# واحد فوٹون اور ایف پی جی اے (FPGA) کی مدد سے چند کوانٹم نوری تجربے

# Some Quantum Optical Experiments with Single Photons and FPGA Programming

*Syed Bilal Hyder Shah*

*Supervised by Dr Muhammad Sabieh Anwar*

May 2022

# خلاصہِ تحقیق

سنگل فوٹون کی تحقیق پر مبنی اس مقالہ کو تین ابواب مین تقسیم کیا گیا ہے۔

**باب اول:** تجرباتی آلات کا دورہ۔

لیب میں آنے والے کسی بھی نئے طالبِ علم کی پہلے سے ہی آلات سے واقفیت ہونا بے حد ضروری ہے، اور اِس باب کا مقصد اِس واقفیت تو یقینی بنانا ہے۔ اس باب میں اُن تمام آلات کا تعارف کرایا گیا ہے کہ جن کا استعمال سنگل فوٹون سے متعلق تجربات میں کیا جائے گا۔ اس باب کو پڑہنے کے بعد طالبِ علم تجرباتی آلات سے اِس حد تک متعرف ہو جائیں گے کہ وہ بلا جھجک تجربہ گاہ میں کام شروع کر سکیں۔

**باب دوم:** یہ باب ان ۴ تجربات کے خلاصون اور نتائج پر مبنی ہے جن میں ہم سنگل فوٹون کی شہتیر کا استعمال کرتے ہین۔ ان تجربات کے نام درج ذیل ہیں۔

تجربہ نمبر ۱- بے ساختہ پیرامیٹرک ڈاون کنورژن۔

تجربہ نمبر ۲- فوٹون کے وجود کا ثبوت۔

تجربہ نمبر ۳- سنگل فوٹون کے پولرائزیشن کی حالت کا یندازہ لگانا۔

تجربہ نمبر ۴- سنگل فوٹون انٹرفیرنس اور کوانٹم ایریزر۔

اِن تمام تجربات کا مقصد سنگل فوٹون کی کوانٹم فطرت کے لہری یا جزوی ہونے کا موازنہ کرنا ہے۔

**باب سوم:** اس باب میں اُن تجربات کے خلاصے اور نتائج تحریر ہیں کہ جن کا مقصد ہم دو الگ الگ، سنگل فوٹون شہتیرون کے مابین کوانٹم انٹینگلمنٹ کے مظاہرے کا مشاہدہ کرنا ہے۔ اِن تجربات کے نام درج ذیل ہیں۔

تجربہ نمبر۱: فریڈمین کی مقامی حقیقت پسندی کی جانچ۔

تجربہ نمبر۲: سی ایچ ایس ایچ کی مقامی حقیقت پسندی کی جانچ۔

**باب چہارم:** یہ باب ایف پی جی اے کی پروگرامنگ کے لیے مرحلہ وار رہنمائی ہے، خاص طور پر اِن سنگل فوٹون کے اتفاق پر مبنی تجربات کے لیے کہ جس طرح کے ہم اپنی تجربہ گاہ میں سرانجام دیا کرتے ہیں۔ یہ باب انتہائی صارف دوست انداز میں لکھا گیا ہے، تاکہ کوئی بھی پروگرامنگ یا ایف پی جی اے سے ناواقف شخص اِس باب کو پڑھنے کے بعد ایف پی جی اے کو باآسانی پروگرام کر سکے۔

مقالے کے اختتام پر موجود اپینڈکس میں پایتھان میں لکھے گئے ایک پروگرام کا کوڈ موجود ہے۔ اِس پروگرام کا مقصد سنگل فوٹون کی گنتی کو ایف پی جی اے سے لے کر کمپیوٹر کی سکرین پر ایک گرافیکل یوزر انٹرفیس کی شکل میں پیش کرنا ہے۔

# ACKNOWLEDGEMENTS

By no means can I claim this achievement all for myself. My first debt is to Dr Muhammad Sabieh, my supervisor for this project. I want to thank him for allowing me to work in his lab and entrusting me with this project. Without his guidance, support, and trust, I could never complete this project. I want to thank the PhysLAB team for their help whenever I needed it, and a special thanks to Umar Hassan, who spent hours teaching me Verilog for FPGA programming.

I dare not forget to mention my peers and friends. I extend my gratitude to Haider Khan and Mariam Arif for being my support and motivation throughout this journey. A special thanks to my friend Mahid Anjum who helped me compile and proofread the thesis. Thanks to my juniors, too, for being such a lovely and caring bunch and keeping me full of spirit.

Finally, I would like to thank my parents and siblings for keeping up with me and supporting me with everything I do.

# Contents

# Chapter 1

# Tour of the experimental apparatus

Knowing the equipment one has to work with is perhaps the most crucial part of an experiment. This chapter serves as a tour of the experimental apparatus for the single-photon experiments. After going through this chapter, the newcomers would not feel alienated in the lab with the unknown equipment.

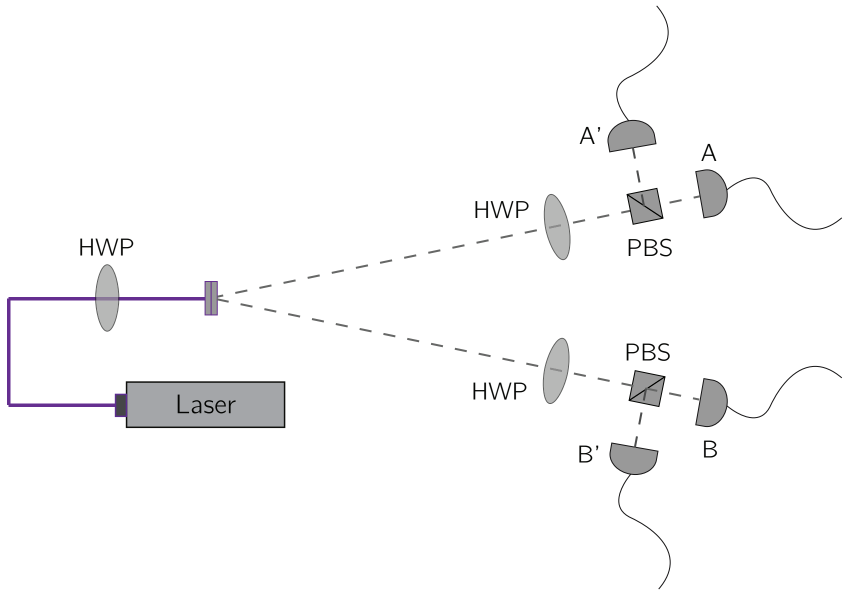


Figure 1.1: *Schematic for an experiment employing four detectors.*

We start by giving a whole overview of the experimental apparatus. Figure 1.1 shows a schematic of an experiment that uses four detectors. We will see many similar schematics in later chapters, but in this chapter we will see what the equipment drawn in this schematic actually looks like.

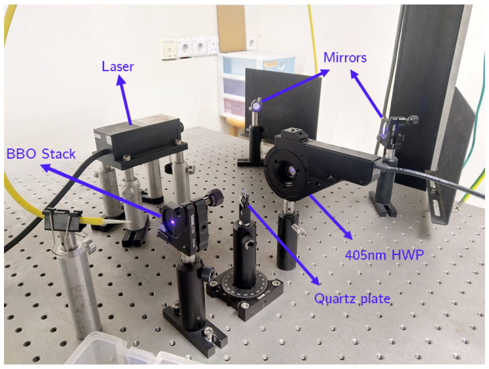
We split the overview of physical components into two parts, the first part deals with the apparatus that is concerned with the production of single photons that we are going to manipulate by inserting different optical components in their path. The second part deals with the apparatus that will collect the single photons after they have passed through the desired optical components. In both parts we discuss the location and at which each component is to be placed with respect to the laser beam, followed by some basic techniques of optical alignment.

After the overview of the experimental apparatus as a whole, we take some time to go into brief details of each and every component. Every component in the setup plays a vital role, missing out on the functional details of any component can result in undesired results in the experimental data that we produce and may damage the equipment as well.

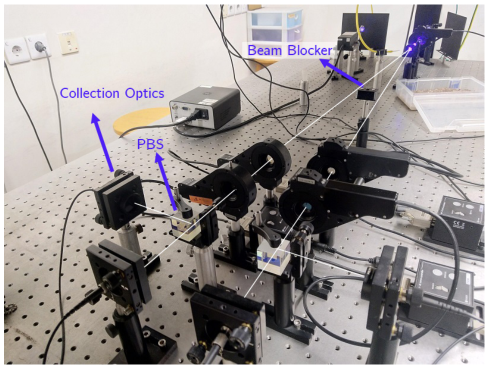
## 1.1 Overview

We use the equipment shown in figure 1.13(c) to make a single photon source for the experiments. Mounting the laser onto a stable stand is not enough to ensure a laser beam that is travelling is a straight line over a long distance. There may be some tilt in the laser beam as it travels over the optical breadboard. To ensure a perfectly straight laser beam without any tilt, we reflect it off two mirrors mounted on kinematic mounts to align it such that the beam passes through the HWP and hits the BBO stack. We can use an alignment ruler to check for any tilt in the beam as demonstrated by ThorLABS insight in this video demonstration. [1]

Figure 1.13(b) shows the apparatus that we use to collect the generated single photons. The BBO stack sends off two simultaneously produced single photons at an angle of 3° from the pump beam. Thus, the apparatus in this portion is set in the path of those single photons. The alignment of this portion is a bit tricky because the the single photons produced by the BBO stack lie in the near-infrared (NIR) regime, and we can not see even a trace of it. We make calculations for approximate position for the detectors and then use back propagation lasers to make sure that these are aligned properly. The height of all the detectors should be equal to the pump beam height to collect the photons efficiently. After collection the photons are taken to the single photon detectors using optical fibres.



(a) *Photon generation.*



(b) *Photon collection.*

Figure 1.2: *4 detector setup with all the components aligned. (a) shows the single photon producing part of the apparatus. This part comprises of the pump laser beam, an HWP, a Quartz crystal, and a stack of BBO crystals. (b) shows the measurement part of the apparatus. This part comprises of beam splitters, wave plates, and collection optics.*

## 1.2 Pump laser

The silicon-based 405 nm pump beam is the powerhouse for all our experiments. It is a class 3B laser with a maximum power output of 500 mW. The power supply it is connected to delivers a max power of 75 mW to the laser head, which is the limit for our experimental apparatus. Class 3B lasers can cause severe damage to the eye in case of direct contact; hence safety goggles must be put on while the laser is powered on.

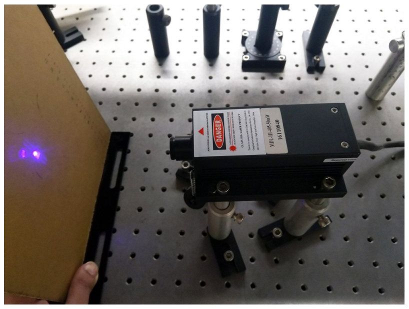


Figure 1.3: *A 405 nm violet laser beam being used to pump photons into the BBO crystal.*

## 1.3 BBO crystal stack

The two Barium Beta Borate (BBO) crystals are situated in the middle of the circular disk mounted on the kinematic mount. The pump laser is aligned such that it hits directly on the BBO; otherwise, it will go to waste. The BBO is the part of our apparatus responsible for generating single photons for all the experiments we will be performing.

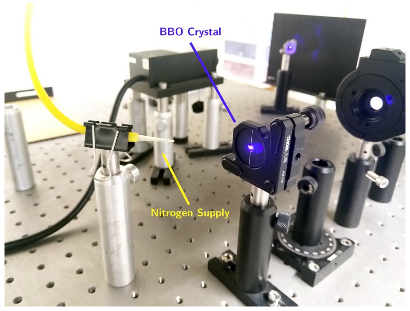


Figure 1.4: *405 nm pump beam hitting the BBO crystal stack. BBO crystals down-convert 405nm photons into two 810nm photons.*

BBO crystals are hydrophobic and should be protected from moisture all the time. We have a continuous nitrogen supply on the BBO crystals for moisture protection. The yellow pipe with a micro-pipette nozzle shown in the picture above is connected to an $N_2$ cylinder outside the lab. Using a nozzle makes sure that we get an adequate flow of $N_2$ over the crystal stack with minimum wastage. We also place silica gel beads near the BBO to act as sacrificial protection, and the lab is air-conditioned to minimize moisture.

## 1.4 Photon collectors

These collectors comprise F220FC-780 collimators, ideal for applications involving light in the near-infrared (NIR) regime, mounted on kinematic mounts and covered with 780 nm long-pass filters. The photons we aim to collect in these experiments are of wavelength 810 nm, so we use the long-pass filters to filter any ambient light that might interfere with the experimental results. Another reason for using these filters is to limit the amount of light going to the detectors as excess light can damage the highly sensitive single-photon detectors. The collimators are followed by optical fibres taking the captured light to the single-photon detectors. Fibre optics are then connected to the F220FC-780 collimators to carry the photons from the collection optics to the single photon detectors.

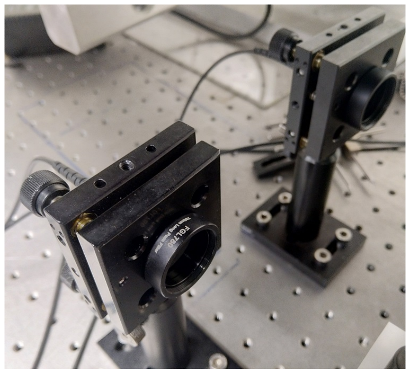


Figure 1.5: *Photon collection setup.*

## 1.5 Single photon counting modules

The Single photon counting modules (SPCMs), shown in the left portion of figure 1.6, contain avalanche photo-diodes. Whenever they receive a photon, they produce an electric pulse in response to it. When terminated by 50 Ω resistance, the output of these SPCMs is a 20 ns square wave with a peak voltage of 2.2V. The device on the right side of the above image is the power supply of the SPCMs rated at 5V and 330mA. Since the SPCMs are highly sensitive, and excess light can damage them, we take extra caution to turn on the intended detector. To avoid confusion regarding which power supply will turn on which detector, the detectors and their respective power supplies are labelled using the same colour.

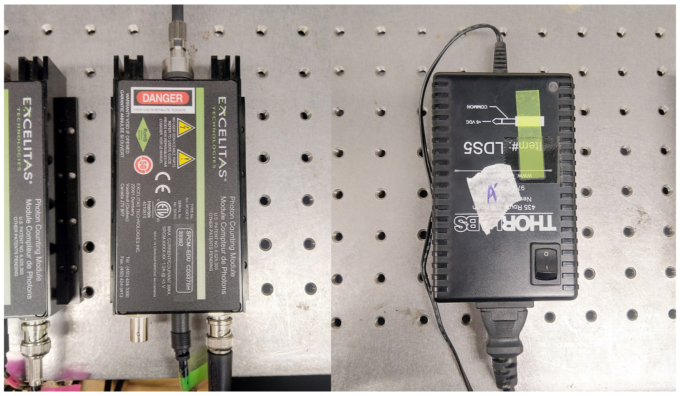


Figure 1.6: *An SPCM on the left and its power supply on the right.*

## 1.6 Polarizing beam splitters

In these experiments, we are using PBS252 polarizing beam splitters from ThorLABS. The 25 in the naming convention means that each edge of these PBSs is 25 mm, and the 2 at the end signifies that the beam splitter is designed to work ideally in the 620-1000 nm range (the NIR regime). These are polarizing beam splitters which means that they split the light based on the polarization state of the incoming beam. The ones that we are using in lab transmit the $|V\rangle$ polarized photons, and reflect the $|H\rangle$ polarized photons.

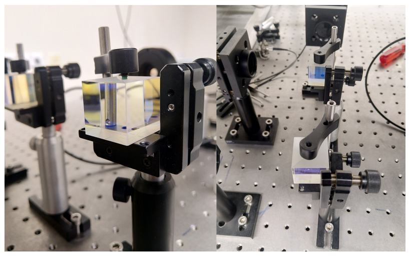


Figure 1.7: *PBS 252 beam splitter shown in different orientations.*

## 1.7 Waveplates

Waveplates are an essential part of all the experiments in this book. They allow us to control the polarization of incoming light. They come in different sizes and mounts. The one shown in the picture is from ThorLABS and comprises a 0.5” diameter crystal mounted in a jacket of 1” diameter. We can place this waveplate into any mount with a diameter to support 1” optical equipment. There are two kinds of waveplates, half waveplates (HWPs) and quarter waveplates (QWPs), and they are quoted with the values of wavelength of light they are designed to work best with.

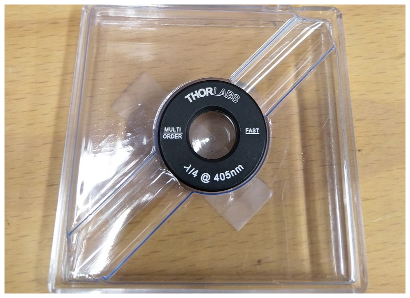


Figure 1.8: *A 405nm multi-order $\lambda/4$ waveplate from ThorLABS.*

## 1.8 Beam displacing prisms

The Beam displacing prisms (BDPs) are a particular variant of beam splitters. Instead of reflecting one component of light, they displace it by 4 mm. These BDPs have a fascinating use case in one of the experiments when we make a Mach Zehnder interferometer using them.

