

QUANTUM KEY DISTRIBUTION USING FPGAs AND LEDs

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science
in the Graduate school of The Ohio State University

By

Akash Gutha, B. Tech.

Graduate program in Electrical and Computer engineering

The Ohio State University

2020

Thesis committee:

Dr. Daniel J. Gauthier, Advisor

Dr. Tawfiq Musah

Copyright by
Akash Gutha
2020

ABSTRACT

In this thesis, I present an integrated Quantum key distribution (QKD) system designed to be low-weight, small-size and low-power. The system is designed to be stationed on drones, hence enabling the system to be deployed to remote locations, new scenarios where information security is of highest priority.

Previous implementations used bulky time-taggers and high-power laser sources which are impractical for extended periods of operation. Our system integrates the bulky time-tagger into an FPGA and replaces the high-power laser source with low-power resonant cavity LEDs. The resonant cavity LEDs are driven directly by the electrical pulses generated from the FPGA. The FPGAs also handle the transmission and reception processes.

The transmission frequency for the system is capped at 25 MHz. The frequency cap is purely a limitation of the single-photon detectors used. The duration of the wave-packets produced by the LEDs are 10 ns. When mounted on an optical bench, a successful key transmission yielded a Quantum bit error rate (QBER) of 7.5%. This test was done without a spectral filter at the output of the transmitter, and the input of the receiver. The QBER increases significantly when spectral filters are introduced into the system. The spectral filter is important in the creation of indistinguishable light sources in our system. Since having indistinguishable light sources are a key part of QKD security, the system as presented is not fully secure.

Key words: Quantum key distribution, Cryptography, Photons, FPGA, LED, Decoy states

Dedication

Dedicated to my family

Acknowledgments

I would like to thank my advisor, Dr. Daniel J. Gauthier, for giving me this opportunity to work in the field of quantum information under his guidance. I am very grateful for his support, patience, time and assistance during my research. I would also like to thank Dr. Paul G. Kwiat for his support and guidance during my research.

I would also like to thank my master's thesis committee member Dr. Tawifq Musah for his time and advice.

I acknowledge the co-operation and support of my colleagues at The Ohio State University - Daniel Sanchez-Rosales, Roderick Douglas Cochran. I also acknowledge the so-operation and support of students from University of Illinois Urbana-Champaign - Samantha Darlene Issac, Andrew Phillip Conrad.

Vita

May 2016 B. Tech. Electronics and Communication engineering, Delhi, India
August 2018 to Present M.S. Electrical and Computer engineering, Columbus, Ohio ‘

Fields of study

Major Field: Electronics and Computer engineering

TABLE OF CONTENTS

Abstract	ii
Dedication	iii
Acknowledgments	iv
Vita	v
List of Figures	ix
List of Tables	xi
Listings	xii
1 Introduction	1
1.1 Public-Key cryptography	1
1.2 Quantum Key Distribution (QKD)	3
1.3 BB84 protocol	3
1.3.1 Example of a successful key exchange	4
1.4 Decoy state QKD	5
1.5 Eavesdropping and Attacks	6
1.6 Indistinguishable Independent Light Sources	6
1.7 Attributions of the work described in this thesis	7
2 Overview of our QKD system	9
3 Design decisions and Choices	11

4 Transmitter (Quantum source)	13
4.1 Overview of the Transmitter system	13
4.2 Electrical setup	14
4.3 System Components	15
4.4 FPGA transmitter module	17
4.4.1 Clock Generator	18
4.4.2 Buffer Memory	18
4.4.3 Data retriever and serializer (DRS)	20
4.4.4 Status controller and Status register	21
4.4.5 Pulse generator	23
4.4.6 Width and Delay control	24
4.5 LED board for optical pulse generation	26
4.6 Bare-metal Transmitter program	27
4.6.1 Program setup	27
4.6.2 Transmission - State machine	30
4.7 Optical setup	31
4.7.1 Resonant cavity Light emitting diode (RC-LED)	32
4.7.2 Optical enclosure/Coupler (LED to Single-mode fiber)	32
4.7.3 Single-mode fiber and In-line variable attenuator	32
4.7.4 Polarization bench (Optical bench)	34
5 Receiver	36
5.1 Overview of the Receiver system	36
5.2 Optical setup	37
5.2.1 Left arm	37
5.2.2 Right arm	38
5.3 Electrical setup	39
5.3.1 Single Photon detectors	39
5.3.2 Impedance matching transmission line board (SPCM Connector)	41
5.4 FPGA Receiver module	43

5.4.1	Clock generation	44
5.4.2	Pulse contractor	44
5.4.3	16-bit Time counter	45
5.4.4	Time tagger and Memory controller	45
5.4.5	Status controller and Status register	48
5.4.6	Buffer memory	48
5.4.7	Scaling up to multiple channels	49
5.5	Bare-metal Receiver program	51
5.5.1	Program Setup	51
5.5.2	Receiver state machine	54
6	Experimental results	56
6.1	Reception speed test	56
6.2	Transmission speed test	57
6.3	Key exchange experiment	59
7	Conclusions	60
	Bibliography	61

LIST OF FIGURES

Chapter 1

1.1	Public key distribution	1
1.2	Public key encryption	2
1.3	A basic Quantum key distribution system	3

Chapter 2

2.1	Full QKD system overview	9
-----	------------------------------------	---

Chapter 4

4.1	Transmitter hardware stack	14
4.2	Transmitter components	14
4.3	Transmitter electrical setup overview in Block diagram	15
4.4	Transmitter module	17
4.5	PLL generation wizard	18
4.6	On-Chip memory generation wizard	20
4.7	Status generation FSM	23
4.8	Delay control circuit diagram	25
4.9	Width control circuit diagram	26
4.10	LED Board	27
4.11	Transmitter - Finite state machine	30
4.12	Transmitter optical setup	31
4.13	Resonant cavity- Light emitting diode (RC650-TO46FW) made by Roithner LaserTechnik GmbH	32
4.14	In-line variable fiber attenuator by Oz-optics (Custom ordered)	33

4.15 Transmitter Optical bench	34
--	----

Chapter 5

5.1 Overview of Receiver system	36
5.2 Receiver Optical bench	38
5.3 Receiver electrical setup overview in Block diagram	39
5.4 Single photon counting module array (SPCM-AQ4C) made by PerkinElmer	40
5.5 Maximum frequency SPAD output	40
5.6 Impedance matched SPCM connector - Front plane	41
5.7 Female SMA Connector, with 5 terminals	42
5.8 Impedance matched SPCM connector - Back plane	42
5.9 Receiver module	43
5.10 Pulse contractor circuit diagram	44
5.11 Buffer memory configuration for Receiver	49
5.12 Receiver platform design	50
5.13 Receiver program - Finite state machine	54

Chapter 6

6.1 Speed test setup for Receiver	56
6.2 Speed test setup for Transmitter	58

LIST OF TABLES

Chapter 1

1.1	Possible basis of polarization for each bit	4
1.2	Example of a successful BB84 protocol based Quantum key exchange	5

Chapter 4

4.1	Resistor values for LED board	26
-----	---	----

Chapter 5

5.1	Polarization sorting in percentages for the receiver optical bench	39
5.2	Resistor values for SPCM connector	42

Chapter 6

6.1	Receiver speed test results	57
6.2	Transmitter speed test results	58
6.3	Key exchange results using a Pseudo-random bit sequence	59

LISTINGS

4.1	Status update logic - Status controller	21
4.2	Pulse generation logic	23
4.3	Light-weight AXI bridge mapping	27
4.4	Heavy-weight AXI bridge mapping	28
4.5	Base pointer construction for Memory bank	28
4.6	Base pointer construction for Status register	28
4.7	Buffer and File pointer declaration	29
5.1	Pulse contractor RTL code	44
5.2	Time tagger verilog implementation	45
5.3	Time tag and address update RTL code	46
5.4	Write signal generator	47
5.5	Generation of status controller for multiple channels	49
5.6	Memory banks and status registers address mapping to a pointer	51
5.7	Clearing the memory banks	52
6.1	Sifted key from the transmitter	59
6.2	Sifted key from the receiver	59

CHAPTER 1

INTRODUCTION

1.1 PUBLIC-KEY CRYPTOGRAPHY

Public key cryptography, also known as asymmetric cryptography uses pairs of public and private keys to establish and maintain secure connection between a sender and a receiver. The public and private keys are generated by complex mathematical algorithms, e.g. RSA, ENIGMA, DES. Public keys are distributed to encrypt information using one-way functions. Private keys are always kept secret, Information encrypted using the public key can only be decrypted using the private key. A simple execution of public key distribution can be seen in figure 1.1.

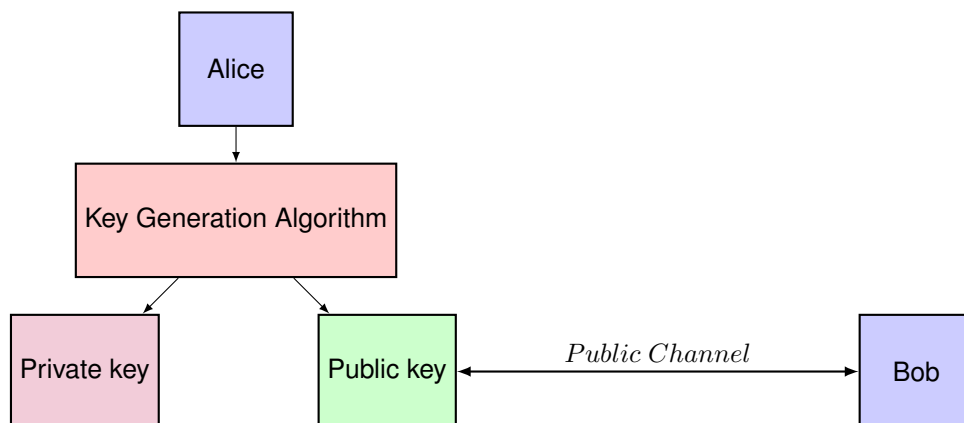


Figure 1.1: Public key distribution

Once the key is distributed, Bob is free to communicate any secret information to Alice if he encrypts it with Alice's public key. Any eavesdropper on the public channel can look at the message but will not be able to make any sense of the message since it's encrypted. The only way to retrieve the original message is to decrypt the message using Alice's private key. This entire process is shown in figure 1.2.

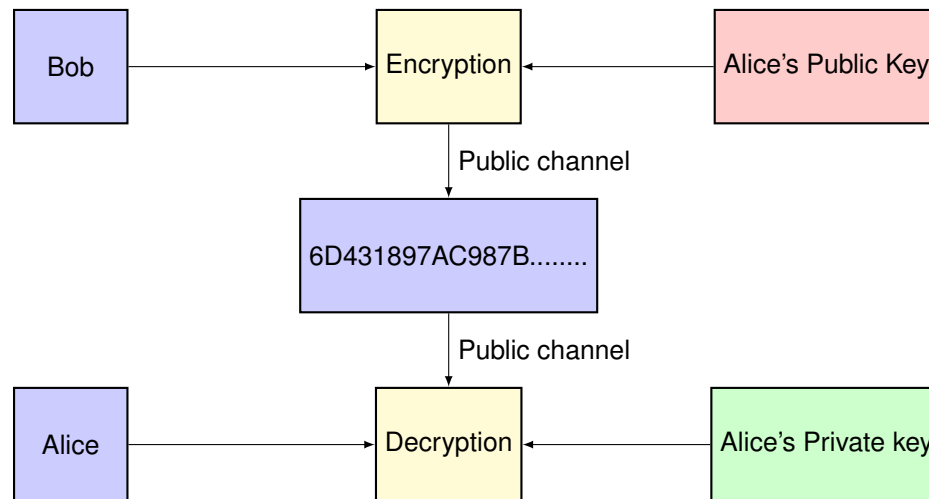


Figure 1.2: Public key encryption

The system may look unbreakable. But the weakest link in the system is access to computation. The most popular algorithm used in today's public key cryptography is RSA [31]. The security of this algorithm is based on the Prime factorization of the product of two large prime numbers. With today's computers it will take hundreds of years to find the private, given that the eavesdropper already has the public key. This is a very long time for the information to be valuable. In reality, the system is still breakable. The time to break is enormously huge which makes the current systems viable.

However with the advent of quantum computers, the basis for this security is shaken [7]. Shor's algorithm [3] can compute the prime factorization in couple of minutes if enough qubits are provided. Today's quantum computers are far away from breaking any cryptography, but it can be only a matter of time before we can make quantum computers with many qubits.

1.2 QUANTUM KEY DISTRIBUTION (QKD)

Quantum key distribution establishes security by distributing the key over a quantum channel [12], which in principle is impossible to be eavesdropped on. This is a fundamental shift from the way public-key cryptography is implemented. There is only a single private key distributed between Alice and Bob. Once a secure key is established and agreed upon, Alice and Bob can communicate on the classical channel using the established key.

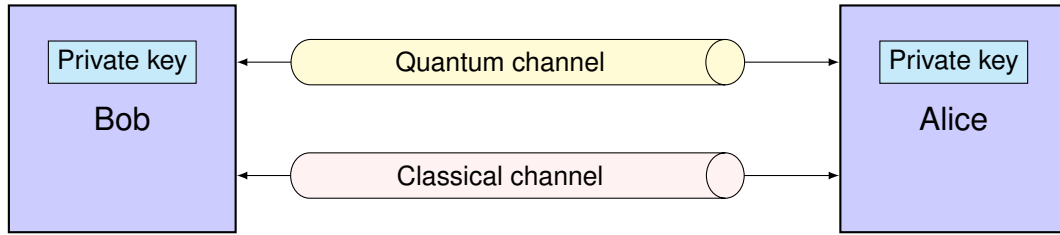


Figure 1.3: A basic Quantum key distribution system

The important aspects of QKD are the quantum properties in a quantum system. By encoding information into quantum properties such as superposition and entanglement, we can detect any third-party eavesdropper. To eavesdrop on the quantum channel, the eavesdropper must perform measurements which will disturb the quantum system, introducing detectable anomalies. There are many well-known protocols for QKD. However, we will be discussing only BB84 protocol [12] (Proposed by Bennett and Brassard in 1984).

1.3 BB84 PROTOCOL

BB84 [12] is a well-known protocol for QKD. The protocol uses photon polarization states as its choice of conjugate pairs. This protocol requires we send only one polarized photon per bit. In our system, these polarizations are generated using single photons sourced from LEDs and a combination of polarization optics.

Given a bit-stream, Alice can choose one of any four non-orthogonal states to polarize her photon with. The four states are:

1. Rectilinear basis (Horizontal-Vertical) \oplus

- Horizontal state \leftrightarrow
- Vertical state \updownarrow

2. Circular basis \bigcirc

- Left-circular state \curvearrowright
- Right-circular state \curvearrowleft

We have four states to choose from and two bits to encode. So we can assign two basis to each bit as shown in table 1.1

Basis/Bit	0	1
Rectilinear basis \oplus	\updownarrow	\leftrightarrow
Circular basis \bigcirc	\curvearrowright	\curvearrowleft

Table 1.1: Possible basis of polarization for each bit

The following steps describes the protocol procedure:

1. Alice chooses a random polarization basis to send her bits.
2. Independent of Alice's choice, Bob chooses a random basis for measurement.
3. Bob completes measurement and ends up with the raw key. This key is very different from Alice's key because of Bob's random choices of measurement basis and photon loss in the quantum channel.
4. Bob sends Alice his choice of basis for all the received bits, without revealing the actual value of measurement.
5. Alice tells Bob the correct choices of basis measurement. Both Alice and Bob will discard the measurement results from different basis choices. This process is called sifting. The key obtained after this is called sifted key. The sifted key should be about 50% of the raw key.

1.3.1 EXAMPLE OF A SUCCESSFUL KEY EXCHANGE

The following example in table 1.2 illustrates the key exchange process using BB84 protocol. Alice sends the bit sequence 01110 encoded as $\curvearrowright \leftrightarrow \curvearrowleft \leftrightarrow \updownarrow$.

Alice's polarization states	\circlearrowleft	\leftrightarrow	\circlearrowright	\leftrightarrow	\updownarrow
Bob's random measurement basis	\oplus		\circ		\oplus
Result of Bob's measurement (with channel loss)	\updownarrow		\circlearrowleft		\updownarrow
Bob's bit sequence (Raw key)	0		1		0
Bob sends Alice his measurement basis	\oplus		\circ		\oplus
Alice tells Bob the correctly chosen basis			✓		✓
Extracted key (Sifted key)			1		0

Table 1.2: Example of a successful BB84 protocol based Quantum key exchange

In practice, we don't have ideal sources and detectors. The imperfections in these sources and detectors show up as errors in the sifted key. The error rate calculated after sifting is called quantum bit error rate (QBER).

