Kyiv School of Economics

School of Engineering

Micro- and Nanoelectronics Programme

Master's Thesis

Implementation RTL-to-GDSII flow for a Croc-based SoC with a hardware BBS pseudorandom generator using Cadence EDA tools

Andrii Rachek

Supervisor
KSE, Mykola Konovalenko, mykolakonovalenko@kse.org.ua

Submission date
20 December 2025

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date: Kyiv, 20.12.2025

Signature 

Content

Abstract 1

1 Introduction 2

1.1 Background: Pseudorandom Number Generation 2

1.2 Statistical testing techniques for PRNG validation 4

1.3 Cryptographically secure PRNGs 6

1.4 Target Hardware Platform: The Croc RISC-V Core 7

1.5 Problem Statement and Thesis Objectives 9

2 Analysis 10

2.1 Design Specifications & Requirements 10

2.2 BBS Algorithmic Adaptation 10

2.3 Target Croc architecture 13

2.4 OBI description 14

2.5 Hardware microarchitecture 18

2.5.1 Top-level 18

2.5.2 Multiplier 19

2.5.3 Modulo operation (MOD) 21

2.5.4 Main Finite State Machine 23

2.5.5 Register interface logic 26

2.6 Physical design strategy 27

2.6.1 SoC dimensions 27

2.6.2 Pin placement strategy 28

2.6.3 Macro placement strategy 31

2.6.4 Power planning strategy 32
3 Implementation 33
3.1 Integration to Croc 33
3.1.1 RTL project structure 34
3.1.2 Generate file list 36
3.2 BBS module simulation 37
3.2.1 Multiplier 37
3.2.2 Module Operation (MOD) 38
3.2.3 BBS functionality 38
3.3 Physical design flow setup 40
3.3.1 Configure design_config.tcl 41
3.3.2 Configure genus_config.tcl 43
3.3.3 Configure innovus_config.tcl 43
3.3.4 Configure mmmc_config.tcl 44
3.3.5 Configure timing constraints 45
3.4 Synthesis flow 45
3.5 Floorplanning implementation 46
3.5.1 Core dimension 47
3.5.2 Pin placement 50
3.5.3 Power grid placement 51
3.5.4 SRAM placement 56
3.5.5 SRAM power grid placement 58
3.5.6 Core power connection 61

4 Results Evaluation 67
4.1 NIST Statistical Test 67
4.2 Synthesis Results 71
4.3 CTS Results 73
4.4 Routing Results 75
4.4.1 Post-routing timings 76
4.4.2 CTS visualization 79
4.4.3 Occupied area visualization 80
4.4.4 Timing map 84
4.4.5 Route congestion heat map 86
4.4.6 Post-route DRC 87
4.4.7 Post-route power estimation 89
5 Conclusions 91
6 Bibliography 94
Appendix A. BBS generator control & Status Register (CSR) 97
Appendix B. Library on C 100
Appendix C. MATLAB script for NIST statistical tests 105
Appendix D. Configuration file mmmc_config.tcl content 107
Appendix E. SoC timing constraint file 109
Appendix F. Floorplan script file 112
Appendix G. Pin placement configuration file 120

List of Acronyms

BBS – Blum-Blum-Shub (pseudorandom generator)

CSR – Control and Status Register

CTS – Clock Tree Synthesis

DEF – Design Exchange Format

DFT – Discrete Fourier Transform

DRC – Design Rule Check

EDA – Electronic Design Automation

FSM – Finite State Machine

GCD – Greatest Common Divisor

GDSII – Graphic Data System II

GPIO – General-Purpose Input/Output

GUI – Graphical User Interface

IO / I/O – Input/Output

IOBOX – I/O box (floorplan region)

IOCELL – I/O cell

IR drop – Supply-voltage drop due to resistivity.

JTAG – Joint Test Action Group (IEEE 1149.1)

LEF – Library Exchange Format

LSB – Least Significant Bit

MSB – Most Significant Bit

NIST – National Institute of Standards and Technology

OBI – Open Bus Interface

PDK – Process Design Kit

PoC – Proof of Concept

PRNG – Pseudorandom Number Generator

RTL – Register-Transfer Level

SRAM – Static Random-Access Memory

STD – Standard cell (standard-cell library)

TCL / Tcl – Tool Command Language

UART – Universal Asynchronous Receiver-Transmitter

VLSI – Very-Large-Scale Integration

WNS – Worst Negative Slack

Abstract

This thesis presents the implementation of an RTL-to-GDSII flow for a Croc-based system-on-chip (SoC) augmented with a hardware Blum–Blum–Shub (BBS) pseudorandom number generator. The work aims to demonstrate the end-to-end integration of a custom security-related peripheral into an open RISC-V SoC, while preserving a reproducible physical design methodology using Cadence EDA tools. The Croc platform is used as a reference SoC, and the generator is integrated as a memory-mapped block in the user domain via the OpenBus Interface (OBI). The BBS hardware architecture includes control logic, a register interface, and modular arithmetic blocks required for iterative modular squaring and bit extraction. Functional correctness is validated through unit-level simulation of arithmetic modules and SoC-level tests driven by firmware that programs the peripheral and checks generated outputs against golden values. A preliminary statistical screening of the produced bitstream is performed using selected tests from NIST SP 800-22, including the Frequency (Monobit) and Discrete Fourier Transform (Spectral) tests. Physical implementation steps are documented, including floorplanning considerations, SRAM placement constraints, and power-distribution planning suitable for the target technology. The chosen 32-bit BBS configuration is intentionally treated as a proof of concept and is not intended to provide practical cryptographic strength because security depends on the modulus size. The results demonstrate a consistent hardware–software integration path on Croc and provide initial evidence that the implemented generator avoids obvious bias and strong periodic artifacts in the evaluated samples.

1 Introduction

1.1 Background: Pseudorandom Number Generation

In engineering practice, *randomness* is primarily a property of a **sequence** rather than of a single value. A sequence is typically regarded as random when its elements are statistically independent and drawn according to a specified probability distribution; the **uniform distribution** is the most common baseline, in which each value in a finite set is equally likely [1].

Random numbers are required in many application domains. They are central to simulation, where stochastic inputs are needed to model natural phenomena and real-world processes realistically, and to sampling, where randomly selected subsets provide insight when exhaustive enumeration is impractical [1]. Randomness is also widely used in *numerical analysis* and Monte Carlo methods, supporting software and hardware development by enabling randomized test stimuli and *randomized algorithms*, which can outperform deterministic methods in efficiency or robustness for certain classes of problems [1].

From an implementation perspective, random values are often represented as *random bits*, which can then be converted into (approximately) uniformly distributed integers by interpreting bitstrings as numbers and discarding out-of-range values when necessary [2]. This motivates a practical classification of generators into **true random** generators and **pseudorandom** generators, with pseudorandom generators further divided into those that satisfy cryptographic security requirements.

A *true random bit generator* (TRBG) produces a sequence of bits intended to be unbiased and statistically independent [2]. In practice, TRBGs rely on naturally occurring physical phenomena (e.g., thermal noise, oscillator instability, and flip-flop metastability). However, producing a bitstream that is free from bias and correlation is challenging, and the generator must be robust against external influence or

manipulation in cryptographic settings [2]. Consequently, TRBG outputs often require conditioning (e.g., de-skewing) to mitigate bias and correlation [2].

A *pseudorandom bit generator* (PRBG/PRNG) is a deterministic algorithm that expands a short, truly random input into a much longer output sequence that is intended to **appear** random. The input is called the *seed*, and the output is a pseudorandom sequence [2]. Deterministic generators are attractive because they are efficient and reproducible, and they can perform well in many applications when an appropriate method is selected [1].

Nevertheless, because PRNG outputs are generated algorithmically, they can exhibit structural weaknesses such as short cycles or other artifacts when poorly designed, motivating careful selection and evaluation [1].

Pseudorandom generators are commonly divided into two classes based on their suitability for security-critical use:

- Non-cryptographic PRNGs. These generators may be adequate for simulation and probabilistic algorithms and may pass standard statistical tests, but they are not designed to resist adversarial prediction. A classic example is the linear congruential generator (e.g., the Lehmer random number generator), which is widely used for simulation yet can be reconstructed from partial output and is therefore unsuitable for cryptographic purposes [2].
- Cryptographically secure PRNGs (CSPRNGs/CSPRBGs). For cryptographic applications, the key requirement is *unpredictability*: many cryptographic systems rely on generating values that an adversary cannot feasibly guess or efficiently search (e.g., secret keys, nonces, challenges) [2]. In complexity-theoretic terms, a CSPRNG is expected to produce outputs that are (i) statistically indistinguishable from truly random sequences for any efficient

distinguisher and (ii) unpredictable in the "next-bit" sense; additionally, the seed must be long enough to make exhaustive search infeasible [2].

This distinction highlights an important methodological point: statistical quality alone is insufficient to claim cryptographic security. Statistical testing is necessary for detecting many non-random artifacts, yet passing such tests does not, by itself, guarantee resistance to prediction or state recovery [2]. In subsequent sections, these definitions and security considerations provide the foundation for discussing how generator outputs are assessed and validated in accordance with the intended application domain [2].

1.2 Statistical testing techniques for PRNG validation

After defining RNG/PRNG classes and their intended applications, the following practical question is how to validate the output sequences. In practice, validation is commonly performed using statistical tests on a finite sample of the generator's output. Such tests are designed to detect specific deviations from the properties expected of an ideal random sequence; consequently, the outcome of each test is inherently *probabilistic*, not a mathematical proof of randomness [2], [5].

A key limitation is that statistical testing is not "complete": there are infinitely many conceivable tests, and passing a chosen finite suite **cannot certify** suitability (especially for cryptography). The NIST documentation, therefore, treats statistical testing as a first screening step and emphasizes that it cannot replace cryptanalytic analysis [5].

Test methodology.

Most widely used PRNG test suites are formulated within statistical hypothesis testing. The *null hypothesis* (H_0) typically states that the tested sequence is random, while the alternative hypothesis (H_a) states that it is non-random. A test statistic is

computed and converted into a *P-value* [5], which is compared against a chosen significance level α. If P-value $\geq \alpha$, H_0 is not rejected; otherwise, H_0 is rejected [5].

The significance level α corresponds to the probability of a *Type I error* (rejecting a truly random sequence), and typical cryptographic practice often uses values around 0.01.

Major groups of statistical tests.

Statistical techniques for evaluating PRNG output can be organized into two broad groups, ranging from basic "sanity checks" to tests targeting deeper structural regularities. This organization follows naturally from the earlier distinction between statistical quality and cryptographic security: statistical testing is necessary for detecting many defects, but it is not, by itself, a security proof [2], [5].

1. Time-domain distribution and dependence tests (local statistics). These tests detect deviations in counts and short-range structure that are unlikely in an ideal random sequence. Typical examples include frequency/monobit, serial (two-bit), poker (m-bit pattern counts), runs, and autocorrelation tests [2]. NIST includes closely related tests such as Frequency (Monobit), Frequency within a Block, Runs, Longest Run, Serial, Approximate Entropy, and Cumulative Sums [5].
2. Global structure, complexity, transform-based tests (long-range regularities). This group targets higher-order regularities that may not be visible in local counts, including periodic components, linear structure, and compressibility and/or algorithmic complexity. Examples include the Discrete Fourier Transform (Spectral) test, which detects periodic features via peak behavior in the frequency domain [5] and Maurer's universal statistical test, motivated by the principle that an ideal random sequence should not be significantly compressible [2]. In the NIST suite, this category

also covers tests such as Binary Matrix Rank, template matching, linear complexity, and the random excursions tests [5].

In summary, statistical testing provides a structured way to search for detectable non-randomness in PRNG output samples [2], and standardized suites (such as NIST's) provide practical procedures and interpretation guidance for cryptographic contexts [5]. Nevertheless, because failures can occur even for good generators (at the chosen α and because no finite suite is exhaustive, results must be interpreted cautiously and complemented by design-specific analysis.

1.3 Cryptographically secure PRNGs

A convenient taxonomy of widely cited CSPRNG families appears in the Handbook of Applied Cryptography (Chapter 5, "Notes and further references"), which lists and contextualizes several provable-security generators and general construction results [2]. The following constructions are explicitly referenced there:

1. Blum-Micali generator (security based on the discrete logarithm assumption).
2. RSA PRBG and the Micali-Schnorr PRBG (RSA-based PRBG and an efficiency-improved variant).
3. Blum-Blum-Shub (BBS), also known as the $(x^2 mod\ n)$ generator.
4. Modified-Rabin generator (a variant related to BBS).
5. Subset-sum based CSPRBG constructions (Impagliazzo and Naor).
6. Syndrome-decoding based CSPRBG construction (Fischer and Stern).

This list is not exhaustive; instead, it provides a set of canonical constructions and a bridge between concrete generators and general complexity-theoretic foundations [2].

In addition to its provable-security framing, the BBS $(x^2 mod\ n)$ family is attractive for pragmatic reasons: pseudorandom sequences are often used because they

are reproducible. Storing only a short seed is sufficient to reproduce a long sequence, avoiding the need to store the entire sequence explicitly [3].

BBS is presented as a CSPRNG under the assumption that integer factorization is intractable [2], [3]. The handbook [2] notes the historical strengthening of its security basis: Blum, Blum, and Shub originally relate the security of the generator to the quadratic residuosity problem, while Vazirani and Vazirani [4] establish a stronger result by proving cryptographic security under the weaker assumption that factoring is hard [2].

The principal parameter controlling the cryptographic security of BBS is the bitlength of the modulus $n = p \cdot q$. This follows directly from the fact that the security argument is conditioned on the intractability of factoring n [2]. Consequently, when n is small (e.g., a 32-bit modulus), factoring is computationally straightforward, and the security premise underlying BBS no longer applies; such a configuration can therefore be restored quickly by recovering p and q and then leveraging that knowledge to analyze or predict generator behavior.

Overall, BBS is a conceptually simple, well-studied CSPRNG whose security discussion cleanly connects to the hardness of factoring; this makes it particularly suitable for illustrating how security scales (or fails to scale) with modulus size in constrained implementations.

1.4 Target Hardware Platform: The Croc RISC-V Core

The System-on-Chip (SoC) Croc is a simple educational project. Due to its simplicity and modularity, it is easy to add new hardware blocks without starting from “scratch”.

Croc is divided into two main parts: *Croc* and *user* domains. Croc domain contains a minimal RISC-V MCU built around an IBEX-based CVE2 core, an OBI

crossbar, and two SRAM banks that are used for both MCU program and data storing [6]. Also, there is a simplified system for downloading firmware via JTAG.

This part of the design is fully functional and ready to use. Also, there are all the necessary scripts to build software for the MCU, run simulations in Verilator and Questa. Also, there is a need for Tcl scripts to create a hardware implementation of SoC using OpenRoad flow and open-source 130 nm IHP technology [6].

The IBEX-based CVE2 core, specifically the CV32E20 configuration, is a production-quality 32-bit RISC-V processor architecture designed for energy-efficient embedded control applications. Derived from the lowRISC Ibex core (formerly Zero-riscy), it features a compact two-stage in-order pipeline consisting of Instruction Fetch (IF) and Instruction Decode-Execute (ID/EX) stages. This design choice minimizes silicon area while delivering predictable timing performance. Architecturally, the core implements the RV32IMCZicsr instruction set, which includes the base integer ISA (I), standard integer multiplication and division (M), compressed instructions (C) for enhanced code density, and Control and Status Register (CSR) operations (Zicsr) necessary for the mandatory Machine Mode privilege level defined in the RISC-V specification. To optimize data throughput, the core utilizes a Harvard architecture with independent OBI ports for instruction and data memory, effectively decoupling memory accesses from instruction fetches.

These characteristics make the core particularly advantageous for implementing lightweight hardware accelerators with memory-mapped interfaces (AXI, OBI). Additionally, the streamlined architecture, which requires only one type of hard macro (SRAM), significantly reduces the complexity of migrating the design across different technology processes or EDA toolchains (Cadence, Synopsys).

1.5 Problem Statement and Thesis Objectives

Scope Limitation.

For this implementation, the data width is set to 32 bits. While cryptographic standards (such as NIST) recommend modulus sizes of 2048 bits or larger to ensure resistance against factorization attacks, a 32-bit width was selected for this Master's thesis to serve as a Proof of Concept. This allows for complete verification of the hardware architecture, bus integration (OBI), and physical design flow (floorplanning) within time constraints, without the complexity of implementing large-integer arithmetic units (e.g., the Toom-Cook algorithm).

2 Analysis

2.1 Design Specifications & Requirements

Specification for design:

- Key and seed width: 32 bits
- Target frequency: 50 MHz
- Key and seed provided by the user.

Croc chip:

- Using Cadence software for synthesis and physical design
- Use SG13G2 130 nm PDK
- Die area: 4.088 mm^2 (2022 um x 2022 um)

2.2 BBS Algorithmic Adaptation

The BBS algorithm consists of three distinct phases:

1. Setup (key generation)
2. Seeding.
3. Generation.

As defined in the specification, the user should provide keys and a seed to the generator.

Setup phase.

Before starting random-bit generation, must first be performed:

1. Choose two Blum prime numbers P and Q. A Number is Blum prime if:

$$(P \ mod \ 4) = 3, (Q \ mod \ 4) = 3$$

2. Calculate Blum integer M. It is just the multiplication of two distinct Blum primes:

$$M = P \cdot Q$$

Seeding phase.

The user should set up the first seed value. Start "seed" X_0 should be such that:

$$GCD(X_0, M) = 1,$$

Where: $X_0 \neq 1$ and $X_0 \neq 0$; GCD stands for "Greatest common divisor" (for example, using Euclidean algorithm).

The flow diagram for the setup and seeding phases is shown below.

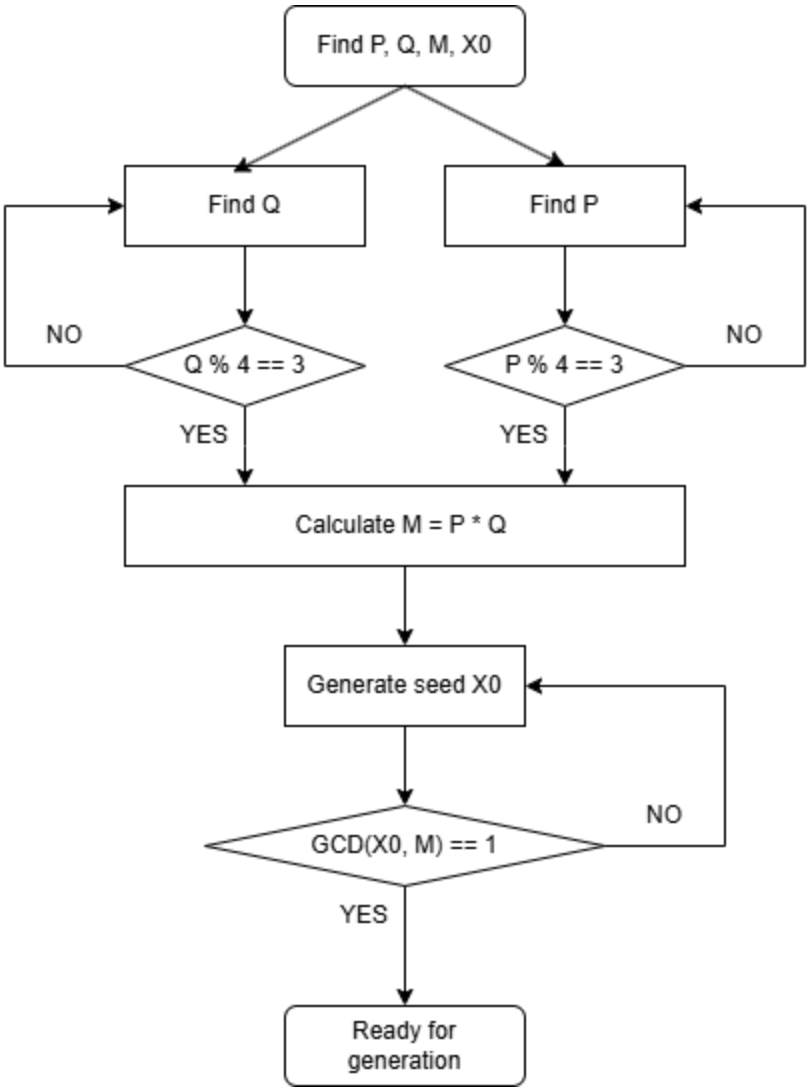


Figure 2.1 – BBS setup and seeding phases flow diagram

Generation phase.

1. Compute the next value of quadratic residue X_{next}:

$$X_{next} = (X_{current}^2 \bmod M)$$

Where: $X_{current} = X_0$ for the first run.

2. Extract the least significant bit.

3. Steps 1,2 are repeated 32 times to get a full 32-bit number.

Diagram flow for the generation phase is shown below. This is a general approach to get an N-bit random number.

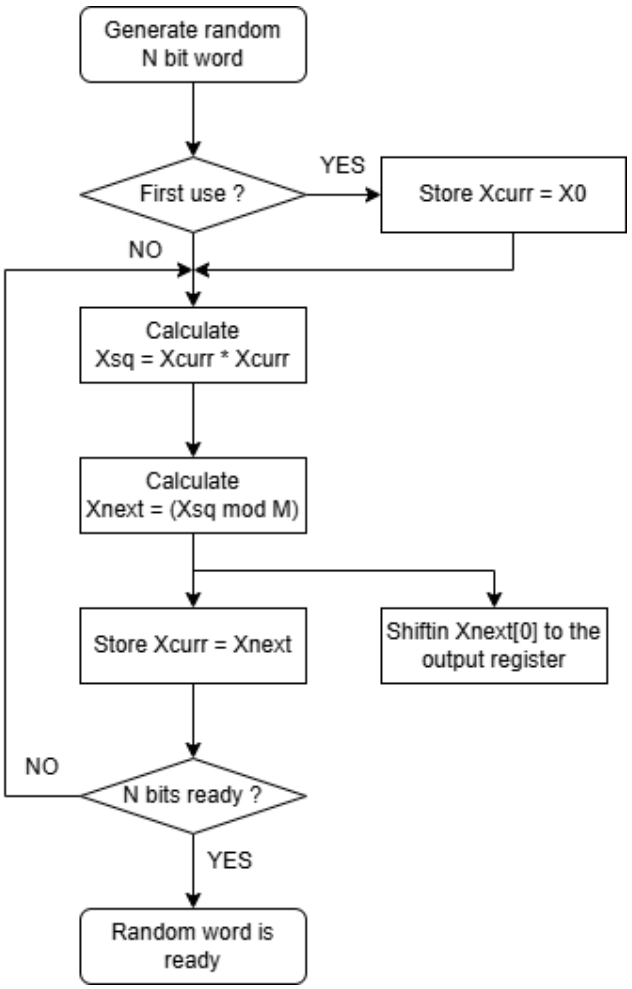


Figure 2.2 – Generation phase flow diagram

This flow diagram shows that every generated bit is shifted into the output serial register. After repeating N times, the resulting N-bit word can be read from this register.

2.3 Target Croc architecture

Croc is a simple educational SoC from ETH Zurich and the University of Bologna, suitable for students' projects.

Croc architecture with a place for the BBS generator is shown below.

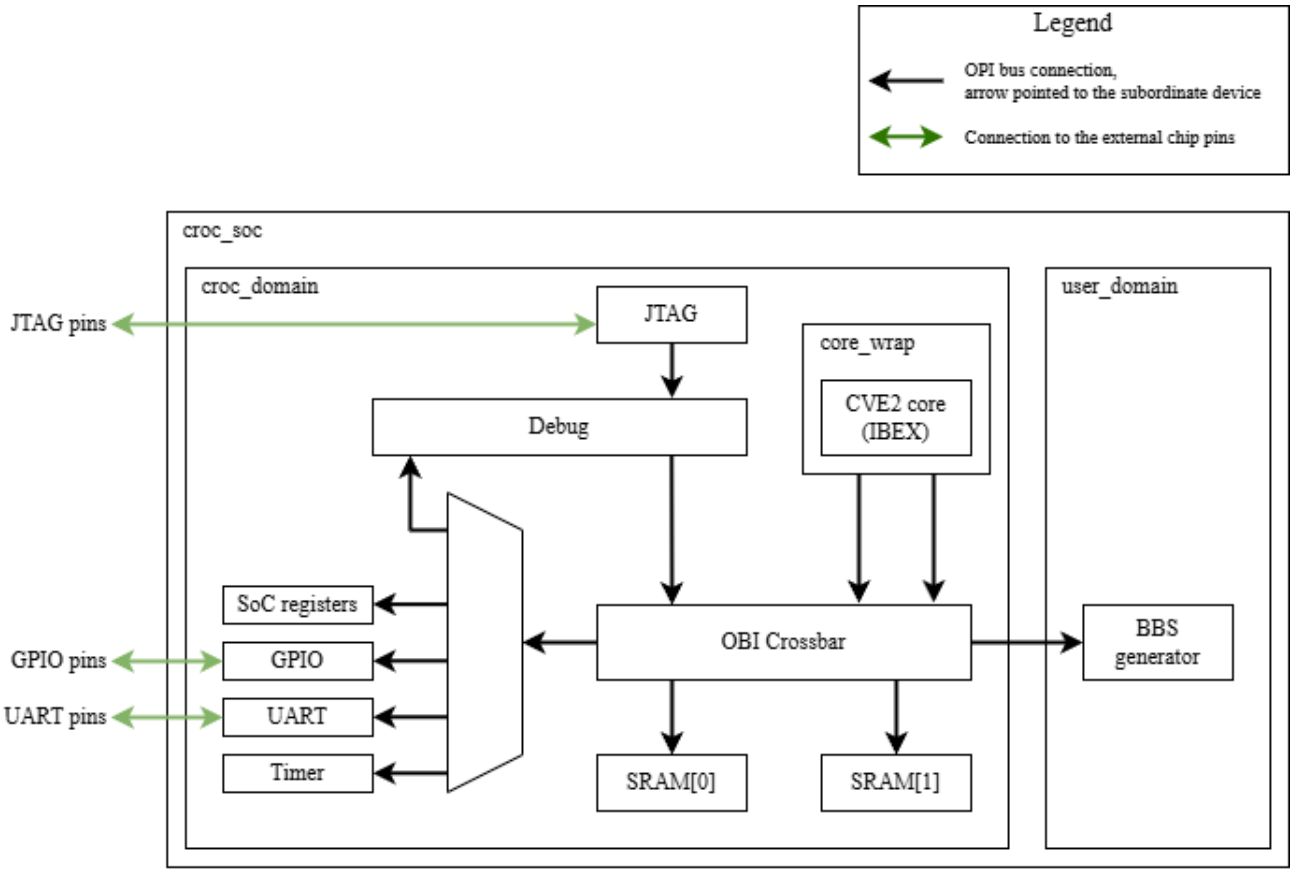


Figure 2.3 – Croc SoC architecture

This SoC contains several parts:

1. Croc domain, that contains CVE2 core (a fork of IBEX), two instances of SRAM, an OBI crossbar with several simple peripherals (UART, Timer, GPIO)
2. User domain, which contains user design, such as BBS generator.

All modules inside Croc domain are ready-to-use and do not require additional efforts to make the SoC work.

CVE2 core is an open-source 32-bit in-order RISC-V core. This is a fork of the lowRISC IBEX core, originally developed by ETH Zurich and the University of Bologna. This architecture uses a 2-stage pipeline and implements the RV32 ISA with a mandatory base instruction set (RV32_zicsr).

SRAM0 and SRAM1 have a total of 1024 32-bit words and are used to store both data and program for the core.

The only way to load a program into the core is to download it via the JTAG interface to the SRAM cores.

User domain shares the same clock domain as Croc domain. This means that no need to add any FIFO or 2-stage synchronizers to the user domain project.

The Croc project contains a default testbench for the whole project. It uses a compiled *.hex file to imitate uploading the core program via JTAG to the SRAM.

2.4 OBI description

RISC-V core and all parts communicate with each other using the OpenBus Interface (OBI). It consists of several main channels (ports):

1. A channel: manager is the source, subordinate is the destination
2. R channel: subordinate is source, manager is destination.

This is illustrated in the figure below.

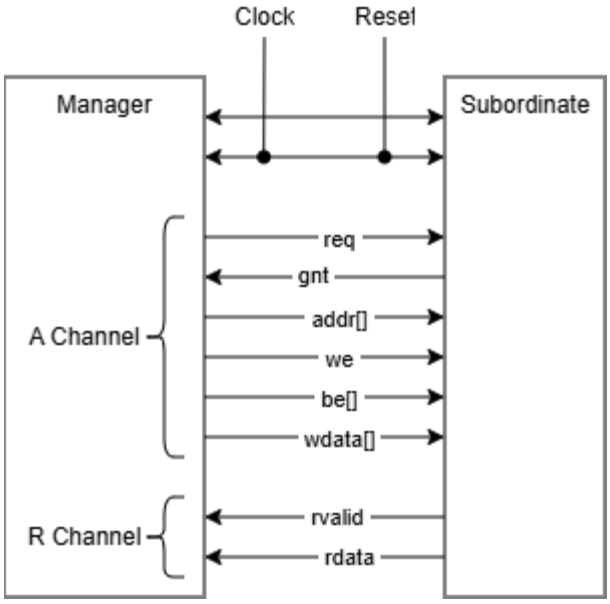


Figure 2.4 – OBI channels

All signals are divided into two channels:

- Address channel (“A Channel”)
- Response channel (“R Channel”)

Core CVE2 supports a minimal set of signals that are described in the table below.

Table 2.1 – OBI signal description

Name	Source	Destination	Description
A Channel signals			
req	Manager	Subordinate	Address transfer request
gnt	Subordinate	Manager	Grant. Ready to accept address transfer
addr[]	Manager	Subordinate	Address
we	Manager	Subordinate	Write (1) or Read (0) request
be[]	Manager	Subordinate	Byte enable
wdata[]	Manager	Subordinate	Write data
R Channel signals			
rvvalid	Subordinate	Manager	Response transfer request
rdata[]	Subordinate	Manager	Read data

All signals have an active level of ‘1’ unless otherwise specified.

Timing diagrams for read and write operations are shown below.

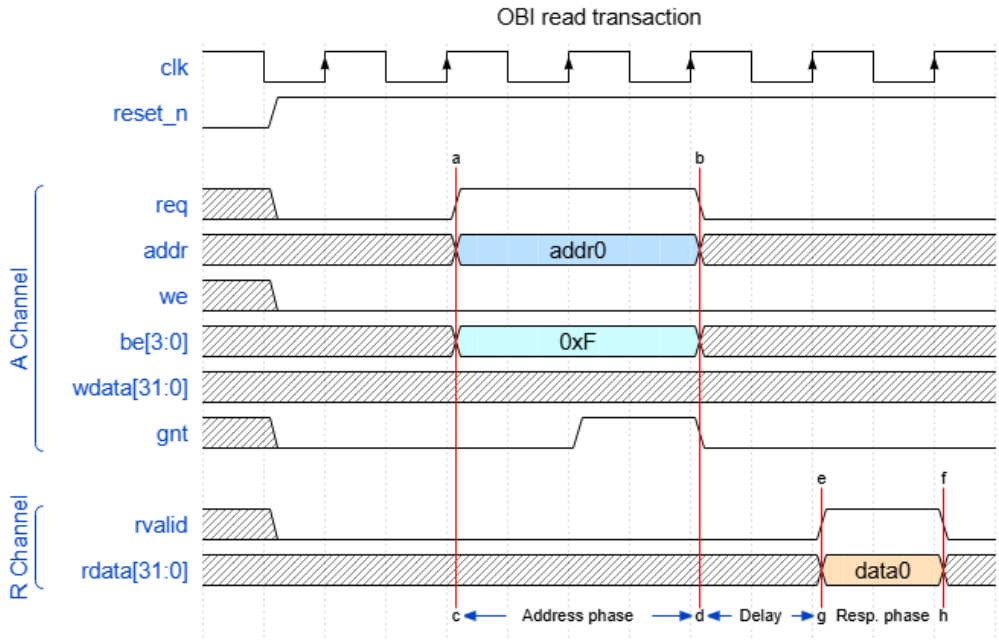


Figure 2.5 – OBI read transaction

Reading transaction starts when the Manager sets *req* signal with *we* set to ‘0’. Along with this, the Manager sets valid values on *addr* and *be*.

After receiving a read request, the Subordinate should accept it by setting *gnt* to ‘1’. By doing this, Subordinate granted that it is ready to process the Manager’s request.

In the next clock period or after some delay, Subordinate sets a valid value on the *rdata* bus and ‘1’ on *rvalid*.

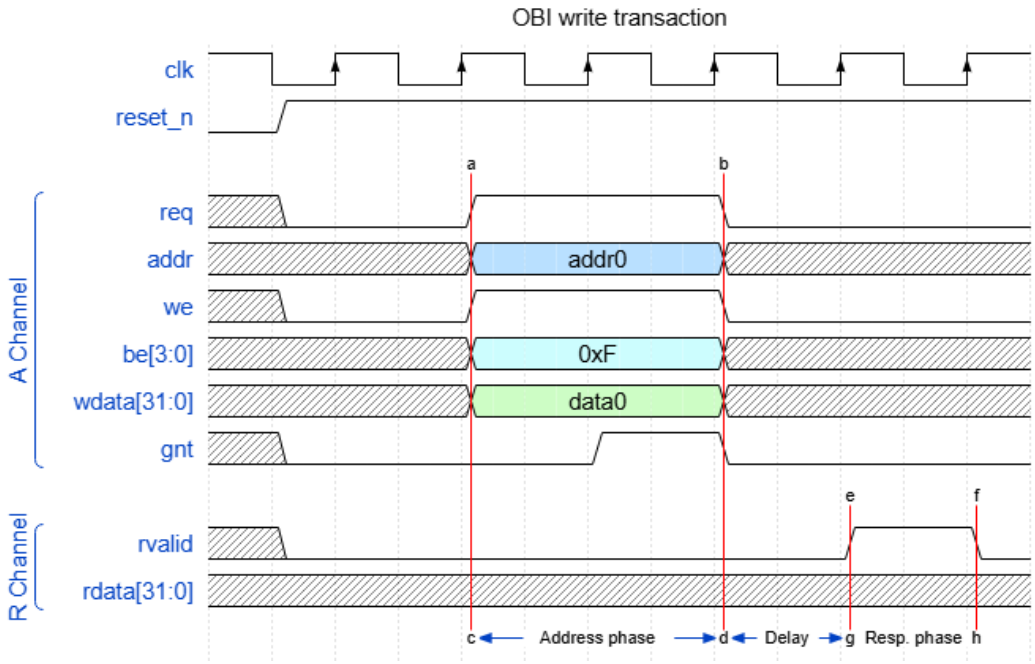


Figure 2.6 – OBI write transaction

Writing transaction starts when the Manager sets *req* signal with *we* set to '0'. Along with this, the Manager sets valid values on *addr*, *be* and wdata.

