SECURE ASIC DESIGN FLOW FOR REVERSE ENGINEERING PREVENTION

A Thesis submitted to the faculty of
San Francisco State University
In partial fulfillment of
the requirements for
the Degree

Master of Science

In

Engineering: Embedded Electrical and Computer Systems

by

Miklesh Naicker

San Francisco, California

May 2019

CERTIFICATION OF APPROVAL

I certify that I have read *Secure ASIC Design Flow for Reverse Engineering Prevention* by Miklesh Naicker, and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirement for the degree Master of Science in Embedded Electrical and Computer Systems at San Francisco State University.

Dr. Hamid Mahmoodi, Ph.D.
Professor of Electrical/Computer Engineering

Dr. Fatemeh Tehranipoor, Ph.D.
Associate Professor of Electrical/Computer Engineering

SECURE ASIC DESIGN FLOW FOR REVERSE ENGINEERING PREVENTION

Miklesh Naicker
San Francisco, California
2019

Due to the rise of fabrication-less Integrated Circuit (IC) design companies that outsource IC manufacturing to third parties, hardware obfuscation is crucial in protecting ICs from reverse engineering. A Look-Up Table (LUT) built using Non-Volatile (NV) latches will be used to replace logic gates to obfuscate designs. Once a LUT is programmed with the correct configuration bits, it enables the LUT to function exactly like a target logic gate and is difficult to reverse engineer since the LUTs are empty memory cells during manufacturing. The NV latches used in the construction of the LUTs is based on eFuse technology that is provided as hard macros from manufactures. To utilize these LUTs in ASIC designs, the normal ASIC flow will be modified to accommodate multiple hard macros during synthesis and automated process of placement, power routing, and net routing during physical design. This will be beneficial to IC designers as they can easily obfuscate their designs using LUTs with minimal overheard during the design process.

PREFACE AND/OR ACKNOWLEDGEMENTS

I would like to take this opportunity to express my profound gratitude to Dr. Hamid Mahmoodi for his excellent guidance, patience, valuable input and support towards the completion of my project. I would also like to thank Dr. Mahmoodi for providing me with Synopsys training and software tools required for the completion of my project. Finally, I would also like to thank my committee member Dr. Fatemeh Tehranipoor for her time.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

1. Introduction 1

    1.1 Why Do We Need Hardware Security? 1

    1.2 Non-Volatile Look Up Tables (NV-LUT) 2

    1.3 Proposed ASIC Flow 3

2. Background for Prototype Chip 5

    2.1 NV-LUTs Made from TSMC eFuse Macro 5

    2.2 Overall Chip Architecture 6

    2.3 Thesis Outline and Steps Required for Completion 7

3. RTL Design 8

    3.1 Core Top 8

    3.2 I/O Cells 11

        3.2.1 Input Only I/O Cells 12

        3.2.2 Output Only I/O Cells 12

        3.2.3 Bi-Directional I/O Cells 13

    3.3 Chip Top 14

    3.4 GPIO 14

        3.4.1 Addresses for Each Benchmark 15

        3.4.2 Clock Gating 16

        3.4.3 Serial to Parallel Input and Parallel to Serial Output 17

3.4.3.1 AES 18
3.4.3.2 DES Area 21
3.4.3.3 DES Performance 23
3.4.3.4 SHA3 (Keccak 512) 26
3.4.4 Direct Connection to Top Level 28
3.4.4.1 B01 30
3.4.4.2 B02 30
3.4.4.3 B04 31
3.4.4.4 B12 31
3.4.4.5 Custom ALU 32
3.4.5 Inout Port for CPU 32
3.4.5.1 CPU 34
4. RTL Verification 35
5. Initial Synthesis 35
5.1 Synthesis Setup 36
5.2 Multi-Corner Setup 37
5.3 Timing Constraints 40
5.4 Additional Synthesis Notes 43
6. Gate Level Verification 44
7. Gate Selection and Replacement 44
7.1 Top Level Modification for Obfuscated Benchmarks 45
8. Re-Synthesis with Design for Test (DFT) 47
8.1 Synthesis Results 47
9. Post Obfuscation Gate Level Verification 56

10. Physical Design 57

10.1 Floor Planning 57

10.2 Placement 60

10.3 Clock Tree Synthesis (CTS) 69

10.4 Routing 70

10.5 Physical Design Results 72

11. Signoff 74

12. Future Work 74

References 76

LIST OF TABLES

Table 1: Benchmarks in Prototype Chip 6
Table 2: Top Level Pin Summary 9
Table 3: Address Table 16
Table 4: B01 Port Connections 30
Table 5: B02 Port Connections 30
Table 6: B04 Port Connections 31
Table 7: B12 Port Connections 31
Table 8: Custom ALU Port Connections 32
Table 9: CPU Port Connections 34
Table 10: Scenario Information 40
Table 11: Clock Information 41
Table 12: Scan Chain Connection for SRAM Version of Benchmarks 46
Table 13: Synthesis Results 47
Table 14: Synthesis Results Continued 48
Table 15: Average, Minimum, and Maximum Area Overhead 53
Table 16: Average, Minimum, and Maximum Power Overhead 54
Table 17: Summary 73

LIST OF FIGURES

Figure 1: NV-LUT Structure 2
Figure 2: Proposed ASIC Flow 3
Figure 3: Utilization of eFuse Bits 5
Figure 4: Overall Architecture of Chip 7
Figure 5: Overview of Core Top 11
Figure 6: Input Only I/O Cell 12
Figure 7: Output Only I/O Cell 12
Figure 8: Bi-Directional I/O Cell 13
Figure 9: Overview of Chip Top 14
Figure 10: Clock Gating Cells 17
Figure 11: Loading Data for the AES Benchmark 18
Figure 12: Loading Data for the AES Benchmark Continued 19
Figure 13: AES Operation 20
Figure 14: AES Output Results 20
Figure 15: Loading Data for the DES Area Benchmark 21
Figure 16: DES Area Operation 22
Figure 17: DES Area Output Results 23
Figure 18: Loading Data for the DES Perf Benchmark 24
Figure 19: DES Perf Operation 25
Figure 20: DES Perf Output Results 25
Figure 21: Loading Data for the SHA3 Benchmark 26
Figure 22: SHA3 Operation 27
Figure 23: SHA3 Output Results 28
Figure 24: Combinational Logic for Directly Connected Benchmarks (Input Side) 29
Figure 25: Combinational Logic for Directly Connected Benchmarks (Output Side) 29
Figure 26: Basic Inout Port 32

Figure 27: CPU Inout Port Connection to I/O Cell.. 34
Figure 28: Synthesis: Topographical Mode vs Standard Mode....................................... 35
Figure 29: Generated Clocks 42
Figure 30: Configuration Bit Converter.. 45
Figure 31: Scan Chain Connection of SRAM Benchmarks.. 46
Figure 32: Total Area of Benchmarks 49
Figure 33: Total Power of Benchmarks 50
Figure 34: Area Overhead.. 51
Figure 35: Power Overhead 52
Figure 36: Average, Minimum, and Maximum Area Overhead Graph........................... 53
Figure 37: Average, Minimum, and Maximum Power Overhead Graph 55
Figure 38: Floor Planning Screenshot.. 59
Figure 39: Power Structure Screenshot.. 62
Figure 40: Power IO Connections Screenshot 63
Figure 41: ESD and eFuse Macro Connections Screenshot 63
Figure 42: Bond Pads... 64
Figure 43: IO Ring 65
Figure 44: IO Ring Screenshot 66
Figure 45: Placement Screenshot... 67
Figure 46: Standard Cell Power and Ground Rails Screenshot 68
Figure 47: CTS Screenshot 70
Figure 48: Routing Screenshot... 72
Figure 49: Final Chip Layout... 73

1. Introduction

1.1 Why Do We Need Hardware Security?

Currently, the cost of owning and operating foundries that can manufacture Integrated Circuits (IC) at advanced nodes are upwards of two to three billion dollars. This has led to the rise of fabless IC design companies that design ICs but rely on other third-party companies that have the manufacturing capability to manufacture their ICs for them. Usually, these companies are located throughout the globe and this creates a complex problem of ensuring trust between the different organizations as Intellectual Property (IP) laws vary widely from country to country [1] [3]. Once the design company submits the physical layout for manufacturing, any organization with adequate resources can easily determine the entire netlist and capabilities of the design by analyzing the physical layout. Due to competition in the market, this would be a huge loss for the IC design companies and since IP laws are enforced differently in each county, this leaves the IC designs to be susceptible to IP Piracy, IP Modification, IP Overuse, and Reverse Engineering (RE) [1] [3]. A survey in the semiconductor industry by Semiconductor Equipment and Materials International (SEMI) reported that 90% of semiconductor companies have experienced IP infringement and 54% of those were reported as extremely serious [2].

To combat these issues, IC designers can deploy different methods of hardware obfuscation. Hardware obfuscation is a way of concealing the important aspects of an IC design to protect the designs from the vulnerabilities listed above. There are different methods of hardware obfuscation that IC designers can use depending on the level of obfuscation the designers need and how much overhead costs such as power, area, and delay they are willing to accept.

1.2 Non-Volatile Look Up Tables (NV-LUT)

We have proposed using Non-Volatile Look Up Tables (NV-LUT) as a way to obfuscate a design. A simple structure of an NV-LUT is shown below in Figure 1.

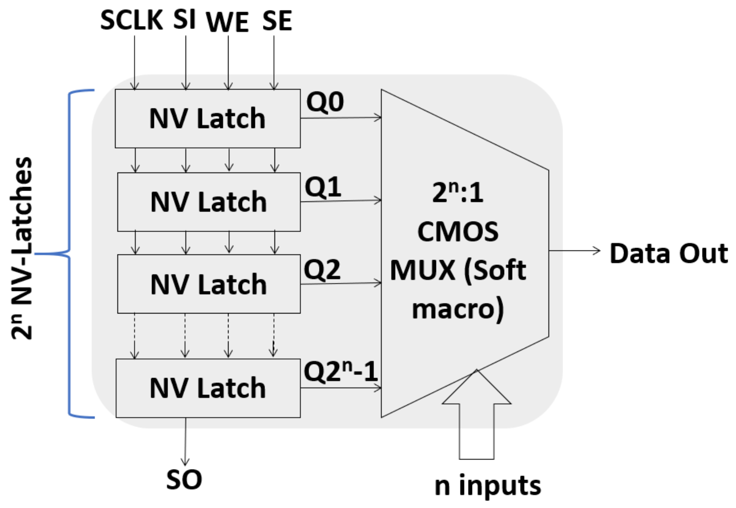

Figure 1: NV-LUT Structure

An NV-LUT can be programmed to function like any logic gate. For example, as shown in Figure 1, the NV-LUT functions as a two input AND gate, NV latches connected to Q0, Q1, and Q2 need to be programmed low (0) and NV latch connected to $Q2^N$ will be programmed high (1). The MUX will have two input lines, A and B. If A and B are both low, the MUX will select Q0 which is low. If A is low and B is high, the MUX will select Q1 which is low. If A is high and B is low, the MUX will select Q2 which is low. Finally, If A and B are both high, the MUX will select $Q2^N$ which is high. The NV-LUT can be scaled for an N number of inputs and can function like any gate when programmed correctly. Since these LUTs will be made up of non-volatile latches, they only need to be programmed once after manufacturing.

The NV-LUT is secure from a RE point of view because the NV latches are empty memory cells in the physical layout. The NV-LUT only acts as a target gate when programmed correctly by the designers after manufacturing; therefore, at the physical layout, the value of the memory cells are not known to any organization trying to reverse engineer the design. They may try all the possible combinations, but when an appropriate number of gates are replaced with NV-LUT, the possible combinations are exponential. The time, effort, and money required to try all the combinations is a good deterrent for any organization to not reverse engineer the design.

1.3 Proposed ASIC Flow

To implement hardware obfuscation using NV-LUTs, the general Application Specific Integrated Circuit (ASIC) flow needs to be modified to the one shown below.

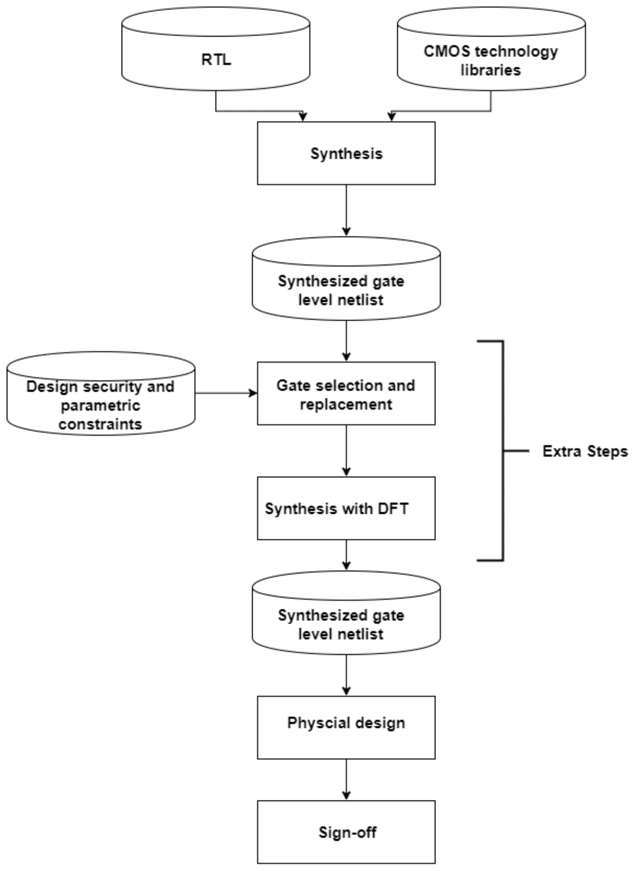
RTL
CMOS technology libraries
Synthesis
Synthesized gate level netlist
Design security and parametric constraints
Gate selection and replacement
Synthesis with DFT
Extra Steps
Synthesized gate level netlist
Physcial design
Sign-off

Figure 2: Proposed ASIC Flow

The starting point of any ASIC design is Register Transfer Level (RTL) code in either Verilog or VHDL. The RTL along with technology libraries and design constraints are given to the Design Compiler or Synthesizer which outputs a synthesized netlist using only the logic gates available in the technology libraries while meeting the timing constraints. This synthesized netlist now can be used for physical design and signoff. For hardware obfuscation with NV-LUTs, two extra steps are required as shown above in Figure 2. The initial synthesized netlist goes through gate section and replacement. In this step, the netlist is analyzed to determine how many and which logic gates within the design need to be replaced with NV-LUTs to have a design that is not easily reverse engineered. After the gate replacement, the initial synthesized netlist now has some RTL constructs. This netlist is now synthesized one more time with the same timing constraints and the resulting synthesized netlist will be used for physical design and signoff.

2. Background for Prototype Chip

2.1 NV-LUTs Made from TSMC eFuse Macro

We have chosen to use non-volatile latches made from eFuse technology. The eFuse latches are provided to us as 32-bit macros from TSMC. All 32 bits of every macro will be utilized regardless of LUT size as illustrated in Figure 3.

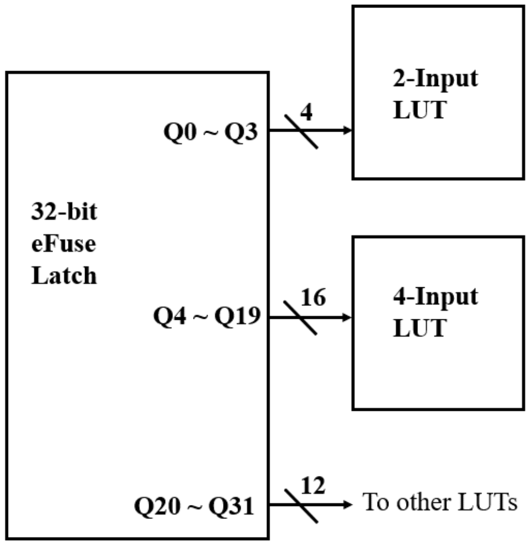

Figure 3: Utilization of eFuse Bits

As shown in Figure 3, a two input LUT needs four latches which are connected to bits Q0 to Q3 of the macro. A four input LUT needs 16 latches which are connected to bits Q4 to Q19 of the macro. Similarly, the remaining 12 bits of the macro can be connected to other LUTs. All 32 bits of the macro does not necessarily need to be connected to only one LUT.

2.2 Overall Chip Architecture

To showcase that NV-LUT based obfuscation can work for designs of different kinds and various complexity the following designs were chosen to be included in the prototype chip.

Source	**Benchmark**	**Description**	**Number of Logic Gates**	**Number of Sequential Gates**	**Number of Total Gates**
OpenCores	DES_perf	DES optimized for performance	13,931	1,984	15,915
OpenCores	DES_area	DES optimized for area	2,023	64	2,087
OpenCores	AES	AES cipher	10,225	554	10,779
OpenCores	SHA3	SHA3 Encryption core (Keccak 512)	11,445	2,245	13,690
ITC'99	b04	Compute min and max	237	69	306
ITC'99	b12	1-player game (guess sequence)	2,252	416	2,668
ITC'99	b01	FSM that compares serial flows	30	5	35
ITC'99	b02	FSM that recognizes BCD numbers	22	4	26
Custom	ALU	Multiplier/Adder/AND	136	0	136
OpenCores	Processor	8-bit Microprocessor	1,389	183	1,572

Table 1: Benchmarks in Prototype Chip

As shown above, there are four encryption cores, one processor core, one Arithmetic and Logic Unit (ALU), and four generic sequential circuit benchmarks. For each benchmark, there are four versions included in the prototype chip: original which is unobfuscated, low obfuscated which has a low-level of obfuscation, medium obfuscated which has a medium-level of obfuscation, and high obfuscated which has a high-level of obfuscation. The low and medium obfuscated versions of the benchmarks will utilize the eFuse registers for logic obfuscation key storage. The high obfuscated versions of the benchmarks store the key in SRAM based LUTs. A General-Purpose Input Output (GPIO) is designed to reduce the chip pin count by connecting only one version of a selected benchmark to the chip level at any given time. The GPIO also performs serial to parallel conversion and parallel to serial conversion for high pin count benchmarks to keep the chip pin count manageable while reducing the area of the entire chip. Figure 4 shows the overall architecture of the entire chip.

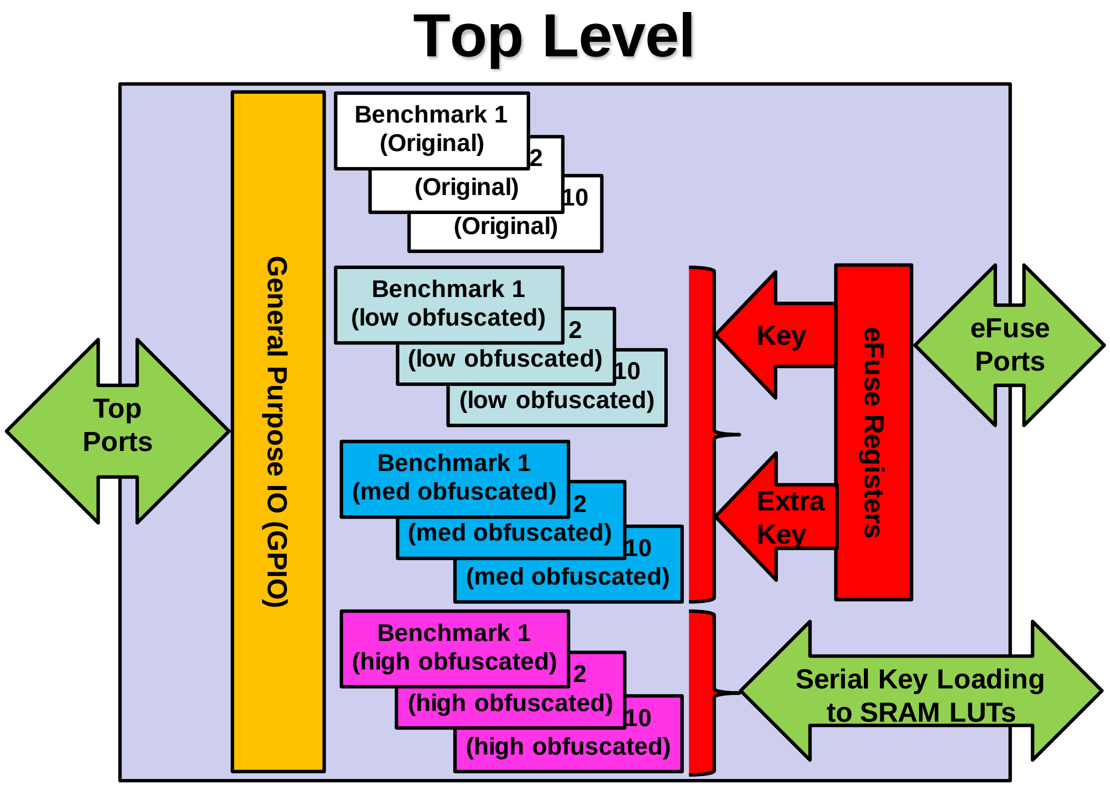

Figure 4: Overall Architecture of Chip

2.3 Thesis Outline and Steps Required for Completion

To have an ASIC chip using NV-LUT based obfuscation ready for manufacturing, there are numerous steps and verifications that need to be completed to get from RTL to physical layout. The steps are listed in order below.

