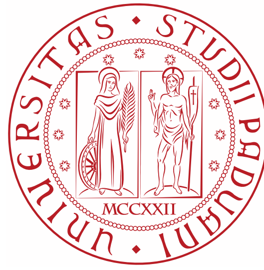


Università degli Studi di Padova

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Study and Design of a Control System for generating and detecting Qubits for the Quantum Exchange of Cryptographic Key

Relatore:
Prof. Paolo Villoresi

Correlatore:
Dott. Giuseppe Vallone

Laureando:
Andrea Stanco

Anno Accademico
2013/2014

Abstract

This thesis presents the realization of a system for quantum information exchange for cryptographic applications. The basic element of quantum information is the *qubit*, which replaces the classical bit concept. This leads to several new implications in the handling and measurement of the information. The purpose of the work is to design all the parts required for the creation of a reliable system. The project, which is named Quantum Advanced Key Exchanger (QuAKE), covers many different technological areas. `VHDL` language describes the hardware core for the driving of the lasers and data reception; `C++` language implements the software part for the management of the entire system. The design is integrated on two boards, each one connected to a computer. The system design faces the problem of linking together two devices with different clock domains. Therefore, a fine synchronization method guarantees the right timing of the system. Furthermore, the unpredictably of the qubits transmission is handled through a suitable time-calibrated receiving windows system. Actually, the work succeeds in improving a previous version of the system increasing its performances, reliability, efficiency and flexibility.

Contents

1 Quantum information and cryptography **7**
- 1.1 Quantum information . . . 7
  - 1.1.1 The qubit . . . 7
- 1.2 Introduction to cryptography . . . 10
- 1.3 Current cryptography technology . . . 11
- 1.4 Quantum Cryptography . . . 12
  - 1.4.1 Quantum cryptographic system . . . 12
  - 1.4.2 BB84 protocol . . . 12
  - 1.4.3 B92 protocol . . . 14

2 QuAKE Experiment **17**
- 2.1 Brief QuAKE description . . . 18
- 2.2 ZedBoard device . . . 19
- 2.3 Software tools . . . 22

3 Hardware Design **23**
- 3.1 Basic System Overview . . . 23
- 3.2 Usable technical knowledge . . . 23
  - 3.2.1 Zynq All Programmable SoC . . . 24
  - 3.2.2 AXI Protocol . . . 25
  - 3.2.3 Finite State Machine . . . 25
- 3.3 Alice Hardware Implementation . . . 27
  - 3.3.1 Laser Controller Module . . . 27
  - 3.3.2 Zeus Synchronization Module . . . 31
  - 3.3.3 Mind Well Memory Manager . . . 32
  - 3.3.4 Other Modules . . . 33
  - 3.3.5 Xilinx-Vivado IP Cores . . . 35
- 3.4 Bob Hardware Implementation . . . 37
  - 3.4.1 Reflex Photon Translator . . . 37
  - 3.4.2 Laser Pinball Communication Sorter . . . 37
  - 3.4.3 Hermes Synchronization Module . . . 43
  - 3.4.4 Bit Ben Word Composer . . . 43
  - 3.4.5 Other Modules . . . 45

4 Softwares Design **47**
4.1 SDK software 47
4.1.1 TCP instructions coding 50
4.2 *uQuake* user interface 51

5 Test and future of QuAKE **55**
5.1 Hardware Testing 55
5.2 Software testing 55
5.3 Whole system testing 56
5.4 Future developments 57

A Clocks Drift Measures **61**

B Zedboard Extension Pmod Interface **65**

C Reset: a-synchronous analysis **67**

D `C++` interrupt routine and `XAxiCdma_SimpleTransfer` **69**

Introduction

Nowadays, there is an increasing need for an efficient and optimized technology for the exchange of the information. Given that, using *qubits* instead of classical bits, the *quantum information* is acquiring more and more relevance in this scenario.
Furthermore, the security of communication is one of the most important aspects in an era in which information technology plays a crucial role. Cryptography is unavoidable for military application and sensitive data communication as well as for privacy security. In addition to that, cryptography represents an essential element for the support of modern information economy. Therefore, the combination of quantum information and cryptography represents one of the most important achievements of the last three decades.

Any device responsible for the information exchange must be fully reliable. Thus, the design of a stable and trustworthy communication system is the first step for the development of new technology and the improvement of the existing one. This project-design work was born within this framework to meet the needs of quantum information field. As a matter of facts, the project starts from the already existing Quantum Advanced Key Exchanger (QuAKE) system, which needs to be improved for better performances and flexibility.

The thesis describes the realization from scratch of a system that works as a foundation for quantum information communication. Since the system might be used in different future quantum cryptography experiments, this report can be considered as an operational manual as well.
The thesis addresses different topics, starting from the introduction of some preliminary notions on quantum information and cryptography that are given in chapter 1. Chapter 2 introduces the QuAKE system which main purpose is to allow the transmission of the information, encoded as qubits, between two parties. Then, the main objectives set for the design of the new version of QuAKE are described. The development of such new project is divided into three different sub-designs: a hardware one and two software programs.
The study and design of the hardware part is described in chapter 3. Starting with a brief analysis on the available technical knowledge, the chapter fully explains the hardware working principle highlighting its modular structure.
Chapter 4 deals with the realization and complete explanation of two different software programs. The first one for the system working and the second one for the system con-

trolling.
The last chapter is dedicated to the description of the tests carried out on the system. After the test and verification of both the software and the hardware parts, the results of the complete test of the whole system are reported.
Four appendices complete the work with a further description and analysis of specific measurements and design of a dedicated device.

The achievements prove that the system works properly and represents a reliable foundation from which future developments can start.

Chapter 1

Quantum information and cryptography

1.1 Quantum information

Quantum information is a quite recent field for the exchange of the information. It is based on quantum mechanics, and represents an evolution of the classical information theory. This technology can be used in many different areas and applications. The main one is cryptography. In quantum information, the central idea is that the basic element for the storing of a single information is the *qubit* instead of the classical bit, which is the most widely used. Therefore, the elaboration, the communication and the storage of the information are done over the qubits. As a matter of facts, the great challenge in quantum information is the handling of the qubits, since they need to follow the *prepare-transform-measure* path [1] in order to realize a reasonable transmission of the information. The processing is performed using protocols. The cryptographic application of quantum information is implemented through the Quantum Key Distribution (QKD) with the usage of BB84 and B92 protocols.
For a clear understanding, the next part will explain the concept of qubit.

1.1.1 The qubit

A qubit represents the quantum information, which is the basic element for the encoding of the information in a quantum system. The therm qubit stands for *quantum bit* and intuitively it can be compared to the computer *bit*. Nevertheless, they represents deeply different concepts. The qubit is a two-level quantum system, described by a two-dimensional Hilbert space. Using two normalized and mutually orthogonal quantum states, it is possible to map the values 0 and 1 of a classical bit:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.1}$$

These two quantum states form a *computational basis* of its space. Thus, unlike a bit that can assume only two allowed values, a qubit can be represented by the superposition of the two states $|0\rangle$ and $|1\rangle$. The general form of the states superposition is described by

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \tag{1.2}$$

where α,$\beta \in \mathbb{C}$ with $|\alpha^2| + |\beta^2|$=1, that is $\langle\psi \,|\, \psi\rangle$=1. The state vectors do not have physical meaning but only describe a global phase. Therefore, choosing real and positive values for α and β, the generic state of a qubit can be written using spherical coordinates:

$$|\psi\rangle = \cos\frac{\theta}{2} |0\rangle + e^{i\phi} \sin\frac{\theta}{2} |1\rangle = \begin{bmatrix} \cos\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} \end{bmatrix} \qquad (0 \leq \theta \leq \pi, 0 \leq \phi \leq 2\pi) \tag{1.3}$$

Using a three-dimensional Cartesian space, this state can be represented as a unit vector lying on a sphere of unit radius know as *Bloch sphere*. Clearly, the Cartesian coordinates of the state can be referred to the polar coordinates:

$$x = cos\phi sin\theta, \quad y = sin\phi sin\theta, \quad z = cos\theta \tag{1.4}$$

The representation of the qubit as a linear combination of the computational basis $\{|0\rangle, |1\rangle\}$ is not necessarily the only one possible. As a matter of facts, the state of a qubit in a two-dimensional Hilbert space can be represented as a linear combination of any two orthonormal state vectors. Therefore, it can be represented with the *conjugate bases*

$$\frac{|0\rangle + |1\rangle}{\sqrt{2}}, \; \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{1.5}$$

$$\frac{|0\rangle + i\,|1\rangle}{\sqrt{2}}, \; \frac{|0\rangle - i\,|1\rangle}{\sqrt{2}} \tag{1.6}$$

The couple of the vectors forming each one of the conjugate bases correspond to the Pauli eigenvectors operators σ_z, σ_x and σ_y. They form a set of mutually orthogonal vectors over the Bloch sphere (Figure 1.1).

Now it is possible to use any two-level quantum system in order to create a physical qubit. For example, the spin of a electron, the spin of a photon or two electronic levels of an atom can be considered for qubit realization. However, due to its nature, the polarization of a light wave is a quite simple task [2]. Hence, many quantum communication systems use the polarization itself to encode the qubit. Thus, it is possible to associate the $|0\rangle$ and $|1\rangle$ states, respectively, to the horizontal and vertical states of polarization of a photon.
From a formal point of view, the new basis is still a computational one:

$$|0\rangle \rightarrow |H\rangle, \quad |1\rangle \rightarrow |V\rangle \tag{1.7}$$

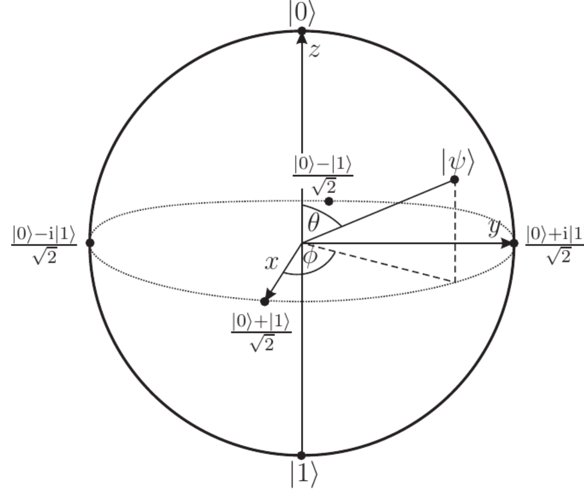


Figure 1.1: The Bloch sphere.

Hence, the diagonal polarization basis (+/-) and the left/right polarization basis (L/R) form the new conjugate bases and are related to the H/V basis as follow:

$$|+\rangle = \frac{|H\rangle + |V\rangle}{\sqrt{2}}, \ \ |-\rangle = \frac{|H\rangle - |V\rangle}{\sqrt{2}} \tag{1.8}$$

$$|L\rangle = \frac{|H\rangle + i\,|V\rangle}{\sqrt{2}}, \ \ |R\rangle = \frac{|H\rangle - i\,|V\rangle}{\sqrt{2}} \tag{1.9}$$

Even in this case, the three bases correspond the Pauli eigenvectors operators σ_z, σ_x and σ_y. Through the *Poincarè sphere* (Figure 1.2), which is quite similar to the Bloch sphere, the light wave polarization states can be visually represented.

Considering the explanation above, it is clear that a qubit can assume an infinite number of states. Therefore, one could think that it could be possible to store an infinite quantity of information in a single qubit. Nevertheless, aside the fact that it is possible to encode a qubit as an arbitrary polarization of a photon, the subsequent measure of the qubit state has a limited precision due to quantum mechanics laws. To be more precise, is not possible to measure the precise state of a qubit since it would require infinite measures. Besides, even with a large number of measures on the same qubit, is not even possible to estimate, with reasonable precision, the state of a qubit, since any measurement will inevitably modify the original qubit state. The theoretical reasons of these two statement derive from the the third postulate of the quantum mechanics theory.
For a complete explanation about qubit measurement and relation with quantum mechanics theory, please refer to [3].

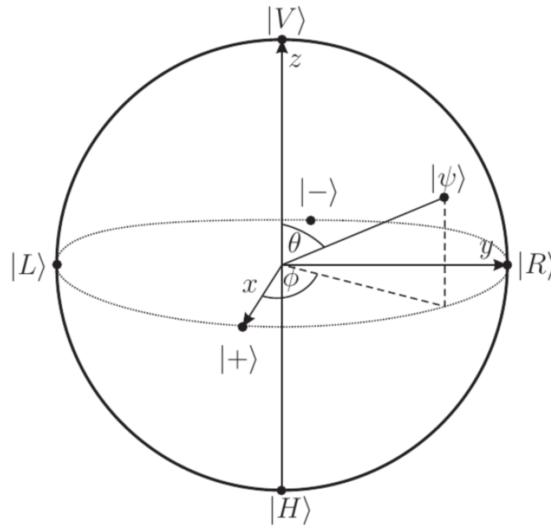


Figure 1.2: The Poincarè sphere.

1.2 Introduction to cryptography

Cryptography science studies the techniques for a secure communication between two parties. The therm comes from the Greek words $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$, "hidden" and $\gamma\rho\acute{\alpha}\varphi\epsilon\iota\nu$, "writing". The purpose of cryptography is to transform a message in order to make it unreadable and information-less but allowing the recovery of the original message only to the designated party.

Cryptography is rooted in ancient ages. It was closely tied with military needs, which boosted the invention of different encryption methods during centuries.
A first important example is the *Caesar cipher*, used to send textual messages. It was based on a simple scheme: every letter was substituted for another one located nth-positions far from it in the alphabet. But the most famous cryptographic system is the *Enigma machine*, used by Germans during the II World War to communicate reserved military information among troupes. The machine was based on an electro-mechanical system and on an alphabetical algorithm, it was easy to use and was considered very secure for that time. As a matter of facts, the main idea of these systems was that the message encryption algorithm was secret and unknown to a third party. Nevertheless, the alphabetic substitution technique allowed the decryption of the algorithm through dedicated analysis of letters frequency. This means that, despite the good efficiency of these systems for that time, they still did not guarantee a very high level of security.

During the last century, the spreading of information engineering increased the need for a secure communication, which fostered the development of cryptography science. Therefore, many new algorithms were invented in order to improve the security of the message transmitted. One of them was the *Vernam cipher* or *one-time-pad* by Gilber

Vernam, back to 1920. The encryption algorithm is used on a binary message and performs a sum modulo two between each bit of the original message and the related bit of a random key. The key has to be as long as the message is. This algorithm is considered to be fully secure as long as the key is used only once.

Nowadays, one of the hardest tasks for cryptography is to exchange the key used by the encryption algorithm.
The quantum cryptography is one of the most recent developed technologies and could become the one used worldwide in the near future.

1.3 Current cryptography technology

Nowadays the most widespread cryptographic system is based on the public/private key algorithm. This system allows most of the today communications and its working principle is based on a mathematical procedure. Basically, the algorithm implements an almost-not-reversible function which allows the transmission of the encrypted message without transmitting the whole cryptographic key.
Let us assume the $\mathcal{X}$ message has to be send through a communication channel. The idea is to encrypt this message with an $\mathscr{F}$ function and the public key in order to create the encrypted message $\mathcal{Y}$. Actually, the only way to obtain $\mathcal{X}$ from $\mathcal{Y}$ is to use the $\mathscr{H}$ function and the private key.

$$\mathcal{X} \overset{\mathscr{F}}{\to} \mathcal{Y}$$
$$\mathcal{Y} \overset{\mathscr{H}}{\to} \mathcal{X} \qquad \mathscr{H} \neq \mathscr{F}^{-1}$$

Here is an example of the information exchange.
Suppose that A needs to send a message to B. Hence, B creates both a public key and a private one. The public key is then sent to A and is visible to anyone. Once it has received the public key, A encrypts its message $\mathcal{X}$, using the $\mathscr{F}$ function and the public key, and obtains the encrypted message $\mathcal{Y}$. Now A sends $\mathcal{Y}$ to B which will decrypt the message using the $\mathscr{H}$ function and the private key. The $\mathcal{Y}$ message too is visible to anyone but cannot be decrypted without the private key. In other words, even knowing the public key and the encrypting $\mathscr{F}$ function, it is not possible to obtain the original message $\mathcal{X}$ in a reasonable period of time. To be more precise, the $\mathcal{Y} \overset{\mathscr{F}^{-1}}{\to} \mathcal{X}$ operation is not impossible but simply represents a mathematical problem with exponential complexity (For a specific description of the public/private key system, please refer to [4]). As a matter of facts, this problem requires an outstanding amount of time to be solved, even for the most powerful computer. Irony of fate, the only computer able to solve this problem in a reasonable period of time could be the quantum one. Nowadays, the quantum computer is far to be completely functioning but probably, in the near future, it will and the quantum cryptography could be one of the safest ways to transmit encrypted messages.

1.4 Quantum Cryptography

The quantum cryptography is the application of quantum information to the world of the cryptography. The first idea of this application was formulated in 1970 by Stephen J. Wiesner creating the basis for the development of this new discipline. Later, in 1984, researchers Charles H. Bennet and Gilles Brassard invented the first quantum cryptography protocol called BB84. Although it was not complex from a theoretical point of view, the quantum cryptography technology was mainly ignored at that time. However, its development continued and from the '90s its consideration started to grow. Nowadays, the quantum cryptography is well considered since its importance for the future of information security.

1.4.1 Quantum cryptographic system

In a quantum cryptographic system, a classical channel and quantum channel[1] are available for the communication between a transmitter, commonly named Alice, and a receiver, commonly named Bob.
The classical channel is a simple physical medium used by Alice and Bob in order to communicate information that are visible to everyone. On the other hand, the quantum channel is a private channel used by Alice and Bob for the communication of reserved information (i.e. a key), which are encoded as qubits. Therefore, the main idea of a quantum cryptographic system is to use the intrinsic unpredictability of a quantum measurement in order to detect the presence of an eavesdropper, commonly named Eve. As a matter of facts, Eve needs to make a measure over the exchanged qubits in order to intercept the private transmission between Alice and Bob. However, the measure will necessarily perturb the qubits quantum state. Therefore, the usage of specific quantum states and of a proper key error rate analysis can allow the detection of the presence of an intruder.
Please note that a quantum cryptographic system is not used for messages encryption but is responsible only for the exchange of a key that will be then used for messages encryption. This specific application is known as Quantum Key Distribution (QKD). A description of two of the most well-known cryptography protocols follows.