After receiving a read request, the Subordinate should accept it by setting *gnt* to '1'. By doing this, Subordinate granted that it is ready to process Manager's request.

In the next clock period or after some delay, Subordinate sets '1' on *rvalid*. Value on *rdata* can be any and will be ignored by the Manager.

2.5 Hardware microarchitecture

2.5.1 Top-level

The top-level block diagram for the BBS generator is shown below.

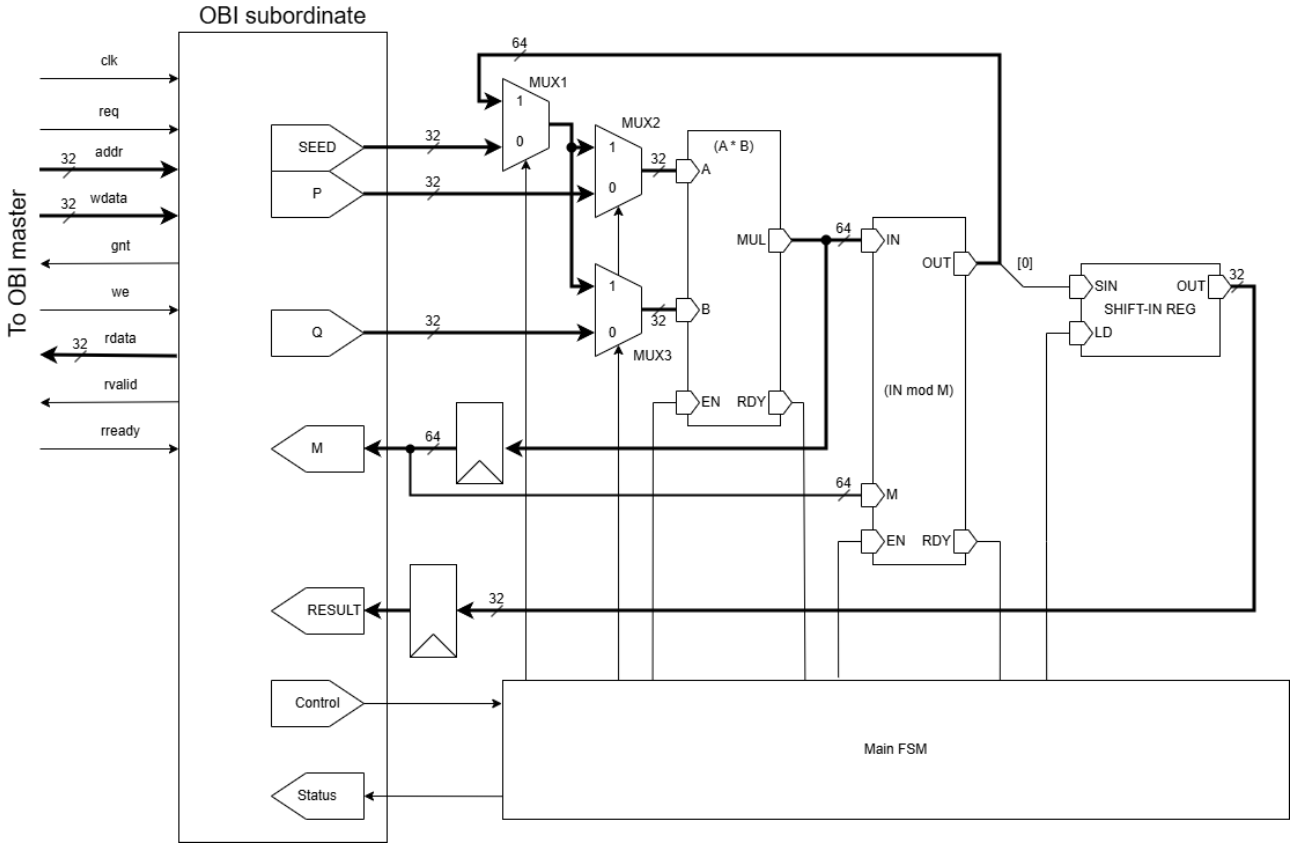


Figure 2.7 – Block diagram of BBS generator

It consists of several big modules:

1. OBI subordinate: provides MCU access to the module registers.
2. Multiplier: multiply two 32-bit unsigned words using the “shift-and-add” algorithm.
3. MOD calculator: makes MOD operations with two 64-bit values.
4. Shift-in register: shifts in the LSB bit from X_{N+1}.
5. Main FSM: controls all data flow, indicates ready signal for user via OBI register

2.5.2 Multiplier

Multiplication is implemented using the shift-and-add algorithm. The block diagram is shown below.

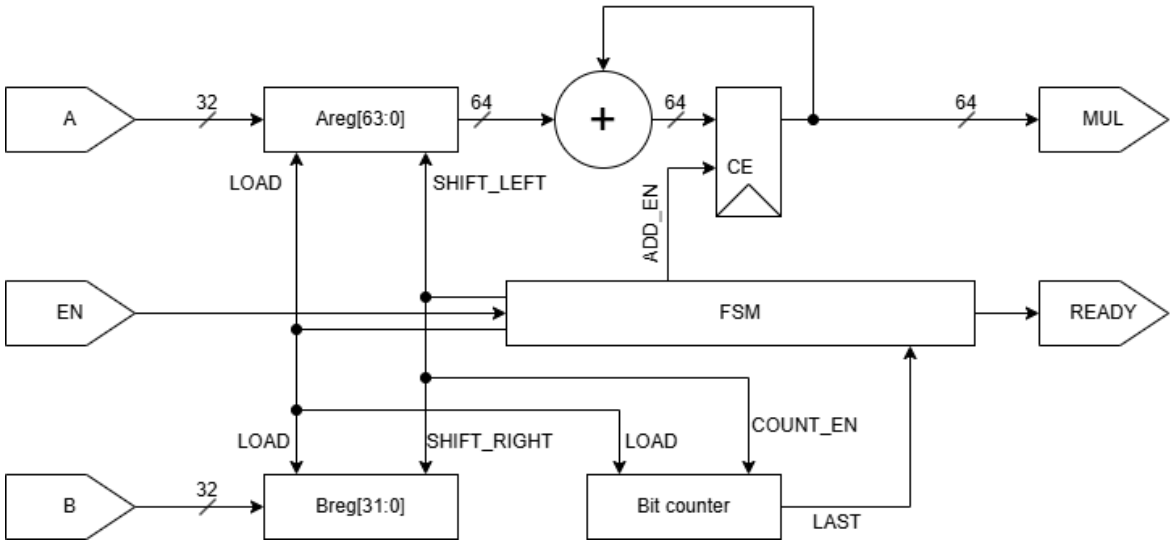


Figure 2.8 – Multiplier 32-bit x 32-bit block diagram

The algorithm “shift-and-add” was chosen because:

- reduce possible combination delay;
- reduce area;
- there is no restriction on calculation time.

This method is structurally equivalent to the manual "long multiplication" process performed in binary.

Mathematically, the product $P = A \cdot B$ is computed by iterating through the bits of the multiplier B, denoted as b_i (where $0 \leq i < 32$). The operation is defined by the summation:

$$P = \sum_{i=0}^{31} b_i \cdot \left(A \cdot 2^i\right)$$

The algorithm flow diagram is shown in the figure below.

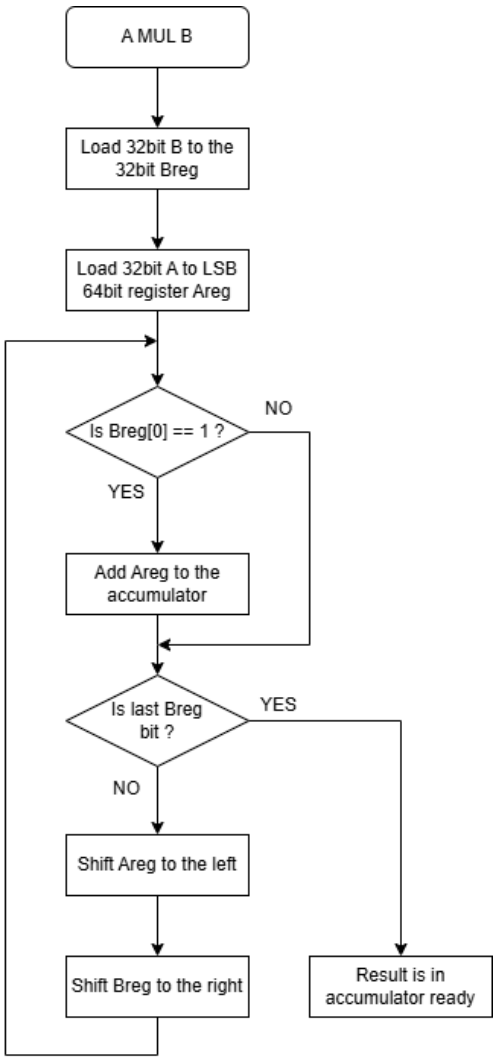


Figure 2.9 – “Shift-and-add” algorithm

According to this algorithm:

1. Multiplier loads two 32-bit values into two shift registers and resets the accumulator.
2. Checks LSB bit of register where saved B value: if it is “1” then add value from A register to the accumulator.
3. Shifts A register to the left, B register to the right.
4. Repeats stages 3,4 32 times.
5. At the end, in the accumulator will be the result of multiplying A and B values

2.5.3 Modulo operation (MOD)

The modulo operation returns the remainder or signed remainder of a division, after one number is divided by another. The block diagram is shown below.

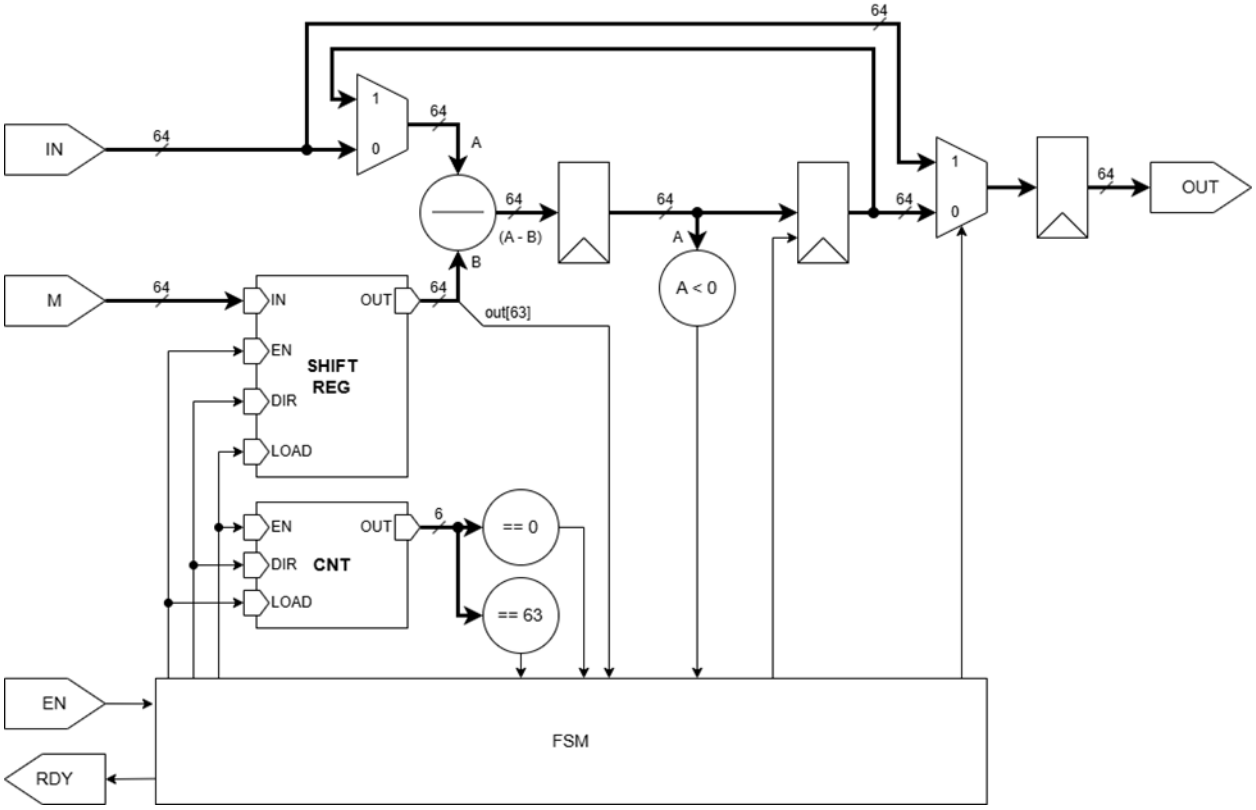


Figure 2.10 – Modulo calculator block schematic

The algorithm is shown below. It implements a “fast” modulo operation that allows calculating the result very fast if M is much smaller than the IN value:

- Shift M to the left until the first “1” is in MSB (this is equivalent to multiplying $M_N = M \cdot 2^N$, where N is the number of 1-bit shifting to the left)
- Try to subtract $IN - M_N$. If the result is negative, shift M_N to the right for one bit and repeat subtraction.
- Repeat subtraction and shifting until shifted M gets the same as before.
- Return the result of the accumulated subtraction operation.

The algorithm flow block schematic is shown below.

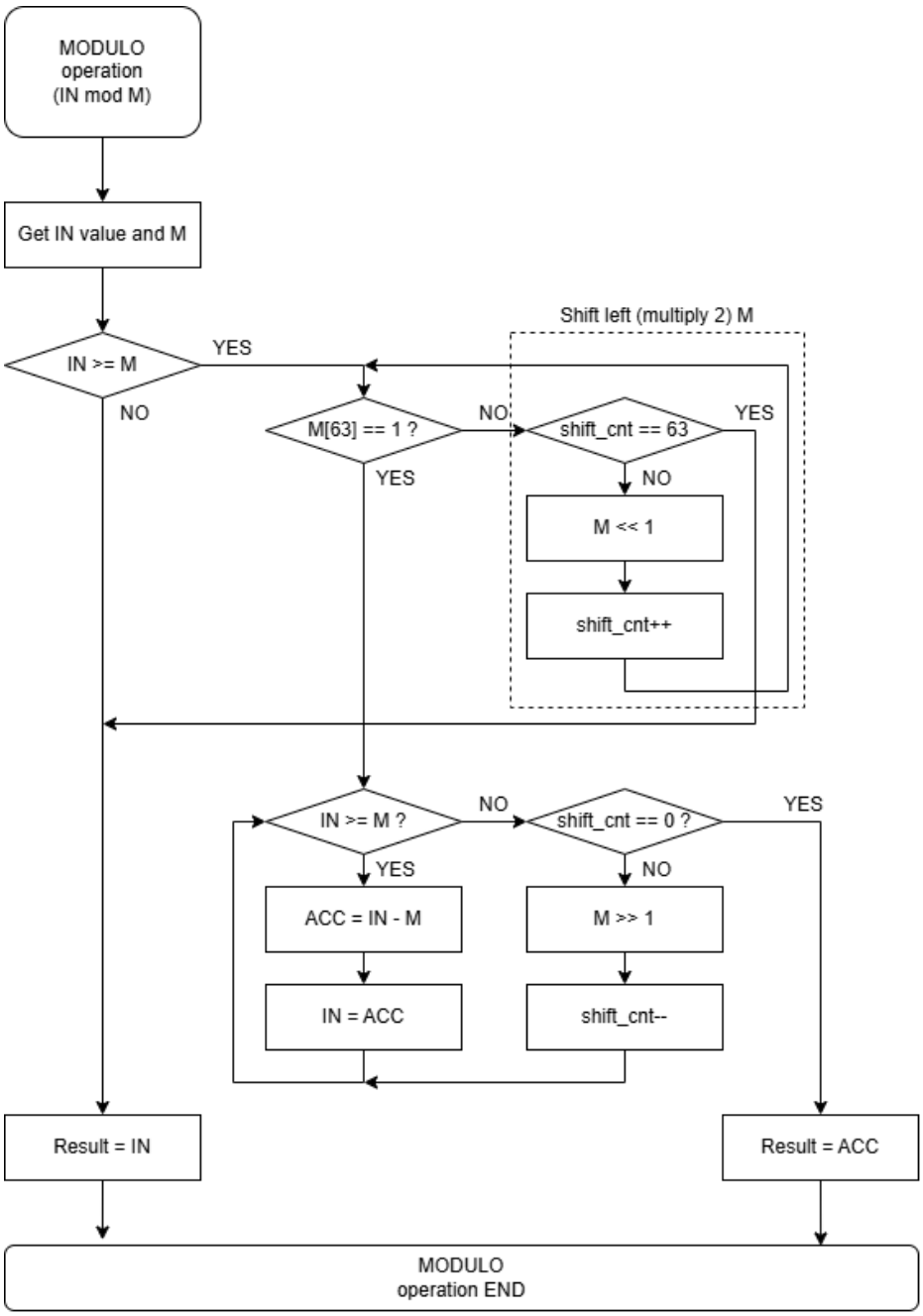


Figure 2.11 – MOD calculator block schematic

2.5.4 Main Finite State Machine

The main FSM receives control signals from the core via OBI registers and runs all submodules. Four modes should be implemented in FSM:

- Mode 1: requires P, Q, and SEED values to start calculation.
- Mode 2: use previously calculated M value and new SEED value.
- Mode 3: use previously calculated M value and last X_{NEXT} value from previous run (continue to generate).
- Mode 4: use new values of P, Q, SEED to restart generation.

To achieve these modes, some OBI registers should be defined:

- Input registers for P, Q, SEED values.
- Control register that contains bits for:
  - START: starts generation one 32-bit word;
  - KEEP_M: if set to “1”, do not recalculate M value from P, Q;
  - USE_XNEXT: if set to “1”, ignore the SEED register value and use the previous calculation result.
- Status register that contains bits for:
  - M_VALID: sets by FSM to “1” if M value is ready;
  - RESULT_VALID: sets by FSM to “1” if the generated value is ready.
- Output registers for M and the generated value.

Detailed descriptions of all described modes are below.

Mode 1 is used after a reset. To run the module in this mode, perform the following steps:

- Write desired values for P, Q, and SEED registers.
- Write “1” to the bit START in the control register (other bits can be in “0” state).
- When the M value is ready, the status bit M_VALID will be set to 1.
- Wait until bit RESULT_VALID in status will be set to “1”.

- A valid value can be read from the result register.

Mode 2 is used when P and Q are the same, but a new SEED value is needed. Can be run after Mode 1 (bit M_VALID should be already set to “1”):

- Update SEED register with new value.
- Set to “1” bits START, KEEP_M in the control register.
- Wait until bit RESULT_VALID in the status register will be set to “1”.
- A valid value can be read from the result register.

Mode 3 can be used if needed to proceed with the calculation with the same P, Q, and XNEXT; this is the default continue mode:

- Set to “1” bits START, KEEP_M, USE_XNEXT in the control register;
- Wait until bit RESULT_VALID in the status register will be set to “1”.
- A valid value can be read from the result register.

Mode 4 is used to restart generation with new parameters of P, Q, and SEED:

- Update registers P, Q, SEED with new values.
- Set to “1” bit START in the control register.
- Wait until bit RESULT_VALID in the status register will be set to “1”.
- A valid value can be read from the result register.

The algorithm flow diagram is shown below.

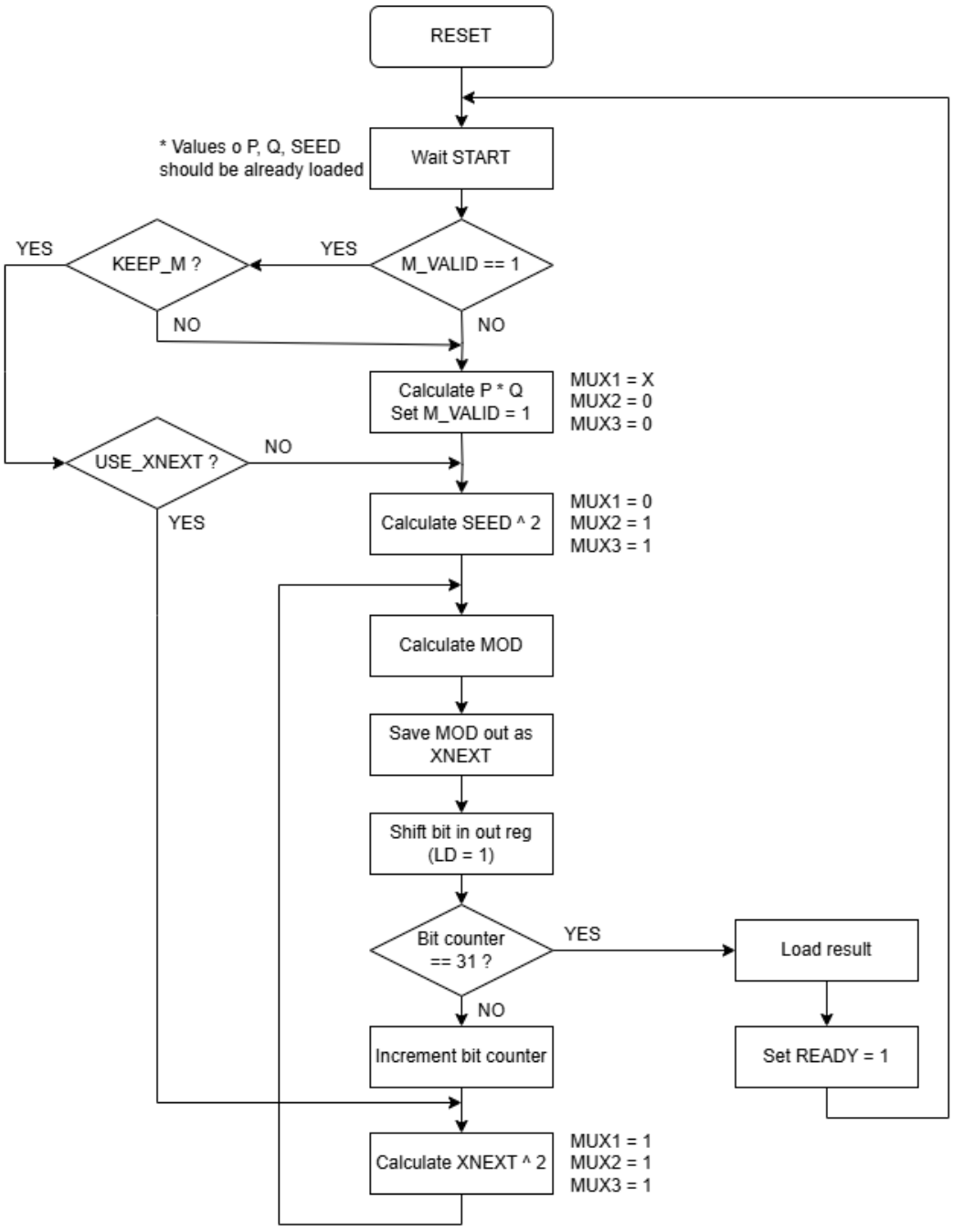


Figure 2.12 – Main FSM flow diagram

2.5.5 Register interface logic

The module block diagram that is responsible for communication with the core via OBI is shown below. It is Subordinate in OBI terminology.

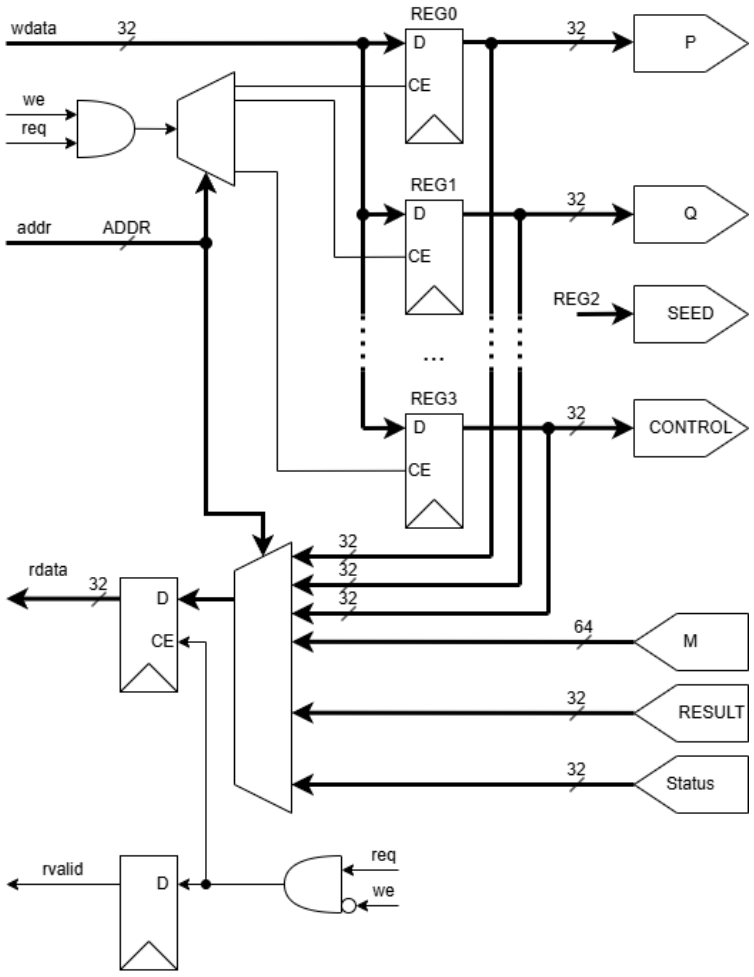


Figure 2.13 – OBI Subordinate block diagram

This module receives control signals from the core (Manager) and processes read and write requests.

It granted core access to read the values of the M, RESULT, and STATUS registers.

Also, this block allows the core to write values for P, Q, SEED, and CONTROL.

2.6 Physical design strategy

In this chapter, expectations from physical design, such as die dimensions, pin placement, and hard macro placement, are described.

2.6.1 SoC dimensions

Main SoC dimensions are shown in the table below.

Table 2.2 – SoC dimensions expectation

Parameter	Value
Die size WxH, um	2022 x 2022
IO box size, um	36
Pad dimensions WxH, um	60x310
Core size, WxH, μm	1330
Core area, mm^2	~1.77
Die area, mm^2	~4.09

In the table indicated:

- Die size: includes the core, IO ring, and IO box;
- IO ring: place where IO cells are placed, on each side of the die;
- IO box: distance between the IO ring and the core, need for power ring placement;
- Core size: area where STD cells will be placed.

All these physical design components are shown in the figure below. The blue color is for the Core area, the green color for the IO box, and the orange color for the IO ring.

For chosen dimensions, there are 1402 μm on each side to place IO cells. This is enough to place 16 IO cells on each side.

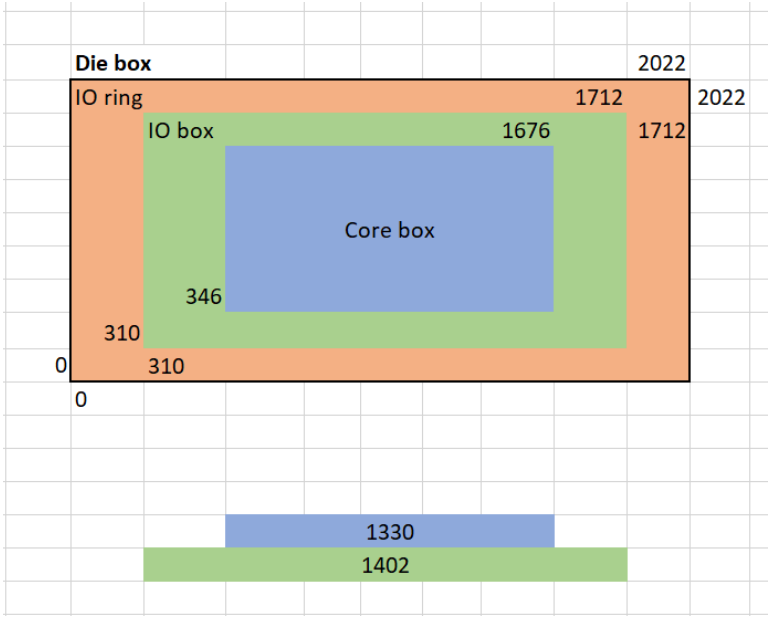


Figure 2.14 – SoC floorplan visualization

2.6.2 Pin placement strategy

According to the top file “rtl/croc_chip.sv”, Croc SoC has:

- 32 General Purpose Input-Outputs (GPIOs);
- JTAG pins;
- 2 pins for UART;
- 9 auxiliary pins (clock input, reset input, etc.);
- 16 for power supply connection.

A more detailed description is provided in the tables below.

Table 2.3 – SoC power pins

Name	Quantity	Description
pad_vdd0… pad_vdd3	4	Connected to the core +1.2V
pad_vss0… pad_vss3	4	Connected to the core ground
pad_vddio0… pad_vddio3	4	Connected to the +3.3V IO structures
pad_vssio0… pad_vssio3	4	Connected to the ground IO structures
Total	16	

Table 2.4 – Croc SoC digital pins description

Name	Direction	Quantity	Description
clk_i	Input	1	Core clock input (20 MHz)
rst_ni	Input	1	Global reset, active ‘0’
ref_clk_i	Input	1	Reference clock for timer (32.768 kHz)
jtag_tclk_i	Input	1	JTAG interface
jtag_trst_ni	Input	1	
jtag_tms_i	Input	1	
jtag_tdi_i	Input	1	
jtag_tdo_o	Output	1	
uart_rx_i	Input	1	UART interface
uart_tx_o	Output	1	
fetch_en_i	Input	1	Fetch enable input
status_o	Output	1	Core busy status
gpio0_io… gpio31_io	In/Out	32	GPIO
unused0_o… unused3_o	-	4	Unused chip pins
Total		48	

The design includes 16 power pins and 48 digital pins. To achieve a uniform distribution of power pins, they may be partitioned into eight VDD/VSS pairs: four for the core supply (VDD/VSS) and four for the I/O supply (VDDIO/VSSIO). It is

convenient to place these pins on each side of the die, with each side accommodating one core supply pair and one I/O supply pair.

The remaining 48 digital I/O pins may be distributed across all four sides of the die. These pins are not subject to additional placement constraints and may be grouped by function. For example, the JTAG interface and other auxiliary pins may be placed on one side, while the 32 GPIO pins, together with four reserved (unused) pins, may be allocated across the other three sides.

Proposal pin placement is shown in the figure below.

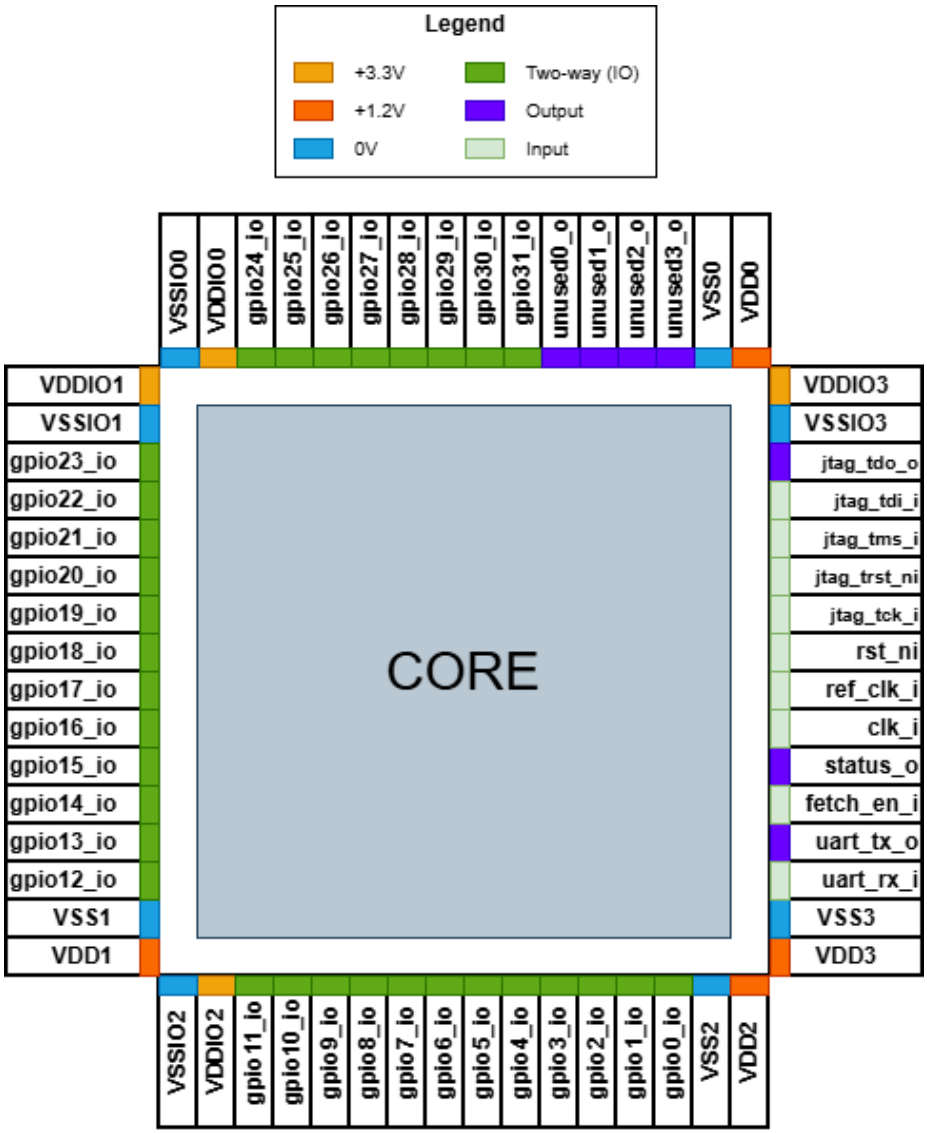


Figure 2.15 – Pins placement proposal

2.6.3 Macro placement strategy

The Corc SoC uses only one type of hard macro – two SRAM blocks. In this context, an appropriate placement strategy is to cluster the SRAM blocks and locate them adjacent to one side of the core area.

