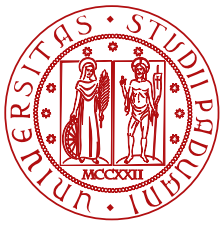


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Sviluppo e Implementazione di algoritmi di post-processing per Comunicazioni Quantistiche su un sistema RISC-V in FPGA

Relatore
Dr. Stanco Andrea
Correlatore
Dr. Bertapelle Tommaso

Laureando
Specia Alex
Matricola 2041787

ANNO ACCADEMICO 2023-2024
Data di laurea: 23 settembre 2024

Sommario

L'obiettivo di questa tesi è quello di creare e testare un sistema basato su una piattaforma Field Programmable Gate Array (FPGA), su cui poter implementare degli algoritmi per comunicazioni quantistiche, usati in particolare nella Quantum Key Distribution (QKD) durante la fase di post-processing. Il sistema sviluppato si basa su un processore che implementa l'architettura RISC-V, uno standard di tipo open source compatto e versatile. La CPU in questione è chiamata NEORV32 ed è anch'essa di tipo open source. La scelta di impiegare questo tipo di configurazione è dovuta alla serie di vantaggi che la contraddistinguono. In primo luogo, essendo un sistema FPGA, può essere riconfigurato tramite programmazione il che lo rende molto versatile. In aggiunta NEORV32 è un processore soft-core anch'esso ampiamente configurabile, disponendo di un'architettura di tipo modulare. Dopodiché possiede una buona efficienza dal punto di vista energetico, potendo attivare e disattivare i vari moduli per svolgere solo le funzionalità essenziali per l'applicazione sviluppata. Questo sistema trova un rilevante utilizzo nelle applicazioni spaziali, essendo particolarmente resistente alle radiazioni spaziali e per le caratteristiche precedentemente elencate. Il principale scopo dei test è quello di valutare le prestazioni del sistema in termini di velocità e consumo di potenza, utilizzando l'implementazione di un algoritmo chiamato biaser. Tale algoritmo sostanzialmente si occupa di produrre dei bit con una distribuzione di probabilità arbitraria, partendo da un flusso di bit generati in modo casuale. Quest'ultimo viene sfruttato nella QKD per riuscire, tramite altri processi di elaborazione specifici, a incrementare l'efficienza del protocollo riguardante la generazione della chiave.

Abstract

The purpose of this thesis is to design and test a Field Programmable Gate Array (FPGA) based platform to implement quantum communication algorithms, which are used in Quantum Key Distribution (QKD) during the post-processing stage. The developed system is based on a processor that implements the RISC-V architecture, an open source standard that is extremely adaptable and simple. The CPU is named NEORV32 and it is also open source. The choice of using this configuration is due to the many advantages it offers. First of all, it is an FPGA system, so it can be reprogrammed, making it highly versatile. Furthermore, NEORV32 is a soft-core processor characterized by a modular architecture, which makes it widely configurable. Moreover it features good efficiency in terms of power consumption, in fact the different modules can be activated or disabled to execute only the essential operations for the developed application. This system is especially used in aerospace applications because it is particularly resilient to space radiation and also due to the features mentioned above. The main purpose of the tests is to evaluate the system's performance in terms of speed and power consumption, through the implementation of an algorithm called biaser. This algorithm essentially creates bits with an arbitrary probability distribution, starting from a randomly generated bitstream, which is used through other specific processes in QKD to increase the protocol's efficiency regarding key generation.

Indice

Capitolo 1

Introduzione

L'obiettivo di questa tesi è quello di creare un sistema basato su una piattaforma Field Programmable Gate Array (FPGA), ovvero la scheda PYNQ-Z2 di Xilinx, su cui poter implementare degli algoritmi di post-processing per la Quantum Key Distribution. Nel dettaglio lo scopo principale è quello di valutare le performance, in termini di velocità e consumo di potenza, del sistema attraverso l'implementazione di un biaser. Quest'ultimo è un algoritmo che, servendosi di una sorgente di bit generati in modo casuale, produce un flusso di bit con una distribuzione di probabilità arbitraria. Il quale viene sfruttato nella QKD per riuscire, tramite altri processi di elaborazione specifici, a incrementare l'efficienza del protocollo riguardante la generazione della chiave. Il design trattato è costituito da un processore soft-core, che implementa l'architettura RISC-V, uno standard di tipo open source compatto e versatile. La CPU in questione è chiamata NEORV32, anch'essa di tipo open source. La scelta di adoperare questa configurazione è dovuta alla serie di vantaggi e caratteristiche che la contraddistinguono. In primo luogo, essendo un sistema FPGA, può essere riconfigurato tramite programmazione, rendendolo estremamente versatile. In secondo luogo, lo stesso processore soft-core è stato ideato per essere configurabile. Esso infatti dispone di un'architettura di tipo modulare, in cui i vari moduli possono essere attivati e disattivati a piacimento. Inoltre risulta essere molto compatto, il che insieme alla sua vasta configurabilità, conferisce a questo sistema una buona efficienza dal punto di vista energetico. Disattivando e attivando opportunamente i moduli messi a disposizione, si possono svolgere solo le operazioni strettamente necessarie al funzionamento dell'applicazione in questione, eliminando i consumi superflui. Questo sistema trova particolare utilizzo nelle applicazioni spaziali, date le sue caratteristiche. Il fatto di essere configurabile a piacimento, il basso consumo di potenza che esso presenta, il peso e le dimensioni fortemente ridotte, lo rendono molto appetibile in questo campo. In aggiunta risulta essere resistente alle radiazioni spaziali, dei disturbi capaci di generare frequenti errori funzionali ai processori usati nell'aerospazio. Essendo l'ISA RISC-V completamente trasparente e NEORV32 ampiamente modificabile, possono essere eseguiti dei

cambiamenti strutturali mirati, per mitigare i malfunzionamenti dovuti all'ambiente circostante.
La tesi si articola in 4 capitoli: il primo capitolo si occupa degli aspetti introduttivi agli argomenti trattati, che sono i dispositivi FPGA, lo standard RISC-V e la Quantum Key Distribution.
Il secondo capitolo descrive il setup sviluppato, elenca i dispositivi che lo compongono e riassume gli step di utilizzo della piattaforma. Dopodiché illustra l'algoritmo su cui si basa il codice prodotto per implementare il biaser e ne spiega brevemente il funzionamento.
Il terzo capitolo si dedica allo svolgimento dei test e ai risultati ottenuti. Presenta alcuni dettagli e problemi riscontrati nelle varie prove, inoltre riporta delle considerazioni di confronto tra i valori ricavati attraverso la piattaforma FPGA e un processore i7 di intel.
Il quarto e ultimo capitolo riporta delle considerazioni finali in merito ai risultati ottenuti e a tutto il lavoro svolto.

1.1 Dispositivi FPGA

1.1.1 Struttura di base

I dispositivi Field Programmable Gate Array (FPGA) sono circuiti integrati digitali, caratterizzati da blocchi e interconnessioni programmabili. La struttura interna di un FPGA è composta da blocchi logici configurabili (Configurable Logic Block, o CLB) disposti a matrice, di cui quelli ai bordi vengono utilizzati anche per la gestione dei segnali di Input/Output, e da interconnessioni programmabili, con le quali è possibile connettere i vari blocchi tra loro.

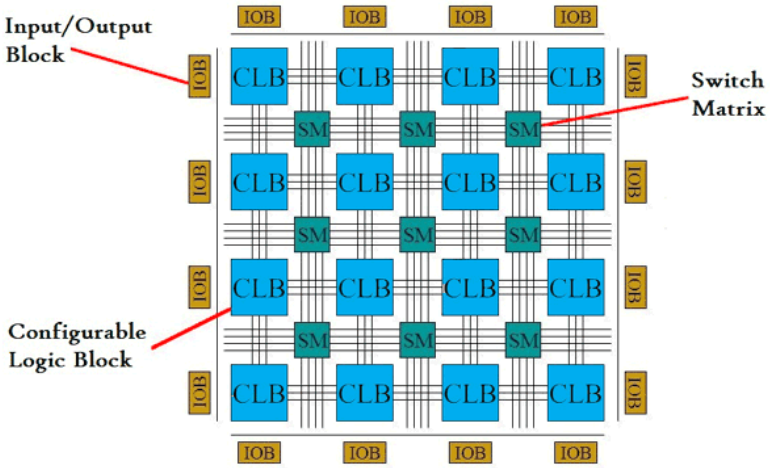


Figura 1.1: Struttura di principio di un FPGA [6]

Solitamente i CLB contengono da 2 (o più) slice, costituite a loro volta da diversi componenti logici o di memoria a seconda della piattaforma FPGA presa in considerazione. Gli elementi più comuni presenti nelle slice sono: Multiplexers, Look Up Tables (LUT) e flip-flops con cui si possono creare funzioni logiche/aritmetiche o anche memorizzare dati. Tuttavia le piattaforme FPGA moderne sono molto complesse e includono anche dispositivi più strutturati come processori, blocchi di memoria RAM, blocchi high-speed per la sezione di Input/Output e così via.

1.1.2 Vantaggi e applicazioni

Le piattaforme FPGA sono molto utilizzate poiché possono essere riprogrammate, andando quindi a modificare l'architettura interna tramite CLB e interconnessioni. Questo aspetto conferisce loro grande versatilità, che si contrappone agli ASIC (Application Specific Integrated Circuit), circuiti molto performanti, e soprattutto costosi, per una specifica applicazione, ma che non possono essere riutilizzati per sviluppare altre funzionalità. Gli FPGA vengono quindi

configurati tramite linguaggi di descrizione hardware come VHDL o Verilog, e vengono programmati tramite il caricamento di una stringa di bit, detto BitStream, generata a partire dalla configurazione specificata, attraverso opportuni software di sviluppo come Vivado. Inoltre a seconda delle applicazioni possono offrire altri vantaggi per quanto riguarda:

- **Risparmio energetico**: possono essere configurati per eseguire solo le operazioni strettamente necessarie allo svolgimento della funzione implementata, elminando tutti gli overhead in modo da aumentare sensibilmente l'efficienza.
- **Prestazioni**: è possibile organizzare la struttura FPGA in più blocchi distinti per manipolare i dati, aumentando così il parallelismo e di conseguenza la velocità d'esecuzione rispetto a una CPU che opera sequenzialmente.
- **Tempi di progettazione**: permettono una rapida prototipazione e verifica dell'effettivo funzionamento dell'hardware, riducendo sensibilmente i tempi di sviluppo ad esempio rispetto alla progettazione con ASIC.

I campi di applicazione per questi dispositivi, data la loro estrema flessibilità, sono innumerevoli: telecomunicazioni, automotive, difesa e aerospazio, industria biomedica. Si tratta di un settore in continua crescita per cui si prospettano ulteriori impieghi in futuro.

1.2 ISA RISC-V

1.2.1 Definizione di ISA

Una Instruction Set Architecture (ISA) è un insieme di istruzioni base che un processore, chiamato anche micro architettura, che la implementa può compiere. Una ISA costituisce sostanzialmente l'interfaccia uomo macchina, cioè l'insieme di operazioni di base disponibili per poter implementare un algoritmo. Si consideri inoltre il fatto che le ISA sono indipendenti dall'hardware che le implementano, rendendo così tutti i dispositivi che supportano una stessa ISA compatibili tra loro. Questo aspetto implica una lunga serie di vantaggi tra cui la riusabilità del software e la sua standardizzazione, una maggiore facilità di utilizzo e supporto. Questi sono solo alcuni dei vantaggi che hanno portato la RISC-V foundation a creare lo standard RISC-V.

1.2.2 RISC-V

RISC-V è uno standard di insieme di istruzioni, detto anche ISA (Instruction Set Architecture), di tipo open source basato sui principi RISC (Reduced Instruction Set Computer). In particolare RISC è un'architettura per microprocessori, le cui principali caratteristiche sono:

- Ottimizzazione per avere poche e semplici istruzioni, con conseguente implementazione hardware semplice e veloce.
- Tutte le istruzioni hanno un formato di pari lunghezza.
- Accesso alla memoria tramite solamente due istruzioni, load e store.
- Tutte le istruzioni possono essere eseguite con un singolo passaggio attraverso la pipeline di elaborazione.

L'ISA RISC-V ha un'architettura di tipo load-store con solamente 49 istruzioni di base. Si tratta di un set minimale di tipo modulare, ovvero modificabile tramite delle estensioni che possono essere aggiunte o rimosse a seconda delle proprie necessità. Lo standard RISC-V comprende in realtà diverse versioni di ISA, utilizzabili per soddisfare più scenari applicativi:

- **RV32I**: spazio di indirizzamento a 32 bit, adatta per sistemi semplici che richiedono un basso consumo di energia e memoria.
- **RV64I**: spazio di indirizzamento a 64 bit, utilizzata per sistemi più complessi della precedente.
- **RV32E**: sottoinsieme di istruzioni di RV32I, aggiunta in modo specifico per supportare piccoli microcontrollori.

- **RV128I**: spazio di indirizzamento a 128 bit, creato per implementazioni molto complesse, poco utilizzato.

