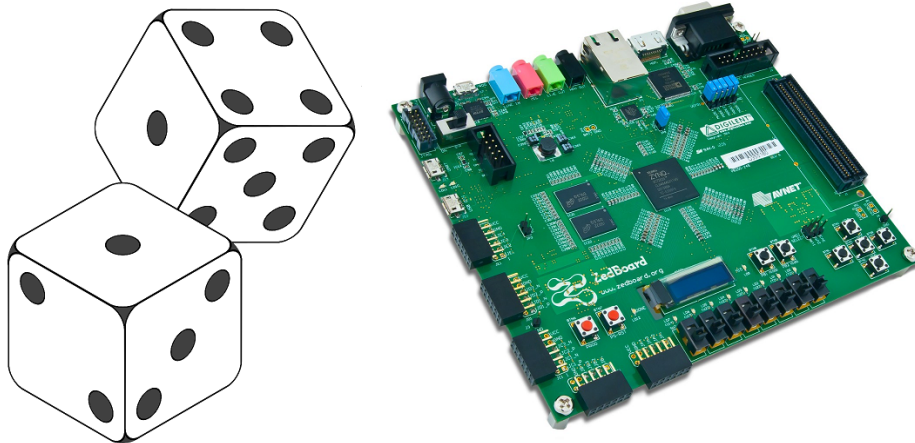


Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2016



Project Title: **Analysis of Random Numbers using FPGAs**

Student: **Sol-Uh Park**

CID: **00888669**

Course: **EIE3**

Project Supervisor: **Dr David Thomas**

Second Marker: **Prof G.A. Constantinides**

Abstract

With increasing amounts of computationally intensive data being analysed in areas such as finance, science and engineering, there has been growing research interest in faster Monte Carlo (MC) simulations. Faster MC simulations would in turn require faster input streams of random numbers, necessitating faster Random Number Generators (RNGs). This project attempts to address current problems in the empirical statistical testing of FPGA-accelerated RNGs; modern FPGA-accelerated RNGs generate numbers at speeds and volumes much higher than existing software-based RNG test suites are capable of processing. The implementation achieved in this project is a software-configurable FPGA/microprocessor embedded system solution for empirical statistical testing, which can analyse up to 2^{44} samples of 32-bit FPGA-accelerated uniform RNGs at throughput rates of 100 million samples per second. The implementation is intended for helping users to analyse custom FPGA-accelerated RNGs for the randomness properties of independence and uniform distribution for more samples and faster than existing test suites. The implementation target platform is the ZedBoard Zynq-7000 ARM/FPGA SoC, whilst the hardware and software portions of the embedded system design are implemented using the Vivado Design Suite and Xilinx Software Development Kit, respectively. Users will be able to test custom RNG algorithms implemented in Vivado HLS C++.

Acknowledgements

I would like to thank Dr David Thomas for all his help and support throughout the project. This project would not have been possible without his teaching, supervision and guidance. I would also like to thank him for the valuable project management skills which I have been fortunate to strengthen under his tutelage. Finally, I would like to thank him for making my cruise into these uncharted waters a challenging, yet enjoyable experience, thus making it a satisfying finisher for my studies at Imperial College London.

Contents

1	Introduction	1
1.1	Project Scope	2
1.2	Target Platform	3
1.3	Report Structure	4
2	Background	5
2.1	Project Motivation	5
2.1.1	Monte Carlo simulations and RNGs in FPGAs	5
2.1.2	Need for higher-throughput RNG implementations	5
2.1.3	Need for higher-throughput RNG Test Suites	6
2.2	RNG Types under Consideration	6
2.2.1	Pseudo and Deterministic RNGs	7
2.2.2	Uniform RNGs	8
2.2.3	32-Bit Unsigned Integer RNGs	8
2.3	Statistical Testing of RNGs	8
2.3.1	How to assess the Statistical Properties of RNGs?	8
2.3.2	Interpreting Empirical Test Results	9
2.4	Empirical Statistical Test Algorithm	10
2.4.1	Chi-Squared Test	11
2.4.2	Computing p -values	11
2.5	Empirical Tests under Consideration	12
2.5.1	Selection Criteria	12
2.5.2	Frequency Test	13
2.5.3	Serial 2-Tuple Correlation Test	14
2.5.4	Serial 3-Tuple Correlation Test	15
2.6	RNG Algorithms under Consideration	17
2.7	Related Work	18
2.8	PS/PL Interconnect - AXI4-Lite	18
2.9	ZedBoard FPGA Resources	19
3	Design Specification	21
3.1	Top-Level Abstract Design Specification	21
3.2	Low-Level Abstract Design Specification	22
3.3	Simulation and Testing Specification	22
3.4	Evaluation Specification	23
4	Top-Level Abstract Design	24
4.1	Test Suite Flowchart	25
4.2	Designing the Programmable Logic (PL)	25

4.3	Controlling the Processing System (PS)	26
4.4	Interfacing between PS and PL	26
4.5	Interconnecting Software/Hardware	27
4.6	Software-configuring the Hardware	28
4.7	System Interaction with the User	29
4.7.1	Before the Test	29
4.7.2	After the Test	29
4.7.3	Output Format	29
4.8	How to compile the test suite from the user's perspective?	30
5	Low-Level Abstract Design	31
5.1	Implementation	31
5.2	System Design Choices	32
5.2.1	Histogram Data Width Concatenation	32
5.2.2	Simplifying the Serial 3-Tuple Test	33
5.2.3	Exploiting Degrees of Freedom	34
5.3	Vivado HLS Hardware Specifications and Optimisations	35
5.3.1	Specifying Data Streams	35
5.3.2	Instruction Pipelining	36
5.3.3	Partitioning Arrays	36
5.3.4	Power-On Initialisation of Arrays to Zero	37
6	Simulation and Testing	38
6.1	Functional Testing using Vivado HLS	38
6.2	Integration Testing using Xilinx SDK	41
6.2.1	Debugging Information	41
6.2.2	Histogram Correctness Testing	42
6.2.3	Maximum Sample Size Testing	42
6.3	Functional Testing using Xilinx SDK	42
7	Evaluation	44
7.1	Hardware Resources	44
7.2	Relative Performance	45
7.3	Comparative Analysis	47
8	Future Work and Conclusion	48
9	User Guide	49
10	References	51

A	Appendix	53
A.1	GitHub Repository	53
A.2	Disregarded Tests	53

1 Introduction

Random Number Generators (RNGs) are software programs or hardware devices which produce streams of reasonably unpredictable numbers. RNGs are widely deployed in areas such as statistical experiments, cryptography and gambling, whilst also being used in more niche ways in music, literature and the arts. As it may become apparent from the various uses of RNGs, different people and different disciplines may have different conceptions of when an RNG can be considered "random enough" for their purposes.

In this project, RNGs for use in Monte Carlo (MC) simulations are of particular interest. In MC simulations, RNGs with insufficiently random streams of numbers may introduce significant systematic bias, potentially leading to misinterpretations being made when examining the simulation's biased observations. It is thus in the developer's best interest to determine whether a proposed RNG's output sequence is "random enough".

Software developers usually verify their RNG's "randomness" empirically, through feeding their RNG's output streams into a battery of statistical tests, such as those offered by the popular Diehard (1995) and TestU01 (2007) test suites. Different statistical tests check for different patterns that are thought to constitute non-randomness. A simple example of a statistical test would be to check whether an integer-producing RNG has outputs that are nearly as often odd as they are even, as expected from a good RNG.

However, for researchers and developers interested in high-performance hardware-accelerated MC simulations, the current lack of dedicated hardware-based RNG test suites can render the testing of hardware-accelerated RNGs difficult. Currently, existing popular RNG test suites are exclusively implemented in software and mainly tailored towards testing software RNGs, which renders the analysis of hardware-generated numbers difficult in many ways.

Existing test suites are from an era focused more on functionality rather than throughput; their algorithms are not tuned for concurrent execution, and their constituent statistical tests are run unoptimised and serially on a general-purpose CPU. Furthermore, hardware RNGs can generate trillions of samples, often used to implement RNGs with periods (after which sequence repeats itself) in the order of quadrillions and higher, whilst existing test suites have maximum sample sizes only in the order of billions (TestU01).

Due to these difficulties faced when attempting to analyse hardware-accelerated RNGs' outputs, we believe that the emergence of faster RNGs for high-performance computing applications requires a matching emergence of faster, parallelised implementations of RNG test suites in their respective specialised processing architectures.

1.1 Project Scope

In this project, a user wishes to test FPGA RNGs for use in MC simulations using an FPGA-accelerated RNG test suite. The test suite envisaged should achieve high throughput rates whilst being capable of processing at least trillions of samples.

The goal of this project is to satisfy these requirements by implementing a software-configurable FPGA/microprocessor embedded system solution for empirical statistical testing of FPGA-accelerated RNGs, enabling the user to provide and test a custom RNG.

In the final implementation, throughput speeds at one sample per clock cycle for sample sizes up to 2^{44} have been achieved at the target FPGA platform's maximum clock rate of 100 MHz, for three statistical tests run in parallel. This can facilitate the testing of up to 2^{44} numbers for three randomness properties in approximately 49 hours.

The exact definition of "random enough" in the context of an RNG stream's quality is not within project scope and is left to be decided by the user, as different types of MC simulations require different levels of different randomness metrics which are not known beforehand. Instead, the implemented solution aims at helping the user evaluate their RNG's quality through providing the user with information on the following quality metrics:

- **Distribution:** Does the random number generator follow the distribution (e.g. uniform distribution) it is supposed to?
- **Independence:** Are samples statistically independent of previous samples?

These questions become particularly important when users need to ensure that the distribution and independence hold over trillions of samples. The methodology behind quantifying and measuring the degree to which these two metrics have been met by the user-provided RNG will be covered in Section 2 (Background).

The test suite should be comparable to existing test suites in terms of user interaction, such that users familiar with existing test suites should be able to comprehend the implemented system's main functionalities, such as the test result format and input parameters of tests.

It is important to restate that the implemented system is a form of software-controlled hardware. Thus, computations involved in empirical testing of RNGs will also be split into software and hardware parts. In particular, the system envisaged should facilitate a workflow similar to the one shown on the next page, where the system's steps are shown

in clockwise order, starting at the top left, in the user’s personal computer.

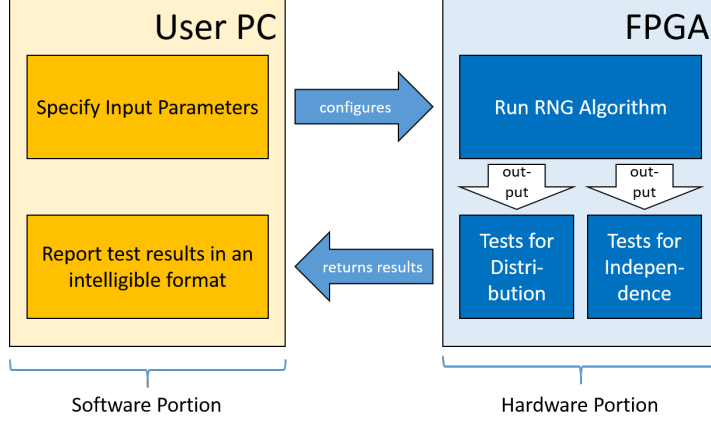


Figure 1: High-level workflow diagram of the proposed test suite

With these requirements in mind, this report investigates the test suite design considerations encountered during the system design and implementation in this project:

- How to split up the embedded design into hardware and software portions?
- Which tests to implement for independence and distribution tests?
- How to design and facilitate user interaction before, during and after test runs?
- How to communicate test results (success/failure) intelligibly to the user?
- How to optimise parallelised system throughput to one sample per clock cycle?
- How to facilitate the parallelised processing of up to 2^{44} samples?

The tests to be implemented will be introduced in Section 2 (Background), and a comprehensive outline of the system’s workflow will be discussed in Section 4 (Top-Level Abstract Design).

1.2 Target Platform

In this project, the ZedBoard Zynq-7000 ARM/FPGA SoC Development Board is used to implement the RNGs and their empirical tests. This FPGA board was chosen a) due to its useful hardware specifications such as the embedded dual-core ARM Cortex A-9 processor easing control of the FPGA, b) due to its integrated software support for creating semiconductor intellectual property cores (IP cores) using high-level synthesis (HLS), as discussed below.

The ZedBoard hardware consists of two parts: The Processing System (PS) portion which consists of the aforementioned ARM Cortex A-9 processor, and the Programmable Logic (PL) portion which consists of the FPGA fabric. Communication between these two parts is facilitated through PS/PL interconnects, as will be outlined in Section 2.8 (PS/PL Interconnect - AXI4-Lite).

The hardware portion of the system is bottom-up designed using the Vivado Design Suite. First, Vivado HLS, which is a subset of the suite, is used to translate C++ specifications of the test suite and of the RNG into IP cores. Next, Vivado IP Integrator is used to create a block design to describe the overall architecture, thus connecting the IP core to the PS. The block design is validated, synthesised and implemented, to then generate the bitstream to program and run the FPGA.

For the software portion of the system, again, Xilinx SDK is used. The SDK is also used to facilitate the reporting and evaluation of the test results, and to benchmark the overall system performance through measuring run-time. The software/hardware interfacing will be outlined in greater detail in Section 4 (Top-Level Abstract Design).

1.3 Report Structure

Section 2 - Background: In order for the content and the contemporary context of this project to be comprehensible, this section will outline the motivations behind the project, alongside the theory behind FPGA acceleration, RNG testing, relevant FPGA hardware concepts, and with discussions of related work.

Section 3 - Design Specification: This section will discuss product deliverables and requirements which will set the technical specifications that the final product must follow.

Section 4 - Top-Level Abstract Design: This section will describe the big picture of the implemented system's architecture. Then, the high-level steps involved in programming the FPGA with the implemented system will be explored from a user's perspective.

Section 5 - Low-Level Abstract Design: This section will elaborate on the design implementation from a lower level of abstraction, through outlining which challenges had to be overcome to achieve the final optimised implementation.

Section 6 - Simulation and Testing: This section will outline the debugging toolchain used in testing the different modular system components.

Section 7 - Evaluation: This section will discuss the system's performance achieved and will benchmark performance with existing software test suites.

Section 8 - Future Work and Conclusion: This section will point out the project's areas of improvement and enhancement, to then finish with some concluding remarks.

Section 9 - User Guide: This section provides a step-by-step manual for compilation and use of the implemented test suite.

2 Background

2.1 Project Motivation

This subsection discusses the motivations for deploying RNGs in FPGAs, to enable discussion of the motivations behind testing RNGs in FPGAs.

2.1.1 Monte Carlo simulations and RNGs in FPGAs

Through their widespread use in solving computationally challenging problems, Monte Carlo (MC) simulations have become an indispensable part of quantitative investigations in finance, science and engineering. Their inherently high computation time, which can span hours or days, is related to the speed of random number generation and the speed of random number consumption; if one does not match the speed of the other, due to data dependencies, the slower component is at risk of becoming a performance bottleneck. Given that high run-time naturally translates to complexity constraints of the computational problem to be solved, this has driven demand for more efficient MC simulation and RNG implementations. Increases in MC simulation efficiency would enable a multitude of new applications that rely on higher throughput.

Recently, [4, 5, 6] have found that FPGAs show significant promise for accelerating MC simulations, primarily through the availability of fine-grain bit-wise operations, allowing high levels of parallelism using hundreds of on-chip arithmetic units. Furthermore, [6] asserts that one way of increasing MC simulation throughput is to formulate computational problems in "embarrassingly parallel" MC simulations, i.e. ones in which subsets of data are processed independently with very little communication whenever possible. FPGAs have been shown by [7] to be suitable for such use due to their high resource-efficiency.