1) Design of RTL
2) *RTL Verification*

3) Initial Synthesis
4) *Gate Level Verification*
5) *Gate Selection and Replacement*
6) Re-Synthesis with *Design for Test (DFT)*
7) *Post Obfuscation Gate Level Verification*
8) Physical Design
9) *Signoff*

The steps that are italicized will not be discussed in this paper. The design of the entire RTL will be discussed in detail. The scripts for synthesis and physical design will also be discussed and the important commands will be highlighted.

3. RTL Design

3.1 Core Top

The decision was made to have a total of 76 pins at the top level that would be used for different purposes. The following table summarizes all the pins at the top level.

Signal Name	Direction	Size (bits)
address	input	6
clk	input	1
reset	input	1
inData	input	4
inData_bi	inout	12
outData	output	8
outData_bi	inout	8
SCLK	input	1
SI	input	1
SO	output	1
enable	input	1
SEL	input	5
PGM	input	1
FCLK	input	1
DIN	input	1
VDDQ	input	3
VSSQ	input	2
DOUT	output	1
scanEn	input	1
testmode	input	1
core VDD	input	4
core VSS	input	4
IO VDD	input	4
IO VSS	input	4
	Total	76

Table 2: Top Level Pin Summary

One pin was needed for a global clock signal that would be shared amongst all the designs and a reset pin for the GPIO. There are ten benchmarks and four different versions of each benchmark, therefore in total 40 benchmarks. A 6-bit address port was chosen because $2^6 = 64$, which is sufficient to individually select each benchmark. For input, 16 bits are used for input which are made of a combination of inout ports and input only ports (inData 4 bits and inData_bi 12 bits). Similarly, 16 bits are used for output which are made from a combination of inout ports and output only ports (outData 8 bits and outData_bi 8 bits). The use of inout ports provide flexibility and help keep the entire chip pin count manageable. The input ports are also required for the CPU benchmark because the CPU has its own 8-bit inout port which it uses to read/write to memory. The prototype chip has three clock domains, clk is the main clock that is used by all designs and clock gating of this signal is discussed in detail later, SCLK is required for the scan chain of the SRAM versions of the obfuscated designs, and FCLK is required for the programming of the eFuse macros. SI and SO are used when programming the SRAM-LUTs. The enable, SEL, PGM, DIN, VDDQ, and DOUT signals are required for the programming of the eFuse macros. Since 18 eFuse macros will be used, SEL was chosen to be 5 bits so the decoding logic can individually select each eFuse macro ($2^5 = 32$ which is greater than the number of fuses, 18). VDDQ is a special high voltage programming pin and one VDDQ pin can be used for a maximum of ten eFuse macros. To satisfy this electrical requirement, three VDDQ pins will be used. Finally, the testmode and scanEn pin are needed for DFT and there are 16 power pins for the entire prototype chip.

Since the GPIO is an internal component, it has no pin limit. The GPIO has separate ports of the top level and for each individual benchmark. The inout ports are split into input only and output only ports as shown in the figure below.

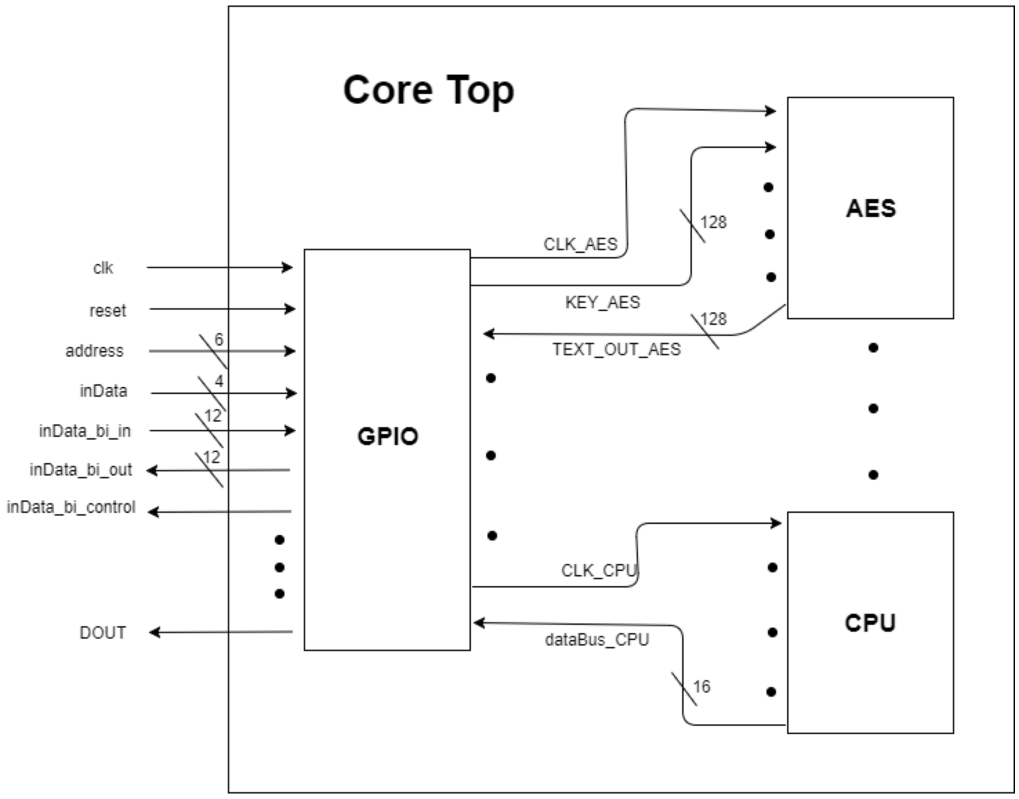

Figure 5: Overview of Core Top

The inout ports are split into input and output only because these signals are then connected to the bi-directional I/O cells. For example, inData_bi is now split into inData_bi_in, inData_bi_out, and inData_bi_control (control signal needed for the I/O cell that is internally generated by the GPIO).

3.2 I/O Cells

I/O cells are special cells that handle signal transmission from the chip or to the chip. It is part of the interface to the external environment. Due to this reason, I/O cells have the ability to drive big loads, it provides voltage consistency, has low switching noise, and provides Electrostatic Discharge (ESD) protection. All signals of any chip must go through I/O cells. There are different types of I/O cells which include input only, output only, bi-directional and even power I/O cells.

3.2.1 Input Only I/O Cells

The input only I/O cell used in this prototype chip is shown below.

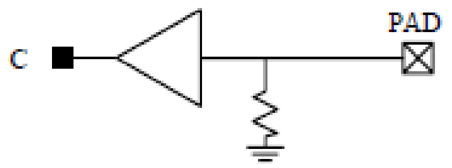

Figure 6: Input Only I/O Cell

The variable C is the actual input signal and PAD is where the signal is applied externally. This I/O cell has a pull-down feature; therefore, if an external signal is not applied, C will be pulled down to ground. Similar to Figure 6, clock signals use an input only I/O cell with a built-in Schmitt Trigger. Schmitt Trigger's help remove noise from a signal and since the clock signal is always changing, using an I/O cell with a built-in Schmitt Trigger can help reduce the noise in the clock signal.

3.2.2 Output Only I/O Cells

The output only I/O cell used in this prototype chip is shown below.

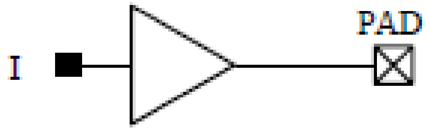

Figure 7: Output Only I/O Cell

The variable I is the actual output signal and PAD is where an external device will be connected to. Since this prototype chip will have to drive test equipment that may have a large capacitance and resistance, the biggest and strongest output only I/O cell was used.

3.2.3 Bi-Directional I/O Cells

The bi-directional I/O cell used in this prototype chip is shown below.

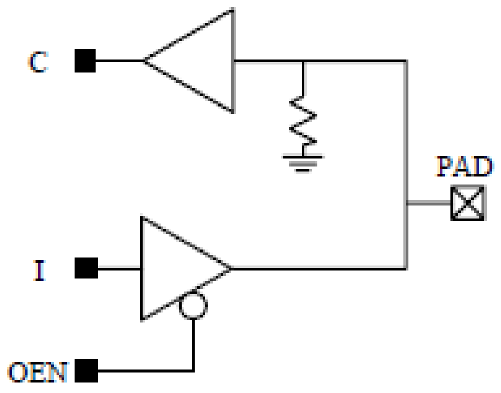

Figure 8: Bi-Directional I/O Cell

The bi-directional I/O cell has an active low tri-state buffer. When OEN is low, the signal I is passed to the PAD (output Mode). When OEN is high, the tri-state buffer is in a high Z mode; therefore, an external device connected to the PAD can drive an input signal through the PAD to C (input mode). For example, inData_bi_in (input only) would be connected to C, inData_bi_out (output only) would be connected to I, and inData_bi_out_control (control signal) would be connected to OEN.

3.3 Chip Top

Knowing the information presented in the previous section, ports from core top are connected to the I/O cells in chip top. The ports from chip top resemble those described in Table 2. This is illustrated in the figure below.

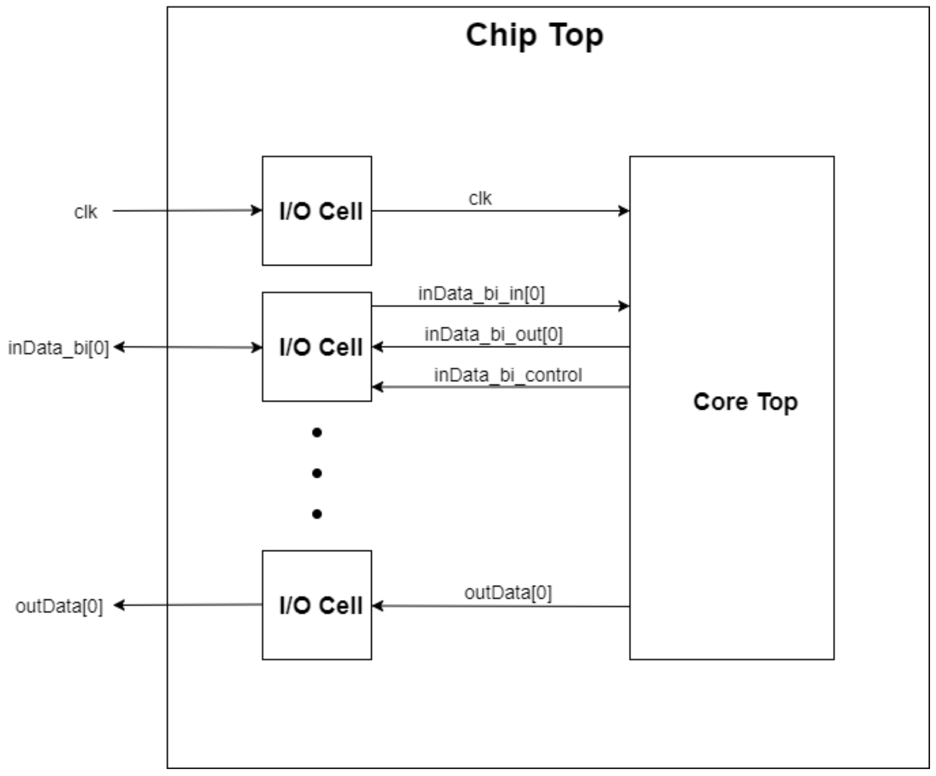

Figure 9: Overview of Chip Top

3.4 GPIO

Due to the area restriction of the prototype chip, the pin count of the entire design was limited to about 76 pins; however, some of the benchmarks listed in Table 1 have pin counts in the hundreds. To overcome this, a custom GPIO was designed to handle serial to parallel input, parallel to serial output, and direct connections for those benchmarks with small pin counts. The decision was made for the GPIO to have 40 pins that would be used for the I/O of all the benchmarks, which ensured the area requirement is met. Only one version of the selected benchmark is connected to chip top at any given time via an addressing mechanism within the GPIO. The GPIO also applies the correct signals of each benchmark based on the selected benchmarks' timing requirements.

3.4.1 Addresses for Each Benchmark

Each benchmark has its own separate address, the following table lists the address for each benchmark. A benchmark name followed by LUT and medium/high implies the obfuscated version of the benchmark using eFuse macros and the level of obfuscation. A benchmark name followed by SRAM implies the obfuscated version of the benchmark using SRAM-LUTs.

Address	Benchmark	Address	Benchmark
0	AES	21	B04 LUT medium
1	AES LUT medium	22	B04 LUT high
2	AES LUT high	23	B04 SRAM
3	AES LUT SRAM	24	B12
4	DES Area	25	B12 LUT medium
5	DES Area LUT medium	26	B12 LUT high
6	DES Area LUT high	27	B12 SRAM
7	DES Area SRAM	28	Custom ALU base
8	DES Perf.	29	Custom ALU LUT medium
9	DES Perf. LUT medium	30	Custom ALU LUT high
10	DES Perf. LUT high	31	Custom ALU SRAM
11	DES Perf. SRAM	32	SHA3
12	B01	33	SHA3 LUT medium
13	B01 LUT medium	34	SHA3 LUT high
14	B01 LUT high	35	SHA3 SRAM
15	B01 SRAM	36	CPU
16	B02	37	CPU LUT medium
17	B02 LUT medium	38	CPU LUT high
18	B02 LUT high	39	CPU SRAM
19	B02 SRAM	40	All Clocks
20	B04		

Table 3: Address Table

The GPIO contains a special mode called All Clocks mode at address 40 where all the clocks in the design will be active.

3.4.2 Clock Gating

The designs have been manually clock gated to ensure power is not wasted by having an active clock for a design that is not being used. This will give us a better power consumption measurement post fabrication for each design. Clock gating is done by manually inserting a clock gating cell (CKLNQD8) provided by TSMC into the RTL. As shown in the figure below, the clock gating cell have four ports. TE (input port) is used when DFT is added to the design so, for now, they are grounded, E (input port) is the enable signal for the clock gating cell, CP (input port) is the input clock and Q (output port) is the output clock. All benchmarks receive the same global clock, if the control signal of the benchmark is high (control signal is changed to high if the corresponding address for the benchmark is selected), then the global clock is passed to the clock of the benchmark. For example, if aesClock is high, then clk_aes follows clk. All other control signals will be low, therefore the clock for the other benchmarks will be grounded.

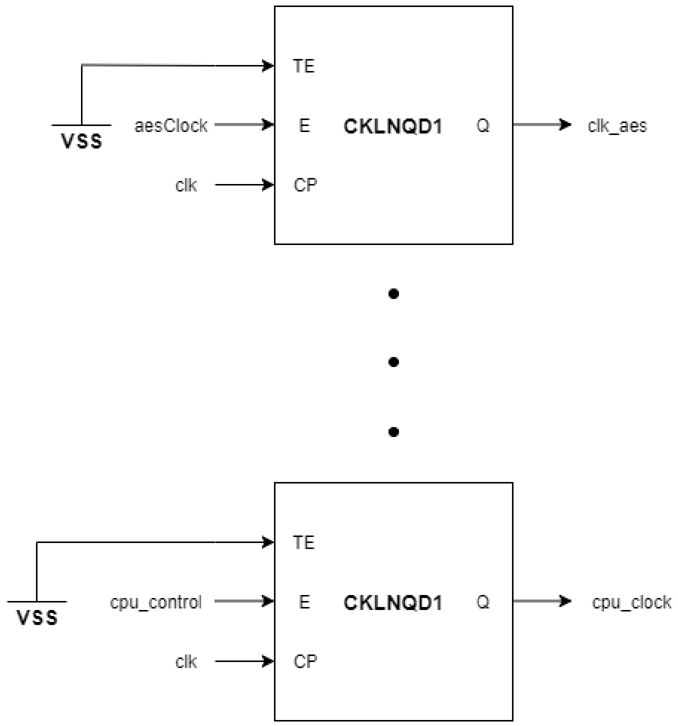

Figure 10: Clock Gating Cells

3.4.3 Serial to Parallel Input and Parallel to Serial Output

For benchmarks that have more pins than chip top, a serial to parallel input and parallel to serial output feature is implemented within the GPIO. Data can be sent to the GPIO, 16 bits per clock cycle which are stored within internal registers. Once the GPIO has all the data needed for the selected benchmark, the GPIO asserts all the controls signals for the benchmark and passes the input data to the benchmark. When the benchmark is done, the GPIO stores the result in an internal register. Finally, the GPIO outputs the results 16 bits per clock cycle. The user only needs to feed the data into chip top that is required for each benchmark. The GPIO is designed to correctly apply all the control signals (reset, load, etc…) with the correct timing requirements for every benchmark. Once the GPIO outputs the result of any benchmark, the GPIO needs to be reset before a new input vector is applied to the same benchmark or when switching to a new benchmark. The following sections document the internal control signals between the GPIO and each benchmark that requires serial to parallel conversion and vice versa. The signals labeled chip top are external signals to the GPIO and the signals identified by the benchmark name are from the GPIO to the benchmark.

3.4.3.1 AES

To begin, the address of the AES benchmark must be applied (in this case, 0) and the reset signal (asynchronous active low) needs to be low for at least one clock cycle. After the GPIO is reset, every clock cycle after the data needed for the AES benchmark needs to be fed in the inData (4 bits) and inData_bi (12 bits) ports. In total, 16 bits per clock

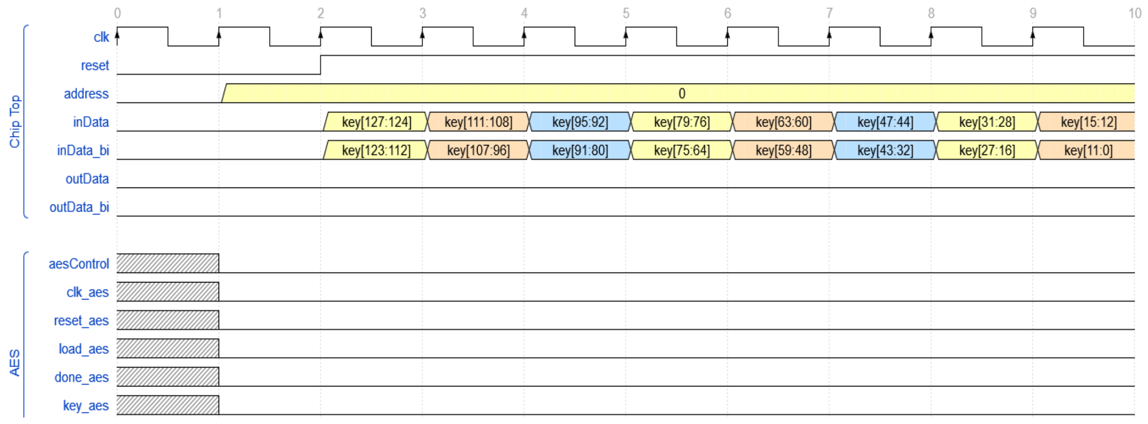

Figure 11: Loading Data for the AES Benchmark

cycle with the higher 4 bits fed to the inData port and the remaining bits to the inData_bi port. The AES benchmarks requires 128-bit key and text data. The loading sequence is shown below.