1.4 DECOY STATE QKD

Decoy state Quantum key distribution [6] is an extension to the BB84 [12] protocol. The BB84 protocol is based on the assumption that the transmitter uses pure single photon sources. In reality, such devices are bulky and hard to integrate into mobile systems. In our system, the single photon sources are attenuated optical pulses with a mean photon number of 1, which results in some packets containing more than one photon. This will make the channel susceptible to photon number splitting (PNS) attacks.

To combat the PNS attacks, Decoy state protocol uses multiple intensity levels at the source (1 Signal state and several decoy states). The highest intensity level is the signal state and the lower intensity levels are the decoy states. Using decoy states has been proven [6] to increase the performance of QKD systems without any reduction in security guaranteed by QKD.

The system presented in this work is designed to deploy a Decoy state QKD system with 1 signal state and 1 decoy state. In this work, the experimental results presented in chapter 6 are only for a variation of BB84 protocol [22] that uses fewer states (3 states instead of 4) for transmission. There will be less emphasis on the decoy state part of the system.

1.5 EAVESDROPPING AND ATTACKS

From a classical perspective, the easiest way to eavesdrop on a quantum channel required by BB84 1.3 is to copy the state as it is and wait for Alice to publish her choice of basis. However, a single quantum state cannot be cloned without changing its state. This is called the no-cloning theorem [1].

Another way to eavesdrop is to do a photon number splitting (PNS) attack. When there are more than two or more photons in a single packet, Eve can split them and send one photon to the receiver and use the rest to gather information. This can be mitigated using the decoy state technique [6]. This is a source side attack.

Denial of service attack is the easiest attack in these types of systems. In the system presented, the attacker can block the quantum channel with any opaque object to deny line-of-sight between the transmitter and receiver. Since line-of-sight is essential for free-space QKD systems, blocking the channel will deny the establishment of a secure channel.

There are many other such attacks that can be done at the source, at the receiver and at the channel. A plethora of such attacks are presented in [27, 8].

1.6 INDISTINGUISHABLE INDEPENDENT LIGHT SOURCES

For a QKD system to be truly secure the sources should be independent of each other and indistinguishable from each other. From the point of view of a potential adversary, having distinguishable sources can make the system susceptible to side-channel attacks. The concept of indistinguishability was first brought forward in the paper "The Universal Composable Security of Quantum Key Distribution" [5]. The concept of indistinguishable sources in the context of QKD has been used in several other works [14, 13, 9, 8, 15].

There are four properties of our photon sources that can be varied and lead to the possibility of the system being vulnerable to a side-channel attack. Our system uses photon polarization states to encode information. To ensure the indistinguishability of our photon sources we need to keep the other three properties indistinguishable, which are the following.

- Spatial mode
- Spectral mode

- Temporal mode

To ensure these properties stay constant we made a few design choices in the system. Since the transmitter should be independently secure, these design choices are restricted to the transmitter. The design choices will be covered in chapters 3, 4.

1.7 ATTRIBUTIONS OF THE WORK DESCRIBED IN THIS THESIS

The work presented in this thesis is only a part of a larger project to create QKD systems on drones involving 3 students from The Ohio State University (OSU) and 2 students from the University of Illinois (UIUC). The students from OSU working on this project are Akash Gutha (Me), Daniel Sanchez-Rosales, Roderick Douglas Cochran. Students from UIUC are Samantha Darlene Issac and Andrew Phillip Conrad.

The goal is to develop compact QKD systems that can be mounted on drones. There are multiple complex parts in this system, which are designed by different students, and integrated into a single system.

Responsibilities as shared by everyone:

1. Akash Gutha: Design and development of FPGA module and the firmware for the transmitter and the receiver. This has been my primary area of focus in this project. I've also helped Daniel Sanchez-Rosales and Roderick Douglas Cochran in setting-up optical experiments that required FPGAs during the development of their systems.
2. Roderick Douglas Cochran: His primary area of focus in this project is developing the transmitter's polarisation bench and designing the sifting algorithm to extract the secure key. He worked on making the 3D-printed LED enclosures. He also designed the SPAD connector.
3. Daniel Sanchez-Rosales: His work has been an overlap of developing the FPGA module with me and developing the optical setup with Roderick Douglas Cochran. He designed the PCB boards used to mount the LEDs.
4. Samantha Darlene Issac: Her responsibility is to develop the receiver optical bench.
5. Andrew Phillip Conrad: His area of focus are the drones. He is developing the optical-locking systems for the drones.

This is an on-going research and has not been completed fully. However, most parts of the project have been completed and are working as intended independently. My systems and firmware designs have been tested and are working as intended. There is still a lot of space for improvement in the functioning of the system.

In this thesis I will presenting an overall discussion of the system after integration. This will not include detailed discussion of the drones and the sifting algorithm. The sifting algorithm is beyond the scope of this thesis. The Drone part of this system was not functioning as intended at the time of writing this thesis.

CHAPTER 2

OVERVIEW OF OUR QKD SYSTEM

The primary objective is to design and develop a compact low-weight, low-power, mobile and secure QKD. Designing the system to be low-weight and low-power lets us deploy the system on drones, which can be useful for military applications and last-mile communication. Many of the design choices we have made for this project are dependent on these requirements. The system's internal functionality has no dependence on the drones. The drones' part of the project is beyond the scope of this thesis and will not be discussed.

In this chapter, we will go through the high-level details of our system. A very basic highest-level view of the system is shown in figure 2.1.

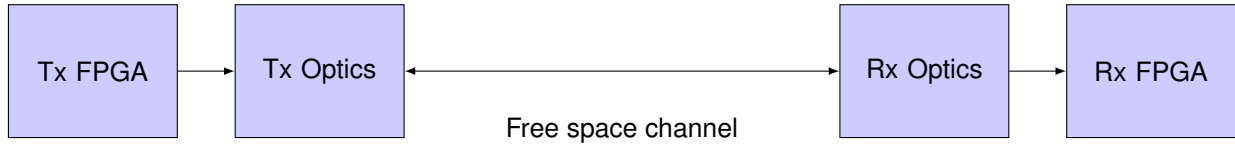


Figure 2.1: Full QKD system overview

The experimental results presented in this work are based on a modified BB84 protocol [21], which uses fewer states (3 states in our experiment) to distribute the key without any significant loss to the secure key rate.

In our system, each state is represented by a photon polarization state. To prepare these states we first generate multi-photon optical pulses using a special type of LEDs called Resonant cavity LEDs. To generate

the optical pulses, the LEDs need to be driven by electrical pulses which are generated from the FPGA (refer 4.4). Since we need only a single photon per packet to combat any PNS attacks, we attenuate the optical pulses using an optical-attenuator to a mean photon number of 1. At this stage, the photon packet is still un-polarized. To polarize the single-photon packet we use a combination of linear polarizers and quarter wave plates, more about this in section 4.7. The polarized states are combined into a single beam using a combination of beam splitters and polarizing beam splitters, this will also be expanded further in section 4.7. The polarized photon is then sent into a free-space channel (free-space generally means air, vacuum, outer space or something similar) to be captured by the receiver.

From the discussion in 1.3, we need to perform a measurement in random basis on the received photon packets. These random measurements are performed by an optical bench, which will be covered in 5.2. Each measurement is captured as a photon-detection event using a single photon detector, more about single photon detectors in 5.4. Each photon detection event generates an electrical TTL pulse, which is read by the FPGA. The free space channel is lossy, every photon packet transmitted doesn't always reach the receiver. To perform the key extraction as shown in 1.2, we need to know the exact time of each received packet. So, the Rx FPGA reads the TTL pulses and time-tags each detection event and stores the data on the SD card. The process of time tagging is explained in detail in 5.9. Now, that we have the raw key stored we can analyze and sift the raw key to extract the secure key and the QBER.

The analysis and QBER extraction will be touched upon in the results chapter 6. The sifting algorithm used to gather our QBER results is outside the scope of discussion for this thesis.

There are four distinct blocks in the system as shown in figure 2.1. We will be discussing each block in detail in the chapters 4 and 5.

CHAPTER 3

DESIGN DECISIONS AND CHOICES

In this chapter, I will discuss some of the important design decision and choices made during the development of various blocks of the system. As discussed in chapter 2, having low power and small sized system was a crucial driver of design decisions. This is usually referred to as having low SWaP (Size, Weight and Power).

1. Why are we using a variant of BB84?

The standard version of BB84 protocol [12] is based on transmitting and receiving 4 distinct states. In the fewer states version we only need to transmit 3 states, but receive 4 states. This translates to having lower light-sources (LEDs), optical components on the transmitter side. This lowers the overall SWaP of the transmitter system. There is also no reduction in secure-key rate achievable by the system [22].

2. Why did we use Resonant cavity LEDs as our light sources?

Resonant cavity LEDs have been known to produce a narrow bandwidth [2, 4] compared to general LEDs. From the discussion in section 1.6, it is a priority to have narrow band-width spectrum to make the sources indistinguishable. Having higher emission intensity [2] when compared to general LED sources is an advantage. Resonant cavity LEDs also have faster reaction times compared to LEDs [2], which lets us operate them at higher frequencies. All the above factors allow us to generate photons in the operating frequency bandwidth with lesser power requirements.

Resonant cavity LEDs have also been successfully used in previous works [16, 25, 10] for Quantum key distribution.

3. Why did we not use laser diodes as our light sources?

From the previous design choice, one might think having a narrower bandwidth is desirable. However, the bandwidth from laser diodes are typically very narrow (much narrower than RC-LEDs) and not the same due to device imperfections. This makes spectrum from each laser distinguishable, leaving the system susceptible to side-channel attacks (refer 1.6). Having a Resonant-cavity LED will allow us to have a narrow bandwidth, yet wide-enough to create indistinguishable sources.

4. Why are we using field programmable gate arrays (FPGAs)?

In the transmitter, as discussed in section 1.6, we need to ensure the optical pulses generated from each LED are indistinguishable from each other. We can design custom circuits that can fine-tune the optical outputs. Since each LED and its output is different from each other we would need to tune each circuit for the respective LED. This task can be performed relatively easily on the FPGA (discussion in 4.4.6) and iterated for changes. The FPGA used in this system is compact in size, weight and uses minimal power, achieving lower SWaP.

In the receiver, as discussed in chapter 2, we need a time-tagger to keep track of the reception time. Commercial time-taggers are usually bulky and are not specifically made to align with our requirements of having low SWaP. Having this system integrated on an FPGA will help us achieve lower SWaP.

CHAPTER 4

TRANSMITTER (QUANTUM SOURCE)

4.1 OVERVIEW OF THE TRANSMITTER SYSTEM

The hardware platform chosen to build the transmitter system is the DE10-Standard development board (fig. 4.2a) developed by Terasic built around the Altera Cyclone V SOC FPGA. The platform packs a two-core CPU and a programmable FPGA. The FPGA also has a high-speed mezzanine card (HSMC) connector. The HSMC connector lets us connect a software defined radio (SDR) to the board. SDR is used for clock synchronization between the transmitter and the receiver. Clock synchronization and SDR will not be discussed in this thesis. The primary reason for choosing the DE10-Standard platform is to have a lower SWaP and SDR capabilities. This lets us build a stacked system as in figure 4.1 that can control and talking to the FPGA system using a bare metal program on the CPU. The system will be expanded further in sections 4.4 and 4.6.

My contribution to the transmitter is development of the FPGA module and the bare-metal program. Discussions with Daniel Sanchez-Rosales have been an important part of the process of developing this system. The LED mounting board was developed by Daniel Sanchez-Rosales. I participated extensively in the thought process of designing these boards with Daniel Sanchez-Rosales and Roderick Douglas Cochran. The optical setup was developed by both Daniel Sanchez-Rosales and Roderick Douglas Cochran.

To connect the LEDs to the FPGA, we designed a custom PCB board that can be mounted with our LED sources. To capture the light emitted from the LEDs we designed an enclosure package that can be 3D printed. The enclosure package can be seen attached to the LED board in figure 4.2b. This enclosure

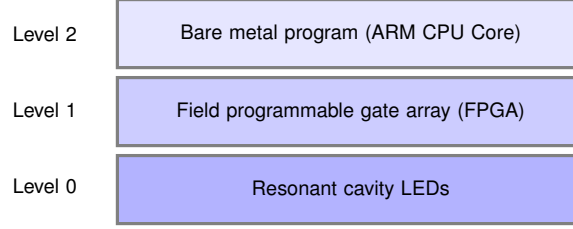
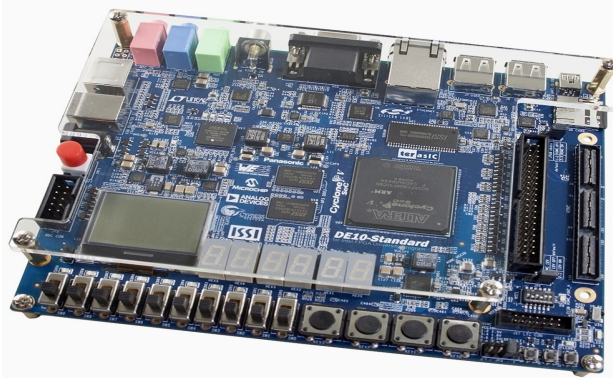
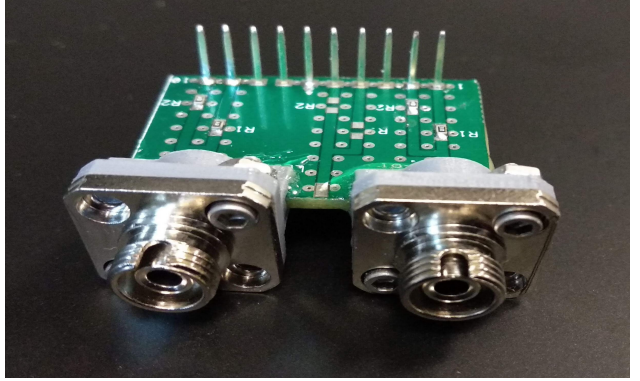


Figure 4.1: Transmitter hardware stack

allows us to plug-in a single mode fiber, ensuring that the core is as close as possible to the light-source. The single-mode fiber has an in-line attenuator. Each optical pulse is attenuated to have a mean photon number closer to 1. Then single photon is then polarized in an optical polarization bench. This polarized photon is launched into free space to be detected by the receiver. The optical setup will be further expanded in the section 4.7.



(a) DE-10 Standard Board



(b) LED mounting board

Figure 4.2: Transmitter components

LED mounting board - A custom PCB board with Pin connectors on one side and LEDs on the other side. The LEDs are soldered to the board and enclosed in 3D printed package

4.2 ELECTRICAL SETUP

There are 2 physical components involved in the electrical setup. The first one is the FPGA board (DE-10 Standard) and the second component is a custom designed PCB board on which the LEDs are mounted. The board on which the LEDs are mounted will be referred to as LED board.

The DE-10 Standard board is packaged with a SD card, an ARM CPU and a FPGA. The SD card is connected to the ARM CPU and it can be accessed by the Linux operating system. The SD card contains the boot section and the storage drive for the Linux system. This gives the CPU programmatic access to the SD card via the Linux file system.

The ARM CPU and the FPGA are connected using an interface called AXI as shown in figure 4.3. The AXI interface is designed for on-chip communication. The interface will be explored further in the section 4.3. The FPGA part of the transmitter will be expanded further in 4.4

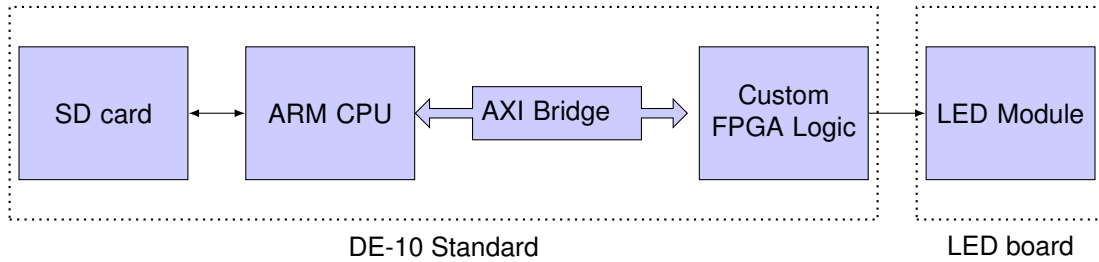


Figure 4.3: Transmitter electrical setup overview in Block diagram

The AXI bridge is used to connect the ARM CPU to SRAM memory banks on the custom FPGA logic. The banks are memory mapped to the CPU and can be accessed via direct memory accesses. The custom FPGA logic also consists of functions that receive the data and do the appropriate pulse generation, pulse shaping and pulse delay. Once the pulse is generated, it is used to drive the LED module which generates the optical pulse. The optical pulse generated at this point is not polarized. The polarization state preparation will be covered in section 4.7.