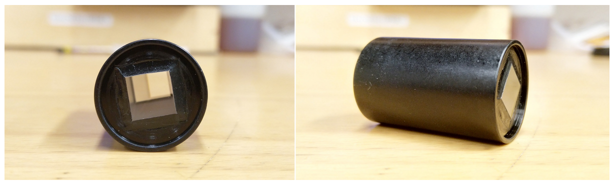
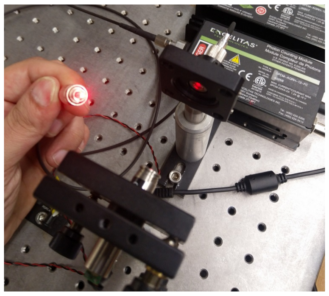
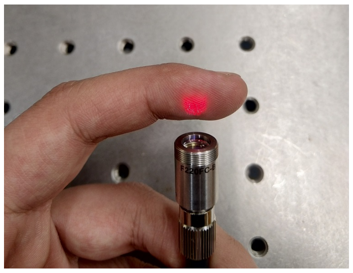


Figure 1.9: *A BD40 beam displacing prisms from ThorLABS.*

## 1.9 Back propagation laser

For back-propagation alignment, we use a 633 nm class 3R laser. It has a power rating of only 5 mW which is not very dangerous. However, direct contact with the eye should be avoided. The laser is coupled into one end of a fibre optic and it passes through to the other end. The output light is not collimated and spreads out as soon as it exits the fibre optic. A collimator can be used to converge the spreading light into a Gaussian beam.


(a) *Laser light travelling through an optic fibre.*


(b) *A collimator converging the light output from the fibre optic into a Gaussian beam.*

## 1.10 Motorized rotation stage and controller

This combination of motorized stage and DC servo controller has made the life of optical scientists a lot easier. This motorized stage has a diameter of 1" so we can attach a wide range of optical instruments with it. Along with the controller, it can give very precise rotation angles without even touching the setup.The controller can be controlled using ThorLABS Kinesis software with great ease. A screenshot of the Kinesis interface is shown in figure 1.12.

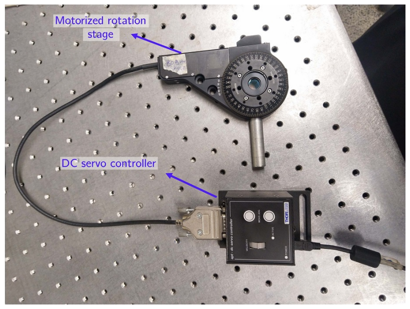
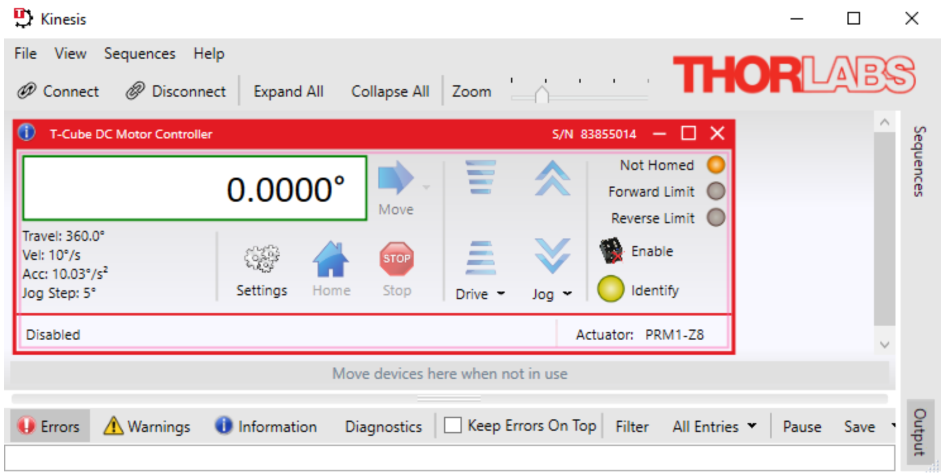


Figure 1.11: *A PRM1Z8 motorized stage connected with a TDC001 DC servo controller.*


Kinesis
File View Sequences Help
Connect Disconnect Expand All Collapse All Zoom
THORLABS
T-Cube DC Motor Controller
S/N 83855014
Move
Settings Home Stop
Drive Jog
Not Homed
Forward Limit
Reverse Limit
Enable
Identify
Move devices here when not in use
Errors Warnings Information Diagnostics Keep Errors On Top Filter All Entries Pause Save
Sequences
Output

0.0000°

Travel: 360.0°
Vel: 10°/s
Acc: 10.03°/s²
Jog Step: 5°

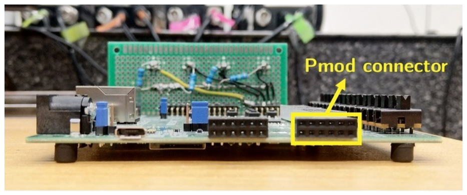
Disabled

Actuator: PRM1-Z8

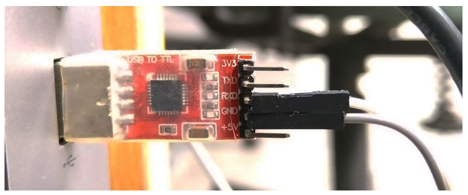
Figure 1.12: *A screenshot of interface of ThorLABS Kinesis.*

## 1.11 Field programmable gate array

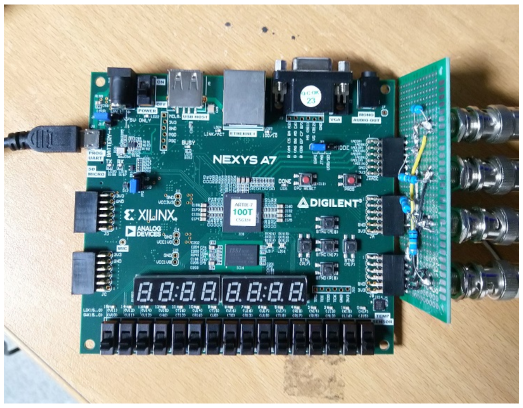
When a pulse is produced by an SPCM, it is carried to the Field programmable gate array (FPGA) using BNC cables. In the FPGA, we have implemented a coincidence counting unit (CCU). We use the Pmod connectors as the input and output interfaces of the FPGA. The data is outputted into a UART serial communication device which transfers data into the PC via a USB port.



(a) *Pmod connector on a Nexys A7.*



(b) *UART serial communication device.*



(c) *Nexys A7 FPGA.*

Figure 1.13: *A physical picture of the FPGA and its accessories.*

## 1.12 Counting software

The counting software we are currently using in the lab is based on Python. The data from the FPGA is received in 7-bit long packets which are then assembled into decimal counts and projected in real-time onto an interactive GUI. The GUI has user-interactive buttons to take a screenshot, collect data for a given time, and safely terminate the communication channel while exiting the software. If the software is closed other than the stop button, the serial communication channel will remain active until it is commanded to close in the terminal.

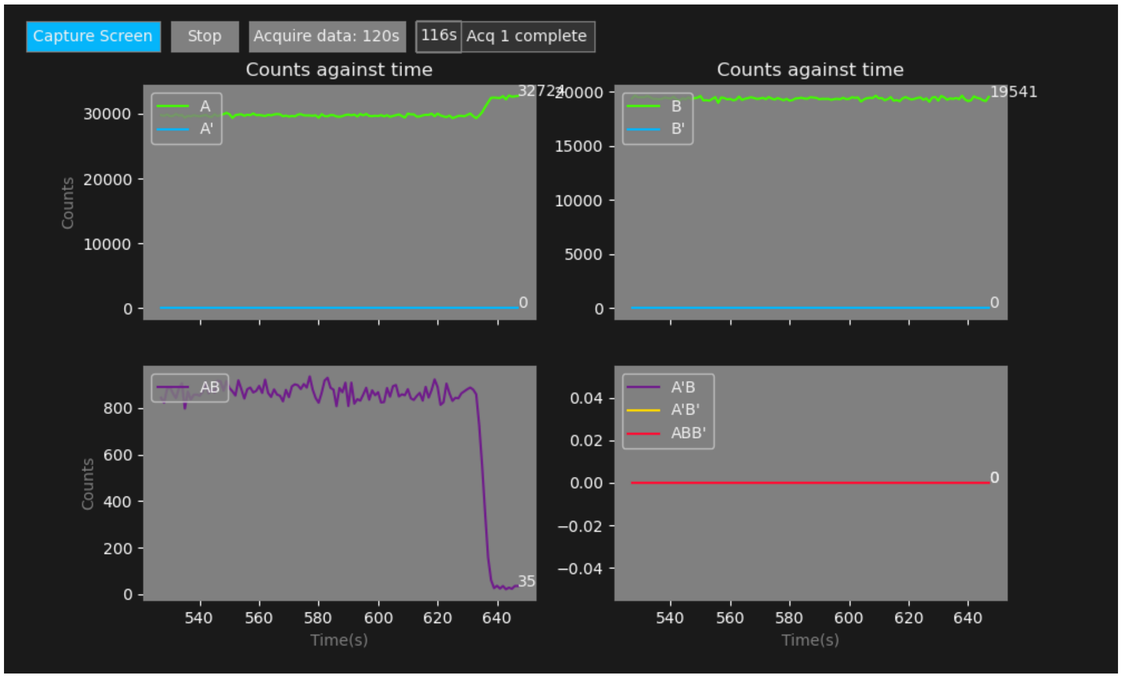


Figure 1.14: *A screenshot of the Python software being used for real-time plotting of counts.*

# Chapter 2

# Single photon experiments: The quantum nature of light

## 2.1 Introduction

This chapter is for the discussion of experiments I performed in the Quantum optics lab. As my guideline, I used the book “Quantum Mechanics in Single Photon Laboratory” and reproduced the results of this book. I produced the results a couple of times. In the first run, I used the old Nexys 2 and LabVIEW. While for the second go, I used the new Nexys A7 and a Python program that I wrote myself for coincidence counting. This chapter will go over the experimental procedure and the results obtained, while the choice of equipment and software lies with the experimenter.

## 2.2 Experiment 1: Spontaneous parametric down-conversion

Spontaneous parametric down-conversion (SPDC) is preliminary to all other experiments to be performed. In this experiment, we will be preparing a single photon beam using BBO crystal and detecting it using our single-photon counting modules (SPCMs). SPDC is the main ingredient for all the experiments outlined in the book I am following. It is the source of the single photons we use in our experiments. The ideal single-photon source would be a photon gun, producing just a single photon whenever needed, but such a device has just resided in human imagination so far. For real-life experiments involving single photons, we need some workarounds. As mentioned in the article by [4], there are many ways of making single-photon sources, some of which include “faint laser pulses”, “entangled photon pairs”, and “Atoms, ions in the gas phase”, “ using organic molecules”. Out of all these methods, parametric down-conversion is the most widely used method worldwide.

SPDC requires non-centrosymmetric crystals like KDP, BBO, LBO or LiNbO3. As the name suggests, this process is spontaneous, i.e. one can not claim for sure if a single photon that is going to hit the crystal will be able to down-convert or not. Only a few thousand out of billions of photons undergo this process. The process inherits the name down-conversion pertaining to the fact that this process converts a photon into two photons of smaller frequency and energy. Lastly, the “parametric” part in the name is because a parameter controls the down-conversion, i.e. the polarization of the incoming photons. The crystals can down-convert photons only for a certain polarization component. If this condition is not met, no matter how much we increase the intensity of the incoming beam, the down-conversion will not take place. In our lab, the crystal we have chosen is BBO (Barium Beta Borate). We stack two of them together at 90° to one another. This way, we can cater for all the polarization of incoming photons. The energy levels in this crystal are such that the electrons in this crystal excite to a high energy level when they absorb a pump photon of 405 nm. However, when these electrons de-excite, they return to the original energy level in a couple of steps, producing photons of equal energy both times. In this way, we split our 405 nm photons into two 810 nm photons. The momentum and energy conservation conditions define the correlation between these two photons [4]. Energy conservation requires,

$$w_p = w_s + w_i, \tag{2.1}$$

and momentum conservation requires,

$$\vec{k_p} = \vec{k_s} + \vec{k_i}. \tag{2.2}$$

$k_p$ and $w_p$ are related to each other by the formula,

$$k_p = \frac{n_p w_p}{c}, \tag{2.3}$$

$n_p$ here is the refractive index of the crystal.

The photons we obtain in SPDC are produced in pairs. We can use this property of photons to make sure what we are receiving are genuinely single photons. We will condition the detection on one detector to the detection on the other detector. This way, we ensure that we have received a photon made due to SPDC. The detection of one photon heralds the detection of the second photon, which is why this mode of detecting single photons is also called *heralded* single-photon detection. Our FPGA has a clock cycle of 10 ns so the time window the two pulses will get to get themselves registered is only $10 \times 2$ ns. A faster FPGA will result in better accuracy of single-photon coincidences detections and improve our results' accuracy. There is a chance of accidental coincidence since we are supplying tens of thousands of photons to the detectors every second. We will need to cater for these counts while using our data for calculations. Accidental counts can be calculated using this formula derived using laws of probability as shown in [2] [6] as,

$$N_{AB}^{(acc)} \approx N_A N_B \Delta t. \tag{2.4}$$

The good thing is that these accidental counts will not be very significant as the coincidence window for our coincidence detection is just 20 ns, while the dead time of SPCMs we are using is 50 ns. To get the real coincidence counts, we must subtract the accidental counts from the raw coincidences as shown in equation 2.5.

$$N^{(real)} = N^{(raw)} - N^{(acc)} \tag{2.5}$$

Furthermore, we need to subtract the background counts from the single detector counts. The data for all the results we produced in the lab went through this data cleaning process.

The two down-converted photons will not necessarily follow the same path as the incoming pump beam. Instead, their path will be determined by the phase-matching angle $\theta_m$, i.e. the angle between the pump beam and the optic axis of the crystal. For the crystal we are using, the phase-matching angle is 30°, giving us a lab angle $\theta_L$ of 3°. These implications need to be considered while purchasing the crystal.

The two output beams of the BBO spread in the form of cones. However, we stick to only the photons emitted horizontally for our experiments and put the detectors in their paths.

### 2.2.1 Optical Alignment

The first step of every optical setup is to align the light source, i.e. the laser. Keeping the laser on a mount is not enough as it is very likely to have some tilt, both horizontally and vertically. For perfect alignment of the laser, we use a couple of mirrors. After the light bounces off these two adjusted mirrors, it will have no tilt, and it can travel in a perfectly straight path on our optical breadboard. We can use an alignment ruler to check if the laser beam follows the desired path. A video on ThorLABS insight gives a lovely demonstration of how to align a laser beam [1].

After we have aligned the laser, we can place other optical elements on the optical table, keeping our laser beam as the reference for their alignment. Schematic shown in figure 2.1 will help us understand the process of alignment.

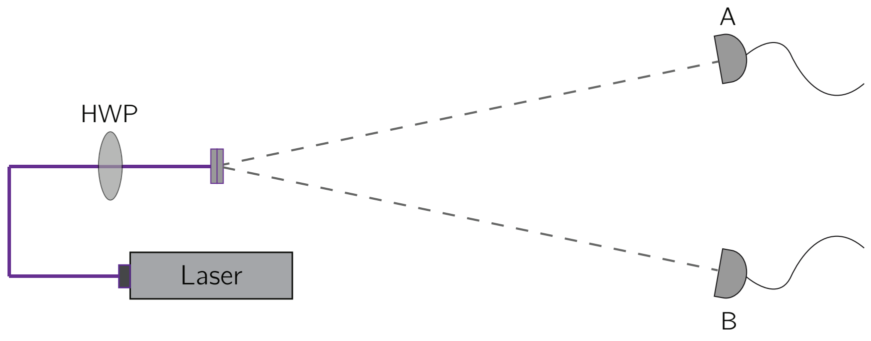


Figure 2.1: *A schematic for experiment 1.*

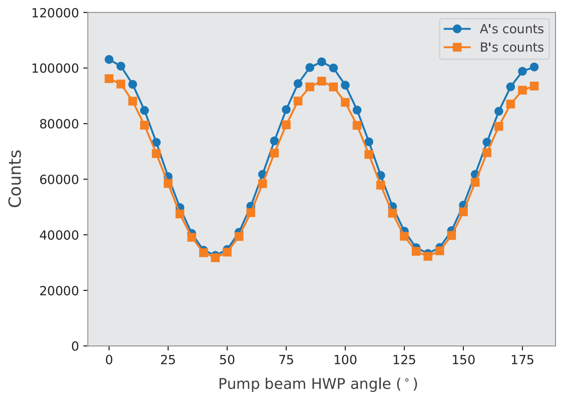
First of all, we shine the laser on the BBO. We have discussed how the output photons leave the BBO crystal by making a 3° angle with the pump beam. These beams contain photons of 810 nm (near-infrared regime), and we can not see them with our naked eyes. Aligning components to a light source that we cannot see is the problem at hand now. To solve this problem, we use the method of back-propagation. Firstly, we need to approximate a position our detectors need to be placed at using simple trigonometry. After placing the detectors on the approximate location, we attach a fibre optic carrying a laser light into the collimator. The collimators output a Gaussian beam that we can use to align the detectors better. We ensure that this light follows a straight path and falls on the BBO crystal. Doing this for both the detectors completes our rough alignment.