1.4.2 BB84 protocol

As already stated, the BB84 protocol was the first quantum cryptography protocol ever invented. Its name derives from its inventors, Bennet and Brassard [5].
The protocol encodes the values 0 and 1 of a randomly generated raw key, using four different quantum states from two conjugate bases. In case of polarization encoded qubits,

[1] Please note that therm *quantum* does not imply that the channel is based on quantum mechanic principles. To be more precise, the channel is still a classical one but its information is processed according to quantum mechanic.

Bit/Basis	+	×
0	$\|H\rangle$	$\|+\rangle$
1	$\|V\rangle$	$\|-\rangle$

Table 1.1: BB84 bit encoding lookup table

the two bases can be chosen as the H/V polarization basis (+) and the 45°diagonal +/- polarization basis (×) ($\{|H\rangle, |V\rangle\}$ and $\{|+\rangle, |-\rangle\}$). Therefore, the protocol works as follow:

Alice-side (*quantum channel*)

1. A raw key is created as a random sequence of bits.
2. Every bit is randomly associated to one of the two possible basis (according to Table 1.1) in order to encode a qubit. The bases assignment is stored for the post-processing phase.
3. Alice sends photons according to polarization states of the qubits.

Bob-side (*quantum channel*)

1. Bob measures the polarization state of the incoming photons with a random association to one of the two bases.
2. The data are saved storing the value (0 or 1) and the basis.

Data-processing (*classical channel*)

1. Bob communicates to Alice which qubits it has detected. As a matter of facts, some of them can have been lost because of channel losses.
2. now Alice and Bob must verify that the Bob basis of the nth-qubit is the same of Alice. When a basis is matched, even the bit is matched. If the basis does not correspond, the bit can be correct with a probability of 50% and must be discarded. The sequence of matched bases/bits forms the *sifted key.*
3. The most important step is to verify the presence of an eavesdropper during the transmission of the qubits. This can be done with a control on the error rate of the raw key, also called Quantum Bit Error Rate (QBER). Therefore, Alice and Bob share a small portion of the raw key on the classic channel, evaluating the percentage of wrong bits. A specific threshold QBER value[2] is used in order to decide if the key is completely safe or not. The exceeding of this threshold

[2]Different threshold value mean different level of security. Generally, a threshold of 11% represents an absolute secure key. For a complete explanation about the threshold values refer to [6].

implies the presence of an eavesdropper during the transmission and so the entire key must be discarded. To be more precise, there is no absolute certainty of the eavesdropper presence since the over-the-threshold QBER can be due to the errors introduced by the quantum channel and optical components.

4. If the QBER value is below the threshold, Alice and Bob can make a procedure of error correction on the sifted key commonly known as *information reconciliation*.

5. The last step is the *privacy amplification*. The key is mapped over a new one, which will be completely uncorrelated from the portion previously shared on the classical channel.

An example of the working principle of BB84 is shown in Table 1.2.

Alice random bits	0	1	0	0	0	0	1	0
Alice random bases	$\times$	$+$	$\times$	$\times$	$\times$	$+$	$+$	$\times$
Transmitted polarization	$\|+\rangle$	$\|H\rangle$	$\|+\rangle$	$\|-\rangle$	$\|-\rangle$	$\|V\rangle$	$\|V\rangle$	$\|-\rangle$
Bob random bases	$+$	$+$	$+$	$\times$	$+$	$\times$	$+$	$\times$
Raw key	1	1	0	0	0	1	1	0
Sifted key		1		0			1	0

Table 1.2: BB84 operation example

Generally, since the Bob bases are wrong 50% of the times, the BB84 efficiency can be considered to be equal to 50%.

1.4.3 B92 protocol

A variant of the BB84 protocol was proposed by Bennet in 1992 and uses only two non-orthogonal states of polarization [7]. This different protocol works as follow:

Alice-side (*quantum channel*)

1. A raw key is created as a random sequence of bits.

2. Every bit is randomly encoded with one of the two possible polarization according to Table 1.3. The polarization assignment is stored for the post-processing phase.

Bit/Party	Alice	Bob
0	$\|V\rangle$	$\|-\rangle$
1	$\|+\rangle$	$\|H\rangle$

Table 1.3: B92 bit encoding lookup table

3. Alice sends photons according to polarization states of the qubits.

Bob-side (*quantum channel*)

1. Bob measures the polarization state of the incoming photons randomly using one of the other two polarization direction. Therefore, Bob has a 50% probability of detecting a photon only when the used polarization direction is correlated with the one used by Alice. If not, Bob does not detect the photon at all.

Data-processing (*classical channel*)

1. Bob communicates to Alice when it has detect a photon. Hence, the sifted key is created using the all bits related to a detected photon.

2. The protocol continues with the same steps of BB84 after the creation of the sifted key.

An example of the operation of the B92 protocol is shown in Table 1.4.

Alice random bits	0	1	0	0	0	0	1	1
Transmitted polarization	$\|V\rangle$	$\|+\rangle$	$\|V\rangle$	$\|V\rangle$	$\|V\rangle$	$\|V\rangle$	$\|+\rangle$	$\|+\rangle$
Bob random directions	$\|H\rangle$	$\|H\rangle$	$\|-\rangle$	$\|-\rangle$	$\|H\rangle$	$\|-\rangle$	$\|H\rangle$	$\|-\rangle$
Bob click		X		X		X	X	
Raw key		1		0		0	1	
Sifted key		1		0		0	1	

Table 1.4: B92 operation example

The protocol has an efficiency of 25%. As a matter of facts, 50% of the times Bob measures the photon with the wrong polarization direction. Furthermore, even with the right polarization direction, there is another 50% possibility not to detect the photon. Thus, this protocol does not offer any efficiency advantages. However, the implementation is clearly easier than the BB84.

Chapter 2

QuAKE Experiment

QuAKE (Quantum Advanced Key Exchanger) is a system for quantum cryptography communication that was developed and realized at Luxor Laboratory of Department of Information Engineering in Padua, Italy. The experiment was intended for quantum cryptography in free space using the B92 protocol. It took several years to realize it with the cooperation of many people. Once its implementation was completed, the possibility for a project upgrade became feasible in order to increase its applications for quantum information exchange. Therefore the new QuAKE system sets these new objectives:

- possibility to switch between B92 and BB84 protocols
- improving synchronization system and frames management
- more flexibility and more settable parameters
- adaptability to different optical setups
- improving hardware reliability
- new on-board software
- possibility to send more than 500 thousand qubits with one key
- TCP connection for computer communication
- new user interface software for full control of the system

Hence, the thesis work required the development and test of all these new features. The project design focused on the hardware and software part, laying foundations for future usage of different optical setups. For a complete description of the old QuAKE, including optical setup explanation, please refer to [8].

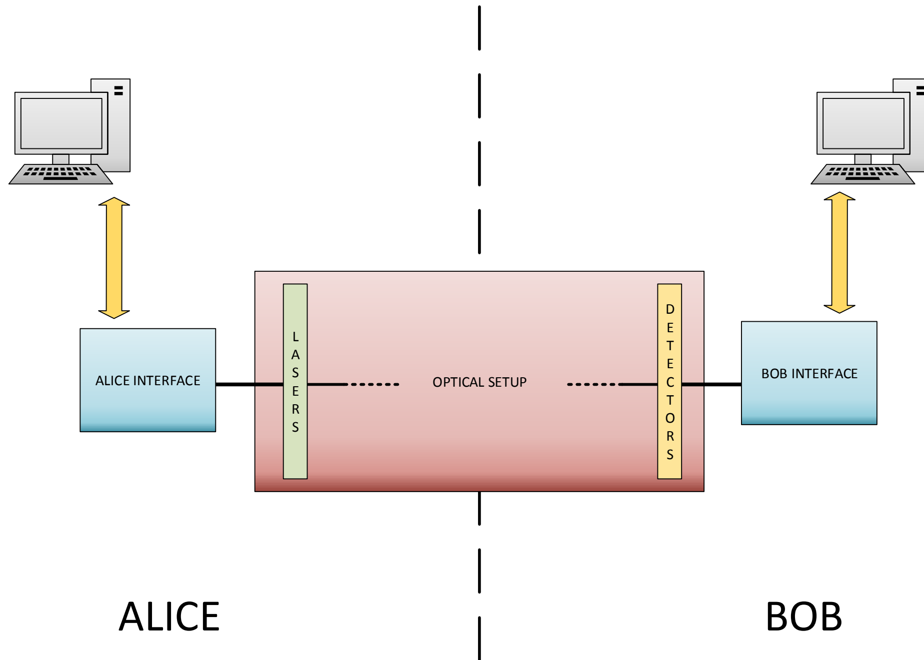


Figure 2.1: QuAKE schematic view.

2.1 Brief QuAKE description

The working principle of the experiment is that the transmitter Alice sends a sequence of qubits based on a key file, and the receiver Bob captures this sequence. Then, a part of the received sequence is compared with the original key for the QBER estimation[1].

Since two different clock domains are not locked one to the other, the system must be periodically resynchronized in order to avoid any transmission error. For this purpose the transmission is divided in groups of n-qubits called *frame*. Every frame starts with a synchronization signal, which is sent through a dedicated laser, and, changing the number of qubits per frame, the quality of the system performances can vary. Therefore, the maximum frame size needs to be chosen carefully with regard to the equipment. Furthermore, Bob implementation must be flexible to photons receivers delay because it can bring the synchronization to a fail condition. The whole system is controlled from a computer which is responsible for the set of the parameters, like the lasers frequency or the frame size, and also manages the start of the transmission as well as the key storing. A schematic view of QuAKE is visible in Figure 2.1.

[1]Please note that, for the testing of the system, the original key file was stored both in Alice and Bob. As a matter of facts, the final user is responsible for the functionality of the key verification through the classical channel.

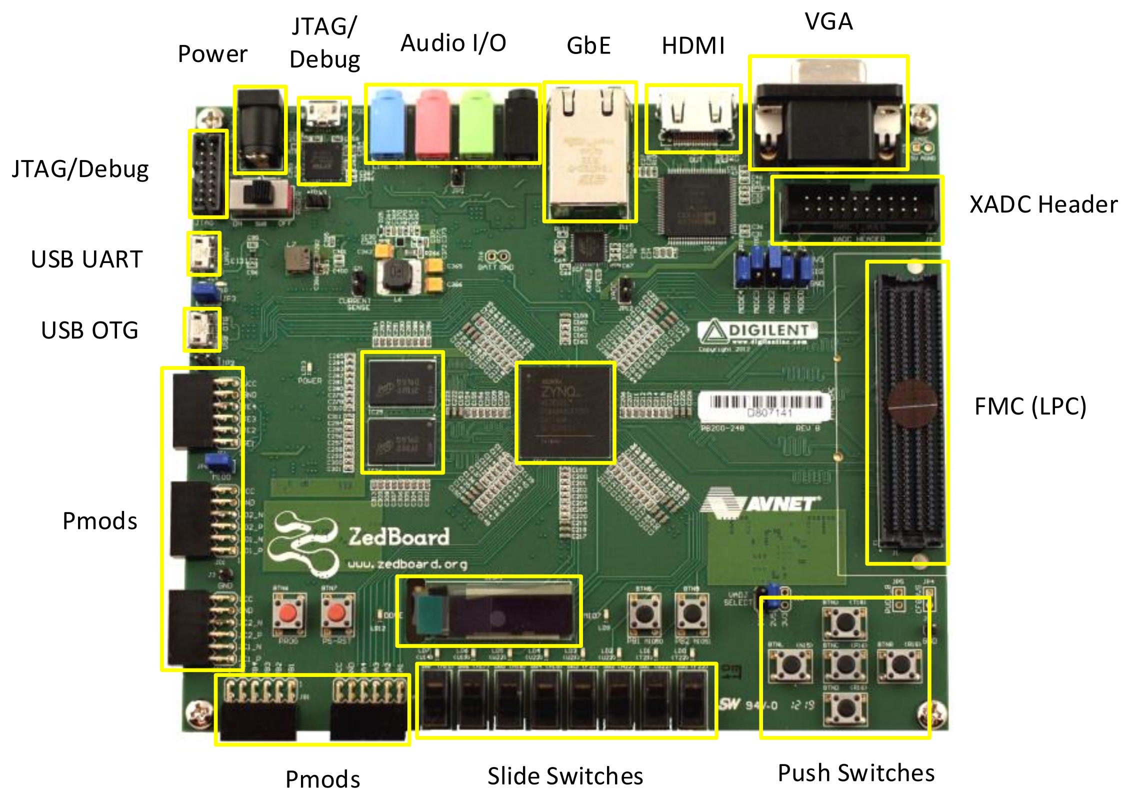


Figure 2.2: Main view of the ZedBoard [9].

2.2 ZedBoard device

For the realization of the transmitter/receiver interface, a programmable board with a Field Programmable Gate Array (FPGA) chip[2] was chosen in order to create all the custom functions required for the operation of QuAKE.
For the new system, one of the boards offered by Luxor Laboratory was the ZedBoard (Figure 2.2), designed and sold by Avnet company. This board offers more design possibilities than others (like the Virtex family) and allowed an easier approach to the project. One of the most important feature of the ZedBoard was the possibility to run an on-board software for the management of the board it-self from a computer.

[2]FPGA is a widely spread technology and is used for many different applications. From a practical point of view, it can be considered as a physical counterpart of VHDL code. For a complete explanation about FPGA see [10].

ZedBoard specifics are listed below [9] and illustrated in Figure 2.3:

- Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC
  - Primary configuration = QSPI Flash
  - Auxiliary configuration options
    - Cascaded JTAG
    - SD Card
- Memory
  - 512 MB DDR3 (128M x 32)
  - 256 Mb QSPI Flash
- Interfaces
  - USB-JTAG Programming using Digilent SMT1-equivalent circuit
    - Accesses PL JTAG
    - PS JTAG pins connected through PS Pmod
  - 10/100/1G Ethernet
  - USB OTG 2.0
  - SD Card
  - USB 2.0 FS USB-UART bridge
  - Five Digilent Pmod™ compatible headers (2x6) (1 PS, 4 PL)
  - One LPC FMC
  - One AMS Header
  - Two Reset Buttons (1 PS, 1 PL)
  - Seven Push Buttons (2 PS, 5 PL)
  - Eight dip/slide switches (PL)
  - Nine User LEDs (1 PS, 8 PL)
  - DONE LED (PL)
- On-board Oscillators
  - 33.333 MHz (PS)
  - 100 MHz (PL)
- Display/Audio
  - HDMI Output

- VGA (12-bit Color)
- 128x32 OLED Display
- Audio Line-in, Line-out, headphone, microphone

- Power
  - On/Off Switch
  - 12V @ 5A AC/DC regulator

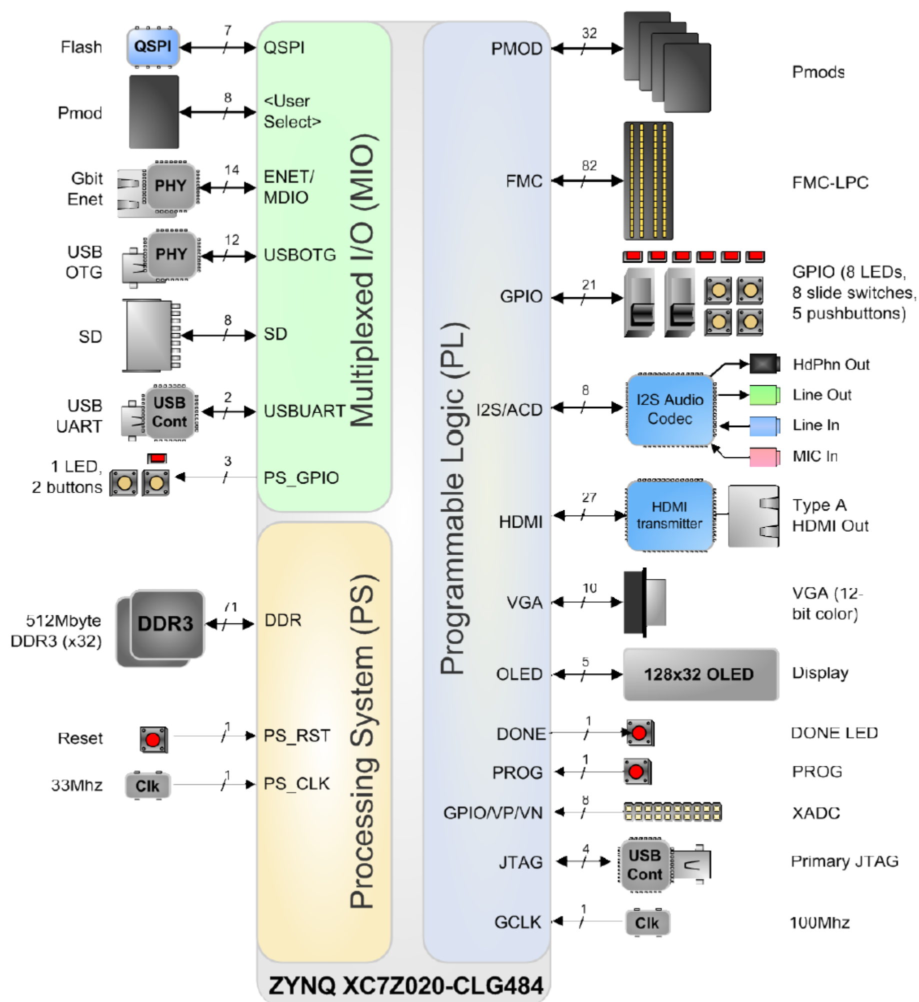


Figure 2.3: ZedBoard Block Diagram [9].

2.3 Software tools

Several software platforms were required for the system development and that is the reason why it was important to study and learn to use each one of them. The software are listed below:

- The hardware part of the project was designed with *Vivado Design Suite* from Xilinx company: this software includes basic tools like `VHDL`-synthesizer, implementation tool and test-benching functions. Furthermore, it allows the creation of custom ip-cores, i.e. a `VHDL` code implemented and saved as a reusable block in other project. But the insertion of the ip-core into a project does not require the classical `VHDL` code for hierarchy description and instantiation. Instead, a useful graphic designer tool allows to put together many ip-cores and link them in a graphical way. This could sound not so crucial but actually, in a project with many modules, writing code for every hierarchy-instance can become really hard and make the design more difficult. Moreover, Vivado includes *Hardware Manager* debug tool, which is essential for testing the true behaviour of the hardware, since it allows to analyze any real signal of the project during the operation of the board.