4.3 SYSTEM COMPONENTS

In this section we will discuss the various interfaces, hardware components involved in the system, and then talk about how the system is controlled by the bare-metal program.

HARD PROCESSOR SYSTEM (HPS)

The ARM CPU in the Cyclone V platform for the DE-10 standard board is a Dual core ARM Cortex-A9 processor. It's a 32-bit processor implementing the ARMv7-A architecture. It is also known as the hard processor system (HPS). The HPS however doesn't include the configurable FPGA. They are independent systems that are connected via two interfaces, namely:

- Heavy weight AXI bridge
- Light weight AXI bridge

Both the interfaces/bridges use the AXI interface as the communication standard. Heavy weight bridge is a wider and faster targeted at large data transfers and frequent data accesses. The light-weight bridge is targeted at slower use cases.

CYCLONE V FPGA

The FPGA included in the DE-10 standard development board is the Altera Cyclone V SE 5CSXFC6D6F31C6N device. The device has a total of 110k Logic Elements. Usually, a logic element in a FPGA is a combination of some-form of combinational logic, a flip-flop and a latch. More advanced implementations of a logic element will include multiplexers, memory blocks. However, as a FPGA designer these details are not necessary to be known in-detail. The programming interface for FPGAs are standard Hardware description languages (HDLs). It is the compiler's duty to figure out how to map the desired logic on-to the logic elements. Knowing the internal details can guide choices or optimizations to your design.

AXI BRIDGE INTERFACE

Advanced extensible interface (AXI) is a parallel multi-master, multi-slave communication interface designed for on-chip communication. It is a subset of an open standard developed by ARM known as Advanced micro-controller bus architecture 3 (AMBA3). The AXI interfaces are the only point of communication between the HPS and the FPGA.

AVALON MEMORY MAPPED INTERFACE

Avalon memory-mapped interface (Avalon-MM) is used to implement read/write interface between master and slaves. This is a simple protocol, yet complex enough to do heavy operations. This flexibility is the reason for Avalon-MM protocol is being used for the buffer memories and status registers in this project.

4.4 FPGA TRANSMITTER MODULE

The FPGA transmitter module receives data from the HPS, which is read and serialized for pulse generation. The generated pulses can be delayed and widened as required to make the output optical pulses generated from each LED indistinguishable (an important property necessary to have secure QKD, refer 1.6). As shown in figure 4.4, Buffer memory is connected to the HPS via the HW AXI bridge, status register is connected to the HPS via the LW AXI bridge. The received data is retrieved and serialized by the DRS block. Status controller is responsible for updating the status register. The serialized data is converted into pulses by a pulse generator. There are a total of 6 pulse paths (3 signal states, 3 decoy states). The 6 pulses are sent through width and delay control blocks. The modified/delayed pulses are used to create the optical pulses in the LED.

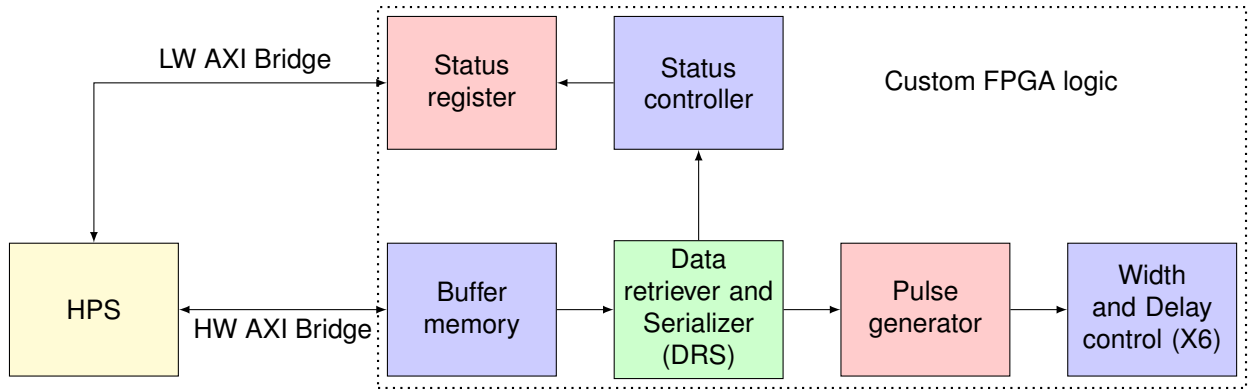
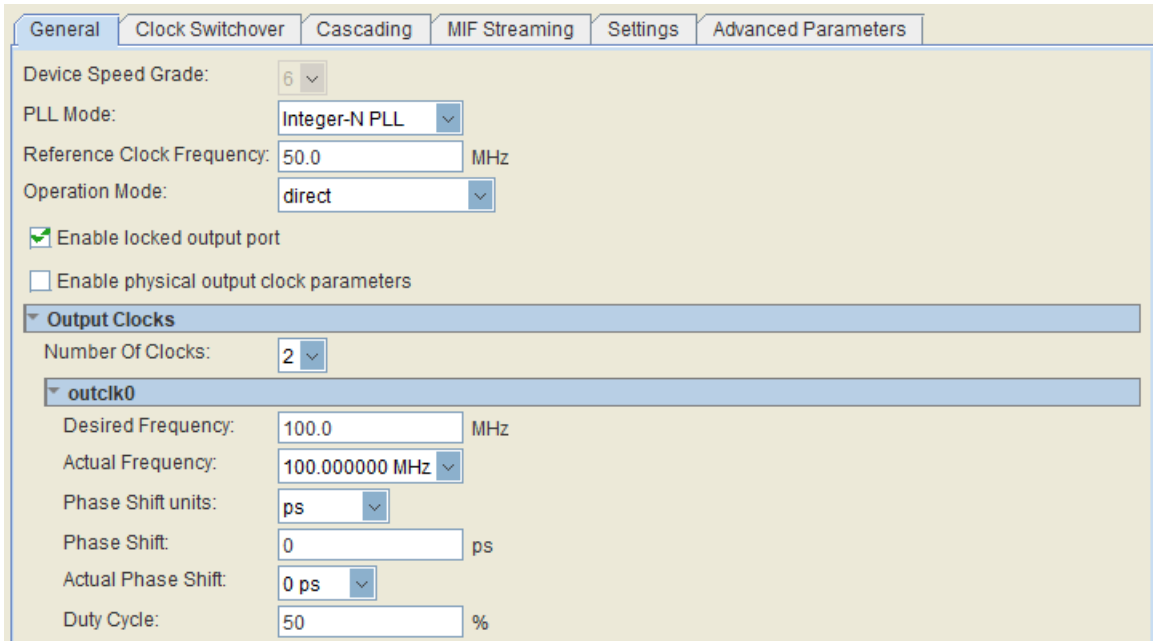


Figure 4.4: Transmitter module

4.4.1 CLOCK GENERATOR

The Board generates four 50 MHz clock inputs externally. These clocks are passed to the FPGA Chip to be used as clock sources for user logic. In our system, we use a 100 MHz clock to run our FPGA logic. It is generated using an Intel PLL IP core [28]. The PLL is configured to use the integer-N Mode. This mode lets us generate clock signals at frequencies that are an integer multiple of the reference frequency and frequencies that are an integer fraction of the reference frequency. In our system, the reference clock is 50 MHz and the output clock is 100 MHz. The configuration wizard is shown in figure 4.5.



The screenshot shows the 'PLL generation wizard' configuration window. It has several tabs: 'General', 'Clock Switchover', 'Cascading', 'MIF Streaming', 'Settings', and 'Advanced Parameters'. The 'General' tab is selected. The configuration includes:

- Device Speed Grade: 6
- PLL Mode: Integer-N PLL
- Reference Clock Frequency: 50.0 MHz
- Operation Mode: direct
- ☒ Enable locked output port
- ☐ Enable physical output clock parameters
- Output Clocks** (expanded section):
 - Number Of Clocks: 2
 - outclk0** (expanded):
 - Desired Frequency: 100.0 MHz
 - Actual Frequency: 100.000000 MHz
 - Phase Shift units: ps
 - Phase Shift: 0 ps
 - Actual Phase Shift: 0 ps
 - Duty Cycle: 50 %

Figure 4.5: PLL generation wizard

4.4.2 BUFFER MEMORY

Reading an SD card at 12.5 MHz using a file system is unreliable. The unreliability is caused due to the operating system, choice of file system and other process running simultaneously on the CPU. SD cards are also not good handling a large amount of small transfers, they are better at handling larger chunks of transfers. This motivates us to introduce a buffer memory in-between the HPS and the FPGA. The HPS reads large chunks of data from the SD card to the buffer memory, while the FPGA can keep serializing the

data from the buffer-memory.

The buffer memory module used to interface with the HPS. The IP chosen to implement the memory buffer is the on-chip memory Intel FPGA IP [29]. The FPGA board also provides us with high speed off-chip RAM which has a capacity of 1 GB. Some part of the 1 GB is reserved for Linux operations, but sufficient memory capacity is provided for our use case. The on-chip memory only provides use a maximum of 512 kB of memory. However, the on-chip memory was preferred because of the following reasons:

- Faster access time than off-chip memory
- Dual port access
- Lower latency

The buffer memory is configured to have dual-port access and operated on a single-clock. Of the two Avalon ports, one port is connected to the HPS via the AXI bridge and the other is connected to the data serializer. The two ports are names S1, S2 respectively. S1 is 64-bits wide, S2 is 32-bits wide.

The choice of 32-bit width for S2 is because of the state machine in DRS (4.4.3). The state machine in DRS is programmed to handle only 32-bits at a time. This is a two-way decision, by the changing the data serializer's capacity we can increase S2's width. 32-bit width is sufficient for our system. The on-chip memory IP puts a restriction on dual-port access, one port cannot be wider than double the width of the smaller port. This restricts our maximum S1 width to 64. Since the S1 port is connected to the HPS via the heavy-weight bridge, having the maximum width lets us push in data faster from the HPS side while being able to read data from S2 port at slower speeds.

Having a smaller buffer defeats the purpose of avoiding reading smaller chunks from the SD card. So, I chose 128 kB as the total buffer memory size.

The buffer memory is further divided into two virtual banks of 64 kB each. This is not a physical distinction. This division is entirely virtual. There will be a status update on the buffer memory status register when the FPGA is finished reading data from a virtual bank. Each bank is reloaded with data from the HPS once the update is received. This lets us have a seamless transfer of data with the help of buffer memory status register.

The buffer memory status register is a PIO (Parallel I/O) core that can be connected to the HPS via the AXI bridge and also be updated from custom logic on the FPGA. This enables status and flag transfers from the FPGA to the HPS with ease. Since the PIO core doesn't need heavy bandwidth, the connection on the

The screenshot shows a configuration window for on-chip memory. It is divided into three sections: Memory type, Size, and Read latency.

- Memory type:**
 - Type: RAM (Writable) (dropdown)
 - ☒ Dual-port access
 - ☒ Single clock operation
 - Read During Write Mode: DONT_CARE (dropdown)
 - Block type: AUTO (dropdown)
 - Note: Tightly Coupled Memory operation require dual port & dual clock sources.
- Size:**
 - ☒ Enable different width for Dual-port access
 - Slave S1 Data width: 64 (dropdown)
 - Slave S2 Data width: 32 (dropdown)
 - Total memory size: 131072 (text input) bytes
 - ☐ Minimize memory block usage (may impact fmax)
- Read latency:**
 - Slave s1 Latency: 1 (dropdown)
 - Slave s2 Latency: 1 (dropdown)

Figure 4.6: On-Chip memory generation wizard

light-weight bridge. This register is updated from the Data serializer and the condition for update will be discussed in 4.4.3.

4.4.3 DATA RETRIEVER AND SERIALIZER (DRS)

As stated in Chapter 2, The data retrieved from the SD card is in parallel format. The 3 signal states and 3 decoy states can be represented in 6 unique characters which require 3 bits of information. In a 32-bit string, 6 unique characters can be stored. So, each 32-bit string is divided into 10 sets of 3-bit data and 2 redundant bits. The DRS is a single block that takes care of retrieving the data and serializing it. There is no external input to this block other than the data bus from buffer memory required to retrieve data.

The DRS state machine is programmed to perform the following actions:

1. Send a read signal and wait for a clock cycle.
2. Read data into a local register.
3. Increments the target address for the next location in the buffer memory.
4. Serialize data 3-bits at a time for the pulse generator. Also, send a generate signal to the pulse generator.

5. After 10 serializations, the state machine will jump to state 1 to retrieve the next piece of data.

The DRS outputs the target address, for the status controller to send updates to the HPS. This will be explained further in section 4.4.4.

4.4.4 STATUS CONTROLLER AND STATUS REGISTER

The status controller takes in the current target address as the input from DRS. It has only one job and that is to indicate the status of the buffer memory, which is the index of virtual bank being serialized. In the following Verilog program, the output state is switched when the target address hits either 0 or the center of the address space. Each output state corresponds to the virtual bank being serialized. The internal status that is being updated is connected to the status register on the AXI bridge. Hence, making the status available for the HPS to poll.

Listing 4.1: Status update logic - Status controller

```
1 module status_controller (
2     input wire clk,
3     input wire rst,
4     input wire [ADDRESS_WIDTH-1:0] i_address,
5     output wire o_status
6 );
7
8 parameter ADDRESS_WIDTH = 20;
9
10 reg int_status = 0;
11 reg [1:0] state;
12
13 assign o_status = int_status;
14
15 always @(posedge clk) begin
16
17     if (~rst) begin
18         state <= 2'b00;
19     end else begin
20         if (i_address == (2**ADDRESS_WIDTH)/2) begin
21             state <= 2'b01;
22         end else if (i_address == 0) begin
23             state <= 2'b01;
24         end
25     end
26 end
```

```

25     end
26
27 end
28
29 always @(posedge clk) begin
30     case (state)
31         2'b00: begin
32             state <= 2'b00;
33             int_status <= 0;
34         end
35         default: begin
36             state <= state + 1;
37             int_status <= 1;
38         end
39     endcase
40 end
41
42 endmodule // status_controller

```

In the code listing 4.1, I present the hardware description code for the status controller. In line 1, the module **status_controller** module is declared. The module **status_controller** has three input ports. The input **clk** is used to feed the clocked registers. The input **rst** is a reset control, used to reset certain registers. The input **i_address** is the input address bus of width **ADDRESS_WIDTH**. The parameter **ADDRESS_WIDTH** is declared in line 8. The module has one output port named **o_status**. The output port is a single bit line used to update the status signal.

In line 10, a register **int_status** is used to store and update the status variable internally. In line 11, a two-bit register is used to store and update the state. In line 13, the internal status register is assigned to the output status port. This will pass any changes made to the internal status register to the output port.

In line 15-27 we declare an always block which will be triggered on the positive edge of the clock. In lines 17,18 we check for reset signal. In lines 20,22 **i_address** represents input address fed from the DRS. we are controlling a variable called state, based on a condition. The condition is **rst** being logic low or the input address being either 0 or half of its maximum allowed value.

