Experiment 7
Hardware-based Security Primitives and Their Applications

on HaHa v3.0 Board

We introduce two most important hardware-based security primitives: physical unclonable functions (PUFs) and true random number generator (TRNG). We also introduce the applications of these security primitives to defend counterfeiting attacks at the chip level.

Instructor: Dr. Swarup Bhunia

Co-Instructors/TAs: Reiner Dizon-Paradis, Chris Vega, and Shuo Yang

Theory Background

1. PUF

Physical unclonable function (PUF) is an emerging potential security primitive for generating volatile secret keys in cryptographic applications. There are different types of PUFs implemented in FPGA [1-8]. A PUF is described as unclonable due to its uniqueness being derived from the uncontrollable variations introduced during the manufacturing process. PUFs offer a high level of protection in cryptographic applications with strong volatile key storage. PUFs are issued a challenge and (ideally) produce a unique and reliable response in return. Since this response is unique to the device, it can, therefore, be used as a device ID or key. In this experiment, we will learn two most popular PUFs: SRAM-based PUF and ring-oscillator (RO)-based PUF.

1.1 SRAM-based PUF:

The usage of SRAM as the PUF medium is appealing for a variety of reasons. Most notably, SRAM is commonly available in most systems and therefore does not require additional hardware. For a standard 6T SRAM (Figure 1), every memory cell is composed of six transistors that are two cross-coupled CMOS inverters and two access transistors. The inverters are designed to be symmetric, match in size, etc., but random variations incurred during manufacturing will result in random mismatches. SRAM PUFs exploit the mismatch which results in each SRAM cell being biased (or skewed) toward a zero or one at power-up. Due to uncontrollable variations in the manufacturing process, different CMOS devices have different physical parameters (e.g., doping-levels, transistor oxide thickness, etc.). When an SRAM is powered-up, these variations affect the power-up state of their associated cells. It has been observed that certain cells have a strong preference to power-up to a ‘1’ or ‘0’ state. Cells that have no preference are deemed neutral and power up at random depending upon the influences of system noise [5]. The more useful cells for PUF output are the ones that strongly prefer ‘0’ or ‘1’.

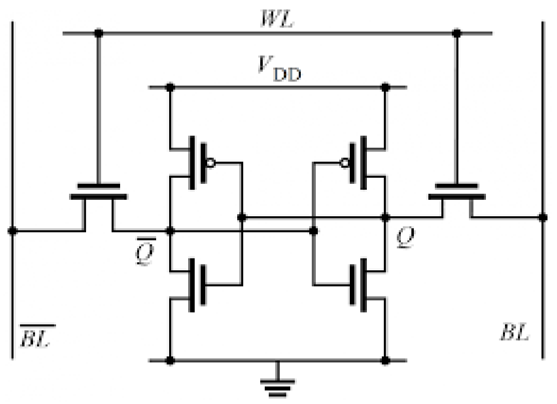


Figure 1 SRAM memory cell used as PUF

1.2 RO-based PUF:

Figure 2 shows a conventional RO-based PUF which generally consists of N identically laid out ROs, two counters, a comparator, and two N-bit multiplexers. Each RO consists of an odd number of inverters and oscillates with a particular frequency. Because of process variation and inherent random noise, each RO oscillates with a slightly different frequency than others even though they are designed to be identical. A response bit is generated by comparing the frequency of two identical ROs, where one RO is faster than the other and generates one (or zero) by a comparator. A pair of ROs is chosen, in order to generate a bit, by applying a digital sequence of length log2(N), known as a challenge. The applied challenge in both MUXs selects RO from each MUX and forms a pair. The counters connected to the MUXs count the number of oscillations of each of the paired ROs for a fixed time interval. This time interval is known as comparison time and impacts the reproducibility of a key. Usually, the comparison time needs to be larger when the difference between the frequencies of ROs in a pair is very low. The comparator generates a bit from the counters' outputs by comparing the corresponding values and set `0' or `1' based on which oscillator from the selected RO pair is faster, i.e., has a greater count for the given comparison time.

