FPGA Implementation of a Pseudo-Random Aggregate Spectrum Generator for RF Hardware Test and Evaluation

Randeep S. Baweja

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Electrical Engineering

Harpreet S. Dhillon, Co-chair

William C. Headley, Co-chair

Alan J. Michaels

Patrick R. Schaumont

September 9, 2020

Blacksburg, Virginia

Keywords: FPGA, signal emulation, T&E.

Copyright 2020, Randeep S. Baweja

FPGA Implementation of a Pseudo-Random Aggregate Spectrum Generator for RF Hardware Test and Evaluation

Randeep S. Baweja

(ABSTRACT)

Test and evaluation (T&E) is a critically important step before in-the-field deployment of radio-frequency (RF) hardware in order to assure that the hardware meets its design requirements and specifications. Typically, T&E is performed either in a lab setting utilizing a software simulation environment or through real-world field testing. While the former approach is typically limited by the accuracy of the simulation models (particularly of the anticipated hardware effects) and by non-real-time data rates, the latter can be extremely costly in terms of time, money, and manpower. To build upon the strengths of these approaches and to mitigate their weaknesses, this work presents the development of an FPGA-based T&E tool that allows for real-time pseudo-random aggregate signal generation for testing RF receiver hardware (such as communication receivers, spectrum sensors, etc.). In particular, a framework is developed for an FPGA-based implementation of a test signal emulator that generates randomized aggregate spectral environments containing signals with random parameters such as center frequencies, bandwidths, start times, and durations, as well as receiver and channel effects such as additive white Gaussian noise (AWGN). To test the accuracy of the developed spectrum generation framework, the randomization properties of the framework are analyzed to assure correct probability distributions and independence. Additionally, FPGA implementation decisions, such as bit precision versus accuracy of the generated signal and the impact on the FPGA's hardware footprint, are analyzed. This analysis allows the test signal engineer to make informed decisions while designing a hardware-based RF test system. This framework is easily extensible to other signal types and channel models, and can be used to test a variety of signal-based applications.

FPGA Implementation of a Pseudo-Random Aggregate Spectrum Generator for RF Hardware Test and Evaluation

Randeep S. Baweja

(GENERAL AUDIENCE ABSTRACT)

Test and evaluation (T&E) is a critically important step before in-the-field deployment of radio-frequency signal hardware in order to assure that the hardware meets its design requirements and specifications. Typically, T&E is performed either in a lab setting utilizing a software simulation or through real-world field testing. While the former approach is typically limited by the accuracy of the simulation models and by slower data rates, the latter can be extremely costly in terms of time, money, and manpower. To address these issues, a hardware-based signal generation approach that takes the best of both methods mentioned above is developed in this thesis. This approach allows the user to accurately model a radio-frequency system without requiring expensive equipment. This work presents the development of a hardware-based T&E tool that allows for real-time random signal generation for testing radio-frequency receiver hardware (such as communication receivers). In particular, a framework is developed for an implementation of a test signal emulator that allows for user-defined randomization of test signal parameters such as frequencies, signal bandwidths, start times, and durations, as well as communications receiver effects. To test the accuracy of the developed emulation framework, the randomization properties of the framework are analyzed to assure correct probability distributions and independence. Additionally, hardware implementation decisions such as bit precision versus quality of the generated signal and the impact on the hardware footprint are analyzed. Ultimately, it is shown that this framework is easily extensible to other signal types and communication channel models.

Contents

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 4

1.3 Publications 5

2 Background **6**

2.1 Prior Work in Software 6

2.2 Prior Work in Hardware 9

2.3 Tool Flow 13

2.4 Summary 14

3 Random Signal Generator **15**

3.1 Linear Digital Amplitude-Phase Modulator 18

3.2 Filter and Interpolator for Random Bandwidth 20

3.3 Random Frequency Generator and Mixer 21

3.4 Random Time Duration 25

3.5 AWGN Generator 27

3.6 Uniform Pseudo-Random Generators 28

3.7 Gaussian Random Generators 31

3.8 Impact of Bit Precision on Statistical Properties 31

3.9 Random Signal Generator Output 36

4 Random Aggregate Spectrum Generator 39

4.1 Aggregation Issues 41

4.2 Periodicity 44

4.3 Hardware Utilization 50

4.3.1 FPGA Resources 51

4.3.2 Bit Precision vs. Hardware Utilization 56

4.3.3 Data Rate vs. Hardware Utilization 61

4.3.4 Parameter Choices vs. Hardware Utilization 64

4.4 Summary 68

5 Conclusions and Future Work 70

Bibliography **74**

List of Figures

2.1 Block diagram of a typical RF transmitter chain. 6

3.1 Xilinx System Generator block diagram of the developed FPGA-based Pseudo-Random Signal Generator. White blocks indicate a collection of blocks and are detailed in the other figures in this chapter. 16

3.2 A waterfall plot of the output of the random signal generator shows signal bursts with random durations, frequency, bandwidth, and SNR. 17

3.3 Xilinx SysGen block diagram of the "LDAPM" block of Figure 3.1, which accepts input bits and outputs the corresponding symbol using a LUT. . . . 18

3.4 Xilinx SysGen block diagram of two of the four filters in the "Filter" block of Figure 3.1. Each filter's output provides a modulated signal with a different bandwidth and is chosen randomly per signal burst using a multiplexer. . . . 20

3.5 Xilinx SysGen block diagram of the "Frequency Generator and Mixer" block of Figure 3.1. The "Black Box" block accepts four pseudo-random numbers and outputs a random frequency, signal power, and filter select bits for each signal burst of randomly chosen duration. 22

3.6 Xilinx SysGen block diagram of the “AWGN” block of Figure 3.1. This block performs four Box-Muller transformations of uniform random variables to generate complex Gaussian random numbers. 28

3.7 The correlation coefficients of the bits on the I and Q branches are close to zero at all points except for where they completely overlap with themselves. 30

3.8 The cross-correlation coefficients of the I and Q branches are close to zero at all times, showing that they are uncorrelated. 30

3.9 The full-precision experimental Gaussian random variable values on both the I and Q branches fit closely to their expected distributions. 32

3.10 Experimental joint Gaussian values are plotted against the theoretical distribution. The two curves nearly overlap, indicating the independence of the two variables. 32

3.11 When the precision is reduced to 8_6 and 8_5, the experimental Gaussian values do not match the theoretical curve at the tails in (a) and at the peak in both. 33

3.12 When the precision is increased to 9_6, the experimental Gaussian values fit their distribution better than with 8_6 and 8_5. The fit is even better at 10_7. 35

3.13 A waterfall plot of the random signal generator output shows signal bursts with random durations, frequency, bandwidth, and SNR. 37

3.14 The symbol error rate (SER) for the QPSK system simulation is very close to the theoretical QPSK SER curve. 38

4.1 The random aggregate spectrum generator is created by summing the output of three random test signal generators and AWGN. 40

4.2 A waterfall plot shows an aggregate spectrum of three simultaneous signals created using three instances of the random signal generator from Chapter 3. Each signal's parameters change independently of each other; each signal has a different center frequency, bandwidth, and SNR, and starts and stops at different times. 41

4.3 The PRNG top level block is shown in (a) and the internal diagram is shown in (b). The enable and reset inputs control the PRNG's operation. The eight constant blocks that feed the Chaos_PRNG block can be modified by the user before run time in order to change the seeds. 46

4.4 The ring generators found inside the Chaos_PRNG block are shown. The ring generator outputs serve as the addresses to the LUTs that contain the prime Galois field residues. 46

4.5 An example hardware utilization report provides a breakdown of how many hardware resources each subsystem uses. 53

4.6 A detailed hardware utilization report provides a detailed breakdown of how many hardware resources each System Generator module uses. 54

4.7 An example timing report shows that the system meets timing and shows how much delay and timing slack is present between each module in the system. . 55

4.8 The hardware utilization of the aggregate signal generator is shown as a function of system bit width. The utilization of most resources increases linearly, except for DSPs, for which the utilization grows exponentially. 56

4.9 The hardware utilization of the aggregate signal generator is shown as a function of system bit width without including resources that do not increase with bit size. 57

4.10 Hardware utilization is shown in percentage as a function of the number of signal branches instantiated in the aggregate system. As the number of branches increases, the hardware utilization of the system grows linearly (excluding the additional adders required). The line for DSP saturates at 9 signal branches due to a limitation in the hardware analysis tool. 60

4.11 Timing fails at 100MHz data rate. The exact location of the violation is the route between the source and destination (columns 6 and 7). There is significant negative slack, meaning that the system has a significant bottleneck that prevents it from running at a higher rate. 62

4.12 Xilinx System Generator highlights the blocks in which the timing violation occurs. 63

4.13 As the number of filter coefficients increases, the hardware utilization of the system barely increases. 64

4.14 As the ROM-depth increases, the hardware utilization of the system increases linearly. 66

4.15 Additional hardware resources are used to sum together the multiple branches without sacrificing dynamic range. This utilization increase is plotted as a function of number of signal branches. 67

List of Tables

3.1 Example values of the LUT-based sine wave generator. 23

3.2 The results of the Kolmogorov-Smirnov (K-S) test, Anderson-Darling (A-D) test, and Lilliefors tests are shown for each level of bit precision tested in this section. If a test passes, the adjacent column shows the maximum significance level for which it passes. Most of the tests fail for bit precisions less than 10_7. 34

4.1 Hardware resource utilization is shown as a function of number of signal branches. As the number of branches increases, resource utilization also increases linearly. 59

4.2 DSP usage vs. bit precision and number of branches. The result for 8-bit and 16-bit branches is experimental, and the 24-bit and 32-bit results are extrapolated. Red marks the point where utilization matches or exceeds the Zedboard Zynq FPGA's capacity. 61

4.3 Hardware resource utilization is shown as a function of system rate. The amount of hardware required does not increase except for when the design does not meet timing or when pipelined stages are added. In the table, (P) indicates when pipelining has been used to increase the system rate. 61

Chapter 1

Introduction

1.1 Motivation

Prior to their deployment, wireless (radio-frequency) applications must be thoroughly tested to ensure that they can successfully operate in a variety of spectral environments. Dynamic Spectrum Access (DSA) applications require a congested test environment full of signals with a variety of parameters in order to evaluate the system's ability to avoid interference and to look for unused spectrum to occupy. For spectrum sensing applications, an appropriate testing environment includes signals with varying power levels and duration. It also includes signals that vary in frequency and bandwidth to test the application's parameter estimation capabilities, and signals with different modulation schemes in order to test its signal classification capabilities.

The current framework for developing radio-frequency (RF) systems (such as spectrum sensors) typically involves software emulation and/or physical deployment to test and evaluate the system. While software solutions are less time-consuming to develop, they do not operate

at speeds fast enough to test high-frequency wideband systems in real time. Additionally, a software-based emulation will usually behave differently than a real system due to the challenge of modeling hardware and channel effects in software. Physical deployment is a costly operation in terms of time and money because it requires lots of expensive equipment, such as transmitters, receivers, and spectrum analyzers, that need to be properly set up and configured in a real-world environment. Many testing scenarios are run using this equipment, and it can take many hours or days to set up and perform all of the tests. To address these limitations, a field-programmable gate array based (FPGA-based) test signal generation framework that combines the best of these methods is presented in this work. The accelerative properties of FPGAs allow for wideband test spectrums to be generated in real time. These spectrums can be piped directly to the system under test, thus eliminating the need to store large data sets or to generate spectral scenarios beforehand. The use of physical hardware to generate and receive test spectrums allows the developer to account for issues such as bit precision and hardware effects. The use of hardware allows the user to plug the test spectrum generator directly into the system under test, and which then enables the consideration of receiver effects and analog phenomena during the test. As a result, the device can be tested with signal environments that more closely resemble those found in real life. In addition, the reconfigurable nature of FPGAs greatly reduces the overhead time between different test runs as compared to using other devices such as application-specific integrated circuits (ASICs).

FPGAs can be reconfigured with new firmware; this process is very different from software development. FPGA firmware is typically written using a hardware description language (HDL) such as Verilog or VHDL (Very High Speed Integrated Circuit Hardware Description Language). The engineer will typically start with a block diagram as a reference design to guide the HDL development. The engineer will write modules and subsystems that have

explicitly defined input and output ports. All wires, registers, and ports have explicitly defined bit widths. Most importantly, HDL code is parallel as opposed to sequential, meaning that all modules and logic blocks are instantiated simultaneously instead of sequentially iterating through the code like in software. In these ways, HDL code is much more hardware-oriented and requires the engineer to consider many more implementation details than is required in software programming. For this reason, some engineers choose not to use FPGAs for the development of their systems, and that is why there are not as many complete FPGA-based implementations of signal generators in the literature.

The architecture of an FPGA also differs from that of a processor. A processor has a fixed architecture that relies on instructions that control the data path and peripheral resources in order to perform computations. An FPGA is made up of many programmable logic blocks (PLBs), switches, interconnecting wires, and input/output ports. The PLBs in an FPGA include block random access memory (BRAM), digital signal processing units (DSPs), look-up tables (LUTs), and registers. Each time the FPGA is flashed with a new design, all of these resources are configured and routed according to a binary image file that is generated from the HDL. There are several FPGA development toolkits (such as Xilinx Vivado) that can aid in the compilation and synthesis of HDL code. These tools are designed to carefully allocate FPGA resources as efficiently as possible and to generate a binary image file to program the FPGA accordingly. There are a limited amount of resources on an FPGA (DSPs, BRAMs, LUTs, etc), so all HDL designs must be carefully written so that they use as few FPGA resources as possible. This consideration results in a trade space between bit precision, number of signal branches, and overall hardware resource utilization, and adds to the difficulty of implementing designs using FPGAs instead of software processors. However, as discussed above, FPGAs provide a number of benefits that outweigh the costs for the implementation of applications such as a real-time pseudo-random aggregate spectrum

generator.

Ideally, the aggregate spectrum generator should generate a variety of signals with non-deterministic parameters so that the device under test observes a multitude of stochastic environments. Here, non-deterministic refers to the statistical nature of the parameter selection and bit generation, which is pseudo-random. There are some commercially available modems that generate frequency, phase, and amplitude modulated test waveforms. These waveforms can be pre-programmed with a variety of changeable parameters such as frequency, bandwidth, and power [1][2]. These are good at simulating realistic signal environments, but are very expensive and do not allow real-time randomization. Other works, described in Chapter 2, can also generate a variety of test signals. However, these signals' parameters cannot be changed pseudo-randomly without user intervention. It is impractical to require the test engineer to stop the simulation, load in new parameters, and run it again. In this way, only one test scenario can be run at a time. The work presented in this thesis improves on the limitations presented in this section.

1.2 Contributions

This thesis introduces a random spectrum generation framework that allows for the real-time generation of test signals with randomly changing parameters. In Chapter 3, the architecture of the test signal generation framework is presented. The framework includes subsystems such as the symbol modulator, filter, frequency mixer, and Gaussian noise generator. Each subsystem is explained in detail, with a focus on how the corresponding parameter randomizations are achieved. The chapter also includes a discussion of the random number generators that are the source of the parameter randomization in the system.

In Chapter 4, several instances of the test signal generator discussed in Chapter 3 are com-

bined to form the random aggregate spectrum generator. The signals from each test signal branch are summed together before additive white Gaussian noise (AWGN) is also added. Each branch acts as separate transmitter and can have its own set of filters, modulators, center frequencies, etc. Each signal varies in duration, frequency, bandwidth, and modulation scheme. Each transmitter can start or stop transmitting at any time, and may transmit different signals in simultaneous bursts. This is why each branch needs its own set of independent modular components (modulators, filters, etc). The framework allows modular components to be easily swapped in and out as desired.

In Chapter 4, the hardware utilization statistics for the random test signal generator and the random aggregate spectrum generator are also analyzed to determine how many signal branches can be instantiated in one FPGA. As mentioned in Section 1.1, there is a natural trade-off between bit precision, quantity of signal generation modules, and overall hardware utilization. This trade-off is discussed in detail in Chapter 4. Prior to these chapters, a summary of the prior work is presented in Chapter 2.

1.3 Publications

- Randeep S. Baweja, Devin Ridge, Harpreet S. Dhillon, and William C. Headley. “FPGA Implementation of a Pseudo-Random Signal Generator for RF Hardware Test and Evaluation.” *(Accepted to IEEE IPCCC, 2020)*

Chapter 2

Background

2.1 Prior Work in Software

Several software implementations of signal generators already exist in the literature. These works implement signal transmit chains (shown in Figure 2.1) with variable parameters using tools such as Matlab, GNU Radio, Liquid DSP, and LabView. However, these implementations are unable to achieve real-time rates higher than 50MHz. In addition, they inherently cannot accurately emulate hardware effects. Most of them also do not provide the ability to randomly change parameters during run time, which is helpful when testing signal classification algorithms. Many implementations only allow one type of signal to be run at once and the simulations must be stopped in order to load new signal parameters.

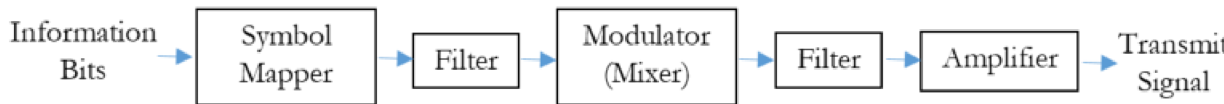


Figure 2.1: Block diagram of a typical RF transmitter chain.

Clark developed a parameterized signal generator in GNU Radio [3]. When it is run, this GNU Radio flow graph generates a spectrum with user-chosen characteristics such as frequency and modulation scheme. Some of these parameters can be changed during run time. In the flow graph, the signal is resampled at a random rate using a polyphase filter bank channelizer. The output of the channelizer is a signal that is split into ten narrowband channels that can be turned on or off by the user during run time by selecting options on the GUI [3]. The narrowband signals are synthesized into an aggregate wideband signal that is displayed on a waterfall plot. This channelizer-synthesizer structure is an efficient way to mix signals up to the same aggregate rate, and this can be used with ten individual narrowband signals to get an aggregate of ten signals. While Clark's signal generator implementation uses an efficient software-based method, it is only able to generate up to 40MHz of bandwidth in real time.

A different software-based arbitrary waveform generator (AWG) was developed by Asami et al. This AWG creates spread spectrum test signals at baseband and intermediate frequencies (IF) [4]. A uniform random generator provides the random bits for the in-phase (I) and quadrature (Q) branches in the system. A spreading code is applied to both of these branches. The signals are then upsampled and filtered before being mixed up to RF using an analog mixer. This AWG's output signal only has fixed characteristics, but it does take into account channel effects such as AWGN and Rayleigh fading. Asami et al. also describe bit error rate testing for their system at different levels of SNR. This can be used to verify that the signals generated by the system match their theoretical specifications. The work done by Asami et al. provides a good framework for generating test signals, but the framework is unable to create a test environment with several concurrent signals that can change during run time.

Beuran et al. implemented a software-based wireless local area network (WLAN) emulator that accepts a test scenario description that is written prior to run time and produces signals

accordingly [5]. The scenarios instruct the simulation to turn on several signals that vary in physical position and signal power. The system is meant to serve as a test bed for network application software as a precursor to testing on hardware. The system also calculates frame rate error (FER) for each of the simulated communication links [5]. The work done by Beuran et al. is useful because it can simulate user-defined scenarios with signals that change during run time to represent the activity of several transmitters. However, this system cannot randomly generate new signal environments; the user must create each scenario beforehand, and this can become tedious if there are many scenarios to test. In this situation, it would be beneficial to have a system that creates its own unique, realistic scenarios each time. In addition, this system is implemented in software and therefore cannot support the real-time generation of high-bandwidth signals.

In order to produce signals with different frequencies, sine and cosine waves with those frequencies must be generated. Several software-based works use ROM-based direct digital synthesis (DDS) to generate a variety of waveforms, including sinusoids of various frequencies [6][7]. This method is memory-efficient and typically involves the use of a frequency control variable that is input into an accumulator that wraps around to zero after reaching its maximum value. The output of the accumulator serves as the address of a LUT (implemented as a ROM) that contains the values of the desired sinusoid. If the frequency control variable is set to 1, then the values in the sinusoid LUT are outputted one by one. If the frequency control variable has a larger value, n, then every nth value is read out, resulting in a sinusoid of higher frequency. This method can be used to generate the carrier sinusoid that is mixed with the baseband waveform. A similar method can be used to create any arbitrary baseband waveform. Xiaodong et al. created a micro-controller-based AWG that uses the ROM-based DDS method [6]. While this implementation involves the efficient generation of waveforms, it is only able to create real-time signals with data rates up to 49MHz. Using this same

method, Nasir et al. were able to achieve rates up to 40 MHz using DDS on a microcontroller programmed with C code via Arduino [7].

In the prior work, software-based test signal generators are shown to have a limited range of frequency and bandwidth [3] [6] [7]. Some of them are also limited by their inability to change parameters during run time. In the next section, the abilities of a variety of hardware-based AWGs are compared to abilities of these software implementations.