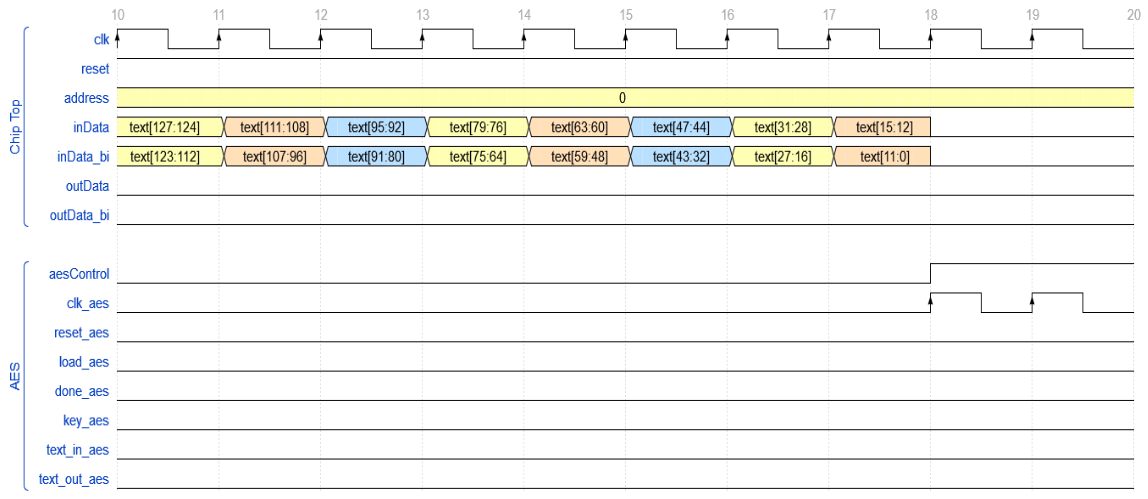

Figure 12: Loading Data for the AES Benchmark Continued

The GPIO is storing all this data into specific internal registers only for this benchmark. While this is occurring as shown in the figures above, all the signals for the AES benchmark are low. Once all the data has been loaded into the GPIO, the GPIO sets the control signal of the AES benchmark high. This enables the clock for the AES benchmark and since this benchmark has an active low reset, the GPIO resets the benchmark for at least two clock cycles.

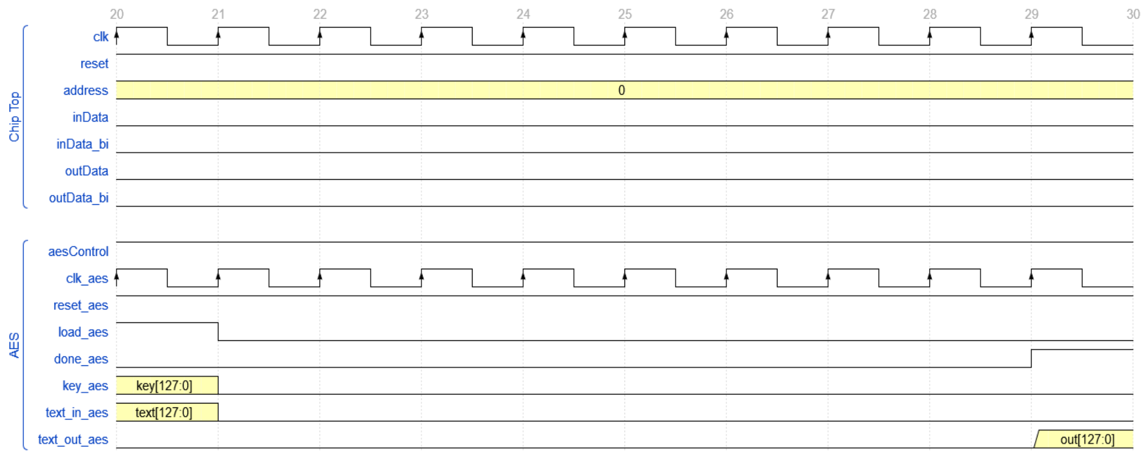

Figure 13: AES Operation

Once the benchmark is reset, the load signal is high for one clock cycle where the key and text for the AES benchmark is applied to the benchmark. Then load, key and text signals are set to low. The GPIO now waits for the AES encryption to be completed. Once the encryption is completed, the done signal is set high by the benchmark and the encrypted data is now available on the test_out_aes port (128 bits) which is stored in an internal register by the GPIO. This is illustrated in the figure above.

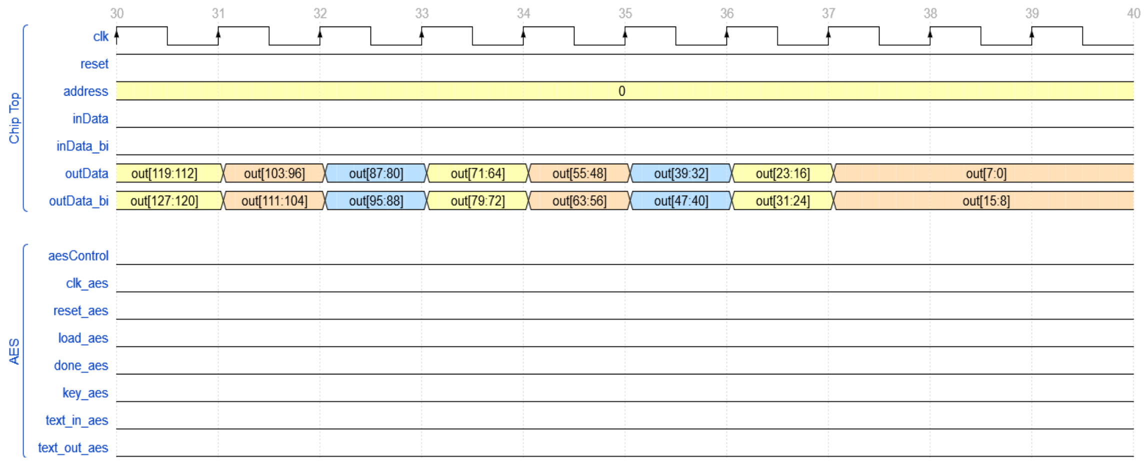

Figure 14: AES Output Results

As shown in Figure 14, once the GPIO has the results from the AES benchmark, all the signals for the AES benchmark are set to low and the GPIO now feeds the results to outData and outData_bi ports, 16 bits per clock cycle. The higher 8 bits of the 16-bit packet is fed to outData_bi and the remaining 8 bits are fed to outData. The last packet of data is held on both outData and outData-bi ports. The GPIO is now at an empty state and needs to be reset before any new input vector is applied to any of the benchmarks.

3.4.3.2 DES Area

To begin, the address of the DES Area benchmark must be applied (in this case, 4) and the reset signal (asynchronous active low) needs to be low for at least one clock cycle. After the GPIO is reset, every clock cycle after the data needed for the DES Area benchmark needs to be fed in the inData (4 bits) and inData_bi (12 bits) ports. In total, 16 bits per clock cycle with the higher 4 bits fed to the inData port and the remaining bits to the inData_bi port. The loading sequence for the DES Area benchmark is shown below.

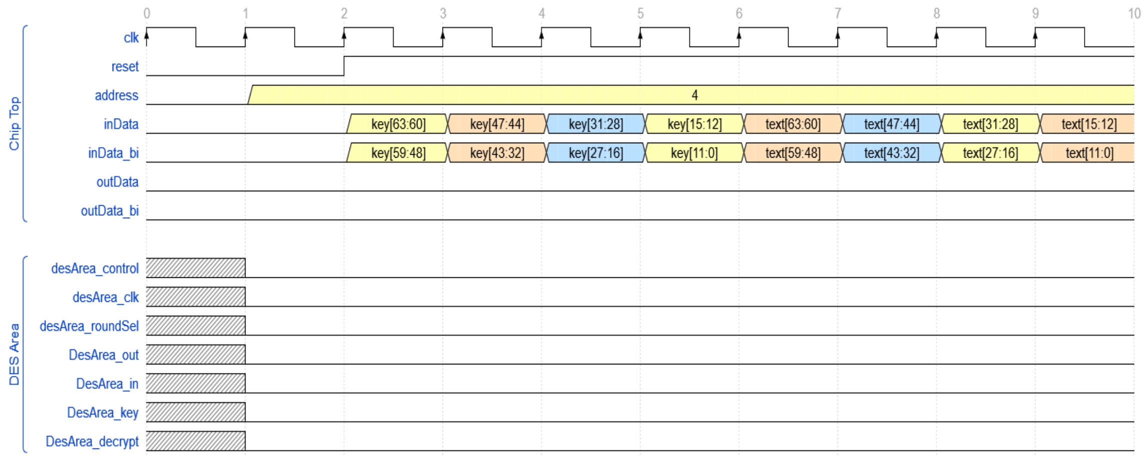

Figure 15: Loading Data for the DES Area Benchmark

The DES Area benchmark has a 56-bit key, therefore key[55:0] is the actual key. The key bit[56] is used to select if the DES Area benchmark will encrypt or decrypt the text. If key[56] is low, then the benchmark is decrypting and if key[56] is high, then the benchmark is encrypting. The key bits key[63:57] are discarded; therefore, they can either be low or random data as it does not affect the operation of the GPIO. The GPIO is storing all this data into specific internal registers only for this benchmark. While this is occurring as shown in the figure above, all the signals for the DES Area benchmark are low. Once all the data has been loaded, the GPIO starts asserting different control signals for the DES Area benchmark.

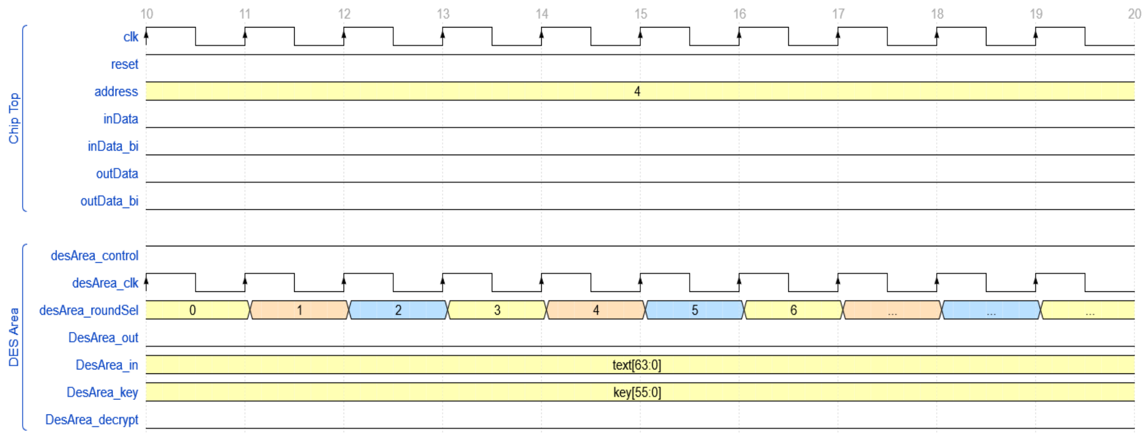

Figure 16: DES Area Operation

The DES Area benchmark does not have a reset signal, so once the data is loaded the GPIO sets the control signal for the DES Area benchmark to high which enables the clock for this benchmark. The roundSel signal now starts counting from 0 to 15 every clock cycle and the key and text input is applied to the benchmark throughout. On the 16th clock cycle, the encrypted data will be available.

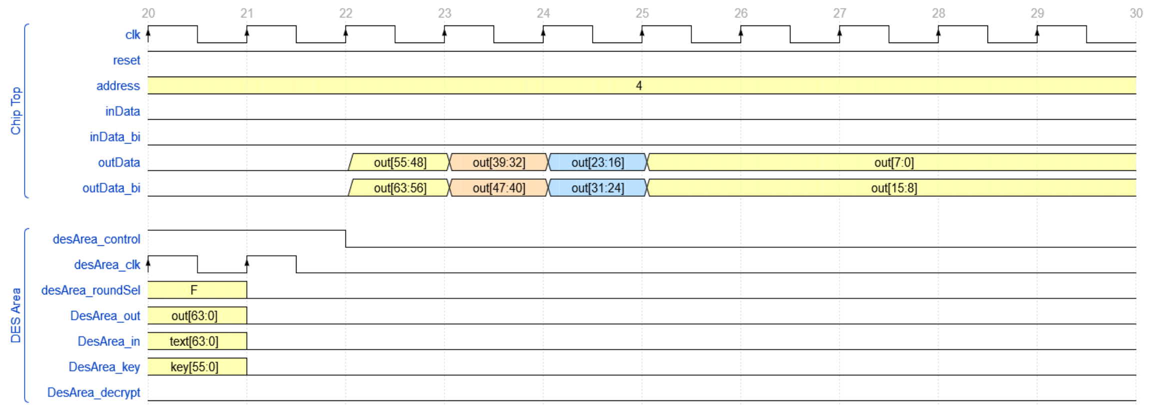

Figure 17: DES Area Output Results

As shown in Figure 17, once the GPIO has the results from the DES Area benchmark, all the signals for the DES Area benchmark are set to low and the GPIO now feeds the results to outData and outData_bi ports, 16 bits per clock cycle. The higher 8 bits of the 16-bit packet are fed to outData_bi and the remaining 8 bits are fed to outData. The last packet of data is held on both outData and outData_bi ports. The GPIO is now at an empty state and needs to be reset before any new input vector is applied to any of the benchmarks.

3.4.3.3 DES Performance

To begin, the address of the DES Perf Benchmark must be applied (in this case, 8) and the reset signal (asynchronous active low) needs to be low for at least one clock cycle. After the GPIO is reset, every clock cycle after the data needed for the DES Perf benchmark needs to be fed in the inData (4 bits) and inData_bi (12 bits) ports. In total, 16 bits per clock cycle with the higher 4 bits fed to the inData port and the remaining bits to the inData_bi port. The loading sequence for the DES Perf benchmark is shown in the figure below.

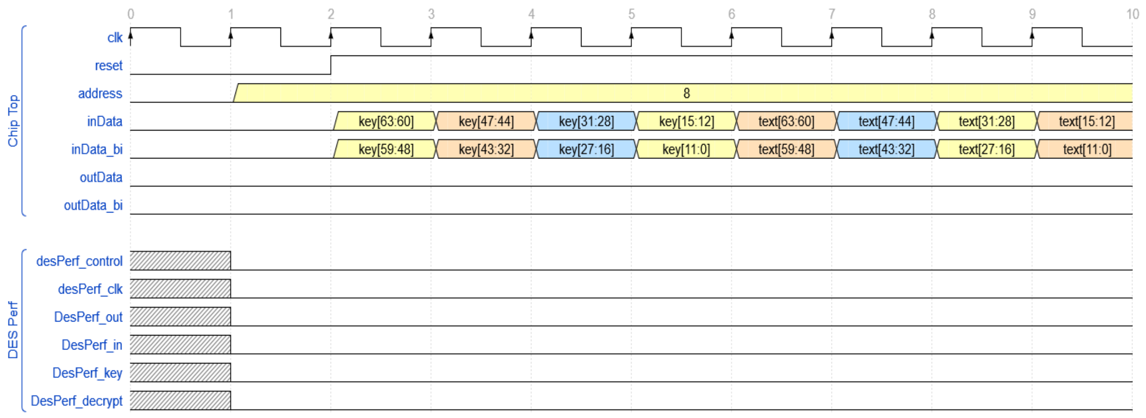

Figure 18: Loading Data for the DES Perf Benchmark

The DES Perf benchmark has a 56-bit key, therefore key[55:0] is the actual key. The key bit[56] is used to select if the DES Perf benchmark will encrypt or decrypt the text. If key[56] is low, then the benchmark is decrypting and if key[56] is high, then the benchmark is encrypting. The key bits key[63:57] are discarded; therefore, they can either be low or random data as it does not affect the operation of the GPIO. The GPIO is storing all this data into specific internal registers only for this benchmark. While this is occurring as shown in the figure above, all the signals for DES Perf benchmark are low. Once all the data has been loaded, the GPIO starts asserting different control signals for the DES Perf benchmark.

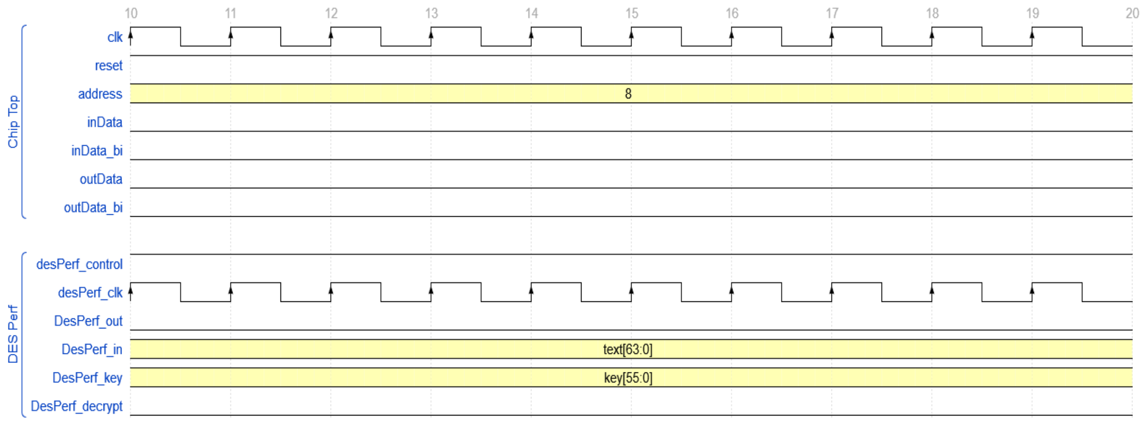

Figure 19: DES Perf Operation

The DES Perf benchmark does not have a reset signal, so once the data is loaded the GPIO sets the control signal for the DES Perf benchmark to high which enables the clock for this benchmark. On the 16th clock cycle, the encrypted data will be available and key and text input is applied throughout the entire 16 clock cycles.

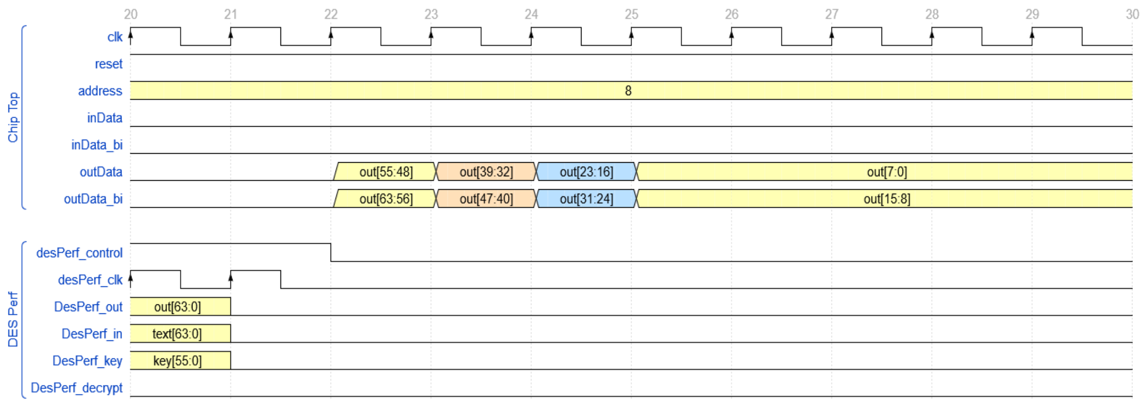

Figure 20: DES Perf Output Results

As shown in Figure 20, once the GPIO has the results from the DES Perf benchmark, all the signals for the DES Perf benchmark are set to low and the GPIO now feeds the results to outData and outData_bi ports, 16-bits per clock cycle. The higher 8 bits of the 16-bit packet are fed to outData_bi and the remaining 8 bits are fed to outData. The last packet of the data is held on both outData and outData_bi ports. The GPIO is now at an empty state and needs to be reset before any new input vector is applied to any of the benchmarks.

3.4.3.4 SHA3 (Keccak 512)

To begin, the address of the SHA3 benchmark must be applied (in this case, 32) and the reset signal (asynchronous active low) needs to be low for at least one clock cycle. After the GPIO is reset, every clock cycle after the data needed for the SHA3 benchmark needs to be fed in the inData (4 bits) and inData_bi (12 bits) ports. In total, 16 bits per clock cycle with the higher 4 bits to the inData port and the remaining bits to the inData_bi port. The SHA3 benchmark is capable of encrypting 576 bits of text, but it is limited to 128 bits to be consistent with the other benchmarks within the design. The loading sequence is shown below.

