IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 6 March 2024, accepted 14 April 2024, date of publication 25 April 2024, date of current version 3 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3393479

RESEARCH ARTICLE

Macro Memory Cell Generator for SKY130 PDK

EMILIO ISAAC BAUNGARTEN-LEON[1,2], (Graduate Student Member, IEEE), SUSANA ORTEGA-CISNEROS[1], (Member, IEEE), GERMAN PINEDO-DIAZ[1,3], (Graduate Student Member, IEEE), MIGUEL ANGEL RIVERA ACOSTA[1], FRANCISCO JAVIER RODRIGUEZ NAVARRETE[1], URIEL JARAMILLO-TORAL[1], (Student Member, IEEE), CRISTIAN TORRES GONZLEZ[1], AND JUAN CARLOS GARCIA LOPEZ[1], (Student Member, IEEE)

[1]Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Ciudad de México Zapopan 45019, México
[2]Department of Design, Science and Technology, Universidad Autónoma de Guadalajara, Zapopan 45129, México
[3]Department of Engineering, Universidad Panamericana Campus Guadalajara, Zapopan 45010, México

Corresponding author: Susana Ortega-Cisneros (susana.ortega@cinvestav.mx)

ABSTRACT The SKY130 Process Design Kit (PDK) offers limited options for Static Random-Access Memory (SRAM) configurations, providing only three predefined memory sizes: 8×1024, 32×256, and 32×512, this poses a challenge for designers who require memories with different characteristics, as they must either design an entire memory architecture from scratch or resort to interleaving techniques with the available memory configurations. To address this issue, we present a novel framework that automates the generation of multiple memory arrays with custom floorplans, leveraging the concept of interleaving memory. Our framework enables designers to create various sizes and configurations memories by combining and interleaving the existing SKY130 PDK memories, additionally, the framework allows designers to easily specify their desired memory size, word length, floorplan, and other essential parameters. The framework then automatically generates multiple memory arrays that meet the specified requirements. Furthermore, it provides the files required by OpenLane, facilitating the seamless integration of these memories into the Register Transfer Level (RTL) to Graphic Data System II (GDSII) flow. The key advantage of our framework lies in its ability to streamline the creation of custom memories by automating the interleaving process and offering flexibility in floorplan design. This significantly reduces design time and effort, empowering designers to efficiently create memories with specific characteristics while adhering to the limitations of the SKY130 PDK. The framework thus serves as a valuable tool for memory design in the context of the SKY130 PDK, opening doors for more efficient and optimized chip designs.

INDEX TERMS Framework, interleaving memory, memory design, OpenLane, SKY130 PDK, SRAM, VLSI.

I. INTRODUCTION

Technology's continuous evolution has led to a significant surge in demand for faster and more efficient computing systems. Because of this demand, the electronics industry has experienced remarkable integration, resulting in billions of transistors being added to single chips. Such advancements, often referred to as System-on-a-Chip (SoC) designs, have revolutionized the computing landscape, however, this integration has brought along various complexities for Very Large Scale Integration (VLSI) designers [1].

Memories play an important role in an array of electronic devices, from smartphones and gaming consoles to high-performance computing in the cloud and advanced driver assistance systems in cars. With an escalating demand for compute-intensive applications like big data analytics, machine learning, and artificial intelligence, memory designers are facing significant challenges in fulfilling the diverse application requirements while ensuring time-to-market and cost targets are met [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Wu-Shiung Feng.

The memory-bound computing problem represents another significant hurdle in the VLSI domain. As the data volume to be processed grows, there is a resultant surge in memory demand, creating a bottleneck that significantly affects system performance [3]. This challenge necessitates the incorporation of new memory management techniques in the designs.

However, creating memory in layout can be an arduous task. Given the complexity of VLSI designs, system design becomes of paramount importance, encompassing a broad spectrum of disciplines. Designers must balance performance, power consumption, and area demands while considering the limitations of the manufacturing process, material properties, and end-user needs [1].

Designers working within such constraints often have to resort to time-consuming and error-prone manual processes to design the entire memory architecture from scratch or apply interleaving techniques with the available memory configurations.

Interleaving memory is a technique designed to mitigate this challenge, where the single memory unit is replaced by multiple memory units or banks. These banks are arranged to allow independent and concurrent access to multiple units, permitting several words to be accessed in each memory cycle [4].

Interleaving memory stands as an effective solution for the limitation imposed by the connection between the main memory module and a processor in conventional systems. In these conventional systems, the speed of data and instruction processing is directly dependent on the delivery speed of the memory, which typically exhibits a longer access time than the processor cycle time. This disparity creates a bottleneck that restricts system performance [4].

The advantages of memory interleaving extend beyond merely improving system performance. For instance, this approach proves highly beneficial in machines equipped with vector processors. In such machines, the data sets' size often renders the caches less efficient than usual. Thus, the memory system must continually supply the processor with a high-speed data stream, which interleaved memory can achieve effectively [4].

Memory interleaving also enhances fault tolerance. In a system with a single memory unit, an operational failure could lead to significant performance disruptions. However, with multiple memory banks, soft logic can be integrated to reconfigure the remaining banks into smaller memory units, ensuring continuity of operation [5].

Memory interleaving is not solely applicable to dividing large memory units into smaller ones; it can also be used to compose large memory units from interconnected smaller ones [4]. This method is currently in use across various hardware accelerators [6], in improving cache memory performance [7], and in advancing energy performance [8].

In the bibliography, various techniques are employed for memory interleaving optimization. In some of these techniques the generated memory, the interconnection algorithm, or both are intellectual property [9], [10], [11]. Another technique involves using simulation tools to test different possible configurations that meet the specifications set by the designer. These tools provide the interconnection that yields better results in system metrics, such as latency, power consumption, or area [12].

Another technique, found in the literature and also used in this work, is to perform multiple iterations of interconnection. The goal is to experimentally determine the optimal values for interconnection parameters, focusing on specific memory characteristics [13].

In the complex realm of VLSI design, technological innovations are continually evolving. An example of such is SKY130 [14], which is a flexible process node developed by Cypress Semiconductor and now accessible to the industry via the SkyWater Technology Foundry. A result of a fruitful collaboration between Google and SkyWater, this provides an open-source Process Design Kit (PDK) and resources for fabricating concrete designs. SKY130 is a mature hybrid technology that possesses a flexible technology stack, replete with a myriad of features. These include support for various voltage levels, local interconnects, multiple metal layers, inductor capabilities, and even optional components like capacitors. Despite these extensive capabilities, the primary focus of the SkyWater open-source PDK at present is geared towards test chips and initial design verification rather than full-scale production contexts [14].

The SKY130 process incorporates the design of memory, particularly, Static Random-Access Memory (SRAM) cells included in its Intellectual Property (IP) libraries. However, these libraries pose a limitation as they offer only a few memory sizes, specifically three: 8×1024, 32×256, and 32×512. This restricted range in memory size significantly limits memory resources, potentially posing a substantial constraint to system designs and applications that require larger or more flexible memory configurations [14], [15].

In light of the constraints delineated above, and the pivotal role memories play in contemporary systems, the tool presented in this research emerges as a robust solution. This tool strategically harnesses the benefits of memory interleaving and leverages the IPs offered by the SKY130 library. The result is an innovative configuration wherein an array of memories operates cohesively as a single SRAM unit. This tool enables customization of the bit width and memory depth according to specific system requirements. Hence, this tool introduces a level of flexibility, scalability, and time design prompt, presenting a compelling solution to the current limitations in memory design and implementation of SKY130 PDK.

The memory interleaving tool transcends conventional design methods, extending beyond the Register-Transfer Level (RTL) design. It introduces a more comprehensive solution by automatically generating all the essential configurations necessary for a successful memory tape-out, a critical

stage in Integrated Circuit (IC) design where the final design files are provided for manufacturing.

One crucial aspect of this tool is its compatibility with the OpenLane design flow, an automated RTL to Graphic Data System II (GDSII) flow based on several components from the open-source semiconductor ecosystem. This flow includes processes such as synthesis, floorplanning, placement, Clock Tree Synthesis (CTS), routing, and verification [16].

In effect, this tool synergizes the concept of memory interleaving with the powerful automation capabilities of the OpenLane flow, enhancing memory design efficiency and reliability. Additionally, the tool simplifies the complex process of memory design and tape-out, offering a streamlined approach that can better handle the specific constraints and requirements in memory design, particularly in the context of the SKY130 technology.