Si noti che queste ISA hanno come base comune un particolare set di istruzioni per supportare l'elaborazione dei numeri interi e per il flusso di controllo denominato con "I", che risulta obbligatorio per qualsiasi implementazione RISC-V. Considerando invece le estensioni possono essere di tre tipi: standard, reserved e non standard. Quelle standard sono previste dalla RISC-V foundation, di cui si elencano in seguito le principali:

1. **M**: aggiunge le istruzioni per le moltiplicazioni e divisioni dei numeri interi.
2. **A**: istruzioni di lettura,scrittura e modifica di tipo atomic. Quest'ultime vengono eseguite senza essere interrotte o influenzate da altre operazioni in esecuzione, senza essere divise.
3. **F**: aggiunge registri e istruzioni necessari alla manipolazione dei numeri in virgola mobile a precisione singola (floating point).
4. **D**: espande i registri e rende l'architettura in grado di operare con numeri in virgola mobile a precisione doppia (double).
5. **C**: implementa istruzioni compresse a 16 bit per diminuire il consumo di energia e l'utilizzo di memoria.

Poi ci sono le estensioni non standard, ovvero non definite dalla RISC-V foundation, create da sviluppatori terzi e rese disponibili sulla rete. Esse possono aumentare le possibili funzionalità delle ISA RISC-V, ma che possono andare in conflitto con altre estensioni standard o non standard. Infine sono presenti le estensioni reserved, non definite dalla foundation e non ancora assegnate a una funzione specifica ma riservate per il futuro.

1.2.3 Il processore NEORV32

NEORV32 è un processore open source basato sull'architettura RISC-V, progettato per poter funzionare all'interno di sistemi più articolati oppure come microcontrollore stand alone. Si tratta di un sistema altamente configurabile, infatti dispone di una ampia gamma di periferiche selezionabili come memorie embedded, timer e interfacce seriali.

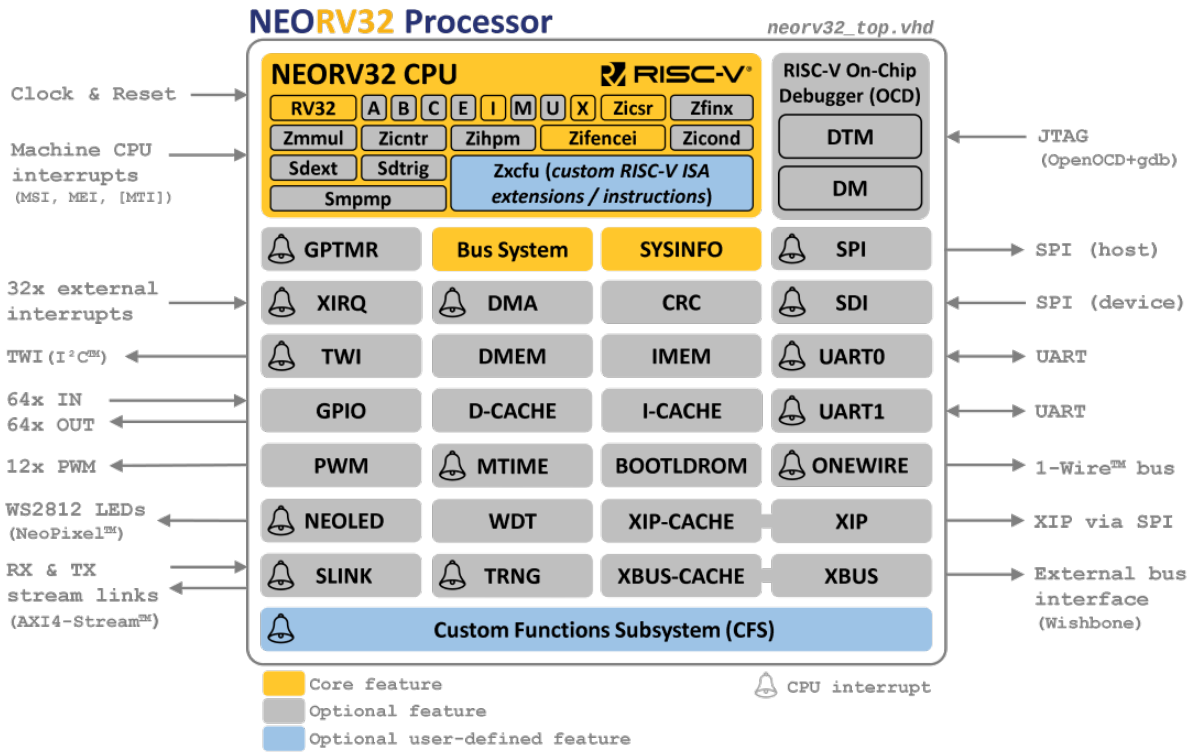


Figura 1.2: Schema a blocchi del processore NEORV32 [4]

Nello specifico si tratta di un processore soft-core, realizzato interamente in linguaggio VHDL adatto per implementazioni FPGA. Inoltre è un processore a 32 bit, che può quindi indirizzare fino a 4GB di memoria e supportare le ISA di RISC-V a 32 bit, come RV32I e RV32E. La peculiarità di NEORV32 è quella di avere, oltre che un buon trade-off tra performance e dimensioni, una alta execution safety. Infatti la CPU prevede precisi meccanismi hardware per qualsiasi malfunzionamento o situazione non prevista nell'esecuzione delle operazioni, in modo tale da ritrovarsi sempre in uno stato noto. Questo permette al sistema di avere sempre un comportamento prevedibile e ben definito, garantendo un alto livello di sicurezza concernente l'esecuzione delle operazioni. Per quanto riguarda invece la vera e propria CPU implementa una architettura di tipo pipeline multiciclo, in cui ogni istruzione viene eseguita come una serie di micro-operazioni. Per aumentare le performance è inserito un buffer di prefetch per disaccoppiare il back-end, cioè l'insieme di blocchi che svolgono l'esecuzione delle operazioni, dal

front-end che è il blocco di fetch delle istruzioni. In questo modo il front-end può eseguire il fetch mentre il back-end è in elaborazione e viceversa.

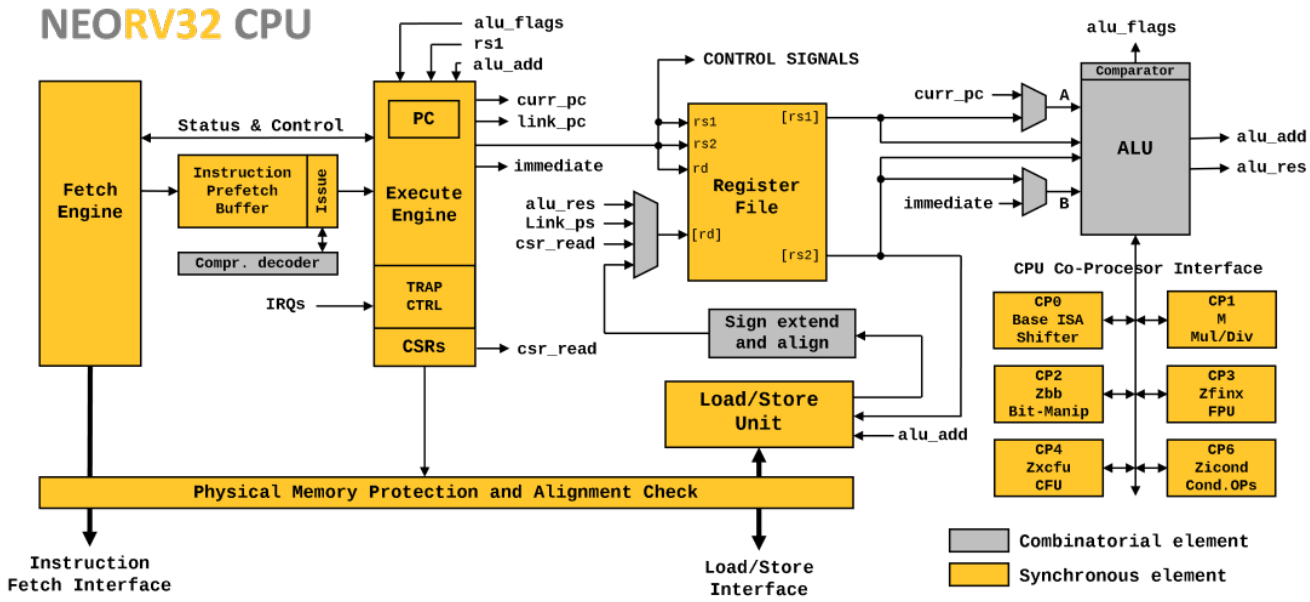


Figura 1.3: Architettura della CPU di NEORV32 [4]

In realtà questa architettura viene classificata come una via di mezzo tra una classica architettura di tipo pipeline a singolo ciclo e una multiciclo, dove ogni stadio viene scomposto in una serie di micro-operazioni. La scelta realizzativa di combinare questi due paradigmi è mirata all'aumento delle prestazioni, ma anche a una minore occupazione di spazio per l'hardware. Inoltre la CPU di NEORV32 segue i principi dell'architettura di Von Neumann e mette a disposizione due interfacce per il bus, differenti per l'accesso ai dati e quello alle istruzioni. I vari accessi vengono gestiti da un unico bus interno con un bus-switch, per poter selezionare tra dati o istruzioni, in cui l'accesso ai dati è a priorità maggiore.

1.3 Quantum Key Distribution (QKD)

1.3.1 Principio di funzionamento

La QKD è un metodo per la distribuzione sicura di chiavi segrete, sfruttando i meccanismi e i principi della meccanica quantistica, utili per lo scambio di informazioni tra due punti in modo criptato. La comunicazione per lo scambio delle chiavi avviene tra un trasmettitore, chiamato comunemente ”Alice”, e un ricevitore chiamato ”Bob”, i quali dispongono di due canali di comunicazione. Il primo canale (quantum channel) serve per trasmettere degli stati quantistici o qubit, con cui è possibile effettuare l'estrazione della chiave crittografica. Mentre nell'altro canale (public channel) le due parti, scambiandosi opportunamente delle informazioni relative agli stati quantistici ricevuti/trasmessi, possono rielaborare i dati scambiati nel quantum channel per generare una chiave condivisa sicura. La sicurezza della QKD si basa su principi fisici, che impediscono la decrittazione delle informazioni passate tramite quantum channel dall'esterno. Uno di questi è il No-cloning theorem [9], il quale proibisce l'esistenza di un meccanismo per creare una copia identica di uno stato quantistico. In altre parole, non è possibile distinguere tra stati quantistici non ortogonali tentando di misurarli, senza che gli stati di partenza vengano influenzati dalla misura stessa. Un'altro risultato fondamentale è sicuramente il collasso della funzione d'onda, di cui si rimanda una descrizione dettagliata in [10].Una comune applicazione in questo senso riguarda il passaggio di stati quantistici tramite quantum channel, generalmente in fibra ottica o spazio libero, utilizzando fasci di fotoni opportunamente polarizzati. Organizzando il sistema in questo modo un eventuale attaccante, denominato in letteratura ”Eve”, non è in grado di estrarre informazioni utili dal quantum channel. Non conoscendo a priori la polarizzazione di ciascun fotone, Eve deve scegliere casualmente la base di polarizzazione per la misura, con la possibilità di compromettere l'integrità del dato. Approfittando poi dell'influenza di Eve sugli stati quantistici da essa misurati, Alice e Bob quindi possono capire in post-processing quali informazioni sono state manipolate dell'attaccante, rimuovendole dalla loro chiave finale. Le precedenti considerazioni rendono questo tipo di comunicazione particolarmente interessante per la crittografia, consentendo una sicurezza di livello Information Theoretic (ITS, Information Theoretic Security), il livello più alto in assoluto.

Nello schema sotto riportato si evidenziano tutte le componenti di un sistema QKD generico, ovviamente si tratta di uno schema di principio che non tiene conto delle caratteristiche e peculiarità presenti in applicazioni specifiche:

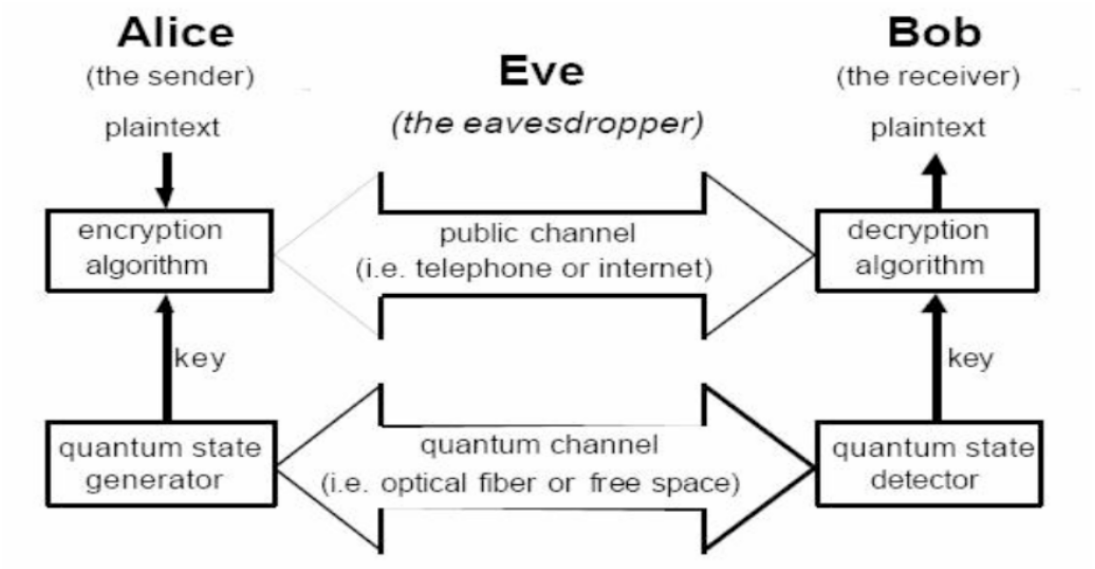


Figura 1.4: Schema di principio Quantum Key Distribution [8]

1.3.2 Il protocollo BB84

Il primo protocollo di comunicazione per QKD è stato inventato nel 1984 da Charles Bennett e Gilles Brassard e prevede l'utilizzo di quattro stati quantistici, chiamati anche qubit. Precisamente si dividono in due coppie di stati che formano le corrispettive basi, la base rettilinea {H,V} e la base diagonale {D,A}:

- **Base {H,V}**: composta dalla polarizzazione Verticale (V) a cui viene associato il valore 1, e da quella Orizzontale (H) a cui viene associato il valore 0.
- **Base {D,A}**: composta dalla polarizzazione Diagonale (D) e antidiagonale (A) a cui vengono associate rispettivamente il valore 1 e 0.