- The on-board software was designed with another Xilinx tool: *Vivado Software Development Kit* or simply *SDK* (Eclipse environment). The software allows a `C` and `C++` code programming along with libraries management and board memory sections assignment. Besides, it is used for the FPGA programming, the boot-image creation and flash memory programming. Therefore, it is almost essential for a complete exploitation of the ZedBoard. *C++* language was used instead of *C* language because it is more flexible and easier to use.

- The user interface software was designed with the *Qt* software usign `C++` language.

Chapter 3

Hardware Design

3.1 Basic System Overview

The whole hardware system is conceived in order to assure a perfect communication between Alice and Bob. Before Alice and Bob can work together, they have to be completely functioning on their own: Alice reads the data and turns on the lasers, Bob reads the lasers and writes the data with no errors. Note that the hardware design is mainly thought from a `VHDL` point of view and not from the cryptographic-transmission one. The cryptographic transmission is the final usage to which the hardware is intended. Therefore, the hardware is designed in order to do the job properly, according to the specifications, and can also work with a classical-coaxial-cable-channel instead of a quantum-channel. One of the great complexity is that two independent systems are linked together with no correlation between their time domains (i.e. their clocks). Hence, the first step is to create a efficient hardware with a modular structure: a module for synchronization, one for lasers control and one for memory management. This allows a clear view of the system and simplifies its design and its testing.

3.2 Usable technical knowledge

The starting point of the project was to understand which technology was already available and could be included in the system reducing the design effort. Therefore, three main items were considered in order to build up the QuAKE hardware with strong basis: the `Zynq`-7000 AP SoC Zedboard processor[1], the `AXI` protocol and the Finite State Machine technology.

[1]Please note that the therm *processor* is used broadly speaking and not to refer to classical CPU micro-processor. To be more precise, there is a kind of misunderstanding in the usage of the `Zynq` therm. Actually, it includes both microprocessor part and FPGA programmable part but is widely used for indicating only the microprocessor. When not specified, the simple therm `Zynq` only stands for the micro-processor side.

3.2.1 Zynq All Programmable SoC

`Zynq` All Programmable SoC allows a simple way to link together the `VHDL`-hardware with the `C++`/`C`-software. Its structure has two main parts: the Processing System (PS) and the Programmable Logic (PL). In Figure 3.1 is visible a Vivado-view of the `Zynq`. Actually the `Zynq` can be added to a Vivado design project as any Xilinx IP Core and all its parameters and options can be set as needed. The PS mainly includes two ARM Cortex-A9 CPU, memory interfaces, cache and minor memories, a clock generation block and a connection to I/O peripherals. Besides, the PL mainly includes `AXI` ports, an analog-to-digital converter, the clock output ports and, of course, the FPGA chip. The `Zynq` also allows to lighten the custom-hardware design, since all the board external communication devices can be initialized directly with the software and do not require a hardware description. Hence, the Ethernet port as well as the serial USB port, can be easily used when writing specific code lines in the software. Furthermore, it allows the usage of the `AXI` protocol, which is widely used in the design for parameters setting and data management.

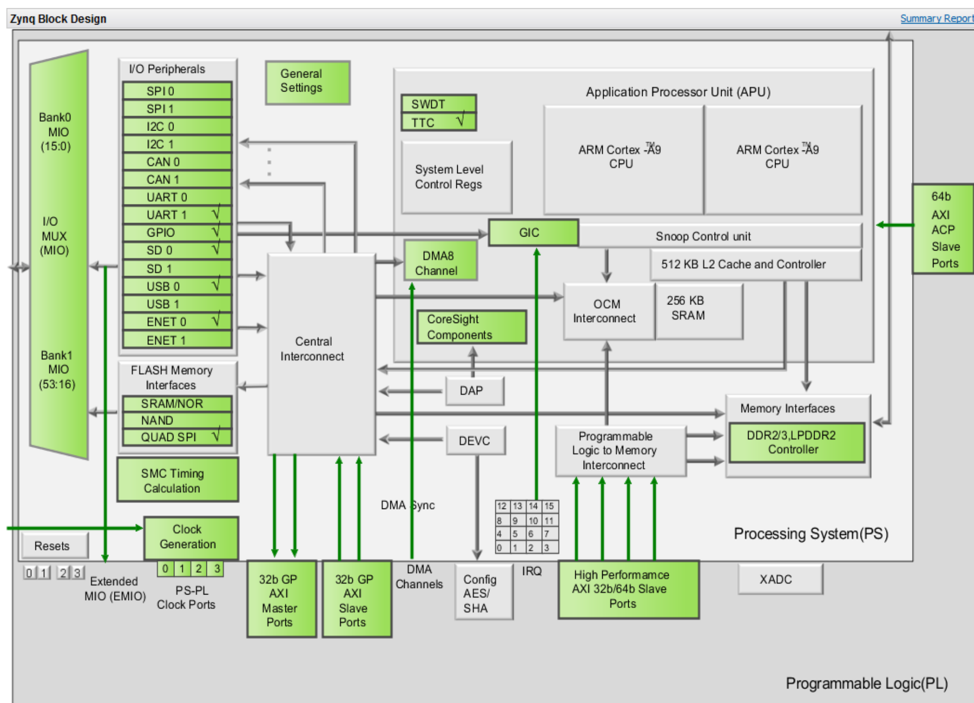


Figure 3.1: Main view of the `Zynq` PS Vivado IP Core.

3.2.2 AXI Protocol

The Advanced eXtensible Interface protocol is a part of ARM AMBA, which is a standard for the connection, the management and the communication of blocks in SoC designs. The main aim is to make easier the handling and the development of multi-processor systems, of its controllers and peripherals. This protocol is very robust and guarantees a safe and fast communication between different modules which are linked together with a master-slave interface system [for a complete description of the protocol refer to [11]]. From a practical point of view, `AXI` protocol is used by most of the Xilinx IP Cores and links together the SDK software with the Vivado hardware: every `AXI`-based module has its own driver, which can be initialized in the software code calling specific functions. Therefore, this protocol is fundamental for the setting of the hardware parameters, the internal management of the data and for the complete control of the hardware behaviour. For example, an hardware parameter can be set using an `AXI GPIO`, which is a module with two outputs with settable length (up to 32 bits) that can be hard-wired to any `VHDL` port. Then, the `AXI GPIO` is controlled with an axi-slave-port, linked to an axi-master-port of the `Zynq`[2], and its specific functions allow to write a custom value on the chosen output, that is the communication of a number to the hardware.

3.2.3 Finite State Machine

A finite state machine, also indicated as `FSM`, is an important element in the design of sequential logical circuits. Basically, it models the behaviour of the circuit abstracting from its *state* of operation. Therefore, the circuit works moving from a state to another and performing a specific task depending on the state. This element is really helpful in designing of complex systems, because it makes the hardware more aware of what it is doing. Furthermore, it organizes the `VHDL` code in a more intuitive way making the code more readable and easier to understand. For this reason, almost every custom `VHDL` module is designed with a dedicated finite state machine.

Mealy and Moore state machine

There are two different types of state machine: Mealy state machine and Moore state machine [12]. In the Mealy machine the output depends on the state and the input, whereas the Moore one depends only on the state and not on the input. In general, the Mealy state machine requires less state (then less hardware) and can work faster than the Moore one. But its outputs can change asynchronously, undermining the predictability of working, and its design could be complicated. Since the system requires a great accuracy in the execution of its functions and since there are no restrictions

[2] Actually, an `AXI Interconnect` module is used to put together many axi-slave-ports and manage them with just one axi-master-port of the `Zynq`. Therefore, the `AXI GPIO` slave-port is not directly connected to the `Zynq`.

on the hardware usage and on the reactivity of the system, only Moore state machines were used.

Coding styles

An `FSM` can be implemented in VHDL code using different styles [13]. The basic two-processes-single-decoder style creates two state-type signals (the state type defines the list of all the possible states): one for the present state and one for the next state. Hence, the first synchronous process simply assigns the next state to the present state, while the second process describes the behaviour of every state (that is the decoder) and assigns the future state to the next state signal. A variant of this style is the two-processes-two-decoders style, which uses two processes: the first one decodes the state transitions and the second one decodes the state to generate the output. In this variant there is only one state-type signal, since the next state signal is not required. The third style is the one-process-one-decoder characterized by one process only, in which both state transitions and outputs assignment are decoded. This style is maybe the most difficult to design because requires to think one clock cycle ahead (the outputs are registered) but deletes any glitch-error possibilities. This last style was chosen for the hardware design.

State encoding

Another important feature of an `FSM` is how the states are encoded. As a matter of facts, the names assigned to states are fictional and the synthesizer will not waste hardware using lots of bits for every name. Therefore, an encoding operation is required in order to assign a more affordable bits sequence to every name. There are different encoding styles like gray, johnson, one-hot and others. Usually this operation is completely automatic and the synthesizer will choose the best one depending on the state machine. However, Vivado synthesizer allows the user to choose between the different encoding styles [for a complete description of all the encoding styles offered by Vivado, refer to [14]]. Anyway, since there are many different state machines on the project[3], the encoding setting was left as automatic. Consulting the after-synthesis log file, it was checked that the chosen encoding style was one-hot for the custom block state machines[4], whereas the chosen one for other state machines was different depending on the module.

[3] Please note that even the `AXI` protocol uses state machines.

[4] Generally the one-hot encoding style is the most recommended for FPGA design [15].

3.3 Alice Hardware Implementation

The Alice hardware was the first thing to be designed. Its task is to read the memory, activate the qubit lasers and send a synchronization signal to Bob. The `Laser Controller Module`, the `Zeus Synchronization Module` and the `Mind Well Memory Manager` were designed in order to do the job. These three modules are the true core of Alice but they can not work without other minor blocks nor the `Zynq`. The following part describes all the custom blocks, their role in the project and their connection and relation with the other modules. A basic schematic of the main custom blocks is visible in Figure 3.2 while the completed Vivado design view is visible in Figure 3.3 and project post-implementation results are visible in Figure 3.4.

3.3.1 Laser Controller Module

As its name says, the **Laser Controller Module** is the block responsible for the lasers lighting. Basically, it reads bits from the `Mind Well` and turns on the equivalent laser. Despite this simple task, in order to make everything works, including the option to switch between BB84 and B92 protocol[5], an elaborate five states `FSM` was designed (Figure 3.5).
As usual in state-machine design, the initial state is an *Idle* state where the `FSM` moves to the next state and asserts *sync_please* only if *start_key* is asserted. This is the beginning of the key transmission. The next state is the *Sync* state, where the `FSM` waits for the *sync_start* from `Zeus` and then moves to one of the two cryptography-protocol state, depending on the *BB-selection* input. In the *BB-84* state, when the two input bits are read, one of the four lasers turns on and the *data_red* output asserts in order to communicate the happened reading to the Mind-Well[6]. But if *empty* input is asserted the `FSM` returns to the *Idle* state because the communication has ended. At this point, the `FSM` moves to the `Slot_Processing` state, where a simple clock-cycles-counter is implemented. After *duty_cycle* clock-cycles, the activated laser is turned off[7] and after *laser_frequency* clock-cycles the frame-counter increments and the `FSM` returns to the protocol state, asserting *sync_please* output. This is true as long as the last qubit of the last frame has been processed. When this happens, the `FSM` returns to the *Sync* state and starts all over with a new frame. At the end of the key transmission, the `FSM` returns to the *Idle* state thanks to *empty* signal assertion.

[5]An easiest way of design would be to delete the protocol-switch and just waste one of the two bits coming from the Mind Well (B-92 just reads the first bit). But, in order to have a great-performance system, this choice was discarded.

[6]The *B-92* state is pretty similar but initially only the first bit is read and *data_red* is not asserted. When the `FSM` returns on *B-92* for an even-time, the second bit is read and *data_red* is asserted. Basically, this system halves the *data_red* frequency.

[7]To allow the outstanding 100% duty-cycle condition, all the lasers are shut down even in the *Protocol* and *Sync* state, since in the *Slot_Processing* state the shut down declaration is unreachable.

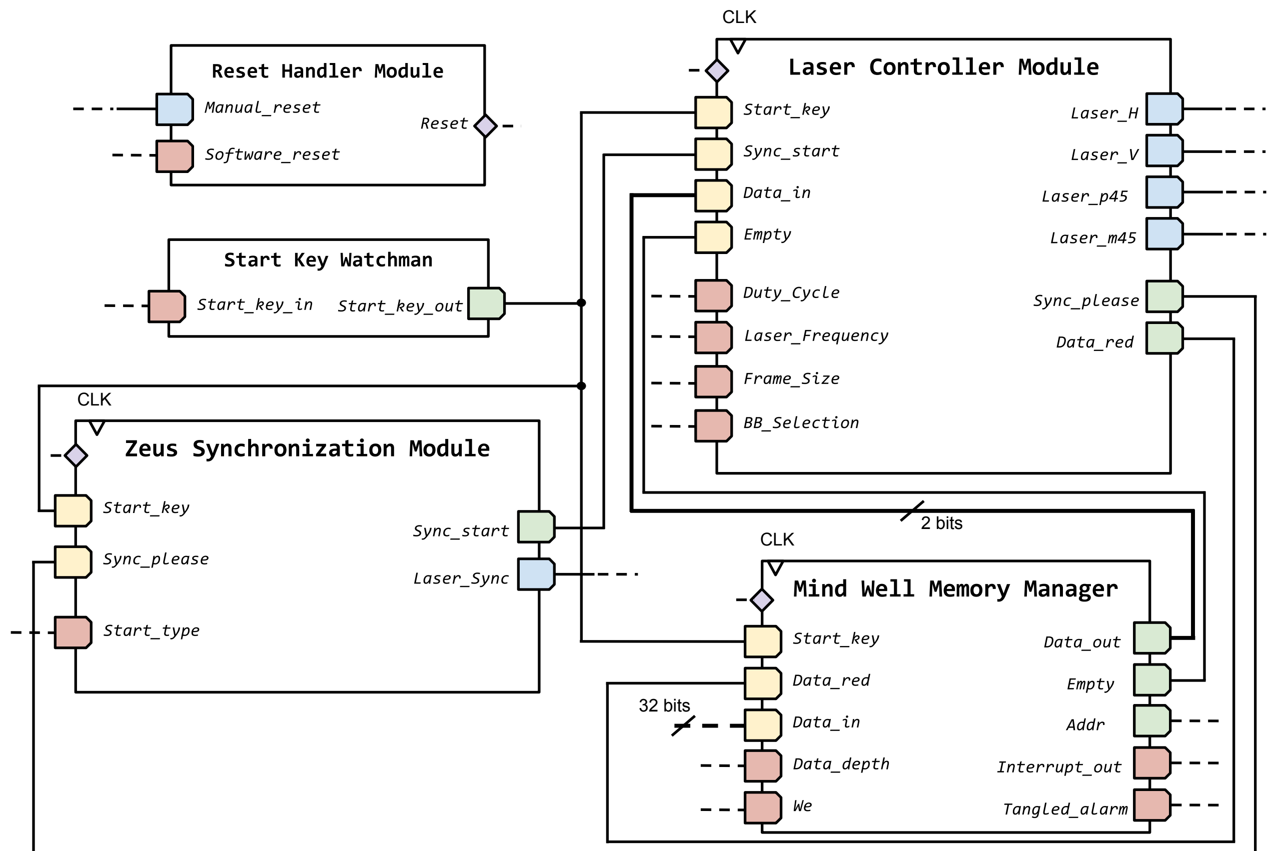


Figure 3.2: A schematic view of some of the custom blocks of Alice. For a clearer view the `System Poiting` and the `Output Laser Multiplexer` are left out. Legend: port with signals between modules are in yellow (input port) and green (output port); ports with signals coming from or linked to the Zynq are in red; ports linked (input or output) to the outside world are in blue.