The remainder of the article is organized as follows. Section II will cover a literature review. Section III introduces the Macro Memory Cell Generator tool. Section IV addresses to explains the development of the Macro Memory Cell Generator tool. Section V illustrates the experimental result. Section VI presents the discussion. Finally, Section VII concludes this work.

II. BACKGROUND AND RELATED WORK

The work presented in [13] emerges as a viable option for users that require customized designs and configurations. OpenRAM is an open-source memory compiler that automates the process of designing SRAM memories in IC, enabling designers to generate customized designs that satisfy specific system requirements. Additionally, it simplifies memory design offering a tool to characterize and evaluate the performance, stability, and power consumption of these designs.

Using OpenRAM, designers can create and tailor SRAM cells to their specific needs within the SKY130 PDK, offering an extra degree of flexibility and control in memory design. Nevertheless, OpenRAM has limitations regarding the size and density of the generated memories, with a maximum suggested size of 4KB, although 8KB and 16KB memories are available, they still present problems when implemented in layout with the PDK SKY130 [13].

In the study presented by [17], there is a notable extension of the capabilities originally introduced in OpenRAM as delineated in [13]. This extension features a multi-port solution, enabling the memory compiler within OpenRAM to utilize multi-ported memories. Such an adoption significantly augments both the throughput and the versatility of the memory system. Central to this enhancement is the introduction of a parameterized bitcell. Remarkably, this bitcell can support any combination of read, write, and combined read-write ports. The SRAM layout is adeptly designed to support any dual-port combination.

From a broader perspective, it's evident that advancements in memory architecture play a pivotal role in the overall efficacy of SoC. As technological applications evolve, the optimization and performance of memory systems, especially those with multi-port configurations, ascend in importance. [17].

In the study detailed by [15], the utility of OpenRAM for SRAM design technology co-optimization is elaborated upon. The culmination of this exploration led to the development of a memory configured for a 130nm node, incorporating 5 metal layers and operating at a nominal supply voltage of 1.8V. This memory boasts an effective bit area of $11.1um^2$/bit. For context, the authors compared this with a D-flip flop designed in the analogous technology, which occupies an area of $41.3um^2$/bit. Furthermore, it's noteworthy to mention that a 1-kilobyte variant is currently in the fabrication phase, under the auspices of the Google/Skywater OpenPDK project, as referenced in [14].

The study [12] analyzes memory design challenges such as optimization, performance, area and latency, as well as the intricate interplay of low-level circuit attributes. Their approach takes advantage of the synergy between CACTI, a high-level tool for design exploration, and OpenRAM. By taking advantage of both tools, the research strives to accelerate the derivation of memory configurations with exceptional area efficiency and minimized power delay product.

The study covers various technology nodes, including 7nm FinFETs, 32nm and 65nm massive CMOS. In particular, the balance between peripheral components and interconnects was found to fluctuate between technology nodes, warranting nuanced design considerations. In addition, the introduction of near-threshold voltage and standard/full swing voltage regimes further enriches the analysis, delineating their impact on memory subset performance.

However, it is imperative to highlight that the focus of [12] is rooted in theoretical exploration and architectural design, circumventing the limitations imposed by the non-manufacturable nature of the PDKs used. This approach, while not directly implementable, serves as a fundamental step in the design of optimized memory configurations that could potentially inform future practical design efforts.

Within the [9] product catalog you can find a set of memory compilers, designed to fit a wide spectrum of configurations and architectures. These configurations cover single port and dual port options, along with log files and read-only memory alternatives. The very architecture of these compilers incorporates SRAM, ternary content-addressable memory, and embedded flash, each of them adapted to different requirements such as low energy consumption or high performance.

Notably, these compilers align with the manufacturing processes of industry giants like TSMC, Samsung, and GlobalFoundries. This compatibility facilitates seamless integration into a broad chip design landscape.

A key advantage of memory compilers is their transformative impact on SoC development. By substantially shortening time to market and mitigating development costs, these

compilers allow you to navigate the intricacies of building complex SoCs with greater agility.

A notable facet of the approach lies in its ability to be customize. Recognizing the uniqueness of each project, they offer a canvas to tailor memory blocks to specific needs, spanning size, performance benchmarks, and power consumption. A final exploration brings us to the diverse range of technologies with which they are offered from 250nm to 7nm.

There are other memory compilation solutions [10] comparable to other companies' products [9]. Ass Generic Memory Compiler (GMC) [10] which is a tool designed to automatically generate multiple types of SRAM with different characteristics. offering a variety of capabilities and a simple user interface, very similar to those offered by other companies [9] for a wide number of configurations and architectures. GMC creates a variety of insights for each supported memory type, such as layout designs, SPICE network lists, Verilog and VHDL models, timing and power libraries, and Design Rule Checking (DRC)/Layout Versus Schematic (LVS) verification reports. Its modular form and its emphasis on the ''tiling'' approach to design generation allow for the production of effective and flexible designs.

The company offers the broadest portfolio of silicon-proven base IPs in the industry, including memory compilers, logic libraries, and General Purpose I/O (GPIOs), supporting a variety of die castings and process technologies from 250nm to 3nm FinFET. This is how GMC fits into Synopsys' broader product portfolio. Synopsys memory compilers are designed with extensive power management capabilities such as light sleep, deep sleep, shutdown, and dual power rails. They are optimized for low power consumption, high performance, and high density. This makes it possible for designers to meet the stringent low-power specifications of contemporary SoCs. In addition, the integration of the STAR memory system and Synopsys memory compilers optimizes integrated memory tests, increasing performance and efficiency of the [11] chip area.

While it is worth acknowledging that [9], [10], [11] offers a suite of sophisticated commercial tools, including proprietary memory compilers that offer an extensive array of memory design and optimization functions, it is essential to consider the limitations associated with commercial software.

Commercial tools often entail significant costs, potentially placing financial burdens on users, especially smaller organizations or individuals. Furthermore, commercial licenses may come with restrictive terms and limited flexibility, limiting accessibility and adaptability for a broader user base, conversely, open-source tools [13], [14], [15], [16], [17], in contrast, tend to be more cost-effective, fostering inclusivity and community-driven development. They often provide greater transparency, enabling users to understand and modify the underlying code, thus encouraging collaboration and innovation within the open-source community. These factors make open-source tools an appealing choice for many seeking memory design and optimization solutions.

III. MACRO MEMORY CELL GENERATOR

The Macro Memory Cell Generator Tool has been created to help design memories more flexibly and quickly when using the SKY130 design kit implementing memory IPs that have been created and tested on-chip. The SKY130 kit comes with three default memory sizes, in case you want a memory with different capacities than the pre-established memories, you would have to make modifications at the layout and RTL level. That's where this innovative tool comes in, making all these modifications on based on requested memory.

At its core, the Macro Memory Cell Generator comprises a collection of Python scripts that facilitate the customization of memory sizes by modifying the parameters within a Verilog file. By working within the constraints of the three SKY130 memory sizes, 8×1024 (T0), 32×256 (T1), and 32×512 (T2), the tool combines and interleaves these memories to generate the desired configuration. This provides designers with enhanced flexibility that caters to an extensive range of project requirements, from the smallest modules to larger, more complex memory architectures.

The user-friendly nature of the tool is evident in the simple input requirement - the user needs only specify the base memory type, *0* for 8×1024, *1* for 32×256, or *2* for 32×512, the word width, the number of memory directions, and the preferred layout floorplan: *c* for column, *r* for row, *g* for grid, *ct* for custom, or *a* for automatic. When a custom floorplan option is selected, two additional parameters are necessitated, the number of block memories on the X axis and the number of block memories on the Y axis, all arranged with a fill priority based on the X position. Conversely, when the automatic floorplan option is chosen, the extra parameters required are the X and Y perimeter that define the user's area in micrometers. With these coordinates in place, the tool will insert as many memory blocks as possible within the designated region, optimizing for space and functionality.

These parameters enable the tool to produce an accurate memory configuration and ensure that the resulting memory layout aligns with the user's design goals.

Beyond the Verilog file modifications, the tool also works hand in hand with OpenLane configuration files. By utilizing the same Python scripts, baseline configurations are subtly manipulated based on the memory type and size, allowing for a seamless integration process that significantly reduces the design time. The Macro Memory Cell Generator thus presents itself as a comprehensive solution for efficient and optimized memory designs, striving to unlock new potentials in the face of the SKY130 PDK memories limitations.

Fig. 1 provides a visual representation of the systematic workflow utilized to generate the memories GDSII file. Within this workflow, the user is guided through a streamlined process that simplifies the creation of memory into manageable steps.

