UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DESIGN OF A RISC-V PROCESSOR WITH OPENRAM MEMORIES

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

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jennifer E. Sowash

December 2019

The thesis of Jennifer E. Sowash
is approved:

Professor Matthew Guthaus, Chair

Professor Jose Renau

Assistant Professor Scott Beamer

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Design of a RISC-V Processor with OpenRAM Memories

by

Jennifer E. Sowash

Memory compilers, such as OpenRAM, are an ideal addition to most digital chip designs as they automate netlist, layout, and characterization of memories. This thesis presents the first place & route of OpenRAM memories and adds write masking, which allows OpenRAM to generate the memories necessary for processor design. To perform synthesis and place & route, we modify OpenRAM's timing and power model and physical model files. To add write masking, we alter OpenRAM's netlist, layout, and characterization. These enhancements culminate in the place and route of a small RISC-V CPU with OpenRAM memories. This CPU is 2.14x smaller than the flip-flop implementation.

Acknowledgments

I would like to thank my parents for always supporting me, and especially my mom, for convincing me to go to a BSOE talk about engineering majors at UC Santa Cruz my senior year of high school. Without this information and encouragement, I would have never gone down this path.

Chapter 1

Introduction

Static random-access memories (SRAMs) are a core component of any processor or integrated circuit (IC) design. However, designing SRAMs by hand is very time consuming, but since they have a regular structure, automation can produce size and configuration variations quickly. However, applying these variations to multiple technologies and tools is challenging. Optimization is also critical because the memory design is essential to the system performance and cost. These requirements are ripe conditions for a memory compiler, such as OpenRAM [6].

In research, many designs are limited by the availability of memories. Standard-cell process design kits (PDKs) are available from foundaries and vendors, but usually these PDKs do not provide memory arrays or memory compilers. If a memory compiler is available, it generally only supports a generic, non-fabricable process technology. Furthermore, commercial industry solutions are often not attainable for researchers due to academic funding restrictions and academic licenses. These solutions are also limited in size and configuration customization. PDKs may have options to request "black-box" memory models, but these could have limited available configurations and are not modifiable. Another option is for researchers to design their own memories, but this is time consuming and often not the point of their research. The resulting memory design usually meets the bare minimum requirements which causes the memory to be inferior and not optimized. This is where OpenRAM comes in.

OpenRAM is an open-source memory compiler development framework that provides reference circuit and physical implementations. OpenRAM is written in Python and currently supports two technologies: NCSU's FreePDK 45nm process [5] and MOSIS's fabricable SC-

MOS 0.35µm process (*SCN4M_SUBM*) [7]. Recent work will extend OpenRAM's supported technologies to include NCSU's FreePDK 15nm [4]. Futhermore, OpenRAM has been used to study SRAM design in a variety of circuit techniques and technologies [2] [12] [13].

Current SRAM design with OpenRAM requires a minimum process technology of four layers. FreePDK 45nm has ten metal layers, designated as *metal1*, *metal2*, and so on, and nine vias that connect between one layer and the adjacent layer. For example, *via1* makes a contact between *metal1* and *metal2*. In OpenRAM, *metal1* is a vertical and horizontal layer, whereas every subsequent layer alternates between always being horizontal or vertical. Conversely, *SCN4M_SUBM* has four metal layers. Support is currently dropped for the three metal layer process, *SCN3M_SUBM*, because the cells use two metal layers and the SRAM routing and power routing use an additional two metal layers.

OpenRAM can be run in front-end mode or back-end mode. Front-end mode generates spice netlists, layout views, and timing and power models. This mode comes with the additional option of skipping supply routing to save time. Front-end mode does not perform the design rule check (DRC) or layout versus schematic (LVS) check and estimates power and delay analytically. On the other hand, the back-end mode generates the same three models and views while also performing DRC and LVS and simulating power and delay. DRC and LVS checks are completed at each level of the hierarchy or at the end while the power and delay simulations are back-annotated or not back-annotated.

Before the writing of this thesis, OpenRAM was not tested with synthesis nor place and route (P&R) and did not support write masked SRAMs. Write masking adds the ability to only write part of a word. Adding these functionalities enhances OpenRAM's ability to be used with Verilog designs, such as a RISC-V processor. This allows researchers and instructors to be able use non-behavioral SRAMs in their designs. In particular, this thesis details and explains the steps required to add these functionalities.

Our contributions to OpenRAM include creating a design flow for the synthesis and place and route of OpenRAM memories in a Verilog design. For this to be possible, it is essential for us to make changes to the .lib and .lef files of which this flow depends on in order to enable OpenRAM memories to be placed and routed. A liberty file (.lib) is an ASCII representation of timing and power parameters associated with any cell. A .lef file is also an ASCII representation, but it represents the physical layout of a cell while also including design

rules for the technology. We also add the option and functionality of write masking to allow for future support of RISC-V processors, such as Berkeley's Rocket [1] or BOOM processors [3]. Finally, we demonstrate that a smaller 32-bit RISC-V processor, Clifford Wolf's PicoRV32 [11], can be P&R'd with OpenRAM's SRAMs using the FreePDK 45nm process technology. Our contributions to OpenRAM are available at:

https://github.com/VLSIDA/OpenRAM

The thesis will be laid out as follows. In Chapter 2, we will discuss the synthesis and place and route design flow including changes to .lib and .lef files. In Chapter 3, we will describe mask writing and explain how it is implemented in OpenRAM. In Chapter 4, we will discuss the PicoRV32 processor and provide the results of place and route of Clifford Wolf's PicoSOC with OpenRAM's SRAMs and with flip-flop memories. We will also provide a case study between the SRAM and D flip-flop (DFF) implemented memory model. Finally, in Chapter 5, we will give conclusions and future work.

Chapter 2

Synthesis and Place & Route Design Flow

One goal of this thesis is to place & route OpenRAM memories in a Verilog design in one of the supported process technologies. To achieve the design flow, we modify .lib and .lef files as well as write scripts to run the synthesis and place and route tools. Both FreePDK 45nm and OpenRAM have these files, but for OpenRAM, the generator, rather than the files themselves, are modified. For the register transfer level (RTL) synthesis tool, we chose Synopsys Design Compiler (DC) and for the place and route tool, we chose Cadence Innovus. RTL synthesis performs logic optimizations and turns Verilog RTL into a Verilog gate-level netlist. Innovus, the place and route tool, places all netlist gates into rows on a chip and generates wires to connect these gates together.

2.1 Design Flow

Figure 2.1 details the design flow for this thesis. A Verilog design, such as a RISC-V processor along with one or more instances of SRAMs from OpenRAM are instantiated in a top-level Verilog module. Synopsys DC accepts .db files while Cadence Innovus accepts .lib files. A .db file is a database format file created when Synopsys Library Compiler (LC) converts an ASCII .lib file to a .db file. Synopsys DC runs a Tcl script, which is a sequence of commands native to the program running it and is used for reproducibility. This script finishes

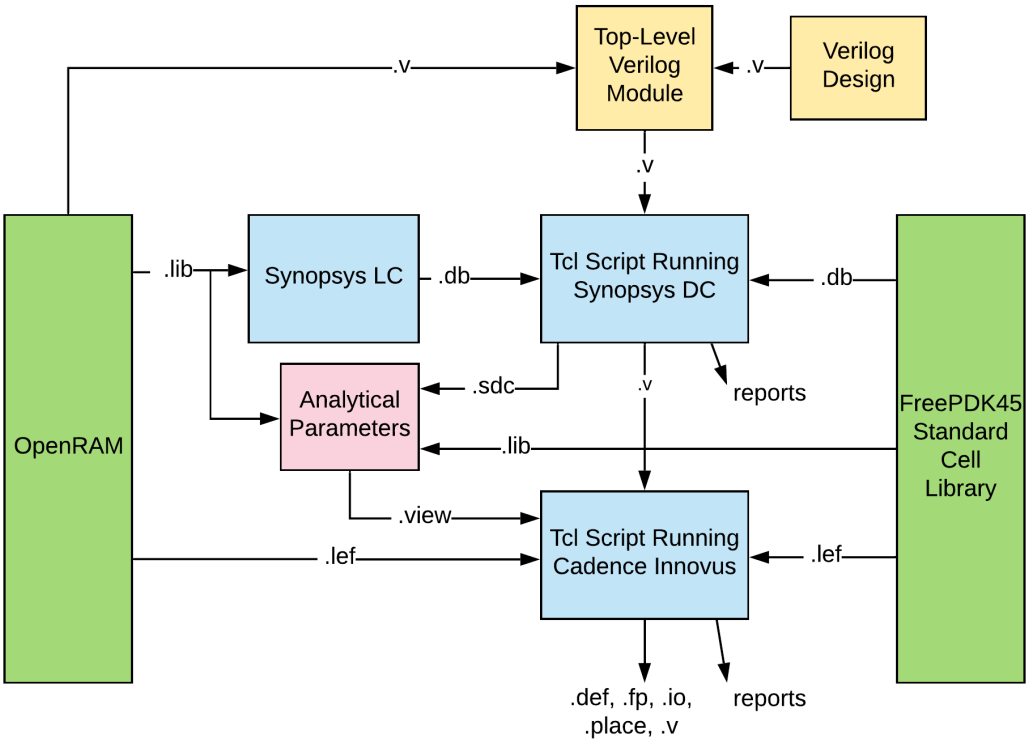


Figure 2.1: A block diagram of the synthesis and place & route design flow with OpenRAM and FreePDK 45nm as the cell library macro and process technology, respectively.

by generating a Verilog netlist file and an .sdc file for Cadence Innovus. An .sdc file is an ASCII text file which contains the constraints and timing assignments for the design.

The .lib files and the .sdc file from DC are used in Innovus to create a .view file. A .view file is an ASCII file used for multi-mode multi-corner (MMMC) setup for timing analysis. The .view file, the .lef files from OpenRAM, and the process technology's .lef file initialize the design into memory. Then, these files are used in the Tcl file along with Innovus commands to place and route the design.

2.2 Synthesis Tool: Synopsys DC

Synopsys DC requires hardware description language (HDL) files of the design and a setup Tcl script for the process technology. The setup Tcl script includes the target standard cell

library and the link SRAM library in .db format. Then, for reproducibility, a Tcl compile script contains the necessary commands for synthesis in DC. In this file, there is a top-level Verilog file which contains all necessary module instantiations, including the SRAMs, for the highest level. Moreover, in a Tcl script, all Verilog files are listed in order to be read by DC if they are part of the module instantiations. SRAMs are not included in this list as they are library components and, thus, referenced by their .db file. A Tcl script can include timing, power, and area results and analysis as well as output a .sdc file and Verilog file for the gate-level netlist.

2.3 Place and Route Tool: Cadence Innovus

Cadence Innovus places and routes any gate-level Verilog netlist. Innovus performs placement, power routing, and design routing as well as providing timing, power, and DRC reports after the design is loaded into program memory. The design is placed and routed in either physical mode or in analytical mode where analytical mode considers timing paths. By only providing the .lef file and the gate netlist Verilog file, the design is run in physical mode. Analytical mode requires an additional .view file that contains an .sdc file from DC synthesis and the .lib files. Also, the supply names must be set before loading the design into memory.

Innovus must be set up to work with FreePDK 45nm standard cell libraries and OpenRAM before a design can be properly placed and routed. While setting up the standard cell libraries is straightforward, setting up OpenRAM's libraries requires more effort. Ensuring that a design can be placed and routed requires reading in the .lef files, ensuring the blockages are correct, ensuring pin routing works, and power routes to the power pins. The first three steps will be covered in this chapter. The latter step and the steps to place and route a design will be covered in Chapter 5 as they do not require changes to .lib or .lef files.

2.4 .lib File Changes

OpenRAM generates a .lib file for every process, temperature, and voltage (V_{DD}) corner. Every NMOS and PMOS process corner can be slow (S), fast (F), or typical (T). These corners are expressed as SS, FF, or TT where the first letter represents the NMOS corner and the second letter represents the PMOS corner. These corners describe whether the carrier mobility is faster or slower than typical. There are also variations, such as SF or FS, but these are not

included to speed up the run time of OpenRAM. The temperature corners set in OpenRAM are 0°C, 25°C, or 125°C. The voltage corners are 10% higher or lower than the supply voltage. For 45nm, this value is 1V, so corners exist for 0.9V, 1.0V, 1.1V.