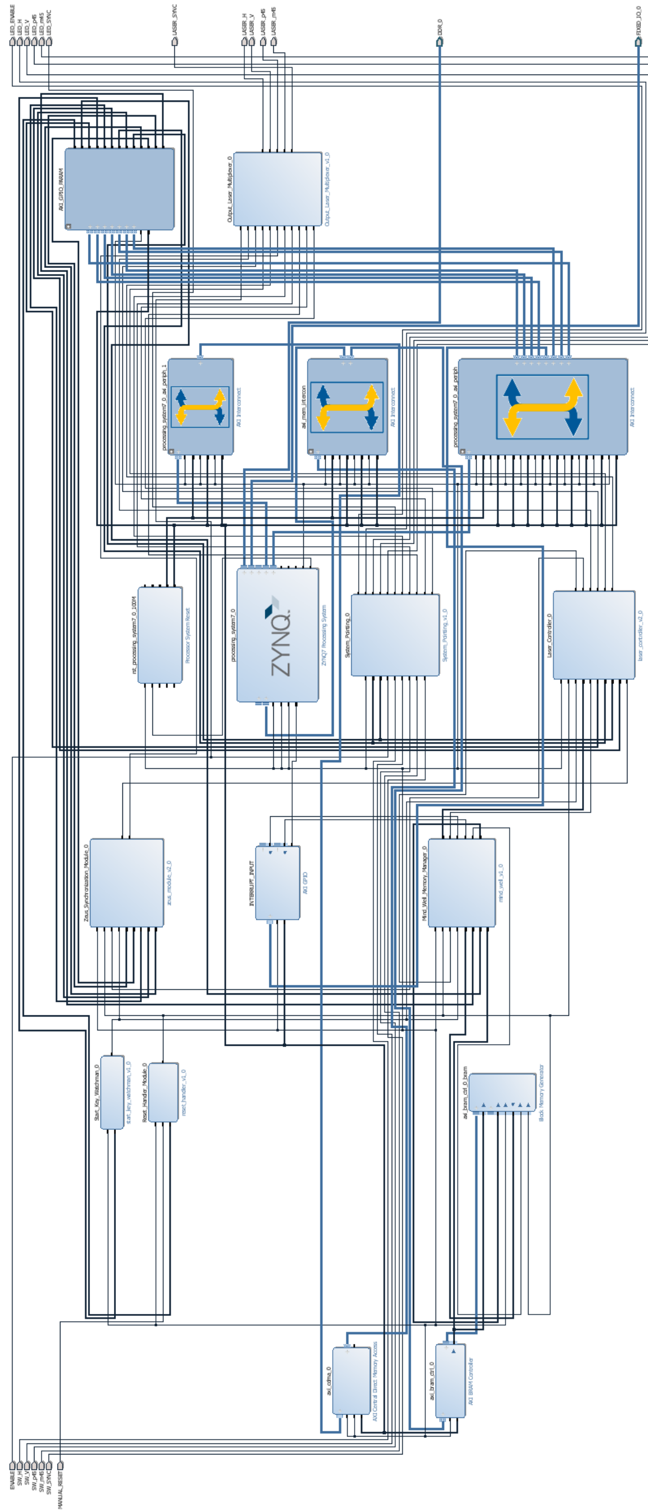
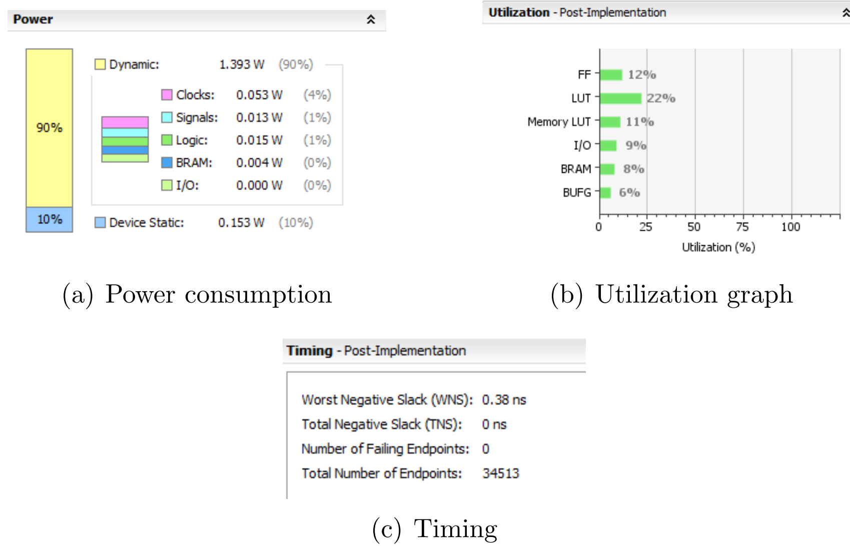


Figure 3.3: Vivado view of the Alice Block Design.



(a) Power consumption

(b) Utilization graph

(c) Timing

Figure 3.4: Alice Vivado post-implementation results which include power consumption (a), utilization graph (b) and timing verification (c).

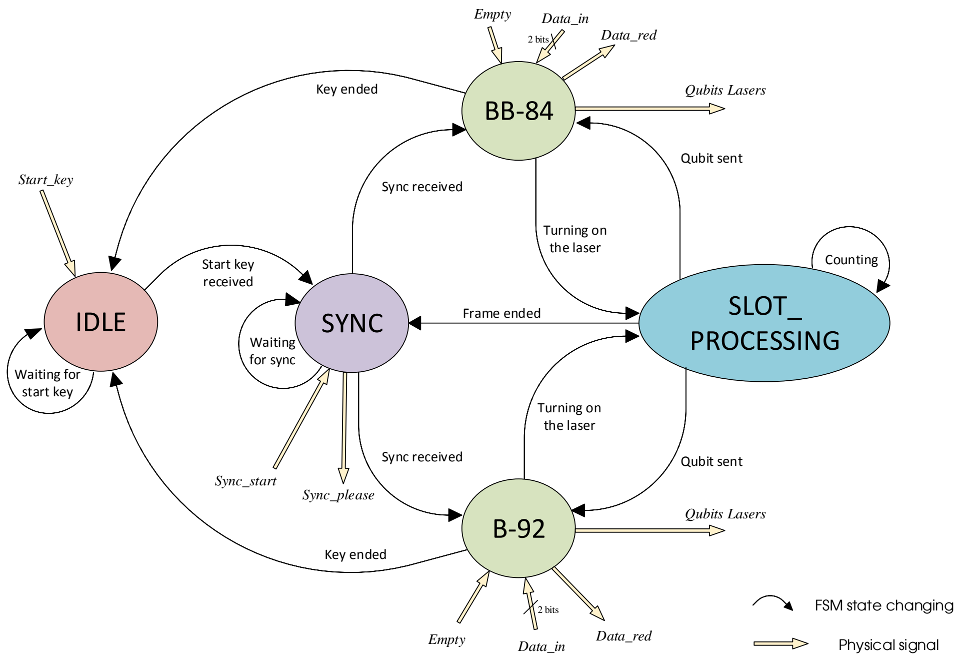


Figure 3.5: Laser Controller Module FSM. For a clearer view, all the parameters signals were removed from the figure.

3.3.2 Zeus Synchronization Module

Keeping Alice and Bob in synchronization is essential in order to prevent any qubit reading or storage error due to the clock drift [see Appendix A for more details]. For this reason, a synchronization management module was designed and named **Zeus Synchronization Module**[8]. It provides the timing of both Alice and Bob through the synchronization laser. The key transmission is divided in frame and, at the begin of every frame, Alice send a synchronization signal in order to timing the qubit acquisition of the incoming frame. The designed `FSM` of `Zeus` is visible in Figure 3.6.

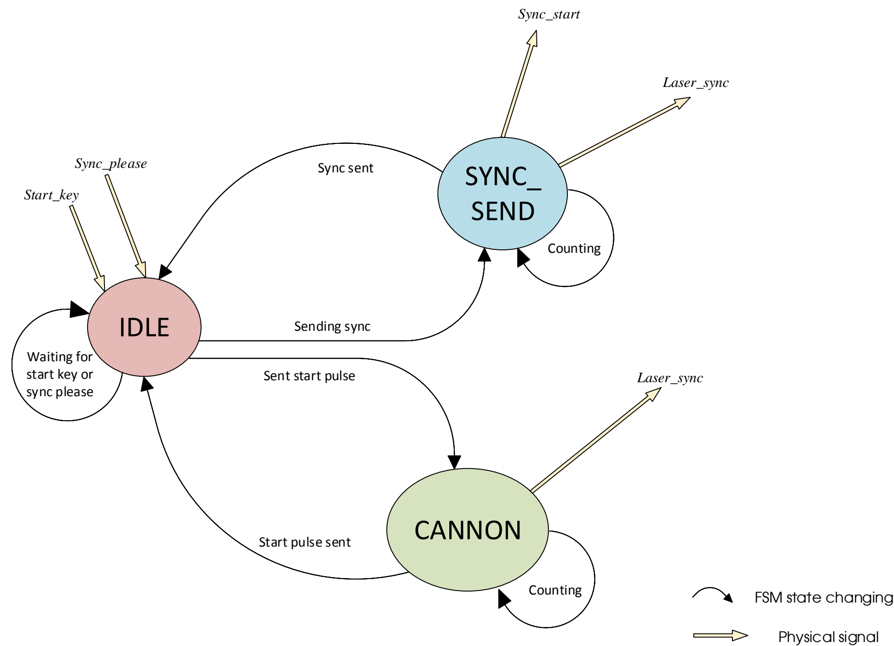


Figure 3.6: Zeus Synchronization Module FSM. For a clearer view, all the parameters signals were removed from the figure.

Zeus receives the *start_key* and the *sync_please* signals, as inputs, and its internal `FSM` remains in the *Idle* state until one of these two signals asserts[9]. The next state depends on the *start_type* input: in the *start-with-laser* mode, the `FSM` moves to the *Cannon* state when *start_key* asserts and launches three impulses[10] with the synchronization laser. Then it returns to the *Idle* state waiting for a *sync_please* signal. When a *sync_start* is required, the `FSM` moves to the *Sync_Send* state and asserts[11]

[8] According to ancient Greek mythology, Zeus defeated Cronus, ancient Greek term for *time*.

[9] Please note that the `FSM` will ignore the *sync_please* signal if no *start_key* signal has arrived.

[10] The impulse frequency and the duty-cycle are both settable in `C++` application.

[11] The lasting time of the syncs-signals is settable in `C++` application.

the *laser_sync* output, after user-chosen clock-cycle. It also asserts *sync_start* but only after a hardware-fixed-time-delay of 500 ns. This delay simplifies the handling of the *single photon detectors* behaviour on Bob-side, since they can have different speeds and reliability depending on the models. Please note that, from here on, *single photon detector* will be abbreviate with SPD.

The *start-with-external-channel* eliminates the need for the laser sync for the *start_key* signal, since it is managed through a wifi or Ethernet connection. The operating principle is quite the same as described above, except for the *start_key* signal. A start signal is sent to Bob through Wi-Fi or Ethernet connection and when Bob receives it, it sends the start back to Alice, which then asserts the *start_key* signal. For more details about Bob synchronization management see 3.4.3.

A future development will offer a synchronization through a GPS signal in order to improve the timing performance, since the synchronization laser may not so reliable, especially on long distances. Furthermore, with a good set up of the Zedboards clock multiplier[12], a generated-from-GPS clock could be used and the clocks-drift issue could be completely solved.

3.3.3 Mind Well Memory Manager

The task of the **Mind Well Memory Manager** is to read the bits, which are used to code the lasers/qubits, from the Block Memory Generator (or `BRAM`). The `FSM` (Figure 3.7) moves from the *Idle* to the *Data_Reading* state when the *start_key* signal asserts. Therefore, it reads the first two bits stored in the first address of the `BRAM` and moves to the next two bits only when *data_red* asserts [for more details about *data_red* behaviour see 3.3.1]. It always handles the address and when 32 bits are read, it increases the *addr* output by four[13]. According to the *bram_depth* and *data_width* hardware parameters and to the *data_depth* software parameter, the `Mind Well` keeps reading the bits changing the address properly, when needed, and returns to the *Idle* state when the inner counter is equal to *data_depth*; besides, the *empty* output signal asserts. The *bram_depth* is also used to send an *interrupt* signal to the `Zynq` every time the `Mind Well` reaches the half and the end of the `BRAM` memory: the `Zynq` then launches an interrupt routine which activates the `XAxiCdma_SimpleTransfer` `C` function in order to move the next part of cryptographic key from the `DDR-RAM` memory to the already-read half of the `BRAM` Memory. A schematic view of the blocks memory connections is visible in Figure 3.8.

A specific control process was designed in order to stop the whole transmission in case of reading/writing superposition: the `Mind Well` checks if the *write enable* signal of the `Axi Bram Controller` is asserted at the half and at the end of the `BRAM`. If so,

[12]The Zedboard allows the usage of a Phase-Locked-Loop or a Digital Clock Multiplier. Both IP Cores, with different performances and features, can multiply an input clock signal to create a faster one.

[13]Note that in one address, apart from the data-width, one byte is always stored.

the `Axi Bram Controller` is still writing to the half of the `BRAM` that is going to be read by the `Mind Well`. This means that the `Axi Bram Controller` writing process is too slow or has accumulated too much delay. At this point, the `Mind Well` stops, returning to the *Idle* state and asserting the *tangled_alarm* signal. The `Mind Well` can restart only after a system-reset that will deassert *tangled_alarm* (i.e. any *start_key* is ignored if *tangled_alarm* is asserted).

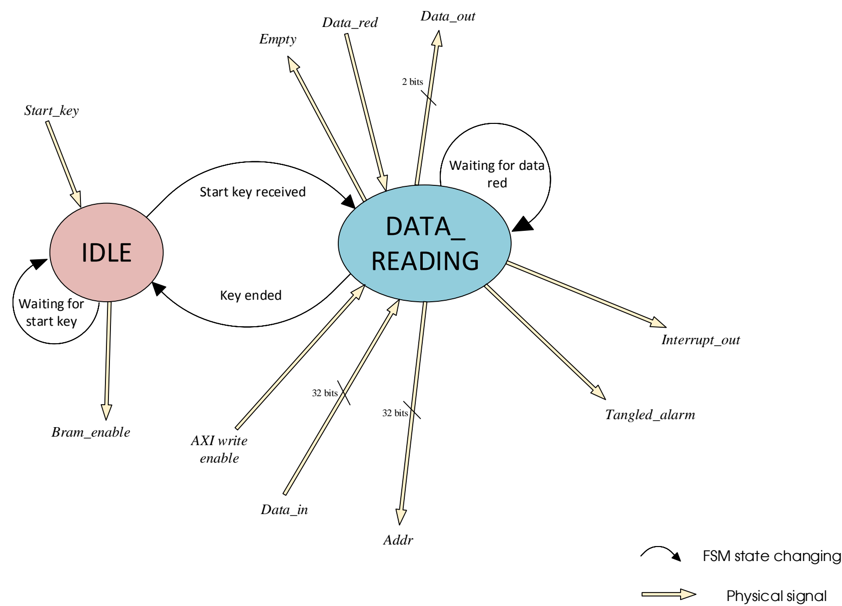


Figure 3.7: Mind Well Memory Manager FSM. For a clearer view, all the parameters signals were removed from the figure.

3.3.4 Other Modules

Start Key Watchman

The key's transmission beginning occurs when the *start_key* signal asserts. This signal is activated from `C++` software through an `AXI GPIO` module but, since it is a software-timing-dependent signal, the assertion time could be too long and the hardware could interpret multiple start signals when just one *start_key* assertion has occurred.
Therefore, the **Start Key Watchman** simply translates a long software-generated start signal into a short[14] and hardware-suitable signal.

[14] A couple of clock-cycles.

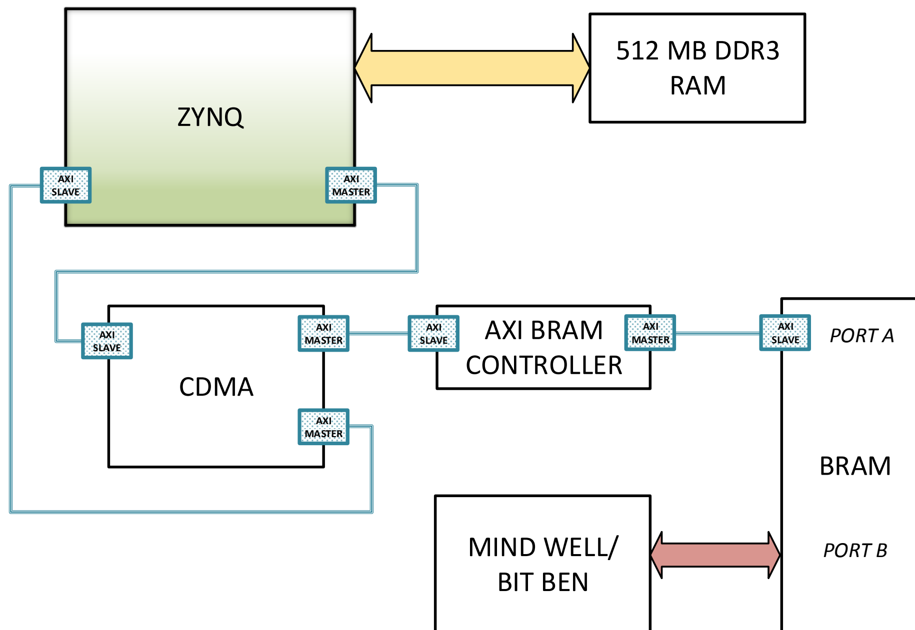


Figure 3.8: Memory scheme. This schematic recaps the linking of the modules responsible for data management. Please note that this configuration is the same for both Alice and Bob.

Reset Handler Module

The whole system has two asynchronous reset signals and the **Reset Handler Module** basically makes a boolean `OR` operation between the *software-axi-gpio* reset and the hardware push button. Asynchronous reset is a classic vhdl-code-style solution but things are not so simple as one would expect. There is a possibility that the designed hardware fails to work properly, since, aside from an asynchronous assertion, a completely asynchronous reset also entails an asynchronous deassertion which could undermine flip-flops reliability [see Appendix C]. Thus, the `Reset Handler Module` provides an asynchronous-assertion-synchronous-deassertion reset output signal. Furthermore, this feature avoids the long-time-signal-software-assertion-issue as in the `Start Key Watchman` and also implements a debounce solution for the push button.

System Pointing

For the setup of the entire optical system, the laser beams have to be perfectly centered to their relative SPD. In order to do this, the lasers have to be turned on and must work continuously with the frequency and duty cycle set by the user[15]. Using the ZedBoard user switch buttons[16], this block activates or deactivates the corresponding laser. The *enable* user switch must be turned on, otherwise the block will keep everything turned off.

Output Laser Multiplexer

The outputs of the lasers have to be linked either to the `Laser Controller Module` outputs or to `System Pointing` outputs. The **Output Laser Multiplexer** works as a simple multiplexer, connecting the outputs to the `Laser Controller Module` outputs, when the *enable* user switch button[17] is turned off, and to the `System Pointing` outputs, when *enable* is turned on.

3.3.5 Xilinx-Vivado IP Cores

As already stated, the project uses some Xilinx IP Cores in addition to the custom-modules. A complete list with description is visible in Table 3.1. Please note that the IP Cores used on Bob are almost the same. Thus, the table is valid for Bob too.

[15] Note that even the synchronization laser beam must be calibrated but since this laser does not send qubits, there are no meaningful duty cycle or frequency parameters and the related output will simply be turned on or off.

[16] One switch button for each lasers plus the *enable* one. These switch buttons are the ones called *Slide Switches* in Figure 2.2

[17] It is the same switch button used in the `System Poiting`.

IP Core	**Name**	**Specifics**
`AXI GPIO`s	Laser param	CH 1: qubit laser duty-cycle CH 2: qubit laser frequency
	Main pulse param	CH 1: start laser duty-cycle CH 2: start laser frequency
	Sync param	CH 1: count before laser CH 2: sync width
	Data param	CH 1: frame size CH 2: encoded laser key length
	Mode param	CH 1: protocol selection CH 2: start type
	Wheel param	CH 1: start key CH 2: reset
	Interrupt (and sync) param	CH 1: interrupt time CH 2: delay before sync (Bob)
	Interrupt input	CH 1: interrupt out CH 2: - Optional interrupt output enable and hard-wired to the `Zynq` interrupt input
`AXI CDMA`	Axi Cdma	-
`AXI BRAM` Controller	Axi Bram Ctrl	-
`Block Memory Generator`	Bram	True dual port BRAM with 8192 element of 32 bits
`Zynq 7 Processing System`	Processing System 7	Zedboard preset plus extra axi-master port, hp axi-slave port and interrupt input

Table 3.1: Xilinx IP Cores used in the design. Please note that this table lists either the cores used in Alice either the ones used in Bob. Minor differences are specified. The `AXI GPIO`s section describes the signals which the channels are linked to. Please note that the channels bit-length is set to 32 bits except for *protocol_selection*, *start_type*, *start_key*, *reset* and *interrupt_out* signals which are only one bit long. Furthermore the *interrupt_out* channel is the only one used as input, all the others are outputs.