In lines 29-40, we declare a positive clock edge triggered state machine which takes the variable state as an input to the case statement and assigns **int_status** to either 0 or 1. In line 31-34 the case for state being equal to 0 is declared. When the state is equal to zero, the **int_status** is set to a value of 0. The default case (declared in lines 35-38) represents all the other cases where the state is not equal to 0, the **int_status** is set

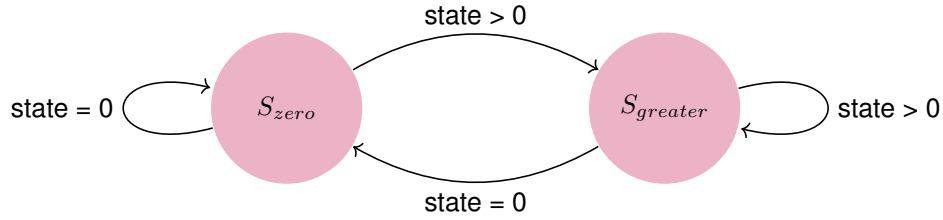


Figure 4.7: Status generation FSM

Outputs from states are S_{zero} : `int_status = 0`, $S_{greater}$: `int_status = 1`

to a value of 1. The state machine is represented in figure 4.7. The input to the state machine is **state**, the output of the system is **int_status**. S_{zero} state represents the first case and the $S_{greater}$ represents the default case.

4.4.5 PULSE GENERATOR

The pulse generator is designed to take in a 3-bit input and generate pulses on 6 different lines. **i_generate** and **i_data** are inputs from the DRS block. **i_generate** is the control variable used to control the generation and the width of the pulses. The longer **i_generate** is pulled up the longer the pulses. **i_generate** is meant to be used in tandem with the incoming data to generate the pulses when data is ready.

Listing 4.2: Pulse generation logic

```

1  assign pulses = int_pulses;
2
3  always @(posedge clk) begin
4      if (~rst) begin
5          int_pulses <= 6'b000000;
6      end else if (i_generate) begin
7
8          if (i_data == 3'b000) begin
9              int_pulses <= 6'b000001;
10         end else if (i_data == 3'b001) begin
11             int_pulses <= 6'b000010;
12         end else if (i_data == 3'b010) begin
13             int_pulses <= 6'b000100;
14         end else if (i_data == 3'b011) begin
15             int_pulses <= 6'b001000;
16         end else if (i_data == 3'b100) begin
17             int_pulses <= 6'b010000;

```

```

18         end else if (i_data == 3'b101) begin
19             int_pulses <= 6'b100000;
20         end else begin
21             int_pulses <= 6'b000000;
22         end
23     end else begin
24         int_pulses <= 6'b000000;
25     end
26 end

```

In the code listing 4.2, I present the hardware description code for the pulse generator. The entire module is not shown, rather the core logic of the module is shown. Line 1 shows an internal pulses variable being assigned to the output variable pulses. In line 3-26, we have a positive edge triggered block that generates pulses using the variable **int_pulses** (internal pulses). In line 4-5, a rst (reset) signal is used to pull all **int_pulses** low. In lines 6-23, the input signal **i_generate** is used to set the variable **int_pulses** with respect to the input data (**i_data**). In lines 23 and 24, I set **int_pulses** variable to low when there is no **i_generate** signal, which will generate the required pulses.

4.4.6 WIDTH AND DELAY CONTROL

The optical pulse generated from our system are affected by the following variables:

1. The propagation delays between different paths inside the FPGA are dependent on the compiler. This can be controlled to some extent using tools such as routing constraint files provided by Intel. However, we still do not have fine-grain control required to ensure the timing between each electrical pulse is constant.
2. Not all LEDs are made same. This means that the optical pulses from different LEDs for similar electric pulses are not always same. They are different in most cases. To ensure the shape of the optical pulses generated by each LED are similar, we need to fine-tune the electrical pulse input for each LED separately.

To mitigate these issues, I had to fine-tune the delays and width of the electrical pulse sent to each LED. I designed two verilog modules to achieve this. The two modules are delay control and width control, which are explained below. By varying the widths and delays and monitoring the output optical pulses we are able to ensure that the optical pulses from every channel is emitted at equal time gaps, making

them indistinguishable from each other. This ensures that our temporal mode stays constant which is a requirement for indistinguishability as discussed in 1.6.

DELAY CONTROL

NOT gates in the FPGAs are usually implemented as look-up tables. Every look-up tables is made up of logic gates that have a certain time delay. On the cyclone V FPGAs, the delay is around 250 ps. I use this property to chain inverters to generate a delayed version of the signal. The downside to this approach is the resolution of the delay is based on the implementation of a NOT gate in the FPGA. On our FPGA, we were able to have a delay of 250 ps for each NOT gate. To ensure that our signal isn't inverted we need to have a even number of NOT gates. This reduces our delay resolution to 500 ps. The circuit model for the implementation is shown in figure 4.8.

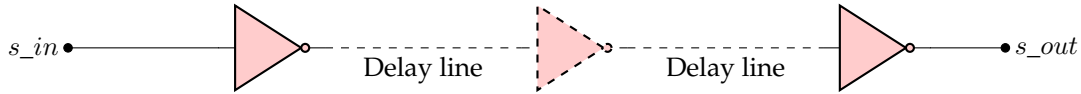


Figure 4.8: Delay control circuit diagram

WIDTH CONTROL

If the shape of the optical pulses does not match, we need to change the widths of the pulses to match the optical pulses. Since the optical pulses will be in reference to each other, we can just narrow the rest of the pulses to match the lowest optical pulse. This realization drove the design choice for this component. We only need to reduce the width of the pulses. The generated pulse will be the widest pulse, and any pulse that needs width adjustment will be narrowed via the width-control module.

The implementation idea was to build up on the delay control and then AND the output with the delayed version to narrow the pulse-width. This gives us a resolution of 500 ps (pulse can be narrowed by 500 ps in one step). The circuit model for the implementation is shown in figure 4.8.

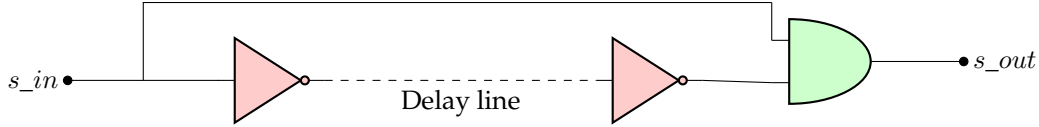


Figure 4.9: Width control circuit diagram

4.5 LED BOARD FOR OPTICAL PULSE GENERATION

The LED board is designed having mounting space for 3 LEDs. Each LED can be controlled by 2 different ports. This gives us 6 channels (3 signals, 3 decoys). Each port has a corresponding path and a resistor. R1 is on the signal path and R2 is on the decoy path. The signal path will drive the LED at full brightness, while the decoy path will drive the LED at a lower brightness (40%).

The LEDs are rated for a maximum forward current of 30 mA, and they are driven by electric pulses at 3.3 V at 12.5 MHz and a pulse width of 10 ns. We can calculate the average voltage as seen by the LED from the above information. The resistance value R1 can be calculated from the simple formula *Voltage/Current*. R2 needs to drive the LED at 40% [6] lower brightness for the decoy-state. The LED is a non-linear device and the translation from R1 to R2 is not straight-forward. The values for R1 and R2 are given in table 4.1.

Resistor	Value (in Ohms)
R1	12
R2	20

Table 4.1: Resistor values for LED board

The board is designed to mount LEDs on one side and have female connector port on the other side. The connector port can then be mounted on the FPGA GPIO pins. The ports at the bottom are labeled 1 through 10. Port 1, 2 are connected to GPIO[1] and GPIO[3] respectively. Port 4, 5 are connected to GPIO[7] and GPIO[9] respectively. Port 9, 10 are connected to GPIO[15] and GPIO[17] respectively.

To send signal pulses we put the decoy paths in high impedance mode. This lets the current take the path of least resistance, which is through the signal path into the LED. The same is true for sending decoy states, the lower brightness is dictated by the lower current passing through the decoy path because of the higher resistance (refer table 4.1).

The work presented in this section is done by Daniel E. Sanchez-Rosales with minor discussions and

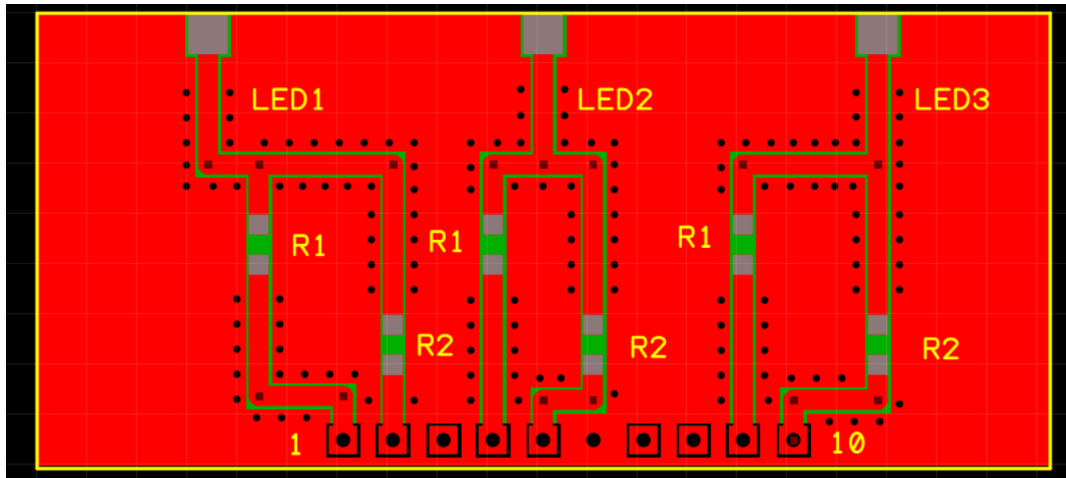


Figure 4.10: LED Board

Designed in ExpressPCB and ordered from expresspcb.com

inputs from me.

4.6 BARE-METAL TRANSMITTER PROGRAM

In this section we will discuss about the transmitter program used to control the FPGA. Programs are typically called bare-metal programs when the access to low-level hardware is done in the most direct way possible. In this case, it is via memory addresses. Each accessible component is memory mapped on the AXI bridge. The program uses a finite-state machine to handle the data transfer and status updates.

The program will be discussed in two sections. The initial setup for the program will be discussed in 4.6.1 and the state machine will be discussed in 4.6.2. Code presented in this section are stored in the SD card and are ran on the HPS system.

4.6.1 PROGRAM SETUP

The first part of the setup is to memory-map the heavy-weight and light-weight bridges. This can be done conveniently by using the *mmap* function provided by Linux.

Listing 4.3: Light-weight AXI bridge mapping


```

1 // we'll actually map in the entire CSR span of the HPS since we want to access
   various registers within that span
2 lw_axi_vbase = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED,
   fd, HW_REGS_BASE);

```

In the code listing 4.3, I create a new mapping for the light-weight bridge in the virtual address space and assign it to the pointer `lw_axi_vbase`. The mapping is done using the `mmap` function provided by Linux. The first argument is the base address which will be `NULL`. The second argument is the length of the address space to be mapped, which is declared as a constant `HW_REGS_SPAN`. The third argument is the desired memory protection of the address space being mapped. Since I need to read and write from the bridges I use `PROT_READ` and `PROT_WRITE` (protection_read and protection_write) to enable reading and writing. `MAP_SHARED` enables the region to be used by any other process running concurrently in the system. `fd` is a pointer to the file descriptor that will be returned by the mapping. I use the `fd` to validate if the mapping was successful. The validation process is not shown. The last argument is the offset of the address being mapped; this offset is stored in a constant `HW_REGS_BASE`.

Listing 4.4: Heavy-weight AXI bridge mapping

```

1 // get the base address for the HW axi bridge
2 hw_axi_vbase = mmap(NULL, HW_FPGA_AXI_SPAN, (PROT_READ | PROT_WRITE),
   MAP_SHARED, fd, ALT_AXI_FPGASLVS_OFST);

```

In the code listing 4.4, I create a mapping for the heavy-weight bridge using the same function `mmap` used to map the light-weight bridge. I pass in the appropriate offset and length of the address space as `ALT_AXI_FPGASLVS_OFFST` and `HW_FPGA_AXI_SPAN`.

Once the bridges are setup, I can setup our pointers to the Buffer memory and the Status register as shown in the code listing 4.5.

Listing 4.5: Base pointer construction for Memory bank

```

1 mem_0_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) MEM_0_BASE
   & (unsigned long) HW_FPGA_AXI_MASK));

```

Listing 4.6: Base pointer construction for Status register

```

1 status_0_lw_axi_base = lw_axi_vbase + ((unsigned long) (ALT_LWFPGASLVS_OFST +
   STATUS_0_BASE) & (unsigned long) HW_REGS_MASK);

```

In the code listings 4.5 and 4.6, I use two pointers **status_0_bank_on_lw_axi_base** and **mem_0_bank_on_axi_base** to read and write from the buffer memory and the status register. To get the real address, the base addresses for the buffer memory and the status register are bitwise anded with their respective masks to get the relative offset. The relative offset is then added to the base address of the respective AXI bridges.

In code listing 4.7, I declare pointer to the file to be read and a buffer. The data needs to be read from a file and read to a buffer. To accomplish this, a file pointer and a local buffer is declared. The buffer is the size of a virtual bank. In line 2 we declare the file pointer. In line 3, we declare an unsigned 8-bit integer buffer of 64 kB width. **BUFFER_SIZE** is a constant equal to 65536 (64 k).

Listing 4.7: Buffer and File pointer declaration

```
1 // FILE pointer and other requirements for the input file
2 FILE *file;
3 uint8_t buffer[BUFFER_SIZE];
```

4.6.2 TRANSMISSION - STATE MACHINE

The state machine is simple. It has four states namely IDLE, RB, CS, WRITE as shown in figure 4.6.2. The state machine is a part of the bare-metal program and runs on the HPS.

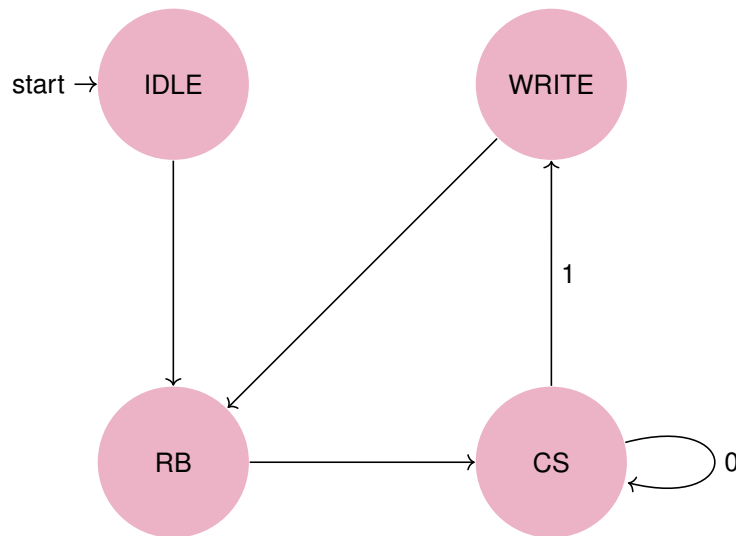


Figure 4.11: Transmitter - Finite state machine

RB - Read bank, CS - Check status

0 - No status update, 1 - Status update received

1. The IDLE state serves the purpose of having a starting point for the FSM.
2. RB stands for Read bank. In this state the state machine reads a chunk of data into the buffer.
3. CS stands for check status. This is a looped state i.e. The machine stays in this state until the result of check status returns true/1. In this state the machine polls the status register continuously to check for updates. If there is an update, the state machine proceeds to the WRITE state.
4. WRITE state is solely reserved for the purposed of loading the local buffer into the appropriate virtual bank on the FPGA.

4.7 OPTICAL SETUP

In this section we will discuss about the optical part of the transmitter. This includes the RC-LEDs, the LED couplers, In-line attenuators and the optical bench. The overview of the system can be seen in figure 4.12. My contribution to the optical setup is limited to participating in design iterations for the optical coupler. All work presented in this section has been done by Roderick Douglas Cochran and Daniel Sanchez-Rosales. My involvement has been limited to discussions.

The optical pulses generated from a single LED is coupled into a single-mode fiber. The single-mode fiber has a in-line variable attenuator, which allows us to manually change the attenuation to vary the mean photon number. The single-mode fiber is then coupled into the optical bench using a collimation package. This same setup is duplicated for all the three LEDs.