We make changes to OpenRAM's .lib files to in order for Innovus to read the design into memory. These changes are as follows. First, the naming scheme of the .lib file must match the naming scheme of the .lef file exactly. The pins for active low chip select (*csb*) and active low write control (*web*) were named *WEb0* and *CSb0* in the .lib file and *web0* and *csb0* in the .lef and .v files. The number at the end of the pins represents the port number. Our solution is to change all letters to lowercase in OpenRAM's lib.py, which generates these changes for the .lib files. Secondly, all .lib files must follow standard unit capitalization rules. For example, millivolts are written as mV, nanoseconds are written as ns, and microwatts are written as uW. We capitalize the symbol for voltage in OpenRAM's lib.py since it was originally written as a lowercase *v*.

2.5 .lef File Changes

The changes to the .lef file are characterized by the changes to Oklahoma State's FreePDK 45nm design flow and by the changes internally to the modules in which OpenRAM generates memories and their corresponding files.

2.5.1 Technology .lef File Changes

Technology .lef files provide design rules and resistances for each metal layer and via type. This file sets requirements, such as unit type, design rules for metal layers, and manufacturing grid sizing. Moreover, .lef files provide standard cell macros for the process technology. A macro is a block that can be easily reused among designs. Examples of macros are D flip-flops, gates, and inverters. These macros are part of the standard cell library provided by the technology.

There are two main problems with the FreePDK 45nm .lef files. First, single-height cells should have site symmetry *Y* and macro symmetry *X Y*. The site defines the placement site in the design. This placement site gives the placement grid for a family of macros, such as I/O or core. Therefore, we add *SYMMETRY Y* to *coreSite*. The macros given in the technology .lef

file have *symmetry X Y*, so they are correct. The second issue is that the cell is not an integer multiple of the site. This size parameter given as height by width specifies the dimension of the site. We solve this issue by dividing the cell width by 2 to make it an integer of site.

2.5.2 Cell Library .lef File Changes

The cell library .lef file provides the cell boundary and pin locations as well as the blockages for each metal layer in a hard macro. A hard macro allows the P&R tool to access the pins; however, the macro itself cannot be altered. However, it can be moved, flipped, or rotated. An example of a hard macro is a memory module, such as OpenRAM's SRAMs.

Our changes to the .lef files in OpenRAM are divided into three phases: preliminary changes in OpenRAM to remove warnings or errors, intermediate changes to allow the SRAM macro to be placed and routed, and advanced changes to improve the .lef file.

Preliminary Changes

The preliminary changes to the .lef files in OpenRAM are to remove a warning about no related standard cells and to make the capacitance units match the FreePDK45 .lef file. The first change requires removing *SITE MacroSite* from the macro as it is not needed in a cell library .lef file. The second change requires dividing capacitance in lef.py in OpenRAM by 1000 since the capacitance in the technology .lef file is in units of pF while the cell library .lef file is in units of fF.

Intermediate Changes

The intermediate changes require fixes to place the standard cells and to be able to use NanoRoute. NanoRoute is Cadence's proprietary digital routing tool which performs routing and interconnect optimization to meet timing, area, signal integrity, and manufacturability convergence. Furthermore, OpenRAM runs lef.py which contains all the functions necessary to create the SRAM macro's .lef file.

The first intermediate change requires decreasing the size of the SRAM in order to place the standard cells. Innovus originally placed a $0.3\mu m^2$ SRAM that caused the program to be killed because it ran out of memory. This occurred because the code in lef.py multiplied each blockage value by 2000 database units. Defining the database units to be 2000 in the .lef file

means that 1μm is equivalent to 2000 database units. Database units are used as a conversion factor in an external program to multiply with micrometers in the .lef file. Therefore, we remove the multiplication of micrometers by database units inside lef.py to correctly size the SRAMs in the generated output.

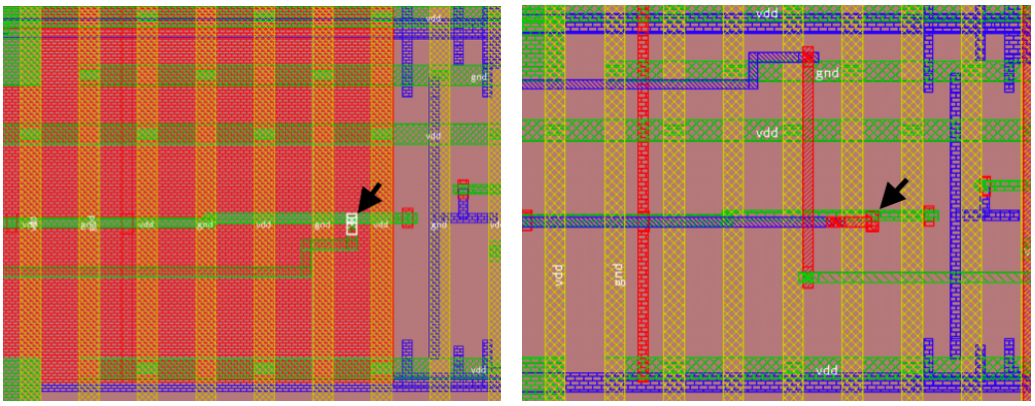
The second change requires fixing the off-manufacturing grid error in order to use NanoRoute. Python floats, by default, leave extraneous decimals at the end of numbers, such as 0.002500001. Therefore, the locations of blockages are no longer considered multiples of the 0.0025μm manufacturing grid even though they are. Our solution is to round float values to four digits in lef.py to match the manufacturing grid in the generated output.

Advanced Changes

Two advanced changes are made to OpenRAM's .lef file. These changes require removing blockages over pins and adding support for polygons. While detailed blockages represent the routing in each metal layer, total blockages apply large metal blockages over areas to account for the lack of detailed routing. Shown in Figure 2.2, after substituting total blockages for detailed blockages, it is clear that OpenRAM's library cells, such as D flip-flops, are missing detailed blockages for interior connections. These missing blockages are polygonal shaped. From Figure 2.4, after adding polygonal blockages to the .lef file, the library cells display all polygonal blockages.

In a .lef file, polygons are written as *POLYGON pt pt pt pt* where each point (pt) is a pair of coordinates that specifies a corner. The number of points can be any number greater than three, and each point must be a 45° or 90° angle. Moreover, each shape, either polygonal or rectangular, is composed of a list of counter clockwise coordinate pairs. The last point in the list is always the same as the first point. Therefore, there will always be one more point than the total number of corners. Thus, in OpenRAM, polygons are defined as any shape that does not have four 90° corners.

Conversely, rectangles are written as *RECT pt pt* in a .lef file where the two points are at opposite corners with 90° angles. For example, the first point is the coordinate pair in the lower left corner and the second point is the coordinate pair in the upper right corner. The 90° angle causes each coordinate to share one point with adjacent coordinates. Therefore, the five points that compose a rectangle are listed as two points.



(a) A large *metal2* blockage (red) over the *metal2* *csb0* pin caused DRC errors upon routing.

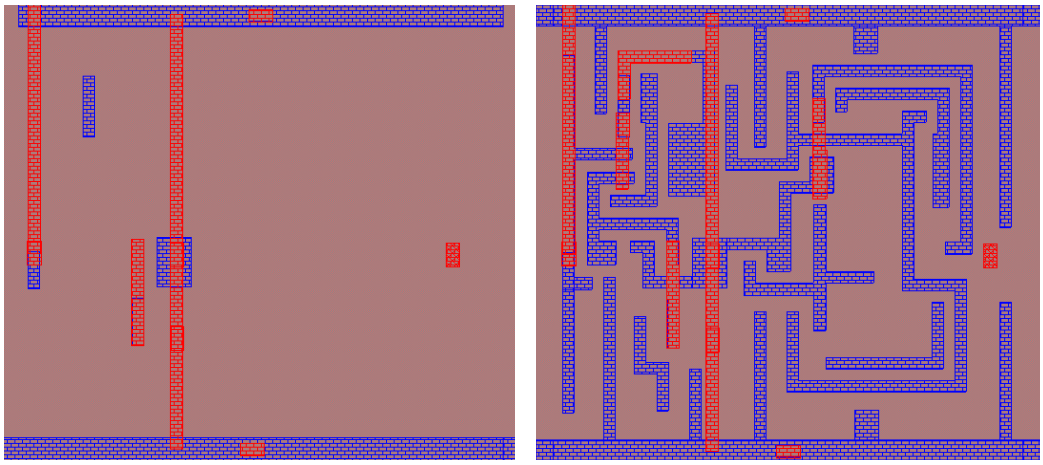
(b) The large *metal2* blockage has been replaced by detailed *metal2* blockages. The pin *csb0* is now accessible and has been routed to.

Figure 2.2: Innovus layout showing before (a) and after (b) demonstrating the removal of the *metal2* total blockage over pin *csb0* (location denoted by the black arrow) in favor of detailed blockages.

The first advanced change is shown in Figure 2.2 where total blockages are converted into detailed routing blockages. Previously, OpenRAM placed *metal1* or *metal2* blockages over most of the library cells since there initially did not exist a way to parse the difference between a blockage and a pin. Since library cells do not use *metal3* for routing, large *metal3* blockages did not exist. A large metal blockage is shown in Figure 2.2(a) where there is a *metal2* blockage over the flip-flop. This causes the pin *csb0* to be blocked leading to a DRC error when routing is attempted in Innovus.

```
def getRectangleBlockages(self,layer):
   blockages = []
   shapes = self.getAllShapes(layer)
   for boundary in shapes:
      ll = vector(boundary[0],boundary[1])
      ur = vector(boundary[2],boundary[3])
      rect = [ll,ur]
      new_blockage = rect
      blockages.append(new_blockage)
   return blockages
```

Figure 2.3: Python code of the *getRectangleBlockages()* function to return rectangular blockages.



(a) D flip-flop with only rectangular blockages and pins.

(b) D flip-flop with pins and rectangular and polygonal blockages.

Figure 2.4: Innovus view of a D flip-flop showing before (a) and after (b) polygonal blockages are added. The internal *metal2* routing (red) out of the flip-flop is shown on the left side of each flip-flop.

When the module is a library cell, blockages are extracted with the *getRectangle-Blockages()* function from Figure 2.3. This function returns all blockages on a given layer in the form *[[llx, lly], [urx, ury]]* where *llx* stands for lower left x-coordinate and *ury* stands for upper right y-coordinate. These coordinates form a rectangular blockage which is expressed as a list of vector pairs. This new rectangular blockage is added to the list of blockages. Once, all shapes in the layer have been converted to blockages, the list of blockages is returned. Once blockages for every cell library metal layer are found, the blockages are printed to the .lef file. The final result is shown in Figure 2.2(b).

Adding functionality to express polygons is another major change to OpenRAM's .lef file. Cells often include polygonal routing. Any blockage shown in Figure 2.4(b) and not in Figure 2.4(a) is a polygonal blockage and, therefore, has any number of points not equal to five. Since the layout is from a place and route tool, no n-well, p-well, or polysilicon layers are shown. As OpenRAM's .lef file writer only wrote rectangular blockages, support for polygonal blockages has to be added to express detailed blockages. As .lef files support polygonal identifiers, polygonal blockages stayed polygons instead of being decomposed into rectangles with an algorithm.

The modification and addition to OpenRAM's code is critical to expressing shapes

```
def getPolygonBlockages(self,layer):
   blockages = []
   shapes = self.getAllShapes(layer)
   for boundary in shapes:
      vectors = []
      for i in range(0, len(boundary), 2):
         vectors.append(vector(boundary[i], boundary[i+1]))
      blockages.append(vectors)
   return blockages
```

Figure 2.5: Python code of the *getPolygonBlockages()* function to return polygonal and rectangular blockages.

with more than two points. The *getPolygonBlockages()* function from Figure 2.5 replaces the *getRectangleBlockages()* function. The new *getPolygonBlockages()* function returns all blockages on a given layer. Each blockage is a list of vector pairs where the number of pairs varies. In Figure 2.3, only two pairs of points, *ll* and *ur*, are allowed. Moreover, new functions are required to rotate and transform rectangles in the hierarchy for output into the .lef file. However, these functions in OpenRAM did not consider more than two points. Therefore, we modify functions to support any number of pairs and create a function to transform and rotate a polygon.

