Technical report: CoPHEE: Co-processor for Partially Homomorphic Encrypted Execution

Mohammed Nabeel[1], Mohammed Ashraf[1], Eduardo Chielle[1], Nektarios G. Tsoutsos[2] and Michail Maniatakos[1]

[1] Center for Cyber Security, New York University Abu Dhabi,
[2] Electrical and Computer Engineering Department, University of Delaware

Abstract. This technical report provides extensive information for designing, implementing, fabricating, and validating CoPHEE: A Co-Processor for Partially Homomorphic Encrypted Execution, complementing the publication appearing in the 2019 IEEE Hardware-Oriented Security and Trust symposium [NAC+19].

Keywords: Data Privacy, Encrypted Execution, Partially-Homomorphic Encryption, Hardware Root-of-Trust, ASIC.

1 `CoPHEE` Design Flow Overview

Chip overview `CoPHEE` is a hardware accelerator for partially homomorphic encryption and can also serve as a hardware root of trust for `Cryptoleq` [MTM16]. `CoPHEE` assists the host processor with the calculation of complex modular arithmetic like modular multiplication, modular exponentiation, and modular inverse, for operand size 64-bit to 2048-bit. In addition, it calculates the great common divisor (GCD) of two numbers and generates random numbers. The host processor communicates with `CoPHEE` through a UART (Universal Asynchronous Receiver-Transmitter) interface. `CoPHEE` is implemented employing Industry standard Netlist –to- GDSII flow using commerical electronic design and aAutomation (EDA) tools from Synopsys and Cadence, before sending it for fabrication at Global Foundries facility using the 65 nm technology node. Typically, a technology node refers to the gate length of the transistor. We used the Multi Project Wafer (MPW) service from MOSIS to fabricate the chip. The frequency target for our fabricated chip is 100 Mhz (due to the maximum speed of the IO pads, as it will be explained later). The chip has two voltage supplies, 3.3 V for the IO pads and 1.2 V for the logic core. As shown in Figure 1, the chip can be divided into two regions: logic core area and IO pad area. All the functions are implemented in the logic core area. The IO pad area contains the IO pads which are connected to the pins of the chip package to interface the logic core with the external world. The packaging of the chip is done with Dual in-line Package (DIP) (Figure 16). We decided to use DIP as it is very easy to plug-in to breadboard, hence easier to prototype.

Design Flow Chip design is composed of various interdependent steps. Performing them in sequence provides a highly efficient and robust design. Most of the steps use Computer-Aided Design (CAD) tools extensively. At high level, the design flow can be divided into front-end and back-end design flow. The front-end design flow can be further divided into design (this Section) and verification (Section 2). The back-end design flow is discussed in Section 4.

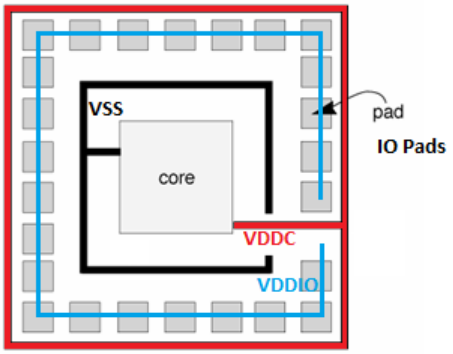

Figure 1: Pad and core area of a chip. VDDIO: voltage supply to IO pad. VDDC: voltage supply to core. VSS: ground.

1. **`Chip specification`**: the first step in a design flow is defining the chip specification. For this, a Product Requirement Document (PRD) is created, where it mentions all features that the chip should support, as well as its performance requirement and physical aspects, like the maximum die area. In the case of `CoPHEE`, the PRD requires support for all the arithmetic operations for large numbers that `Cryptoleq` needs (modular multiplication, modular exponentiation, greater common divisor, modular inverse) and security-related functions (random number generator and secure multiplexer) to accelerate processing on encrypted data and/or serve as a root of trust. In addition, a way for the host computer communicates to the chip for requesting operations and reading their results should also be included. Other parameters that should be accounted for are: 1) the die area limit defined by the foundry. In our case, it is $9mm^2$. 2) the maximum clock frequency that the chip can run. In `CoPHEE`, that frequency is 100 MHz. It is due to an architectural decision of supplying the clock externally to avoid complex integration of clock sources on chip and limitations of the IO pad.

2. **`Architectural design`**: the next step consists of coming up with the chip architecture that performs the required functions by respecting the constraints mentioned in the PRD. For example, coming up with the functional blocks to perform the required tasks and their interconnection, defining clock input, debug features and how the handshake and data transfer with external devices should be done.

3. **`System modeling`**: system modeling is an optional step. In this step, a software model of the chip with the expected functionality is created. This software model serves as reference to the logical design, in order to compare the hardware correctness and performance. We did our system modeling using python which replicates the exact functionality intended in the chip.

4. **`Logical design`**: with the architectural design and system model as reference, the logic design of the chip is constructed through implementation in the Register Transfer Level (RTL) code using a High Level Description Language (HDL), like Verilog or VHDL. Both languages are equally powerful, but since the EDA tools used in the development of this chip have a better support for Verilog, we decided to write our RTL code using it.

In the following sections, we give a top-down breakdown of the logic design with the help of RTL-like algorithms that we adapted from software implementations, wherever necessary.

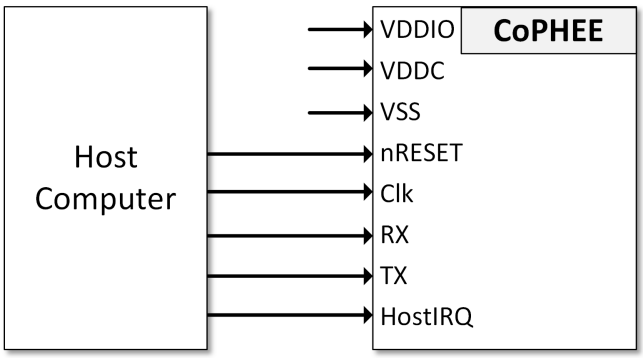

Figure 2: IO pins of `CoPHEE` and how it is connected to host computer.

1.1 External interface

The `external interface` is the set of the package pins where the IO pads of the chip are connected. Figure 2 shows the main IO pins of `CoPHEE` and how they interface with the host computer. Table 1 describes each of those pins. Through the receiver line `RX` of the UART interface, the host computer programs the value of the operands and then triggers the desired operation (e.g. modular multiplication).

The `UART` interface has a default baud rate of 1/25000 of the input clock frequency (Clk). The proportion 1/2500 was chosen as the post-silicon validation plan, considering that the clock provided to `CoPHEE` will be 24 Mhz. Thus, we get the baud rate of 9600 bps, which is supported by the majority of the devices. The transaction size supported is 8 bit without parity. The UART block waits for a read or write signature to start any operation. A signature consists of an 8-bit pattern (read: 0x4D, write: 0x34). Once the block receives a read signature from the host computer, the next four 8-bit data are treated as single word representing a 32-bit read address. When a write signature is received, and the address is passed, the UART block expects another 32 bits write data. The main reasons for choosing UART, which is a slow interface, as the communication interface with host computer is due to its less complex design, easy connection to a computer, and facilitated testing during FPGA verification and post-silicon validation compared to faster alternates like SPI or more complex interfaces like USB.

Starting an operation Through the UART interface, an operation is triggered when the trigger bit is set. Once that happens, `CoPHEE` starts the execution of the required operation, and when the operation finishes, `CoPHEE` sets the interrupt line `HostIRQ` to inform the host that the operation has been completed. Once the host computer receives this interrupt, it reads the result of the operation via UART using the transmitter line `TX`, and clears the interrupt `HostIRQ`. There is also a `GPIO` to assist with debugging and post-silicon validation.

1.2 Internal data flow

Typically, in any System on Chip (SoC), different design blocks communicate with each other using a well defined protocol called bus protocol. Figure 3 shows the internal bus architecture diagram of `CoPHEE`. It is a single-master two-slave system. The master communicates to the slaves using a 32-bit AHB-Lite bus protocol [Lim]. We have made the AHB-Lite design parameterizable to facilitate the addition of masters or slaves to the bus. A master is the design block that can initiate a transaction, while a slave can only respond to it. In `CoPHEE`, the UART is the only master in the bus and the slaves

Table 1: IO pin description of CoPHEE.

