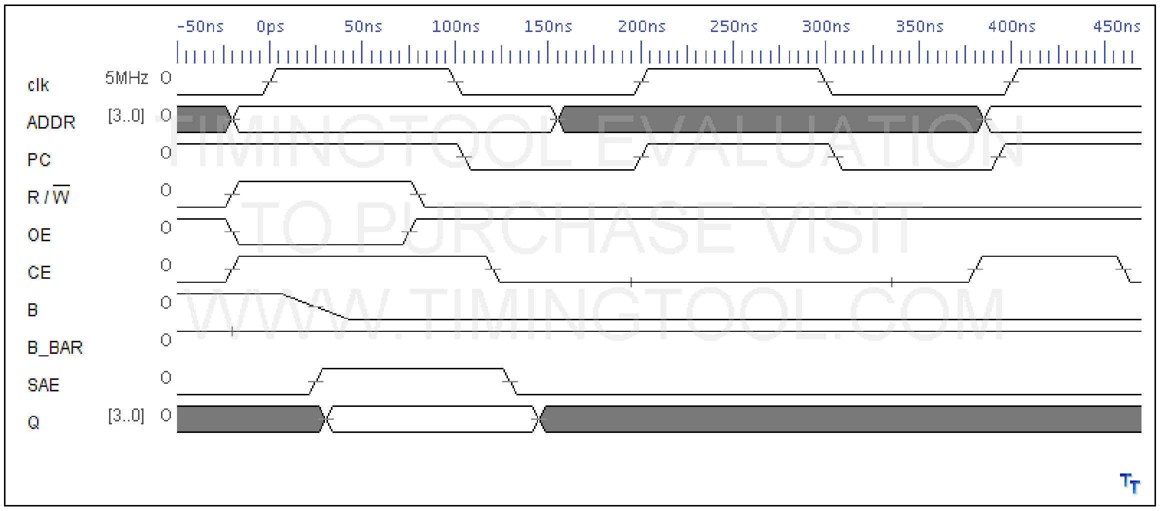


Figure 5.9 Read operation

The figure 5.9 shows the read operation timing diagram of the memory circuit. During the read operation the address and the control signals will be applied on its respective pins before the setup time required by the flip flops to capature the data. And these signals are held steady until the hold time requirement of the flipflops. After clock edge rise the address will be decoded with brief amount of propagation delay. The data will be available at the output pins.

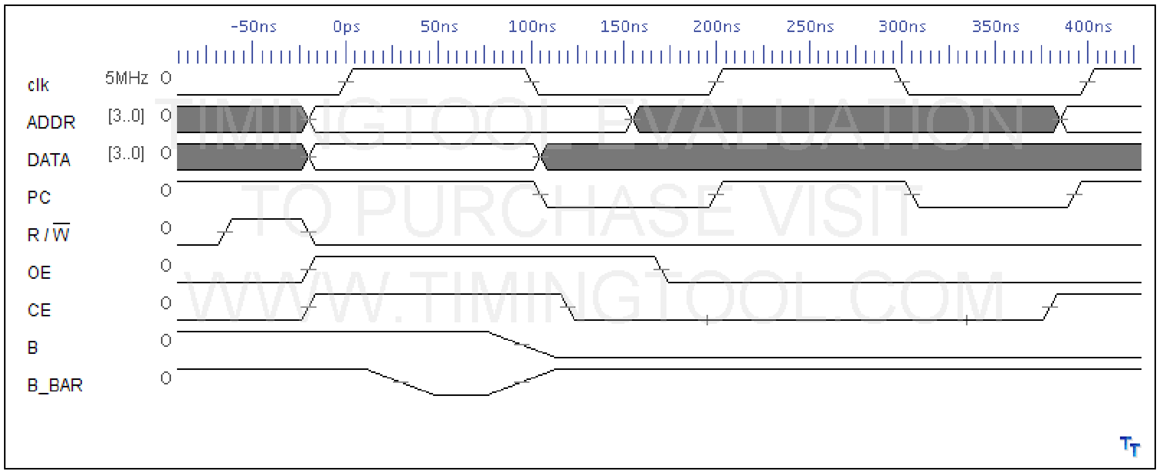


Figure 6.0 writes operation

The figure 6.0 shows the write operation timing diagram of the memory circuit**.** The function is similar to that of the read except that the data signals are included. Since the controls signals will be calibrated for write operation the signals at the output data pins will be high impendence. After rise edge of the clock the data presented on the data input pins will be written to the memory array selected by the applied address input signal.

4.6.2 Incorporation of hierarchical decoder into Memory Layout

The hierarchical decoder generated is a single module of the memory compiler. The layout generated by this module is integrated with the memory as shown in the following figures.