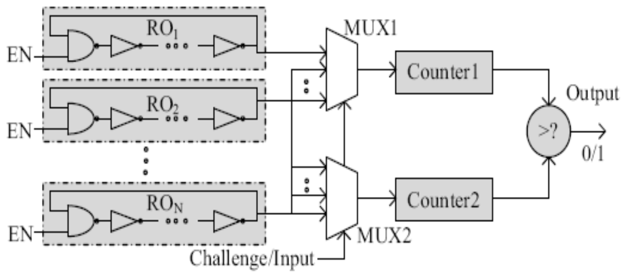


Figure 2 RO-PUF

1.3 PUFs Evaluation:

Reliability: The Reliability of a PUF determines how often a PUF can generate the same response to a given challenge. A PUF must generate the same response at all operating conditions.

Uniqueness: Uniqueness measures how well a single PUF is differentiated from other PUFs based on its challenge-response pair. Different PUFs must generate different responses for a given challenge in order to separate one from another. The average inter-chip fractional hamming distance for an ideal PUF must be 0.5. If a PUF circuit is instantiated on several different chips, then each of the PUF instantiations is expected to produce unique responses when supplied with the same challenge C. In general, C is an m-bit binary input which is supplied to the each of the PUF instantiations. The responses R1, R2, R3, and R4 are n-bit binary strings (Figure 3).

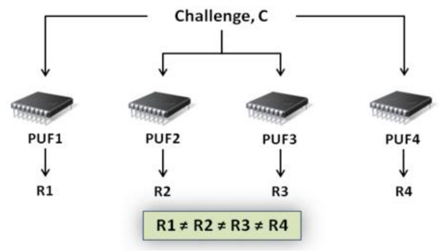


Figure 3 PUF Uniqueness

Randomness: Entropy can be used to measure the randomness of a PUF. It's the reflection of the amount of 1s and 0s in a sequence.

Diffuseness: The diffuseness, as discussed earlier, is very similar to the uniqueness. How responses of the same PUF are different for different challenges.

2. True Random Number Generator:

2.1 TRNG and an example: RO-based TRNG

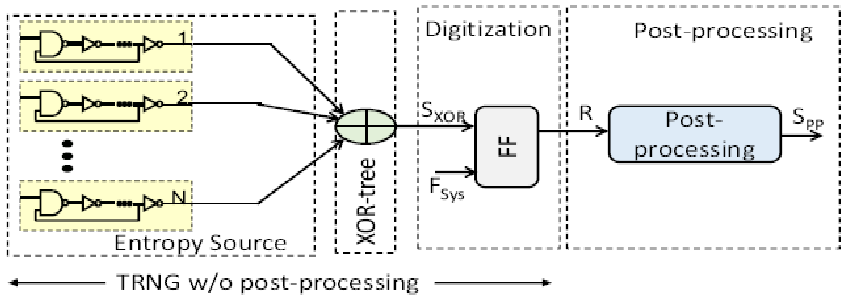


Figure 4 RO-based TRNG

A random number generator (RNG) is an important security block widely used in most cryptographic applications such as one-time pads, session, and temporary keys, nonce, seeds, challenges for authentication, zero-knowledge protocols, hardware metering, generation of primes, secure communications, secured servers and processors, VPN access, and customer-facing web access. A quality RNG generates statistically independent and unpredictable sequences of random numbers. Compromising an RNG often means compromising an entire system. A true RNG (TRNG) translates random physical phenomena such as thermal noise, atmospheric noise, shot noise, radio noise, flicker noise, clock jitter, phase noise, noise in a compact memory etc. into random digits. A TRNG must have uniform statistics; non-uniform statistics due to biased random sequences help attackers to guess the random numbers. Generally, random numbers are generated by comparing two symmetric devices which possess some process variation (PV) and random inner noise [9-11].

TRNG consists of a source of randomness (entropy) and a randomness extractor. Metastability of logic cells and timing jitter of Ring Oscillators (ROs) are the most common sources of entropy for a TRNG. The simplicity of implementation and entropy collection have made the RO-based TRNG most popular. Figure 4 shows a conventional RO-based TRNG where the RO outputs are combined by an XOR-tree. The output of the XOR-tree, S_{XOR}, is sampled constantly by a synchronous D flip-flop driven by the system clock to convert RO jitter into a random digital sequence. Jitter, in this case, represents the deviation from ideal RO behavior caused by random process variation and temporal variations such as random physical noise, environmental variations, and aging. Without jitter, the ROs will possess almost identical phase and the output of XOR-tree would be almost constant which is undesirable. Among the variations, only the random physical noise (thermal noise, atmospheric noise, shot noise, radio noise, flicker noise, etc.) improves the randomness/entropy of the bitstream output by the TRNG.