Once we write the code to include polygonal blockages for each metal layer, we make a function in lef.py to determine the shape identifier. The shape identifier, either *POLYGON* or *RECT*, is based upon the number of pairs in each blockage. When the number of pairs is two, the blockage is a rectangle, and the *RECT* shape identifier is used. These identifiers along with the pairs of points are printed to the .lef file as a list of numbers. The result is shown in 2.4(b).

Chapter 3

Implementation of Write Masking

Write masking is a necessary feature to be able to support advanced architectures. For example, since RISC-V processor caches often require byte writing, mask writing is used to implement the memories for these processors. If the word size is 32-bits and the write granularity is a byte, meaning that the write size is 8-bits, there are 4 write mask bits where each write mask bit corresponds to 8-bits. When *wmask[3]* is selected, only bits *31:24* are written to.

In this chapter, we will first discuss OpenRAM's SRAM structure and port types. Then, we will describe how write masking is added to OpenRAM in order for the functionality to be supported by OpenRAM.

3.1 OpenRAM's Structure

Following Figure 3.1, OpenRAM's SRAMs consist of a six transistor (6T) bit cell array in the upper right. Although, no flip-flops are shown in Figure 3.1, below the write driver array are the data input flip-flops, and to the left of the control logic are the control logic flip-flops. Moreover, to the left of the address decoder are the address flip-flops. Additionally, there is one address decoder, wordline driver, column multiplexer (column mux) array, and control logic module per port. There is also one precharge array and sense amplifier (sense amp) array per read port and one write driver per write port.

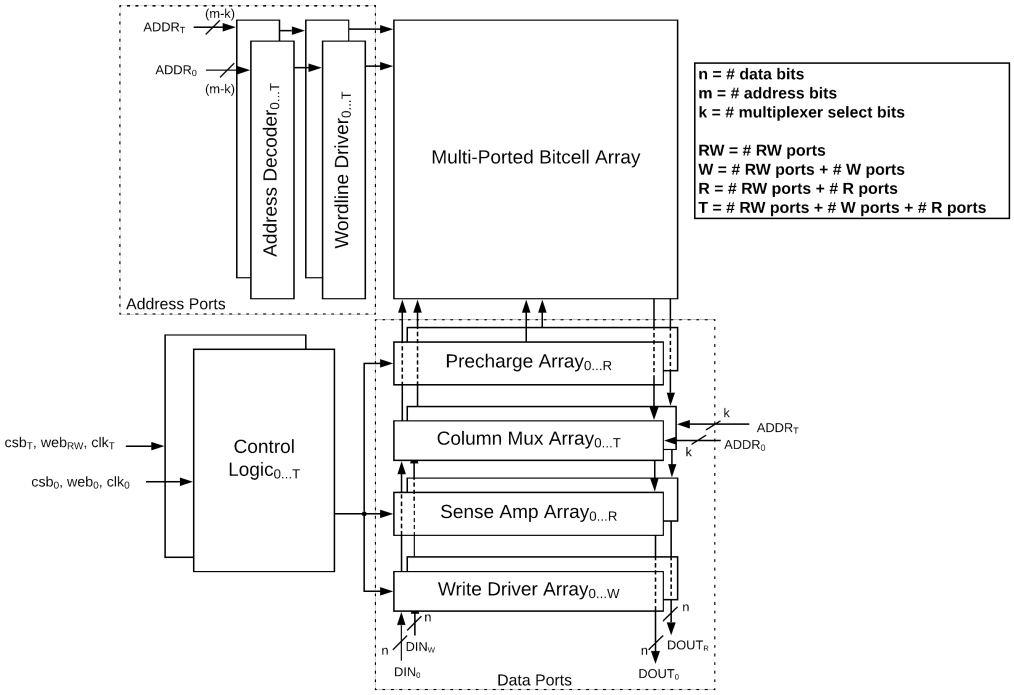


Figure 3.1: Block diagram of a mutli-ported SRAM in OpenRAM without write masking. The block locations for first port approximately follow the actual layout. Additional ports are symbolic and serve to demonstrate the port naming scheme [8].

3.1.1 Data Port and Control Logic

A data port consists of precharge, column mux, sense amp, and write driver arrays. The precharge array precharges the bitlines to drive a "1" on both bitlines. The precharge happens during the first phase of the clock cycle. On the other hand, the sense amplifier senses a voltage difference between the bitlines when a read operation is performed to improve performance.

The column multiplexer is an optional block that is used when there are multiple words per row. It uses the lower address bits to select the associated word in the row. When there is more than one word per row, the sense amp and write driver instances inside the sense amp and write driver arrays are spaced out so that the arrays are the same width as the column mux array.

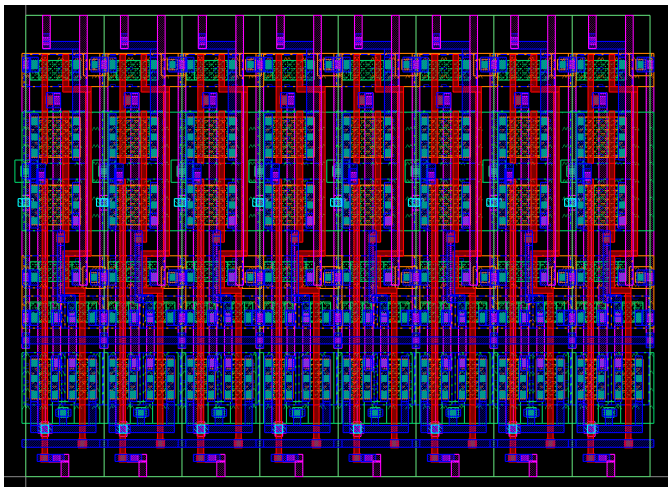


Figure 3.2: A write driver array consisting of 8 write drivers in OpenRAM corresponding to an 8b x 16 SRAM.

The write driver sends the input data signals onto the bitlines for a write operation when the write enable signal from control logic is "1". The write driver array is tri-stated to write to the bitline through the column mux. There is one write driver for each input data bit which comprises the write driver array as shown in Figure 3.2.

The control logic includes an externally provided active low chip select (*csb*), active low write enable (*web*), and a system clock (*clk*) incorporated into a standard synchronous memory interface. The input control signals, *csb* and *web*, are stored using D flip-flops to ensure the signals are valid for the entire clock cycle. During a read operation, data is available after the negative clock edge. Furthermore, the internal control signals are generated using a replica bitline (RBL) structure for the timing of the sense amplifier enable and output data storage [6].

3.1.2 Address Port

The address decoder takes, as inputs, the row of address bits and asserts the appropriate wordline. This ensures that the correct memory cells are read or written. The wordline driver acts as a buffer between the address decoder and the bit cell array. They are sized based upon the width of the memory array to drive the row select signal across the bit-cell array.

3.1.3 Port Types

OpenRAM currently supports both single-port and dual-port configurations. The single-port types are read (R), write (W), and read-write (RW). RW ports perform both read and write operations, R ports perform only read operations, and W ports perform only write operations. OpenRAM also has custom bit cells for multiple port (multi-port) configurations. These configurations are 1 RW; 1 R, 1 W; and 1 RW, 1 R. If other configurations are desired, these configurations can be created as long as the number of ports does not exceed two, hence, a dual-port configuration. OpenRAM's layout currently only supports up to two ports because more metal layers are required for configurations with more ports. Regular placement of ports and resizing transistors to satisfy read and write stability allow for these various configurations to arise. However, since these cells are not custom-made, they are not area-efficient, but may be better than choosing to make the design with flip-flops.

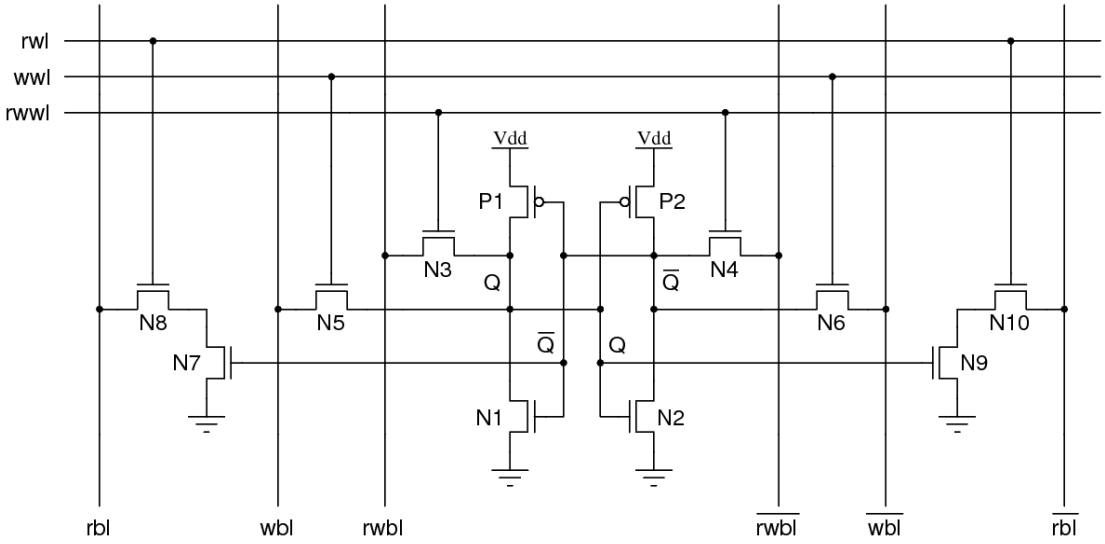


Figure 3.3: Bitline locations in multi-port supporting RW, R, and W ports [8].

From Figure 3.3, a multi-port SRAM has two access transistors, two sets of bitlines, and one set of word selection lines per port. The single inverter pair, *N1*, *P1* and *N2*, *P2*, stores a bit. The read port consists of access transistors *N8* and *N10* with one additional transistor per bitline *N7* and *N8*. Meanwhile, the write port consists of access transistors *N5* and *N6*, and the RW port consists of access transistors *N3* and *N4* [8]. Therefore, the RW and W ports look very similar in layout since the only difference is that the transistor locations are reversed.

3.2 Write Mask Overview

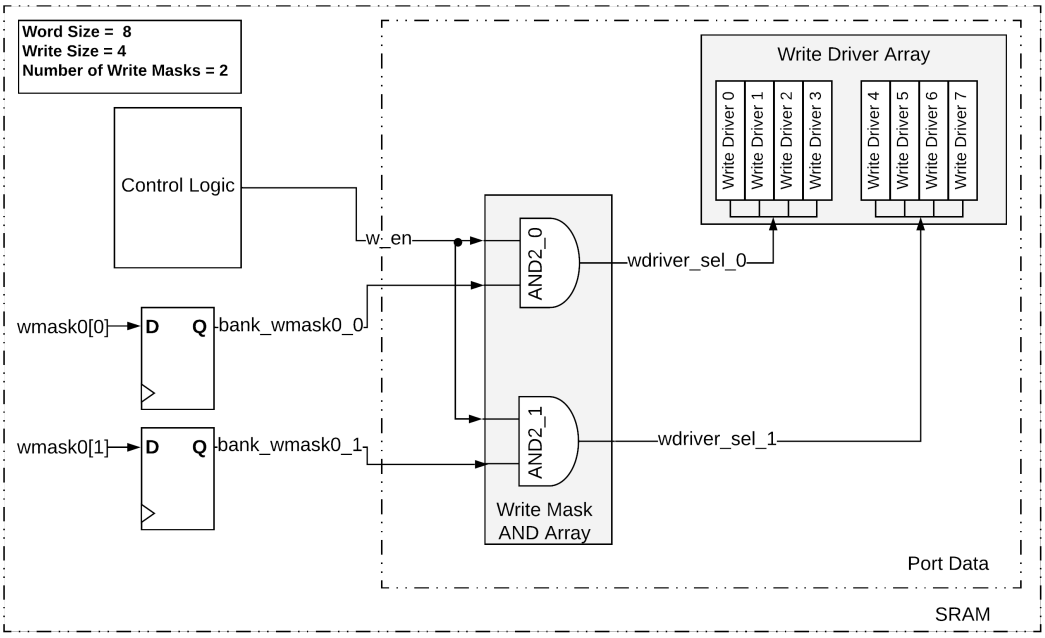


Figure 3.4: Block diagram of write masking in OpenRAM. For simplicity, the port module is removed, and for accuracy, the first number in each pin name represents the port number while the second, or only, number represents the bit.