2.2 Prior Work in Hardware

Arbitrary waveform generators similar to the ones discussed in Section 2.1 have also been implemented on FPGAs and DSPs. Generally, these works are able to generate signals at higher rates than the software implementations and are able to provide some parameter flexibility, though not during run time. Digital hardware implementations of most of the RF transmit chain (Figure 2.1) can be found in the literature (except for the amplifier, which is done in analog). In addition, some work, including that of Dong et al. and Salerno et al., use the DDS method described in Section 2.1 in their FPGA implementations of AWGs to generate carrier sinusoids [8][9]. They provide an equation (shown below) that relates the frequency control variable to the actual output frequency of the DDS system:

$$f_o = f_c K / 2^N \tag{2.1}$$

where f_o is the output frequency, f_c is the reference frequency (clock rate), K is the value of the frequency control variable, and N is the width of the LUT containing the sinusoid. According to the sampling theorem, $K < 2^{N-1}$ [9]. Similar equations that summarize this relationship can also be found in [10].

Salerno et al. use four DDS blocks and a multiplexer to generate sinusoids at four times the speed as compared to using one block. Each LUT has one fourth of the sinusoid's values such that consecutively reading one value from each DDS block during one clock cycle yields four consecutive sinusoid values [9]. The signals produced in this implementation have frequencies only in the range of hundreds of kilohertz and bandwidths of 50 kHz, but the main takeaways from the works are the LUT-based DDS implementations.

Tie-liang et al. and Hu et al. also use the LUT-based DDS method to generate an aggregate of sinusoids [11] [12]. In this implementation, the sinusoids are unmodulated, so the aggregate spectrum only contains impulses at the corresponding frequencies (in the range of a few kilohertz) [12]. These works successfully created an aggregate test signal environment but with limited variation and frequency. Bisiaux et al. created an aggregate spectrum (with modulated sinusoids) with center frequencies up to 3.2GHz and bandwidths up to 100MHz [13], which is much higher than the ones created by the software versions described in the previous section. However, this implementation still does not provide a dynamic signal generation framework, as the parameters cannot be changed during run time.

An advantage of using the ROM-based DDS approach described previously is its simplicity. Any arbitrary waveform can be stored in the ROM-based LUT, and each address is consecutively read out to generate the waveform (in this case a carrier sinusoid). This approach also allows for higher output speeds than some other approaches. The main issue with this approach is that the memory usage increases linearly as the number of addresses is increased in order to improve the frequency resolution. This results in an increase in memory and power consumption. However, there are techniques to reduce memory consumption without sacrificing frequency resolution [14].

There are also several other approaches to DDS. One approach uses pipelined parabolic approximations and LUTs to estimate sine values. While this implementation operates at

similar rates as the simple ROM-based DDS method, it is more complex and uses more power [14]. Another example of a ROM-less approach is one that uses interpolation algorithms that perform phase-to-amplitude conversions [10]. This algorithm saves memory because it does not require a ROM-based LUT, but its performance is limited in that its maximum output frequency is much smaller than the sampling rate of the system [10]. Another computation-based algorithm is the well-known Coordinate Rotation Digital Computer (CORDIC) algorithm. This algorithm, which involves 2-D vector rotations, is very flexible and can be used to generate low-frequency signals without compromising the resolution of the produced signal [15][16]. This overcomes one of the weakness of ROM-based DDS, which is the trade-off between ROM-size and frequency resolution. The CORDIC algorithm is also more efficient than other computational methods (such as Taylor series approximations) because it only requires adders, subtractors, and shifters (no multipliers or dividers, which are resource-hungry). However, the CORDIC algorithm and other computational methods are more difficult to implement than the ROM-based method [16]. For systems that only require limited frequency resolution, the simple ROM-based approach is sufficient and uses less hardware. However, for systems that require more frequency choices, the size of the ROM grows linearly, and the CORDIC algorithm is more appropriate. In this thesis, the simple ROM-based method is used due to its simplicity and efficiency, but future work could use the CORDIC method instead. In general, DDS solutions provide better control of the frequency generation (better tuning resolution) as compared to analog techniques such as phase-locked loops (PLLs) [10].

Overall, the hardware implementations discussed in this section are able to produce high-frequency, wideband signals in real time, and thus improve on the software-based signal generators in Section 2.1. However, in all of these works, the signal characteristics do not change during run time, and the parameters are not randomly chosen; they are fixed by

the user before run time. In order to properly test RF algorithms, a variety of signals with different parameters must be presented in rapid succession. This tests the system's capability to respond to a variety of signals in a spectral environment. The work discussed in this thesis addresses this issue by randomly changing signal parameters during run time.

Some of the other signal modules developed in this thesis are also guided by prior work. Individual components of a signal generator, such as modulators, mixers, and filters, have previously been developed for FPGAs. Jammu et al. implemented binary linear digital modulators using Xilinx System Generator with the constraint of using as few blocks as possible [17]. They generated hardware description language (HDL) code and evaluated the hardware resource utilization on an FPGA for the following modulation schemes: binary amplitude, binary frequency, binary phase, and differential phase shift keying [17]. Al Safi and Bazuin introduced a few novel approaches to create the same digital modulators as Jammu et al. using DDS with a LUT and a reverse-address accumulator [18]. Vu et al. created a 16 quadrature-amplitude modulation (16-QAM) transmitter and receiver for an FPGA using System Generator [19]. While the motivation behind these works was not to create a full-blown test signal generator, each introduced novel approaches to FPGA-based signal generation that guided some of the design decisions discussed in this thesis.

Other platforms, such as DSP chips, have also been used to implement test signal generators [20] [21] [22]. DDS methods similar to those previously described are used in these works, and frequencies in the range of a few hundred MHz are achieved. DSPs are microprocessors specialized for signal processing, and are programmable and flexible. However, they are not able to run at rates as high as FPGAs [23]. Unlike FPGAs, they can only support the concurrent generation of a few wideband signals. FPGAs can be clocked at a higher rate and thus, can produce a wider spectrum that can fit more wideband signals. FPGAs are also more flexible than ASICs because they are reprogrammable and thus have a faster

development time That is why an FPGA was the chosen target device for the work done in this thesis.

2.3 Tool Flow

The works discussed in Sections 2.1 and 2.2 used a variety of software tools to develop signal generation frameworks. Matlab, Simulink, Xilinx System Generator, and GNU Radio are among the most widely used. There are also some waveform development efforts done using LabView [24][25]. Only a few of the authors of the works discussed in this chapter wrote their own HDL because it takes much longer to develop a system using HDL than it does using software tools. Since the target platform for the work presented in this thesis is an FPGA, Xilinx System Generator and Simulink were the tools chosen for development. The System Generator library contains hundreds of pre-written HDL blocks, which the engineer can use to create FPGA designs in Simulink. System Generator is able to generate HDL code based on this user-created model [18]. It can also provide a hardware utilization report that details the usage of specific FPGA resources such as ROMs and DSP slices. It can also produce a preliminary report that shows if the system will pass a timing check.

A design created in System Generator can then be ported into a Xilinx Vivado design as an IP core. Xilinx Vivado can compile and synthesize the design, run place and route, and download the resulting binary image file onto a target FPGA to program it. System Generator and Vivado together cover the entire tool flow from designing a system to downloading it onto an FPGA without requiring the user to write any HDL code.

2.4 Summary

In order to properly test RF algorithms, they must be presented with a variety of spectral scenarios that provide signals with different characteristics. Having a congested test environment can help evaluate DSA and spectrum sensing applications, as an example. In the prior work, test signal generators have been developed using a variety of hardware and software platforms. The hardware-based AWGs are able to generate realistic, wideband signals at higher frequencies than their software counterparts. However, these generators do not randomly vary the signal parameters during run time. Instead, the signal parameters are fixed prior to run time, and new scenarios have to be set up before each test run. The work presented in this thesis builds on the prior work to create a random aggregate spectrum generator that can provide several signals with random parameters, such as frequency, bandwidth, modulation scheme, and signal-to-noise ratio, that change non-deterministically during run time.

Chapter 3

Random Signal Generator

In this chapter, a novel FPGA-based pseudo-random test signal generator is introduced. The System Generator block diagram of the developed system is presented in Figure 3.1. The system shown in the diagram represents one of the signal branches that are summed together to create a random aggregate spectrum (presented in Chapter 4). This signal generator branch produces a signal that exhibits a random change in several key characteristics of a communication signal, namely its modulation format, center frequency, bandwidth, power, duration, and start time. The signal's center frequency, bandwidth, power, and duration can all be seen changing randomly in the waterfall plot shown in Figure 3.2. In order to generate this signal, random select bits are used to choose the features in the modules described in this section.

This chapter breaks down each of the components of the developed generator as follows. Sections 3.1 and 3.2 present the symbol generator which generates and filters the random bits for Linear Digital Amplitude Phase Modulations (LDAPMs) such as phase shift keying (PSK), quadrature amplitude modulation (QAM), and amplitude shift keying (ASK) (see Grami's and Lathi's books for more information on digital modulation schemes [26] [27]).

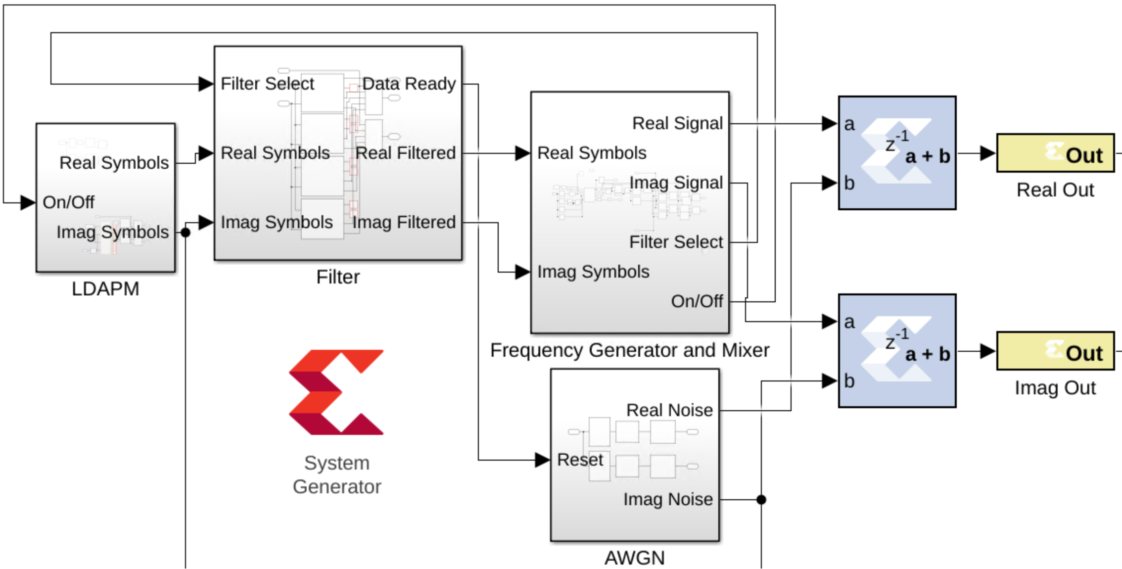


Figure 3.1: Xilinx System Generator block diagram of the developed FPGA-based Pseudo-Random Signal Generator. White blocks indicate a collection of blocks and are detailed in the other figures in this chapter.

Sections 3.3 and 3.4 present the sine wave generator, which generates a carrier wave of random frequency that is mixed with the modulated symbols for a random duration of time, and Section 3.5 presents the AWGN generator [28].

The random signal generator framework employs five uniform pseudo-random generators and two Xilinx White Gaussian Noise Generators (that also contain uniform pseudo-random generators within their hierarchy). An analysis of the underlying random number generators is provided in Sections 3.6-3.8. Each of the generators are statistically analyzed to verify that they are independent from one another in order to ensure there is no bias between the random elements in the system (the bias can be caused by using the same structure and initial conditions in each generator). Each random generator must also prove that it fits the expected distribution so that the random parameters are not biased or skewed from the expected behavior. This analysis becomes especially important when dealing with hardware solutions, as developed in this thesis, due to bit precision considerations. Typically, there is a

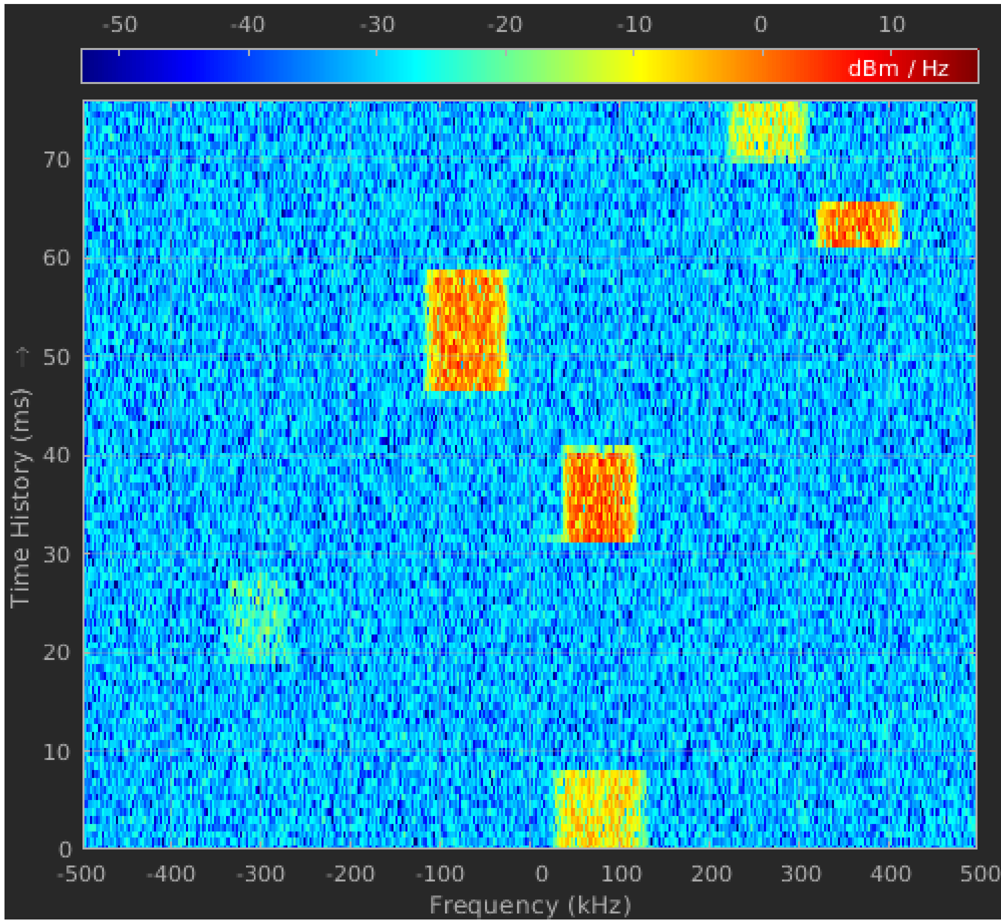


Figure 3.2: A waterfall plot of the output of the random signal generator shows signal bursts with random durations, frequency, bandwidth, and SNR.

trade-off between precision and performance, and in this chapter, performance is defined by how well the generators match their distributions and how correlated their outputs become once fewer bits of precision are used.

For the following analysis, a QPSK modulation format is assumed. Results can be extended to other LDAPMs.

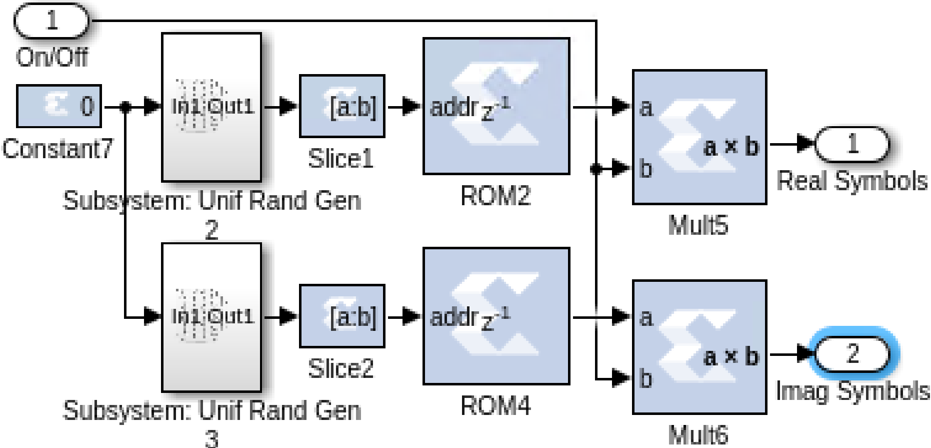


Figure 3.3: Xilinx SysGen block diagram of the "LDAPM" block of Figure 3.1, which accepts input bits and outputs the corresponding symbol using a LUT.

3.1 Linear Digital Amplitude-Phase Modulator

Linear digital amplitude-phase modulated (LDAPM) signals carry information bits in the signal's phase and amplitude. Amplitude shift keying (ASK) signals are amplitude modulated, phase shift keying (PSK) signals are phase modulated, and quadrature amplitude modulation (QAM) signals are both amplitude and phased modulated [27]. Some of these modulation schemes only transmit symbols on the in-phase/real (I) branch of the signal (ASK, binary PSK), while the rest transmit on both the in-phase and quadrature/imaginary (Q) branches. The distance between the symbols on the constellation diagram for each modulation schemes determines the likelihood of a symbol error. Modulation schemes that have symbols that are placed on opposite ends of the constellation (like BPSK) tend to have a lower symbol error rate (SER) than schemes that place symbols closer to each other (64-QAM, for example). However, higher-order schemes have better bandwidth efficiency.

The Linear Digital Amplitude-Phase Modulator (LDAPM), shown in Figure 3.3, takes in random data bits and outputs complex symbols on the I and Q branches. The I/Q symbol components are chosen using a LUT of possible values for the modulation scheme that

was selected for the signal. The random data bits that serve as the address for the LUT are pseudo-randomly chosen from the uniform distribution, which is the maximum entropy distribution over a given interval. In other words, this distribution is the "most random" because each outcome is equally likely. The LDAPM uses random bits because real communication signals are often whitened, which involves scrambling the bits prior to transmission to improve the spectral properties of the signal. This scrambling causes the bits to appear random.

The uniform random bits are drawn from a slice of the output of an 8-bit leap-forward linear feedback shift register (LFSR). A leap-forward LFSR is needed so that consecutive 8-bit outputs are uncorrelated. A traditional LFSR will provide one new bit per cycle, whereas an 8-bit leap-forward LFSR will create eight new bits per cycle [29]. The LFSR used in this implementation was originally part of a hierarchical block of a Xilinx AWGN generator [28].

The required size of the LFSR slice depends on the order of the modulation scheme. For example, 4-PSK or QPSK requires two bits since there are four possible symbols. An 8-bit LFSR allows for up to $2^8 = 256$ different symbol options. The LDAPM modulation scheme in each branch can easily be changed by swapping out LUTs prior to run time or by leveraging a multiplexer during run time. The slice size for each modulation scheme is also modified accordingly.

For a fixed point architecture, the bit precision used to represent the LDAPM symbols affects their accuracy. For example, if only three bits of decimal precision are used, then symbols can only have a resolution of 0.125. This would mean that even 8-PSK could not be accurately represented because it requires values $\frac{1}{\sqrt{2}}$ (or about 0.7071) and the closest value that can be represented is 0.75, which results in an error of 0.0429. Using eight bits of precision leads to an error of about 6.875×10^{-5} . The LDAPM signals are represented using 16 bits, 12 of which are for decimal precision. This allows symbols increments as small as $\frac{1}{4096}$ to

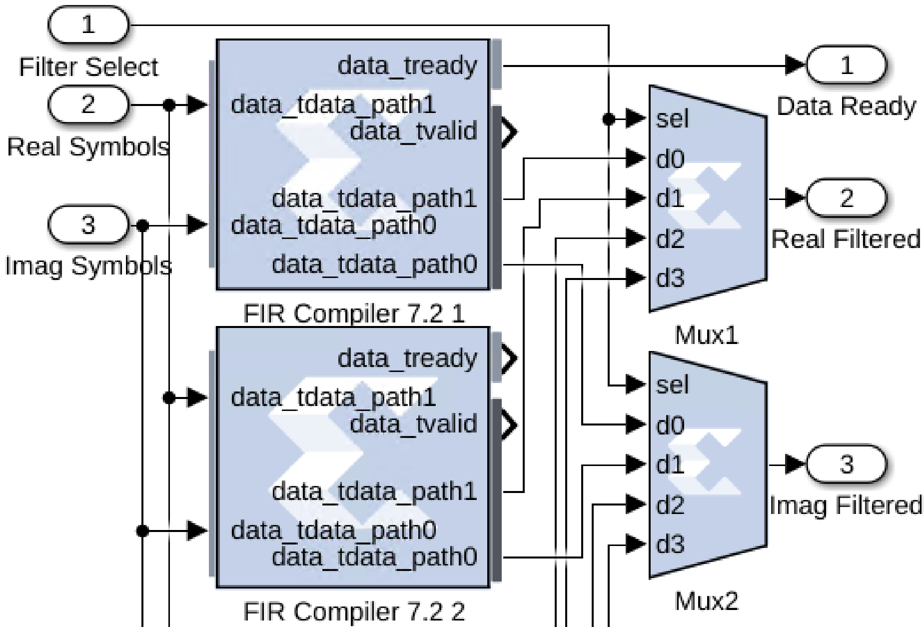


Figure 3.4: Xilinx SysGen block diagram of two of the four filters in the "Filter" block of Figure 3.1. Each filter's output provides a modulated signal with a different bandwidth and is chosen randomly per signal burst using a multiplexer.

be represented, with extra integer bits included as buffer for calculations further down the transmit chain.