This process starts with the selection of the memory type, the user then determines the data width and address size according to specific requirements, followed by the choice of arrangement. Then the generated files are imported into

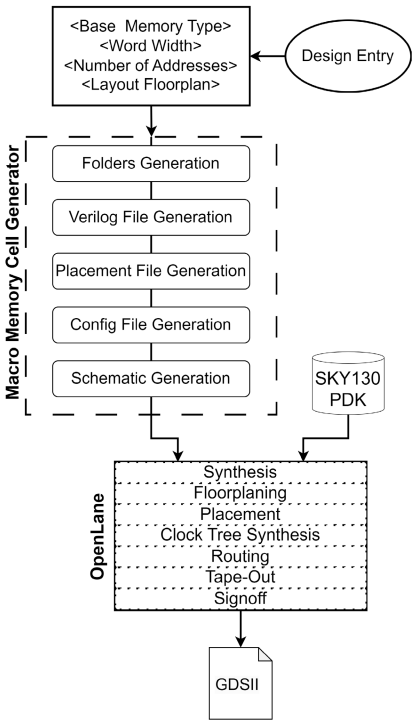


FIGURE 1. Workflow diagram for memory configuration. This illustration details the step-by-step process from selecting the memory type to executing the OpenLane flow, leading to the generation of the GDSII file of the memories.

OpenLane. Finally, the OpenLane flow is executed, this can be accomplished without the need for any additional changes or adjustments. The design of this workflow reflects a thoughtful balance of flexibility and efficiency, enabling users to achieve the desired memory structure with minimal complexity.

A. TOOL INSTALLATION AND USE

Prerequisites:

- Python 3.6+, pip, svgwrite, cairosvg
- Git 2.34+
- OpenLane

The initial phase to execute the Macro Memory Cell Generator Tool starts with obtaining the necessary files by downloading the repository from GitHub.

```
$ git clone
 ↪  git@github.com:Baungarten-CINVESTAV/
 ↪  SKY130-Macro-Memory-Cell-Generator.git
```

Then to generate the desired memory execute the *imem_generator*, which is the main Python file, with the command:

```
$ cd SKY130-Macro-Memory-Cell-Generator
$ python3 imem_generator.py [mt] [wn]
 ↪  [ad] [p] [op1] [op2]
```

The parameter *mt* represents the base memory type from the SKY130 library and is characterized as an integer data type. Its value is restricted to *0*, *1*, or *2*, each corresponding to distinct SKY130 memory sizes. Specifically, an *mt* value of *0* corresponds to a memory size of T0, while a value of *1* corresponds to a size of T1. When *mt* is set to *2*, the associated memory size is T2.

The data width, denoted as *wn*, is determined by the word width and is expressed as an integer data type. The constraints on *wn* are dependent on the value of *mt*. If *mt* is set to *0*, then *wn* must be a multiple of 8. However, if *mt* takes on a value of *1* or *2*, the requirement for *wn* changes, and it must be a multiple of 32.

The parameter *ad* represents the number of addresses of the desired memory. It must always be a multiple of 2 to ensure proper alignment and functionality. The minimum required value of *ad* varies depending on the memory type parameter *mt*. If *mt* is set to *0*, the minimum allowable value for *ad* is 2048. In the case where *mt* equals *1*, the minimum requirement for *ad* is 512. Finally, if *mt* is set to *2*, the minimum value required for *ad* is 1024. These conditions are integral to defining the correct memory structure and must be adhered to in order to maintain the integrity and efficiency of the memory management system.

The *p* parameter signifies the placement or arrangement of the memory in the die area, and its value determines the specific structure configuration. If set to *g*, the memory is arranged in a *grid* formation. A value of *r* leads to a *row* arrangement, whereas *c* corresponds to a *column* configuration. For more specialized requirements, *ct* can be selected for *custom* arrangements, allowing for unique configurations tailored to specific needs. Alternatively, the *a* option ensures that the memory is arranged *automatically* by the system. It's important to note that the choice of arrangement is conditional on the data width *wn* being either 8 or 32 when the option *a* is chosen, as these constraints ensure proper alignment and optimal performance within the given memory structure.

The parameter *op1* represents option and is defined as an integer data type, serving different purposes depending on the placement configuration defined by *p*. If the placement *p* is set to *ct*, *op1* refers to the *number of columns*. Conversely, if the placement *p* is set to *a*, then *op1* signifies the *x_space* and refers to the user-defined space on the x-axis, measured in micrometers. This dimension guides the automatic arrangement, ensuring that the memory is laid out according to the spatial constraints and specific requirements of the user's design.

The *op2* parameter, denoted as option two, is an integer data type and serves a specific function depending on the chosen placement configuration *p*. If the placement *p* is set to *ct*, then *op2* corresponds to the *number of rows*, and it must not exceed 10 to ensure a successful GDSII file creation. Conversely, if *p* is set to *a*, then *op2* refers to *y_space*, representing the size on the y-axis, measured in micrometers. The precise meaning of *op2* thus varies according to the overall design specification, and careful consideration must be given to its value to maintain the structural integrity

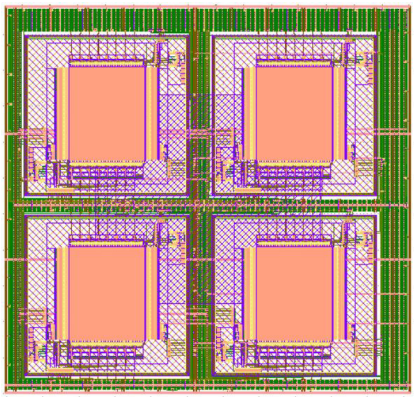


FIGURE 2. Visual representation of the GDSII file corresponding to the SRAM_8_4096_0_grid memory.

and alignment of the memory within the defined spatial constraints.

B. OUTPUT STRUCTURE

After executing the *imem_generator* Python file, the console presents a memory implementation summary. This includes the base memory type and its size, data width, number of addresses, chosen arrangement, generated memory size, count of memories in serial and parallel setups, total base memories used, and a completion message.

As an integral component of the primary repository, the Macro Memory Cell Generator Tool incorporates a *designs* folder. This folder serves as a storage for all the memory structures generated and adheres to a specific naming convention: SRAM_<wn>_<ad>_<mt>_<p>. The tool automatically generates this folder, which encompasses crucial configuration files, constraint file, placement file, and five essential sub-folders: *GDS*, *LEF*, *LIB*, *src*, and *V_BB*. This comprehensive setup is essential for compatibility with OpenLane. In addition, the tool provides an illustrative schematic representation of the interleaved memory, available in both PNG and SVG formats, facilitating a visual understanding of the memory organization and design.

C. USAGE EXAMPLE

1) GRID ARRANGE

In the present demonstration, a grid floorplan configuration is adopted, characterized by a memory size of 8×4096. Upon finalizing the choice of memory, it becomes imperative to appropriately set the parameters for the *imem_generator* file. Specifically for this configuration, the parameters are defined as: *mt = 0*, *wn = 8*, *ad = 4096*, and *p = g*. The corresponding command line invocation, which aligns with these parameters, should be structured as follows:

```
$ python3 imem_generator.py 0 8 4096 g
```

To execute this command correctly, consider that must be located in the repository directory.

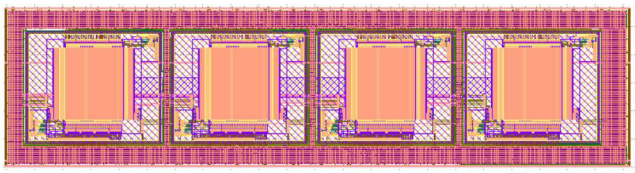


FIGURE 3. Visual representation of the GDSII file corresponding to the SRAM_32_1024_1_row memory.

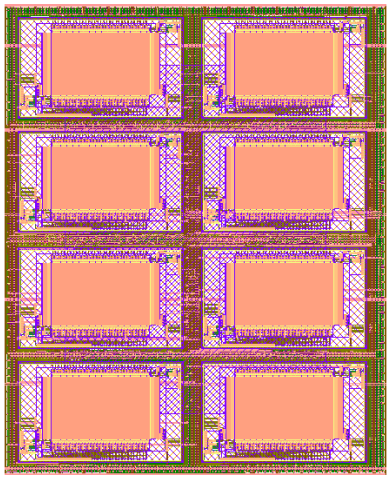


FIGURE 4. Visual representation of the GDSII file corresponding to the SRAM_64_2048_2_custom memory with 2 columns and 4 rows arrangement.