IO Pin	Direction	Description
VDDIO	in	3.3V voltage supply for IO pads
VDDC	in	1.2V voltage supply for the core logic
VSS	in	ground supply for IO pads and core logic
nRESET	in	active low reset
Clk	in	clock input (max. frequency: 100 Mhz)
RX	in	UART receive line from host
TX	out	UART transmit line to host
HostIRQ	out	interrupt to the host processor
GPIO	out	for post-silicon debugging

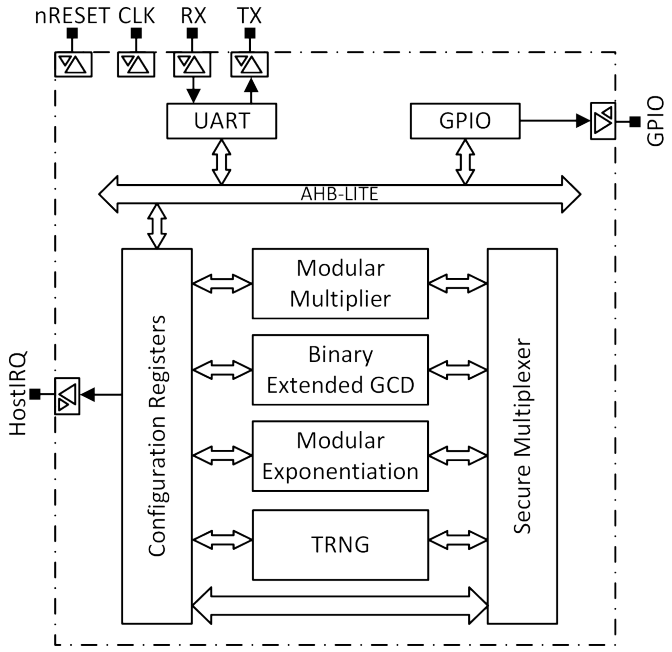

Figure 3: Bus architecture diagram of CoPHEE. Resets and clock are connected to all blocks (their connections are not shown).

are the configuration registers unit and GPIO (General Purpose IO). The GPIO is added mainly to assist debugging and verification. One can blink an LED through GPIO to signal an error during a regressive testing. The configuration registers unit is the slave which contains the registers for the key, modulus, operands, triggering an operation, result of an operation, and the status of the operation. Table 2 shows the 39 Configuration Registers of CoPHEE. Our configuration registers map to the 0x4002_0000 – 0x4002_FFFF memory range, while the GPIO maps to the 0x4003_0000 – 0x4003_FFFF range. In our design, the memory base address follows the ARM Cortex M series memory map convention for peripheral addresses.

1.3 Design blocks

CoPHEE implements a set of modular arithmetic hardware accelerators: 1) an interleaved modular multiplier, 2) a binary extended greater common divisor (GCD) algorithm, which

Table 2: CoPHEE Configuration Registers.

Register Name	Description	Bit Size
UARTMTX_PAD_CTL	IO Pad control for UART TX	32
UARTMRX_PAD_CTL	IO Pad control for UART RX	32
HOSTIRQ_PAD_CTL	IO pad control for Host Interrupt	32
GPIO0_PAD_CTL	IO pad control for GPIO	32
UARTM_BAUD_CTL	Baud control for UART	32
UARTM_CTL	UART control (parity, polarity, etc.)	32
SPARE0	Reserved register	32
SPARE1	Reserved register	32
SPARE2	Reserved register	32
CLEQCTL2	Log_2 of N	32
SIGNATURE	Stores Chip ID	32
CLCTLP	Trigger bits for modular blocks	32
CLCTL	Control bits	32
CLSTATUS	Flag bits (busy, inverse error, etc.)	32
N	Modulus N	1024
NSQ	Square of N	2048
ARGA	Argument A for modular blocks	2048
ARGB	Argument B for modular blocks	2048
ARGC	Argument C for modular blocks	2048
RAND0	Random number 0 for secure mux	1024
RAND1	Random number 1 for secure mux	1024
GFUNC_RES_ADDR	Result register for secure multiplexer	2048
MUL_RES	Result register for multiplication	2048
EXP_RES	Result register for exponentiation	2048
INV_RES	Result register for inversion	2048
DBG_REG	Debug register	2048

calculates the GCD of two operands and the modular inverse of the first operand with second operand considered as modulus, and 3) a modular exponentiation unit based on the Montgomery multiplier [MVOV96]. The chip also comprises of a True Random Number Generator (TRNG) and the Secure multiplexer for selecting on encrypted data. This section gives design details about each block.

1.3.1 Modular multiplier

We have implemented the classical interleaved modular multiplier [MVOV96]. We chose this algorithm since other modular multiplication algorithms, like the Montgomery multiplication, need to convert the operands into a different domain before doing the actual multiplication, thus, nullifying any performance benefit from a faster multiplication.

As described in Algorithm 1, the module takes two operands X and Y as input along with the modulus M, and outputs the result R, where $R = XY \; mod \; M$. Once X, Y, and M are programmed at their corresponding location in the configuration registers, the host processor sets the trigger input En to high during one clock cycle upon which the modular multiplier starts the operation. Afterwards, in every clock cycle (given by the input Clk) Y is multiplied by the least significant bit (LSB) of X. The multiplication is just a selection, adding Y in case the LSB of X is 1 or zero, otherwise. X is then shifted right by one position. Each multiplication produces an intermediary result which is accumulated to the result of previous cycles. After each multiplication and accumulation (and in the same

Algorithm 1: Classical Interleaved Modular Multiplier

```
1  INPUT: X[N-1 :0], Y[N-1 :0], M[N-1 :0], En, Clk;
2  OUTPUT: R[N-1 :0], Done;
3  if En == 1 @ Positive edge of Clk then
4  |  R = Y*X[0];
5  |  X = X >> 1;
6  |  while Any of the bit is 1 in X @ Positive edge of Clk do
7  |  |  R = (R << 1) + Y*X[0];
8  |  |  if R > M then
9  |  |  |  if R > 2M then
10 |  |  |  |  R = R - 2M;
11 |  |  |  else
12 |  |  |  |  R = R - M;
13 |  |  |  end
14 |  |  end
15 |  |  X = X >> 1;
16 |  end
17 |  Done = 1;
18 end
```

cycle), the result goes through a modular reduction. The modular multiplier stops once it reaches the most significant bit of X set to 1 (which, as discussed in the threat model, can leak information through timing and power side channels). Finally, the output *Done* is asserted as high during one clock cycle to inform the configuration register slave that the final result is ready at output R.

1.3.2 Modular exponentiation

We used the Montgomery multiplier [MVOV96] in order to implement the modular exponentiation. The main advantage of the Montgomery multiplication is regarding modular reductions. While modular reduction in the classical interleaved modular multiplier is achieved through subtractions, the Montgomery modular multiplier does it using a right shift operation, which enables it to run much faster. However, the operands need to be converted to the Montgomery domain before performing the Montgomery multiplication. In order to do the conversion, the operands are multiplied by a modular reduced version of 2^N, where N is the bit width of the modulus. Since it is necessary to convert the operands to and back from the Montgomery domain, it is only beneficial to use the Montgomery multiplication in the modular exponentiation, which executes several multiplications before converting the result back from the Montgomery domain.

Algorithm 2 presents the Montgomery multiplication. Whenever the En goes high for one clock cycle, the block starts the operation where, in each cycle, the input Y is multiplied by LSB of the input X and, then, added to the result of the previous cycle shifted right by one position. Another operand M is also added to the result if the previous result (unshifted) is odd, i.e., if its LSB is 1. It is important to notice that the addition Y + M is pre-calculated, meaning that the algorithm does not have more than one addition per clock cycle. Subtraction is required in the interleaved modular multiplication for the modular reduction, while in the Montgomery modular multiplication, the modular reduction is achieved by the right shift of the previous result (line 8). That is the main reason why the Montgomery multiplier is faster than the interleaved multiplier. X is shifted right in parallel to the addition.

The modular exponentiation operates in three stages. Figure 4 shows the modular exponentiation block. In the first stage, the input X (the base of the exponentiation) is

Algorithm 2: Montgomery Multiplier

```
1  INPUT: X[N-1 :0], Y[N-1 :0], M[N-1 :0], En, Clk;
2  OUTPUT: R[N-1 :0], Done;
3  if En == 1 @ Positive edge of Clk then
4  |   R = Y*X[0] + X[0]*Y[0]*M;
5  |   X = X >> 1;
6  |   cnt = N-1;
7  |   while Any of the bits is 1 in cnt @ Positive edge of Clk do
8  |   |   R = (R >> 1) + Y*X[0] + R[0]*M;
9  |   |   X = X >> 1;
10 |   |   cnt = cnt - 1;
11 |   end
12 |   if R > M then
13 |   |   R = R - M;
14 |   end
15 |   Done = 1;
16 end
```

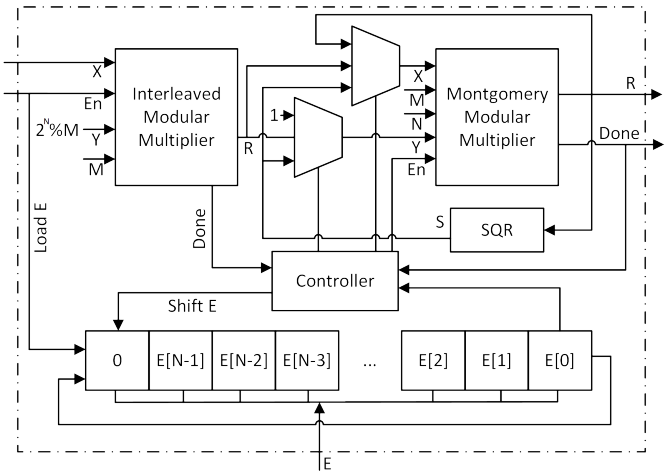

Figure 4: Modular exponentiation block.

converted to the Montgomery domain through a modular multiplication with $2^N \, mod \, M$, where N is the bit width of the modulus M. Then, the second stage the exponentiation is performed using the right to left binary exponentiation [MVOV96], presented in Algorithm 3. The controller implements this algorithm, where it controls the operands entering the Montgomery multiplier depending upon the value of each bits in E (exponent). The controller scans through each bits of the E starting from its LSB and, for every bit regardless of its value, it does repeated squaring of the input X using the Montgomery multiplier until it reaches the most significant bit (MSB) of E with value 1 (line 9). The final result of the exponentiation in the Montgomery domain is the Montgomery multiplication of all the repeated squared values S whenever the bits in E is 1 (line 7). At the end of the binary exponentiation, we have the value of X^E in the Montgomery domain. To convert back from the Montgomery domain, the controller block triggers the Montgomery multiplier by multiplying the result by 1 (line 12).