3.4 Bob Hardware Implementation

The main task of Bob is to be perfectly synchronized with Alice in order to receive all the qubits sent and to store all the data. The `Laser Pinball Communication Sorter`, `Hermes Synchronization Module` and `Bit Ben Word Composer` are the equivalent of Alice's `Laser Controller`, `Zeus` and `Mind Well` and they work in a similar way. Bob uses the `Zynq` and other minor modules in a quite similar way as Alice does. Since there are very minor differences in the usage of the Xilinx IP Cores, there is no description of them in this section: please refer to Table 3.1 for details. Besides, the following part describes all the custom blocks, their role in the project and their connection and relation with the other modules. A basic schematic of the main custom blocks is visible in Figure 3.9, while the completed Vivado design view is visible in Figure 3.4; project post-implementation results are visible in Figure 3.11.

3.4.1 Reflex Photon Translator

This module is thought to translate the incoming SPDs signals (i.e. the received lasers) from the Alice time-domain (asynchronous) to the Bob time-domain (synchronous). The main purpose is to make these signals visible to Bob[18]. Through a vhdl-process, sensitive to both the clock and the SPDs signals, the outputs assert simultaneously with the SPDs signals, but they deassert on the second clock rising-edge, after the SPDs assertion. Hence, the output signals assertion-time is at least one clock-period long, making the output signals timing-suitable for the `Laser Pinball` reading process. In order to reduce the background noise due to the SPD dark counts, the *reflex_enable* input activates or deactivates the module sensitivity to the SPDs inputs. Only when the *reflex_enable* is asserted, an input reading and its output assertion can occur [for more details about *reflex_enable* behaviour see 3.4.2].

3.4.2 Laser Pinball Communication Sorter

As the name suggests, the **Laser Pinball Communication Sorter** plays an important role in the whole Bob hardware: it reads the qubits and sends them to the `Bit Ben`, according to a perfect timing, in order to manage the *slots* and the *frames*[19] which can be set through the *duty_cycle* and *laser_frequency* `C++` settable parameters.

The module starts when the *start_key* signal asserts and its `FSM` (Figure 3.12) moves from the *Idle* state to the *Sync* state, waiting for the *sync_start* to assert. Then, it moves to the *Slot_Reading* state, asserting the *reflex_enable* output. As described in the `reflex` module, the assertion of *reflex_enable* allows the reading of the qubits, but its activation-time is designed to minimize the noise and not to maximize the qubits

[18] In the worst case scenario, due to different factors, the SPD assertion-period could be shorter than Bob clock-period.

[19] Actually this is the only module aware of the "slot" and the "frame" concept.

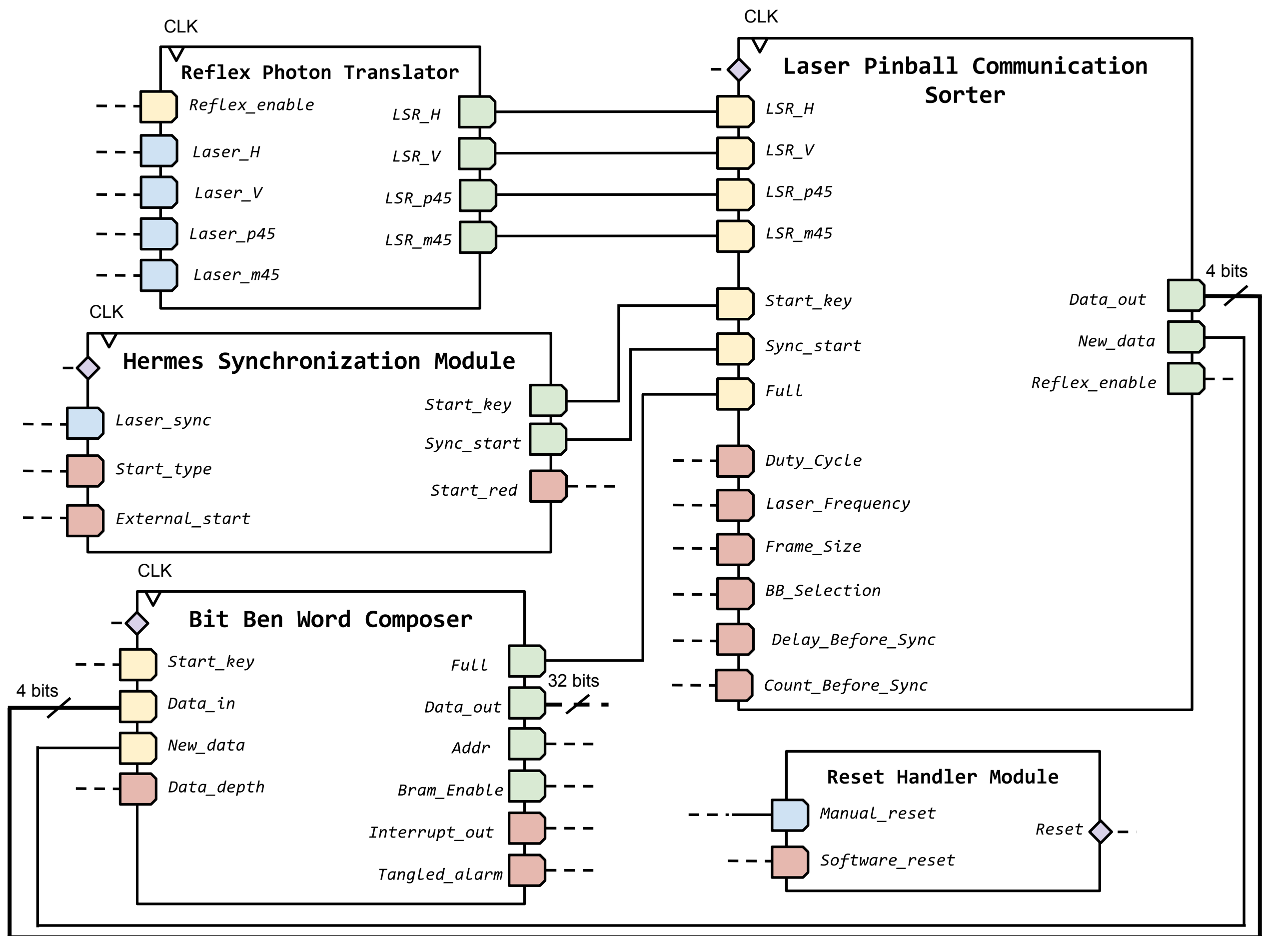


Figure 3.9: Schematic view of some of the custom blocks of Bob. Legend: port with signals between modules are in yellow (input port) and green (output port); ports with signals coming from or linked to the `Zynq` are in red; ports linked (input or output) to the outside world are in blue

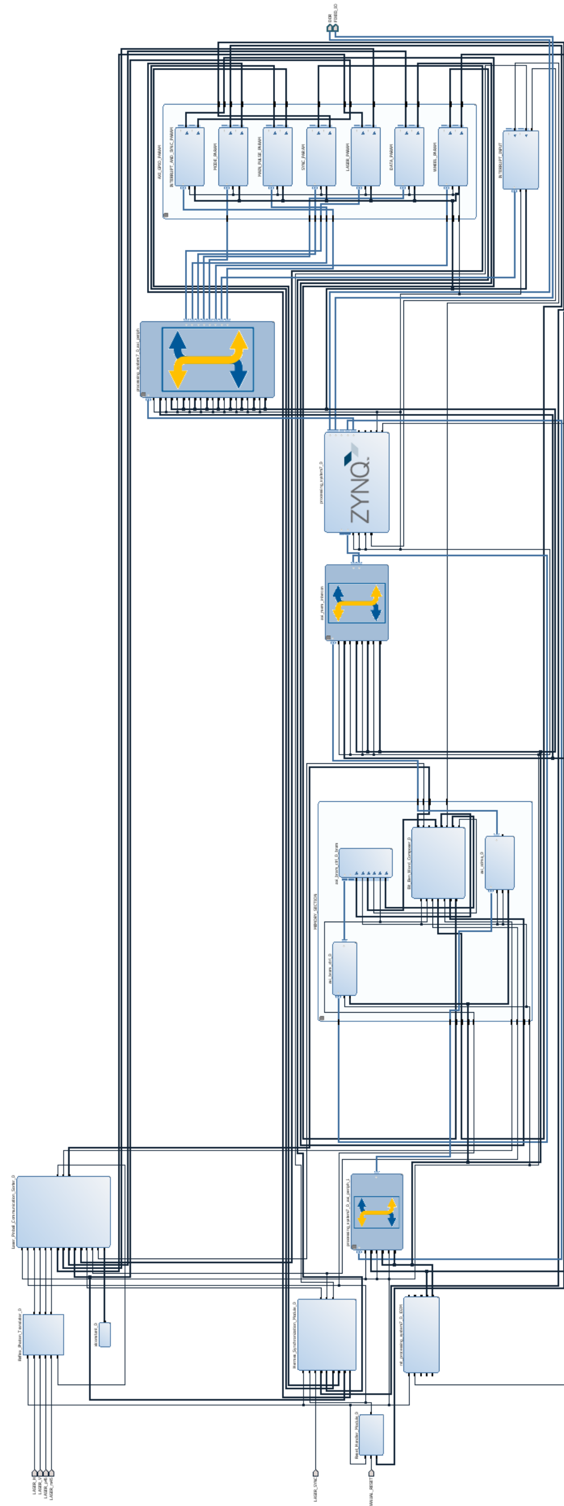
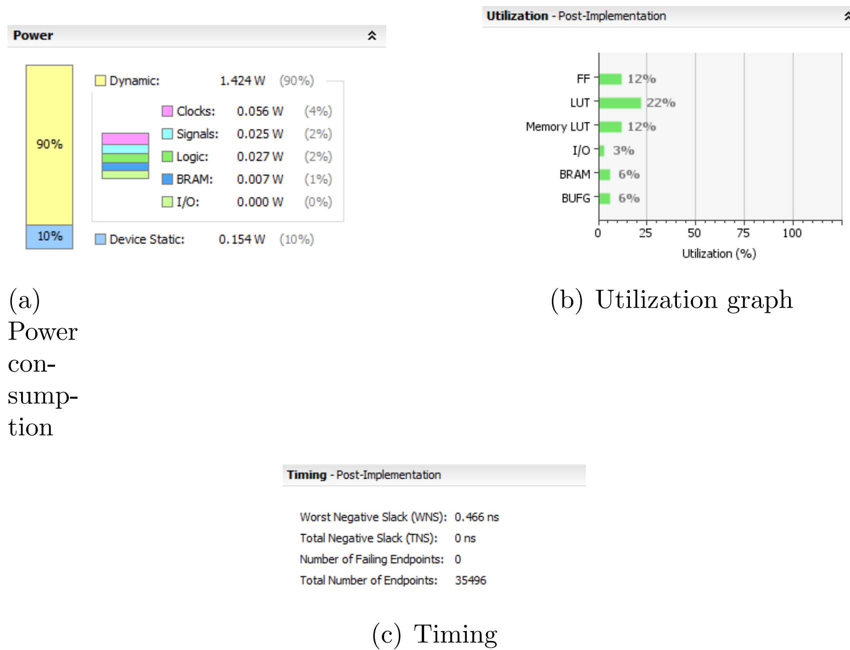


Figure 3.10: A Vivado view of the Bob Block Design



(a) Power consumption

(b) Utilization graph

(c) Timing

Figure 3.11: Bob Vivado post-implementation results which include power consumption (a), utilization graph (b) and timing verification (c).

reception. That is the reason why its assertion-period is always set to the number of clock-cycles equal to the laser's duty-cycle minus one clock-cycle. This kind of system allows the reading of the qubits only for the clock-cycles in which the likelihood of receiving a qubit covers the entire clock-period. For a clear understanding see Figure 3.13. Furthermore, there is no need for special counting system for the extreme laser condition of 100% duty-cycle, because the reading-time would be equal to a duty-cycle of 90%. This implies that, even in the worst case scenario, there is at least one free-clock-cycle (that is the slot-last-one) which is used to update the *data_out* output without any data loss or superposition with the next incoming qubit/slot. Besides, it is also used to assert the *new_data* output which is sent to the `Bit Ben` in order to timing the data transfer between the modules [see 3.4.4 for more details]. Otherwise, the *data_out* managing and timing would have been much more complex.

The *data_out* is designed as a four-bits (one for each laser) input-output port in order to store every qubits reception within a single slot. If any SPD clicks inside the *reflex_enable* assertion time, the corresponding data-out-bit will assert and will maintain its value[20] till the end of the slot. This means that if, within a slot, more than one SPD clicks, a multiple "1" vector will be stored in *data_out*. Thanks to that, a precise SPDs click-history is stored in the memory. Note that the number of bits

[20]This assignment is done as follow: `DATA_OUT<=DATA_OUT or LSR_H or LSR_V or LSR_p45 or LSR_m45`.

required for Bob is twice as much as that of Alice, so for the BB84 protocol Bob uses four bits, while with the B92 protocol it uses just two bits. It is also important to stress that the `Laser Pinball` is not aware of the type of protocol used, because this distinction is managed by the `Bit Ben`. Hence, even with the B92 protocol, it reads from all the four inputs but the `Bit Ben` will ignored two output-bits which, obviously, cannot be asserted since the corresponding SPDs will be disconnected.

As in Alice's side, the slots and frames are processed: at the end of the frame the `FSM` returns to the *Sync* state and waits for the next *sync_start*, unless the *full* signal is asserted. In that case, the transmission has ended and the `FSM` returns to the *Idle* state.

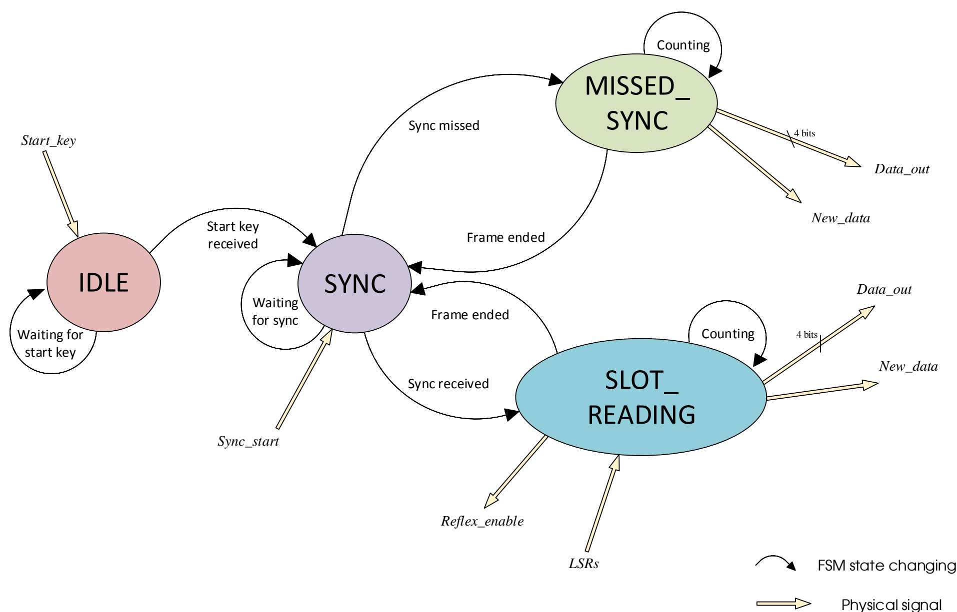


Figure 3.12: Laser Pinball Communication Sorter FSM. For a clearer view, all the parameters signals were removed from the figure.

If the synchronization laser fails to arrive to Bob (which is not a remote possibility), the `Laser Pinball` will not store any qubit of the entire frame, because it will be still waiting for the *sync_start*. This wrong storage corrupts the whole transmission, since the data are shifted by the number of missed bits. Consequently, the received key cannot be compared to the original one and there is no way to recover it. To avoid this circumstance, an emergency system was designed. In the *Sync* state, through an internal counter, it is checked that the *sync_start* is asserted by the time the *delay_before_sync*-time is reached. If not, the *sync_start* assertion can occur up to an

extra time of 50 ns. After this extra time, if no *sync_start* assertion has occurred, the FSM moves to the *Missed_Sync* activating the emergency system. In this state all the data-out-bits are forced to one for the entire frame period[21] and then the FSM returns to the *Sync* state. This means that for every slot of the frame a "1111" vector is stored . This information will be interpreted as a wrong qubits transmission but will not corrupt the entire key.