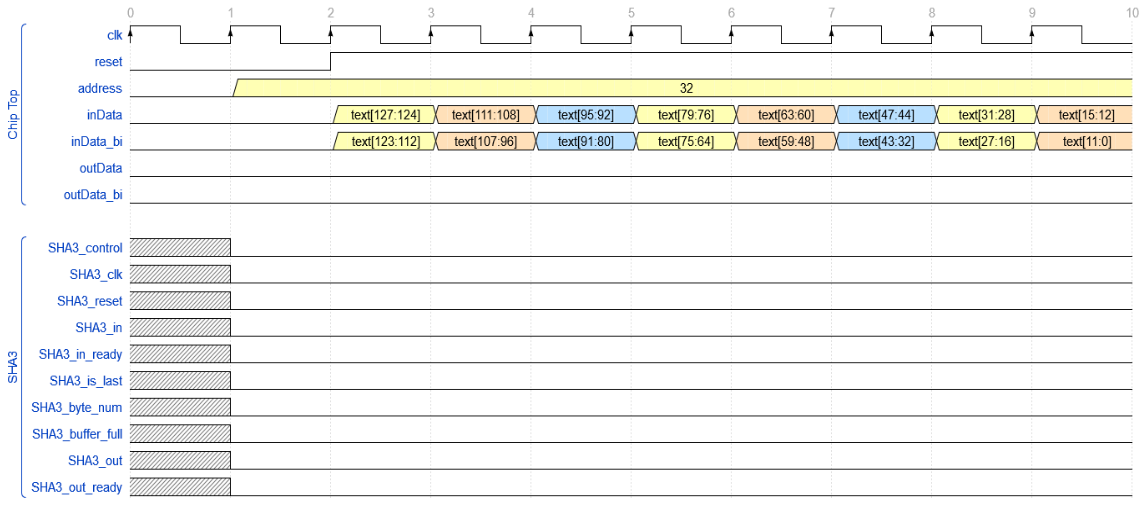

Figure 21: Loading Data for the SHA3 Benchmark

The GPIO is storing all this data into specific internal registers only for this benchmark. While this is occurring as shown in the figure above, all the signals for the SHA3 benchmark are low. Once all the data has been loaded into the GPIO, the GPIO sets the control signal of the SHA3 benchmark to high. This enables the clock for the SHA3 benchmark and since this benchmark has an active high reset, the GPIO resets the benchmark for at least two clock cycles.

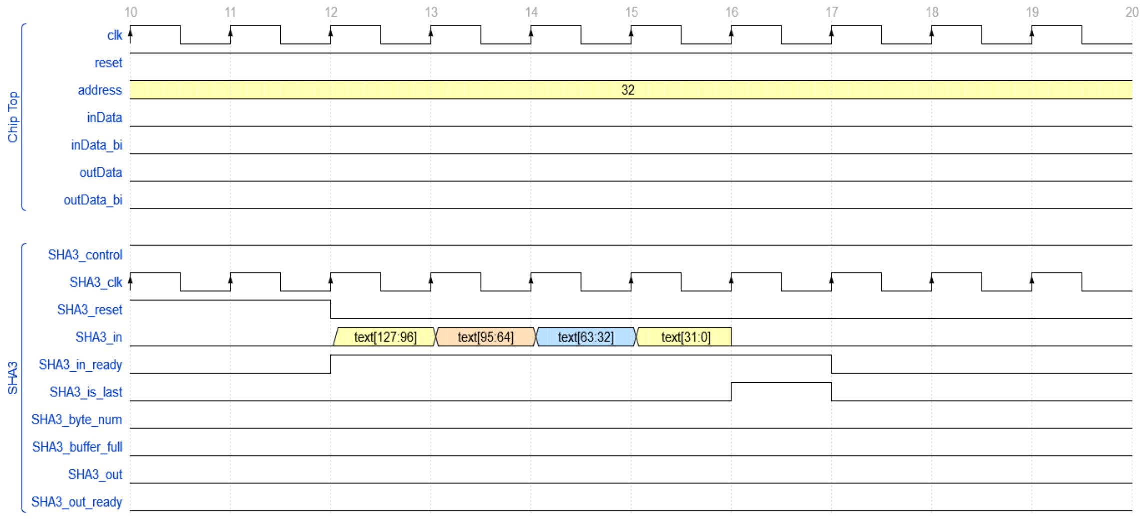

Figure 22: SHA3 Operation

Once the benchmark is reset, the in_ready signal is high for the entire time data is being loaded to the SHA3 benchmark. Data is serially loaded via the in port 32 bits per clock cycle. Once all 128 bits of data is loaded, a 32-bit packet of 0's are loaded and the is_last signal is set to high. After one clock cycle, is_last and in_ready are set to low. The signal byte_num is always grounded. The signal buffer_full never goes high because the input of the SHA3 is limited to 128 bits whereas this benchmark can handle upto 576 bits. If the input data exceeds 576 bits, then the buffer_full signal would be high. The GPIO now waits for the SHA3 encryption to be completed.

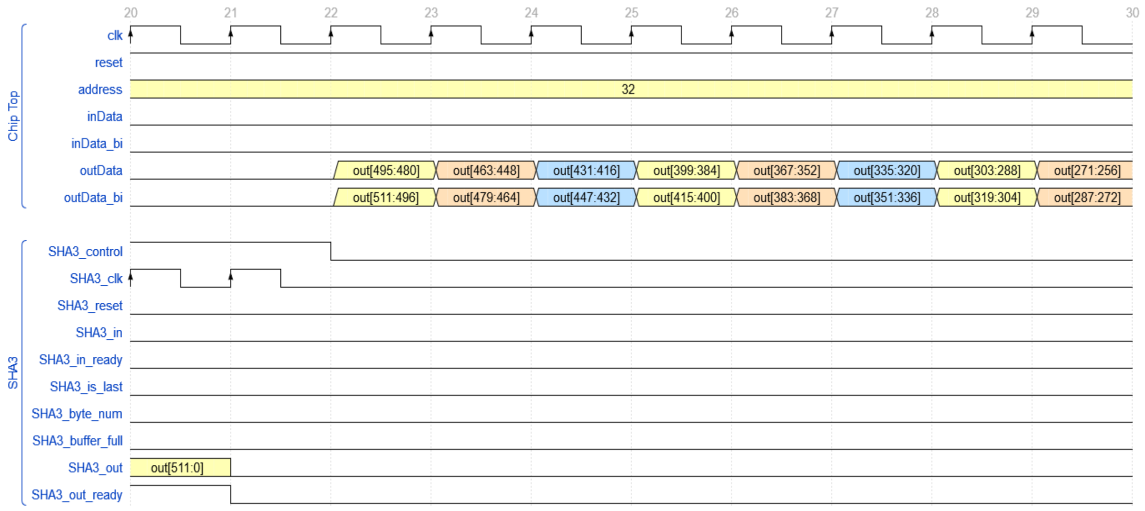

Figure 23: SHA3 Output Results

As shown in Figure 23, once the SHA3 encryption is completed, the out_ready signal is set high by the benchmark and the encrypted data is now available on the out port which is stored in an internal register by the GPIO. Once the GPIO has the results from the SHA3 benchmark, all the signals for the SHA3 benchmark are set to low and the GPIO now feeds the results to outData and outData_bi ports, 16 bits per clock cycle. The higher 8 bits of the 16-bit packet are fed to outData_bi and the remaining 8 bits are fed to outData. The last packet of data is held on both outData and outData_bi ports. The GPIO is now at an empty state and needs to be reset before any new input vector is applied to any of the benchmarks.

Note: SHA3 benchmark has a 512-bit output, the entire 512-bit sequence is not shown; however, it has a deterministic pattern where 16-bits of data per clock cycle is fed to the output signals.

3.4.4 Direct Connection to Top Level

Benchmarks with a low pin count that do not require any serial to parallel or parallel to serial conversion are directly connected to the top level.

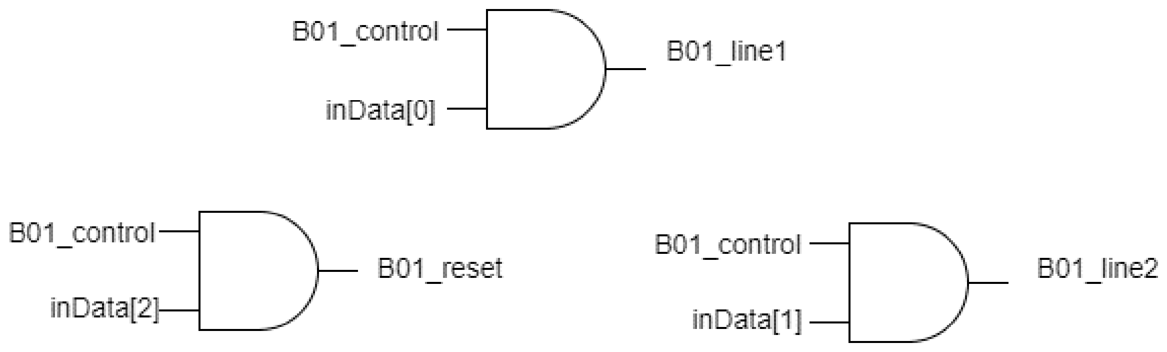

Figure 24: Combinational Logic for Directly Connected Benchmarks (Input Side)

This is done by having an enable signal (B01_control), when the enable signal is high the input signals of the benchmarks (B01_clock, B01_reset, B01_line1, B01_line2) follows their corresponding signal from top level (clk, inData[2], inData[1], inData[0]) as shown in the figure above. The enable signal is toggled by applying the address for a specific benchmark and only one benchmark is active at a time. If the enable signal is low, the input signals of the benchmark are all low.

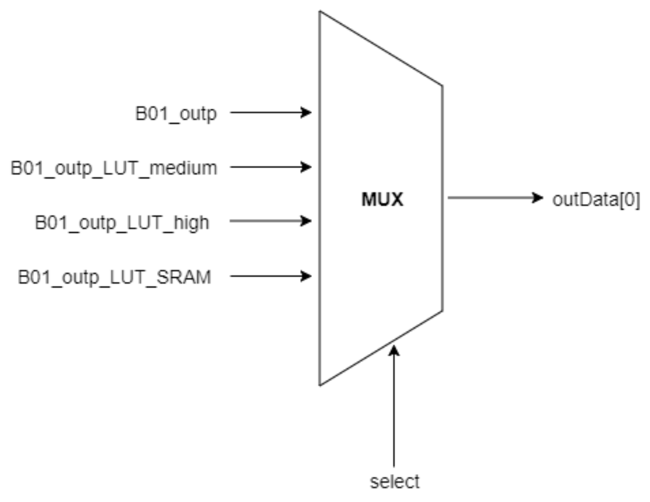

Figure 25: Combinational Logic for Directly Connected Benchmarks (Output Side)

The output signals of the directly connected benchmarks are connected through a multiplexer logic as shown above. Depending on the select signal, the correct benchmark output will be passed to the output port at the top level and only one benchmark is active at a time. The select signal is toggled by selecting the correct address for each benchmark.

Note: Figures 24 and 25 are simplified versions used only to explain the concept. In reality, they are far more complex.

The following sections document the port connections between the benchmarks and the top level. (All versions [base, LUT medium, LUT high, SRAM] of a benchmark have the same input and output connections to the top level)

3.4.4.1 B01

Benchmark Signal	Top Level Signal	Direction	Size (bits)
clock	clk	input	1
reset	inData[2]	input	1
line2	inData[1]	input	1
line1	inData[0]	input	1
outp	outData[0]	output	1
overflow	outData[1]	output	1

Table 4: B01 Port Connections

3.4.4.2 B02

Benchmark Signal	Top Level Signal	Direction	Size (bits)
clock	clk	input	1
reset	inData[1]	input	1
line	inData[0]	input	1
flag	outData[0]	output	1

Table 5: B02 Port Connections

3.4.4.3 B04

Benchmark Signal	Top Level Signal	Direction	Size (bits)
clock	clk	input	1
reset	inData[3]	input	1
restart	inData[2]	input	1
enable	inData[1]	input	1
average	inData[0]	input	1
dataIn	inData_bi[7:0]	input	8
outData	outData[7:0]	output	8

Table 6: B04 Port Connections

3.4.4.4 B12

Benchmark Signal	Top Level Signal	Direction	Size (bits)
clock	clk	input	1
reset	inData_bi[5]	input	1
start	inData_bi[4]	input	1
k	inData_bi[3:0]	input	4
nloss	outData[5]	output	1
speaker	outData[4]	output	1
nl	outData[3:0]	output	4

Table 7: B12 Port Connections

3.4.4.5 Custom ALU

Benchmark Signal	Top Level Signal	Direction	Size (bits)
mode	inData_bi[2:1]	input	2
c	outData_bi[7:0]	input	8
b	inData_bi[6:3]	input	4
a	inData[3:0]	input	4
carryIn	inData_bi[0]	input	1
carryOut	inData_bi[7]	output	1
z	outData[7:0]	output	8

Table 8: Custom ALU Port Connections

3.4.5 Inout Port for CPU

The CPU benchmark has direct connections for all it ports to top level as well as a special inout port that it uses to read and write to memory. This port is handled differently than regular input only or output only ports.

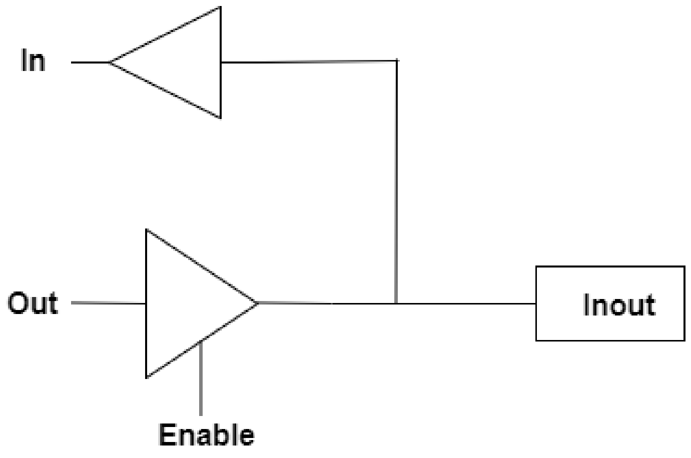

Figure 26: Basic Inout Port

The above figure shows an example of a basic inout port. When Enable is high, the data (Out) is passed to the Inout port (Output mode). If Enable is low, the tri-state buffer is in a high Z mode; therefore, an external device connected to the Inout port can drive an input signal through the same Inout port to the input port (In).

Having the necessary information presented above and recalling that the inout ports in the GPIO have been split into input and output only, the GPIO uses similar logic to configure outData_bi (inout port) either in input or output mode depending on the status of the CPU. The figure below illustrates how the inout port of the CPU is handled within the GPIO and how it is connected to the I/O cell. First, if the CPU benchmark is not selected outData_bi is either in input only or output only mode depending on which benchmark is selected. When the CPU benchmark is selected, outData_bi is connected to the inout port of the CPU (dataBus). The CPU benchmark has a signal named write, if high, the CPU is in write mode and if low, the CPU is in read mode. Using this information, if write signal is high, then the GPIO sets outData_bi in output mode. This is accomplished by setting the internal control signal (enable) to low, placing the internal buffer in high Z mode and the external control signal (outData_bi_control) for the I/O cell low (active low for write mode). Now the data from cpu dataBus is passed to the outData_bi_out signal and to the I/O cell. If write signal is low, then the GPIO set outData_bi in input mode. This is accomplished by setting the internal control signal (enable) to high and the external control signal (outData_bi_control) for the I/O cell high, placing the I/O cell buffer in high Z mode. Now the data from outData_bi_in is passed from the I/O cell to the CPU dataBus.

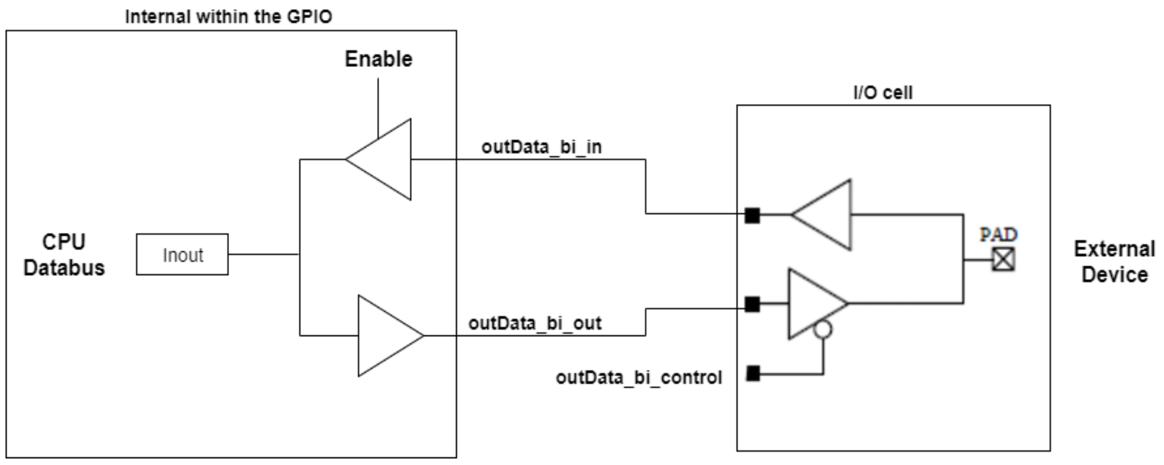

Figure 27: CPU Inout Port Connection to I/O Cell

3.4.5.1 CPU

The following table documents the port connections between the CPU and top level. (All versions [base, LUT medium, LUT high, SRAM] of a benchmark have the same input and output connections to the top level)

Benchmark Signal	**Top Level Signal**	**Direction**	**Size (bits)**
clock	clk	input	1
reset	inData[0]	input	1
writeBar	inData_bi[11]	output	1
write	inData_bi[10]	output	1
romEnable	inData_bi[9]	output	1
ramEnable	inData_bi[8]	output	1
addressBus[15:8]	inData_bi[7:0]	output	8
addressBus[7:0]	outData[7:0]	output	8
dataBus	outData_bi	Inout	8

Table 9: CPU Port Connections

4. RTL Verification

Throughout the entire design process, a direct test bench that has a couple of test cases for each benchmark is utilized to ensure the design functions as expected while the top-level RTL for the prototype chip is completed. This direct test bench is not complete and only serves a purpose of identifying major design issues. Once the RTL is completed, a thorough verification is required. This verification is completed by writing many SystemVerilog test benches for all the designs in the prototype chip. These test benches have self-checking abilities that compare the results of the benchmarks with the reference design programmed in either C or Verilog. This ensures that the benchmarks itself are verified since they are open source designs. Most importantly, it verifies the functionality of the GPIO and the entire top level.

5. Initial Synthesis

Once the RTL is verified and bug-free, it is ready for synthesis. Synthesis is chosen to be done in topographical mode.

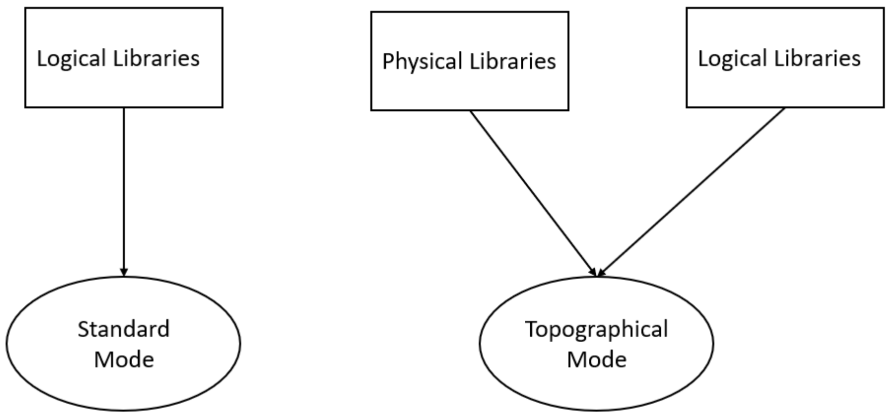

Figure 28: Synthesis: Topographical Mode vs Standard Mode

As shown in Figure 28, topographical mode requires both the logical and physical libraries compared to standard mode that only requires logical libraries. During synthesis and optimization, topographical mode does under the hood placement. Therefore, the distance between the nets of the connected cells are known. Since the distance is known and the synthesizer has physical library information, it calculates the actual net parasitics that are used for timing analysis rather than fan-out based analysis which is done in standard mode. This results in a better timing estimation and a more optimized design.