2.2 TRNG Evaluation:

Unlike the PUF output, the TRNG output has to be different from measurement to measurement so that attacker cannot guess future output from previous TRNG output. NIST test is used to measure the randomness of a TRNG output [4]. NIST's statistical test suite is popularly used to evaluate the quality of randomness for random and pseudorandom number generators designed for cryptographic applications and can also be used for PUF [9-11]. There is a total of 15 NIST tests and different tests require a different minimum length of bitstreams. For example, rank test, linear complexity test, and overlapping template matching test require at least 38912, 106, and 106 bits long bitstreams respectively. On the other hand, frequency test, block frequency test, and runs test require a minimum of 100-bit long bitstream

3. Application of PUFs in Supply Chain Integrity

PUFs can serve as a root of trust and can provide a key which cannot be easily reverse engineered. One of the biggest issues in any supply chain management is being able to verify that the product ordered is the product received. The electronics industry makes great strides to verify the integrity of their product lines. For example, the distributor your group used to complete the Bill of Materials for the previous experiment, Digi-Key, is a member of the trade association Electronic Component Industry Association, or ECIA, which strives to ensure that no counterfeit devices enter the supply chain. However, one can't always order from verified suppliers. In this case, it is necessary to order components from unauthorized resellers, increasing the risk of acquiring counterfeit goods. One emerging research area to counteract this problem is in the area of Physically Unclonable Functions. The gist of the idea of a PUF is that the manufacture of the goods will use some sort of intrinsic process variation to generate a unique signature for every device. This signature is entered into a database so that the end user can, using the same signature generation process, generate and verify the signature of their device.

Applications of PUF includes cryptographic key generation memoryless key storage, device authentication, PUF-based RFID for anti-counterfeiting, Intellectual Property (IP) protection etc.

💻Experiment Set-up: Configuration

The instruments needed for this experiment are the HAHA V3.0 Board, a USB A to B cable, and a computer. Figure 5 shows the instruments connections.

The software needed to program the microcontroller is Microchip Studio. Use GOWIN FPGA Designer to program the FPGA. MATLAB or Python is recommended to process the data.

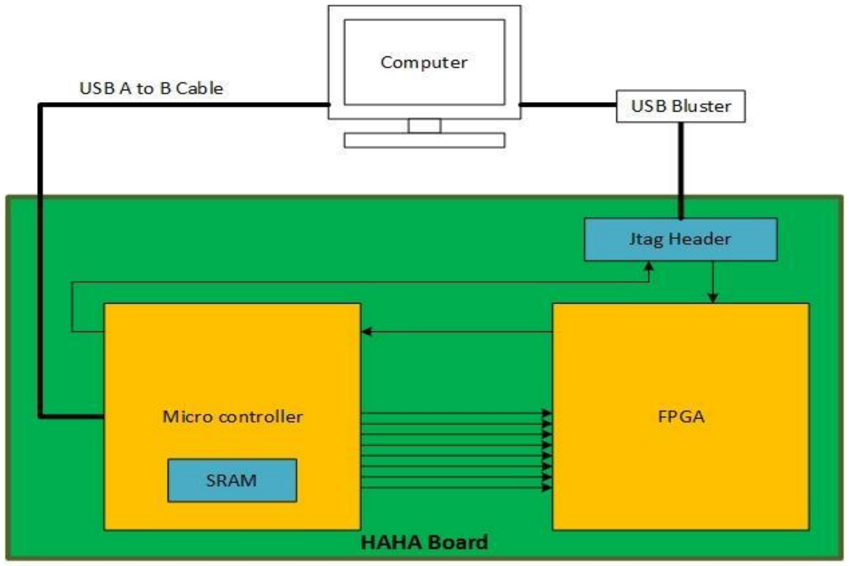


Figure 5 Experiment set-up.

Instructions and Questions

Part I SRAM PUF

In this part, you will implement an SRAM PUF on the HAHA V3.0 Board. Read 64 bytes of the SRAM power-up states from the Microcontroller to generate a signature. Only given a USB A to B cable, your PC cannot collect data directly from the Microcontroller. Therefore, you have to send data to the FPGA and then export.