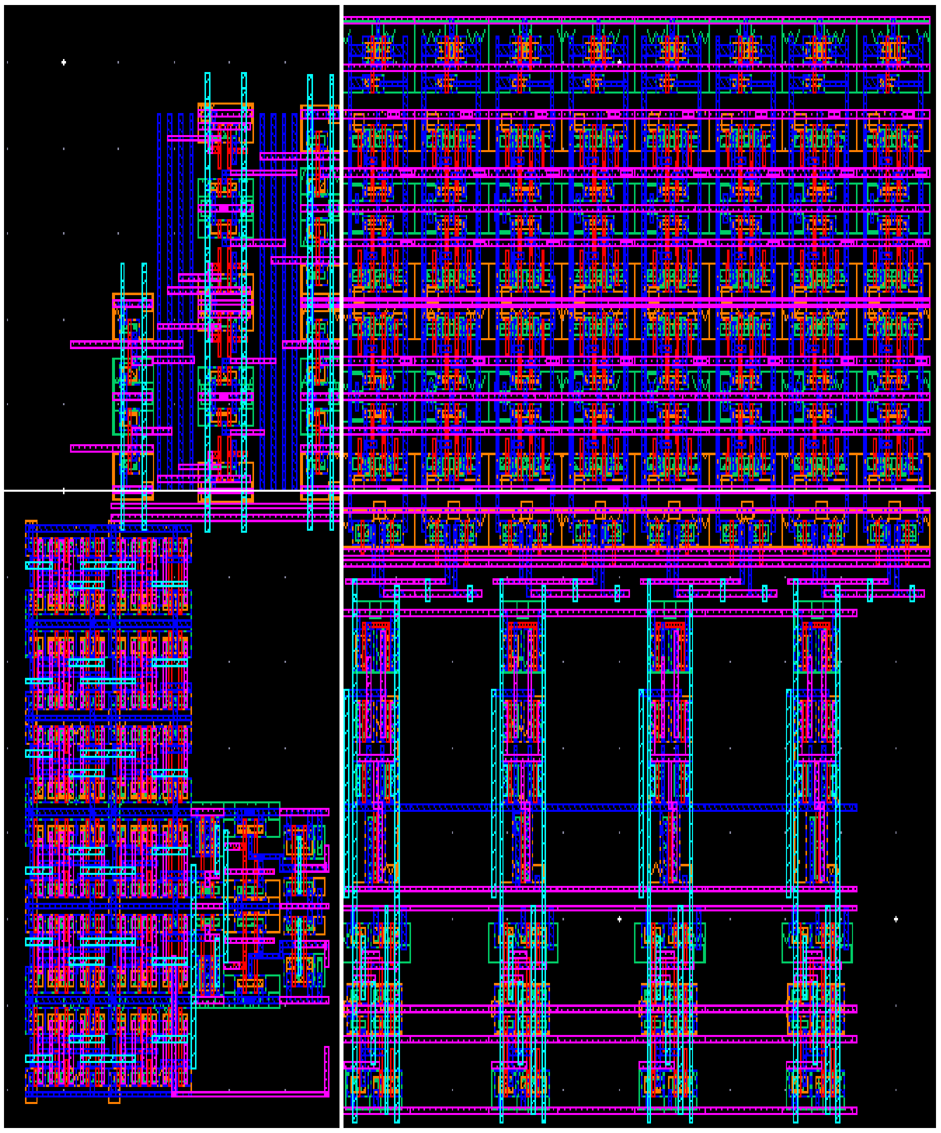


Figure 6.1 8 X 4 SRAM memory layout

The hierarchical decoder generated has its final stage of inverters pitch matched with the memory cells. The connection between hierarchical decoder and the memory cells takes place by abutment of the layouts. This abutment of the layout is shown in the figure 6.1.

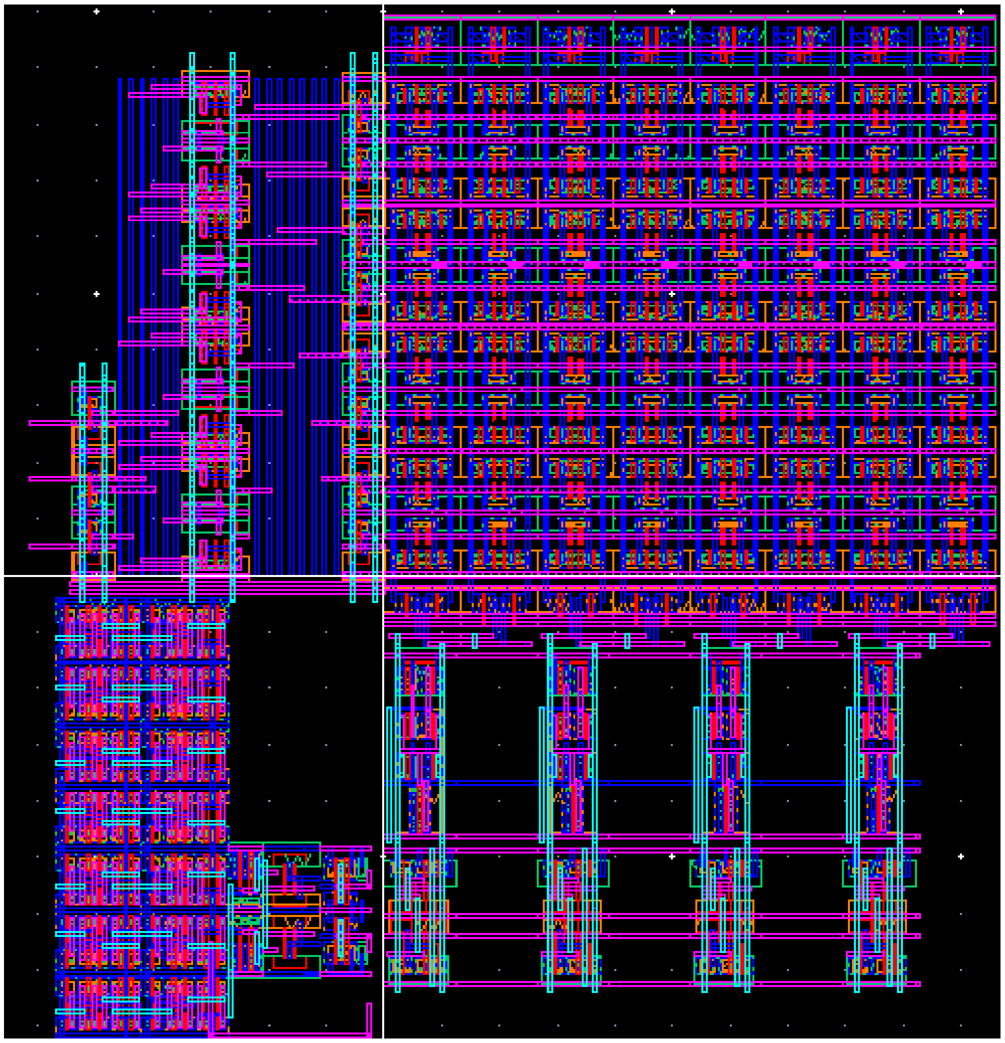


Figure 6.2 16 X 4 SRAM memory layout

The Figure 6.2 shows the 16 X 4 SRAM memory layouts. The compiler was give the parameters 16 memory word and each word is of size 4 bits. Hence the arrangement will generate the 3:8 decoders. In the figure 6.2 the connection between the decoder and memory array takes place by abutment. As we can see every two column has one column multiplexer. The column multiplexer is connected to the write driver and sense amplifier. The control logic and registers for applying the signals is shown in the bottom left corner of the figure 6.2.