3.2 Filter and Interpolator for Random Bandwidth

After the symbols are generated by the LDAPM, they are passed through a filter and interpolator to adjust the bandwidth of the signal and to reduce inter-symbol interference (ISI). Xilinx's finite impulse response (FIR) Compiler block allows the user to specify a set of filter coefficients prior to run time. In order to enable randomized bandwidths during run time, the developed system utilizes several FIR Compiler blocks with different coefficient sets. A multiplexer with uniform random select bits is used to randomly choose which filter's output is used for each signal burst. In the analysis that follows, four filters are used, two of which

are shown in Figure 3.4. The nominal bandwidth choices offered by these filters are $\frac{1}{12}$, $\frac{1}{10}$, $\frac{1}{8}$, and $\frac{1}{6}$ of the final desired signal bandwidth. These values were chosen so that a difference in bandwidth could be observed between different signals shown on the spectrum analyzer, but any range of nominal bandwidths can be chosen.

Naturally, there is a trade-off between the number of possible bandwidths and the amount of space available on the FPGA to accommodate the FIR filters. To alleviate this limitation, future work could leverage a filter block with variable interpolation rates so that only one filter and set of coefficients is required. The bandwidth variation would then result from changes in the interpolation rate. The downside to this approach is that other filter parameters, such as span and roll-off factor, would not be randomized during run time. In addition, using a bank of Xilinx FIR filters (the approach developed in this thesis) allows for increased modularity.

As was the case for the LDAPM, there is a trade-off between the bit precision of the filter coefficients and the bandwidth of the signal. As the number of bits is reduced, the resulting quantization noise distorts the signal's spectral shape [30]. Similar to the LDAPM, the filter and interpolator utilizes 16 bits of precision. Finally, there is a trade-off between hardware usage and the order, or span, of the filter. A higher order filter typically results in better spectral performance (in terms of roll-off, etc.) at the expense of requiring more physical space on the chip [30]. This trade-off is discussed in more detail in Section 4.3.4.

3.3 Random Frequency Generator and Mixer

After passing through the filter and interpolator, the symbols are mixed up to a pseudo-randomly chosen center frequency. This process first requires the generation of a sine wave of that chosen frequency. While there is a built-in sine wave block available in the Xilinx

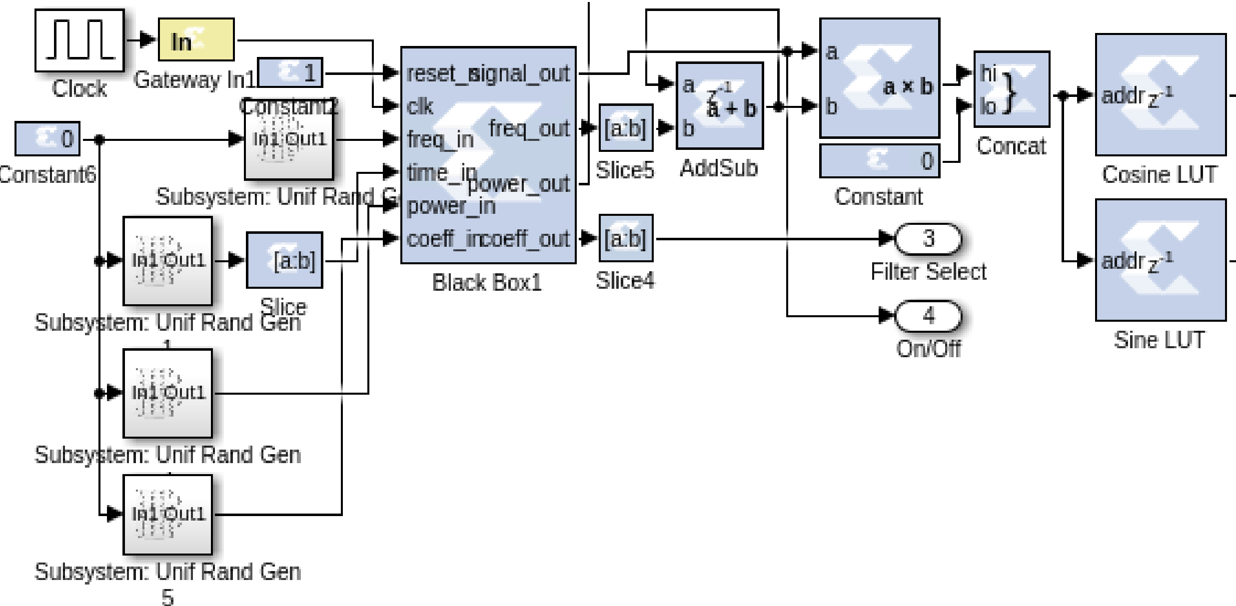


Figure 3.5: Xilinx SysGen block diagram of the "Frequency Generator and Mixer" block of Figure 3.1. The "Black Box" block accepts four pseudo-random numbers and outputs a random frequency, signal power, and filter select bits for each signal burst of randomly chosen duration.

System Generator library, the block does not allow the random frequency to change during run time. The sine wave block only allows the user to modify the sinusoid's frequency prior to run time before the design is recompiled. The frequency-varying functionality is provided in the developed signal generator system using an oversampled LUT-based sine/cosine wave (stored in the Sine and Cosine LUTs in Figure 3.5). As discussed in Section 2.2, this ROM-based DDS method is efficient for smaller LUT sizes. The size of the ROM-based LUT increases linearly if more frequency options are required, so the CORDIC algorithm is more efficient if the system requires higher frequency resolution. For the implementation developed in this thesis, a 256-address LUT was used as a starting point, but a larger table or the CORDIC method could be implemented in future work.

Using the ROM-based DDS method, a change in the step value by which the LUT address is incremented causes the frequency of the sine wave to change (see Table 3.1). The LUT

Table 3.1: Example values of the LUT-based sine wave generator.

LUT Index	Sine Wave Value	Sine Output Plots
0	0.0000	
1	0.3827	
2	0.7071	
3	0.9239	
4	1.0000	
5	0.9239	
6	0.7071	
7	0.3827	Index Step Size 1 (0,1,2,3,....14,15)
8	0.0000	
9	-0.3827	
10	-0.7071	
11	-0.9239	
12	-1.0000	
13	-0.9239	
14	-0.7071	
15	-0.3827	Index Step Size 2 (0,2,4,6,....12,14)

address step value is chosen randomly from an 8-bit uniform pseudo-random number generator and is added to an accumulator (Add-Sub block in Figure 3.5) that wraps around to zero after reaching 255. The output of the accumulator provides the addresses to the LUT that contains the sine wave values. A step size of 2 results in a sine wave of twice the frequency as a step size of 1. A 256-address LUT allows for 64 positive and negative center frequency options for each signal burst. The relationship between the step size and the resulting frequency is summarized in Equation 3.1:

$$N = fS/R_s, \quad 0 \leq f \leq R_s/4 \tag{3.1}$$

$$N = S - fS/R_s, \quad -R_s/4 \leq f \leq 0 \tag{3.2}$$

where N is the step size, S is the size of the sinusoid's LUT, R_s is the sample rate, and f is the output frequency. These relations are similar to the ones found in [10]. The maximum step size given in the example above is 255, but that step size results in the same frequency as

a step size of 1, but in reverse (negative frequency). If the step size is 128, the values always remain at zero because the 0th and 128th LUT values are both zero. At step size 129, the address values are 0, 129, 2, 130, 3, 131, etc. In this case, the corresponding output values start around zero and gradually increase, resulting in a sinusoid that changes amplitude with each iteration, which is undesired behavior. Any step size above 64 ($S/4$) results in a sinusoid with variable amplitude, zero amplitude, or the negative frequency corresponding to step sizes 1 to 64 (as is the case with step sizes of 255 to 192). In a case where the sample rate is 1Msps, step sizes (N) of 1 to 64 result in frequencies that range from 3.9kHz to 250kHz and step sizes 192 to 255 yield frequencies between -3.9kHz and -250kHz. The output frequency increases/decreases by 3.90625 kHz with every change in step size. Additional logic was added to the random frequency generator to ensure that the signals stay within the $\frac{-R_s}{2}$ to $\frac{R_s}{2}$ range, inclusive of their bandwidth. To do this, the step values corresponding to the frequencies of $\frac{-R_s}{2} + \frac{B}{2}$ to $\frac{R_s}{2} - \frac{B}{2}$ are found, where B represents the maximum bandwidth of the signal. The randomly chosen step value is compared with these bounds, and if the value is out of range, it is mapped to one of the valid ones. For example, if the step size is between 65 and 127, then 64 is subtracted from it to map it into the acceptable range of 1 to 64. If the step size is 128 to 191, then 64 is added to map it into the range of 192 to 255. This mapping still results in a uniform distribution, but over a different range.

To change the baseband signal's center frequency, the resulting sinusoid is mixed with the filtered baseband signal. This complex-valued mixing operation is performed using multiplication, addition, and subtraction blocks based on the following equations:

$$P_I = X_I Y_I - X_Q Y_Q \tag{3.3}$$

$$P_Q = X_I Y_Q - X_Q Y_I, \tag{3.4}$$

where P is the output product, and X and Y are the inputs. The subscripts, I and Q, represent the real and imaginary components, respectively. The resulting signal is given 32 bits of precision with 24 fractional bits to accommodate the multiplication of the symbols and the sine wave, each of which are given 16 bits (12 for the decimal portion). The product of multiplication requires twice the bits of the larger multiplicand.

3.4 Random Time Duration

In order to vary the duration of each signal burst, a uniform pseudo-random generator of size eight is used to generate a number between 0 and 255. This number serves as the maximum count value in the custom HDL block (the "Black Box" in Figure 3.5) that was written to randomize burst duration. The custom block was designed such that its output changes from 1 to 0 (or vice versa) when the count value is reached. This output is used as an enable signal at the input of the sine wave LUT. This same signal is also applied to the input of the interpolating raised-cosine filters (applied in Mult5 and Mult6 in the LDAPM). If applied after the filters, this would cause a large discontinuity in the signal, leading to the deterioration of the signal's spectrum.

Each time the enable signal is toggled, the custom block accepts a new random number as the next maximum count value, along with a new randomly chosen frequency and SNR, in order for each new signal burst to have its own randomized parameters. In the system described in this work, the rate provided to the custom block is 1kHz. This causes the block to provide new random values at $\frac{1}{100}$ of the symbol rate (which is 100ksps) so that a minimum of 100 symbols are transmitted for each new parameters. As a result, the maximum possible signal burst length is 0.255 seconds (25500 symbols) and the minimum is 0.001 seconds (100 symbols).

The uniform distribution is chosen for this block because it maximizes entropy or randomness. This makes it equally likely to select any value in the range of 0 to 255. The probability of signal detection typically increases if the observation window is longer (for static channels), so it is useful to test different observation lengths. However, in communications, time duration is sometimes modeled as a Poisson distribution [31]. In the future, other distributions can be derived from the uniform distribution by using inverse transform sampling. This method generates any probability distribution given its cumulative distribution function (CDF) [32]. The uniform distribution over the interval 0 to 1 can be used since the CDF of a distribution ranges from 0 to 1. The inverse CDF (or quantile function) of the desired distribution is applied to the uniform distribution in order to obtain the target distribution. For example, in order to obtain the exponential distribution, shown in Equation 3.5, the inverse (Equation 3.7) of the CDF (Equation 3.6) is applied to the uniform distribution, where λ is the rate of the exponential distribution, x is the input to the distribution, and U is a value drawn from the uniform distribution.

$$\lambda e^{-\lambda x}, x \geq 0 \tag{3.5}$$

$$1 - e^{-\lambda x}, x \geq 0 \tag{3.6}$$

$$-\frac{1}{\lambda} log(1 - U) \tag{3.7}$$

Similarly, to generate a Rayleigh distribution (Equation 3.8), the inverse of the Rayleigh CDF (Equation 3.9) is applied to the uniform distribution [32], where σ is the rate or scaling factor, x is the input to the distribution, and U is a value drawn from the uniform distribution.

$$\frac{x}{\sigma} e^{\frac{x^2}{2\sigma^2}} \tag{3.8}$$

$$\sigma \sqrt{-log(U)} \tag{3.9}$$

Thus the use of the uniform distribution serves as a starting point for further development which would entail applying the inverse CDF to the uniform random bits. The inverse CDFs could be implemented as a small System Generator module that can easily be swapped out before run time in order to change the distribution as desired. Depending on the complexity of the inverse CDF, this module may add latency to the system and may increase its hardware utilization slightly.

3.5 AWGN Generator

The developed framework is designed to accommodate scenarios in which the system under test is wired directly to the random test signal generator (e.g. not utilizing RF front ends). As a result, AWGN needs to be added to the system. In order to emulate an AWGN channel, this system uses two Xilinx White Gaussian Noise Generator (WGNG) blocks, the architecture of which is shown in Figure 3.6, to generate noise samples for the I and Q branches of the signal. The WGNG block uses the concepts of the Box Muller algorithm and the Central Limit Theorem and is implemented using leap-forward LFSRs [28]. The Box Muller algorithm transforms several uniform random variables into one Gaussian random variable. According to [33], a "good" WGNG requires six bits of decimal resolution to achieve a high-accuracy fit to the Gaussian distribution. More specifically, according to [34], seven bits of precision is sufficient to produce a deviation of less than 0.1dB. The output of the Xilinx WGNG has width of 12 bits, including seven bits after the decimal [33]. A rigorous discussion of the Gaussian distribution and required bit precision will be presented in Sections 3.7 and 3.8.

To achieve a target signal-to-noise ratio (SNR), the AWGN samples produced by the Xilinx block are normalized and the transmit signal is multiplied by a pseudo-randomly chosen

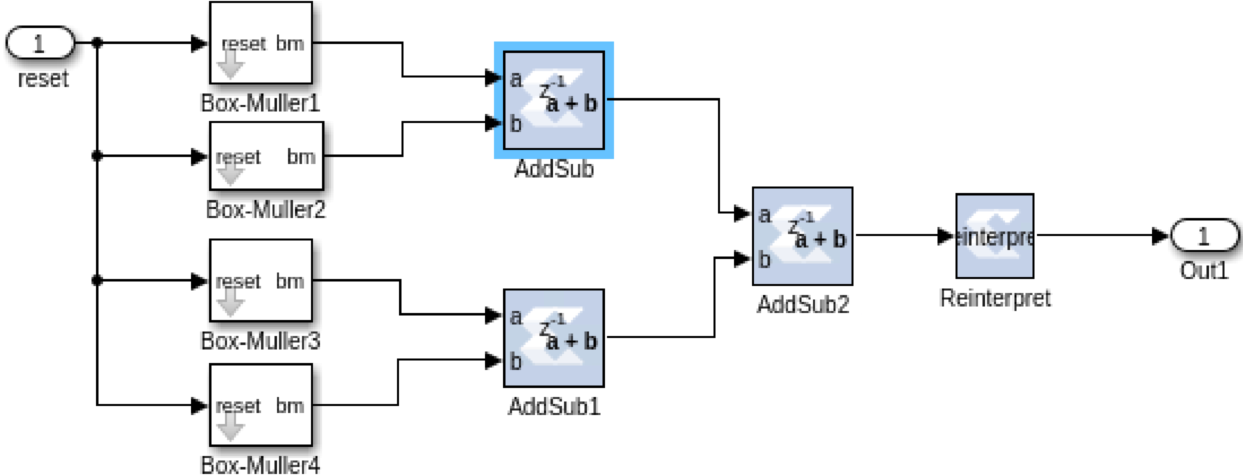


Figure 3.6: Xilinx SysGen block diagram of the "AWGN" block of Figure 3.1. This block performs four Box-Muller transformations of uniform random variables to generate complex Gaussian random numbers.

value. The number of bits used to choose an SNR thus sets the amount of unique SNRs that can be used. For the analysis that follows, an 8-bit uniform pseudo-random generator, similar to the ones used for random time and frequency, is used to generate a number between 0 and 255 that is appropriately scaled to provide SNRs between -10dB and 14.06dB.

3.6 Uniform Pseudo-Random Generators

Two different identically-distributed uniform pseudo-random number generators are used to create the in-phase (I) and quadrature (Q) components of the modulated symbols in the LDAPM. The two generators can be shown to be independent if their joint distribution is the same as the product of their marginal distributions [31].

For QPSK, which is used as an example, only one bit is required to represent the symbols on each branch. To prove the independence of the two branches, the outputs of both 1-bit uniform random number generators are counted for ten thousand bits (trials) and the results

are shown in (3.10)-(3.12), where $f_X(x)$ denotes the probability mass function.

$$f_{X_1,X_2}(x_1,x_2) = \begin{cases} 0.24910 & x_1 = 0, x_2 = 0 \\ 0.25089 & x_1 = 0, x_2 = 1 \\ 0.24820 & x_1 = 1, x_2 = 0 \\ 0.25175 & x_1 = 1, x_2 = 1 \end{cases} \tag{3.10}$$

$$f_{X_1}(x_1) = \begin{cases} 0.50002 & x_1 = 0 \\ 0.49998 & x_1 = 1 \end{cases} \tag{3.11}$$

$$f_{X_2}(x_2) = \begin{cases} 0.49733 & x_2 = 0 \\ 0.50267 & x_2 = 1 \end{cases} \tag{3.12}$$

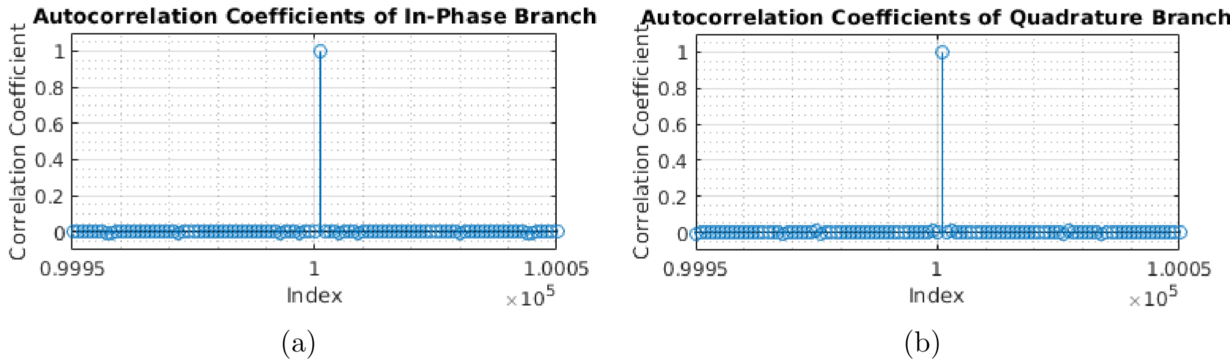
Overall, the maximum departure from the standard uniform distribution in the trials is 0.018%, so it can be reasonably concluded that the uniform generators (and thus the I and Q branches of the modulated symbols) are independent of each other.

Next, we want to show that the bit streams generated by the uniform generators are uncorrelated. To do this, the correlation coefficients are calculated for the autocorrelation of the I and Q branches and their cross-correlation [31]. The mathematical definition of correlation coefficient is provided in equation 3.13.

$$\rho_{X_1,X_2} = \frac{Cov(X_1,X_2)}{\sqrt{Var(X_1)Var(X_2)}} \tag{3.13}$$

where $Var(X_1)$ is the variance of X_1 and $Cov(X_1,X_2)$ is the covariance of the two variables, X_1 and X_2, defined as:

$$Cov(X_1,X_2) = E[(X_1 - E[X_1])(X_2 - E[X_2])] \tag{3.14}$$



(a) (b)

Figure 3.7: The correlation coefficients of the bits on the I and Q branches are close to zero at all points except for where they completely overlap with themselves.