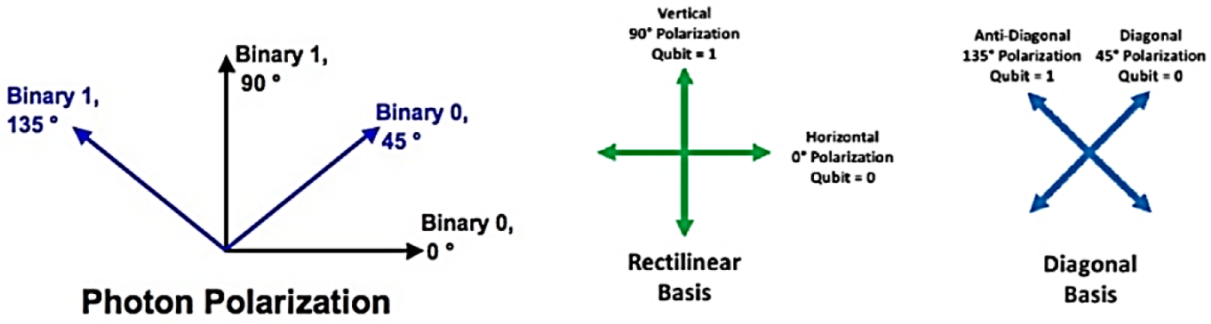


Figura 1.5: Basi e polarizzazioni protocollo BB84 [7]

Inizialmente Alice invia un fascio di fotoni polarizzati casualmente, scegliendo in maniera random la base e il dato da trasmettere per ogni fotone tramite il quantum channel. A questo punto Bob, non sapendo quali polarizzazioni sono state effettuate in partenza, sceglie casualmente la base da utilizzare per poter ricevere ogni singolo fotone, con il risultato di avere il 50% di probabilità d'errore. Secondo questa procedura una buona parte dell'informazione trasmessa verrà quindi persa, in primo luogo per l'errore in ricezione causato da Bob. In secondo luogo possono esserci dei bit corrotti a causa delle interferenze causate da un possibile attaccante, oppure in seguito a dei disturbi relativi all'ambiente esterno e alla natura del mezzo di comunicazione. Fatto ciò, il ricevitore comunica i dati ricevuti tramite il public channel, ovvero un canale di comunicazione classico. Le due parti confrontano poi quali dati sono stati ricevuti correttamente, senza dichiararli esplicitamente, scartando gli altri. Una volta che Bob comunica i dati ricevuti con Alice viene sfruttata una serie di algoritmi per eliminare i bit errati, eliminare i possibili bit rintracciati da Eve e creare una chiave affidabile. Il processo di rielaborazione computazionale dei dati nel suo insieme viene chiamato post-processing. Si sottolinea però che, una volta analizzati i dati ricevuti da Bob, viene fatta una quantificazione del numero di dati validi rispetto al

numero di dati grezzi forniti da Alice. Se la percentuale va al di sotto di una certa soglia, il protocollo riconosce la situazione come un malfunzionamento della rete, oppure come l'intervento di un'entità esterna che sottrae informazioni. Questa soglia in realtà viene stimata in funzione della tipologia di attacco adottato da Eve per estrarre i qubit. L'esempio più semplice di attacco è detto intercept-resend e in questa casistica la soglia ottima ricavata ammonta al 12.5%. Tuttavia esistono molteplici tipi di attacchi molto efficaci, che possono far scendere questa percentuale anche fino all'11%. Nel caso in cui questa percentuale sia sensibilmente più bassa di quella attesa, la chiave non viene generata affatto, poiché considerata inaffidabile ed eventualmente viene riavviato il protocollo. La figura 1.6 in seguito esemplifica il funzionamento del protocollo appena descritto.

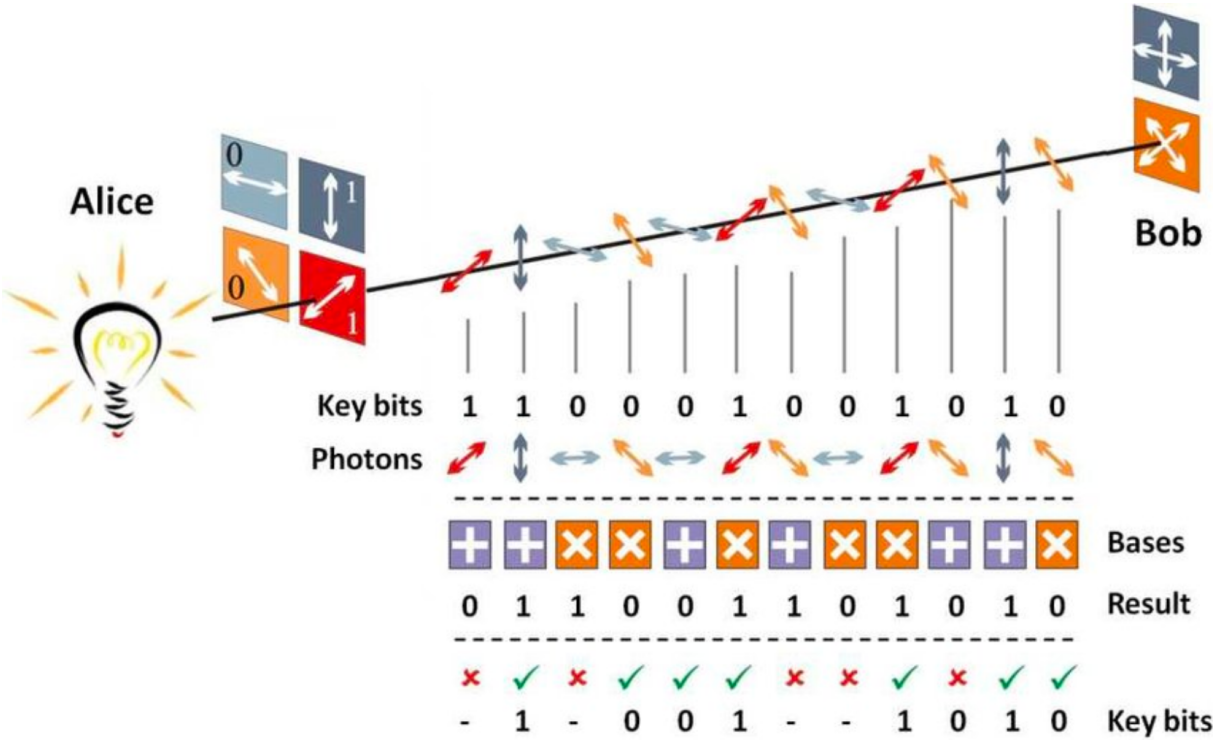


Figura 1.6: Il protocollo BB84 [5]

Capitolo 2

Design e implementazione

2.1 Schede utilizzate

Si riportano in seguito le caratteristiche delle schede che compongono il setup utilizzato per lo svolgimento dei test. La prima da citare è la board PYNQ-Z2 di Xilinx, basata su ZYNQ XC7Z020-1CLG400C SoC usata per lo sviluppo FPGA, su cui è stato configurato il processore NEORV32. Per questo scopo la scheda dispone di una sezione di logica programmabile (PL o Programmable Logic) tramite porta JTAG, Quad-SPI flash o MicroSD. In particolare la PL si organizza in:

- 630 KB di RAM
- 13,300 slices, ciascuna con 6 LUTs e 8 flip-flops
- 220 DSP slices
- Convertitore analogico digitale Xilinx (XADC)

Inoltre nella board è presente un processore ARM® Cortex®-A9 dual-core, chiamato per semplicità PS (Processor System) caratterizzato da:

- 512 MB DDR3 RAM
- 16 MB Quad-SPI Flash
- Frequenza di clock pari a 650 MHz

Sono disponibili anche molti altri connettori come HDMI, Pmod e Ethernet oltre ad altri dipositivi accessori come 4 pulsanti, 2 switch e 6 Led. Tramite USB viene implementato anche un modulo UART collegato al PS e non accessibile dalla parte FPGA della scheda. In aggiunta, per necessità del setup descritte più avanti, è stato fatto uso di un modulo Pmod USBUART per

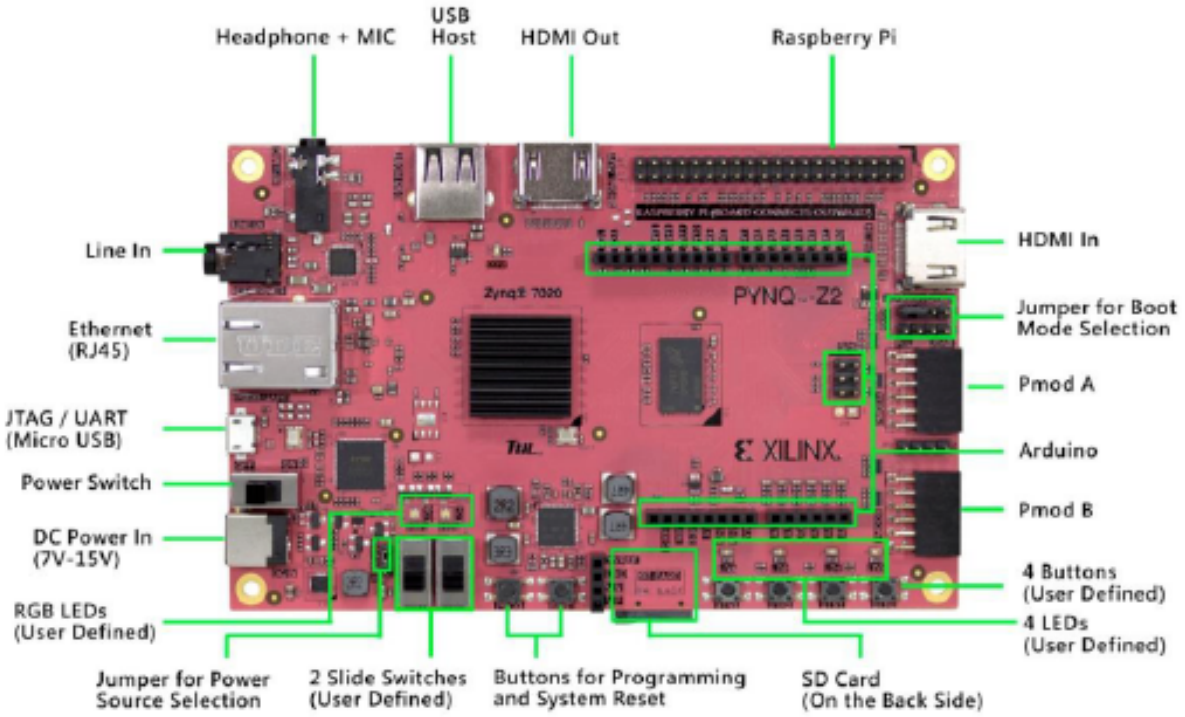


Figura 2.1: Scheda FPGA di Xilinx PYNQ-Z2 [14]

la comunicazione. La scheda pmod USBUART della DIGILENT sostituisce sostanzialmente la combinazione tra un modulo Pmod RS232 per la comunicazione seriale e un adattatore seriale-USB. Consente due modalità di funzionamento selezionabili spostando lo switch presente nella parte superiore: LCL se la board collegata tramite pmod al modulo è alimentata autonomamente, SYS per prelevare l'alimentazione direttamente dalla porta USB.