### 2.2.2 Experimental method and results

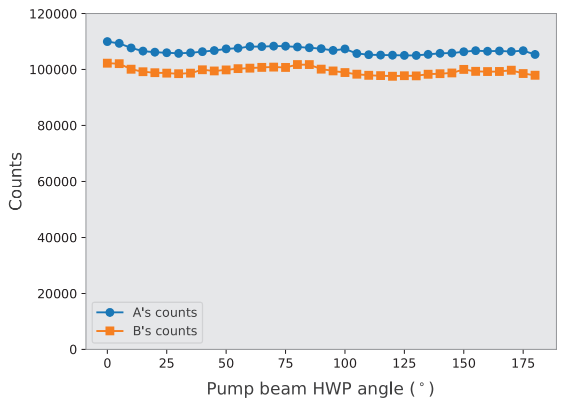
We will remove the back-propagation laser and attach the SPCMs back into the setup to fine-tune the detectors. By this step, we must be receiving some photons in both the detectors. We need to tweak the tilt of the detectors and the BBO crystal using the rotating knobs of the mount to maximize the coincidence detection counts.

So far, we have not changed the polarization of the pump beam, and the input to BBO is $|V\rangle$. This means that the down-conversion was done hitherto by only one of the two BBO crystals. This misalignment will be evident once we change the input polarization of the pump beam by rotating the pump beam HWP. If the other BBO crystal is misaligned, we should get dips in counts when the input state becomes $|H\rangle$. Figure 2.2(a) represents the experimental data supporting this hypothesis.

We have got the dips in counts while rotating the HWP from $0°$ to 180 ° These dips correspond to 45 ° and 135 ° HWP angles when the polarization on input to BBO changes to $|H\rangle$. This shows that the BBO crystal that is responsible for down-converting $|H\rangle$ photons is unaligned. To align this crystal, we give $|H\rangle$ input to the BBO and tune the BBO tilt using the knobs on the kinematic mount until the coincident counts maximize. We will need to turn only one knob of the rotating mount, i.e. horizontal or vertical, as one of the crystals is already aligned. After the alignment is complete, we will rotate the HWP again to check if the two BBO crystals have been properly aligned. Figure 2.2(b) shows the single photon counts on after proper alignment of both BBO crystals in the stack.



(a) *One BBO unaligned.*



(b) *Both BBOs aligned.*

Figure 2.2: *Pump beam HWP angles vs individual detector counts. (a) shows dips in counts with one of the BBO crystals unaligned, while (b) shows the counts with both BBO crystals properly aligned.*

The main object of interest for us are the instances where two detectors click simultaneously. These coincidence detections signify the detection of single photons. No significant dips in the coincidence counts show that the BBO stack is able to down-convert all the input polarization of photons. The graph shown in figure 2.3 compares the variation of coincidence counts with pump beam HWP orientation for when the BBO is aligned and unaligned.

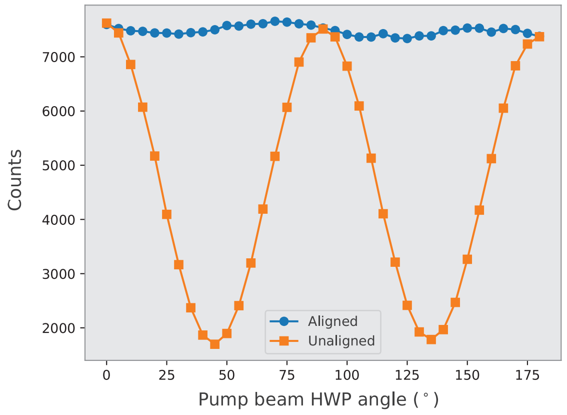


Figure 2.3: *Comparison of single photon (coincidence) counts before and after alignment.*

With our single-photon source aligned and perfectly working, we can move on to perform further experiments on the single-photon beam.

## 2.3 Experiment 2: Proof of the existence of photons

This experiment serves to demonstrate the grainy nature of light and will act as a test of whether the source we have captured in the previous experiment is truly a quantum source or not. For this purpose, we will use a parameter called degree of second-order coherence $g^{(2)}(0)$. If our light source is as quantum as we claim to be, the photons should hit the detectors individually and not in packets. This is called *anti-bunching*. In the case of anti-bunched photons, $g^{(2)}(0)$ should give us a value of 0, and if we have a classical source where we do see bunching, we should get a value of at least 1 with an upper bound of 2 in case of a chaotic light source. The general expression for $g^{(2)}(0)$, as shown in [3], is given by equation 2.6,

$$g^{(2)}(0) = \frac{\langle I(t)^2 \rangle}{\langle I(t) \rangle^2}. \tag{2.6}$$

To check for the grainy nature of light, we will need to make some alterations to the experimental setup.

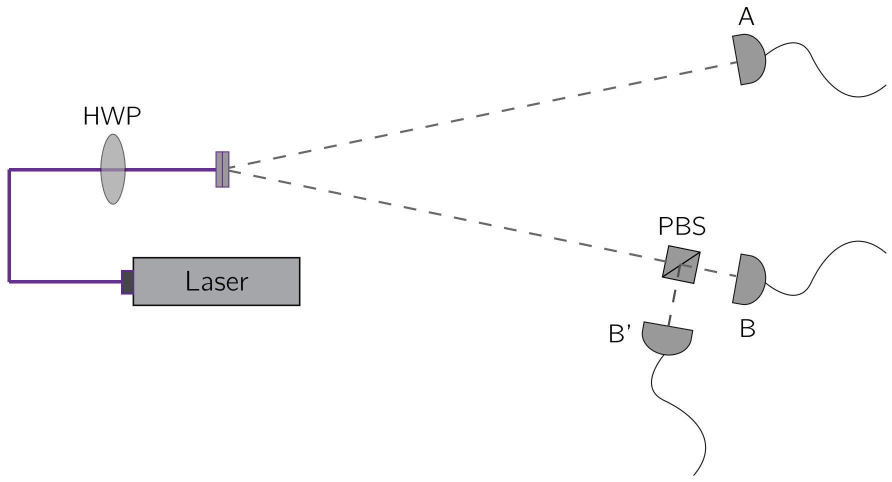


Figure 2.4: *Schematic for experiment 3. Addition of a PBS and a detector to the setup.*

We place a polarizing beam splitter in the path of the idler beam. This beam splitter will help us to distinguish between the photons. If the incoming photon is in state $|V\rangle$, it will go straight through the beam splitter, and if the input state is $|H\rangle$, it will be reflected by the beam splitter. We will include another detector in the setup to detect these reflected photons. The logic behind this experiment is that if we consider the photon as a particle (inspecting the grainy nature of light), it should be able to take only one of the two paths at a time and result in detection on one detector only. The expression for $g^{(2)}(0)$, in this case, can be written in terms of the probabilities of detection. From [6], we find this expression to be,

$$g^{(2)}(0) = \frac{P_{ABB'}(0)}{P_{AB}(0)P_{AB'}(0)}. \tag{2.7}$$

In terms of counts on the detectors, this expression turns to,

$$g^{(2)}(0) = \frac{N_A N_{ABB'}}{N_{AB} N_{AB'}}. \tag{2.8}$$

We will need to incorporate accidental counts in this setup experiment as well. The derivation from [6] shows that the probability of accidental counts comes out to be,

$$\begin{aligned}
P'_{ABB'} &= P^{AB} P'_{B'} + P_{AB'} P'_{B}, \\
&= P_{AB} N_{B'} \Delta t + P_{AB'} N_B \Delta t.
\end{aligned}$$

Here $\Delta t$ is the coincidence time window. In our case, it is 20 ns. We can derive an expression for $g^{(2)}(0)'$ in terms of detector counts as.

$$\begin{aligned}
g^{(2)}(0)' &= \frac{P'_{ABB'}}{P_{AB} P_{AB'}}, \\
&= \frac{P_{AB} N_{B'} \Delta t + P_{AB'} N_B \Delta t}{P_{AB} P_{AB'}}, \\
&= \frac{N_{B'} \Delta t}{P_{AB'}} + \frac{N_B \Delta t}{P_{AB}}, \\
&= N_A \Delta t \left( \frac{N_{B'}}{N_{AB'}} + \frac{N_B}{N_{AB}} \right).
\end{aligned}$$

This shows that we can get a contribution to the value of $g^{(2)}(0)$ from the accidental counts, so we should subtract them from the measured counts before making any calculations.

### 2.3.1 Experimental method and results

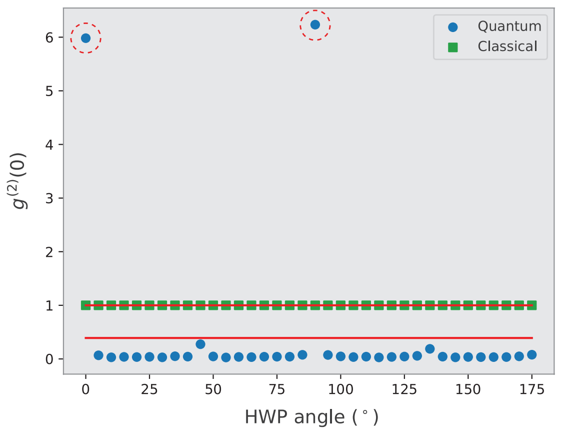
We first need to align the new detector and the PBS. The PBS and the new detector should be placed such that the PBS is equidistant from both the detectors. The procedure will be similar to the one we followed for the previous experiment. To rough align the new detector, we first use the back-propagation technique. After this, we turn the pump beam HWP to 45° to produce down-converted $|VV\rangle$ photons from the BBO stack. $|V\rangle$ photons will go straight through the PBS and give us maximum single-photon detections on detector B (maximum AB counts). As B is already aligned, this detection scheme gives us the reference counts. We then turn the pump beam HWP to 0° which will change the BBO output to $|HH\rangle$, and the photons hitting the BBO will now be reflected towards $B'$. Now we fine-tune the new detector B' such that we get almost the same number of counts as we were getting on detector B. This completes our additional alignment for this setup.

For this experiment, we will change the input state of the PBS and calculate values of $g^{(2)}(0)$ for different states. To change the polarization of the input to BBO, we use an HWP, which we can add to either the pump beam's or idler beam's path. We will record data for the counts against the orientation of HWP. Since we claim that our source is quantum, we need to show that it has anti-bunching, and for that reason, we need to get 0 value of $g^{(2)}(0)$ irrespective of the input state. For comparison, if we do not condition the detection on detector A, The light hitting the BBO will be a classical light, and for that, we should get a $g^{(2)}(0)$ value close to 1. Expression for classical $g^{(2)}(0)$ in terms of detector counts can be derived as,

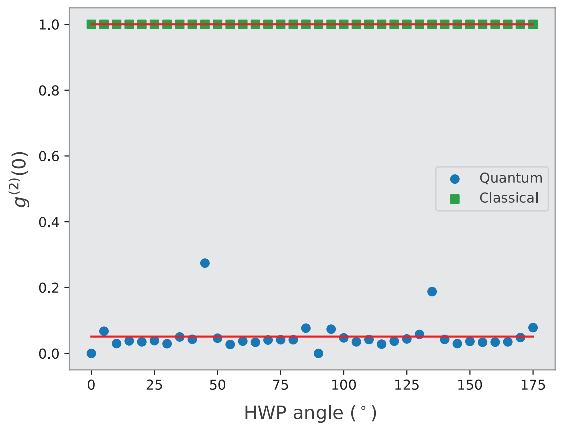
$$g^{(2)}(0) = \frac{\langle I_B(t) I_{B'}(t)\rangle}{\langle I_B(t)\rangle\langle I_{B'}(t)\rangle}, \tag{2.9}$$

$$= \frac{\langle N_B N_{B'}\rangle}{\langle N_B\rangle\langle N_{B'}\rangle}. \tag{2.10}$$

Figure 2.5(a) was obtained when we calculated the $g^{(2)}(0)$ for different pump beam HWP angles. The classical result was obtained by considering the total counts hitting the $B$ and $B'$ detectors and using the equation 2.6.



(a) $g^{(2)}(0)$ *with outliers.*



(b) $g^{(2)}(0)$ *without outliers.*

Figure 2.5: *Graphs for comparison of quantum and classical* $g^{(2)}(0)$. *(a) has a couple of outliers corresponding to the angles that are multiples of* $45^\circ$. *While (b) has those outliers removed.*

The points marked by the red circles are the outliers. These outliers occur at the points where we have all $|H\rangle$ or $|V\rangle$ polarized photons because at these polarizations either $N_{AB}$ or $N_{AB'}$ counts should almost be zero, resulting in a denominator in equation 2.8 that approaches zero. To take care of these outliers we assign them the value 0 that we should ideally obtain for the quantum case and obtain the graph shown in figure 2.5(b). For the quantum case the values of $g^{(2)}(0)$ gave us the following statistics in the lab experiment.

| | |
|---|---|
| Mean | 0.0512 |
| Standard error | 0.0082 |
| Confidence interval | 115 |

Table 2.1: *Table quoting the experimental statistics.*

To quantify the anti-bunching, we add a delay between the two signal channels coming from the SPCMs. By adding the delay, we will miss the photons that were produced simultaneously, and as a result, we will start getting photons that are bunched together. These bunched photons will start giving us values of $g^{(2)}(\tau)$ asymptotically reaching 1 for increasing values of $\tau$. To add a delay between the two channels, we use the delay box from Stanford research systems, inc. The following graph is obtained experimentally, and it conforms to the simulated and experimental results in [3] and [6].

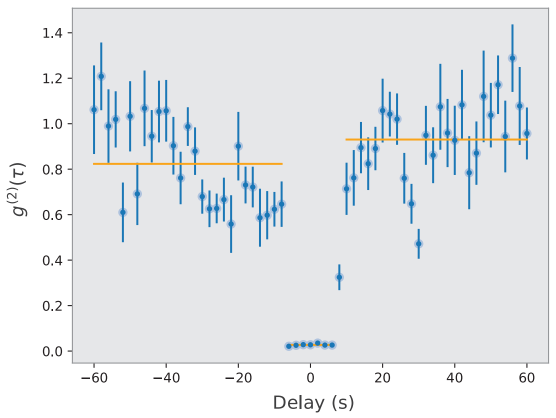


Figure 2.6: *Graph for $g^{(2)}(\tau)$ obtained for different delays introduced. The lines between the points represent the mean for that particular cluster. Error bars have different length for each point.*

## 2.4 Experiment 3: Estimating state of polarization of single photons

In the previous experiment, we saw that the photons are discrete particles of light and that the light has a grainy nature. Despite having a grainy, particle-like nature, the photons still act like waves as well. They have a polarization associated with them like a classical electromagnetic wave does. In this experiment, we will use the laws of probability to determine the polarization state of the single photons. The polarization state of single photons is encoded as a quantum state. This quantum state is defined in general as,

$$|\psi\rangle = A\,|H\rangle + Be^{i\phi}\,|V\rangle\,. \tag{2.11}$$

$A$ and $B$ are the probability amplitudes for the measurements in state $|H\rangle$ and $|V\rangle$ components, respectively and $e^{i\phi}$ is the phase difference between the two components. For the normalization of probabilities, equation 2.12 must be satisfied.

$$A^2 + B^2 = 1 \tag{2.12}$$

For a single photon beam, if we make several measurements in the $|H\rangle$ and $|V\rangle$ basis, we can determine the values of $A$ and $B$. Measurement in $|H\rangle$ is basically the projection of $|\psi\rangle$ on $|H\rangle$. Mathematically, it is shown by an inner product in the Dirac notation as,

$$\begin{aligned}\langle H|\psi\rangle &= A\,\langle H|H\rangle + Be^{i\phi}\,\langle H|V\rangle\,,\\ &= A.\end{aligned}$$

The probability of measuring the photon in state $|H\rangle$ will be,

$$P(|H\rangle\,|\,|\psi\rangle) = |\,\langle H|\psi\rangle\,|^2 = A^2. \tag{2.13}$$

Similarly, we can determine the value of $B$ by making a projection in $|V\rangle$. Or we can just use the normalization condition for that as shown in equation 2.14.

$$B = \sqrt{1 - A^2} \tag{2.14}$$

Determining the value of $A$ and $B$ just required measurements in $\{|H\rangle\,, |V\rangle\}$ basis. To determine $\phi$ we will require measurements in $\{|D\rangle\,, |A\rangle\}$ and $\{|L\rangle\,, |R\rangle\}$ basis. In terms of $|H\rangle$ and $|V\rangle$ these polarizations are defined as,

$$
\begin{aligned}
|D\rangle &= \frac{1}{2}(|H\rangle + |V\rangle),\\
|A\rangle &= \frac{1}{2}(|H\rangle - |V\rangle),\\
|L\rangle &= \frac{1}{2}(|H\rangle + i\,|V\rangle),\\
|R\rangle &= \frac{1}{2}(|H\rangle - i\,|V\rangle).
\end{aligned}
$$

The projection of $|\psi\rangle$ on these will be given by,

$$
\begin{aligned}
P(|D\rangle\,|\,|\psi\rangle) &= |\,\langle D|\psi\rangle\,|^2,\\
&= |\,\Big\langle D\Big|(A\,\langle H|H\rangle + Be^{i\phi}\,\langle H|V\rangle)\Big\rangle\,|^2,\\
&= \frac{1+2ABcos\phi}{2},
\end{aligned}
$$

and

$$
\begin{aligned}
P(|L\rangle\,|\,|\psi\rangle) &= |\,\langle L|\psi\rangle\,|^2,\\
&= |\,\Big\langle L\Big|(A\,\langle H|H\rangle + Be^{i\phi}\,\langle H|V\rangle)\Big\rangle\,|^2,\\
&= \frac{1+2ABsin\phi}{2}.
\end{aligned}
$$

Using the equations above, we can get a single equation that will yield a unique value of $\phi$,

$$
\phi = tan^{-1}\left(\frac{P(|L\rangle\,|\,|\psi\rangle) - 0.5}{P(|D\rangle\,|\,|\psi\rangle) - 0.5}\right). \tag{2.15}
$$

### 2.4.1 Generating polarization states and changing measurement basis

We have seen that by using different measurement basis, we can determine the state of polarization of the photons. This section deals with how we can experimentally generate a photon in a specific state and change the measurement basis. We will choose the idler beam for this experiment.