1.3.3 Modular inverse

The modular inverse of one number in relation to another exists when the two numbers are co-prime, i.e., their great common divisor (GCD) is 1. A well-known algorithm for

Algorithm 3: Binary Exponentiation followed by exiting Montgomery Domain

```
1  INPUT: X[N-1 :0], E[N-1 :0], M[N-1 :0], En, Clk;
2  OUTPUT: R[N-1 :0];
3  R = 1;
4  S = X;
5  while Any of the bits is 1 in E do
6  |  if E[0] == 1 then
7  |  |  R = Montgomery multiplication of R and S ;
8  |  end
9  |  S = Montgomery mutiplication of S and S;
10 |  E = E >> 1;
11 end
12 R = Montgomery multiplication of R and 1.
```

calculating the GCD and modular inverse is the Binary Extended GCD [MVOV96]. This algorithm calculates the GCD of two numbers and the inverse of the first argument in relation to the second, as one can see in Algorithm 4. Similar to other modules, this block also starts operating once the host processor toggles the input En after programming the inputs X (number to be inverted) and M (modulus). The goal of the algorithm is to find A and B in Equation 1, where G is the GCD of X and M. If G is 1 then A is the modular inverse of X with modulus M.

$$A * X + B * M = G \tag{1}$$

Given X and M, we create two instances of Equation 1, as listed in Equations 2 and 3, where the initial values of the set $\{A_x, B_x, A_y, B_y\}$ are $\{1, 0, 0, 1\}$, which means that the initial values of set $\{X_g, Y_g\}$ are $\{X, M\}$, as stated at line 3 of Algorithm 4.

$$X_g = A_x * X + B_x * M \tag{2}$$

$$Y_g = A_y * X + B_y * M \tag{3}$$

The algorithm then iterates doing the following three steps:

1. While both X_g and Y_g are even, divide them by 2 and multiply the GCD G by 2.
2. If only one of them is divisible by 2, their corresponding equation is divided by 2. In this case, if any of the coefficients in the equation is not divisible by 2, we add $X * Y - X * Y$ to the right side of the equation and restructure the equation to make it in the form $A * X + B * Y$, as shown in lines 15 and 23 of Algorithm 4. Then, both coefficients become divisible by 2.
3. When both X_g and Y_g are not divisible by 2, the algorithm checks if X_g is greater than Y_g. If so, the Equation 2 is updated by subtracting itself from the Equation 3. Otherwise, the Equation 3 is updated by subtracting itself from the Equation 2.

Once X_g and Y_g are equal, the algorithm terminates. The output G contains the GCD of $\{X, M\}$ and the output $Done$ is set to high for one clock cycle to inform that the process has ended. If the GCD is 1 then the modular inverse of X with respect to M is available in the output INV.

1.3.4 Random number generator

The host processor can either directly send a random number to `CoPHEE` or it can use the True Random Number Generator (TRNG) module implemented in the chip. The TRNG

Algorithm 4: Binary Extended GCD

```
1  INPUT: X[N-1 :0], M[N-1 :0], En, Clk;
2  OUTPUT: G[N-1 :0], INV[N-1 :0] Done;
3  G = Not(X[0] | Y[0]); X_g = X; Y_g = M; A_x = 1; B_x = 0; A_y = 0; B_y = 1;
4  if En == 1 @ Positive edge of Clk then
5      while X_g[0] == Y_g[0] == 0 @ Positive edge of Clk do
6          X_g = X ≫ 1; Y_g = Y ≫ 1; G = 2 * G;
7      end
8      while X_g != Y_g do
9          while X_g[0] == 0 @ Positive edge of Clk do
10             X_g = X_g ≫ 1;
11             if A_x[0] == B_x[0] == 0 then
12                 A_x = A_x ≫ 1; B_x = B_x ≫ 1;
13             else
14                 A_x = (A_x + Y) ≫ 1; B_x = (B_x − X) ≫ 1;
15             end
16         end
17         while Y_g[0] == 0 @ Positive edge of Clk do
18             Y_g = Y_g ≫ 1;
19             if A_y[0] == B_y[0] == 0 then
20                 A_y = A_y ≫ 1; B_y = B_y ≫ 1;
21             else
22                 A_y = (A_y + Y) ≫ 1; B_y = (B_y − X) ≫ 1;
23             end
24         end
25         if X_g > Y_g then
26             X_g = X_g − Y_g; A_x = A_x − A_y; B_x = B_x − B_y;
27         else
28             Y_g = Y_g − X_g; A_y = A_y − A_x; B_y = B_y − B_x;
29         end
30         G = X_g ≪ G; INV = A_x; Done = 1;
31     end
32 end
```

design is based on bi-stable circuit [EHK+03]. There are 16 individual TRNG blocks spread across the chip, as shown in Fig. 12, to improve randomness by exploiting the process variations of the chip. As mentioned in [EHK+03], a random number stream is generated after XORing the 16 individual TRNG blocks. The output of the XOR is post-processed using Von Neumann corrector to remove possible 0/1 bias. It is also possible to bypass the Von Neumann corrector. After this step, the least significant bit (LSB) of the random number is set to 1 to make sure the random number is odd. Then, a GCD between the random number and N is performed to check if they are co-prime (when the GCD is 1). If they are not co-prime, then the random number is incremented by 2 until the GCD becomes 1.

As explained in Section 1.3.5, the random number is used by the secure multiplexer to re-encrypt zero or the second input. To make sure that the output of the secure multiplexer is not repeated, the random number needs to be updated after every operation of the secure multiplexer. As generating a random number using the TRNG is time consuming, once the random number is generated and used, the next random number is calculated by squaring the existing random number, as shown in Equation 4, where R and R_{new} represent the existing and the new random number, respectively, and M the modulus. A explicit request is necessary in order to generate a new random number using the TRNG unit.

Algorithm 5: Secure Multiplexer

```
1  INPUT: X[N-1 :0], Y[N-1 :0], FKF[N-1 :0], RAND0[N-1 :0], RAND1[N-1 :0], M[N-1 :0] En,
     Clk;
2  OUTPUT: R[N-1 :0];
3  if En == 1 @ Positive edge of Clk then
4  |   RAND0 = Modular Exponentiation of RAND0 with exponent √M;
5  |   RAND1 = Modular Exponentiation of RAND1 with exponent √M;
6  |   D = Modular Exponentiation of X with exponent FKF;
7  end
8  if D <= 0 then
9  |   R = Modular multiplication of RAND0 and RAND1 ;
10 else
11 |   R = Modular multiplication of RAND0 and Y;
12 end
```

$$R_{new} = R^2 \ mod \ M \tag{4}$$

1.3.5 Secure multiplexer

The `Cryptoleq secure multiplexer` is essentially a state machine which makes use of all the above mentioned blocks. As mentioned in Algorithm 5, the secure multiplexer takes two inputs X and Y along with a function of the private key FKF and two random numbers $RAND0$ and $RAND1$, and checks (at line 8) the sign of the decryption of X (calculated at line 6). If it is less than or equal to zero, the output R receives the re-encryption of zero, else the output receives the re-encryption of Y. Both lines 4 and 5 produce an encryption of zero. The reason for having two encryptions of zero is to ensure that both conditions of the `leq test` produce similar power and timing characteristics[1]. In addition, for every call to the secure multiplexer, two new random numbers are generated, so that consecutive executions of the secure multiplexer do not produce the same output. This prevents any eavesdropping attack that tries to infer information about the ciphertexts by comparing the inputs and outputs.

1.3.6 Other blocks

The following additional blocks are also present in `CoPHEE` in order to assist with communication and control: 1) UART master, to interface with the external host computer, 2) configuration registers unit, to store the operands, modulus and results, 3) GPIO, to assist debugging in the post-silicon validation, and 4) AHB bus interconnect, to transfer data inside the chip, where the UART is the master and the configuration registers unit and GPIOs are the slaves.

The main purpose of the UART master block is to convert the read and write commands from the host processor to the AHB read and write protocol. As discussed in Section 1.1, during a reading request, the UART master receives a read signature followed by a 32-bit address. Once it receives the address, it initiate an AHB read transaction with this address on the bus and waits for a response from the slave. The slave responds with a 32-bit data that is transmitted to the host computer. During a write command, the UART master initiates an AHB write transaction passing a 32-bit address and data.

[1] While we refrained from side-channel resistant versions for the other units, since they are well-documented in the literature, we made an effort to make `Cryptoleq`'s secure multiplexer side-channel resistant since it is a completely new operation.

