Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Magnus Øverbø

Cryptanalysis of Irregularly Clocked LFSR

using approximate RBP search on FPGA

Master's thesis in Masters of Science Information Security: Technology
Supervisor: Prof. Slobodan Petrovic
June 2019

NTNU
Norwegian University of Science and Technology

Magnus Øverbø

Cryptanalysis of Irregularly Clocked LFSR

using approximate RBP search on FPGA

Master's thesis in Masters of Science Information Security: Technology
Supervisor: Prof. Slobodan Petrovic
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

NTNU
Kunnskap for en bedre verden

Preface

This Master's thesis in AIMT at NTNU was carried out during the spring semester of 2019 with Prof. Slobodan Petrovic as supervisor. Prof. Petrovic started this research[1] on the theoretic background of using approximate search as a method for obtaining the Levenshtein distance between two bit sequences of unequal lengths. Then utilise this in the generalised correlation attack[2] to recover the initial state of the irregularly clocked Linear Feedback Shift Register (LFSR).

The topic of this thesis is to assess the practical implementation of this method on Central Processing Unit (CPU) and Field-Programmable Gate Array (FPGA) using an unconstrained Approximate Row-wise Bit-Parallel (ARBP) search[1] using the shift-AND algorithm developed by Baeza-Yates and Gonnet[3].

The reader's background in this thesis is assumed to be familiar with computer architecture and logic operations in electronic circuitry. These aspects are explained, but familiarity is still expected.

Magnus Øverbø
2019-06-01

Acknowledgement

A special thanks to Prof. Slobodan Petrovic for the guidance, supervision and feedback provided throughout the thesis.

Thanks to Mr. Høgli for reviewing my draft.

Magnus Øverbø

Abstract

Cryptanalysis on a cipher system the utilising 0/1 clocking Binary Rate Multiplier (BRM) as the keystream generator renders Siegenthaler's classical correlation attack[4] unusable since the output sequence and the undecimated bit sequence generated by the irregularly clocked Linear Feedback Shift-Register (LFSR) are of different lengths. Instead, by utilising the generalised correlation attack developed by Golic and Mihaljevic[2] the Levenshtein distance between the two sequences can be utilised as the basis for correlation.

In this thesis, we explore the application of implementing an unconstrained Approximate Row-wise Bit-Parallel (ARBP) search, using Wu and Manber's shift-AND algorithm[5], to obtain the Levenshtein distance between the ciphertext and the undecimated bit sequence of the irregularly clocked LFSR as the correlation metric. Our findings shows that the FPGA will perform better than a Central Processing Unit (CPU) implementation, as it operates with constant mean execution time, estimated by $T_{tot} = R_{ops} \times 4f$. Where R_{ops} represents the number of search state values which must be updated throughout the search, and f is the FPGA designs clock rate.

The CPUs processing time is shown to be of linear growth, based on the length of the search word as given by its periodic increase by a factor of $\lceil \frac{M}{w} \rceil$, where M is the search word length, and w is the size of the CPUs machine word.

However, overall the time required for processing a real-world cipher system using Field-Programmable Gate Array (FPGA) requires a large amount of resources. Even a feedback polynomial of $L = 32$, with $M = 1024$ and a clock rate of $f = 2.39GHz$ will require 43 days to complete, while $M = 4096$ would require 695 days to complete. Even so, given 700 FPGAs running simultaneously a search can be completed within a day at the cost of the additional FPGAs needed, making it possible given enough resources are available.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Background

Large number calculations on a Central Processing Unit (CPU) is not possible without a library providing a layer of abstraction, as CPUs can maximum hold 64b in a single register. This limitation is a problem for cryptography requiring fast computations of large numbers since it creates latency. A Field-Programmable Gate Array (FPGA) is a stateless system consisting of a large amount of interconnectable Configurable Logic Block (CLB), Random Access Memory (RAM) module and several interfaces. FPGA is perfect for constructing logical circuits and performing logical operations as it generates a circuit with programmable logic without having a fixed register size like CPUs. Applications for FPGAs are among others, signal processing, string matching and cryptanalysis as explored in this thesis. It has already been shown to outperform CPUs in spam filtering by Borgund[6], and in text searching, as studied by Irwin et. al[7] and Michailidis and Margaritis[8]. The downside is that the clock frequency on a consumer-grade FPGA is much slower than a CPU, but its primary strength is that it performs operations in parallel and concurrently.

The theoretical application of performing cryptanalysis using a constrained approximate bit-parallel search algorithm is given in [1]. It details a cipher system built on the 0/1-clocking Binary Rate Multiplier (BRM) which generates the irregularly clocked keystream to encrypt the plaintext. The paper discusses the general correlation attack[2] and utilising a constrained Approximate Row-wise Bit-Parallel (ARBP) search to recover the initial state of an irregularly clocked Linear Feedback Shift Register (LFSR).

This thesis investigates the applicability of performing cryptanalysis of irregularly clocked LFSRs by employing an unconstrained ARBP search on an FPGA. It evaluates the Levenshtein distance[2] between the intercepted ciphertext and all undecimated bit sequences generated by the initial states of the clocked LFSR. The thesis uses an unconstrained ARBP search with the shift-AND algorithm developed by Wu and Manber[5], based on Baeza-Yates and Gonnets previous work[3], to obtain the Levenshtein distance.

Approximate search requires processing the M bit search word and the ciphertext sequence, which is tried embedded into the N bit search text allowing for K errors, where the search text is the undecimated keystream sequence of length $N = 2M$ bit. Given the limitation on a CPUs register size w is mainly 64b, this thesis sought to determine if an FPGA could perform an unconstrained ARBP search faster than a standard CPU, and what limitations occurs in relation to cryptanalysis.

1.2 Related Research

Petrovic[1] explore the theoretical applicability of using a version of the unconstrained approximate, bit-parallel search algorithm developed by Wu and Manber[5], to calculate the constrained

Levenshtein distance between two bit sequences. This distance metric has been explored in the past by Golic and Mihaljevic[2] to perform cryptanalysis of a cipher system which utilises BRM as its keystream generator. More specifically, a 0/1 clocking generator, based on two separate LFSRs. Golic and Mihaljevic[2] proposed a generalised correlation attack based on the constrained Levenshtein distance as Siegenthaler's[4] classical correlation attack, using Hamming distance, is not possible with irregular clocked LFSR as the sequences are of different length.

This thesis seeks to build on Golic and Mihaljevic work regarding the estimation of Levenshtein distance between an output sequence of length M and an undecimated LFSR sequence of $N = 2M$. The thesis implements an unconstrained ARBP search as developed by Wu and Manber[5]. The thesis also seeks to implement this algorithm on both a standard CPU architecture and using a specialised hardware, FPGA. which is capable of generating large complex circuits with programmable logic. The reason for this is that FPGAs do not have constraints on register size, w, as opposed to CPUs with $w = 64b$. Given this advantage, it was desirable to determine how efficient the implementation of this algorithm would be on an FPGA in the setting of cryptanalysis.

The topic of efficiency in bit-parallel searches on CPU and FPGA is well researched. Irwin et.al[7] focused on exact string matching on an FPGA to improve the searching within an IDS. He utilised the exact bit-parallel matching as developed by Baeza-Yates and Gonnet[3]. Their tests based itself on searching a set of texts, given a set of search words where $M = 32b$. It evaluated its findings against the same application run inside MatLab, which is not an ideal comparison as MatLab is a scripting language. However, their result showed that FPGAs were 160 times faster than MatLab.