We can generate many polarization states by inserting an HWP or a QWP in the path of a beam in the $|H\rangle$ state. The table 2.2 shows the orientations of the optical components required for the generation of certain polarization states starting from the $|H\rangle$ state.

| State | HWP angle | QWP angle |
|---|---|---|
| $\|H\rangle$ | 0° | 0° |
| $\|V\rangle$ | 45° | 0° |
| $\|D\rangle$ | 22.5° | 0° |
| $\|A\rangle$ | -22.5° | 0° |
| $\|L\rangle$ | 0° | 45° |
| $\|R\rangle$ | 0° | -45° |

Table 2.2: *Waveplate angles required to produce certain polarization states.*

To change the measurement basis, we need to use a QWP followed by an HWP oriented at the angles specified in table 2.3.

| Measurement basis | QWP angle | HWP angle |
|---|---|---|
| $\|H\rangle, \|V\rangle$ | 0° | 0° |
| $\|D\rangle, \|A\rangle$ | 45° | 22.5° |
| $\|L\rangle, \|R\rangle$ | 45° | 0° |

Table 2.3: *Waveplate angles required to change measurement basis.*

Correspondence to detector counts of the polarization state is summarized in table 2.4.

| Measurement basis | AB' detection | AB detection |
|---|---|---|
| $\|H\rangle, \|V\rangle$ | $\|H\rangle$ | $\|V\rangle$ |
| $\|D\rangle, \|A\rangle$ | $\|D\rangle$ | $\|A\rangle$ |
| $\|L\rangle, \|R\rangle$ | $\|L\rangle$ | $\|R\rangle$ |

Table 2.4: *Measurement basis and corresponding detector counts.*

The probabilities of detection that we need to calculate $A, B$ and $\phi$ can simply be determined by these formulae,

$$P_{AB} = \frac{N_{AB}}{N_{AB} + N_{AB'}}, \tag{2.16}$$

and

$$P_{AB'} = \frac{N_{AB'}}{N_{AB} + N_{AB'}}. \tag{2.17}$$

### 2.4.2 Experimental method and results

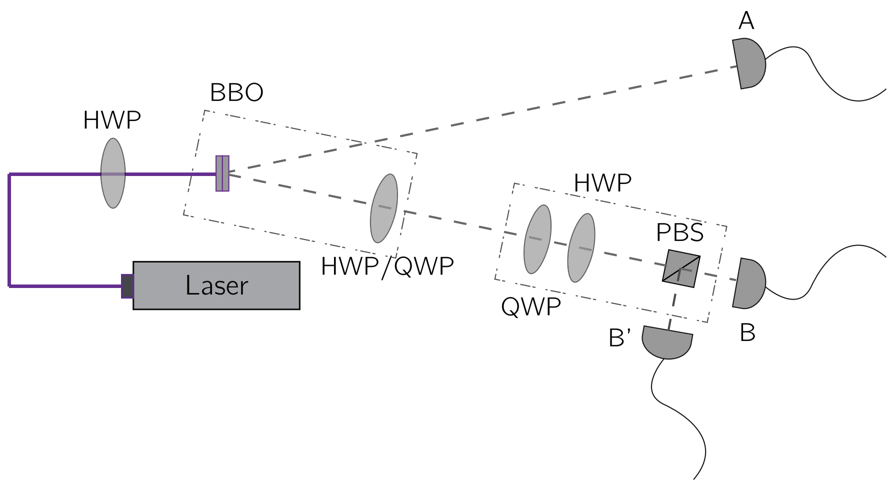


Figure 2.7: *Schematic for experiment 3.*

The first dotted block containing the BBO crystal and an HWP/QWP represents the state generation part of the experiment, while the other block represents the state measurement part. We generate the states given in the table 2.3 by using the corresponding waveplate angles and then measure the counts in different basis. Finally, we determine the state of polarization by calculating $A$, $B$, and $\phi$ as described in the previous sections.

Table 2.5 shows our experimental results.

| Input | Prediction | Measurement |
|---|---|---|
| $\|H\rangle$ | A = 1.000, B = 0.000 | A = 0.996, B = 0.089, $\phi = 0.387$ |
| $\|V\rangle$ | A = 0.000, B = 1.000 | A = 0.214, B = 0.977, $\phi = -0.138$ |
| $\|D\rangle$ | A = 0.707, B = 0.707, $\phi = 0.000$ | A = 0.686, B = 0.727, $\phi = 0.310$ |
| $\|A\rangle$ | A = 0.707, B = 0.707, $\phi = 3.142$ | A = 0.733, B = 0.681, $\phi = 2.817$ |
| $\|L\rangle$ | A = 0.707, B = 0.707, $\phi = 1.571$ | A = 0.538, B = 0.843, $\phi = 1.127$ |
| $\|R\rangle$ | A = 0.707, B = 0.707, $\phi = -1.571$ | A = 0.792, B = 0.611, $\phi = -1.320$ |

Table 2.5: *Comparison of expected and experimental results for experiment 3.*

The differences in experimental outcome and theoretical predictions can be accounted for by considering the experimental error while orienting the waveplates and the imperfections in the waveplates themselves. A Python code can be found in Appendix

B that can be used to troubleshoot the waveplates. It uses Jones calculus to theoretically predict the polarization state after it passes through different arrangements of wavevplates. This code can predict graphs that should be produced when a certain waveplate is rotated. A mismatch in the experimental data and the theoretical prediction indicates a possibility of having a faulty waveplate in the setup. Other than troubleshooting, this program can also use used to verify the results of experiment 3.

## 2.5 Experiment 4: Single-photon interference and quantum erasure

In this experiment, we will work with the which-path information. Which-path information refers to the information about the path that a single photon will take. If we know for sure where the photon will go, we have complete which-path information. For example, if we send in a known polarization state of either $|H\rangle$ or $|V\rangle$ through a PBS, we know for sure if it will reflect or go straight through. In this case, we have complete which-path information of the photon. The toll for getting complete which-path information is that we lose the wave-like property of photon, and the nature of photon changes to completely particle-like.

In this experiment, we will change the nature of photons between wave-like and particle-like by varying the amount of information we know of its path. While changing this, we pass the single photons through a Mach Zehnder interferometer to determine the nature of the photon. If we have complete which-path information, we have converted the photon into a particle, and particles do not interfere with one another. In this case, we will not see any interference fringes. However, suppose we do not have any information about the photon's path. In that case, it will behave like a wave, interfere with itself, and we will see an interference pattern appearing at the output detector. Another interesting aspect of this experiment is that we place the polarizer (the equipment to measure which-path information) after the beam displacing prisms where the interference is supposed to occur. This shows that the act of determining the state of polarization in the present changed the nature of the photon in the past when it was passing through the interferometer. This experiment is analogous to Wheeler's "delayed-choice experiment" [5].

### 2.5.1 Experimental method and results

We make an interferometer in the path of the single photons by using two beam displacing prisms (BDPs), two HWPs, and a polarizer, as shown in the figure 2.8.

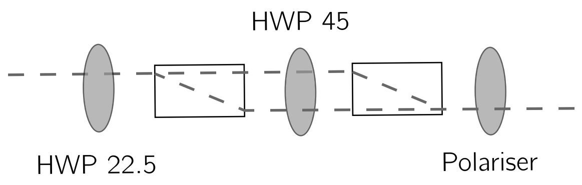


Figure 2.8: *Mach Zehnder interferometer made using BDPs.*

We set the pump beam HWP at 0°, so the BBO's output is $|HH\rangle$ and input to the interferometer in the idler beam will be $|H\rangle$. The first HWP is set at 22.5° which will convert the state to an equal superposition state,

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|H\rangle + |V\rangle). \tag{2.18}$$

The BDPs are designed such that they let the $|H\rangle$ state photons pass through undeviated and shift the $|V\rangle$ state photons to the right by 4 mm. When the beam in the superposition state passes through the BDPs, it will split into two portions, one portion will contain $|H\rangle$ photons, and the other will contain $|V\rangle$ photons. The second HWP is oriented at an angle of 45°, so the two beams will flip their polarization when they pass through this HWP. Now when the beams pass through the next BDP, the $|H\rangle$ portion will go undeviated, and the $|V\rangle$ portion will get deviated by 4 mm again. The figure above shows that the two beams have covered the same path length after they split and before they combine. There should be no phase difference in the two beams that interfere. However, if we make the beams cover different path lengths, which we can do by changing the tilt of one of the BDPs in tiny steps using a piezoelectric stepper motor, we can induce a path difference in the interfering beams. When this path difference is such that the two beams are in phase, we see constructive interference, and we will get a peak in single photon counts, and when the two beams are out of phase, we will get destructive interference which will correspond to a dip in single-photon counts.

We will be able to observe all these phenomena provided that the photons act as waves and they do interfere. This will be determined by the last component in the interferometer, i.e. the polarizer determining the which-path information. If we set the polarizer at 0° or 90°, we will be able to distinguish between the $|H\rangle$ and $|V\rangle$ photons, and while doing so, we will be able to determine which beam path out of the two did our photon probe through. We have determined the which-path information, which means now we will be deprived of the interference fringes. On the other hand, if we turn the polarizer to 45°, we will not be able to determine which path did the photon take. No which-path information will result in maximum visibility of the interference fringes. Equation 2.19 gives us a simple formula that defines the visibility of the fringes,

$$V = \frac{N_{max} - N_{min}}{N_{max} + N_{min}}. \tag{2.19}$$

We will insert this interferometer in the path of idler beam as shown by the schematic below.

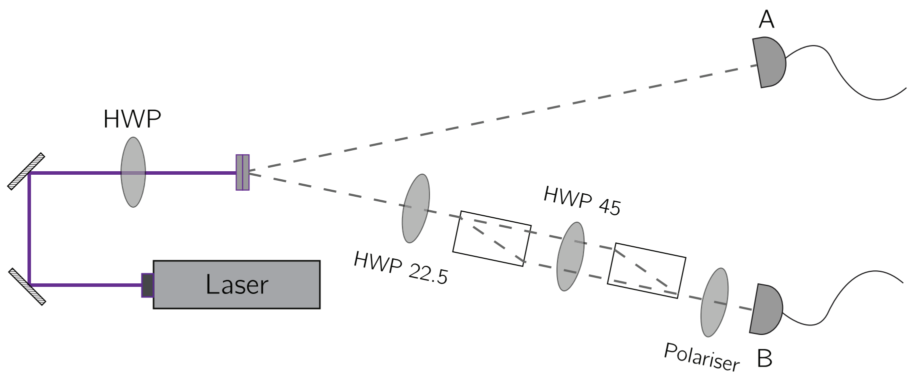


Figure 2.9: *Schematic for experiment 4.*

Since the beam emerging from the second BDP has displaced by 4 mm from the original beam path, we need to move the detector $B$ by 4 mm so that it can catch the photons that have displaced on their way through the interferometer. Initially, to get the interference pattern, we need to tilt the BPDs by a significant amount. For our experiment, we took steps from -100,000 to +100,000 at 100 steps/sec to initially get the tilt where we get interference fringes. To scan these fringes with better accuracy, we tilt the BDP from -1500 to +1500 steps at a speed of 20 steps/s.

Figure 2.10 shows the interference fringes we obtained for different polarizer angles.

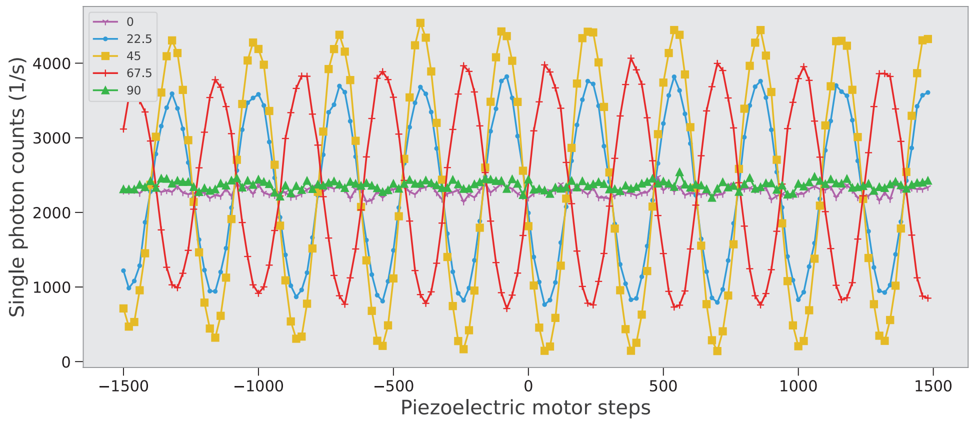


Figure 2.10: *Graph for interference fringes corresponding to different polarizer angles.*

Table 2.6 shows the visibility calculated for different polarizer angles.

| Polarizer angle (°) | Predicted V | Experimental V |
|---|---|---|
| 0 | 0 | 0.06 |
| 22.5 | 0.71 | 0.51 |
| 45 | 1 | 0.73 |
| 67.5 | 0.71 | 0.54 |
| 90 | 0 | 0.07 |

Table 2.6: *Predicted and measured visibility for different states.*

# Chapter 3

# Photon pair experiments: Entanglement

## 3.1 Introduction

Until now, we have dealt with the photons in the idler beam only and considered only a single stream of photons. From here onwards, we will include the signal beam in investigations and use the joint polarization state to represent the whole system.

When photons are generated in the BBO crystal, they get correlated because of the law of conservation of energy,

$$w_p = w_s + w_i, \tag{3.1}$$

and the law of conservation of momentum,

$$\vec{k_p} = \vec{k_s} + \vec{k_i}. \tag{3.2}$$

In these expressions, the subscripts $p$, $s$ and $i$ represent pump beam, signal beam and idler beams respectively. These correlations are quantum, and we say that the two photons are entangled because of these correlations. Entangled quantum particles violate local realism, i.e. entangled photons seem to give a random output when measured individually, but we see very strong correlations in the measured state of the photons when we compare the experimental outcomes. It is as if measuring the state of one photon collapses the entanglement and determines the fate of the other photon in the entangled pair.

A two-photon state is in general given by,

$$\begin{aligned}|\psi\rangle &= A\,|H\rangle_A\,|H\rangle_B + Be^{i\phi}\,|V\rangle_A\,|V\rangle_B\,, \\ &= A\,|HH\rangle + Be^{i\phi}\,|VV\rangle\,.\end{aligned}$$

An *entangled state* is defined as the state from which we can not extract the state of a single particle. Following are the states with maximum entanglement known as the Bell states,

$$\left|\phi^+\right\rangle = \frac{1}{\sqrt{2}}(|HH\rangle + |VV\rangle), \tag{3.3}$$

$$\left|\phi^-\right\rangle = \frac{1}{\sqrt{2}}(|HH\rangle - |VV\rangle), \tag{3.4}$$

$$\left|\psi^+\right\rangle = \frac{1}{\sqrt{2}}(|HV\rangle + |VH\rangle), \tag{3.5}$$

$$\left|\psi^-\right\rangle = \frac{1}{\sqrt{2}}(|HV\rangle - |VH\rangle). \tag{3.6}$$

(3.7)

## 3.2 Experiment 5: Freedman's test of local realism

This experiment is the most basic and experimentally easy to perform test for local realism. In this experiment, we begin with a two detector setup like the one we used for experiment 1 and insert linear polarizers in the path of both the beams. Let $\theta_A$ and $\theta_B$ be the angles at which the polarizers are oriented. The probability of detection of a photon pair provided the polarizer angles is defined by,

$$P(\theta_A, \theta_B) = \frac{N(\theta_A, \theta_B)}{N_t}, \tag{3.8}$$

where $N_t$ represents the total number of photon pairs reaching the detectors in time $t$, while $N(\theta_A, \theta_B)$ represents the number of coincidence detections in the same time window.