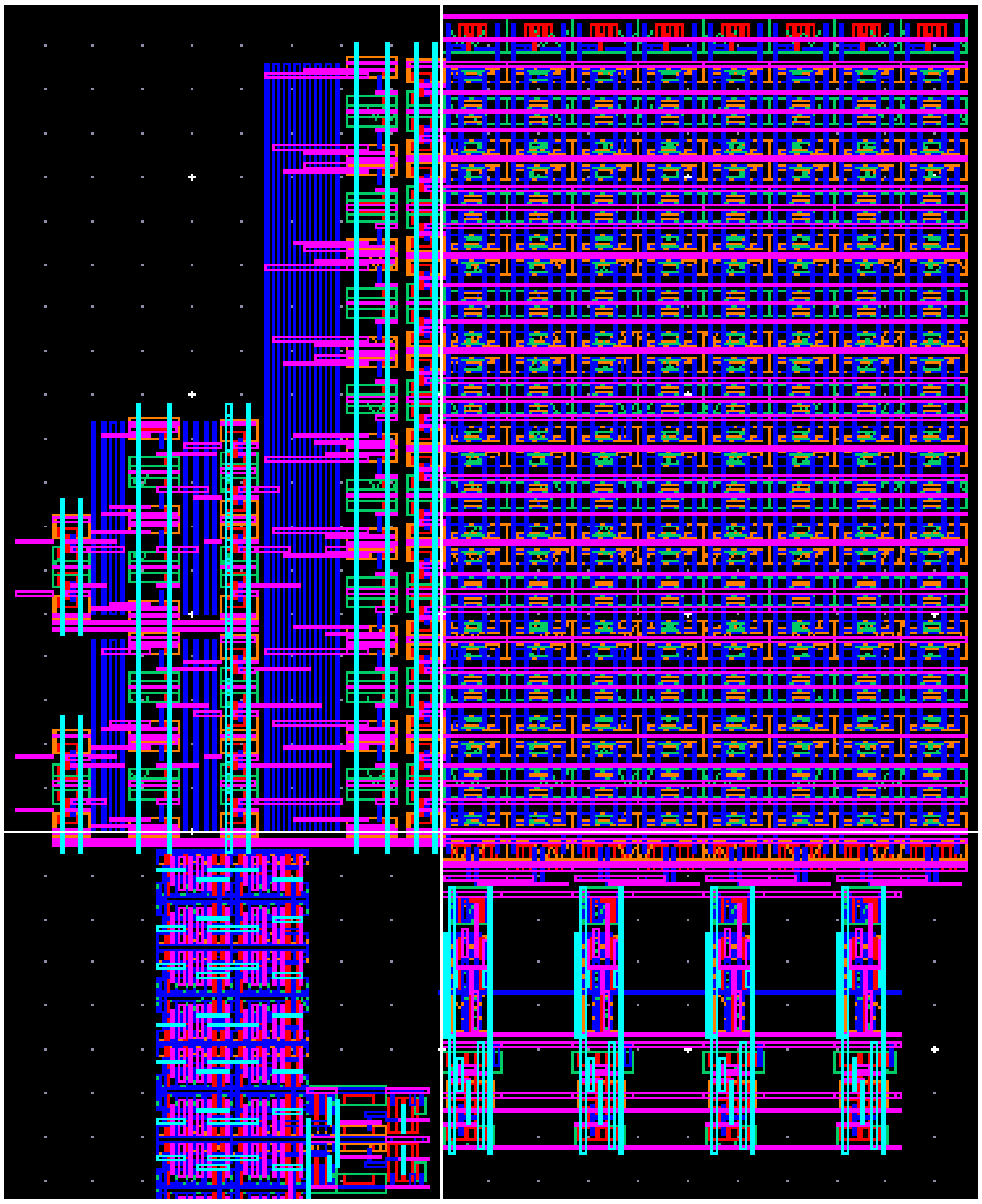


Figure 6.3 32 X 8 SRAM memory layout

The figure 6.3 shows the layout of 32 X 8 SRAM memory layout generated by the compiler. In this case the number of words of the memory is 32 and hence the compiler chose the 4:16 hierarchical decoder for its row decoding.

CHAPTER V

RESULTS AND CONCLUSION

The simulations of the hierarchical decoder prove that the implementation of the memory structure over the pass transistor decoder will yield good results. Major problem encountered during the implementation was routing the connections between the NAND gate stage and INVERTER gate stage. As we see from the figure 4.2, the 2:4 address decoder layouts the problem of routing was solved by adding the vertical metal layers in between the NAND gates stage and connecting the gates to these vertical wires. The connections for the next stage are obtained using this vertical wire added in between.

One of the disadvantages of incorporating this method is that as the decoder size increases the vertical length of metal route will also increase. This in turn will increase the voltage drops and the RC delay through the long wire. This can be eliminated by implanting the repeater logic in to the design.

In memory implementation we usually don’t opt for large decoders instead we use small local block decoders with small memory arrays. The global decoder is used to activate these small decoders. The method of using local and global decoders is more efficient. The voltage drop and RC delay will not be a significant when we use this method for decoding logic.

5.2 Future extensions of the memory compiler

The future extensions planned for the OPENRAM memory compiler is as shown in the figure 6.4.