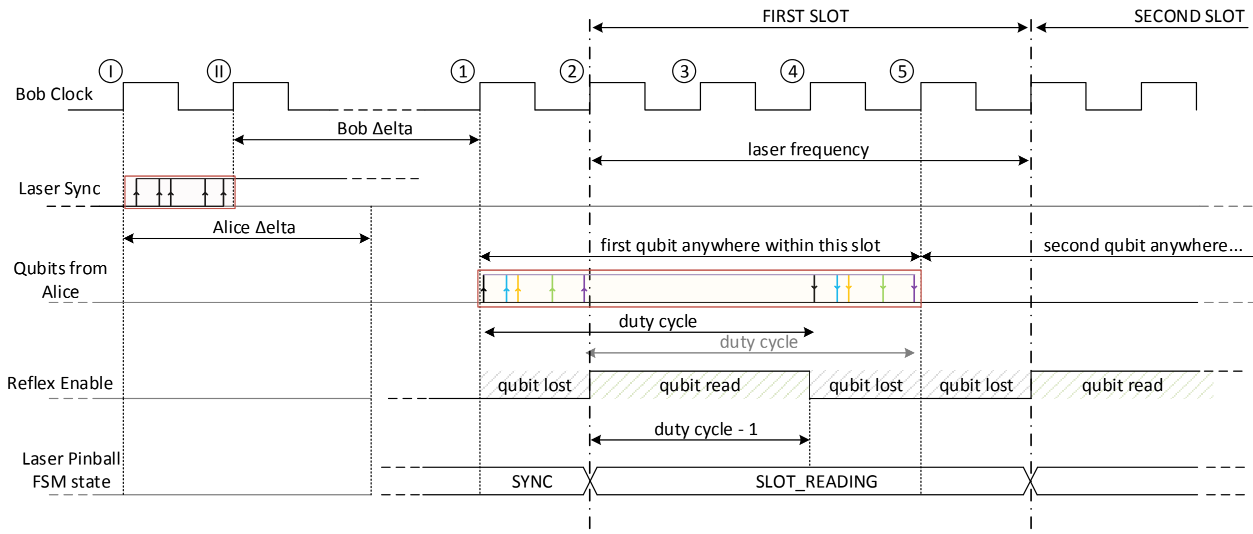


Figure 3.13: An example time plot of the qubits reading system supposing a laser period of four clock cycle with a three clock cycles duty cycle. Due to the clocks unknown skew, Alice asserts the synchronization laser between I and II clock rising edge of Bob. Hence, Bob can read it on II rising edge. After the assertion of the synchronization laser, Alice waits 500 ns plus a clock cycle (the one needed to move from *Sync* state to *BB-84/B-92* state) before asserting one of the qubit laser. On the other hand, Bob waits a custom delay (which would be equal to 500 ns in an ideal condition) plus a clock cycle (the one needed to move from *Sync* state to *Slot_reading* state) before asserting the *reflex_enable* output. Therefore, Alice asserts the qubit laser between 1 and 2 clock rising edge (with deassertion between 4 and 5) and, supposing a tuned Bob delay, Bob asserts the *reflex_enable* on clock rising edge 2. Please note that the qubit could be emitted within the laser assertion period, but the right moment is not deterministic. Thus, this system deletes one clock cycles of the qubit laser and the *reflex_enable* is not asserted when the laser qubit could not be asserted.

[21]Actually is not the whole frame-period but is the frame-period minus 50 ns in order to recover the extra time delay. If two or more consecutive synchronization lasers are missed, there is the 50 ns-correction only for the first one, since the next ones will be correctly timed.

3.4.3 Hermes Synchronization Module

The role of **Hermes Synchronization Module** is quite similar to the that of Alice `Zeus Synchronization Module` but with more responsibility within the design, since this module needs to understand the timing specifications sent from Alice through the synchronization laser. Then, it needs to use this information to perfectly time the qubits receiving and storage. The first step is to time the start of the transmission, by decoding the start signal that is sent with the synchronization laser (three impulses with settable duty cycle and frequency) or through the classic channel (Wi-Fi or Ethernet).
In the *start-with-laser* mode, the `FSM` (Figure 3.14) remains in the *Idle* state until a start signal occurs and then moves to the *Decoding_Start* state, in order to assert the *start_key* output[22], which will be used by all the other modules. Therefore, the `FSM` moves to the *Sync_Send* state, which operates similar to its corresponding in the `Zeus Synchronization Module`, and sends a *sync_start* signal to the `Laser Pinball`, so that it can time the qubits acquisition. The *sync_start* is sent after a user-chosen number of clock-cycles, which is set through the *delay_before_sync* parameter[23]. This system is used to manage the SPD activation delay, which could be different for the qubits SPDs and the synchronization-laser photon detector[24]. In the *start-with-external-channel* the `FSM` ignores the synchronization laser until the *external_start* signal is asserted[25]. When this signal asserts, the `FSM` moves to the *Sync_Send* state and then operates as described above; meanwhile the Start_key_Manager generates the main *start_key* signal. In both cases, when the communication is over, the `FSM` returns to the *Idle* state, thanks to the *full* signal coming from the `Bit Ben`.

3.4.4 Bit Ben Word Composer

The **Bit Ben Word Composer**[26] is responsible for the management and storage of the incoming bits from the `Laser Pinball`. The working principle is quite similar to the `Mind Well` with a two-state `FSM` (Figure 3.15). As usual, when *start_key* asserts, the `FSM` moves from the *Idle* state to the *Word_Composing* state where the input bits[27] are neatly stored in the 32-bit output port, which is connected to the *data_input* of

[22] A Start-Key-Manager sub-module is instantiated to manage the creation of the *start-key* signal like the `Start_Key_Watchman` does [see 3.3.4].

[23] Settable in `C++` application.

[24] Note that on Alice side this delay is hardware-set to 500 ns and in an ideal situation the *delay_before_sync* would be set to this value.

[25] This signal comes from an `AXI GPIO` an it is activated when a start signal arrives through Ethernet/Wi-Fi channel.

[26] The name plays on word Big Ben, which reminds the idea of a clock, i.e. something that accumulates seconds and increases the minutes needle. This behaviour is quite similar for this module: it accumulates the bits until the first 32-bit-vector is ready and then sends it to the memory.

[27] The module can discerns between BB84 and B92 reading all the four bits in the first case and only the first and third bit in the second case.

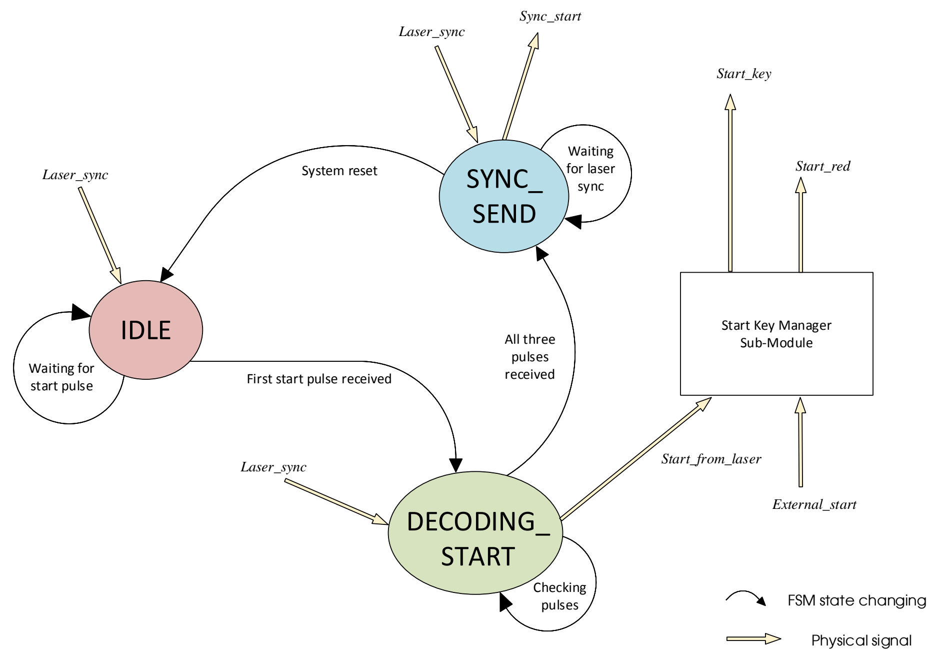


Figure 3.14: Hermes Synchronization Module FSM. For a clearer view, all the parameters signals were removed from the figure.

the `BRAM` memory. When the vector is completed, the *bram_enable* output is asserted for one clock-cycle and the data is written to the `BRAM`. Note that the `BRAM` memory has a *write_enable* input port but is not used for the writing because it is linked to the `Bit Ben` *write_enable* output-port, which is always asserted. Since there is no need for reading, the *bram_enable* port can be used instead of the *write_enable*. This system improves the `BRAM` performance: the *bram_enable* works faster than the *write_enable* and there is less power consumption (the `BRAM` is off for most of the time). The data are stored according to the address specified by the *addr* output port, which increases by four every time a 32-bit-vector is completed. According to the *bram_data_depth* input parameter, when the half and the end of the `BRAM` is reached, the *interrupt_out* signal is asserted. Like in the `Mind Well`, this interrupt is used to launch a software routine in order to move the data stored in the free-from-writing half of the `BRAM` to the `DDR` memory. A schematic view of the blocks memory connections is visible in Figure 3.8. The module also counts the number of data and when the transmission has ended, the *full* output is asserted and the `FSM` returns to the *Idle* state.

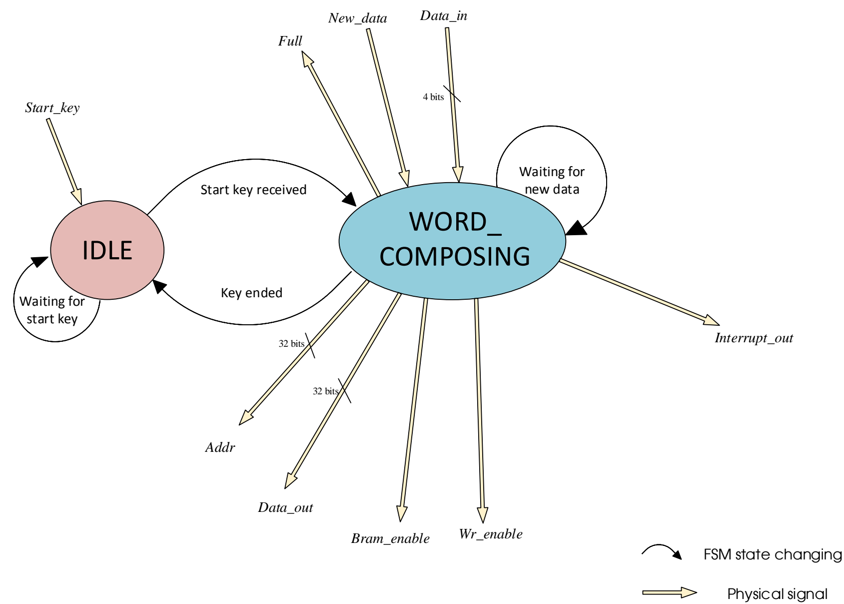


Figure 3.15: Bit Ben Word Composer FSM. For a clearer view, all the parameters signals were removed from the figure.

3.4.5 Other Modules

Like in the transmission side, Bob too needs a `Reset Handler Module` [see 3.3.4] but it does not need the `Start Key Watchman`, since its function is done by the `Hermes Synchronization Module` as described above.

Chapter 4

Softwares Design

The second part of the design requires the implementation of two different software programs. The first one is responsible for the data and parameters management as well as for the connection of the system with the hardware. The second software is the graphical user interface, which allows the full control of the apparatus.

4.1 SDK software

The on-board software was written in order to send all the parameters and the commands to the hardware. Furthermore, it manages the storage of the encoded key moving it from `RAM` to `BRAM`[1] or from `BRAM` to `RAM`, depending on the board role. In order to use the TCP connection, the *lwip* library is included in the project. Furthermore, a `global.h` header file contains some useful constant definitions and the description of the `structure` *parameters*, which absorbs all the `GPIO` parameters in order to make them more accessible throughout the code.
The software is shared by the following `C++` files and functions:

- `main.cc` file contains the `main` function which starts all the software. It sets all the needed variables, the system (`GPIO`s, `CDMA` and interrupts) and the TCP connection calling many different functions spread in all the `C++` software files.

  initEthernet() is the function that initializes the Ethernet parameters such as the board ip address, subnet mask, gateway and others. It also initializes the lwip calling the `init_lwip` function.

  decodeCommand(std::string payload) is the function responsible for decoding the incoming string from the network. Depending on the identifier, it calls different functions in order to carry out the operation required by the user. See 4.1.1 for more details about the TCP string coding.

[1] According to the `BRAM`-Buffer system implemented by the `Mind Well` and the `Bit Ben`.

bramBufferingIsr(void *InstancePtr) is one of the most important functions of the code. It is called every time an interrupt occurs on the related GPIO[2]. Therefore, it manages the BRAM-RAM memory moving. On Alice-side, when it is called, it fills half of the BRAM with a new portion of key. Clearly the operation is reversed on Bob and the function moves half portion of the BRAM to the RAM memory. A flag signal allows to determine which half of the BRAM must be filled. The moving operation is done with an XAxiCdma_SimpleTransfer [for more details about it, please see D].

resetHardware() function uses XGpio_DiscreteWrite to assert and deassert the input of the Reset Handler Module, which will launch the hardware *reset* signal.

initKeyToBram() initializes the first portion of the key in the BRAM. Calling the XAxiCdma_SimpleTransfer, it moves the first BRAM_SIZE*32 bits of the key in the BRAM, filling it completely. The BRAM half-empty-fill operation will be handled by the interrupt routine **bramBufferingIsr**. Note that this function exists only on Alice board.

std::vector <std::string>tokenize(std::string s) separates all the strings of parameters and returns a collection of strings, each one containing a parameter. In other words, it searches the *comma* char and, when it founds it, it extracts the parameter as a sub-string, assigning it to a position of the strings vector that has to be returned.

parameters decodeStringParam(std::string s) is the function that extracts all the real parameter values from the TCP string. Basically, it calls the tokenize() function which returns all the parameters as single strings and then it converts every string[3] to an integer number, assigning it to the correct element of the parameters structure.

setupQuakeParams(parameters parameters) sends correct parameters values through the whole hardware, calling the related XGpio_DiscreteWrite[4]. Note that Alice and Bob have quite the same parameters with the exception of the *delay_before_sync*, which is present in Bob only. In all, there are eleven parameters for Alice and twelve for Bob.

startKeyTransmission() function uses XGpio_DiscreteWrite to assert and deassert the input of the Start Key Watchman, which will launch the hardware *start_key* signal. This function exists in the Alice SDK software only. A

[2]Basically, the GPIO produces an interrupt signal every time its input (the *interrupt_out* signal coming from the Mind Well or from the Bit Ben) changes its value.

[3]Note that this is not true for the protocol and start-type parameters. In these cases, an *if* statement assigns 0 or 1 to the relative parameters, depending on the received strings.

[4]This function write a value on one of the two channels of the GPIO. Note that every GPIO channel is hard-wired to a hardware parameter. Hence, selecting the right channel, this function will physical set the parameter value.

similar version of this function is still present in Bob SDK software in order to give the start to Bob hardware, in case of the *start-with-ethernet* option.

acquireKey(std::string payload) is the function responsible for the acquisition of the key sent from computer. Thanks to a `for` loop, the function gets four elements at time of the payload containing all the sent key. It passes this four-element substring to the `stringToU32` function and assigns the returning u32 element to the i-th element of the key variable. Note that this function is present only in Alice.

u32 stringToU32(std::string s) converts a four-element string in one u32 element. In other words, it extracts four char from the string and builds up a u32 element assigning every char to eight of its bits.

sendKeyToPC(std::string payload) is called from `decodeCommand` when Bob receives a `QETK` command. It simply calls `sendData` function passing the key pointer and length as parameters and print some notices.

allocateKey(std::string payload) is a function present only in Bob and it creates the required memory in order to store the incoming key from Alice.

- `platform_zynq.cc`

platform_setup_gpio() initializes all the `GPIO`s.

platform_setup_cdma() initializes the `AXI CDMA`.

platform_setup_interrupts(void) sets up all the required parameters in order to use interrupts.

platform_enable_interrupts() enables the interrupt option of the `GPIO`.

init_platform() simply calls the four functions `platform_init_name`[5] which set the system *timer* required for the usage of interrupts, the `GPIO`s, the `CDMA` and the *interrupt service*. The function is called from the `main` function during the setup of the system.

cleanup_platform() is the last function called before the end of the program. It disables the data and the instruction caches.

- `ethernet.cc`

std::string transfer_data() is the function responsible for the analysis of the TCP command sent from the computer. Basically, it searches for the `END;` delimiter (see 4.1.1) in the `commandBuffer` string (see the function below), returns all the data that precede the delimiter and deletes all the command from the `commandBuffer`. The function is called in the `main` function. Hence, is the function that "moves" the TCP data from the `ethernet.cc` file to the `main.cc` file.

[5] Defined in turn in the same file.

sendData(u32 *key, int key_length) is used in Bob in order to prepare the key sending from the board to the computer. The function is called in the `main` function passing the key pointer and length as parameters. The function sends the `QEKB` identifier and calls the `sendDataChunck`, which will actually send the key.

sendDataChunck() is called by the `sent_callback` until the whole key has been sent. It takes a portion of the key equal to the output buffer size[6] and sends it to the computer using the `tcp_sent` function.

err_t recvCallback(..,..,..,..) is a *callback* function that is called every time a TCP packet is received. The function gets the incoming packet and appends it to the `commandBuffer` (static string variable). Note that the `commandBuffer` is filled by this function and emptied by the `transfer_data` function.

err_t sent_callback(..,..,..) is a *callback* function present only in Bob software and is specular of the `recvCallback` function. It is called every time the computer is ready to receive a new TCP packet. The function calls the `sendDataChunck` function if there are data to send. Otherwise it sends the delimiter in order to close the TCP command. Note that the data transfer from the board to the computer is implemented only in Bob and is used only to extract the key from the board and analyze it on the computer. On future developments the bidirectional communication will be enhanced and implemented on Alice as well.

err_t acceptCallback(..,..,..) is called from the TCP in order to establish a new connection.

startTcpServer() is called by the `main` function and setup all the element for a new connection, like the TCP structure and port connection.

4.1.1 TCP instructions coding

The board is connected to the computer through an Ethernet cable and a solid TCP connection is implemented using the `SDK` *lwip* library. The great advantage of a TCP connection is that it sends the data using packets, which prevents the loss of data in the communication.
All the instructions are coded according to a custom-made protocol. Every instruction is sent as follow:

$$\underbrace{\texttt{QE XX}}_{Instruction\ identifier} + \underbrace{\texttt{D A T A}}_{Possible\ instruction\ data} + \underbrace{\texttt{END;}}_{Instruction\ delimiter}$$

[6]Please note that the key length could not be multiple of the buffer size. Therefore the function handles this possible issue checking whether the remaining key size is greater or smaller than the buffer size.

Set Parameters command is coded with `QESP` identifier followed by all the parameters. Every parameter is separated from the others by a *comma* char. Note that the parameters are sent as a sequence of chars and not as integer type numbers. The `SDK` software does the conversion char⇒ number once the whole command has been received.