An inequality can be established using locality, reality and hidden variable assumptions. This inequality is known as the Freedman's inequality, and is written as,

$$\delta = \left| \frac{N(22.5^\circ) - N(67.5^\circ)}{N_o} \right| - \frac{1}{4} \leq 0. \tag{3.9}$$

here $N(\phi)$ represents the relative angle between the two polarizers. The detailed derivation of the Freedman's inequality can be found in [6] for interested readers. To determine the same quantity $\delta$ in another way, we can use quantum mechanical predictions and the polarizer properties. As mentioned in [6], the equation for it comes out to be,

$$\delta = \frac{\epsilon_A \epsilon_B}{2\sqrt{2}} - \frac{1}{4}. \tag{3.10}$$

Here $\epsilon_A$ and $\epsilon_B$ represent the transmittance of polarizers $A$ and $B$, respectively. The classical and quantum predictions are in contradiction with one another. Classical hypotheses state that the value of $\delta$ should be less than 0, while quantum mechanically, $\delta$ should attain a value greater than 0. There is one experimental apparatus restriction that must be taken into account. From the equation above, it can be easily seen that to attain a value of at least 0, the average transmittance of the polarizers should be 0.84 at the bare minimum.

### 3.2.1 Experimental method and results

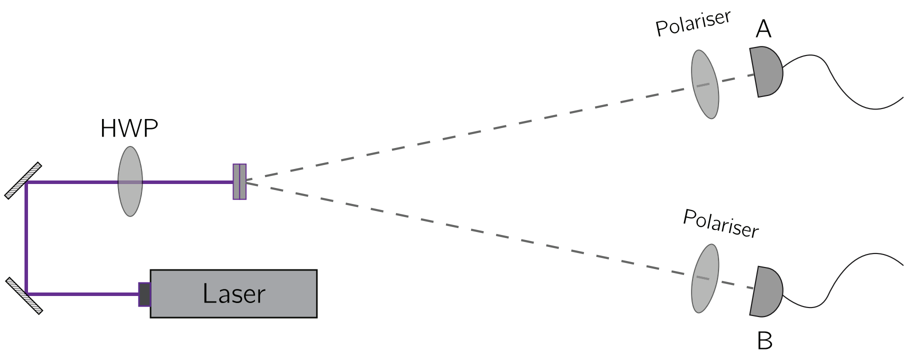


Figure 3.1: *Schematic for experiment 5.*

The experimental setup is similar to that of experiment 1 with the addition of polarizers in the two down-converted beam paths as shown in figure 3.1. We generate the required polarization state by turning the pump beam HWP at 22.5°. This will turn the pump bean polarization to $|D\rangle$. We assume the BBO crystals to be very thin and stacked closely so that there is no significant phase $\phi$ that we need to cater for. The output of the BBO stack in this case will be the Bell state,

$$\left|\phi^+\right\rangle = \frac{1}{\sqrt{2}}(|HH\rangle + |VV\rangle).$$

The transmittances of the polarizers used in our experiment are measured to be $\epsilon_A = 0.857$ and $\epsilon_B = 0.867$. The average transmittance is $\epsilon_{avg} = 0.862$ so our polarizers should suffice for the minimum requirement of transmittance to violate local realism. The predicted value of $\delta$ that we should get using these transmittances can be derived as,

$$\begin{aligned}\delta &= \frac{\epsilon_A \epsilon_B}{2\sqrt{2}} - \frac{1}{4}, \\ &= \frac{0.857 \times 0.867}{2\sqrt{2}} - \frac{1}{4}, \\ &= 0.0127.\end{aligned}$$

The experimental values obtained for use in equation 3.9 are,

$$\begin{aligned}N(22.5°) &= 2732 \pm 5, \\ N(67.5°) &= 542 \pm 3, \\ N_o &= 8232 \pm 7.\end{aligned}$$

These values lead us to the value of $\delta = 0.01600 \pm 0.00012$, which clearly violates Freedman's inequality and hence proves entanglement in the pair of photons.

## 3.3 Experiment 6: CHSH Test of local realism

Another more advanced test for local realism is the CHSH test. Its experimental apparatus contains four detectors. Like Freedman's test, this experiment also revolves around disapproving an inequality based on the local realism and hidden variable theories. In this case, as mentioned in [6], our inequality is the CHSH inequality written as,

$$|S| = E(a,b) - E(a,b') + E(a',b) + E(a',b'). \tag{3.11}$$

Where $E(\alpha,\beta)$ represents the expected value of a local realistic measurement for the analysis angle of $\alpha$ in the signal beam and $\beta$ in the idler beam. Its value can be experimentally determined by the following equation, as mentioned in [6],

$$E(\alpha,\beta) = P_{HH} + P_{VV} - P_{HV} - P_{VH} = cos(2(\alpha-\beta)). \tag{3.12}$$

Using this, $|S|$ becomes,

$$\begin{aligned} |S| &= E(a,b) - E(a,b') + E(a',b) + E(a',b'), \\ &= cos(2(a-b)) - cos(2(a-b')) + cos(2(a'-b)) + cos(2(a'-b')). \end{aligned}$$

If $|S|$ attains a value greater than 2, this will violate local realism. So the inequality under consideration is,

$$|S| \leq 2.$$

### 3.3.1 Experimental method and results

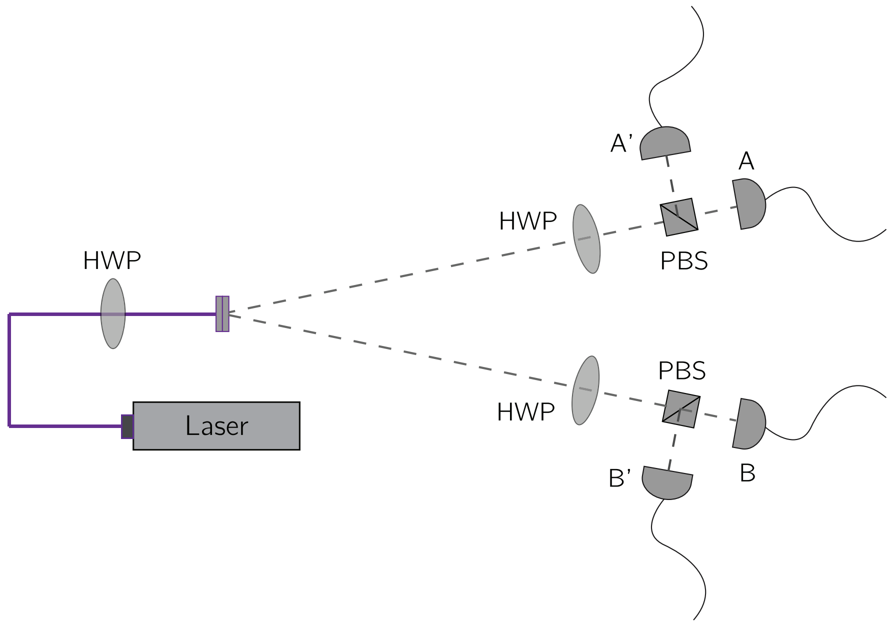


Figure 3.2: *Schematic for experiment 6.*

Figure 3.2 shows a schematic for this experiment and the components can be aligned by following the steps outlined for the previous experiments.

For a analysis angles $a = -45°, b = -22.5°, a' = 0°, b' = 22.5°$ and the Bell state $|\phi^+\rangle$ we get maximum violation of CHSH inequality and get a value of $|S| = 2\sqrt{2}$ . The experimental value will be less than this because of imperfections while preparing and measuring the state. The experimental value I obtained while performing the experiment came out to be

$$S = 2.327 \pm 0.082$$

The test passes with a 4 standard error margin, which is a clear violation of local realism. The photon pairs were entangled.

# Chapter 4

# FPGA: Introduction and programming

## 4.1 Introduction

This chapter is a step by step guide to FPGA programming, especially for coincidence counting in single-photon experiments. It is written in a highly user-friendly way, such that anyone with little to no programming or FPGA knowledge can program an FPGA for use in the experiments in a single-photon quantum physics lab.

### 4.1.1 What is an FPGA? A historical overview.

When digital electronics were still in infancy, discrete logic chips (AND, OR, NOT, NAND etc.) were used to make entire logic circuits. It is still possible, in theory at least, to make any digital circuit using just these chips. However, we prefer not to resort to that technique any longer because of its slow speed, high cost and difficulty of implementation.

The concept behind the logic design is still the same, but now we have much more efficient methods to implement the circuits. One circuit implementation method that takes the lead over any other method when it comes to speed is application-specific integrated circuits (ASICs). These circuits are extremely fast, but there is a price to pay for this speed. These circuits can be programmed only once. One small mistake or one need for change in the implementation of the circuit will render the entire batch of old circuits useless.

Another modern way of implementing logic on a chip is by the use of microprocessors. Microprocessors contain fixed hardware components like registers, logic units, and control units. When we use a microprocessor, we use this fixed hardware repeatedly to accomplish the required task. This gives us much room for innovation but to no one's surprise, using hardware in this way is highly inefficient.

At this point, the FPGAs come into play. FPGAs are made with rows of gates stacked over one another, but the gates are left unconnected. Based on the design needs, the gates are connected through wiring channels; changing the wire connections changes the circuit implementation. The design of an FPGA can be elaborated into three significant components.

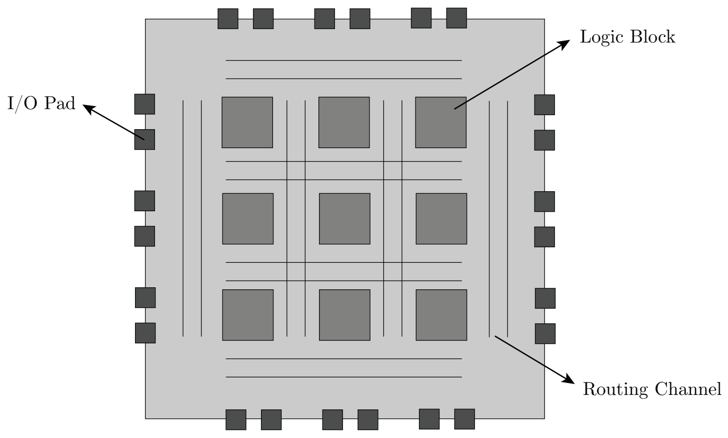


Figure 4.1: *Illustration for FPGA's overall design.*

- **Logic Blocks:** Each logic block in the FPGA typically consists of a four-input[1] lookup table (4-LUT) and an optional D Flip-Flop. A 4-LUT is made up of four layers of 2-to-1 multiplexer circuits. A 4-LUT has four control inputs and 16 standard inputs connected to registers storing values of either 0 or 1. A 4-LUT can implement any logic gate if we alter the values stored in the registers. The D flip-flop halts the output of the logic block until it registers a

[1]Some advanced FPGAs use LUTs with six control inputs as well

rising clock edge (we will get into the details of clock cycles further down the road)

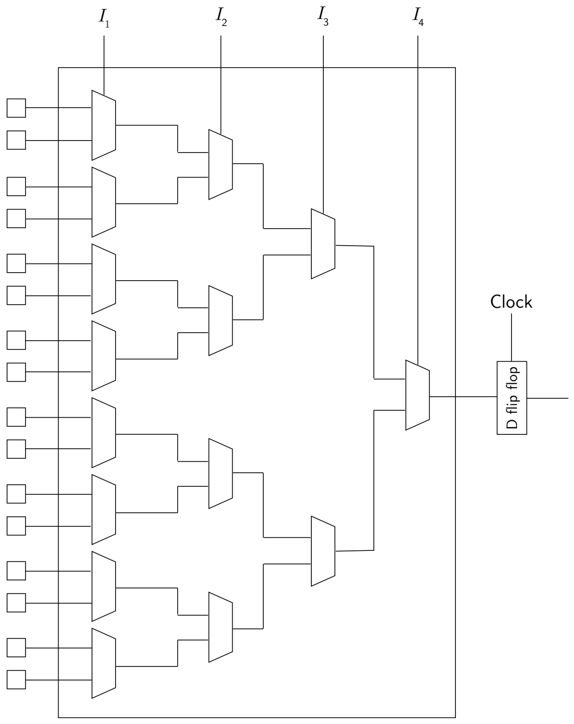


Figure 4.2: *A schematic for a 4LUT made using 16 MUXs.*

- **Routing Channels:** A modern FPGA may contain tens of thousands of logic blocks, and all of these logic blocks are interconnected by a web of routing wires and programmable interconnection switches. This allows us to connect all the logic blocks to make a circuit that we desire just by writing a code that configures the routing channels.
- **I/O Pads:** The circuit we made would be useless if we could not communicate with it. For this purpose, we have I/O pads that enable connection to a range of different instruments and peripherals.

It is to be noted that there is a difference in programming for a microprocessor and an FPGA. A microprocessor has fixed hardware, and the commands need to be fed continuously for it to process them. In contrast, an FPGA needs to be *configured* only once after power-up, the circuit gets implemented, and the required task can be performed as many times as required. For this reason, programming an FPGA is often called **configuration**. To configure an FPGA, we download a bit-stream file onto an FPGA that consists of 0's and 1's. It contains instructions from a hardware- configuration code written in HDL (Hardware Description Language). This bit-stream file implements circuitry on the FPGA chip. This implementation stays written on the FPGA as long as it is powered on. No need to send further commands to FPGA.

## 4.2 Task for FPGA in our experimental setup.

Before exploring the world of FPGAs, let us look at what implementation we want and what task we want to accomplish in our experiments.

### 4.2.1 Overview

A comprehensive view of our need would be to register the pulses generated by the SPCMs and detect if any of these pulses arrived simultaneously. The SPCMs output a square wave of amplitude 2.2V and 20 ns whenever a photon strikes its sensor. We want some way to process and count these narrow pulses and communicate this information to a computer. FPGAs are equipped with high-speed clocks, making it possible for us to register these pulses. The built-in memory of the FPGA temporarily stores the information of counts until it is sent to a computer in the form of bits (0s and 1s) through a serial communication port.

### 4.2.2 Implementation

#### Registering a pulse

Every FPGA is equipped with an external oscillator that generates a square wave (clock signal). This signal enters the FPGA through a specific physical pin, and we can use this pin's input to use the clock signal wherever we need it to trigger some event. Both the clock signal and the output of SPCMs are square waves, i.e. series of rising and falling edges of a voltage signal. The clock signal is a square wave of constant frequency, which we can utilize to monitor an input pulse (another square wave). For this to work, the clock signal frequency must be at least as high as the input we want to register. In our case, the pulse generated by an SPCM is 20 ns, while the FPGA clock of the Nexys A7 is 100MHz, i.e. a square pulse of 10 ns.

Hence, registering the pulses coming from the SPCMs should not pose a problem for our FPGA.

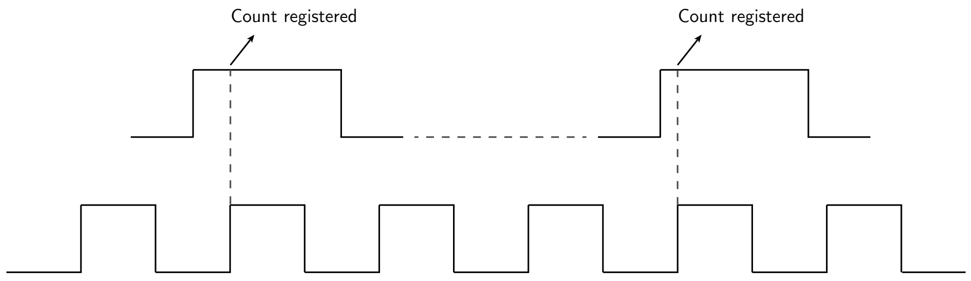


Figure 4.3: *Pulse detection at rising edges.*

### Registering a coincidence

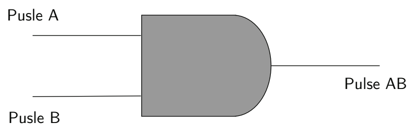


Figure 4.4: *Registering coincidence using AND operation.*

When our clock signal coincides with a pulse coming from the SPCM, the FPGA produces a pulse signifying a detection event. While simply detecting the incoming pulses, it is also possible to look for two pulses coming in simultaneously from two channels by simply applying an AND gate on these detection pulses.