Figure 3.4 describes how write masking is implemented. Two variables are important for write masking: write size (*write_size*) and number of write masks (*num_wmasks*). Write size specifies the size of the write, such as 8-bits. Therefore, this number must be an integer multiple of the word size and be between one and half of the word size. If it's more than half of the word size, it's either not an integer, or the whole word will always be written removing the need for a write mask. The number of write masks is calculated by the word size divided by the write size. If this value is not an integer, OpenRAM will return an error because the user specified a non-valid write size. Currently, if there is more than one write port, such as in a 1 RW, 1 W memory, there will be one write mask per write port since each port shares the same configuration file.

Write masking is added to OpenRAM as an optional feature and gets turned on when the property *write_size* is set in the configuration file. In other words, the functionality of the

current OpenRAM implementation should not change when no write size is specified. Write masking is divided into four steps: adding a write mask to Verilog, implementing the spice netlist, adding the layout of the write mask, and adding the write mask bus to the .lib file. While the first three steps will be described in the sections below, adding the write mask bus to the .lib file involved creating a bus type called *wmask*. This type defines the size of the write mask, the data type which is bit, and the base type which is array. After creating the type, we add the write mask bus to the .lib file. This bus contained information, such as the capacitance, direction, and setup and hold timing.

We complete testing in three parts: testing the Verilog with a testbench, functional testing, and testing the layout. The Verilog testbench ensures that the digital simulation output is correctly written while functional tests assess the spice model's ability to pass transient circuit simulations. Layout testing involves ensuring that the added components pass DRC and LVS checks.

3.3 Write Mask Verilog Implementation

To gain a basic understanding of write masking, we create the Verilog module first. To generate Verilog code, OpenRAM writes to a Verilog file based upon the memory specifications from the configuration file. In the Verilog file, write masking adds a parameter *NUM_WMASKS*, an input write mask bus, and a write mask register. The input write mask bus is sized to be equal to the number of write masks. Moreover, the code for a write mask is added to each write port.

The Verilog implementation of the write mask is verified with a testbench. The testbench performs a series of partial writes and compared the assertion to reads to the respective address.

The Verilog code in Figure 3.5 demonstrates how a write mask functionally works. The memory in this example is an 8b x 16 SRAM with 1 RW port and 4-bit write masks. A write operation in an RW port occurs when the signals *web0* and *csb0* are low. Therefore, since there is only one port, port 0 is being written to. For each write mask, the corresponding data input bits are written to the corresponding memory bits at that address. Without a write mask, the code would differ in that the entire data input bus would be written to the memory at the particular address.

```
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
  always @ (negedge clk0)
  begin : MEM_WRITE0
    if ( !csb0_reg && !web0_reg ) begin
         if(wmask[0])
           mem[ADDR0_reg][3:0] = DIN0_reg[3:0];
         if(wmask[1])
            mem[ADDR0_reg][7:4] = DIN0_reg[7:4];
     end
    end
```

Figure 3.5: Verilog code demonstrating what a write operation looks like with write masking.

3.4 Write Mask Netlist and Functional Tests

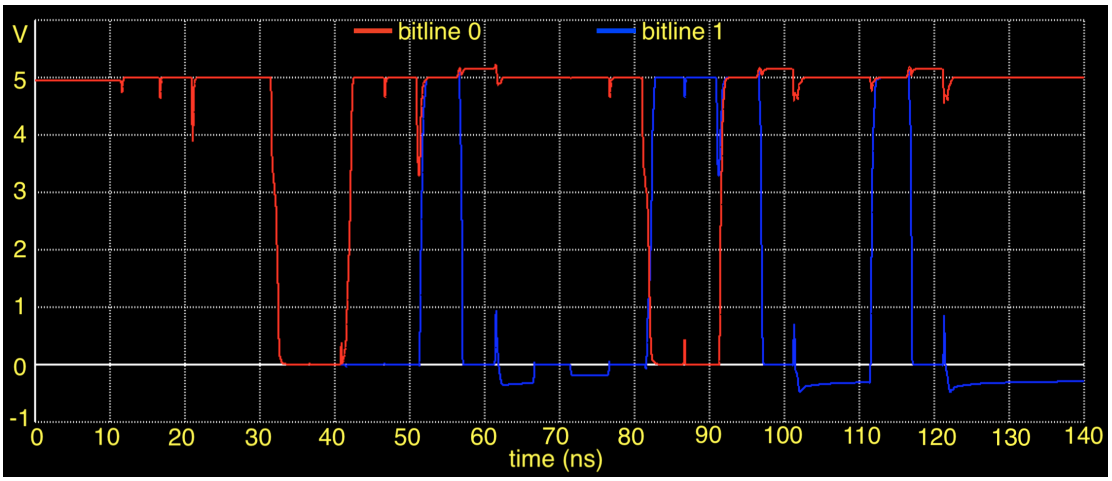
An important component to any memory compiler is a spice netlist, used for verification, simulation, and testing. Therefore, the write mask must be added to the spice netlist generated by OpenRAM. Additionally, the netlist must be verified to be correct through functional testing.

3.4.1 Write Mask Netlist

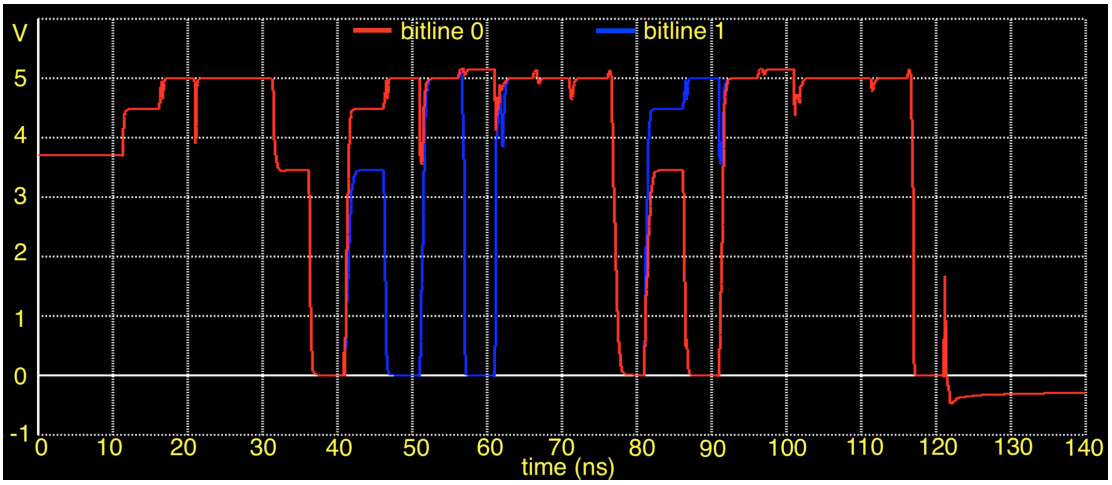
There are three steps to add any circuit to a spice netlist in OpenRAM. These steps are adding the pins, adding the modules, and connecting the instances. First, the pins must be added to the lower level of the hierarchy, which creates the pin names in the spice netlist. In the next step, the module is created, which generates the cell type, such as a write driver. Here, any cell parameters, such as size, are given for the layout. Additionally, each module with a different layout is given a different name. Thirdly, the module is added to the netlist, which adds a subcircuit (subckt) to the subckt hierarchy.

In each higher level of the hierarchy, we add and connect instances, such as a write driver array, which join the pins. Thus, creating and adding instances is the precursor to being able to place the layout. In each level of the spice hierarchy, the net names are different for readability. The only exception is naming *vdd* and *gnd* pins. Greater detail about the write mask's hierarchy will be given in the layout portion of this thesis.

3.4.2 Write Mask Functional Tests



(a) Without precharging on writes, the bitlines contain incorrect values leading to these values being written after the negative clock edge.



(b) Precharging sets the bitlines to the correct value by the negative clock edge where a write operation begins.

Figure 3.6: ngspice's results showing the lowest two bits from bitline 0 (red) and bitline 1 (blue) without precharging (a) and with precharging (b). At 70ns and 100ns, the value of bitline 1 stays "0" while precharging pulls the bitline up to "1". Similarly, at 80ns and 120ns, the value of bitline 0 stays "1" while precharging pulls the bitline down to "0".

The functional tests are a series of random writes, partial writes, and reads to random addresses with random input data. Partial writes are added to test a netlist which has write masking. Partial writes write the selected bits of the write mask to a previously initialized address where a whole write takes place since memory is not initialized.

During functional testing, we discover an issue where the wrong values are written when a partial write follows a read to the same address. To determine the issue, we use ngspice, a spice simulator, to plot the results from the bitlines in the functional test as shown in Figure 3.6. From Figure 3.6(a), the part of the word that is not written has floating bitlines. When these bitlines have a value from a previous read, the bit cell contents are destroyed because the capacitance of the bitline is significant enough to pull down the bit cell. The solution is to precharge the writes in addition to the reads because every bit in the bitline is no longer always getting written. From Figure 3.6(b), with a clock period of 10ns, at the positive edge of the clock between 60 and 70ns, the precharge pulls bitline 1 up to "1". Without the precharge as in 3.6(a), the value remained "0" at the negative clock edge, and the incorrect value is written due to a previous read pulling the bitline down.

3.5 Write Mask Layout

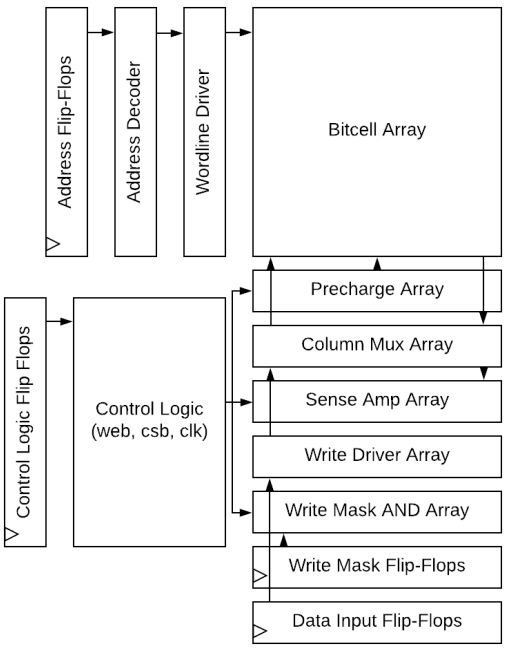


Figure 3.7: Block diagram of a single-port SRAM with write masking where the block locations approximately follow the actual layout. Here, flip-flops, the write driver array, and the write mask AND array are included.

After adding a write mask to the spice netlist and verifying it with functional tests, the next step adds the write mask to OpenRAM's layout. OpenRAM saves the layout as .gds files for both technologies and converts the layout to .mag files for SCMOS technology when testing. A .gds file is a binary file format that represents geometric shapes, text labels, and other information about the layout in hierarchical form while a .mag file is a Magic file format which contains the same information, but with the hierarchy separated by file. Glade [9] views .gds files while Magic [10] views .mag or .gds files. Glade is a VLSI layout and schematic editor while Magic is a VLSI layout tool and editor.

The layout of a single-port of the write mask is shown in Figure 3.7. The write mask AND array goes below the write driver array while the write mask flip-flops are placed between the write mask AND array and the data input flip-flops. The write mask flip-flops will never be longer than the data input flip-flops because the number of write masks must be between half of the number of data input bits and the number of data input bits. This is equivalent to half the word size. Furthermore, the logic for the layout of the write mask is added to OpenRAM in four levels of hierarchy.

3.5.1 Write Driver Array

Each layout in this chapter, except for the multi-port layout, is of the same 8b x 16 SRAM with 1 RW port and 4-bit writes. Although a typical implementation will include byte writing, for simplicity of the images, an 8-bit word, 4-bit write SRAM is chosen. For compatibility with write masks, we alter the write driver array to have multiple enable layout pins instead of a single layout enable pin. This change is shown in Figure 3.8. In the figure, there are two enable wires since there are two write masks in higher levels of the hierarchy where each write mask corresponds to 4 write drivers, or 4-bits.