In topographical mode, synthesis is done in two steps for a more optimized design. First, synthesize the RTL and generate a gate level netlist. This gate level netlist is then used in physical design to generate floorplan information, such as macro placement, core size, utilization, IO cell placement etc. Next, synthesize the RTL again but with the floorplan information given to the synthesizer. Since the synthesizer does under the hood placement, the placement results are more refined now because floorplan information is available to the synthesizer. This results in a better optimized netlist.

Due to the short timeline and the relatively low speed for the prototype chip, the decision was made to do only one synthesis run. The following sections highlight the important aspects of the synthesis script, the entire synthesis script can be found in the appendix.

5.1 Synthesis Setup

First, all the logical files need to be loaded into memory. This includes all the different characterization corners available, which are used for multi-corner synthesis. This will be discussed later.

source $filename

The file, *$filename* is the file that contains the location of all the logical libraries (.db files) and is loaded into memory by using the *source* command. Next, the RTL files need to be loaded into memory and parsed for any syntax errors, if any, using the following commands.

analyze -library WORK -format verilog {$RTL file names}

elaborate -architecture verilog -library WORK $top module name

The *analyze* command parses all the RTL files. The *library* option specifies the working directory (*WORK*), *format* option specifies the language of the RTL (*verilog*) and all the RTL files are specified within the curly brackets (*{$RTL file names}*). The *elaborate* command links the RTL files to the general technology library designs. The *architecture* option specifies the structure of the RTL which is essentially the language the RTL is written in (*verilog*), *library* option specifies the working directory (*WORK*) and *$top module name* is the name of the top module in the entire design. Finally, the physical library needs to be imported into memory. This is accomplished by using the *create_mw_lib* command.

create_mw_lib -technology $techfile -mw_reference_library {$reference libraries} $design name

The *technology* option specifies the location of the technology file (*$techfile*), *mw_reference_library* option specifies the location of all the Milkyway libraries (*{reference libraries}*), and *$design name* is the name of the design chosen by the designer.

5.2 Multi-Corner Setup

ASIC designs need to function in many environments because upon startup, when all the internal components are cold, the design will function slightly slower. When the internal components of the design heat up, the design will function slightly faster. During the manufacturing process, process variation will cause some transistors in the design to function faster than expected or slower than expected. To combat this, the design needs to be optimized for all these situations. Luckily, manufactures provide logical libraries of standard cells that are characterized in many operating corners such as worst case, best case, low temperature, and others. To optimize the design in multiple operating corners requires a multi-corner setup using the following commands.

create_scenario $scenario name

set_operating_conditions -analysis_type bc_wc -max $max condtion name -max_library $max library name -min $min condition name -min_library $min library name

set_operating_conditions -analysis_type bc_wc -max $max condtion name -max_library $max library name -min $min condition name -min_library $min library name -cells $specific cells

set_tlu_plus_files -max_tluplus $max file location -min_tluplus $min file location -tech2itf_map $map file location

Since multiple scenarios can be created each scenario is given a specific name by using the *create_scenario* command followed by the scenario name (*$scenario name*). Then the operating condition of this scenario is set using the *set_operating_conditions* command. The *analysis_type* option followed by *bc_wc* specifies to use the min library for best case analysis and use the max library for worst case analysis. The *max* option followed by *$max condition name* alerts the synthesizer to use the operating condition named *$max condition name* for worst case analysis. The *max_library* option followed by *$max library name* specifies which library the max operating condition is located in. A similar setup is required for the best case analysis using the min library options. The *set_operating_conditions* command by default sets the operating conditions of all the cells in the design to what is specified in the command options; however, if designs contain cells that are not characterized with the same operating conditions as specified in the max and min libraries, warnings are thrown. For example, in our case, the netlist contained IO cells which have an input voltage of 2.5 V, whereas everything else in the design has an input voltage of 1 V. This mismatch will cause warnings. To rectify this, the *set_operating_conditions* can be used once more with the *cells* option along with *$specific cells* to notify the synthesizer to use different max and min libraries for these specific cells. These max and min libraries will contain characterization corners where the input voltage is 2.5 V rather than 1 V.

Multi-corner setup requires parasitic information for all the different scenarios. This is accomplished by using the *set_tlu_plus_files*, where the *max_tluplus* option is followed by the file location (*$max file location*) and the *min_tluplus* option is followed by the file location (*$min file location*) alerts the synthesizer to use the parasitic information in these files for best case and worst case analysis in this scenario. This command requires a map file that is provided by the *tech2itf_map* option followed by file location (*$map file location*).

Finally, to finish up the scenario setup, the timing constraints need to be applied for each scenario. This will be discussed in the next section. For our prototype chip, nine scenarios were created. The following table summarizes the scenario setup information.

Note: To keep multi-corner setup as simple as possible, the parasitic information used in all the scenarios were the same, hence the reason it is excluded from the following table.

Scenario Name	**Min Library**	**Max Library**
s1	Best Case (BC)	Worst Case (WC)
s2	Maximum Leakage (ML)	Worst Case (WC)
s3	Low Temperature (LT)	Worst Case (WC)
s4	Maximum Leakage (ML)	Worst Case at Low Temperature (WCL)
s5	Best Case (BC)	Worst Case at Low Temperature (WCL)
s6	Low Temperature (LT)	Worst Case at Low Temperature (WCL)
s7	Maximum Leakage (ML)	Worst Case at Zero Temperature (WCZ)
s8	Low Temperature (LT)	Worst Case at Zero Temperature (WCZ)
s9	Best Case (BC)	Worst Case at Zero Temperature (WCZ)

Table 10: Scenario Information

5.3 Timing Constraints

Each scenario requires its own timing constraints to have a multi-corner multi-mode (MCMM) option, but since our goal is only for multi-corner the timing constraints in all the scenarios are the same. To create the clock constraints the following command is used.

create_clock $clock source -period $period in nS -name $clock name

To create the clock, *create_clock* command is used followed by the source of the clock (*$clock source*). The *period* option specifies the period of the clock (*$period in nS*) in the default time units specified in the technology file. Finally, clock can be given a name which it can be referenced by throughout the synthesis script, by using the *name* option followed with the name of the clock (*$clock name*). The following table summarizes all the clocks in the prototype chip.

Clock	**Period (nS)**
clk	10
SCLK	100
FCLK	10,000

Table 11: Clock Information

Once the clocks are defined, the input/output delay of all the top-level ports need to be defined. The is demonstrated by the following command.

set_input_delay/set_output_delay $delay $pin name -clock $clock name

The delay is set using the *set_input_delay* for input pins and *set_output_delay* for output pins. The command is followed by the actual delay (*$delay*) and to which pin this delay applies to (*$pin name*). Finally, this command requires the reference clock of the delay which is set by using the *clock* option followed by the clock name (*$clock name*). Each input and output pin need this delay information for their respective clocks. In our case, the delay was three nanoseconds, which was the added delay from the IO cells. Please see the synthesis script in the Appendix to see all the input/output delays of each pin with respect to their clocks.

Since the GPIO has internal clock gating cells, the clocks beyond these cells need to be defined differently. Once the clock is an input to a logic cell (clock gating cell), it becomes an explicit stop pin and the timing analysis for the clock signal ends there. As shown in Figure 29, only U1, U2, and UCG1 are considered in the timing analysis. However, for correct timing analysis, we need the clock signal after the clock gating cell to be considered in the timing analysis. Therefore, U3 and U4 also need to be considered in the timing analysis. For this to happen, a special *create_generated_clock* command needs to be used.

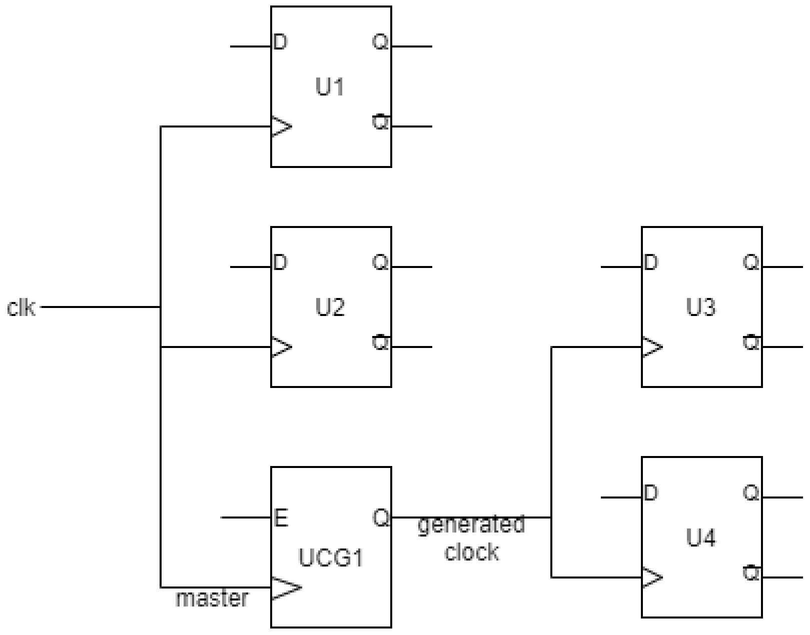

Figure 29: Generated Clocks

create_generated_clock -source $source pin $output pin -combinational

The *source* option of the *create_generated_clock* command specifies the input pin of a generated clock (*$source pin*) followed by the output pin (*$output pin*). The *combinational* option specifies that a combinational cell generates the clock, if a flip-flop is generating the clock then the *divide_by* option is used rather than the *combinational* option.

Some additional timing constraints were also added to the list of constraints.

set_max_transition $transition time $clocks -clock_path

The *set_max_transition* followed by a transition time (*$transition time*) sets the maximum allowable transition time for all the specified clock signals (*$clocks*). The *clock_path* option alerts the synthesizer to set the maximum transition on only the clock signals rather than the data signals.

set_clock_uncertainty $uncertainty time $clocks

The *set_clock_uncertainty* command sets the uncertainty of the specified clocks (*$clocks*) by a certain time period (*$uncertainty time*).

set_clock_latency $latency time $clocks

The *set_clock_latency* command sets the latency of the specified clocks (*$clocks*) by a certain time period (*$latency time*).

set_max_fanout $fanout number $input ports

The *set_max_fanout* command sets the maximum fan-out (*$fanout number*) of all the specified input ports (*$input ports*).

5.4 Additional Synthesis Notes

Once all the timing constraints are set within each scenario, the design can be synthesized by using the *compile_ultra* command. To accommodate the gate selection and replacement scripts, all the individual benchmarks need to be flattened. This was accomplished by selecting the design using the command *current_design $design name* and then running the *ungroup -all* command. The gate selection script is not capable of handling complex gates, therefore, the *set_dont_use* command was placed on the complex gates so the synthesizer would not use them. Once the synthesis is completed, various reports can be generated along with the Verilog netlist for the entire design or individual designs. The gate replacement script is not capable of handling some special characters such as “[]” or multi-dimensional registers. The *change_names* command is used to alert the synthesizer not to use those special characters when the Verilog netlist is generated.

Finally, the results of synthesis will have no setup violations, but it will have numerous hold violations. The synthesizer has limited options to fix hold violations; therefore, all hold violations will be addressed during Clock Tree Synthesis (CTS) in physical design.

Please refer to the appendix to see the synthesis script and the exact usage for all the commands described above.

6. Gate Level Verification

Once the gate level netlist is generated from synthesis, it is once again verified using the same SystemVerilog test benches used in RTL verification. Since the top-level ports do not change, verification can be completed by changing the RTL to the gate level netlist and re-running the simulation. Even though the synthesizer is a well-trusted algorithm capable of synthesizing correctly, verification of the gate level netlist is completed to ensure that synthesis was successful.

7. Gate Selection and Replacement

The gate level netlist for individual benchmarks is now analyzed to determine which gates of the benchmark will be replaced with LUTs. The number of LUTs per benchmark is predetermined by the number of configuration bits allowed for the level of obfuscation. Once the gates are chosen to be replaced, the netlist is analyzed to determine if it can be reverse engineered within five days. If the netlist cannot be reverse engineered within five days, then gate replacement follows.

Knowing which gates need to be replaced in each design, the gate replacement script replaces the specific gates within a netlist and reroutes the netlist if needed. It also generates the configuration bits that is needed to be programmed in the LUTs. These configuration bits enable the LUTs to function correctly as the gates they are replacing. The gate replacement script does initial verification to make sure the configuration bits are generated correctly.

7.1 Top Level Modification for Obfuscated Benchmarks

Previously, the obfuscated netlists were not ready, all four versions of the benchmarks in RTL were the same. They were only included so that the RTL verification would ensure that the top-level was wired correctly and the GPIO functions as expected. Now after gate selection and replacement, the LUT medium, the LUT high, and the SRAM versions of the benchmark are available. The top-level netlist is now modified to replace all the placeholder benchmarks with the actual obfuscated benchmarks. The fuse macro module is now inserted into the netlist along with a configuration bit converter module.

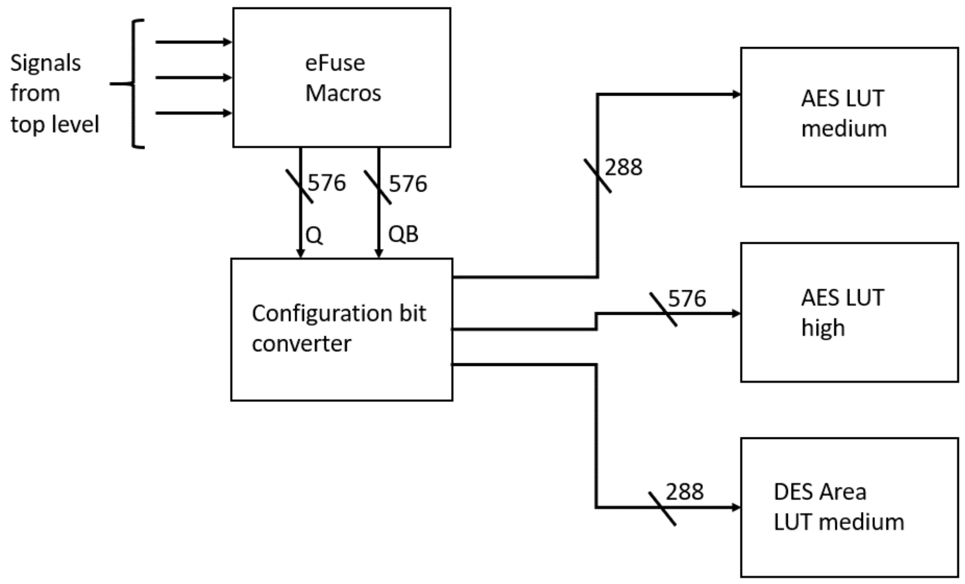

Figure 30: Configuration Bit Converter

As shown in Figure 30, the configuration bit converter takes the entire stream of Q and QB from the eFuse macros and then converts the stream of Q and QB into the specific configuration bits for each design. Q and QB from the eFuse macros are programmed to be the configuration bits for SHA3 LUT high, so using the configuration bit converter the configuration bits of the other benchmarks are derived from this, which is internally handled within the configuration bit converter.

The SRAM version of the obfuscated benchmarks do not require extra modules. They need to be connected in a scan chain format as shown in the Figure 31 below.

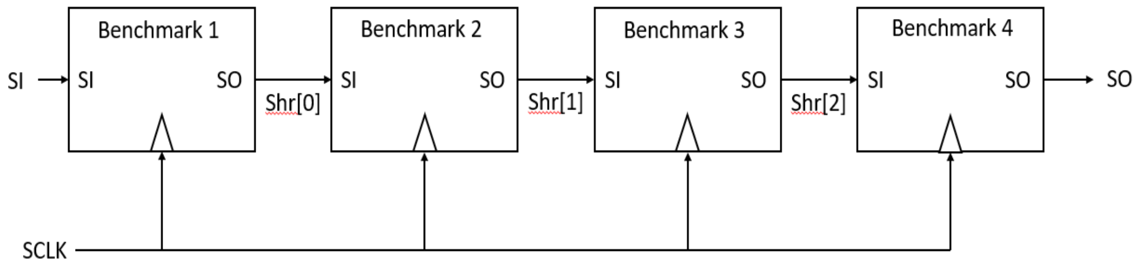

Figure 31: Scan Chain Connection of SRAM Benchmarks

The table below shows the order of the scan chain.

Benchmark	**SI**	**SO**
ALU	SI	shr[0]
B01	shr[0]	shr[1]
B02	shr[1]	shr[2]
B04	shr[2]	shr[3]
B12	shr[3]	shr[4]
CPU	shr[4]	shr[5]
DES Performance	shr[5]	shr[6]
DES Area	shr[6]	shr[7]
AES	shr[7]	shr[8]
SHA3	shr[8]	SO

Table 12: Scan Chain Connection for SRAM Version of Benchmarks

Once all the obfuscated benchmarks are properly integrated into the top level, the entire top level is verified using the direct test bench.

8. Re-Synthesis with Design for Test (DFT)

The entire netlist now goes through one final synthesis run. The only difference between the initial synthesis and final synthesis is now, synthesis does not need to accommodate the gate selection and replacement scripts. The restriction on using only simple gates is removed so the synthesizer can use all the gates available in the library. Finally, Design for Test (DFT) is included in the final synthesis script.

8.1 Synthesis Results

Benchmark	**Total Area Benchmark**	**Total Power (mW)**	**Power Overhead (%)**	**Area Overhead (%)**
SHA3	48202.20	1.74	0	0
SHA3 LUT medium	95828.53	1.57	-10	98.80530515
SHA3 LUT high	143468.18	1.59	-9	197.6382438
SHA3 SRAM	78406.56	1.93	11	62.66178974
AES	23421.96	0.70	0	0
AES LUT medium	71260.45	1.02	45	204.2463096
AES LUT high	118931.90	1.03	46	407.7794438
AES SRAM	53878.00	1.37	94	130.0319836
DES_Area	3882.24	0.65	0	0
DES_Area LUT medium	51523.69	0.21	-69	1227.163926
DES_Area LUT high	99138.86	0.22	-66	2453.650923
DES_Area SRAM	34162.20	0.60	-8	779.9610511
DES_Perf	39597.48	2.31	0	0
DES_Perf LUT medium	87201.49	2.69	17	120.2197945
DES_Perf LUT high	134801.90	2.71	18	240.4304974
DES_Perf SRAM	69690.96	2.99	30	75.99847284

Table 13: Synthesis Results

Benchmark	Total Area Benchmark	Total Power (mW)	Power Overhead (%)	Area Overhead (%)
CPU	3801.96	0.09	0	0
CPU LUT medium	51441.25	0.12	32	1253.019214
CPU LUT high	99067.22	0.12	40	2505.688085
CPU SRAM	33902.28	0.47	425	791.7053379
B12	7344.36	0.25	0	0
B12 LUT medium	54785.29	0.38	52	645.950487
B12 LUT high	102442.22	0.39	56	1294.842006
B12 SRAM	37341.36	0.73	192	408.4358633
B04	1158.48	0.06	0	0
B04 LUT medium	48771.13	0.07	21	4109.924124
B04 LUT high	96410.06	0.09	44	8222.116737
B04 SRAM	31303.80	0.43	590	2602.144192
B02	79.92	0.00	0	0
B02 LUT medium	47038.81	0.01	71	58757.3684
B02 LUT high	93911.66	0.01	80	117407.0791
B02 SRAM	2531.16	0.04	960	3067.117109
B01	90.72	0.01	0	0
B01 LUT medium	47003.53	0.01	14	51711.65009
B01 LUT high	93882.14	0.01	17	103385.6018
B01 SRAM	835.56	0.02	197	821.0317456
ALU	288.72	0.03	0	0
ALU LUT medium	47309.17	0.05	68	16285.83023
ALU LUT high	94197.14	0.07	124	32525.77516
ALU SRAM	1012.32	0.08	141	250.6234435

Table 14: Synthesis Results Continued

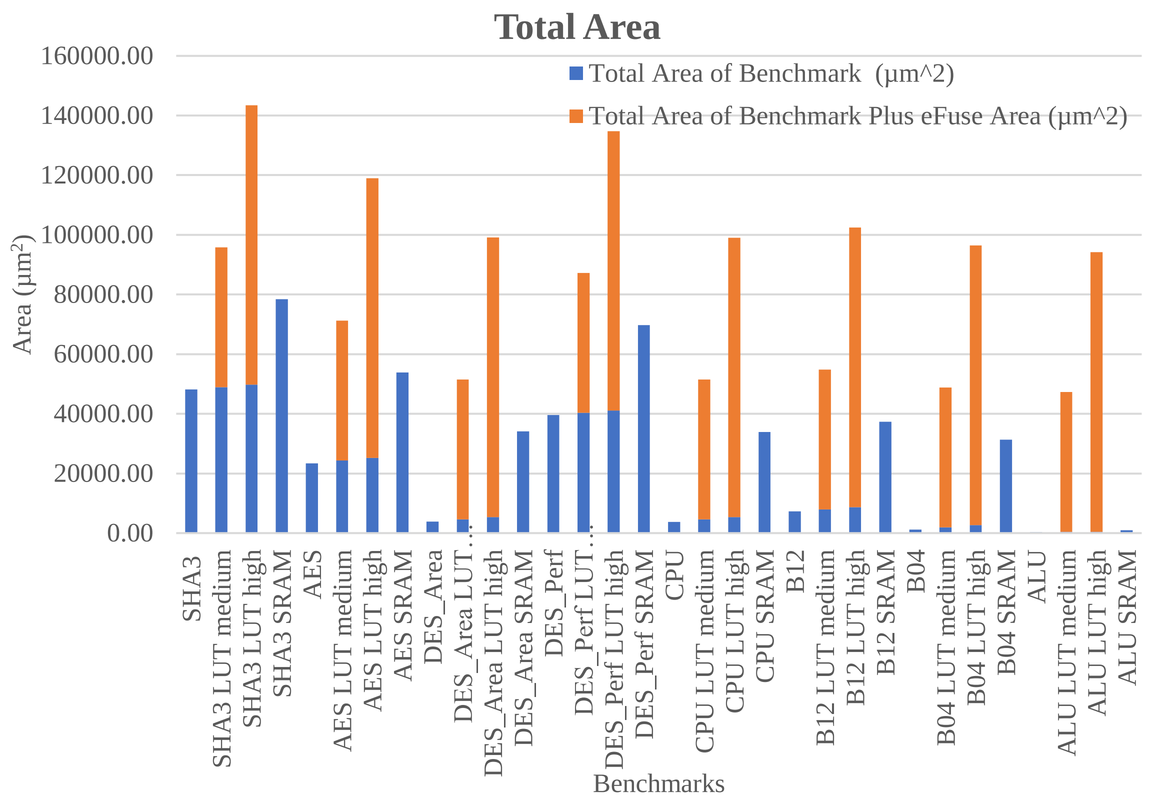

Figure 32: Total Area of Benchmarks

As shown in Figure 32, the medium and high obfuscated benchmarks do not show significant area increase, whereas the SRAM obfuscated benchmarks show a large area increase. This is reasonable since 3200 registers are added to these benchmarks, but when the eFuse macro area is added for the medium and high obfuscated benchmarks then the area of those benchmarks increases greatly. This is due to the large area required by the eFuse macro. One eFuse macro has an area of 5200 µm^2. The medium obfuscated benchmarks require nine eFuse macros while the high obfuscated benchmarks require 18 eFuse macros; therefore, the area adds up drastically.