Borgund[6] explored a similar project to Irwin et.al[7], and tested the increased efficiency of pattern recognition in spam email using an FPGA. The test performed was trying to match multiple search words within the text of emails using the Levenshtein distance and approximate bit-parallel search on FPGA. Borgunds results were evaluated against the same test on CPU, finding the FPGA was 55 times faster.

A different paper details the same discovery, using the approximate bit-parallel search algorithm as developed by Wu and Manber[5]. Michailidis and Margaritis[8] research is more closely related to this thesis because their tests used long patterns testing. They found that the FPGA was 9-340 times faster than a CPU given $M = 1024b$. One difference between this thesis and Michailidis and Margaritis is the implementation. Michailidis and Margaritis preloaded the FPGA with all data required for the search, meaning it only had to handle the searching of fixed strings which is very efficient. In this thesis, an entire system was implemented to automate the search in relation to cryptanalysis, generate the data sets and transfer data.

In regards to specific hardware, Tran, Schindel, Liu and Schmidt[9] implemented approximate bit-parallel search of $M = 512b$ on a CPU capable of running a 512b register per CPU core. Although achieving excellent results in regards to CPU implementation and multithreading solutions, their problem arises when $M > 512b$ as the physical limits result in degraded performance. The cost of these CPUs is also a consideration, even though they will not have an application for cryptanalysis given its the limitation of $M = 512b$.

Even though the efficiency of searching on FPGA is researched, its focus has been on sequences

shorter than w. Testing of search words longer than w is avoided on the background that the search will lose efficiency. In this thesis, it is sought to verify this and test how long search words can be processed, allowing $K \geq \frac{1}{4}M$ errors.

This thesis implement a complete system for automated search with cryptanalysis as the focus, on both CPU and FPGA. It performs an unconstrained ARBP search for all initial states of an irregularly clocked LFSR and is further used to obtain the search speed of both CPU and FPGA. Because of this, the system must be capable of generating the search word and the search text automatically. The search algorithm then tries to embed the search word into the search text, and report matches to the host, allowing K errors to occur.

1.3 Scope of the thesis

For the scope of this thesis, the implementations must simulate the cipher system and generate an arbitrary length search word, generate the search text, search all initial states of an LFSR given its degree and polynomial, and record matches of interest. Part of the thesis is to implement this search on both CPU and FPGA where most aspects of the information system are known. Meaning the LFSRs feedback polynomials, ciphertext and the BRM are known. Then perform the cryptanalysis by searching all undecimated bit sequences produced by the irregularly clocked LFSR, provided by the set of possible initial states, and logging the correlation metrics obtained from the search.

Further cryptanalysis of the cipher system is out of scope for this thesis. Meaning the one-to-one search for the combination of plausible undecimated bit sequences and *clocking* LFSR is out of the scope of this thesis.

1.4 Contributions

This thesis seeks to evaluate how efficient FPGAs are in relation to CPUs for searching of N bit long binary sequences using $\frac{1}{2}N$ long search words in the application of cryptanalysis. The reasoning for this is to evaluate the FPGAs applicability as a platform for obtaining the Levenshtein distance between two sequences and use it as the basis for cryptanalysis with the general correlation attack by Golic and Mihaljevic[2]. The search implemented in this thesis is the unconstrained, approximate row-wise bit-parallel search, using the shift-AND[3, 5] algorithm.

1.5 Research Questions and Hypothesis

The thesis objective is to perform and evaluate the application of running an unconstrained ARBP shift-AND search on both FPGA and CPU. In doing so, the following hypothesis is developed based on past research and the thesis problem.

FPGA will perform an unconstrained ARBP search faster than a CPU, given a normalised clock rate.

The following research questions are posed based on the hypothesis to prove or disprove the hypothesis.

- How long search words can practically be processed.

- How fast is the search performing.
- How do changes to polynomial, search word length and error threshold affect the performance.
- How do the FPGA and CPU compare, given a normalised CPU clock speed.
- What are the limitations given each architecture.

1.6 Thesis Outline

The thesis consists of three main chapters. Chapter 2 goes through and explains the background theory behind cryptanalysis, searching, and the system architectures used in the thesis project. Chapter 4 goes through both system architectures and explains how the implementations designs, discovered issues, and testing is also analysed to ensure the data collected would be valid. Chapter 6 discusses the results collected and the subsequent data analysis.

Chapter 1 entails the problem, defines the scope, discuss past research and poses the research questions. Chapter 2 explores the background theory and concepts which the thesis bases itself on. Some key aspects are Non-Deterministic Finite Automaton (NFA), unconstrained ARBP search, and Golic and Milosevic's proposed generalised correlation attack[2] and FPGAs. The background theory provides the reader with knowledge of the required topics for this thesis.

The research methodology is discussed in chapter 3. It discusses the methods used for research, experiment, software development and data analysis. In chapter 4, the implementation is explained for both CPU and FPGA. It discusses the considerations made during the design and implementation, and analyses implementations to ensure they do not provide data which would be considered tainted or incorrect.

Chapter 5 lists and explains the exact setup for the experiment and data collection. It presents the overview of the data sets along with initial assumptions and assessments. Following this chapter, it discusses the analysis of the gathered data, in chapter 6. Based on the obtained data sets, it compares the implementations using a common factor before analysing the data sets behaviour to draw a conclusion based on the characteristics and comparative analysis.

Lastly, chapter 7, summarises the findings and concludes this thesis before providing suggestions for future work. The attached appendices list the data sets collected and the final set of source code for both implementations.

2 Cryptanalysis of irregularly clocked LFSR using ARBP search

In this thesis, all binary registers and sequences are depicted with the Most Significant Bit (MSB) as the left-most bit and Least Significant Bit (LSB) as the right-most bit. Bit-wise shift operations are left-shift operations, and numbering of registers and sequences are from right to left.

2.1 Stream Ciphers

> A stream cipher is a class of encryption algorithms, which operates on the individual characters or bits of a plaintext using an encryption transformation which varies with time. – Chapter 6.1, Handbook of Applied Cryptography, Menezes et al. [10]

The stream cipher system in this thesis is a symmetric cipher, meaning the same key is used for both encryption and decryption. The cipher system in this thesis is also a synchronous stream cipher since the keystream is produced independently of the plaintext and ciphertext.

The cipher system in figure 1 operates in such a way that each bit of a plaintext(P) is encrypted by bit from a pseudorandom bit sequence(X), the keystream. The ciphertext is generated as $X_i \oplus P_i = Z_i$, by performing the bit-wise operation XOR on each pair of plaintext and keystream bits, denoted by the i-th bit. To recover the plaintext, the same procedure using the bit-wise XOR operation is employed, but on each pair of ciphertext and keystream bits, as $X_i \oplus Z_i = P_i$.

2.2 Linear Feedback Shift Registers

A Linear Feedback Shift Register (LFSR) is a binary shift-register, S, where a polynomial denotes the feedback function and all bits are shifted each time it is clocked. Given the polynomial, the shift-register produces a pseudorandom output bit sequence, which is used as the stream cipher's keystream. This operation is detailed in, for example, [11].