Since, previously and without a write mask, the enable pin spanned the entire write driver array, the enable pins in each write driver overlapped when there was no column multiplexer. However, with more than one word per row, the other arrays in the data port are spaced out to match the column multiplexer array, which removes the enable overlap. Here the enable pin in the write driver array connects the individual drivers instead. Therefore, the enable pin in the custom-made write driver cell is shortened to add spacing between individual write drivers. The *metal1* enable pin is shortened to extend to the edge of other *metal1* wires in the layout to

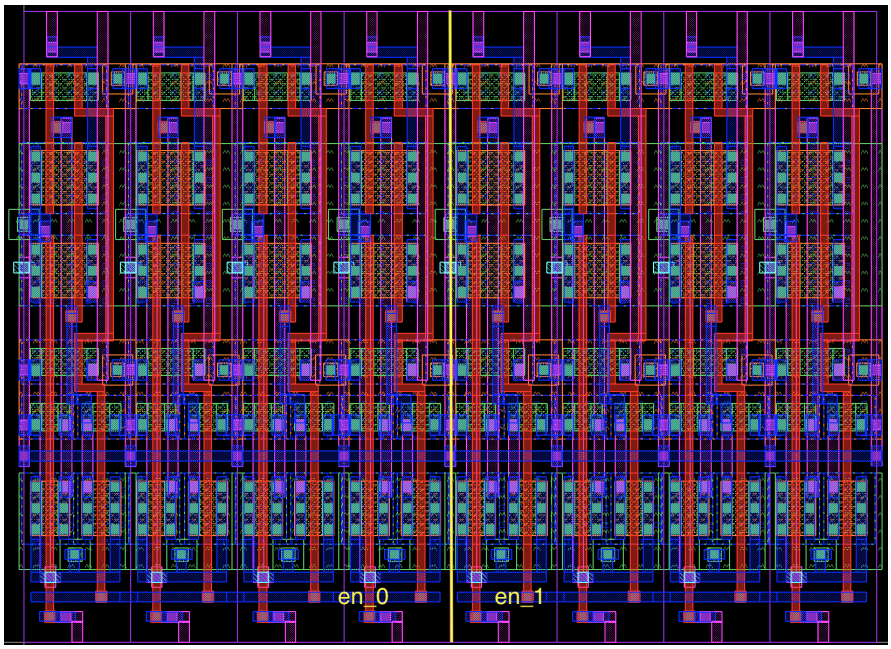


Figure 3.8: FreePDK 45nm layout of the write driver array in an 8b x 16 SRAM with 4-bit writes and 1 RW port. This configuration has no column multiplexer. The yellow line indicates the division between the 8 write drivers into 4-bits each for the write mask AND array.

guarantee no *metal1* DRC spacing issues.

3.5.2 Write Mask AND Array

Adding the write mask AND array, shown in Figure 3.9, involves spacing out the AND gates, sizing the gates, and providing connections between gates. In each layout image, the pin names at the respective level of hierarchy are shown in yellow. First, the AND gates must be placed above their respective enable connections in the write driver. Therefore, if the width of the AND gate is longer than the number of write drivers and spacing between those write drivers, the AND gate cannot be laid out.

To drive the required number of write drivers, the AND gates need to be sized appropriately. An AND gate of size 1 cannot scale to drive large loads. An AND gate consists of a single inverter and a NAND gate. Each inverter in each AND gate drives a number of write drivers equal to write size; thus, the fanout is equal to the write size. Hence, the size of the AND gate in the write driver AND array must be equal to the write size divided by the stage effort.

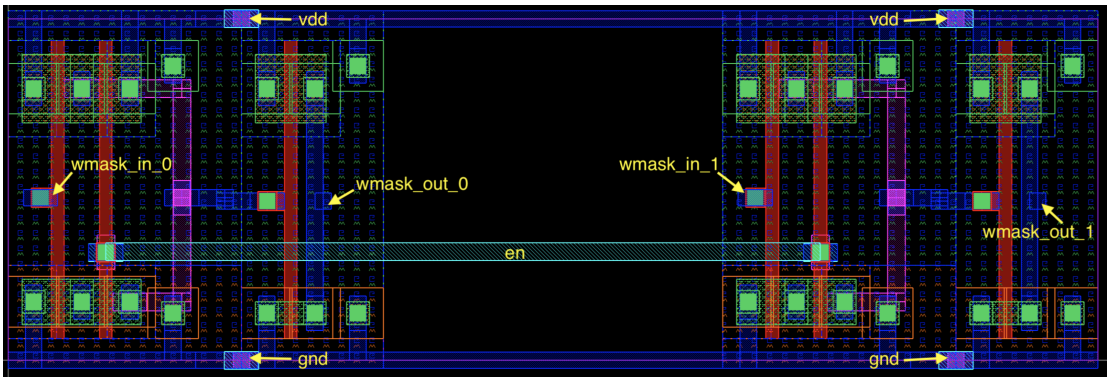


Figure 3.9: FreePDK 45nm layout of the write mask AND array.

However, a single inverter module will be slow when the size becomes large since the NAND gate will always be size 1.

We create the *pdriver* module to instantiate a specific number and polarity of inverters sized to drive a certain load. Therefore, the *pdriver* module is added to the *pand2* module, which is a 2 input AND gate, in place of the single inverter module. The *pand2* module employed the write driver AND array now consists of a NAND gate followed by a varying, negative number of inverters.

The input and output pins of each AND gate in the write driver AND array are added to this hierarchical level, and the *B* pins are routed together to form the enable pin. The *en* pin is routed horizontally in *metal3* since there are *metal1* conflicts and *metal2* is a vertical layer. This requires a via from *metal1* to *metal2* and from *metal2* to *metal3* to reach *metal3*. Furthermore, the enable wire is made a pin. Otherwise, the enable pins will appear disconnected at higher levels and routing will be impossible since a wire only exists in lower levels. Additionally, the *A* and *Z* pins from the AND gates are copied as layout pins to the write driver AND array and assume the pin names, *wmask_in_*{} and *wmask_out_*{}, for the pins that exist at this level of hierarchy.

To avoid DRC errors from larger *pdriver* modules, the *vdd* and *gnd* pins are placed at the boundary between the NAND gate and the *pdriver* module. Placing the pin in the middle of the *vdd* and *gnd* wire can cause DRC issues with larger *pdriver* modules. Finally, for redundancy, the *vdd* and *gnd* pins must be connected together in case of a via manufacturing error.

3.5.3 Data Port

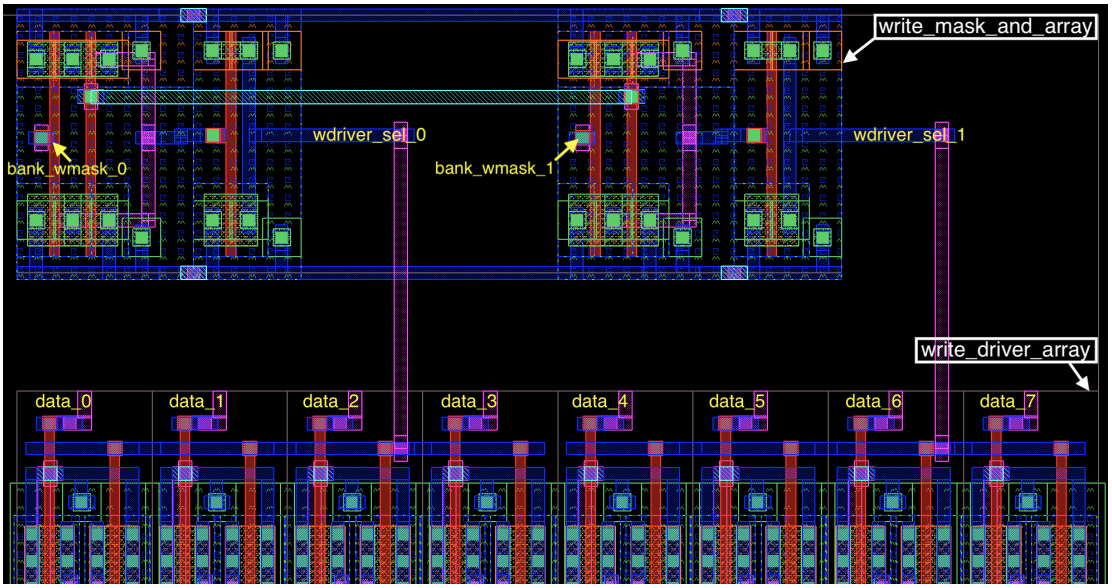


Figure 3.10: FreePDK 45nm layout of the bottom of the data port with boxed white module names.

Shown in Figure 3.10, the data port level requires placing the write mask AND array and routing between write mask AND array and the write driver array. The layout pins from the write driver array and the write mask AND array must have been previously copied from the lower levels to appear at this higher level. For ease of layout, the instances are upside down at this level of hierarchy; thus, the write mask AND array are placed below the write driver array, which is flipped at the SRAM level.

One challenging element to adding the write mask is discerning a way to route the write driver select pins. Since routing left of the output of the AND gate will cause DRC issues, the solution is to add the connection from the output of the AND gate to the enable of the write driver after the next bitline data pin. This location would cause no DRC errors in both supported technologies. The location of the next bitline data pin is determined by finding the position where the write driver's bitline data pin, *data_{}*, is right of the AND gate's output pin, *wdriver_sel_{}*. If this position is not found, the write mask AND array is too long for the corresponding number of write drivers. Assuming the connection is found, it is routed horizontally from the output pin in *metal1* and routed vertically to the enable pin in *metal2*.

Special attention is paid to *metal2* to *metal2* spacing to avoid DRC errors. Furthermore, a suitable location for the next write mask bit is always separated by the write size. Therefore, every additional bitline location is found by multiplying the write mask bit by the write size and adding it to the appropriate bitline data number.

3.5.4 Bank

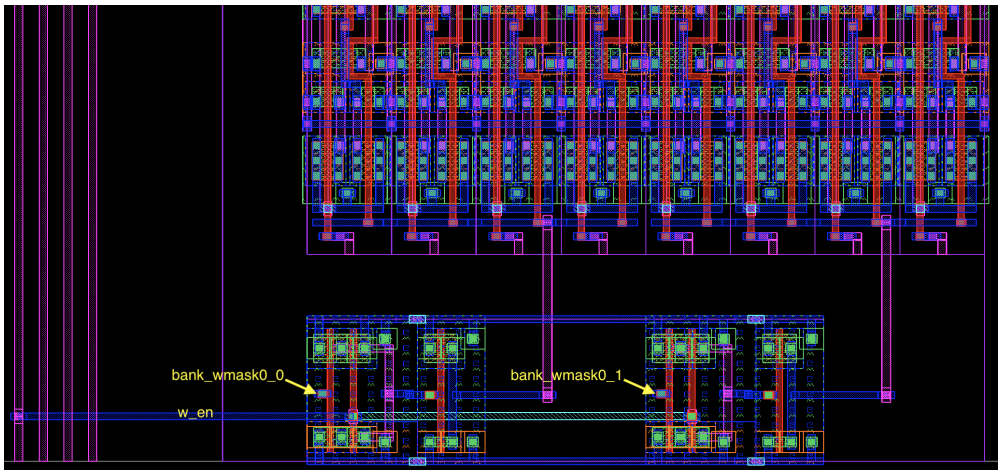


Figure 3.11: FreePDK 45nm layout of the bottom of bank. Relevant pin names are replaced with names indicating the port number.

Alterations at the bank level, shown in Figure 3.11, involve bringing layout pins from the data port level to the bank level and routing the write enable. The *bank_wmask_*{} pins are copied to *bank_wmask0_*{} pins to match the netlist names. This higher level pin includes the port number, which in this case is 0, to signify which port the bank belongs to. Moreover, the *w_en* pin from the write mask AND array is routed to the *metal2* wires on the left side of the bank. These *metal2* wires connect to the control logic at the SRAM level. Without the optional write mask parameter, the *w_en* pin would instead be routed from the write driver to the *metal2* wire, as it was previously.