It should also be noted that, in the target technology, the uppermost routing layer (TopMetal2) has a preferred horizontal routing direction. This constraint informs the orientation of the core power stripes. To minimize IR drop, the design should maximize the number of TopMetal2 power stripes traversing the SRAM macros. Accordingly, rotating the SRAM blocks by 90° (R90) is advantageous; the SRAM LEF definition permits this rotation.

A proposed SRAM placement is illustrated in the figure below.

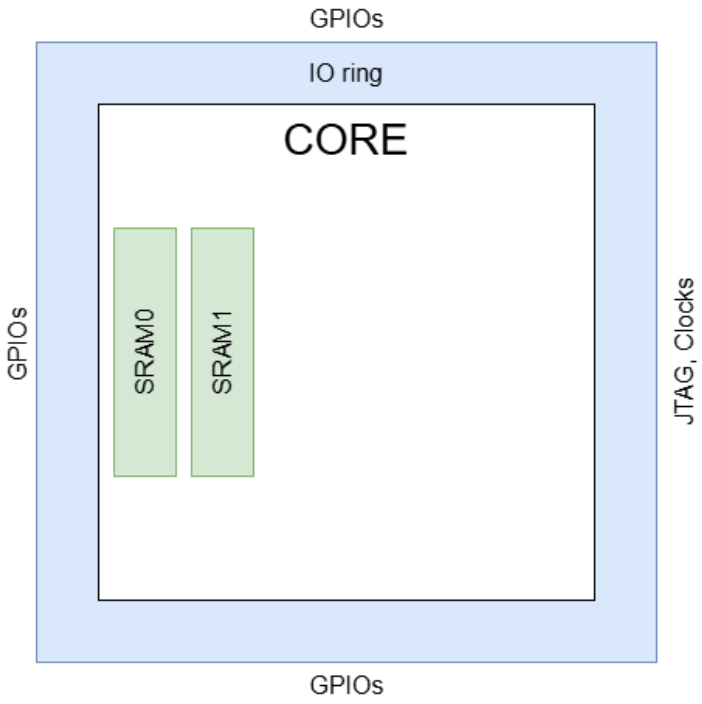


Figure 2.16 – Proposal SRAM placement

The left side of the die was selected to avoid obstructing the JTAG and clock pins.

2.6.4 Power planning strategy

Standard power-distribution techniques can be applied to the Croc SoC, including a single-core ring with a common ground rail (VSS) and a 1.2 V core supply rail (CORE_VDD). All ground pins (VSSIO and VSS) are tied to the VSS ring, whereas only the VDD pins are connected to the CORE_VDD ring.

The VDDIO pins are shorted together via the built-in connectivity provided by the pin cells, filler cells, and corner cells.

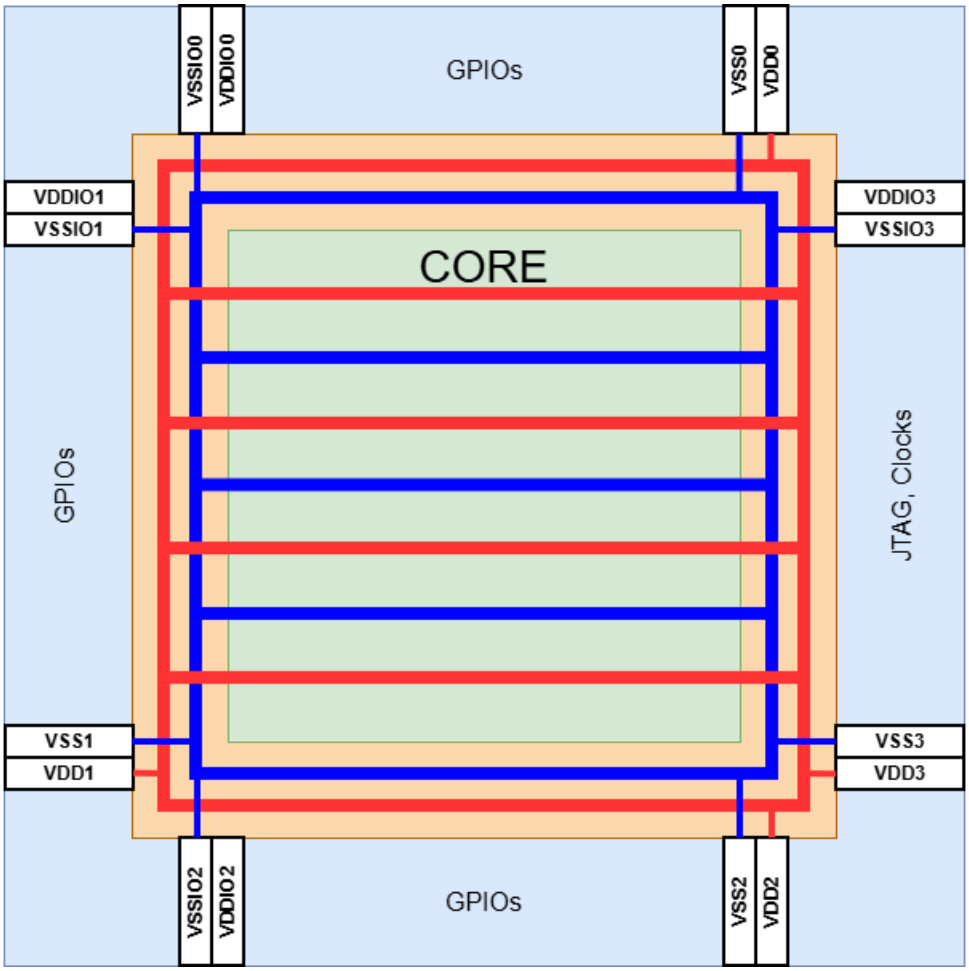


Figure 2.17 – Croc SoC power planning

3 Implementation

The implemented module was named “bbs32,” and this name will be used in the following description.

3.1 Integration to Croc

Module “bbs32” is inside the user domain. It is connected to the user subordinate interface.

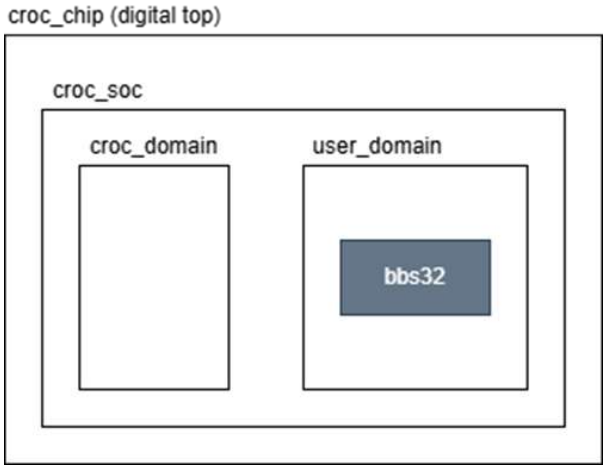


Figure 3.1 – BBS generator inside Croc project

Its base address is configured in “croc_pkg.sv” package as: UserBaseAddr = 32'h2000_0000.

```
rtl > croc_pkg.sv
11  package croc_pkg;
55    ////////////////////////////////////
56    // Croc Main interconnect addressing ///
57    ////////////////////////////////////
58
59    localparam bit [31:0]   PeriphBaseAddr    = 32'h0000_0000;
60    localparam bit [31:0]   PeriphAddrRange   = 32'h1000_0000;
61
62    localparam bit [31:0]   SramBaseAddr      = 32'h1000_0000;
63    localparam int unsigned NumSramBanks      = 32'd2;
64    localparam int unsigned SramBankNumWords  = 512;
65    localparam int unsigned SramBankAddrWidth = cf_math_pkg::idx_width(SramBankNumWords);
66    localparam int unsigned SramAddrRange     = NumSramBanks*SramBankNumWords*4;
67
68    localparam bit [31:0]   UserBaseAddr      = 32'h2000_0000;      Philippe Sauter, 7 mon
69    localparam bit [31:0]   UserAddrRange     = 32'h6000_0000;
```

Figure 3.2 – Module ‘bbs32’ base address configuration

3.1.1 RTL project structure

Project files added to the folder “rtl/user_domain”:

- Git directory
    - rtl
        - user_domain
            - bbs32.sv
            - bbs32_reg_pkg.sv
            - bbs32_reg_top.sv
            - bbs32_top.sv
            - mod64.sv
            - mul32.sv

Also, need to modify files:

- rtl/user_domain.sv: contains instantiation of “bbs32_top.sv”
- rtl/user_pkg.sv: contains “bbs32” module address, offset, and special structure for address map rule.

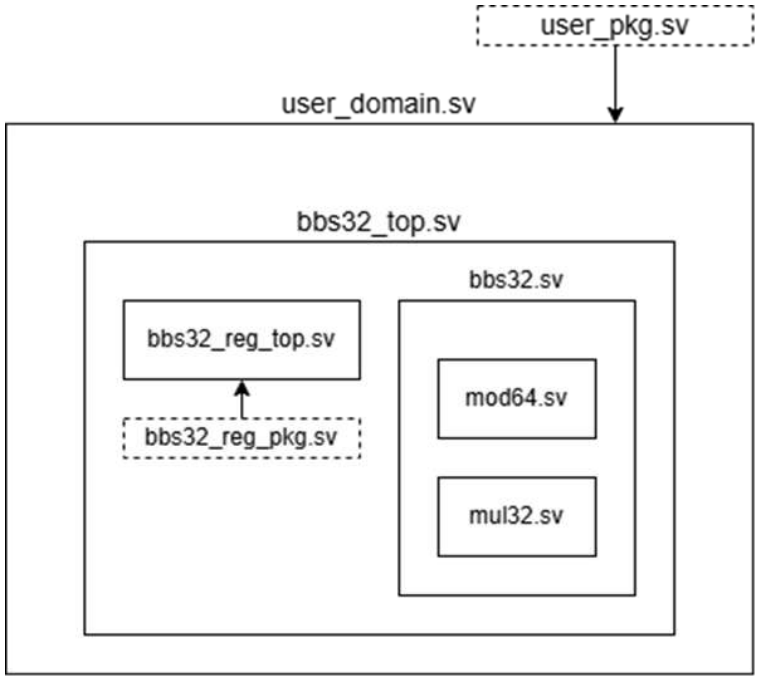


Figure 3.3 – BBS generator modules inside Croc project

To embed BBS32 module into Croc project file [git]/Bender.yml, need to be modified as shown below:

- Add a new target named “sg13g2 and add two files for clock and SRAM primitives.
- Add register package at Level #0.
- Add BBS32 RTL into section RTL->Level #2.

```
24  sources:
25    # Level 0
26    - target: ihp13
27      files:
28        - ihp13/tc_clk.sv
29        - ihp13/tc_sram_impl.sv
30
31    - target: sg13g2
32      files:
33        - sg13g2/tc_clk.sv
34        - sg13g2/tc_sram_impl.sv
35
36    - rtl/croc_pkg.sv
37    - rtl/user_pkg.sv
38    - rtl/soc_ctrl/soc_ctrl_reg_pkg.sv
39    - rtl/gpio/gpio_reg_pkg.sv
40    # add your design files containing anything but modules (packages) here
41    - rtl/user_domain/bbs32_reg_pkg.sv
```

Figure 3.4 – Bender file: add new target

```
24  sources:
25    # Level 0
26    - target: ihp13
27      files:
28        - ihp13/tc_clk.sv
29        - ihp13/tc_sram_impl.sv
30
31    - target: sg13g2
32      files:
33        - sg13g2/tc_clk.sv
34        - sg13g2/tc_sram_impl.sv
35
36    - rtl/croc_pkg.sv
37    - rtl/user_pkg.sv
38    - rtl/soc_ctrl/soc_ctrl_reg_pkg.sv
39    - rtl/gpio/gpio_reg_pkg.sv
40    # add your design files containing anything but modules (packages) here
41    - rtl/user_domain/bbs32_reg_pkg.sv
42
```

Figure 3.5 – Bender file: add register package

```
38    # RTL
39  - target: not(netlist_yosys)
40    files:
41      # Level 1
42      - rtl/core_wrap.sv
43      - rtl/soc_ctrl/soc_ctrl_reg_top.sv
44      - rtl/gpio/gpio_reg_top.sv
45      - rtl/gpio/gpio.sv
46      # Level 2
47      - rtl/croc_domain.sv
48      - rtl/user_domain.sv
49      # add your design files with modules here
50      - rtl/user_domain/mul32.sv
51      - rtl/user_domain/mod64.sv
52      - rtl/user_domain/bbs32.sv
53      - rtl/user_domain/bbs32_top.sv
54      - rtl/user_domain/bbs32_reg_top.sv
55      # Level 3
56      - rtl/croc_soc.sv
```

Figure 3.6 – Bender file: add SystemVerilog sources

3.1.2 Generate file list

To generate file list that can be passed to the Cadence flow tool, the following Make target can be used:

- make yosys-flist BENDER=../ext_bin/bender BENDER_TARGETS="asic sg13g2 rtl synthesis"

Where: “BENDER=” is the path to the executable file of the Bender program; “BENDER_TARGETS=” configure it to use sg13g2 target instead of the default.

3.2 BBS module simulation

All tests were performed using the free version of Vivado 2023.1 simulator.

3.2.1 Multiplier

In this testbench, several pairs of random 32-bit values (a, b) are generated and checked to verify that this module outputs the correct values.

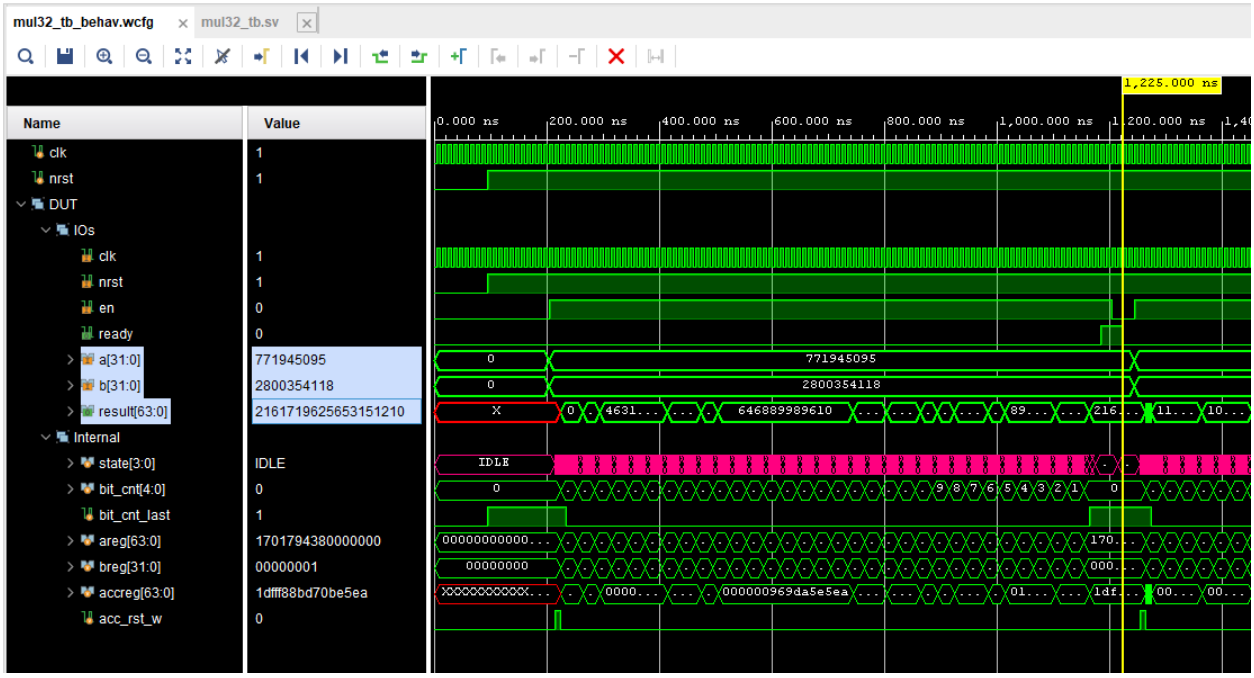


Figure 3.7 – Multiplier testbench waveforms

After reset, the deactivated testbench generates two 32-bit unsigned numbers:

- Input a = 771945095
- Input b = 2800354118

After setting input *en* to “1”, the multiplier starts to work and, after 32 clock periods, sets the output ready to “1”. The calculated result is 2161719625653151210.

To repeat the calculation, input *en* should be set to ‘0’ until the multiplier sets the output *ready* to “0”.

3.2.2 Module Operation (MOD)

This test sends several pre-defined constants and checks if the module calculator returns the correct value.

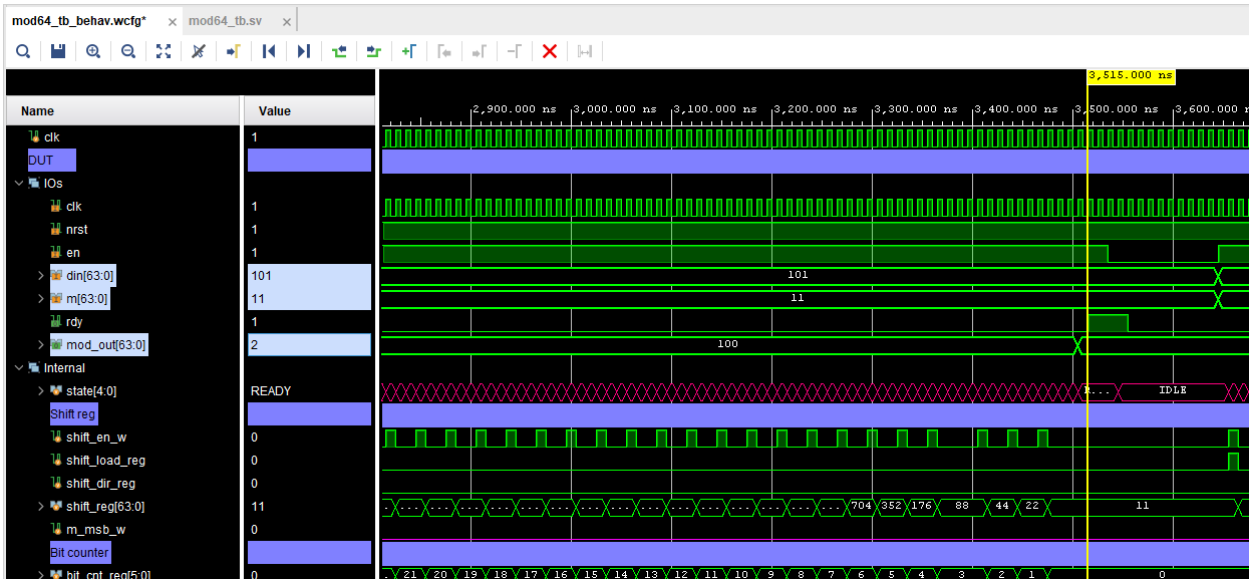


Figure 3.8 – MOD calculation module testbench waveforms

Ot waveform above is an example of calculation (101 mod 11), and the answer is 2. This value was returned on the output *mod_out* when *rdy* output is “1”.

3.2.3 BBS functionality

To test the BBS generator functionality, the Croc testbench was modified to run the whole SoC, including simulation loading the HEX firmware into the SRAM via JTAG.

To do this, a C library was implemented that works with the “bbs32” module, see Appendix B.

Default Croc C program “Hello world” in “sw” folder was modified as follows:

- Write predefined P, Q, SEED values using “bbs32_write_*” functions
- Run module
- Read results from module using “bbs32_read_result”

- Compare with “golden value” and print via UART message “Expected result” if read value is the same as “golden”.

```
80
81      //Write P, Q, Seed
82      uint32_t res = 0u;
83      bbs32_write_pq(29711, 45543);
84      bbs32_write_seed(56686);
85      bbs32_first_run();
86      bbs32_wait_ready();
87      res = bbs32_read_result();
88      printf("Result 0x%x", res);
89      if (res != 0x6E341593) printf("--> Unexpected result\n");
90      else                   printf("--> Expected result\n");
91      uart_write_flush();
92
93      bbs32_next_run();
94      bbs32_wait_ready();
95      res = bbs32_read_result();
96      printf("Result 0x%x", res);
97      if (res != 0xE62F5EBC) printf("--> Unexpected result\n");
98      else                   printf("--> Expected result\n");
99      uart_write_flush();
100
101     return 1;
102 }
```

Figure 3.9 – Modified “Hello world” C program

Firstly, run a C program:

- Add to PATH environment variable path to the cross-compiler suitable for Ubuntu 18.04 (riscv64-elf-ubuntu-18.04-nightly-2022.11.12-nightly/riscv/bin)
- Add environment variable LD_LIBRARY_PATH with path to the folder with library "libmpfr.so.6”
- Run *make* command inside “sw” folder.

After that, return to the folder “xrun” and run the command *make sim*.

The result of executing this testbench is shown below.

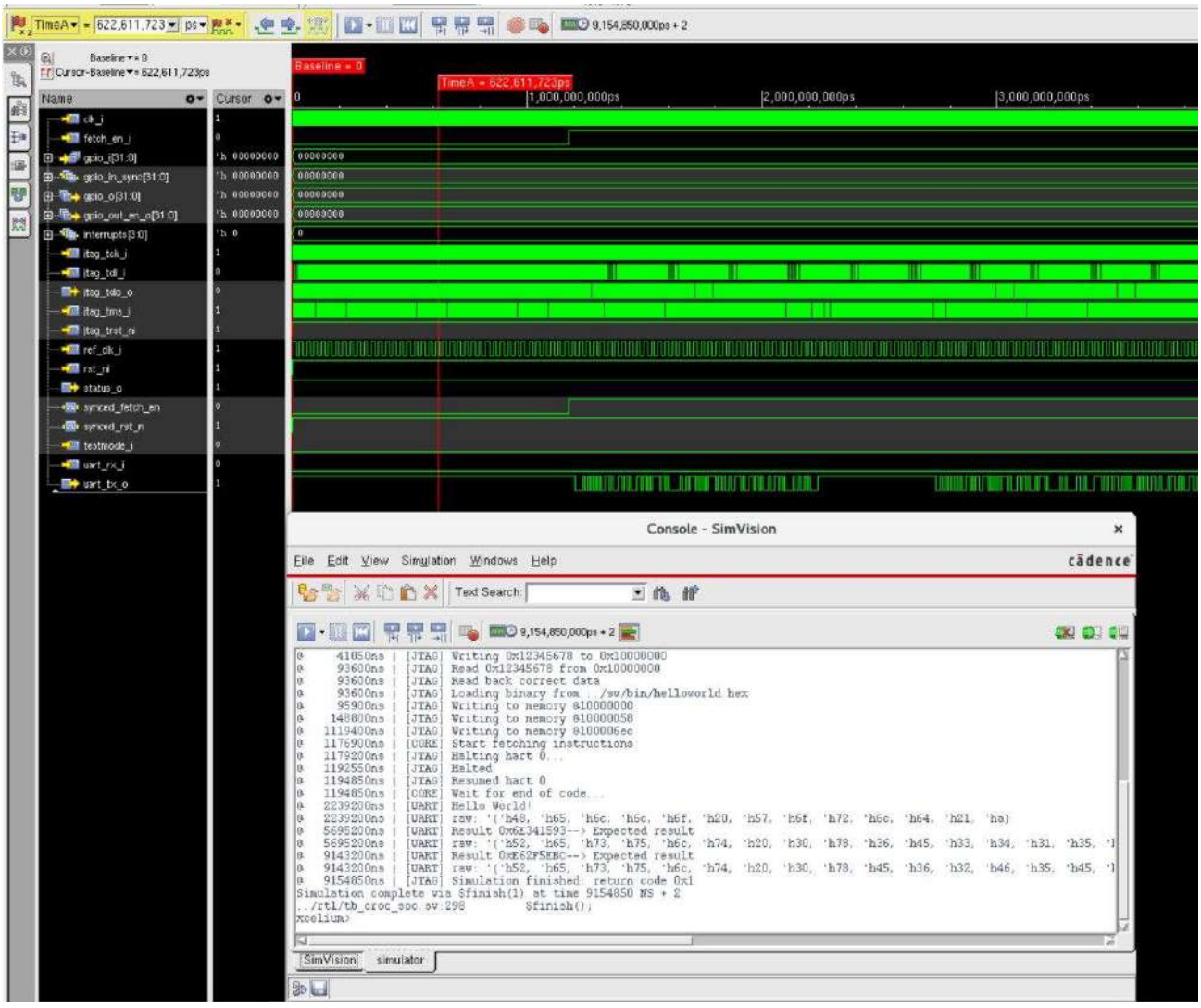


Figure 3.10 – Result simulation modified “Hello world” using Xcellium

As can be seen, this testbench outputs expected results.

3.3 Physical design flow setup

To implement physical design in Cadence Innovus, the “foundation Flow Template” feature was used. To generate a flow template, perform the following steps:

- Run Innovus without GUI: innovus -no_gui
- Run command: write_flow_template -directory ./scripts

These create a directory “script” with a flow template.

Proposal structure for project:

- Top directory:
    - input: directory with Croc filelist, constraints, additional Tcl scripts (for example, for floorplan)
    - rtl: directory with Croc project
    - scripts: directory with Cadence flow template.

After that, you need to configure the generated template.

3.3.1 Configure design_config.tcl

Open file “design_config.tcl” and add to it beginning paths to the standard library cells and IO cells as shown below.

Table 3.1 – Add to “design_config.tcl” paths to IO and STD libraries

```
set GATELIB     "$env(GATELIB_IHP)"

set curr_dir [file normalize [file dirname [info script]]]
set proj_dir [file normalize $curr_dir/../]
set out_dir [file normalize $proj_dir/out]

set io_library   "$GATELIB/io/rev1.2.2/ixc013g2_iocell"
set std_library  "$GATELIB/stdcells/rev1.3.2/ixc013g2ng_stdcell"
```

After that, add to the init_step procedure the following commands that add STD, IO, and SRAM libraries to the design:

Table 3.2 – Add to “design_config.tcl” LEF and filelist

```
create_flow_step -name init_design -owner design -write_db {
    #- setup library information
    read_mmmc              [get_db init_flow_directory]/mmmc_config.tcl
    read_physical -lefs    [list \
                                $std_library/lef/ixc013g2ng_tech.lef \
                                $std_library/lef/ixc013g2ng_stdcell_v5p7.lef \
                                $std_library/lef/ixc013g2ng_phys.lef \
                                $io_library/lef/ixc013g2_iocell_v5p8.lef \
                                [glob -nocomplain -directory \
                                    /pdks/SG13G2/open_pdk_v0.2.0/libs.ref/sg13g2_sram/lef/ \
                                    RM_IHPSG13_1P_*_c?_bm_bist.lef] \
                                ]
#Add filelist
set src_rtl_list    $proj_dir/input/croc.flist
    read_hdl -language sv -f $src_rtl_list
    elaborate              "croc_chip"

#- initialize library and design information
    init_design
}
```

After that, add the floorplan Tcl script to be sourced in the initialize floorplan step and route rules.

Table 3.3 – Add “design_config.tcl” floorplan TCL and rout rules

```
create_flow_step -name init_floorplan -owner design {
    #- initialize floorplan object using DEF and/or floorplan files
    source $proj_dir/input/floorplan.tcl

    #- create route_rule
    create_route_rule -name cts_2x_space_2x_width \
        -spacing_multiplier {Metal2:Metal5 2} \
        -width_multiplier {Metal2:Metal5 2} \
        -generate_via
    create_route_rule -name cts_2x_space \
        -spacing_multiplier {Metal2:Metal5 2}

    #- define route_types and/or route_rules
    create_route_type -name route_type_leaf  \
        -top_preferred_layer Metal3 \
        -bottom_preferred_layer Metal2 \
        -preferred_routing_layer_effort high
    create_route_type -name route_type_trunk \
        -route_rule cts_2x_space \
        -top_preferred_layer Metal4 \
        -bottom_preferred_layer Metal3 \
        -preferred_routing_layer_effort high
    create_route_type -name route_type_top   \
        -route_rule cts_2x_space_2x_width \
        -top_preferred_layer Metal5 \
        -bottom_preferred_layer Metal4 \
        -preferred_routing_layer_effort high

    #- finish floorplan with auto-blockage insertion
    finish_floorplan  -fill_place_blockage soft 20.0
}
```

3.3.2 Configure genus_config.tcl

Open “genus_config.tcl” and add information about the process node, top, and bottom routing layers.

Table 3.4 – Configure “genus_config.tcl”

```
create_flow_step -name init_genus -owner flow {


  # Physical Synthesis attributes  [get_db -category phys]
  #-------------------------------------------------------------------------------
  set_db design_process_node              130
  set_db design_top_routing_layer         TopMetal2
  set_db design_bottom_routing_layer      Metal1
}
```

3.3.3 Configure innovus_config.tcl

This file contains information about what cells will be used for CTS, tieoff, and filler cells:

- Tieoff cells: used to connect constant “0” and “1” to the VSS and VDD, respectively.
- CTS cells: clock buffers, inverters, and clock-gating cells.
- Fillers: cells used to fill gaps between STD cells.

Table 3.5 – Configure “innovus_config.tcl”

```
create_flow_step -name init_innovus -owner flow {
  # Design attributes  [get_db -category design]
  #--------------------------------------------------------------------------------
  set_db design_process_node              130
  set_db design_top_routing_layer         TopMetal2
  set_db design_bottom_routing_layer      Metal1

<...>

  # Tieoff attributes  [get_db -category add_tieoffs]
  #--------------------------------------------------------------------------------
  set_db add_tieoffs_cells              "LOGIC0JI LOGIC1JI"

<...>

  set_db cts_buffer_cells           [ list BUJIX20 BUJIX16 BUJIX12 BUJIX8 BUJIX4 BUJIX2 BUJIX1 ]
  set_db cts_inverter_cells         [ list INJIX0 INJIX1 INJIX2 INJIX4 INJIX8 INJIX12 INJIX16
INJIX20 ]
  set_db cts_clock_gating_cells     [ list LGCNJIX1 LGCPJIX1 LSGCNJIX1 LSGCPJIX1 ]

<...>

  # Filler attributes  [get_db -category add_fillers]
  #--------------------------------------------------------------------------------
  set_db add_fillers_cells          [ list FEED1JI FEED2JI FEED3JI FEED5JI FEED10JI FEED25JI ]

}
```

3.3.4 Configure mmmc_config.tcl

This file configures all timing information for all cells and the hard macro, and specifies which data should be used for fast/slow analysis. The full contents of the file are in Appendix D.

In this file, the SoC timing constraints are defined as shown in the table below.

Table 3.6 – Configure timing constraint file in mmmc_config.tcl

```
################################################################################
## CONSTRAINT MODES
################################################################################
source $proj_dir/input/instances.tcl
create_constraint_mode  \
    -name cm_func \
    -sdc_files $proj_dir/input/constraints.sdc
```

3.3.5 Configure timing constraints

Full content of timing constraint file *constraints.sdc* are in Appendix E.

According to this file, the SoC has three clock domains. They are described in the table below.

Table 3.7 – Croc SoC clock domains

Clock domain name	Port name	Period / Frequency	Description
clk_sys	clk_i	12.5 ns / 80 MHz	Main clock for SoC
clk_jtag	jtag_tck_i	20.0 ns / 50 MHz	JTAG clock
clk_rtc	ref_clk_i	50.0 ns / 20 MHz	Timer reference clock input

3.4 Synthesis flow

The synthesis process in Genus (part of Cadence EDA) follows a three-stage pipeline: syn_generic, syn_map, and syn_opt [11, page 32]. The accuracy of timing reports, specifically Worst Negative Slack (WNS), improves at each stage as wire delay estimation becomes more realistic. A brief description of each step is provided in the tables below [11, pp. 32-33].

Table 3.8 – Genus synthesis flow description

Generic Synthesis (syn_generic)	
Description	This stage translates RTL into a technology-independent netlist
Function	Elaborates HDL and maps logic to generic boolean gates
Wire delay	0 ns (Ideal). Since no placement or physical data is applied yet, interconnects are typically treated as ideal or estimated without physical context
WNS estimate	Measures pure logic depth. A positive WNS here confirms the architectural logic speed is sufficient for the target clock
Technology Mapping (syn_map)	
Description	This stage maps generic logic to specific standard cells from the technology library
Function	Replaces generic gates with physical cells (e.g., NAND2, DFF) defined in the library
Wire delay	Statistical (WLM). Delays are estimated using vendor-supplied Wire Load Models (WLM) based on fanout, as precise placement is not yet available
WNS estimate	Provides a rough timing check using statistical wire approximations
Optimization (syn_opt)	
Description	This final stage refines the design to meet timing, area, and power constraints
Function	Resizes cells, buffers signals, and performs logic restructuring
Wire delay	Delays are calculated using Physical Layout Estimation (PLE) based on loaded LEF/QRC data
WNS estimate	Highly accurate. Correlates closely with the final Place & Route results due to physical awareness

3.5 Floorplanning implementation

The floorplan script is placed in “input/floorplan.tcl”; see Appendix F.