The LFSR is represented by its current state and its feedback polynomial, $C(D)$, of degree L, which is also the size of the binary shift-register. Church[12] represents polynomials as $L+1$ long bit sequences, which is not applicable in software or hardware. To compute a polynomial it has to be represented as a L bit long register, e.g. the polynomial $C(D) = x^3 + x + 1$ is represented by a 3b register as; $1\ 0\ 1 \rightarrow x^3 + x$.

A LFSR computes its next state by the following method and is calculated each time the LFSR is clocked. Store the MSB S_L, calculate the feedback value using the polynomial and current state, shift the register and discard the current MSB. Insert the feedback value as the LSB, S_1.

First, the MSB is temporarily stored because it is the current states output bit, and is discarded by the future shift-operation. The feedback value, $F(S)$, is calculated using the feedback polynomial and the current state, as shown below. It produces a single bit output, which is assigned as the LSB,

S_1 after the shift operation occurs.

$$F(S) = (\sum_{i=1}^{L} C(D)_i \wedge S_i) \mod 2 \tag{2.1}$$

Following obtaining the feedback value, the next LFSR state is set by shifting the current state by a single bit. The shift results in discarding the MSB, which is why it is temporarily stored, and after the shift occurs, the LSB is set to the feedback value.

$$\begin{aligned} S_i &= S_{i-1} \quad \text{for} \quad i \in \{2, .., L\} \\ S_1 &= F(S) \end{aligned} \tag{2.2}$$

When the LFSR is clocked n number of times the resulting pseudorandom output bit sequence, X, produced will be n bit long. X is then further used to produce the ciphertext Z, as illustrated in figure 1, by XOR-ing the X with plaintext P as denoted below.

$$Z_i = P_i \oplus X_i \quad \text{for} \ \ i \in \{1, .., n\} \tag{2.3}$$

An important factor regarding the feedback polynomials used for an LFSR is that they should be primitive polynomials. Utilising these polynomials will ensure that the maximum length period, $2^L - 1$, is generated. This is because these polynomials generate all possible states of the LFSR and the maximum output, except for the 0 state, before it repeats itself. A list of primitive polynomials for $GF(2^L)$ is, for example, provided by [13], which is derived from [12].

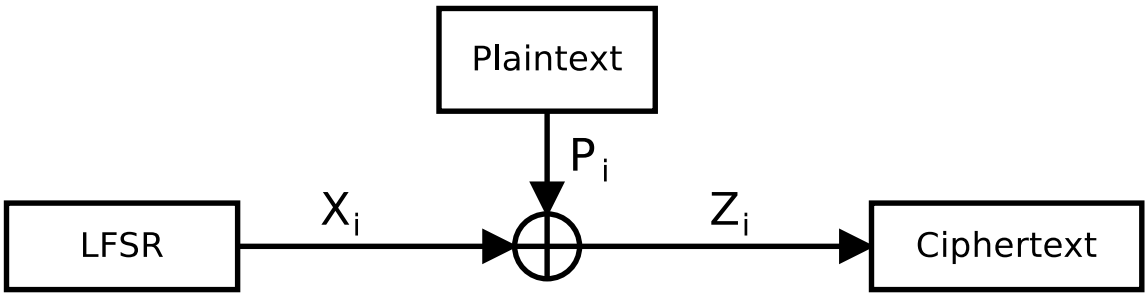

Figure 1: Encryption using LFSR

2.2.1 Clock Controlled Generators: Irregularly Clocked LFSR

In this thesis, the pseudorandom bit sequence is generated by an irregularly clocked LFSR, which produces a decimated bit sequence of greater linear complexity. The method employed is a Binary Rate Multiplier (BRM) called 0/1-clocking, see, for example, [14].

The BRM used in this thesis operates as shown in figure 2. It depicts two LFSR producing two separate bit sequences, X and Y. These are fed into the decimation function producing the X' keystream bit sequence. This bit sequence is then XOR-ed with the plaintext, P, producing the ciphertext, Z.

The decimation function operates by clocking both LFSRs, X and Y simultaneously generating the bits X_i, Y_j. If $Y_j = 1$ LFSR X is clocked once more, producing X_{i+1}. X_{i+1} is then added to the irregularly clocked bit sequence X' as X'_j. If $Y_j = 0$, X_i is just forwarded to X' as bit X'_j. The resulting bit sequence X' is then used as the keystream and is XOR-ed with P producing the ciphertext Z.

The maximum number of deletions between X and X' is equal to the length M of the decimated bit sequence, given that the probability $Pr(Y_j = 1) = 1$. The minimum number of deletions is 0 given that the $Pr(Y_j = 1) = 0$, which occurs if the clocking LFSR is initialised with the 0 state. The clocking LFSR should produce a sequence of independent identically distributed binary variables, Y, with $Pr(Y_j = 1) = 0.5$ for all initial states of the LFSR.

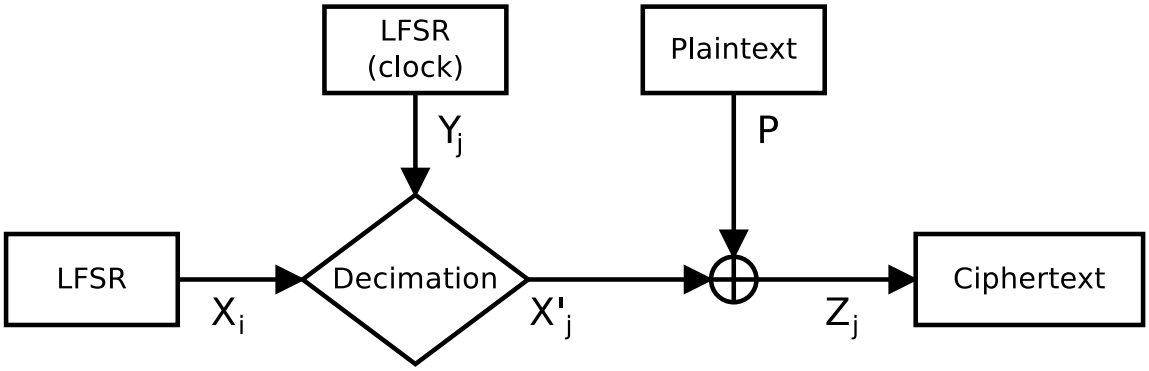

Figure 2: Cipher system using the 0/1-clocking BRM as keystream generator

The LFSRs in this thesis utilise primitive feedback polynomials for two reasons. First, using primitive polynomials of the same degree L in a BRM pseudorandom generator results in a linear complexity of $L(2^L - 1)$[15]. Secondly, the overall period is $(2^L - 1)^2$. The primitive polynomials used in this thesis were selected from [13, 12] for $GF(2^L)$, where $2 \leq L \leq 32$.

The linear complexity is based on the shortest period it generates, and the linear complexity is defined as the shortest LFSR capable of generating the given sequence of bits.

2.3 Cryptanalysis

Given the cipher system, there are requirements for the generation of keystreams to ensure it has adequate security. Regarding the statistical properties of the generated sequence, Golomb's postulates, as listed in [16], have to be satisfied and can be tested through various statistical tests and autocorrelation. The linear complexity must be high, and the BRM 0/1-clocking LFSR used in this thesis has a linear complexity of $L(2^L - 1)$[15]. Lastly, the periods generated must be long, and in this case, the overall period is $(2^L - 1)^2$[15].