3.5.5 SRAM

The final step in adding the write mask for a single-port SRAM requires placing instances and adding connections at the SRAM level. The write mask flip-flops were originally

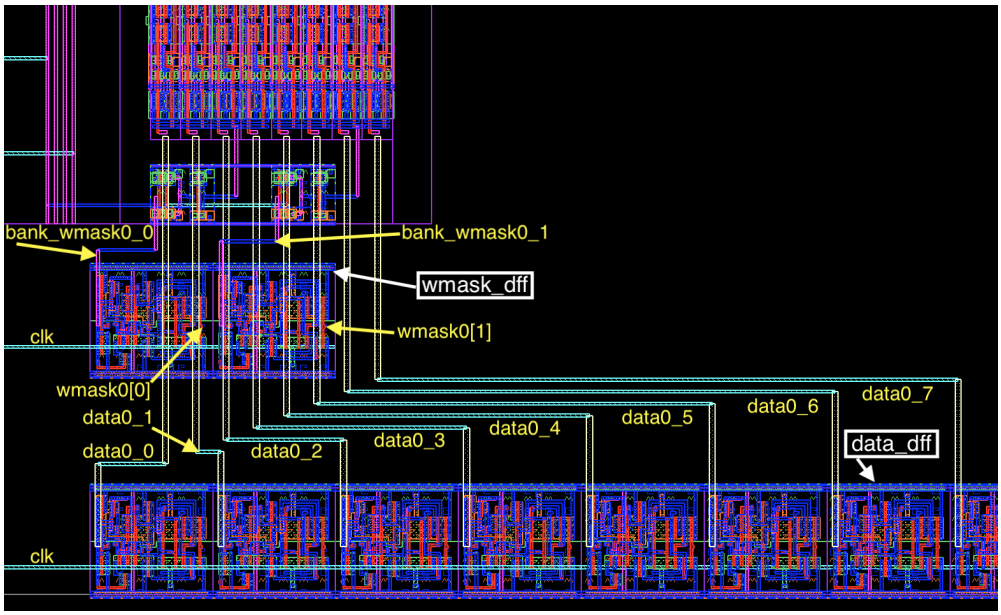


Figure 3.12: FreePDK 45nm layout of the SRAM showing the bottom of the write driver and the local flip-flops, indicated with boxed white names. The write mask flip-flops are above the data input flip-flops.

placed below the data input flip-flops. However, this requires that both the data and write mask flip-flops are routed in *metal3* and *metal4* to the data port. Therefore, the locations are swapped, as seen in Figure 3.12. When a write mask is included, the *bank_wmask0_*{} wires are channel routed in *metal1* and *metal2* from the write mask flip-flops to the write mask AND gates. On the other hand, the *data0_*{} signals are channel routed in *metal3* and *metal4* from the data input flip-flops to the write drivers. Without a write mask, the data input flip-flops are routed in *metal1* and *metal2* as they were previously.

The last steps of the SRAM routing consist of routing from the control logic and routing supplies. First, the clock must be connected to the write mask flip-flops in *metal3*. As it was previously, the wire for the write enable signal connects from the control logic to the vertical *metal2* wire on the left of the port. When the supply routing option in the SRAM configuration file is enabled, the *metal3* and *metal4* supply grid is placed, where possible, at regular intervals and connects each *vdd* and *gnd* pin. To show detail, the final layout of the 1 RW port SRAM presented in Figure 3.13 excludes supply routing.

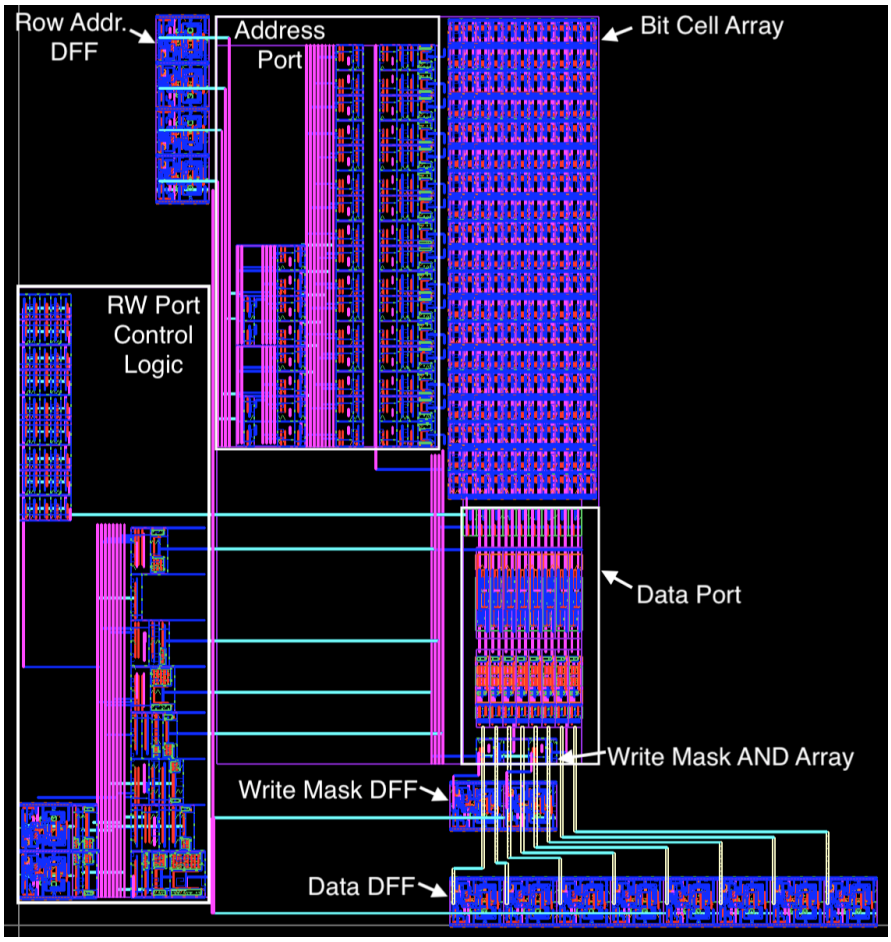


Figure 3.13: FreePDK 45nm layout of a 8b x 16 SRAM with 4-bit writes and 1 RW port. Write mask components and major SRAM elements are labeled. Due to OpenRAM's automated layout, area efficiency is future work.

3.5.6 Multiport

Each write port in the SRAM is supplied with the same write mask size. As expected, multi-port with multiple read ports is not affected. OpenRAM implements a multi-port layout by mirroring the placement of the second port, as seen in Figure 3.14. This mirroring requires two different write mask AND array instances as it is no longer possible to have a direct *metal1* connection from the vertical *metal2* wires to the write mask AND array. Therefore, the second write port's *en* pin extends to the right side of the write mask AND array in *metal3*. This allowed for the *w_en* pin to be routed in *metal1* to connect to the right side of the write mask AND array

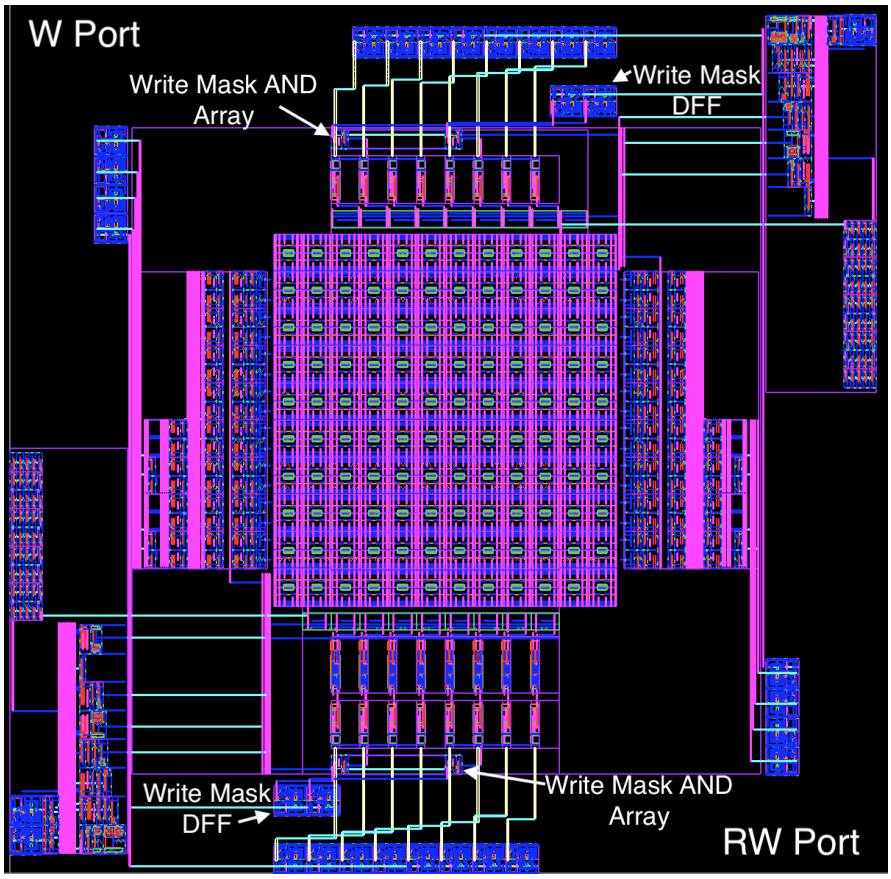


Figure 3.14: FreePDK 45nm layout of the 8b x 16 SRAM with 4-bit writes and 1 RW, 1 W port. Write mask components and ports are labeled.

without causing a DRC error.

Chapter 4

The Place and Route of PicoSOC, a RISC-V Processor

Adding write masking to OpenRAM allows for RISC-V processors' memories to be generated in OpenRAM. In this chapter, we will use PicoSOC, a system on a chip (SOC), which incorporates a PicoRV32 processor [11] and respective SRAM memories. To demonstrate our improvements to OpenRAM, we will explain the place and route of two PicoSOC layouts: one with SRAM memories and one with flip-flop memories. In addition, we will discuss timing and area results between the two layouts.

4.1 Clifford Wolf's PicoSOC and Its Memories

Clifford Wolf's PicoRV32 is the processor component of PicoSOC. PicoRV32 is a 32-bit RISC-V processor that implements the RV32IMC ISA. The core has three variations: PicoRV32, PicoRV32_axi and PicoRV32_wb. PicoRV32 is the standard PicoRV32 CPU while PicoRV32_axi is a version of the CPU with a AXI4-Lite interface. Finally, PicoRV32_wb is a version of the CPU with the Wishbone Master interface [11].

Figure 4.1 displays an example of how PicoSOC would look connected to a circuit. PicoSoC includes PicoRV32 along with a memory controller that interfaces to external SPI flash and a simple UART core connected directly to SoC TX/RX lines [11]. The SRAM size is configurable, but is a 1kB scratchpad by default.

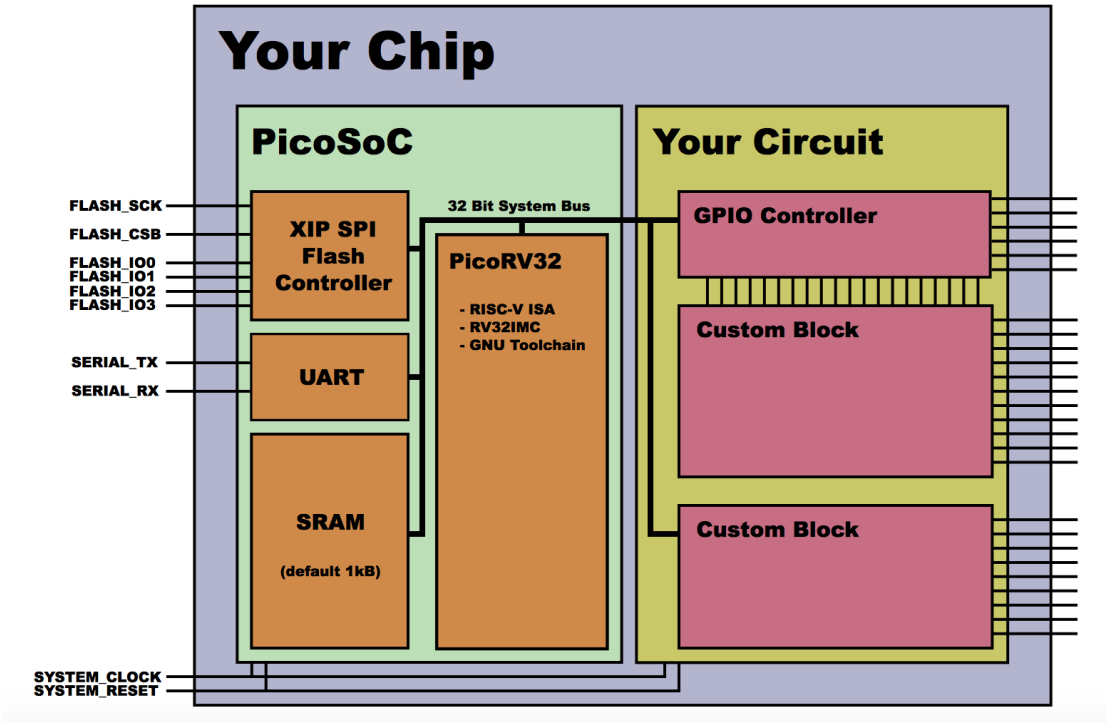


Figure 4.1: An example SoC using PicoRV32 [11].