The configuration registers unit consists of 32-bit byte-addressable registers. There are around 800 registers in the unit, some of which are readable and writable (e.g. operands, modulus), while some are only readable (e.g. result) and others are only writable (e.g. operation trigger). The AHB-Lite bus matrix interconnect supports a single master and two slaves. The main logics in the AHB bus are the decoder logic that identifies the master's accesses to slaves, and the muxing logic, which multiplexes the data read and the ready signal from the slaves.

2 Verification

Once the logic design is complete, it is necessary to verify if the RTL code meets the specifications of the architecture document and system model. This process is called `Verification`. Figure 5 shows a typical structure of the testbench where the Design Under Test (DUT) is applied with required stimulus by the stimulus generator and the DUT response is verified by the monitor. As summarized in Table 3, the functional verification is done in three phases:

1. `Block-level simulation`: each individual modular arithmetic block is tested individually in the first phase through simulation, where stimulus from the testbench are applied directly to the inputs of the design block and the response is checked for correctness.

2. `Top-level simulation`: in the second phase of the verification, the blocks are verified from the top level, i.e., the DUT is the HDL description of `CoPHEE`. Here, the stimulus is generated by the Bus Function Model (BFM) for UART. BFM for UART is part of the top level testbench and takes care of the UART protocol needed to communicate with `CoPHEE`. In order to accelerate testing, the baud rate of the UART is set to 1/4 of the system clock, much faster than the original design (around 1/10000). This phase is also simulation-based. The testbench written in Verilog is simulated using the industry standard simulation tool Synopsys VCS. As the modulus size is as big as 2048 bits, it is impossible to do an exhaustive test. To cope with that, we use the function `$random` to generate random stimulus to increase the confidence of the verification.

3. `FPGA-based validation`: the main limitation of simulation-based verification is the slow runtime. Thus, running an exhaustive test through simulation is infeasible. Therefore, once we have confidence in simulation-based verification, we move to a FPGA-based validation for a more exhaustive test, since the runtime is much shorter. However, due to size limitations of the FPGA in conjuction with our 2048-bit modules, it is impossible to accommodate the whole 2048-bit `CoPHEE` design. Hence, we implemented a scaled-down version of `CoPHEE`, where maximum data-width supported is 256 bits. The `CoPHEE` RTL is written in such a way that the width is a parameter which accepts any power of two greater than 32. We utilized the Digilent Nexys 4 during testing. We have opted for this board after the resource requirement reported by Xiling Vivado synthesis tool for 512-bit data exceeded the capacity of the Kintex-7 and Virtex 5 FPGA boards. Our option was to scale-down to 256-bit data, which fits in the Nexys 4. Thus, we decided for this board, since it is less complex than the Kintex-7 and Virtex 5. One of the slide switch in the Nexys 4 is used as reset. The board has an on-board FTDI chip for USB to UART interface which is used to communicate with `CoPHEE`. The validation setup, written in Python, runs in the Linux terminal. It programs random operands and modulus, triggers all function units, and compares the result to the one calculated in software. The advantage of the FPGA-based validation is that the the same setup can be used

Table 3: Pre-silicon verification phases.

Verification phases	Stimulus generation	Design under test	Modulus size (bit)	Runtime
block-level	Verilog testbench	arithmetic blocks	up to 2048	slow
top-level	Verilog testbench	CoPHEE top-level	up to 2048	very slow
FPGA	Python code	CoPHEE top-level	up to 256	fast

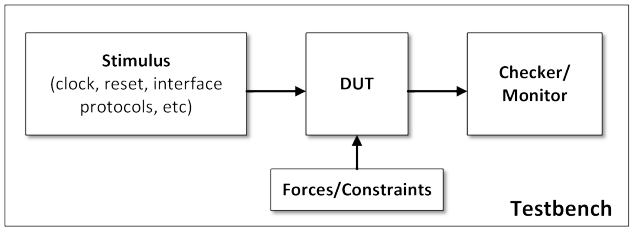

Figure 5: High-level view of the testbench.

for the post-silicon validation. The maximum frequency we could run the design in the FPGA is 25 Mhz, and the UART baud rate used for testing is 9600 bps. This frequency is generated by dividing the 100 Mhz crystal oscillator clock provided in Nexys 4 board.

3 Synthesis

Synthesis is the process of converting a circuit from its RTL code into logic gates, known as gate-level netlist. This process is typically automated using synthesis tools. Besides the RTL code, the synthesis tool requires a list of constraints along with a standard cell library. The constraints contain information about area, clock frequency, and power. The standard cell library is a collection of low-level electronic logic functions, such as logic gates and memory elements. In addition to mapping the RTL code into logic gates, the synthesis tools also optimizes the netlist to ensure that the circuit meets the targeted frequency, area, and power. Following the synthesis, a verification process is performed in order to verify if the tool has correctly generated the gate-level netlist.

The CoPHEE RTL code is synthesized using a 65nm standard cell library from Global Foundries and a clock constraint of 100 Mhz. As the UART and GPIOs are the only interfaces of CoPHEE (i.e., both are asynchronous), there is no specific IO timing constraint. Following common practices, the standard cell library used for synthesis was the one characterized for the worst voltage (1.08V), temperature (125C), resistance, and capacitance. Synthesizing with such a library ensures that we can achieve the target frequency.

For synthesis, we used the Synopsys Design Compiler (DC), while for post-synthesis and formal verification we used the Cadence Conformal, and we were able to ensure that the RTL code and the synthesized netlist are functionally equivalent. In Table 4 we presents the area and timing estimations of the major CoPHEE blocks after synthesis. The largest design is the binary extended GCD, followed by the configuration registers that store a total of 2.73 KB. As expected, the modular multiplier and the modular exponentiation unit are roughly the same size, and the rest of the modules occupy significantly less area.

4 Physical Design

Physical Design is the process of layout implementation and analysis of a design netlist ensuring to certain user and foundry that specific constraints are met in order for the chip

Table 4: Post-synthesis area and timing estimations.

Blocks	**Area** (um^2)	**Worst path delay** (ns)
Configuration registers	512,724	4.770
Binary extended GCD	548,454	9.433
Interleaved mod. multiplier	315,487	9.374
Modular exponentiation	304,064	9.389
AHB bus	1,261	4.850
UART master	3,466	4.930
TRNG	28,658	NA

to be fabricated. The physical design can be subdivided into PnR (Place and Route) and SignOff Analysis. Sections 4.1 and 4.2 describe them in detail. The inputs to physical design process can be broadly classified into two types: design-related and technology-related. The design-related files are the design netlist and the constraints. We obtained the netlist in the Verilog format and the constraints in the Synopsys Design Constraint (SDC) format after the synthesis from Synopsys Design Compiler. The technology-related files are:

- `Technology`: this file is specific to the technology and contain basic information related to the physical implementation. It contains, for example, various metal/via layers, their width, pitch, preferred routing direction, etc.
- `Physical libraries`: it includes the physical libraries of all the standard cells and macros used in the design. These libraries capture the physical description of each primitive standard cells, macros, IO pads, etc. The physical description involves the cell size, their pin shapes and locations, etc. These are usually in the LEF (Library Exchange Format) format.
- `Timing libraries`: it regards the timing libraries for all the primitives used in the design. These libraries defines the function along with the area, delay, power, transition, pin capacitance, setup/hold characteristics of each cell available in the library. Each logic cell in the physical library should have a corresponding entry in the timing library for them to be used in the valid timing paths in the design. The libraries are usually either NLDM (Non Linear Delay Model) or CCS (Composite Current Source). We used CCS libraries as they are more accurate.
- `Interconnect technology`: this file is used for parasitic extraction. The file contains details about RC (parasitic) values of various metal layers and vias. It also captures the RC for possible variations of width, thickness of metal layers due to variation in the lithographic processes. This information is needed to extract parasitic of every net in the design layout, to be used for timing and rail analysis.
- `Physical verification rule deck`: The DRC (Design Rule Check) rule deck contains foundry rules of manufacturing to be verified in every design layout for it to be qualified for tape out. The LVS (Layout Versus Schematic) rule deck defines rules for device extraction from layout and to compare the layout against the schematic netlist.

The chip was fabricated using the Multi Project Wafer (MPW) program provided by Metal Oxide Semiconductor Implementation Service (MOSIS). We chose the metal layer stack `5_02_00_00_LB` from the available different stacks. The metal and via layers are M1, V1, M2, V2, M3, V3, M4, V4, M5, WT, BA, WA, BB, VV, and LB. These layers also have preferred routing direction (horizontal or vertical). The tool ensures that it uses each metal layer in its preferred routing direction most of the time in order have optimal

Table 5: CAD tools used for implementation and analysis.

Stage	Tool	Vendor
Place and Route	IC compiler	Synopsys
RC Extraction	starRC-XT	Synopsys
Static Timing Analysis	PrimeTime	Synopsys
Rail Analysis	PrimeRail	Synopsys
Formal Verification	Formality	Synopsys
Layout strmin and editing	Virtuoso	Cadence
Physical Design Rule Check	PVS	Cadence
Layout Versus Schematic	PVS	Cadence
Signoff fill insertion	Calibre	Mentor

usage of routing resources. This stack provided us enough signal routing resources in first 5 metal layers and the top two to be used mostly for power/gnd and clock structure. The layer LB wasn't used in the layout except for the IO PADs. The external bonding to the chip during packaging is done using these LB cups on the pads. The Global Foundries Process Design Kit (GF-PDK) for 65LPE process provided us all the technology-related files. We got the physical and timing libraries for standard cells from ARM. ARAGIO supplied us the IO Pad libraries. The chip was implemented flat without any physical hierarchy. Several CAD tools are available for physical implementation and analysis. Table 5 shows the cad tools we used along with their stages and EDA vendor.