Program the Microcontroller. It needs to do two things: read and send. Refer to the datasheet to find the address range for the SRAM inside the Microcontroller. In this range, read 64 bytes power-up states. Note that when the Microcontroller is running, some SRAM addresses will be occupied, and these addresses will no longer hold their power-up states. Therefore, chose the addresses for your 64 bytes carefully and make sure they won't be occupied by running your program. Use the 9 interconnections between the microcontroller and the FPGA to send data. Generate a clock signal on PORTC0 and send data on PORTA.

Program the FPGA. It needs to do two things: receive and store. Instantiate a RAM in it that is big enough to some of the 64-byte data without causing error. Add signals from the internals of this RAM into the RAO file. Use the clock signal provided by the Microcontroller. After storing, export data to your computer from the GOWIN Analyzer Oscilloscope.

Process your data with tools, such as MATLAB, Python, etc., according to the requirements given below and answer questions.

1) Refer to the datasheet of the Microcontroller, what is the size of the SRAM in it?
2) You should read the power-up states of the SRAM cells, i.e., the values are not changed by the program after power up. What addresses will be occupied by running your program? What addresses do you choose to get your 64-byte signature? (Hint: for affected addresses, their contents will look "much less random" than the power-up values.)
3) Use the 9 interconnections between the Microcontroller and the FPGA to communicate. How can you generate a square wave from the Microcontroller I/O pin? What's the frequency?
4) Turn in your code for the Microcontroller. Give comments to make it easier to read.
5) Turn in your code for the FPGA. Turn in a screenshot of the GOWIN Analyzer Oscilloscope window after storing all data.
6) Export data to a CSV file. Convert only the relevant data (remove all other information) in the CSV file from hexadecimal to binary, which will be signature S1. How many bits does S1 have? Include your S1 in the report in both hexadecimal and binary format.
7) For S1, how many percentages of the bits are 0 and how many percentages are 1? What is the mean value ($\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$)? [6]

8) Power OFF the whole board when taking different readings (remove all USB connections). Change the power supply for the microcontroller to be adjustable voltage source: 3.0V and 2.7V. Then power ON the board and read out the signature again. These will be your signature S2 and S3. Include S2 and S3 in the report in both hexadecimal and binary format.
9) Are S2 and S3 the same with S1? How many bits are different? How much is the fractional ***intra*** hamming distance? [6]
10) Turn in your code for calculating the mean value and the ***intra*** hamming distance for this part.

Part II RO-PUF

In this part, you should design a top module with the provided code to generate a 128-bit key.

After compiling your code, you should compare symmetrical ROs to obtain the key. This must also include placing the MUXes and counters symmetrically. Use the FloorPlanner tool to find the names/locations of the PUF elements after Place/Route. Right click on the Chip view and select “Show Multi-View” where you will see used resources on the FPGA. Expand on the top module and then expand “Primitives” where you will see all the PUF elements. Right click on any of these items and select “Edit Constraint” where you will see the locations in the bottom pane. Then, create a script to generate location assignments for all the PUF elements. See the Appendix and [12-14] for more information.

1) Use the provided Verilog/VHDL codes to place the RO and PUF elements in such a way that they are physically symmetric. Turn in your code for the top module and any additional circuits that were created by your group.
2) Run the code to generate a 128-key. It is your K1. Turn in your K1.
3) Change voltages to 3.0V and 2.7V separately to collect the Key for each case. They are your K2 and K3. Include them in the report.
4) Are K2 and K3 the same with K1? How many bits are different? How much is the fractional ***intra*** hamming distance? [6] Analyze the reliability.

Part III SRAM as a TRNG

In this part, you will modify your design for Part I so that it can work as a TRNG instead of a PUF. You can repeat your steps in Part I to collect multiple S1’s (under the same voltage and temperature condition). See if all the bits of S1 will stay the same whenever you collect them and find if there are bits that act more randomly than others. Use these bits to generate an 8-bit TRNG.

1) What type of SRAM cells is a good candidate for TRNG? Select 8 bits from them. Include their addresses in your report and explain the procedure that you find them.
2) Write code to generate an 8-bit random number with your SRAM in the Microcontroller. Turn in your code.