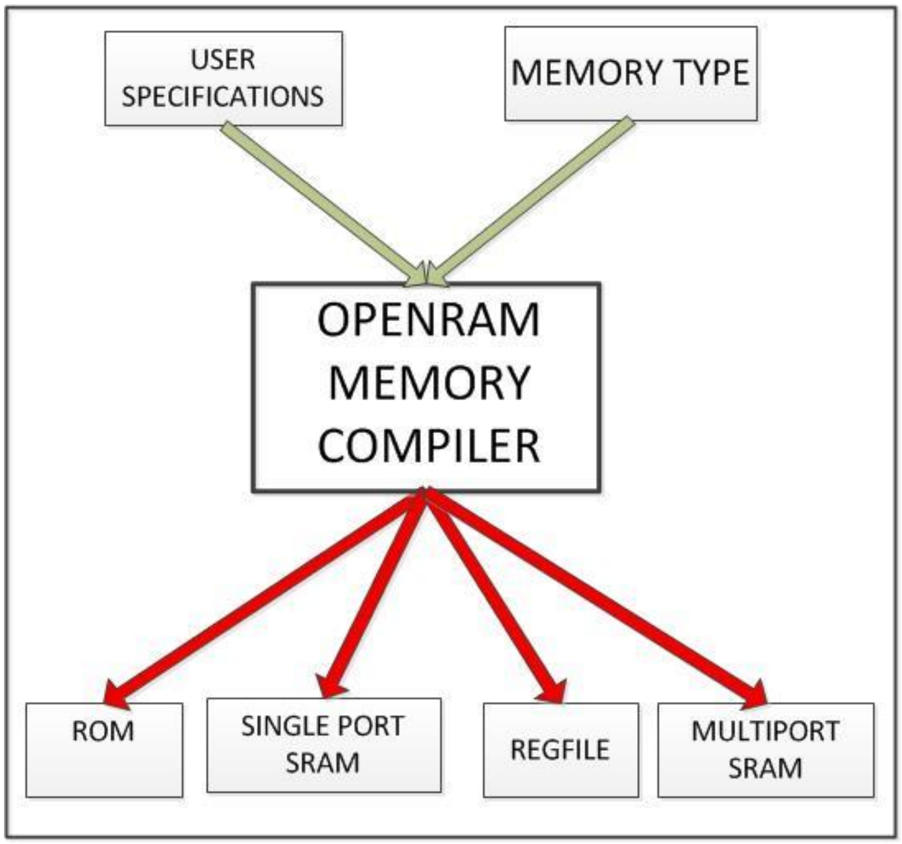


Figure 6.4 Future Enhancements to OPENRAM compiler

The current compiler is capable of generating only the single port SRAM memory architecture. In future we plan on extending the compiler to produce different memory structures like Multiport SRAM, register file memory and ROM. The future compiler will also have options to generate the large size memory structures by incorporation of bank architecture. The extension also includes the option of making compiler technology independent by dynamic generation of library cells required for memory construction.

REFERENCES

[1] N. H. E. Weste and D. Harris, "CMOS VLSI Design: A Circuits and Systems Perspective," 3rd ed: Addison Wesley, 2004, pp. 786-800.

[2] Harris, David Money, Harris,Sarah L, "*Digital Design and Computer Architecture,"* Amsterdam: Morgan Kaufmann Publishers; 2007.

[3] Bhasker J, Chadha R,"*Static Timing Analysis for Nanometer Designs: A Practical Approach," New* York: Springer; 2009.

[4] J. E. Stine, C. Jun, I. Castellanos, G. Sundararajan, M. Qayam, P. Kumar, J. Remington, and S. Sohoni, "FreePDK v2.0: Transitioning VLSI education towards nanometer variation-aware designs," in *Microelectronic Systems Education, 2009. MSE '09. IEEE International Conference on*, 2009, pp. 100-103.

[5] "International Technology Roadmap for Semiconductors," Semiconductor Industry Association2003.

[6] C. Saint and J. Saint, *IC Mask Design, Essential Layout Techniques*, 1st ed.: McGraw-Hill, 2002.

[7] Hennessy, John L, Patterson, David A, Arpaci-Dusseau,Andrea C."*Computer Architecture: A Quantitative Approac,"* Amsterdam: Morgan Kaufmann, 2007.

[8] E. Brunvand, *Digital VLSI Chip Design with Cadence and Synopsys CAD Tools*, 1st ed.: Addison Wesley, 2009.

[9] D. Hodges, H. Jackson, and R. Saleh. "Analysis and Design of Digital Integrated Circuits: *In Deep Submicron Technology. 3rd Ed. New York: McGraw Hill*," 2004. Pp.368-377.

[10] J. E. Stine, Matthew Guthaus, Jeff Butera, Bin Wu, Brian Chen, Ping-Yao Li(past Chase Peters, Marcelo Siero, Tom Golubev), "*openram open source memory compiler manual*," University of California at Santa Cruz; Oklahoma state university:2013.

,"

[11] Michael wieckowski,"*GDSMILL PYTHON Package,"* http://michaelwieckowski.com/wp-content/uploads/2012/01/GdsMillUserManual.pdf .

APPENDICES1

OPENRAM COMPILER SETUP SCRIPT

```
#source this file before using the openram compiler required environmental
#variables and python environment will be setup
#========================================
if ($?LD_LIBRARY_PATH) then
   echo "Your LD_LIBRARY_PATH is setup correctly"
else
   setenv LD_LIBRARY_PATH /usr/lib:/usr/X11R6/lib:/lib:/usr/local/lib
endif
#========================================
########### Cadence software ############
setenv PYTHON /home/manju/Desktop/Openram/PYTHON
setenv PYTHONPATH
/home/manju/Desktop/Openram/opencorrection/openram/trunk/gdsMill:/home/manju/Desktop/O
penram/opencorrection/openram/trunk/compiler:/home/manju/Desktop/Openram/opencorrection/
openram/trunk/freepdk45
setenv OPENRAM_HOME /home/manju/Desktop/Openram/opencorrection/openram/trunk
setenv OPENRAM_TECH
/home/manju/Desktop/Openram/opencorrection/openram/trunk/freepdk45
setenv DRCLVS_HOME /home/manju/Desktop/pcells/Freepdk/ncsu_basekit/techfile/calibre


# Mentor Graphics' Calibre DRC/LVS/xRC (PeX)
```

```
setenv CADENCE /import/cadence1

setenv MGC_HOME /import/comet1/Mentor/ixl_cal_2012.4_25.21

setenv CALIBRE $MGC_HOME


# Set the PDK_DIR variable to the root directory of the FreePDK distribution

setenv PDK_DIR /home/manju/Desktop/pcells/Freepdk

setenv CDS_LIC_FILE  ${CADENCE}/license/license.dat:5280@cadence.ceatlabs.okstate.edu

setenv LM_LICENSE_FILE 1717@avatar.ecen.okstate.edu:27000@avatar.ecen.okstate.edu

setenv CDS_LIC_ONLY 1

#============= Aliases =================

alias prepend 'if (-d \!:2) if ("$\!:1" \!~ *"\!:2"*) setenv \!:1 "\!:2":${\!:1}'

alias extend  'if (-d \!:2) if ("$\!:1" \!~ *"\!:2"*) setenv \!:1 ${\!:1}:\!:2'

#=======================================


#============= Library Definitions ====

prepend LD_LIBRARY_PATH $PYTHON/lib : $CALIBRE/lib

#prepend LD_LIBRARY_PATH $PYTHON/lib

#=======================================


#============= Path Definitions ========

set path = ($PYTHON/bin : $CALIBRE/bin  \ $path)

#set path = ($PYTHON/bin  \ $path)

#=======================================
```

APPENDICES2

PROCEDURE TO COMPILE PYTHON CODE TO OBTAIN PCELLS

Procedure to compile python code to obtain pcells using Ciranova pycell studio

The following instruction has to be followed as it is to obtain the Pcells because it will be easy for you to compile without any problems.

1) First in your home directory make a directory named as pcells you can do it by typing following command

%mkdir ~/pcells

2) Change the directory to pcells by typing following command

%cd pcells

3) Make other directory called Freepdk inside pcells directory the command is as follows

%mkdir Freepdk

4) Change the directory to Freepdk directory as follows

%cd Freepdk

5) In the Freepdk directory type the following command to obtain the technology files and the python code required for obtaining Pcells

%svn co https://svn.unity.ncsu.edu/osi/freepdk45/trunk .