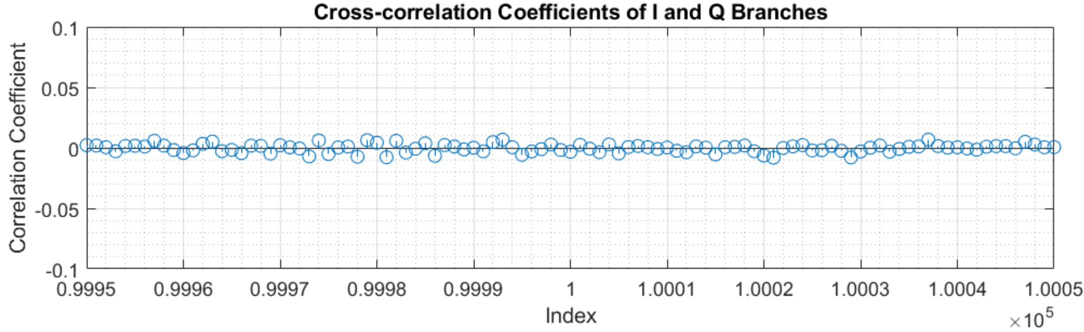


Figure 3.8: The cross-correlation coefficients of the I and Q branches are close to zero at all times, showing that they are uncorrelated.

where $E[X_1]$ is the expectation of X_1. The correlation coefficients are shown in the stem plots in Figures 3.7 and 3.8. The correlation coefficient peaks when the sequence completely overlaps with itself (at 10000 symbols) in the case of the I and Q autocorrelations. The correlation approaches zero otherwise. In the cross-correlation plot, the coefficients remain near zero even when the I and Q sequences overlap. This shows that the I and Q branches are uncorrelated, which is expected since X_1 and X_2 were argued to be independent above, and independence usually implies that they are also uncorrelated.

As mentioned in Section 3.4, the uniform distribution can be used to generate other distributions by implementing the inverse transform method. The result of this transform creates a distribution that is still independent from the other random generators because the

underlying uniform distributions were shown to be independent.

3.7 Gaussian Random Generators

Using the same approach as with the uniform random generators, the correlation coefficients between the two identically distributed AWGN generators (one for each branch) were found to exhibit the same behavior as in the uniform case. Additionally, just as with the uniform generators, the auto-correlation coefficients of the Gaussian generators approach zero except when the sequences completely overlap, and the cross-correlation coefficients always remain near zero.

Next, the Gaussian generators are tested for independence. If the joint distribution of the two Gaussians is equivalent to the product of the two marginal Gaussian distributions, then it can be concluded that the two are independent. Matlab's Kolmogorov-Smirnov (K-S) test function was used to demonstrate that the joint distribution is also Gaussian. The histogram plots of all three Gaussian simulations are plotted against the expected curves in Figures 3.9 and 3.10. Note that the x-axis (and y-axis in the joint case) is scaled by the standard deviations. Since the standard deviation of a Gaussian random variable is assumed to be 1, the axis has the same scaling as the real number line.

3.8 Impact of Bit Precision on Statistical Properties

The impact of bit precision on the performance of the pseudo-random generators must be analyzed in order to make decisions on how much hardware to allocate for each random generator. The performance of the generators is evaluated in terms of how well they fit their respective distributions and remain uncorrelated. For the discrete distributions, such as the

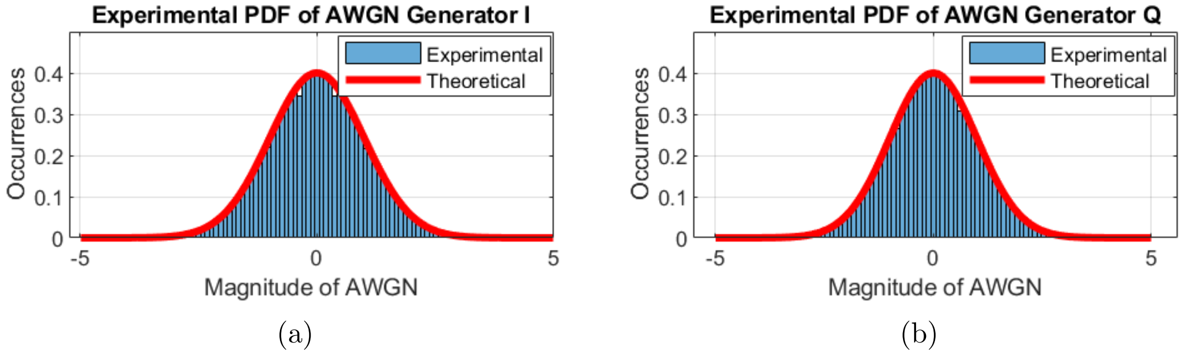


Figure 3.9: The full-precision experimental Gaussian random variable values on both the I and Q branches fit closely to their expected distributions.

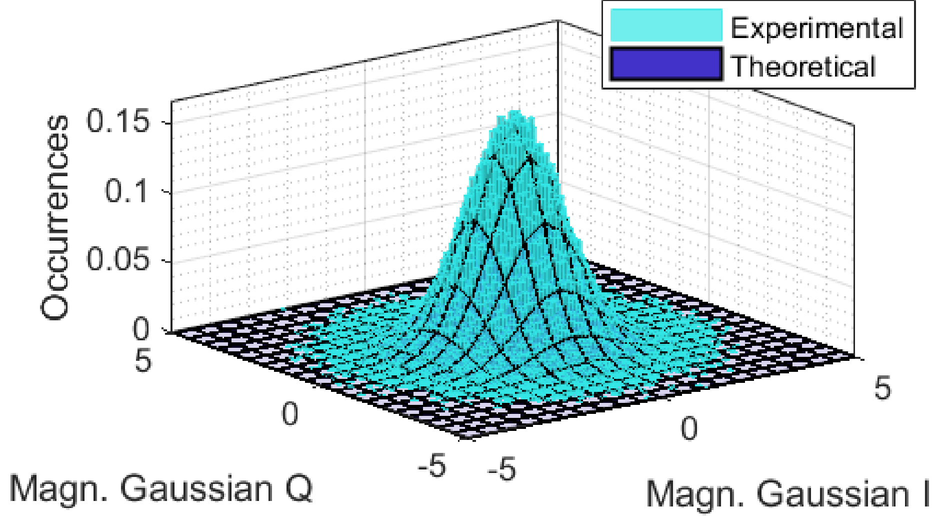


Figure 3.10: Experimental joint Gaussian values are plotted against the theoretical distribution. The two curves nearly overlap, indicating the independence of the two variables.

uniform distribution, the trade-off analysis between bit precision and performance is simple. The trade space lies solely within the argument that using fewer bits reduces the number of values that can be represented. For the uniform distribution between 0 and 255, the bit precision must be equal to eight in order to maintain a full range of values. A reduction in bit precision here would result in fewer choices of frequency (refer to Section 3.3). If four bits are used to select a frequency, then only 16 count value options (four frequency options) are available versus 256 count values (64 frequencies).

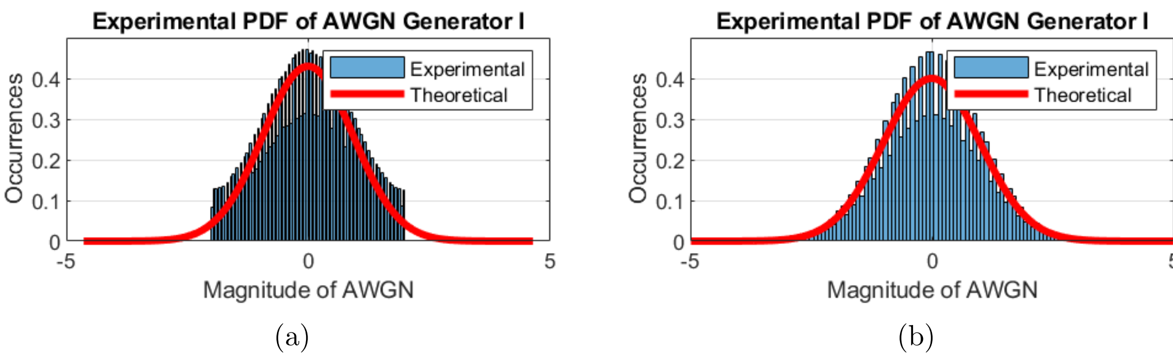


Figure 3.11: When the precision is reduced to 8_6 and 8_5, the experimental Gaussian values do not match the theoretical curve at the tails in (a) and at the peak in both.

The AWGN generators see a greater impact in terms of fit to distribution because the values of the distribution are not integers, and thus must be treated more carefully with respect to bit precision. The default data type of the Gaussian generators is floating point double. However, the system discussed in this paper uses two's complement fixed point notation, which means that the range of values that can be represented is more limited. This limitation varies depending on the number of bits allocated for the integer and fractional portion of the values. In discussion that follows, the impact of using different levels of bit precision is analyzed. For brevity, the number of bits used to represent the fixed point number are written in the format X_Y, where X is the total number of bits and Y is the number of bits used for the fractional portion. For example, an 8-bit number with six bits of fractional precision will be written as 8_6.

The result of using 8_6 fixed point values to represent the Gaussian distribution is shown in Figure 3.11(a). Since the Gaussian distributions of the I and Q branches were shown to be identical, only the distribution of the I branch is shown. The tails of the Gaussian distribution have been cut off due to the use of only two bits to represent the integer portion of the number. The rest of the distribution has been distorted as a result. The distribution resulting from the use of a 16_14 fixed point number is identical to that of using an 8_6 number, meaning

Table 3.2: The results of the Kolmogorov-Smirnov (K-S) test, Anderson-Darling (A-D) test, and Lilliefors tests are shown for each level of bit precision tested in this section. If a test passes, the adjacent column shows the maximum significance level for which it passes. Most of the tests fail for bit precisions less than 10_7.

Prec.	**K-S P/F**	**K-S %**	**A-D PF**	**A-D %**	**Lil P/F**	**Lil %**
6_4	F	-	F	-	F	-
7_4	F	-	F	-	F	-
7_5	F	-	F	-	F	-
8_5	F	-	F	-	F	-
8_6	F	-	F	-	F	-
9_5	F	-	F	-	F	-
9_6	F	-	P	0.1	F	-
10_5	F	-	F	-	F	-
10_6	F	-	P	0.1	F	-
10_7	P	5	P	5	P	3
11_6	F	-	P	0.1	F	-
11_7	P	5	P	5	P	3
11_8	P	5	P	5	P	3
12_7	P	5	P	5	P	3
12_8	P	5	P	5	P	3
16_13	P	5	P	3	P	5
16_14	F	-	F	-	F	-

that the limitation here is in the 2-bit integer precision and not the fractional precision.

Three well-known normal distribution tests are used to statistically check the normality of the Gaussian random variables at different precision levels. The Kolmogorov-Smirnov (K-S) test, the Anderson-Darling (A-D) test, and the Lilliefors test are used with a significance of 5%. The full-precision floating point representation passes these tests. However, as the bit precision is reduced, the normality tests start to fail. The results of these tests are shown in Table 3.2. The result of each test (pass or fail) is shown in columns 2, 4, and 6. If the test passes, the highest significance value that it passes for is shown in the adjacent column.

The tests fail for most bit precision values lower than 10_7, even if the integer portion is greater than three bits wide. This means that the limitation now is the binary point, which

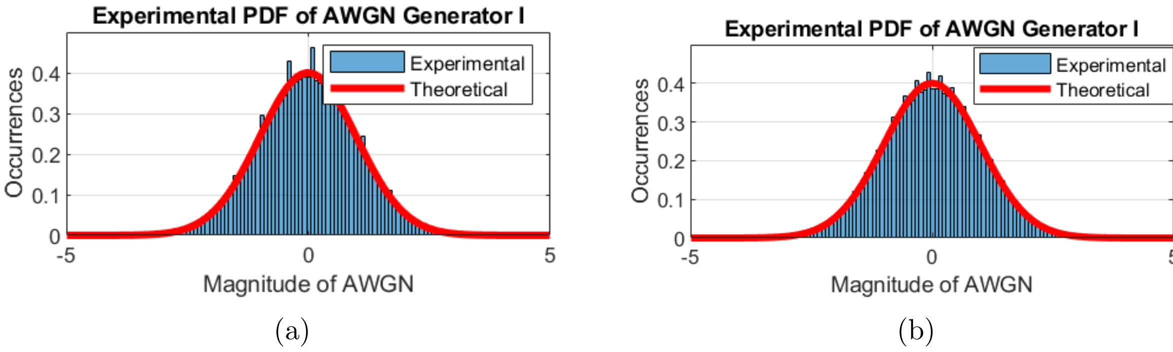


Figure 3.12: When the precision is increased to 9_6, the experimental Gaussian values fit their distribution better than with 8_6 and 8_5. The fit is even better at 10_7.

needs to be at least seven bits wide (the tests also fail for a precision of 11_6, but pass at 11_7, which supports this conclusion). At all higher precisions in the table, all tests passes, except for 16_14. Increasing the integer portion to three bits wide in the 16_13 format causes the tests to pass again. It is safe to conclude that three integer bits and seven fractional bits are required to ensure that the Gaussian samples fit the expected distribution. As mentioned in Section 3.5, [34] claims that the minimum number of bits required to represent the Gaussian distribution with sufficient accuracy is seven. The analysis here agrees with that claim. All formats with more than three integer bits and seven fractional bits pass the tests. The plots for 9_6 and 10_7, provided in Figures 3.12(a) and 3.12(b), show that the experimental curve for 10_7 shows some improvement over that of 9_6 near the center of the distribution. The plots for higher precisions did not show any improvements. Therefore, for this analysis, 10_7 was chosen for the AWGN generators.

In all of the trials of bit precision discussed in this section, it was found that the correlation coefficients exhibit the same behavior as those calculated in Section 3.7. This means that the reduction in bit precision does not affect the correlation of the Gaussian random values.

3.9 Random Signal Generator Output

The waterfall plot in Figure 3.13 shows an example output of the pseudo-random test signal generation framework with signal bursts of different durations, center frequencies, bandwidths, and SNRs. All of the parameters discussed in this chapter can be observed changing randomly throughout the run. The symbol error rate plot in Figure 3.14 shows the theoretical curve for QPSK symbol error rate (green) against the empirical symbol error rate (red) derived from the output of the signal generator generator. As can be observed, the empirical and theoretical curves align well, thus indicating that the signals generated by the developed system are valid. The symbol error rate was tested individually with different center frequencies and filter coefficients to prove that the system will perform as expected when the parameters change during run time.

The test signals generated by this framework are realistic and can have a variety of characteristics that make them useful for testing RF algorithms. However, most RF algorithms cannot be properly tested with just one signal at a time. Most of the real world environments in which these systems are deployed are crowded with many signals in different frequency bands that turn on and off and can vary in bandwidth and SNR. For this reason, the framework developed in this chapter must be extended to allow for the concurrent generation of these signals. A random aggregate spectrum generator that solves this limitation is presented in Chapter 4.

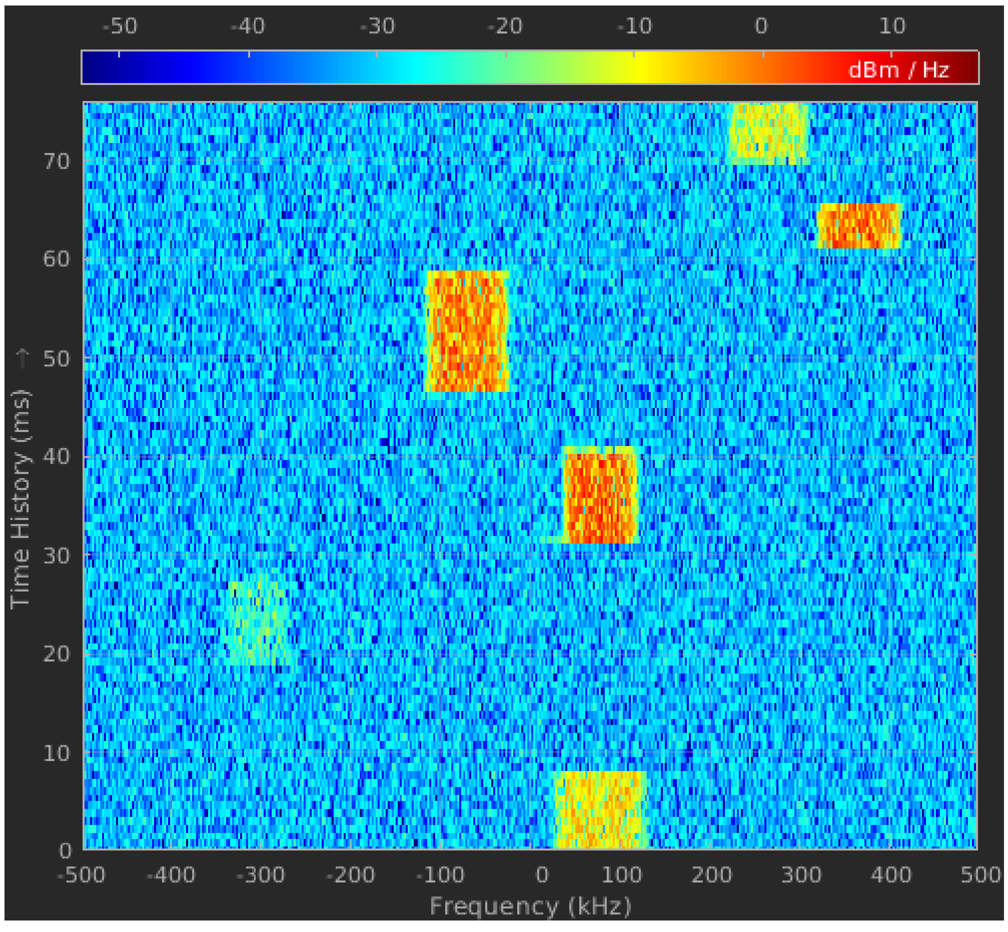


Figure 3.13: A waterfall plot of the random signal generator output shows signal bursts with random durations, frequency, bandwidth, and SNR.

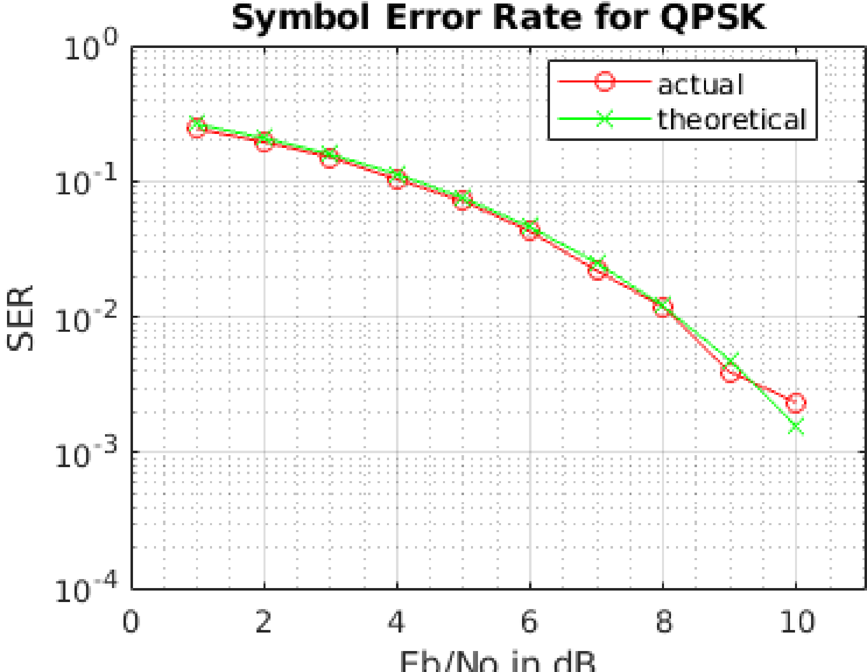


Figure 3.14: The symbol error rate (SER) for the QPSK system simulation is very close to the theoretical QPSK SER curve.

Chapter 4

Random Aggregate Spectrum Generator

In Chapter 3, a novel FPGA-based pseudo-random test signal generator was presented. This system was able to generate realistic signals that pseudo-randomly vary in characteristics such as center frequency, bandwidth, duration, and SNR. However, these individual signals are not sufficient to adequately test RF algorithms, which are deployed in crowded signal environments. In this chapter, the pseudo-random test signal generator developed in Chapter 3 is expanded to create an aggregate spectrum of several test signals. This aggregate of signals is a more realistic representation of a typical deployment environment for an RF system, which could be a Dynamic Spectrum Access (DSA), spectrum sensing, or signal classification system, for example.