4.1 Place and Route (PnR)

`PnR` is the first step in physical design and involves various steps like die size estimation, floor planning, power planning, placement, clock tree synthesis (CTS), routing, and post-route optimization. Formal verification is also performed at each of these stages to ensure netlist equivalency.

4.1.1 Die size estimation

`Die size estimation` is the first stage of PnR, which involves estimating the die and core area. The estimation starts considering the maximum final utilization (FU) that one wants to achieve. A layout designer does not want the final utilization to be too high as a small disturbance from an optimization or ECO (Engineering Change Order) leads to large disturbance in layout timing, recovering from which will be time consuming and difficult. Nevertheless, a very relaxed utilization means more silicon and, thus, more cost. Hence, the designer needs to find a sweet spot which he arrives after multiple iteration of PnR. It is important to notice that the design utilization often increases during PnR due to timing and design rule optimization. This growth in utilization is noted for your design after initial trials and, thus, the initial utilization (IU) is determined. Based on the aspect ratio and the size and shape of the hard macros in the design, width and height of the core region is determined, as one can see in Equations 5, 6, 7, and 8, where IU is the initial utilization, FU is the final utilization, x is the % growth in utilization due to clock tree synthesis and timing optimization, RCA is the required standard cell area, SA is the synthesis area, MA is the macro area (including RAMs or any physical IPs), DW and DH are the die width and height, respectively, CW and CH are the core width and height, respectively, HIO is the height of the IO PAD, and CIO is the core to IO spacing.

$$IU = FU/(1 + x/100) \tag{5}$$

$$RCA = (SA - MA)/IU \tag{6}$$

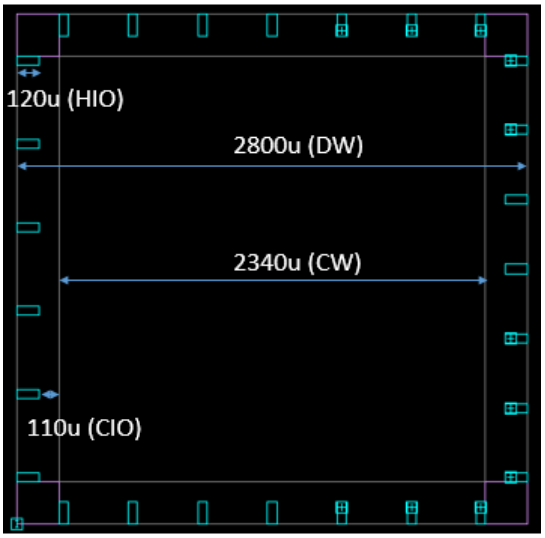

Figure 6: Chip physical parameters.

Table 6: Layout physical parameters.

Parameter	**Value**
IU (Initial Utilization)	36 %
FU (Final Utilization)	47 %
MA (Macro Area)	0 um^2
HIO (IO PAD Height)	120 u
CIO (Core to IO spacing)	110 u
A (Aspect ratio)	1
RCA (Required std cell Area)	1956692.52 um^2
CW (Core Width)	2340 u
CH (Core Height)	2340 u
DW (Die Width)	2800 u
DH (Die Height)	2800 u

$$DW = CW + HIO + CIO \tag{7}$$

$$DH = CH + HIO + CIO \tag{8}$$

Fig. 6 shows the above physical parameters with respect to our layout. Table 6 captures other relevant details of this stage. The minimum chip size supported by Global Foundries for 65lpe through MOSIS is $9mm^2$. This is more than the estimated die size of our chip. Thus, we decided to use the entire area as there was no reason in reducing the chip area.

4.1.2 Floor planning

`Floor planning` is the step where an actual layout core and die boundaries are created (e.g. the location of the IO pads and memory elements). This process is based on the die size estimation. After the floor planning, the design is ready to be taken to the detailed placement of standard cells. There could be multiple iterations of floorplan based on the netlist revision, timing and congestion results from post-placement or routing. The floor planning usually involves the five following steps:

1. **`Design import`**: this involves importing the design netlist and constraints along with technology LEF, timing and physical libraries of standard cells.
2. **`Die/core outline creation`**: here a layout outline is created based on the die size estimation.
3. **`Signal and power PAD distribution`**: this step involves the distribution of signal and power pads along the periphery following foundry IO guidelines.
4. **`Placement of hard/soft macros`**: all the hard macros (RAM/ROM/PLL, etc) used in the design are then placed mostly manually and sometimes with the guidance of AMP (Automatic Macro Planning) tools. The macro placement is guided by data flow in the design and pad/pin distribution. This is followed by hard and soft placement blockages insertion around the macros to control the placement of standard cells in narrow channels or close to these macros to avoid routing congestion.
5. **`ENDCAP and TAPCELL insertion`**: as per foundry requirement for some technologies, specific cells named ENDCAP cells are inserted on either edge of the layout. TAP cells are also inserted in the layout to tie the wells in order to avoid LATCHUP violations.

In our implementation, after we created a milkyway database (Synopsys database) using the technology and other libraries, the design netlist and constraints were imported. The IO pad distribution guidelines were provided in the TCL format before the layout outline creation. The file contains the order and edge for each IO pad in the design adhering to IO pad placement guidelines from Global Foundries. Thus, we could distribute all the IO pads along the edges during die and core boundaries creation. Layout outline along with IO pad placement is shown in Figure 6. We have 27 IO pads in total, 8 of them are for VDD/VSS core power/ground supply, 8 DVDD/DVSS for IO power/ground, and the remaining 11 are signal pads. One supply and one ground pad were enough for us, but we decided to utilize the empty spaces in the IO pad ring with more of them to make the power structure robust. The light green rectangular structures in the periphery are the pads. The empty spaces between the pads are filled with filler pads to maintain continuity of the internal power/ground and other special signals. These pad fillers can be noticed in the Figure 7. The tool also creates horizontal structures called rows of equal height throughout the core region. The height of the each row is dependent on the technology and this information is picked up by the tool from the technology files provided. These rows define the legal locations for standard cell placement. ENDCAPs and well tie cells were then distributed in the core region as per the foundry requirement. Figure 7 also shows tap/tie cells placed in a staggered fashion in the design layout. Figure 8 shows an enlarged view of the distribution. We then ran the recommended ZIC (Zero Interconnect Delay) analysis to check the quality of the input netlist for timing and were good to go.

4.1.3 Power planning

`Power planning` or **`power network synthesis (PNS)`** is the creation of a well designed power structure to deliver power from the power pads to every standard cells and macros used in the design. The structure should also ensure that both static and dynamic voltage drop to each cell in the design is within the recommended range (3.3% static, 5.5% dynamic). These numbers are based on our prior experience. Having a dense power distribution keep you safe on voltage drop but consumes lot of routing resources and might make the design congested or unroutable for other signal nets in the design. Hence power structure should neither be too aggressive nor relaxed. A few PnR iterations along with rail analysis (Section 4.2.3) might be needed to finalize the power structure. A power structure consists of:

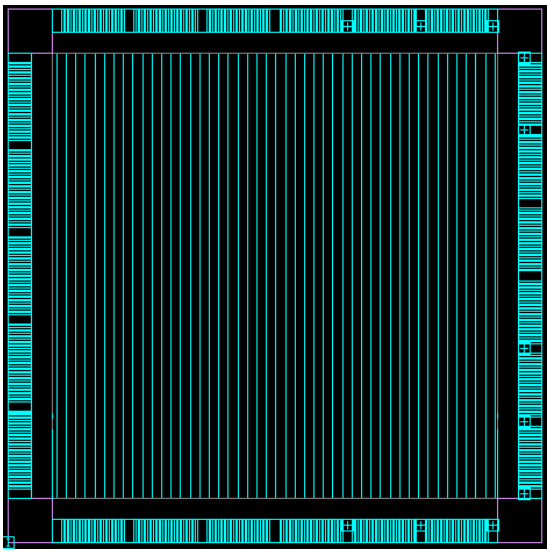

Figure 7: Tap cell distribution.

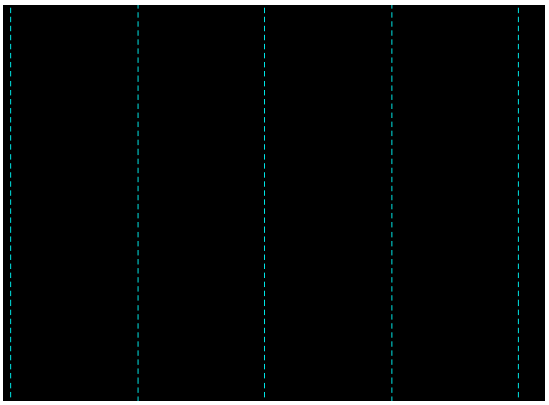

Figure 8: Tap cell distribution enlarged.