Upon execution, a directory named SRAM_8_4096_0_grid is systematically generated within the primary designs directory. The spatial layout of this specific memory configuration can be ascertained from the *config.json* constraint file. Within this file, the parameter DIE_AREA is distinctly allocated with values *0 0 1115 1082*, delineating the dimensions of the memory in micrometers. As delineated in Fig. 1, the subsequent procedure necessitates the exportation of the generated directory to OpenLane. Following this, the initiation of the RTL to GDSII workflow ensures the generation of the corresponding GDSII file for the selected memory configuration. A detailed visual representation of this GDSII file, specifically for the SRAM_8_4096_0_grid memory, is presented in Fig. 2.

2) ROW ARRANGE

For this demonstration, a memory size of 32×1024 has been selected. The pertinent parameters have been set as: *mt = 1*, *wn = 32*, *ad = 1024*, and *p = r*. The appropriate command line invocation that corresponds to these parameters should be articulated as:

```
$ python3 imem_generator.py 1 32 1024 r
```

Subsequent to the execution, a directory entitled SRAM_32_1024_1_row is produced. The encompassing area of this specific memory configuration is delineated by the dimensions *0 0 2217 570*. Refer to Fig. 3 for a GDSII file

showing a 32×1024 memory configuration with a row layout and using base memory type T1.

3) CUSTOM ARRANGE

In this particular demonstration, the targeted memory size is set at 64×2048, utilizing the base memory type T2. The arrangement is custom-designed as a rectangle encompassing 2 columns and 4 rows. To generate the GDSII file that accurately reflects this memory specification, the following parameters must be set: *mt = 2*, *wn = 64*, *ad = 2048*, *p = ct*, *opt1 = 2*, and *opt2 = 4*.

Fig. 4 shows the GDSII file of the SRAM_64_2048_2_ custom memory. For memories encompassing multiple rows or columns, a custom arrangement selection is imperative.

IV. DEVELOPMENT AND IMPLEMENTATION

A. DESCRIPTION HARDWARE IMPLEMENTATION

The architecture design presented in this paper represents an approach to memory expansion by combining serial and parallel block memories. This design expands the address and data width of a memory system. Furthermore, depending on the defined memory type (T0, T1, T2), the based module can be instantiated N times to complete the required address and data from the Python generation file.

The generator module in hardware was described in Verilog HDL, as a parameterized module with parameters ALL_MEM_DATA_WIDTH, ALL_MEM_NUM_ADDRESSES, and MEM_TYPE. These parameters are modified in Python to generate the appropriate combination of memories.

The following Verilog code illustrates how the parameters are implemented.

```
module memory_generator_sky130
 ↪  #(parameter ALL_MEM_DATA_WIDTH =
 ↪  W_n, parameter
 ↪  ALL_MEM_NUM_ADDRESSES = Ad_n,
 ↪  parameter MEM_TYPE = MT)(
// Port0 signals
        `ifdef USE_POWER_PINS
           inout vccd1,
           inout vssd1,
        `endif
        input clk0,
        input csb0,
        input web0,
        input
         ↪  [(ALL_MEM_DATA_WIDTH/8)-1:0]
         ↪  wmask0,
        input
         ↪  [$clog2(ALL_MEM_NUM_ADDRESS
         ↪  ES)-1:0] port0_address,
        input [ALL_MEM_DATA_WIDTH-1:0]
         ↪  port0_datain,
        output[ALL_MEM_DATA_WIDTH-1:0]
         ↪  port0_dataout,
// Port1 signals
        input clk1,
        input csb1,
        input
         ↪  [$clog2(ALL_MEM_NUM_ADDRESS
         ↪  ES)-1:0] port1_address,
        output [ALL_MEM_DATA_WIDTH-1:0]
         ↪  port1_dataout
);
```

By default, Verilog cannot synthesize parameters that lack specific values, such as ALL_MEM_DATA_WIDTH defined as W_n, or MEM_TYPE defined as MT. The use of keywords such as W_n, Ad_n, and MT is necessary because these variables are intended to be substituted for their actual value in a Python script based on user values.

The RTL design uses for-generate constructs and multiplexers to create a parameterized architecture capable of combining various data selection configurations. This parameterized model facilitates simultaneous support of parallel and serial data.

```
genvar i,j;
generate
for (i=0; i < NUM_SERIAL_MEMORIES;
 ↪  i=i+1) begin: SERIAL_MEMORY
    for(j=0; j < NUM_PARALLEL_MEMORIES;
     ↪  j=j+1) begin: PARALLEL_MEMORY
    .
    <base memory instantiation>
    .
    end
end
```

The top wires of the chip are referred to as port0 and port1 as are the base modules, clk0, csb0, web0, wmask0, addr0, din0, dout0, clk1, csb1, addr1 and dout1. Number of serial and parallel memories are calculated based on parameters defined by:

```
    localparam NUM_SERIAL_MEMORIES =
     ↪  ((ALL_MEM_NUM_ADDRESSES) /
     ↪  (SINGLE_MEM_NUM_ADDRESSES));
    localparam NUM_PARALLEL_MEMORIES =
     ↪  ((ALL_MEM_DATA_WIDTH) /
     ↪  (SINGLE_MEM_DATA_WIDTH));
```

Finally, the output data is assigned by shifting the port1_bus_odata where is concatenated all memory outputs. The shift is obtained by:

$$(port\_addr * ALL\_MEM\_DATA\_WIDTH) \quad (1)$$

B. UNIVERSAL VERIFICATION METHODOLOGY FOR MEMORY TOOL IMPLEMENTATION

The verification of the generated memories by the tool is a critical aspect of ensuring their functionality and reliability. To address this need, a robust verification environment based

on the Universal Verification Methodology (UVM) has been developed.

The UVM verification environment implemented for this work is showed in the Fig. 5, consisting of two distinct sections: the OpenRAM UVM environment and the OpenRAM Design Under Test (DUT). The OpenRAM environment consists of several components such as openram_base_test, openram_env, port0_scb, port1_scb, port0_agent, port1_agent, port0_monitor, port1_monitor, port0_driver, and port1_driver.

The full random test is designed to assess the memory generator's ability to handle arbitrary data inputs effectively. It encompasses a wide range of parameters, including addresses, masks, enables, and data values, with each of these elements being generated randomly during the test execution.

The Simultaneous Read/Write Test aims to verify that the memory generator can effectively manage and maintain data integrity when subjected to simultaneous read and write requests.

C. PYTHON CODE DEVELOPMENT

The Python code for the Macro Memory Cell Generator comprises five main functions, each with a specific role in the framework functionality. The following pseudocode illustrates the order in which each function is executed.

```
Folder_Name = Src_generator(wn,ad,mt,p,
 ↪  Rows,Columns)
Die_Area = Placement_File_Generator(wn,
 ↪  ad,mt,p, Rows,Columns,Folder_Name)
Copy_macro_files(mt, Folder_Name,
 ↪  Die_Area)
Config_File_Generator(,Folder_Name)
draw_diagram(wn,ad,mt,Folder_Name)
```

These functions perform the five key steps: Folder Generation, Verilog File Generation, Placement File Generation, Config File Generation, and Schematic Generation mentioned in Fig. 1.

1) FOLDER AND VERILOG FILE GENERATION

The *Src_generator* function is responsible for generating the main directory SRAM_<wn>_<ad> _<mt>_<p> within the *design* folder. It also creates the *src* folder and the top module Verilog file, which incorporates user-configured settings changing the global parameters ALL_MEM_DATA_WIDTH, ALL_MEM_NUM_ADDRESSES, and MEM_TYPE. The pseudocode of this function is shown below.

```
Current_directory = Current_Path()
Designs_directoy = Current_directory +
 ↪  "/../designs"
New_Folder = make_dir(Designs_directoy,
 ↪  "SRAM_<wn>_<ad>_<mt>_<p>")
Copy_File("memory_generator_python.v",
 ↪  "memory_generator_sky130.v")
change_parameter("memory_generator_sky
 ↪  130.v","W_n",<wn>)
change_parameter("memory_generator_sky
 ↪  130.v","Ad_n",<ad>)
change_parameter("memory_generator_sky
 ↪  130.v","MT",<mt>)
shutil.move("memory_generator_sky130.v"
 ↪  ,"New_Folder/src")
```

The first step is to get the path of the current directory and assign it to the Current_directory variable, then create another variable that points to the designs directory, which is where all the memories generated by the framework are located, create a new folder with the name SRAM and the parameters selected by the user, followed by a copy of the memory_generator_python.v file. and in the copy replaces the key names W_n, Ad_n and MT, by the values entered by the user <wn>, <ad>, and <mt> and finally moves the modified verilog file to the newly created folder.