2.1.2 Need for higher-throughput RNG implementations

As discussed, slow RNGs are at risk of becoming MC simulation performance bottlenecks. MC simulations, especially in their embarrassingly parallel form, thus require resource-efficient, high-speed input streams of high-quality random numbers. [7] has found that FPGA-based RNGs are faster and more resource-efficient than their software-optimised counterparts. [6] has benchmarked RNG algorithms optimised for four different hardware platforms, finding that the FPGA shows the highest performance levels, whilst providing an order of magnitude higher performance per joule than any other platform. Furthermore, it is possible and convenient to feed an FPGA-implemented RNG's output sequence directly into MC simulations on the same FPGA, making the platform attractive for such use.

2.1.3 Need for higher-throughput RNG Test Suites

FPGA-accelerated RNGs, like any RNG, have to be tested for their quality. However, their high performance can become a problem if test suites are several orders of magnitude behind in terms of throughput and maximum sample size that can be tested for.

[10] notes that popular software test suites present many performance bottlenecks. Existing popular test suites such as Diehard are not parameterisable, and can only consume up to 2.5M 32-bit integers, whilst recent FPGA-based RNGs such as in [7] can generate 32-bit numbers at 800MHz+ without any optimisation. This would consume more than 320-times the Diehard maximum sample size every second. Whilst the newer TestU01 test suite library can consume up to 2^{38} 32-bit integers for some of its tests, this still implies an analysis of only 6 minutes worth of 800MHz random numbers, whilst [6] notes that in highly parallel architectures, RNGs with periods ranging larger than 2^{160} are of research interest, which is several orders of magnitude larger than TestU01 constraints currently allow for empirical testing.

Additionally, interfacing RNG sequence data between FPGA and software is often much slower than directly performing pipelined and parallelised testing on-chip, such that unnecessary performance bottlenecks can occur when trying to port hardware-generated numbers to software for analysis.

These factors altogether motivate the analysis of random numbers using FPGAs.

2.2 RNG Types under Consideration

RNGs are widely deployed in man-made models where an element of unpredictability alongside simulation repeatability are desired features of the environment. Such applications are common in areas such as cryptography, statistical experiments and simulations. Different types of RNGs exist for different types of application areas, depending on what property of an RNG is deemed as important to make them fit for purpose.

In areas such as cryptography, a traceable pattern in the sequence of generated numbers should be avoided at all costs, as this may pose security risks to individuals and institutions. Here, the "quality" of an RNG would mainly refer to the unpredictability of successive numbers. In the event of an RNG's seed value and a large string of generated numbers being leaked, it should be hard for adversaries to reverse-engineer the number sequence such that already generated and future numbers could be computed.

In contrast, in MC simulations, the RNG's speed of generating numbers may matter more; as aforementioned, in MC simulations, slow RNGs are at risk of becoming simula-

tion bottlenecks. As another requirement, simulation RNGs should have a long enough period (after which the output sequence of numbers repeats itself). This is due to long periods generally improving the quality of the simulation observations made, especially in higher run-time MC simulations requiring larger quantities of random numbers.

Whilst applications in areas such as cryptography and gambling may at times view the sequence's unpredictability as more important than its speed of generation, in contrast, in MC simulations, fast streams of resource-efficiently generated, "unpredictable enough" random numbers are desired. As aforementioned, in this project, the client wishes to test RNGs designed to be used in MC simulations. Using this information, the RNGs under consideration in this project can be limited to the following types.

2.2.1 Pseudo and Deterministic RNGs

RNGs can be broadly classified into two categories: True random number generators (TRNGs), where the generated numbers are purely random and based on polling a physical process such as atmospheric noise, and pseudo random number generators (PRNGs), where the numbers generated approximate a perfectly random sequence. PRNGs operate through an underlying algorithm, which generates output sequences of numbers deterministically; the next number is a mathematical function of the current number.

TRNGs are of particular interest in areas such as cryptography. In contrast, PRNGs are of interest in computer simulations; due to the deterministic nature of PRNG algorithms, PRNG sequences are reproducible, provided identical initial conditions. Since MC simulations need to exhibit the property of simulation repeatability, the FPGA-accelerated RNGs tested, implemented and discussed in this project will be pseudorandom and deterministic, i.e. for identical seed values and number of samples generated, the RNGs should always output identical sequences of numbers.

Upon generating an amount of numbers equal to a PRNG's period, the next number will again be the first generated value, and so on. Thus, PRNGs can be viewed as finite state machines, where each generated number is its own state, where the RNG algorithm is a transition function from the current state to the next state, and where the period is the total number of states. For example, for a simple counter RNG, the RNG would increment the current value by 1 until the current value reaches its highest possible value, where it resets to 0. A counter RNG's period would be the highest possible value + 1.

Hereinafter, unless noted otherwise, the term RNG will imply a PRNG.

2.2.2 Uniform RNGs

RNGs can be further grouped into uniform RNGs and non-uniform RNGs. Uniform RNGs generate numbers which approximately occur with identical frequency over its defined range of possible output values. In contrast, non-uniform RNGs intentionally follow different frequency distributions such as Gaussian or exponential distributions. The standard approach for implementing a non-uniform RNG is to take a uniform RNG and use mathematical methods to plot its output to a desired non-uniform distribution.

[6] has shown that RNGs on the FPGA platform achieve their highest performance for optimised uniform RNGs in particular, with a throughput higher by several orders of magnitude compared to optimised Gaussian or exponential RNGs. Given these findings, and given the tight project time frame, only uniform RNGs are within project scope.

2.2.3 32-Bit Unsigned Integer RNGs

Popular RNG test suites, including Diehard and TestU01, usually evaluate 32-bit RNGs by default. RNG benchmarking literature also frequently reports RNG performance metrics for 32-bit numbers. For performance benchmarking purposes, this project likewise implements tests for 32-bit integer RNGs. For simplicity's sake during project implementation and evaluation stage, the range of integers will be considered to be unsigned. For TRNGs, this would imply that given a long enough sample size, every number between zero and 4,294,967,295 ($2^{32} - 1$) would occur at some point in the sample sequence.

2.3 Statistical Testing of RNGs

2.3.1 How to assess the Statistical Properties of RNGs?

For the analysis of RNG algorithms and their statistical properties, a well-established approach is to apply theoretical, then empirical testing on the RNG.

In theoretical statistical testing, the behaviour of the RNG's underlying algorithm is examined mathematically without running the RNG algorithm. This is usually done through studying the correlations within one period of the RNG's output sequence. Since the algorithms' underlying mathematical structure is assumed to remain unchanged irrespective of the developer's chosen platform for implementation, and given that there already exists extensive background literature on the theoretical statistical profile of many popular RNGs, theoretical statistical testing is considered to be outside of project scope.

In empirical statistical testing, the RNG algorithm is run in conjunction with an empirical statistical test suite. As aforementioned, RNG algorithms perform mathematical operations on the current value of the sequence to generate the next value. When

deployed with empirical statistical tests, together, they function as an MC simulation: The RNG's generated output is analysed in real-time, such that every time a sample is produced by the RNG, it is immediately consumed by an empirical test to be analysed, and the process is repeated for the total number of samples requested. In this project, each FPGA clock cycle, one sample is produced and analysed by three empirical tests in parallel.

2.3.2 Interpreting Empirical Test Results

An empirical test's goal is to find patterns indicating non-randomness according to a measure of what is thought to constitute non-randomness. To reuse a previous example, a simple heads/tails test would check whether an integer-producing RNG has outputs that are nearly as often odd as they are even, as would be expected from a good RNG. Passing more empirical tests provides stronger confidence that the sequence under scrutiny is indeed random, and that its underlying RNG and its algorithm are good. Furthermore, [1] suggests that it could be generalised that a *bad* RNG is one that fails *simple* tests and a *good* RNG is one that passes *complicated* tests.

Each empirical test transforms samples generated from a random sequence u_0, u_1, \dots, u_n to a real number Y , the test statistic, from which a pass or fail decision can be made. The evaluation of Y is specific to the type of test, and will be discussed in the next subsection. However, classifying whether an RNG has passed a specific test is an ambiguous process as it is impossible to prove whether a sequence is perfectly random based on empirical testing alone. Instead, TestU01 and Diehard convert Y into a p -value (probability value) which can take on values ranging from 0 to 1, and is then reported to the user. The method of transforming Y to its p -value is specific to each test, and those relevant to the empirical tests implemented in this project will be discussed in this subsection. The p -value can be used by the user as guidance when deciding whether an RNG has almost certainly passed a certain test, or whether retesting is required with different parameters.

p -values are commonly known in the context of hypothesis testing, where conventionally a null hypothesis H_0 is rejected or accepted based on whether the reported p -value falls into a rejection area R , usually defined by a significance level threshold α . However, TestU01 does not let users specify R nor α . The TestU01 documentation [2] cites as reasons that in the context of RNG testing, the RNG sample size is usually huge and can be increased at will, making a fixed R and α inappropriate.

Thus, unlike the standard procedure recommended in many statistical textbooks, where one should reject H_0 if and only if $Y \in R$, TestU01 simply reports the p -value. Both TestU01 and Diehard documentation [3] note that for perfectly random RNGs, the reported p -value would approximate a random variable $U(0, 1)$ under H_0 , i.e. values be-

tween 0 and 1 should happen with about equal probability. An RNG is usually considered to fail a test if the p -values are clustered close to 0 or 1 for multiple iterations of a test with different parameters. [1] defines the p -value as follows:

$$p = P[Y \geq y | H_0] \quad (1)$$

where y is the value taken by test statistic Y .

In order to achieve the aforementioned goal of implementing a system with input/output comprehensible for users familiar with existing RNG test suites, in this project, each implemented test reports p -values using the definition above, as this methodology is well-established and easily understood by the target demographic.

2.4 Empirical Statistical Test Algorithm

Given the FPGA's fine-grain bit-wise computation possibilities, the approach used in this project is to make use of this peculiarity through base conversion; noting that all 32-bit unsigned integers from 0 to 4,294,967,295 inclusive can be represented in base-2, each of the resulting constituent bits 0 to 31 can be analysed separately and in parallel for certain randomness properties in the tests under consideration.

More accurately, each test compares an expected histogram of some randomness figure to the observed distribution of this figure, similarly to the example shown below.

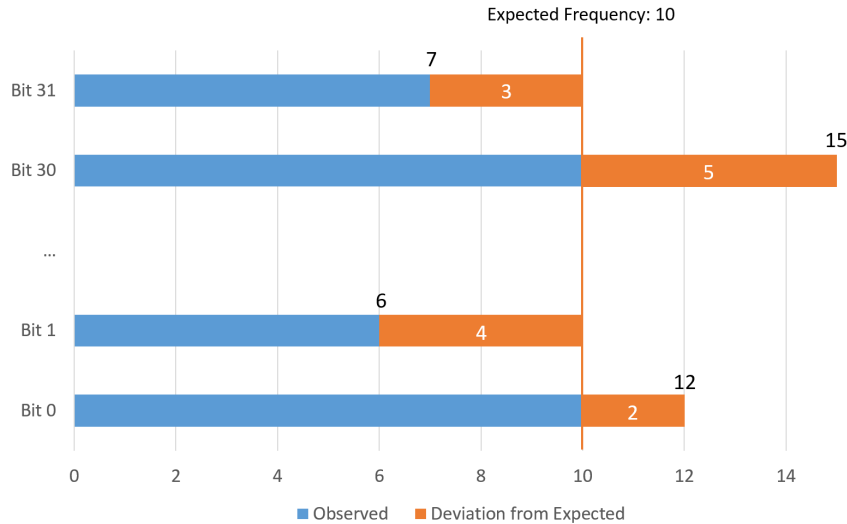


Figure 2: Example observed frequency histogram for arbitrary expected frequency 10

The measured deviations (amount of error) between observed and expected distributions are then used for the analysis of whether these are likely to have occurred by chance. This can be achieved through chi-squared tests.

2.4.1 Chi-Squared Test

In the chi-squared test, the test statistic Y is used to determine the likelihood of an observed frequency distribution being consistent with its theoretical distribution. The observations must be independent. Observations are grouped into k categories and the total number of observations is denoted by n . Then, Y is given by:

$$Y = \sum_{s=0}^{k-1} \frac{(Y_s - np_s)^2}{np_s} \quad (2)$$

where Y_s is the observed number of 1s in category s and p_s is the expected probability of an observation occurring in category s .

In this project, each implemented test is a variation of the chi-squared test. In particular, each test provides different Y_s , p_s , n and k according to the randomness property investigated, from which each test's own Y is computed.

For each test, n is a function of the tested sample size and how many samples are required for one observation, k is dependent on the test, and Y_s varies depending on the randomness property investigated by each test and the tested RNG. Given that all tests implemented in this project are a form of chi-squared test, it will thus be required to determine these four parameters as inputs for the calculation of their respective Y .

To summarise, the parameters required for the computation of Y are as follows:

Parameter	Description
n	total number of observations
k	number of observation categories
p_s	expected probability of observing category s
Y_s	number of observations in category s

where $0 \leq s \leq k - 1$.

2.4.2 Computing p -values

p -values are computed by the chi-squared cumulative distribution function. Provided a test's degrees of freedom v (which differs for each test), the function is given by:

$$p = F(Y; v) = \frac{\gamma(\frac{v}{2}, \frac{Y}{2})}{\Gamma(\frac{v}{2})} \quad (3)$$

where $\gamma(s, x) = \int_0^x t^{s-1} e^{-t} dt$ and $\Gamma(s) = \int_0^\infty t^{s-1} e^{-t} dt$ are the lower incomplete Gamma function and the Gamma function, respectively.

In the implemented system, approximations of the chi-squared cumulative distribution functions are used to compute p -values. As aforementioned, an RNG will usually fail a test if the p -values are clustered close to 0 or 1 for multiple iterations of the test. Since users are mainly interested in whether their RNGs have failed, it was decided that slight precision losses of the p -value computations were appropriate, given that their hardware resource footprint could be minimised. The algorithms deployed to approximate p -values and their documentation are public domain available [12], and their version used in this project is available on the project’s GitHub repository (see appendix).

2.5 Empirical Tests under Consideration

2.5.1 Selection Criteria

Each test implemented in this project has been selected using the following criteria:

- It analyses one target RNG quality metric (either distribution or independence)
- It has light FPGA resource footprint, such that other tests can be run in parallel
- It can be optimised to process one sample per clock cycle
- It meets FPGA design timing constraints, allowing 100MHz maximum frequency

Initially, five algorithms were shortlisted for their diverse range of randomness properties investigated, and for their perceived straightforward implementation: Frequency Test, Serial Correlation Test, Gap Test, Count the 1s Test and Poker Test.

However, when trying to synthesise their optimised algorithms in Vivado HLS, it was found that Gap Test, Count the 1s Test and Poker Test were not realisable in the target platform. The reasons behind these were related to exceeding timing constraints and requiring more than the available amount of hardware resources which will be outlined in Section 2.9 (ZedBoard FPGA Resources). An early indicator of the disregarded tests’ infeasible design was the notably increased synthesis time in Vivado HLS (20min+)

In the interest of their possible future implementation in higher performance FPGAs or in more efficient hardware description languages such as Verilog/VHDL, discussions of the disregarded tests and their methodologies can be found in more detail in the appendix.

It was thus decided that only three of the working tests, the frequency test, and two variations of the serial correlation test, would be included in the final parallelised

implementation. The frequency test analyses distribution correctness, whilst the serial correlation tests analyse sample independence. The following subsections provide the necessary background to enable discussion of the implementation and evaluation of these three tests. It is important to remember that all of these three tests are variations of the chi-squared test and thus follow similar methodologies for Y and p -value computations.