Note:- you might want to type this command twice to get all the files and get a response saying that the kit is at 174 revision

6) Downloading and installing Ciranova Pycell Studio software

- The pycells studio can be downloaded can be downloaded from the following website www.ciranova.com
- Depending on the Linux version and the cadence layout editor you are using you can download the suitable software for you
- In our case the we are using the 64 bit Red Hat Linux and IC6.1.5 cadence so we have downloaded the following pycells studio

Linux 64 Pycell studio(Release 4.6, Openaccess 22.41.004)

- The name of this directory will be lengthy one and it will be in the tar.gz form i.e,
  Ciranova_linux_rhe140_gcc44x_py262.tar.gz

- You can untar the directory by typing the following command in the directory where you have downloaded the file

%tar -xvzf \

ciranova_installer_linux_rhe140_gcc44x_64_4.6_l1_Py262.tar.gz

- Copy this directory to the pcells directory which you created before in your home directory you can do this as following the command is given assuming that the above untared directory is in Downloads directory which is also in home directory

%cp -rf \\ ~/Downloads/ciranova_installer_linux_rhe140_gcc44x_64_4.6_l1_Py262 \\ ~/pcells

- Now we shall install the pycells studio this is done as follows change the directory to ciranov…………………………….py262 directory by using following command

%cd ~/pcells/ciranova_installer_linux_rhe140_gcc44x_64_4.6_l1_Py262

- In this directory type

% ./installer

- the prompt will ask for path to install the software you can give any path you want but it will be easy if type following command it will be easy later for setting up environment so just type this

% ../cni

- the above command will install the software in the pcells directory with the name cni.

- This will install several packages one of them is python2.6.2 but we used python2.6.5 for compiling and we downloaded it from the following link www.python.org

7) The next task is modifying the scripts which setup the environment for cadence and pycell studio you have to modify three scripts as follows we will show you the scripts in the Freepdk kit and the modified script so you can make out the changes made and modify your scripts accordingly.

$PDK_DIR /home2/manju/pcells/Freepdk

The three scripts which you have to modify and they are in the following paths

- $PDK_DIR/ncsu_basekit/cdssetup/icoa_setup.csh
- $PDK_DIR/ncsu_basekit/cdssetup/setup.csh
- $PDK_DIR/ncsu_basekit/gentech/sshaft/src/setup.csh

ACTUAL SCRIPTS

ICOA_SETUP.CSH

```
#########################################################################
#  ICOA Setup Script
#########################################################################

# BEGIN customizable section
setenv CDSHOME          /afs/eos/dist/cadence2010/ic
setenv OA_HOME                  /afs/eos/dist/cadence2010/oa
if (`arch` == sun4) then
   setenv CNI_ROOT        /afs/bp/contrib/pycell425/.install/solaris
   setenv CNI_PLAT_ROOT    ${CNI_ROOT}/plat_solaris_32
else
   setenv CNI_ROOT        /afs/bp/contrib/pycell425/.install/linux32
   setenv CNI_PLAT_ROOT    ${CNI_ROOT}/plat_linux_gcc411_32
endif

# New Path Entries
set newdirs = ( \
                    $CDSHOME/tools/bin \
                    $CDSHOME/tools/dfII/bin \
                    $OA_HOME/bin \
                    $CNI_PLAT_ROOT/3rd/bin \
                    $CNI_PLAT_ROOT/3rd/oa/bin/linux_rhel30_gcc411_32/opt \
                    $CNI_PLAT_ROOT/bin \
                    $CNI_ROOT/bin \
                    /afs/eos/dist/cadence2010/edi/bin \
                    /afs/eos/dist/cadence2008/edi/tools/bin \
                    )

# New LD_LIBRARY_PATH Entries
set newlibs = ( \
                    $CDSHOME/tools/lib \
                    $OA_HOME/lib \
                    $CNI_PLAT_ROOT/3rd/lib \
                    $CNI_PLAT_ROOT/3rd/oa/lib/linux_rhel30_gcc411_32/opt \
                    $CNI_PLAT_ROOT/lib \
                    )

# New Environment Variables
setenv SKIP_CDS_DIALOG
setenv CDS_LIC_FILE 2101@ece-lic-03.ece.ncsu.edu:2101@ece-lic-
04.ece.ncsu.edu:2101@ece-lic-05.ece.ncsu.edu
setenv CDS_Netlisting_Mode Analog
#setenv LD_ASSUME_KERNEL 2.4.1
```

```
setenv PYTHONHOME $CNI_PLAT_ROOT/3rd
setenv PYTHONPATH
$CNI_ROOT/pylib:$CNI_PLAT_ROOT/pyext:$CNI_PLAT_ROOT/lib:.
# The following gets rid of the annoying messages that pycell prints to stdout
setenv CNI_LOG_DEFAULT /dev/null


# END customizable section



# Extend path
foreach dir ($newdirs)
   set present = `printenv PATH | /bin/grep $dir`
   if ($present == "") then
      set path= ( $dir $path )
   endif
   unset present
end
unset dir newdirs

# Extend LD_LIBRARY_PATH
if ( ! $?LD_LIBRARY_PATH ) then
   setenv LD_LIBRARY_PATH
endif
foreach dir ($newlibs)
   set present = `printenv LD_LIBRARY_PATH | /bin/grep $dir`
   if ($present == "") then
      setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${dir}
   endif
   unset present
end
unset dir newdirs
```

SETUP.CSH

```
###############################################################################
#
#                                                      #
# FreePDK Setup Script                                            #
#    2/23/2008 by Rhett Davis (rhett_davis@ncsu.edu)                    #
#                                                      #
###############################################################################
#

# Set the PDK_DIR variable to the root directory of the FreePDK distribution
setenv PDK_DIR /home2/manju/times/pcells4/Freepdk
```

```
# Set CDSHOME to the root directory of the Cadence ICOA installatio
#setenv CDSHOME /import/cadence1/IC_06.15.504

if !(-f ${PWD}/.cdsinit ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/cdsinit ${PWD}/.cdsinit
endif

if !( -f ${PWD}/cds.lib ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/cds.lib ${PWD}/cds.lib
endif

if !( -f ${PWD}/lib.defs ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/lib.defs ${PWD}/lib.defs
endif

if !(-f ${PWD}/.runset.calibre.drc ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.drc ${PWD}/.runset.calibre.drc
endif

if !(-f ${PWD}/.runset.calibre.lvs ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.lvs ${PWD}/.runset.calibre.lvs
endif

if !(-f ${PWD}/.runset.calibre.lfd ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.lfd ${PWD}/.runset.calibre.lfd
endif

if !(-f ${PWD}/.runset.calibre.pex ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.pex ${PWD}/.runset.calibre.pex
endif

set present = `printenv PYTHONPATH`
if ($present == "") then
  setenv PYTHONPATH $PDK_DIR'/ncsu_basekit/techfile/cni'
else
  setenv PYTHONPATH $PYTHONPATH':'$PDK_DIR'/ncsu_basekit/techfile/cni'
endif

setenv MGC_CALIBRE_DRC_RUNSET_FILE ./.runset.calibre.drc
setenv MGC_CALIBRE_LVS_RUNSET_FILE ./.runset.calibre.lvs
setenv MGC_CALIBRE_PEX_RUNSET_FILE ./.runset.calibre.pex
```