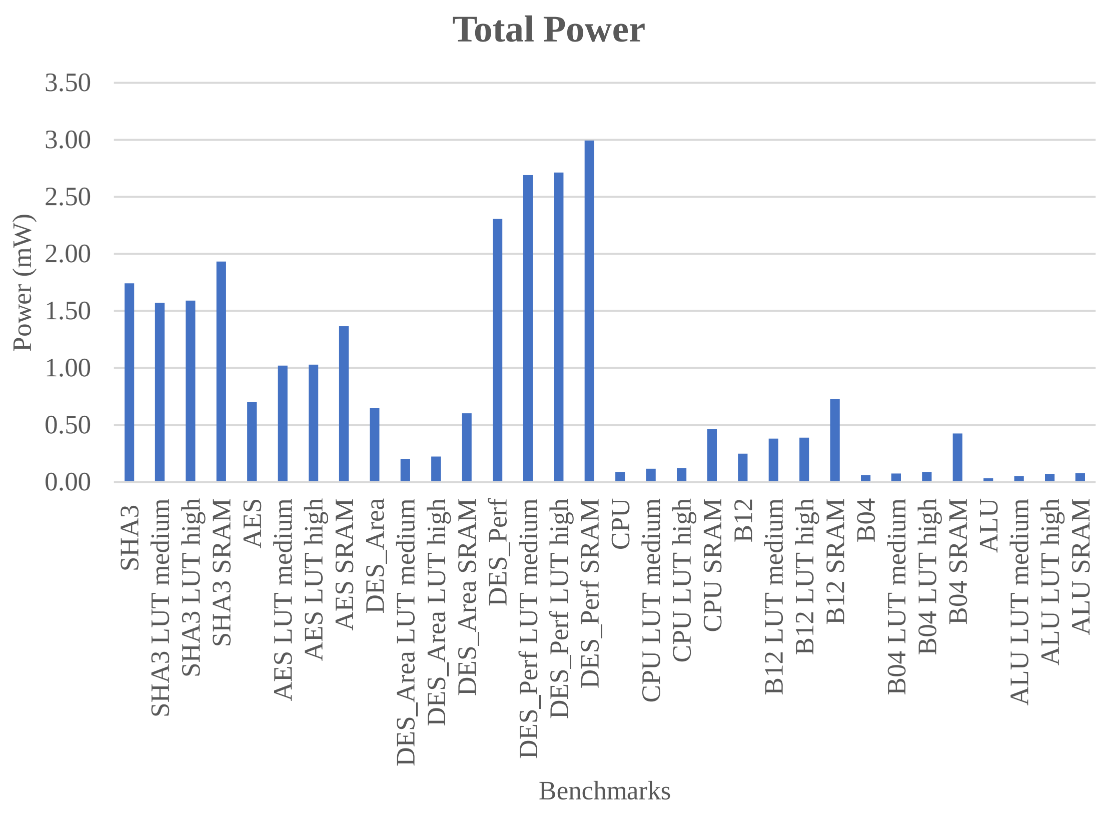

Figure 33: Total Power of Benchmarks

As shown in Figure 33, the power results correlate with the area results. The medium and high obfuscated benchmarks do not show significant power increase, but the SRAM obfuscated benchmarks show a large power increase. This is anticipated because in the SRAM obfuscated benchmarks 3200 registers have been added and registers consume more power compared to combinational cells due to constant switching activity.

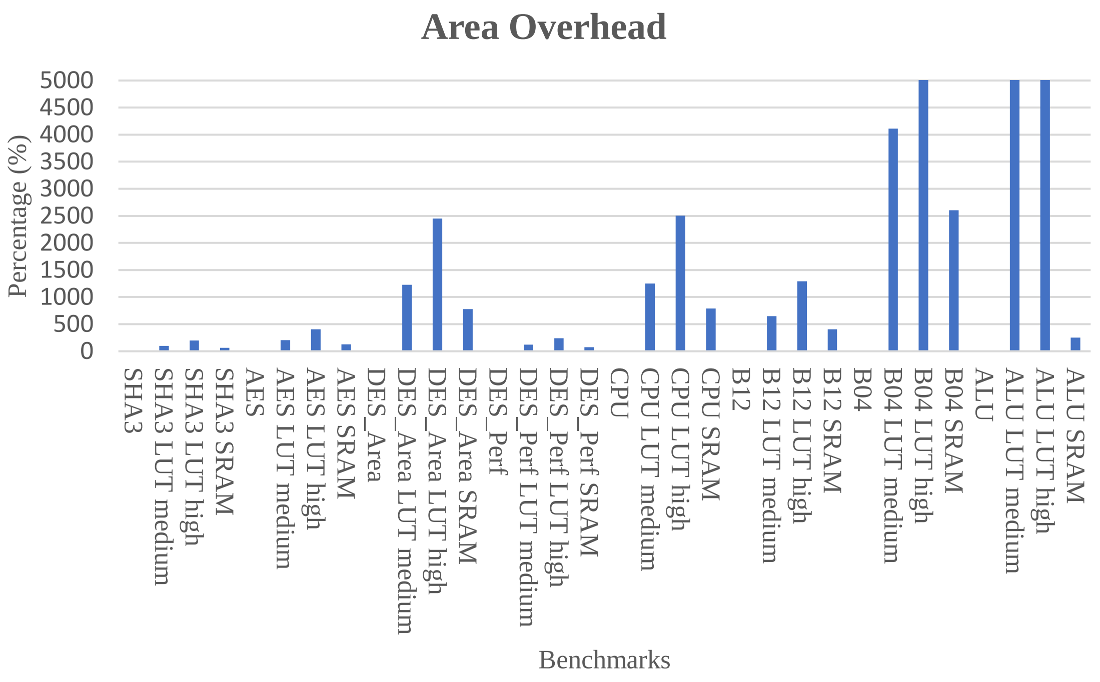

Figure 34: Area Overhead

As shown in Figure 34, when considering the area of the eFuse macros, there is an extremely large area overhead. All the benchmarks on the right of the graph are extremely small benchmarks; therefore, it is reasonable to see a large area overhead. The more complex benchmarks which are on the left of the graph show more favorable results as the area overhead is not as drastic. By analyzing the graph, it is reasonable to predict that as the design gets more complex the area overhead will be minimal even with the large eFuse macros. If single bit NV latches are used to build LUTs rather than large 32-bit eFuse macros, the area overhead can be further reduced.

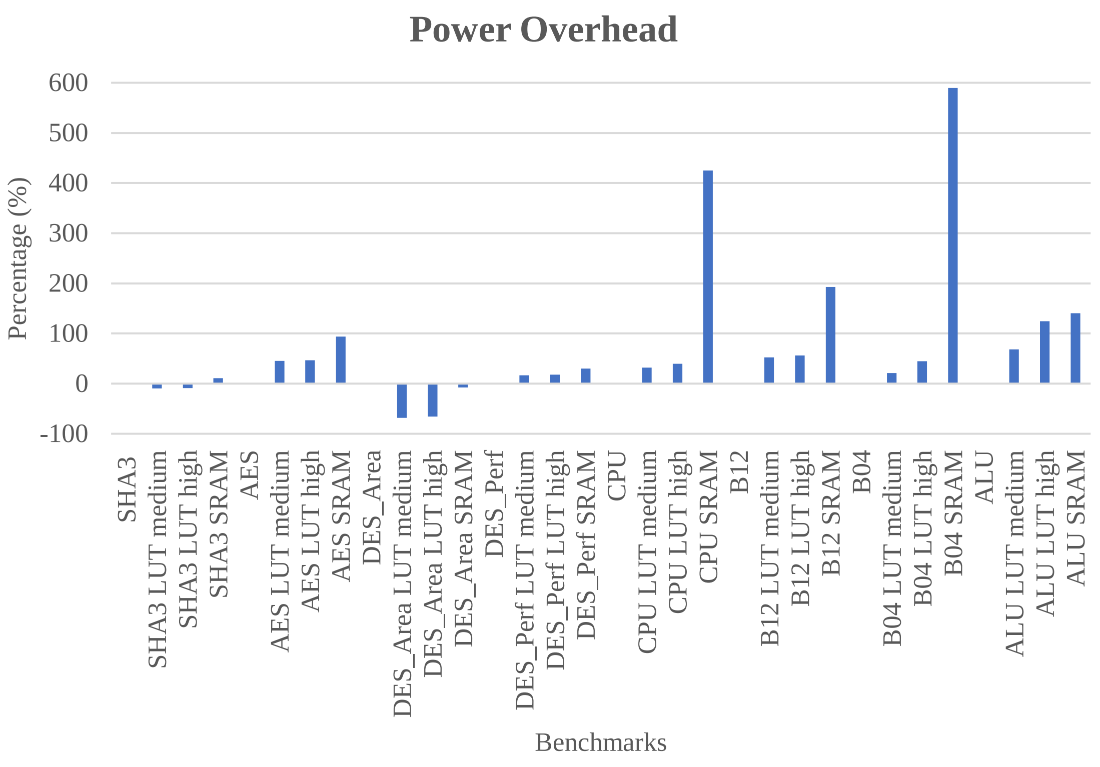

Figure 35: Power Overhead

As shown in Figure 35, the power overhead correlates with the area overhead. The smaller and less complex benchmarks show a large power overhead, but as the benchmarks get bigger and more complex the power overhead is reasonable. In all cases, the SRAM obfuscated benchmarks show the largest power overhead. This is expected as 3200 registers have been added to those designs which contribute heavily towards the large power overhead.

Benchmark	Average Area Overhead (%)	Area Overhead Minimum (%)	Area Overhead Maximum (%)
SHA3	119.7017796	62.66178974	197.6382438
AES	247.352579	130.0319836	407.7794438
DES_Area	1486.9253	779.9610511	2453.650923
DES_Perf	145.5495882	75.99847284	240.4304974
CPU	1516.804212	791.7053379	2505.688085
B12	783.0761188	408.4358633	1294.842006
B04	4978.061685	2602.144192	8222.116737
B02	59743.85489	3067.117109	117407.0791
B01	51972.76121	821.0317456	103385.6018
ALU	16354.07628	250.6234435	32525.77516

Table 15: Average, Minimum, and Maximum Area Overhead

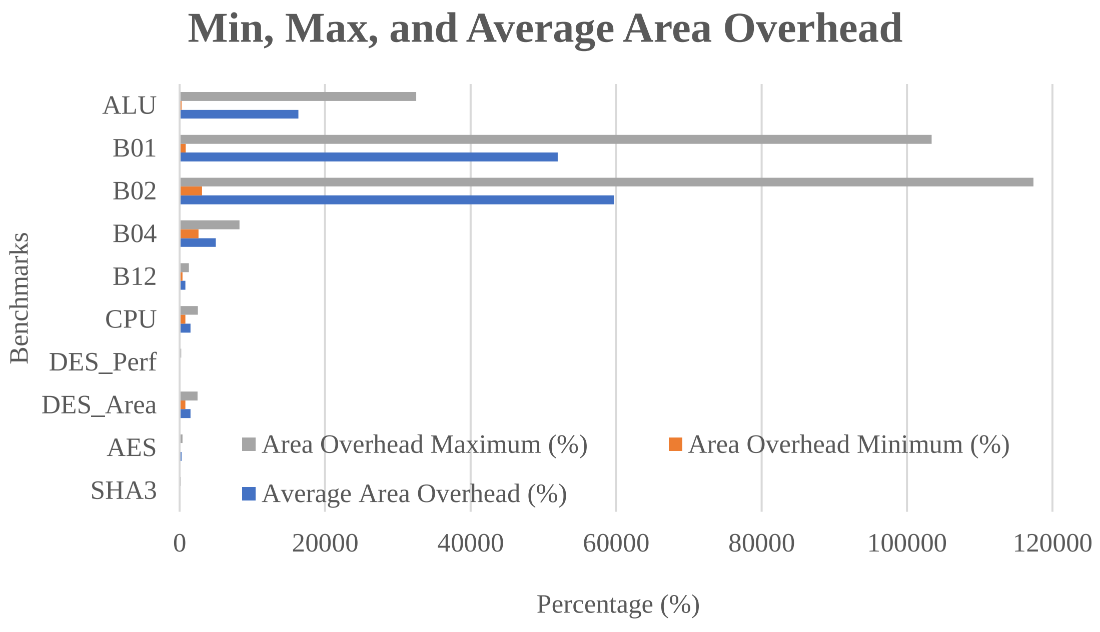

Figure 36: Average, Minimum, and Maximum Area Overhead Graph

Table 15 shows the average, minimum, and maximum area overhead for all three versions of the obfuscated benchmarks. Figure 36 displays the information in Table 15 in a graphical form. The average area overhead for all the benchmarks are large. As expected for all the benchmarks, the minimum area overhead came from the SRAM obfuscated versions whereas the maximum area overhead came from the high obfuscated versions of the benchmarks.

Benchmark	**Average Power Overhead (%)**	**Power Overhead Minimum (%)**	**Power Overhead Maximum (%)**
SHA3	-3	-10	11
AES	62	45	94
DES_Area	-47	-69	-8
DES_Perf	21	17	30
CPU	165	32	425
B12	100	52	192
B04	932	21	590
B02	370	71	960
B01	76	14	197
ALU	111	68	141

Table 16: Average, Minimum, and Maximum Power Overhead

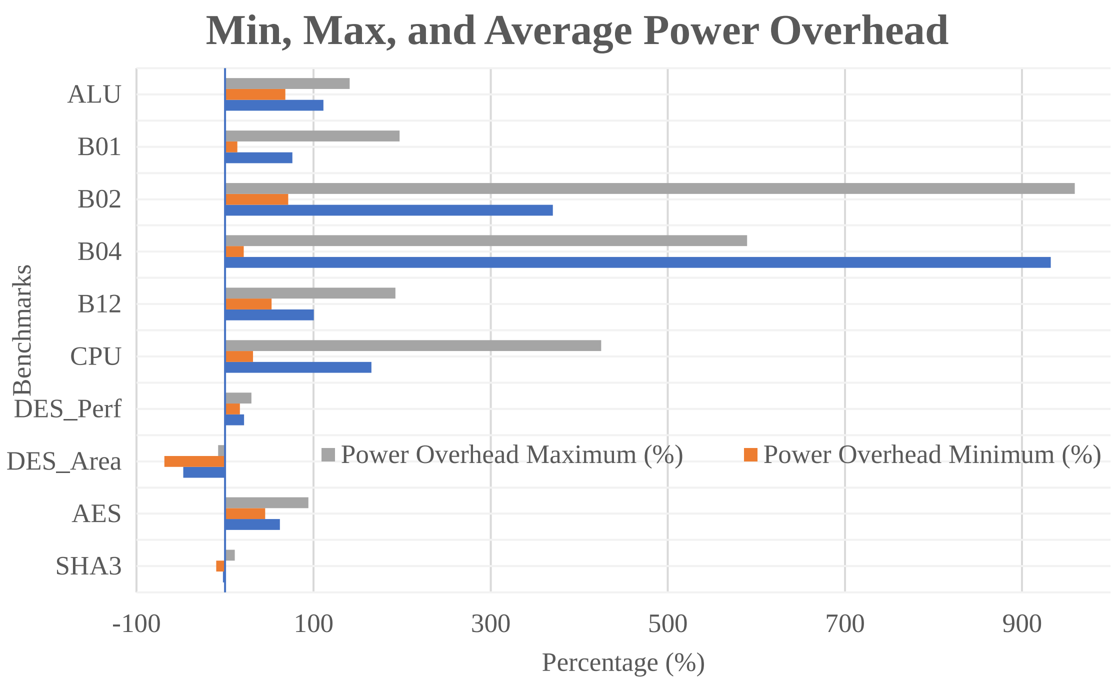

Figure 37: Average, Minimum, and Maximum Power Overhead Graph

Table 16 shows the average, minimum, and maximum power overhead for all three versions of the obfuscated benchmarks. Figure 37 displays the information in Table 16 in a graphical form. The power overhead results correlate with the area overhead results. The minimum power overhead came from the medium obfuscated versions of the benchmarks and the maximum power overhead came from the SRAM obfuscated versions of the benchmarks. Interestingly, the DES Area benchmark required less power after obfuscation.

The results from benchmarks B01 and B02 are disregarded because regardless of the number of gates replaced; they were not successfully obfuscated since these benchmarks are extremely simple.

9. Post Obfuscation Gate Level Verification

Once the final gate level netlist is ready, it needs to be verified. This verification step is extremely important because three major changes have been made to the final gate level netlist compared to the initial gate level netlist. Firstly, the entire top level has been substantially redesigned because the placeholder benchmarks have been removed and the actual obfuscated benchmarks have been inserted. This required some modification to the top-level module. Also, a new eFuse macro module along with a configuration bit converter module was added and wired to the LUT based obfuscated benchmarks. These connections and the overall functionality of the newly added modules need to be verified. The SRAM designs connected in a scan chain format also need their connections to be verified.

Secondly, the gate replacement script is a completely new concept and idea that is introduced in ASIC flow by our research team and anything that is a result of this script needs extensive verification. This is the first trial of the script, so it is relatively untested. When replacing the gates, the gate replacement script has to reroute some of the signals and need to generate the configurations bits that will be programmed to the LUTs for the LUTs to function as the logic gates they are replacing. Extensive verification is required so that the obfuscated netlist is shown to function as expected along with verifying if the configuration bits are generated correctly.

Finally, DFT has been introduced for the first time in this netlist. Although DFT does not change the netlist drastically, it does replace most of the flip-flops in the design with special scan flops. And then connects all these scan flops in a scan chain format, so after manufacturing test patterns can be loaded via the scan chain to test if there are any manufacturing defects within the design.