2) PLACEMENT FILE GENERATION

The *Placement_File_Generator* function utilizes user-provided parameters to generate a configuration placement file. This file plays a critical role in arranging the memory components in the desired order, optimizing the overall memory layout.

Equation 2 defines the DIE AREA (D_A) parameter which is the area occupied by the generated block memory. Here, N_m is the number of memories, Dx_i and Dy_i are constants determined by the dimensions of the base memories T0, T1, and T2 along the x-axis and y-axis. Conversely, bx_i and by_i biases represent the offset space between two memories along the x and y directions. The *Placement_File_Generator* function automatically adjusts these biases, considering factors such as the number of memories, the type of base memory, and the need to avoid routing congestion errors. The determination of specific offsets bx_i and by_i for each memory configuration was achieved through an empirical approach to avoid interconnect congestion.

$$D_A = \sum_{i=1}^{Nm} (Dx_i + bx_i)\,(Dy_i + by_i) \tag{2}$$

This function can be divided into three sections.

In the first section, based on the parameters <wn>, <ad>, and <mt>, the function Memory_Type calculates and stores the size of the selected base memory and the total number of memories required.

The second part involves calculating the position of each memory. This calculation considers factors such as the type of placement, the number of memories, and the size of the base memory obtained in the previous section.

Finally, the third part of the function involves writing the obtained coordinates to the macro.cfg file and determining

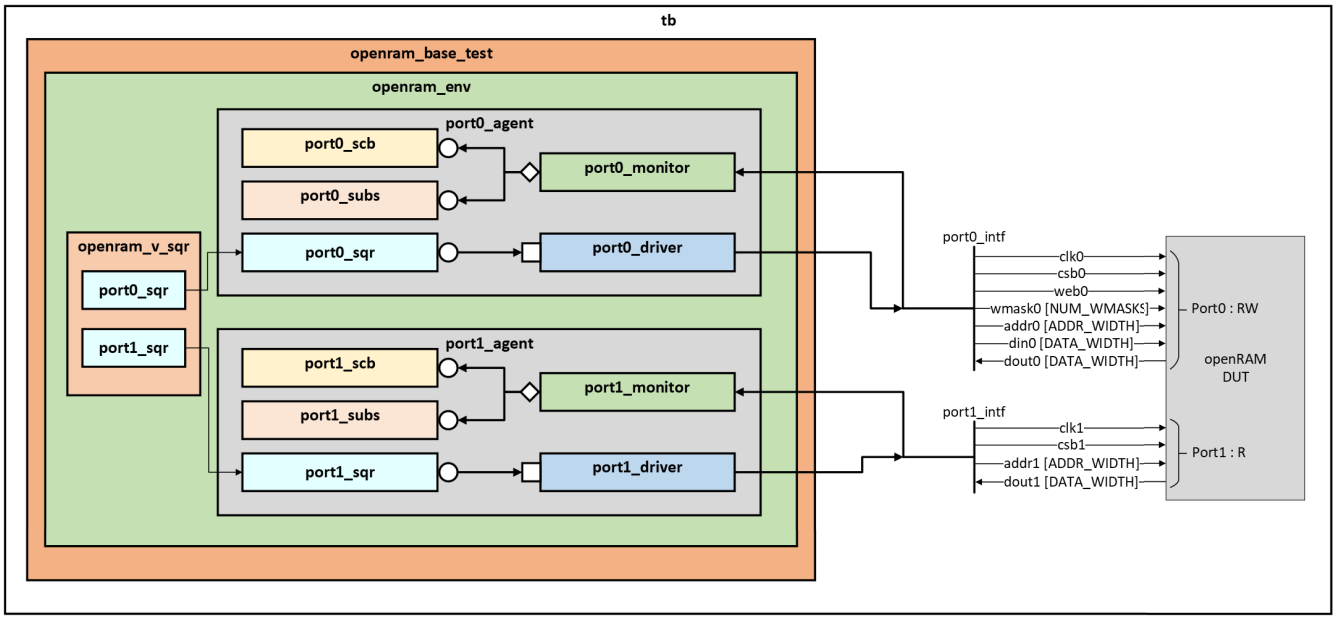


FIGURE 5. UVM diagram used to test and verify randomly memory write and read behavior.

the coordinates with the highest values in both the X and Y axes which are used to calculate the die area of the generated memory.

The following pseudocode illustrates the execution of the *Placement_File_Generator* function.

```
Memory_Type(MT,W_n,Ad_n)
 size_X = selected MT
 size_Y = selected MT
 SINGLE_MEM_NUM_ADDRESSES  = Ad_n
 SINGLE_MEM_DATA_WIDTH = W_n
 Total_Mem = Number_of_memories(W_n,
  ↪  Ad_n)

Placement_Type(p)
 coordinates = Square_placement(MT,
  ↪  Total_Mem)
 coordinates = row_placement(W_n,MT,
  ↪  Total_Mem)
 coordinates = column_placement(W_n,MT,
  ↪  Total_Mem)
 coordinates =
  ↪  Custom_placement(W_n,MT,Rows,
  ↪  Columns,Total_Mem)

file_path = f"designs/{Folder_Name}
 ↪  /macro.cfg"
open(file_path)
for i in Total_Mem:
 file.write(name[i] coordinates[i])
```

```
 if(coordinates[i] > coordinates[i-1])
  die_area = coordinates[i]
return die_area
```

3) CONFIG FILE GENERATION

The *Config_File_Generator* function dynamically generates the constraint file (*config.json*). This file is custom-tailored to meet precise memory requirements, accommodating various memory types, quantities, and placement configurations with meticulously crafted constraints.

The essence of this function lies in its remarkable flexibility for modifying variables within a JSON file. This capability is demonstrated in the following code. Here, the function first specifies the names of Verilog files and instances of the base memory (T0, T1, T2). Subsequently, it initiates the JSON file editing process. Utilizing the json.load command, allowing the ability to seamlessly edit any parameters within the file.

```
Config_File_Generator(MT,
 ↪  Folder_Name,Die_area)
Mem_Name = Verilog_File_Memory_Name(MT)
inst_Name = Instantiation_Name(MT)
Lib_name = Library_File_Name(MT)
open('onfig_python_script.json') as
 ↪  Json:
 F_Json = json.load(Json)

 F_Json['<prameter>']='<Variable
  ↪  configuration>'
```

The main constraints are comprehensively elucidated in the subsection KEY OPENLANE PARAMETERS for clarity and precision.

Alternatively, the *Copy_macro_files* function duplicates crucial folders from the macro_files folder to the generated memory folder in the design path, The files copied include *GDS*, *LEF*, *LIB*, and *V_BB*, all of which contain essential data

for a seamless RTL to GDSII conversion process for the base memory.

4) SCHEMATIC GENERATION

Lastly, the *draw_diagram* function contributes by generating schematic representations of the requested memory components. These visual outputs aid in better understanding and visualization of the generated memory structures.

D. KEY OPENLANE PARAMETERS

It's important to note that when working with OpenLane and integrating SKY130 IP memories as macros, specific constraints play a crucial role in ensuring successful memory fabrication and integration. Here's a breakdown of these constraints and their significance:

- VERILOG_FILES: Specifies the Verilog source files that form the core of the macro.
- VERILOG_FILES_BLACKBOX: Used to treat certain Verilog files as black boxes. Include Verilog files corresponding to the macro modules used in the core.
- EXTRA_LEFS: Specifies the Library Exchange Format (LEF) files associated with pre-hardened macros incorporated into the core design.
- EXTRA_LIBS: Specifies the library files of pre-hardened macros used in the current design. Optional but aids in improving timing analysis.
- EXTRA_GDS_FILES: Used for specifying GDSII files associated with pre-hardened macros integrated into the core.
- MACRO_PLACEMENT_CFG: Provides fine-grained control for manual macro placement with instance names, positions, and orientation information.
- SYNTH_USE_PG_PINS_DEFINES: Enables the usage of power and ground pins defined within the macro module during synthesis.
- VDD_NETS: Specifies power supply nets or connections required for the macro module.
- GND_NETS: Defines ground nets or connections needed for proper grounding within the macro module.

The previous constraints are mandatory and exhibit similarity across all the requested memories. However, the key differences arise when incorporating different base memories from the SKY130 IP memory. Each base memory introduces its unique GDSII, LEF, library, and Verilog files into the design, which need to be set up in a specific way.