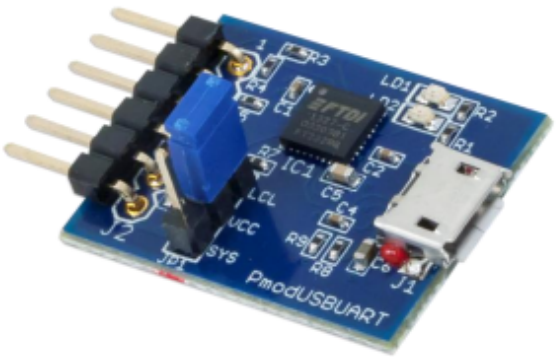


Figura 2.2: Modulo pmod USBUART DIGILENT [15]

2.2 Sistema di sviluppo

2.2.1 Configurazione hardware NEORV32

Per riuscire a implementare il processore NEORV32 nella board PYNQ-Z2 sono stati predisposti innanzitutto gli strumenti software, in modo da poter generare il bitstream per la configurazione del processore. A questo scopo è stata installata la toolchain relativa a NEORV32 (supporto ISA RV32E/RV32I), con cui è possibile compilare programmi in linguaggio C e creare un corrispondente file per l'implementazione in NEORV32. Successivamente è stato inserito in un progetto VHDL il processore tramite i file sorgente open source in una impostazione di base. Su questa base in VHDL poi è stata instanziata una top entity tramite software Vivado, per poter manipolare l'architettura e le risorse del processore in base all'algoritmo da testare. Dopodichè sono stati aggiunti i file di constraint al progetto per specificare la frequenza di clock (per la board PYNQ-Z2 sono imposti 125 MHz) e i collegamenti alle varie periferiche. Nel dettaglio la struttura hardware prevede l'impiego di un pulsante di reset per riavviare il proessore nel caso di malfunzionamenti, oltre che una sezione di IN/OUT composta da 8 porte digitali su cui in questo caso sono stati collegati gli 8 led disponibili nella board, similmente alla configurazione standard.

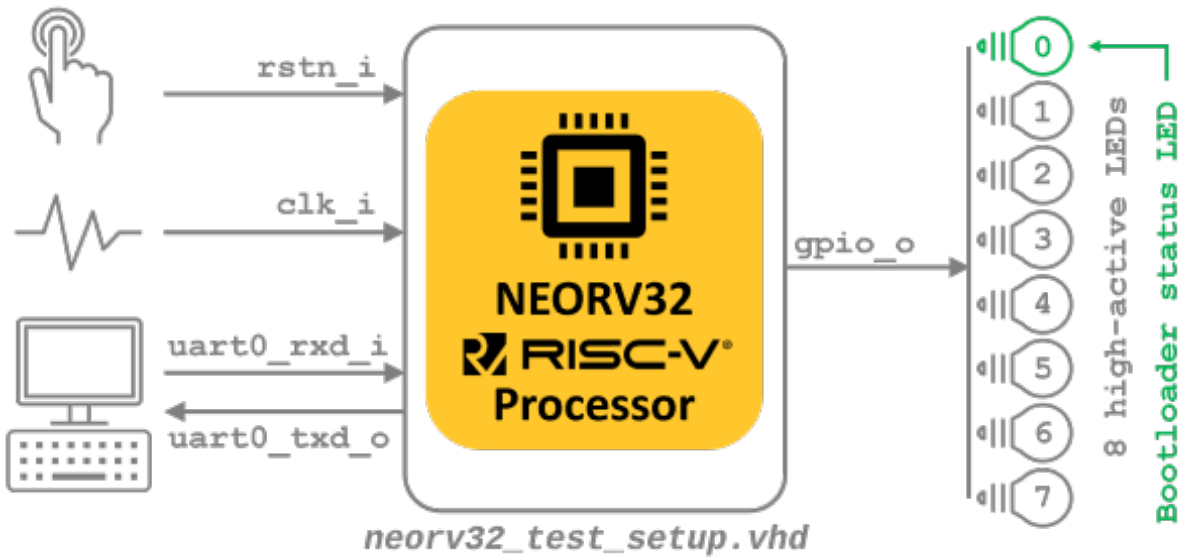


Figura 2.3: Setup hardware di base NEORV32 [4]

Inoltre è prevista anche la comunicazione seriale con il pc tramite modulo UART abilitato e implementato dal NEORV32, sfruttando però le porte pmod della board, poichè il modulo UART presente via USB è accessibile solo alla PS e non alla PL. Sfruttando quindi una porta

pmod in combinazione al modulo pmodUSB UART si garantisce il trasferimento dei dati in seriale tramite cavo USB.

2.2.2 Caricamento del software e Bitstream

Per quanto riguarda la programmazione vera e propria del processore, essa può avvenire essenzialmente in due modi, sfruttando o meno il bootloader interno dell'architettura, attivabile tramite top entity. Le due modalità di caricamento del programma in linguaggio C sono:

- **Indirect Boot configuration**: richiede l'abilitazione del bootloader interno e del modulo UART per poter caricare il programma. Compilando quest'ultimo tramite la toolchain viene generato grazie al linker un file eseguibile, che sarà poi il file da inviare al bootloader. Utilizzando il modulo UART è possibile comunicare tramite riga di comando con il bootloader ed eseguire l'upload del programma nella memoria boot ROM, specificando il file eseguibile come input. Questo processo avviene in modo indipendente dal trasferimento del bitstream, che deve essere svolto a priori, per consentire al bootloader e alla boot ROM di essere concretamente inizializzati.

- **Direct Boot configuration**: in questa modalità non è strettamente necessario includere il bootloader e il modulo UART nella configurazione per il caricamento del programma. Tuttavia è indispensabile abilitare, tramite le generics della top entity, una sezione di memoria interna del processore mirata a questa funzionalità (MEM_INT_IMEM). Sempre grazie alla toolchain è possibile, opzionalmente al caso appena citato, generare a partire dal codice C un file con estensione VHD, che rappresenta il codice bnario estrapolato dal linker. In buona sostanza consiste in un file contenente un array di valori per la scrittura diretta nella memoria MEM_INT_IMEM, chiamato Application Image. Il file VHD deve essere quindi importato nel progetto Vivado e viene generato il bitstream per la configurazione. Al contrario del caso precedente le impostazioni hardware e software del processore avvengono simultaneamente durante il trasferimento del bitstream.

Negli esperimenti svolti è stato scelto il Direct Boot configuration come metodo di approccio per l'upload del software. In particolare è stato fatto uso della toolchain in un sistema operativo Linux (obbligatorio per questo tipo di toolchain, non direttamente disponibili su Windows), gestendo la generazione del bitstream e del setup per la board su ambiente Windows, tramite il software Vivado e i file estrapolati dalla toolchain.

2.2.3 Setup completo e step di utilizzo

In merito agli aspetti relativi alle interconnessioni tra le varie componenti, per chiarezza espositiva, in figura 2.4 è raffigurato un semplice schema a blocchi del setup completo, adoperato nello svolgimento dei test.

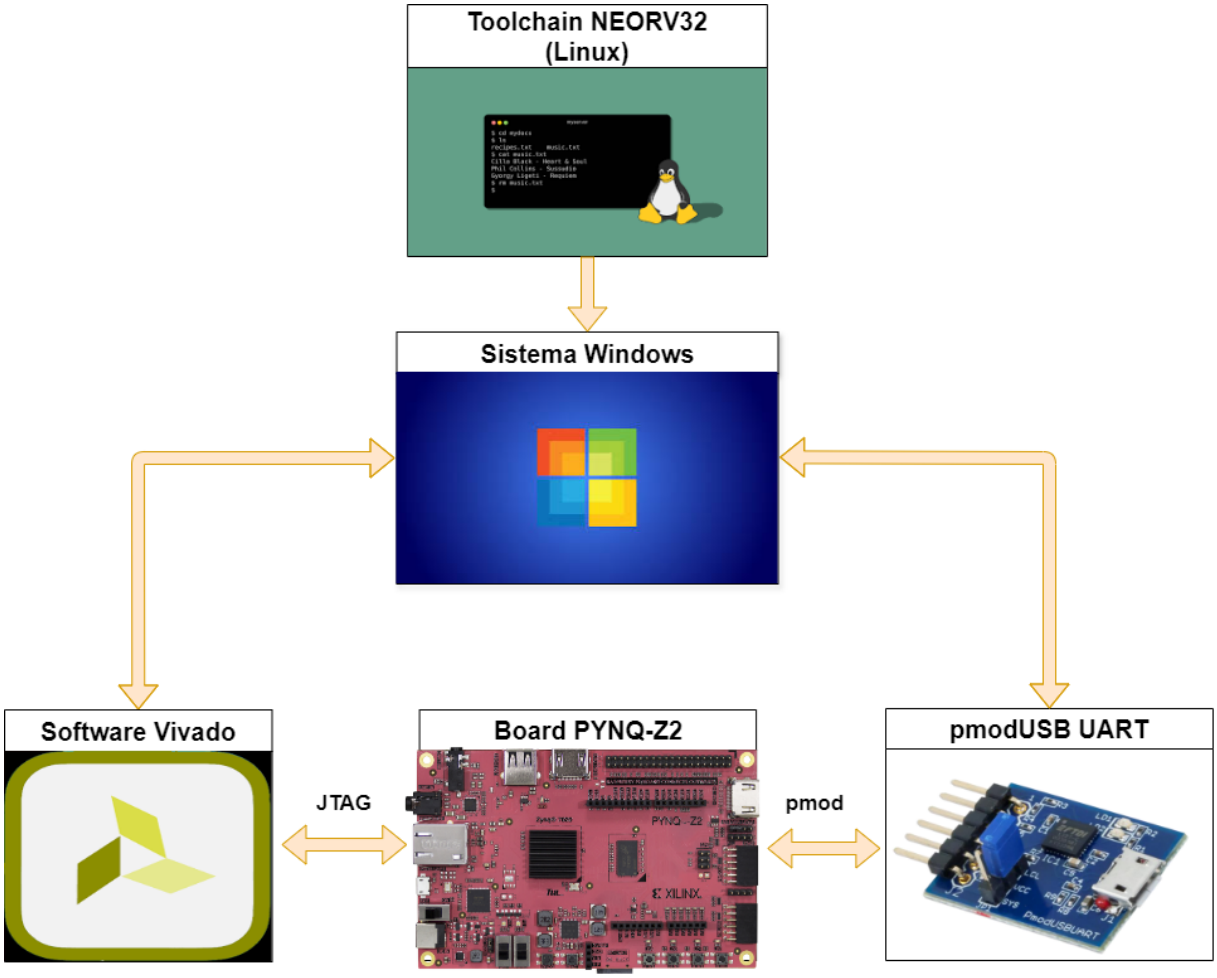


Figura 2.4: Schema a blocchi del setup utilizzato

Riassumendo la procedura per l'impostazione hardware e software di NEORV32 per l'esecuzione di un algoritmo, in linea di principio gli step che sono stati seguiti sono:

1. Configurazione hardware su programma Vivado
2. Compilazione dell'algoritmo tramite toolchain e creazione dell'Application Image
3. Trasferimento dell'Application Image nel progetto Vivado
4. Caricamento del BitStream tramite porta JTAG della board

2.3 Algoritmo di Knuth-Yao

2.3.1 Ruolo nella QKD

Prima di descrivere dettagliatemene in cosa consiste l'algoritmo di Knuth-Yao, è opportuno andare ad evidenziare quale funzione esso ricopre nella QKD. Il protocollo BB84, sebbene sia molto sicuro, presenta alcune criticità tra cui l'efficienza nella generazione della chiave tra Alice e Bob. Scegliendo casualmente la base di polarizzazione in ricezione e trasmissione si ha una perdita dei dati grezzi di almeno il 50%, che non potranno essere usati per la creazione della chiave, abbassando di molto il key rate in uscita. Questa uniformità nella scelta delle basi è stata rimossa da Hoi-Kwong Lo, H. F. Chau e M. Ardehali [12] quando nel 2004 proposero delle modifiche rispetto al protocollo BB84 che permettono teoricamente di raggiungere asintoticamente una efficienza prossima al 100%. Le modifiche in questione sono essenzialmente due: la prima è quella di una scelta non uniforme delle basi di polarizzazione per incrementare l'efficienza, ovvero viene introdotto un bias sulle probabilità di scelta delle basi. Infatti introdurre una base preferenziale per Alice e Bob consente ai due di avere statisticamente meno perdite di dati dovute alla scelta errata della base in ricezione. La seconda riguarda l'aggiunta di un'analisi dei dati e delle probabilità molto più complessa nella parte di post-processing, che si suddivide a seconda della base scelta, per garantire la sicurezza sui dati trasferiti nonostante le modifiche apportate. Questa innovazione ha provocato il bisogno di avere un meccanismo per la creazione del bias di probabilità per la scelta della base, attraverso appositi algoritmi di biasing. Knuth e Yao hanno contributo fortemente a questo tipo di algoritmi sia dal punto di vista teorico che implementativo, ponendo le basi dei successivi sviluppi per gli algoritmi di biasing.