SETUP.CSH

```
###############################################################################
#  SSHAFT Setup Script
###############################################################################
```

```
# BEGIN customizable section

setenv MUSE_HOME /afs/eos.ncsu.edu/lockers/research/ece/wdavis
setenv PDK_DIR
/afs/eos.ncsu.edu/lockers/research/ece/wdavis/users/wdavis/svn/freepdk45/trunk
setenv SSHAFT_HOME $PDK_DIR/ncsu_basekit/gentech/sshaft

# New Path Entries
set newdirs = ( \
                    $SSHAFT_HOME/bin \
                    )

# New Environment Variables
#setenv XKEYSYMDB /usr/X11/lib/X11/XKeysymDB

# Environment variables for the automated design flow
setenv SSHAFT_PERL $SSHAFT_HOME/lib/pl
setenv SSHAFT_SKILL $SSHAFT_HOME/lib/il
setenv SSHAFT_TCL $SSHAFT_HOME/lib/tcl

# END customizable section



# Extend path
foreach dir ($newdirs)
    set present = `printenv PATH | /bin/grep $dir`
    if ($present == "") then
        set path= ( $path $dir )
    endif
    unset present
end
unset dir newdirs

# Setup Python 2.4
# (Need to prepend these directories, since some versions of Linux
# include older versions of Python)
if ( `arch` == sun4 ) then
  set present = `printenv PATH | /bin/grep $MUSE_HOME/bin`
  if ($present == "") then
    set path= ( $MUSE_HOME/bin.sun4v $path )
  endif
  setenv PYTHONHOME $MUSE_HOME/tools/Python-2.4.1
  set present = `printenv PYTHONPATH | /bin/grep $MUSE_HOME`
  if ($present == "") then
    setenv PYTHONPATH $PYTHONHOME/Lib:$PYTHONHOME/Lib/lib-
tk:$PYTHONHOME/build/lib.solaris-2.8-sun4u-2.4:$SSHAFT_HOME/lib/py
  endif
else
```

```
  set present = `printenv PATH | /bin/grep $MUSE_HOME/bin`
  if ($present == "") then
    set path= ( $MUSE_HOME/bin.lnx86 $path )
  endif
  setenv PYTHONHOME $MUSE_HOME/tools/Python-2.4.3
  set present = `printenv PYTHONPATH | /bin/grep $MUSE_HOME`
  if ($present == "") then
    setenv PYTHONPATH $PYTHONHOME/Lib:$PYTHONHOME/Lib/lib-
tk:$PYTHONHOME/build/lib.linux-i686-2.4:$SSHAFT_HOME/lib/py
  endif
endif
```

<u>MODIFIED SCRIPTS</u>

ICOA_SETUP.CSH

```
############ Cadence software #############

setenv CADENCE /import/cadence1

setenv CDS_LOG_PATH ~/cadence/FREEPDK45logs

setenv CDS_LOG_VERSION sequential

setenv CDS_LIC_FILE ${CADENCE}/license/license.dat

setenv OSU_FREEPDK ~/pcells/Freepdk/osu_soc

setenv LD_ASSUME_KERNEL 2.4.1

setenv OA_COMPILER gcc44x

setenv MGC_HOME /import/mentor1/ixl_cal_2011.2_16.14

setenv PYTHONHOME /import/vlsi2/python


#==========================================
# NCSU Setup:
#==========================================

setenv CDSHOME          /import/cadence1/IC_06.15.504
setenv OA_HOME                    $CDSHOME/oa_v22.41.007
setenv CNI_ROOT         ~/pcells/cni
setenv CNI_PLAT_ROOT    ${CNI_ROOT}/plat_linux_gcc44x_64
setenv OA_PLUGIN_PATH   $OA_HOME/data/plugins
setenv OA_PLUGIN_PATH
${CNI_PLAT_ROOT}/3rd/oa/data/plugins:${CNI_ROOT}/quickstart:${OA_PLUGIN_PATH}

# New Path Entries
set newdirs = ( \
                $CDSHOME/tools/bin \
                $CDSHOME/tools/dfII/bin \
                $OA_HOME/bin \
                $CNI_PLAT_ROOT/3rd/bin \
                $CNI_PLAT_ROOT/3rd/oa/bin/linux_rhel40_gcc44x_64/opt \
                $CNI_PLAT_ROOT/bin \
                $CNI_ROOT/bin \
                $PYTHONHOME/bin \
                /import/cadence1/EDI_10.13.002/bin \
                /import/cadence1/EDI_10.13.002/tools/bin \
                )

# New LD_LIBRARY_PATH Entries
set newlibs = ( \
                $CDSHOME/tools/lib \
                $OA_HOME/lib/linux_rhel40_gcc44x_64 \
                $CNI_PLAT_ROOT/3rd/lib \
                $CNI_PLAT_ROOT/3rd/oa/lib/linux_rhel40_gcc44x_64/opt \
                $CNI_PLAT_ROOT/lib \
                $PYTHONHOME/lib  \
                )

# New Environment Variables
setenv SKIP_CDS_DIALOG

setenv CDS_Netlisting_Mode Analog


setenv PYTHONPATH
$CNI_ROOT/pylib:$CNI_PLAT_ROOT/pyext::$CNI_PLAT_ROOT/lib:

# The following gets rid of the annoying messages that pycell prints to stdout
setenv CNI_LOG_DEFAULT /dev/null

# END customizable section

# Extend path
foreach dir ($newdirs)
   set present = `printenv PATH | /bin/grep $dir`
   if ($present == "") then
      set path= ( $dir $path )
   endif
   unset present
end
```

```
unset dir newdirs


# Extend LD_LIBRARY_PATH
foreach dir ($newlibs)
    set present = `printenv LD_LIBRARY_PATH | /bin/grep $dir`
    if ($present == "") then
        setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${dir}
    endif
    unset present
end
unset dir newdirs

#==========================================

source /home2/manju/pcells/Freepdk/ncsu_basekit/cdssetup/setup.csh

#===========================================
echo "You are now set up to use Cadence properly to use FreePDK"
#==========================================
```