Part IV RO-based TRNG

In this part, you will write a VHDL/Verilog code for RO-based TRNG with ten 5-stage ROs (you can use Verilog code from RO-PUF). Use the same method as in Part II to make the ROs symmetric. Then collect 10^5 bits as your random number under different conditions as required below.

1) Write a VHDL/Verilog code for RO-based TRNG with ten 5-stage ROs. Constrained routing: make the ROs symmetric. Collect 10^5 bits for each of the following cases.

10^5-bit number	Voltage	Number of ROs	Number of stages
N1	3.3V	10	9
N2	3.3V	20	5
N3	3.3V	30	5
N4	3.0V	10	5
N5	2.7V	10	5

2) Change the stage of ROs to 9. (N1)
3) Change the number of ROs to 20 and 30. (N2 and N3)
4) Change the operating voltage to 3.0V and 2.7V. (N4 an N5)
5) Analyze the robustness of RO-based TRNG you designed.
6) Optional: perform NIST test. (See the optional Part VI)

Optional Follow-up

Part V Uniqueness evaluation

Exchange your HAHA V3.0 board with another group temporarily to finish this part. Get back your own HAHA V3.0 Board afterward.

1) Run your group's ***own SRAM PUF code*** on other group's HAHA V3.0 Board. Get signature S4. Compare signature S4 with the signature S1. Are they the same? How many bits are different? How much is the fractional ***inter*** hamming distance? [6]
2) Run your group's ***own RO PUF code*** on other group's HAHA V3.0 Board. Get signature and calculate the ***inter*** hamming distance with the signatures you got in Part II.

As UF EDGE students do not have access to a second board, you can finish the extra requirement in Part VI to get full bonus points.

Complete this part for up to 10% extra points.

Part VI Randomness evaluation

Evaluate the randomness of your random numbers N1-N5 using the test suite provided in [4]. The more tests you do, the better. Three different tests make the full credit for this part. Include your results in your report.

For UF EDGE students, you can run 5 more tests in order to earn full optional points.

Complete this part for up to 10% extra points.

☞ Lab Report Guidelines

Deliverable:

1. In your report, answer ALL the questions.
2. Give all the code for all the steps as required.
3. Give photos/screenshots as required.

Demonstration:

Submit videos that fulfill the requirements below.

1. For Part I, show how you get the data and store it in the FPGA.
2. For Part II, show how you get the signature.
3. For Part III, show that you can get different random numbers when you run it multiple times.
4. For Part IV, show how you get the random numbers under 1 of the 5 listed conditions.
5. No demo for Part V.

Your videos should follow the format below.

1. start with a commentary (3-4 sentences) on what this experiment is about and what are the goals.
2. followed by a commentary (3-4 sentences) on the steps you follow to do the experiments.
3. Next, you move the camera to show the experiment setup (the HaHa v3.0 board, connection with Analog Discovery 2 and computer), go through the steps one by one on the setup - how you apply the inputs, what components of the HaHa v3.0 board are involved, what you expect to see - and then show the outputs - e.g., recorded signal (if any), output LED or seven segment displays, etc.
4. Next, add a commentary on the key lessons learnt through this experiment. If you've done any advanced options or explored anything beyond that is covered by the lab instructions, describe them too.
5. The video should have no mistakes.

References and Further Reading

[1] http://rijndael.ece.vt.edu/puf/background.html

[2] https://en.wikipedia.org/wiki/Hamming_distance

[3] Gassend, Blaise, et al. "Silicon physical random functions." Proceedings of the 9th ACM conference on Computer and communications security. ACM, 2002.

[4] Rukhin, Andrew, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Booz-Allen and Hamilton Inc Mclean Va, 2001.

[5] Holcomb, Daniel E., Wayne P. Burleson, and Kevin Fu. "Power-up SRAM state as an identifying fingerprint and source of true random numbers." *IEEE Transactions on Computers* 58.9 (2009): 1198-1210.

[6] Böhm, Christoph, Maximilian Hofer, and Wolfgang Pribyl. "A microcontroller sram-puf." *Network and System Security (NSS), 2011 5th International Conference on*. IEEE, 2011.

[7] Suh, G. Edward, and Srinivas Devadas. "Physical unclonable functions for device authentication and secret key generation." *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007.

[8] A. Maiti et al., “A systematic method to evaluate and compare the performance of physical unclonable functions,” In Embedded Systems Design with FPGAs, P. Athanas, D. Pnevmatikatos, and N. Sklavos (Eds.). Springer, New York, 245267.