2.5.2 Frequency Test

The frequency test evaluates whether the deviation between observed and expected distribution of 1s for each of the 32 bits is likely to have occurred by chance. This can be done through declaring an array of size 32 to track the number of 1s observed in each bit. Through accumulating an array element by 1 whenever the respective bit is a 1 for the sample number, the frequency test can be algorithmically described as follows:

Algorithm 1: Frequency Test Algorithm

```

input : seed and sampleSize
output: histogram of freqCount of size 32

1 for  $i \leftarrow 1$  to sampleSize do                                // for every sample
2   for  $j \leftarrow 0$  to 31 do                                       // for every bit
3     freqCount [ $j$ ]  $\leftarrow$  freqCount [ $j$ ] + ((seed >>  $j$ )&1); // +1 per observation
4   seed  $\leftarrow$  seedNext(seed);                                   // generate next sample

```

where the function *seedNext* is the user-specified RNG algorithm, generating the next sample based on the current sample.

In order to calculate the frequency test's p -value based on the resulting *freqCount* histogram, Y has to be determined first, using the following parameters:

Parameter	Description	Value
n	total number of observations	<i>sampleSize</i>
k	number of observation categories	32
p_s	expected probability of observing category s	$\frac{1}{2}$
Y_s	number of observations in category s	RNG-specific

where $0 \leq s \leq 31$.

Given an expected 50% chance of a 1 occurring in each of the tracked 32 bits, $p_s = \frac{1}{2}$ for all bits s , whilst $k = 32$. Using Y and the number of degrees of freedom given by $v = k = 32$ (since none of the bit buckets convey information about another), these parameters can be used to compute the p -value.

2.5.3 Serial 2-Tuple Correlation Test

The serial n -tuple correlation test evaluates whether the observed distribution of n -tuple successive numbers is approximately uniformly distributed as expected. In the simplest case for $n = 2$, the 2-tuples can be categorised into the four possible buckets 00, 01, 10, 11. Note that in order to satisfy the chi-squared test's assumption of independent observations, the n -tuples must be non-overlapping.

In its algorithmic implementation, this test can be realised through declaring an array of size $4 \times 32 = 128$ to track the 4 different kinds of 2-tuples as observed in each of the 32 bits. Taking into account the required non-overlapping observations property, the test can be algorithmically described as follows:

Algorithm 2: Serial 2-Tuple Correlation Test Algorithm

```

input  : seed and sampleSize
output: histogram of serial2Count of size 128

1 for  $i \leftarrow 1$  to sampleSize do
2    $seedPrev \leftarrow seed$ ;           // cache the previous sample
3    $seed \leftarrow seedNext(seed)$ ;    // generate the next sample
4    $serialEnable \leftarrow !serialEnable$ ; // prevent sample overlapping
5   if  $serialEnable = true$  then
6     for  $j \leftarrow 0$  to 31 do (           // for every bit
7       if  $((seedPrev \gg j) \& 1) = 0$  then
8         if  $((seed \gg j) \& 1) = 0$  then           // Tuple 0: 00
9            $serial2Count[j * 4]++$ 
10        else  $serial2Count[j * 4 + 1]++$ ;         // Tuple 1: 01
11      else
12        if  $((seed \gg j) \& 1) = 0$  then           // Tuple 2: 10
13           $serial2Count[j * 4 + 2]++$ 
14        else  $serial2Count[j * 4 + 3]++$ ;         // Tuple 3: 11

```

In order to calculate the p -value of the serial 2-tuple correlation test's results, Y has to be determined first, using the following parameters:

Parameter	Description	Value
n	total number of observations	$\frac{sampleSize}{2}$
k	number of observation categories	128
p_s	expected probability of observing category s	$\frac{1}{4}$
Y_s	number of observations in category s	RNG-specific

where $0 \leq s \leq 127$.

For each of the 32 bits, there are 4 buckets. Thus, $k = 32 \times 4 = 128$ and $v = k - 32 = 96$ (since a fourth tuple bucket can be calculated given the first three tuple buckets' values, and this can be done for each of the 32 bits). Given that each bit has an expected 50% chance of a 1 occurring in each sample, each tuple is expected to be uniformly distributed with $p_s = \frac{1}{4}$ for all tuples s . Using k , p_s and v , first Y and then the p -value can be evaluated, similarly to the frequency test.

2.5.4 Serial 3-Tuple Correlation Test

The same test can be scaled to triples. 3-tuples can be categorised into the eight possible buckets 000, 001, 010, 011, 100, 101, 110, 111. Note that in order to satisfy the chi-squared test's assumption of independent observations, the 3-tuples must be non-overlapping.

In its algorithmic implementation, this test can be implemented through declaring an array of size $8 \times 32 = 256$ to track the 8 different kinds of 3-tuples as observed in each of the 32 bits. Taking into account the required non-overlapping observations property, the test can be algorithmically described as shown on the next page.

In order to calculate the p -value of the serial correlation test's results, Y has to be determined first, using the following parameters:

Parameter	Description	Value
n	total number of observations	$\frac{\text{sampleSize}}{3}$
k	number of observation categories	256
p_s	expected probability of observing category s	$\frac{1}{8}$
Y_s	number of observations in category s	RNG-specific

where $0 \leq s \leq 255$.

For each of the 32 bits, there are 8 buckets. Thus, $k = 8 \times 32 = 256$ and $v = k - 32 = 224$ (since an 8th bucket can be calculated given the first 7 buckets' values, and this can be done for each of the 32 bits). Given that each bit has an expected 50% chance of a 1 occurring, each tuple is expected to be uniformly distributed with $p_s = \frac{1}{8}$ for all tuples s . Using k , p_s and v , first Y and then the p -value can be evaluated, similarly to the previous two tests.

Algorithm 3: Serial 3-Tuple Test Algorithm

input : *seed* and *sampleSize*

output: histogram of *serial3Count* of size 256

```
1 enableSerial = 0;

2 for  $i \leftarrow 1$  to sampleSize do
3   seedPrevPrev  $\leftarrow$  seedPrev;          // cache previous previous sample
4   seedPrev  $\leftarrow$  seed;                  // cache previous sample
5   seed  $\leftarrow$  seedNext(seed);           // generate next sample
6   enableSerial ++;

7   if enableSerial=3 then                    // prevent sample overlapping
8     enableSerial = 0;
9     for  $j \leftarrow 0$  to 31 do              // for every bit
10      if ((seedPrevPrev >>  $j$ )&1) = 0 then
11        if ((seedPrev >>  $j$ )&1) = 0 then
12          if ((seed >>  $j$ )&1) = 0 then        // Tuple 0:  000
13            serial3Count [ $j * 8$ ]++;
14          else serial3Count [ $j * 8 + 1$ ]++ ;    // Tuple 1:  001
15        else
16          if ((seed >>  $j$ )&1) = 0 then        // Tuple 2:  010
17            serial3Count [ $j * 8 + 2$ ]++;
18          else serial3Count [ $j * 8 + 3$ ]++ ;    // Tuple 3:  011
19      else
20        if ((seedPrev >>  $j$ )&1) = 0 then
21          if ((seed >>  $j$ )&1) = 0 then        // Tuple 4:  100
22            serial3Count [ $j * 8 + 4$ ]++;
23          else serial3Count [ $j * 8 + 5$ ]++ ;    // Tuple 5:  101
24        else
25          if ((seed >>  $j$ )&1) = 0 then        // Tuple 6:  110
26            serial3Count [ $j * 8 + 6$ ]++;
27          else serial3Count [ $j * 8 + 7$ ]++ ;    // Tuple 7:  111
```

2.6 RNG Algorithms under Consideration

As aforementioned, the goal of the implemented system is to facilitate the testing of a user-specified RNG algorithm. In the following, some popular RNGs will be introduced in ascending order of quality, which were implemented and tested for in this project to simulate the failing of different combinations of tests when the user is testing custom RNGs. The results of these tests will be outlined in Section 6 (Simulation and Testing).

RANDU: RANDU is an RNG widely used in the 1970s despite its bad randomness properties. Its generated numbers corresponding to the recurrence:

$$V_{j+1} = 65539 \times V_j \bmod 2^{31} \quad (4)$$

where V_0 is an odd number. The generated numbers are not uniformly distributed and the generator produces only odd numbers. The generator is expected to fail all tests.

Simple Counter: A simple counter outputs all possible 32-bit unsigned integer values in ascending order, and resets to zero upon reaching the final possible value. This RNG is expected to pass the frequency test, but is expected to fail the serial test in both its variations, as the ascending order number stream obscures tuple distributions.

ranqd1: This RNG algorithm is from [15], using a suggested recurrence relation:

$$V_{j+1} = 1664525 \times V_j \quad (5)$$

Marsaglia's Xorshift Generator: An Xorshift generator takes an exclusive-or of a number with a bit-shifted version of the current RNG value. The version implemented in this project is a public domain available one [16]. This RNG is expected to pass all tests.

```
1 static unsigned long x=123456789, y=362436069, z=521288629;
2
3 unsigned long xorshf96(void) { //period 2^96-1
4 unsigned long t;
5     x ^= x << 16;
6     x ^= x >> 5;
7     x ^= x << 1;
8     t = x;
9     x = y;
10    y = z;
11    z = t ^ x ^ y;
12    return z;
13 }
```

2.7 Related Work

As inspiration for the RNG tests which were tried to be implemented on the FPGA, the test suites TestU01 and Diehard have been used, alongside existing literature discussing the theory behind RNG testing from Jansson [8] and Knuth [9]. As aforementioned, the selected test algorithms have been modified and optimised to suit the FPGA platform better, as has been introduced in Sections 2.4 (Empirical Statistical Test Algorithm) and 2.5 (Empirical Tests under Consideration) and as will be discussed in greater detail in Section 5 (Low-Level Abstract Design).

For benchmarking of the implemented test suite in Section 7.2 (Relative Performance), a battery of tests provided in TestU01, called *Alphabit* will be used. The reasons behind choosing Alphabit is that it is arguably the closest and most modern contender for targeting users wishing to test their hardware RNGs, as indicated by their documentation [2] (p.164) noting that "*Alphabit has been defined primarily to test hardware random bits generators*". Furthermore, two of their constituent tests methodologically resembled the frequency and serial test implemented in this project. This will also be discussed in greater detail in Section 7.2 (Relative Performance).

Other than existing software test suites and academic literature, to the best of our knowledge, there currently exists no directly related work on the implementation of RNG tests using FPGAs, as this seems to not yet have been attempted. Whilst there have been some attempts to address test suite throughput issues in software, e.g. with [11] using multi-threading in software to achieve a 4x speedup of the TestU01 library, this is still several orders of magnitude too slow for faster FPGA RNGs.

2.8 PS/PL Interconnect - AXI4-Lite

Given the ZedBoard's peculiar system architecture, the PS and PL portions of the board have to communicate using proprietary bus architectures. The bus architecture designed by ARM in conjunction with Xilinx for their commercially available hardware, including the ZedBoard, is the Advanced Microcontroller Bus Architecture (AMBA). AMBA offers several types of 32-bit interfaces between IP cores.

The Advanced eXtensible Interface (AXI) is the main protocol for connecting IP cores in the Vivado Design Suite, with the newest version of AXI being AXI4. AXI4 interfaces need to be used when connecting differently clocked hardware blocks; given that PS and PL have differing clock rates (33.333 MHz and 100 MHz, respectively), it is essential to use AXI4 interfaces to interconnect these portions of the ZedBoard.

There are three types of AXI4 interfaces available: AXI4 (the default), AXI4-Lite

(lighter memory footprint) and AXI4-Stream (for burst data transfers). In this project, the AXI4-Lite slave interface is used to allow the HLS-generated test suite block design to be controlled by the PS. In AXI4-Lite interfaces, data can move in both directions between a specified master and slave, simultaneously. Although AXI4-Lite does not allow burst transactions like its heavier-weight alternatives do, this was not found to be an issue, as the proposed design does not require the facilitation of data bursts between IP blocks.

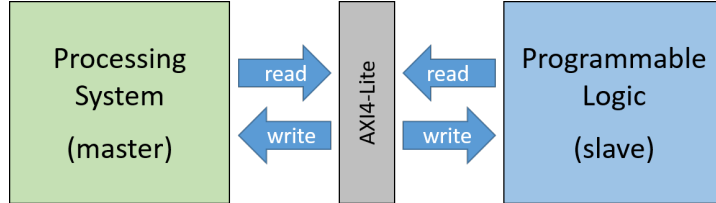


Figure 3: AXI4-Lite serves reads and writes between ZedBoard PS and PL portions

The implementation of AXI4 in the context of the proposed design is specified in Vivado HLS, as will be outlined in Section 4.4 (Interfacing between PS and PL).

2.9 ZedBoard FPGA Resources

When implementing a proposed design on the ZedBoard platform, the board’s available hardware resources cannot be exceeded. To help developers help plan ahead whether their designs are realisable in their proposed form on their chosen FPGA platform, Vivado HLS offers a *Synthesis Report* functionality which can estimate hardware utilisation for every major type of FPGA hardware resource type post-HLS and prior to actual implementation.

To minimise the proposed system’s development time, it was essential early on in the development stage to understand the variety of the target FPGA board’s hardware resources and how to make effective use of the resource types relevant for this project implementation.

In the following, the available hardware resources as provided by the ZedBoard platform will be outlined, using Xilinx guides as references [13, 14]. It will also be noted whether and how heavily the resource type was made use of in the final system implementation.

- **BRAM_18K:** Not used except for storing the AXI4-Lite interface. 18kbit Block Random Access Memory can help load/store up to 2048 bytes of data quickly. In their dual-port form, BRAMs can facilitate a read/write for each clock cycle, however, they can only access one address per clock cycle, making them unsuitable for use in implementing the throughput-critical histograms in this project.

- **DSP48E:** Not used, however, might be worth considering for user-implemented RNGs. Digital Signal Processing slices have 25×18 twos-complement multipliers, 48-bit accumulators and other functions useful for DSP applications. However, the computations made in hardware are mainly bit-wise computations and accumulators, making the DSP48E redundant in the test suite implementation.
- **FF:** Used moderately. Flip-flops can be thought of as registers, used to maintain or change its stored value once per clock cycle.
- **LUT:** Used extensively in all tests, so much that some disregarded tests became unimplementable according to Vivado HLS. Lookup tables can be used to efficiently encode Boolean functions, which are used extensively in this project.

However, care should be taking when interpreting these estimates. The surprising deviation between estimated hardware utilisation and actual hardware utilisation will be outlined in Section 7.1 (Hardware Resources).

3 Design Specification

In this section, the requirements captured in the planning process of this project will be discussed, and measures taken to allow their implementation and facilitation using the software tools and hardware provided. From these specifications, the key features necessary for the design will become apparent, which will serve as background for the discussion of the low-level workings of the proposed design later in this report.

3.1 Top-Level Abstract Design Specification

As aforementioned, fundamentally, the system should be addressing the needs of a user wishing to empirically test trillions of custom FPGA-RNG-generated samples at higher throughput rates and at higher maximum sample sizes compared to existing test suites.

Vivado HLS C++

In this project, the test suite and the example RNGs were written in Vivado HLS C++ to reduce development time given the tight project time frame, whilst ensuring sufficient granular command of the system behaviour through directive commands in Vivado HLS, as outlined in detail in Section 5.3 (Vivado HLS Hardware Specifications and Optimisations).

Alternatively, it could have been possible to implement these in a hardware description language such as Verilog/VHDL, which could have been fine-tuned for better performance. However, the expected higher implementation time was seen as inappropriate for the tight project time frame.

Specifiable RNGs and input parameters

The user should be able to configure the test suite, through providing Vivado HLS C++ code of the custom RNG to be tested, and through specifying test suite input parameters for the desired sample size and the desired seed value.