### Storing the counts

Once we have registered the required counts, we can save them temporarily into registers (a combination of gates that can store bits) until it is time to output them to the computer.

### Sending output to computer

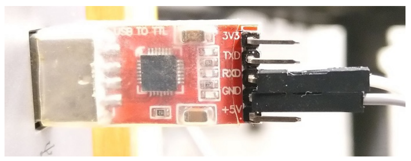


Figure 4.5: *Silicon lab's UART communication bus.*

The data is now stored in registers, ready to be sent to a computer. An FPGA can output data in the form of bits only, so we send the data stored in registers bit by bit to the computer over a UART serial communication channel. We need to communicate to the computer the rate at which we will be transferring the data. This rate is known as the **baud rate**.

## 4.3 Selecting the right FPGA

FPGAs may seem to be the solution to every problem one may face; however, there are some implications one must consider before buying an FPGA. One FPGA can not suit everyone's needs, and in the worst case, one may buy something expensive yet entirely useless for the desired application.

### 4.3.1 I/O options

FPGAs are specific to the industry they are destined to serve, especially in the I/O department. An FPGA designed for use in multimedia devices or broadcasting purposes may have many multimedia connectors like video interface ports and audio jacks. On the other hand, the FPGA designed for use in some outer-space equipment may not contain any multimedia interface because no one needs audio jacks in outer space. Needless to say, these I/O connectors will require some area on the FPGA board and will add to the cost of the device as well.

For the coincidence counting experiments, we will only require PMOD connectors. Pins in PMOD connectors can be used as digital I/O communication pins, and we will be using them for just that. Input will be a square pulse generated by the SPCM, and the output will be bits sent to the computer via the UART communication serial device.

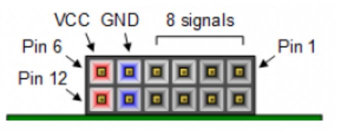


Figure 4.6: *Front view of a PMOD connector.*

### 4.3.2 Frequency

The frequency of an FPGA is a raw measure of its speed—the higher the frequency, the faster the FPGA. The frequency of the FPGA is to be selected according to the application.

In our case, the signal being produced by our APDs is a square wave of about 20 ns, so our FPGA's clock speed should be such that it can register every count precisely without a miss. For this, we require the frequency of our clock to be at least $(1/20) \times 10^{-9} = 50 \times 10^{6}$ Hz. Our previous device, the Nexys2, had a clock speed of 50MHz, but the upgraded Nexys 7A operates with a crystal producing a clock of 100MHz. It was possible to miss a few counts with our previous FPGA, but with the newer one, we would not be missing a beat.

### 4.3.3 Cost

Depending on the budget, one may not want to go over the minimum requirements of the experimental needs, and that makes sense. However, with a reasonable budget, a device that does a bit more than the minimum might be a better choice as it can be used if the experimental setup needs to be expanded or improved. Factors that impact the cost of a device are primarily its clock speed, available logic units, I/Os, and RAM.

### 4.3.4 Manufacturer

Who is making the FPGA is also a thing to consider while shopping for one. Xilinx makes the FPGA we are using. Xilinx is the primary producer of FPGAs worldwide, and so its supporting software set is one of the best ones available. Another major producer of FPGAs is Altera.

Selecting an FPGA is a tricky task and may require some time investment to find the suitable device that fits our needs. It will, however, be worth it as it is vital to have the right equipment before beginning the experiment.

## 4.4 Programming the FPGA

Now that we have the proper hardware, we can shift our focus to programming it for our experiments.

### 4.4.1 Programming language

FPGAs are built different from microprocessors, and the simple programming languages like C, Java, or Python that we use to program microprocessors will be of no use here. We have to implement circuit designs using logic gates and wires on FPGA. So to program an FPGA, we need a Hardware Description Language (HDL). In HDL, we define physical connections between different logic gates.

We have two options of HDL to choose from.

1. **Verilog:** Verilog was created by Prabhu Goel, Phil Moorby, Chi-Lai Huang, and Douglas Warmke with "Gateway Design Automation" as the rights-holding company. Its syntax is more like C and is easier to understand and work with

for people with programming experience in C. It is the language that we chose to configure our FPGA.

2. **VHDL:** It was initially developed by the US Department of defence in 1987 and is more to depth and strict in syntax than Verilog.

### 4.4.2 Implementation

In Verilog, we divide the tasks we need to perform into portions of code called modules. When we run the FPGA after configuration, all these modules will run simultaneously. There will not be a program flow that will dictate the sequence of execution of the modules, but the modules will work in sync to perform the required task. Modules that require the clock signal's input will get it through a global routing channel that consists of interconnected wires. Feeding the clock signal through the global routing channel ensures that each module gets the clock simultaneously, and there is no delay in the working of the modules that need to work in sync.

The implementation of a coincidence counting unit on an FPGA will require five modules. Verilog implementations and functional details for all these modules are given below.

- **baud_rate_counter:** Baud rate is the rate at which our FPGA and computer agree to communicate, and it binds all data transfer. This module will use the FPGA clock to produce a pulse signal of the required baud_rate (frequency). We have chosen a commonly used baud rate of 19200 bits/s for our setup. We need to send a pulse after $100 \times 10^6/19200 = 5208$ clock cycles to achieve this. We define a register named baud_rate_count and increment its value by 1 on every rising edge of the clock cycle. When the value stored in this register reaches 5208, we send out a pulse signal of the same time as a clock signal, i.e. 10 ns, and reset the value in the baud_rate_count register to zero. We need to input the clock signal into this module, and it gives the baud rate clock of 19200 cycles per second in the register named baud_rate_clk.

```
module baud_rate_counter (input clock_50, output reg baud_rate_clk);

reg [31:0] baud_rate_count;

    always@(posedge clock_50)
        begin
        baud_rate_count <= baud_rate_count +1;

        if (baud_rate_count >= 2604)
        begin
            baud_rate_clk <= 1;
            baud_rate_count <=0;
        end

        else
            baud_rate_clk <= 0;
       end

endmodule
```

- **data_triggering:** Now that we have determined the communication rate of our FPGA and computer, we need to decide the time after which we want to send the data from an FPGA to a computer. The following module is designed to create a data triggering signal of 10Hz, i.e. we will be sending out data to the FPGA every 1/10th of a second. We use the baud rate clock we produced in the baud_rate_count module and produce a signal of 10 Hz by sending out a pulse after every 1920 positive edges of the baud rate signal. To do this, we take the data from baud_rate_clk register as input, increment the value of the register named data_trigger_count by one at every rising edge of the baud_rate_clk, send out a signal of the same width as the clock signal once the value of data_trigger_count reaches 1920, and then reset the value of data_trigger_count back to zero. We send the data into a register named data_triggering, which other modules can access in the program.

```
module data_triggering (input baud_rate_clk, output reg data_trigger);

reg [31:0] data_trigger_count;

    always @(posedge baud_rate_clk)

        begin
        data_trigger_count <= data_trigger_count  +1;
        // Currently set to 1,920 to produce a clock of 10Hz
        if (data_trigger_count == 12'b11110000000)
            begin
            data_trigger <= 1;
            data_trigger_count <=0;
            end
        else
            data_trigger <= 0;
       end

endmodule
```

- **counter:** This module implements the main task in our program. It takes in a clock signal, data_trigger and a pulse from one of the detectors as input and sends the data collected for 1/10 sec out into a register whose name can later be defined by the user. For this module, we take the clock signal, data_trigger, and a pulse from one of the SPCMs as input, and we define a wire named x which will contain a signal edge if either data_triggering or pulse contains a signal. Whenever we get a signal on this wire, if the signal is from an SPCM pulse, we increment the value in the register defined by the user to contain the count. If it was from the data_trigger register, we reset the value in the register to zero. This marks that the data collected by the FPGA so far has been forwarded to the computer, and the FPGA can start collecting data for the next cycle.

```verilog
module counter(input clock_50, input data_trigger, input pulse,
output reg [31:0]q);

    wire x;

    or o1 (x, data_trigger, pulse);

    always @ (posedge x)
        begin

        if (data_trigger)
            q <=0;

        else
            q<=q+1;

        end

endmodule
```

- **coincidence_pulse and three_detector_coincidence:** We construct these modules to send out a signal if two pulses reach the FPGA at the same time. We perform a simple AND operation on the coincidences of interest and store the output in the specified registers in this function. This module gets triggered whenever one of its inputs gets a signal.

```verilog
module coincidence_pulse (input a, input b, output reg y);

    always @(*)
        begin
        y = a && b;
        end

endmodule
```

```verilog
module three_detector_coincidence (input a, input b, input c, output reg y);

    always @(*)
        begin
        y = a && b && c;
        end

endmodule
```

- **data_out:** This module is designed to send the data generated and collected by all other modules to the computer via a UART serial communication channel. This module is triggered with the baud_rate_clk register that we defined earlier. It selects the registers containing the number of single detector counts and coincidence counts one by one and sends the binary representation of the stored value to the computer. Computers receive the data in terms of bytes. Each byte contains 8 bits. The registers storing the data are 32 bits long, and we divide the data into five packets of length 7 bits each. Three zeros append the last packet to compensate for the remaining space in a byte. Each packet is followed by a sequence of starting and stopping bits that mark the end and start of a new packet. These starting and stopping bits are to be used by the serial communication channel for its reference, and they will not appear in the final data that the computer gets. All that the computer will get in the end are five 7 bit packets from which we need to extract the required information.

- **coincidence:** This module is like the "main" function that we define in most high-level languages. Its function is to provide a workflow for all the modules we defined earlier. It interconnects the registers containing all the values to the required modules and provides a flow to the program.

Complete Verilog code and a schematic of the whole circuitry that will be implemented on the FPGA after burning this code on it is shown in appendix B.

## 4.5 Software

We will be using Xilinx Vivado for programming our new Nexys A7. The old Nexys 2 was programmed using Xilinx ISE, which Xilinx has now discontinued, and the Nexys A7 does not support ISE. Now that the computer receives the data, we need to interpret these bits into numbers and display the data in real-time onto the screen for easy, continuous data monitoring and collection.

### 4.5.1 Reading data on computer

For data logging, I had two options to choose from:

- **LabVIEW:** A licenced software from National Instruments. It is a systems engineering software for applications that require test, measurement, and control with rapid access to hardware and data insights. It is designed for an extensive range of applications and has many installation requirements.
- **Python:** Using the open-source programming language, we can communicate with the serial port and obtain data. Using matplotlib, we can generate graphs as well.

I chose Python because I had some prior experience with it which allowed me to create a decent GUI for the required task. Furthermore, it is open-source so that anyone can replicate and improve the code for similar experiments. It can reach a wider audience this way. It is not as resource hungry as LabVIEW and can be installed on almost any device. Figure 4.7 shows a screenshot of the GUI.

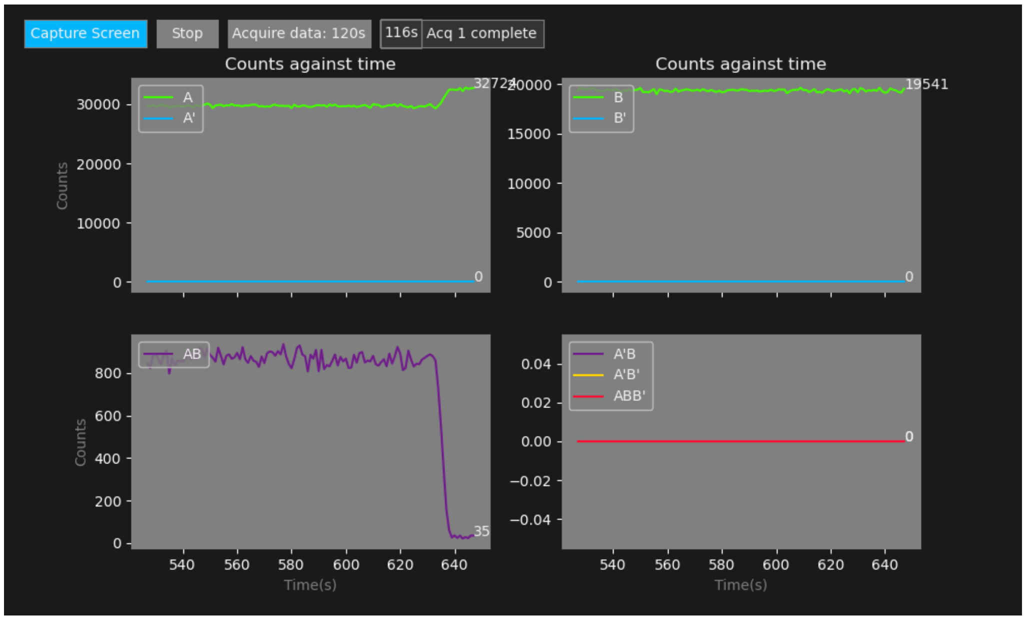


Figure 4.7: *GUI of the counting program written in Python*

Commented Python code for this software can be found in the appendix. Users can make relevant changes to change the time for data acquisition, the width of data being displayed etc., by going through the comments in the code.