[9] Md. Tauhidur Rahman et al., "TI-TRNG: Technology Independent True Random Number Generator. Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference (DAC), Article 179, pp. 1-6, 2014.

[10] Mehrdad Majzoobi, Farinaz Koushanfar, Srinivas Devadas, FPGA-Based true random number generation using circuit metastability with adaptive feedback control, Proceedings of the 13th international conference on Cryptographic hardware and embedded systems, September 28-October 01, 2011, Nara, Japan.

[11] Berk Sunar, William J. Martin, Douglas R. Stinson, A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks, IEEE Transactions on Computers, v.56 n.1, p.109-119, January 2007.

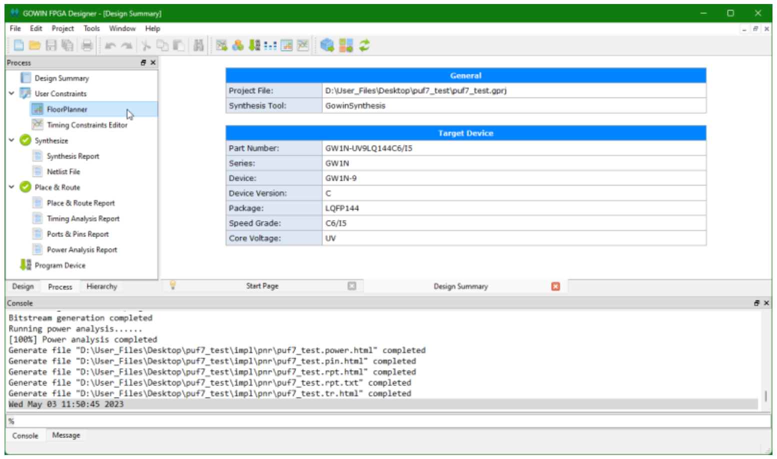
[12] GOWIN User Guide, http://cdn.gowinsemi.com.cn/SUG550E.pdf

[13] GOWIN Design Physical Constraints, http://cdn.gowinsemi.com.cn/SUG935E.pdf

[14] GOWIN FPGA Primitive, https://www.gowinsemi.com/upload/database_doc/39/document/5bfcff2ce0b72.pdf

Appendix: Accessing/changing locations of PUF elements

1. Before accessing GOWIN's FloorPlanner tool to obtain original placements of the PUF elements, please make sure to perform synthesis and place/route operations first.
2. Double click on **FloorPlanner**. A new window will appear.

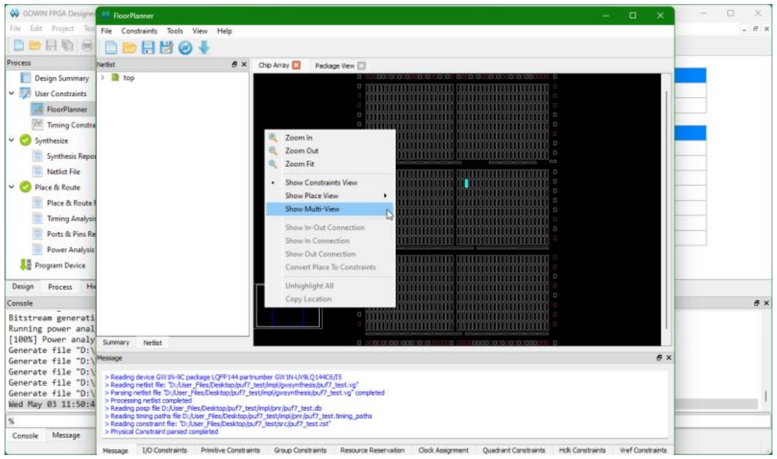


General	
Project File:	D:\User_Files\Desktop\puf7_test\puf7_test.gprj
Synthesis Tool:	GowinSynthesis

Target Device	
Part Number:	GW1N-UV9LQ144C6/I5
Series:	GW1N
Device:	GW1N-9
Device Version:	C
Package:	LQFP144
Speed Grade:	C6/I5
Core Voltage:	UV