3.5.1 Core dimension

To set the main dimensions (such as the die and core areas), the command create_floorplan is used. In the floorplan script file, only the die and IO box sizes were set. Other dimensions are calculated by script

```
## =========================================================
## Create floorplan: die, IO, Core
## =========================================================

set padD    310; # pad depth (edge to core)
set padW     60; # pad width (beachfront)

#Set die WxH
set dieW   2022
set dieH   2022
#Set IO box WxH
set ioboxW 36
set ioboxH 36

#Calculate IO box right-top
set ioR     [expr $dieW - $padD]
set ioT     [expr $dieH - $padD]
#Calculate core dim.
set coreL   [expr $padD + $ioboxW]
set coreB   [expr $padD + $ioboxH]
set coreR   [expr $ioR - $ioboxH]
set coreT   [expr $ioT - $ioboxW]


#From: "Innovus Stylus Common UI Text Command Reference" p. 1004 (TCRcom.pdf)
#define box_size:
#   die box:  left bottom right top
#   IO box:   left bottom right top
#   core box: left bottom right top
create_floorplan \
    -site CoreSite \
    -no_snap_to_grid \
    -box_size \
        0      0      $dieW  $dieH \
        $padD  $padD  $ioR   $ioT \
        $coreL $coreB $coreR $coreT
```

The two figures below show that the die size and the IO box meet expectations.

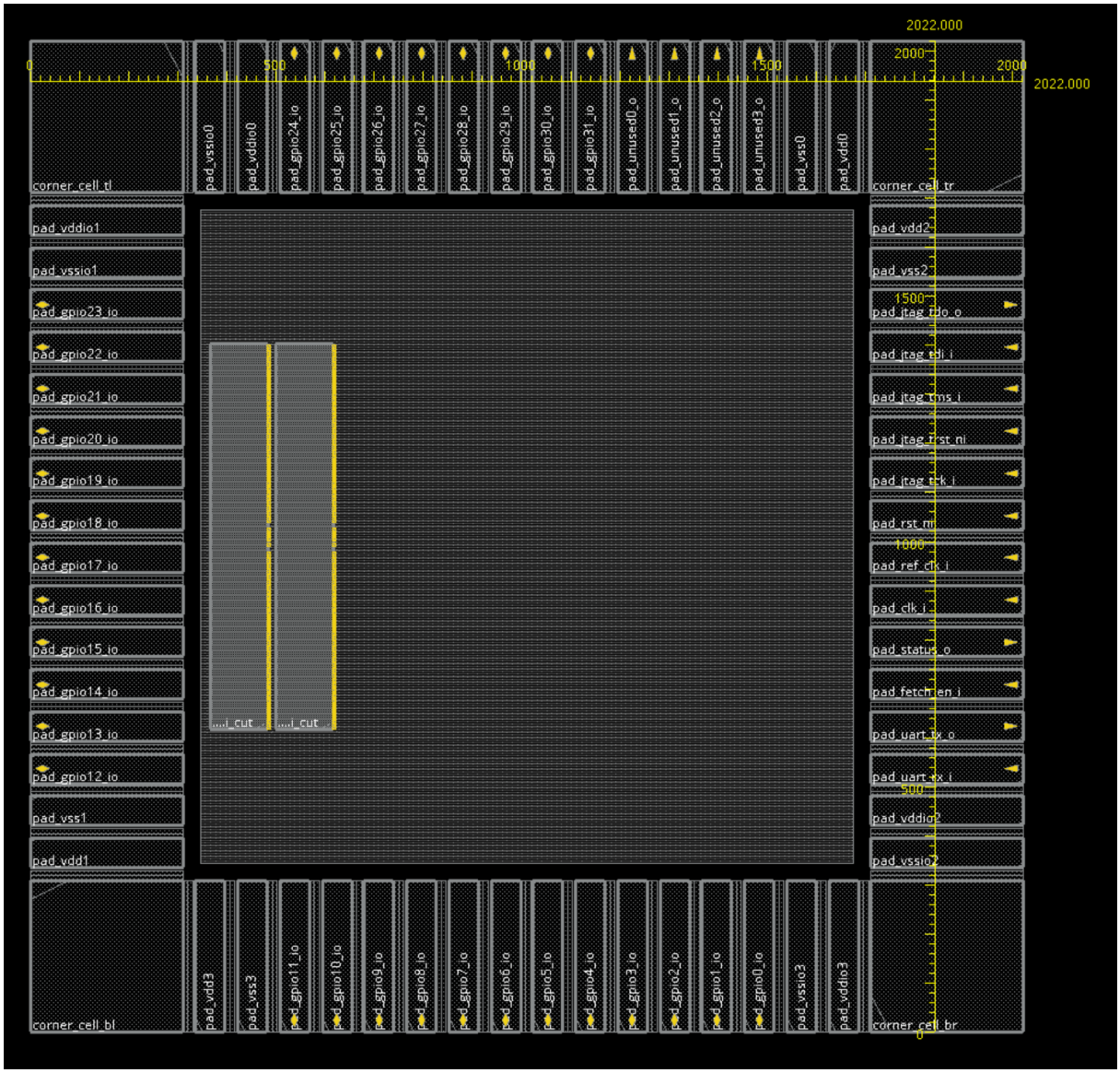


Figure 3.11 – SoC die dimension

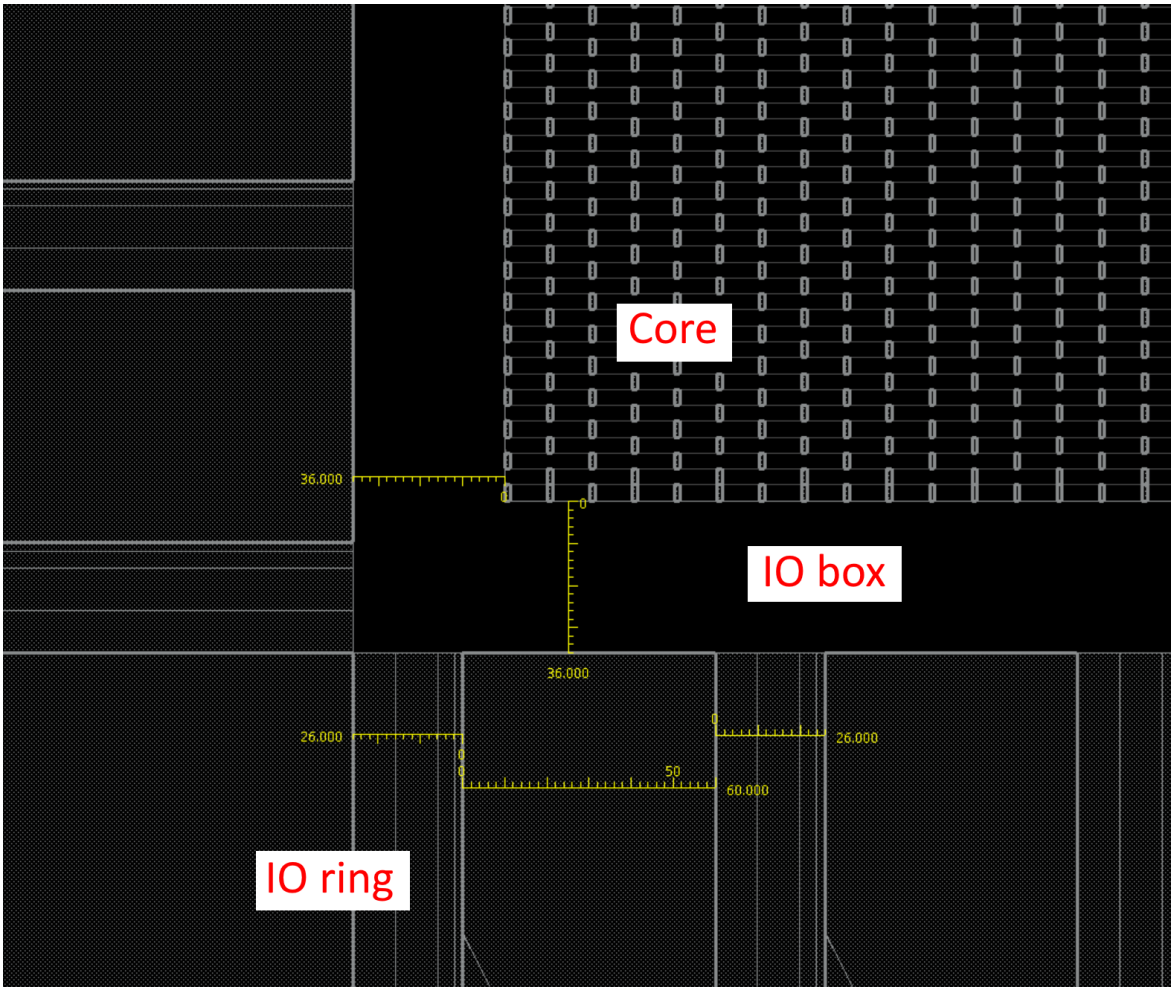


Figure 3.12 – SoC IO box dimension

Core size can be easily calculated:

$$CORE = DIE - 2 \cdot IOCELL - 2 \cdot IOBOX$$

Where: *DIE* – is die size; *IOCELL* – is IO cell height (310 um); *IOBOX* – IO box size;

$$CORE = 2022 - 2 \cdot 310 - 2 \cdot 36 = 1330[um]$$

3.5.2 Pin placement

To place pins in order as specified earlier, a special IO file was used. It contains information on the order and side on which the pins should be placed. Also, there are special corner cells configured to connect the IO cell rings. Full content for this file (placed in“input/io_padring.tcl”) can be found in Appendix G.

To apply this file to the design, there is a command *read_io_file*.

```
read_io_file $input_dir/io_padring.tcl
```

The result of this command is shown in the screenshot below.

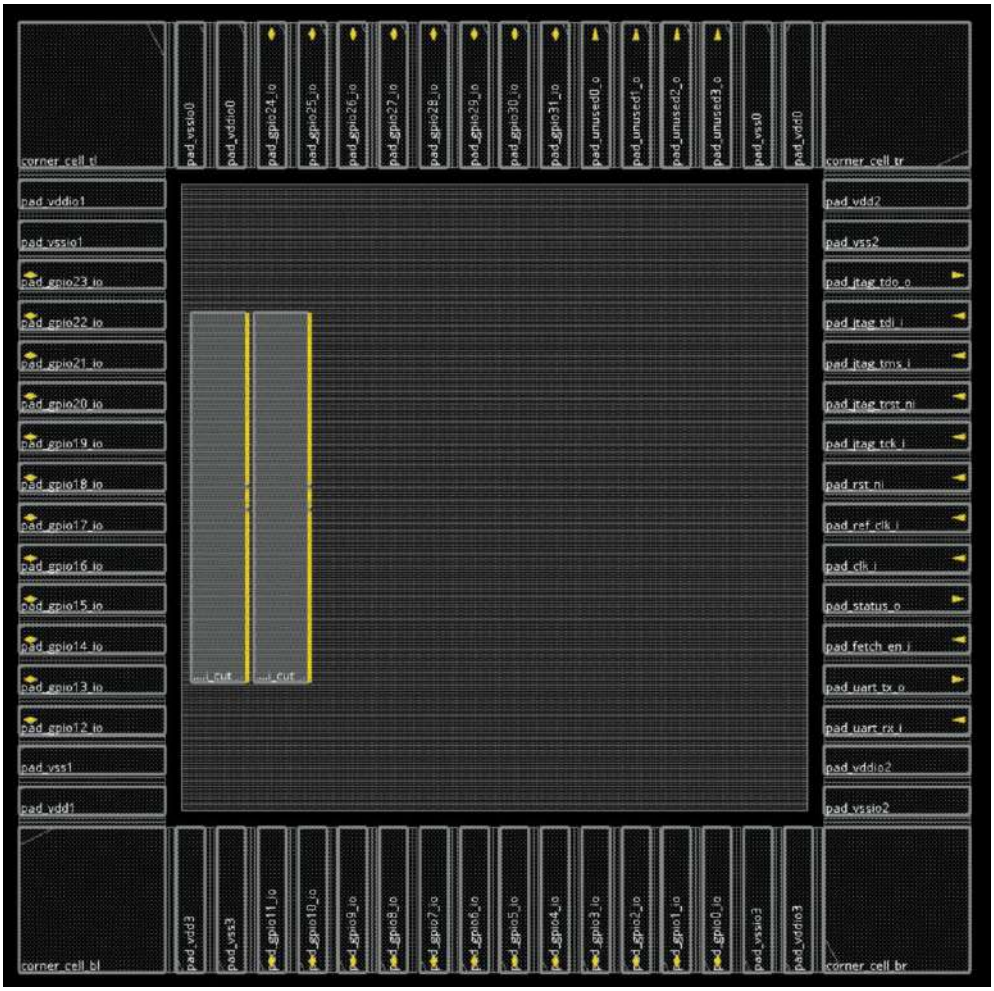


Figure 3.13 – SoC IO placement

3.5.3 Power grid placement

During power grid placement, power nets are first created, and the tool is configured to map the corresponding cell pins to these nets.

In this SoC, there are only two main power nets: VSS and core VDD (+1.2V). IO VDD will be connected automatically using the resources of the IO cells and corner cells.

```
## ========================================================
## Create power nets
## ========================================================
create_net -name VSS -ground
create_net -name CORE_VDD -power
create_net -name IO_VDD -power

#Connect IO pads with these global nets
connect_global_net VSS \
    -type pg_pin \
    -override \
    -pin_base_name VSSPAD \
    -inst_base_name *

connect_global_net VSS \
    -type pg_pin \
    -override \
    -pin_base_name VSSCORE \
    -inst_base_name *

connect_global_net IO_VDD \
    -type pg_pin \
    -override \
    -pin_base_name VDDPAD \
    -inst_base_name *

connect_global_net CORE_VDD \
    -type pg_pin \
    -override \
    -pin_base_name VDDCORE \
    -inst_base_name *
```

After that need to create a core ring for uniformly distributing core VDD and VSS around the core area. It was created inside the IO box reserved using the command *add_rings*.

For current implementation, the core ring consists of two rings (core VDD, VSS) and is placed on TopMetal1. Wire with a width of 10 μm, space between them is set to 6 μm. See the Tcl commands set below.

```
## =========================================================
## Add core rings
## =========================================================
set_db add_rings_target default ;
set_db add_rings_extend_over_row 0 ;
set_db add_rings_ignore_rows 0 ;
set_db add_rings_avoid_short 0 ;
set_db add_rings_skip_shared_inner_ring none ;
set_db add_rings_stacked_via_top_layer TopMetal2 ;
set_db add_rings_stacked_via_bottom_layer Metal1 ;
set_db add_rings_via_using_exact_crossover_size 1 ;
set_db add_rings_orthogonal_only true ;
set_db add_rings_skip_via_on_pin {  standardcell } ;

# Add Core ring on TopMetal1 for VSS adn CORE_VDD
set ringW       10
set ringSpacing 6
#
add_rings \
    -nets {VSS CORE_VDD} \
    -type core_rings \
    -follow core \
    -layer {top TopMetal1 bottom TopMetal1 left TopMetal1 right TopMetal1} \
    -width "top ${ringW} bottom ${ringW} left ${ringW} right ${ringW}" \
    -spacing "top ${ringSpacing} bottom ${ringSpacing} left ${ringSpacing} right ${ringSpacing}" \
    -center 1 \
    -threshold 0 \
    -jog_distance 0 \
    -snap_wire_center_to_grid none
```

The result of this command is shown in the screenshot below.

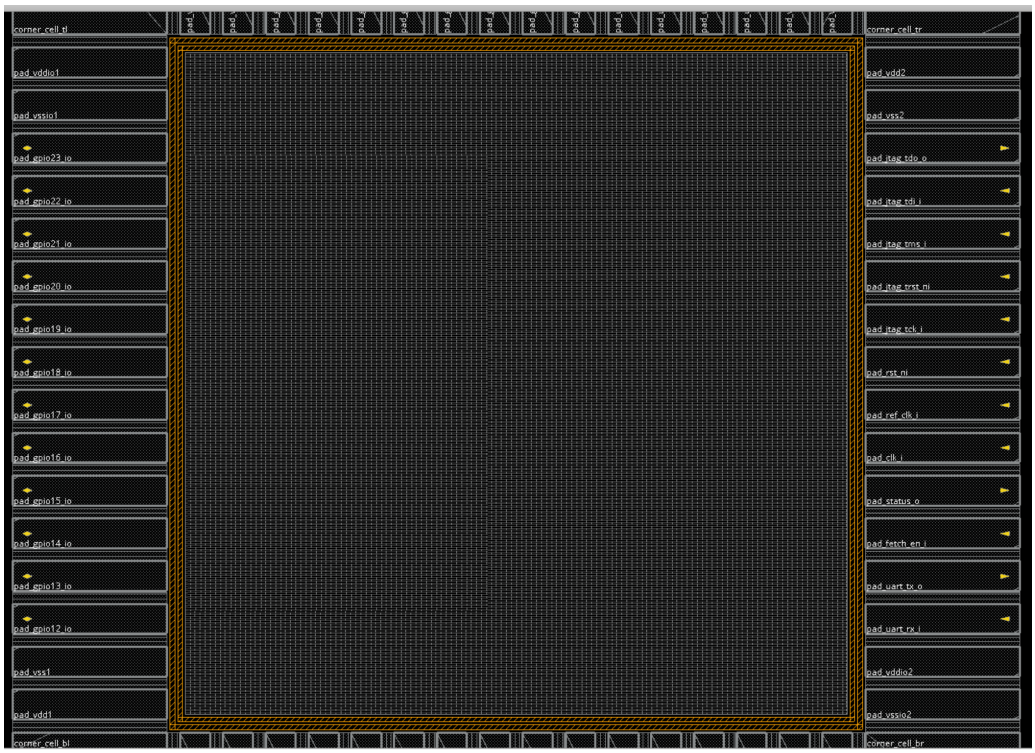


Figure 3.14 – IO ring around core

The next step is adding power stripes on top of the core to distribute power uniformly. For this, TopMetal2 is used, with a stripe width of 10 μm and a horizontal direction. Each stripe is located opposite the IO fillers. This approach will provide a better connection of IO cells to the ring in the future.

The Tcl script for this is shown below.

```
## ========================================================
## Add core stripes on TopMetal 1
## ========================================================

set coreOffset      63
set ioPadSpace      86
#Configure horisontal stripes position
set hstripeW        10
set horzVSSstripe   {0 3 8 11 14}
set horzVCOREstripe {1 4 6 10 13}
set horzStripeMetal {TopMetal2}

# Add horizontal stripes for VSS
foreach iter ${horzVSSstripe} {
    add_stripes \
        -direction horizontal \
        -layer ${horzStripeMetal} \
        -nets "VSS" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${hstripeW}/2] \
        -number_of_sets 1 \
        -width ${hstripeW}
}

# Add horizontal stripes for core VDD
foreach iter ${horzVCOREstripe} {
    add_stripes \
        -direction horizontal \
        -layer ${horzStripeMetal} \
        -nets "CORE_VDD" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${hstripeW}/2] \
        -number_of_sets 1 \
        -width ${hstripeW}
}
```

The result of this part of the script is shown below.



Figure 3.15 – Horizontal stripes on TopMetal2

After that, the script connects the IO cells to the core ring using the *route_special* command.

```
## ========================================================
## Connect power pins to the core ring
## ========================================================

#Connect IO cells pad pins to the power ring
#Connect all ports of pad pins, use only top metal 1 & 2
route_special \
    -nets {CORE_VDD VSS} \
    -connect {pad_pin} \
    -allow_jogging {0} \
    -allow_layer_change {1} \
    -pad_pin_layer_range {TopMetal2 TopMetal1} \
    -pad_pin_port_connect {all_port all_geom}
```

The result of this command is shown in the two screenshots below. The yellow color indicates TopMetal1, and the magenta color indicates TopMetal2.

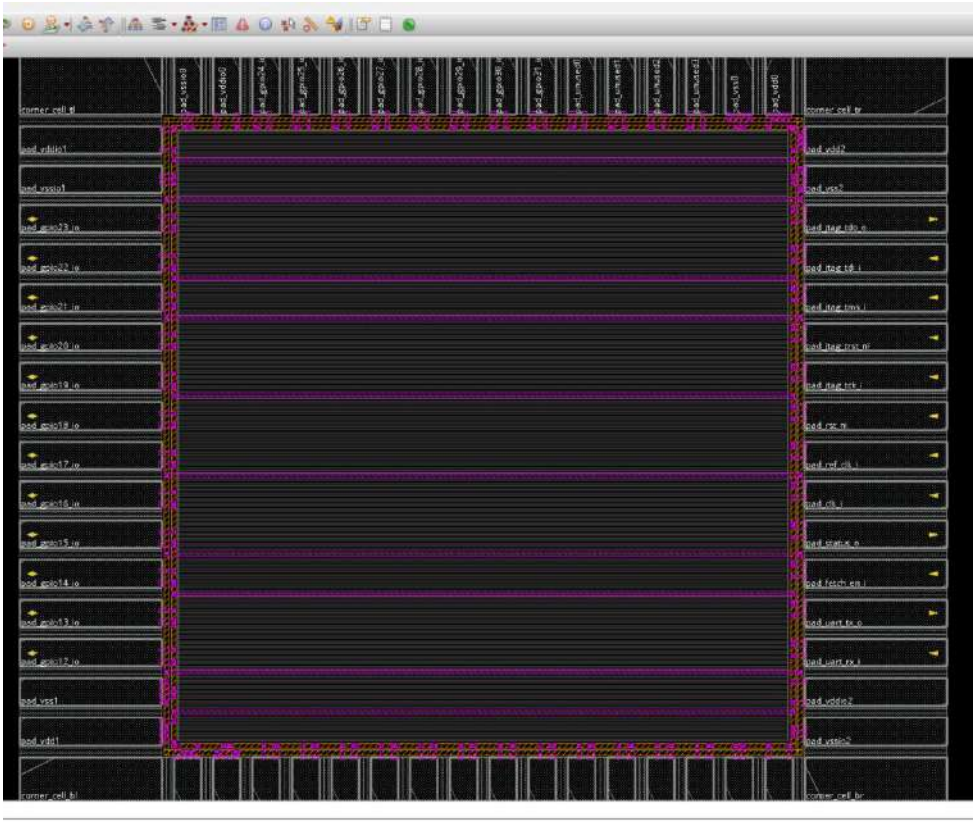


Figure 3.16 – Adding connections from IO cells to the core ring

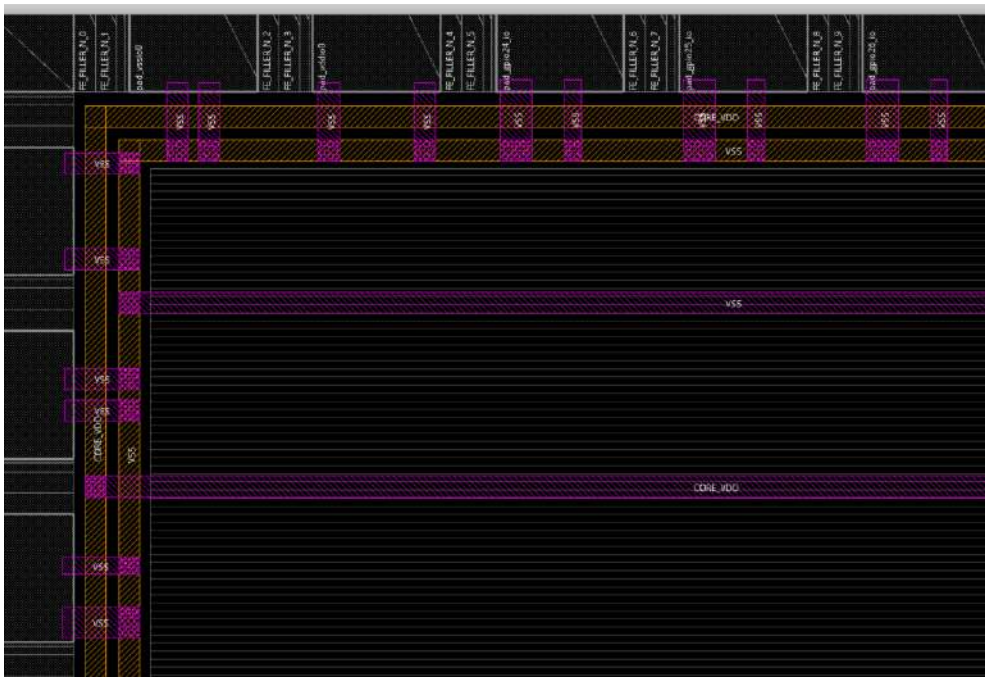


Figure 3.17 – Connection to the core ring, right top corner

3.5.4 SRAM placement

To place SRAM hard macros, need to perform the following operations:

- place macro on active core area;
- rotate macro on 90° (R90);
- place the rotated macro in its final place.

The Tcl script that does this is shown below.

```
## ======================================================
## Place SRAM IPs
## ======================================================

# Vertical distance between SRAMs
set bramH       118
set bramW       785
set bramToBram  15
set bramToCore  20
set bramCell    {RM_IHPSG13_1P_256x64_c2_bm_bist}

# Add SRAM instnces into list
set bramList {}
lappend bramList {i_croc_soc_i_croc_gen_sram_bank\[0\].i_sram_gen_512x32xBx1.i_cut}
lappend bramList {i_croc_soc_i_croc_gen_sram_bank\[1\].i_sram_gen_512x32xBx1.i_cut}

#Configure coordinates for SRAMs AFTER rotation
set bram0X      [expr ${coreL}+${bramToCore}]
set bram0Y      [expr ${coreB}+${coreH}/2-${bramW}/2]
set bram1X      [expr ${bram0X}+${bramH}+${bramToBram}]
set bram1Y      ${bram0Y}

# SRAM #0
#Step 1: put somewhere on core
update_floorplan_obj -obj [lindex ${bramList} 0] -move {850 400}
#Step 2: rotate
flip_or_rotate_obj -rotate {R90} -objs [lindex ${bramList} 0]
#Step 3: put on place
update_floorplan_obj -obj [lindex ${bramList} 0] -move "${bram0X} ${bram0Y}"

# SRAM #1
#Step 1: put somewhere on core
update_floorplan_obj -obj [lindex ${bramList} 1] -move {850 400}
#Step 2: rotate
flip_or_rotate_obj -rotate {R90} -objs [lindex ${bramList} 1]
#Step 3: put on place
update_floorplan_obj -obj [lindex ${bramList} 1] -move "${bram1X} ${bram1Y}"
```

After this, the Tcl script below adds a halo around each SRAM and fixes it in place on the core.

```
# Configure halo around BRAMs
set_macro_place_constraint \
    ${bramList} \
    -halo_sharing true

# Add halo
#-halo_deltas left bottom right top
set haloV    7.3
set haloH    5.6
create_place_halo -halo_deltas ${haloH} ${haloV} ${haloH} ${haloV} -cell ${bramCell} -orient R90

# Fix SRAM placement
foreach bramPath $bramList {
    set_db [get_db insts ${bramPath}] .place_status fixed
}
```

The result of these commands is shown in the screenshot below.

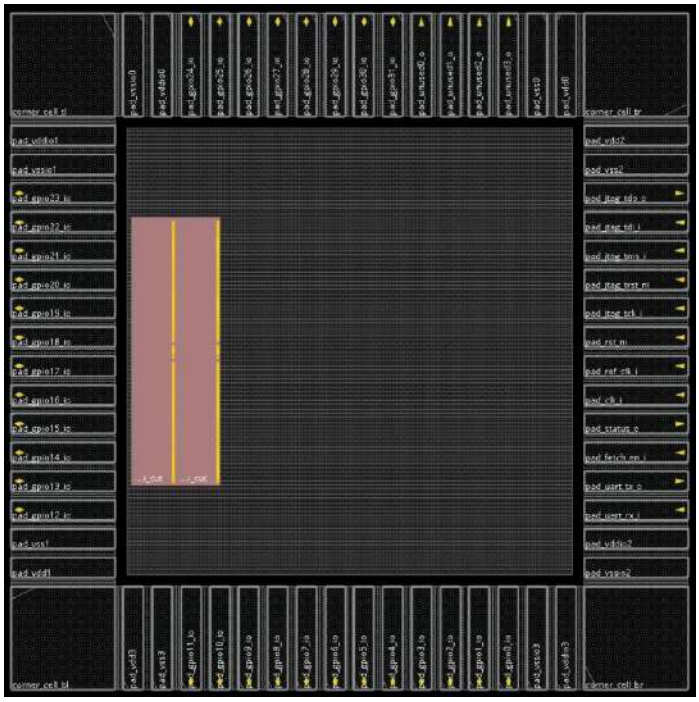


Figure 3.18 – Die view after SRAM placed

3.5.5 SRAM power grid placement

As usual, SRAM consumes a lot of power; there is a need for an additional power ring around it and also additional stripes over it.

For the current implementation, a separate power ring was created around each SRAM instance; the horizontal part is on Metal 4, the vertical part on Metal 5. The ring wire width is set to 2 μm. All these configurations are set by the script below.

```
## =======================================================
## Add ring around BRAMs
## =======================================================

set bramRingMetalV  {Metal4}
set bramRingMetalH  {Metal3}
set bramRingW       2
set bramRingS       0.8
set bramRingOffsetV 2.4
set bramRingOffsetH 0.8

# Select each SRAM and run 'add_rings' on each
foreach bramPath ${bramList} {
    deselect_obj -all
    select_obj [get_db insts ${bramPath}]

    # -user_defined_region {x1 y1 x2 ya ...}
    add_rings \
        -nets {VSS CORE_VDD} \
        -type block_rings \
        -around selected \
        -layer "top ${bramRingMetalH} bottom ${bramRingMetalH} left ${bramRingMetalV} right
${bramRingMetalV}" \
        -width ${bramRingW} \
        -spacing ${bramRingS} \
        -offset "top ${bramRingOffsetV} bottom ${bramRingOffsetV} left ${bramRingOffsetH} right
${bramRingOffsetH}" \
        -threshold 0 \
        -jog_distance 0 \
        -snap_wire_center_to_grid none
}
deselect_obj -all
```

After that, the following Tcl script adds vertical stripes over SRAM on TopMetal1; the stripe width is set to 6 μm.

```
## =======================================================
## Add stripes for BRAMs
## =======================================================

set bramVStripeMetal    {TopMetal1}
set bramVStripeW        6
set bramVStripeS        6
set bramVStripeY1       [expr ${bram0Y}*1.0-7.2]
set bramVStripeY2       [expr ${bram0Y}*1.0+${bramW}*1.0+6.675]
set bramVStripe0X2      [expr ${bram0X}+${bramH}]
set bramVStripe1X2      [expr ${bram1X}+${bramH}]

# Add vertical stripes for BRAM0 manualy
# -area {x1 y1 x2 y2 ...}
add_stripes \
    -direction vertical \
    -layer ${bramVStripeMetal} \
    -nets "VSS CORE_VDD" \
    -width ${bramVStripeW} \
    -spacing ${bramVStripeS} \
    -number_of_sets 2 \
    -start_offset 16 \
    -stop_offset 16 \
    -area "${bram0X} ${bramVStripeY1} ${bramVStripe0X2} ${bramVStripeY2}"

# Add vertical stripes for BRAM1 manualy
# -area {x1 y1 x2 y2 ...}
add_stripes \
    -direction vertical \
    -layer ${bramVStripeMetal} \
    -nets "VSS CORE_VDD" \
    -width ${bramVStripeW} \
    -spacing ${bramVStripeS} \
    -number_of_sets 2 \
    -start_offset 16 \
    -stop_offset 16 \
    -area "${bram1X} ${bramVStripeY1} ${bramVStripe1X2} ${bramVStripeY2}"
```

The result of these commands is shown in the screenshot below.