2.3.2 Formalizzazione ed esempi

Il lavoro svolto da Donald Knuth e Andrew Yao [17] riguarda la generazione di numeri casuali con probabilità non uniforme, concentrandosi in particolar modo sugli algoritmi di sampling, sulla loro modellizzazione e rappresentazione mediante l'uso di alberi binari e sull'analisi prestazionale di quest'ultimi. In generale un algoritmo di sampling riceve in input un flusso di bit generati in modo totalmente casuale, per generare in uscita una sequenza di numeri interi, ciascuno con probabilità $p_i = a_i/m$ $(i = 1, \dots n)$ arbitraria. Viene supposto quindi che esista una operazione primitiva, chiamata FLIP, in grado di generare un bit $\mathcal{B} \in \{0, 1\}$ perfettamente casuale, cioè con $p_0 = p_1 = 1/2$ per garantire il flusso di bit in ingresso al sampler (random bit model). Knuth e Yao forniscono un metodo per poter rappresentare un qualsiasi sampler mediante un albero binario radicato, chiamato in letteratura DDG tree (Discrete Distribution Generating) dotato di due proprietà: ogni nodo interno ha esattamente due figli, ad ogni nodo foglia viene etichettato uno dei numeri da produrre in uscita $\{1, \dots, n\}$, mentre i nodi interni

non posseggono etichette. La costruzione del DDG tree da loro proposta (che si dimostrerà essere ottima in termini di efficienza) avviene nel seguente modo: si supponga di avere una certa distribuzione di probabilità discreta ed ogni probabilità sia associata a un numero i. Per ogni valore di probabilità associata $p_i \in (0,1)$ a ciascun numero si consideri la sua rappresentazione in codice binario $(0.p_{i1}p_{i2}\ldots)_2$. Allora il DDG Tree per ogni suo livello j deve contenere esattamente una foglia etichettata con il numero i se $p_{ij} = 1$, altrimenti nessuna. La costruzione del DDG tree si complica nel momento in cui le probabilità non possono essere rappresentate con un numero finito di bit, ma con una codifica periodica che porterebbe alla costruzione di un albero di altezza infinita. Tuttavia, anche se le probabilità sono periodiche, è possibile trovare una rappresentazione con un numero di nodi finito. Per semplicità si può chiarire la costruzione dei DDG tree tramite due esempi, uno che include un insieme di probabilità diadiche e l'altro un insieme di probabilità con rappresentazione binaria periodica.

Esempio 1 Sia una distribuzione $p = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ associata all'insieme di numeri da generare $S = (1,2,3,4,5)$. La rappresentazione delle probabilità è quindi $p_1 = p_2 = p_3 = (0.010)_2$ e $p_4 = p_5 = (0.001)_2$, essendo tutte diadiche il DDG tree può essere costruito direttamente:

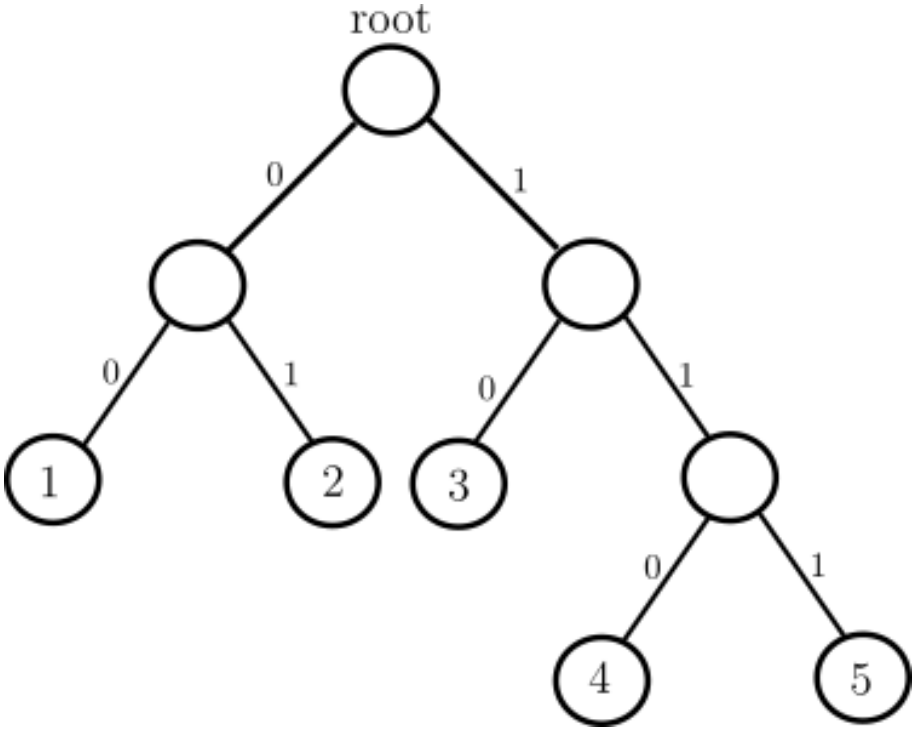


Figura 2.5: DDG Tree con probabilità diadiche

Esempio 2 Sia una distribuzione $p = (\frac{3}{10}, \frac{7}{10})$ associata all'insieme di numeri $S = (1,2)$, la rappresentazione delle probabilità è $p_1 = (0.0\overline{1001})_2$ e $p_2 = (0.1\overline{0110})_2$. Non avendo probabilità diadiche non è possibile rappresentare direttamente il DDG Tree con un numero finito di nodi,

però è possibile utilizzare dei rami di ritorno (vedi ramo in rosso in figura 2.6) per tenere conto della periodicità delle decodifiche:

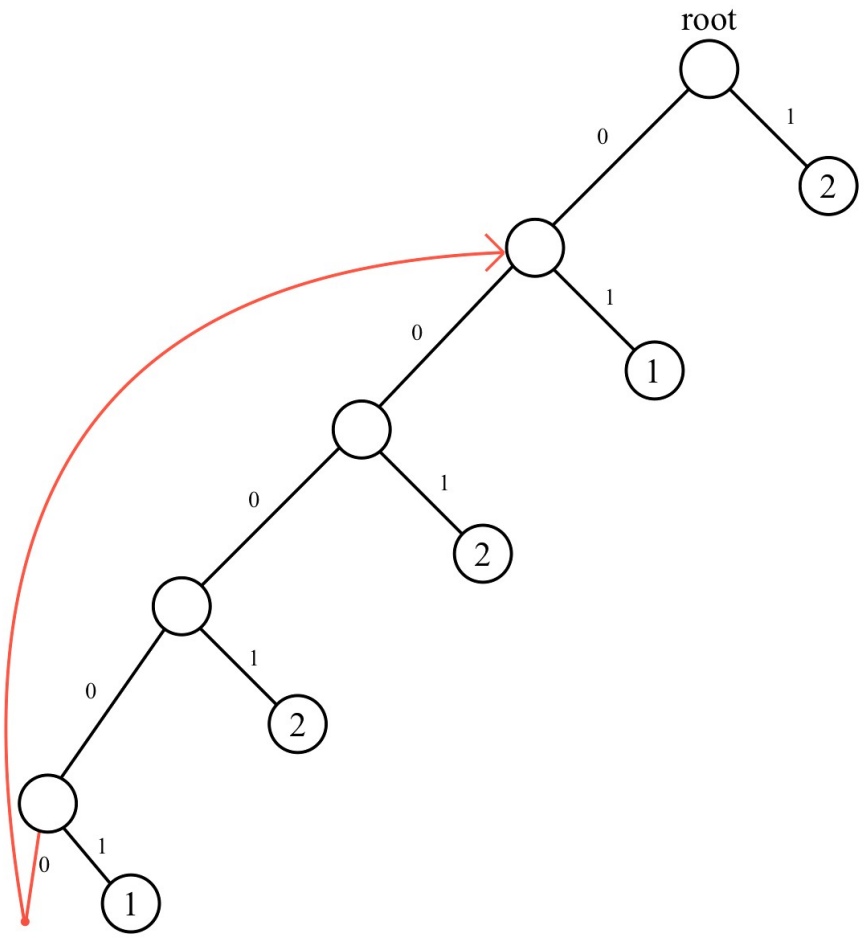


Figura 2.6: DDG Tree con probabilità non diadiche

Una volta costruito il DDG Tree l'algoritmo di Knuth-Yao funziona come segue: partendo dal nodo radice si ottiene tramite FLIP un bit $\mathcal{B}$, se $\mathcal{B} = 0$ si procede avanzando verso il figlio sinistro del nodo, invece se $\mathcal{B} = 1$ si avanza verso il figlio destro. Se il nuovo nodo raggiunto è una foglia l'algoritmo restituisce il numero con cui è etichettata quella foglia, altrimenti si ripete l'algoritmo estraendo un nuovo bit dall'operazione di FLIP. Dato un nodo appartenente all'albero $x \in T$, indicando con $d(x)$ il suo livello e con $l(x)$ la sua etichetta, attribuendo convenzionalmente il livello 0 alla radice e l'etichetta 0 ai nodi interni, le probabilità ottenute reiterando questo processo possono essere scritte in relazione al livello delle foglie:

$$p_i = P[T\ restituisca\ i] = \sum_{x \mid l(x)=i} 2^{-d(x)} \quad \forall i \in (1, \ldots, n)$$

Secondo queste convenzioni il valore atteso di FLIP L_T usati per generare mediamente un numero è uguale al valore atteso della profondità delle foglie:

$$E[L_T] = \sum_{x \mid l(x)>0} d(x)\, 2^{-d(x)}$$

Il successivo teorema fornisce dei bound stretti relativi al minimo valore atteso di bit consumati L_T da un sampler qualsiasi, data una distribuzione di probabilità p. Inoltre stabilisce che il DDG Tree costruito nel modo proposto è ottimo in termini di efficienza, dove l'efficienza per un sampler è associata alla quantità di bit (FLIP) consumati per la generazione dei numeri casuali a probabilità non uniforme.

2.3.3 Teorema di Knuth-Yao

Si introduce preliminarmente la definizione di Entropia di Shannon, necessaria per chiarificare l'enunciato del teorema:

Definizione Entropia di Shannon
Data una variabile aleatoria discreta X con alfabeto $A_x = \{a_1, \ldots, a_n\}$, per ogni evento a appartenente a X si definisce la funzione informazione come

$$i_X : A_X \to \Re, \quad i_X(a) = \log_2 \frac{1}{p_X(a)}$$

L'entropia di Shannon di una variabile aleatoria discreta X è definita come il valore atteso dell' informazione:

$$H(X) = E[i_X(X)] = \sum_{a \in A_X} p_X(a) i_X(a) = \sum_{a \in A_X} p_X(a) \log_2 \frac{1}{p_X(a)}.$$

Enunciato *Sia $p := (p_1, \ldots, p_n)$ associata a una variabile aleatoria X, dove $n > 1$. Ogni algoritmo di sampling con DDG Tree T e distribuzione in uscita p il cui valore atteso di bit in input è minimo (tra tutti i T' la cui distribuzione in uscita è uguale a p) soddisfa $H(X) \leq E[L_T] < H(X) + 2$, dove questi bound sono i più stretti possibile. In aggiunta, T contiene esattamente un nodo foglia con etichetta i al livello j se e solo se $p_{ij} = 1$, dove $(0.p_{i1}p_{i2}\ldots)_2$ è la codifica binaria dei p_i.*

La doppia disuaglianza del teorema può essere dimostrata in due parti, una per l'upper bound e una per il lower bound utilizzando il seguente risultato:

Lemma Sia Y una variabile aleatoria e sia f una funzione, allora vale la seguente proprietà sull'entropia [18]:

$$H(f(Y)) \leq H(Y).$$

Dimostrazione lower bound $E[L_T] \geq H(X)$

Dato un albero T, sia Y la variabile aleatoria con distribuzione:

$$P = \{P(f) : \text{ f e' foglia di T }\}$$

dove $P(f) = 2^{-d(f)}$ con $d(f)$ la profondità della foglia f. Si può notare che:

$$H(Y) = \sum_{f:\text{f è foglia di T}} 2^{-d(f)} log \frac{1}{2^{-d(f)}} = \sum_{f:\text{f è foglia di T}} d(f) 2^{-d(f)} = E[L_T]$$

Si osserva che la variabile aleatoria X è ottenibile mediante l'applicazione di una funzione f alla variabile Y (la funzione f può mappare ad esempio uno o più valori di Y, in un qualche singolo valore di X) dal Lemma otteniamo:

$$E[L_T] = H(Y) \geq H(f(Y)) = H(X).$$

□

Dimostrazione upper bound $E[L_T] < H(X) + 2$

Data $p = (p_1, \ldots, p_n)$ la distribuzione di probabilità di X si può esprimere ogni p_i come somma di potenze di 1/2. Precisamente scrivendo

$$p_i = p_i^{(1)} + p_i^{(2} + \ldots \quad (1)$$

dove ogni termine $p_i^{(j)}$ o è pari a 0 oppure una potenza di 1/2. Usando tutti i termini p_i^j per ogni i e per ogni j, si può scrivere una nuova distribuzione di probabilità data da:

$$(p_1^{(1)}, p_1^{(2)}, \ldots, p_2^{(1)}, p_2^{(2)}, \ldots, p_n^{(1)}, p_n^{(2)}, \ldots) \quad (2)$$

si costruisca quindi un albero T che ha una foglia associata ad ogni una delle potenze di 1/2 che compare in (1), cioè se $p_i^{(j)} = 2^{-k_{ij}}$, per qualche intero k_{ij}, allora l'albero T avrà una foglia alla

profondità $k_i j$ e vale che

$$1 = \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} \sum_{j \geq 1} p_i^{(j)} = \sum_{i=1}^{n} \sum_{j \geq 1 : p_i^{(j)} > 0} 2^{-k_{ij}}$$

quindi un tale albero esiste a causa della diseguaglianza di Kraft. Sia Y una distribuzione di probabilità data dalla equazione (2) ed essendo questa diadica si ha che $E[L_T] = H(Y)$. La v.a. che si vuole generare è funzione di Y, ovvero per ottenere X da Y si etichettano tutte le foglie aventi probabilità $p_i^{(j)}$ che appaiono nell'espansione (1) di p_i con lo stesso valore x_1 della v.a. X, ovvero con il valore x_i che ha probabilità di occorrere p_i. Si osserva che

$$H(Y) = -\sum_{i=1}^{n} \sum_{j \geq 1} p_i^{(j)} \log p_i^{(j)} = \sum_{i=1}^{n} \sum_{j : p_i^{(j)} > 0} j 2^{-j}$$