# Appendix A

# Commented Python code for the coincidence counting software

```
from serial import *
from time import sleep
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.widgets as wgt
import os

def FindHeader(s):

    ''' We have inserted a random 35 bit long in the FPGA code. At the
    moment we are receiving 8 counts; after every session of these counts
    we ask the FPGA to this random long. We use this packet to mark the
    send and receive of an 8 count session. We find the packet first to
    make sure we are receiving the correct byte sequence.'''

    counter = 0
    byte = decto7bit(s.read(1)[0])
    # print(byte)

    ''' Data is being received in 7-bit packets, these five 7-bit packets
    make up the header that we are hoping to receive from the FPGA'''

    start = ["0001001", "1001100", "0100101", "0101010", "1011101"]

    trial = []
    '''Packets are received in reverse order aso we make a reverse array
```

```
of these packets in this variable.'''
for i in range(1, len(start)+1):
    trial.append(start[-i])

while True: # Looking for the packet in this loop.

    counter += 1
    ans = ""

    if counter > 100:
        print("Couldn't find the packet.")
        break

    if byte != trial[0]:
        print("Returning from stage 1")
        byte = decto7bit(s.read(1)[0])
        print(byte)
        continue

    ans = byte + ans
    byte = decto7bit(s.read(1)[0])

    if byte != trial[1]:
        print("Returning from stage 2")
        byte = decto7bit(s.read(1)[0])
        print(byte)
        continue

    ans = byte + ans
    byte = decto7bit(s.read(1)[0])

    if byte != trial[2]:
        print("Returning from stage 3")
        byte = decto7bit(s.read(1)[0])
        print(byte)
        continue

    ans = byte + ans
    byte = decto7bit(s.read(1)[0]))

    if byte != trial[3]:
        print("Returning from stage 4")
        byte = decto7bit(s.read(1)[0])
        print(byte)
        continue
        ans = byte + ans
        byte = decto7bit(s.read(1)[0])

        if byte != trial[4]:
            print("Returning from stage 5")
            byte = decto7bit(s.read(1)[0])
            print(byte)
            continue

        ans = byte + ans

        break

    return ans


'''Simple function to convert the decimal representation of a number
we receive from the FPGA into binary that we need to combine and read.'''
def decto7bit(byte):

    bit = ""

    while byte >= 1:
        if byte%2 == 0:
            bit = "0" + bit
        elif byte%2 == 1:
            bit = "1" + bit
        byte = int(byte/2)

    if len(bit) < 7:
        for i in range(7-len(bit)):
            bit = "0" + bit

    return bit


'''Converting a number from binary representation to decimal for us to
read as output. '''
def bin2dec(data):
    len_data = len(data)
    ans = 0
    for i in range(len_data):
        ans = ans + int(data[i])*(2**(len_data-1-i))
    return ans
```

```
'''Every 1/10th of a second we receive a set of 8 counts. This function
receives, processes, and returns one of those sets.'''
def GetIndvCounts(s):

    '''Receinig those 8 counts plus the header requires 46 bytes.'''
    data = s.read(51)

    temp = []
    for i in data:
        temp.append(decto7bit(i)) # dec27bit sends back a string.
    data = temp

    counts = []
    for i in range(9): # We are to receive a total of 9 counts
        temp = ""
        '''# Each count is received in 5 7-bit long portions, we
        combine those portions here.'''
        for j in range(5):
            temp = data[j+i*5] + temp
        counts.append(temp)

    ans = []

    '''Here we convert the binary string representations into
    decimal counts.'''
    for i in counts:
        ans.append(bin2dec(i))

    return ans

def GetCounts(s):

    ''' This function checks if there are 51 bytes available to be read,
    and reads if there are. It repeats this 10 times and collects data
    for 1 second this way. It then returns the sum of counts for 1 whole
    second.'''

    num_iter = 0
    total_counts = []

    while True:

        '''Returns an array of total counts received in 1 sec.'''
        if num_iter >= 10:
            return np.sum(total_counts, 0)
        if s.in_waiting > 51:
            total_counts.append(np.array(GetIndvCounts(s)))
            num_iter += 1

        sleep(0.03)

def SaveStop(val):
    plt.savefig("saved")

def Stop(val):
    os._exit(1)


''' Function to run the whole program and return data for the asked '''
def AcquireTrigger(val):
    global acquire
    acquire = 1

def ContinuousCheck(): # Function to monitor counts for past 3 min

    acquisition_time = 120
    ack_reset = acquisition_time # This value needs to be changed once more
    global acquire
    '''Represents the state of acquisition 0 for do not acquire, 1
    for acquire'''
    acquire = 0

    s = Serial("COM4", 19200)
    s.close()
    s.open()
    s.bytesize = 7
    s.stopbits = 2

    FindHeader(s)

    # plt.style.use('dark_background')
    # plt.style.use('seaborn-dark')

    plt.rcParams.update({
    "lines.color": "0.5",
    "patch.edgecolor": "0.5",
    "text.color": "0.95",
    "axes.facecolor": "0.5",
    "axes.edgecolor": "0.5",
    "axes.labelcolor": "0.5",
    "xtick.color": "white",
    "ytick.color": "white",
    "grid.color": "lightgray",
    "figure.facecolor": "0.1",
    "figure.edgecolor": "0.1",
    "savefig.facecolor": "0.1",
    "savefig.edgecolor": "0.1"})

    fig, ((ax1, ax2),(ax3, ax4)) = plt.subplots(nrows = 2, ncols = 2,
    figsize = (10,6), sharex = True)

    b1ax = plt.axes([0.02, 0.93, 0.12, 0.045])

    save_stop_button = wgt.Button(b1ax, "Capture Screen", color="grey",
    hovercolor="#05b5fa")

    save_stop_button.on_clicked(SaveStop)

    b2ax = plt.axes([0.15, 0.93, 0.06, 0.045])

    stop_button = wgt.Button(b2ax, "Stop", color="grey",
    hovercolor="#05b5fa")

    stop_button.on_clicked(Stop)

    b3ax = plt.axes([0.22, 0.93, 0.14, 0.045])

    acquire_button = wgt.Button(b3ax, "Acquire data: "+str(ack_reset)+"s",
    color="grey", hovercolor="#05b5fa")

    acquire_button.on_clicked(AcquireTrigger)

    time = np.array([1])
    tic = 1
    counts = GetCounts(s)
    counts = np.array([GetCounts(s)])

    temp_data = []
    acquisition = 0 # Represents the initital number of acquisition

    for i in range(180):

        tic += 1

        time = np.append(time, tic)

        counts = np.append(counts, [GetCounts(s)], axis = 0)
```

```python
A = counts[:, 0]
B = counts[:, 1]
BP = counts[:, 2]
AP = counts[:, 3]
AB = counts[:, 4]
ABP = counts[:, 5]
APB = counts[:, 6]
APBP = counts[:, 7]
ABBP = counts[:, 8]


if acquire == 1:

    print(".", end = " ")

    if ack_reset < 1:

        acquire = 0
        b4ax = plt.axes([0.37, 0.93, 0.16, 0.045])

        display = wgt.Button(b4ax, "  "+str(ack_reset)+"s | Acq "+
        str(acquisition)+" complete", color="0.2", hovercolor="0.2")

        print("Acquisition no.", acquisition, "completed.")
        np.savetxt("data"+str(acquisition)+".txt", temp_data)
        acquisition += 1
        ''' Restored to the original value, 120 in this case'''
        ack_reset = acquisition_time
        temp_data = []

    else:

        b4ax = plt.axes([0.37, 0.93, 0.04, 0.045])

        display = wgt.Button(b4ax, str(ack_reset)+"s", color="0.2",
        hovercolor="0.2")

        temp_data.append([A[-1], B[-1], BP[-1], AP[-1], AB[-1],
                          ABP[-1], APB[-1], APBP[-1], ABBP[-1]])
        ack_reset -= 1

ax1.cla()
ax2.cla()
ax3.cla()
ax4.cla()
```

```
    ax1.plot(time, A, color = "#45fc03", label = "A")
    ax1.plot(time, AP, color = "#05b5fa", label = "A'")
    ax1.text(time[-1], A[-1], A[-1])
    ax1.text(time[-1], AP[-1], AP[-1])

    ax2.plot(time, B, color = "#45fc03", label = "B")
    ax2.plot(time, BP, color = "#05b5fa", label = "B'")
    ax2.text(time[-1], B[-1], B[-1])
    ax2.text(time[-1], BP[-1], BP[-1])

    ax3.plot(time, AB, color = "#701c8c", label = "AB")
    ax3.plot(time, ABP, color = "#FFD700", label = "AB'")
    ax3.text(time[-1], AB[-1], AB[-1])
    ax3.text(time[-1], ABP[-1], ABP[-1])

    ax4.plot(time, APB, color = "#701c8c", label = "A'B")
    ax4.plot(time, APBP, color = "#FFD700", label = "A'B'")
    ax4.plot(time, ABBP, color = "#FD0E35", label = "ABB'")
    ax4.text(time[-1], APB[-1], APB[-1])
    ax4.text(time[-1], APBP[-1], APBP[-1])

    ax1.legend(loc = 'upper left')
    ax1.set_title("Counts against time")
    ax1.set_ylabel("Counts")

    ax2.legend(loc = 'upper left')
    ax2.set_title("Counts against time")

    ax3.legend(loc = 'upper left')
    ax3.set_xlabel("Time(s)")
    ax3.set_ylabel("Counts")

    ax4.legend(loc = 'upper left')
    ax4.set_xlabel("Time(s)")

    plt.pause(0.001)

while True:

    tic += 1

    time = np.delete(time, 0)
    time = np.append(time, tic)

    counts = np.delete(counts, 0, axis = 0)
    counts = np.append(counts, [GetCounts(s)], axis = 0)

    A = counts[:, 0]
    B = counts[:, 1]
    BP = counts[:, 2]
    AP = counts[:, 3]
    AB = counts[:, 4]
    ABP = counts[:, 5]
    APB = counts[:, 6]
    APBP = counts[:, 7]
    ABBP = counts[:, 8]

    ax1.cla()
    ax2.cla()
    ax3.cla()
    ax4.cla()

    if acquire == 1:

        print(".", end = " ")

        if ack_reset < 1:

            acquire = 0
            b4ax = plt.axes([0.37, 0.93, 0.16, 0.045])

            display = wgt.Button(b4ax, "  "+str(ack_reset)+
            "s | Acq "+str(acquisition)+" complete", color="0.2",
            hovercolor="0.2")

            print("Acquisition no.", acquisition, "completed.")
            np.savetxt("data"+str(acquisition)+".txt", temp_data)
            acquisition += 1

            ''' Restored to the original value, 120 in this case'''
            ack_reset = acquisition_time
            temp_data = []

        else:

            b4ax = plt.axes([0.37, 0.93, 0.04, 0.045])
            display = wgt.Button(b4ax, str(ack_reset)+"s", color="0.2",
                                 hovercolor="0.2")

            temp_data.append([A[-1], B[-1], BP[-1], AP[-1], AB[-1],
                              ABP[-1], APB[-1], APBP[-1], ABBP[-1]])
```

```
                ack_reset -= 1

        ax1.plot(time, A, color = "#45fc03", label = "A")
        ax1.plot(time, AP, color = "#05b5fa", label = "A'")
        ax1.text(time[-1], A[-1], A[-1])
        ax1.text(time[-1], AP[-1], AP[-1])

        ax2.plot(time, B, color = "#45fc03", label = "B")
        ax2.plot(time, BP, color = "#05b5fa", label = "B'")
        ax2.text(time[-1], B[-1], B[-1])
        ax2.text(time[-1], BP[-1], BP[-1])

        ax3.plot(time, AB, color = "#701c8c", label = "AB")
        ax3.plot(time, ABP, color = "#FFD700", label = "AB'")
        ax3.text(time[-1], AB[-1], AB[-1])
        ax3.text(time[-1], ABP[-1], ABP[-1])

        ax4.plot(time, APB, color = "#701c8c", label = "A'B")
        ax4.plot(time, APBP, color = "#FFD700", label = "A'B'")
        ax4.plot(time, ABBP, color = "#FD0E35", label = "ABB'")
        ax4.text(time[-1], APB[-1], APB[-1])
        ax4.text(time[-1], APBP[-1], APBP[-1])

        ax1.legend(loc = 'upper left')
        ax1.set_title("Counts against time")
        ax1.set_ylabel("Counts")

        ax2.legend(loc = 'upper left')
        ax2.set_title("Counts against time")

        ax3.legend(loc = 'upper left')
        ax3.set_xlabel("Time(s)")
        ax3.set_ylabel("Counts")

        ax4.legend(loc = 'upper left')
        ax4.set_xlabel("Time(s)")

        plt.pause(0.001)

    plt.show()

    s.close()

ContinuousCheck()
```

# Appendix B

# Python program for waveplate troubleshooting.

```
import cmath as m
import pylab as pl
import numpy as np
import matplotlib.pyplot as plt

def MatrixMultiply(matrix1, matrix2):
        matrixr = [[0],[0]]
        for i in range(2):
            for j in range(1):
                for k in range(2):
                    matrixr[i][j] += matrix1[i][k] * matrix2[k][j]

        return matrixr

def Send(a, b, phi, tq1, th1, tq2, th2):

    # Making input state
    inp = [[a],[b*m.exp(complex(0,phi))]]


    # Making Waveplates

    # State Generating QWP
    qe11 = complex((m.cos(tq1))**2, (m.sin(tq1))**2)
    qe123 = complex(m.cos(tq1)*m.sin(tq1), -m.cos(tq1)*m.sin(tq1))
    qe14 = complex((m.sin(tq1))**2, (m.cos(tq1))**2)
    qwp1 = [[qe11,qe123],[qe123,qe14]]

    # Basis Changing QWP
    qe21 = complex(m.cos(tq2)**2,m.sin(tq2)**2)
    qe223 = complex(m.cos(tq2)*m.sin(tq2),-m.cos(tq2)*m.sin(tq2))
    qe24 = complex(m.sin(tq2)**2,m.cos(tq2)**2)
    qwp2 = [[qe21,qe223],[qe223,qe24]]

    # State Generating HWP
    hwp1 = [[m.cos(2*th1), m.sin(2*th1)],[m.sin(2*th1), -m.cos(2*th1)]]

    # Basis Changing HWP
    hwp2 = [[m.cos(2*th2), m.sin(2*th2)],[m.sin(2*th2), -m.cos(2*th2)]]

    # Passing through apparatus

    result = MatrixMultiply(qwp1, inp)

    result = MatrixMultiply(hwp1, result)

    return result


# Input state
a = 1
b = 0
phi = 0


# Waveplate angles (All angles in radians)

# State generation waveplates
tq1 = 0
th1 = m.pi/8

# Basis changing waveplates
tq2 = 0
th2 = 0


# Lists to store data

hlist = []
vlist = []
tlist = []
tq1list = []
```

```
tq2list = []
th1list = []
th2list = []


# Making Waveplate Rotations

# State Generating QWP

for i in np.arange(0, 2*m.pi, 0.01):
    tq1 = i
    result = Send(a, b, phi, tq1, th1, tq2, th2)
    tq1list.append(tq1 * 180/m.pi)
    hlist.append(abs(result[0][0])**2)
    vlist.append(abs(result[1][0])**2)

'''
# State Generating HWP
for i in np.arange(0, m.pi, 0.01):
    th1 = i
    result = Send(a, b, phi, tq1, th1, tq2, th2)
    th1list.append(th1 * 180/m.pi)
    hlist.append(abs(result[0][0])**2)
    vlist.append(abs(result[1][0])**2)
'''
'''
# Basis Changing QWP
for i in np.arange(0, 2*m.pi, 0.01):
    tq2 = i
    result = Send(a, b, phi, tq1, th1, tq2, th2)
    tq2list.append(tq2 * 180/m.pi)
    hlist.append(abs(result[0][0])**2)
    vlist.append(abs(result[1][0])**2)
'''
'''
# Basis Changing HWP
for i in np.arange(0, 2*m.pi, 0.01):
    th2 = i
    result = Send(a, b, phi, tq1, th1, tq2, th2)
    th2list.append(th2 * 180/m.pi)
    hlist.append(abs(result[0][0])**2)
    vlist.append(abs(result[1][0])**2)
'''

plt.plot(tq1list, hlist)
plt.plot(tq1list, vlist)
```

# Appendix C

# Complete Verilog code and resulting circuit schematic.

```

// This is the main function which counts the single and coincidence photon detection
// pulses and sends the corresponding count rates to PC via UART every 1/10th of a
// second
module coincidence( output UART_TXD, input clock_50, input A, input B, input C,
input D);

// data_trigger is turned on every 1/10th of a second and begins the data
// stream out
// baud_rate_clk is the clock to output data at the baud rate of 19200
// bits/second
    wire baud_rate_clk;
    wire data_trigger;
    wire Coincidence_0;
    wire Coincidence_1;
    wire Coincidence_2;
    wire Coincidence_3;
    wire Coincidence_4;

// Counts the baud clock until it reaches 1920, which occurs every 1/10th of a second
    reg [14:0] data_trigger_count;

// Turns on every 1/10th of a second for one 100 MHz clock pulse signal and
// resets the photon detection counters
    reg data_trigger_reset;

// Counts the 100 MHz clock pulses until it reaches 5208 in order to time the
// baud clock
    reg [31:0] baud_rate_count;

// Represents the top level design entity instantiation of the number of
// coincidences counted
    wire [31:0] Count_top_0;
    wire [31:0] Count_top_1;
    wire [31:0] Count_top_2;
    wire [31:0] Count_top_3;
    wire [31:0] Count_top_4;

// Output registers of coincident photon counts
    reg [31:0] Count_out_0;
    reg [31:0] Count_out_1;
    reg [31:0] Count_out_2;
    reg [31:0] Count_out_3;
    reg [31:0] Count_out_4;

// Represents the top level design entity instantiation of the number of counts.
    wire [31:0] A_top;
    wire [31:0] B_top;
    wire [31:0] C_top;
    wire [31:0] D_top;

// Output registers of single photon counts.
    reg [31:0] A_out;
    reg [31:0] B_out;
    reg [31:0] C_out;
    reg [31:0] D_out;

// Generation of four coincidence pulses from the input pulses.
    coincidence_pulse CP0( .a(A), .b(B), .y(Coincidence_0));
    coincidence_pulse CP1( .a(A), .b(C), .y(Coincidence_1));
    coincidence_pulse CP3( .a(D), .b(B), .y(Coincidence_2));
    coincidence_pulse CP2( .a(D), .b(C), .y(Coincidence_3));
    three_detector_coincidence CP4(.a(A), .b(B), .c(C), .y(Coincidence_4));

// Counts for a baud rate of 19200 and produces the baud rate clock signal.
    baud_rate_counter BRC1 (.clock_50(clock_50), .baud_rate_clk(baud_rate_clk));

// Uses the baud rate clock signal and generates a trigger signal every 1/10th
// of a second.
    data_triggering DT1 (.baud_rate_clk(baud_rate_clk), .data_trigger(data_trigger));

// Outputs the data in 32-bit registers and resets every 1/10th of a second
    counter C0 ( .clock_50(clock_50), .data_trigger(data_trigger),
    .pulse(Coincidence_0), .q(Count_top_0) );
    counter C1 ( .clock_50(clock_50), .data_trigger(data_trigger),
    .pulse(Coincidence_1), .q(Count_top_1) );
    counter C2 ( .clock_50(clock_50), .data_trigger(data_trigger),
    .pulse(Coincidence_2), .q(Count_top_2) );
    counter C3 ( .clock_50(clock_50), .data_trigger(data_trigger),
    .pulse(Coincidence_3), .q(Count_top_3) );
    counter C4 ( .clock_50(clock_50), .data_trigger(data_trigger),
    .pulse(Coincidence_4), .q(Count_top_4) );
    counter CA ( .clock_50(clock_50), .data_trigger(data_trigger), .pulse(A),
    .q(A_top));
    counter CB ( .clock_50(clock_50), .data_trigger(data_trigger), .pulse(B),
    .q(B_top));
    counter CC ( .clock_50(clock_50), .data_trigger(data_trigger), .pulse(C),
    .q(C_top));
    counter CD ( .clock_50(clock_50), .data_trigger(data_trigger), .pulse(D),
    .q(D_top));