- **`PG IO pads`**: Power and Ground IO pads receive power from an external source and drive the internal power distribution network.
- **`PG rings`**: they form a ring of power and ground lines in the higher metal layers around the core region and are connected to the power pads.
- **`PG straps`**: PG straps are the ones that runs horizontally and vertically in power and ground pairs throughout the core region. They are connected to each other as well as to the power ring using power via arrays. The frequency of the power straps can be tuned to generate an optimal power distribution network.
- **`PG rails`**: these are rails that runs along the standard cell rows. The metal layer for rail creation depends on the layer at which the power pins of the standard cells are designed for, which is defined during the standard cell library development. These power pins are usually designed to be on the top and bottom edges of the standard cells and to run horizontally along the rows.

We created a core power ring as seen as the red and white lines around the core in Figure 9. They are in metal layers BA (red vertical) and BB (white horizontal). The thick

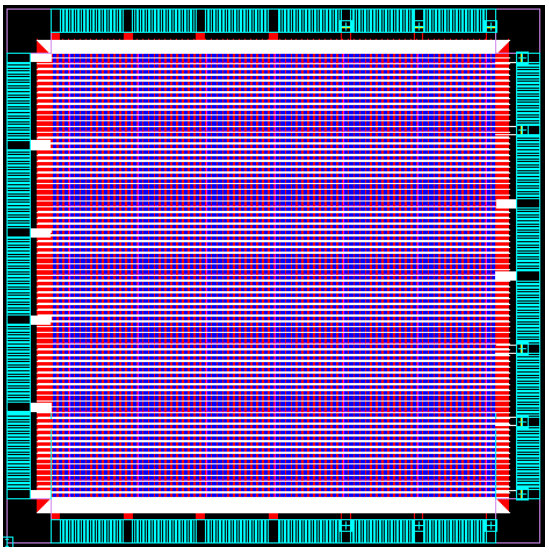

Figure 9: Power network.

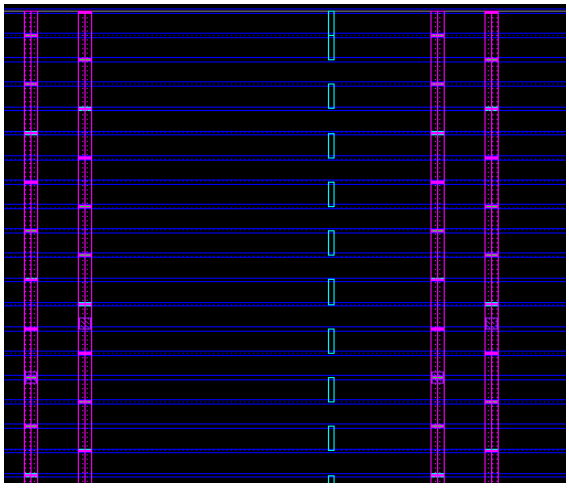

Figure 10: Power network M4 M1 enlarged.

white and red lines connects the ring to the pads. We have power straps created in layers BA, BB, M5 and M4. The mesh structure in Figure 9 inside the core region shows the power straps distribution. Our PG rails are in M1 and are connected to the power straps in M4 through power via stack from M4. Please note that the M4 strap runs vertically (preferred routing direction vertical) and vias can be dropped on every intersection of them with horizontal M1 rails. Figure 10 shows the via connection from M4 (purple vertical straps) to M1 (blue horizontal rails). Alternately it is also possible to have straps in M3 and M2 and drop vias from M2 straps to the M1 rails. However, this would increase the routing congestion and it was discarded.

4.1.4 Placement And Optimization

After floorplanning and power planning, the next challenge is to place the standard cells in the design netlist in accordance with the foundry and user's placement constraints. This process is know as `placement`. The standard cells should be placed in legal locations defined by row and site in the technology and should not overlap each other. The CAD

tool takes care of this and certain checks post-placement can confirm the adherence to the guidelines. CAD tools consider any specific hard/soft placement blockages created by the user in the layout while placing the design. If power structure is present in the lower layers (e.g. M2), the area below these power straps is partially blocked in order to not create accessibility issues to the pins of the standard cells. Placement also takes standard cell connectivity, design and library constraints in to consideration. The netlist is further optimized to fix design rule violations (e.g. the maximum transition, fanout, capacitance, etc) and for performance, area and power. This involves addition/deletion of buffers, up-sizing or down-sizing of cells, and also logic restructuring to an extend. The layout designer evaluates the post-placement timing, area, power and congestion to evaluate the quality of netlist, floorplan, power plan and, sometimes, the library, in case it is immature.

Prior to the standard cell placement, we grouped and distributed the TRNG module using bounds/regions in the chip to leverage On Chip Variation (OCV). Bound is a feature in the tool used to restrict placement of specific cells according to the user specification. Figure 12 shows the TRNG distribution. The standard cells in the TRNG module were not allowed to be optimized by the tool. The red highlighted groups of standard cells in the image compose the TRNG module. After fixing these groups in position, the rest of the standard cells went through placement and optimization. The design was then analyzed for timing, congestion, area and power and passed the requirements. We enabled high threshold voltage (HVT) and regular threshold voltage cells (RVT) during optimization. HVT are low-leakage high-delay cells and RVT are medium-leakage medium-delay cells. LVT (low threshold) cells which are high-leakage low-delay cells were used only in the final timing closure. We used this kind of approach to limit the power leakage as in accordance to standard practices.

Figure 11 shows the placement distribution of various important modules of the design after placement. The Binary Extended GCD is presented in red, Interleaved Modular Multiplier in green, Modular Exponentiation in blue, and purple refers to the remaining modules. As one can see, the GCD module consumes a good portion of the design and is confined to right side of the chip. The modular exponentiation and multiplier follows. We performed a trial placement run by creating specific regions in the chip for these modules. A region restricts the placement of the module to a specified location of the chip. The timing results have not turn out to be better than the one provided without regions and, hence, was discarded.

4.1.5 Clock Tree Synthesis

`Clock three synthesis (CTS)` is a method that aims at minimizing the insertion delay and skew through the insertion of buffers or inverters along the clock path. The insertion delay (ID) of a design is the worst arrival time to the sinks, and the skew is the difference between the longest and shortest ID.

Nowadays, most of the designs are sequential and they need to be properly clocked for the desired operation. Without CTS, the clock would usually be a high fanout net (HFN), which is considered ideal for timing analysis before CTS. Nevertheless, it is important to create a tree structure on this HFN to deliver the clock signal to all the sequential elements (sinks) in the design possibly at the same time. Minimizing the ID reduces the OCV impact and minimizing skew helps in timing closure. Some CAD tools work on timing closure rather than minimizing skew which involves introducing additional skew in some paths to improve timing in other paths referred as useful skew.

Prior to the clock tree synthesis, the layout designer needs to decide on the CTS constraints. Assuming that the layout designer tries to achieve the best ID and skew possible, we list below some important CTS constraints:

- **`Non Default Rule (NDR)`**: these are special rules defined to make the clock more

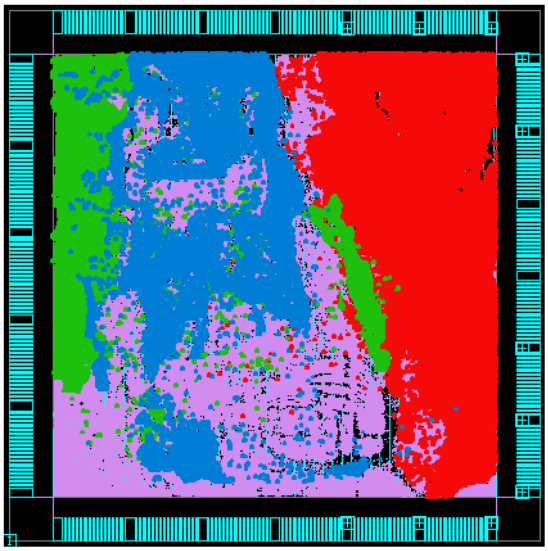

Figure 11: Placement distribution (binary extended GCD: red; interleaved modular multiplier: green; modular exponentiation: blue; other modules: purple).

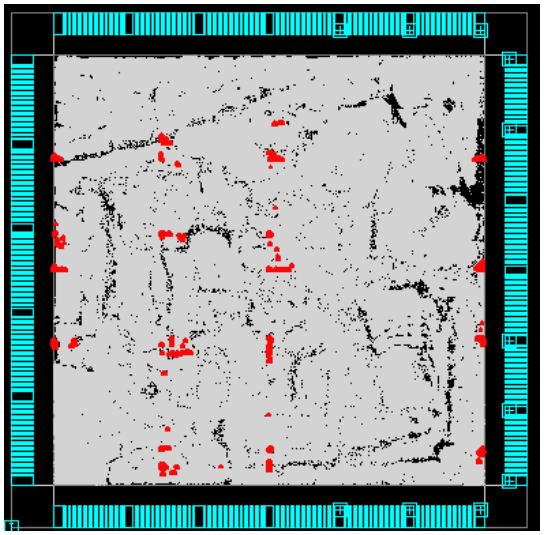

Figure 12: TRNG distribution (red).

robust and avoid signal integrity issues in the design.