In quanto $p_i^{(j)}$ è 0 oppure una potenza di 1/2. Sia

$$T_i = \sum_{j : p_i^{(j)} > 0} j 2^{-j} \quad \text{in modo che} \quad H(Y) = \sum_{i=1}^{n} T_i = E[L_T].$$

Notando che per ogni probabilità p_i, possiamo sempre trovare un numero naturale n_i tale che $2^{-(n_i - 1)} > p_i \geq 2^{-n_i}$, ovvero $n_i - 1 < -log p_i \leq n_i$, per cui $p_i^{(j)} > 0$ solo se $j \geq n_i$, e quindi si può scrivere T_i come:

$$T_i = \sum_{j : j \geq n_i, p_i^{(j)} > 0} j 2^{-j}.$$

Calcolando poi la differenza tra T_i e $-p_i \log p_i + 2p_i$ si ottiene:

$$\begin{aligned}
&T_i + p_i \log p_i - 2p_i < T_i - p_i(n_i - 1) - 2p_i = T_i - (n_i - 1 + 2)p_i \\
&= \sum_{j : j \geq n_i, p_i^{(j)} > 0} j 2^{-j} - (n_i + 1) \sum_{j : j \geq n_i, p_i^{(j)} > 0} 2^{-j} = \sum_{j : j \geq n_i, p_i^{(j)} > 0} (j - n_i - 1) 2^{-j} \\
&= -2^{-n_i} + 0 + \sum_{j : j \geq n_i + 2, p_i^{(j)} > 0} (j - n_i - 1) 2^{-j} \\
&= -2^{-n_i} + \sum_{k : k \geq 1, p_i^{(k + n_i + 1)} > 0} k 2^{-(k + n_i + 1)} \quad \text{(cambio di variabili nella somma)} \\
&\leq -2^{-n_i} + \sum_{k : k \geq 1} k 2^{-(k + n_i + 1)} \quad \text{(estendendo la somma su tutti i termini)} \\
&= -2^{-n_i} + 2^{-(n_i + 1)} \sum_{k : k \geq 1} k 2^{-k}
\end{aligned}$$

$= -2^{-n_i} + 2^{-(n_i+1)} \times 2 = 0.$ (essendo $\sum_{k:k\geq 1} k2^{-k} = 2$)

Mettendo insieme tutti i risultati ottenuti si mostra infine che

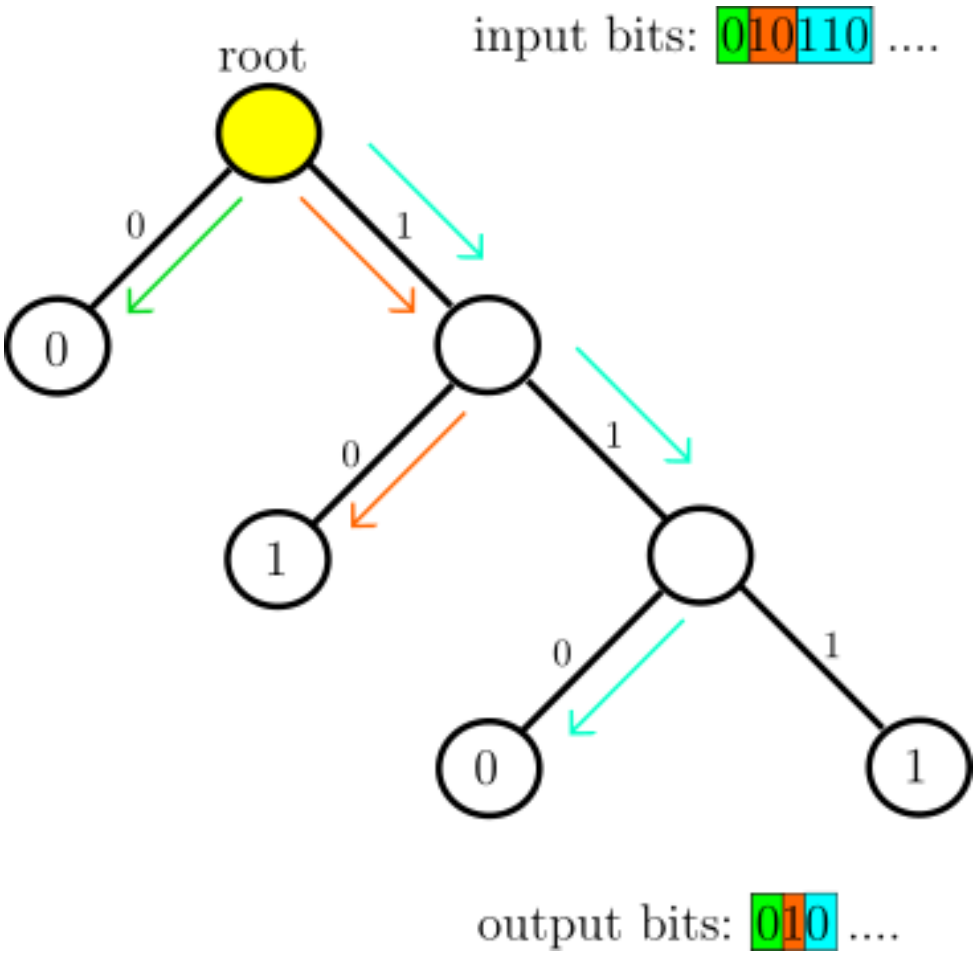
$$E[L_T] = \sum_{i=1}^{n} T_i < -\sum_{i=1}^{n}(p_i \log p_i - 2p_i) = -\sum_{i=1}^{n} p_i \log p_i + 2\sum_{i=1}^{n} p_i = H(x) + 2$$

□

2.3.4 Implementazione in NEORV32

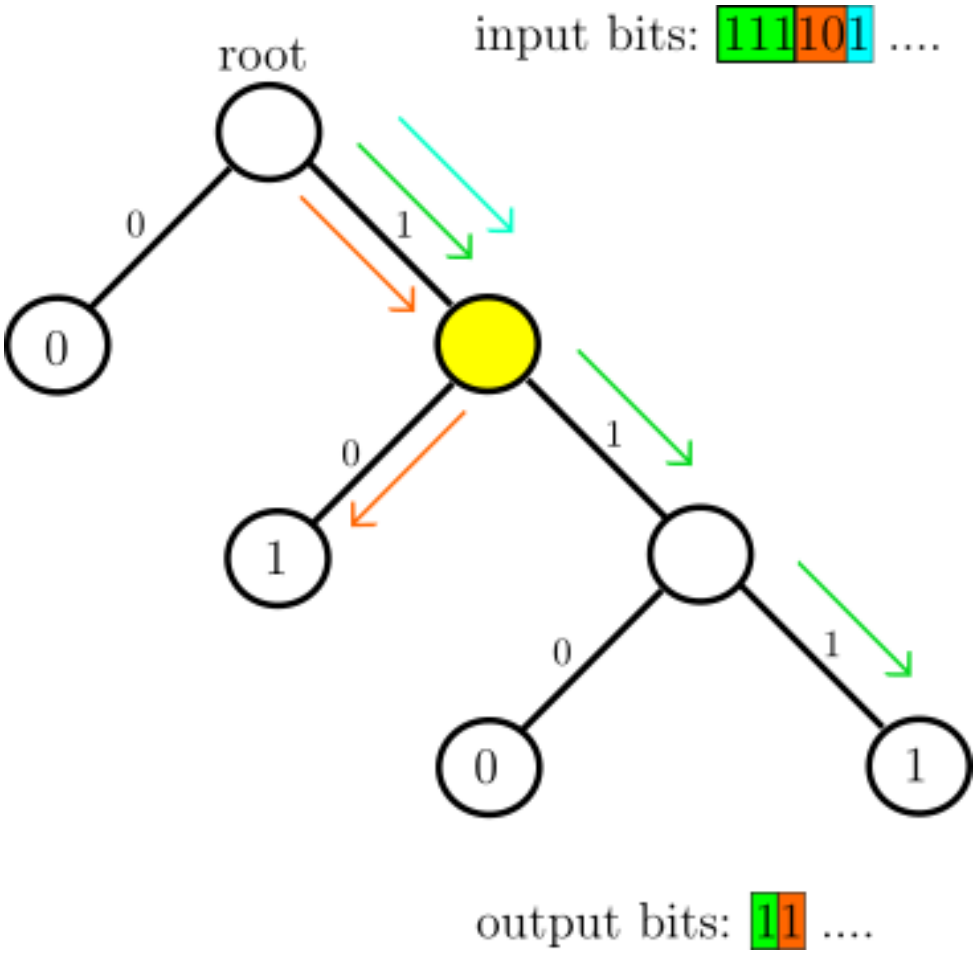
Nella piattaforma PYNQ-Z2 è stato implementato un algoritmo di biasing in codice C che rispecchia il funzionamento descritto da Knuth e Yao, sebbene non sia stato strutturato esattamente come la teoria consiglia, ovvero utilizzando il DDG Tree. Questo perchè la costruzione di una struttura dati come un albero binario in un codice C e la sua gestione, comporterebbe un forte rallentamento in termini di velocità d'esecuzione e la diminuzione del bitrate in uscita, con delle conseguenti prestazioni non adatte a una comunicazione di tipo QKD. Pertanto si è fatto uso di un algoritmo high performance personalizzato, che non utilizza esplicitamente una struttura dati ad albero, ideato in modo tale da migliorare il più possibile le prestazioni. Nello specifico sono state usate delle look-up tables in cui ognuna rappresenta un nodo del DDG Tree dell'algoritmo di Knuth-Yao. Ciascuna look-up table possiede due campi, il primo campo contiene i bit in uscita dell'algoritmo, nel dettaglio un bit per le foglie mentre un campo vuoto per i nodi interni. Il secondo invece funge da prefisso per lo start, cioè imposta il nodo di partenza per l'iterazione successiva. L'algoritmo ciclicamente analizza un numero finito di bit in input e mappa quest'ultimi nel corrispondente nodo del DDG Tree. Se la sequenza di bit analizzata porta ad una foglia, viene prelevato il bit in uscita da essa contenuto, impostando la radice come nodo di start. Altrimenti non viene generato nessun bit in uscita e il nodo raggiunto, che in tal caso risulta essere interno, viene impostato tramite prefisso come prossimo nodo di start. Si forniscono nel paragrafo successivo delle rappresentazioni grafiche per delucidare il processo appena illustrato.

Esempi Si supponga di utilizzare un DDG Tree che produce in uscita bit con probabilità $p_0 = 5/8$ e $p_1 = 3/8$, e che l'algoritmo gestisca la sequenza random in ingresso in pacchetti di sei bit. Nelle due figure sono rappresentate due casistiche, con differenti input e output. Nella prima figura il pacchetto di bit viene gestito interamente, viene prodotto l'output e il processo termina in una foglia. L'algoritmo quindi non deve modificare il nodo di start e alla prossima iterazione ripartirà dalla radice. Nella seconda invece l'esecuzione termina in un nodo interno, che diventa automaticamente il nuovo punto di partenza. In giallo sono segnati i successivi nodi di start, mentre le frecce indicano il percorso all'interno dell'albero per ciascun gruppo di bit evidenziato.



Esempio 1

Figura 2.7: Esempio di gestione completa del pacchetto di bit, la radice è il prossimo nodo di start.



Esempio 2

Figura 2.8: Esempio di gestione parziale del pacchetto di bit, il nodo interno è il prossimo nodo di start.

L'algoritmo impostato per lo svolgimento dei test possiede la stessa distribuzione di probabilità descritta dall'esempio sopra riportato, ovvero $p_0 = 5/8$ e $p_1 = 3/8$. Tramite la corrispondente struttura di look-up tables è stato implementato nel processore in FPGA, accertandosi sperimentalmente del suo corretto funzionamento. Cioè andando a misurare le probabilità ottenute, usando una sequenza di bit generata in modo casuale di lunghezza maggiore possibile.

Capitolo 3

Test e Risultati

3.1 Panoramica sullo svolgimento dei test

Tramite il setup è stato caricato nella piattaforma FPGA il codice C che implementa l'algoritmo di Knuth-Yao. Il quale riceve in ingresso uno stream di bit casuali generati tramite un RNG (Random Number Generator), esegue il biasing dei bit ricevuti in input e infine produce in uscita una sequenza di bit con distribuzione di probabilità non uniforme. La scelta di generare i random bit internamente al sistema, invece di riceverli direttamente da un dispositivo esterno, è stata giustificata da varie prove sulla trasmissione via UART. Di fatto la gestione di grandi quantità di informazioni si è dimostrata inaffidabile con questo tipo di protocollo, essendo per certi versi troppo semplice e incerto per questi fini. Nel dettaglio il programma che esegue il testbench sull'algoritmo svolge le seguenti operazioni:

1. **Generazione look-up tables e bit casuali**: in prima istanza viene creato un array di bit pseudocasuali tramite RNG, da usare come input per il biaser. Dopodiché vengono create le look-up tables per poter procedere con l'esecuzione, contenute anch'esse in un array.