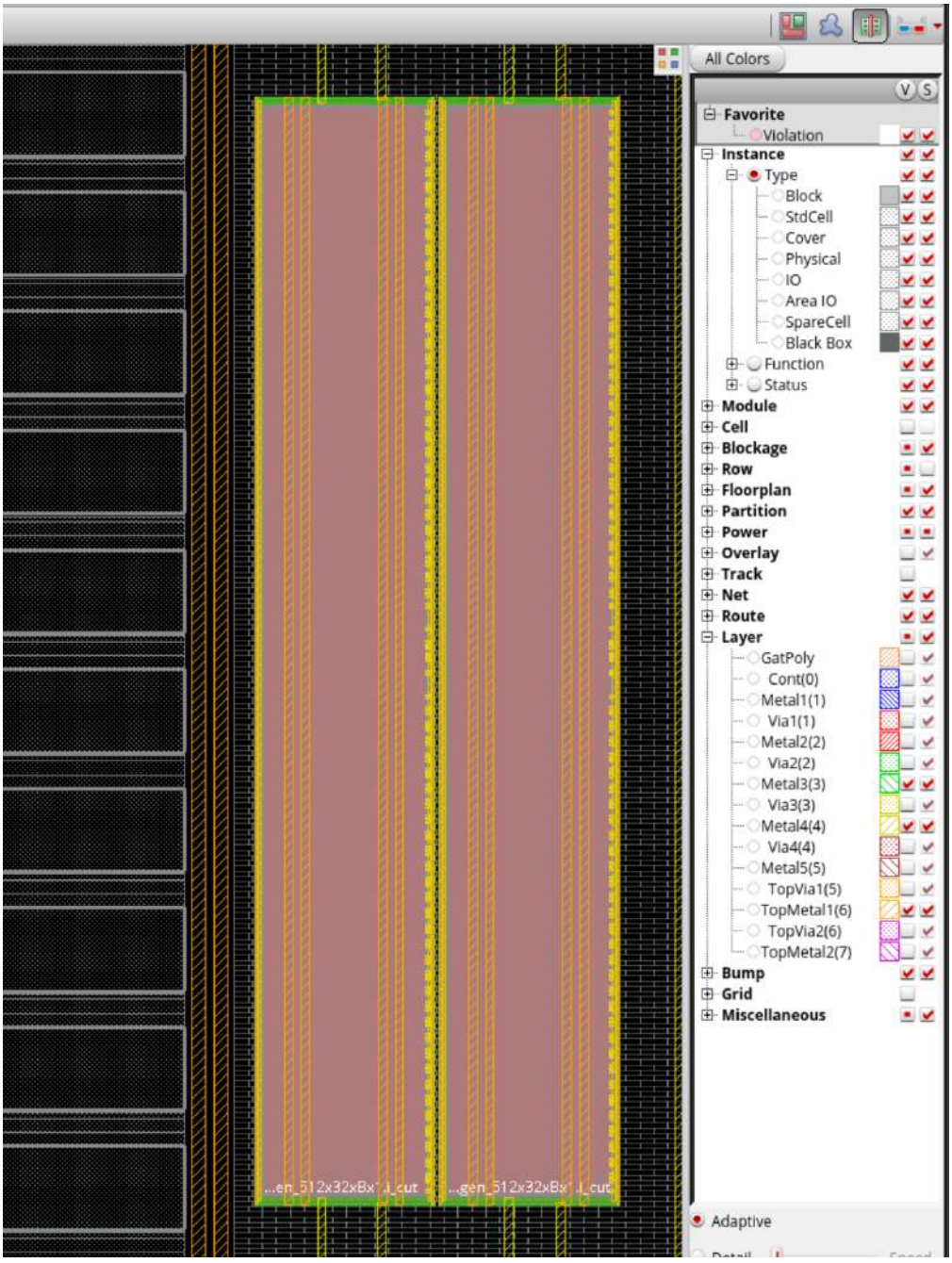


Figure 3.19 – SRAM power ring and stripes

3.5.6 Core power connection

On this step, the core needs to be connected to the core ring.

First of all, create additional vertical stripes on Metal 4, set with a width of 6 μm. The script below does this.

```
## ========================================================
## Add core stripes on Metal 4
## ========================================================

#Configure vertical stripes position
set vstripeW        6
set vertVSSstripe   {0   1.57 3 8 11 14}
set vertVCOREstripe {0.5 2    4 6 10 13}
set vertStripeMetal {Metal4}

# Add vertical stripes for VSS
foreach iter ${vertVSSstripe} {
    add_stripes \
        -direction vertical \
        -layer ${vertStripeMetal} \
        -nets "VSS" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${vstripeW}/2] \
        -number_of_sets 1 \
        -width ${vstripeW}
}

# Add vertical stripes for core VDD
foreach iter ${vertVCOREstripe} {
    add_stripes \
        -direction vertical \
        -layer ${vertStripeMetal} \
        -nets "CORE_VDD" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${vstripeW}/2] \
        -number_of_sets 1 \
        -width ${vstripeW}
}
```

Here, the same approach is used as for the horizontal stripes on TopMetal2 made earlier. Vertical stripes are also connected to the core ring in places opposite to the IO fillers. The result of this connection is shown below.

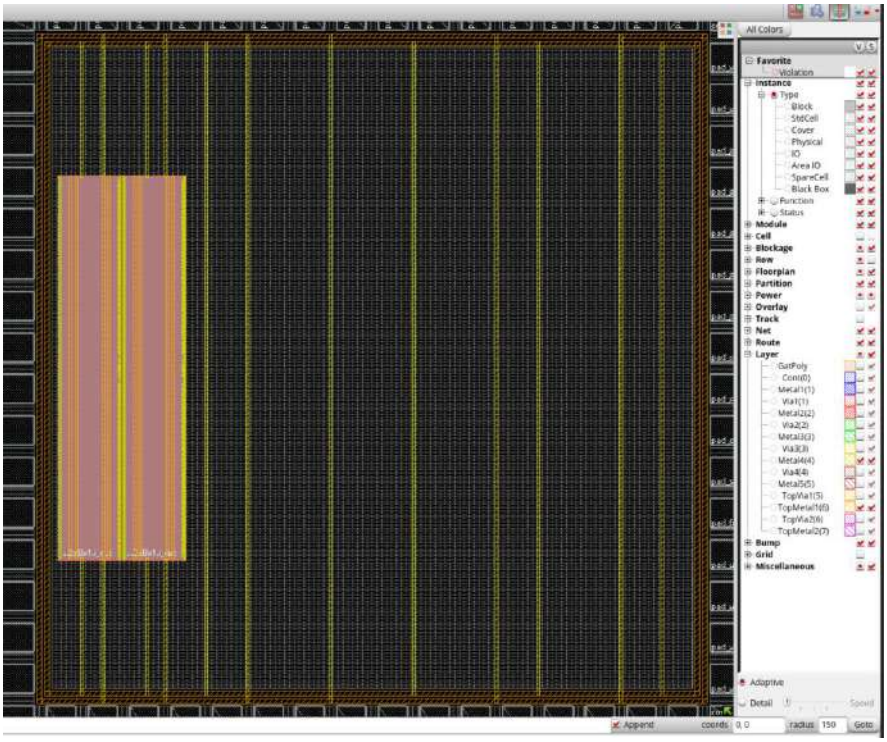


Figure 3.20 – Vertical stripes on Metal4

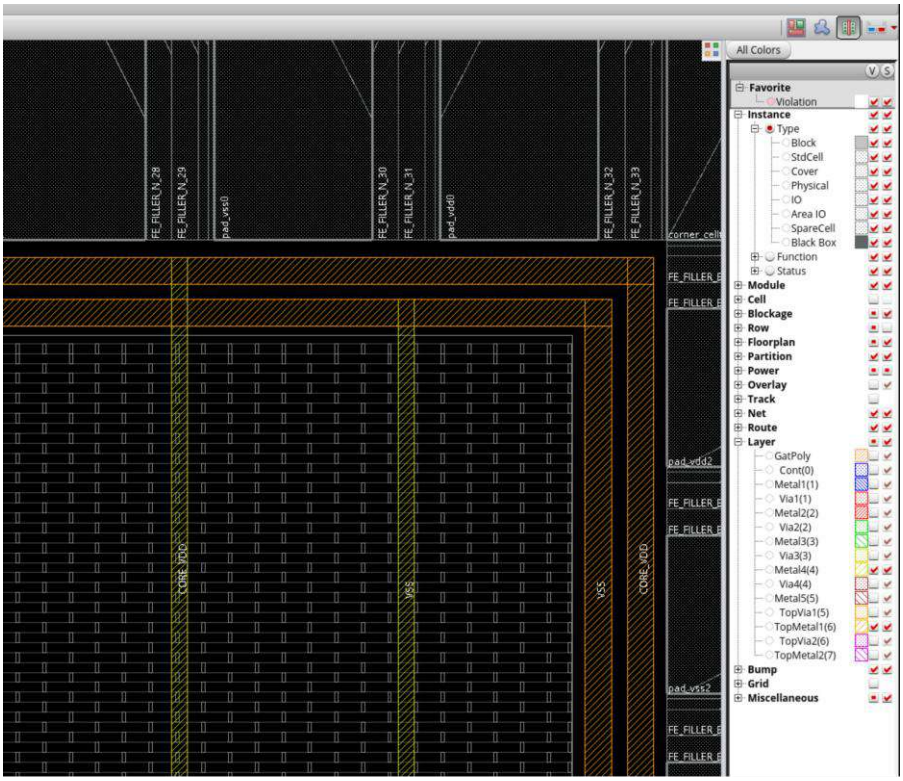


Figure 3.21 – Vertical stripes on Metal 4, right up corner

After this, add stripes on the core using Metal1. They need to connect the STD cells to the power lines. The Tcl script below does this.

```
## ========================================================
## Add Core pins on M1
## ========================================================

route_special \
    -nets {CORE_VDD VSS} \
    -connect {core_pin} \
    -core_pin_layer {Metal1} \
    -allow_jogging {0} \
    -allow_layer_change {0}
```

The screenshot below shows a zoomed version of the core layout with the power stripes on Metal1. Also, power stripes and their connections are displayed.

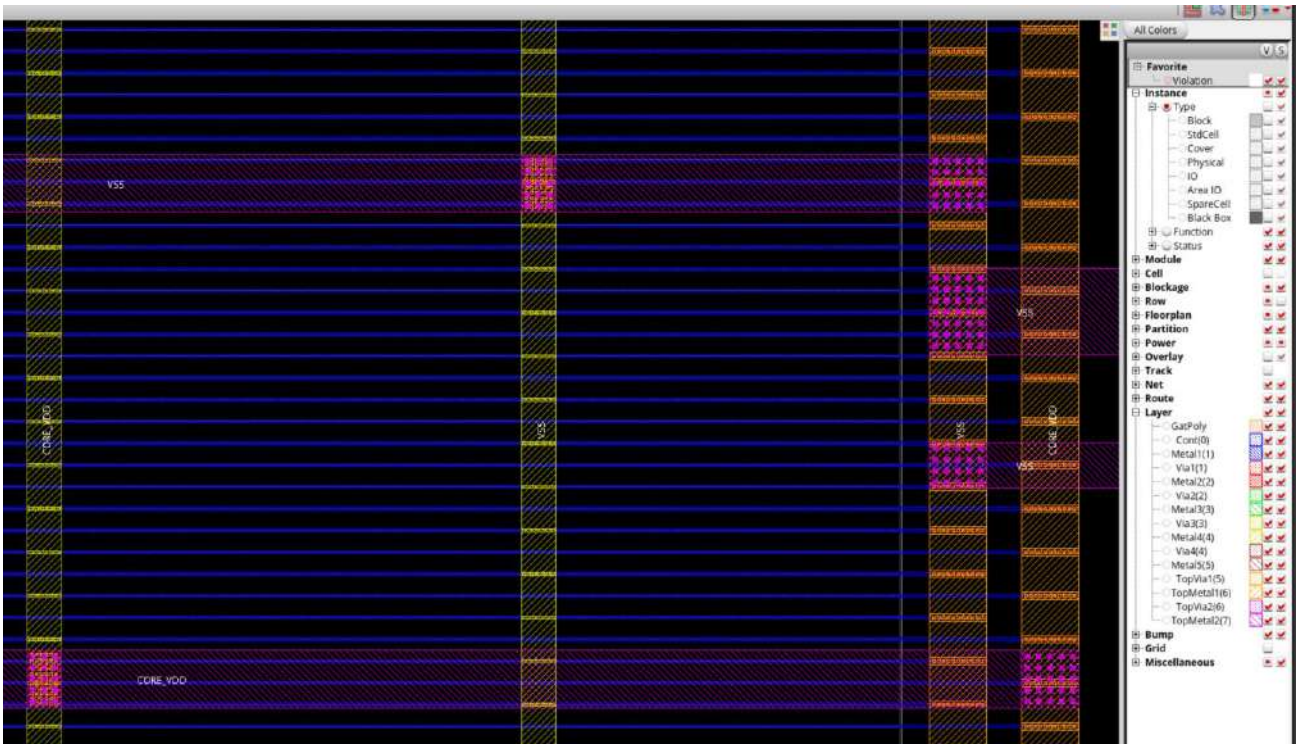


Figure 3.22 – Follow pins on core

In the end, welltaps will be placed on the core.

As there are no separate welltap cells in SG13G2, cell FEED3JI is used as a well-tap cell. The maximum distance between welltaps can be defined by additional rules LU.a and LU.b, which state that the maximum distance between any portion of N+Active and the PWell tie is 20 µm.

The script below calculates the distance between welltaps using the FEED3JI cell width.

```
## =======================================================
## Add welltaps
## =======================================================

# p.2560 add_well_taps
# Get welltap interval from "SG13G2 Layout Rules Rev. 2.0", see "7.2 Latch-up Guidelines", p.67
# Rules LU.a, LU.b is equal to 20 um MAX
# Welltap edge-to-edge distance should not be greater than this value
# Cell FEED3 width is 1.26 um, interval center-to-center calculated as:
# interval = welltap_width + LU.a

set coreSite        0.42
set intervalRule    20
set welltapWidth    1.26; #for FEED3
set wtInterval      [expr ((int($intervalRule/$coreSite)*$coreSite) + $welltapWidth) -
$coreSite*1]

#All well taps here: [get_db add_fillers_cells]

add_well_taps \
    -cell  FEED3JI \
    -prefix WELLTAP \
    -cell_interval ${wtInterval} \
    -start_row 1 \
    -checker_board
```

The script below connects all welltap cells to the core VDD and VSS to prevent DRC errors.

```
connect_global_net \
    CORE_VDD \
    -type pg_pin \
    -pin_base_name VDD \
    -inst_base_name WELLTAP_* \
    -override

connect_global_net \
    VSS \
    -type pg_pin \
    -pin_base_name VSS \
    -inst_base_name WELLTAP_* \
    -override
```

The screen below shows a zoomed-in view of the core with welltap cells.

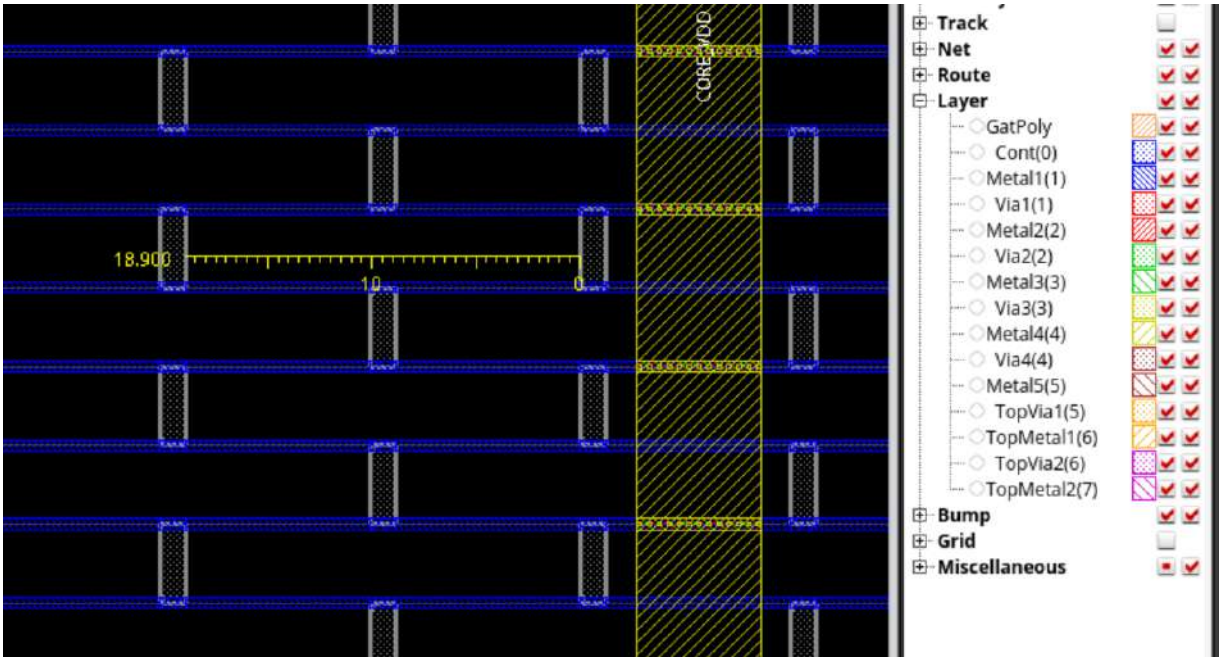


Figure 3.23 – Welltap cells arrangement on core

The screenshot below shows the general view of the SoC floorplan.

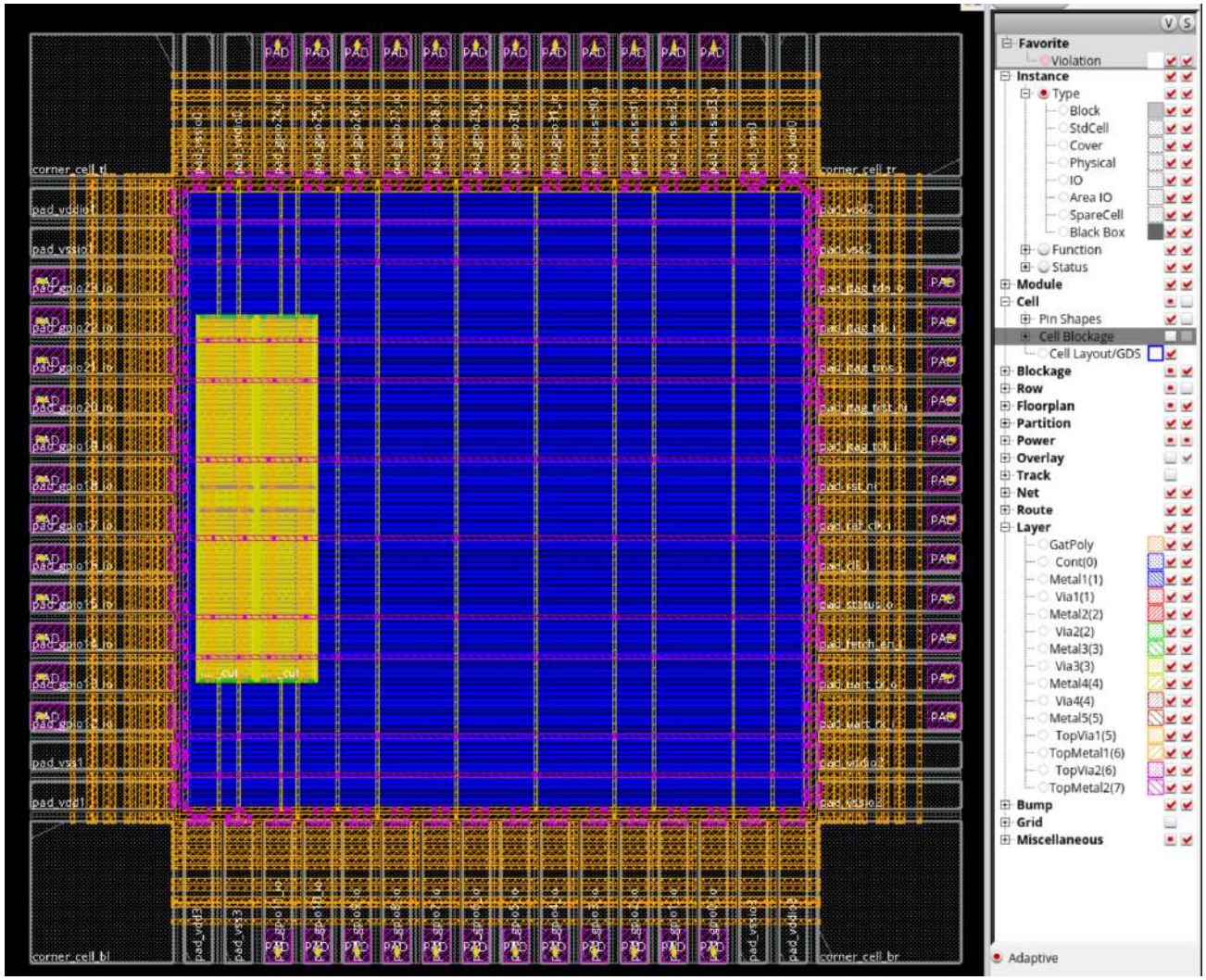


Figure 3.24 – Implemented SoC floorplan

4 Results Evaluation

4.1 NIST Statistical Test

To analyze the algorithm, two tests from “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications” were used:

- Monobit test (2.1)
- Discrete Fourier Transform (Spectral) Test (2.6)

The MATLAB script for both tests is shown in Appendix C.

To get data from the BBS generator, the testbench “xilinx_user_sim/bbs32_analyze_sim/bbs32_analyze_tb.sv” was used. It runs the BBS generator with parameters shown in the table below and makes a CSV file with 1-bit values.

Table 4.1 – Testing P, Q, and Seed values for statistical analysis

P	Q	Seed	Number of total output bits
29711	45543	56686	2048

Monobit test (2.1)

This test evaluates the overall balance between zeros and ones across the entire sequence. Its objective is to determine whether the observed counts of ones and zeros are approximately equal, as expected for a truly random sequence. Specifically, the test examines whether the proportion of ones is close to 1/2, implying that the number of ones and zeros are roughly the same. Passing this test is a prerequisite for the applicability of all other tests.

Test description:

- Convert 0 and 1 to the -1 and +1 and sum the converted values

$$S_n = X_1 + X_2 + \cdots + X_n$$

- Compute the test statistic:

$$s = \frac{|S_n|}{\sqrt{n}}$$

- Compute P-value:

$$P_{value} = erfc\left(\frac{s}{\sqrt{2}}\right)$$

Where: *erfc* – MATLAB complementary error function.

- If the P-value is less than 0.01, the sequence is not random [5]

Discrete Fourier Transform (Spectral) Test (2.6)

This test examines the peak magnitudes in the Discrete Fourier Transform (DFT) of the sequence. Its purpose is to identify periodic components—i.e., repetitive structures occurring at nearby frequencies—that would indicate a deviation from the assumption of randomness. Specifically, the test evaluates whether the fraction of spectral peaks exceeding the 95% threshold differs significantly from the expected 5%.

Test description:

- Convert 0 and 1 to -1 and +1 values to create a sequence $X = x_1, x_2, \ldots, x_n$.
- Apply the Discrete Fourier transform (DFT) on X to produce $S = DFT(X)$.
- Calculate $M = |S'|$, where S' is subsequence of first n/2 elements in S.
- Compute 95% peak height threshold value T:

$$T = \sqrt{\left(\log\frac{1}{0.05}\right)n}$$

- Compute expected theoretical (95%) number of peaks N_0:

$$N_0 = \frac{0.95 \cdot n}{2}$$

- Compute the actual observed number of peaks N_1 in M that are less than T.

- Compute the normalized difference between the observed and the expected number of frequency components *d*:

$$d = \frac{(N_1 - N_0)}{\sqrt{\frac{n \cdot 0.95 \cdot 0.05}{4}}}$$

- Compute P-value:

$$P_{value} = erfc\left(\frac{|d|}{\sqrt{2}}\right)$$

- If the computed P-value is less than 0.01, then conclude that the sequence is non-random [5].

Results from the MATLAB script for the monobit test are shown in the screenshots below.

```
>> bbs_test
Checking P = 29711 ... Good
Checking Q = 45543 ... Good

Calculate M = P*Q = 1353128073

Start seed is X0 = 56686
Calculate GCD(X0, M) = 1
X0 value is good

Generates 1798 random bits...

Done, largest value = 1350617854, log2 = 30.330972

********************************************************************************
Frequency (Monobit) Test (2.1)
P-value is 0.037956
Sequence is random, P >= 0.01
```

Figure 4.1 – Monobit test result

Results from the MATLAB script for the Spectral test are shown in the screenshots below.

```
**************************************************************************
Discrete Fourier Transform (Spectral) Test (2.6)
Warning: Imaginary parts of complex X and/or Y arguments ignored.
> In bbs_test (line 99)
Threshold = 88.152293
N0 = 8.540500e+02
N1 = 865
d = 2.369750
P-value is 0.017800
Sequence is random, P >= 0.01
>>
```

Figure 4.2 – Spectral test result

For the analyzed 2048-bit sample, with significance level alpha set to. 01, the output was not rejected by the Frequency (Monobit) and DFT (Spectral) tests. This provides limited evidence against obvious bias or strong periodicity, but does not certify randomness or cryptographic security.

4.2 Synthesis Results

Synthesis report after step *syn_opt* is shown below.

```
Run Flow Summary
--------------------

Steps run:  report_start init_genus report_area_genus report_late_summary_genus report_late_paths report_power_genus report_finish

Step status:
    report_start                  success
    init_genus                    success
    report_area_genus             success
    report_late_summary_genus     success
    report_late_paths             success
    report_power_genus            success
    report_finish                 success

-------------------------------------------------------------------------------------------
| Snapshot    | flow.cputime (s) | flow.realtime (s) | timing.setup.tns (ns) | timing.setup.wns (ns) |
|-------------+------------------+-------------------+-----------------------+-----------------------|
| syn_generic | 1:16:04          | 1:25:24           |                    -9 |                -0.260 |
| syn_map     | 0:10:53          | 0:13:16           |                    -0 |                -0.168 |
| syn_opt     | 0:04:32          | 0:05:44           |                    -0 |                -0.168 |
-------------------------------------------------------------------------------------------
```

Figure 4.3 – Report after syn_opt stage

This report shows that from the beginning, the project has the worst negative slack (WNS) for setup time, -0.26 ns, with total negative slack (TNS) -9 ns. But at each step, TNS values decreased, indicating that the tool resolves issues with Slack.

The critical path after the optimization stage is in the JTAG clock group, as shown in the report screenshot below.

```
Timing
--------

Analysis view  Clock    Period
------------------------------
av_func_slow   clk_jtg  20000.0
               clk_rtc  50000.0
               clk_sys  12500.0


  Analysis      Cost    Critical          Violating
    View        Group   Path Slack   TNS     Paths
-------------------------------------------------
av_func_slow  clk_jtg      -167.8 -167.8         1
              clk_rtc    No paths    0.0         0
              clk_sys         0.0    0.0         0
              default      9509.4    0.0         0
-------------------------------------------------
Total                             -167.8         1
```

Figure 4.4 – Critical path after syn_opt stage

Timing analysis was done using *av_func_slow* view. It includes timing information from *.lib files as shown in the table below.

Table 4.2 – Libraries for av_func_slow view

Library name (.lib)	Library set
ixc013g2ng_stdcell_slow_1p08V_125C	Slow
ixc013g2_iocell_slow_1p08V_3p0V_125C	Slow
RM_IHPSG13_1P_*_c?_bm_bist_slow_1p08*_125C	Slow

These libs contain information for the slowest possible combination of voltage and temperature: the lowest core voltage, 1.08 V, and the highest temperature, +125°C. They used to calculate setup timings as the worst-case scenario for it.

To calculate, the hold WNS will be used in the *av_func_fast* view. It includes timing information from *.lib files as shown in the table below.

Table 4.3

Library name (.lib)	Library set
ixc013g2ng_stdcell_fast_1p32V_m40C	Fast
ixc013g2_iocell_fast_1p32V_3p6V_m40C	Fast
RM_IHPSG13_1P_*_c?_bm_bist_fast_1p32*_m55C	Fast

These libs contain information for the fastest possible combination of voltage and temperature: the highest core voltage of 1.32 V and a medium temperature of +40°C (+55°C for SRAM). They used to calculate hold times assuming the worst case.

Cell count after syn_opt is shown on the screen below.

```
45   Instance Count
46   --------------
47   Leaf Instance Count              22776
48   Physical Instance count              0
49   Sequential Instance Count         5927
50   Combinational Instance Count     16849
51   Hierarchical Instance Count          5
```

Figure 4.5 – Cells count after syn_opt stage

4.3 CTS Results

After the post-CTS step, the worst setup and hold timings are:

- Setup time worst: -0.105 ns (violated)
- Hold time worst: 0.00 ns (met)

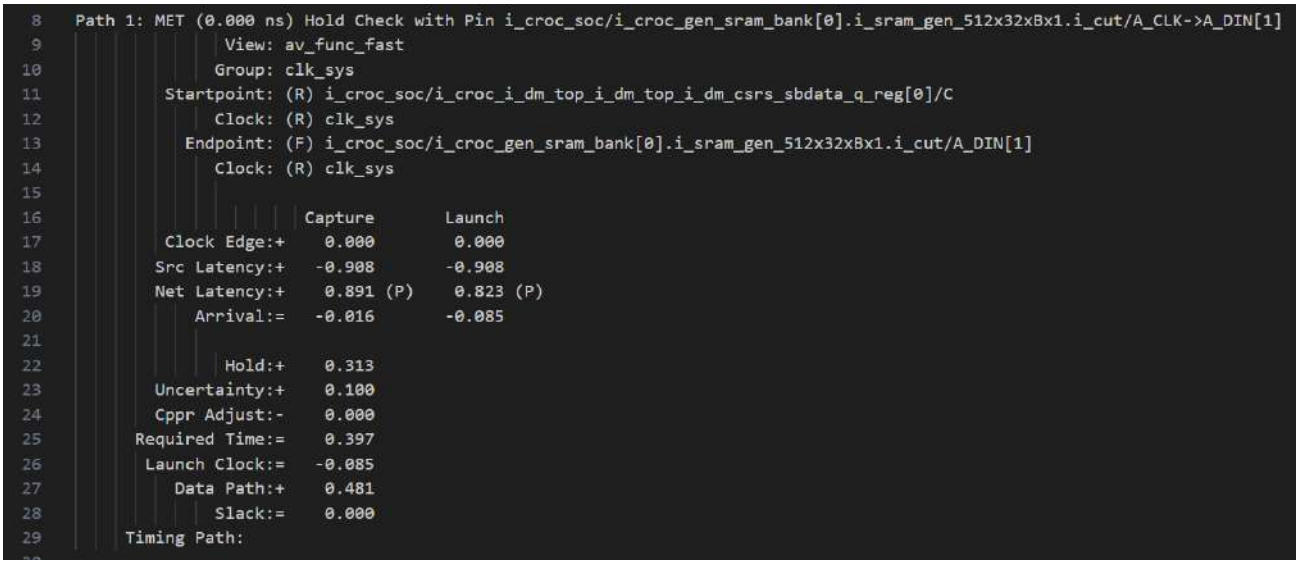
The WNS setup is negative, meaning the setup after the post-CTS stage is still violated. It should be improved at *route* stage.

```
Path 1: VIOLATED (-0.105 ns) Late Output Delay Assertion
           View: av_func_slow
          Group: clk_jtg
     Startpoint: (R) i_croc_soc/i_croc_i_dmi_jtag_i_dmi_jtag_tap/td_o_reg/C
          Clock: (F) clk_jtg
       Endpoint: (R) jtag_tdo_o
          Clock: (R) clk_jtg

                    Capture       Launch
     Clock Edge:+    20.000       10.000
    Src Latency:+     0.000       -1.533
    Net Latency:+     0.000 (P)    1.429 (P)
        Arrival:=    20.000        9.896

   Output Delay:-     5.000
    Uncertainty:-     0.100
    Cppr Adjust:+     0.000
  Required Time:=    14.900
   Launch Clock:=     9.896
      Data Path:+     5.110
          Slack:=    -0.105
```

Figure 4.6 – Post-CTS worst setup time



```
Path 1: MET (0.000 ns) Hold Check with Pin i_croc_soc/i_croc_gen_sram_bank[0].i_sram_gen_512x32xBx1.i_cut/A_CLK->A_DIN[1]
              View: av_func_fast
             Group: clk_sys
        Startpoint: (R) i_croc_soc/i_croc_i_dm_top_i_dm_top_i_dm_csrs_sbdata_q_reg[0]/C
             Clock: (R) clk_sys
          Endpoint: (F) i_croc_soc/i_croc_gen_sram_bank[0].i_sram_gen_512x32xBx1.i_cut/A_DIN[1]
             Clock: (R) clk_sys

                      Capture       Launch
       Clock Edge:+    0.000        0.000
      Src Latency:+   -0.908       -0.908
      Net Latency:+    0.891 (P)    0.823 (P)
          Arrival:=   -0.016       -0.085

             Hold:+    0.313
      Uncertainty:+    0.100
      Cppr Adjust:-    0.000
     Required Time:=    0.397
      Launch Clock:=   -0.085
         Data Path:+    0.481
             Slack:=    0.000
  Timing Path:
```

Figure 4.7 – Post-CTS worst hold time

4.4 Routing Results

A *post-route* view of the SoC in Cadence Innovus is shown in the figure below.