```

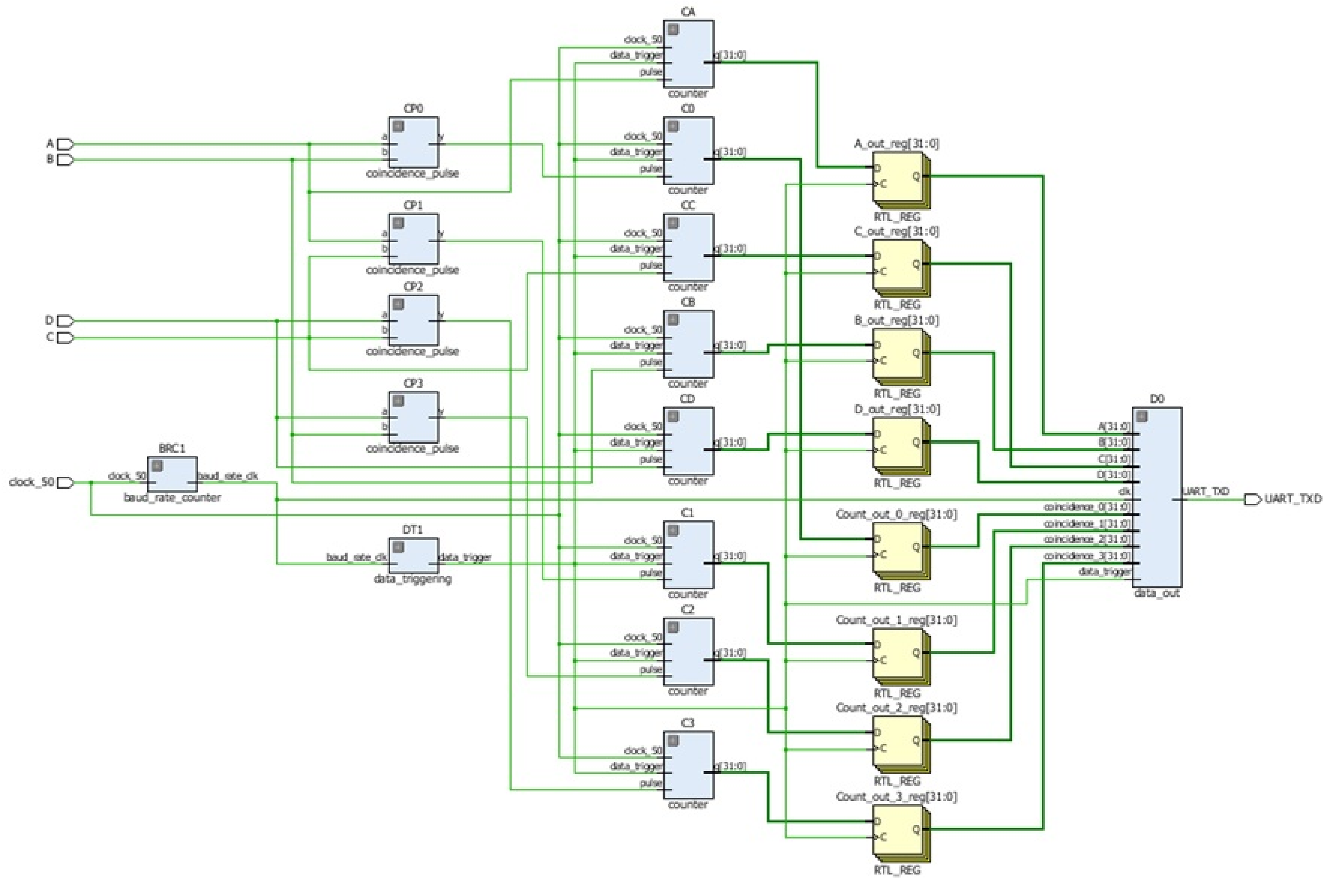
```

// This process updates the counts output arrays every 1/10th of a second
    always@( posedge data_trigger)
        begin
        A_out <= A_top;
        B_out <= B_top;
        C_out <= C_top;
        D_out <= D_top;
        Count_out_0 <= Count_top_0;
        Count_out_1 <= Count_top_1;
        Count_out_2 <= Count_top_2;
        Count_out_3 <= Count_top_3;
        Count_out_4 <= Count_top_4;
        end

// Sends the A, B, C, D and the Coincidence counts out on the RS-232 port
    data_out D0 ( .A(A_out), .B(B_out), .C(C_out), .D(D_out),
    .coincidence_0(Count_out_0), .coincidence_1(Count_out_1),
    .coincidence_2(Count_out_2), .coincidence_3(Count_out_3),
    .coincidence_4(Count_out_4), .clk(baud_rate_clk), .data_trigger(data_trigger),
    .UART_TXD(UART_TXD));

endmodule


// This function ANDs two pulse signals to form one coincidence pulse signal
module coincidence_pulse (input a, input b, output reg y);

    always @(*)
        begin
        y = a && b;
        end

endmodule

module three_detector_coincidence (input a, input b, input c, output reg y);

    always @(*)
        begin
        y = a && b && c;
        end

endmodule


// This function uses the baud rate clock signal and generates a trigger signal
// every 1/10th of a second
module data_triggering (input baud_rate_clk, output reg data_trigger);

reg [31:0] data_trigger_count;

    always @(posedge baud_rate_clk)

        begin
        data_trigger_count <= data_trigger_count  +1;
        //Currently set to 1,920 so we get clock of 10Hz
        if (data_trigger_count == 12'b11110000000)
            begin
            data_trigger <= 1;
            data_trigger_count <=0;
            end
        else
            data_trigger <= 0;
       end

endmodule

// This counter specifically counts for a baud rate of 19200 and produces a
// corresponding baud rate clock signal
module baud_rate_counter (input clock_50, output reg baud_rate_clk);

reg [31:0] baud_rate_count;

    always@(posedge clock_50)
        begin
        baud_rate_count <= baud_rate_count +1;

        if (baud_rate_count >= 5208)
        begin
            baud_rate_clk <= 1;
            baud_rate_count <=0;
        end

        else
            baud_rate_clk <= 0;
       end

endmodule

// This function counts voltage pulses
module counter(input clock_50, input data_trigger, input pulse, output reg [31:0]q);

    wire x;

    or o1 (x, data_trigger, pulse);

    always @ (posedge x)
        begin

        if (data_trigger)
            q <=0;

        else
            q<=q+1;

        end

endmodule

// This function sends out up to four single photon counts and up to four coincidence
// counts to the PC through serial communication (UART)
module data_out(input [31:0] A, input [31:0] B, input [31:0] C, input [31:0] D,
input[31:0] coincidence_0, input [31:0] coincidence_1, input [31:0] coincidence_2,
input [31:0] coincidence_3, input [31:0] coincidence_4, input clk,
input data_trigger, output reg UART_TXD);

```

```
    reg [5:0] index;
    reg [0:31] incremental;
    reg [31:0] out;
    reg [3:0] data_select;

    always @ (posedge clk)

        begin

        if (index == 6'b111111 && data_trigger == 1)
            begin
            index <= 6'b000000;
            UART_TXD <= 1;
            // Sending out a header number to detect before taking counts
            out <= 32'b10011001100010010101010101011101;
            data_select <= 3'b000;
            end

        else if (index == 6'b000000)
            begin
            index <= 6'b000001;
            UART_TXD <= 0;
            end

        else if (index == 6'b000001)
            begin
            index <= 6'b000010;
            UART_TXD <= out[0];
            end

        else if (index == 6'b000010)
           begin
            index <= 6'b000011;
            UART_TXD <= out[1];
            end

        else if (index == 6'b000011)
            begin
            index <= 6'b000100;
            UART_TXD <= out[2];
            end

        else if (index == 6'b000100)
            begin
            index <= 6'b000101;
            UART_TXD <= out[3];
            end

        else if (index == 6'b000101)
            begin
            index <= 6'b000110;
            UART_TXD <= out[4];
            end

        else if (index == 6'b000110)
            begin
            index <= 6'b000111;
            UART_TXD <= out[5];
            end

        else if (index == 6'b000111)
            begin
            index <= 6'b001000;
            UART_TXD <= out[6];
            end

        else if (index == 6'b001000)
            begin
            index <= 6'b001001;
            UART_TXD <= 0;
            end

        else if (index == 6'b001001)
            begin
            index <= 6'b001010;
            UART_TXD <= 1; // the first stop bit
            end

        else if (index == 6'b001010)
            begin
            index <= 6'b001011;
            UART_TXD <= 0; // the second start bit
            end

        else if (index == 6'b001011)
            begin
            index <= 6'b001100;
            UART_TXD <= out[7];
            end

        else if (index == 6'b001100)
            begin
            index <= 6'b001101;
            UART_TXD <= out[8];
            end

        else if (index == 6'b001101)
            begin
            index <= 6'b001110;
            UART_TXD <= out[9];
            end

        else if (index == 6'b001110)
            begin
            index <= 6'b001111;
            UART_TXD <= out[10];
            end

        else if (index == 6'b001111)
            begin
            index <= 6'b010000;
            UART_TXD <= out[11];
            end

        else if (index == 6'b010000)
            begin
            index <= 6'b010001;
            UART_TXD <= out[12];
            end

        else if (index == 6'b010001)
            begin
            index <= 6'b010010;
            UART_TXD <= out[13];
            end

        else if (index == 6'b010010)
            begin
            index <= 6'b010011;
            UART_TXD <= 0; // the termination bit
            end

        else if (index == 6'b010011)
            begin
            index <= 6'b010100;
            UART_TXD <= 1; // the second stop bit
            end

        else if (index == 6'b010100)
            begin
            index <= 6'b010101;
            UART_TXD <= 0; // the third start bit
            end

        else if (index == 6'b010101)
            begin
            index <= 6'b010110;
            UART_TXD <= out[14];
            end

        else if (index == 6'b010110)
            begin
            index <= 6'b010111;
            UART_TXD <= out[15];
            end

        else if (index == 6'b010111)
            begin
            index <= 6'b011000;
            UART_TXD <= out[16];
            end

        else if (index == 6'b011000)
            begin
            index <= 6'b011001;
            UART_TXD <= out[17];
            end

        else if (index == 6'b011001)
            begin
            index <= 6'b011010;
            UART_TXD <= out[18];
            end

        else if (index == 6'b011010)
            begin
            index <= 6'b011011;
            UART_TXD <= out[19];
            end

        else if (index == 6'b011011)
            begin
            index <= 6'b011100;
            UART_TXD <= out[20];
            end

        else if (index == 6'b011100)
            begin
            index <= 6'b011101;
            UART_TXD <= 0; // the termination bit
            end

        else if (index == 6'b011101)
            begin
            index <= 6'b011110;
            UART_TXD <= 1; // the third stop bit
            end

        else if (index == 6'b011110)
            begin
            index <= 6'b011111;
            UART_TXD <= 0; // the fourth start bit
            end

        else if (index == 6'b011111)
            begin
            index <= 6'b100000;
            UART_TXD <= out[21];
            end

        else if (index == 6'b100000)
            begin
            index <= 6'b100001;
            UART_TXD <= out[22];
            end

        else if (index == 6'b100001)
            begin
            index <= 6'b100010;
            UART_TXD <= out[23];
            end

        else if (index == 6'b100010)
            begin
            index <= 6'b100011;
            UART_TXD <= out[24];
            end

        else if (index == 6'b100011)
            begin
            index <= 6'b100100;
            UART_TXD <= out[25];
```

```
                end

        else if (index == 6'b100100)
                begin
                index <= 6'b100101;
                UART_TXD <= out[26];
                end

        else if (index == 6'b100101)
                begin
                index <= 6'b100110;
                UART_TXD <= out[27];
                end

        else if (index == 6'b100110)
                begin
                index <= 6'b100111;
                UART_TXD <= 0;  // termination bit
                end

        else if (index == 6'b100111)
                begin
                index <= 6'b101000;
                UART_TXD <= 1; // the fourth stop bit
                end

        else if (index == 6'b101000)
                begin
                index <= 6'b101001;
                UART_TXD <= 0; // the fifth start bit
                end

        else if (index == 6'b101001)
                begin
                index <= 6'b101010;
                UART_TXD <= out[28];
                end

        else if (index == 6'b101010)
                begin
                index <= 6'b101011;
                UART_TXD <= out[29];
                end

        else if (index == 6'b101011)
                begin
                index <= 6'b101100;
                UART_TXD <= out[30];
                end

        else if (index == 6'b101100)
                begin
                index <= 6'b101101;
                UART_TXD <= out[31];
                end

        else if (index == 6'b101101)
                begin
                index <= 6'b101110;
                UART_TXD <= 0;
                end

            else if (index == 6'b101110)
                begin
                index <= 6'b101111;
                UART_TXD <= 0;
                end

            else if (index == 6'b101111)
                begin
                index <= 6'b110000;
                UART_TXD <= 0;
                end

            else if (index == 6'b110000)
                begin
                index <= 6'b110001;
                UART_TXD <= 0;
                end

            else if (index == 6'b110001 && data_select == 4'b0000)
                begin
                index <= 6'b000000;
                data_select <= 4'b0001; // increments data_select to begin output of B
                out <= A;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0001)
                begin
                index <= 6'b000000;
                data_select <= 4'b0010; // increments data_select to begin output of B
                out <= B;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0010)
                begin
                index <= 6'b000000;
                data_select <= 4'b0011; // increments data_select to begin output of C
                out <= C;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0011)
                begin
                index <= 6'b000000;
                data_select <= 4'b0100; // increments data_select to begin output of D
                out <= D;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0100)
                begin
                index <= 6'b000000;
                // increments data_select to begin output of Coincidence_0
                data_select <= 4'b0101;
                out <= coincidence_0;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0101)
                begin
                index <= 6'b000000;
                // increments data_select to begin output of Coincidence_1
                data_select <= 4'b0110;
                out <= coincidence_1;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0110)
                begin
                index <= 6'b000000;
                // increments data_select to begin output of Coincidence_2
                data_select <= 4'b0111;
                out <= coincidence_2;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b0111)
                begin
                index <= 6'b000000;
                // increments data_select to begin output of Coincidence_3
                data_select <= 4'b1000;
                out <= coincidence_3;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b1000)
                begin
                index <= 6'b000000;
                // increments data_select to begin output of Coincidence_4
                data_select <= 4'b1001;
                out <= coincidence_4;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110001 && data_select == 4'b1001)
                begin
                index <= 6'b110010;
                UART_TXD <= 1; // the fifth stop bit
                end

            else if (index == 6'b110010)
                begin
                index <= 6'b111111;
                UART_TXD <= 0; // the start bit of the termination byte
                end

            else
                begin
                index <= 6'b111111;
                UART_TXD <= 1; // sets all subsequent bits to negative voltage
                end
```

```
    end

endmodule
```

In the schematic given below, starting from the left, the small white blocks represent the inputs to the FPGA. These are the four inputs we get from the single-photon detectors, and a clock signal coming from the oscillator placed on the FPGA. The blue blocks after that represent the modules we wrote in the FPGA programming chapter. All the functionality that we implemented in the modules is being implemented inside these blue blocks in the schematic. Following these blue block are some yellow blocks, These yellow blocks represent the registers in which the value is temporarily stored before it is sent to the PC. Finally, the rectangle at the end represents the UART communication bus that will send all the data stored in the registers to the attached PC via serial communication..


A
B
D
C
clock_50
CP0
CP1
CP2
CP3
coincidence_pulse
BRC1
baud_rate_counter
baud_rate_clk
DT1
data_triggering
data_trigger
CA
C0
CC
CB
CD
C1
C2
C3
clock_50
data_trigger
pulse
q[31:0]
counter
A_out_reg[31:0]
C_out_reg[31:0]
B_out_reg[31:0]
D_out_reg[31:0]
Count_out_0_reg[31:0]
Count_out_1_reg[31:0]
Count_out_2_reg[31:0]
Count_out_3_reg[31:0]
RTL_REG
D0
A[31:0]
B[31:0]
C[31:0]
D[31:0]
clk
coincidence_0[31:0]
coincidence_1[31:0]
coincidence_2[31:0]
coincidence_3[31:0]
data_trigger
data_out
UART_TXD

# Bibliography

[1] *How to align a laser*. Thorlabs Insights. 2020.
https://www.youtube.com/watch?v=qzxILY6nOmA

[2] *Experimental test of the violation of local realism in quantum mechanics without bell inequalities*. G. Di Giuseppe, F. De Martini, and D. Boschi. Phys. Rev. A, 56:176–181, Jul 1997.
https://journals.aps.org/pra/abstract/10.1103/PhysRevA.56.176

[3] *Quantum optics: an introduction*. M. Fox. Oxford master series in atomic, optical, and laser physics. Oxford Univ. Press, Oxford, 2006.
https://cds.cern.ch/record/1001868

[4] *Single-photon sources*. B. Lounis and M. Orrit. Reports on Progress in Physics, 68(5):1129–1179, apr 2005.
https://doi.org/10.1088/0034-4885/68/5/r04

[5] *Realization of quantum wheeler's delayed-choice experiment.* J.-S. Tang, Y.-L. Li, X.-Y. Xu, G.-Y. Xiang, C.-F. Li, and G.-C. Guo.Nature Photonics, 6(9):600–604, 2012.

[6] *Quantum Mechanics in the Single Photon Laboratory*. M. H. Waseem, F. Illahi, & M. S. Anwar. IOP Publishing, July 2020. Online ISBN: 978-0-7503-3063-3, Print ISBN: 978-0-7503-3061-9.
https://doi.org/10.1088/978-0-7503-3063-3