The SRAMs used for place and route are shown in Table 4.1. The default 1kB SRAM scratchpad is used so that OpenRAM could quickly generate the supply grid. The SRAM scratchpad is included in PicoSOC while the register file is internal to PicoRV32. Since there is no circuit being added the design, the right side of Figure 4.1 is disregarded. While a dual-port register file provides better performance, a single-port register file results in a smaller core [11]. However, with the dual-port register enable parameter turned off, the register files are implemented with OpenRAM's SRAM because the register file becomes a 1 R, 1 W port instead of 2 R, 1 W port SRAM. Therefore, dual-port must be chosen for both designs for consistency.

Table 4.1: SRAM sizes in PicoSOC [11].

SRAM Function	Word Size (Width)	Number of Words (Depth)	Capacity (Width×Depth)	Port Type	Write Mask Size
Register File	32-bits	32	128 bytes	1 R, 1 W	-
SRAM Scratchpad	32-bits	256	1024 bytes	1 RW	8-bits

4.2 Place and Route Design Flow

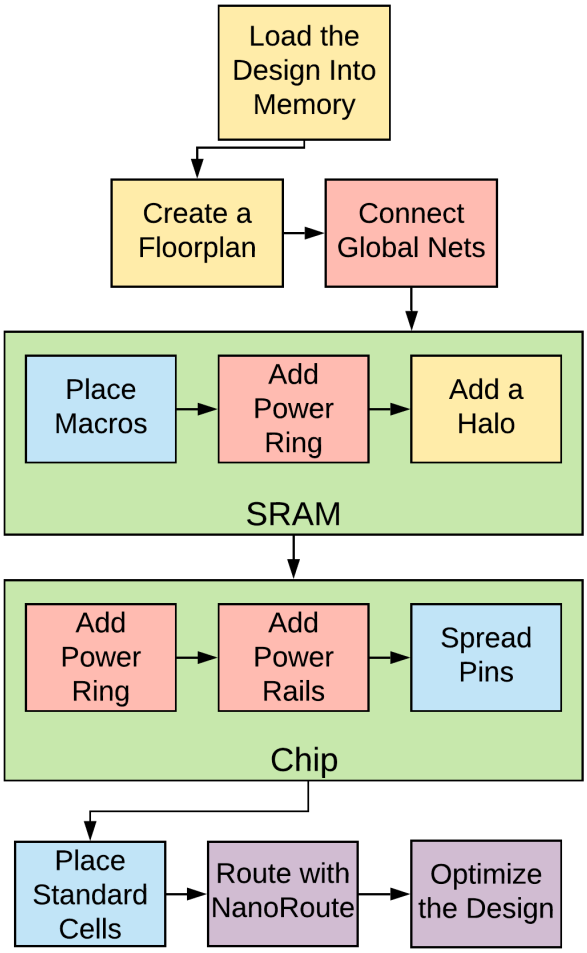


Figure 4.2: Block diagram demonstrating the steps required to place and route in Innovus. Several indicated steps involve the SRAM macro or chip itself.

Figure 4.2 shows the design flow to place and route a design in Innovus. In Chapter 2, we described how the design was loaded into memory. After the design is in Innovus' memory, a floorplan is created to fit the cells in the design with a perimeter wide enough for a power rail. Then, the supply connectors are tied high and low to connect global nets. After the SRAM macros are placed, a power ring is added around the macros with a halo to prevent standard cells from being placed on the ring. Finally, a power ring is placed around the chip in the area

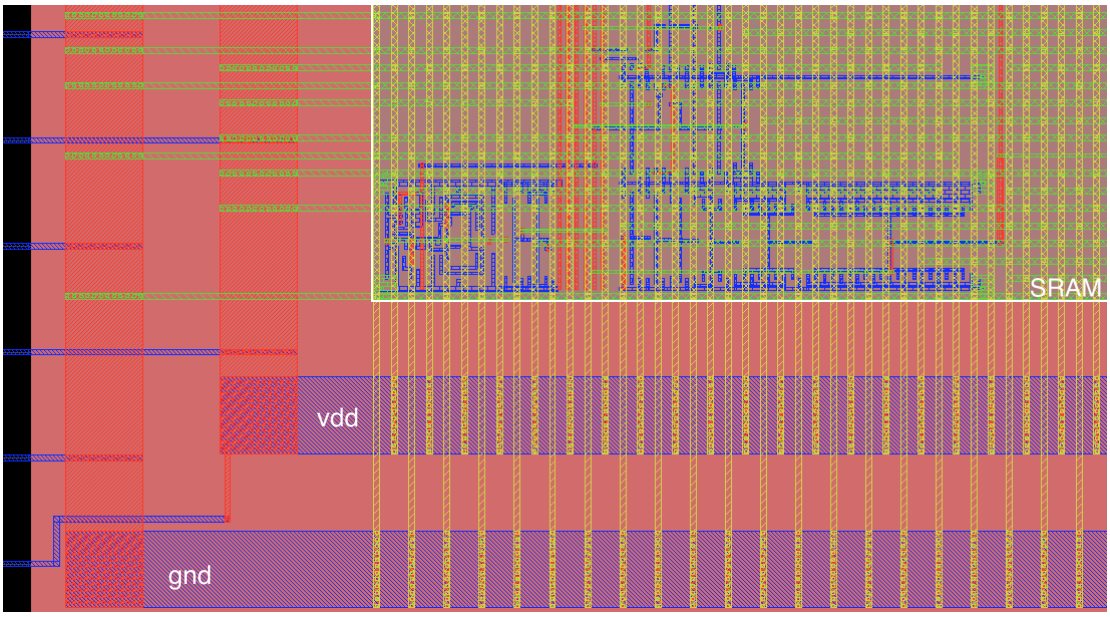


Figure 4.3: Supply routing from the power ring to an SRAM's supply grid. The lower left corner of the SRAM is pictured with a halo (red) around the power ring.

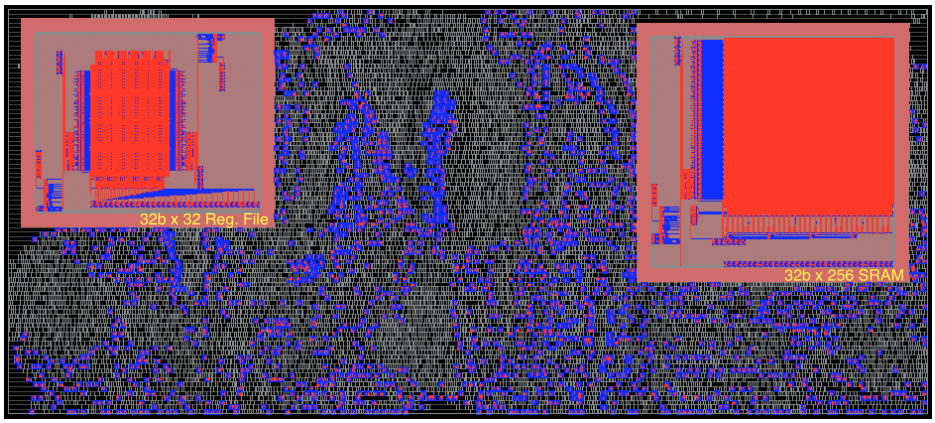
that was left when the floorplan was created.

After the initial placement of macros and power rings, the design is routed and optimized for timing. First, supply rails must be added horizontally across the chip. The addition of supply rails is called special routing in Innovus. This routing allows the standard cells to connect to the supply rails. Additionally, the power rails connect from the power ring to the power grid inside the SRAM. The next step in place & route is to spread the pins out along the border. Afterwards, standard cells are placed and the design is routed with NanoRoute. Here the routing algorithm can be chosen to be driven by timing and signal integrity (SI). After routing, the design is optimized to meet setup and hold timing if it is not yet achieved. Afterwards, files can be created to record timing, power, and DRC results.

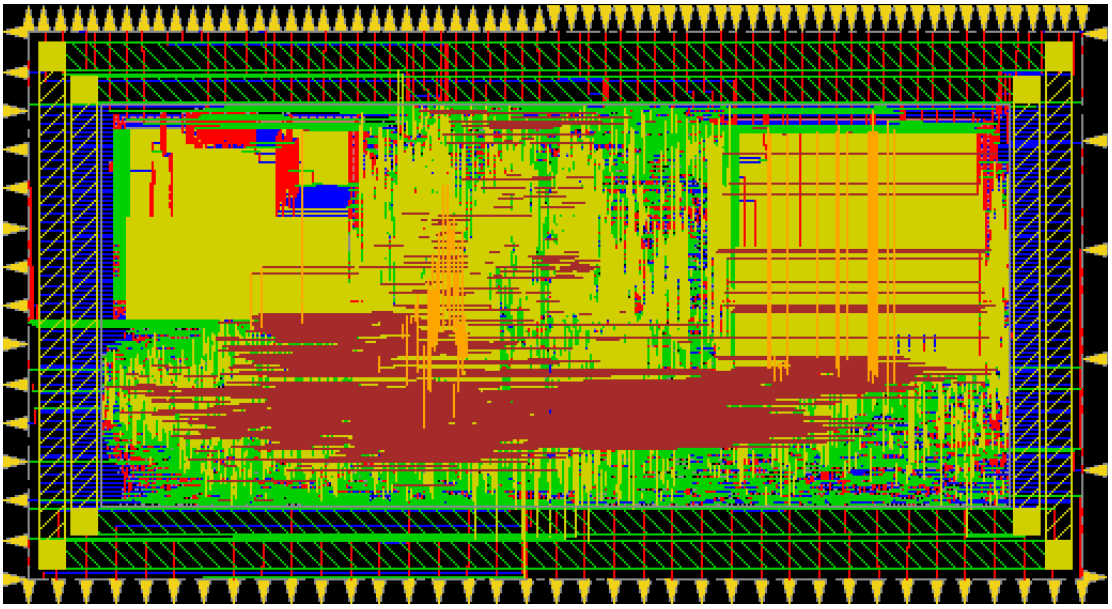
In order to successfully place and route with OpenRAM's SRAMs, the final hurdle, mentioned in Chapter 2, is to connect each wire in the supply grid to the power ring around each SRAM. By default, Innovus only connects *vdd* and *gnd* supply rails in the SRAM to the power ring one or two times. Adjusting the number of supply rail connections involves altering the special route settings in Innovus. By only special routing to pins abutting the bottom and left block boundaries, it is possible to route to each supply rail. This change is observed in Figure

4.3, which shows each *vdd* and *gnd* rail connected to the power ring.

4.2.1 PicoSoC Placed and Routed



(a) Layout with routing of standard cells and *metal3* and *metal4* turned off to show detail.



(b) Layout with routing and all metal layers turned on.

Figure 4.4: FreePDK 45nm layout of PicoSOC with OpenRAM's SRAMs for the memories.