SETUP.CSH

```
###############################################################################
#
#                                                       #
# FreePDK Setup Script                                              #
#    2/23/2008 by Rhett Davis (rhett_davis@ncsu.edu)                     #
#                                                       #
###############################################################################
#

# Set the PDK_DIR variable to the root directory of the FreePDK distribution
setenv PDK_DIR /home2/manju/pcells/Freepdk

if !(-f ${PWD}/.cdsinit ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/cdsinit ${PWD}/.cdsinit
endif

if !( -f ${PWD}/cds.lib ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/cds.lib ${PWD}/cds.lib
endif

if !( -f ${PWD}/lib.defs ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/lib.defs ${PWD}/lib.defs
endif

if !(-f ${PWD}/.runset.calibre.drc ) then
```

```
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.drc ${PWD}/.runset.calibre.drc
endif

if !(-f ${PWD}/.runset.calibre.lvs ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.lvs ${PWD}/.runset.calibre.lvs
endif

if !(-f ${PWD}/.runset.calibre.lfd ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.lfd ${PWD}/.runset.calibre.lfd
endif

if !(-f ${PWD}/.runset.calibre.pex ) then
  cp ${PDK_DIR}/ncsu_basekit/cdssetup/runset.calibre.pex ${PWD}/.runset.calibre.pex
endif

set present = `printenv PYTHONPATH`
if ($present == "") then
  setenv PYTHONPATH $PDK_DIR'/ncsu_basekit/techfile/cni'
else
  setenv PYTHONPATH $PYTHONPATH':'$PDK_DIR'/ncsu_basekit/techfile/cni'
endif

setenv MGC_CALIBRE_DRC_RUNSET_FILE ./.runset.calibre.drc
setenv MGC_CALIBRE_LVS_RUNSET_FILE ./.runset.calibre.lvs
setenv MGC_CALIBRE_PEX_RUNSET_FILE ./.runset.calibre.pex
```

SETUP.CSH

```
###########################################################################
#  SSHAFT Setup Script
###########################################################################

# BEGIN customizable section

setenv PDK_DIR /home2/manju/pcells/Freepdk
setenv SSHAFT_HOME $PDK_DIR/ncsu_basekit/gentech/sshaft

# New Path Entries
set newdirs = ( \
                    $SSHAFT_HOME/bin \
                    )

# New Environment Variables
#setenv XKEYSYMDB /usr/X11/lib/X11/XKeysymDB

# Environment variables for the automated design flow
setenv SSHAFT_PERL $SSHAFT_HOME/lib/pl
setenv SSHAFT_SKILL $SSHAFT_HOME/lib/il
setenv SSHAFT_TCL $SSHAFT_HOME/lib/tcl

# END customizable section



# Extend path
foreach dir ($newdirs)
   set present = `printenv PATH | /bin/grep $dir`
   if ($present == "") then
      set path= ( $path $dir )
   endif
   unset present
end
unset dir newdirs
```

from the above scripts you can see the scripts are drastically changed to match our cadence and the python environment as you can see from the variable PYTHONHOME it is the path where our new python is installed.

8) Now you should do following things once the script is modified.
In the following path you will find the file named Makefile

$PDK_DIR/ncsu_basekit/gentech/sshaft/src/py/Makefile-

The first line of the file defines a variable called SCRIPTPATH=, set this path to your python binary executable in our case the path is as follows

SCRIPTPATH=/import/vlsi2/python/bin/python

Just know that we installed the python in the following path /import/vlsi2/python .

In the same directory type command ***%make*** this will copy the SCRIPTPATH which is nothing but the python binary executable path and add it to all python files located in the following path

$PDK_DIR/ncsu_basekit/gentech/sshaft/bin/

9) Next you will need to create a new script that adds both cadence and pycell studio to your environment at the same time. Before that please verify that you have modified the icoa_setup.csh correctly you can do that by sourcing it and starting the virtuoso and pyros by typing the following commands.

%virtuoso -64 (for starting cadence)

%pyros (for starting pycell studio)

10) The python file named setup.py in the following path contains environment setting for various tools.

$PDK_DIR/ncsu_basekit/gentech/sshaft/lib/py/setup.py

11) Now we will edit this file to match your new Virtuoso & Pycell script. It can be done as follows

- Open a fresh terminal (important)
- Type the following command %printenv > env1
- Source the icoa_setup script located in the following path

$PDK_DIR/ncsu_basekit/cdssetup/icoa_setup.csh

- Type the following command %printenv > env2
- Source the setup.csh script located in the following path

$PDK_DIR/ncsu_basekit/gentech/sshaft/setup.csh

- There are two setup scripts the first setup script will be sourced when you source the icoa_setup.csh so no need to worry about that.
- Copy the env1 env2 files in to the following path

%cp ~/env* $PDK_DIR/ncsu_basekit/gentech/sshaft/src/py/

- Now change the directory to

%cd $PDK_DIR/ncsu_basekit/gentech/sshaft/src/py/

- In the directory type the following command

%python envparse.py env1 env2 setup.txt

this will parse the differences between env1 env2 files and put the result in the file setup.txt.