The pseudo-random aggregate spectrum generator (shown in Figure 4.1) uses several instances of the blocks described in Sections 3.1 - 3.4 that are summed together in front of the AWGN generator described in Section 3.5. Each signal branch has its own set of filters, modulation LUTs, and ranges of center frequency and SNR. These modular components can

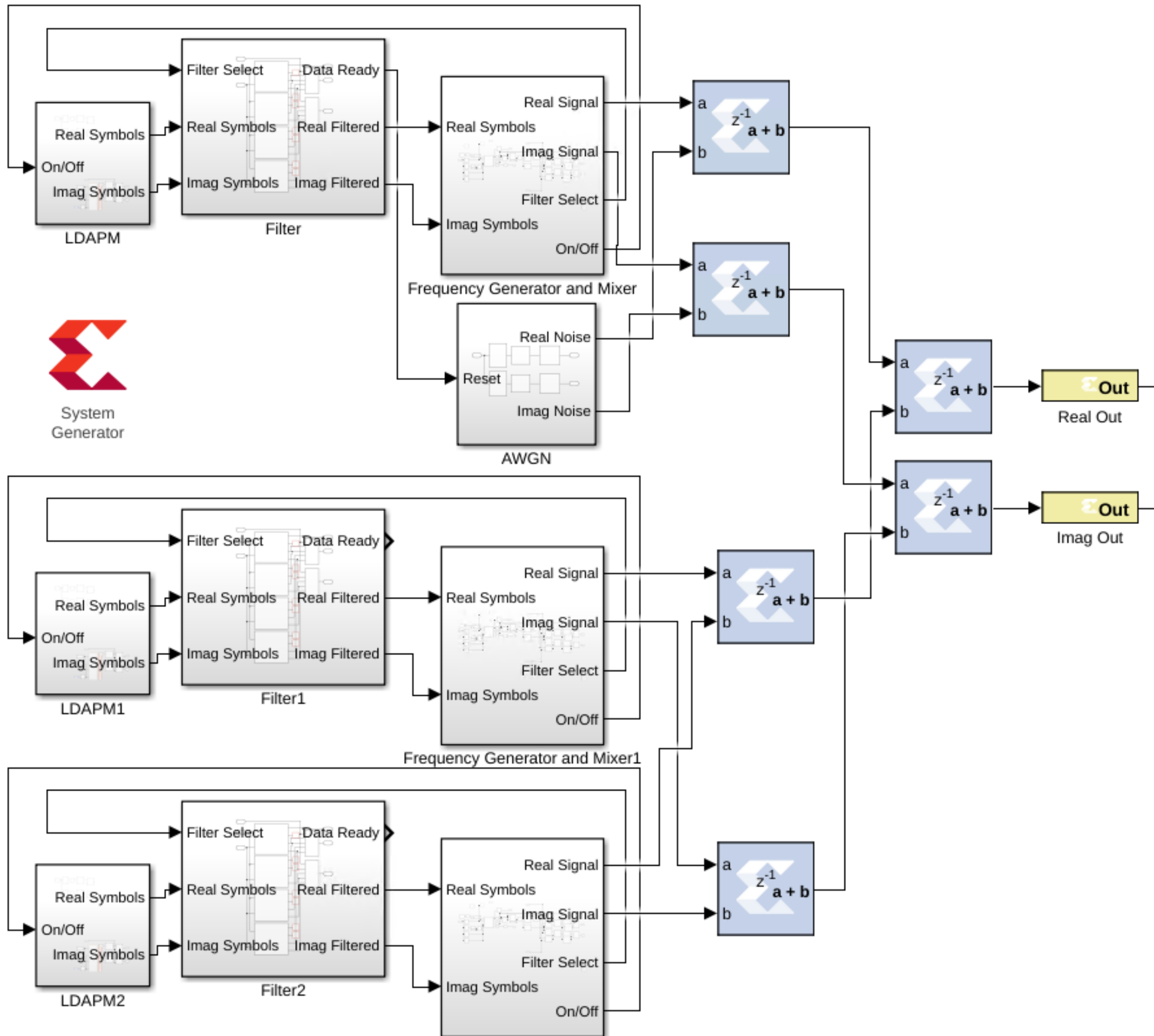


Figure 4.1: The random aggregate spectrum generator is created by summing the output of three random test signal generators and AWGN.

be switched out for ones with different characteristics as desired. For example, if a certain bandwidth or roll-off factor is desired, a new Xilinx FIR Compiler block (or new coefficients) can be swapped into the filter bank to offer the desired filter characteristics. If a different modulation scheme is desired, the LUT containing the current modulation scheme within the LDAPM subsystem needs to be swapped out for the new one.

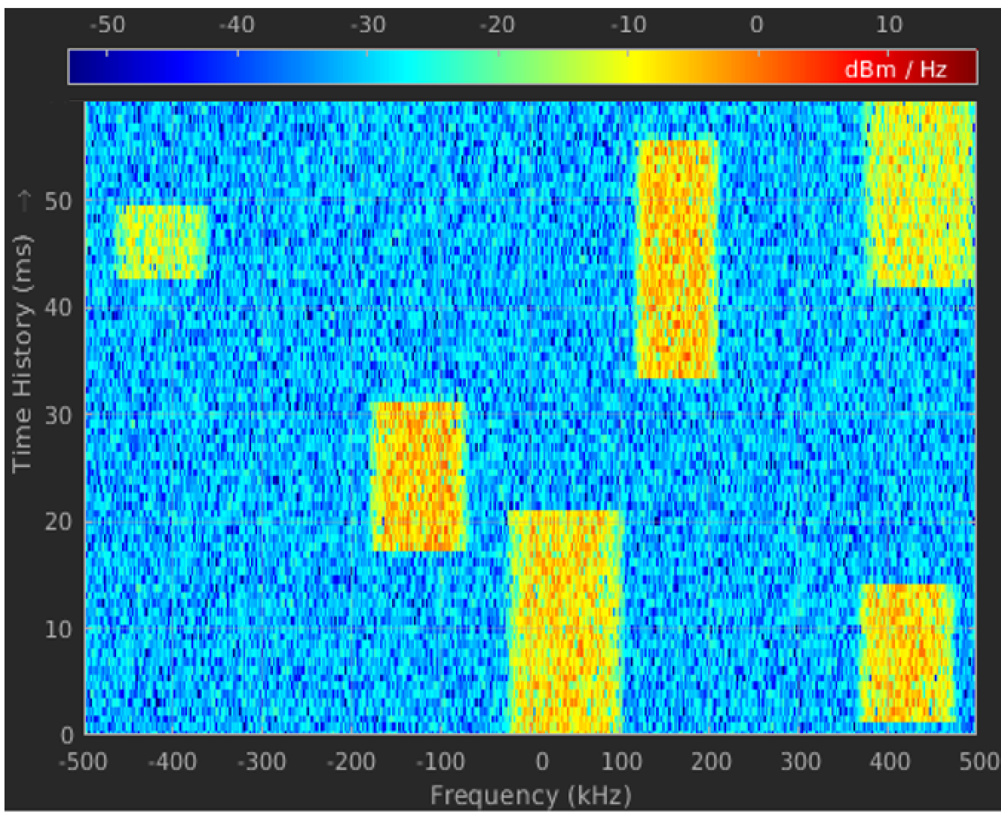


Figure 4.2: A waterfall plot shows an aggregate spectrum of three simultaneous signals created using three instances of the random signal generator from Chapter 3. Each signal's parameters change independently of each other; each signal has a different center frequency, bandwidth, and SNR, and starts and stops at different times.

4.1 Aggregation Issues

Adding several signal branches together causes a few problems to arise. The first is dynamic range. If all of the signals happen to be on at once, the output power increases in magnitude and may not be properly represented when a limited amount of bits is available. If there are only 16 bits available for a signed value, then the range of values that can be represented is -32768 to 32767 (-2^{15} to $2^{15} - 1$), assuming the use of one bit for the sign. This is also assuming that no bits are used for decimal representation. If only one low-power signal is on, or if none of the signals are on, then a very low value needs to be represented. This results in a trade-off between sensitivity (ability to represent low values) and dynamic range

(ability to represent a wide range of values). This relationship is summarized by Equation 4.1:

$$S = \left\lfloor \frac{2^N - 1}{a2^M} \right\rfloor \tag{4.1}$$

where S is the maximum number of signals, N is the number of integer bits (excluding the sign bit), a is the chosen SNR scaling factor, and M is the number bits used to choose an SNR value. In a 16-bit signed system, if eight bits each are used for the integer and fractional portions, then the range of the magnitudes that can be represented is 2^{-8} (or 0.00390625) to $2^7 - 1$ (or 127), assuming that one of the integer bits is used for the sign. As an example, if the range of possible SNRs for each branch is 0.1 to 25.5 ($a = 0.1$, $M = 8$), as described in Section 3.5, then a sum of up to five signals can be properly represented ($\frac{127}{25.5} \approx 5$).

The second issue with using several branches is developing the logic to avoid co-channel interference if desired. Frequency bands are not specified for each signal branch in the aggregate signal generator developed in this thesis. This enables the testing of RF algorithms such as DSA applications, which switch between frequency bands depending on which ones are unoccupied (white spectrum space). However, some communication systems use fixed frequency bands. Others, such as spectrum hopping signals, rely on pattern detection based on frequency. In the future, this system can be modified such that a specific range of frequencies can be selected for each branch if desired. The current version of the developed aggregate signal generator does not allow for this. Defining specific frequency bands for each branch could cause the algorithm under test to detect a pattern between certain bands and the their corresponding signal characteristics. For example, if one signal branch is assigned the frequency band between 10 and 20 MHz and all of the signals transmitted in that band have similar characteristics, then the signal classification algorithm may categorize all of the signals from that branch based on frequency range instead of the characteristics it is

supposed to use to classify the signals. If the algorithm is supposed to guess which modulation scheme is being used to transmit information but each modulation scheme is tied to a certain frequency band, then the algorithm may ignore the phase and amplitude information used for modulation and instead classify based on the frequency band. To prevent this, frequency bands are not specified. However, logic is added to prevent overlapping frequency bands (but this logic can be removed for cases where co-channel signals are permitted, such as for co-channel interference cancellation algorithms). This is done in a similar manner to the frequency limitation logic discussed in Section 3.3, where the LUT address increment values were compared to a range of acceptable values. In the aggregate signals case, the LUT address increment values from all of the branches are compared to ensure that none of them cause co-channel interference. As discussed in Section 3.3, an increment in step size, N, causes an increase in frequency that satisfies Equation 3.1. With a sample rate of 1MHz, each step increment causes an increase in frequency by 3.90625kHz. To prevent co-channel interference, the distance between the center frequencies of two adjacent signals must be greater than or equal to the sum of half of their respective bandwidths. In other words, $f_2 - f_1 \geq \frac{B_1+B_2}{2}$, where f_1 and f_2 are the center frequencies of the signals, and B_1 and B_2 are their corresponding bandwidths. To simplify the logic, the difference between the two center frequencies is compared to the maximum possible bandwidth of the signals, which is 125ksps in the system described in Chapter 3. Since each step increment represents an increase in 3.90625kHz, the difference between the step sizes corresponding to each of the frequencies must be larger than $\frac{125kHz}{3.90625kHz} = 32$.

The spectrum of the aggregate signal generator is shown in Figure 4.2 with up to three distinct signals on at once. The parameters in each of the branches are independent of each other because their underlying distributions were shown to be independent in Chapter 3 . As a result, each signal can be seen turning on and off at different times and can be seen

having different center frequencies, bandwidths, and SNRs (indicated by different colors). An example of two signals behaving differently can be seen at 40-50ms, where two signals are present at -425kHz, 175kHz, and 425kHz, and turn on and off at different times. It is impossible to tell which signal branch produces which signal since there are no specified frequency bands, as mentioned above, and the same modular components are provided for each branch in this run. As mentioned in previous sections, the filters and modulators can easily be swapped out for different ones to produce different signals as desired.

The pseudo-random aggregate spectrum generator shown in Figure 4.1 only has three signal branches, but more branches can be added as desired. As the number of branches increases, the total hardware utilization of the system starts to become an issue. A more detailed discussion of the system's hardware utilization is presented in Section 4.3.

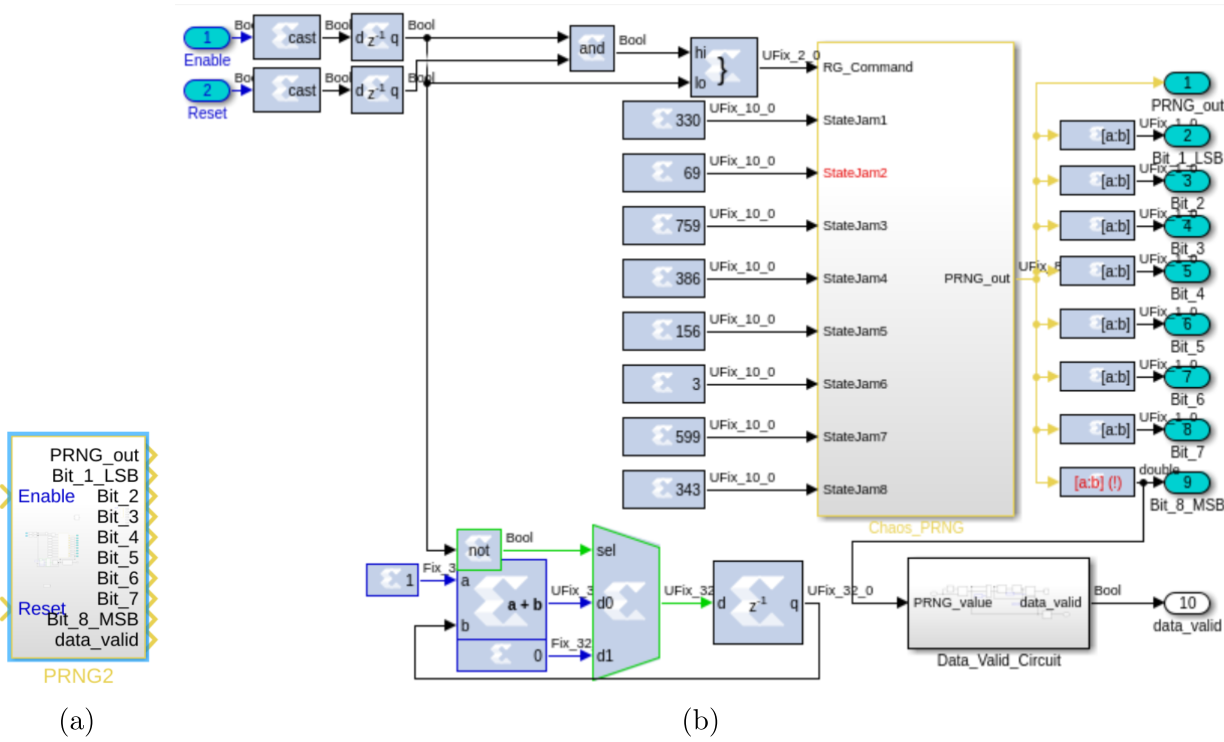
4.2 Periodicity

As more signal branches are added, more random data is also required in order to serve all of the blocks. The random data sequence must have a period that is long enough so that it does not repeat in order to prevent the system under test from classifying on repeating patterns. The period is defined as the number of bits that are outputted by the random generator before the sequence repeats. The 8-bit leap-forward LFSR-based uniform generators discussed in Section 3.6 have a period of $2^{47} - 1$ bits because the underlying polynomials are irreducible polynomials of order 47 [33][35]. While this seems like a very large number, this depends on the usage of the random generator. If this generator were used for a 100MHz signal, it would provide enough non-repeating data for 1.4×10^6 seconds, or about 389 days. If the generator were to provide data for 100 signals, then there would be enough data to last 3.89 days, assuming that the generator runs at 100 times the signal rate and that a mechanism

is in place so that it can provide bits to all 100 signals during one signal cycle. The random generator and the 100 signal branch inputs must be properly synchronized so that all of the random values are distributed appropriately and no bits or clock cycles are wasted. Without synchronization mechanisms in place, the maximum possible data rate is reduced due to the need for parameter estimation [36]. In addition, the initial conditions can be chosen such that the system's conditional Lyapunov exponents are negative, meaning that the sequence will eventually converge and that two random generators will produce the same behavior if presented with the same inputs (in other words, they will synchronize)[37].

Generally, test signals are not generated for a long enough time for the 3.89 day limit to be a problem. However, for Gaussian noise generators, the periodicity can become a problem because background noise is needed for the entire duration of testing. Strictly speaking, AWGN has infinite bandwidth and is typically represented at very high bandwidths in the testing of communication systems, so it inherently requires a longer sequence of random bits. For instance, using the random generator to produce 1GHz of noise would limit the non-repeating interval of the LFSR to 1.4×10^5 seconds, which is 38.9 days. Any increase in the noise bandwidth would result in a further decrease in period. If one random generator has to provide bits to several different modules, this becomes a severe problem because the sequence may repeat before the test run is over. Any such pattern can result in training bias when testing signal detection systems.

In order to avoid issues with the repetition period, a different pseudo-random number generator (PRNG) is implemented (shown in Figures 4.3 and 4.4). This residue number system-based (RNS-based) PRNG relies on the work done by Michaels in [38], the implementation of which is described as follows. Eight co-prime Galois field values are used to represent the state of the PRNG. The Galois field values are de-rotated and the Chinese Remainder Theorem (CRT) is applied to them. The size of the CRT outputs is then reduced by taking



(a) (b)

Figure 4.3: The PRNG top level block is shown in (a) and the internal diagram is shown in (b). The enable and reset inputs control the PRNG's operation. The eight constant blocks that feed the Chaos_PRNG block can be modified by the user before run time in order to change the seeds.

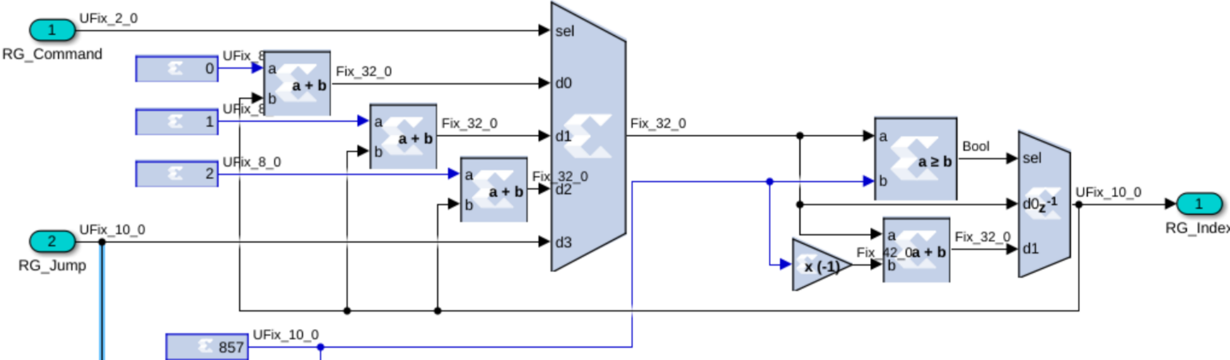


Figure 4.4: The ring generators found inside the Chaos_PRNG block are shown. The ring generator outputs serve as the addresses to the LUTs that contain the prime Galois field residues.

the modulus with 2^8 such that the resulting width is eight bits. The residues are then stored in eight 1024-address LUTs. Eight ring generators provide the addresses for each of these LUTs. The 10-bit addresses in the ring generators are incremented until they reach their corresponding prime values, after which the moduli of the addresses are taken so that they wrap around to zero. The LUT addresses then continue to increment and the process repeats [38]. The outputs of the ring generators provide a new address to the LUTs during each clock cycle, which causes the LUTs to output a different value each time. The outputs of the LUTs are summed together and truncated to eight bits to produce an 8-bit uniform random output. The CRT calculations mentioned above are represented by the following equation:

$$X = (\sum_{n}^{i=1}(M/p_i) \times ((M/p_i)_{p_i}^{-1}x_i)_{p_i}) \mod M \tag{4.2}$$

where X is the remainder or residue, p_i is the prime that the GF is generated on, M is the product of those primes, and x_i is the Galois field element that is being summed [38]. The consecutive reading out and summing of LUT values produces the random sequence.

In Equation 4.2, M also represents the number of possible combinations of LUT outputs. As an example, the prime numbers used in this implementation are $857, 859, 877, 887, 907, 911, 919$, and 929, so $M = 4.0397 \times 10^{23}$. These values were chosen because they are in the upper range of the prime numbers that can be represented using ten bits (which can represent numbers up to 1023). The size of the prime numbers was chosen to be ten bits because it balances the trade-off between repeat interval and bit precision, as will be discussed later on. As described earlier, the ring generators produce LUT address values that turn over after reaching the maximum (a prime value). Since all of the prime numbers are mutually prime, the addresses will not all turn over at the same exact time until the number of iterations reaches the least common multiple (LCM) of the primes. Each combination of addresses is

unique until they all turn over at once. Since the prime numbers used are mutually prime, the product of the primes, M, is also the LCM of the primes. Therefore, M is equivalent to the repeat period of the sequence because the system repeats only after M cycles.

This new repetition period is more than enough for an exhaustive signal testing routine. If this PRNG is used to provide random bits for a 1GHz signal, the sequence will repeat after 4.0397×10^{14} seconds, or 4.6756×10^{9} days, or about 12.81 million years. This seems like overkill, but if a system uses one PRNG for all of its random bits, this time frame quickly decreases. If there are 1000 signals, each with a bandwidth of 1GHz, the period reduces to 12.81 thousand years. If all of the select bits for each random component are drawn from one PRNG, and if the AWGN generator is also sourced from this generator, this number drops even further. The AWGN generator can be implemented by applying the Box-Muller transform to several uniform distributions, as discussed in Chapter 3.5. Assuming that ten uniform distributions are sourced from the PRNG to produce 1GHz of AWGN, the period of the system reduces to 1281 years. This is a much more reasonable number but is still more than plenty for typical signal testing scenarios.