Reporting histograms and p -values for each test

Upon finishing the simulation, the test results should be returned to the user in an eligible output format, which should aid users in forming a test pass, fail or retest needed decision for their custom RNG. As aforementioned, the test suite is intended for use in helping its users determine whether their proposed custom RNG algorithms are good enough for their intended application area. Hence, the results should be reported to the user based on a) histograms to see bit-level weaknesses of the implemented RNGs and b) p -values.

User feedback

It would be useful for the user to know the entered parameters and when to expect results.

Architecture Hardware/Software Split The hardware/software split of the embedded system’s computations should be well-reasoned; one-off, resource-intensive computations should generally be moved to software to reduce hardware resource footprint, whilst repeated, performance-critical computations should generally be hardware-accelerated.

3.2 Low-Level Abstract Design Specification

Balance between throughput and hardware resources used

Important to note is that in this project, maximising throughput efficiency and hardware resource efficiency of the test suite are almost equally important. Trivially, the former is important to ensure faster test runs, whilst in this project, the same also applies for the latter; it becomes essential to parallelise as many resource-efficient tests as possible, when the test suite has reached the maximum throughput rates possible for the target FPGA platform, which has been achieved in this project.

To achieve the above qualitative statements, some quantitative performance objectives have to be met, leading to the system requirements outlined below. Explicitly, the implemented test suite must:

- process samples equal to the FPGA’s maximum frequency (100MHz)
- implement at least two tests in parallel
- be reducing hardware resources used whenever possible
- compute p -values precise enough for trillions of samples
- be realisable with the Vivado Design Suite and Xilinx SDK

Implementation

In order for the system to be realisable, the system’s constituent components must be

- Synthesisable in Vivado HLS and Vivado IP Integrator
- Implementable in Vivado IP Integrator
- Correctly software/hardware interfaceable in Xilinx SDK

3.3 Simulation and Testing Specification

In order for the system to be considered to be useful, it needs to perform all calculations correctly, or when numerical approximations are made, as discussed in Section 5.2 (System Design Choices), their trade-offs must be reasonable and the underlying computations need to be accurate. In order to verify the test suite’s correctness, and in order to

simulate and evaluate the user’s experience when using the test suite, different RNGs will be tested in Vivado HLS and Xilinx SDK, using the RNGs outlined in Section 2.6 (RNG Algorithms under Consideration), to check whether every feature works as intended.

More specifically, in order for the test suite to be considered correct to a high degree of confidence, there must be a set of test bench codes at every major compilation stage of the design to check that each portion of the design is working as expected.

3.4 Evaluation Specification

First, the hardware resources estimated by Vivado HLS, as well as the actual hardware resources used by the final system will be analysed.

Furthermore, in order for the system to be put into context with related work, TestU01’s *Alphabit* battery of tests will be used as a benchmark. Methodologically, the performance of relevant tests within Alphabit will be analysed in terms of throughput.

Their findings will lead to a discussion of the advantages and disadvantages of the test suite and its use in its intended and some potential unintended application areas.

4 Top-Level Abstract Design

In this section, the implemented design will be viewed from its highest levels of abstraction, such that the overall split of hardware and software functionalities of the embedded system becomes clear. Afterwards, the different computations performed in software and hardware will be outlined, followed by an analysis of their designed interaction. Throughout these discussions, the design decisions made in this project will be explained.

When viewed holistically, the implemented system incorporates all elements outlined in the top-level abstraction diagram below.

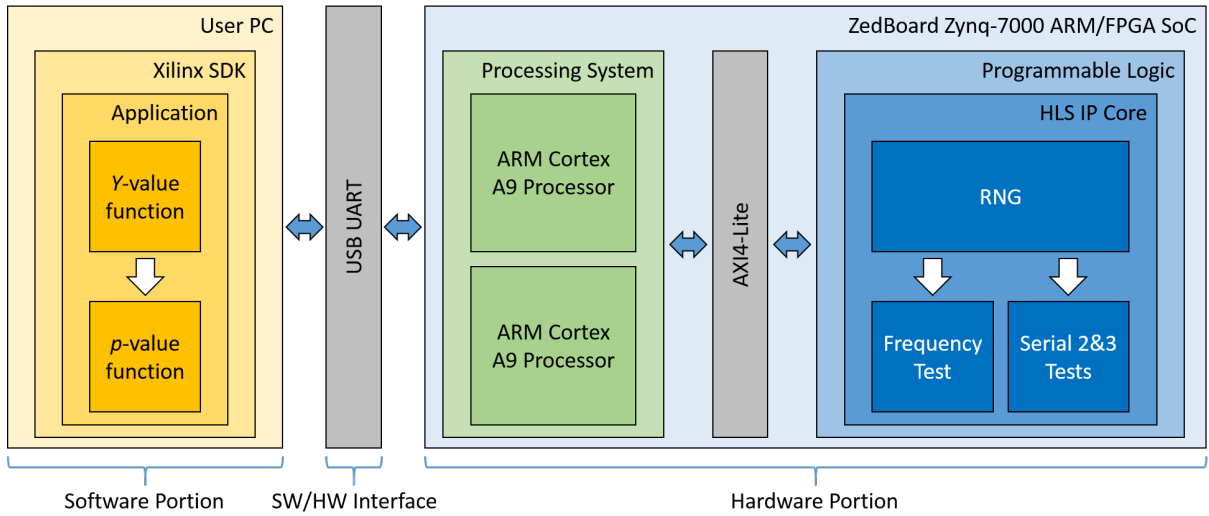


Figure 4: Top-level design schematic of the implemented system

Before jumping into discussions of how these elements are technically implemented in Section 5 (Low-Level Abstract Design), the following subsections will provide discussions of a top-level abstraction of design through breaking the design down into its main constituent components of hardware (PL and PS) and software.

The system will first be viewed in a holistic flowchart to outline how the Top-Level Design Specifications outlined in Section 3.1 (Top-Level Design Specification) have been realised in this project. Next, the hardware portion of the design will be outlined, with an examination of the Programmable Logic (PL) and the Processing System (PS) separately, followed by a discussion of how to interconnect PL/PS. Then, the software portion of the design will be outlined, and how interfacing of hardware/software is managed in the implementation. Finally, these components will be viewed from the user's perspective when trying to compile the design, outlining the compilation stages involved in porting and running the RNG and the test suite on the FPGA.

4.1 Test Suite Flowchart

In this section, the top-level architecture will be specified with its main components. When these pieces are put together, the resulting flowchart is as follows from the user's perspective, when the design has been successfully compiled and ported on the FPGA.

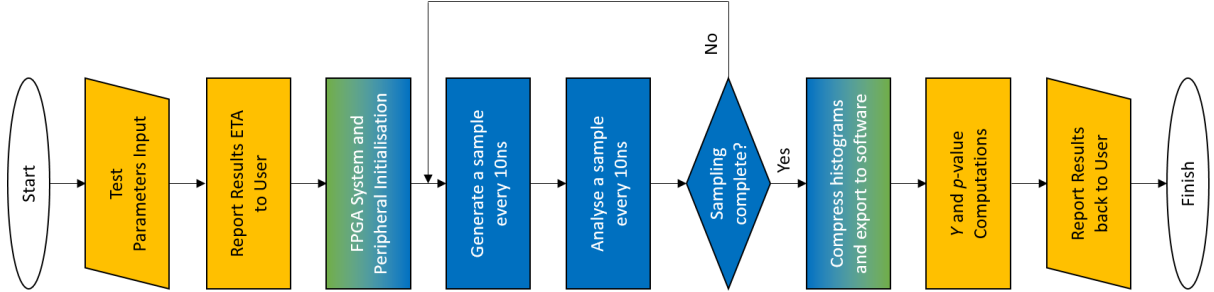


Figure 5: Flowchart of a test suite run in Xilinx SDK

where the orange, blue-green and blue boxes indicate that the processing is performed in software, in PS/PL or just in PL, respectively, and where the parallelogram, rectangle and diamond boxes indicate that the processing performed is input/output processing, data processing or decision-making, respectively.

4.2 Designing the Programmable Logic (PL)

In this project, to minimise data dependency complexity between interconnected IP cores, all Vivado HLS C++ code has been compiled into a single Vivado HLS block. This IP core is then connected to the PS IP Core using the Vivado IP Integrator. Important to note is that due to the chosen "one HLS block" approach, the user is required to compile the custom RNG algorithm alongside the test suite, in Vivado HLS. It is also important to remember that in order for the PS to be able to communicate with the PL-implemented test suite IP core, the AXI4-Lite interface needs to be specified in Vivado HLS.

The HLS-generated IP block should consist of the user-defined RNG on the number-generating side, and the Frequency Test and Serial Tests on the number-consuming side.

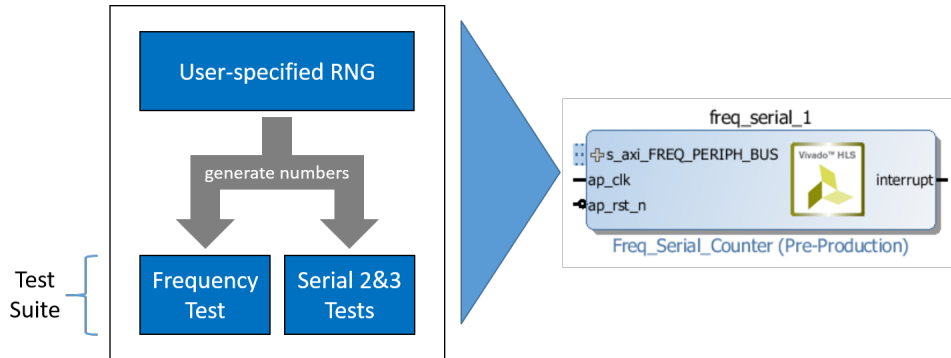


Figure 6: HLS IP Core structure (left) and representation in Vivado IP Integrator (right)

The IP block should function algorithmically as shown below, where the functions `freqTest`, `serialTwoTest`, `serialThreeTest` and `seedNext` are all run in parallel.

Algorithm 4: Test Suite IP Core Algorithm as specified in Vivado HLS

```

input  : seed and sampleSize
output: histogram of freqCount of size 32, histogram of serial2Count of size 128
          and histogram of serial3Count of size 256

1 for  $i \leftarrow 1$  to sampleSize do
2   freqTest (seed);
3   serialTwoTest (seed);
4   serialThreeTest (seed);
5    $seed \leftarrow \text{seedNext}(seed)$ ;           // call user-defined RNG algorithm

```

When this algorithm is compiled, it is worth noting that the `seedNext` function is inline expanded during compilation, substituting the function call site with the body of the called function. This way, there is a clear separation between user-configurable RNG and the test suite during user-specification of the RNG in Vivado HLS C++.

Note also that due to the RNG sequence data dependence between RNG and test suite, the achievable speed of the test suite will depend on the speed of the user-supplied RNG algorithm. Hence, care should be taken by the user to optimise the RNG to achieve close to one generated number per clock cycle.

4.3 Controlling the Processing System (PS)

For running the PS, two options were available: Running FPGA applications using a version of Linux, or running applications "bare-metal", i.e. through direct communication with hardware. Whilst Linux might have enabled closer control over code execution, in the end, the bare-metal approach was chosen for sake of simplicity. In particular, the required functions implemented were found to be straightforward in nature, which helped completely specify communication with hardware without operating system support. Some of the hurdles that had to be overcome to ensure PS control correctness will be discussed in Section 6 (Simulation and Testing).

4.4 Interfacing between PS and PL

The HLS IP Core created has to be interfaced with the PS in order to be controllable by the user. This is facilitated through AXI4-Lite interfaces as introduced in Section 2.8 (PS/PL Interconnect - AXI4-Lite). In particular, the data sent to the PL will be input parameters, with the results reported back being the histograms of each respective test.

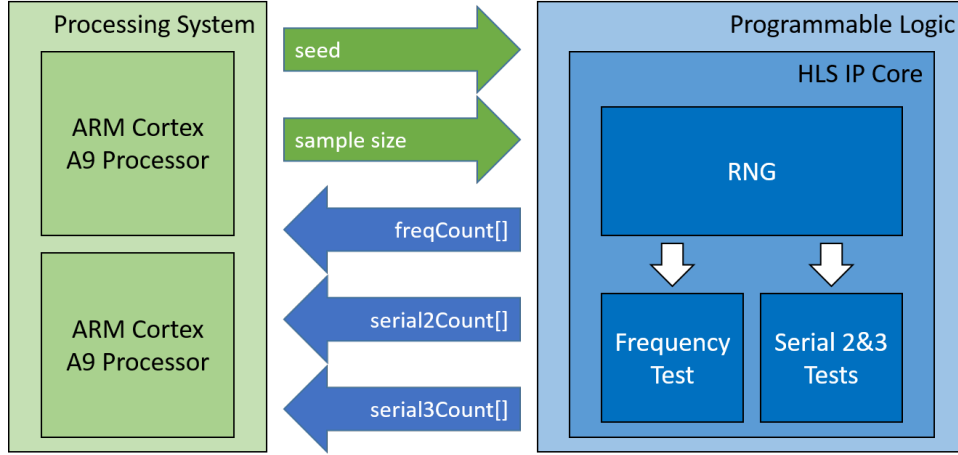


Figure 7: Overview of data (excluding metadata) exchanged between PS and PL

The outputted histograms can then be compared with the expected distributions for each test, thus allowing for the Y and p -value computations on the user's PC.

4.5 Interconnecting Software/Hardware

The physical interconnecting between the user's PC and the ZedBoard is enabled through USB-to-UART connections. The ZedBoard has an integrated USB-to-UART bridge connected to a PS UART peripheral, which, when conceptualised from a higher level of abstraction, will connect to the software as shown in Figure 8.

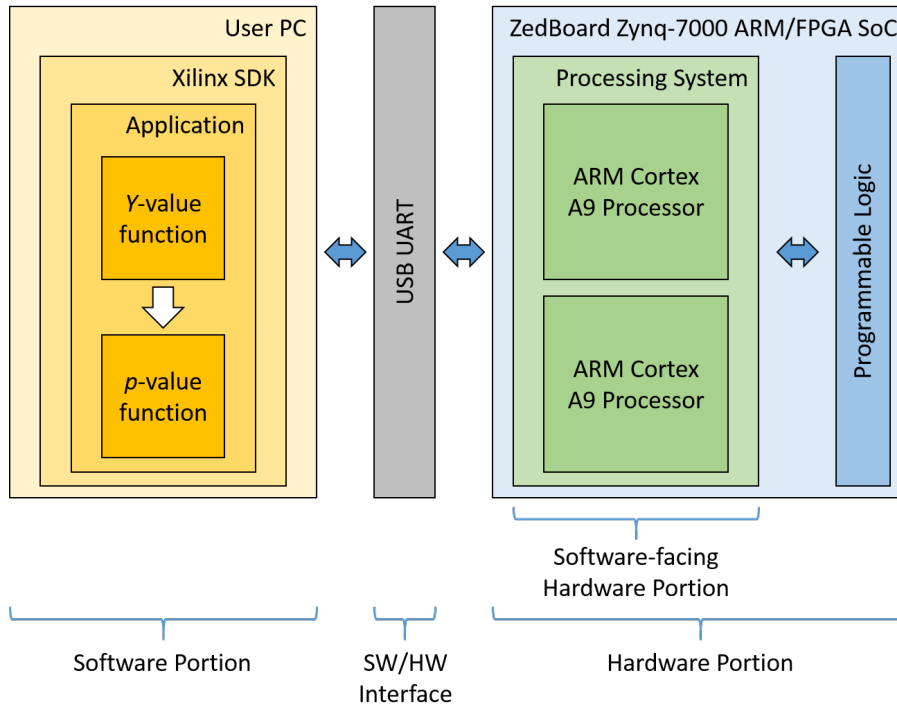


Figure 8: Overview of interface between software and PS

Xilinx SDK will be required for the programming and controlling of the FPGA from the user's PC. In Xilinx SDK, different applications can be programmed, using an automatically generated board support package, which allows for the ZedBoard to be run using the user PC's operating system. The provided project files on GitHub (see appendix) will allow compilation of a board support package for use with the three tests in parallel.