PicoSOC is synthesized and placed and routed, both with OpenRAM's SRAMs and flip-flops for the scratchpad and register file. Both designs are routed for timing and SI driven optimizations. These optimizations cut down on the area of the design, particularly in the

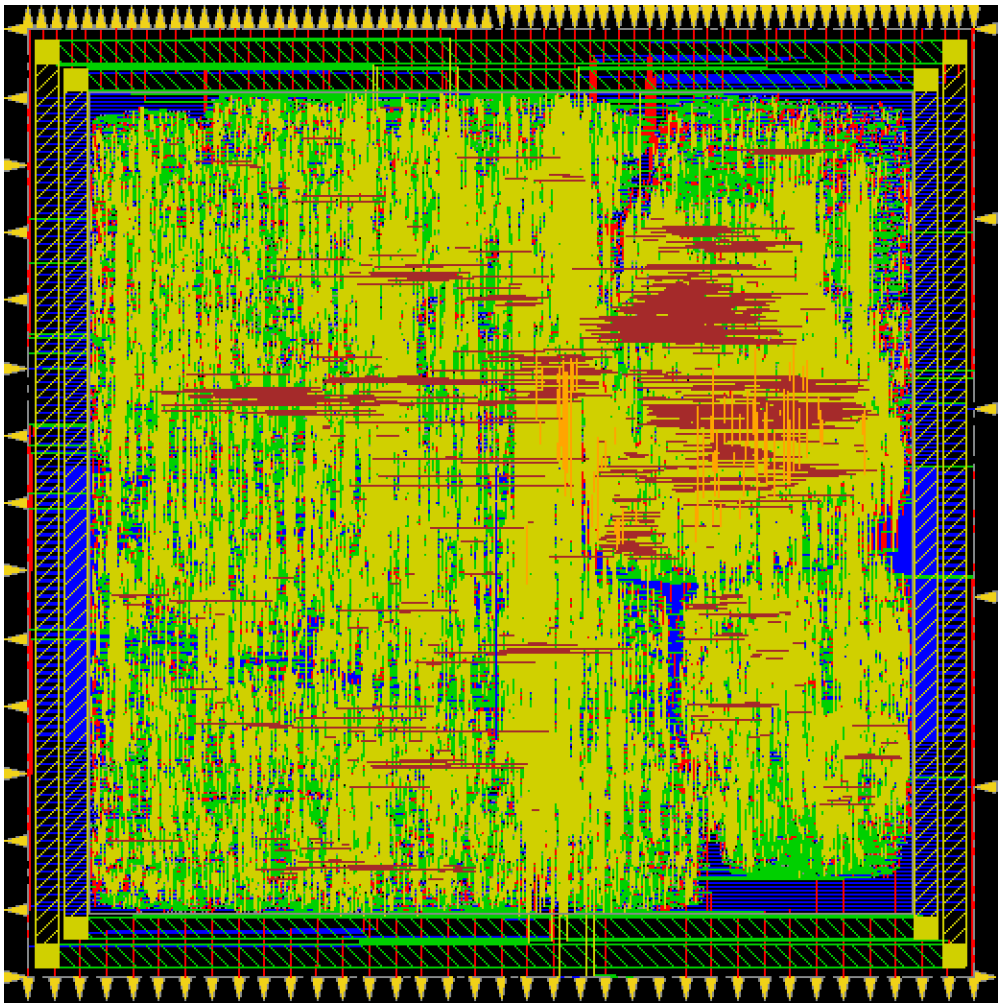


Figure 4.5: FreePDK 45nm layout of PicoSOC with flip-flops for the memories.

SRAM case.

Figure 4.4 shows the layout of the processor with SRAM. This design incorporates two custom bit cells: a 1 W, 1 R bit cell and a 1 RW bit cell from OpenRAM. In the 1 W, 1 R SRAM, the read port is on the top, mirrored, and the write port is on the bottom. In the 32b x 256 SRAM with 1 RW port, the 8-bit write is shown. Due to the configuration of the SRAM, OpenRAM creates the SRAM with 4 words per row. Therefore, the column multiplexer and row address flip-flops are present in the design. In Figure 4.4(a), all internal *metal3* and *metal4* routing in the SRAM are turned off to show detail by removing the power grid. Therefore, the data input flip-flops and other internal components appear disconnected. In addition, the power

rings are removed, as they are power routing, and Figure 4.4(a) only extends to the inner chip's power ring.

In the PicoSOC layout, we decide the placement of the SRAMs by recording where Innovus placed them with the standard cells. This location is then manually recorded for consistency and placement of the halo ring. The final floorplan is chosen by removing area until all the components could still be placed and routed without any DRC errors. This leads to a rectangular shape since the bottom of the floorplan lacked standard cells, indicating it could be removed. Furthermore, the pins that are part of busses are placed on the top and the bottom of the layout as this is where the sides are longer.

Figure 4.5 displays the layout of PicoSOC with D flip-flops. Here the single-port register file and data memory are flip-flops. Unlike the SRAM layout, the flip-flop layout remains square since it is dense on all sides. For consistency, the pins in the flip-flop layout are placed at the same locations as the PicoSOC layout with SRAMs.

4.3 PicoSoC Case Study

The case study uses results from place and route instead of from synthesis for accuracy. Table 4.2 compares the area between the two designs. The placement density before routing is 51.70% for the layout with SRAMs and 65.69% for the layout with flip-flops. However, using SRAMs instead of flip-flops for the memories yields a smaller layout. With the power ring, the layout with SRAMs is 2.14x smaller. This is also impressive because the register file's SRAM from Figure 4.4(a) lacks area efficiency due to nature in which OpenRAM currently lays out a single column in 1 R, 1 W SRAMs. From Figure 4.6, the area consumed by custom bit cells grows much slower with size than flopped area, which follows with the results from the place and route.

Table 4.2: Area without and with the power ring for both PicoSOC configurations.

	Without Power Ring		With Power Ring	
Configuration	Height (μm)	Width (μm)	Height (μm)	Width (μm)
SRAMs	500	220	580	300
Flip-Flops	530	530	610	610

Table 4.3 compares the timing between the two designs for both setup and hold time before optimization. The periods are chosen by using a clock period from each design that

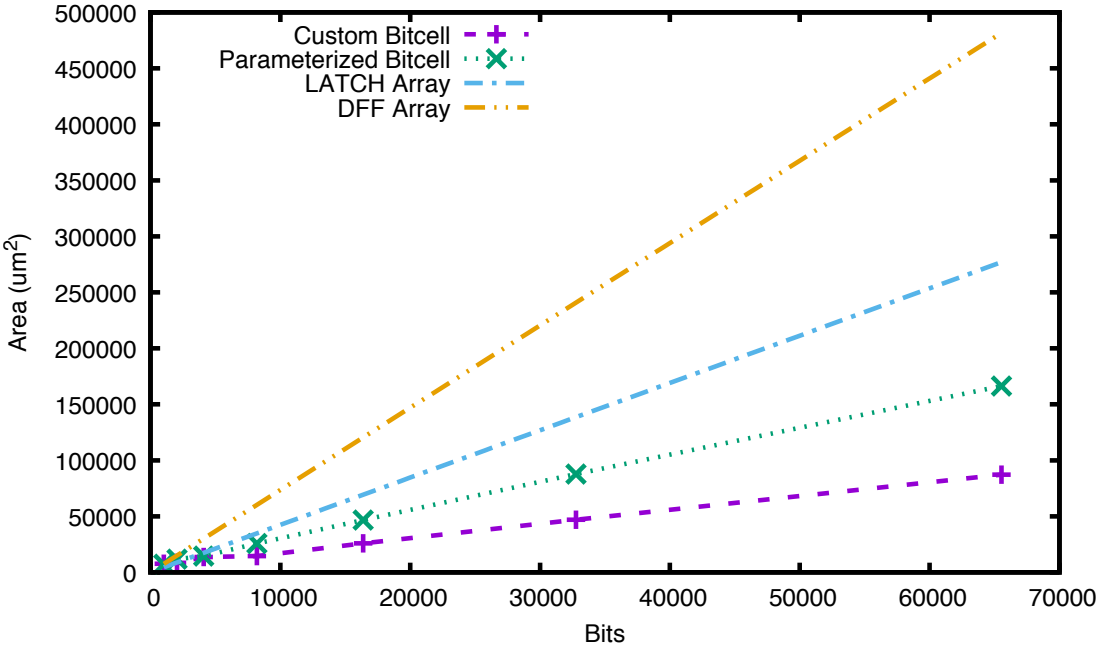


Figure 4.6: FreePDK 45nm area comparison of 32-bit word memories of varying sizes which shows that both the custom and parameterized bit cell are more efficient than latch or D flip-flop arrays [8].

meets setup time in synthesis. This period is further adjusted in Innovus until both designs meet setup time. However, hold time is not met until the designs are optimized after routing since period does not improve hold time.

Table 4.3: Post-route setup and hold time for both PicoSOC configurations before post-route optimizations.

		Setup Time		Hold Time	
Configuration	Period (ns)	Worst Negative Slack (ns)	Total Negative Slack (ns)	Worst Negative Slack (ns)	Total Negative Slack (ns)
SRAMs	7	0.179	0.000	-0.055	-0.488
Flip-Flops	6	0.063	0.000	-0.039	-0.159

Optimizing for timing allows the design to meet hold time and improve setup time. In the design with SRAMs, the critical path for setup time is from the SRAM scratchpad's data output flip-flop to the first operation flip-flop inside PicoRV32. For hold time, the critical path is between the latched read flip-flop in PicoRV32 and an address flip-flop in OpenRAM's register

file. On the other hand, in the flip-flop design, the critical path for setup time is between the latched branch flip-flop and the next program counter flip-flop in PicoRV32. For hold time, the critical path is between two CPU register flip-flops. Table 4.4 shows that with these optimizations both setup and hold timing are met. Since the PicoSOC design with flip-flops does not interface with SRAMs, it meets timing with a smaller period of 6ns. Conversely, the design with SRAMs requires a period of 7ns to meet setup time. After post-route optimizations, the flip-flop design can benefit from a smaller clock period since the setup slack is 2.093ns. However, the design cannot meet setup time before timing optimizations.

Table 4.4: Post-route setup and hold time for both PicoSOC configurations after post-route optimizations.

		Setup Time		Hold Time	
Configuration	Period (ns)	Worst Negative Slack (ns)	Total Negative Slack (ns)	Worst Negative Slack (ns)	Total Negative Slack (ns)
SRAMs	7	0.328	0.000	0.002	0.000
Flip-Flops	6	2.093	0.000	0.000	0.000

Chapter 5

Conclusions

This thesis presents three main contributions to OpenRAM. It improves OpenRAM's .lib and .lef files, adds write masking, and demonstrates that OpenRAM SRAMs could be placed and routed with a small processor, PicoRV32. With these changes, we attain a design flow for the synthesis and place and route of OpenRAM memories in a Verilog design. Furthermore, write masking adds the ability for OpenRAM to support RISC-V processor memories. The final layout in Innovus of PicoSOC with SRAM memories consumes less area than the flip-flop counterpart. Additionally, both layouts are able to meet setup and hold timing, but at different clock periods. Future work involves placing and routing a larger RISC-V processor, such as UC Berkeley's Rocket or BOOM, with OpenRAM's memories. It would also be interesting to include the register file in the place & route after OpenRAM can layout register file designs such as 2 R, 1 W ports.

Bibliography

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] S. Ataei, J. Stine, and M. R. Guthaus. A 64 kb differential single-port 12T SRAM design with a bit-interleaving scheme for low-voltage operation in 32 nm SOI CMOS. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 499–506, Oct 2016.

[3] Christopher Celio. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018.

[4] NCSU EDA. FreePDK15. `https://www.eda.ncsu.edu/wiki/FreePDK15:Contents`, 2017.

[5] NCSU EDA. FreePDK45. `https://www.eda.ncsu.edu/wiki/FreePDK45:Contents`, 2018.

[6] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, Nov 2016.

[7] MOSIS. SCN4M and SCN4M_SUBM. https://www.mosis.com/pages/Technical/Layermaps/lm-scmos_scn4m, 2019.

[8] Hunter Nichols, Michael Grimes, Jennifer Sowash, Jesse Cirimelli Low, and M. R. Guthaus. Automated Synthesis of Multi-Port Memories and Control. In *2019 IEEE/International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 1–6, Oct 2019.

[9] Peardrop Design Systems. Glade. https://peardrop.co.uk, 2017.

[10] Magic Development Team. Magic VLSI Layout Tool. http://opencircuitdesign.com/magic/, 2017.

[11] Clifford Wolf. PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/cliffordwolf/picorv32, 2019.

[12] B Wu and M. Guthaus. A Bottom-Up Approach for High Speed SRAM Word-line Buffer Insertion Optimization. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2019.

[13] B Wu, J. Stine, and M. R. Guthaus. Fast and Area-Efficient Word-Line Driver Topology Optimization. In *International Symposium on Circuits and Systems (ISCAS)*, 2019.