The verification of the final gate level netlist is crucial because of the three major changed to the netlist described above. Verification is once again completed using the same SystemVerilog test benches that have been used in the RTL verification. The test benches are now slightly modified to include two programming tasks. One programming task programs the eFuse macros to store the correct key value for the LUT based obfuscated benchmarks and the second programming task programs all the flops in the SRAM based benchmarks to store the correct configuration bits. After the programming tasks are completed, verification tasks begin.

10. Physical Design

After post obfuscation gate level verification, the synthesized netlist is now ready for physical design. The four major steps of physical design are floor planning, placement, clock tree synthesis (CTS), and routing. In floor planning, the core size is determined and IO cell placement as well as, the order is finalized. During placement, routing and placement blockages are defined, the power structure of the entire chip is created, and all cells are placed within the core area. During CTS, clock tree synthesis is completed while hold violations are finally addressed and fixed. After CTS, all the signal nets within the design are routed.

10.1 Floor Planning

Before the core area is defined and created, physical only cells need to be created. Physical only cells are power IO cells that provide power and ground connections to the core as well as supply a different high voltage power the IO cells themselves. Physical only IO cells include Electrostatic Discharge (ESD) protection and filler cells. The following command is used to create physical only cells.

create_cell $cell name $instance name

The *create_cell* command requires the name of the cell ($cell name) and the instance name ($instance name) given to the cell once it is created. Once all the physical cells are created, the order of the IO cells needs to be defined and on which side of the chip each IO cell will be placed. This is accomplished by using the following command.

set_pad_physical_constraints -pad_name $pad instance name -side $side -order $order

The *pad_name* option followed by the instance name of the IO cell (*$pad instance name*) for the *set_pad_physical_constraints* command defines which IO cell this constraint will be applied to. The *side* option specifies on which side of the chip the IO cell will be placed and the *order* option specifies in which order the IO cell will be placed on that side of the chip.

create_floorplan

To create the core area, the *create_floorplan* command is used. With this command, there are many options that are not shown above but are available to use. The length and width of the chip can be explicitly defined by using the appropriate options or the size of the chip can be automatically determined by the tool from specifying the utilization ratio among others. The *create_floorplan* command creates the core area and places the IO cells according to the order defined by the *set_pad_physical_constraints* command. Once the core area is created, macro locations are defined next, using the following command.

set_fp_macro_options $macro names -x_offset $X -y_offset $Y

The *set_fp_macro_options* command requires the instance names of the macros (*$macro names*) followed by the *x_offset* and *y_offset* options that define the X (*$X*) and Y (*$Y*) values of these macros.

create_placement_blockage

create_routing_blockage

The *create_placement_blockage* and *create_routing_blockage* commands are used to create placement and routing blockages. The placement blockages restrict the placement of standard cells within a specified area and are useful to reduce congestion by having the standard cells placed a specific distance away from the macros. Similarly, during routing in specific areas of the chip, routing blockages restrict the use of specific metal layers. Routing blockages are useful in eliminating shorts by not routing over macros with metal layers used within the macro. There are different options that can be used with the *create_placement_blockage* and *create_routing_blockage* commands to define the location of the blockages.

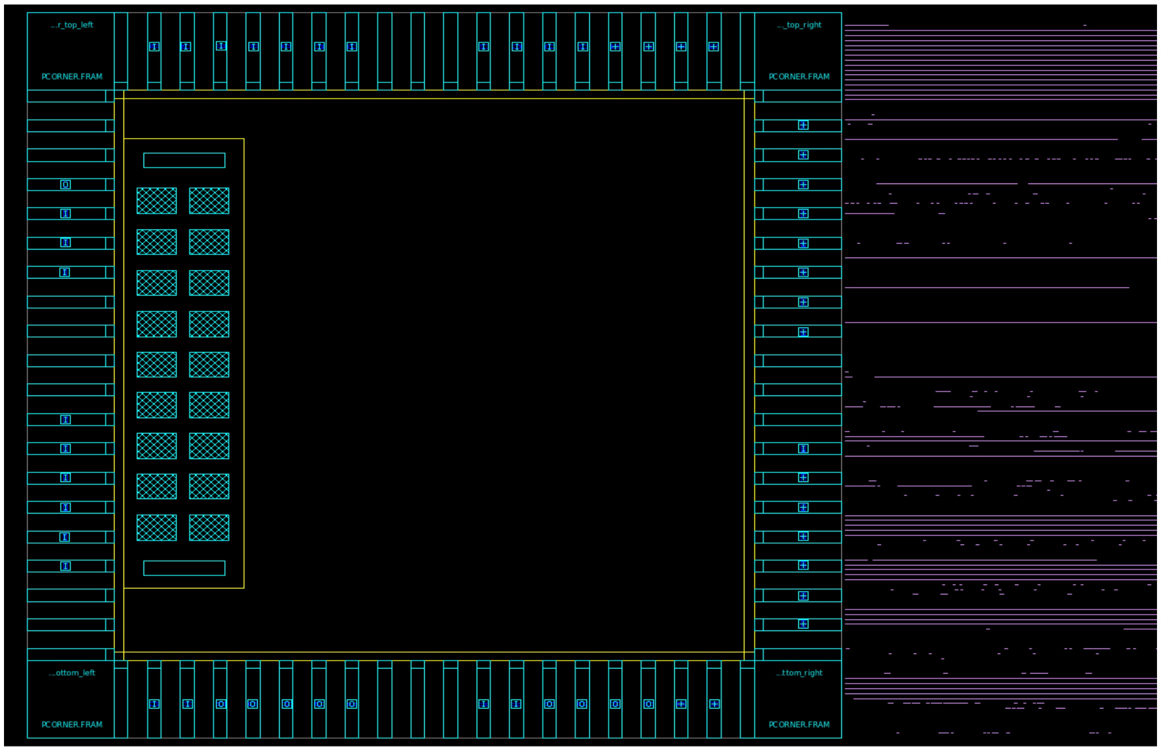

Figure 38: Floor Planning Screenshot

As shown in Figure 38, the blue rectangles on the outer edge are the IO cells and the blue rectangles in the interior are the eFuse macros and ESD cells. The yellow rectangles are placement blockages and to the right, the purple lines are the standard cells which have not been placed within the core yet.

10.2 Placement

Before the standard cells can be placed within the core, the power structure of the entire chip is created. First, the outer power rings are created followed by the inner power straps that run horizontally and vertically across the chip.

create_rectilinear_rings -nets {$name of nets} -width {$horizontal width $vertical width} -layers {$horizontal layer $vertical layer}

create_power_straps -direction $direction -start_at $starting position -nets {$name of nets} -width $width -layers $metal layer

To create the outer power rings, the *create_rectilinear_rings* command is used. The *nets* option specifies which nets (*{$name of nets}*) the outer rings are comprised of. The *width* option defines the width (*{$horizontal width $vertical width}*) of the rings and the *layers* options define which metal layers (*{$horizontal layer $vertical layer}*) will be used to create the rings. To create the inner power straps, the *create_power_straps* command is used. The *direction* option defines if this is a vertical or horizontal power strap (*$direction*), the *start_at* option defines the starting point (*$starting position*), and the *nets* option specifies which nets (*{$name of nets}*) the power straps are comprised of. Finally, the *width* option specifies the width (*$width*) of the power strap and the *layer* option defines which metal layer (*$metal layer*) is used to create the power strap.

Once the power structures are created, the power IO cells and macros are routed using the following commands.

derive_pg_connection -power_net $power net name -power_pin $power pin name -cells $name of cells to route -reconnect

preroute_instances -connect_instances specified -primary_routing_layer specified -specified_horizontal_routing_layer $horizontal metal layer -specified_vertical_routing_layer $vertical metal layer -cells $name of cells to route

Before any physical connections are made, the logical connectivity between the power nets and IO cell or macro need to be defined using the *derive_pg_connection* command. The *power_net* option identifies the name of the power net (*$power net name*) that will be connected to the specific power pin (*$power pin name*) specified by the *power_pin* option. Finally, the *cells* option defines which cells ($*name of cells to route*) the logical connectivity exists for. The physical routing is done by using the *preroute_instance* command. The *connect_instances* and *primary_routing_layer* options need to be followed by the *specified* keyword to define which metal layers and cells name to be routed. The *cells* option specifies which cells ($*name of cells to route*) are routed and which metal layer is used for vertical (*$vertical metal layer*) and horizontal ($*horizontal metal layer*) routing by utilizing the *specified_horizontal_layer* and *specified_vertical_layer* options.

Figure 39 shows the completed power structure of the chip. Three outer rings of nets VDD, VSS, and VDDQ are created along with internal power straps of VDD and VSS running throughout the entire chip horizontally and vertically. Finally, the power IO cells are connected to the outer rings and the power routing of the macros are completed.

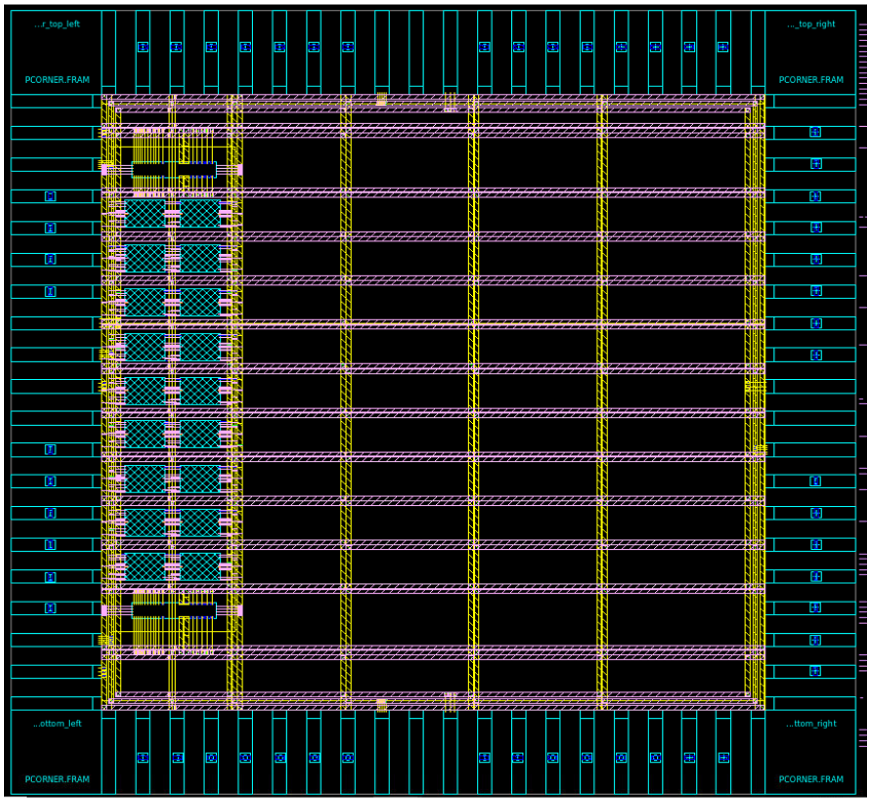

Figure 39: Power Structure Screenshot

Figure 40 and Figure 41 show the connection of the power IO cells to the outer rings and the power routing for the ESD and eFuse macros.

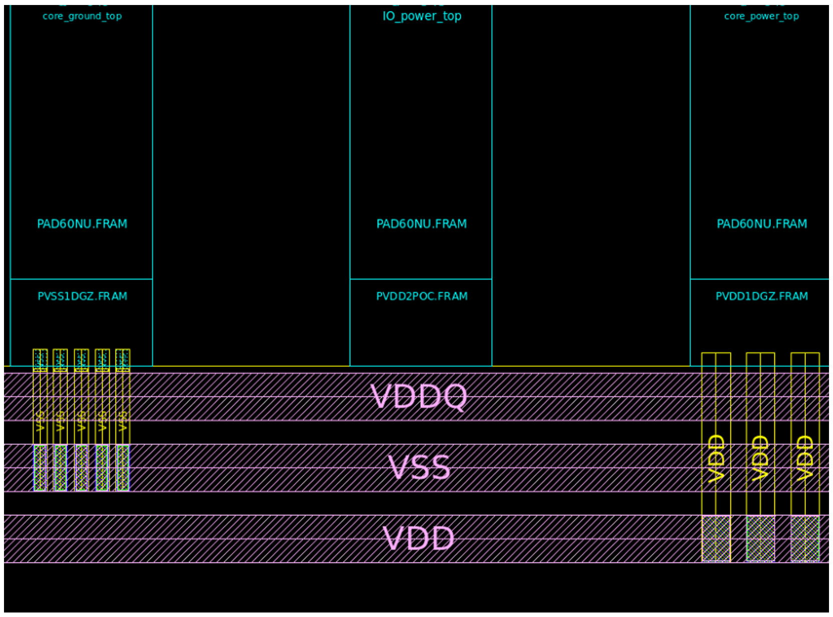

Figure 40: Power IO Connections Screenshot

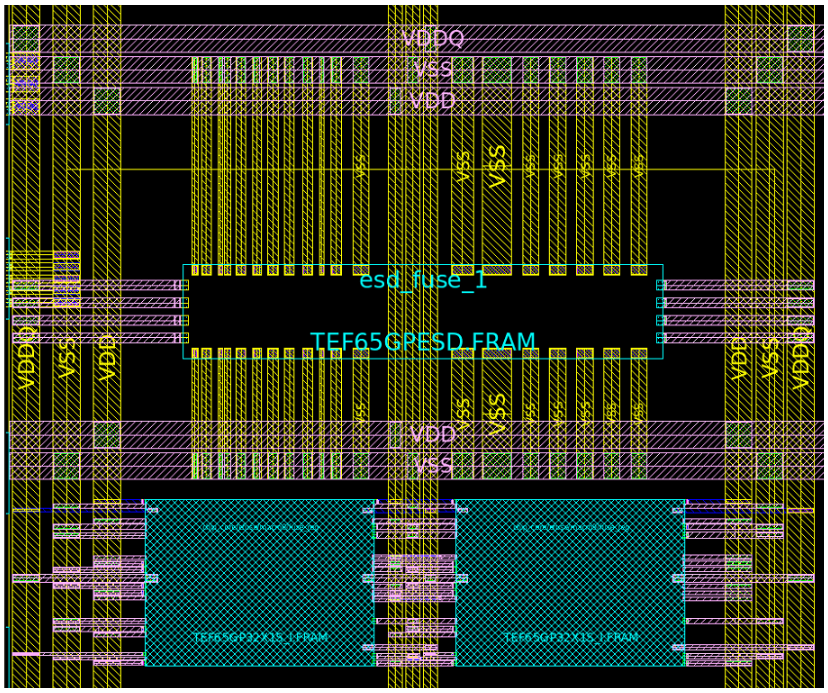

Figure 41: ESD and eFuse Macro Connections Screenshot

Bond pads connect the core pins to the package pins as shown in the figure below. These same bond pads are then connected to the IO cells that drive the signals to and from the core.

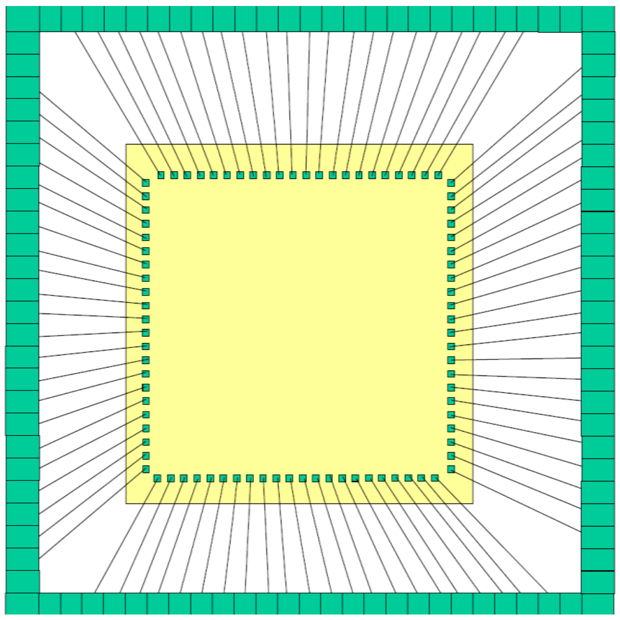

Figure 42: Bond Pads

As shown in Figure 39, the green exterior rectangles are the package pins which are connected to the bond pads. The bond pads are in the core and represented by the small green rectangles within the yellow rectangle (core). The three types of bond pads include flip chip, wire bond, and Circuit Under Pad (CUP). CUP bond pads are used to save chip area because CUP bond pads are placed directly on top of the IO cells. The following commands are used to create and place bond pads.

create_cell

move_objects

The *create_cell* command is used to create the bond pads and the *move_objects* command is used to move the bond pads to the correct location. When placing CUP bond pads, the orientation of the IO cell and bond pads need to be the same as well as their 0,0 origin needs to be aligned.

To ensure Electrostatic Discharge (ESD), dummy IO cells need to be inserted into the IO ring. Specifically, PVDD1DGZ and PVDD2DGZ need to be inserted in alternating order for ESD protection because these cells have built-in ESD protection within them. These are considered dummy IO cells because these cells do not have a bond pad that connects them to a pin package. Finally, the remaining spaces within the IO ring need to be filled with filler cells. Figure 40 represents a completed IO ring with dummy and filler cells inserted.

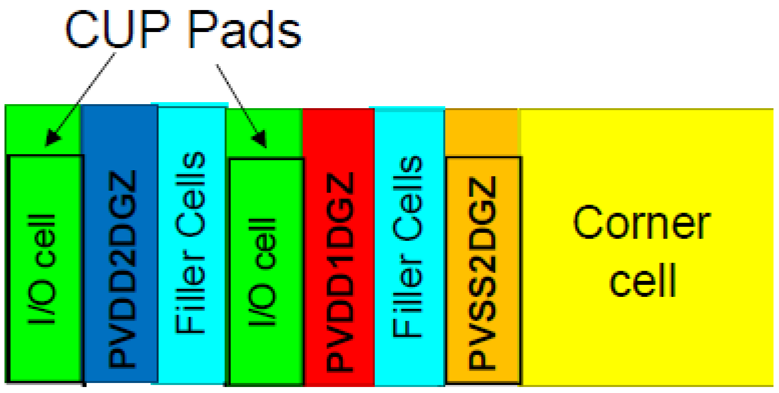

Figure 43: IO Ring

The following commands are required for the creation and placement of dummy cells as well as the insertion of filler cells.

create_cell $cell name $instance name

set_pad_physical_constraints -pad_name $pad instance name -side $side -order $order

insert_pad_filler -cell $filler cell name

Creating and placing dummy IO cells are the same as creating physical only cells and placing them using the same *create_cell* and *set_pad_physical_constraints* command. To insert filler cells, the *insert_pad_filler* command is utilized. The *cell* option specifies the name of the filler cells (*$filler cell name*). Figure 41 shows a section of the chip IO ring.

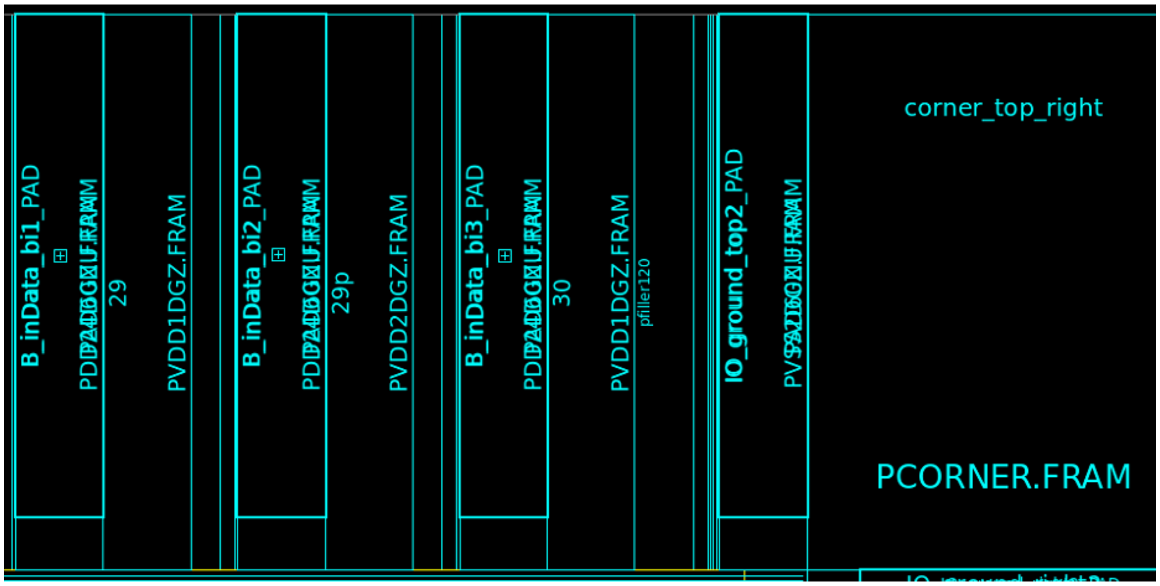

Figure 44: IO Ring Screenshot

The chip is now ready for standard cell placement, which is accomplished using the following command.

place_opt

The *place_opt* command has many options that can be used such as *congestion* option which alerts the tool to focus on congestion issues during placement or *area_recovery* which alerts the tool to downsize the gates that are not in the critical path to save area. Similarly, there are other options available and should be used when desired or needed.