4.6 Software-configuring the Hardware

The implemented system makes it easy for the software to take away parts of the p -value computations from hardware through shared components. As aforementioned, chi-squared tests, Y and p -value calculations are shared by all tests; calculations which can be conveniently performed in software upon computing the raw histograms in hardware.

The software portion of the system is not just responsible for computations, but also for managing test parameter input from the user, and formatting test feedback output to the user. In particular, the interactions between software and ZedBoard work as below.

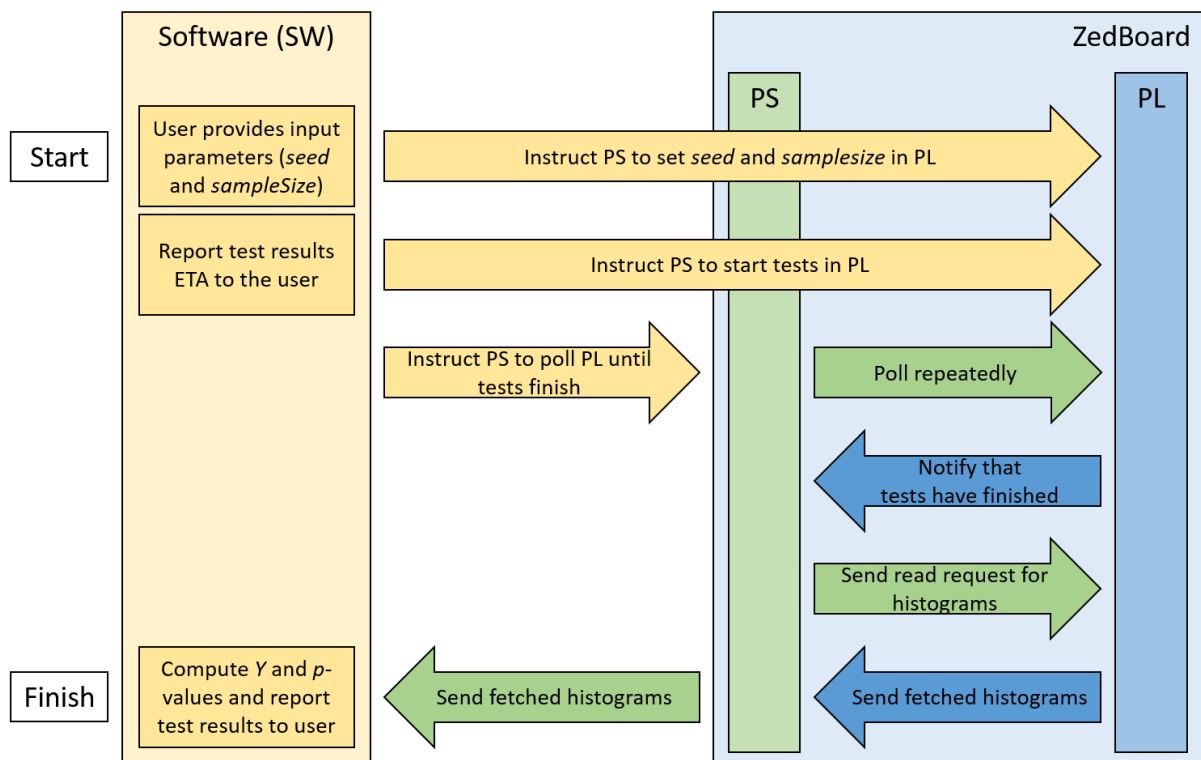


Figure 9: Overview of instructions exchanged between software, PS and PL

Through polling, the PS waits for the PL portion to finish the histogram functions. Upon finishing computations, the PS sends the PL's histogram output to software, where the histograms are used for the Y and p -value calculations. Finally, when computations finish, the results are reported to the user in a format specified below.

4.7 System Interaction with the User

When the user has ported the design to the FPGA and has started a test run, it becomes important to inform the user of important test run information. Given that the test can take up to almost 49 hours for 2^{44} samples, which is the maximum allowed sample size, it was deemed important to let the user know some critical pieces of information before the test, as well as the results themselves after the test run has completed. Given that in this system's I/O, polling was used instead of interrupt handlers operating during run-time, the information communicated to the user can be split into before the test has begun, and to after the test has finished.

4.7.1 Before the Test

Results' estimated time of arrival (ETA): Given that the test runs can become very long, and given that the length of the test can be calculated in advance through multiplying the PL's clock frequency with the requested sample size, it was deemed important to let the user know how long the test suite would run for. Note that this calculation assumes that the user's RNG is optimised to 1 sample per clock cycle generation speed, hence the ETA may deviate from the actual if the user supplies a slower RNG.

Reminder of the user's chosen test parameters: In case the user has erroneously entered a wrong parameter, it would be more helpful if this became clear early on rather than when the user confusedly examines unexpected test results after long test runs.

An example of the communication before the test is outlined in Figure 10 (next page).

4.7.2 After the Test

Performance metrics: The measured test run speed will be reported to the user.

Test results: The test results include raw histogram data and computed Y and p -value data, relevant for test pass/fail decisions by the user.

4.7.3 Output Format

Initially, it was planned to output the data in a `.csv` file. However, due to time constraints, this could not be achieved. As a compromise however, the user can view the test information and test results in the console, where the data has been formatted such that the `printf` functions output in `.csv`-friendly format; for convenience when copy/pasting the console information into spreadsheet software such as Microsoft Excel, all information is automatically split into separate cells, through separating values with commas in the `printf` functions. An example test output, when pasted into Excel 2016, is shown in Figure 11.

```

-----,
Sample Size, 4294967296
Seed, 0
Offset, 0
-----,
Results ETA, 42.95, seconds

```

Figure 10: Communication to user before the test (viewed in Xilinx SDK console)

	A	B	C	D	E	F	G	H	I	J
1	-----	-----								
2	Sample Size	4.29E+09								
3	Seed	0								
4	Offset	0								
5	-----	-----								
6	Results ETA	42.95 seconds								
7	Test completed in	42992865 us (42.95 seconds).						
8	PL clock cycles per sample:	1								
9	-----	-----								
10	Frequency Histogram	524288	524288	524288	524288	524288	524288	524288	524288	(...)
11	Serial 2-Tuple Histogram	0	1048576	0	0	524288	0	0	524288	(...)
12	Serial 3-Tuple Histogram	0	0	349525	0	0	349525	0	0	(...)
13	-----	-----								
14	Test	Y								
15	-----	-----								
16	Frequency Test	0								
17	Serial 2-Tuple Test	35651584								
18	Serial 3-Tuple Test	30944631								
19	-----	-----								
20	Test	p-value								
21	-----	-----								
22	Frequency Test	1								
23	Serial 2-Tuple Test	nan								
24	Serial 3-Tuple Test	nan								
25	-----	-----								

Figure 11: Communication to user after the test (copy/pasted to Microsoft Excel 2016)

4.8 How to compile the test suite from the user's perspective?

The user has to compile the design in three software programmes, as outlined below.




1		2		3	
	Vivado HLS		Vivado IP Integrator		Xilinx SDK
	Design PL		Interconnect PS/PL		Control PS/PL
	User inserts custom RNG algorithm in provided Vivado HLS project files		User replaces IP core in provided project files with updated IP core		User connects FPGA with SDK and programs bitstream on FPGA
	User synthesises updated project files and exports RTL to create an updated IP core for use in Vivado IP Integrator block design		User generates updated block design, synthesises and implements the design and generates bitstream for use in Xilinx SDK		User specifies test suite input parameters, runs the test and analyses the test results

Figure 12: Test suite compilation steps (outlined in detail in Section 9 (User Manual))

5 Low-Level Abstract Design

This section outlines some of the more interesting technical problems which were faced in the facilitation of the implementation of the aforementioned top-level features of the test suite system, and the design choices which were made along the way, and why.

5.1 Implementation

Figure 13 shows a top-level block diagram of the hardware implementation, and its block diagram upon its synthesis in Vivado IP Integrator.

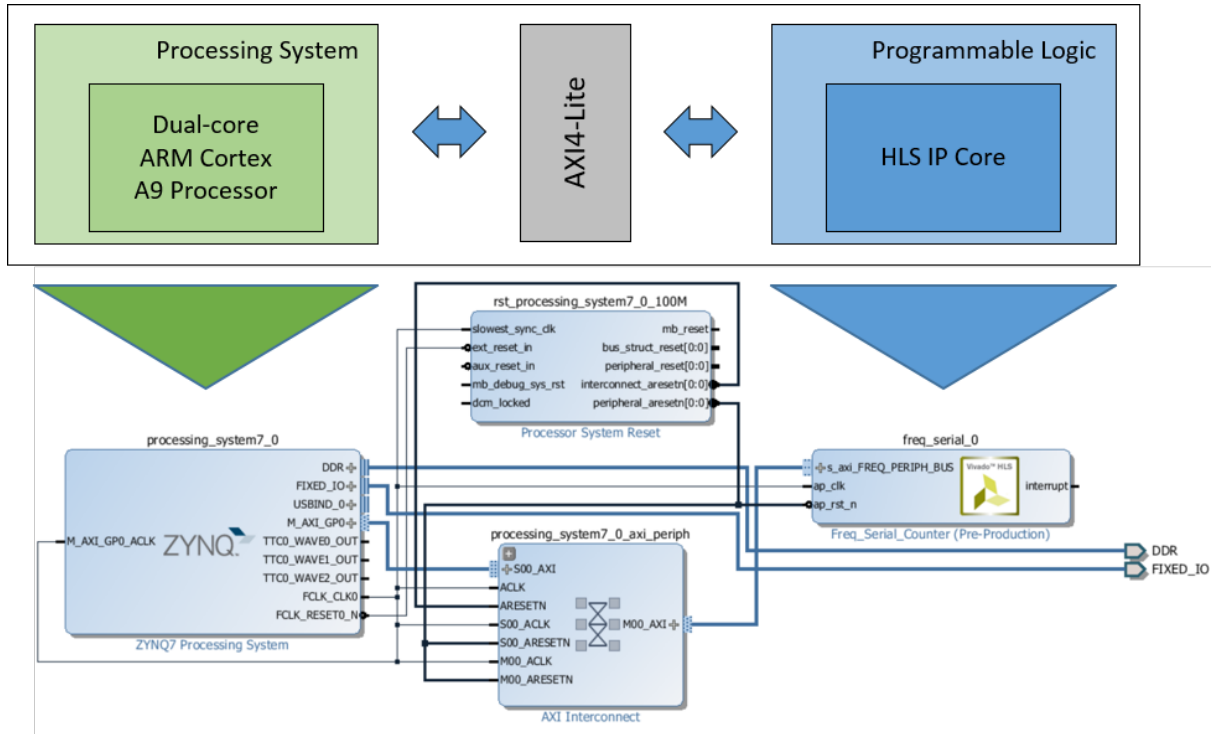


Figure 13: Block Diagram of AXI4-Lite-interfaced system (top) and its representation in Vivado IP Integrator (bottom)

From the left of the Vivado IP Integrator diagram, the IPs are, in clockwise order:

- **Processing System:** This IP represents the ARM Cortex Processor, which can be customised. This block functions as the controller of the design, hence the AXI4-Lite Master general output port `M_AXI_GPO`. Note that the AXI4 Master interface contains a large number of constituent signals, which are combined to provide a higher level of abstraction of the interconnects between AXI4 Master/Slave.
- **AXI Peripheral Reset:** This IP is responsible for driving the reset signals for all the peripherals that are connected to the PS via AXI4.
- **User's RNG / Test Suite:** This IP represents the HLS-generated component, which the user has to export from Vivado HLS. Note the block needs to be controlled

and instructed by the PS, hence the AXI4-Lite Slave input port `s_axi_FREQ_PERIPH_BUS`, which contains a large number of constituent signals.

- **AXI Peripheral Interconnect:** This IP is responsible for handling the complex interconnecting between the PS' AXI4 Master Interface and the HLS-generated block's AXI4 Slave interface.

Note that the `interrupt` output port of `freq_serial_0` is unconnected. As will be discussed in Section 8 (Future Work and Conclusion), this port could have been used to enhance the implemented system, however, due to insufficient implementation time, this could not be realised in this project.

5.2 System Design Choices

5.2.1 Histogram Data Width Concatenation

When testing a first version of the implemented design in Xilinx SDK for sample sizes larger than 2^{32} , surprisingly, the frequency test and serial test accumulators would start to overflow, reporting undefined negative values as their results. It was found that this was due to there being a 32-bit data width limit as set forth by the AXI4-Lite standard for array declarations in Vivado HLS, such that histogram arrays with data types specifying more than 32 bits would not be communicated correctly between PS and PL.

The initial implementation had to be reconfigured to accommodate for this unforeseen design constraint. Initially, the histograms were designed such that they would leave the PL in their raw, unprocessed form; frequency histograms were designed to count up to 2^{44} numbers (since, in the worst case scenario, a bit counts a 1 every sample) and the serial 2-tuple and 3-tuple histograms up to 2^{43} , since number of observations is $\frac{sampleSize}{2}$ and $\frac{sampleSize}{3}$, respectively.

However, in order for the histograms to be communicable with the PS, the results of the PL-generated histograms were concatenated to 32 bits. This has been achieved through right-bitshifting `freqCount` by $44 - 32 = 12$ bits and `serialTwoCount` and `serialThreeCount` by $43 - 32 = 11$ bits.

More specifically, when the PL has finished the tests internally and is about to send off the histograms to the PS, as an intermediary step, the final 44-bit and 43-bit accumulator arrays are first concatenated and cached into 32-bit arrays, and then sent in this 32-bit form to the software, via the PS. In context of the implemented system, this PS/PL/Software interaction will work as shown in Figure 14 (note that the PS' role as the data stream intermediary between PL and software is omitted, but implied in the schematic).

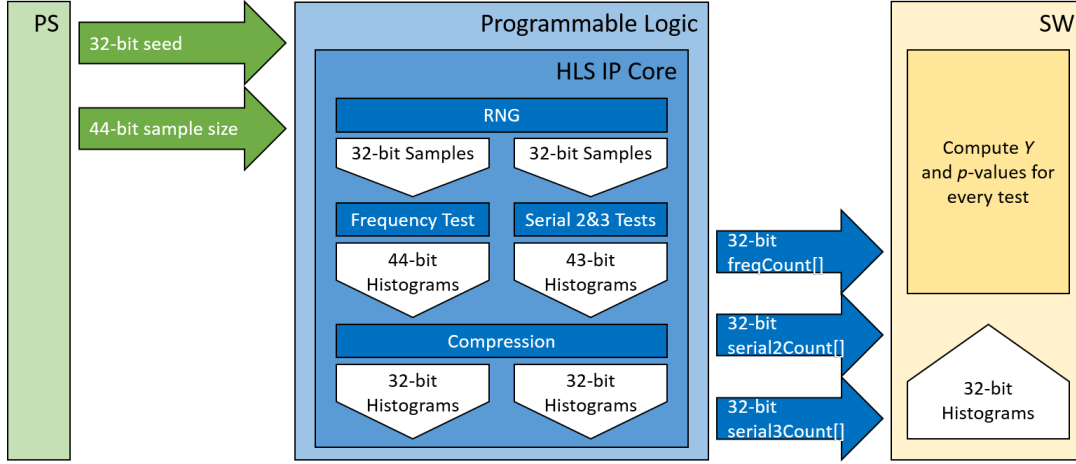


Figure 14: Overview of bit widths of PL input/output streams

As may become clear from this concatenation approach, the testing of small sample sizes of 2^{12} and lower now became infeasible, whilst sample sizes between 2^{12} and 2^{44} , especially towards the lower end would become less accurate, possibly leading to misinterpretations of the reported p -values. However, since the user is assumed to wish to test for trillions of numbers, where observational errors in the order of thousands ($2^{12} = 4096$) would have no significant impact on Y and p -value calculations (and did not significantly impact, as shown later), this was thought to be a sensible design trade-off.