Furthermore, to ensure the effective interleaving of memory components in the layout, specific constraints become imperative. These include:

- PL_TARGET_DENSITY: It reflects how spread the cells would be on the core area. 1 = closely dense. 0 = widely spread
- FP_PDN_HPITCH: Defines horizontal power distribution pitch, indicating the distance between two adjacent horizontal power rails, in micrometers.
- FP_PDN_VPITCH: Establishes vertical power distribution pitch, specifying the distance between two adjacent vertical power rails, in micrometers.
- FP_PDN_VSPACING: Manages vertical power distribution spacing by determining the distance between two adjacent vias in a vertical power rail, in micrometers.
- FP_PDN_HOFFSET: Specifies horizontal power distribution offset, indicating the distance between the left edge of the die and the first horizontal power rail, in micrometers.
- FP_PDN_VOFFSET: Determines vertical power distribution offset by specifying the distance between the bottom edge of the die and the first vertical power rail, in micrometers.
- GRT_ADJUSTMENT: Supports route adjustment considerations, allowing to determine how much space should be left between two adjacent routes.
- ROUTING_CORES: Facilitates effective configuration of routing cores by specifying the number of routing cores to be used during routing.
- DIE_AREA: Used to specify the area of the die in the OpenLane configuration files. It is defined as a rectangle with four coordinates: *x1, y1, x2, y2*. The first two coordinates *x1, y1* represent the bottom-left corner of the rectangle, and the last two coordinates *x2, y2* represent the top-right corner of the rectangle. The unit of measurement for these coordinates is micrometers μm.

These constraints collectively facilitate the seamless integration of SKY130 IP memories as macros within the OpenLane design flow, ensuring efficient synthesis, placement, and routing while preserving the integrity of the macros' functionality and physical characteristics.

V. EXPERIMENTAL RESULTS

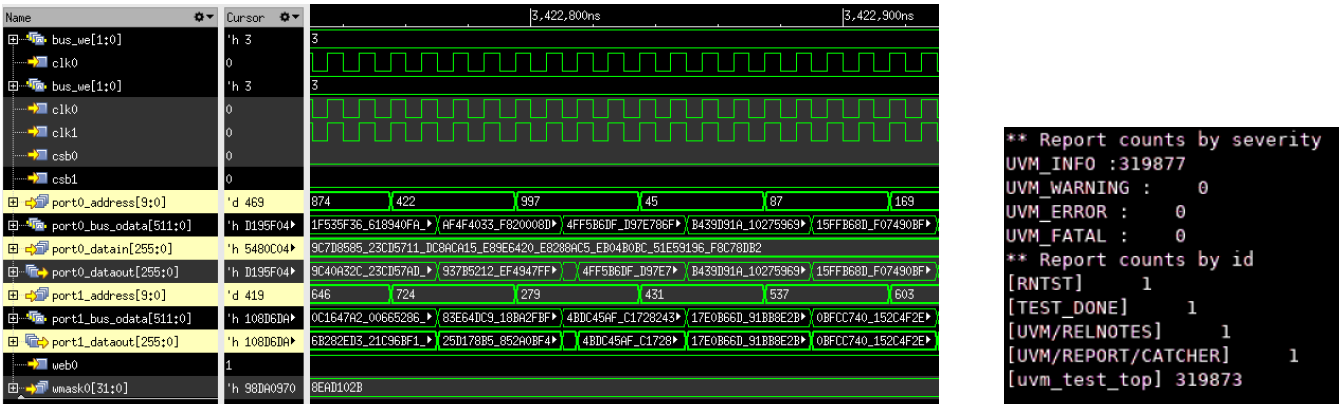
This section about the Macro Memory Cell Generator framework is presented in three distinct subsections, each contributing to a comprehensive understanding of our approach:

A. DEVELOPMENT PHASE: CONSTRAINT CONFIGURATION

In this subsection, we delve into the critical aspect of constraint configuration. The constraints governing memory generation play a pivotal role in the subsequent phases. An empirical methodology is provided insight, employed to derive these configurations.

As previously mentioned, the constraints VERILOG_ FILES, VERILOG_FILES_BLACKBOX, EXTRA_LEFS, EXTRA_ LIBS, EXTRA_GDS_FILES, SYNTH_USE_PG_ PINS_ DEFINES, VDD_ NETS, and GND_NETS are imperative for utilizing the SKY130 IP memory effectively. The selection of these constraints is contingent upon the specific base memory chosen for the design.

On the other hand, the parameters are set as PL_TARGET_ DENSITY: 0.05, FP_PDN_HPITCH: 350, FP_PDN_



(a) (b)

FIGURE 6. UVM-based testbench full random testcase. a) Waveforms of the DUT top wires, b) UVM reporter summary.

VPITCH: 100, FP_PDN_VSPACING: 5, FP_PDN_ HOFFSET: 10, FP_PDN_VOFFSET: 13, and GRT_ ADJUSTMENT: 0.1. The selection of these parameters involved a series of iterations guided by empirical methodology. Their values were calibrated with to ensure the successful implementation of over 150 different memory configurations into GDSII format.

Furthermore, the DIE_AREA constraint's definition is contingent on the MACRO_PLACEMENT_CFG constraint, with considerations encompassing routing congestion, the number of memories, and their arrangement. Lastly, the configuration of ROUTING_CORES could be adapted to the user's processor and the desired number of core usage during the routing phase, defaulting to 1.

B. UVM SIMULATION

The module (Design Under test) DUT in testbench is instanced with serial and parallel memories for the different types of memories (T0, T1, T2) in order to have memory different blocks with 2^{10} ADDRESSES and 256 BLOCK_DATA_WIDTH.

The testcase used for this UVM-based testbench is multiple full random addresses and full random data. It has been performed 100,000 operations in each port, where port0 does random writes/reads, 20,000 reads in each port concurrently, and Byte per Byte checking.

Fig. 6 a) shows the simulation waveform where can notice the randomized values for address and data in both ports. On the other hand, Fig. 6 b) is the UVM reporter, where no errors have been detected during the test.

C. TESTING PHASE: DIVERSE MEMORY INSTANCES

By utilizing the Macro Memory Cell Generator framework, we successfully automated the generation of a diverse set of 151 memory instances at the layout level. These memory instances exhibit a wide range of configurations, including data widths spanning from 8 bits to 128 bits and memory capacities ranging from 512 to 32768 memory addresses. The arrangement of these memories varies, with options including column, row, and grid layouts, as well as custom configurations tailored to specific design requirements.

In Fig. 7, both base memories T1 and T2 are depicted, illustrating the correlation between memory size and die area measured in mm^2. The representation encompasses memories organized in column, row, and grid arrangements. Notably, the absence of specific size and arrangement information, such as 32×4096 (T1) with a column floorplan, signifies the inherent limitations of the previously mentioned constraint configuration. In such cases, a custom constraint configuration becomes necessary. Nevertheless, the framework can still rely on a base reference configuration, requiring minimal adjustments, to accommodate these variations and streamline the memory design process.

Fig. 8 and 9 offer a graphical representation of the size-to-die area ratio for memories configured with custom layouts using the T1 and T2 base memories, respectively. The label at the top of each data in those figures represents the number of columns and rows, e.g. Fig. 4 uses a custom configuration with a T2 base memory, the memory configuration has 2 columns and 4 rows so the corresponding data for that implementation can be found in Fig. 9 with a memory size of 64×2048 and a top label of 2_4 which correspond to <number of columns>_<number of rows>.

Out of the 151 generated memory instances, 63 were derived from the T0 base memory. In Fig. 10, The comprehensive visualization illustrates the relationship between memory size and die area in mm^2 for memories organized in column, row, and grid arrangements. Meanwhile, Fig. 11 provides a detailed illustration of the size-to-die area ratio for memories configured with custom layouts, showcasing the versatility and adaptability of our framework.