- **`Clock Routing Layers`**: the designer tries to route clock nets mostly in higher layers as they are less resistive and hence lesser voltage drop. Thus, NDR rules can be honoured easily. It addition, it does not affect much the signal routing in lower layers.
- **`CLK Buffer/Inverter List`**: the layout designer need to make a decision on the clock buffers/inverters that should be used for building the clock network. The first decision is to to pick up the right threshold voltage group and then the proper driving strength for these buffers/inverters. HVT cells consume less power but are susceptible to On Chip Variation (OCV) while LVT cells consumes a lot of power. Hence, generally RVT cells which are a midway between HVT and LVT are used for CTS.

Table 7: Buffers/inverters used for CTS

Buffers	**Inverters**
BUFH_X4M_A9TR	INV_X4M_A9TR
BUFH_X5M_A9TR	INV_X5M_A9TR
BUFH_X6M_A9TR	INV_X6M_A9TR
BUFH_X9M_A9TR	INV_X9M_A9TR

Table 8: Post-CTS statistics

Parameter	**Value**
clock name	HCLK
CTS synthesis corner	slow
Number of levels	45
Number of Sinks	67628
Number of clock tree buffers	9921
Global Skew	162 ps
Longest Insertion delay	2.410 ns
Shortest Insertion delay	2.248 ns
Standard cell utilization	43.63 %

We performed CTS using the buffers/inverters listed in Table 7 from the ARM standard cell library. The middle part of the name indicates the driving strength of the cell (e.g. X9M). As we discussed in the CTS constraints above, this list of buffers are RVT cells of medium driving strength helping us to achieve reduced OCV, robust clock network and less power.

A Non-Default Rule (NDR) of double width and double spacing was created and assigned to the clock trunk nets. Trunk nets are all the clock nets except those connected to sinks directly. Clock nets were then routed using metal layers M4, M5, BA, and BB, and are assigned as soft fixed. Being soft fixed, their change, during signal routing is restricted and, thus, keep the clock network intact. This was followed by a multi-corner optimization for fixing design rule violations, setup and hold timing closure. Table 8 shows post-CTS statistics. It describes important metrics like the insertion delay, skew, clock tree buffer/inverter count, etc. The layout designer considers these metrics to evaluate how good is the clock tree. We achieved a decent skew of 162 ps with nominal count of clock buffers/inverters.

4.1.6 Signal Routing and Optimization

The process of establishing physical connectivity between the gates using available metal layers, based on the logical connectivity in the design is termed **`signal routing`**. A standard router provided by CAD tools follows all the routing guidelines or rules from the foundry and ensures that the design is completely routed with no opens or DRC violations. Congestion, cross-talk, and timing are also important parameters considered during routing. In the cases where the design is unroutable or ends up with opens or DRC or timing violations, the reason needs to be identified and suitable fixes applied. Some fixes may even need to go back to floorplan stage. Then, the design goes again through various stages and the routing stage to verify the fix. Before routing, parasitics are estimated based on a basic trial routing. Trial route is often used to estimate timing and congestion and may differ from actual detail route. This difference leads to deviation of post route timing results from pre route. In order to recover from any DRV/timing violation which might have been caused by these routing differences, further round of optimization involving mostly buffer addition/deletion or up/down sizing is done .

Table 9: Design statistics through PnR

Parameter	**Initial**	**Place**	**CTS**	**Route**
# of Standard cells	693333	775548	784795	786526
# of Sequential cells	67628	67628	67628	67628
# of Combinational cells	625708	707923	717170	718901
# of Buffer/Inverter cells	91139	707923	175163	176894
Standard Cell Utilization	35.45 %	43.44 %	46.35 %	46.56 %
# of Signal nets	695425	777637	771789	773520
HVT cells	100 %	55.8 %	67.7 %	67.8 %
RVT cells	0 %	44.2 %	32.3 %	32.2 %
Total wire length (μm)	NA	NA	1242079	46668040

Table 10: Redundant via statistics

Layer	**# of multi-cut vias**	**# of total vias**	**% of multi-cut vias**
V1	2003289	2579845	77.65
V2	2129266	246275	86.46
V3	688693	869281	79.23
V4	417410	544818	76.61
WT	76473	134395	56.90
WA	59085	100430	58.83

Design for yield requires redundant vias to be inserted wherever possible. Connectivity between metal layers are made possible with vias or cuts between them. A single via is enough for establishing the connectivity, but multiple vias for the same connection safe guard the connection from possible malformation of vias during manufacturing. The white spaces in the design are then filled with decoupling capacitance fillers and normal standard cell fillers. Decoupling capacitance (DCAP) fillers are special fillers available in the library that can store some charge and deliver in case of sudden dynamic requirement of power.

We took the optimized post-CTS database and followed the same procedure described above to generate a DRC and a timing clean layout database. The analysis stage after PnR requires the design information to be passed on to it in some specific formats. For this purpose, we dump the DEF, GDSII, formal netlist and LVS netlist. The DEF (Design Exchange Format) captures all the design information needed for signoff parasitic extraction. The GDSII carries the design information for physical verification. Formal netlist is used for formal verification and the LVS netlist is for LVS in physical verification. Both formal and LVS netlists are similar, except that the LVS netlist also contains physical-only cells (e.g. filler cells, tap cells) and the PG pin connections. Table 10 shows the percentage of redundant vias for various via layers. We were able to achieve more than 75 % conversion of single to multi-cuts for lower via layers V1, V2, V3, V4, but a lower percentage for higher layers. Please note that V1 is the via that connects Metal 1 to Metal 2 and so on. Table 9 shows an interesting design statistics over various stages in the PnR. Please note the increase in the standard cell count as the design move from initial to final routing stage. It can be noted as evident from the buffers/inverters count that this is primarily because of the buffers/inverters inserted in the design to fix design rule violations, clock tree synthesis and timing fixes. The design started with 100 % HVT cells and ended up with 67.8 %. They were swapped with RVT cells, also for timing and DRV fixes. Wire length numbers for CTS captures only clock net wire length while for routing is the entire nets route lengths.

4.2 SignOff Analysis

After PnR, `SignOff analysis` qualifies the design for fabrication. Implementation tools are less accurate compared to signoff tools. This is primarily because of the trade-off between turn around time and accuracy. Therefore, SignOff analysis is required for static timing analysis (STA), physical verification involving DRC and LVS, and rail analysis. Formal verification is also performed for logic equivalence between the final netlist from PnR and the initial netlist from synthesis.

4.2.1 Static Timing Analysis

`Static timing analysis (STA)` is the timing evaluation of the final layout in SignOff timing tool (primeTimeSI), using an accurate parasitic data from signOff extracter (starRC). That is necessary because, PnR tools usually uses quick and inaccurate parasitic extraction for timing evaluation of the layout through out various stages, since an accurate parasitic extraction takes considerable more time than the normal extraction generally employed by PnR tools. For that reason, we need to analyze the final layout with signoff extraction and timing analysis tools.

An STA tool performs timing analysis after reading in the design netlist, an accurate parasitic data, and timing library in order to create various violation reports. The layout designer goes through these reports and generate fixes for them if required. The fixes are implemented in the PnR tool which is followed by a new signoff extraction and STA. This process is termed Engineering Change Order (ECO).

The parasitcs are extracted from the DEF dumped from final DRC clean layout using StarRC, a signoff extraction tool.[2] That provide us the SPEF (Standard Parasitic Exchange Format), which is fed into primeTimeSI (a static timing analyzer tool) along with the netlist for timing analysis. We used an uncertainty of 200ps for setup, and 50ps for hold as recommended by Global Foundries. The design is also analyzed for design rule violations, such as the maximum transition, capacitance, and fanout. The few violations we had after the initial STA were fixed using ECO.

Variation in fabrication parameters can cause the cells to be slow, typical or fast. These are referred as process corners. A device manufactured in slow process corner would operate slow while one fabricated in fast process corner would operate fast. In addition device timing characteristics also varies with change in operating temperature and supply voltage. Gate delay increases with, increase in temperature and decrease in operating voltage. Similarly RC (Resistance and Capacitance) of the interconnects are also modelled for worst, typ and best corners. Hence designer need to analyze timing in all these possible combinations (scenarios) of process, voltage and temperature and RC corners to ensure the performance of the chip. Timing libraries for these possible operating voltages and temperatures are obtained as a part of standard cell library kit. SignOff extraction generates SPEF for each of the RC corners. Table 11 list various scenarios for which setup and hold timing were checked before tapeout.

4.2.2 Physical Verification

`Physical verification (PV)` is the process of verifying the final layout against foundry manufacturing rules. It involves Design Rule Check (DRC), Layout Versus Schematic (LVS) and Antenna checks. A clean DRC layout from PnR with no shorts, spacing violation or opens still need to go through signoff PV in order to to be ready for tapeout. Layout information dumped in the GDSII format from the final post-route optimized layout is read by the layout tool (Virtuoso) along with the GDSII files for all the primitives used in

[2] We scaled the resistance and capacitance by 5% in the PnR tool in order to fix a slight the miscorrelation that we observed between PnR and STA timing results.