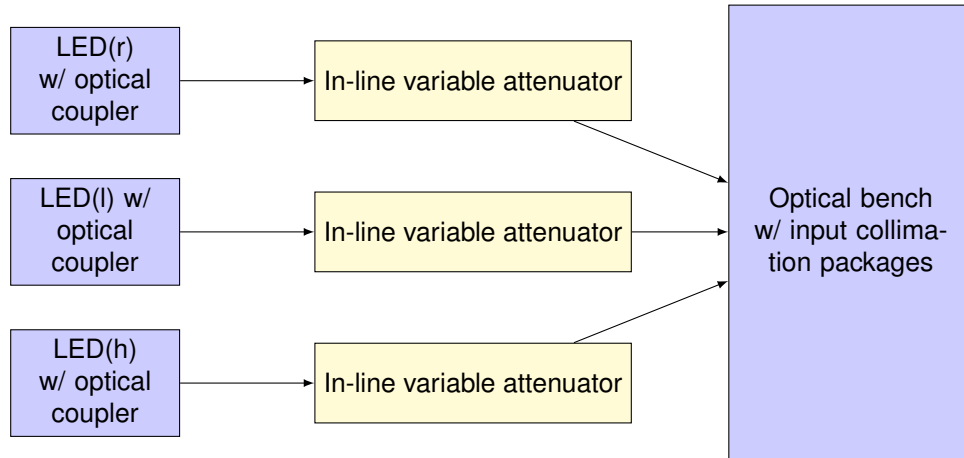


Figure 4.12: Transmitter optical setup

The optical setup for transmitter is made-up of four distinct modules:

1. Resonant cavity Light emitting diode (RC-LED)
2. Optical coupler (LED to Single-mode fiber)
3. In-line variable fiber attenuator
4. Polarization bench (Optical bench)

4.7.1 RESONANT CAVITY LIGHT EMITTING DIODE (RC-LED)

We are using a 650 nm Resonant cavity LED manufactured by Roithner LaserTechnik GmbH, mounted in a standard TO-46 package. It has a rated output of 1 mW. Using a RC LED over common LED provides us some necessary benefits [2].

1. Improved spectral purity.
2. Enhanced light emission due to reflective mirror on the bottom side.
3. RC LEDs are capable of enhanced spontaneous emission due to optical resonance cavity.
4. Faster turn on and off times.

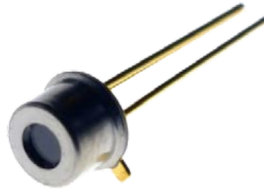


Figure 4.13: Resonant cavity- Light emitting diode (RC650-TO46FW) made by Roithner LaserTechnik GmbH

4.7.2 OPTICAL ENCLOSURE/COUPLER (LED TO SINGLE-MODE FIBER)

Usually when working with optics, many devices come with coupling mechanisms that makes pairing fibers into them. The optical devices in the system are just off-the-shelf LEDs. To squeeze out maximum coupling from the LEDs to the fibers, we designed and 3D-printed an optical coupler. The coupler can be seen in figure 4.2b.

4.7.3 SINGLE-MODE FIBER AND IN-LINE VARIABLE ATTENUATOR

Each LED produces an optical pulse of multiple photons. These photons are made of all spatial modes. We are only interested in having a single photon with a single spatial mode. To filter the spatial modes, we use a single-mode fiber coupled to the LED as described in 4.7.2. The single-mode fiber doesn't ensure that



Figure 4.14: In-line variable fiber attenuator by Oz-optics (Custom ordered)

the optical pulse contains a single photon. We need to attenuate the optical signal to have an approximate mean photon number of 1. To achieve a mean photon number of 1, we need to have ability to attenuate the optical signal in the fiber. This is accomplished by an in-line variable fiber attenuator (fig. 4.14).

We custom ordered these fibers with special specifications. First, the fiber length is 1 feet, to make sure that the fibers are not short enough to be able to go around the system and not long that they end up dangling, once we deploy the system on drones. Second, the fiber is black jacketed to reduce noise induced from the environment.

Three separate channels are used to achieve three different polarization states. So, we need three in-line variable fiber attenuators to control the mean photon number in each channel. Each fiber is a single-mode fiber, making sure we have the same mode in every packet from each channel. This ensures our spatial mode is kept constant which is a requirement for indistinguishability as discussed in 1.6.

4.7.4 POLARIZATION BENCH (OPTICAL BENCH)

To prepare polarization states we designed a polarization bench, which takes in three inputs (since we are using a fewer state version of the BB84 protocol, discussed in 1.4). The three inputs are fed from the LEDs connected via single-mode fibers. The bench components and their arrangements are shown in figure 4.15.

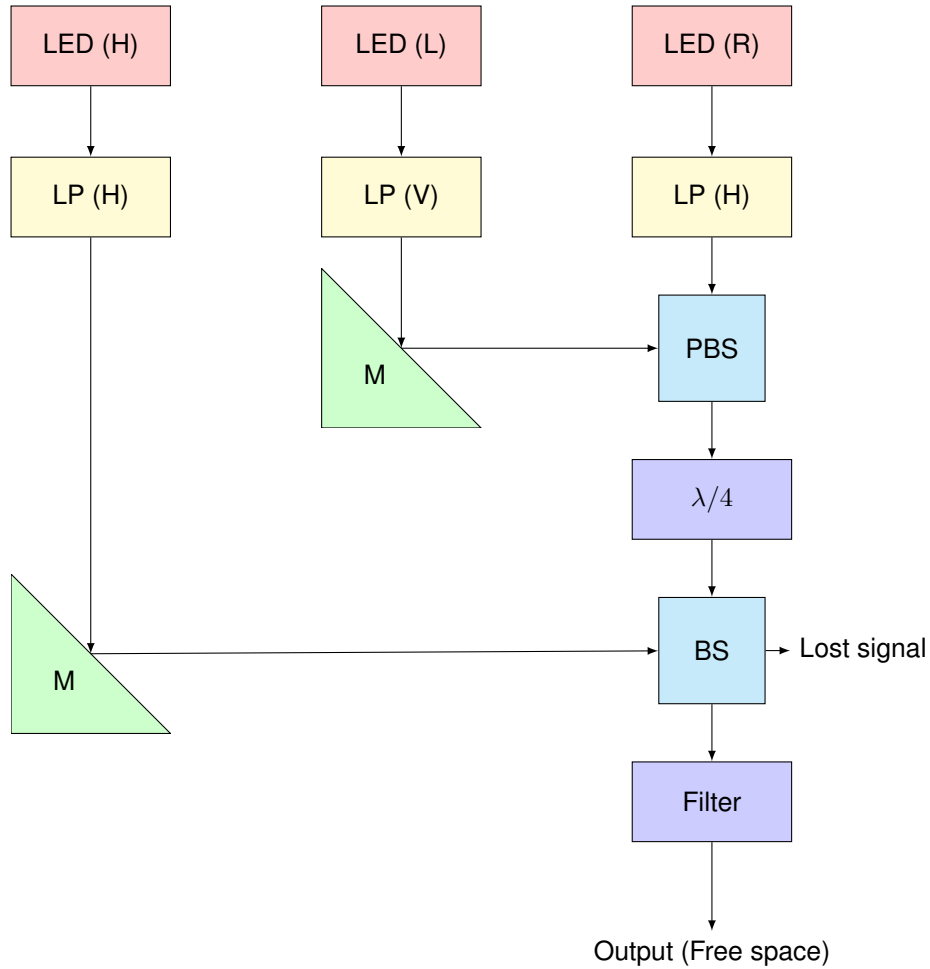


Figure 4.15: Transmitter Optical bench

LED - Light emitting diode, LP - Linear polarizer

PBS - Polarizing beam splitter, Filter - Spectral filter (650 nm)

LED(H), LED(L), LED(R) labelling is just to represent the channel

All the LEDs produce unpolarized light. We are labeling the LEDs as LED (channel name). Each

LED will only be responsible for producing the photon packets that will be eventually polarized into their corresponding states.

LED(H) is connected to LP(H) via the attenuated fiber. The optical pulse is then lineally polarized in H basis. This signal is sent to the combining beam splitter (BS) to be combined with the rest of the signal and be launched into the filter. The beam splitter (BS) transmits only 50% of the signal, reducing the overall signal by 50%.

L, R polarizations are prepared in the following procedure. First, polarizing the L, R channels into H, V. Then a Polarizing beam splitter is used to combine H, V into a single beam. Passing the beam through a quarter wave plate ($\lambda/4$) will turn linear polarizations to circular polarizations (L and R). There is no loss in this stage. After the quarter wave plate, we send the signal to the combining beam splitter, where we lose 50% of this signal.

The filter is used to filter out the frequencies other than 650 nm. The filter helps in mitigating side channel attacks based on spectral difference. Since this is a part of the entire transmission system we need to ensure the total mean photon number at the end of this bench, which is the spectral is equal to 1 to combat PNS attacks. The filter ensures that our output spectrum stays constant ensuring that our spectral mode stays constant, which is requirement for indistinguishability as discussed in 1.6.

CHAPTER 5

RECEIVER

5.1 OVERVIEW OF THE RECEIVER SYSTEM

The hardware platform used to build the transmitter system is identical to the transmitter platform used in section 4.1. Figure 5.1 shows a high-level view of the Receiver system.

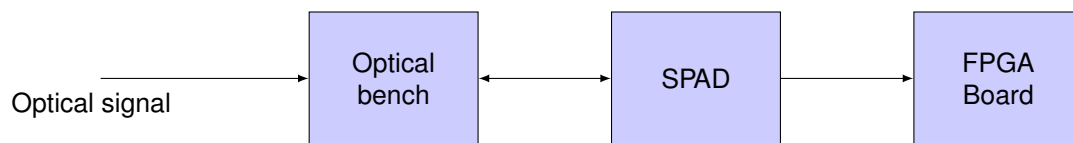


Figure 5.1: Overview of Receiver system

SPAD - Single photon counting avalanche photodetector

My contribution to the Receiver is the development of the FPGA module and the bare-metal program. The SPAD connector mounting board was developed by Roderick Douglas Cochran, with help from Daniel Sanchez-Rosales. Samantha Darlene Issac designed and developed the receiver optical system with inputs from Daniel Sanchez-Rosales and Roderick Douglas Cochran.

The optical bench is the input to our receiver system. The optical output of the transmitter is aligned with the receiver optics to ensure maximum signal coupling. The main function of receiver optics is to select a random basis for measurement and take the measurement. The optical bench will be explored further in

5.2.

The SPAD (Single photon counting avalanche photodetector) is used to detect the photon as measured by the optical bench. The detection process and device output are discussed in detail in 5.3. The SPAD generates an electrical TTL pulse for each single photon detection. This pulse is launched into a transmission line and captured on the FPGA board.

The basic function of the FPGA in the receiver system is to time-tag the incoming data. The time-tagged data is then sifted through for key extraction. The system and its various pieces are covered in sections 5.3 and 5.4.

5.2 OPTICAL SETUP

The optical bench for the receiver is shown in figure 5.2, was designed and developed at UIUC by Samantha Darlene Issac. The input to the system is the optical pulses from transmitter and the ambient noise. The ambient noise is filtered out using a filter. The receiver's primary job is to do a measurement in random bases.

A beam splitter splits the optical signal into two different paths equally (i.e. with a 50% probability). So, the filtered input regardless of the polarization state is split into the left and right arm equally.

5.2.1 LEFT ARM

In the left arm, the first optical component is a polarizing beam splitter. This polarizing beam splitter can sort H, V polarization with 100% purity. Hence, every optical pulse in H, V polarization that enters the left arm will be sorted into the correct detector.

When L or R polarization enter the left arm, they have an equal chance of being reflected or transmitted. Hence, 50% of them end up in either leg of the arm.

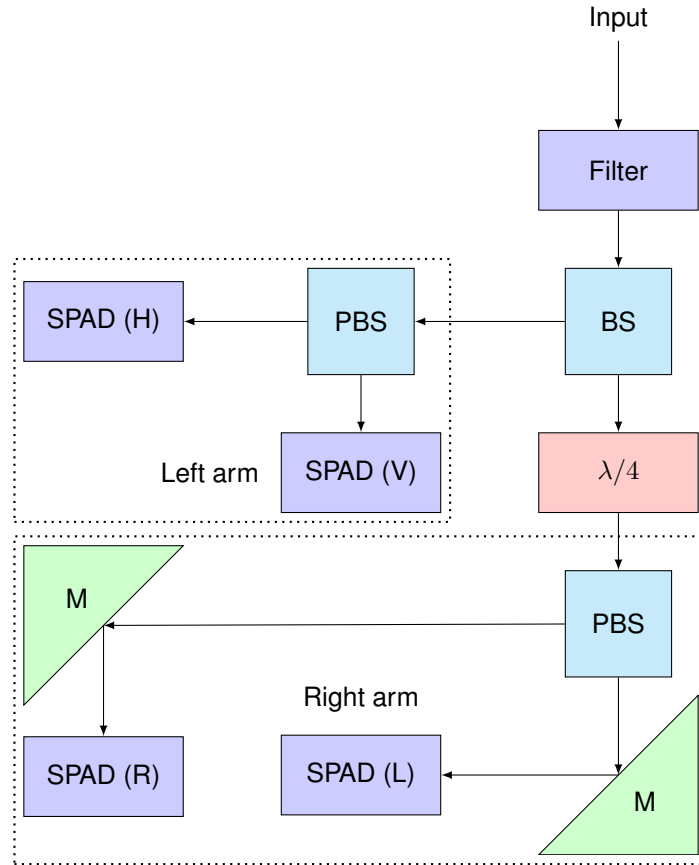


Figure 5.2: Receiver Optical bench

PBS - Polarizing beam splitter, BS - Beam splitter

M - mirror, $\lambda/4$ - Quarter wave plate

5.2.2 RIGHT ARM

In the right arm, the first optical component is a quarter wave plate ($\lambda/4$). The quarter wave plate converts H, V bases to L, R and L, R bases to H, V respectively.

At the polarizing beam splitter H, V are sorted with 100% purity. Hence, sorting the polarizations that were originally L, R with 100% accuracy. The original H, V bases that are now L, R are split equally into the two legs of the arm.

Original Polarization	H-Detector	V-Detector	L-Detector	R-Detector
H	50%	0%	25%	25%
V	0%	50%	25%	25%
L	25%	25%	50%	0%
R	25%	25%	0%	50%

Table 5.1: Polarization sorting in percentages for the receiver optical bench

5.3 ELECTRICAL SETUP

There are 3 distinct physical components involved in the electrical setup as shown in figure 5.1. The SPAD has four output channels and each channel expects a $50\ \Omega$ termination via a BNC output. To capture the outputs from SPAD with load matching we designed a PCB with SMA connectors which when mounted on the input GPIO pins will create a load of approximately $50\ \Omega$. The PCB will be discussed further in section 5.3.2.

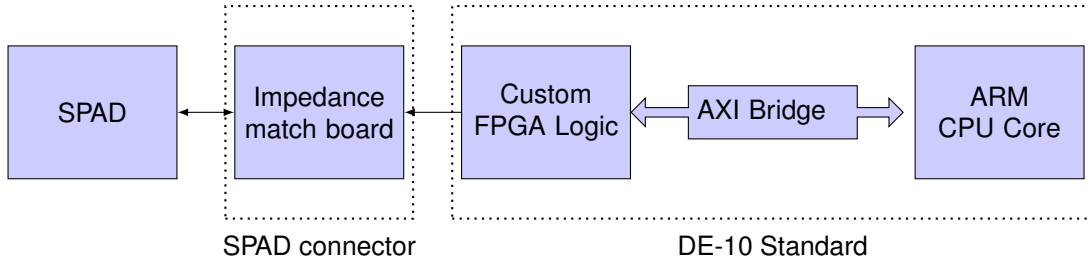


Figure 5.3: Receiver electrical setup overview in Block diagram

5.3.1 SINGLE PHOTON DETECTORS

The security of QKD relies heavily on the transmission of single photons ensuring that PNS attacks cannot be done. Single photon detectors provide the opportunity to transmit single photons and capture it reliably on the receiver. There are many technologies [11] that have different targeted uses. The technology of choice for QKD is single photon counting avalanche photodetector (SPAD).

SPADs are operated in Geiger-mode [11], which is in the reverse-breakdown region. The SPAD is usually biased in the breakdown region and near the Geiger-mode. A single photon hitting the diode will introduce a small bias ΔV , which sets off an avalanche breakdown [11]. The breakdown will generate



Figure 5.4: Single photon counting module array (SPCM-AQ4C) made by PerkinElmer

The array contains 4 counting modules

Output: TTL pulses, 25 ns wide at 4.5 V; Dead time: 50 ns; Dark count rate: 500 counts/sec

a signal, which when paired with a quenching circuit [11] can be used to create pulse for each photon detection.