Given these requirements, a keystream generator can be seen as adequately secure and resistant against attacks. The cryptanalysist's job then becomes to find a scientific method to break the cipher system. In this case the task is to recover the initial states of the two LFSRs, X and Y, used to gen-

erate the ciphertext, on the premise that the following parameters are known, the LFSRs feedback polynomials, $C(D_X)$ and $C(D_Y)$, and the ciphertext Z of length M.

2.3.1 The Brute-Force Attack

Given the cipher system used in this thesis, recovering the initial state of both LFSR using a brute force attack would require evaluating all combinations of initial states of X and Y. The search scope would consist of $(2^L - 1)^2$ decimated keystreams X', which must be evaluated against the ciphertext to recover the plaintext. Such a search quickly becomes infeasible to complete, e.g. given a polynomial of degree 100, the search scope would consist of approximately $1.606E + 60$, as opposed to a single LFSR with only approximately $1.267E + 30$.

Even though a brute-force method will find the correct solution eventually, the time requirements are usually well beyond the acceptable ones. Besides, there is a possibility of many possible solutions that could be found given that the plaintext is too short or too generic.

2.3.2 Generalised Correlation Attack

Given Siegenthalers classical correlation attack[4] is based on the Hamming distance of two equal length bit sequences, it is not usable given the cipher system in this thesis. When a BRM-based generator generates the pseudorandom sequence, the undecimated sequence and the output sequence will be of different lengths, and Hamming distance is rendered unusable.

Golic and Mihaljevic[2] explored and developed a generalised correlation attack, which is based on a constrained Levenshtein distance between two bit sequences of different lengths. The two lengths must be of non-negative difference when calculating the correlation. Which in this case is always positive. Golic and Mihaljevic propose the same cipher system, as illustrated in figure 2. Where $Pr(Y_j = 1) = 0.5$, and the $Pr(P_j = 1) = p \neq 0.5$ for all j. These parameters generate the binary sequence Z, the ciphertext.

To evaluate the distance between a LFSR generated bit sequence and the ciphertext, $2^L - 1$ initial states must be evaluated and subsequently added to either of the possible hypotheses, H_0 and H_1. Where the resulting Levenshtein distance d of the variable D is used for classifying the evaluations, H_0 a representing a plausible solution, and H_1 representing a non-match. After evaluation two sets of probability distributions are generated $Pr(D|H_0)$ and $Pr(D|H_1)$.

The evaluation is based on the length of the ciphertext, M and the selected threshold value, t, which is set to achieve the desired probabilities for false-negatives $P_m = 10^{-3}$ and false-positives $P_f \approx 2^{-L}$.

Following the initialisation, the Levenshtein distance between the generated undecimated bit sequence for each initial state of X is calculated using constrained search, explained in section 2.4.6. This constrained search allows match, substitution and deletions to occur, with the additional requirement of only allowing a run of one deletion. The resulting Levenshtein distance d is then classified as H_0 or H_1 according to t. The resulting set of $Pr(D|H_0)$ will contain the most plausible initial states which could generate the solution.

2.4 Search

Efficient searching algorithms is a well-researched topic, and for this thesis, the method employed for searching is Wu and Manbers Approximate Row-wise Bit-Parallel (ARBP) search algorithm[5]. This search is built on the work regarding the exact search by Baeza-Yates and Gonnet[3] using a bit-parallel algorithm. Baeza-Yates and Gonnet was the first to release the bit-parallel search for exact pattern matching, which was based on finite automata theory, like Knuth, Morris and Pratts algorithm[17], and it exploits the finite alphabet as Boyer and Moores algorithm[18] do.

A general problem in searching is the handling of large patterns as a Central Processing Unit (CPU) loses efficiency when handling data larger than a single machine word, w. Several methods for performing approximate search have been implemented, and the unconstrained approximate row-wise bit-parallel search algorithm based on Wu and Manbers shift-AND algorithm is explained in the following sections. The following sections details the concepts of the Non-Deterministic Finite Automaton (NFA)[3, 5], bit-parallel search[3, 5], the shift-AND algorithm and the difference between constrained and unconstrained search.

An exact search is defined as the search word, or pattern, P exists as a direct subset of the search text T, $P \subset T$. An approximate search is defined as, P is of distance k to Q, if it can be transformed through k number of insertions, deletions or substitutions[5]. This is explained further in section 2.4.2, finite automata, and section 2.4.3, bit-parallel search.

2.4.1 Bit-Parallelism and Bit-operations

Bit-Parallelism, as explained by Navarro[19], chapter 1.3.1, is the representation of bit sequences and processing of bit-wise operations in a computer and hardware. In hardware, many bits can be represented as a single register of size w, which then can be updated in a single operation, e.g. given the logical AND operation in figure 7, two registers of w size can be processed as a single operation of parallel logic operations reducing the number of operations by w. This parallelism is the aspect referred to and allows for high-speed processing of complete registers.

For a standard CPU w is 64b, but on an Field-Programmable Gate Array (FPGA) w can be set to an arbitrary size given the FPGA has allocatable space for it. This ability is the reason why FPGAs are more efficient in performing bit-wise operations, as CPUs must perform the same task repeatedly until the entire value is processed.

Given this thesis uses bit-wise operations frequently, especially in the case regarding implementation, the used bit-wise operations are explained below. Arithmetic operations are not used in the search, besides incrementing counters, so these concepts are not explained.

Bitwise shift

This operation is denoted by "« n" or "» n", which produces a left or right shift of n positions. The operation will move all bits in the register n positions the declared direction, which results in discarding the n outermost bits and insert n 0-values at the shifted positions in the register. An example of a left-shift of three is given to the right.

$$1101101 << 3 = 1101000$$

Logical AND

This operation is denoted by "&". Given two registers, only the positions where a bit is set in both registers produce a set bit in the resulting register, figure 7. This produces the truth table on the right for each bit in the register.

0	0	0
0	1	0
1	0	0
1	1	1

Logical Inclusive OR / Logical OR

This operation is denoted by "|". Given two registers, all positions with a set bit, regardless of the register, will produce a set bit in the resulting register. This produces the truth table on the right for each bit in the register.

0	0	0
0	1	1
1	0	1
1	1	1

Exclusive OR (XOR)

This operation is denoted by " ^ ". Given two registers only positions with a bit set in one of the register, will produce a set bit in the resulting. This produces the truth table on the right for each bit in the register.

0	0	0
0	1	1
1	0	1
1	1	0

2.4.2 Finite Automaton

As described by Navarro[19], an automaton is a directed cyclic or acyclic graph where every vertex depicts a state $q \in Q$, and edges represent functional transitions $D(q_i, \alpha_j)$ between states and are labelled by $\alpha \in \Sigma \cup \{\epsilon\}$. D associate each state $q \in Q$ with a set $Q' \subset Q$ for its set of edges, $\alpha \in \Sigma \cup \{\epsilon\}$. In the automaton there exists an initial state, $I \in Q$, regular states and final states $F \subseteq Q$. Given this, the automaton is defined by $A = (Q, \Sigma, I, F, D)$.

The automaton is a non-deterministic finite automaton, if there exist more than one transitions from $q \in Q$, given a single label α. If there for every $q \in Q$ exist only one transition for every α it is said to be a deterministic finite automaton. Given figure 3, the left is an NFA, because $D(q_0, \alpha_0) = q_0, q_1$ and $D(q_2, \alpha_1) = q_1, q_3$. Given the Deterministic Finite Automata (DFA), every state $q \in Q$ have only one transition with the same α.