VI. DISCUSSION

A significant portion of the work detailed in Section II shares a close association with the OpenRAM framework, which served as the foundation for constructing the base memories within the SKY130 PDK. The Macro Memory Cell Generator tool's execution relies on specific functions,

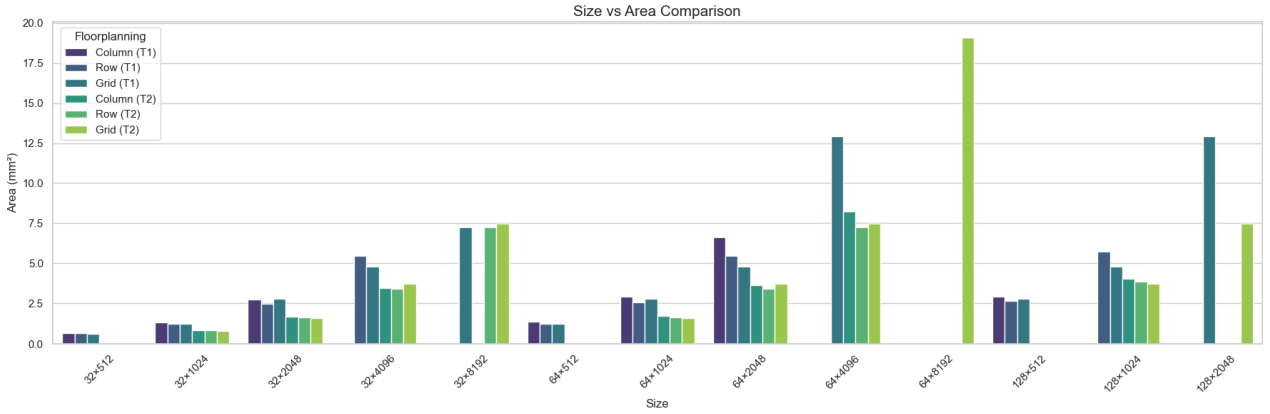


FIGURE 7. Results of size vs area comparison for T1 and T2 memory type with column, row and grid arrangement.

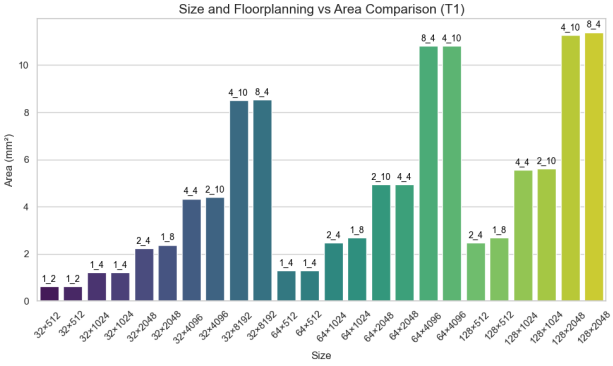


FIGURE 8. Results of size vs area comparison for T1 memory type with custom arrangement.

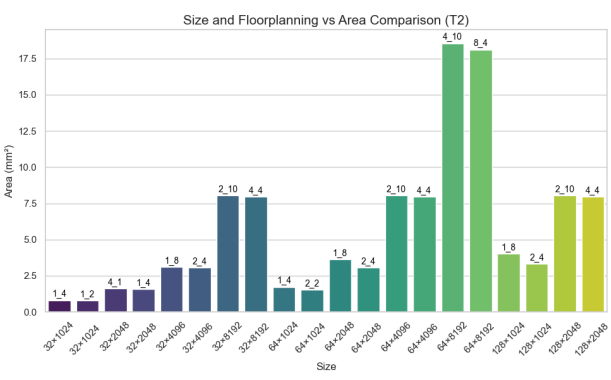


FIGURE 9. Results of size vs area comparison for T2 memory type with custom arrangement.

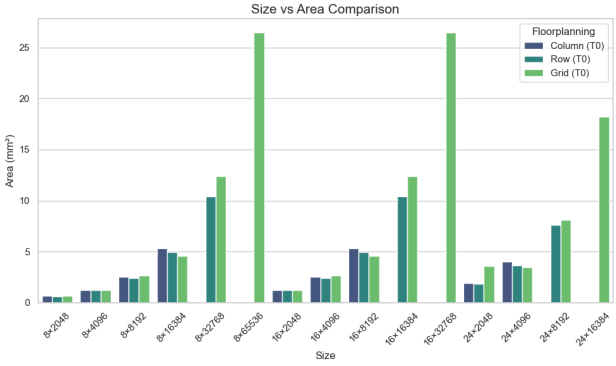


FIGURE 10. Results of size vs area comparison for T0 memory type with column, row and grid arrangement.

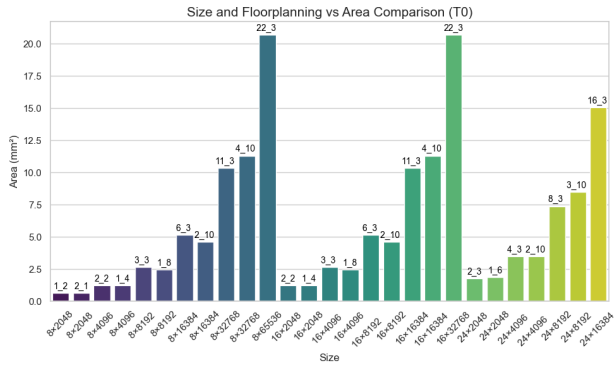


FIGURE 11. Results of size vs area comparison for T0 memory type with custom arrangement.

as discussed in section IV-C. This design choice affords flexibility for adapting the tool to incorporate the latest versions and improvements of memories generated with OpenRAM. However, considering that the memories within the SKY130 PDK have undergone rigorous testing and silicon manufacturing, we opted for the initial implementation to be based on these foundational memories ensuring that the memories generated by this tool can be manufactured. This decision doesn't prevent the possibility of integrating current and future enhancements.

The constraints configuration file holds a pivotal role in the OpenLane workflow. A well-optimized constraints file is essential for efficient system implementation at the layout level. The configurations elucidated in section IV-D and V-A were meticulously derived and fine-tuned through empirical methodologies. These methodologies prioritize two key factors: obtaining the GDSII file without violation errors and ensuring the uniformity of constraints across various memories. Consequently, the configuration file presents consistent restrictions for all memories, deviating only when selecting different base memory types and determining the

final memory area. The size of the ultimate memory is intricately linked to the quantity of memories used. While employing a generalized configuration file streamlines tool development, it may not achieve the best optimization for each individual memory, given its limitations regarding memory range.

Beyond the 151 memories featured in the preceding chapter, it is imperative to acknowledge that the tool may not autonomously generate memories without human intervention or errors within the RTL to GDSII flow. Nevertheless, the tool serves as a foundational starting point for generating memories beyond those covered in this research. Achieving this entails adjustments in the configuration file and/or memory positioning.

Fig. 7 illustrates the area utilization of each memory, revealing a notable trend when comparing implementations based on T1 and T2 memories. T2 memories tend to occupy less area, primarily due to their inherent capacity advantage. A T2 memory effectively implements double the storage capacity of a T1 memory without doubling the space requirement, consequently, implementing a T2 memory necessitates the instantiation of two T1 memories to achieve equivalent storage capacity. Furthermore, it is evident that higher-capacity memories can only be effectively implemented using a grid arrangement, with the principal constraint being routing congestion when adopting the column and row arrangements.

Fig. 8 and 9 showcase custom implementations utilizing T1 and T2 memories. While various possibilities exist for custom arrangements, our initial tests focused on comparing area utilization when employing more columns than rows and vice versa. The outcomes vary depending on the base memory type. With a T1 base memory, an increase in area is observed when using more columns, whereas with a T2 base memory, an increase in area occurs when using more rows than columns. The disparities in area utilization between implementing more rows or columns range from 1.4% to 6.1%.

Fig. 10 delves into area utilization with the base memory T0 in column, row, and grid arrangements. Upon inspecting the data, a discernible pattern emerges. When the number of instantiated memories is fewer than 16, and a row arrangement is employed, a tendency toward lower area utilization is evident. Conversely, when the number of memories surpasses 16, a smaller area footprint is attributed to the grid arrangement.

VII. CONCLUSION

This research presented a comprehensive framework called Macro Memory Cell Generator, designed to address the memory limitations of the SKY130 PDK by automating the generation of custom memory arrays with varying sizes and configurations. Through a meticulous empirical methodology, we optimized critical design parameters, resulting in over 150 memories successfully implemented at the layout level. Our work has shown that the tool can efficiently generate memories based on SKY130 IP memories while accommodating custom configurations with minimal adjustments. The tool's flexibility ensures compatibility with the latest advancements and improvements in memory design related to OpenRAM.

Furthermore, our study has shed light on the impact of memory arrangement on die area utilization. We observed that certain base memories, such as T2, offer advantages in terms of capacity-to-space efficiency, with grid arrangements proving particularly useful for high-capacity memories.

While the tool's capabilities are substantial, we acknowledge its limitations in generating memories beyond the scope of this research without human intervention. Nevertheless, it serves as a valuable starting point for memory designers, streamlining the memory creation process.