Every SPAD available in the market has a dead time [11], which is the time after a photon detection where the diode is very likely to see another photon generated from the avalanche process. Since detecting photons from the avalanche process is not desirable, the bias is temporarily reduced below breakdown, letting the diode stabilize. This time where nothing happens is known as the dead time of the detector. The dead-time is important in the context of the overall speed of the system. The dead time determines how fast can we detect single photons. Hence, capping the overall system frequency.

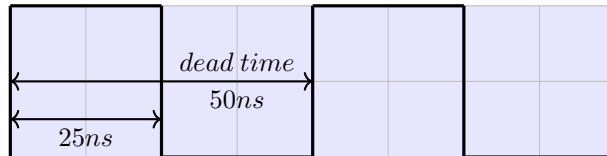


Figure 5.5: Maximum frequency SPAD output

Our setup uses single photon counting module array [30] made by PerkinElmer. This particular SPD was chosen because the of operating frequency [30] (60% peak efficiency at 650 nm) is very close to the operating frequency of the LEDs, which make the choice of the LEDs and this SPD a perfect pair. This device has also been used in previous work of QKD [23, 17, 18]. We will refer to this detector array as single photon counting module array (SPCM). SPCM is a four-channel card that has four independent single photon detectors, which fits our requirements perfectly. It has a dead time of 50 ns. Thus, capping the

operating frequency at 25 MHz. This limitation is shown in figure 5.5. Some of these detector's efficiency is sensitive to frequency of incident photon. The SPCM has a peak photon detection efficiency of 60% at 650 nm [30], which is very close to our operating photon frequency. The dark count rate is moderately low at 500 counts/sec.

5.3.2 IMPEDANCE MATCHING TRANSMISSION LINE BOARD (SPCM CONNECTOR)

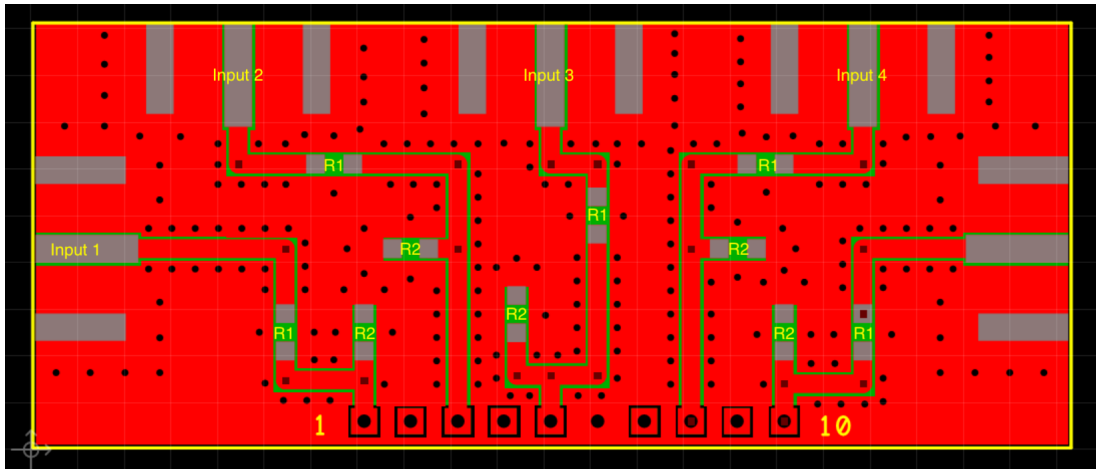


Figure 5.6: Impedance matched SPCM connector - Front plane

Designed in ExpressPCB and ordered from expresspcb.com

The SPCM has four outputs corresponding to four different on-board SPADs. Each channel expects to be impedance matched at $50\ \Omega$. The output impedance can be matched by using a BNC cable with an impedance of $50\ \Omega$. BNC connectors are big and clunky, while SMA connectors are small and compact. So, we use a BNC to SMA cable with an impedance of $50\ \Omega$ and have a custom PCB with Female SMA connectors as input ports. A female SMA connector is shown in figure 5.7.

The output from SPCM is a TTL (transistor-transistor logic) pulse at 4.5 V. This output is higher than the operating voltage for the GPIO pins on the FPGA board, which is 3.3 V. As shown in figures 5.6 and 5.8, each input has 5 inputs (3 on the front place, 2 on the back plane) as required by the SMA connector. 4 terminals are Ground terminals, while the center terminal is the Signal.

The Resistors R1 and R2 are used as a voltage divider for the signals. The values are reported in 5.2. These values don't produce an accurate 3.3V at the output, rather they are meant move the voltage level



Figure 5.7: Female SMA Connector, with 5 terminals

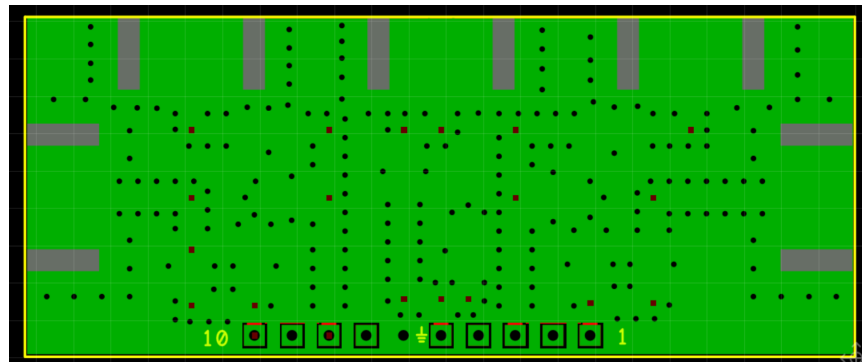


Figure 5.8: Impedance matched SPCM connector - Back plane

Designed in ExpressPCB and ordered from expresspcb.com

closer to 3.3V.

Resistor	Value (in Ohms)
R1	16
R2	37

Table 5.2: Resistor values for SPCM connector

The work presented in this subsection has been a collaboration of ideas from the three of us (Akash Gutha, Daniel Sanchez-Rosales and Roderick Douglas Cochran). Once the design was done, Roderick was responsible for making and ordering the PCB.

5.4 FPGA RECEIVER MODULE

The FPGA receiver module has four channels for four inputs. As shown in figure 5.9, each channel has a time tagger, memory controller, status controller, buffer memory to capture and process incoming signals. The receiver's job is to store the detection time tags for incoming signals for each channel. Each time-tag is 16 bits wide (2 bytes). The buffer memory is 128 kB wide and is divided into two virtual banks of 64 kB each. The data is sent from the buffers to the HPS in chunks of 64 kB via the AXI bridge. The virtual bank statuses are updated through the 8-bit wide status register. Since each time-tag is 2 bytes wide, the total buffer memory can store 64 kilo-hits, each virtual bank can store 32 kilo-hits.

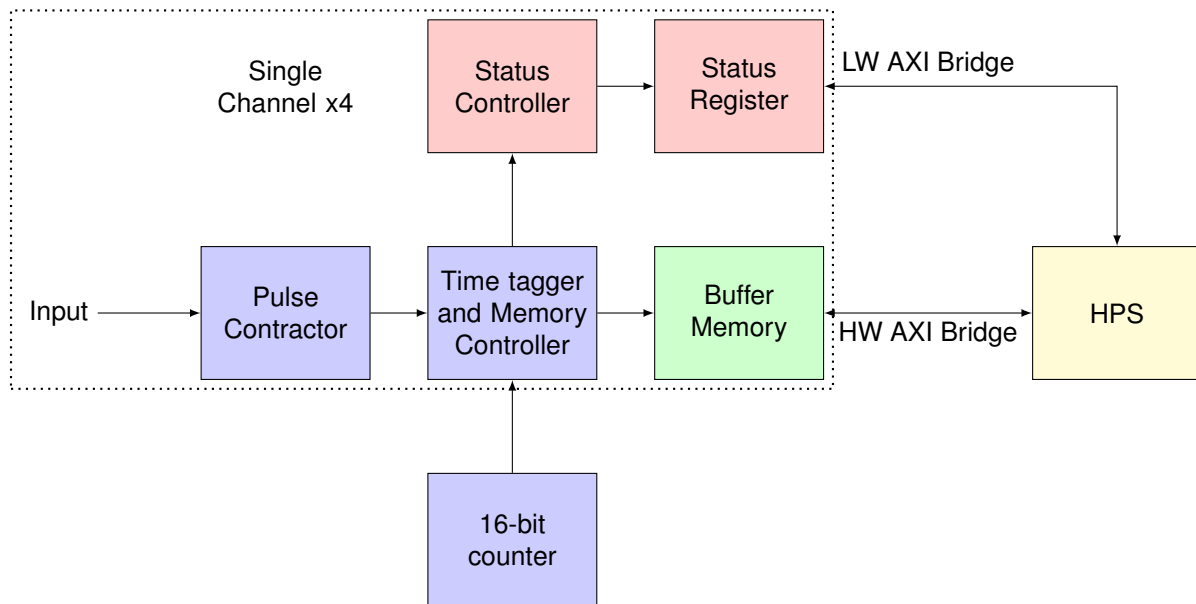


Figure 5.9: Receiver module

5.4.1 CLOCK GENERATION

Similar to the clock generator in section 4.4, we use a PLL IP from Intel to generate required clocks. The reference clock is 50 MHz. The PLL is configured to be in integer-N mode. The desired clock frequency for our receiver system is 100 MHz. 100 MHz was chosen for the following reasons. First, to receive reliably at 25 MHz we need to have a sampling frequency equal to or faster than 25 MHz. Second, to have a time tagger resolution of 10 ns we need the time-tagger running at 100 MHz. The time-tagger is elaborated in the section 5.4.3.

5.4.2 PULSE CONTRACTOR

The first component the incoming signal passes through is the pulse contractor. Its function is to take a pulse wider than the pulse width of the reference clock and shorten it to the period of the reference clock. In our case, the incoming pulses are 25 ns wide and the reference clock is running at 100 MHz (period= 10 ns). We will refer to this as the contracted pulse in further discussion. The Verilog implementation is shown in code listing 5.1 and the equivalent circuit is shown in figure 5.10.

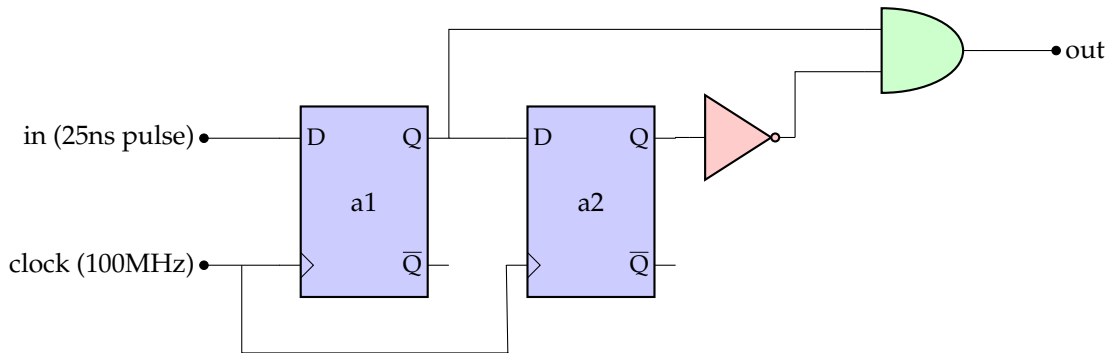


Figure 5.10: Pulse contractor circuit diagram

The code listing 5.1 is the Verilog implementation of the circuit shown in figure 5.10. In lines 3-6, we have a positive clock edge triggered block which takes the input and delays the input. The two delayed versions are **a1** and **a2**. In line 1, I create the contracted pulse by performing a logical **AND** operation on the two delayed pulses, which is assigned to the output port **out**.

Listing 5.1: Pulse contractor RTL code

```

1  assign out = a1 & (~a2);
2
3  always@(posedge clk) begin
4      a1 <= in;
5      a2 <= a1;
6  end

```

5.4.3 16-BIT TIME COUNTER

The 16-bit counter in fig 5.4 runs at 100 MHz. This block acts as the primary timekeeper in our system. Every photon detection will be tagged with the value from this counter. One concern about having such a small counter is counter overflow. Since each channel in the SPCM has a dark count rate of 500 counts/sec, we are guaranteed to have at-least one hit before the counter overflows. This ensures that we don't miss any time information and adversely affecting our timing information. The time can be linearized later in post-processing. As shown in the code below, we increment the counter every clock cycle.

Listing 5.2: Time tagger verilog implementation

```

1  always @(posedge fpga_clk) begin
2      time_tag_counter <= time_tag_counter + 1;
3  end

```

In the code listing 5.2, I present the HDL implementation for the time tagger. In line 1, I declare a block triggered at the positive edge of the **fpga_clk**. The **fpga_clk** is a 100 MHz clock. In line 2, I increment the **time_tag_counter** by 1 every clock cycle.

5.4.4 TIME TAGGER AND MEMORY CONTROLLER

This is a single block, that takes care of both time tagging the channel inputs and storing them into buffer memory. The input to this block is the contracted pulse and the time counter. The time tagger will be discussed first, followed by the memory controller.

I designed the time tagger to have a resolution equivalent of the time period of the operating clock frequency (20 ns in this case). There are other implementations that have much higher resolution [26, 20, 19, 24], on the order of pico-seconds. The decision to pick a lower resolution for the project was concluded from the following reasons. First, having a finer time resolution requires higher bit-width to represent

each hit. This will increase the data storage requirements for the equal amount of hits. Second, the timing resolution is not a significant constraint for sifting the data. The timing resolution can be taken care of by accommodating the time ambiguity into the sifting algorithm, giving us some headroom for choosing a lower resolution.

The time tagger is implemented as level triggered logic. The trigger is the contracted pulse. Since each contracted pulse is not wider than the period of the clock, we can tag the pulse with the corresponding 16-bit time. This value is stored in a temporary register which will be written to the buffer memory by a state machine. The trigger also increments the target address register for every pulse triggered. The implementation of this functionality is shown in the code below.

Listing 5.3: Time tag and address update RTL code

```

1  always @(posedge clk) begin
2      if (~rst) begin
3          int_address <= {ADDRESS_WIDTH{1'b1}};
4          int_data <= 0;
5      end else begin
6
7          if (i_detection) begin
8              int_data <= i_time_tag;
9              int_address <= int_address + 1;
10         end else begin
11             int_data <= int_data;
12             int_address <= int_address;
13         end
14     end
15 end

```

In the code listing 5.3, I present a part of the memory controller responsible for the target address update logic and time tag capture logic. In lines 2-4, the reset block sets the **int_address** to its max value, and **int_data** to 0. The **int_address** and **int_data** variables represent internal address and internal data variables. In lines 7-13, I check for a detection event by monitoring the **i_detection** variable. If there is a detection event, the time tag is recorded into the internal data variable and internal address is incremented by 1. In the case of no detection, the variables are not changed.

Next, the state machine responsible to time-tag and write the data into the buffer memory will be discussed. The state machine has four states. The four states will be referred to as states 0,1,2,3 respectively.

1. The default state is state 2. State 2 also uses level triggered logic to branch of to State 0 on a pulse

detection. If there is no pulse detected, state 2 stays in a loop until the next detection.

2. When there is a detection, the state machine jumps to state 0. state 0 pulls the write signal high, which writes the time tag in the temporary register to the buffer memory.
3. The state machine then automatically jumps to state 1, which is an extension of state 0. state 1 is to remove any timing miss-match occurred while sending the write signal. The state machine then automatically jumps to state 2 to loop until the next detection.
4. State 4 serves no meaningful purpose other than to explicitly set the signals to known values in-case of any failure.

The code for this state machine is presented and discussed in the code listing 5.4.

Listing 5.4: Write signal generator

```
1  always @(posedge clk) begin
2      if (~rst) begin
3          state <= 2'b10;
4      end else begin
5          case (state)
6              2'b00: begin
7                  state <= 2'b01;
8                  int_write <= 1'b1;
9              end
10             2'b01: begin
11                 state <= 2'b10;
12                 int_write <= 1'b1;
13             end
14             2'b10: begin
15                 if (i_detection) begin
16                     state <= 2'b00;
17                     int_write <= 1'b0;
18                 end else begin
19                     state <= 2'b10;
20                     int_write <= 1'b0;
21                 end
22             end
23             2'b11: begin
24                 state <= 2'b00;
25                 int_write <= 1'b0;
26             end
27         endcase
28     end
29 end
```

```

28     end
29
30 end

```

In the code listing 5.4, I present the verilog realization of the state machine responsible for generating the write signals. The **state** variable is used to keep track of the current state of the fsm. **int_write** is the internal write variable used to issue write signals. The lines 7-8 represent state 0. In this state we issue a write signal, which is carried forward throughout state 1 in lines 11-12. The lines 15-16 represent state 2. In state 2, the fsm monitors the **i_detection** variable to issue a write command. The write command is issued by states 0 and 1. Hence, state 2 falls back to state 0 when there is a detection event. The lines 24-25 represent state 3, which is used to handle cases of system failure.