5.2.2 Simplifying the Serial 3-Tuple Test

The serial 3-tuple test, in its initial form, could have helped analyse all 32 bits of every 3-tuple of random numbers. However, when trying to synthesise this test, it was found that this was not implementable on the target FPGA platform, as it used up all available LUTs, even when implemented as the only test. In order to allow for this test to be run in parallel with the frequency test and the serial 2-tuple test, a compromise was made.

Instead of analysing all 32 bits like the other two implemented tests, the final implemented version of the serial 3-tuple test analyses only for 16 bits, either every odd bit, or every even bit. The control of this is facilitated through the user specifying an `offset` variable in Xilinx SDK to 1 for all even (bits 0, 2, ... 30) or 0 for all odd (bits 1, 3, ... 31). This variable is then fed into the PL portion of the ZedBoard to toggle on/off the correct 3-tuple counters for analysis of the custom RNG.

The user can still analyse all 32 bits for their 3-tuple independence through running the test twice with offsets set to 0 and 1 once each, this way still allowing for a convenient speed-up compared to existing test suites.

5.2.3 Exploiting Degrees of Freedom

Both the serial 2-tuple test and the serial 3-tuple test were enhanced in terms of resource footprint, through exploiting the fact that the accumulator values are statistically dependent of one another; the last bucket can be calculated using all previous bucket values, and the expected total number of observations, as shown below for serial 2-tuple tests:

Tuple	00	01	10	11
Frequency	x	y	z	$\frac{sampleSize}{2} - (x + y + z)$

Similarly, for serial 3-tuple tests:

Tuple	000	001	...	101	110	111
Frequency	a	b	...	f	g	$\frac{sampleSize}{3} - (a + b + c + d + e + f + g)$

For the serial 2-tuple test, this optimisation can reduce the number of counters by 32 counters to $128 - 32 = 96$, whereas for the serial 3-tuple test, a reduction of 16 counters to $128 - 16 = 112$ could be achieved. The restoration of the missing tuple values is then performed in software, using the formulae above, allowing for the three parallelised tests to be realisable on the target platform. When viewed as a subsystem of the implementation, the accounting for degree of freedom calculations works as shown below:

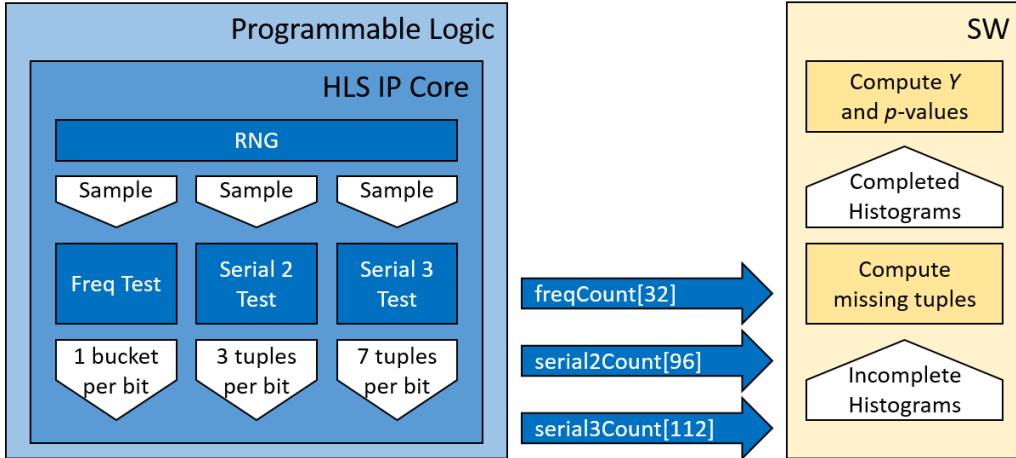


Figure 15: Overview of tuple completion in PL/software. Note the notable reduction in required serial test array elements ($128 - 32$ and $128 - 16$ for 2- and 3-tuples, respectively)

Note that when coupled with the histogram data width concatenation methodology above, there are some inaccuracies in the obtained values for the restored tuple values. However, as will be shown in Section 6 (Simulation and Testing), this did not pose any practical problems for the RNGs tested in the evaluation of whether they had passed or failed the tests. This is partly due to the anticipated and tested large sample sizes, making calculation errors caused by these system enhancements negligible for the purposes of testing trillions of RNG samples.

5.3 Vivado HLS Hardware Specifications and Optimisations

Vivado HLS allows for optimisation commands, which helped meet the goal of achieving one sample per clock cycle throughput. These commands can be directly embedded in C++ code to granularly specify at function- or variable-level how the hardware should behave. In the following section, the optimisations behind the implemented empirical statistical tests will be discussed, and how speed-ups were achieved in Vivado HLS C++.

5.3.1 Specifying Data Streams

For the HLS-generated block (consisting of the user's RNG and the three test parallelised tests) to be able to interact with the PS, the top function needs to include definitions of the intended data streams which will enter/leave the HLS block from/to the PS.

This is achieved in Vivado HLS C++ through declaring data types for the input parameters `seed`, `offset` and `n` (for sample size), and the histogram inside the top level function `freq_serial`'s arguments, as shown below. Note that `ap_uint<x>` is a Vivado HLS data type for arbitrarily-sized unsigned integers of `x` bits.

```
1 void freq_serial(ap_uint<32> freqStream[32], ap_uint<32> serialTwoStream
   [96], ap_uint<32> serialThreeStream[112], ap_uint<32> seed, ap_uint<44>
   n, ap_uint<1> offset) {
2
3   /*...code for the three tests and the user-specified RNG...*/
4 }
```

Vivado HLS automatically detects whether an argument is intended to be an input or output port in hardware, through analysing inside the function whether the arguments are read from or written to.

In order to allow these HLS block data streams to communicate with the PS, as aforementioned in Section 2.8 (PS/PL Interconnect - AXI4-Lite), the AXI4-Lite interface has to be specified. This is achieved through `#pragma` hardware specifications, as below.

```
1 void freq_serial(/*...aforementioned input parameters...*/){
2
3   #pragma HLS INTERFACE s_axilite port=freqStream bundle=FREQ_PERIPH_BUS
4   //repeat pragma command for serialTwoStream, serialThreeStream, seed, n,
   offset
5
6   /*...code for the three tests and the user-specified RNG...*/
7 }
```

The command `bundle=FREQ_PERIPH_BUS` gathers all top function parameters in a single AXI4-Lite connection in Vivado IP Integrator as seen in Figure 13, allowing for simplified interconnecting with the PS, through bundling together all external ports.

5.3.2 Instruction Pipelining

To achieve one sample per clock cycle throughput, each outermost loop within the C++ code, including the main sampling loop, must be fully pipelined. This is achieved through inserting a `#pragma` optimisation directive below each loop declaration, as below.

```

1 //top level function main loop, analysing sampleSize n
2 for (int idx = 0; idx < n; idx++) {
3     #pragma HLS PIPELINE
4     //freqTest() || serialTwoTest() || serialThreeTest() || seedNext()
5 }
```

However, this optimisation directive alone does not guarantee one sample per clock cycle throughput, as the pipelined loop's constituent instructions may be too slow; the pipelined design will take as many clock cycles to process one sample, as the slowest constituent instruction will takes to process one sample. One example of such a slow instruction used heavily in this system are array accumulations, e.g. `freqCount[x] += 1`.

5.3.3 Partitioning Arrays

By default, Vivado HLS synthesises an array declared in Vivado HLS C++ as a BRAM in hardware, which allows for one read or one write per clock cycle of one element of the array. For the frequency test for instance, it was found that when pipelining is enabled, it would still take 64 clock cycles to analyse one 32-bit sample. This was found to be due to a BRAM representation of the array `freqCount[32]` not allowing for more than one data access per clock cycle, to a maximum of one of its addressable elements per clock cycle, as aforementioned in Section 2.9 (ZedBoard FPGA Resources). In its dual-port version, BRAMs can process a 32-bit sample in 32 clock cycles, but this is too slow.

In order to overcome the BRAM's throughput limitations, and in order to achieve the goal of accelerating the tests to one sample per clock cycle, the `freqCount[32]`, `serialTwoCount[96]` and `serialThreeCount[112]`, arrays are fully partitioned when they are declared in C++, as below.

```

1 ap_uint<44> freqCount[32];
2 #pragma HLS ARRAYPARTITION variable=freqCount complete dim=1
3 ap_uint<43> serialTwoCount[96];
4 #pragma HLS ARRAYPARTITION variable=serialTwoCount complete dim=1
5 ap_uint<43> serialThreeCount[112];
6 #pragma HLS ARRAYPARTITION variable=serialThreeCount complete dim=1
```

These optimisation directives command Vivado HLS to synthesise the histogram counters using FFs and LUTs instead of BRAMs, allowing for a read/write for every counter per cycle to achieve 1 sample per clock cycle throughput.

It should be noted that the resulting increase in LUTs used in the design goes up by more than twice compared to an equivalent design with non-partitioned arrays. However, given this project's primary goal of achieving throughput rates of one sample per clock cycle for multiple tests, in order to maximise the efficiency in this configuration of tests, the complete partitioning approach was taken.

5.3.4 Power-On Initialisation of Arrays to Zero

Chronologically, this was the final implementable optimisation discovered and implemented in the system. Initially, when the three tests were coded such that they would be explicitly initialised to zero inside the HLS block prior to actually starting to run the tests, the *Synthesis Report* function in Vivado HLS would report that the available LUTs had been overshoot by 106%.

Since this was a one-off computation, it was attempted to initialise the histograms through some data stream signal from software. However, since the aforementioned concatenation design choice had already been made, which converts the 44-bit counters to 32-bit counters, such that the 44-bit counters become invisible to outside the PL, there was no longer any external access possible to the HLS block's intrinsic histogram counters.

Through research, it was found that it is instead possible to implicitly initialise all arrays to zero through declaring them as `static`, which would set them to zero when powering on the FPGA. This way, the design could be fitted on the FPGA, with only 36% of available LUTs used according to Vivado HLS, with a 32-bit counter RNG connected to the three-test suite. This frees up significant space for more tests to be run in parallel, and/or for efficient LUT-heavy RNGs to be implemented by the user.

However, it is important to note that the user is now required to power off/on and reprogram the FPGA prior to consecutive test runs, which may be considered inconvenient. This slight trade-off of user-experience was reasoned to be appropriate, given that reducing test run-time and required resources were main goals of this project.

Given the unexpected gains in available FPGA resources, it was attempted to implement a correctly functioning version of the full serial 3-tuple test. Unfortunately, there was insufficient time to debug this test in time for the system testing to start, hence only the 16-bits-analysing serial 3-tuple test is considered in this report discussion, alongside the full frequency test and the full serial 2-tuple test.

6 Simulation and Testing

The testing of the system for the correctness of its intended functionalities was mainly done through analysing the output streams of the HLS block; first, the *Run C Simulation* functionality in Vivado HLS has been used to design a test bench for testing the Vivado HLS C++ code prior to exporting the IP block as RTL. Second, Xilinx SDK has been used extensively to debug the system to the extent that every function outlined in the previously introduced top-level design system in Figure 4 is implemented and, to the best of our knowledge, working as intended.

More specifically, the system testing methodology used can be roughly be split into:

- **Functional Testing:** Are the individual components working as intended?
- **Integration Testing:** Is the interaction between components working as intended?

Note that whilst there could have been more ways of testing the system, such as the *Vivado Simulator* tool, the *Timing Analysis* function or even Xilinx’s proprietary debugging IPs, all of which are part of the Vivado IP Integrator, there was too little time to become familiarised with all these functionalities. Thus, the focus in this project was instead on debugging and testing the design in C/C++ style code through `printf` functions in Vivado HLS and Xilinx SDK.

In particular, the analysis focused on whether the system output streams were computed and communicated as intended. This approach was found to be satisfactory for ensuring a correct design. The more granular testing methods mentioned were considered to be too microscopic for the tight time constraints of this project.

6.1 Functional Testing using Vivado HLS

Note that due to the *Run C Simulation* function being software-based, higher test suite sample sizes ($> 2^{36}$) tend to become inadequate for testing, as the run-time becomes too significant, causing the software to freeze. Instead, the C simulations used in this project mainly focused on feeding in smaller example input, to see if the output histograms are computed, concatenated and formatted correctly.

This is achieved through coding a C testbench (available on GitHub, see appendix) which calls the IP Core top level function using example parameters for `seed`, `sampleSize` and `offset` (toggling to 0 or 1 for even or odd samples for the serial 3-tuple test), to then check the histogram outputs for whether they are computed as intended. Note that *p*-value computations will be done in Xilinx SDK and are hence not the focus of this

Vivado HLS test. Only histograms are checked at this stage.

The four RNGs under consideration in this project have been run using the following parameter values:

Input Parameter	sampleSize	seed	offset
Value	$2^{20} = 1048576$	1	1

Below are the expected frequencies of each category occurring for all three tests, accounting for 12-bit (frequency test) or 11-bit (serial tests) concatenation.

Test	Frequency	Serial 2	Serial 3
Expected Frequency	$\frac{sampleSize}{2*2^{12}} = 128$	$\frac{sampleSize}{4*2*2^{11}} = 64$	$\frac{sampleSize}{8*3*2^{11}} = 21.33$

On the next page, the results are graphically outlined in Figure 16. The further away an RNG's observed frequency is from the expected frequency (128, 64 and 21.33, respectively for each test), the worse is the RNG's performance according to the test's idea of what constitutes randomness. When the results were analysed, it was found that each RNG was behaving as intended for the selected input parameters.

The order in which they were listed in Section 2.6 (RNG Algorithms under Consideration) was indicative of their ascending order of quality, with RANDU having volatile distributions for all three tests, and Xorshift having nearly perfect uniform distribution for all three tests. Note that the Simple Counter would have shown notably better distributions if its period 2^{32} had been reached. This highlights the importance of varying the sample size parameter for consecutive test runs, especially when initial test result screenings reveal borderline pass/fail p -values close to 0 or 1, as explained in Section 2.3.2 (Interpreting Empirical Test Results). To quote Knuth [9] p. 46, this is due to larger n tending to smooth out locally nonrandom behaviour, when blocks of numbers with a strong bias followed by blocks of numbers of the opposite bias.

Note that due to the aforementioned degrees of freedom design choice made, the last bit-wise counter for each tuple-counter (every 4^{th} counter for 2-tuples, every 8^{th} counter for 3-tuples) is sometimes slightly deviating from the expected frequency. This is, however, far less noticeable for trillions of samples.

The results from the tests are as would be expected; the distributions roughly match the theoretical distribution of these results for the tested RNGs. Worse RNGs deviate further from the expected value, whilst the better RNGs' distributions closely resemble the ideal. Hence, there is a higher level of confidence that the HLS IP block is working as intended, which has been confirmed further using Xilinx SDK.