2. **Esecuzione ciclica dell'algoritmo**: l'algoritmo viene eseguito N volte (nei test svolti $N = 100$) per valutare poi le prestazioni al caso medio. Questo per non dipendere troppo da misure fuorvianti, dovute a rallentamenti del sistema che sporadicamente possono presentarsi. I bit in uscita al biaser vengono salvati ancora una volta in un array.

3. **Calcolo dei tempi di esecuzione**: Attraverso determinate istruzioni viene calcolato il tempo di esecuzione per ogni iterazione, e infine viene ricavato il valore medio.

4. **Print dei risultati ottenuti**: in conclusione viene calcolata la statistica sui bit in output, scorrendo l'array di output ottenuto. Dopo aver raccolto tutte le informazioni, tramite apposite funzioni di print, vengono inviate sfruttando il modulo UART al pc.

Per inserire il codice in NEORV32 è stata verificata l'adattabilità delle istruzioni. Non solo andando a verificarne il funzionamento, ma si è anche tenuto in considerazione l'impatto di queste, soprattutto per quanto riguarda l'occupazione di memoria. Infatti NEORV32 supporta le librerie standard di C (non sempre per intero, dipende dalla versione), tuttavia implementare certe istruzioni, che non sono particolarmente ottimizzate, può far aumentare sensibilmente il peso del programma. Per darne l'idea una istruzione tra tutte quelle testate risultava particolarmente gravosa per la memoria, ovvero la funzione di print, che richiedeva un costo di circa 60 KB aggiuntivi. Ora considerando che la parte FPGA dispone di 630 KB di memoria, si può ben capire come non sia un aspetto irrilevante. Questo perché nelle prove fatte sull'algoritmo, per misurare le prestazioni del sistema, si è cercato di usare la quantità di dati maggiore possibile in ingresso al biaser, cosicché da arrivare a dei risultati affidabili. Avendo a disposizione più dati in input, l'errore sulle probabilità in uscita rispetto a quelle desiderate tende a diminuire, in confronto ad averne un numero esiguo, come si mostrerà dalle misure fatte. Se la generazione dei numeri casuali presenta un piccolo sbilanciamento sulle probabilità, per il comportamento idealmente casuale dell'RNG, con l'aumentare dei bit generati questo tende a diventare trascurabile. Essendo che le risorse disponibili non sono così abbondanti, si è cercato di ridurre al minimo i consumi di memoria per avere un discreto numero di dati su cui effettuare il biasing. In modo da avere infine una distribuzione di probabilità con un'errore accettabile rispetto a quella ideale.

3.2 Gestione delle librerie

Prima di effettuare i test si è dovuto abilitare e inserire tutte le risorse software, in particolare le librerie, per l'operatività delle periferiche e del codice. NEORV32 mette a disposizione, nella build standard, le API (Application Peripheral Interface) per le periferiche più comuni, come il modulo UART, timer, RNG e così via. Ossia sono accessibili dei file header che possono essere inclusi per poter utilizzare istruzioni specifiche del processore, per gestire le singole periferiche. In questo caso sono state inclusi, oltre alla libreria standard di C, la libreria standard del processore "neorv32", oltre alle API del modulo UART e dell'RNG integrato.

3.2.1 Modulo UART

Concentrandosi sul modulo UART, sono stati configurati tutti i vari parametri per poter comunicare, tra cui il BAUDRATE che è stato impostato a 19200 bit/s. Questa periferica è stata sfruttata per l'invio dei risultati ricavati dal biaser al pc, utilizzando le istruzioni della API. Tutto ciò non perché non si possa utilizzare le istruzioni standard di C, ma risultava largamente svantaggioso, soprattutto per il consumo di memoria e per il tempo impiegato.

3.2.2 RNG

Dopodiché la generazione dei bit casuali è stata implementata attraverso delle istruzioni che sfruttano il modulo TRNG (”True Random Number Generator”), attivabile nella configurazione della piattaforma. Si tratta di un RNG che si basa su oscillatori ring in free-run per generare numeri casuali. Un aspetto importante da sottolineare è che questo RNG non è utilizzabile in crittografia, cioè non garantisce una casualità perfetta e può sussistere una cross-correlazione tra i numeri generati e i segnali provenienti dall’ambiente circostante. Tuttavia per dei test puramente prestazionali è più che sufficiente.

3.2.3 Timer

La versione di NEORV32 adoperata non supporta le istruzioni C per la misura dei tempi di esecuzione, di conseguenza è stata utilizzata la libreria ”time”. Sfruttando il timer dell’architettura è possibile, tramite un registro di conteggio interno, andare a calcolare il tempo di esecuzione per le N iterazioni dell’algoritmo. Semplicemente è stato ricavato per differenza tra il valore assunto dal contatore alla fine e all’inizio delle iterazioni, dividendo per la frequenza di clock:

$$t_{mis} = \frac{count_{stop} - count_{start}}{N * f_{clk}}$$

3.2.4 Inserimento delle look-up tables

In realtà anche la struttura di look-up tables è stata inserita nel codice attraverso un file header. Nel dettaglio è stato usato un codice in linguaggio Python che, partendo dalla distribuzione di probabilità voluta, genera le look-up tables associate. Dopodiché lo stesso codice crea il file header contenente una funzione che, quando richiamata, restituisce l’array di LUT.

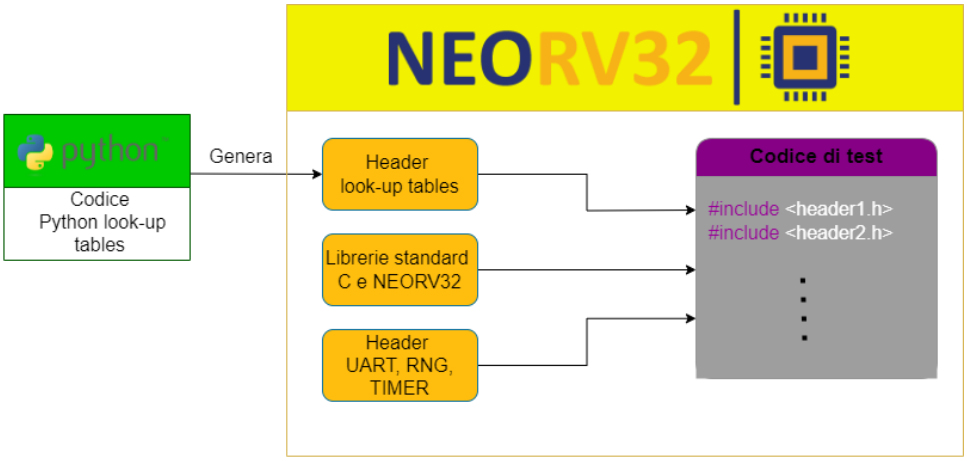


Figura 3.1: Schema di gestione delle librerie incluse durante i test

3.3 Gestione della memoria

L'utilizzo della memoria del processore è suddivisa in due: la zona di memoria per il caricamento del programma, che viene chiamata neorv32_rom, e quella per il salvataggio e la manipolazione dei dati, chiamata neorv32_ram. Le due zone devono essere configurate prima della compilazione del software, a meno che non si voglia mantenere il setup di default, a seconda del consumo di memoria previsto. I parametri modificabili sono: la dimensione e l'indirizzo di base di ciascuna memoria. Queste modifiche possono essere fatte in due modi, per poterle rendere persistenti o meno:

- **flags di compilazione**: sono particolarmente utili per la segnalazione di eventuali errori nel codice, oltre che per l'impostazione della memoria. Sfruttando opportuni flags è possibile comunicare al linker le modifiche da fare sui parametri sopra citati. Tuttavia devono essere reinseriti ad ogni compilazione per mantenere la stessa organizzazione.

- **linker script**: riscrivendo direttamente i campi all'interno del linker script, le modifiche diventano permanenti fino alla nuova riscrittura. Permettendo di eseguire più compilazioni con la stessa configurazione senza specificare flags.

Si osserva poi che i parametri devono coincidere con i corrispettivi presenti nel software Vivado, prima del caricamento del bitstream. Le risorse rese disponibili dal sistema risultano essere di 512 KB, in accordo con l'hardware manager di Vivado, mentre la dimensione effettiva del programma di test risultava essere di circa 27KB. Dunque si è scelto di settare la dimensione di neorv32_rom a 32KB. Infatti la memoria interna viene generata solo per potenze di due, eventualmente approssimando alla potenza di due più vicina. Se si vuole definire un numero qualsiasi come parametro, si deve tenere in considerazione che la struttura hardware potrebbe non corrispondere a quella voluta. Pertanto la dimensione di neorv32_ram è stata impostata a 256 KB, la massima ottenibile. Il consumo di memoria dati è dettato principalmente dai tre array di grandi dimensioni inizializzati nel programma: l'array delle LUT, quello contenente i random bit e l'ultimo contenente i bit prodotti dal biaser. I problemi sostanziali nell'impiegare queste strutture sono stati i frequenti errori d'accesso alla memoria. Nell'istanziare normalmente gli array il processore riportava diversi errori in fase di esecuzione, tra cui "memory access fault" e "store address misaligned". In pratica, con una mole eccessiva di dati, apparivano sempre errori di lettura o accesso a zone non permesse della memoria. Per evitare queste situazioni si è deciso di istanziare gli array utilizzando le funzioni malloc e calloc di C, per garantire una maggiore sicurezza sull'accesso ai dati e la loro allocazione. Avendo queste necessità sono state aggiunte ulteriori impostazioni per la RAM, visto che di default non prevede l'allocazione di un heap, inizializzato con dimensione nulla. Tenendo in considerazione le caratteristiche del layout della

RAM, è stato configurato tramite linker un heap di 128 KB per la gestione degli array. In figura 3.2 è rappresentato il layout della neorv32_ram:

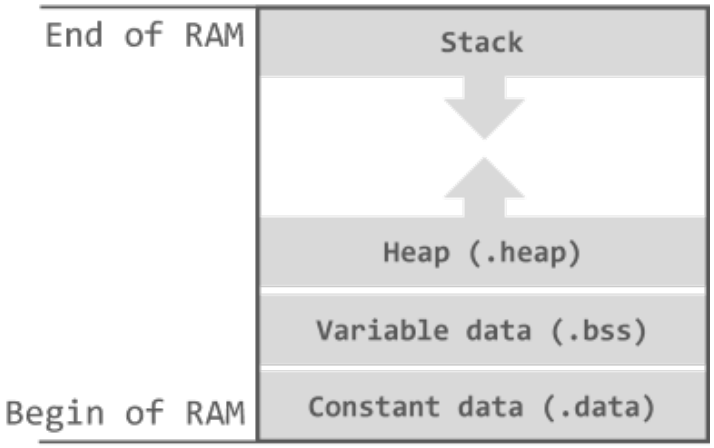


Figura 3.2: Layout di default meomria RAM

Si noti come lo stack sia discendente, a partire dall'ultima locazione della RAM, mentre l'heap cresce a partire dalla prima locazione libera dopo le costanti di programma. Con questa soluzione non si sono più rilevati errori di accesso alla memoria, d'altra parte lo spazio occupabile dagli array è stato abbastanza ridotto. Sono stati svolti vari test con varie dimensioni della sequenza di bit per il biasing, fino a 40000 elementi. Per chiarezza, i due array per i bit in input e output dell'algoritmo hanno lunghezza pari al numero di elementi su cui fare il biasing, mentre l'array di LUT è a dimensione costante (circa 3 KB). Si è verificato sperimentalmente che, andando ad aumentare ulteriormente le dimensioni, si presentano altri errori in esecuzione, molto simili ai precedenti. Tutto ciò a causa di un possibile riempimento eccessivo dello stack, che comporta collisioni con l'heap ed errori di indirizzamento persistenti.