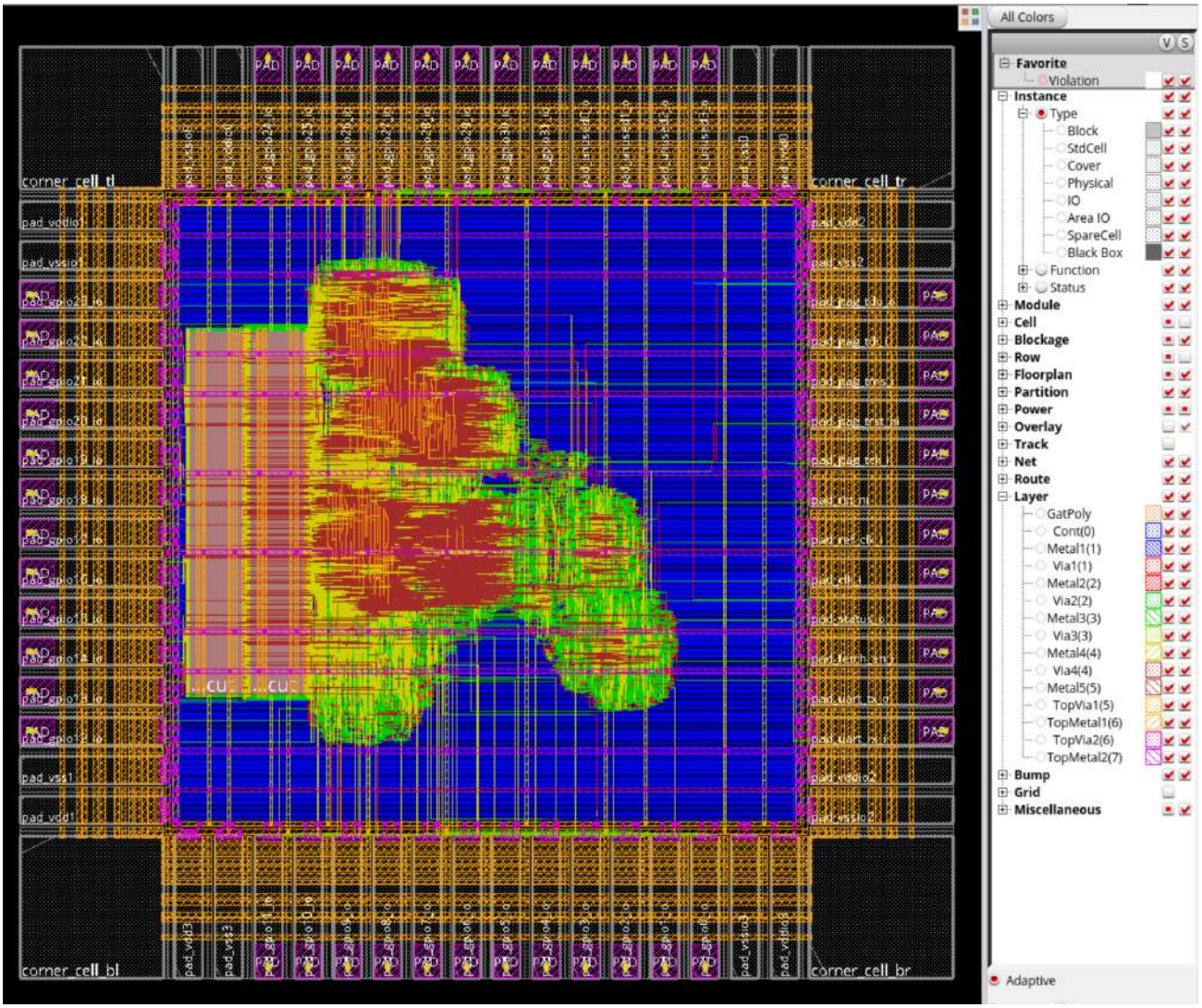


Figure 4.8 – Croc SoC post-rout view

4.4.1 Post-routing timings

After the post-rout stage, the worst hold, and setup timings are:

- Setup time worst: -0.099 ns (violated)
- Hold time worst: 0.001 ns (met)

Compared with the post-CTS results (setup WNS: -0.105 ns; hold WNS: 0.0 ns), both metrics improved at the post-route stage. But still, the setup is violated.

```
Path 1: VIOLATED (-0.099 ns) Late Output Delay Assertion
           View: av_func_slow
          Group: clk_jtg
     Startpoint: (R) i_croc_soc/i_croc_i_dmi_jtag_i_dmi_jtag_tap/td_o_reg/C
          Clock: (F) clk_jtg
       Endpoint: (R) jtag_tdo_o
          Clock: (R) clk_jtg

                    Capture       Launch
     Clock Edge:+    20.000       10.000
    Src Latency:+     0.000       -1.533
    Net Latency:+     0.000 (P)    1.449 (P)
        Arrival:=    20.000        9.916

   Output Delay:-     5.000
    Uncertainty:-     0.100
    Cppr Adjust:+     0.000
  Required Time:=    14.900
   Launch Clock:=     9.916
      Data Path:+     5.083
          Slack:=    -0.099
```

Figure 4.9 – Post-route worst setup time

```
Path 1: MET (0.001 ns) Hold Check with Pin i_croc_soc/i_croc_gen_sram_bank[1].i_sram_gen_512x32xBx1.i_cut/A_CLK->A_DIN[3]
              View: av_func_fast
             Group: clk_sys
        Startpoint: (R) i_croc_soc/i_croc_i_dm_top_i_dm_top_i_dm_csrs_sbdata_q_reg[1]/C
             Clock: (R) clk_sys
          Endpoint: (F) i_croc_soc/i_croc_gen_sram_bank[1].i_sram_gen_512x32xBx1.i_cut/A_DIN[3]
             Clock: (R) clk_sys

                      Capture       Launch
       Clock Edge:+    0.000        0.000
      Src Latency:+   -0.908       -0.908
      Net Latency:+    0.866 (P)    0.807 (P)
          Arrival:=   -0.041       -0.101

             Hold:+    0.299
      Uncertainty:+    0.100
      Cppr Adjust:-    0.000
     Required Time:=   0.358
      Launch Clock:=  -0.101
        Data Path:+    0.459
             Slack:=   0.001
  Timing Path:
```

Figure 4.10 – Post-route worst hold time

The next step in setup time violations is to find out if there are any other violated paths and where they are located. According to the report from *postroute/setup.analysis_summary.rpt* TNS is equal to WNS (see screenshot below), which means that there is only one violated path.

```
# SETUP                   WNS     TNS   FEP
#-------------------------------------------
  View : ALL            -0.099  -0.099     1
     Group : Async       1.484   0.000     0
     Group : in2out        N/A     N/A     0
     Group : reg2out    -0.099  -0.099     1
     Group : in2reg      6.985     0.0     0
     Group : reg2reg     0.182     0.0     0
```

Figure 4.11 – Post-route setup WNS and TNS

According to the post-route report, the critical path is within the JTAG logic, as depicted below.

```
#-------------------------------------------------------------------------------------------------------------------------
# Timing Point                                          Flags  Arc         Edge  Cell         Fanout  Trans   Delay  Arrival
#                                                                                                    (ns)    (ns)    (ns)
#-------------------------------------------------------------------------------------------------------------------------
  jtag_tck_i                                           -      jtag_tck_i  F     (arrival)         1  0.004   0.000   8.467
  jtag_tck_i                                           -      -           F     (net)             1      -       -       -
  pad_jtag_tck_i/DOUT                                  -      PAD->DOUT   F     ixc013_i16x       1  0.004   1.062   9.529
  soc_jtag_tck_i                                       -      -           F     (net)             1      -       -       -
  i_croc_soc/CTS_cid_buf_00072/Q                       -      A->Q        F     BUJIX8            4  0.094   0.184   9.713
  i_croc_soc/CTS_62                                    -      -           F     (net)             4      -       -       -
  i_croc_soc/i_croc_i_dmi_jtag_i_dmi_jtag_tap/g3268/Q  -      A->Q        R     INJIX0            1  0.120   0.203   9.916
  i_croc_soc/i_croc_i_dmi_jtag_i_dmi_jtag_tap/CTS_1    -      -           R     (net)             1      -       -       -
  i_croc_soc/i_croc_i_dmi_jtag_i_dmi_jtag_tap/td_o_reg/Q -    C->Q        R     DFRRQJIX1         1  0.190   0.471   10.387
  soc_jtag_tdo_o                                       -      -           R     (net)             1      -       -       -
  postroute_FE_PSBC1_soc_jtag_tdo_o/Q                  -      A->Q        F     INVJIX2           1  0.091   0.080   10.467
  FE_PSBN1_soc_jtag_tdo_o                              -      -           F     (net)             1      -       -       -
  postroute_FE_PSBC2_soc_jtag_tdo_o/Q                  -      A->Q        R     INJIX8            1  0.055   0.085   10.552
  FE_PSBN2_n                                           -      -           R     (net)             1      -       -       -
  pad_jtag_tdo_o/PAD                                   -      DIN->PAD    R     ixc013_b16m       2  0.126   4.447   14.999
  jtag_tdo_o                                           -      -           R     (net)             2      -       -       -
  jtag_tdo_o                                           -      jtag_tdo_o  R     -                 2  1.477   0.000   14.999
#-------------------------------------------------------------------------------------------------------------------------
```

Figure 4.12 – Post-route setup path violation

All of this means that the Croc project with the Ibex core, BBS32, does not have setup violations and can operate at the specified frequency in the constraint file, 80 MHz, even for the *av_func_slow* view.

To avoid any violations caused by the JTAG critical path, the user can use a lower frequency than specified (less than 12.5 MHz).

4.4.2 CTS visualization

Below is shown a clock tree visualization in Innovus after routing was finished.

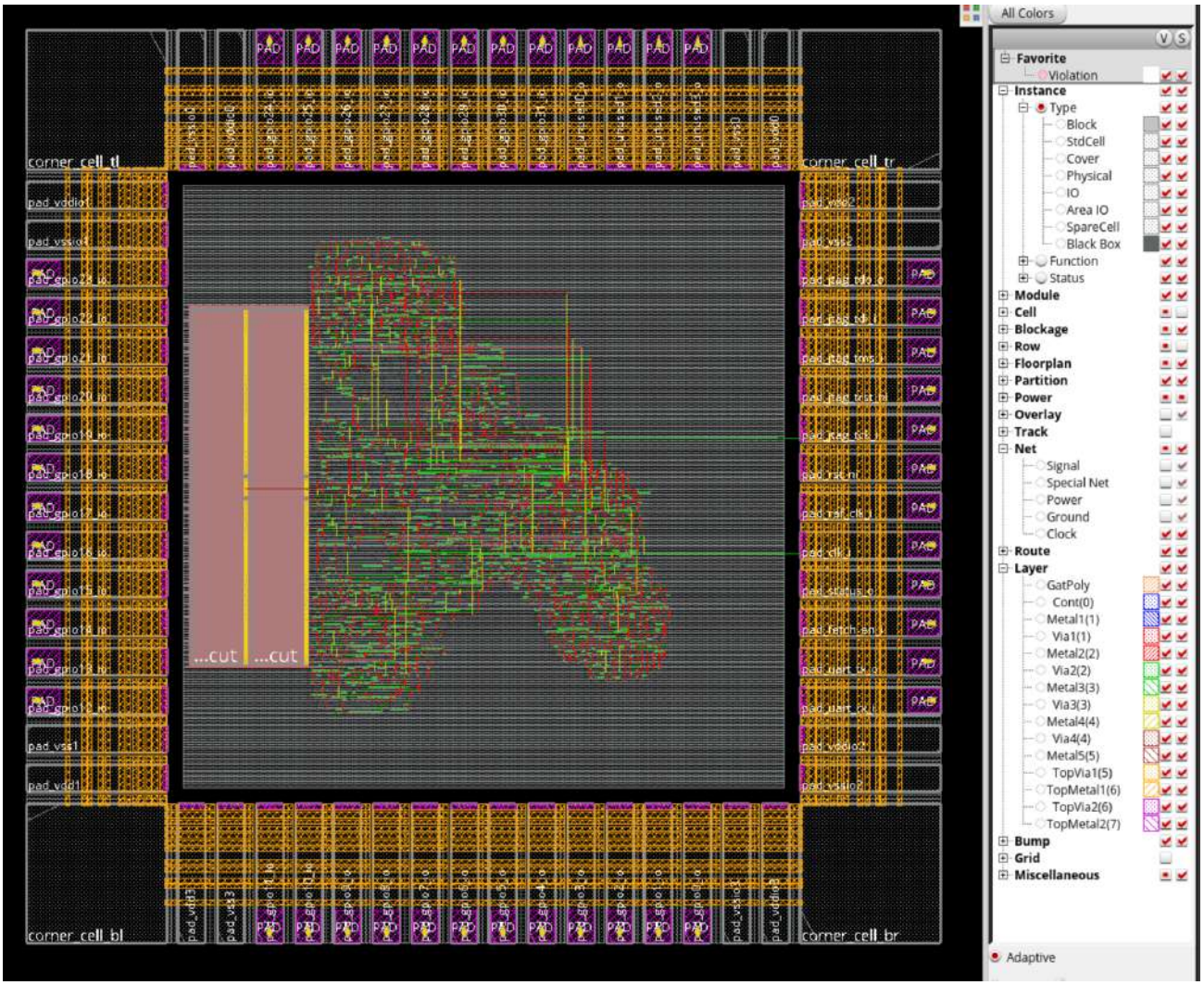


Figure 4.13 – Croc SoC clock tree visualization

4.4.3 Occupied area visualization

Below are some visualizations of occupied area for basic Corc SoC modules, including the BBS32 generator.

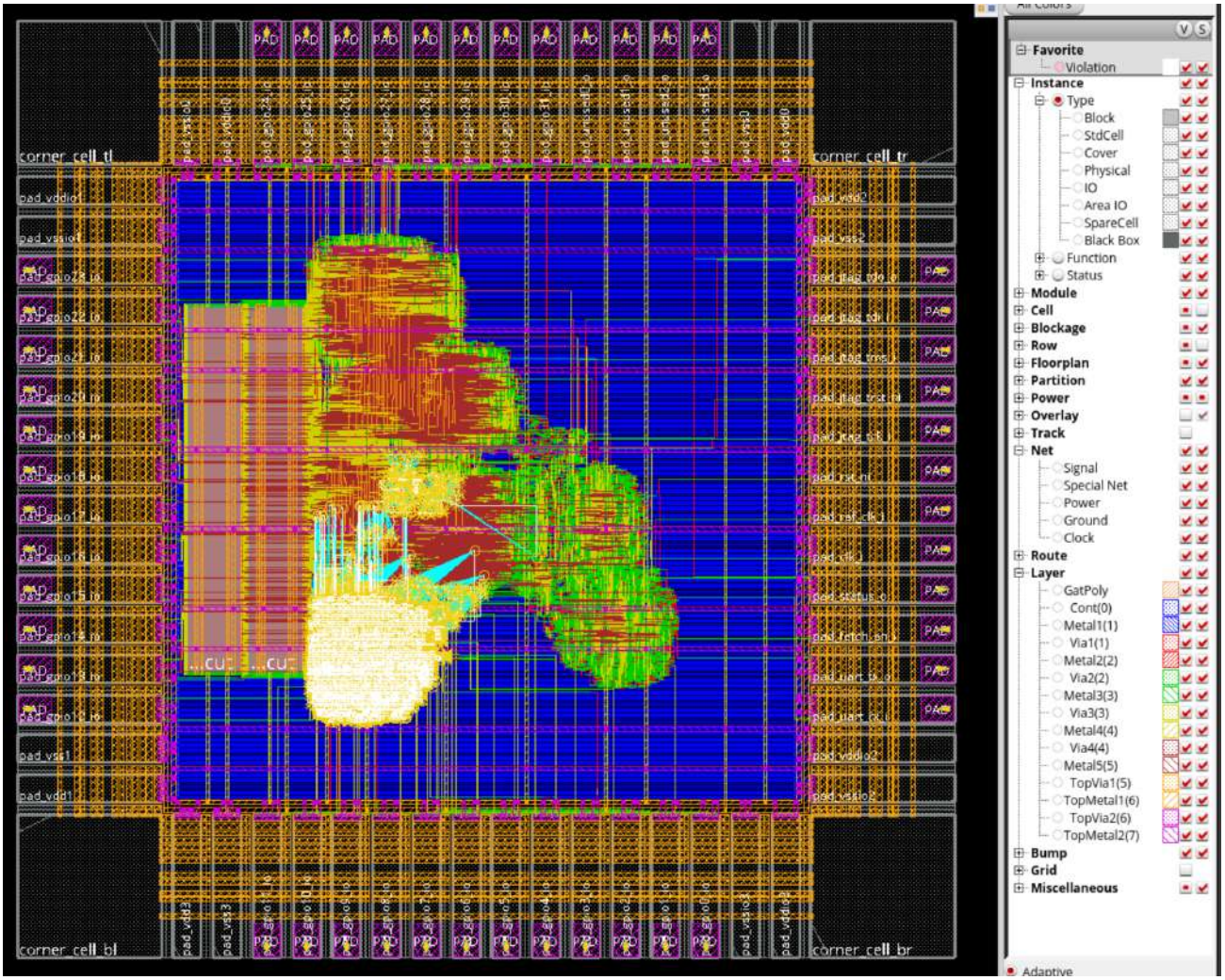


Figure 4.14 – Area occupied by the BBS32 generator

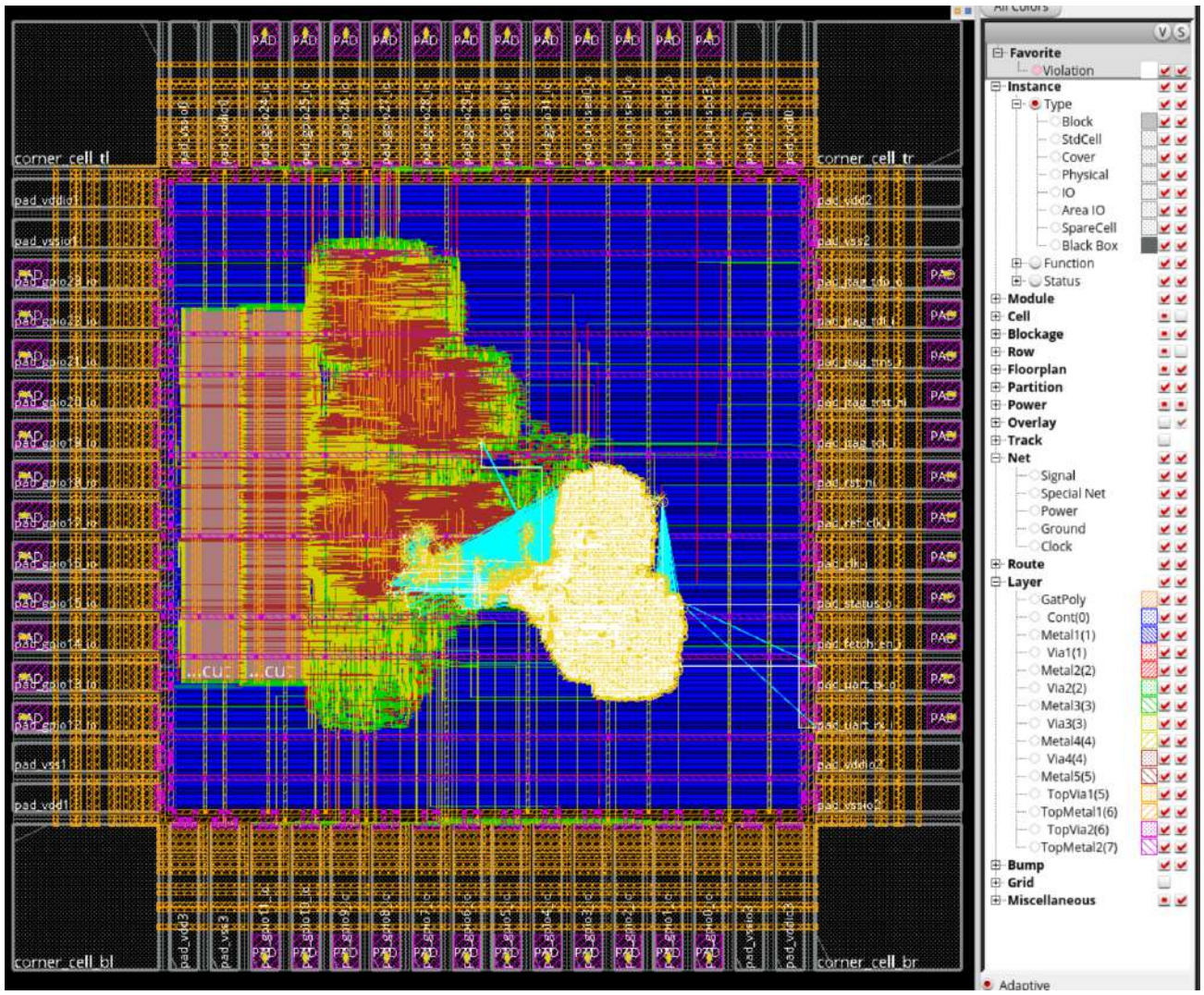


Figure 4.15 – Area occupied by UART

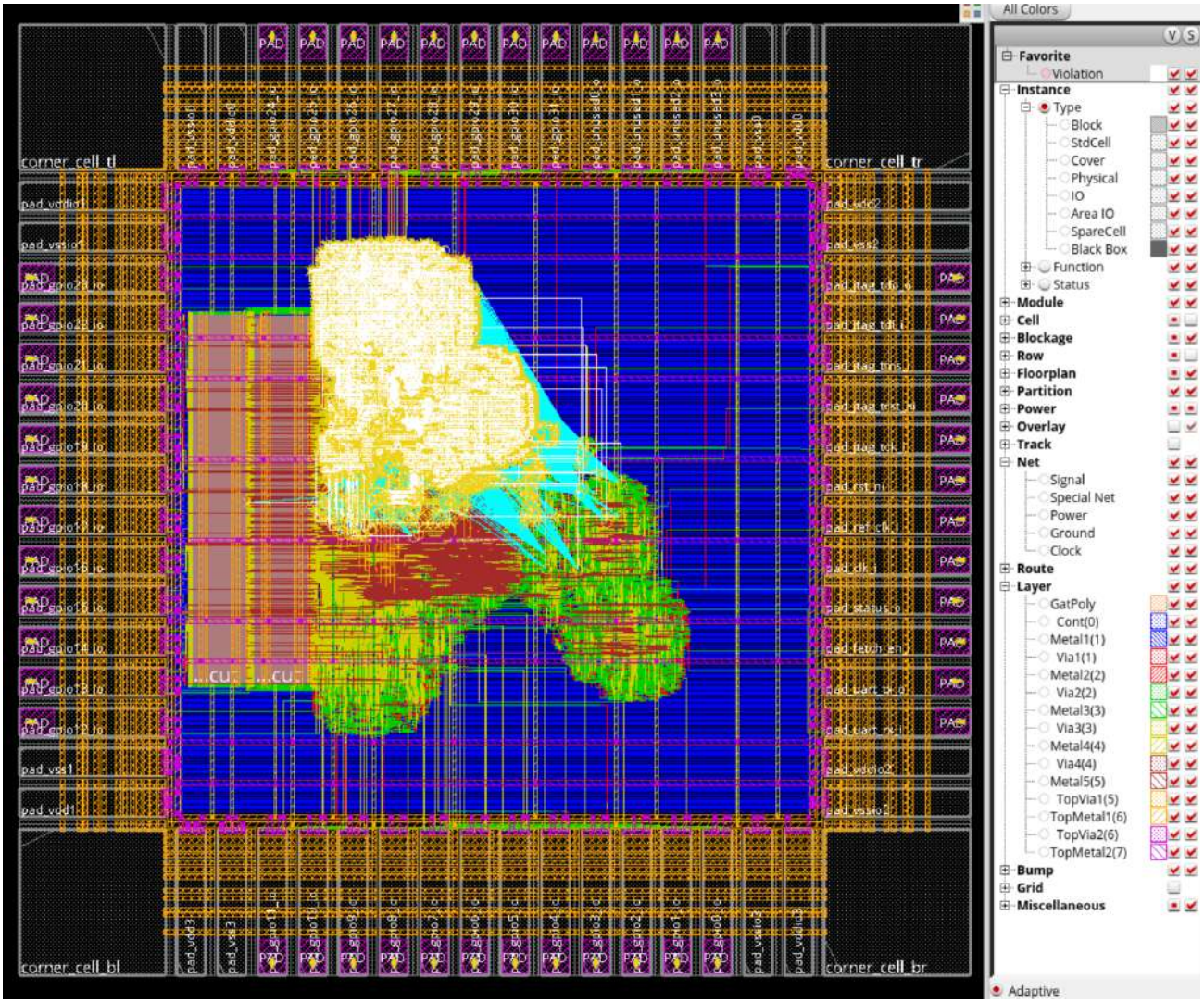


Figure 4.16 – Area occupied by Ibex core

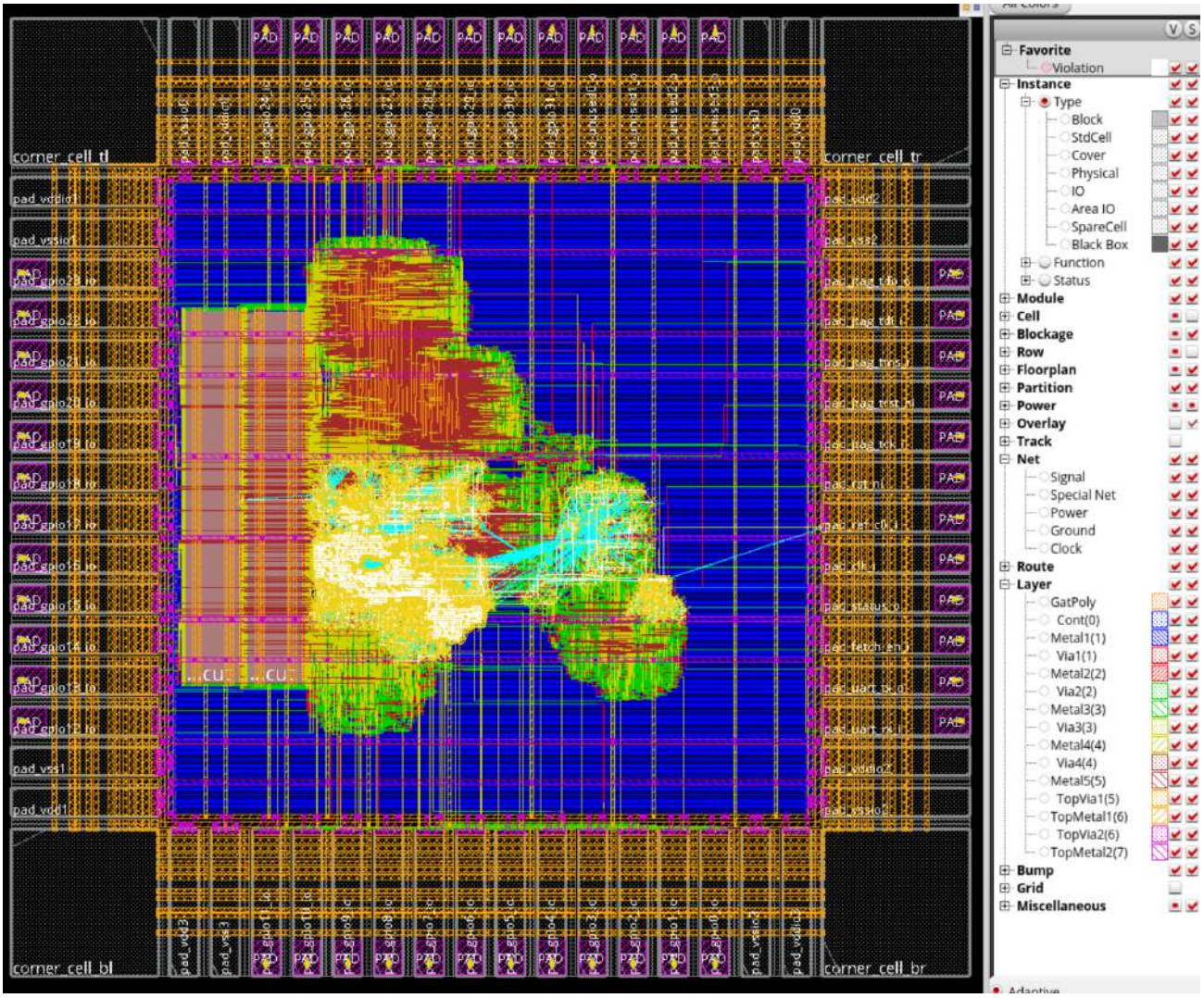


Figure 4.17 – Area occupied by Timer

4.4.4 Timing map

Below is shown a Timing heat map visualization for post-route setup and hold timings.

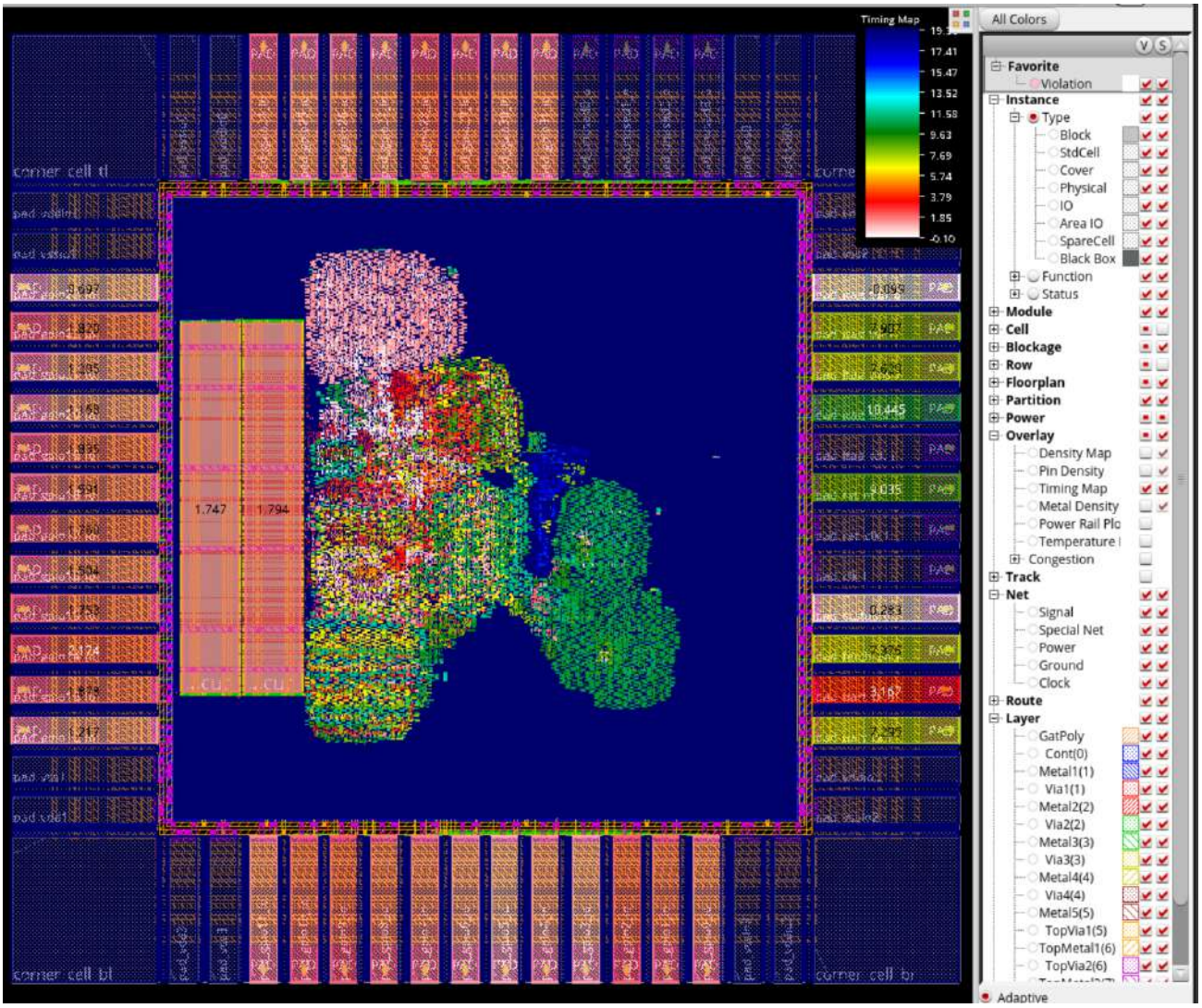


Figure 4.18 – Post-route setup timing heat map

The lowest setup time margin is in the area occupied by the Ibex core. BBS32 has a medium setup margin.

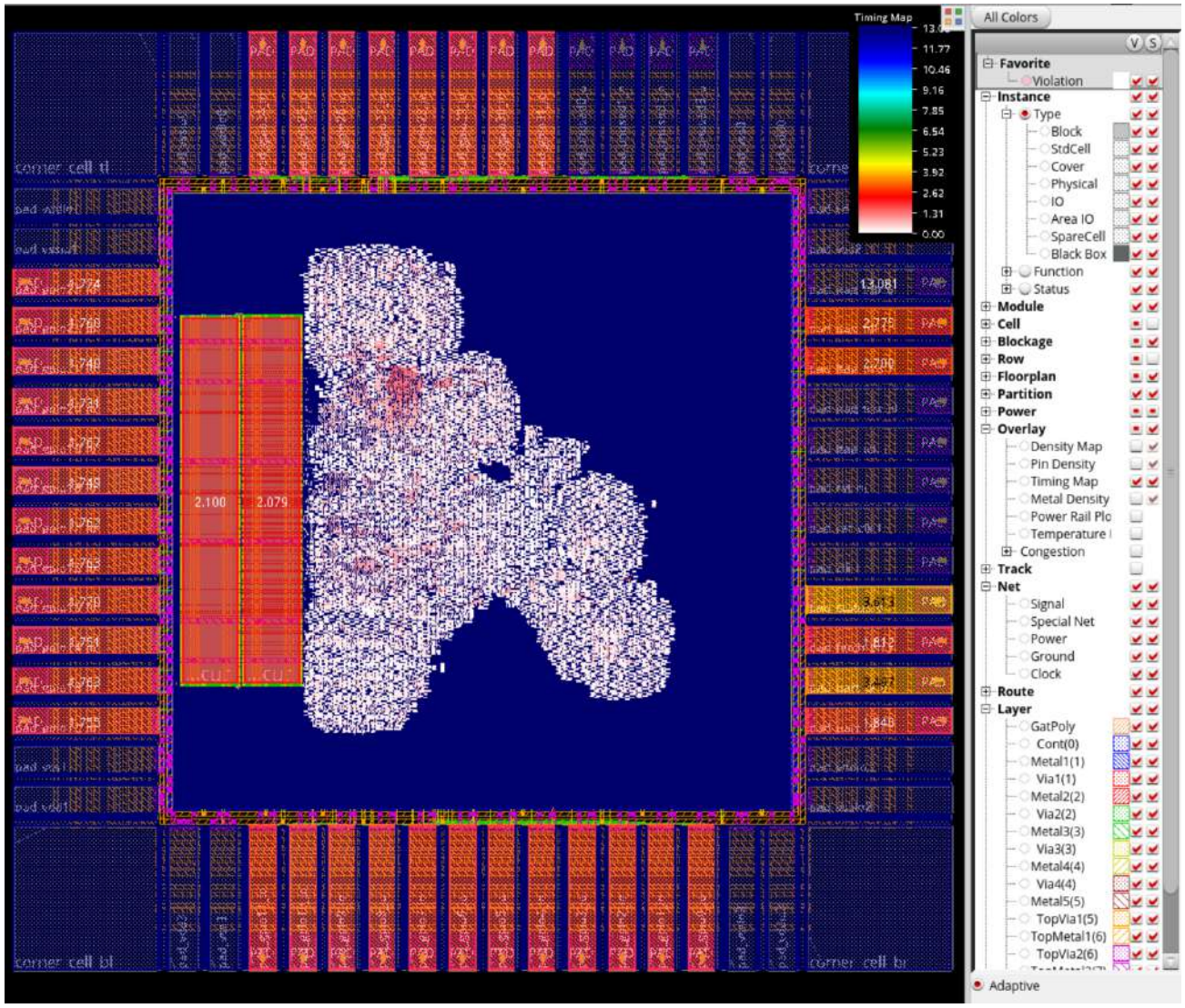


Figure 4.19 – Post-route hold timing heat map

According to the hold timing heat map, there are no hold violations.