The example hardware utilization report in Figure 4.6 in the next section shows the utilization for both the new PRNG and the LFSR-based uniform random generator (Subsystem: Unif Rand Gen). The PRNG-based approach uses significantly more LUTS (315 vs. 16) and registers (82 vs. 30). However, this is a relatively small price to pay for the increase in repetition period achieved by the PRNG, especially considering that the FPGA has 53200 LUTs an 106400 registers available. However, the PRNG's repeat period is longer than is typically necessary to test an RF system, so steps can be taken to reduce its hardware footprint without significantly compromising the period of the produced sequence. Specifically, the bit width of the prime numbers can be reduced, which also reduces the size of the LUTs that store the rotated Galois field residues. The implementation discussed in this section

uses ten bits, which results in a period of 128.1 years (given the usage scenario described earlier). If this is reduced to eight bits, the period is reduced to 9.54 days (using the prime values 211, 223, 227, 229, 233, 239, 241, 251). This is sufficient for most testing scenarios. A detailed discussion of the system's hardware resources is presented in Section 4.3.

The RNS-based approach described in this section was chosen because of its efficiency benefits. Using this approach, each integer is represented by its remainder from performing the modulus operation, in this case $\mod (256)$. This reduces the number of bits needed to represent any integer to just eight. The use of modular residues allows mathematical operations to be calculated more efficiently because they result in no carry-out bits. This allows parallel computations to occur without relying on the results of other calculations [39].

The structure of the PRNG was selected to match the efficient calculations described earlier. The CRT residues are calculated beforehand and stored in LUTs in the Chaos_PRNG subsystem in Figure 4.3(b). Ring generators, an example of which is shown in Figure 4.4, produce the addresses for each of the LUTs. The ring generator's operation is controlled by the RG_Command input to the mulitplexer. If the selection is 0, then the output of the multiplexer remains constant. If the command is 1, the value increments by 1 each time. If the command is 2, the value increments by 2 each time, which speeds up the rate at which the ring generator values turn over. If the command is 3, then the output of the multiplexer is reset to the State_Jam values (seeds), which are configured in the constant blocks in Figure 4.3(b). The incorporation of these commands gives other subsystems the ability to control the operation of the PRNG such that it provides a value only when it is requested. The PRNG can also be reset to an initial state that is represented by the State_Jam seeds if needed. In the PRNG, a counter circuit and a data_valid circuit also implemented (at the bottom of Figure 4.4) to provide counter values for debug and to provide a notification of when the random output bits are ready to be used, respectively.

Besides producing a sequence with a longer period, the new PRNG has an interface that allows the user to load in new random seeds more easily before run time. The LFSR approach requires instantiating a new 48-bit initial state vector for each new generator, and then changing the corresponding initial state variable names in eight different places three layers deep in the hierarchy. This quickly becomes tedious. The PRNG only has eight seeds that can easily be changed in the constant blocks in the second level of the hierarchy (Figure 4.3). In the future, a software interface could be added to the PRNG to allow for run-time modification of the PRNG's control bits and seeds. The test engineer could benefit from modifying the PRNG's operation during run time. The engineer could command the PRNG to run, stop, or reset. There could be two sets of seeds in the system so that the engineer could modify one bank of seeds via the software interface while the other set is actively being used by the generator. This could be used to extend the repeat period of the produced sequence indefinitely if needed.

4.3 Hardware Utilization

As will be shown in this section, the test signal generator described in Chapter 3 is small enough to fit onto an average FPGA. When several of these signal generators are instantiated in an aggregate test signal generator, the hardware resource utilization of the designs becomes an important consideration. A limited amount of signal branches can fit onto a chip of a given size. Steps can be taken to increase the amount of branches that can fit on the chip at the expense of bit precision, as will be explored in this section.

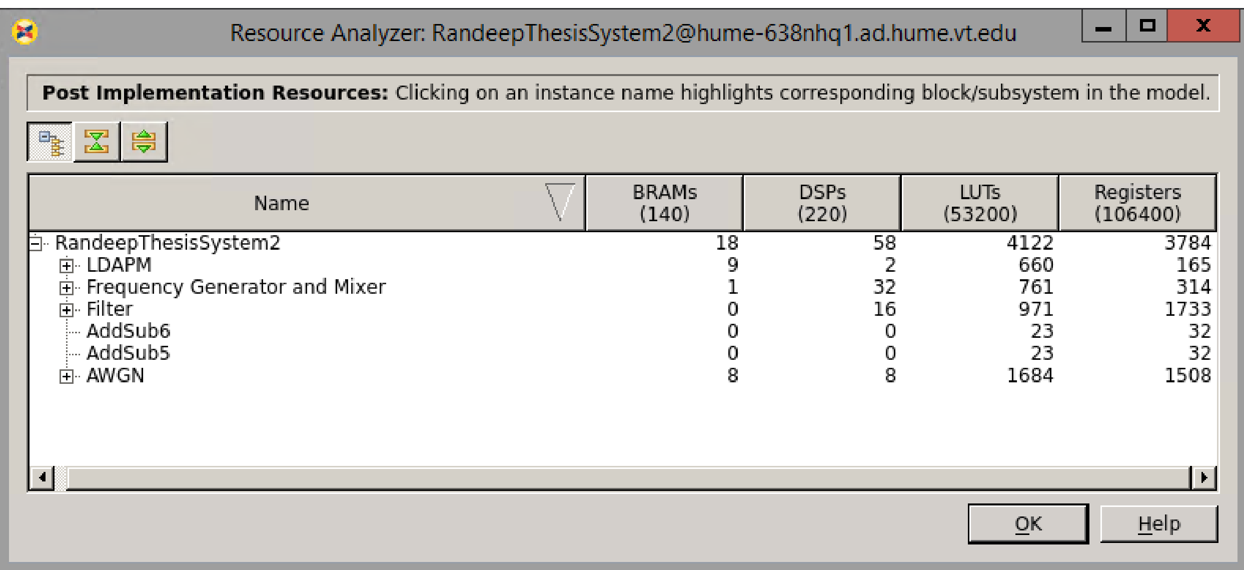
4.3.1 FPGA Resources

An FPGA is a configurable logic device that allows the programmer to reflash it using a binary image file. The binary image file configures the device according to the HDL design from which it was generated. This binary image file is generated after the HDL design is compiled, synthesized, and routed using an FPGA development tool such as Xilinx Vivado. In order to save time, Xilinx System Generator can provide initial hardware utilization reports after performing synthesis but before performing the place and route implementation. This is useful because it allows the engineer to analyze the FPGA resources utilization of the design and make changes without first having to run place and route, which is a time-consuming process.

The hardware utilization report provided by System Generator highlights the usage in four categories: block RAMs (BRAMs), digital signal processing blocks (DSPs), look-up tables (LUTs), and registers, based on a chosen target device. Block RAM is the memory resource in an FPGA and can be used as a single or dual port read and write memory [40]. DSP blocks are units specialized to efficiently perform mathematical operations such as addition and multiplication. These supplement the other logic blocks in the FPGA that are used for mathematical operations [40]. LUTs, which exist in the programmable logic blocks of an FPGA, are used for combinational logic (such as logic gates) and can also be used as additional memory blocks [41]. This can be done by changing the settings of the System Generator block to use distributed memory (DRAM) instead of BRAM. This is useful because most devices have more LUTs than BRAMs. It is important to clarify that unless explicitly stated, the LUTs described in this section refer to the blocks of FPGA resources and not the ROM tables mentioned in the rest of this thesis. The ROM-based LUTs described in the previous sections are actually implemented using BRAMs by default [40].

The target device used for the hardware utilization reports is the Zedboard Zynq Development and Evaluation kit [42]. This device was chosen because it provides an average amount of FPGA resources while remaining within budget for most engineering projects (it costs on the order of 500 USD). This device contains 140 BRAMs, 220 DSPs, 53200 LUTs, and 106400 registers. An example hardware utilization report generated for the system developed in Chapter 3 using this device is shown in Figure 4.5. Notice that the hardware utilization is broken down for each sub-unit of the system. These sub-units can be expanded even further to provide the utilization statistics down to each individual System Generator block (see Figure 4.6). In addition to hardware utilization, System Generator can perform a timing analysis to verify that the design passes timing. If a design passes timing, it means that there is no place in the design that violates the setup and hold time for any logic gate/block. Setup and hold time is the time required to guarantee that a block's logic value is stable. In other words, a design will pass timing if the propagation time between two clocked registers is less than the clock period of the system. If this is violated, the timing report shows where exactly the violations occurred. The timing report also shows the amount of logic delay, routing delay, and timing slack between each logic block. An example timing report is shown in Figure 4.7. As long as each logic route has a positive slack time, the design will pass timing.

In Figures 4.5 and 4.6, the hardware utilization for one signal generator is shown. The system uses the bit precision values described in Chapter 3: 16_12 for the LDAPM and sine wave generators, 32_24 after complex mixing, and 32_24 for the final signal after the addition of AWGN. Eight bits are used for the selection of each random parameter and the size of the LFSRs and PRNGs is held constant (8-bit output). This report shows that the design uses 12.86% of the BRAMs, 26.36% of the DSP blocks, 7.75% of the LUTs, and 3.56% of the registers available in the Zedboard's Zynq 7000 FPGA. The PRNG in the LDAPM uses



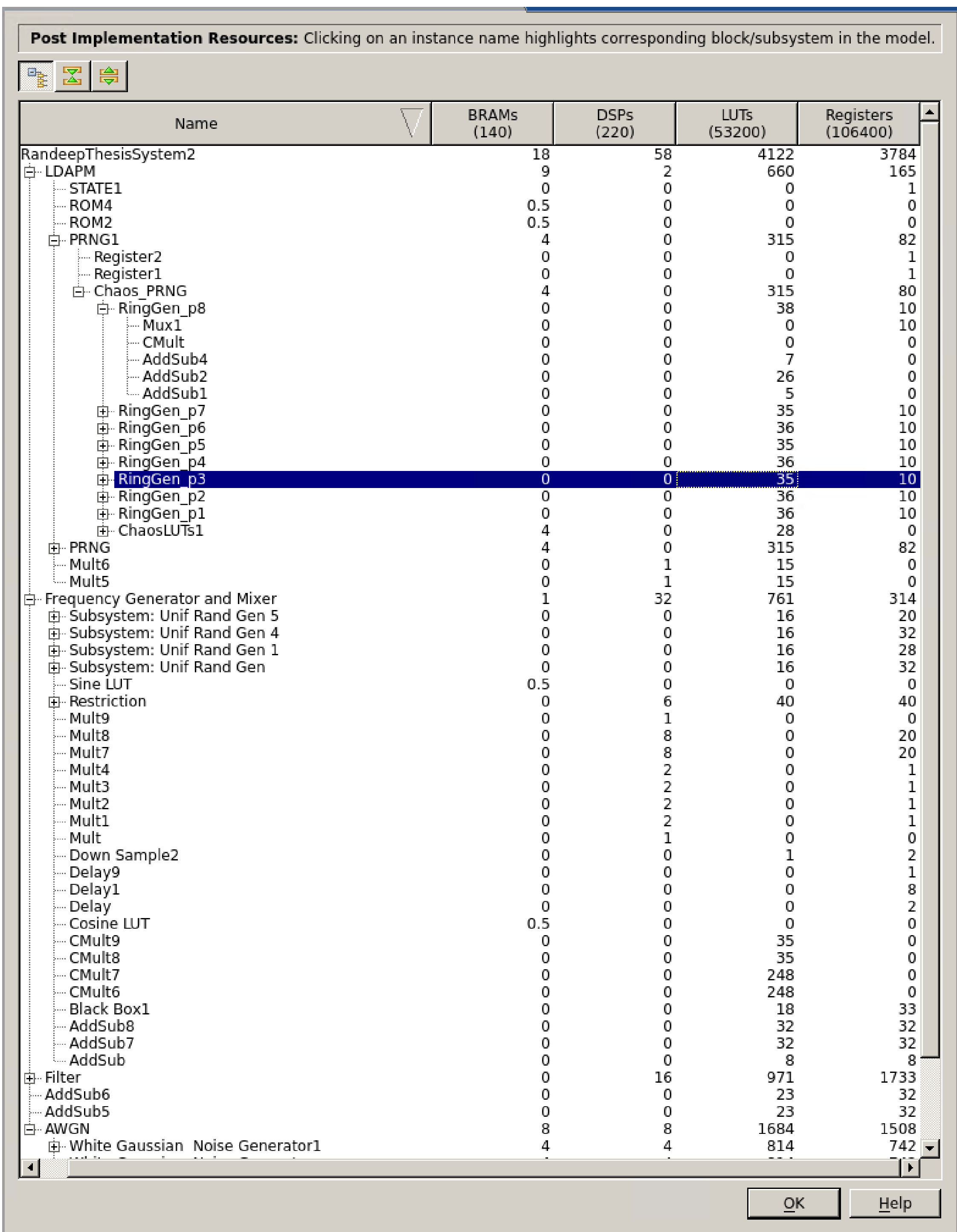
Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
RandeepThesisSystem2	18	58	4122	3784
LDAPM	9	2	660	165
Frequency Generator and Mixer	1	32	761	314
Filter	0	16	971	1733
AddSub6	0	0	23	32
AddSub5	0	0	23	32
AWGN	8	8	1684	1508

Figure 4.5: An example hardware utilization report provides a breakdown of how many hardware resources each subsystem uses.

the most BRAMs of any of the subsystems. Most of the BRAMs are used for the PRNG's ROM-based LUTs that store the prime Galois field residues. The DSPs in the LDAPM are used for the multiplication blocks used for on/off signal gating. Most of the LDAPM's 660 FPGA LUTs are used in the PRNG's ring generators, which use a multiplexer and other combinational logic that is synthesized using FPGA LUTs.

Just like in the LDAPM, most of the DSPs in the Frequency Generator and Mixer are used for multiplication operations. Most of the LUTs are used for multiplications with a constant. This is because constant multiplication results are tabulated in tables similar to multiplication tables that are taught to elementary school students, but with larger numbers. This tabulated method results in the use of fewer logic blocks, especially when multiplying with larger constants. The remaining LUTs are used for addition operations, which are tabulated in a similar way.

The FIR filters in the Filter block each use 4 DSPs for complex multiplication operations and 215-240 LUTs for multiplication with constants (the filter coefficients). The AWGN



Post Implementation Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
RandeepThesisSystem2	18	58	4122	3784
LDAPM	9	2	660	165
STATE1	0	0	0	1
ROM4	0.5	0	0	0
ROM2	0.5	0	0	0
PRNG1	4	0	315	82
Register2	0	0	0	1
Register1	0	0	0	1
Chaos_PRNG	4	0	315	80
RingGen_p8	0	0	38	10
Mux1	0	0	0	10
CMult	0	0	0	0
AddSub4	0	0	7	0
AddSub2	0	0	26	0
AddSub1	0	0	5	0
RingGen_p7	0	0	35	10
RingGen_p6	0	0	36	10
RingGen_p5	0	0	35	10
RingGen_p4	0	0	36	10
RingGen_p3	0	0	35	10
RingGen_p2	0	0	36	10
RingGen_p1	0	0	36	10
ChaosLUTs1	4	0	28	0
PRNG	4	0	315	82
Mult6	0	1	15	0
Mult5	0	1	15	0
Frequency Generator and Mixer	1	32	761	314
Subsystem: Unif Rand Gen 5	0	0	16	20
Subsystem: Unif Rand Gen 4	0	0	16	32
Subsystem: Unif Rand Gen 1	0	0	16	28
Subsystem: Unif Rand Gen	0	0	16	32
Sine LUT	0.5	0	0	0
Restriction	0	6	40	40
Mult9	0	1	0	0
Mult8	0	8	0	20
Mult7	0	8	0	20
Mult4	0	2	0	1
Mult3	0	2	0	1
Mult2	0	2	0	1
Mult1	0	2	0	1
Mult	0	1	0	0
Down Sample2	0	0	1	2
Delay9	0	0	0	1
Delay1	0	0	0	8
Delay	0	0	0	2
Cosine LUT	0.5	0	0	0
CMult9	0	0	35	0
CMult8	0	0	35	0
CMult7	0	0	248	0
CMult6	0	0	248	0
Black Box1	0	0	18	33
AddSub8	0	0	32	32
AddSub7	0	0	32	32
AddSub	0	0	8	8
Filter	0	16	971	1733
AddSub6	0	0	23	32
AddSub5	0	0	23	32
AWGN	8	8	1684	1508
White Gaussian Noise Generator1	4	4	814	742

OK Help

Figure 4.6: A detailed hardware utilization report provides a detailed breakdown of how many hardware resources each System Generator module uses.

Post Implementation Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

Violation type setup Status : PASSED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Dest
1	0.524	9.16	6.445	2.715	2	RandeepT...	Rand
2	0.427	9.251	6.447	2.804	2	RandeepT...	Rand
3	0.403	9.28	6.447	2.833	2	RandeepT...	Rand
4	0.381	9.299	6.447	2.852	2	RandeepT...	Rand
5	0.557	9.278	6.419	2.859	2	RandeepT...	Rand
6	0.547	9.302	6.419	2.883	2	RandeepT...	Rand
7	0.565	9.306	6.419	2.887	2	RandeepT...	Rand
8	0.299	9.536	6.419	3.117	2	RandeepT...	Rand
9	3.107	6.38	0.456	5.924	0	RandeepT...	Rand
10	3.005	6.387	0.456	5.931	0	RandeepT...	Rand
11	2.936	6.456	0.456	6	0	RandeepT...	Rand
12	2.936	6.456	0.456	6	0	RandeepT...	Rand
13	3.031	6.456	0.456	6	0	RandeepT...	Rand
14	3.031	6.456	0.456	6	0	RandeepT...	Rand
15	3.031	6.456	0.456	6	0	RandeepT...	Rand
16	3.031	6.456	0.456	6	0	RandeepT...	Rand
17	2.797	6.595	0.456	6.139	0	RandeepT...	Rand
18	2.733	6.659	0.456	6.203	0	RandeepT...	Rand
19	2.414	6.978	0.456	6.522	0	RandeepT...	Rand
20	2.261	7.131	0.456	6.675	0	RandeepT...	Rand
21	2.356	7.131	0.456	6.675	0	RandeepT...	Rand
22	2.356	7.131	0.456	6.675	0	RandeepT...	Rand
23	2.356	7.131	0.456	6.675	0	RandeepT...	Rand
24	2.356	7.131	0.456	6.675	0	RandeepT...	Rand
25	2.356	7.131	0.456	6.675	0	RandeepT...	Rand
26	2.087	7.305	0.456	6.849	0	RandeepT...	Rand
27	2.087	7.305	0.456	6.849	0	RandeepT...	Rand
28	2.087	7.305	0.456	6.849	0	RandeepT...	Rand

Figure 4.7: An example timing report shows that the system meets timing and shows how much delay and timing slack is present between each module in the system.

generators use 4 BRAMs each for ROMs, 4 DSPs for multipliers, and 814 LUTs for LFSRs and combinational logic in the Box-Muller modules.

Overall, the block RAMs in the FPGA are typically used for ROM-based SysGen LUTs, the DSPs are predominantly used for multiplication operations, the LUTs are used for most other combinational logic (such as addition and constant multiplication), and the registers are used to store values between each of the stages within each block.

4.3.2 Bit Precision vs. Hardware Utilization

In order to determine how many signal branches can be instantiated in the aggregate spectrum generator, the impact of system bit width on the FPGA's hardware utilization must be analyzed. Hardware utilization reports similar to those shown in Section 4.3.1 are generated with the following system bit widths: 8-bit, 16-bit, 24-bit, and 32-bit. Figure 4.8 shows the system's hardware utilization as a function of bit width, split by each category of resource.

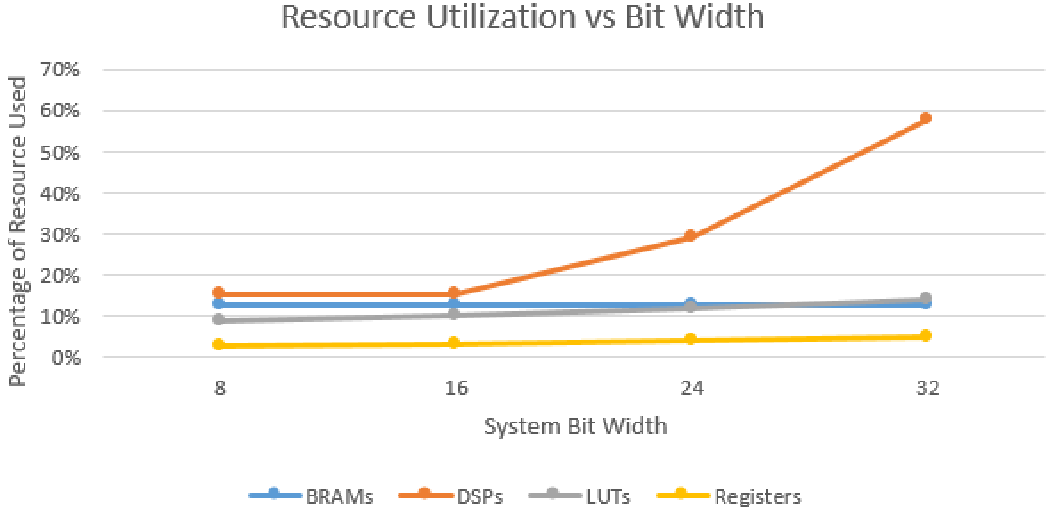


Figure 4.8: The hardware utilization of the aggregate signal generator is shown as a function of system bit width. The utilization of most resources increases linearly, except for DSPs, for which the utilization grows exponentially.