Ultimately, this work contributes to the advancement of memory design methodologies within the SKY130 PDK. It empowers designers to efficiently create memories tailored to specific requirements, facilitating more optimized chip designs. As we move forward, we anticipate further enhancements and applications of the Macro Memory Cell Generator, fostering innovation and efficiency in memory design.

REFERENCES

[1] Y.-K. Chen and S. Y. Kung, "Trend and challenge on system-on-a-chip designs," *J. Signal Process. Syst.*, vol. 53, nos. 1–2, pp. 217–229, Nov. 2008.

[2] Synopsys. (2021). *Memory Chip Design*. Accessed: Apr. 17, 2024. [Online]. Available: https://blogs.synopsys.com/from-silicon-to-software/2021/12/08/memory-chip-design/

[3] P.-H. Chen, "Intelligence everywhere: The challenges and opportunities for semiconductor designs," in *Proc. Int. Symp. VLSI Design, Autom. Test (VLSI-DAT)*, Hsinchu, Taiwan, Apr. 2021, pp. 1–2.

[4] A. R. Omondi, *The Microarchitecture of Pipelined and Superscalar Computers*. The Netherlands, Europe: Springer, 1999.

[5] S. Das and S. Dey, "FPGA based design of a fine-grained fault tolerant interleaved memory," in *Proc. IEEE Int. Conf. Adv. Commun., Control Comput. Technol.*, Ramanathapuram, India, May 2014, pp. 565–568.

[6] C.-H. Yang, Y.-C. Wu, Y.-L. Chen, C.-H. Lee, J.-H. Hung, and C.-H. Yang, "An FM-index based high-throughput memory-efficient FPGA accelerator for paired-end short-read mapping," *IEEE Trans. Biomed. Circuits Syst.*, vol. 17, no. 6, pp. 1331–1341, Dec. 2023.

[7] H. Lee, S. Lee, Y. Jung, and D. Kim, "T-CAT: Dynamic cache allocation for tiered memory systems with memory interleaving," *IEEE Comput. Archit. Lett.*, vol. 22, no. 2, pp. 73–76, Dec. 2023.

[8] B. Khargharia, S. Hariri, and M. S. Yousif, "An adaptive interleaving technique for memory performance-per-Watt management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 7, pp. 1011–1022, Jul. 2009.

[9] Dolphin Technology. *Products, Memory Products*. Accessed: Apr. 17, 2024. [Online]. Available: https://www.dolphin-ic.com/about/market.html

[10] R. Goldman, K. Bartleson, T. Wood, V. Melikyan, and E. Babayan, "Synopsys' educational generic memory compiler," in *Proc. 10th Eur. Workshop Microelectron. Educ. (EWME)*, Tallinn, Estonia, May 2014, pp. 89–92, doi: 10.1109/EWME.2014.6877402.

[11] Synopsys | EDA Tools. *Semiconductor IP and Application Security Solutions*. Accessed: Apr. 17, 2024. [Online]. Available: https://www.dolphin-ic.com/products/memory.html

[12] A. Parmar, K. Prasad, N. Rao, and J. Mekie, "FastMem: A fast architecture-aware memory layout design," in *Proc. 23rd Int. Symp. Quality Electron. Design (ISQED)*, Santa Clara, CA, USA, Apr. 2022, pp. 120–126, doi: 10.1109/isqed54688.2022.9806258.

[13] M. Guthaus, J. Stine, N. Chandramoorthy, J. Liu, N. Bafekrpour, S. Shah, M. Sarwar, R. Huang, and S. Sinha, "OpenRAM: An open-source memory compiler," Dept. Comput. Eng., Univ. California Santa Cruz, Santa Cruz, CA, USA, Tech. Rep., 2016. Accessed: Apr. 17, 2024. [Online]. Available: https://escholarship.org/content/qt8x19c778/qt8x19c778_noSplash_b2b3fbbb57f1269f86d0de77865b0691.pdf, doi: 10.1145/2966986.2980098.

[14] Google and SkyWater. *SkyWater Open Source PDK*. Accessed: Apr. 17, 2024. [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/
[15] M. Guthaus, H. Nichols, J. Cirimelli-Low, J. Kunzler, and B. Wu, "Enabling design technology co-optimization of SRAMs through open-source software," in *IEDM Tech. Dig.*, San Francisco, CA, USA, Dec. 2020, pp. 41.7.1–41.7.4, doi: 10.1109/IEDM13553.2020.9372047.
[16] A. Ghazy and M. Shalan, "OpenLane: The open-source digital ASIC implementation flow," in *Proc. Workshop Open-Source EDA Technol. (WOSET)*, 2020, pp. 1–5.
[17] H. Nichols, M. Grimes, J. Sowash, J. Cirimelli-Low, and M. R. Guthaus, "Automated synthesis of multi-port memories and control," in *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Cuzco, Peru, Oct. 2019, pp. 59–64, doi: 10.1109/VLSI-SoC.2019.8920314.

EMILIO ISAAC BAUNGARTEN-LEON received the bachelor's degree in biomedical electronics from Universidad Autnoma de Guadalajara, in 2019, and the Master of Science degree in electrical engineering (MSEE) from Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV), where he is currently pursuing the Ph.D. degree, concentrating on the implementation of digital systems at the layout level. During the master's study, he has collaborated on a research project with the German Aerospace Center.

SUSANA ORTEGA-CISNEROS received the degree in communications and electronics engineering from the University of Guadalajara, Mexico, the master's degree in science from the IPN Center for Research and Advanced Studies, Mexico, and the Ph.D. degree in computer science and telecommunications from the Autonomous University of Madrid, Spain. She is a Researcher in electronic design with CINVESTAV Unidad Guadalajara and a member of the National System of Researchers. Her research interests include integrated circuit design; physical design, layout, and simulation of digital integrated circuits; methodologies and algorithms for modeling, analysis, and design of electronic circuits; and handling of software for the development of integrated circuits.

GERMAN PINEDO-DIAZ received the degree in communications and electronics engineering from the Autonomous University of Zacatecas, and the Master of Science degree from CINVESTAV, Guadalajara, specialized in electronic design laboratory, where he is currently pursuing the Ph.D. degree. His area of expertise lies in embedded systems design and the application of artificial intelligence in medicine. His research interests include VLSI integrated circuit design, functional circuit verification, and designing accelerators for neural network operations on FPGA platforms.

MIGUEL ANGEL RIVERA ACOSTA received the Ph.D. degree in electrical engineering from the Center for Research and Advanced Studies of the National Polytechnic Institute (CINVESTAV) in 2020. Currently, he is a Design Verification Technical Leader with Circuify Semiconductors, and an Assistant Teacher at CINVESTAV. His research interests include ASIC design for digital image processing, computer vision, and hardware development for artificial intelligence algorithms inference.

FRANCISCO JAVIER RODRIGUEZ NAVARRETE received the M.S.E.E. degree in electronic design from Cinvestav Unidad Guadalajara, Mexico, in 2020, where he is currently pursuing the Ph.D. degree. He has collaborated with the government of Jalisco on projects that include the design of processors based on RISC-V and tools for integration and schematic data analysis, simulation, and emulation of various systems. His field of expertise includes computer architecture, digital logic design/verification using HDL, data analysis, test automation, and software development.

URIEL JARAMILLO-TORAL received the bachelor's degree from the Technological Institute of Aguascalientes, in 2019, and the Master of Science degree in electrical engineering (MSEE), specialized in encryption systems and the generation of integrated circuit layouts from Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV), where he is currently pursuing the Ph.D. degree with a focus on the implementation of digital systems at the circuit design level and the development of encryption systems resilient to quantum computers. During his master's studies, he actively collaborated on research projects related to these areas.

CRISTIAN TORRES GONZLEZ received the bachelor's degree in electrical and electronic engineering with a specialty in electronics from the Faculty of Engineering, National Autonomous University of Mexico (UNAM), in 2020. In 2022, he entered the Center for Advanced Studies of the National Polytechnic Institute (CINVESTAV), Guadalajara Campus, where he actively participates in the area of digital systems.

JUAN CARLOS GARCIA LOPEZ received the degree from the University of Guadalajara, in 2020. He is currently pursuing the master's degree in electrical engineering (MSEE) with Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV). He is an Electronics and Communications Engineer. He has been involved in research projects at the institution. His current focus is on the implementation of asynchronous systems and neural networks at the RTL design level as he works toward the completion of his master's degree. He received the CENEVAL EGEL Prize for Academic Excellence.