Table 11: Timing scenarios

Process (P)	**Voltage** (V)	**Temp.** ($°C$)	**RC**	**Timing checked**
Slow	1.08	125	Worst	Setup, hold
Slow	1.08	125	RC best	Setup, hold
Typical	1.2	25	RC typical	Setup, hold
Fast	1.32	125	RC worst	hold
Fast	1.32	125	RC best	hold
Fast	1.32	-40	RC worst	hold
Fast	1.32	-40	RC best	hold

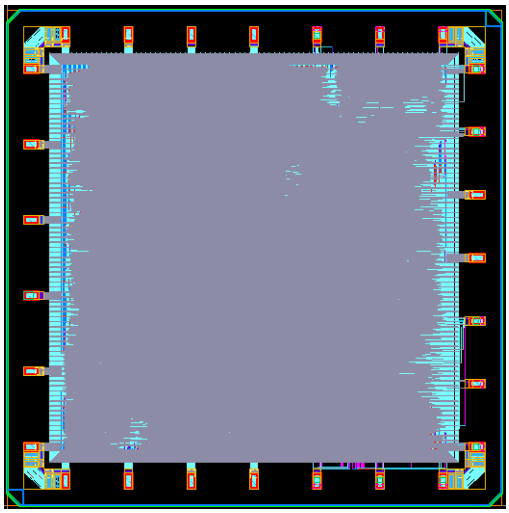

Figure 13: Chip GDS view from Virtuoso.

the design. This process is called streamIn. It merges the information from the primitive GDS to the design GDS.

The next step is to place a seal ring around the design as per foundry guidelines. A streamOut process from Virtuoso now dumps a complete design GDS. We take this GDS through an additional process to insert dummy metal/poly fills to meet the foundry metal/poly density requirements. This process is done using tool Calibre with the fill insertion script provided by MOSIS.

The fill GDS obtained from Calibre is merged with the design GDS in virtuoso and a final design GDS ready for DRC/LVS analysis is written. We used Cadence PVS tool to run through DRC, LVS and Antenna checks. Any violations noticed were fixed and the checks rerun to ensure a clean GDS for the tapeout. Figure 13 shows an image of design layout from Virtuoso after the streamIn process and addition of seal ring. Thin lines running around the chip boundary with diagonal routing around corners is the seal ring.

4.2.3 Rail Analysis

Analysis of voltage drop and ground bounce in its static and dynamic operations is termed `rail analysis`. If the drop is beyond certain limit, it affects the performance of the layout. The higher the voltage drop (IR), the higher the delay of gates, causing timing failure depending on the magnitude of the drop. Any IR violations observed in the analysis has to be fixed in the implementation tool and re-analyzed. Static drops are usually fixed by strengthening power structure. Dynamic drop fixes may include downsizing of high

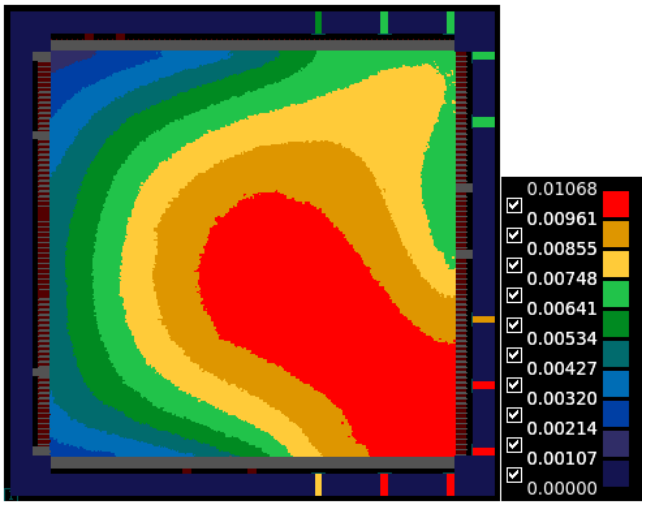

Figure 14: Chip static rail drop. The red region has the most static drop and the dark blue the least.

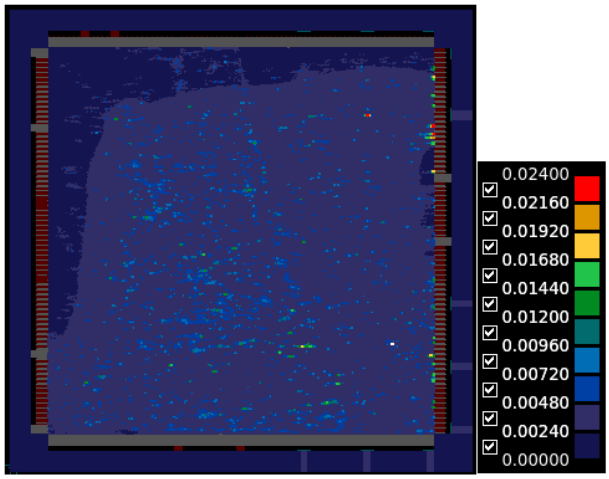

Figure 15: Chip dynamic rail drop.

driving strength cells or placing gates away from them and addition of DCAP (Decoupling cap) fillers.

We have used the Synopsys primeRail for rail analysis. The milkyway database and signal parasitics are imported to the tool to perform the analysis. Because we had a robust power structure, there was no static or dynamic violation. The worst static and dynamic drop was 10.6 mV and 24 mV, respectively. Figure 14 shows the static effective rail voltage drop. Legends shows color corresponding to the drop values. Most of our power pads are situated to the left of the chip and hence the highest static drop is seen to the right side (red in the image). Dynamic maps are shown in Figure 15. Dynamic drop we observed are specific to some gates and are seen as red spots in the image.

5 Post-Silicon Validation

Once the chip comes back from the foundry, one needs to verify if it meets all the specification that it is designed for. This process is known as **`post-silicon validation`**.

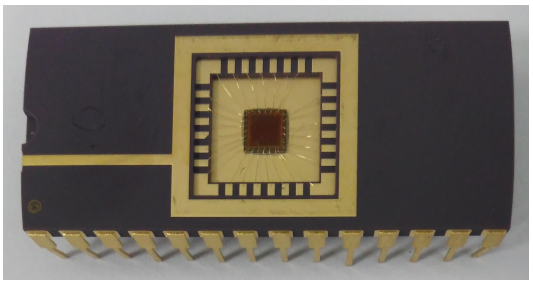

Figure 16: CoPHEE die in DIP-28 package.

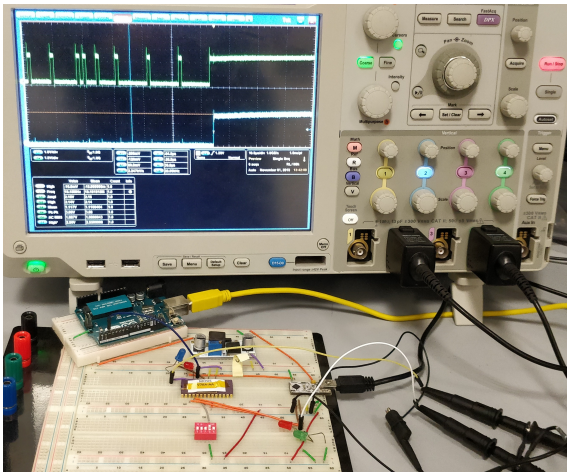

Figure 17: Photo of CoPHEE experimental setup.

In an industry standard chip design flow, where there is a huge volume of chip manufactured, chips are also tested for manufacturing defects. To assist this test, separate testing circuits called Design For Testability (DFT) are inserted during design phase. Identifying such defects, if any, helps fixing the fabrication flow and, thus, improve the yield (less number of chip with defect). In addition, it helps to ship only working chips to customers. We have skipped this step of testing for manufacturing defects as our devices are not produced as part of a high volume manufacturing process, but in a controlled single wafer manufacturing process. Thus, the chance of having such defects is very low. Moreover, one needs high-end costly testers for such testing.

Figure 16 shows the CoPHEE chip received from fabrication. CoPHEE was packaged in a 28 pin DIP, and was connected to a breadboard for silicon bring up and testing. For interfacing with a host computer, we used a UMFT230XA development board that features an FTDI chip for USB-to-UART conversion. The UMFT230XA board can provide a 3.3V supply for the IO pad of CoPHEE, as well as a clock output (used as the clock source of the chip). Moreover, the required 1.2V supply was generated using a DC-DC adjustable step-down module that converts the 5V source of the UMFT230XA board. In addition, an Arduino was responsible for receiving interrupt signals from CoPHEE and for transmitting these events to the host computer. Our post-silicon validation setup is shown in Fig. 17.

Our post-silicon validation confirmed that the fabricated chip is fully functional, with a discovered bug in the "debug read" path that reads random numbers and tests their randomness. Specifically, there was a hard-coded bit-width value in the read path of the configuration registers, which prevents us from reading the debug register (last register in the path). Interestingly, this bug escaped our FPGA-based validation, as the latter was performed on the scaled-down version where the 256 bit-width was incorrectly hard-coded.

References

[EHK+03] Michael Epstein, Laszlo Hars, Raymond Krasinski, Martin Rosner, and Hao Zheng. Design and implementation of a true random number generator based on digital circuit artifacts. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 152–165. Springer, 2003.

[Lim] ARM Limited. Amba 3 ahb-lite protocol specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033a/index.html(accessed: 03.27.2018).

[MTM16] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation. *IEEE Transactions on Information Forensics and Security*, 11(9):2123–2138, 2016.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.

[NAC+19] M. Nabeel, M. Ashraf, E. Chielle, N.G. Tsoutsos, and M. Maniatakos. Cophee: Co-processor for partially homomorphic encrypted execution. In *IEEE Hardware-Oriented Security and Trust (HOST)*, 2019.