Key Transfer includes many different commands. On Alice-side, a key transfer is the command responsible for the transmission of the key from the computer to the board. The key is sent divided in many blocks[7] and the first one is coded with a `QEKB` identifier followed by the first portion of the key[8]; all the following blocks are coded with a `QEKK` identifier followed by a portion of the key. On Bob-side things are a little bit different since the key transfer must be done on the reverse way. When the user requires the key to be transfered from the board to the computer, a command with only the `QEKT` identifier is sent to the board. Therefore, Bob interprets the command and starts to transfer the key from its `RAM`, labeling every TCP packets with a `QEKK` identifier (`QEKB` for the first packet) and the `END;` delimiter. Between them a part of the key relies.

Start Key command is coded with a `QEST` identifier with no data section.

Close Board Software command is coded with a `EXIT` identifier with no data section.

4.2 *uQuake* user interface

Once the entire system was designed and set up, a user interface was required in order to manage the board operation. For this purpose, the uQuake software was created, using the Qt Creator software, which allows to design and realize a new software from scratch including the Graphical User Interface (uQuake GUI is visible in figure 4.2).

Alice and Bob interface

The software has two different working options: thanks to a hidden menu on the top right, the user can switch between Alice and Bob software revealing and hiding specific GUI sections. When Alice is selected, the *Synchronization Needle* menu is hidden; on the contrary, when Bob is selected the *Start Key Transmission* menu is hidden. On both mode, a TCP connection can be established with the board using the *Connect to* buttons and according to the IP and port custom parameters.

[7] See 4.2 for more details about block division.

[8] Please note that the key is transmitted with a one-bit⇒one-qubit coding. Hence, a char sent through the network describes four or eight qubits, depending on the quantum cryptography protocol.

Set parameters

Once the connection to the board is established, pressing the *Set parameters* push-button will send all the parameters to the board using the default values. In order to make a user-friendly software, the parameters have an intuitive form and unit of measure but these can be different from the effective value sent to the board. For example, the qubit laser frequency is expressed in MHz, but the board does not have the knowledge of MHz and it uses the number of clock cycles for computing the assert-deassert time of the qubit laser. Therefore, a 1 MHz frequency will be elaborated by the software which will sent the number 100 to the board, i.e. the board will consider the qubit laser frequency as 100 clock cycles. Obviously, this example implies that the board uses a 100 MHz clock. If the board uses a different clock frequency, this must be set up in the dedicated parameter menu on the bottom left of the uQuake GUI[9]. Indeed the qubit laser frequency value will be calculated upon the clock frequency and with a 200 MHz clock frequency, the actual value will be doubled. The user can change each value simply by clicking on the dedicated spin box button. At any parameter change, the software makes a new dispatch of all the parameters to the board. Furthermore, moving the mouse over a parameter will display a specific help text with a complete parameter description.

Data verification

The key transfer between board and computer can be activated with the *Key Transfer* push-button. As a matter of facts, this push-button sends the key from the computer to the board or requires the board to send the key to the computer.
In Alice-mode, the original qubits file is read from a text file containing zeros and ones chars which represent the qubits to be sent. Hence, the software builds up a bytes array, putting together eight chars for every byte. The final bytes array is divided in blocks of 10000 bytes[10] [please refer to 4.1.1 for details about block structure]; this prevents to overload the board CPU, since it can manage small portion of key at a time instead of handle the whole key at once. If the BB-84 is selected, then the software also reads the bases text file and merges it with the qubits file, taking one char from one and one char from the other at a time.
In Bob-mode, pushing the *Key Transfer* button sends a request to the board and the software reads the incoming data from Bob. Once all the data are received, they are analyzed and compared to the original key. The final results are printed on the Qt terminal showing the QBER, the number of missed-sync qubits and the number of time where no SPDs clicked.
Besides, the *Verify Key* and *Print Qubits* push-buttons were used as a quick debugging

[9]Since the clock frequency change is a thorny thing, a check box must be activated in order to modify the default 100 MHz value. Besides, a pop-window will appear warning the user to verify that the actual clock frequency was already changed on the hardware.

[10]Obviously there is a block truncation in case of a byte-converted-key not multiple of 10000.

functions and, sending dedicated command, allowed a verification of the internal data transfer as well as an approximate key analysis in earlier system tests. Since the complete key transfer and analysis is implemented with *Key Transfer*, these buttons are not required anymore and probably will be deleted in future developments.

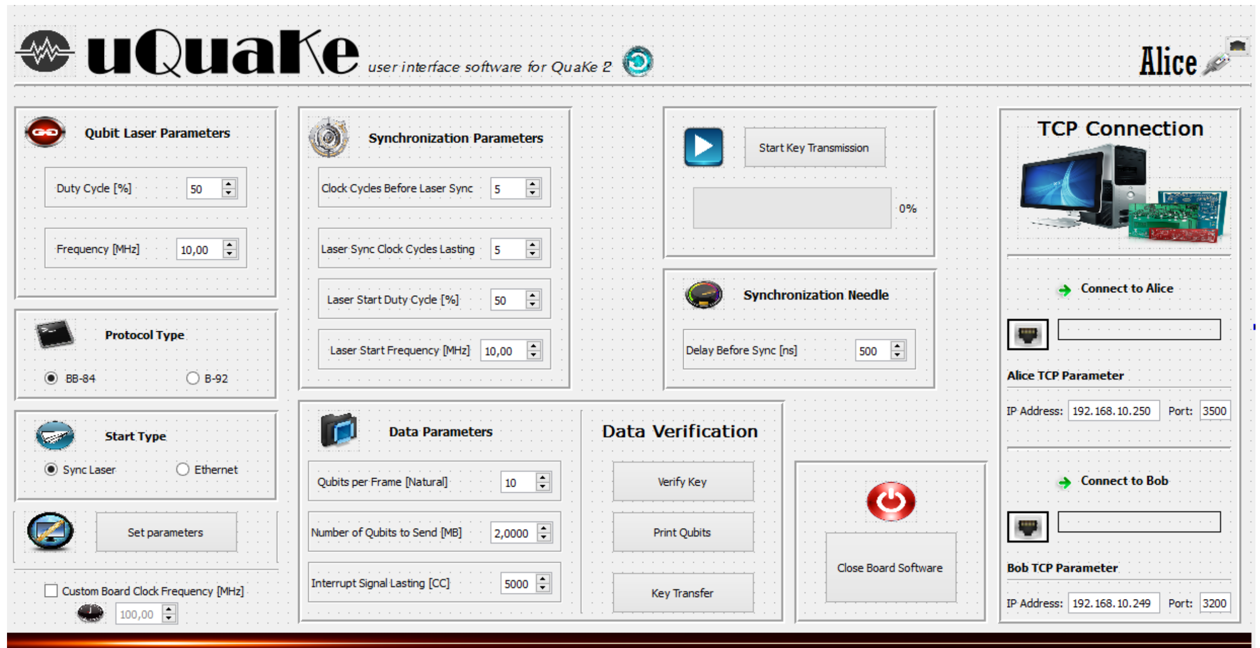


Figure 4.1: uQuake main view. Please note that this is not a running-view of the program but a Qt perspective. Hence, all the elements are visible.

Chapter 5

Test and future of QuAKE

5.1 Hardware Testing

Once the hardware was finished, many different tests were done in order to verify the functionality of the system. A thorny point was to test the right communication between software and hardware. Hence, with a basic `SDK` software, the parameters setting system was tested as well as the `BRAM` filling-emptying system. Furthermore, the custom modules were tested together[1] creating a dedicated Vivado project which included both Alice and Bob modules. The project did not contain the IP Cores since a hardware test-benching test was done and IP Cores utilization required a software part. During this important hardware test, many different parameters combinations were examined in order to ensure that the system could support every required conditions. That is, for example, the usage of any laser frequencies up to 50 MHz with variable duty cycle up to 100% or the usage of any frame size or protocol type. Furthermore, the effectiveness of the Alice-Bob synchronization system was deeply verified[2]. All these tests allowed to correct some prime design errors and to fix some bugs. Therefore the hardware was verified to work perfectly.

5.2 Software testing

Several tests were done in order to verify the functionality of both software programs. First of all, the tests checked the setting of the parameters from uQuake to the hardware, passing through the `SDK` software. An essential tool was the `Vivado Hardware Manager`, which allowed to view the real hardware behaviour. In other words, it was

[1]Please note that all the custom modules were already tested individually during the design with dedicated test-benches.

[2]Unfortunately, this test verified only the goodness of the system and not its robustness to clock-drift, since a true test could be done only with the two boards together. In other words, the test used a mock clock frequency created by the computer and, though the possibility of having two different clocks shifted relative to one another, there was no possibility to create two clocks with different drifts.

possible to verify that setting the qubit laser to 10 MHz, for example, the physical output signal had that frequency for sure. Furthermore, it was verified the functionality of the interrupt for the `BRAM` management as well as the TCP connection. Unfortunately, the TCP connection presented some speed issues, since it was not as fast as expected. This error is probably due to the initialization-bug of the *lwip* library and its fixing is on schedule for future developments. However, the data transmission through the TCP connection was tested to be completely safe.

A basic test was done to verify the working principle of the *start-with-ethernet* configuration. This type of start was not completely implemented on software-side and was postponed to future developments. The test simply confirmed the functionality of Bob, i.e. the correct operation of the *external_start* and *start_red* signals. The same thing was done for the *tangled_alarm* signal. Its functionality was tested and it was verified that the system can recognize over-write errors. But its completely implementation was postponed to future developments. As a matter of facts, the over-write error is really a remote possibility and was not considered as essential for the prime level system. Thus, all these tests allowed to correct some prime design errors and to fix some bugs.

5.3 Whole system testing

Once the whole design was completed, the system was tested in order to verify its fully operation. The two boards were connected together through normal 50 Ω-coaxial cables with SMA connectors. The Zedboards were provided with the ZEPIs (see Appendix B) linked to pmod ports and a 100 MHz clock frequency was used both on Alice and Bob. The tests explored different parameters combinations and were carried out as follow:

- key length between 40000 and 80 million qubits. The test verified the transmission of different keys with a variable length. The system revealed that, increasing the key length, the `SDK` software tended to work slowly. However, this aspect does not change the precision of the system and a faster data-handling is on schedule for future developments. For convenience, once the maximum number of qubits to send was successfully tested, a key length of 8 million qubits was used for testing the other parameters.

- qubit laser frequency between 10 KHz and 10 MHz. Actually, the system did not presented any issue with the changing of this parameter and works perfectly with different duty cycle values as well. The maximum value tested for duty cycle was 20% with a 10 MHz frequency, i.e. a duty cycle equal to two clock cycles (remember that on Bob side the actual duty cycle is a clock cycle less than the Alice duty cycle). Please note that with a 100 MHz board clock frequency, the maximum qubit laser frequency is 100 MHz but with no control on the duty

cycle, which is fixed to 100%. Otherwise, the next allowed frequency is 50 MHz with a 50% or 100% duty cycle. These high frequencies was not considered for this primal tests, since they implied high-performing laser devices. Hence, higher frequencies testing is on schedule for future developments and with a higher clock frequency.

- protocol switch between BB84 and B92. As expected, this test did not reveal any errors. For convenience, during the testing of the other parameters, the BB-84 protocol was selected.

- frame size between 10 and 300000 qubits. This test was one of the most important since it estimates the real boundary of the system functionality. The qubit laser frequency was set to 10 MHz with 50% duty cycle, since this frequency is more comparable with the clock frequency and is the one where the clock drift is more remarkable. According to the clock drift measures [see Appendix A], the system would not present any error with a synchronization every 20 ms (i.e. a frame size of 200000 with a 10 MHz qubit laser frequency). As a matter of facts, a zero QBER was detected with any frame size up to 200000 qubits. Besides, extreme tests were made with a frame size up to 300000 qubits. It was noticed that the system tended to mismatch the qubits-reading around the 250000th qubit. Therefore, an absolute safe frame size boundary could be fixed to 200000 qubits with a qubit laser frequency of 100 MHz. To be more precise, the synchronization must happen at most every 20 ms. Please note that this boundary could have to be lowered if a smaller duty cycle value is used, because the clock drift could be even more remarkable. Therefore, a complete testing of the relation between duty cycle and synchronization time is on schedule for future tests. For convenience, the frame size was set to 1000 qubits during the testing of the other parameters.

These results can be considered positive, since they prove the right working of the system, and confirm correctness of the overall system.

5.4 Future developments

The system may be further improved. As a matter of facts, increasing the functionality and the reliability of both the hardware and the software parts, the system can increase its adaptability to very different scenarios. Thus, this new version of QuAKE can become a trustworthy landmark for future experiments and applications.
In particular, future developments could include:

- hardware development
  - implementation of a *tangled_alarm* signal on Bob for greater data-storage security

  - usage of higher clock frequency for better synchronization flexibility

- software development
  - continuously-working-system with an automatic key send-store-analysis application over many keys. This application allows the usage of the system for quantum information exchange intensive experiments
  - improvement of TCP speed connection
  - further increase of the key length
  - full implementation of the *start-with-ethernet* option
  - improvement of the interrupt routine adding the handling of the *empty*, *full* and *tangled_alarm* signals
  - uQuake graphical interface improvement for adjustment to different OSs

- hardware and software development
  - usage of GPS for the synchronization to improve timing performances
  - usage of an SD card for the key-storage in order to have no limits to the key size
  - usage of Linux OS on the ZedBoard for exploring a wide area of new possibilities

Conclusions

This work described the realization of a brand new system for QKD experiments, starting from the earlier QuAKE project. Using the FPGA technology along with `C++` programming, the new version of QuAKE was designed with the aim of providing a clear and easy-to-use system. The initial objectives were well achieved. Now the system has:

- a brand new reliable hardware with a modular structure
- a reliable synchronization system with extended frame size possibility
- the possibility to set many different parameters for improved flexibility to different scenarios
- the possibility to send up to 160 million qubits
- the possibility to switch between BB84 and B92 protocols
- new software with TCP connection

Both hardware and software new features have been developed, implemented and successfully tested.

In the near future, it will be also tested the actual capability of the system to work with different optical setups.

Furthermore, the system offers a great potential for the implementation of further features and facilities. As a matter of facts, future developments could expand the system functionality with the usage of the GPS technology, Linux OS and SD memory card.

Recently, satellite and free-space QKD has experienced a great progress, widening the scientific community interest. Therefore, this new version of QuAKE can improve the reliability of different quantum information experiments thanks to its increased flexibility and adaptability to a wide range of scenarios.

Appendix A

Clocks Drift Measures

Usually the clock of any digital system is generated from an analog device which is commonly known as Voltage Controlled Oscillator or VCO. Generally, this device has a quite simple structure (but the design could be very demanding) and provides a sine wave output signal[1] at a frequency that can vary from hundreds of kHz to tens of MHz. Obviously, talking about a real device, the output frequency is not 100% perfect but can vary within a range of values, with kind of precision, and there can be also some harmonic frequencies. The frequency error within a period is measured in p.p.m. (parts per million) and, typically, a MHz VCO has an error that varies between 50 and 100 p.p.m., which is a negligible error in most of the cases. But the summing of this error on every period can lead to a significant error. This problem is known as clock-drift and could compromise the reliability of the communication between Alice and Bob, especially with long-time transmission. For example, depending on the specific clock-drift, Alice could be one clock-cycle ahead or late than Bob and vice-versa.
For this reason, a measure of the Zedboards' clock-drift was done and the results were used to set the maximum frame-size. According to the Zedboard specification 2.2, there are two different oscillators: a 33.333 MHz PS-side oscillator and a 100 MHz oscillator on PL-side. Therefore, two measures were required for every board[2] in order to find which oscillator is the best one.
The measure was done implementing a custom clock-counter. Depending on the switch user buttons combination, the hardware counted one million (10 ms), one hundred million (1 s) or one billion clock cycles (10 s). Go-signal and stop-signal were asserted, respectively, at the beginning and at the end of the count. Using the Tektronix MSO 3054 Digital Oscilloscope, the gap between go and stop was measured. The results were quite the same for both boards but an important difference arose from the measures. As a matter of facts, the PS oscillator has less clock-drift than the PL one. The PL

[1] The VCO actually works because is an unstable circuit and can produces such a wave.

[2] Please note that the native frequency of the oscillator can be multiplied using PLLs or DCMs. The `Zynq` Vivado block offers many different outputs for different clock frequency obtained from the PS oscillator and a 100 MHz custom-output was used for the measure.

oscillator gains[3] about 35 ÷ 40 ns every 10 ms, whereas the PS gains only 20 ÷ 25 ns. But this deviation represents the error from the ideal time measurement. The system does not work referring to an external time but it works referring Bob clock to Alice clock. Therefore, since the measure approximation is within 5 ns, a safety frame time could be around 10 ms. Furthermore, the clock chosen to be used in the system was the PS one. Figures A.1 and A.2 show, respectively, the go-stop system with large time-scale and the zoomed view on the edge of the stop-signal with the indication of the time between the rising edge and the triggered go-signal. Both images represent a measure done on the PL oscillator.

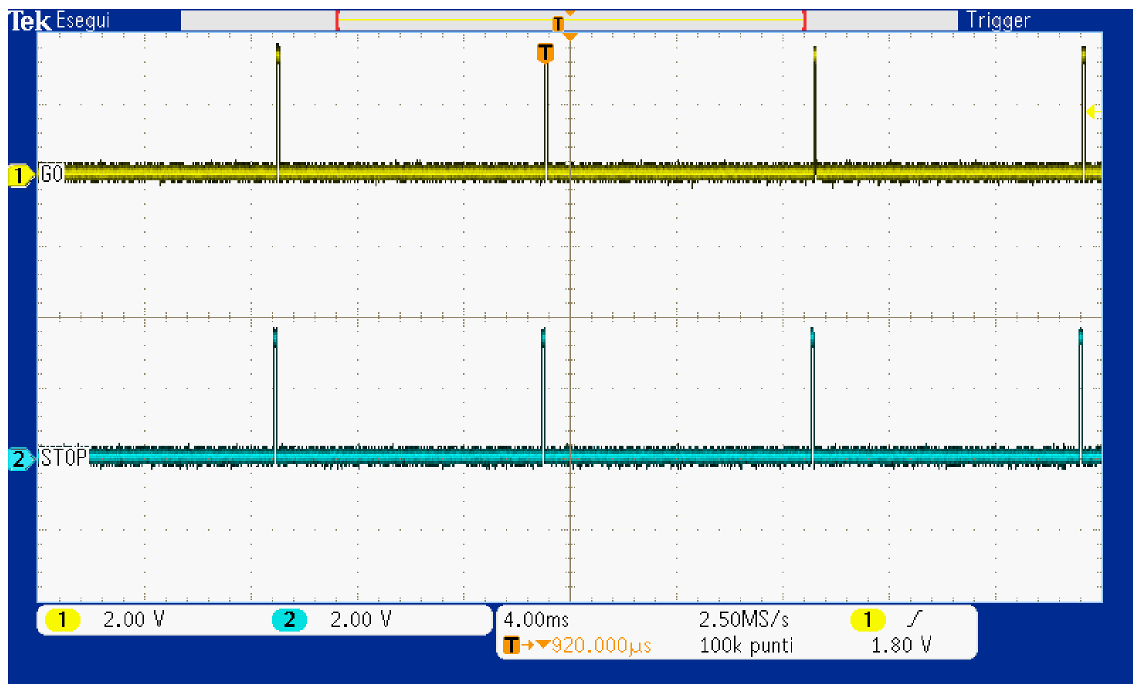


Figure A.1: View of the go-stop counting system set to one million counts. The yellow waveform represents the go-signal, the blue one the stop-signal. The spikes on every waveform are the asserting-deasserting of the signals. Once the hardware has finished the count, it starts all over. Therefore, the first go-signal of the figure is ment to be joint with the second stop-signal. The next go-signal asserts right after the deassertion of the previous stop-signal and so on.