3.4 Analisi dei risultati

3.4.1 Risultati ottenuti

In seguito agli aggiustamenti fatti per cercare di rendere il sistema più efficiente dal punto di vista hardware, si elencano in figura 3.3 le risorse di PYNQ-Z2 impiegate per l'implementazione finale dell'algoritmo:

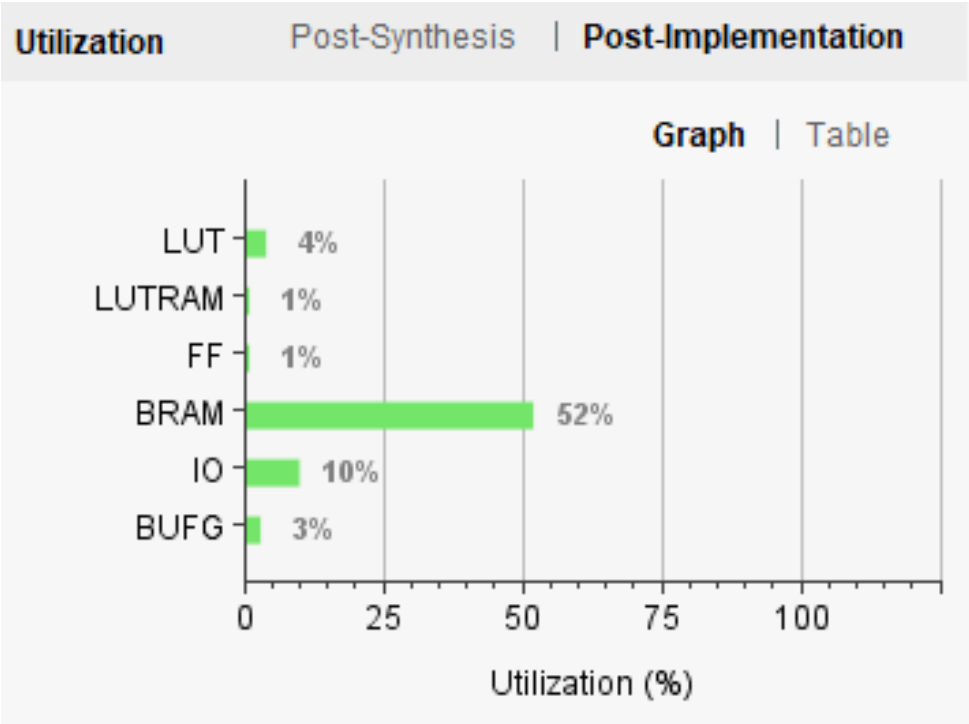


Figura 3.3: Risorse hardware utilizzate su piattaforma FPGA

Passando invece alla misura delle prestazioni, sono stati condotti dei test non solo sulla piattaforma FPGA, ma anche su un normale processore general purpose di un computer. In modo da poter confrontare i risultati ottenuti tramite NEORV32, con una CPU di uso comune. Tutto ciò non con l'obiettivo di stimare la vera e propria differenza di performance tra i due, dato che si tratta di sistemi totalmente diversi, con caratteristiche e campi di applicazione distinti. L'obiettivo della valutazione è solamente capire, a grandi linee, dove si pone la potenza computazionale di questa implementazione FPGA, rispetto a quella di un qualsiasi CPU. Senza dare una precisa classificazione tra dispositivi, ma semplicemente ricavando un ordine di grandezza sulle performance raggiunte. Si elencano in tabella i risultati dei test ottenuti da NEORV32 al variare del numero di random bit usati.

Nella tabella sono trascritte: la distribuzione di probabilità ottenuta (p_0,p_1), il rate dei random bit processati, il bitrate in uscita al biaser, il numero di bit prodotti e il tempo di esecuzione:

n. elementi	p_0	p_1	processing rate [Mbps]	biaser gen. rate [Mbps]	n. bit biaser output	t_{mis} [ms]
10	0.64583	0.35416	2.56016	1.53609	6	0.31242
100	0.61818	0.38181	1.66883	0.91785	55	0.47937
1000	0.63116	0.36883	1.17960	0.67001	568	6.78193
10000	0.62436	0.37563	0.90605	0.51717	5700	88.29508
20000	0.62452	0.37547	0.84754	0.48420	11426	188.77990
30000	0.62561	0.37438	0.81987	0.46847	17142	292.72847
40000	0.62551	0.37449	0.79614	0.45489	22855	401.93831

Tabella 3.1: Dati ottenuti dai test sul processore NEORV32

Da notare come il rapporto tra il rapporto tra il bitrate di dati generati e quello dei bit processati, mediando tra le misure fatte, risulta essere di all'incirca 1.749. Il quale rispetta giustamente le previsioni descritte dal teorema di Knuth-Yao, che sancisce i seguenti buond:

$$H(x) = -\frac{5}{8}\, log_2 \frac{5}{8} - \frac{3}{8} log_2\, \frac{3}{8} = 0.954, \quad 0.954 < E[L_T] < 2.954$$

Per confrontare questi valori è stato fatto un singolo test su un processore intel i7-12700h di 12^a generazione, sufficiente per comprendere le differenze sostanziali tra le due CPU:

n. elementi	p_0	p_1	processing rate [Mbps]	biaser gen. rate [Mbps]	n. bit biaser output	t_{mis} [ms]
1000000	0.624771	0.375229	5060.085351	2891.666735	571466	1.576347

Tabella 3.2: Dati ottenuti dal test su intel i7 12^a generazione

Inoltre sono stati quantificati mediamente i consumi e le temperature raggiunte relative alle due piattaforme, per poter valutare non solo la velocità di elaborazione dei bit e di esecuzione, bensì anche il trade-off tra performance e dissipazione di potenza. Tutto questo è stato fatto attraverso dei semplici software di monitoraggio per le due CPU, quindi si sottolinea che i valori ricavati sono delle stime approssimative e non dei valori esatti. Nell'estrazione della potenza per la CPU i7 è stato stimato prima il consumo a riposo (7 W), cioè senza programmi in esecuzione eccetto quelli strettamente necessari al funzionamento del pc. Il quale è stato usato come offset da togliere al consumo totale di 29 W, ricavato durante l'esecuzione del programma, per migliorare l'accuratezza della stima. Essendo che i7, a differenza di NEORV32, deve gestire anche le funzionalità del sistema operativo, si sarebbe compiuto un errore di valutazione troppo grossolano nel trascurare l'offset. In tabella 3.3 sono presenti i valori medi ottenuti per il consumo di potenza e la temperatura:

CPU	potenza totale [W]	potenza in esecuzione [W]	temperatura [°C]
intel i7	29	22	93
neorv32	0.5	0.5	44.5

Tabella 3.3: Misure sul consumo di potenza e temperatura

3.4.2 Considerazioni e confronti

Analizzando i valori ottenuti dai test si vede come i bitrate dell'algoritmo utilizzando la piattaforma FPGA sono nell'ordine del Megabit, mentre con la CPU intel si arriva a una velocità di elaborazione dei dati molto maggiore, superiore al Gigabit. Tuttavia bisogna tenere conto di molteplici fattori che motivano il divario significativo:

- **Frequenza di clock**: il primo aspetto che impatta fortemente sulle differenti velocità delle due CPU è la frequenza di clock, che avvantaggia molto il processore intel. Per la precisione NEORV32 lavora a 125 MHz, mentre per il core di intel i7 è stata misurata la frequenza di clock durante l'esecuzione del codice per avere un valore affidabile. La misura riporta una frequenza di 4.2 GHz per cui il fattore che separa le due CPU è di:

$$K = \frac{f_{i7-base}}{f_{NEORV32}} = \frac{4.2 * 10^9}{125 * 10^6} = 33.6$$

  Sicuramente si tratta di un parametro moltiplicativo elevato, ma che da solo non giustifica a pieno questa discrepanza tra i bitrate prodotti, che ammonta a un fattore superiore a 10^3.

- **Architettura e ottimizzazioni**: le CPU messe a confronto sono ampiamente diverse e sono state progettate con filosofie diverse. Il processore i7 possiede un'architettura fortemente orientata alle performance, per di più dispone di una lunga serie di caratteristiche orientate in tal senso. Basti citare il funzionamento multicore, la disponibilità di 25 MB di cache interna e così via. Invece NEORV32 è un processore molto compatto e ha un'architettura molto più snella e semplice. Quest'ultimo offre tutt'altri tipi di vantaggi, tra cui una buona efficienza dal punto di vista energetico e soprattutto un'altissima configurabilità, che si collega strettamente anche ai consumi.

- **Trade-off tra performance e consumo di potenza**: sebbene la potenza computazionale di NEORV32 risulti più bassa, dalle misure fatte si evidenzia un'efficienza energetica superiore rispetto all'altra CPU. Del resto per la prima il consumo è di circa 500 mW con una temperatura media di 44.5 °C, mentre per la seconda si raggiungono i 22 W con una temperatura media di 93°C. Nonostante le stime fatte non siano troppo accurate, riescono comunque a rendere l'idea sui vantaggi che riesce ad offrire in termini di consumo e temperatura. Questi vantaggi per determinate applicazioni possono rivelarsi molto utili, come nel settore aerospaziale. Si osservi che queste misure sono state incluse solo con l'obiettivo di ottenere un quadro generale sulle prestazioni delle due CPU, lasciando una caratterizzazione più approfondita come suggerimento ad analisi future.

Capitolo 4

Conclusioni

In definitiva è stato sviluppato un sistema sulla piattaforma FPGA PYNQ-Z2, basata sul processore RISC-V NEORV32, per implementare un algoritmo di biasing applicato alla Quantum Key Distribution. Nel dettaglio il programma in questione si fonda sull'algoritmo di Knuth e Yao e ha come scopo quello di generare bit con probabilità non uniforme e arbitraria. Tale funzionalità viene sfruttata nelle comunicazioni QKD, durante la fase di post-processing, per aumentare l'efficienza sui dati utili nella generazione della chiave di comunicazione. Sono stati fatti poi dei test, scegliendo una distribuzione di probabilità sui bit di $(p_0 = 5/8, p_1 = 3/8)$ per valutare la correttezza del codice e le performance dell'apparato costruito. Dopodiché si sono confrontati i risultati ottenuti da quest'ultimo con quelli di un processore general purpose, una CPU intel i7 di 12^a generazione, in modo da avere un punto di riferimento su cui soppesare il lavoro svolto. I risultati ottenuti evidenziano una sostanziale inferiorità del processore in FPGA, in termini di velocità d'esecuzione e bitrate, rispetto all'altro. Per quanto riguarda i tempi di esecuzione abbiamo poco più di 1 ms per i7 con 1000000 di bit manipolati, comparati con gli oltre 400 ms di NEORV32 con appena 40000 elementi. Mentre per i bitrate dei dati generati in uscita dell'algoritmo si arriva a circa 2.89 Gigabit nel primo caso, invece nel secondo caso, con un numero significativo di elementi, il valore scende sotto il Megabit. Tuttavia NEORV32 si è dimostrato essere migliore dal punto di vista del risparmio energeico, con un consumo in esecuzione medio di appena 500 mW, contro i 22 W dell'altra CPU. Nonostante ciò si ribadisce quanto esplicitato in precedenza, cioè che le due CPU possiedono caratteristiche e campi di applicazione totalmente diversi. Questa analisi è servita unicamente per comprendere a che punto sono le prestazioni di questo sistema FPGA, rispetto a un processore generico, per capire in che direzione impostare gli sviluppi futuri. In questa tesi è stata creata una struttura di base, senza ottimizzazioni molto avanzate. Dunque è certamente possibile approfondire vari aspetti per migliorare le prestazioni raggiunte. Come ad esempio approfondire la gestione delle risorse di NEORV32, che in questo caso è stata particolarmente critica. Oppure impiegare o elaborare ottimizzazioni aggiuntive per la piattaforma e molto altro.

Bibliografia

[1] C. Maxfield, *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.

[2] I. Grout, *Digital systems design with FPGAs and CPLDs*. Elsevier, 2011.

[3] A Waterman, K Asanović et al., «The RISC-V Instruction Set´ Manual Volume I: User-Level ISA Document Version 20191214,» *RISC-V International, Tech. Rep.*, 2019.

[4] S Nolting e U Martinez-Corral, «The neorv32 risc-v processor,» *Fischer," The neorv32 risc-v processor*, 2022.

[5] A. Carrasco-Casado, V. Fernández e N. Denisenko, «Free-space quantum key distribution,» *Optical Wireless Communications: An Emerging Technology*, pp. 589–607, 2016.

[6] «What is FPGA Chip.» (2024), indirizzo: `https://www.raypcb.com/an-introduction-to-fpga/`.

[7] L. V. Cherckesova, O. A. Safaryan, A. N. Beskopylny e E. Revyakina, «Development of Quantum Protocol Modification CSLOE–2022, Increasing the Cryptographic Strength of Classical Quantum Protocol BB84,» *Electronics*, vol. 11, n. 23, p. 3954, 2022.

[8] J Aditya e P. S. Rao, «Quantum cryptography,» *Proceedings of computer society of India*, 2005.

[9] S. Weigert, «No-cloning theorem,» in *Compendium of quantum physics*, Springer, 2009, pp. 404–405.

[10] S. Gao, *Collapse of the Wave Function: Models, Ontology, Origin, and Implications*. Cambridge University Press, 2018.

[11] Q. Cryptography, «Public key distribution and coin tossing,» in *Proc. of IEEE Int. Conf. on Comput. Sys. and Sign. Proces., Bangalore, India*, 1984, pp. 175–179.

[12] H.-K. Lo, H. F. Chau e M. Ardehali, «Efficient quantum key distribution scheme and a proof of its unconditional security,» *Journal of Cryptology*, vol. 18, pp. 133–165, 2005.

[13] C. Lee, I. Sohn e W. Lee, «Eavesdropping detection in BB84 quantum key distribution protocols,» *IEEE Transactions on Network and Service Management*, vol. 19, n. 3, pp. 2689–2701, 2022.

[14] «PYNQ-Z2 Reference Manual v1.0.» (2018), indirizzo: `https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf`.

[15] «PmodUSBUART™ Reference Manual.» (2015), indirizzo: `https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf`.

[16] F. Saad, C. Freer, M. Rinard e V. Mansinghka, «The fast loaded dice roller: A near-optimal exact sampler for discrete probability distributions,» in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 1036–1046.

[17] D. Knuth, «The complexity of nonuniform random number generation,» *Algorithm and Complexity, New Directions and Results*, 1976.

[18] A. El Gamal e Y.-H. Kim, *Network information theory*. Cambridge university press, 2011.