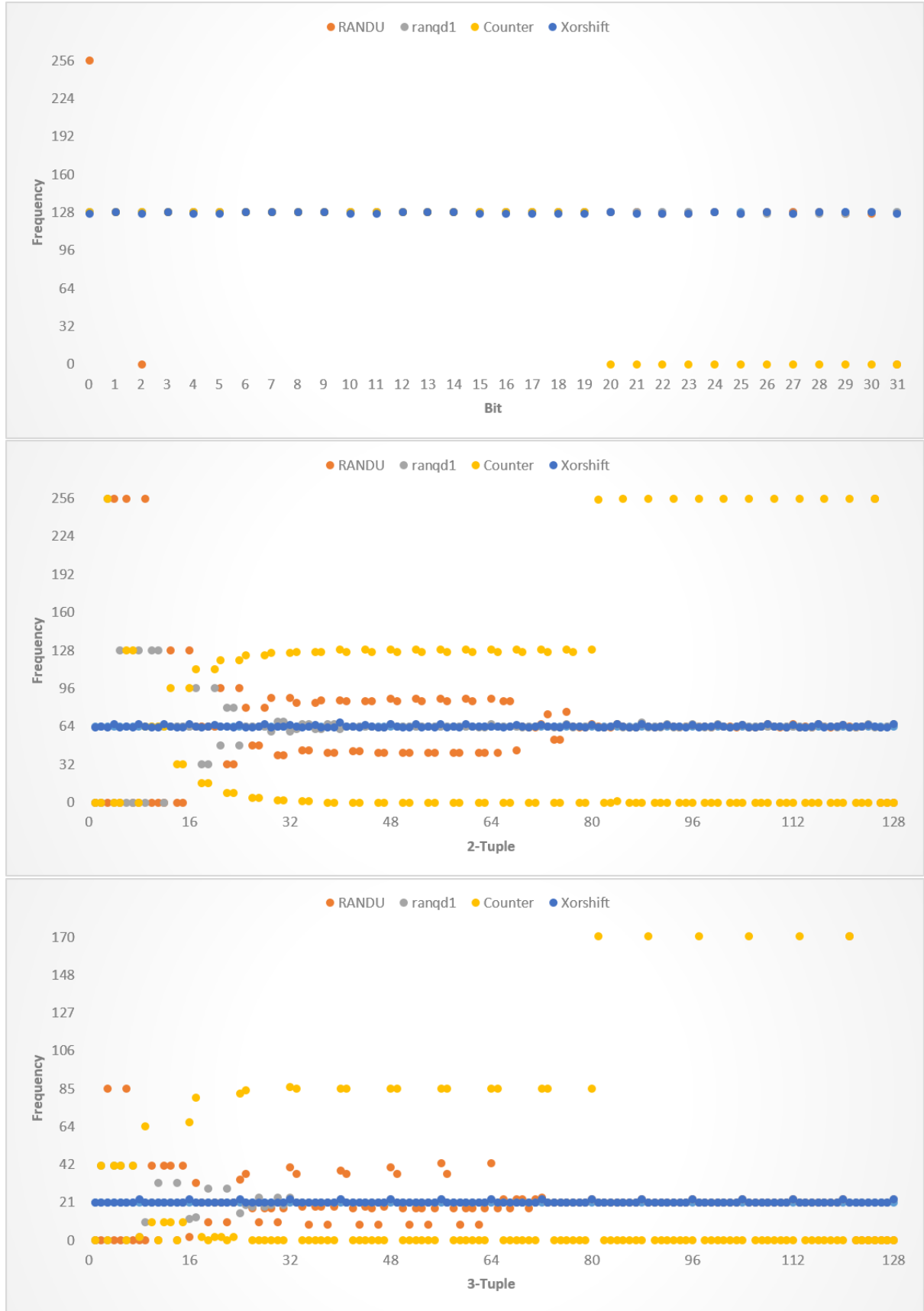


Figure 16: Frequency, Serial 2-Tuple and Serial 3-Tuple distributions for the four RNGs

6.2 Integration Testing using Xilinx SDK

Debugging the hardware/software interface in Xilinx SDK turned out to be surprisingly difficult, perhaps one of the most difficult parts of the project, because the various subsystems of the proposed system were of a tightly coupled nature, and because various Xilinx handbooks, guides, forum posts and C/C++ library documentation and standards had to be referred to and fully understood in order to fully grasp the system, as well as the generated and supplied functions as part of the generated board support package (BSP).

As aforementioned, the chosen way of interacting with the ZedBoard was a bare-metal approach. Debugging-procedure-wise, various `printf` functions were distributed across the code to read current values from the ZedBoard, in order to facilitate a stronger understanding of whether a bug was caused by hardware, software, or in both.

6.2.1 Debugging Information

In order to help understand the synthesised and implemented software/hardware interface for system debugging, various pieces of information have been read from the ZedBoard in order to offer insights into the initially erroneously implemented architecture and its inner workings, to help spot the errors. The following information in particular helped identify and resolve the aforementioned 32-bit histogram concatenation problem:

- **Array TotalBytes:** Stores the number of bytes allocated to each array. Each word is 4 bytes, hence at least $4 \times 32 = 128$ bytes should be available for frequency counters, $4 \times 96 = 384$ for serial 2 counters and $4 \times 112 = 448$ for serial 3 counters.
- **Array BitWidth:** Stores the maximum length of each element of the array. Was thought to be 44 for `freqCount` and 43 for both `serialCount` arrays, but turned out that it can be, and thus should be, a maximum of 32 bits for every counter.
- **Array Depth:** Stores the number of elements in each array. Should be 32 for frequency counters, 96 for serial 2-tuple counters and 112 for serial 3-tuple counters

Xilinx SDK `printf` functions were used to output the console excerpt below, showing that the histogram counters were correctly sized in the final system implementation.

```
1 Freq Total Bytes: 128
2 Freq Bit Width: 32
3 Freq Depth: 32
4 Serial 2 Total Bytes: 512
5 Serial 2 Bit Width: 32
6 Serial 2 Depth: 96
7 Serial 3 Total Bytes: 512
8 Serial 3 Bit Width: 32
9 Serial 3 Depth: 112
```

6.2.2 Histogram Correctness Testing

In order to check whether the hardware/software interfacing works as intended, a similar approach to the aforementioned functional testing methodology in Vivado HLS has been adopted. The four RNGs have each been compiled once with the test suite, then tested for 2^{38} samples for identical parameters (seed = 1, offset = 1, sample size = 10^{11}). The four test run result outputs can be viewed in the GitHub repository (see appendix).

The observed distributions have then been compared to their expected distributions in a similar fashion to the previously shown Figure 16 distribution graphs. The analysis again indicated a correct implementation of hardware/software interfacing, thus providing stronger evidence that the design is functioning correctly as intended.

6.2.3 Maximum Sample Size Testing

In order to ensure that the design works correctly for the intended trillions of RNG samples, the test suite has been run with periods of 2^{40} , 2^{42} and finally for the target maximum period of 2^{44} with the simple counter RNG (since this RNG was simple to implement and achieved the required matching generation speed of one sample per clock cycle). The results of these test runs (available on GitHub) were in accordance with a correctly functioning suite for the maximum sample size, indicating that the system works correctly for trillions and tens of trillions of samples.

6.3 Functional Testing using Xilinx SDK

Finally, the correctness of the computations performed in software had to be verified. As aforementioned, the software computations include the completion of histograms for the degrees of freedom optimisation made, and the Y and p -value calculations, accounting for the 32-bit concatenation design decision made. All of these were tested for using various `printf` functions distributed throughout the software-part of the Xilinx SDK code, to check whether the computations were working as intended.

For the histogram completion computations, there was already code available from the aforementioned Vivado HLS testbench, which was already tested for correctness. The verified Vivado HLS testbench was copied almost 1:1 into Xilinx SDK. For the remaining computations, the Y computations were simple accumulations, and the p -value function used was a public domain available algorithm, such that this part of the testing was found to be relatively straightforward.

It is worth noting that in this stage, it was found that this test suite lets the p -values converge extremely fast either to nan (not a number) for failure or 1 for success, almost in a binary fashion, with no value being reported between these two. As aforementioned

in Subsection 2.3.2 (Interpreting Empirical Test Results), it was expected that for good RNGs, the p -value computations would approximate a random variable $U(0, 1)$ under null hypothesis H_0 , i.e. values between 0 and 1 should happen with about equal probability.

The reason why this was not case in this implementation is partly because the number of buckets is vastly smaller in this project than for most tests in TestU01, such that test failure/non-failure is extremely evident; in this project, the test suite computes bit-wise distributions and expects them to be uniform, such that even moderate deviations between expected and observed distribution found in one of the 32 frequency counters or in one of the 128 serial counters influences the p -value computations significantly, whilst existing test suites check for distributions using counters across the entire decimal range of numbers. Since the Y -value computations in TestU01 and in this project are not mathematically equivalent, due to the binary vs decimal difference in testing, a p -value between 0 and 1 for consecutive runs of TestU01's tests is likely to show as a 1 in this project's implemented test suite.

However, for the project goal of helping the user analyse the properties of distribution and independence through the analysis of trillions of numbers at high throughput rates, as the quickly converging p -value still suffices for this purpose. Furthermore, should the p -value computations themselves be considered to be too high-level for understanding the low-level workings of underlying RNG algorithm, the user will find the auxiliary histogram reporting functionality as part of this test suite more informative for addressing the custom RNG's randomness properties, as being able to see bit-level weaknesses can be helpful. In fact, in this project, the test suite was helpful for reverse-engineering RNGs' algorithms to strengthen our understanding of their statistical profiles, helping to implement the tested RNGs correctly in the first place.

In summary, the test suite's simulation and testing has established that the test suite is functioning correctly for up to 2^{44} to a high degree of confidence, whilst also showing that the implemented test suite can be useful for detecting non-randomness patterns through extremely quickly converging p -values, whilst allowing for bit-level analysis of the RNG's randomness properties.

7 Evaluation

7.1 Hardware Resources

The test suite HLS IP block, when synthesised in Vivado HLS with the Xorshift RNG, reports the following synthesis report summary, outlining estimated resource usage:

Resource	Utilisation	Available	Utilisation %
BRAM_18K	2	280	2%
DSP48E	0	220	0%
FF	11046	106400	10%
LUT	19596	53200	36%

However, when the test suite is synthesised and actually implemented with the Xorshift RNG in Vivado IP Integrator, the following project summary is reported:

Resource	Utilisation	Available	Utilisation %
BRAM	3	140	2.14%
FF	11455	106400	10.77%
LUTRAM	58	17400	0.39%
LUT	3670	53200	6.90%

This shows a stark deviation between the expected and the realised LUT usage in particular, which was a leading consideration when evaluating how the design should be outlined; first, the target sample throughput rates were reached, but then the focus was on reducing hardware footprint of already implemented tests, which, according to the tables, was not necessary to the degree achieved in this project. More time could have been dedicated to the implementation and correctness testing of additional tests instead.

The high deviation between expected vs realised hardware utilisation figures was a very surprising finding, given that the resources required in the final implementation was checked only towards the end of the project time line, as it was assumed that Vivado HLS estimates would not deviate several orders of magnitude from the realised implementation's hardware resource requirements. In retrospect, this should have been verified earlier on in the project; the tests disregarded due to their estimated heavy LUT-usage may have been able to be implemented after all.

The reasons behind this deviation between expected and actual hardware resource utilisation are unknown. Perhaps Vivado IP Integrator has become more efficient recently than Vivado HLS estimations have been able to account in its latest version (Vivado Design Suite 2015.4 was used towards the beginning of this project, 2016.2 was used towards the end).

7.2 Relative Performance

The goal of this project was to outperform existing test suites in terms of throughput and maximum sample size. Both of these were met, as benchmarked and discussed below.

The performance of the hardware-accelerated test suite’s constituent tests has been compared to the TestU01’s *Alphabit* battery of tests, which consists of 9 tests. The motivation behind this choice, as aforementioned in Section 2.7 (Related Work), were a) Alphabit is arguably the closest and most modern contender for targeting users wishing to test their hardware RNGs and b) the similarity of two of the test suite’s constituent tests to the implemented tests in this project.

Alphabit’s 9 tests are actually four types of tests with different permutations of parameters used. Two of these four types, the `MultinomialBitsOver` and `HammingCorr` tests are of interest for benchmarking. The `MultinomialBitsOver` test analyses overlapping 16-bit numbers for distribution correctness similar to the frequency test, whilst the `HammingCorr` test applies a correlation test on the Hamming weights of 32-bit numbers, similar to the serial test.

Note that it was acknowledged in this project that the two TestU01 tests’ documentation [2] (p.116 and p.149) shows that the randomness property investigated is mathematically fundamentally different from the implemented tests. However, for run-time benchmarking purposes, it was reasoned that they provide a sufficient proxy for the order of magnitude differences between the implemented system and a generic CPU-run modern test suite. Hence, these two tests’s run-time were selected as performance benchmarks.

For run-time benchmarking, a Windows 7 64-bit PC with Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz has been used to run Alphabit. Alphabit has been run using RANDU as its input RNG. Through collecting data of multiple test runs, the run-time has been noted for bit sample sizes ranging from 2^{32} to 2^{40} in steps of $\times 2$, equivalent to 32-bit integer sample sizes ranging from 2^{27} to 2^{35} ($32 = 2^5$). This averaging was required in order to determine mean throughput rate values for benchmarking, since the project implemented system’s throughput rate is nearly constant for any sample input.

Since RANDU was used, which in its unoptimised form takes 2 clock cycles to produce one sample in Vivado HLS as demonstrated in example test runs on GitHub, the test suite’s sample throughput rate has been halved, for benchmarking purposes, to 50MHz. Furthermore, from Alphabit’s side, the lower throughput rate of the two tests has been used as a benchmark, given that there is concurrent execution potential of the suite in the future. The benchmarked run-time values for the two selected tests and of the project test suite are shown below.

Test or Test Suite	Throughput (32-bit samples per second)
MultinomialBitsOver	8831811
HammingCorr	39759409
Alphabit	8831811
Project Test Suite	50000000

These metrics would indicate a 5.66 times speed increase when using the implemented test suite. However, this can also be put in a different light. As Alphabit shows rather volatile average throughput levels for its constituent tests, the benchmarked run-time values and their resulting throughput computations are shown below for each of the nine constituent tests.

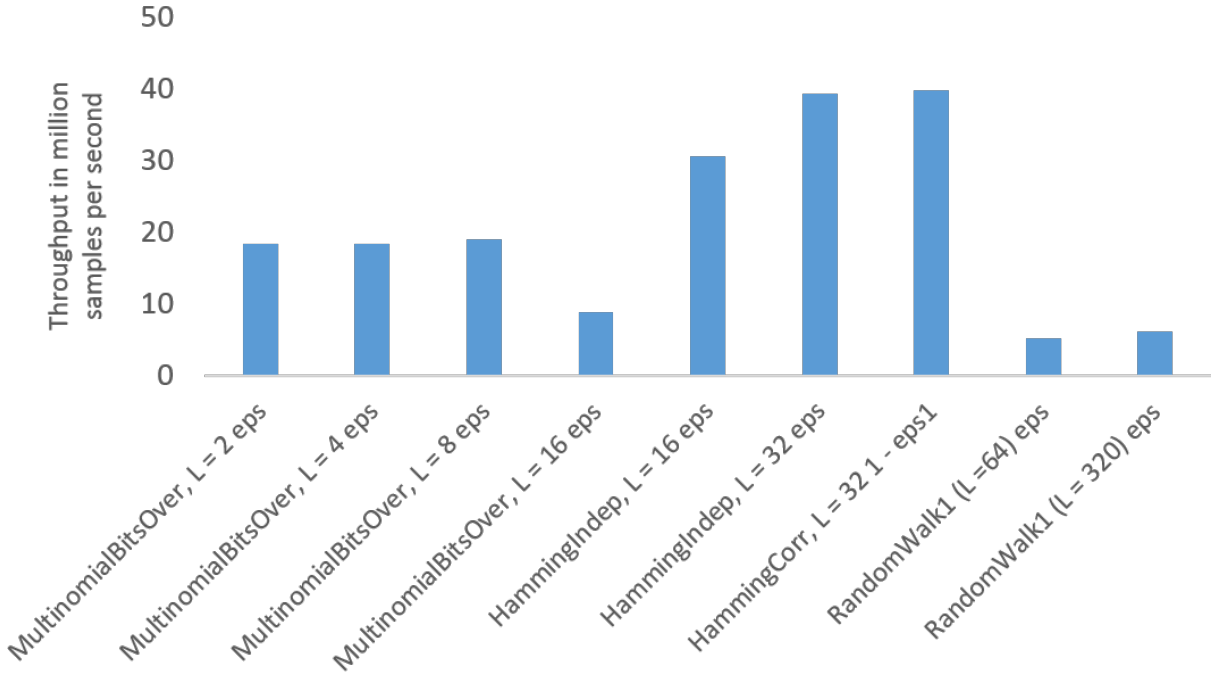


Figure 17: Measured 32-bit sample throughput rates of Alphabit's 9 constituent tests

Assuming parallel execution of all nine tests of Alphabit at their mean throughput rate of 20633656 32-bit samples per second (the underlying spreadsheet calculations are available on GitHub), this would indicate that the test suite is still 2.42 times faster.