Given an exact searching algorithm, each state has two transitions D. Either a horizontal match transitioning it to the next state, or a mismatch returning the search to I, while the final value of

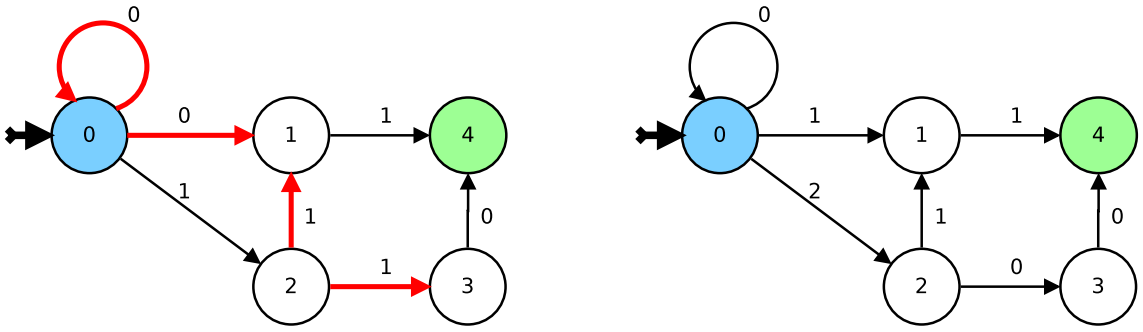

Figure 3: Automaton, NFA on the left, and DFA on the right

the search pattern denotes a final state F marking a match.

In both Non-Deterministic Finite Automaton (NFA) and DFA, transitions from one state to another will set the new state active. In a DFA only one state is active at a time because only one transition exists from any state given a specific α. In NFA, multiple states can be active at the same time, as there can exist more than one transition for α given a current state.

2.4.3 Bit-Parallel Search

Both Baeza-Yates and Gonnet's[3], and Wu and Manber's[5] proposed bit-parallel search algorithms that are based on NFAs. Gonnets search is an exact pattern matching which is given by the NFA in figure 4. Wu and Manber[5] proposed an approximate search based on Baeza-Yates and Gonnets exact search algorithm, giving the NFA in figure 5.

Given the exact search by Baeza-Yates and Gonnet[3] only horizontal transitions are allowed, indicating a direct match from the current state to the next value in the pattern. As shown in figure 4. In this event, both the pattern and search text is incremented, so the next state in the NFA is evaluated against the next $t \in T$.

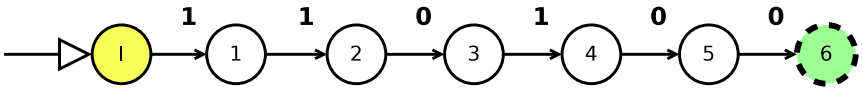

Figure 4: NFA for exact search of pattern 001011

When the search state reaches the end, state 6 in figure 4, a full match is registered. The state of the search is represented as an M bit register R, the pattern length, and based on the algorithm, the currently active states are represented by a set bit or unset bit. Eg. $R = 001001$, represent two partial matches are occurring, and have reached position one of the pattern and position four of the pattern. Example of exact search is shown in section 2.4.4 for a short example of the shift-AND algorithm for exact search, but the update function for the search state is given below.

$$R'_0 \leftarrow ((R_0 << 1)|0^{m-1}1)\&B[t_j]$$

Exact search utilises a single transition representing a direct match. Approximate search implements four transitions to allow K errors to occur. The NFA by Wu and Manber[5] has four transitions from every state until the $F \subseteq Q$ states denoting a match with k errors. The transitions are defined as, match (horizontal), insertion (vertical), substitution (diagonal) and deletion (dashed diagonal).

Given the NFA for approximate search with $K = 2$, figure 5, the marked transitions are defined as follows by Wu and Manber[5], and described by Navarro[19]. Horizontal transitions for a match are the same as for exact search, and both the search text and pattern is advanced. A vertical transition represents an insertion of a symbol into the pattern, and only the text advances. Solid diagonal lines represent substitutions, and both the search text and pattern is advanced. Dashed diagonal lines, ϵ, mean that a character of the pattern is deleted, the pattern is advanced, but not the search text. Finally, the state I allows for an arbitrary number of bits to be skipped until the first transition is made.

The search state of an exact search is represented by a single M bit register that is updated throughout the search. Approximate search organises this as a set of search states. $R^K_{k=0}$, where R_0 denotes the search state without errors, R_1 allows a single error, until $R_{k=K}$ allowing K errors to occur during the search. The two update functions for calculating the new search states are given below, see, for example [20].

$$R'_0 \leftarrow ((R_0 << 1)|0^{m-1}1)\&B[t_j]$$
$$R'_k \leftarrow ((R_k << 1)\&B[t_j])|R_{k-1}|(R_{k-1} << 1)|(R'_{k-1} << 1)|0^{m-1}1$$

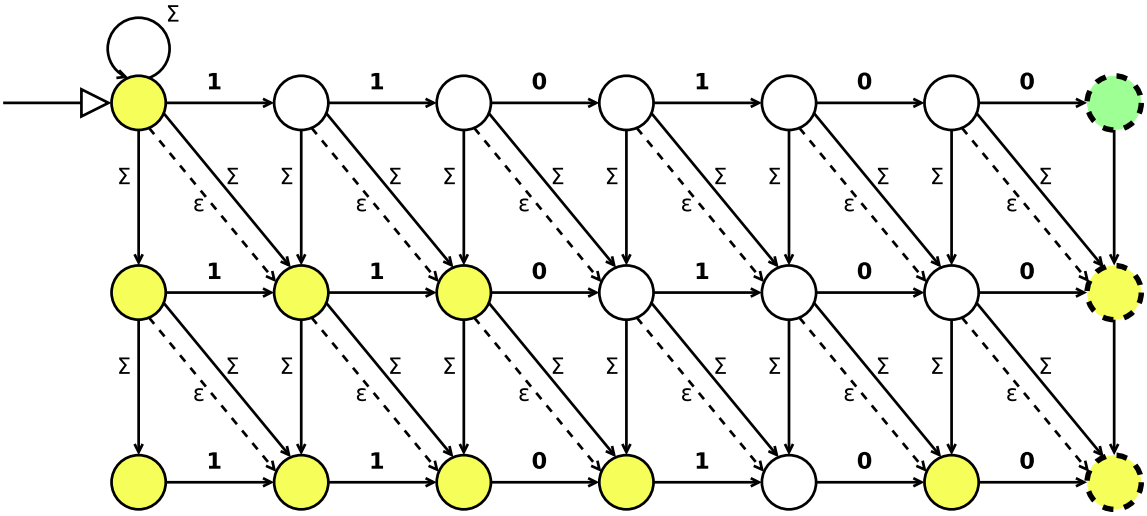

Figure 5: Non-Deterministic Finite Automaton, allowing two errors

The horizontal transition for R_0 is particular and updated the same way as an exact search, while the rest are updated using the second function above and described below. The second function implements all four transitions given for $t_j \in T_j$ as created by Wu and Manber[5] and explained by Navarro[20]. In the list below, R'_{k-1} refers to the new search state value of the search state with one less error than is currently being updated. R_{k-1} Refers to the old value of the search state with one less error allowed. R_k is the non-updated value of current search state, and R'_k is the updated search state value.