4.4.5 Route congestion heat map

Below is shown a heat map for routing congestion. It uses results from the post-route step to show where routing resources are used most.

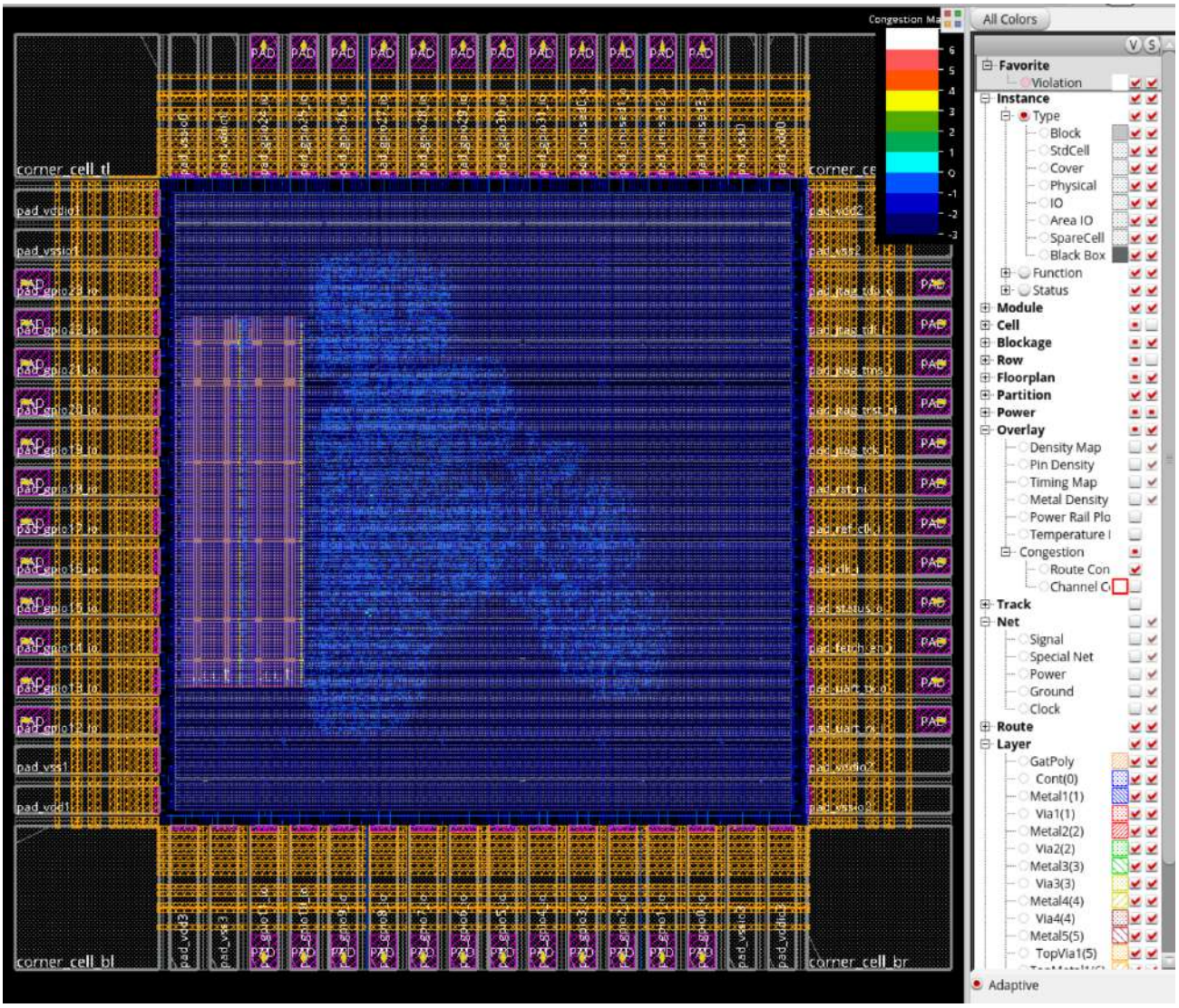


Figure 4.20 – Croc SoC congestion heat map

As can be seen in the screenshot above, in the current implementation, the routing resource utilization is low, with only a few points showing medium routing congestion.

4.4.6 Post-route DRC

The entire post-route project was checked using the Innovus GUI menu, as shown below.

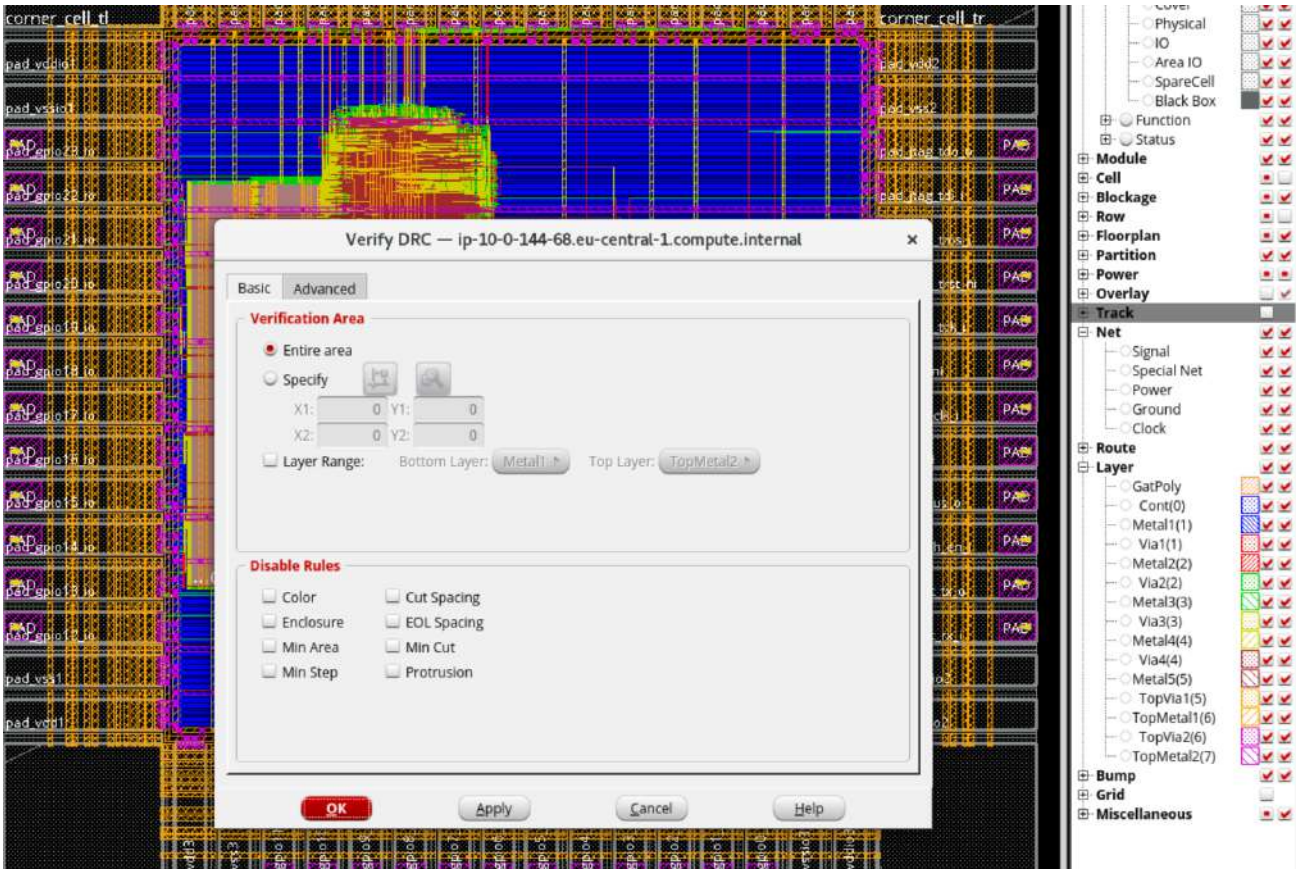


Figure 4.21 – Innovus DRC check menu

The result was stored in the file *croc_chip.drc.rpt*. Its content is shown below.

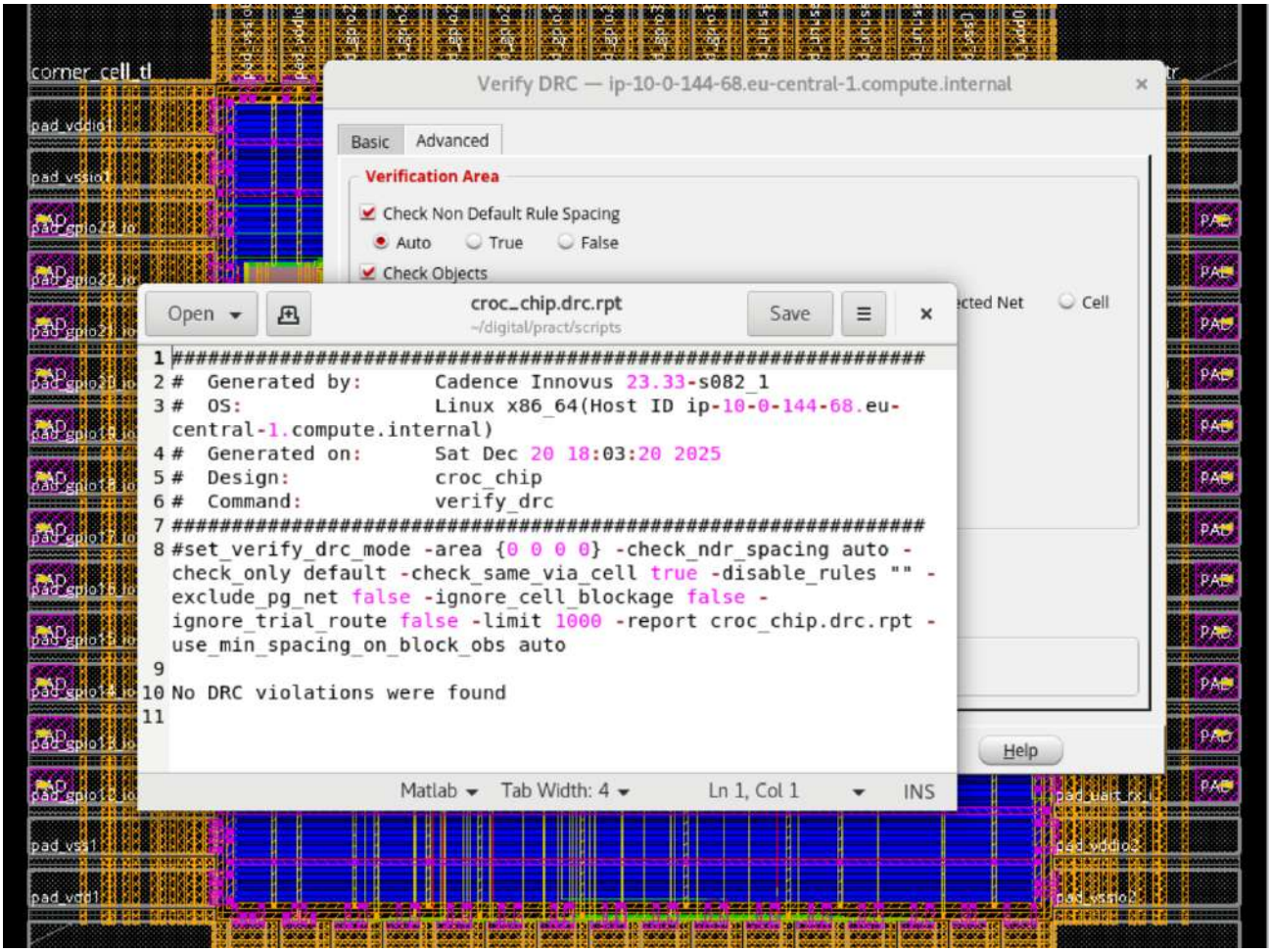


Figure 4.22 – DRC result after post-route stage

No DRC violations were found.

4.4.7 Post-route power estimation

This chapter contains power estimation from Innovus command *report_power* using statistical power analysis:

- Do not have a user-defined activity file;
- Set sequential global activity to 20%;
- Set primary input activity to 20%.

For this configuration, the received values are depicted in screenshots below (power unit is 1 mW).

```
Total Power
--------------------------------------------------------------------
Total Internal Power:       31.00281891         0.5347%
Total Switching Power:      43.66855835         0.7531%
Total Leakage Power:      5723.54110313        98.7122%
Total Power:              5798.21248039
--------------------------------------------------------------------
```

Figure 4.23 – Post-route total power estimation

Group	Internal Power	Switching Power	Leakage Power	Total Power	Percentage (%)
Sequential	7.992	0.8033	0.01039	8.806	0.1519
Macro	2.981	0.001003	5724	5727	98.76
IO	18.68	39.16	0.003511	57.84	0.9976
Combinational	1.074	1.846	0.008369	2.928	0.0505
Clock (Combinational)	0.2801	1.855	0.0002236	2.135	0.03683
Clock (Sequential)	0	0	0	0	0
Total	31	43.67	5724	5798	100

Figure 4.24 – Post-route power estimation distribution between groups

The most significant leakage power has group “Macro” which consists of two SRAM hard macros. For the current analysis, this can be ignored because statistical analysis assumes that all circuit elements are identical and toggle at the configured rate. This assumption is not valid for all hard macro and for SRAM too.

Re-calculated percentages excluding SRAM macros are shown in the table below.

Table 4.4 – Statistical power estimation excluding SRAM

Group	Total power, mW	Percentage (%)
Sequential	8.81	12.63
IO	55.84	80.1
Combinational	2.93	4.20
Clock (Combinational)	2.14	3.06
Clock (Sequential)	0.00	0
Total	69.71	100

The IO group consumes the most power from analysis. In a real application, this may not hold because different scenarios (programs) will produce different I/O toggling rates. To estimate power consumption more precisely, a user-defined activity file or a VCD file from a simulation for a particular scenario is usually used.

5 Conclusions

This thesis presented a proof-of-concept implementation of a hardware Blum-Blum-Shub (BBS) pseudorandom generator embedded into the Croc SoC, together with a practical RTL-to-GDSII workflow based on Cadence tools. The selected configuration uses a 32-bit data path, intentionally chosen to enable end-to-end validation of the hardware microarchitecture, OBI-based bus integration, and the physical design setup within typical thesis time constraints. While cryptographic guidance recommends modulus sizes on the order of 2048 bits or more to resist factorization, the 32-bit configuration is sufficient to demonstrate the complete design methodology and to highlight how security fundamentally depends on parameter size.

At the SoC level, the generator was integrated into the Croc user domain as an OBI subordinate mapped into the system address space (UserBaseAddr = 0x2000_0000). This approach follows a standard accelerator/peripheral integration pattern: the core configures the generator via memory-mapped registers and retrieves the generated output through the same interface.

At the microarchitectural level, the design decomposes BBS operation into a small number of deterministic hardware blocks orchestrated by a main FSM. The implementation includes an OBI register interface, a shift-and-add multiplier, a dedicated modulo unit, and a shift register for assembling output bits. The FSM supports multiple operating modes, enabling flexible software-driven use without modifying hardware.

The arithmetic blocks were selected to favor clarity and controllable timing over peak throughput. In particular, a shift-and-add multiplication method was used to reduce combinational complexity and area, with the explicit assumption that longer computation latency is acceptable for the intended proof-of-concept and validation-focused use case.

Functional correctness was validated at both the unit and SoC levels. Unit simulations covered the multiplier and modulo blocks. At the same time, SoC-level validation was performed by running a modified Croc “Hello world” firmware that programs the BBS registers through a C library, and starts generation. Checks returned results against expected (“golden”) values. The simulation log indicates that the produced results match the expected outputs for the tested configuration, supporting correct integration and register-level behavior.

To provide an initial statistical screening of the generated bitstream, two tests from NIST SP 800-22 were applied: the Frequency (Monobit) test and the Discrete Fourier Transform (Spectral) test. The data for this analysis was produced by a dedicated testbench that generated 2048 output bits using a fixed parameter set. The resulting *P-values* for both tests exceeded the 0.01 threshold, and therefore the sequence was not rejected as non-random by these two detectors.

These outcomes should be interpreted as evidence that the implemented generator does not exhibit specific fundamental imbalances (monobit) or strong periodic artifacts detectable by the spectral test on the evaluated sample. However, statistical testing remains a screening tool rather than proof: NIST’s recommended methodology involves testing sets of sequences, analyzing the proportion of passing sequences, and checking the distribution of *P-values* for uniformity. Consequently, stronger empirical claims would require more extended sequences, multiple independent samples across different seeds (and potentially different keys), and a broader test selection.

The primary limitation is the deliberate reduction of BBS parameters to 32-bit width. This makes the design suitable for architectural demonstration but unsuitable for practical cryptographic deployment, because the modulus can be factored efficiently at this size.

Finally, a complete RTL-to-GDSII signoff typically includes quantitative reporting of power analysis using simulation or a user-defined activity file, and IR-drop margins. Where these analyses are not yet finalized, they should be treated as future work required to convert the demonstrated flow setup and floorplanning strategy into a tapeout-ready implementation.

6 Bibliography

[1] Donald Ervin Knuth “The art of computer programming. Volume 2 Seminumerical Algorithms”, 3rd ed, 1998, Addison Wesley Longman, pp.10-26.

[2] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). “Handbook of Applied Cryptography”. CRC Press. Chapter 5. Pseudorandom Bits and Sequences, pp. 169-190. [Online]. Available: https://cacr.uwaterloo.ca/hac/about/chap5.pdf

[3] Blum, L., Blum, M., & Shub, M. (1986). "A Simple Unpredictable Pseudo-Random Number Generator." SIAM Journal on Computing, 15(2), 364–383. [Online]. Available: https://epubs.siam.org/doi/10.1137/0215025. Accessed: 2025-11-11.

[4] Vazirani, U. V., & Vazirani, V. V. (1984). "Efficient and Secure Pseudo-Random Number Generation." 25th Annual Symposium on Foundations of Computer Science, 193-202. [Online]. Available: https://ieeexplore.ieee.org/document/715948. Accessed: 2025-11-11.

[5] “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”. NIST Special Publication 800-22 Revision 1a, April 2010.

[6] Croc System-on-Chip. [Online]. Available: https://github.com/pulp-platform/croc

[7] “The RISC-V Instruction Set Manual Volume I”. Unprivileged Architecture. RISC-V International. Version 20250508. Chapter 6, pp. 69-72. [Online]. Available: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf

[8] “OBI 1”. Silicon Labs, Inc. v1.6.0, 3/15/2023. [Online]. Available: https://github.com/openhwgroup/obi/blob/072d9173c1f2d79471d6f2a10eae59ee387d4c6f/OBI-v1.6.0.pdf

[9] Sarah L. Harris, David Harris “Digital Design and Computer Architecture. RISC-V Edition”. Elsevier Inc, 2022, pp. 393-541

[10] Hubert Kaeslin “Top-Down Digital VLSI Design From Architectures to Gate-Level Circuits and FPGAs”. Microelectronics Design Center Department of Information Technology and Electrical Engineering ETH Zürich Switzerland. Published by Elsevier, 2015.

[11] “Genus Physical Guide”. Product Version 23.14. Cadence Design Systems, 2025.

[12] “Innovus Stylus Common UI User Guide”. Product Version 23.15. Cadence Design Systems, 2025.

[13] “Innovus Stylus Common UI Text Command Reference”. Product Version 23.15. Cadence Design Systems, 2025.

[14] “Innovus User Guide”. Product Version 23.15. Cadence Design Systems, 2025.

[15] “Innovus Foundation Flows Guide”. Product Version 23.10. Cadence Design Systems, 2023.

[16] Prof. Adam Teman “RTL2GDS Demo Part 4: Place and Route on a simple block”. EnICSLabs, Bar Ilan University, 2025. [Online]. Available: https://enicslabs.com/wp-content/uploads/2025/10/04-Place-and-Route.pdf

[17] Prof. Sung Kyu Lim “Tutorial I: Cadence Innovus”. ECE6133: Physical Design Automation of VLSI Systems. Georgia Institute of Technology, 2/7/2022. [Online]. Available: https://limsk.ece.gatech.edu/course/ece6133/lab/innovus.pdf

[18] T. Manikas “Tutorial for Cadence Innovus Place & Route”. Southern Methodist University, 2/26/2019. [Online]. Available: https://s2.smu.edu/~manikas/CAD_Tools/CPR/lib/Innovus_PR_Tutorial2019Feb.pdf

[19] João Vaz “Introduction to Innovus place & route (version 1.2021)”. Sistemas Eletrónicos Integrados (Period 1), Instituto Superior Técnico 2021/2022. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382296398/tutorial_innovus_PR_2021.pdf

[20] DEF file in VLSI Design | Data Exchange Format. [Online]. Available: https://teamvlsi.com/2020/08/def-file-in-vlsi-design-exchange.html

[21] Luca Benini, Frank K. Gürkaynak “Floorplanning”, 26 March 2025. [Online]. Available: https://vlsi.ethz.ch/w/images/9/9f/06_DEF.pdf.

[22] “IHP SG13G2 Process Specification Rev. 1.4”, 2023-10-17.

[23] “SG13G2 Layout Rules Rev. 2.0”, 2024-03-08.

[24] “ixc013g2ng_stdcell_typ_1p20V_25C Library”

[25] “IHP IO Pad Cells IXC013. Architecture Documentation” Rev. 1.0, 2020-04-21

Appendix A. BBS generator control & Status Register (CSR)

Implemented the BBS generator that operates with 8 OBI registers. They are described below.

Table 6.1 – Register map

Relative byte address	Register name
0x00	P value
0x04	Q value
0x08	Seed value
0x0C	Control register
0x10	M 64-bit value (LSB)
0x14	M 64-bit value (MSB)
0x18	Output (result) value
0x1C	Status register

Table 6.2 – Register #0, P value

Bits	Name	Read / Write	Description	Default value
31:0	P[31:0]	R/W	Value of P	0

Table 6.3 – Register #1, Q value

Bits	Name	Read / Write	Description	Default value
31:0	Q[31:0]	R/W	Value of Q	0

Table 6.4 – Register #2, Seed value

Bits	Name	Read / Write	Description	Default value
31:0	SEED[31:0]	R/W	Value of SEED	0

Table 6.5 – Register #3, Control

Bits	Name	Read / Write	Description	Default value
0	START	R/W	Start calculation. Set to “1” to start	0
1	KEEP_M	R/W	Skip M calculation, configuration only valid if M has been calculated previously	0
2	USE_XNEXT	R/W	Use XNEXT value to calculate a random value. Otherwise, use the SEED value. Configuration only valid if M has been calculated previously and KEEP_M is set to “0”	0
31:3	Reserved[28:0]	-	Always “0”	0

Table 6.6 – Register #4, M value (LSB)

Bits	Name	Read / Write	Description	Default value
31:0	M[31:0]	R	Calculated M value, LSB	0

Table 6.7 – Register #5, M value (MSB)

Bits	Name	Read / Write	Description	Default value
31:0	M[63:32]	R	Calculated M value, MSB	0

Table 6.8 – Register #6, output random word

Bits	Name	Read / Write	Description	Default value
31:0	RAND_WORD[31:0]	R	Calculated a random 32-bit word	0

Table 6.9 – Register #7, status register

Bits	Name	Read / Write	Description	Default value
0	M_VALID	R	Will be set to “1” if M value is valid	0
7:1	Reserved[6:0]	R	Reserved, always “0”	0
8	RESULT_VALID	R	Will be set to “1” if the Result is ready (valid)	0
31:9	Reserved [22:0]	R	Always “0”	0

Appendix B. Library on C

There are several C files:

- bbs32.h: header where defined functions, prototypes, and defines need to work with registers;
- bbs32.c: contains implemented C functions;
- config.h: need to be modified to work with the implemented module “bbs32”;
- helloworld.c: modified Croc default C program

File “bbs32.h”

```
#pragma once

#include <stdint.h>
#include "config.h"


#define BBS32_P_REG_OFFSET          0x000
#define BBS32_Q_REG_OFFSET          0x004
#define BBS32_SEED_REG_OFFSET       0x008
#define BBS32_CONTROL_REG_OFFSET    0x00C
#define BBS32_M_LSB_REG_OFFSET      0x010
#define BBS32_M_MSB_REG_OFFSET      0x014
#define BBS32_RESULT_REG_OFFSET     0x018
#define BBS32_STATUS_REG_OFFSET     0x01C

#define BBS32_STATUS_M_VALID        ((uint32_t)0x01u)
#define BBS32_STATUS_RESULT_VALID   ((uint32_t)(1u << 8u))

#define BBS32_CONTROL_START         ((uint32_t)0x01u)
#define BBS32_CONTROL_KEEP_M        ((uint32_t)0x02u)
#define BBS32_CONTROL_USE_XNEXT     ((uint32_t)0x04u)

//Write
void bbs32_write_pq(uint32_t p, uint32_t q);
void bbs32_write_seed(uint32_t seed);

//Control
void bbs32_first_run();
void bbs32_next_run();
void bbs32_updated_seed_run();
void bbs32_updated_all_run();
void bbs32_wait_ready();
void bbs32_wait_not_ready();

//Read
uint32_t bbs32_read_result();
//For debug only:
uint32_t bbs32_read_seed();
```

File “bbs32.c”

```
#include "bbs32.h"
#include "util.h"
#include "config.h"

//Write
void bbs32_write_pq(uint32_t p, uint32_t q)
{
    *reg32(BBS32_BASE_ADDR, BBS32_P_REG_OFFSET) = p;
    *reg32(BBS32_BASE_ADDR, BBS32_Q_REG_OFFSET) = q;
}

void bbs32_write_seed(uint32_t seed)
{
    *reg32(BBS32_BASE_ADDR, BBS32_SEED_REG_OFFSET) = seed;
}

//P,Q,SEED should be already written to the registers
void bbs32_first_run() {
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = 0x0;
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = BBS32_CONTROL_START;
    bbs32_wait_not_ready();
}

void bbs32_next_run() {
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = (BBS32_CONTROL_START |
BBS32_CONTROL_KEEP_M | BBS32_CONTROL_USE_XNEXT);
    bbs32_wait_not_ready();
}

void bbs32_updated_seed_run() {
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = (BBS32_CONTROL_START |
BBS32_CONTROL_KEEP_M);
    bbs32_wait_not_ready();
}

void bbs32_updated_all_run() {
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = (BBS32_CONTROL_START);
    bbs32_wait_not_ready();
}

void bbs32_wait_ready() {
    uint32_t tmp;
    do {
        tmp = *reg32(BBS32_BASE_ADDR, BBS32_STATUS_REG_OFFSET);
        tmp &= (BBS32_STATUS_M_VALID | BBS32_STATUS_RESULT_VALID);
    } while (tmp != (BBS32_STATUS_M_VALID | BBS32_STATUS_RESULT_VALID));
    //
    tmp = *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET);
    tmp &= ~BBS32_CONTROL_START;
    *reg32(BBS32_BASE_ADDR, BBS32_CONTROL_REG_OFFSET) = tmp;
}

void bbs32_wait_not_ready()
{
    uint32_t tmp;
    do {
        tmp = *reg32(BBS32_BASE_ADDR, BBS32_STATUS_REG_OFFSET);
        tmp &= BBS32_STATUS_RESULT_VALID;
    } while (tmp == BBS32_STATUS_RESULT_VALID);
}

//Read
uint32_t bbs32_read_result()
{
    return (*reg32(BBS32_BASE_ADDR, BBS32_RESULT_REG_OFFSET));
}
```

File “config.h”

```
// Copyright 2022 ETH Zurich and University of Bologna.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0
//
// Nils Wistoff <nwistoff@iis.ee.ethz.ch>
// Paul Scheffler <paulsc@iis.ee.ethz.ch>

#pragma once

// Address map
#define SOCCTRL_BASE_ADDR 0x03000000
#define UART_BASE_ADDR    0x03002000
#define GPIO_BASE_ADDR    0x03005000
#define TIMER_BASE_ADDR   0x0300A000
#define BBS32_BASE_ADDR   0x20000000


// Frequencies
#define TB_FREQUENCY 20000000
#define TB_BAUDRATE    115200

// Peripheral configs
// UART
#define UART_BYTE_ALIGN 4
#define UART_FREQ       TB_FREQUENCY
#define UART_BAUD       TB_BAUDRATE
```

File “helloworld.c”

```
// Copyright (c) 2024 ETH Zurich and University of Bologna.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0/
//
// Authors:
// - Philippe Sauter <phsauter@iis.ee.ethz.ch>

#include "uart.h"
#include "print.h"
#include "timer.h"
#include "gpio.h"
#include "util.h"

#include "bbs32.h"

/// @brief Example integer square root
/// @return integer square root of n
uint32_t isqrt(uint32_t n) {
    uint32_t res = 0;
    uint32_t bit = (uint32_t)1 << 30;

    while (bit > n) bit >>= 2;

    while (bit) {
        if (n >= res + bit) {
            n -= res + bit;
            res = (res >> 1) + bit;
        } else {
            res >>= 1;
        }
        bit >>= 2;
    }
    return res;
}

int main() {
    uart_init(); // setup the uart peripheral

    // simple printf support (only prints text and hex numbers)
    printf("Hello World!\n");
    // wait until uart has finished sending
    uart_write_flush();

    // // toggling some GPIOs
    // gpio_set_direction(0xFFFF, 0x000F); // lowest four as outputs
    // gpio_write(0x0A);  // ready output pattern
    // gpio_enable(0xFF); // enable lowest eight
    // // wait a few cycles to give GPIO signal time to propagate
    // asm volatile ("nop; nop; nop; nop; nop;");
    // printf("GPIO (expect 0xA0): 0x%x\n", gpio_read());

    // gpio_toggle(0x0F); // toggle lower 8 GPIOs
    // asm volatile ("nop; nop; nop; nop; nop;");
    // printf("GPIO (expect 0x50): 0x%x\n", gpio_read());
    // uart_write_flush();

    // // doing some compute
    // uint32_t start = get_mcycle();
    // uint32_t res   = isqrt(1234567890UL);
    // uint32_t end   = get_mcycle();
    // printf("Result: 0x%x, Cycles: 0x%x\n", res, end - start);
    // uart_write_flush();

    // // using the timer
    // printf("Tick\n");
    // sleep_ms(10);
    // printf("Tock\n");
    // uart_write_flush();

    // //Simple check check user periphery
```

```
    // bbs32_write_seed(0xdeadbeef);
    // uint32_t read_res = bbs32_read_seed();
    // if (read_res != 0xdeadbeef) {
    //     printf("ERROR: value 0x%x\n", read_res);
    // }
    // else {
    //     printf("Success\n");
    // }
    // uart_write_flush();

    //Write P, Q, Seed
    uint32_t res = 0u;
    bbs32_write_pq(29711, 45543);
    bbs32_write_seed(56686);
    bbs32_first_run();
    bbs32_wait_ready();
    res = bbs32_read_result();
    printf("Result 0x%x", res);
    if (res != 0x6E341593) printf("--> Unexpected result\n");
    else                   printf("--> Expected result\n");
    uart_write_flush();

    bbs32_next_run();
    bbs32_wait_ready();
    res = bbs32_read_result();
    printf("Result 0x%x", res);
    if (res != 0xE62F5EBC) printf("--> Unexpected result\n");
    else                   printf("--> Expected result\n");
    uart_write_flush();

    return 1;
}
```