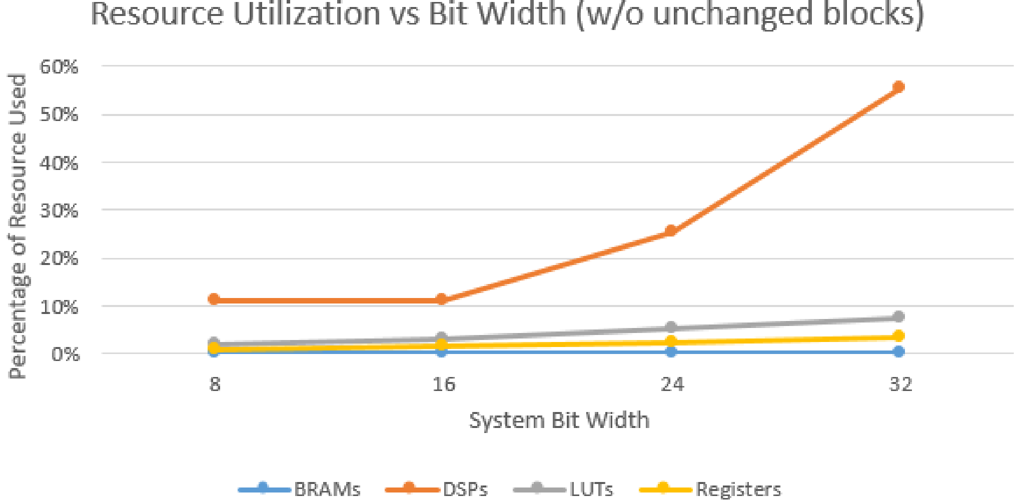


Figure 4.9: The hardware utilization of the aggregate signal generator is shown as a function of system bit width without including resources that do not increase with bit size.

Across each of the trials, the percentage of BRAM utilization stays fairly constant, and the usage of LUTs and registers increases gradually. The usage of DSPs was found to have the greatest rate of increase, especially above 16 bits. It is important to note that in the system, there are a few blocks that do not change in bit width and therefore have the same hardware utilization across all trials. Specifically, the width of the random parameters and the size of the Gaussian random numbers stays constant between the runs. When these blocks are included, the true extent of the impact of bit width on hardware utilization cannot be observed. In order to more clearly show the effect of changing the bit width of the system, a plot similar to 4.8 is produced without including the hardware utilization of blocks for which the usage does not change (see Figure 4.9). In this plot, the effects of changing bit width are more pronounced without the inclusion of the unaffected blocks in the calculations. BRAM usage does not change across the different bit widths, while the usage of LUTs and registers increases slightly with bit width. The DSP usage shows a marked increase, especially after 16 bits.

Upon detailed examination of the hardware utilization reports for each trial, it was found that most of the hardware utilization increase was caused by the adders and multipliers. As the bit width of all of the blocks increases, more LUTs are required for addition and constant multiplication, since the results of the operations are often tabulated in LUTs. In the case of variable multiplication, there is an increase in the use of registers because multiplication typically occurs in several clock stages, each of which requires a register. Most importantly, there is an increase in the usage of DSP blocks because non-constant multiplication occurs almost exclusively in DSP blocks, which are specialized for this operation. The increase in the utilization of DSP blocks is exponential because of the nature of binary multiplication. The output of the multiplication operation is twice as wide as the size of the larger multiplicand. Therefore, the multiplicands in the next stage are twice as wide as the ones in the first stage. This increase compounds at every stage of multiplication, which means twice as many bit-wise multiplications are needed at each stage of calculation. If the multiplication chain starts with two bits, and there are eight stages of multiplication, then $2^8 = 256$ bits are required in the last stage. If the chain starts with four bits, $4^8 = 65536$ bits are required, which is equivalent to 256^2. Doubling the initial bit width results in an exponential increase in bit width at the end of the chain. As bit width increases, more multiply and accumulate (MAC) modules within the DSP blocks are required, so this results in an exponential increase in the utilization of DSP blocks.

If several branches are instantiated, like in Figure 4.1, the hardware utilization increases linearly, except for the additional adder blocks needed to sum the branches and the AWGN block, of which only one is required. The primary limitation here is the DSP blocks (220 DSPs are available in the Zedboard's Zynq 7000 FPGA). A 16-bit system is used for this example. Since each branch requires 26 DSPs without the AWGN generator, it can be calculated that less than nine such branches can be instantiated ($9 \times 26 + 8 = 242 > 220$). In Table 4.1, the

Table 4.1: Hardware resource utilization is shown as a function of number of signal branches. As the number of branches increases, resource utilization also increases linearly.

Num Branches	**BRAMs**	**DSPs**	**LUTs**	**Registers**
1	18	34	5330	3365
3	38	86	12496	7063
5	58	138	19694	10793
7	78	190	26860	14491
8	88	216	30570	16396
9	98	220	34362	18245
Total Available:	140	220	53200	106400

hardware utilization results for a 16-bit multi-branch aggregate signal generator are shown as a function of number of signal branches. When seven branches are instantiated, the DSP usage is 190/220, or about 86% (190 is highlighted in yellow in the SysGen report to indicate that DSP usage is near capacity). The trend from 1 branch onward shows an increase of 26 additional DSPs per branch. The first branch shows 34 DSPs because 8 DSPs are used for the AWGN block. At 8 branches, 216 out of the 220 DSPs are used, and System Generator highlights this number in red. At 9 branches, the report shows that the system uses 220 DSPs, while the use of the other resources continues to increase linearly. According to the DSP utilization calculations and results with using 8 branches, 9 branches should not fit on the chip. However, the hardware utilization display does not exceed the maximum number of DSPs even though more may be required. That is why the trend line for DSPs reaches saturation in Figure 4.10. If the report value was allowed to exceed 220, it would show 234 DSPs are needed. A hardware utilization report that is gathered after running the full implementation of the design would throw an error if the DSP usage exceeds the maximum, but the pre-implementation (post-synthesis) hardware report does not.

In the simulation that generated the data in Figure 4.10, a 16-bit system was specified. Using the data in Figure 4.5 and extrapolating, Table 4.2 was generated to show how many branches can fit onto the Zedboard Zynq at different levels of bit precision. Eight branches

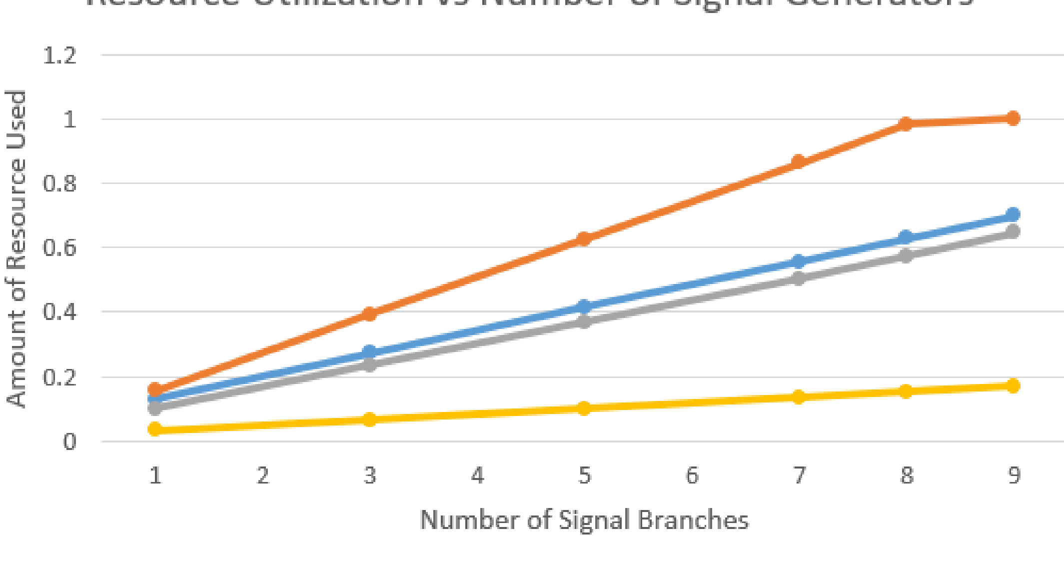


Figure 4.10: Hardware utilization is shown in percentage as a function of the number of signal branches instantiated in the aggregate system. As the number of branches increases, the hardware utilization of the system grows linearly (excluding the additional adders required). The line for DSP saturates at 9 signal branches due to a limitation in the hardware analysis tool.

will fit if an 8-bit or 16-bit system is implemented. Only three 24-bit branches will fit onto the Zedboard Zynq, and only one branch will fit if 32 bits are used. In order to add more signal branches to the system, a larger device, such as a Zynq Ultrascale+ device, will be required.

Following the DSP limitation, the next most limited resource is the BRAMs. For a 16-bit system, BRAMs will be maxed out at 14 branches. However, there is a work-around. In many System Generator blocks, there is an option to use distributed memory instead of BRAM. In an FPGA, memory blocks are implemented using LUTs. LUTs are the next highest used resource, so a suitable balance between BRAMs and LUTs has to be found in order to maximize the number of branches that will fit onto the device. This trade-off only

Table 4.2: DSP usage vs. bit precision and number of branches. The result for 8-bit and 16-bit branches is experimental, and the 24-bit and 32-bit results are extrapolated. Red marks the point where utilization matches or exceeds the Zedboard Zynq FPGA's capacity.

Branches	**8-bit**	**16-bit**	**24-bit**	**32-bit**
1	34	34	64	127
3	86	86	176	365
5	138	138	288	603
7	190	190	400	841
8	216	216	512	1079
9	220	220	624	1317

occurs if the device has sufficient DSPs.

4.3.3 Data Rate vs. Hardware Utilization

The maximum rate of software implementations in the literature was found to be less than 50MHz [6][7]. FPGA-based signal generator implementations were able to exceed a rate of 100MHz [13]. The system developed in this thesis is tested at several system rates to compare it with the prior work. The resulting hardware utilization and timing status is shown in Table 4.3. In the table, (P) indicates when pipelining has been used to increase the data rate.

Table 4.3: Hardware resource utilization is shown as a function of system rate. The amount of hardware required does not increase except for when the design does not meet timing or when pipelined stages are added. In the table, (P) indicates when pipelining has been used to increase the system rate.

Data Rate	**BRAMs**	**DSPs**	**LUTs**	**Registers**	**Timing**
0.1MHz	18	34	5330	3365	PASS
1MHz	18	34	5330	3365	PASS
10MHz	18	34	5330	3365	PASS
100MHz	18	74	6390	6672	FAIL
125MHz (P)	18	34	5372	3475	PASS
154MHz (P)	18	34	5372	3507	PASS
294MHz (P)	18	34	5372	3508	PASS

The hardware utilization of the system stays the same for 0.1MHz to 10MHz, and the system passes timing. When the data rate is increased to 100MHz, the design fails the timing check and the hardware utilization increases. Specifically, more DSPs, LUTs, and registers are used by Xilinx Vivado in order to pipeline the design in an effort to get it to pass timing. Xilinx Vivado automatically attempts to optimize the System Generator blocks based on the trade-off between data rate and hardware resources. Pipelining improves performance by splitting a process, such as multiplication, into smaller stages that require less time to execute. New registers are placed in between each of the new stages in order to hold the value of the current stage until the next clock cycle arrives. This change results in an increase in latency but it allows the FPGA to be clocked at a higher rate because each stage requires less time to execute. This means that it will take more clock cycles to get the first output of the pipelined system, but the subsequent values will be calculated at a faster rate.

ation type setup   Status : FAILED

Slack (ns)	ıs)	ıs)	ıs)	of	Source	Destination
-3.057	1...	7...	5...	11	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/AddSub8	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Mult
-3.057	1...	7...	5...	11	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/AddSub7	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Mult
0.648	8...	6...	2...	2	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Cosin...	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Mult
0.736	8...	6...	2...	2	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Cosin...	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Mult
2.762	7...	3...	3...	13	RandeepThesisSystem2_32bit/AWGN/Down Sample1	RandeepThesisSystem2_32bit/AddSub6
2.762	7...	3...	3...	13	RandeepThesisSystem2_32bit/AWGN/Down Sample	RandeepThesisSystem2_32bit/AddSub5
3.217	6...	4...	1...	1	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/AddSub	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Sine
3.217	6...	4...	1...	1	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/AddSub	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Cosir
3.294	6...	4...	2...	3	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Down ...	RandeepThesisSystem2_32bit/Filter/FIR Compiler 7.2 1
3.294	6...	4...	2...	3	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Down ...	RandeepThesisSystem2_32bit/Filter/FIR Compiler 7.2 1
3.294	6...	4...	2...	3	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Down ...	RandeepThesisSystem2_32bit/Filter/FIR Compiler 7.2 2
3.294	6...	4...	2...	3	RandeepThesisSystem2_32bit/Frequency Generator and Mixer/Down ...	RandeepThesisSystem2_32bit/Filter/FIR Compiler 7.2 2

Figure 4.11: Timing fails at 100MHz data rate. The exact location of the violation is the route between the source and destination (columns 6 and 7). There is significant negative slack, meaning that the system has a significant bottleneck that prevents it from running at a higher rate.

Despite Vivado's attempts to optimize the hardware at 100MHz, the system still fails timing. Figure 4.11 shows that the slack time is insufficient by 3 nanoseconds. This means that the maximum logic delay between two clocked registers is 13 nanoseconds. If the design were

pipelined to ensure that the maximum logic delay is less than 10 ns, the system would pass timing at 100MHz. According to the timing report, the sources of the violation are in the Frequency Generator subsystem between the adders and multipliers that come after the complex mixing operations. System Generator highlights these blocks in the diagram when the corresponding lines in the timing report are examined (see Figure 4.12). In order to pipeline the design, latency is added to the CMult blocks that exist between the adder and multiplier blocks, and this requires 80 additional registers. This causes the design to pass timing with nearly 2 ns of positive slack (fifth row of Table 4.3), which means the rate could be increased to 125MHz. The increase in rate causes a 42-LUT and 30-register increase in hardware utilization in the filter block.

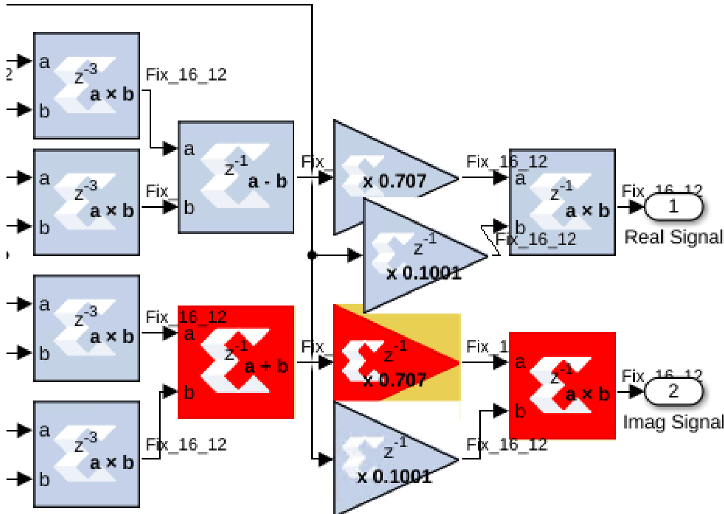


Figure 4.12: Xilinx System Generator highlights the blocks in which the timing violation occurs.

After the CMult blocks are pipelined, the next bottleneck is in the LDAPM. Adding latency to the LDAPM subsystem (in the multiplication blocks) results in a positive slack time of 3.5 ns (max rate is 153.8MHz) and uses 32 additional registers. Adding latency to the downsampling block in the Frequency Generator subsystem further improves this to achieve

6.6 ns of slack (294MHz). At this point, the limiting factor is in Xilinx System Generator blocks such as the AWGN generator and the FIR Compiler. This means that unless these blocks are redesigned (by the use of additional pipelining, for example), the maximum rate that this system can achieve is 294 MHz. This rate is similar to those achieved in the literature. When run at this new rate, the system passes timing with a slack of 0.018 ns, so 294MHz is about as fast as it can be clocked. In the future, efforts can be taken to improve upon the System Generator blocks so that an even higher data rate can be achieved.

4.3.4 Parameter Choices vs. Hardware Utilization

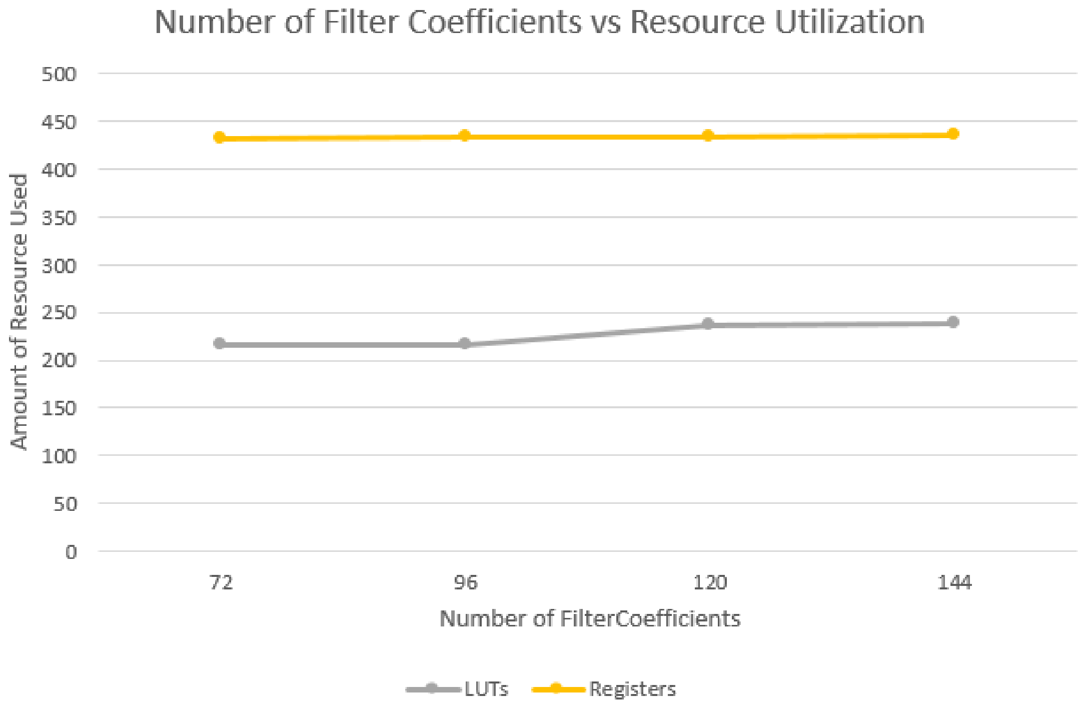


Figure 4.13: As the number of filter coefficients increases, the hardware utilization of the system barely increases.

As discussed in Section 3.2, there is a trade-off between bandwidth options and hardware utilization. Using the method described in Section 3.2, if more filter options are added, then

the hardware utilization increases linearly based on the fact that each FIR filter block uses the same amount of resources (see the hardware utilization report in Figure 4.6). That is why the alternative approach that uses one filter with different interpolation rates is discussed in Section 3.2. There is also a trade-off between hardware usage and the span, or order, of a filter. The higher the order, the better the spectral performance, but also the more coefficients that need to be stored. However, the effect of adding more coefficients is not detrimental. Figure 4.13 shows the trend of LUT and register utilization as the number of coefficients increases. The usage of registers is nearly identical for all four cases, and the number of required LUTs only increases by 21 when going from 96 to 120 coefficients. The other resources' utilization does not change, so they are omitted from the plot. Most of the LUTs are used for convolution and multiplication operations in the filter, so small changes in the number of coefficients does not result in a significant change in the amount of LUTs needed. A larger change (like from 72 to 120 coefficients) requires more LUTs because many more constant multiplication operations need to be performed with the coefficients. This change is not as significant as the impact of using several FIR filter blocks instead of one block with variable interpolation rates.