The setup.txt file which we obtained is as follows.

```
'NewEnvVars':['PDK_DIR', 'SKIP_CDS_DIALOG', 'CNI_LOG_DEFAULT', 'CDS_LIC_FILE',
'PYTHONPATH', 'CDSHOME', 'MGC_CALIBRE_DRC_RUNSET_FILE', 'OA_HOME',
'OSU_FREEPDK', 'CDS_LOG_VERSION', 'OA_COMPILER',
'MGC_CALIBRE_LVS_RUNSET_FILE', 'CNI_PLAT_ROOT', 'MGC_HOME',
'CDS_Netlisting_Mode', 'CADENCE', 'LD_ASSUME_KERNEL', 'CDS_LOG_PATH',
'OA_PLUGIN_PATH', 'PYTHONHOME', 'CNI_ROOT',
'MGC_CALIBRE_PEX_RUNSET_FILE', ],
   'AppendEnvVars':['LD_LIBRARY_PATH', ],
   'PrependEnvVars':['PATH', ],
   'PDK_DIR':'/home2/manju/pcells/Freepdk',
   'SKIP_CDS_DIALOG':'',
   'CNI_LOG_DEFAULT':'/dev/null',
   'CDS_LIC_FILE':'/import/cadence1/license/license.dat',

'PYTHONPATH':'/home2/manju/pcells/cni/pylib:/home2/manju/pcells/cni/plat_linux_gcc44x_64
/pyext:/home2/manju/pcells/cni/plat_linux_gcc44x_64/lib:.:/home2/manju/pcells/Freepdk/ncsu_b
asekit/techfile/cni',
   'CDSHOME':'/import/cadence1/IC_06.15.504',
   'MGC_CALIBRE_DRC_RUNSET_FILE':'./.runset.calibre.drc',
   'OA_HOME':'/import/cadence1/IC_06.15.504/oa_v22.41.007',
   'OSU_FREEPDK':'/home2/manju/pcells/Freepdk/ncsu_basekit/osu_soc',
   'CDS_LOG_VERSION':'sequential',
   'OA_COMPILER':'gcc44x',
   'MGC_CALIBRE_LVS_RUNSET_FILE':'./.runset.calibre.lvs',
   'CNI_PLAT_ROOT':'/home2/manju/pcells/cni/plat_linux_gcc44x_64',
   'MGC_HOME':'/import/mentor1/ixl_cal_2011.2_16.14',
   'CDS_Netlisting_Mode':'Analog',
   'CADENCE':'/import/cadence1',
   'LD_ASSUME_KERNEL':'2.4.1',
   'CDS_LOG_PATH':'/home2/manju/cadence/FREEPDK45logs',

'OA_PLUGIN_PATH':'/home2/manju/pcells/cni/plat_linux_gcc44x_64/3rd/oa/data/plugins:/hom
e2/manju/pcells/cni/quickstart:/import/cadence1/IC_06.15.504/oa_v22.41.007/data/plugins',
   'PYTHONHOME':'/import/vlsi2/python',
   'CNI_ROOT':'/home2/manju/pcells/cni',
   'MGC_CALIBRE_PEX_RUNSET_FILE':'./.runset.calibre.pex',

'LD_LIBRARY_PATH':':/import/cadence1/IC_06.15.504/tools/lib:/import/cadence1/IC_06.15.50
4/oa_v22.41.007/lib:/home2/manju/pcells/cni/plat_linux_gcc44x_64/3rd/lib:/home2/manju/pcells
/cni/plat_linux_gcc44x_64/3rd/oa/lib/linux_rhel40_gcc44x_64/opt:/home2/manju/pcells/cni/plat_
linux_gcc44x_64/lib:/import/vlsi2/python/lib',

'PATH':'/import/cadence1/EDI_10.13.002/tools/bin:/import/cadence1/EDI_10.13.002/bin:/import
/vlsi2/python/bin:/home2/manju/pcells/cni/bin:/home2/manju/pcells/cni/plat_linux_gcc44x_64/bi
n:/home2/manju/pcells/cni/plat_linux_gcc44x_64/3rd/oa/bin/linux_rhel40_gcc44x_64/opt:/home
2/manju/pcells/cni/plat_linux_gcc44x_64/3rd/bin:/import/cadence1/IC_06.15.504/oa_v22.41.007/
bin:/import/cadence1/IC_06.15.504/tools/dfII/bin:/import/cadence1/IC_06.15.504/tools/bin:',
```

- Insert the content of the setup.txt file into the setup.py file which is located in the following path

$PDK_DIR/ncsu_basekit/gentech/sshaft/lib/py/setup.py

as shown below.

```
...
Env = {
  'cadence':{
(insert contents of setup.txt here, deleting the previous contents)
  },
  'synopsys':{
...
```

- Finally you are ready to run the command which generates the pcells before that change the directory to the following path

%cd $PDK_DIR/ncsu_basekit/gentech/sshaft/lib/py/

- Source the setup.csh script locates as shown below

%source $PDK_DIR/ncsu_basekit/gentech/sshaft/setup.csh

- Then type the command

%gentech.py –log gen.log

- Assuming that all the above steps above followed correctly, you should see the new technology library is compiled. One key way to note success is that the file **$PDK_DIR/ncsu_basekit/lib/NCSU_TechLib_FreePDK45/tech.db** should exist. Also, the **gen.log** file should look similar to the following:

```
 Begin gentech.py
Executing: cngenlib -c --techfile
/home2/manju/pcells/Freepdk/ncsu_basekit/techfile/cni/Santana.tech pkg:pycells
NCSU_TechLib_FreePDK45
/home2/manju/pcells/Freepdk/ncsu_basekit/lib/NCSU_TechLib_FreePDK45 >& cngenlib.log
Chdir run/cds
Writing command to exe.il: pdkAppendTechfile( ?stepargs list( "../../gen.log" t 5 1 "low"))
Executing: virtuoso -log CDS.log -nograph -replay exe.il
        Begin pdkAppendTechfile
        Loading "/home2/manju/pcells/Freepdk/ncsu_basekit/techfile/FreePDK45.tf" into
"/home2/manju/pcells/Freepdk/ncsu_basekit/lib/NCSU_TechLib_FreePDK45"...
        Techfile successfully appended.
        CDF Data for Devices library successfully updated.
        Finished pdkAppendTechfile (elapsed time: 0h 0m 4s actual)
Step returned with value None
Chdir ../..
Executing: cp -r /home2/manju/pcells/Freepdk/ncsu_basekit/techfile/customvia/*
/home2/manju/pcells/Freepdk/ncsu_basekit/lib/NCSU_TechLib_FreePDK45
Finished gentech.py (elapsed time: 0h 0m 59s actual)
```

- We got two Deprecation warnings which ignored by defaults and which is not harmfull.

- And one more important thing when your starting the cadence start with the following command other wise you might have some problems using pcells and this command starts the 64 bit Virtuoso.

So the command is

%virtuoso -64

VITA

Manju Kiran Subbarayappa

Candidate for the Degree of

Master of Science/Arts

Thesis: IMPLEMENTATION OF HIERARCHICAL PREDECODER/DECODER STRUCTURE IN OPENRAM OPENSOURCE MEMORY COMPILER

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical and computer engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2013.

Completed the requirements for the Bachelor of Engineering in Electronics and communication at Sir M Visveswaraya Institute of Technology, Bangalore, and Karnataka, India in Year July 2011.

Experience:

Worked as the research assistant at Oklahoma state university, Department of Electrical engineering.