1. The horizontal transition is given by $((R_k << 1)\&B[t_j])$ which evaluates the next state for correct match. The shifted search state value is OR-ed with the bit-mask of character t_j. Any matching bits will denote the new search state, R'_k.
2. The vertical transition is given by $R'_k|R_{k-1}$, by adding any missing bits from the non-updated search state with one less error allowed. This represents inserting missing characters into the pattern by advancing the text, but not the pattern.
3. The diagonal substitution transition is given by $R'_k|(R_{k-1} << 1)$, by adding missing bits from

the shifted register representing a non-updated search state with one lower error. This is representing advancing the text and the pattern given the NFA.

4. The diagonal deletion of a character in the pattern is given by $R'_k|(R'_{k-1} << 1)$. This is done by adding any missing bits from the updated search state of one less error to the new search state, through a logical inclusive OR operation.
5. In addition, as given by [20], an additional OR operation is performed. $R'_k|0^{m-1}1$ which accounts for an unrepresented initial state, and simply sets the Least Significant Bit (LSB) of R'_k to 1.

2.4.4 Exact Bit-Parallel Search using Shift-AND

The search algorithm shift-AND by Baeza-Yates and Gonnet[3] for exact search is based on a NFA and utilise register of M length to represent the current search state. The search state represents all currently active states which have a partial or complete match. The NFA for the search is given in figure 4 which deletes all non-match values until an initial match is found.

First, the search word, P, is processed to get the set of prefixes in the alphabet, $\Sigma = \{B[t_0], B[t_1]\}$. Since Σ is binary, the prepossessing is given by, $B[t_0] = \sim P$, the inverse of the pattern, and $B[t_1] = P$. For each $t_j \in T$ being searched, the current search state is updated according to the shift-AND algorithm. Where the shift-AND calculation for an exact match is denoted below.

$$R' = ((R << 1)|0^{m-1}1)\&B[t_j]$$

Given the search text $T = 10010110$ and search word $P = 001011$, the exact shift-AND search updates the search values as follows. Given the initial search state is $R = 0$, the updated search state value is R', with each set bit representing a match or partial match. This R' is then used as the R value for every t_j being searched.

$$\begin{aligned}
T &= 10010110 \\
p &= 001011 \\
B[t_j = 0] &= 110100 \\
B[t_j = 1] &= 001011 \\
R &= 000000
\end{aligned}
\qquad
\begin{aligned}
(R << 1|1) &= 000001 \\
B[t_1 = 0] &= 110100 \\
R' &= 000000 \\
\hline
(R << 1|1) &= 000001 \\
B[t_2 = 1] &= 001011 \\
R' &= 000001 \\
\hline
(R << 1|1) &= 000011 \\
B[t_3 = 1] &= 001011 \\
R' &= 000011
\end{aligned}
\qquad
\begin{aligned}
(R << 1|1) &= 000111 \\
B[t_4 = 0] &= 110100 \\
R' &= 000100 \\
\hline
(R << 1|1) &= 001001 \\
B[t_5 = 1] &= 001011 \\
R' &= 001001 \\
\hline
(R << 1|1) &= 010011 \\
B[t_6 = 0] &= 110100 \\
R' &= 010000
\end{aligned}
\qquad
\begin{aligned}
(R << 1|1) &= 100001 \\
B[t_7 = 0] &= 110100 \\
R' &= \mathbf{100000} \\
\hline
(R << 1|1) &= 000001 \\
B[t_8 = 1] &= 001011 \\
R' &= 000001
\end{aligned}$$

2.4.5 Approximate Row-wise Bit-Parallel Search

As shown in explained in section 2.4.3, the approximate search is built upon the bit-parallel shift-AND search by Wu and Manber[5]. Which is built on top of the exact bit-parallel search algorithm by Baeza-Yates and Gonnet[3]. It is implemented as two separate functions, one for the initial R_0 state, which is calculated in the same manner as an exact search. From this value, the next error level is calculated using the second function which represent all transitions possible from the

current state allowing K errors.

$$R'_k \leftarrow ((R_k << 1)\&B[t_j])|R_{k-1}|(R_{k-1} << 1)|(R'_{k-1} << 1)|0^{m-1}1$$

Even though this is a high-speed search at the hardware level, many calculations have to be performed as given by $R_{ops} = N \times (K + 1)$, which represent the total number of search state updates needed to be performed for an entire approximate search. The example later in this section with T of length $N = 9$ and $K = 2$ errors allowed. This results in 27 R_{ops} to complete the search.

Given the NFA in figure 5 with $T = 100101110$, $p = 001011$ and $K = 2$ errors allowed, we get the following approximate search states.

$$\begin{aligned} T &= 100101110 \\ p &= 001011 \\ B[t_j = 0] &= 110100 \\ B[t_j = 1] &= 001011 \\ R_0 &= 000000 \\ R_1 &= 000001 \\ R_2 &= 000011 \end{aligned}$$

$$\begin{aligned} B[t_1 = 0] &= 110100 \\ R'_0 &= 000000 \\ R'_1 &= 000001 \\ R'_2 &= 000111 \end{aligned}$$

$$\begin{aligned} B[t_2 = 1] &= 001011 \\ R'_0 &= 000001 \\ R'_1 &= 000011 \\ R'_2 &= 001111 \end{aligned}$$

$$\begin{aligned} B[t_3 = 1] &= 001011 \\ R'_0 &= 000011 \\ R'_1 &= 000111 \\ R'_2 &= 001111 \end{aligned}$$

$$\begin{aligned} B[t_4 = 1] &= 001011 \\ R'_0 &= 000011 \\ R'_1 &= 001111 \\ R'_2 &= 011111 \end{aligned}$$

$$\begin{aligned} B[t_5 = 0] &= 110100 \\ R'_0 &= 000100 \\ R'_1 &= 011111 \\ R'_2 &= \mathbf{111111} \end{aligned}$$

$$\begin{aligned} B[t_6 = 1] &= 001011 \\ R'_0 &= 001001 \\ R'_1 &= 011110 \\ R'_2 &= \mathbf{111111} \end{aligned}$$

$$\begin{aligned} B[t_7 = 0] &= 110100 \\ R'_0 &= 010000 \\ R'_1 &= \mathbf{111111} \\ R'_2 &= \mathbf{111111} \end{aligned}$$

$$\begin{aligned} B[t_8 = 0] &= 110100 \\ R'_0 &= \mathbf{100000} \\ R'_1 &= \mathbf{111111} \\ R'_2 &= \mathbf{111111} \end{aligned}$$

$$\begin{aligned} B[t_9 = 1] &= 001011 \\ R'_0 &= 000001 \\ R'_1 &= \mathbf{101011} \\ R'_2 &= \mathbf{111111} \end{aligned}$$

2.4.6 Constrained vs Unconstrained Search

Given the previous sections on correlation attacks, section 2.3.2, the idea of the constrained search was introduced. This section briefly explains the idea of unconstrained and constrained search. These search methods are implemented when calculating the edit distance between two sequences and are used by Golic and Mihaljevic[2] when calculating the constrained Levenshtein distance.

In the NFA for approximate search method by Baeza-Yates and Gonnet[3] the set of operations consists of match, insert, substitute and delete. An unconstrained search using the Levenshtein distance calculates the number of operations required to transform sequence A into B using any number of operations. Restrictions are not imposed on the operations and allow any number of consecutive runs by a single operation.