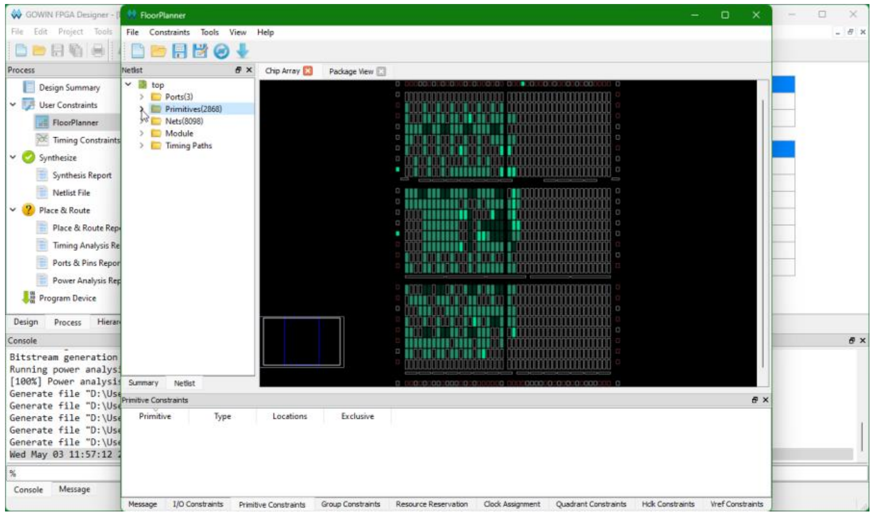
```
Bitstream generation completed
Running power analysis......
[100%] Power analysis completed
Generate file "D:\User_Files\Desktop\puf7_test\impl\pnr\puf7_test.power.html" completed
Generate file "D:\User_Files\Desktop\puf7_test\impl\pnr\puf7_test.pin.html" completed
Generate file "D:\User_Files\Desktop\puf7_test\impl\pnr\puf7_test.rpt.html" completed
Generate file "D:\User_Files\Desktop\puf7_test\impl\pnr\puf7_test.rpt.txt" completed
Generate file "D:\User_Files\Desktop\puf7_test\impl\pnr\puf7_test.tr.html" completed
Wed May 03 11:50:45 2023
%
```

3. Right click anywhere on the *Chip Array*. Click on **Show Multi-View**.

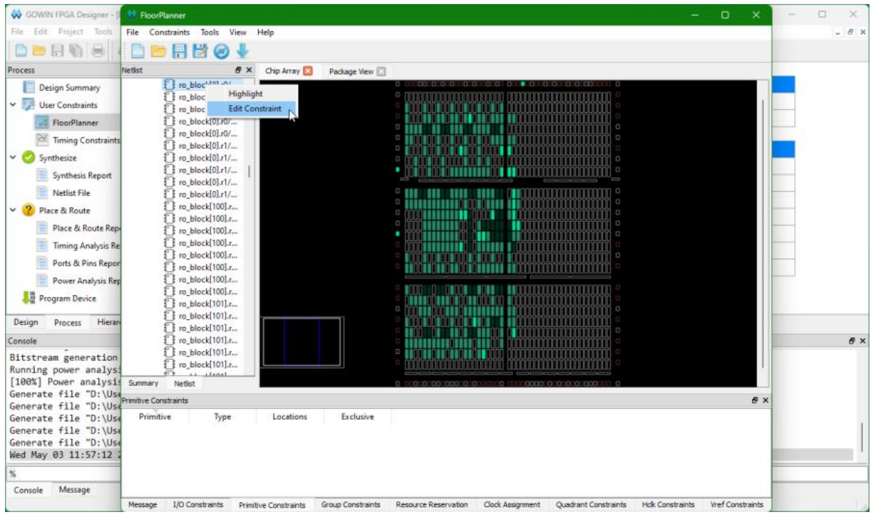


```
> Reading device GW1N-9C package LQFP144 partnumber GW1N-UV9LQ144C6/I5
> Reading netlist file: "D:/User_Files/Desktop/puf7_test/impl/gwsynthesis/puf7_test.vg"
> Parsing netlist file "D:/User_Files/Desktop/puf7_test/impl/gwsynthesis/puf7_test.vg" completed
> Processing netlist completed
> Reading posp file D:/User_Files/Desktop/puf7_test/impl/pnr/puf7_test.db
> Reading timing paths file D:/User_Files/Desktop/puf7_test/impl/pnr/puf7_test.timing_paths
> Reading constraint file: "D:/User_Files/Desktop/puf7_test/src/puf7_test.cst"
> Physical Constraint parsed completed
```