5.4.5 STATUS CONTROLLER AND STATUS REGISTER

The status controller takes in the current target address as the input. It has only one job and that is to indicate the status of the buffer memory, which is the index of virtual bank data is being currently written to. In the following verilog program, the output state is switched when the target address hits either 0 or the half-way mark. Each output state corresponds to the virtual bank being written. The internal status that is being updated is connected to the status register on the AXI bridge. Hence, making the status available for the HPS to poll.

The Verilog implementation is similar to the code shown in code listing 4.1.

5.4.6 BUFFER MEMORY

The memory IP used in the receiver is the on-chip memory Intel FPGA IP, which was also used in 4.4. It is also a dual-port access, single clock operation configuration as shown in figure 5.11.

The two ports are S1, S2 respectively. S1 is connected to the Memory controller logic on the FPGA, while S2 is connected to the HPS. S1 is 16 bits wide, this makes it easier to write our 16-bit wide time tag to an address without the complexity of dealing with byte positions. Since only one time-tag is written per detection, there is no advantage in using a wider bus. S2 is 32-bit wide and is connected to the HPS via HW AXI bridge. In theory, this bus can be as wide as it can. But there is a restriction of 2X width for dual-port memories in this specific IP.

Memory type

Type: RAM (Writable) ▾

☒ Dual-port access

☒ Single clock operation

Read During Write Mode: DONT_CARE ▾

Block type: AUTO ▾

Tightly Coupled Memory operation require dual port & dual clock sources.

Size

☒ Enable different width for Dual-port access

Slave S1 Data width: 16 ▾

Slave S2 Data width: 32 ▾

Total memory size: 131072 bytes

☐ Minimize memory block usage (may impact fmax)

Figure 5.11: Buffer memory configuration for Receiver

5.4.7 SCALING UP TO MULTIPLE CHANNELS

Up until now the discussion has been about a single channel. This discussion will be about scaling up the single channel into four different channels.

To generate multiple blocks for the custom FPGA blocks we can use the **generate** keyword in verilog. The generate loop creates physical copies of the component on the FPGA during synthesis. The code listing 5.5 shows how different status controllers are generated for each channel.

Listing 5.5: Generation of status controller for multiple channels

```

1 generate
2 for (gi = 0; gi < N_MEMS; gi++) begin: ctrlers
3     status_controller
4     # (
5         .ADDRESS_WIDTH (ADDRESS_WIDTH)
6     )
7     status_ctrl (
8         .clk (clk),
9         .rst (rst),
10        .i_address (i_addresses [ ((gi+1) * ADDRESS_WIDTH) - 1 : (gi * ADDRESS_WIDTH) ]),
11        .o_status (o_statuses [ gi ])
12    );
13 end

```

In line 2, the for loop of the generate loop is declared. `gi` is a genvar used to index the for loop. In lines 3-6 we declare the module name that will be generated by the loop. The required parameters are also passed into the module. In lines 7-12 we declare an instance of the module and pass the required wires/regs.

For each channel we need to instantiate a buffer memory and a status register. This had to be done manually in the platform designer and the entire setup is shown in figure 5.12.

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		hps	Arria V/Cyclone V Hard Process...					
		f2h_cold_reset_req	Reset Input	hps_0_f2h_cold_res...				
		f2h_debug_reset_req	Reset Input	hps_0_f2h_debug_r...				
		f2h_warm_reset_req	Reset Input	hps_0_f2h_warm_re...				
		f2h_stm_hw_events	Conduit	hps_0_f2h_stm_hw_...				
		memory	Conduit	memory				
		hps_io	Conduit	hps_0_hps_io				
		h2f_reset	Reset Output	hps_0_h2f_reset				
		f2h_sdram0_clock	Clock Input	Double-click to export	clk			
		f2h_sdram0_data	Avalon Memory Mapped Slave	Double-click to export	[f2h_sdra...	#		
		h2f_axi_clock	Clock Input	Double-click to export	clk			
		h2f_axi_master	AXI Master	Double-click to export	[h2f_axi_c...	#		
		f2h_axi_clock	Clock Input	Double-click to export	clk			
		f2h_axi_slave	AXI Slave	Double-click to export	[f2h_axi_c...	#		
		h2f_lw_axi_clock	Clock Input	Double-click to export	clk			
		h2f_lw_axi_master	AXI Master	Double-click to export	[h2f_lw_a...			
		f2h_irq0	Interrupt Receiver	Double-click to export			IRQ 0	IRQ 31
		f2h_irq1	Interrupt Receiver	Double-click to export			IRQ 0	IRQ 31
<input checked="" type="checkbox"/>		clk	Clock Source	exported				
<input checked="" type="checkbox"/>		status_0	PIO (Parallel I/O) Intel FPGA IP		clk	0x0000_0000	0x0000_000f	
<input checked="" type="checkbox"/>		status_1	PIO (Parallel I/O) Intel FPGA IP		clk	0x0000_0010	0x0000_001f	
<input checked="" type="checkbox"/>		status_2	PIO (Parallel I/O) Intel FPGA IP		clk	0x0000_0020	0x0000_002f	
<input checked="" type="checkbox"/>		status_3	PIO (Parallel I/O) Intel FPGA IP		clk	0x0000_0030	0x0000_003f	
<input checked="" type="checkbox"/>		mem_0	On-Chip Memory (RAM or ROM)...		clk	# multiple	multiple	
<input checked="" type="checkbox"/>		mem_1	On-Chip Memory (RAM or ROM)...		clk	# multiple	multiple	
<input checked="" type="checkbox"/>		mem_2	On-Chip Memory (RAM or ROM)...		clk	# multiple	multiple	
<input checked="" type="checkbox"/>		mem_3	On-Chip Memory (RAM or ROM)...		clk	# multiple	multiple	

Figure 5.12: Receiver platform design

In the figure 5.12, The window shown is called the Platform-designer. It is a part of the Intel Quartus FPGA programming package. This is a GUI interface that lets us connect different peripherals to each other visually. The connections are realized using compatible interfaces like Avalon-MM (as discussed in 4.3). There are many columns shown in the picture, the important columns are the connections column, base and end memory offset column.

The first component in the list is the HPS, which is expanded to show the various interfaces that can be used. The `h2f_axi_master` is the Heavy weight (HW) AXI Bridge, while the `h2f_lw_axi_master` is the light-weight (LW) AXI Bridge. The HW bridge is connected to the memory buffers (`mem_0-3`). The LW bridge is connected to the status registers (`status_0-3`).

Each status register is an instance of the Parallel IO core IP. The core provides different control variables as registers on the bridge. Together with the 1 byte required by the status register each core takes up 16 address spaces. `status_0` spans from `0x0000_0000` in the base column to `0x0000_000f` in the end column. Since the base and end column are representing offsets, `status_0` occupies the first 16 address spaces and the proceeding registers occupy the proceeding 16 bits of address space each.

In the next section, we will discuss about the bare-metal program running on the receiver HPS.

5.5 BARE-METAL RECEIVER PROGRAM

In this section we will discuss about the bare metal receiver program used to control the FPGA. The receiver program's function is to poll status registers and copy data from the virtual bank that corresponds the status update. Data is written in chunks of 64 kB. The discussion will be divided into two parts:

1. Program setup
2. Receiver state machine

5.5.1 PROGRAM SETUP

The pointers to memory mapped HW and LW AXI bridges are initialized and calculated using *mmap*, similar to 4.6.1.

For the receiver, we need access to four buffer memories and four status registers. These are already instantiated in the platform designer as shown in figure 5.12. Since each bank is divided virtually into two banks, we will setup pointers to the start of the memory and the mid-way address of the memory. We also need to setup pointers to the status registers.

Listing 5.6: Memory banks and status registers address mapping to a pointer

```
1 mem_0_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) MEM_0_BASE
   & (unsigned long) HW_FPGA_AXI_MASK));
2 mem_1_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) MEM_1_BASE
   & (unsigned long) HW_FPGA_AXI_MASK));
3 mem_2_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) MEM_2_BASE
   & (unsigned long) HW_FPGA_AXI_MASK));
```



```

4  mem_3_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) MEM_3_BASE
    & (unsigned long) HW_FPGA_AXI_MASK));
5
6  mem_0_mid_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) (
    MEM_0_BASE + MEM_0_SPAN / 2) & (unsigned long) HW_FPGA_AXI_MASK));
7  mem_1_mid_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) (
    MEM_1_BASE + MEM_1_SPAN / 2) & (unsigned long) HW_FPGA_AXI_MASK));
8  mem_2_mid_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) (
    MEM_2_BASE + MEM_2_SPAN / 2) & (unsigned long) HW_FPGA_AXI_MASK));
9  mem_3_mid_bank_on_axi_base = (uint8_t *) (hw_axi_vbase + ((unsigned long) (
    MEM_3_BASE + MEM_3_SPAN / 2) & (unsigned long) HW_FPGA_AXI_MASK));
10
11 status_0_lw_axi_base = lw_axi_vbase + ((unsigned long) (ALT_LWFPGASLVS_OFST +
    STATUS_0_BASE) & (unsigned long) HW_REGS_MASK);
12 status_1_lw_axi_base = lw_axi_vbase + ((unsigned long) (ALT_LWFPGASLVS_OFST +
    STATUS_1_BASE) & (unsigned long) HW_REGS_MASK);
13 status_2_lw_axi_base = lw_axi_vbase + ((unsigned long) (ALT_LWFPGASLVS_OFST +
    STATUS_2_BASE) & (unsigned long) HW_REGS_MASK);
14 status_3_lw_axi_base = lw_axi_vbase + ((unsigned long) (ALT_LWFPGASLVS_OFST +
    STATUS_3_BASE) & (unsigned long) HW_REGS_MASK);

```

The code presented in 5.6 is similar to the code presented in listing 4.5 and 4.6. In lines 1-4, I map the four pointers to the virtual addresses of the four memory banks. In lines 6-9, I find the central virtual address for each bank by memory mapping the calculated central base address $((\text{MEM_X_BASE} + \text{MEM_X_SPAN})/2)$. In lines 11-14, I memory map the status registers on the light-weight bridge and assign the pointers.

Next, we ensure that buffers are clean by setting all values in the buffer to 0. This is just a sanitary step to remove any data in the buffer memory from previous writes.

Listing 5.7: Clearing the memory banks

```

1  memset(mem_0_bank_on_axi_base, 0, MEM_0_SPAN);
2  memset(mem_1_bank_on_axi_base, 0, MEM_1_SPAN);
3  memset(mem_2_bank_on_axi_base, 0, MEM_2_SPAN);
4  memset(mem_3_bank_on_axi_base, 0, MEM_3_SPAN);

```

In the code listing 5.7, I use the **memset** function provided by the Linux system. In line 1, we call the function **memset** and pass three arguments. The first argument is the pointer to the address you want to set the value of. The second argument is the actual value to be copied to the address, which is 0 in our case. The third argument is the length of the address space if you want to set multiple addresses at once. The pointer

mem_0_bank_on_axi_bridge points to the first buffer memory calculated in section 5.5.1. **MEM_0_SPAN** is a constant equal the length of the address space occupied by the first buffer memory. In lines 2-4, I continue to clear the remaining banks.

5.5.2 RECEIVER STATE MACHINE

All the states in the state machine are defined as an enumeration as shown below. Each channel in the receiver is given a separate state (CHECK_STATUS_X) for status polling and memory bank reading (READ_BANK_X). Once the values are read to the local buffer it's written to file (WRITE_TO_FILE).

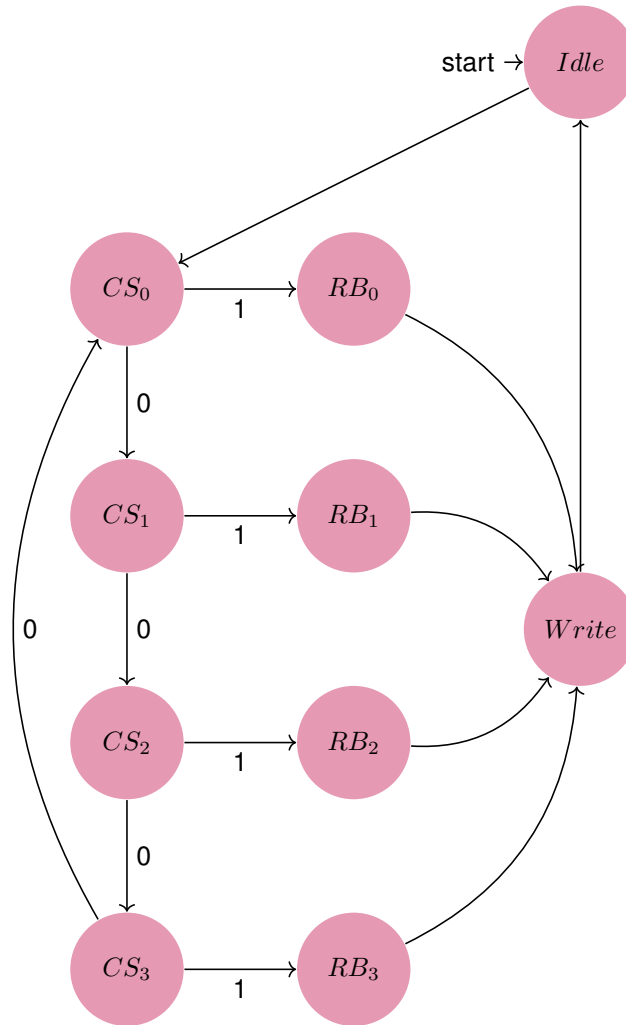


Figure 5.13: Receiver program - Finite state machine

CS - Check Status, RB - Read Bank

0 - No Status update, 1 - Status update received

The FSM starts in the Idle state and then proceeds into a infinite loop of status checks until it finds a

status update. Once a status update is found, the corresponding bank is read from the memory bank into a local buffer. The buffer is then written is padded with a 16-bit number representing the channel number (0 or 1 or 2 or 3). Once the bank is written to a file, the state machine falls back to Idle state momentarily before proceeding into an infinite loop of status polling.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, I will discuss the various experiments performed to prove that our transmitter and receiver systems are working at the intended speeds. I will also discuss a successful key exchange (not completely secure) between the systems that yielded a QBER of 7.5%.

6.1 RECEPTION SPEED TEST

The maximum input speed we want our receiver to be handling reliably is at 25 MHz on all 4 input channels. To test our reception speeds reliably, I setup the test-bench show in figure 6.1. I developed a SPCM simulator that emulates the outputs from the SPCM (refer 5.5). I can simulate an arbitrary number of hits received on an SPCM at its maximum capacity. Each hit is time-tagged with 2 bytes of memory, I can calculate the file size to be expected. The results are presented in table 6.1.

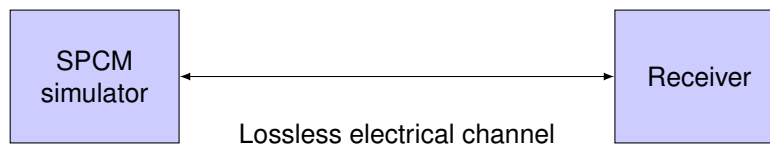


Figure 6.1: Speed test setup for Receiver

The receiver was designed with 64 kB buffer memory to support maximum output frequency (25 MHz)

from the SPCM. However, running this experiment I found that the system cannot handle maximum frequency output from all the four channels. This result can be seen clearly in table 6.1. A 64 kB buffer size will give us two virtual banks of 32 kB. Each virtual bank can store 16 kilo-hits. For the un-successful case from table 6.1, the receiver has to complete transfer of 512 virtual banks from the buffer to the SD card in less than 0.7 seconds. This excessive load of transfers was the cause of bottleneck in the system.

Increasing the buffer memory to 128 kB clears the bottleneck of the system. We can increase the buffer memory further, but we quickly hit the limit of 768 kB of total On-chip memory size synthesizable on the Cyclone V FPGA. There are other optimizations that can be performed on the bare-metal program side, such as ensuring the multiple virtual banks are buffered before performing a write operation. However, I did not implement this optimization for the work presented in this thesis.