In addition to hardware trade-offs in the filters, there is also a hardware trade-off with the size of the ROM-based SysGen LUTs (used for the modulation schemes in the LDPAM and the sinusoid generators in the Frequency Generator subsystem). As the size of the ROM increases, the number of frequency options increases and the granularity of the sinusoid improves at the expense of needing additional hardware resources. At low ROM depths, i.e. 256-1024 entries, the ROM uses only half of a BRAM (BRAM usage is shown in half-block increments). The 0.5 BRAM utilization occurs for all bit widths of 64 or less because the BRAMs have a maximum bit width of 72. Using a 4096-address ROM results in the use of 2 BRAMs. The BRAMs in the Zynq FPGA exist as 36-kilobit blocks. Since the ROM

entries are 16 bits wide, a 4906-address ROM results in the use of 78.5 kilobits. Xilinx Vivado therefore estimates the usage of BRAMs to be two blocks. Figure 4.14 shows that there is an exponential increase in hardware utilization as the ROM depth also increases exponentially, meaning that the relationship between ROM depth and hardware utilization is linear (note that the X-axis grows exponentially). When the ROM size grows to 8192 or larger, LUTs and registers start being used to connect the growing number of BRAMs needed to represent a larger ROM.

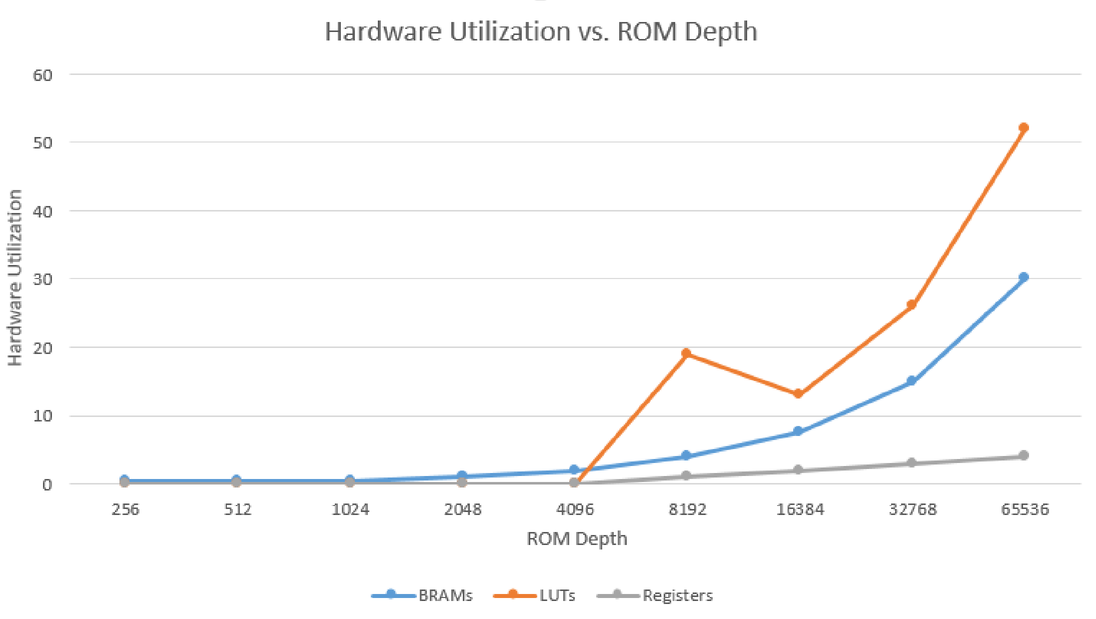


Figure 4.14: As the ROM-depth increases, the hardware utilization of the system increases linearly.

Finally, as discussed in Section 4.1, there is a trade-off between dynamic range and the number of signal branches instantiated in the aggregate spectrum generator. For the multi-branch simulations run in Section 4.3.2, each time a new branch is added, the width of the final output is increased accordingly to prevent problems with dynamic range. This results in an increase in the hardware utilization of the adder blocks. Figure 4.15 shows that while the relationship between register use and number of signal branches is linear, the utilization

curve for LUTs grows more exponentially. This is because when one branch is added, the bit widths of all of the adders down stream has to be increased, and the adders primarily use LUTs. Seeing as LUTs are the third most used resource after BRAMs and DSPs, the designer must be careful to find a balance between dynamic range and hardware utilization. The range of SNR values can be decreased to allow the inclusion of additional signals without requiring additional bit width to represent the aggregate. Reducing the maximum SNR by half results in the ability to include twice as many signal branches in the system (see Equation 4.1).

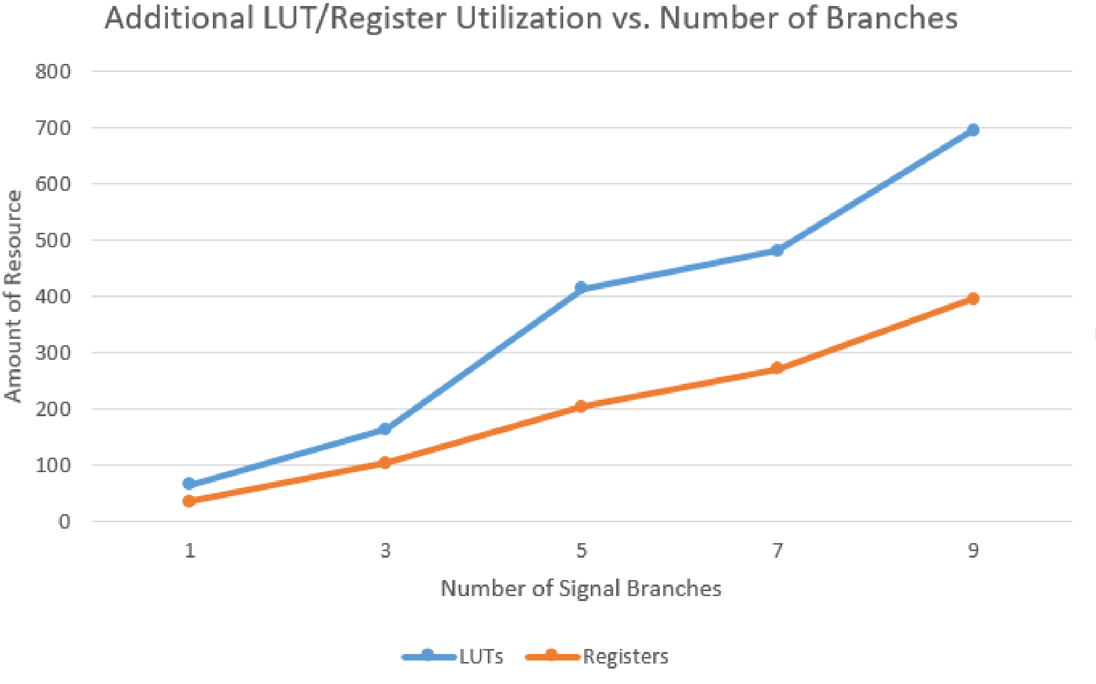


Figure 4.15: Additional hardware resources are used to sum together the multiple branches without sacrificing dynamic range. This utilization increase is plotted as a function of number of signal branches.

4.4 Summary

A novel, FPGA-based random aggregate spectrum generator was presented in this chapter. This generator builds on the work presented in Chapter 3 to provide a multi-signal environment with randomly varying signal parameters to test RF system such as DSA applications, spectrum sensors, and signal classifiers. The combination of several signal branches causes problems with dynamic range and co-channel interference. As more signal branches are added, the maximum possible value of the aggregate output signal increases. This causes an increase in the number bits needed to represent the aggregate signal. The maximum SNR value can be reduced to accommodate more signal branches because there is a trade-off between the maximum SNR value and the number of signals that can be represented. With several signals on at once, there is also a possibility that the signals can overlap, causing co-channel interference. This issue is solved by adding logic that checks if the center frequencies of each branch are too close to one another. In the future, this feature can be removed or modified to accommodate systems that require the use of fixed frequency bands or several hopping frequencies.

In this chapter, a new PRNG was also presented as an improvement on the leap-forward LFSRs discussed in Section 3.6. The repetition period of the LFSRs is not long enough to supply random bits to all of the generator's components for a test run that lasts more than a few days. The new PRNG uses significantly more hardware resources but is able to produce a sequence that repeat after thousands of years (using the scenarios described in this chapter) given the use of 10-bit primes. The period of the PRNG's sequence is reduced to 9.54 days when 8-bit primes are used, and the hardware utilization is also reduced slightly. However, the utilization is still significantly greater than that of the LFSR-based approach. The test signal engineer can use this trade-off between the bit width of the prime values and the hardware footprint of the PRNG to optimize the subsystem according to the needs of

the testing routine and the amount of hardware resources that are available.

The hardware utilization of the aggregate spectrum generator was also analyzed as a function of the number of signal branches instantiated in the system and as well as the bit widths of each of the signal branches. It was found that eight 16-bit signal branches could fit onto the Zedboard's Zynq FPGA. As the bit width is increased, fewer branches are able to fit. The overall hardware utilization of one branch increases exponentially as the bit width is increased due to the limitation of DSP resources. This exponential increase is due to the nature of binary multiplication in hardware. As the size of the multiplicands increases, the use of multipliers, which are implemented using DSPs, increases exponentially. As the number of signal branches increases, the hardware utilization also increases exponentially due to the increase in the amount and size of the adders needed to sum the signals together.

In the discussion about timing, it was shown that pipelining the design increases the maximum rate at which the system can be clocked at the expense of requiring more hardware (especially registers and LUTs). One signal branch of the aggregate spectrum generator can be clocked at 294MHz, which is similar to the rates seen in the literature for the hardware-based solutions. The hardware utilization of filter coefficients was also analyzed to conclude that increasing the number of coefficients does not affect the hardware utilization very much; increasing the number of filter blocks instantiated has a much larger impact. Finally, it was found that as the size of the ROMs used in the system is increased linearly, the hardware utilization also increases linearly.

Chapter 5

Conclusions and Future Work

A novel, FPGA-based pseudo-random aggregate test spectrum generator framework was presented in this work. The spectrum generator was shown to generate realistic wideband signals that can be used to accurately test RF algorithms without requiring expensive in-the-field testing. The system is able to generate signal bursts that have random duration, frequency, SNR, and bandwidth. An aggregate of several randomly-parameterized signals creates a spectral environment that can thoroughly test and evaluate RF systems such as DSA systems and signal classifiers. The use of an FPGA allows the user to make changes to the framework and regenerate the design with ease. The FPGA can also produce higher rates than software solutions.

The random aggregate signal generator's design is modular so that individual components can be swapped out as desired. The LUTs containing the modulation symbols in the LDAPM can easily be replaced by new LUTs containing different symbols. The LUTs containing the sinusoids in the random frequency generator subsystem can also be swapped out in a similar way. The filter and interpolator subsystem contains a bank of FIR filters that can each be swapped out to change the filter's properties (such as bandwidth). The system's PRNG, as

well as the PRNGs' seeds, can be swapped out as desired in order to balance the trade-off between the period of the sequence and the PRNG's hardware utilization. In the future, different channel propagation modules could be implemented in each of the signal branches to represent channel effects such as multi-path and Rayleigh fading. Each branch would have its own unique channel module so that each branch can represent a separate transmitter. The channel effects modules would be interchangeable like the blocks described earlier.

The random parameters in the system were analyzed to ensure independence and statistical fit to their distributions. The impact of bit precision on these performance metrics was also analyzed, showing that at least 10 bits, with 7 bits of decimal precision, are required to properly represent a Gaussian value. The performance of the PRNGs used in this work was analyzed in terms of their repeat period. A PRNG with a much longer repeat period was used as a replacement for the Xilinx leap-forward LFSRs used initially, at the expensive of using more hardware. This trade-off can be fined tuned by changing the widths of the prime Galois fields as desired.

Hardware utilization reports were generated and analyzed to provide insight on the effect of different levels of bit precision on the hardware footprint of the design. Bit precision affects the size of the system, and thus dictates how many modules and signal branches can be added to the system. The system was tested with 8, 16, 24, and 32 bits to show an exponential trend in hardware utilization due to the limitation of DSP resources. It was also shown that as the number of signal branches increases, the hardware utilization increases exponentially due to the need for more adders and additional bit width to properly represent all of the signals. The system was pipelined and tested at different clock rates to find the maximum possible clock rate of the system, 294MHz. With a 294MHz rate, the system can support the generation of many concurrent wideband signals. The analyses performed in this thesis allow the signal engineer to make more intelligent design decisions to balance the inherent

trade-offs between hardware utilization, data rate, and bit precision.

In the future, work can be done to improve the efficiency of the system in terms of hardware utilization and timing. Individual components, such as the Gaussian noise generators and FIR filters, could be redesigned using HDL instead of relying on Xilinx System Generator blocks. These two blocks limit the maximum clock rate of the system to 294MHz. If the blocks are pipelined further, this clock rate can be increased. The FIR filters could also be redesigned in HDL so that one filter can provide many bandwidth options, instead of using a new Xilinx FIR Compiler block for each additional bandwidth option. This can be done by using a filter with a fixed set of coefficients and randomly changing the interpolation rate of the filter to adjust the signal bandwidth during run time. The only limitation of this approach is that the other filter parameters, such as span and roll-off factor, could not be randomized during run time without instantiated additional filter blocks. A combination of this approach and the multiplexed FIR filter bank approach used in this work could alleviate this limitation, where each interpolation filter has different filter parameters.

Another example of improvement would be the use of a channelizer and synthesizer to bring all of the signals to the same aggregate rate instead of using complex mixing, which is a hardware-intensive operation. Channelizers and synthesizers are more hardware efficient because they are implemented using FFTs and IFFTs, which have less redundant hardware than other implementations. A channelizer-synthesizer pair was used in some of the prior work [3], and can be utilized to improve the work presented in this thesis.

It would also be useful to provide a time-stamped report of all the signal parameters that were used during run time. Having this record of ground truth is critically important for the supervised learning of signal classifiers, as an example. In real-world testing, it is difficult to retain an accurate record of ground truth as the simulations are running. This limits the engineer's ability to compare the real signal's characteristics to the signal classifier's

predictions. The test spectrum framework presented in this thesis could be modified so that it outputs a record of which signal parameters are used at any given time in the duration of the run.

Finally, a software interface could be added to the random aggregate spectrum generator to allow the user to modify some system characteristics during run time. The user could, for example, modify a PRNG's seeds or disable a certain module or branch. If desired, the random select bits in the system could be sourced via software so that the user has the ability to choose a specific known value for testing purposes. When the system under test is being debugged, it can be useful to test the system using the same conditions as in a prior run in order to check for expected behavior.

Bibliography

[1] L3Harris Technologies Inc., *RF Waveform Generators*, 2019.

[2] Keysight Technologies, *Wireless Network Emulators*, 2020.

[3] W. H. Clark IV, *gr-signal_exciter*, 2017.

[4] K. Asami, Y. Furukawa, M. Purtell, M. Ueda, K. Watanabe, and T. Watanabe, "WCDMA testing with a baseband IF range AWG," *International Test Conference*, 2002.

[5] R. Beuran, J. Nakata, T. Okada, L. T. Nguyen, Y. Tan, and Y. Shinoda, "A multi-purpose wireless network emulator: QOMET," in *22nd International Conference on Advanced Information Networking and Applications - Workshops*, pp. 223–228, 2008.

[6] L. Xiaodong, S. Yanyan, and L. Shubo, "A MCU-based arbitrary waveform generator for SLH power amplifer using DDS technique," *International Conference on Electronic Measurement and Instruments*, 2007.

[7] N. A. A. M. Nasir, M. N. Mohtar, and N. A. M. Yunus, "Development of signal generator for lab on a chip application," *IEEE ICSIMA*, 2018.

[8] W. Dong, Q. Liu, S. Peng, and H. Li, "Design and realization of arbitrary radar wave-

form generator based on DDS and SOPC technology," *International Conference on Electronic Measurements & Instruments*, 2009.

[9] N. Salerno, L. Simone, S. Cocchi, V. Piloni, M. Maffei, and O. Cocciolillo, "Wideband arbitrary waveform generator for enhanced spaceborne SAR," *European Radar Conference*, 2008.

[10] G. Wang, "An FPGA-based spur-reduced numerically controlled oscillator," *International Conference on System Science and Engineering (ICSSE)*, 2012.

[11] L. Tie-liang and Q. Yu-lin, "An approach to the single-chip arbitrary waveform generator," *International Conference on ASIC*, 2001.

[12] W. Hu, C. L. Lee, and X. Wang, "Arbitrary waveform generator based on direct digital frequency synthesizer," *International Symposium on Electronic Design, Test and Applications*, 2008.

[13] P. Bisiaux, F. Rivet, Y. Veyrac, and Y. Deval, "Experimental demonstration of a 65nm integrated CMOS waveform generator for 5G sub-6GHz standard," *IEEE ICECS*, 2018.

[14] A. M. Sodagar and G. R. Lahiji, "A pipelined ROM-less architecture for sine-output direct digital frequency synthesizers using the second-order parabolic approximation," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, pp. 250–257, 2001.

[15] V. Gautam, K. C. Ray, and P. Haddow, "Hardware efficient design of variable length FFT processor," *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2011.

[16] E. O. Garcia, R. Cumplido, and M. Arias, "Pipelined CORDIC design on FPGA for

a digital sine and cosine waves generator," *International Conference on Electrical and Electronics Engineering*, 2006.

[17] B. R. Jammu, H. K. Botcha, A. V. Sowjanya, and N. Bodasingi, "FPGA implementation of BASK-BFSK-BPSK-DPSK digital modulators using System Generator," *ICCPCT*, 2017.

[18] A. Al Safi and B. Bazuin, "FPGA based implementation of BPSK and QPSK modulators using address reverse accumulators," *IEEE UEMCON*, 2016.

[19] X.-T. Vu, N. A. Duc, and T. A. Vu, "16-QAM transmitter and receiver design based on FPGA," *IEEE International Symposium on Electronic Design, Test & Applications*, 2010.

[20] Z. Shenghua, X. Dazhuan, J. Xueming, Z. Shishan, and Z. Hongrong, "An arbitrary waveform generator for SAR test-bench application," *Asia-Pacific Microwave Conference Proceedings*, 2005.

[21] M. Yeary, R. Fink, D. Beck, M. Burns, and D. Guidry, "A spline function, DSP based mixed-signal arbitrary waveform generator," *IEEE Instrumentation and Measurement Technology Conference*, 2002.

[22] S. Prasad and S. Sanyal, "Design of arbitrary waveform generator based on direct digital synthesis technique using code composer studio platform," *International Symposium on Signals, Circuits, and Systems*, 2007.

[23] Intel Corporation, *FPGA vs. DSP Design Reliability and Maintenance*, 2007.

[24] L. Yueqin and L. Jinping, "Arbitrary waveform generator based on LabVIEW," *International Conference on E-Learning, E-Business, Enterprise Information Systems, and E-Government*, 2009.

[25] G. Liu, Y. Qiu, Z. Yu, and C. Liu, "Design and implementation of software radio based signal generator using LabVIEW," *International Symposium on Systems and Control in Aerospace and Astronautics*, 2006.

[26] A. Grami, *Introduction to Digital Communications.* Elsevier, first ed., 2016.

[27] B. P. Lathi and Z. Ding, *Modern Digital and Analog Communication Systems.* Oxford University Press, fourth ed., 2009.

[28] Xilinx, *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator*, 2017.

[29] G. Xiao-chen and Z. Min-xuan, "Uniform random number generator using leap ahead LFSR architecture," *International Conference on Computing and Communications Security*, 2009.

[30] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing.* MIT Lincoln Laboratory, fourth ed., 2006.

[31] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering.* Pearson Prentice Hall, third ed., 2008.

[32] L. Devroye, *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[33] A. Ghazel, E. Boutillon, J.-L. Danger, G. Gulak, and H. Laamari, "Design and performance analysis of high speed AWGN," *IEEE PACRIM*, 2001.

[34] A. J. Michaels and C. C. Lau, "Quantization effects in digital chaotic communication systems," *IEEE MILCOM*, 2013.

[35] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design.* IEEE, 1990.

[36] G. Kolumbán and M. P. Kennedy, "The role of synchronization in digital communications using chaos–part ii: Chaotic modulation and chaotic synchronization," *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications*, vol. 45, no. 11, pp. 1129–1140, 1998.

[37] T. L. Carroll and L. M. Pecora, "Synchronizing chaotic circuits," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 4, pp. 453–456, 1991.

[38] A. J. Michaels, "A maximal entropy digital circuit," *IEEE ISCAS*, 2011.

[39] C. A. Gayoso, C. González, L. Arnone, M. Rabini, and J. C. Moreira, "Pseudorandom number generator based on the residue number system and its FPGA implementation," *7th Argentine School of Micro-Nanoelectronics, Technology and Applications*, 2013.

[40] G. Krishna and S. Roy, *Advanced Engineering: Fundamentals of FPGA Architecture*. Technical and Scientific Publisher, 2017.

[41] M. M. Mano, C. R. Kime, and T. Martin, *Logic and Computer Design Fundamentals*. Pearson, fifth ed., 2016.

[42] Avnet, Inc., *ZedBoard - Zynq SoC Development Board with FMC LCP/JTAG/UART Interface*, 2020.