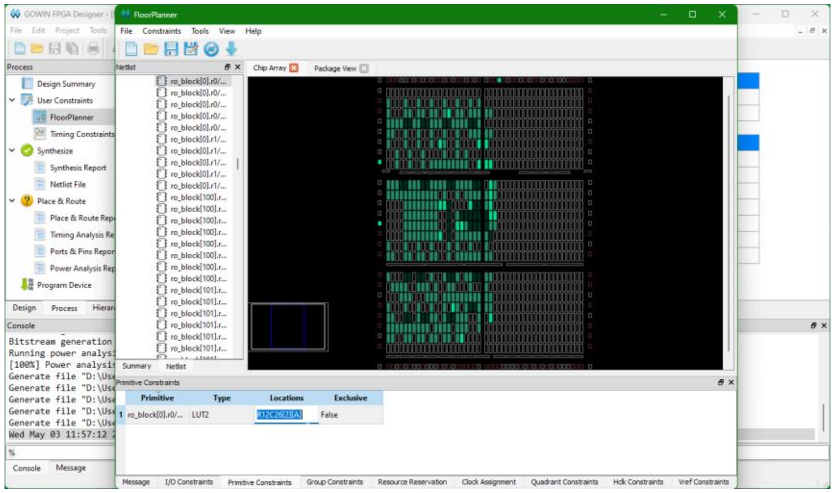
4. You will now see the utilized FPGA resources. On the left sidebar, expand the *top* module. Then, expand the **Primitive** folder.



5. Here you can find all the instantiated modules. Select a module and right click on it. If you want to change its location, select **Edit Constraint**.

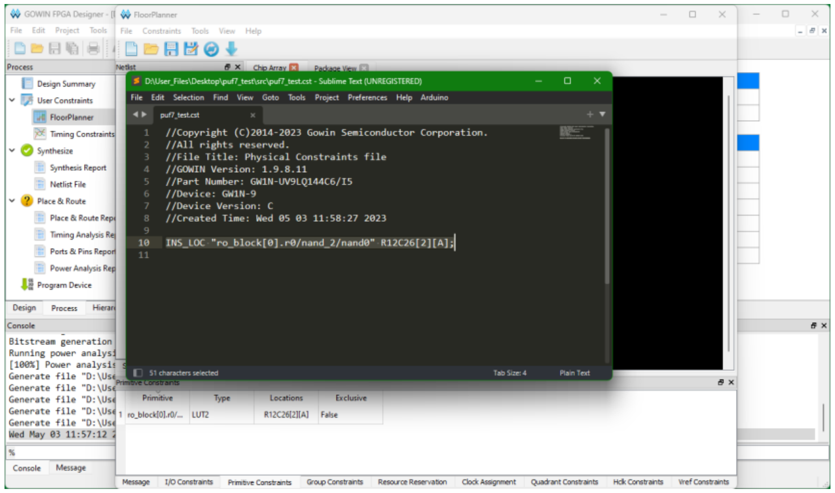


6. In the bottom pane, you will now see a new entry in the *Primitive Constraint* tab. Here you can change the location of the selected module. For now, save the CST file.


Primitive Constraints

	Primitive	Type	Locations	Exclusive
1	ro_block[0].r0/...	LUT2	R12C26[2][A]	False

7. Open the CST file in any code/text editor. You will now find a new entry on it that starts with **INS_LOC**.


D:\User_Files\Desktop\puf7_test\src\puf7_test.cst - Sublime Text (UNREGISTERED)

```
//Copyright (C)2014-2023 Gowin Semiconductor Corporation.
//All rights reserved.
//File Title: Physical Constraints file
//GOWIN Version: 1.9.8.11
//Part Number: GW1N-UV9LQ144C6/I5
//Device: GW1N-9
//Device Version: C
//Created Time: Wed 05 03 11:58:27 2023

INS_LOC "ro_block[0].r0/nand_2/nand0" R12C26[2][A];
```

8. Observe how the locations are named in the *Chip Array* view. Create a script that automates the symmetric placements of PUF elements and writes the new locations directly to the CST file. After making changes to CST file, open the **FloorPlanner** tool to observe changes.