As opposed to unconstrained search, constrained search can restrict the set of operations and how they are used in the search, with the same goal of transforming A into B. As discussed concerning cryptanalysis, in [2, 1], the constrained Levenshtein distance is calculated based on the constraints proposed. For cryptanalysis, only match, deletion and substitution are allowed operations, since insert operations do not occur in the cipher system and can be disregarded. Restrictions are also set on the number of consecutive runs of deletes to one since the 0/1 clocking BRM skips maximum one bit at a time.

2.5 Central Processing Unit

A Central Processing Unit (CPU)[21] is in its most basic definition is an electronic circuit component which carries out the computer programs set of instructions, in order to complete the desired task. A CPU has a specific set of instructions available, and only a subset of these can be utilised by a computer program. This mapping is handled by the compilation process when generating the computer program from source.

CPUs[21] are found in almost all commercial hardware, from computers to refrigerators, to provide some functionality. CPUs are found as different components based on architecture and functionality required. Generally, CPUs are mainly thought of existing in computers and servers, but System on a Chip (SoC) systems like Arduino employ a Reduced Instruction Set Computer (RISC)[22] to manage the tasks it is being employed for. A RISC employs a reduced instruction set which limits the operations that the CPU can perform. Eg. Arduino and similar devices can be found in drones, toys and devices where space is limited. These RISC have lower capability and speed than regular CPUs found in computers and servers.

A standard commercial CPU consists of between two and eight cores, that are capable of running multiple threads simultaneously. This totals in 4-16 processes being able to run simultaneously within a computer, but with the addition of scheduling[21], the CPU can handle an arbitrary number of processes simultaneously. Until the wait time between scheduled CPU access causes latency.

A CPU is restricted by its instruction set, but also its register size which puts a physical limit of the number of bits it can process. Given a CPU with 64b architecture the maximum register size, or machine word size w, that the CPU can handle in a single operation is 64b. If the search word is longer than w, it is split into $\lceil \frac{M}{w} \rceil$ machine words and stored in separate address space locations, this directly affects the processing time of each search word, as it requires $\lceil \frac{M}{w} \rceil$ times the number of CPU operations to complete the processing of the entire search word.

While RISC has less capability, there exists CPUs with much higher capacity. Commercially available CPU, i.e. Intel Xeon Platinum 8180 processor consists of 28 cores, 56 threads and a clock frequency of 2.5GHz (Max 3.8GHz) and is capable of rendering 512b vectors. A similar CPU was explored by Tran et al. [9], testing its capabilities of processing long patterns using the ARBP search by Wu and Manber[5]. This thesis also utilises the unconstrained Wu and Manber search, but with even longer search words on a regular 4-core, 64b CPU. As Tran explains, two issues arose. First, since $M = 512b$ was the maximum length they could represent per CPU core, longer search words were not tested given it would be split into several parts, causing an efficiency loss during computation. Secondly, when performing a bit-wise left-shift operation, it required 31 additional steps because the 512b vector is represented by 16 32b registers. Tran et al. [9] tested the runtime of implementing the CPU specific 512b vector against a regular 8-core CPU using multithreading.

Tran et al. [9] results were that the 512b CPU vector was superior. Which is easily understood as the natively implemented 512b vector is almost equivalent to a hardware implementation of a 512b register.

To handle large numbers, longer than w bits, in a program running on a CPU special libraries that

support arbitrary-precision arithmetic has to be utilised. E.g. the GNU Multiple Precision Arithmetic Library (GMP) library[23] provides this abstraction layer for the C programming language. It allows allocation and handles many arithmetic and bit-wise operations. It contains three separate data types of different aspects, MPZ, MPF, and MPN. MPZ is for standard signed integers, MPF is for floating point, and MPN is for very low-level operation on positive integers. MPN is meant for low-level applications, has an extensive function library, but suffers from being harder to implement than MPZ. MPZ has a much lower threshold for implementation, a vast library and well documented. However, it does not provide a function for bit-wise left-shift operations, meaning certain support functionality would require custom implementation..

2.6 Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) is a hardware device used to test and implement programmable logic circuits at near hardware-level efficiency. It is based on a semiconductor device with a matrix configuration that is comprised of Configurable Logic Block (CLB) connected via routable connections. An FPGA is usually designed to be infinite programmable, while its counterpart Application-Specific Integrated Circuit (ASIC) is programmed from the manufacturer for a specific task. Given this, an ASIC is more suited for the production of finished designs, while FPGA is used for development or when the system must be re-configurable. This is one of the reasons intricate systems such as signal processing, medical devices, automotive control units and ASIC

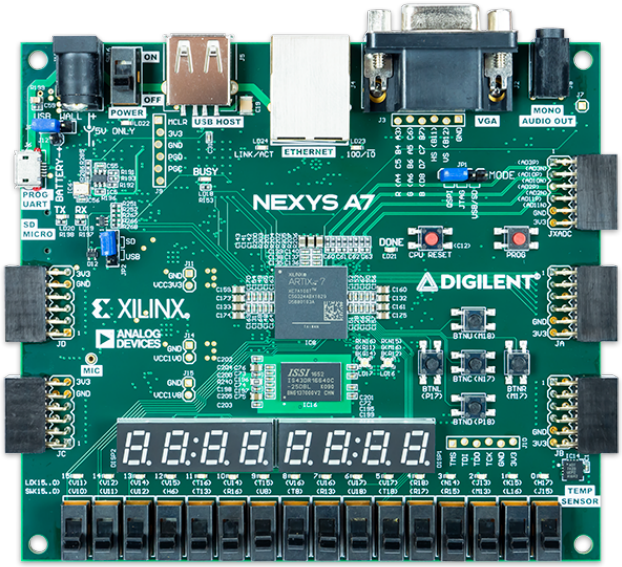

Figure 6: Nexys A7 100T XC7A100T CSG324C Xilinx chip
Source: https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-a7/nexys-a7-top-600.png

prototyping utilise FPGA as they can be reconfigured, reprogrammed, and with relative ease extend its physical capabilities.

As opposed to Central Processing Unit (CPU), the FPGA has a much lower clock frequency and higher price range, but currently, consumer-grade FPGA boards have become affordable with a clock rate of 50-150MHz. Current FPGAs have overall increased capabilities in the form of CLB density, Block-RAM (BRAM), DSP blocks, clock frequency, and external connections, as seen by the Nexys A7 in figure 6. In the case of the Xilinx Artix-7 FPGA chip a single CLB[24] is comprised of two logic slices where each logic slice is comprised of four 6-input Look-Up Table (LUT), four flip-flops[25], carry-chain logic[26] and four additional flip-flops used for latching.

A LUT is a truth table for combinational logic for a set of inputs, and these are used to resolve the combinational logic. A flip-flop[25] is a fundamental component that can store information, e.g. setting a flip-flop means it will hold the value and present it as the output. This is how registers are managed in FPGAs. Carry-chain logic[26] is the basic functionality which makes an FPGA effective in regards to arithmetic operations. Given an arithmetic operation producing a carry, the direct output is sent as output, while the resulting carry is routed through the carry-chain logic to the next logic element in the register. These building blocks are combined into a CLB and routing allows for CLB to be interconnected carrying out complex logic.