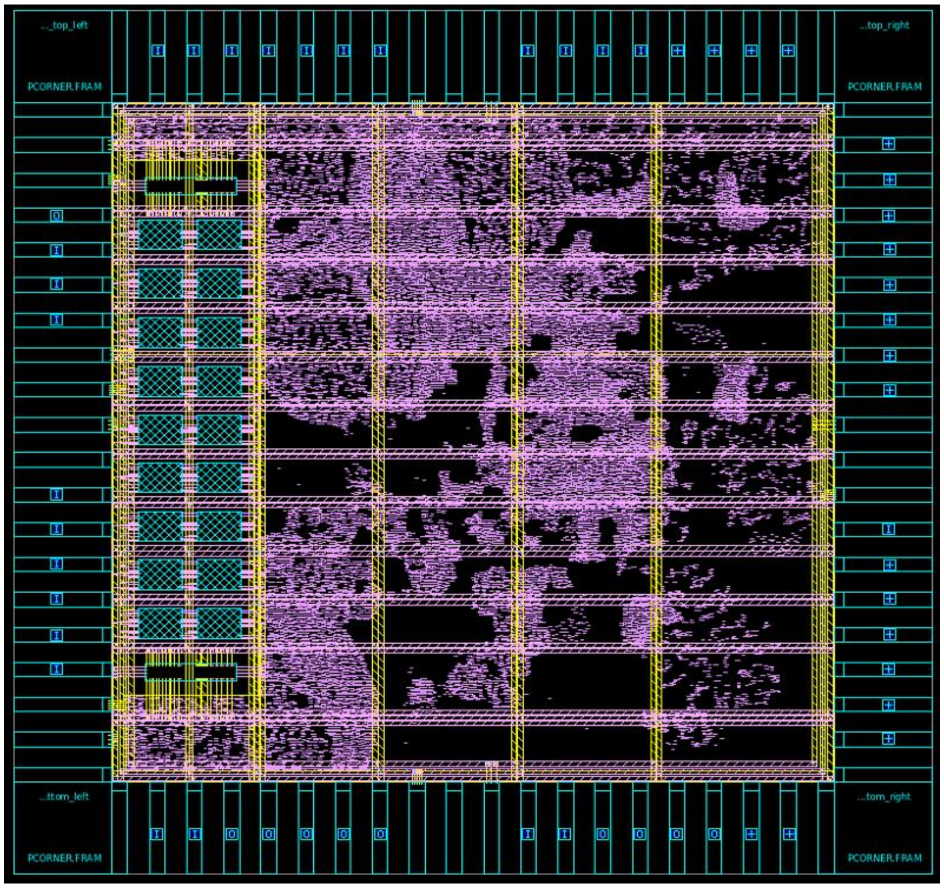

Figure 45: Placement Screenshot

Figure 45 shows the results of the *place_opt* command. All the standard cells are now placed within the core area. The black spots on the chip are not empty, this screenshot is not zoomed in enough for the standard cells placed in those spots to appear. Once the standard cells are placed, the power and ground rails of the standard cells need to be created using the following command.

preroute_standard_cells -nets {$name of nets} -connect $direction of connection

The *nets* option specifies the name of the power and ground nets (*{$name of nets}*). The *connect* option specifies in which direction the power and ground rails are created (*$direction of connection*). The rails can be created either horizontally or vertically depending on the placement of the standard cells.

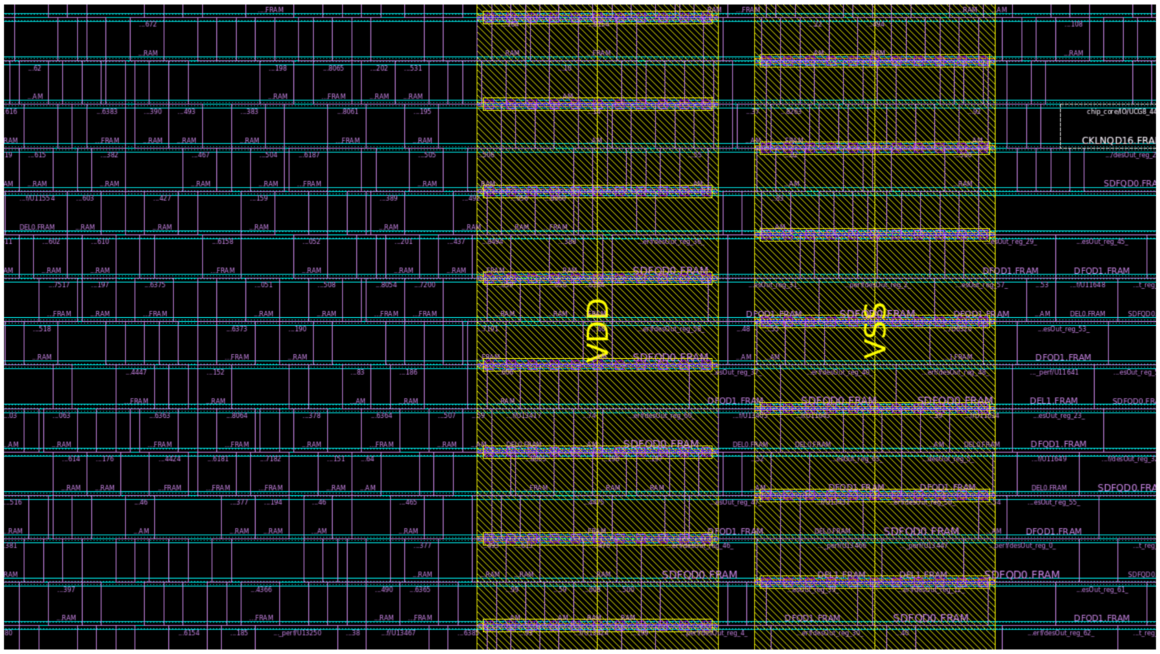

Figure 46: Standard Cell Power and Ground Rails Screenshot

Figure 46 shows the results of the *preroute_standard_cells* command. The light blue lines are the power and ground rails that connect all the standard cells. The rails are created in alternating order as VDD, VSS, VDD, VSS and so on to avoid shorts. These standard cells rails are then connected to the internal power straps. These are the yellow metals shown in Figure 46.

10.3 Clock Tree Synthesis (CTS)

The design is now ready for CTS. CTS optimizes the clock tree routes so that the clock signal arrives at the same time for all the cells that require a clock signal within the core. CTS is executed in multiple steps, first the multi-corner optimization needs to be enabled so CTS optimizes the clock tree in all the scenarios that were defined during synthesis. Then initial CTS is executed using the following command.

clock_opt -no_clock_route -only_cts

The *clock_opt* command followed by the *no_clock_route* and *only_cts* options specifies that only initial CTS is to be completed and the clock signals are not to be routed. Once this is executed, as mentioned in Section 5.3, the hold violations will now be addressed. Appropriate options for hold time fixing for all clocks are enabled and the following command is executed.

clock_opt -no_clock_route -fix_hold_all_clock

The *clock_opt* command followed by the *no_clock_route* option specifies that clock signals will not be routed. The *fix_hold_all_clock* option alerts the tool to fix as many hold violations as possible with buffer and delay cell insertion. The *clock_opt* command has many other options that enable features for reducing area or congestion etc. These options can be used as needed. The final step in CTS is to route the actual clock nets using the following command.

route_zrt_group -all_clock_nets -reuse_existing_global_route true

The *route_zrt_group* command followed by the *all_clock_nets* and *reuse_existing_global_route true* options specifies to only route the clock nets.

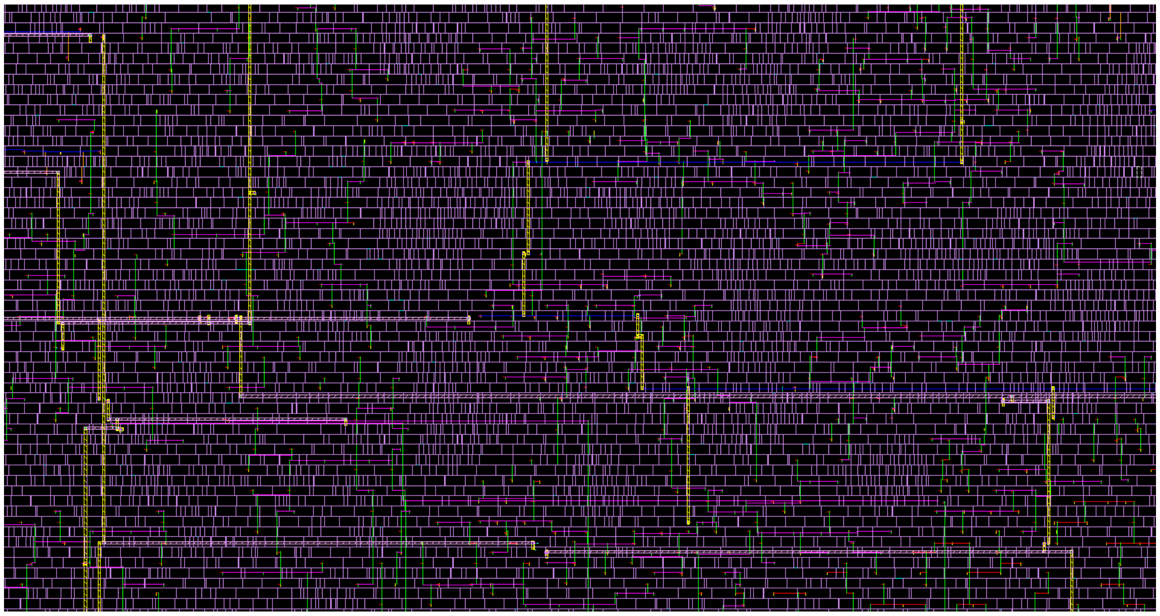

Figure 47: CTS Screenshot

Figure 47 shows the results of CTS. As shown above, only the clock nets have been routed.

10.4 Routing

After CTS, the chip is now ready for signal routing. Similar to CTS, signal routing is also done in multiple steps to achieve the best results. First, only initial routing is completed using the following command.

route_opt -initial_route_only

The *route_opt* command followed by the *initial_route_only* option alerts the tool to complete initial routing only. Once initial routing is completed, options for signal integrity are enabled. This will analyze the routing completed so far to reduce crosstalk interference among the nets. To fix all these crosstalk issues the following command is executed.

route_opt -skip_initial_routing -xtalk_reduction

The *route_opt* command followed by the *skip_initial_routing* option alerts the tool that initial routing has been completed. The *xtalk_reduction* option alerts the tool to re-route the nets that have crosstalk issues. The first two steps of routing will introduce many DRC errors which now need to be addressed. The option to prioritize design rule checking will be enabled and the following command is executed.

route_opt -incremental -only_design_rule

The *route_opt* command followed by the *incremental* option specifies that only improvements to the current routing need to be made. The *only_design_rule* option alerts the tool to fix nets that have DRC violations. Finally, antenna violations need to be fixed by either layer hopping or inserting antenna diodes. The options for fixing antenna rule violations are enabled and the final routing command shown below is executed.

route_opt -incremental -max_number_iterations $iteration

The *route_opt* command followed by the *incremental* options specifies that only improvements to the current routing need to be made. The *max_number_iterations* options followed by the number of iterations (*$iteratrion*) set how many times this final routing command will be executed. Having a high iteration number will ensure most DRC and antenna rule violations will be fixed, but this increases the runtime greatly.

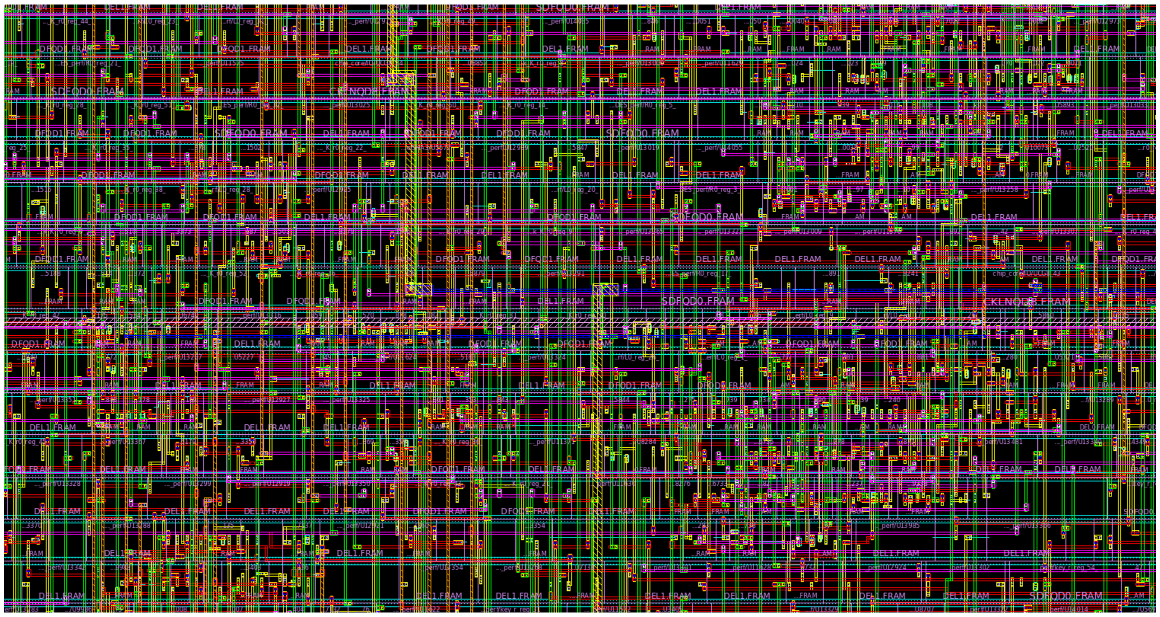

Figure 48: Routing Screenshot

Figure 48 shows the results of the routing. In comparison to Figure 47, the routing is much denser as all the signals are now routed.

10.5 Physical Design Results

The total area of the completed chip is 3.54 mm^2 and has a power consumption of 39.85 mW. The design has a total of 230,255 gates where 177,412 of those gates are combinational and 52,689 are sequential. The following table summarizes the physical design results.

Specifications	Results
Technology	TSMC 65 nm
Area	3.54 mm^2
Power	39.85 mW
Cells	230,255
Combinational	177,412
Sequential	25,689
Clock	
clk	100 MHz
SCLK	10 MHz
FCLK	100 kHz
Supply Voltage	
core	1.0 V
IO, eFuse programming	2.5 V

Table 17: Summary

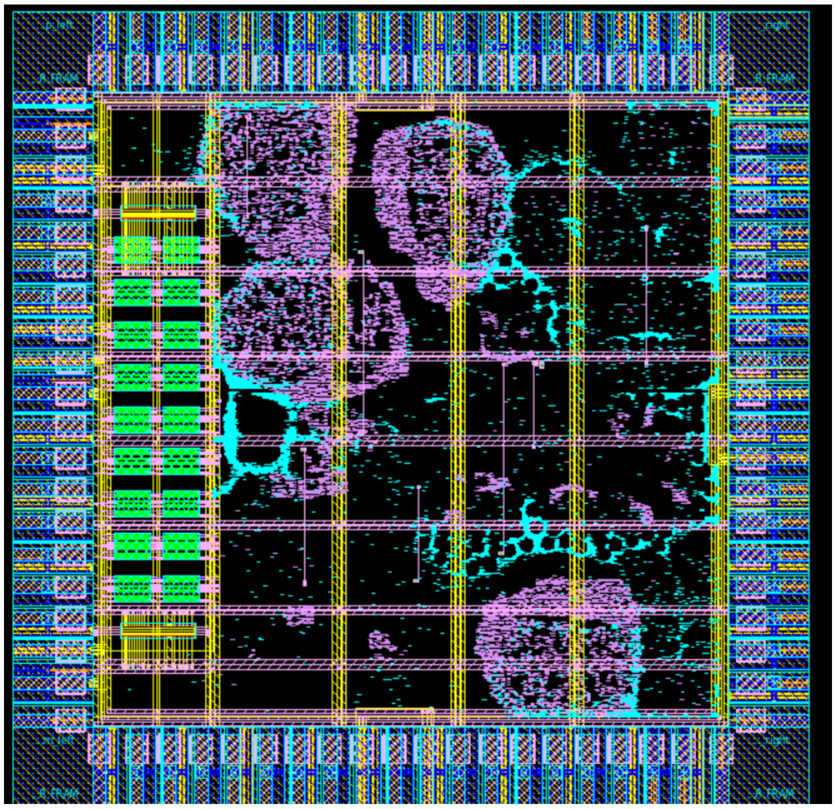

Figure 49: Final Chip Layout

11. Signoff

The design is now almost complete and needs to go through multiple signoff stages. At this point, there are a few timing violations that need to be resolved. A netlist is now generated by ICC which will be analyzed in PrimeTime. PrimeTime is a timing analysis program that will identify all the remaining timing violations and generate an Engineering Change Order (ECO) script. This script will contain information to either size up some logic cells or insert more delay or buffer cells to fix remaining timing violations. This ECO script will be imported into ICC to achieve timing closure. This process may be repeated multiple times until timing closure is achieved.

The final layout is imported in another program called RedHawk. RedHawk is used to analyze the power structure and voltage drops throughout the chip. This is completed to ensure the power structure for the chip is capable of supplying the power for all the cells within the core. If there is too much voltage drop, more power straps and IO power cells need to be added to the chip.

Once again, the layout is imported in another program called Calibre. Calibre is used for Layout vs Schematic (LVS) and DRC checks. There are additional antenna rules and metal fill checks that need to be completed using Calibre. Once all the signoff steps pass, a final netlist is generated from the layout which goes through verification using the same SystemVerilog test benches. However this time, the verification is done using timing information. Since all the nets are routed and the clock tree is implemented, the timing information is now available that will be used during final verification.

12. Future Work

In the front end, wrapper scripts need to be written to better incorporate the entire synthesis with the gate selection/replacement process. Currently, once synthesis is completed then the netlist is handed to the gate selection/replacement scripts. After this, the obfuscated benchmarks are now ready and the top-level netlist needs to be modified manually. Ideally, this would be completed in a single step through a wrapper script that does initial synthesis, gate selection/replacement, modification of top-level netlist, and re-synthesis to get final gate level netlist.

In the signoff stage, Calibre and RedHawk programs need to be explored in depth. Currently, only the basic features of these programs are used, but they need to be explored in detail so their full potential can be utilized. A wrapper script needs to be written to better incorporate the ECO scripts from PrimeTime into ICC. Currently, the ECO script is manually imported from PrimeTime into ICC, since this process happens multiple times a wrapper script can be written to automate the process.

In a research point of view, different types of memory technologies and architectures need to be explored. As shown in Section 8.1 the overhead is extremely large; therefore, different memories and architectures need to be explored to reduce overhead while meeting hardware security goals.

References

[1] Amir, Sarah, et al. “Comparative Analysis of Hardware Obfuscation for IP Protection.” *2017 Great Lakes Symposium on VLSI*, 2017, pp. 364-368.

[2] Baugarten, Alex, et al. “Preventing IC piracy using reconfigurable logic barriers.” *IEEE Design & Test of Computers*, 2010.

[3] Koushanfar, Farinaz. “Active Hardware Metering by Finite State Machine Obfuscation.” *Hardware Protection through Obfuscation*, 2017, pp.161-187.

[4] Mahmoodi, Hamid, et al. “Hybrid STT-CMOS Designs for Reverse-engineering Prevention.” *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC),* 2016, pp. 1-6.