Appendix C. MATLAB script for NIST statistical tests

MATLAB script for NIST statistical tests (monobit and spectral).

```
close all;
clear;


T = readtable("bbs32_rtl_results.csv");
Bnum = height(T);

res1 = table2array(T);
res = transpose(res1);
res_ones = 2 .* res - 1;

%****************************************************************************
%Frequency (Monobit) Test (2.1)
%****************************************************************************
disp(" ")
disp("****************************************************************************");
disp("Frequency (Monobit) Test (2.1)");

sum_all = sum(res_ones, "all");
%Compute the test statistic
s_obs = abs(sum_all)/sqrt(Bnum);
%Compute P-value
p1_val = erfc(s_obs/sqrt(2));
fprintf("P-value is %f\n", p1_val);
test_p_value(p1_val);


%****************************************************************************
%Discrete Fourier Transform (Spectral) Test (2.6)
%****************************************************************************
disp(" ")
disp("****************************************************************************");
disp("Discrete Fourier Transform (Spectral) Test (2.6)");

res_fft = fft(res_ones);
res_fft_half = res_fft(1,1:Bnum/2);
f = (0:length(res_fft_half)-1)*100/length(res_fft_half);        % Frequency vector

mag = abs(res_fft_half);
phas = unwrap(angle(res_fft_half));

subplot(2,2,1);
plot(f,res_fft_half);
grid on
title('DFT raw');

subplot(2,2,2);
area(f,mag);
grid on
title('Magnitude');

subplot(2,2,3);
plot(f,phas);
grid on
title('Phase');

%Use only half of all results
%res_fft_half = res_fft(1,1:Bnum/2);
%Calculate M = modulus(S´) ≡ |S'|, where S´ is the substring consisting of the first n/2 elements
s_mod = abs(res_fft_half);
%Peak height threshold value
treshold = sqrt(log2(1/0.05)*Bnum);
fprintf("Threshold = %f\n", treshold);
%expected theoretical (95 %) number of peaks
n0 = 0.95*Bnum/2;
%the actual observed number of peaks
n1 = get_n1(s_mod, treshold);
```

```
fprintf("N0 = %u\n", n0);
fprintf("N1 = %u\n", n1);

%Compute d
d_val = (n1-n0)/sqrt(Bnum*0.95*0.05/4);
fprintf("d = %f\n", d_val);

%Compute P-value
p_val = erfc(abs(d_val)/sqrt(2));

fprintf("P-value is %f\n", p_val);
test_p_value(p_val);




%Functions definitions ************************************************

function res = test_p_value(p_in)
    if (p_in >= 0.01)
        disp("Sequence is random, P >= 0.01");
        res = 1;
    else
        disp("Sequence is NOT random, P < 0.01");
        res = 0;
    end
end

function n1_num = get_n1(fft_abs, treshold)
    n1_num = 0;
    for itr=1:length(fft_abs)
        if fft_abs(itr) < treshold
            n1_num = n1_num + 1;
        end
    end
end
```

Appendix D. Configuration file mmmc_config.tcl content

The content of the file “mmmc_config.tcl” is shown below.

```
# Generated using: Flowkit 23.14-s002_1

set GATELIB      "$env(GATELIB_IHP)"
set io_library   "$GATELIB/io/rev1.2.2/ixc013g2_iocell"
set std_library  "$GATELIB/stdcells/rev1.3.2/ixc013g2ng_stdcell"
set sram_library [file normalize "$GATELIB/../open_pdk_v0.2.0/libs.ref/sg13g2_sram/lib/"]

set out_dir [file normalize [file dirname [info script]]/../out]

#############################################################################
## LIBRARY SETS
#############################################################################
create_library_set \
    -name ls_slow_libs \
    -timing " \
        $std_library/lib/ixc013g2ng_stdcell_slow_1p08V_125C.lib \
        $io_library/lib/ixc013g2_iocell_slow_1p08V_3p0V_125C.lib \
        [glob -nocomplain -directory $sram_library/
RM_IHPSG13_1P_*_c?_bm_bist_slow_1p08*_125C.lib]"

create_library_set \
    -name ls_fast_libs \
    -timing " \
        $std_library/lib/ixc013g2ng_stdcell_fast_1p32V_m40C.lib \
        $io_library/lib/ixc013g2_iocell_fast_1p32V_3p6V_m40C.lib \
        [glob -nocomplain -directory $sram_library/
RM_IHPSG13_1P_*_c?_bm_bist_fast_1p32*_m55C.lib]"

create_library_set \
    -name ls_typ_libs \
    -timing " \
        $std_library/lib/ixc013g2ng_stdcell_typ_1p20V_25C.lib \
        $io_library/lib/ixc013g2_iocell_typ_1p2V_3p3V_25C.lib \
        [glob -nocomplain -directory $sram_library/ RM_IHPSG13_1P_*_c?_bm_bist_typ_1p20*_25C.lib]"

#############################################################################
## TIMING CONDITIONS
#############################################################################
create_timing_condition \
    -name tc_slow_libs \
    -library_sets { ls_slow_libs }

create_timing_condition \
    -name tc_fast_libs \
    -library_sets { ls_fast_libs }

create_timing_condition \
    -name tc_typ_libs \
    -library_sets { ls_typ_libs }

#############################################################################
## RC CORNERS
#############################################################################
create_rc_corner \
    -name rc_typ_corner \
    -temperature 25.0 \
    -cap_table $std_library/lef/captable/SG13.captable \
    -qrc_tech $GATELIB/../pdk/sg13g2_oa_ads2024_rev1.3.2/PVS_SG13/qrc/qrcTechFile \
    -pre_route_res 1.0 \
    -pre_route_cap 1.0 \
    -pre_route_clock_res 0.0 \
    -pre_route_clock_cap 0.0 \
    -post_route_res {1.0 1.0 1.0} \
    -post_route_cap {1.0 1.0 1.0} \
    -post_route_cross_cap {1.0 1.0 1.0} \
    -post_route_clock_res {1.0 1.0 1.0} \
```

```
    -post_route_clock_cap {1.0 1.0 1.0} \
    -post_route_clock_cross_cap {1.0 1.0 1.0}

##############################################################################
## DELAY CORNERS
##############################################################################
## Delay corner config
## ------------------------------------------------------------------------------------------------
## Index          0                 1              2                 3                   4
##        delay_corner_name library_set_name Library_name operational_conditions RC_corner_name
## ------------------------------------------------------------------------------------------------
set delay_corners_table [ list \
           { dc_slow   tc_slow_libs   ixc013g2ng_stdcell_slow_1p08V_125C   worst    rc_typ_corner
} \
           { dc_fast   tc_fast_libs   ixc013g2ng_stdcell_fast_1p32V_m40C   best     rc_typ_corner
} \
           { dc_typ    tc_typ_libs    ixc013g2ng_stdcell_typ_1p20V_25C     typical  rc_typ_corner
} \
]

foreach dc_corner $delay_corners_table {
    create_delay_corner \
        -name [ lindex $dc_corner 0 ] \
        -timing_condition [ lindex $dc_corner 1 ] \
        -rc_corner [ lindex $dc_corner 4 ]
}

##############################################################################
## CONSTRAINT MODES
##############################################################################
source $proj_dir/input/instances.tcl
create_constraint_mode  \
    -name cm_func \
    -sdc_files $proj_dir/input/constraints.sdc

##############################################################################
## ANALYSIS VIEWS
##############################################################################
## Analysis views config
## ------------------------------------------------------------------------
## Index          0                 1                      2
##        analysis_view_name delay_corner_name constraint_mode_name
## ------------------------------------------------------------------------
set analysis_views_table [ list \
        { av_func_slow dc_slow cm_func } \
        { av_func_fast dc_fast cm_func } \
]

foreach analysis_view $analysis_views_table {
    create_analysis_view \
        -name [ lindex $analysis_view 0 ] \
        -delay_corner [ lindex $analysis_view 1 ] \
        -constraint_mode [ lindex $analysis_view 2 ]
}

##############################################################################
## ACTIVE VIEWS
##############################################################################
set_analysis_view \
    -setup              av_func_fast \
    -hold               av_func_slow
```

Appendix E. SoC timing constraint file

File “input/constraints.sdc”

```
# Copyright 2024 ETH Zurich and University of Bologna.
# Solderpad Hardware License, Version 0.51, see LICENSE for details.
# SPDX-License-Identifier: SHL-0.51

# Authors:
# - Philippe Sauter <phsauter@iis.ee.ethz.ch>

# Backend constraints

#############
## Global ##
#############

# source instances.tcl

# Macro names as produced by the yosys synthesis
# Used for manual macro placement

set CROC            i_croc_soc/i_croc
set USER            i_croc_soc/i_user
set IBEX            $CROC/i_core_wrap.i_ibex
set SRAM            $CROC/gen_sram_bank
set JTAG            $CROC/i_dmi_jtag
set SRAM_512x32     gen_512x32xBx1.i_cut

# memory banks
set sram {\[0\].i_sram/}
set bank0_sram0 $SRAM$sram$SRAM_512x32
set sram {\[1\].i_sram/}
set bank1_sram0 $SRAM$sram$SRAM_512x32

set JTAG_ASYNC_REQ [get_nets $JTAG/i_dmi_cdc.i_cdc_req/*async_*]
set JTAG_ASYNC_RSP [get_nets $JTAG/i_dmi_cdc.i_cdc_resp/*async_*]


##############################
## Driving Cells and Loads ##
##############################

# As a default, drive multiple GPIO pads and be driven by one.
# accomodate for driving up to 2 74HC pads plus a 5pF trace
set_load [expr 2 * 5.0 + 5.0] [all_outputs]
# set_driving_cell [all_inputs] -lib_cell sg13g2_IOPadOut16mA -pin pad
set_driving_cell [all_inputs] -lib_cell ixc013_b16m -pin pad


##################
## Input Clocks ##
##################
puts "Clocks..."

# We target 80 MHz
set TCK_SYS 12.5
create_clock -name clk_sys -period $TCK_SYS [get_ports clk_i]

set TCK_JTG 20.0
create_clock -name clk_jtg -period $TCK_JTG [get_ports jtag_tck_i]

set TCK_RTC 50.0
create_clock -name clk_rtc -period $TCK_RTC [get_ports ref_clk_i]


#################################
## Clock Groups & Uncertainties ##
#################################
```

```
# Define which clocks are asynchronous to each other
# -allow_paths re-activates timing checks between asyncs -> we must constrain CDCs!
set_clock_groups -asynchronous -name clk_groups_async \
     -group {clk_rtc} \
     -group {clk_jtg} \
     -group {clk_sys}

# We set reasonable uncertainties in their transistion timing
# and transition (rise/fall) times for all clocks (ns)
set_clock_uncertainty 0.1 [all_clocks]
set_clock_transition  0.2 [all_clocks]


####################
## Cdcs and Syncs ##
####################
puts "CDC/Sync..."

# Clock Domain Crossings: paths going from a FF with one clock to an FF with another another)
# to increase the metastability-recovery window we do not wants any additional delays in these
paths
# so we deactivate the hold checking (as it may add buffers)

# Constrain `cdc_2phase` for DMI request
set_false_path -hold                  -through $JTAG_ASYNC_REQ
set_max_delay  [expr $TCK_SYS * 0.35] -through $JTAG_ASYNC_REQ -ignore_clock_latency

# Constrain `cdc_2phase` for DMI response
set_false_path -hold                  -through $JTAG_ASYNC_RSP
set_max_delay  [expr $TCK_SYS * 0.35] -through $JTAG_ASYNC_RSP -ignore_clock_latency


#############
## SoC Ins ##
#############
puts "Input/Outputs..."

# Reset should propagate to system domain within a clock cycle.
set_input_delay -max [ expr $TCK_JTG * 0.10 ] [get_ports rst_ni]
set_false_path -hold   -from [get_ports rst_ni]
set_max_delay $TCK_SYS -from [get_ports rst_ni]


##########
## JTAG ##
##########
puts "JTAG..."

set_input_delay  -min -add_delay -clock clk_jtg [ expr $TCK_JTG * 0.10 ]     [get_ports
{jtag_tdi_i jtag_tms_i}]
set_input_delay  -max -add_delay -clock clk_jtg [ expr $TCK_JTG * 0.50 ]     [get_ports
{jtag_tdi_i jtag_tms_i}]
set_output_delay -min -add_delay -clock clk_jtg [ expr $TCK_JTG * 0.10 / 2 ] [get_ports
jtag_tdo_o]
set_output_delay -max -add_delay -clock clk_jtg [ expr $TCK_JTG * 0.50 / 2 ] [get_ports
jtag_tdo_o]

# Reset should propagate to system domain within a clock cycle.
set_input_delay -max [ expr $TCK_JTG * 0.10 ] [get_ports jtag_trst_ni]
set_false_path -hold    -from [get_ports jtag_trst_ni]
set_max_delay $TCK_JTG  -from [get_ports jtag_trst_ni]


##########
## GPIO ##
##########
puts "GPIO..."

set_input_delay  -min -add_delay -clock clk_sys [ expr $TCK_SYS * 0.10 ] [get_ports {gpio*
fetch_en_i}]
set_input_delay  -max -add_delay -clock clk_sys [ expr $TCK_SYS * 0.30 ] [get_ports {gpio*
fetch_en_i}]
```

```
set_output_delay -min -add_delay -clock clk_sys [ expr $TCK_SYS * 0.10 ] [get_ports {status_o
gpio*}]
set_output_delay -max -add_delay -clock clk_sys [ expr $TCK_SYS * 0.30 ] [get_ports {status_o
gpio*}]


##########
## UART ##
##########
puts "UART..."

set_input_delay  -min -add_delay -clock clk_sys [ expr $TCK_SYS * 0.10 ] [get_ports uart_rx_i]
set_input_delay  -max -add_delay -clock clk_sys [ expr $TCK_SYS * 0.30 ] [get_ports uart_rx_i]
set_output_delay -min -add_delay -clock clk_sys [ expr $TCK_SYS * 0.10 ] [get_ports uart_tx_o]
set_output_delay -max -add_delay -clock clk_sys [ expr $TCK_SYS * 0.30 ] [get_ports uart_tx_o]
```

Appendix F. Floorplan script file

File “input/floorplan.tcl” content is shown below

```
#suspend; #type "resume" to resume
#Configure GUI
#gui_show
#set_layer_preference pinObj -is_visible 1

#Read flowtool top dir
set flowtool_top_dir [file normalize $::env(FLOWTOOL_TOP_DIR)]
set input_dir        [file normalize $flowtool_top_dir/../input]

## =============================================================
## Floorplan definition
## =============================================================

# From Croc:
# [QFN48]
#                   package die
#   pins               48   56
#   I/O                40   40
#   Core power          4    4
#   Core ground         -    4
#   Pad  power          4    4
#   Pad  ground         -    4
#
#
# final dimensions:                   [um]
#   die area (4sqmm)       2000.0 x 2000.0
#   io cell dimensions      310.0 x   60.0

set padD    310; # pad depth (edge to core)
set padW     60; # pad width (beachfront)

#From Croc:
# set coreH  1400;
# set coreW  1400;


## =============================================================
## Create floorplan: die, IO, Core
## =============================================================
#Set die WxH
set dieW   2022
set dieH   2022
#Set IO box WxH
set ioboxW 36
set ioboxH 36

#Calculate IO box right-top
set ioR     [expr $dieW - $padD]
set ioT     [expr $dieH - $padD]
#Calculate core dim.
set coreL   [expr $padD + $ioboxW]
set coreB   [expr $padD + $ioboxH]
set coreR   [expr $ioR - $ioboxH]
set coreT   [expr $ioT - $ioboxW]


#From: "Innovus Stylus Common UI Text Command Reference" p. 1004 (TCRcom.pdf)
#define box_size:
#   die box:  left bottom right top
#   IO box:   left bottom right top
#   core box: left bottom right top
create_floorplan \
    -site CoreSite \
    -no_snap_to_grid \
    -box_size \
```

```
        0      0      $dieW  $dieH \
        $padD  $padD  $ioR   $ioT \
        $coreL $coreB $coreR $coreT


#Total core area:
set coreW       [expr $coreR - $coreL]
set coreH       [expr $coreT - $coreB]
set coreArea    [expr $coreW * $coreH]
set coreAreamm2 [expr {($coreW * $coreH) * 1e-6}]
puts "\[SCRIPT\] Core size is $coreW x $coreH um, core area [format "%.2f mm^2" $coreAreamm2]"

# suspend; #type "resume" to resume

#Read IO configuration file
#See "Innovus User Guide" p. 244 (innovusUF.pdf), "Creating an I/O Assignment File"
#Also add corner cells, see 'topright','topleft' etc
read_io_file $input_dir/io_padring.tcl

# Add fillers between IO pads
add_io_fillers -cells {filler10u filler1u filler2u filler4u}

# suspend; #type "resume" to resume

## =========================================================
## Create power nets
## =========================================================
create_net -name VSS -ground
create_net -name CORE_VDD -power
create_net -name IO_VDD -power

# suspend; #type "resume" to resume

#Connect IO pads with these global nets
connect_global_net VSS \
    -type pg_pin \
    -override \
    -pin_base_name VSSPAD \
    -inst_base_name *

connect_global_net VSS \
    -type pg_pin \
    -override \
    -pin_base_name VSSCORE \
    -inst_base_name *

connect_global_net IO_VDD \
    -type pg_pin \
    -override \
    -pin_base_name VDDPAD \
    -inst_base_name *

connect_global_net CORE_VDD \
    -type pg_pin \
    -override \
    -pin_base_name VDDCORE \
    -inst_base_name *

# suspend; #type "resume" to resume

set createPGpinsManualy     0

if {${createPGpinsManualy} > 0} {
    #Create PG pins for VDD/CORE/VSS
    set pgW 54.20
    set pgH 71.20
    set pgWgap 2.9
    set pgHgap 2.535

    set horPGpins {}
    #               [pinXstart NetName PinName HorizotalStart]
    lappend horPGpins [list 336  VSS      VSSIO2 $pgHgap]
    lappend horPGpins [list 422  IO_VDD   VDDIO2 $pgHgap]
```

```
    lappend horPGpins [list 1540 VSS      VSS2   $pgHgap]
    lappend horPGpins [list 1626 CORE_VDD VDD2   $pgHgap]
    lappend horPGpins [list 336  VSS      VSSIO0 [expr $dieH-$pgH-$pgHgap]]
    lappend horPGpins [list 422  IO_VDD   VDDIO0 [expr $dieH-$pgH-$pgHgap]]
    lappend horPGpins [list 1540 VSS      VSS0   [expr $dieH-$pgH-$pgHgap]]
    lappend horPGpins [list 1626 CORE_VDD VDD0   [expr $dieH-$pgH-$pgHgap]]

    # TCRcom.pdf, p. 1023
    # -geometry layername x1 y1 x2 y2
    # Specifies the geometry of the physical pin.
    # x1 : Specifies the lower-left x coordinate, in microns, of the power/ground pin.
    # y1 : Specifies the lower-left y coordinate, in microns, of the power/ground pin.
    # x2 : Specifies the upper-right x coordinate, in microns, of the power/ground pin.
    # y2 : Specifies the upper-right y coordinate, in microns, of the power/ground pin.

    foreach pgPin $horPGpins {
        lassign $pgPin pinX pinNet pinName pinHgap
        create_pg_pin \
            -net $pinNet \
            -name $pinName \
            -geometry TopMetal2 \
                [expr $pinX+$pgWgap] \
                [expr $pinHgap] \
                [expr $pinX+$pgWgap+$pgW] \
                [expr $pinHgap+$pgH]
    }

    # suspend; #type "resume" to resume

    set verPGpins {}
    lappend verPGpins [list 336  CORE_VDD VDD1   $pgHgap]
    lappend verPGpins [list 422  VSS      VSS1   $pgHgap]
    lappend verPGpins [list 1540 IO_VDD   VDDIO1 $pgHgap]
    lappend verPGpins [list 1626 VSS      VSSIO1 $pgHgap]
    lappend verPGpins [list 336  CORE_VDD VDD3   [expr $dieW-$pgH-$pgHgap]]
    lappend verPGpins [list 422  VSS      VSS3   [expr $dieW-$pgH-$pgHgap]]
    lappend verPGpins [list 1540 IO_VDD   VDDIO3 [expr $dieW-$pgH-$pgHgap]]
    lappend verPGpins [list 1626 VSS      VSSIO3 [expr $dieW-$pgH-$pgHgap]]

    foreach pgPin $verPGpins {
        lassign $pgPin pinY pinNet pinName pinWgap
        create_pg_pin \
            -net $pinNet \
            -name $pinName \
            -geometry TopMetal2 \
                [expr $pinWgap] \
                [expr $pinY+$pgWgap] \
                [expr $pinWgap+$pgH] \
                [expr $pinY+$pgWgap+$pgW]
    }
}


## =======================================================
## Add core rings
## =======================================================
set_db add_rings_target default ;
set_db add_rings_extend_over_row 0 ;
set_db add_rings_ignore_rows 0 ;
set_db add_rings_avoid_short 0 ;
set_db add_rings_skip_shared_inner_ring none ;
set_db add_rings_stacked_via_top_layer TopMetal2 ;
set_db add_rings_stacked_via_bottom_layer Metal1 ;
set_db add_rings_via_using_exact_crossover_size 1 ;
set_db add_rings_orthogonal_only true ;
set_db add_rings_skip_via_on_pin {  standardcell } ;
set_db add_rings_skip_via_on_wire_shape {  noshape }

# suspend; #type "resume" to resume

# Add Core ring on TopMetal1 for VSS adn CORE_VDD
set ringW       10
set ringSpacing 6
```

```
#
add_rings \
    -nets {VSS CORE_VDD} \
    -type core_rings \
    -follow core \
    -layer {top TopMetal1 bottom TopMetal1 left TopMetal1 right TopMetal1} \
    -width "top ${ringW} bottom ${ringW} left ${ringW} right ${ringW}" \
    -spacing "top ${ringSpacing} bottom ${ringSpacing} left ${ringSpacing} right ${ringSpacing}" \
    -center 1 \
    -threshold 0 \
    -jog_distance 0 \
    -snap_wire_center_to_grid none


## ============================================================
## Add core stripes on TopMetal 1
## ============================================================

set coreOffset      63
set ioPadSpace      86
#Configure horisontal stripes position
set hstripeW        10
set horzVSSstripe   {0 3 8 11 14}
set horzVCOREstripe {1 4 6 10 13}
set horzStripeMetal {TopMetal2}

# suspend; #type "resume" to resume

# TCRcom.pdf, p.2918
# Add horizontal stripes for VSS
foreach iter ${horzVSSstripe} {
    add_stripes \
        -direction horizontal \
        -layer ${horzStripeMetal} \
        -nets "VSS" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${hstripeW}/2] \
        -number_of_sets 1 \
        -width ${hstripeW}
}

# Add horizontal stripes for core VDD
foreach iter ${horzVCOREstripe} {
    add_stripes \
        -direction horizontal \
        -layer ${horzStripeMetal} \
        -nets "CORE_VDD" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${hstripeW}/2] \
        -number_of_sets 1 \
        -width ${hstripeW}
}


## ============================================================
## Connect power pins to the core ring
## ============================================================

# suspend; #type "resume" to resume

#Connect IO cells pad pins to the power ring
#Connect all ports of pad pins, use only top metal 1 & 2
route_special \
    -nets {CORE_VDD VSS} \
    -connect {pad_pin} \
    -allow_jogging {0} \
    -allow_layer_change {1} \
    -pad_pin_layer_range {TopMetal2 TopMetal1} \
    -pad_pin_port_connect {all_port all_geom}


## ============================================================
## Place SRAM IPs
## ============================================================
```

```
# Vertical distance between BRAMs
set bramH       118
set bramW       785
set bramToBram  15
set bramToCore  20
set bramCell    {RM_IHPSG13_1P_256x64_c2_bm_bist}

# Add SRAM instnces into list
set bramList {}
lappend bramList {i_croc_soc/i_croc_gen_sram_bank\[0\].i_sram_gen_512x32xBx1.i_cut}
lappend bramList {i_croc_soc/i_croc_gen_sram_bank\[1\].i_sram_gen_512x32xBx1.i_cut}

#Configure coordinates for BRAMs AFTER rotation
set bram0X      [expr ${coreL}+${bramToCore}]
set bram0Y      [expr ${coreB}+${coreH}/2-${bramW}/2]
set bram1X      [expr ${bram0X}+${bramH}+${bramToBram}]
set bram1Y      ${bram0Y}

# suspend; #type "resume" to resume

#Enable floorplan view
# gui_set_draw_view fplan

# SRAM #0
#Step 1: put somewhere on core
update_floorplan_obj -obj [lindex ${bramList} 0] -move {850 400}
#Step 2: rotate
flip_or_rotate_obj -rotate {R90} -objs [lindex ${bramList} 0]
#Step 3: put on place
update_floorplan_obj -obj [lindex ${bramList} 0] -move "${bram0X} ${bram0Y}"

# SRAM #1
#Step 1: put somewhere on core
update_floorplan_obj -obj [lindex ${bramList} 1] -move {850 400}
#Step 2: rotate
flip_or_rotate_obj -rotate {R90} -objs [lindex ${bramList} 1]
#Step 3: put on place
update_floorplan_obj -obj [lindex ${bramList} 1] -move "${bram1X} ${bram1Y}"

# suspend; #type "resume" to resume

# Configure halo around BRAMs
set_macro_place_constraint \
    ${bramList} \
    -halo_sharing true

# Add halo
# TCRcom.pdf, p. 1033
#-halo_deltas left bottom right top
set haloV    7.3
set haloH    5.6
create_place_halo -halo_deltas ${haloH} ${haloV} ${haloH} ${haloV} -cell ${bramCell} -orient R90

# suspend; #type "resume" to resume

# Fix SRAM placement
foreach bramPath $bramList {
    set_db [get_db insts ${bramPath}] .place_status fixed
}


## =========================================================
## Add ring around BRAMs
## =========================================================

set bramRingMetalV  {Metal4}
set bramRingMetalH  {Metal3}
set bramRingW       2
set bramRingS       0.8
set bramRingOffsetV 2.4
set bramRingOffsetH 0.8

# suspend; #type "resume" to resume
```

```
# Select each SRAM and run 'add_rings' on each
foreach bramPath ${bramList} {
    # p. 1416
    deselect_obj -all
    select_obj [get_db insts ${bramPath}]

    # p. 2901
    # -user_defined_region {x1 y1 x2 ya ...}
    add_rings \
        -nets {VSS CORE_VDD} \
        -type block_rings \
        -around selected \
        -layer "top ${bramRingMetalH} bottom ${bramRingMetalH} left ${bramRingMetalV} right
${bramRingMetalV}" \
        -width ${bramRingW} \
        -spacing ${bramRingS} \
        -offset "top ${bramRingOffsetV} bottom ${bramRingOffsetV} left ${bramRingOffsetH} right
${bramRingOffsetH}" \
        -threshold 0 \
        -jog_distance 0 \
        -snap_wire_center_to_grid none
}
deselect_obj -all


## =========================================================
## Add stripes for BRAMs
## =========================================================

set bramVStripeMetal    {TopMetal1}
set bramVStripeW        6
set bramVStripeS        6
set bramVStripeY1       [expr ${bram0Y}*1.0-7.2]
set bramVStripeY2       [expr ${bram0Y}*1.0+${bramW}*1.0+6.675]
set bramVStripe0X2      [expr ${bram0X}+${bramH}]
set bramVStripe1X2      [expr ${bram1X}+${bramH}]

# suspend; #type "resume" to resume

# Add vertical stripes for BRAM0 manualy
# -area {x1 y1 x2 y2 ...}
add_stripes \
    -direction vertical \
    -layer ${bramVStripeMetal} \
    -nets "VSS CORE_VDD" \
    -width ${bramVStripeW} \
    -spacing ${bramVStripeS} \
    -number_of_sets 2 \
    -start_offset 16 \
    -stop_offset 16 \
    -area "${bram0X} ${bramVStripeY1} ${bramVStripe0X2} ${bramVStripeY2}"

# Add vertical stripes for BRAM1 manualy
# -area {x1 y1 x2 y2 ...}
add_stripes \
    -direction vertical \
    -layer ${bramVStripeMetal} \
    -nets "VSS CORE_VDD" \
    -width ${bramVStripeW} \
    -spacing ${bramVStripeS} \
    -number_of_sets 2 \
    -start_offset 16 \
    -stop_offset 16 \
    -area "${bram1X} ${bramVStripeY1} ${bramVStripe1X2} ${bramVStripeY2}"


## =========================================================
## Add core stripes on Metal 4
## =========================================================

#Configure vertical stripes position
set vstripeW        6
```

```
set vertVSSstripe   {0   1.57 3 8 11 14}
set vertVCOREstripe {0.5 2    4 6 10 13}
set vertStripeMetal {Metal4}

# suspend; #type "resume" to resume

# Add vertical stripes for VSS
foreach iter ${vertVSSstripe} {
    add_stripes \
        -direction vertical \
        -layer ${vertStripeMetal} \
        -nets "VSS" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${vstripeW}/2] \
        -number_of_sets 1 \
        -width ${vstripeW}
}

# Add vertical stripes for core VDD
foreach iter ${vertVCOREstripe} {
    add_stripes \
        -direction vertical \
        -layer ${vertStripeMetal} \
        -nets "CORE_VDD" \
        -start_offset [expr ${coreOffset}+${ioPadSpace}*${iter}-${vstripeW}/2] \
        -number_of_sets 1 \
        -width ${vstripeW}
}


## =========================================================
## Add Core pins on M1
## =========================================================

# suspend; #type "resume" to resume

# p. 1317
route_special \
    -nets {CORE_VDD VSS} \
    -connect {core_pin} \
    -core_pin_layer {Metal1} \
    -allow_jogging {0} \
    -allow_layer_change {0}


## =========================================================
## Add welltaps
## =========================================================

# suspend; #type "resume" to resume

# p.2560 add_well_taps
# Get welltap interval from "SG13G2 Layout Rules Rev. 2.0", see "7.2 Latch-up Guidelines", p.67
# Rules LU.a, LU.b is equal to 20 um MAX
# Welltap edge-to-edge distance should not be greater than this value
# Cell FEED3 width is 1.26 um, interval center-to-center calculated as:
# interval = welltap_width + LU.a

set coreSite        0.42
set intervalRule    20
set welltapWidth    1.26; #for FEED3
set wtInterval      [expr ((int($intervalRule/$coreSite)*$coreSite) + $welltapWidth) -
$coreSite*1]

#All well taps here: [get_db add_fillers_cells]

add_well_taps \
    -cell  FEED3JI \
    -prefix WELLTAP \
    -cell_interval ${wtInterval} \
    -start_row 1 \
    -checker_board

connect_global_net \
```

```
    CORE_VDD \
    -type pg_pin \
    -pin_base_name VDD \
    -inst_base_name WELLTAP_* \
    -override

connect_global_net \
    VSS \
    -type pg_pin \
    -pin_base_name VSS \
    -inst_base_name WELLTAP_* \
    -override

## ======================================================
## Connect all Cells to the power
## ======================================================

connect_global_net CORE_VDD -type pg_pin -pin_base_name VDD -all
connect_global_net VSS -type pg_pin -pin_base_name VSS -all
# For SRAM
connect_global_net CORE_VDD -type pg_pin -pin_base_name VDD! -all
connect_global_net CORE_VDD -type pg_pin -pin_base_name VDDARRAY -all
connect_global_net VSS -type pg_pin -pin_base_name VSS! -all
```

Appendix G. Pin placement configuration file

File “input/io_padring.tcl” content is shown below

```
(globals
    version = 3
    io_order = default
)
(iopad
        (topright
                (inst name="corner_cell_tr"
                 orientation=R90
                 cell=corner
                )
        )
        (top
                (inst  name="pad_vssio0" )
                (inst  name="pad_vddio0" )
                (inst  name="pad_gpio24_io" )
                (inst  name="pad_gpio25_io" )
                (inst  name="pad_gpio26_io" )
                (inst  name="pad_gpio27_io" )
                (inst  name="pad_gpio28_io" )
                (inst  name="pad_gpio29_io" )
                (inst  name="pad_gpio30_io" )
                (inst  name="pad_gpio31_io" )
                (inst  name="pad_unused0_o" )
                (inst  name="pad_unused1_o" )
                (inst  name="pad_unused2_o" )
                (inst  name="pad_unused3_o" )
                (inst  name="pad_vss0" )
                (inst  name="pad_vdd0"
                 orientation=MX
                 )
        )
        (topleft
                (inst  name="corner_cell_tl"
                 orientation=R180
                 cell=corner
                )
        )
        (left
                (inst  name="pad_vdd1" )
                (inst  name="pad_vss1" )
                (inst  name="pad_gpio12_io" )
                (inst  name="pad_gpio13_io" )
                (inst  name="pad_gpio14_io" )
                (inst  name="pad_gpio15_io" )
                (inst  name="pad_gpio16_io" )
                (inst  name="pad_gpio17_io" )
                (inst  name="pad_gpio18_io" )
                (inst  name="pad_gpio19_io" )
                (inst  name="pad_gpio20_io" )
                (inst  name="pad_gpio21_io" )
                (inst  name="pad_gpio22_io" )
                (inst  name="pad_gpio23_io" )
                (inst  name="pad_vssio1" )
                (inst  name="pad_vddio1" )
        )
        (bottomleft
                (inst  name="corner_cell_bl"
                 orientation=R270
                 cell=corner
                )
        )
        (bottom
                (inst  name="pad_vdd3" )
                (inst  name="pad_vss3" )
                (inst  name="pad_gpio11_io" )
                (inst  name="pad_gpio10_io" )
```

```
                (inst  name="pad_gpio9_io" )
                (inst  name="pad_gpio8_io" )
                (inst  name="pad_gpio7_io" )
                (inst  name="pad_gpio6_io" )
                (inst  name="pad_gpio5_io" )
                (inst  name="pad_gpio4_io" )
                (inst  name="pad_gpio3_io" )
                (inst  name="pad_gpio2_io" )
                (inst  name="pad_gpio1_io" )
                (inst  name="pad_gpio0_io" )
                (inst  name="pad_vssio3" )
                (inst  name="pad_vddio3" )
        )
        (bottomright
                (inst  name="corner_cell_br"
                 orientation=R0
                 cell=corner
                )
        )
        (right
                (inst  name="pad_vssio2" )
                (inst  name="pad_vddio2" )
                (inst  name="pad_uart_rx_i" )
                (inst  name="pad_uart_tx_o" )
                (inst  name="pad_fetch_en_i" )
                (inst  name="pad_status_o" )
                (inst  name="pad_clk_i" )
                (inst  name="pad_ref_clk_i" )
                (inst  name="pad_rst_ni" )
                (inst  name="pad_jtag_tck_i" )
                (inst  name="pad_jtag_trst_ni" )
                (inst  name="pad_jtag_tms_i" )
                (inst  name="pad_jtag_tdi_i" )
                (inst  name="pad_jtag_tdo_o" )
                (inst  name="pad_vss2" )
                (inst  name="pad_vdd2" )
        )
)
```