[3]Please note that the following data include the measures of both boards and also evaluate the clock jitter.

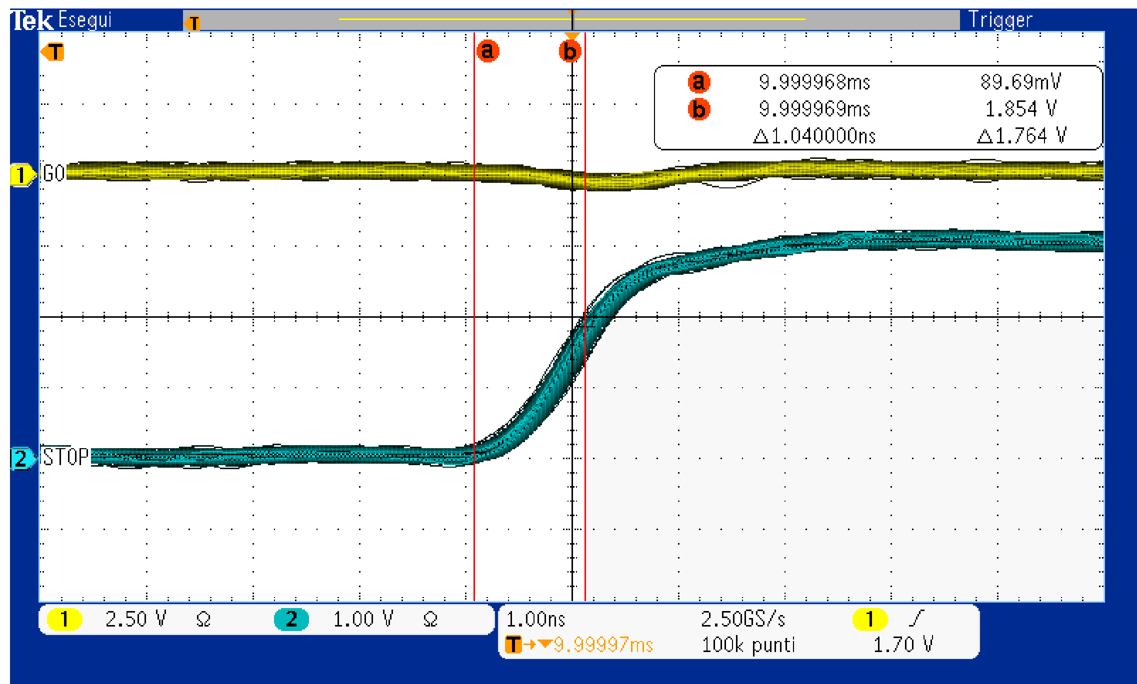


Figure A.2: Zoomed-in view of the stop signal rising edge. Two extra cursors were added in order to improve the precision of the measure: the b cursor was placed on the half of the rising edge; on the other hand, the a cursor was placed right before the change of the last digit of its position-time indicator. This allowed to use the cursors-time-difference-indicator and have a hundreds-fempto precision. Please note that the thickness of the wave form is due to the infinite persistence set on the oscilloscope. This allowed to place the cursor in a good spot regarding the clock jitter effect. Also note that the thickness does not represent the final value of clock jitter, since this is not a predictable phenomenon and the waveform will become thicker as time goes on.

Appendix B

Zedboard Extension Pmod Interface

An important step was the creation of an external board in order to make the Zedboard fully operational for the project. The Zedboard provides up to five *Pmods* output connectors with eight signals each, a doubled ground and power supply outputs.
Digilent company offers many *Pmods* interfaces for many different aims but as no one was appropriate for this purpose, we had to build our own interface in order to link the Zedboards to our equipment. Using the *Eagle* software, we designed an interface that could be linked to the Zedboard through two *Pmods* connectors and provides eight output signals with SMA connectors and eight more outputs through a male pin header. An extra 3-pin ground connection was also added for any extra purpose. The interface was called Zedboard Extension Pmod Interface or Zepi.
In order to reduce the cross-talk effect, the ground plane was laid down on both faces of the PCB. As a matter of facts, minimizing the gap between the signal track and the ground track is the only way to reduce this kind of effect [16]. Hence, a double-ground plane allows very short distance even with multiple signal tracks laid very close together, like in this case.
Furthermore, some considerations about the network matching were made. The outputs of the first two *Pmods* (JA and JB) have a 200 Ω output series resistor and do not match the 50 Ω characteristic impedance of the network (a 50 Ω SMA-coaxial cable is used). Therefore, the Zepi is designed to be used on the 50Ω-matched-Pmods (JC and JD) which do not have the 200 Ω series resistor. But the network matching is highly frequency-dependent [17] and this configuration still suffers from network matching issue, since the outputs will be used in a wide range of frequencies. In conclusion, the usage of the Zepi with the not-matched *Pmods* is not forbidden but simply not recommended. The Eagle layout is visible in figure B.1 and the finished Zepi is visible in figure B.2.

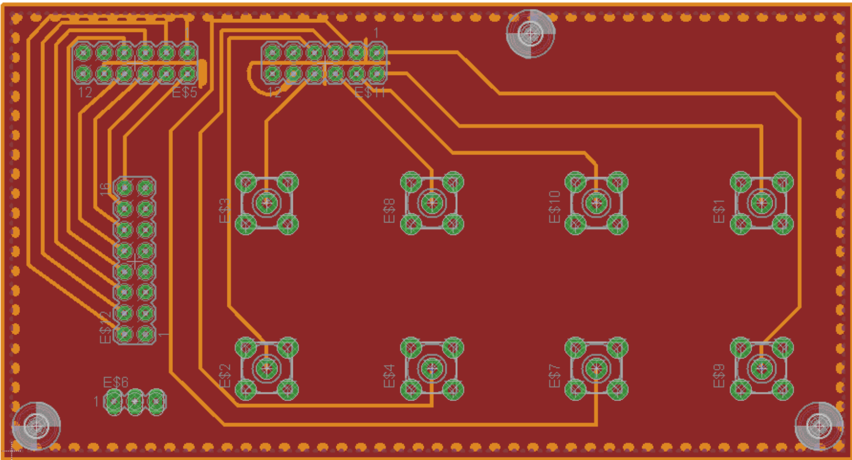


Figure B.1: Eagle-view of the Zepi layout

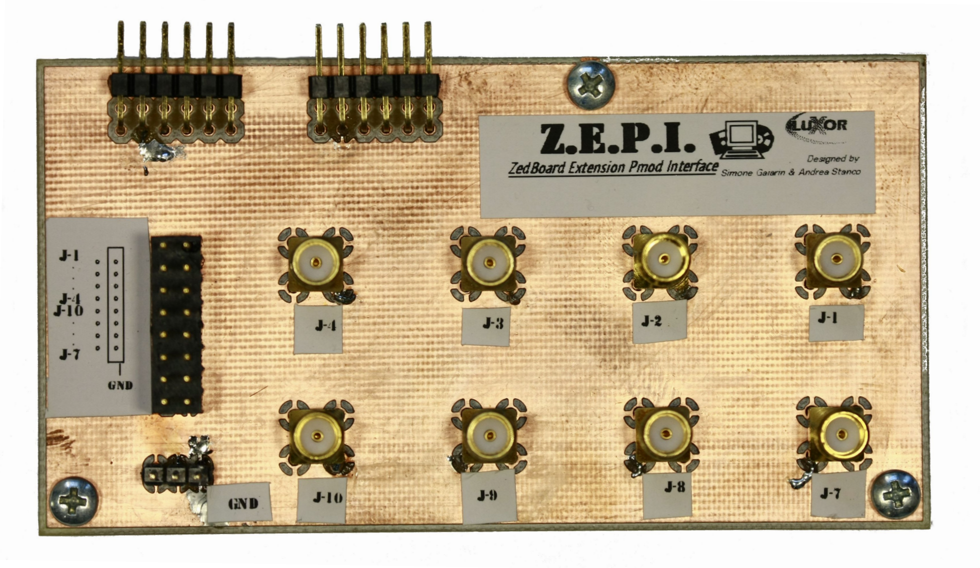


Figure B.2: View of the Zepi

Appendix C

Reset: a-synchronous analysis

Usually, a very common way to reset flip-flops is to use an asynchronous reset which is not bounded to a clock domain and its assertion and deassertion can be read independently. This mode works most of the times and no issue occurs, but there is a possibility that the flip-flops end up in a metastable condition with metastable output and unpredictable behaviour. The reason of such an error is that a flip-flop needs a slightest time to set up after reset has occur. Therefore, the time between the reset assertion and the first clock rising edge must be greater than the flip-flop reset recovery time (Figure C.1). If not, the flip-flop will not be ready to operate when the clock rising edge occurs and, as a result, it will fall into metastability.

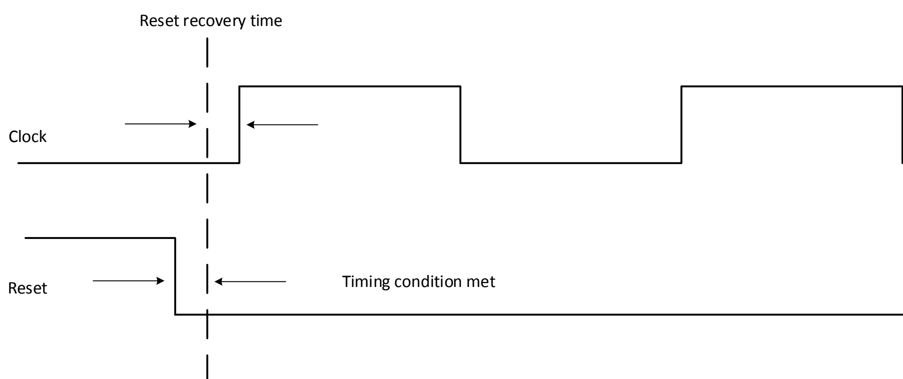


Figure C.1: Reset recovery time scheme

On the other hand, a fully synchronous reset does not have this kind of issue, since

its assertion and deassertion are read only on clock edges, and flip-flops have an entire clock cycle to set up after the reset deassertion. But the synchronous reset has another issue: the reset pulse (time between assertion and deassertion) must be greater than a clock period[1] or could not be captured by the flip-flop.
Hence, according to [18], a good solution takes the best from the two worlds: asynchronous assertion guarantees that the reset is captured by the flip-flop, whereas synchronous deassertion avoids the recovery time issue.

[1]To be more precise, the synchronous reset could work even with very short pulse but the pulse must fall over the clock edge. However, this represents a very critical situation too.

Appendix D

C++ interrupt routine and XAxiCdma_SimpleTransfer

Within the SDK software the interrupt routine is essential for a correct management of the data. Therefore, a part of the code of the **bramBufferingIsr(void *InstancePtr)** function [4.1] of Alice is reported. The code reads the value of the interrupt GPIO and if it is equal to a logical 1 value[1] it makes the memory transfer as follow: if the interrupt_in_gpio flag variable is equal to 1, then it moves a portion of the key, that corresponds to half-BRAM, from the first not-read address of the RAM to the base address of the BRAM or to the address of the half of the BRAM. As a matter of facts, the `XAxiCdma_SimpleTransfer(CDMA Driver instance, source address, destination address, length n, NULL, NULL)` moves n-elements from the source address to the destination one. Clearly, on Bob-side the destination and the source address are swapped.
For more details about the `XAxiCdma_SimpleTransfer` function, please see [19].
Note that the AXI BRAM controller and the BRAM are not visible to the system, since they are linked directly to the AXI CDMA. Thus, they are not istantiated by the Zynq. They are visible only as addresses.
Here is the portion of the code:

```
interrupt_in_gpio = XGpio_DiscreteRead(&GPIO_Interrupt::interrupt_in , 1);

if (interrupt_in_gpio == 1) {
 if (first_memory_half) {

 XAxiCdma_SimpleTransfer(&CDMA::axiCdmaInstance , (u32) key_current_position ,
  BRAM_BASE_ADDRESS, BRAM_SIZE_IN_BYTE / 2, NULL, NULL);

  first_memory_half = 0;
```

[1]Note that the AXI GPIO sends an interrupt signal even when its input deasserts; but in that case no memory transfer is required.

```
  }
  else {

  XAxiCdma_SimpleTransfer(&CDMA::axiCdmaInstance, (u32) key_current_position,
  BRAM_BASE_ADDRESS + BRAM_SIZE_IN_BYTE / 2, BRAM_SIZE_IN_BYTE / 2, NULL, NULL);

  first_memory_half = 1;
  }
key_current_position += BRAM_SIZE_IN_BYTE / (2 * 4);
}
```

List of Figures

1.1 The Bloch sphere. 9
1.2 The Poincarè sphere. 10

2.1 QuAKE schematic view 18
2.2 ZedBoard view 19
2.3 ZedBoard Block Diagram 21

3.1 Zynq PS IP Core 24
3.2 Alice custom blocks 28
3.3 Alice Vivado Block Design 29
3.4 Alice Vivado post-implementation results 30
3.5 Laser Controller Module FSM 30
3.6 Zeus Synchronization Module FSM 31
3.7 Mind Well Memory Manager FSM 33
3.8 Memory scheme 34
3.9 Bob custom blocks 38
3.10 Bob Vivado Block Design 39
3.11 Bob Vivado post-implementation results 40
3.12 Laser Pinball Communication Sorter FSM 41
3.13 Laser Pinball qubits reading time plot 42
3.14 Hermes Synchronization Module FSM 44
3.15 Bit Ben Word Composer FSM 45

4.1 uQuake GUI 53

A.1 Clock-drift zoomed-out view. 62
A.2 Clock-drift zoomed-in view. 63

B.1 Eagle-view of the Zepi layout 66
B.2 View of the Zepi 66

C.1 Reset recovery time scheme 67

List of Tables

1.1 BB84 bit encoding lookup table 13
1.2 BB84 operation example 14
1.3 B92 bit encoding lookup table 14
1.4 B92 operation example 15

3.1 Xilinx IP Cores 36

Bibliography

[1] G. Benenti, G. Casati, G. Strini *Principles of Quantum Computation and Information Vol. 1: Basic Concepts*, World Scientific Publishing Company, 1th edition, 2004. ISBN: 978-9-812-38830-8

[2] B. E. A. Saleh, M.C. Teich, *Fundamentals of Photonics*, John Wiley & Sons, 2nd edition, 2007. ISBN: 978-0-471-35832-9

[3] G. Benenti, *Principles of quantum computation and information: Basic tools and special topics*, World Scientific Publishing Company, 1th edition, 2007. ISBN: 978-9-812-56345-3

[4] James F. Kurose, Keith W. Ross, *Computer Networking: A Top-Down Approach*, Prentice Hall, 6th edition, 2012. ISBN: 978-0-273-76896-8

[5] C. Bennett and G. Brassard, *Quantum cryptography: Public key distribution and coin tossing*, Proceedings of International Conference on Computers, System and Signal Processing, Bangalore, India, Dec. 10-12, 1984

[6] Simone Gaiarin, *Signal Handling for Space Quantum Key Distribution*, Master Thesis, Università di Padova, 2013

[7] C. Bennett, *Quantum cryptography using any two nonorthogonal states*, Phys. Rev. Lett. 68, 1992, pp. 3121-3124.

[8] Davide Bacco, *Comunicazione Quantistica Finalizzata alla Realizzazione di Chiavi in Spazio Libero*, Master Thesis, Università di Padova, 2010

[9] Zedboard hardware guide, online available `http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf`

[10] Clive Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 1st edition, 2004. ISBN: 0-7506-7604-3

[11] `AXI` protocol Xilinx reference guide, online available `http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf`

[12] M. Morris Mano, Charles R. Kime, *Logic and Computer Design Fundamentals* , Prentice Hall, 4st edition, 2007. ISBN: 978-0-131-98926-9

[13] Andrew Rushton, *VHDL for Logic Synthesis*, Wiley, 3rd edition, 2011. ISBN: 978-0-470-68847-2

[14] `Vivado Synthesis` documentation, available `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug901-vivado-synthesis.pdf`

[15] K.Kuusilinna, V. Lahtinen, T. Hämäläinen and J. Saarinen, *Finite State Machine encoding for VHDL synthesis*, IEE, No.1 January 2001

[16] Tim Williams, *The Circuit Designer's Companion*, Newnes, 2nd edition, 2005. ISBN: 978-0-7506-6370-0

[17] David M. Pozar, *Microwave Engineering*, John Wiley & Sons, 4th edition, 2011. ISBN: 978-0-470-63155-3

[18] Steve Kilts, *Advanced Fpga Design*, Wiley-Interscience, 1st edition, 2007. ISBN: 978-0-470-05437-6

[19] `AXI CDMA` documentation, online available `http://www.cs.indiana.edu/hmg/le/project-home/xilinx/ise_13.2/ISE_DS/EDK/sw/XilinxProcessorIPLib/drivers/axicdma_v2_01_a/doc/html/api/xaxicdma_8h.html#a20`