As shown in Section 7.1 (Hardware Resources), given that only 6% of available LUTs were used for three empirical tests, if just a few more parallelised tests were implementable on the FPGA optimised to one sample per clock cycle, it becomes evident that the FPGA implementation would become exponentially more efficient than the test suite due to parallel execution on the FPGA vs serial execution in software.

7.3 Comparative Analysis

Looking beyond the achieved goals of analysing a) more samples b) much faster than TestU01's Alphabit battery of tests, some disadvantages become clear. For instance, the achieved implementation allows for fewer and less complicated randomness metrics to be investigated in one test run. In fact, TestU01 provides hundreds of more complicated, usually parameterisable tests, which can be pick & mixed such that different test combinations can be selected for a variety of randomness metrics in one test run. Hence, hardware RNG developers may find it useful to continue performing more complicated parts of RNG analysis in software, whilst moving longer, simpler sample size testing into hardware using this project implementation.

In turn, similarly to hardware RNG developers currently testing their RNGs in software, this project's implemented system may entice software developers to test their RNGs in hardware; given that the RNG implementation is written in Vivado HLS C++, some might consider quickly writing a C++ version of their RNG algorithm and testing over longer periods using this system a viable option to establish a baseline performance prior to moving on to more advanced testing of their RNG in question, as RNG analysis is faster on the implemented system than is currently feasible with existing test suites, for the randomness properties investigated.

As addressed in Section 6.3 (Functional Testing using Xilinx SDK), for chi-squared tests, the implemented test suite reports input parameters, Y and p -value computations and the raw histogram data. In contrast, for chi-squared tests in TestU01, only the input parameters and the computed p -values are reported without the underlying histograms; this is likely due to it being infeasible to output all histograms in TestU01, as they would be very large in terms of number of buckets due to their decimal analysis methodology.

This is unlike the binary (bit-wise) analysis in the implemented test suite leading to very few buckets (32 and 128 for frequency and serial tests, respectively), as discussed in Section 6.3 (Functional Testing using Xilinx SDK), TestU01 is simply not being designed to test hardware RNGs where bit-wise comparisons can be analysed efficiently and can be more interesting to the user. This marks a notable difference to TestU01, differentiating this project test suite positively from existing test suites for bit-level empirical statistical analysis of custom RNGs.

However, it should be noted that for TestU1, given that its test suite is currently designed to be serially executed, there is significant opportunity in the future for parallel execution in software, which could also allow developers and researchers to increase the sample size TestU01 can test for, to match the emergence of newer, faster, longer period RNGs with better statistical profiles.

8 Future Work and Conclusion

In conclusion, from carrying out this project, the maximum throughput rate and maximum sample size restrictions of existing popular test suites have been successfully lifted for some empirical statistical tests checking for distribution and independence, as was planned. It was found that testing FPGA RNGs directly on the FPGA is both a run-time- and resource-efficient approach, even when specified in Vivado HLS.

Given that the test suite can be viewed as a form of MC simulation, and given that previous research outlined in Section 2.1 (Project Motivation) has found FPGAs to be efficient for implementing MC simulations, this project’s efficient implementation is in accordance with these previous findings. The discussed hardware optimisations open up more possibilities in the future for porting more complicated tests to the FPGA platform. With every additional test run in parallel with these existing FPGA-accelerated tests, the time it would take to test them serially with existing test suites can be decreased exponentially. Furthermore, when refined in more granular hardware description languages such as Verilog/VHDL, the design could be increased in efficiency even further.

It is very unfortunate that the most pivotal optimisations and resolvings of misconceptions happened towards the end of the project timeline, leaving little time for action. Thus, ironically, perhaps some of the more interesting takeaways from this project for future work may be the many byproducts of this final implementation which had to be discarded early on due to misleading estimations in Vivado HLS. Hence, one major takeaway from this project is that estimations are estimations, and that being a hopeless optimist can sometimes be very helpful for developers using HLS because the reality is that the majority of LUT estimations in HLS don’t apply for one reason or another.

In terms of future work, when using the same target ZedBoard hardware platform, more enhancements could be made, such as, but not limited to, a) overclocking the ZedBoard [17], b) increasing maximum allowed sample size through increasing the histogram sizes in the PL (prior to AXI4-related 32-bit concatenation), c) programming an interrupt handler to report preliminary results while the test continues to run and d) make modifications of the software code to allow for testing of non-uniform RNGs.

Perhaps most interestingly however for future work is the PS’ underlying dual-core processor which could enable multithreaded execution of tests; the resource-light HLS block could be put into the design twice with the RNG seed value pre-computed in software to be forwarded by 2^{44} samples, so that one HLS block analyses from the user-specified seed value onwards, whilst the other one analyses from the 2^{44th} sample onwards, such that up to 2×2^{44} samples can be analysed, effectively at twice the clock speed.

9 User Guide

1. Install the Vivado Design Suite 2016.2 and the Xilinx Software Development Kit
2. Download the GitHub repository. The link is provided in the appendix.
3. Open Vivado HLS 2016.2
 - (a) Create a new project, specifying the ZedBoard as the target platform and setting time constraints to 10ns
 - (b) Include the contents of `FYP_Vivado_HLS_Files` as source files to the project
 - (c) Open the `simpleCounter.cpp` file. Specify a custom RNG inside the `ap_uint<32> seedNext` function. Do not edit any other part of the `.cpp` file.
 - (d) In the taskbar, click the *C Synthesis* button
 - (e) If synthesis is successful, view in synthesis report the latency and ensure that the design has an iteration latency of 1
 - (f) In the taskbar, click the *Run C Simulation* button and verify that the preliminary tests are in accordance with your expectations. Make adjustments to RNG algorithm if necessary.
 - (g) When satisfied, in the taskbar, click the *Export RTL* button. Upon success, take note of the project directory `simpleCounter/solution1/impl/ip` and close Vivado HLS.
4. Open Vivado 2016.2. This is also known as the Vivado IP Integrator.
 - (a) Click *Open Project*, go to `FYP_Vivado_IP_Integrator_Files` and open the `FYP.vvg1` project file. This should open the project.
 - (b) In the Flow Navigator on the left side of the GUI, press open block design. a block design similar to Figure 13 should appear.
 - (c) In the Block Design Diagram on the right side of the GUI, press the icon for *IP Settings* (fourth icon from the bottom), go to *Repository Manager* and specify the Vivado HLS-generated `ip` folder directory. Vivado should detect one IP.
 - (d) Remove the existing `freq_serial_0` block by clicking on it and pressing Delete.
 - (e) In the Block Design Diagram, press the icon for *Add IP* (icon with a plus symbol) and search for the string `hls` and press enter. A new IP block `freq_serial_0` should appear.
 - (f) In the Block Diagram, click the new *Run Connection Automation* hyperlink.
 - (g) Click on the Block Diagram and press CTRL+S. At the bottom of the Flow Navigator, press *Generate Bitstream*. This process can take a few minutes.

- (h) Upon finishing, click **File/Export/Export Hardware** and when requested in a pop-up, toggle on *Include bitstream*. If prompted, say yes to overwriting existing files in directory.
5. Go to **File/Launch SDK**. This launches Xilinx SDK to launch the test suite.
- (a) In the Project Explorer, open **testbench/helloworld.c**. At the top of the file, specify the desired input parameters.
 - (b) Power on the FPGA and connect two Micro-USB cables to the J14 and J17 sockets of the ZedBoard.
 - (c) Go to **Xilinx Tools/Program FPGA** and press **Program**.
 - (d) Go to **Run/Run Configurations**, go to the relevant **testbench.elf**, go to the **STDIO Connection** tab, toggle on the setting and specify the correct port and set the BAUD rate to 115200.
 - (e) In the Project Explorer, right-click testbench, and go to **Run As/4 Launch on Hardware (GDB)**
 - (f) The test should begin. Check that your input parameters are correct.
 - (g) When tests finish, click on the console and copy/paste into a spreadsheet software of your choice.

10 References

- [1] Pierre L’Ecuyer, Richard Simard. *TestU01: A C Library for Empirical Testing of Random Number Generators*. ACM Transactions on Mathematical Software, Vol. 33, No. 4, Article 22, Publication date: August 2007.
- [2] Pierre L’Ecuyer, Richard Simard. *TestU01 A Software Library in ANSI C for Empirical Testing of Random Number Generators User’s guide, detailed version*.
- [3] G Marsaglia. *DIEHARD: a battery of tests of randomness (1996)*.
<http://stat.fsu.edu/geo/diehard.html>
- [4] De Schryver, Christian, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuik, and Ralf Korn. *An energy efficient FPGA accelerator for Monte Carlo option pricing with the Heston model*. Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, pp. 468-474. IEEE, 2011.
- [5] Phillip J. Kinsman, Nicola Nicolici. *NoC-Based FPGA Acceleration for Monte Carlo Simulations with Applications to SPECT Imaging*. Computers, IEEE Transactions on 62, no. 3 (2013): 524-535.
- [6] David B. Thomas, Lee Howes, Wayne Luk. *A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation* ACM/SIGDA international symposium on FPGAs, pp. 63-72. ACM, 2009.
- [7] David B. Thomas, Wayne Luk *FPGA-Optimised Uniform Random Number Generators using LUTs and Shift Registers*. 2010 International Conference on Field Programmable Logic and Applications. IEEE.
- [8] Jansson, Birger. *Random number generators (1966)*.
- [9] Knuth, Donald E. *Seminumerical algorithms*. (2007).
- [10] David B. Thomas, Wayne Luk. *High Quality Uniform Random Number Generation Using LUT Optimised State-transition Matrices*. Journal of VLSI Signal Processing 47, 7792, 2007.
- [11] Alin Suciu, Radu Alexandru Toma, Kinga Marton. *Parallel Implementation of the TestU01 Statistical Test Suite* Intelligent Computer Communication and Processing (ICCP), 2012 IEEE International Conference on (pp. 317-322). IEEE.
- [12] Jacob F. W. *How to Calculate the Chi-Squared P-Value (2012)*
codeproject.com/Articles/432194/How-to-Calculate-the-Chi-Squared-P-Value
- [13] Xilinx. *7 Series FPGAs Memory Resources - User Guide* UG473 v1.11 Nov 12, 2014

- [14] Xilinx. *7 Series DSP48E1 Slice - User Guide* UG479 (v1.8) November 10, 2014.
- [15] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery *Numerical Recipes in C. The Art of Scientific Computing, 2nd Edition* (p.284), 1992.
- [16] <http://stackoverflow.com/questions/1640258>
- [17] <https://forums.xilinx.com/t5/7-Series-FPGAs/How-to-get-135-MHz-on-Zedboard/td-p/697376>
- [18] Abdel-Rehim, Wael MF, Ismail A. Ismail, and Ehab Morsy. *Testing randomness: Implementing poker approaches with hands of four numbers*. International Journal of Computer Science Issues 9, no. 4 (2012).

A Appendix

A.1 GitHub Repository

The GitHub repository below has been used to store the relevant files to compile the project, and to store the test outputs outlined throughout the report. When copy/pasting the link below, special care should be taken, as some operating systems will not allow correct copying of the `_` underscore character.

https://github.com/sp4713/fyp-fpga-random-number/tree/master/Appendix_code

The Vivado HLS tests have been run through setting the input parameters as follows:

- `ap_uint<32> test_seed = 1;` (as RANDU requires an odd seed value)
- `ap_uint<44> test_sample_size = 1048576;` ($= 2^{20}$)
- `ap_uint<1> test_offset = 1;` (serial 3-tuple test will analyse odd bits (1,3,...,31))

A.2 Disregarded Tests

Gap Test

This test checks whether the observed distribution of gaps is distributed as expected. A gap is defined as the distance between two 1s, e.g. the bitstream 10001 implies a gap of size 3 between the two successive 1s. Gaps may have the values $0, 1, 2, \dots, f, \dots$ and each gap of size s will be counted in a classification category, where the classification category f counts all gaps of size larger or equal to f . Hence, $0 \leq s \leq f$ which implies that $k = f + 1$. Denote the probability that the gap equals s by p_s . For base-2 numbers, p_s is given by: $p_s = \left(\frac{1}{2}\right)^s \times \frac{1}{2}$ where $(r = 0, 1, \dots, n - 1)$.

Using k and p_s , Y can be computed. Using $v = f - 1$, the p -value can be computed.

Count the 1s Test

This test counts the number of 1s in the 32 bits of each generated number. This count figure (from 0 to 32 inclusive) is then converted into a "letter", and 5 successive letters are converted into a "word". The goal is to evaluate whether the deviation between observed and expected distribution of the words is likely to have occurred by chance.

Given that words consist of 5 consecutive letters and given that there are 33 different letters, $k = 33^5 = 39135393$. Since for uniform RNGs, every bit has a constant 50% probability of a 1 occurring, the probability of each letter α occurring is given by: $p_\alpha = \frac{\binom{32}{\alpha}}{2^{32}}$ for $0 \leq \alpha \leq 32$. Then, the probability of each word s occurring is given by $p_s = p_{\alpha1} \times p_{\alpha2} \times p_{\alpha3} \times p_{\alpha4} \times p_{\alpha5}$ for $0 \leq s < 33^5$. Again, $v = k - 1 = 33^5 - 1$. Using k

and p_s , Y can be evaluated. The p -value is obtained using Y and v .

Poker Test

The classical poker test considers five successive integers and checks whether the observed quintuple frequencies correspond to the expected distribution. Inspired by poker hands consisting of five cards, the following integer combinations can occur: All different (abcde), one pair (aabcd) two pairs (aabbcc) three of a kind (aaabc), four of a kind (aaaaa) and poker (aaaaa).

In this project, given that the numbers under consideration are 32 bit numbers, the test will be modified such that hands of 4 integers will be used. 32 bits are more readily divisible by 4, which allows for a more efficient implementation of a 4-card test compared to a 5-card one. To model hands of 4 different integers, 4 pairs of 2 bits will be grouped together. This results in $\frac{32}{4 \times 2} = 4$ hands for chi-squared analysis, with the chance of each number occurring equaling $\frac{1}{4}$. [18] computes the 4-card probabilities as:

- Four of a kind: 0.001
- Three of a kind: 0.036
- Two pairs: 0.027
- One pair: 0.432
- All different: 0.504

Obtaining $k = 4 \times 5 = 20$ and $v = k - 1 = 19$, and using the above probability values as p_s values for $0 \leq s \leq 19$, it is possible to compute Y to obtain the p -value.