SPCM simulator (25 MHz)	Buffer size	Hits sent (per channel)	Received file size	Success
Single channel	64 kB	1 Mega-hits	2 MB	✓
Single channel	64 kB	2 Mega-hits	4 MB	✓
4 channels	64 kB	2 Mega-hits	15 MB	X
4 channels	128 kB	4 Mega-hits	32 MB	✓
4 channels	128 kB	8 Mega-hits	64 MB	✓

Table 6.1: Receiver speed test results

Maximum speed achievable - 25 MHz on all four channels

6.2 TRANSMISSION SPEED TEST

Testing the transmitter was complex than the receiver. Having a lossy free space channel made it hard to quantify transmission speeds reliably. To test our speeds reliably we setup the test-bench show in figure 6.2, with a fiber optical channel with a mean photon number equal to 1. Having a reliable fiber optical channel with a known loss helped us characterize our transmitter with much higher reliability.

The test could have been setup to test the electrical pulses generated from the transmitter. But the electrical pulse to optical pulse conversion is not linear i.e. The exact time of single photon hitting the SPAD is probabilistic, while the electrical pulse is deterministic. So, I decided to test the transmitter with the LEDs.

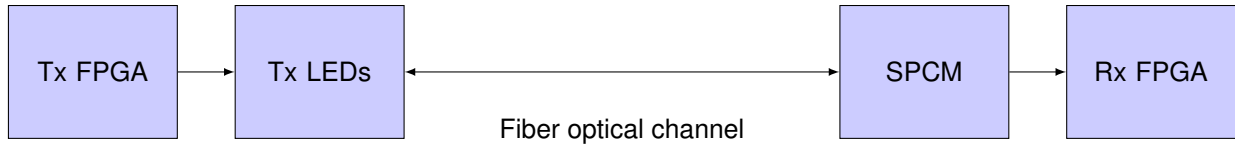


Figure 6.2: Speed test setup for Transmitter

Mean photon number = 1

I conducted two experiments on this setup to test the transmitter. First, to send a known number of hits and calculate the expected file size at the receiver. Second, transmit a pseudo-random bit sequence and compare the output at the receiver to match the sequence. Running both the experiments I was able to conclude that the transmitter was working as expected.

The results for the first method are presented in table 6.2. Running the experiment with a mean photon number of 1, including a dark count rate of 500 counts/sec we expect the received data to be closer, but lesser than the total file size expected. We run the receiver for roughly the same amount of time as the transmitter, ensuring we don't collect dark counts outside the transmission region. The results were as expected, receiving one virtual bank of data less than the expected in the first case and receiving two virtual banks of data less than expected in the second case.

Transmitter (25 MHz)	Hits sent (per channel)	Received file size	Success
4 channels	4 Mega-hits	32,704 kB (64 kB less than 32 MB)	✓
4 channels	8 Mega-hits	65,408 kB (128 kB less than 64 MB)	✓

Table 6.2: Transmitter speed test results

The second method is a brute-force method, I transmitted a known pseudo-random bit sequence. Collected the data file from the receiver and manually matched the data with the pseudo-random bit sequence. With this test being successful, I concluded that the transmitter is working as intended.

6.3 KEY EXCHANGE EXPERIMENT

For this experiment I mounted the transmitter and receiver modules on a standard optical table available in most optical-physics labs. The transmitter output and the receiver input are aligned to each other manually. To conduct the experiment, I generated a large pseudo-random bit sequence (10 Mb) using a linear feedback shift register (LFSR) and ran the transmission experiment. By manually looking at small portion of the data and confirming that the reception was successful, I converted the time-tagged data into a bit-sequence. Then we ran the sifting algorithm using the known pseudo-random bit sequence from the LFSR and the received data to extract a sifted key and the QBER.

Length of transmission (in bits)	1048567
Length of received raw key (in bits)	86622
Length of sifted key (in bits)	35689
Quantum bit error rate (QBER)	7.65%

Table 6.3: Key exchange results using a Pseudo-random bit sequence

A part of the sifted key from both the transmitter and the receiver is presented below.

Listing 6.1: Sifted key from the transmitter

```
1 HRRRHLRLLRRRHRHHLHRLHLRRRHHRRHHRLRLHHLHHHLHLHHHLLRLHHHLRLRLRHLLRLHLRRHHRRHLRHR
```

Listing 6.2: Sifted key from the receiver

```
1 HRRRHLRLLRRRHRHVLHRLHLRRRHHRRHVLRLHHLHHHLHLHHVLLRLHHRRRLRLRHLLRLHLRRHHRRHLRHR
```


CHAPTER 7

CONCLUSIONS

To conclude, we built a QKD transceiver system that has a low SWaP and can be easily deployed on drones. It operates at a maximum frequency of 25 Mhz. We were able to use low power RC LEDs (650 nm) and FPGAs to implement indistinguishable light sources, that can operate at the maximum frequency of operation. We were also able to use a single LED source to generate both the signal and decoy states. The transmitter and the receiver FPGA modules can transmit and receive at the maximum operating frequency without sacrificing any system performance. The system is capable of transmitting keys through the quantum channel and has a QBER of 7.65%.

In the future, the transmitter and the receiver FPGA modules can be improved in a number of ways discussed below.

The transmitter can use interrupts to implement the data transfer system instead of a polling method, which will alleviate a lot of burden from the HPS caused by continuous polling. A bigger buffer memory can be used to accommodate more data in a single transfer. Better delay and width control methods can be explored to get better resolution. The bare-metal program can be improved by adding automatic start-up system instead of a manual start-up system. The transmitter can also benefit from having a remote control on the Linux system to run different experiments while the drones are in-flight.

The receiver can also be designed with an interrupt system instead of a polling system, which will alleviate a lot of burden from the HPS. Better time-to-digital converter system can be explored for better timing resolution. Better encoding schemes for the time-tagged data can save storage space and increase our maximum key storage capacity.

BIBLIOGRAPHY

- [1] W. K. Wootters and W. H. Zurek. “A single quantum cannot be cloned”. In: *Nature* 299.5886 (Oct. 1982), pp. 802–803. ISSN: 1476-4687. DOI: 10.1038/299802a0. URL: <https://www.nature.com/articles/299802a0> (visited on 04/12/2020).
- [2] E. F. Schubert et al. “Resonant cavity light-emitting diode”. In: *Applied Physics Letters* 60.8 (Feb. 24, 1992), pp. 921–923. ISSN: 0003-6951, 1077-3118. DOI: 10.1063/1.106489. URL: <http://aip.scitation.org/doi/10.1063/1.106489> (visited on 04/15/2020).
- [3] P.W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Proceedings 35th Annual Symposium on Foundations of Computer Science. Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.
- [4] Roel G. Baets et al. “Resonant-cavity light-emitting diodes: a review”. In: *Light-Emitting Diodes: Research, Manufacturing, and Applications VII*. Light-Emitting Diodes: Research, Manufacturing, and Applications VII. Vol. 4996. International Society for Optics and Photonics, July 3, 2003, pp. 74–86. DOI: 10.1117/12.476588. URL: <https://www.spiedigitallibrary.org/conference-proceedings-of-spie/4996/0000/Resonant-cavity-light-emitting-diodes-a-review/10.1117/12.476588.short> (visited on 04/15/2020).
- [5] M. Ben-Or et al. “The Universal Composable Security of Quantum Key Distribution”. In: *arXiv:quant-ph/0409078* (Sept. 13, 2004). version: 1. arXiv: quant-ph/0409078. URL: <http://arxiv.org/abs/quant-ph/0409078> (visited on 04/16/2020).
- [6] Hoi-Kwong Lo, Xiongfeng Ma, and Kai Chen. “Decoy State Quantum Key Distribution”. In: *Physical Review Letters* 94.23 (June 16, 2005), p. 230504. DOI: 10.1103/PhysRevLett.94.230504. URL: <https://link.aps.org/doi/10.1103/PhysRevLett.94.230504> (visited on 11/08/2019).
- [7] Daniel J. Bernstein. “Introduction to post-quantum cryptography”. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer, 2009,

- pp. 1–14. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_1. URL: https://doi.org/10.1007/978-3-540-88702-7_1 (visited on 04/24/2020).
- [8] Valerio Scarani et al. “The Security of Practical Quantum Key Distribution”. In: *Reviews of Modern Physics* 81.3 (Sept. 29, 2009), pp. 1301–1350. ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.81.1301. arXiv: 0802.4155. URL: <http://arxiv.org/abs/0802.4155> (visited on 03/20/2020).
- [9] Tom Close, Erik M. Gauger, and Brendon W. Lovett. “Overcoming phonon-induced dephasing for indistinguishable photon sources”. In: *New Journal of Physics* 14.11 (Nov. 2012). Publisher: IOP Publishing, p. 113004. ISSN: 1367-2630. DOI: 10.1088/1367-2630/14/11/113004. URL: <https://doi.org/10.1088%2F1367-2630%2F14%2F11%2F113004> (visited on 04/16/2020).
- [10] Tobias Heindel et al. “Quantum key distribution using quantum dot single-photon emitting diodes in the red and near infrared spectral range”. In: *New Journal of Physics* 14.8 (Aug. 2012). Publisher: IOP Publishing, p. 083001. ISSN: 1367-2630. DOI: 10.1088/1367-2630/14/8/083001. URL: <https://doi.org/10.1088%2F1367-2630%2F14%2F8%2F083001> (visited on 04/15/2020).
- [11] Andreas Beling and Joe C. Campbell. “Chapter 3 - Advances in Photodetectors and Optical Receivers”. In: *Optical Fiber Telecommunications (Sixth Edition)*. Ed. by Ivan P. Kaminow, Tingye Li, and Alan E. Willner. Optics and Photonics. Boston: Academic Press, Jan. 1, 2013, pp. 99–154. DOI: 10.1016/B978-0-12-396958-3.00003-2. URL: <http://www.sciencedirect.com/science/article/pii/B9780123969583000032> (visited on 03/29/2020).
- [12] Charles H. Bennett and Gilles Brassard. “Quantum cryptography: Public key distribution and coin tossing”. In: *Theoretical Computer Science* 560 (Dec. 2014), pp. 7–11. ISSN: 03043975. DOI: 10.1016/j.tcs.2014.05.025. arXiv: 2003.06557. URL: <http://arxiv.org/abs/2003.06557> (visited on 03/29/2020).
- [13] L. C. Comandar et al. “Near perfect mode overlap between independently seeded, gain-switched lasers”. In: *Optics Express* 24.16 (Aug. 8, 2016), p. 17849. ISSN: 1094-4087. DOI: 10.1364/OE.24.017849. URL: <https://www.osapublishing.org/abstract.cfm?URI=oe-24-16-17849> (visited on 04/16/2020).
- [14] Shi-Hai Sun et al. “Experimental Demonstration of Passive-Decoy-State Quantum-Key-Distribution with Two Independent Lasers”. In: *Physical Review A* 94.3 (Sept. 27, 2016), p. 032324. ISSN: 2469-9926, 2469-9934. DOI: 10.1103/PhysRevA.94.032324. arXiv: 1609.02653. URL: <http://arxiv.org/abs/1609.02653> (visited on 04/16/2020).
- [15] Horace P. Yuen. “Security of Quantum Key Distribution”. In: *IEEE Access* 4 (2016). Conference Name: IEEE Access, pp. 724–749. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2528227.

- [16] Hyunchae Chun et al. "Handheld free space quantum key distribution with dynamic motion compensation". In: *Optics Express* 25.6 (Mar. 20, 2017), p. 6784. ISSN: 1094-4087. DOI: 10.1364/OE.25.006784. URL: <https://www.osapublishing.org/abstract.cfm?URI=oe-25-6-6784> (visited on 04/15/2020).
- [17] Heasin Ko et al. "Critical side channel effects in random bit generation with multiple semiconductor lasers in a polarization-based quantum key distribution system". In: *Optics Express* 25.17 (Aug. 21, 2017), p. 20045. ISSN: 1094-4087. DOI: 10.1364/OE.25.020045. arXiv: 1706.08705. URL: <http://arxiv.org/abs/1706.08705> (visited on 04/16/2020).
- [18] Andrés Aragonese et al. "Bounding the outcome of a two-photon interference measurement using weak coherent states". In: *Optics Letters* 43.16 (Aug. 15, 2018), p. 3806. ISSN: 0146-9592, 1539-4794. DOI: 10.1364/OL.43.003806. URL: <https://www.osapublishing.org/abstract.cfm?URI=ol-43-16-3806> (visited on 04/16/2020).
- [19] Guiping Cao, Haojie Xia, and Ning Dong. "An 18-ps TDC using timing adjustment and bin realignment methods in a Cyclone-IV FPGA". In: *Review of Scientific Instruments* 89.5 (May 2018), p. 054707. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.5008610. URL: <http://aip.scitation.org/doi/10.1063/1.5008610> (visited on 04/15/2020).
- [20] Ke Cui, Xiangyu Li, and Rihong Zhu. "A high-linearity time-to-digital converter based on dynamically delay-adjustable looped carry chains on FPGAs". In: *Review of Scientific Instruments* 89.8 (Aug. 2018), p. 084704. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.5038146. URL: <http://aip.scitation.org/doi/10.1063/1.5038146> (visited on 04/15/2020).
- [21] Nurul T. Islam. *High-Rate, High-Dimensional Quantum Key Distribution Systems*. Springer Theses. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-98928-0 978-3-319-98929-7. DOI: 10.1007/978-3-319-98929-7. URL: <http://link.springer.com/10.1007/978-3-319-98929-7> (visited on 03/17/2020).
- [22] Nurul T. Islam et al. "Securing quantum key distribution systems using fewer states". In: *Physical Review A* 97.4 (Apr. 30, 2018). Publisher: American Physical Society, p. 042347. DOI: 10.1103/PhysRevA.97.042347. URL: <https://link.aps.org/doi/10.1103/PhysRevA.97.042347> (visited on 04/15/2020).
- [23] Gwenaelle Mélen et al. "Handheld Quantum Key Distribution". In: *Conference on Lasers and Electro-Optics (2018), paper FTu3G.1*. CLEO: QELS_Fundamental Science. Optical Society of America, May 13, 2018, FTu3G.1. DOI: 10.1364/CLEO_QELS.2018.FTu3G.1. URL: https://www.osapublishing.org/abstract.cfm?uri=CLEO_QELS-2018-FTu3G.1 (visited on 04/16/2020).

- [24] (:Unav). "Employing FPGA DSP blocks for time-to-digital conversion". In: (2019). Publisher: Polish Academy of Sciences Committee on Metrology and Scientific Instrumentation. DOI: 10.24425/MMS.2019.130570. URL: <http://journals.pan.pl/dlibra/publication/130570> (visited on 04/15/2020).
- [25] Annika Dugad et al. "Investigating the Effectiveness of Measurement-Device-Independent Quantum Key Distribution with Weak Coherent Pulses". In: (2019). Conference Name: APS Meeting Abstracts, G70.385. URL: <http://adsabs.harvard.edu/abs/2019APS..MARG70385D> (visited on 04/15/2020).
- [26] Haojie Xia, Guiping Cao, and Ning Dong. "A 6.6 ps RMS resolution time-to-digital converter using interleaved sampling method in a 28 nm FPGA". In: *Review of Scientific Instruments* 90.4 (Apr. 2019), p. 044706. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.5084014. URL: <http://aip.scitation.org/doi/10.1063/1.5084014> (visited on 04/15/2020).
- [27] Feihu Xu et al. "Secure quantum key distribution with realistic devices". In: *arXiv:1903.09051 [quant-ph]* (Feb. 19, 2020). arXiv: 1903.09051. URL: <http://arxiv.org/abs/1903.09051> (visited on 04/16/2020).
- [28] intel. "IOPLL Intel® FPGA IP Core User Guide". In: (), p. 13. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altera_iopll.pdf (visited on 04/16/2020).
- [29] intel intel intel. "Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide". In: (), p. 50. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram_rom.pdf (visited on 04/16/2020).
- [30] Elmer Perkin. "Single Photon Counting Module Array". In: (), p. 8. URL: <https://www.pacer-usa.com/Assets/User/398-SPCM-AQ4C.pdf> (visited on 04/19/2019).
- [31] R L Rivest, A Shamir, and L Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: (), p. 15.