Given the programmable interconnects between an FPGAs CLBs, complex hardware systems can be implemented on an FPGA. This inter-connectivity enables one to design independent and parallel circuits to process its work effectively. This is exemplified in figure 7 illustrating a basic logic-AND

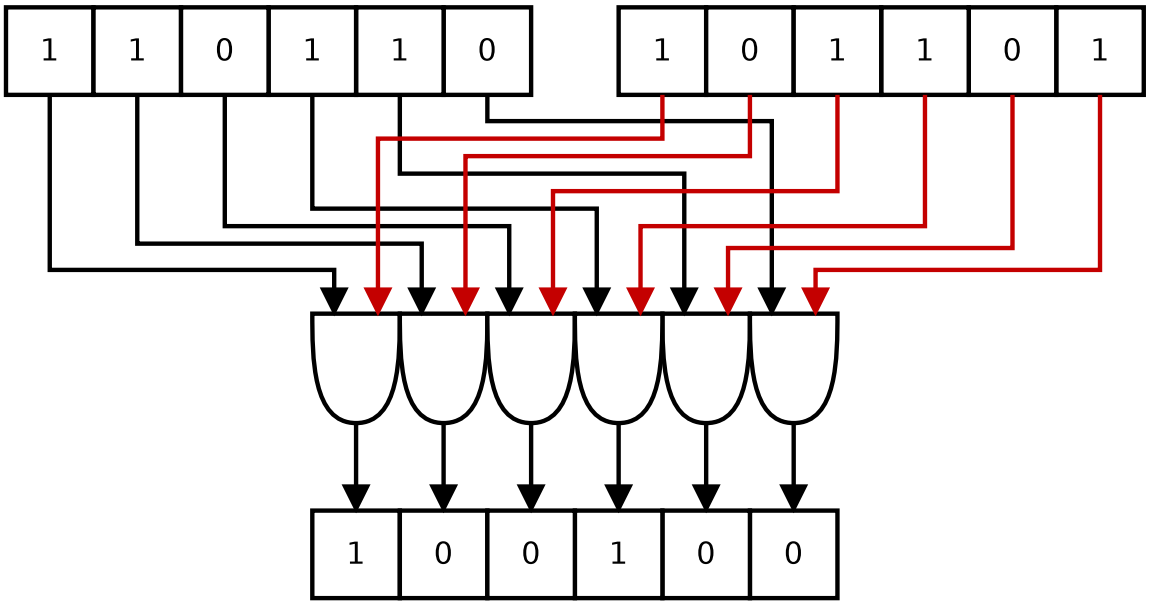

Figure 7: AND operation on two registers

operation. Two 6b registers generate a single 6b output through a bit-wise AND-operation. The FPGAs architecture allows an arbitrary number of bits to be processed simultaneously based on its design, while CPU can only process up to w bits simultaneously based on its physical constraints.

The AND-operation is implemented by using CLBs, but since these are of limited quantity in the FPGA, implementation of Random Access Memory (RAM) is essential aspects of an FPGA. In addition to Double Data Rate 2 (DDR2), BRAM is memory structures on the FPGA dedicated to synchronous memory utilisation. BRAM has the advantage of being very simple, larger storage capacity than CLB, and easily implemented in an FPGA design[27, 28]. It is implemented using a standard design which is interpreted upon synthesis which allocates BRAM instead of CLBs to contain the data. Representing the error table of size $(K+1) \times M$ in CLB is not feasible as it quickly becomes larger than the number of available CLB.

2.6.1 FPGA Development Methodology

In difference to compiling an executable from source code for an operating system, generating a functional program for an FPGA is more complicated and requires many stages as it generates a hardware circuit. Xilinx[29] describes the design flow overview in their documentation. These stages have to be run each time a change occurs in order to generate a functional bit-stream file for the FPGA. Their description is summarised below.

Design Entry This stage constitutes the generation and implementation of the desired design. This is where one generates source code and define constraints, such as timing constraints and pin-out mapping for the FPGA board.

Functional Verification Here the functionality is verified through manual and automated simulation pre and post-synthesis. Before synthesis, it is run a behavioural simulation to verify RTL code and to confirm the design is functional. After programming the FPGA, functional verification can be performed on the implemented design.

Design Synthesis Synthesising the design involves verifying the source code syntax, analysing the design hierarchy for the optimal design and architecture, generating the netlist describing the design.

Design Implementation This is run post-synthesis, which performs the following tasks. Translate, map, place and route the design, and generating the programming file. Translate merges netlists and constraints into a design file. Map fits the design into the FPGAs resources. Place and route, places and routes the design to the timing constraints to create the optimal utilisation of the board. Programming file generation produces a bit-stream file used by the FPGA to run the implemented design.

Timing Verification This is run after map, or place and route and verifies the timing of the implemented design.

Xilinx Device Programming Program the FPGA using the generated bit-stream file produced by the design implementation. It is uploaded through the desired method, usually via JTAG on consumer-grade FPGA boards.

2.6.2 FPGA Design Debugging

Debugging an FPGA is done via many options at different stages and consists of electronic debugging, code validation, simulation and testing. Electronic debugging is the task of verifying and debugging the system after it has been implemented on the FPGA and testing its functionality via measurement tools to verify the operation is correct. E.g. using an oscilloscope to verify the bit-stream and clock frequency or output of a register through debug ports. Simulation and debugging in software is mostly the same thing, this involves running the simulation, as shown in figure 9, and stepping through the code to verify it operates and performs the tasks correctly. Code validation is run throughout the process of synthesis and implementation, and verifies that the code and resulting schematics is functional.

The main debug process is through simulation, but as shown in figure 9 this quickly becomes cluttered. The simulation shows the state of registers plotted along a timeline based on a simulate clock rate and updates the registers based on the synthesised code. However, even though this is a powerful tool, it can be prone to errors as the simulation generates a perfect clock signal which it is not a real-world example and thus may result in errors on hardware. In figure 9 the following data, among others, are shown, the output 16b LED-register, LFSR polynomial, search position, BRAM allocation and BRAM operating status is shown.

2.7 Universal Asynchronous Receiver-Transmitter

Universal asynchronous receiver-transmitter (UART)[30] is a hardware device or program block which implements the conversion and transfer of parallel data over a serial bridge using a specific baud rate. The protocol is straightforward, but have a slow transfer speed with a baud rate of maximum 115200bps, meaning the cost of this protocol is easily understood to be time. USB 2.0, which the Nexys A7 board utilise, has a theoretic transfer speed of 480Mbps, as opposed to UART 115.2kbps. However, the implementation of USB2.0 communication is much more complicated and requires more space on the FPGA. Using Timothy Goddards open source UART implementation[31], both the FPGA requirements and implementation time is shortened severely, but at the cost of data transfer speed. Therefore UART should only be used if the data transfer can be kept minimal or is insignificant to the operation of the system. This thesis only utilises the transmission aspect of the UART protocol because it was not necessary to receive data. Which results in a reduced implementation of the UART protocol.

The UART protocol functions as a system of two communicators, A and B in figure 8, each with an interconnected receiver and transmitter, utilising a common baud rate. The transmitter of A is connected to the receiver of B and vice versa, using a dedicated line.

To send data, a fixed 7-9b register containing data set on the transmitter (A). The transmitter encapsulates the data by pre-pending a single start bit to the data register. Then it appends one or

zero parity bits (yellow), and lastly one or two stop bits (pink). This is then transferred in serial to the receiver at the specified baud rate. The receiver B, then receives the data and reconstructs original data register and presents it to the recipient as a fixed size register.

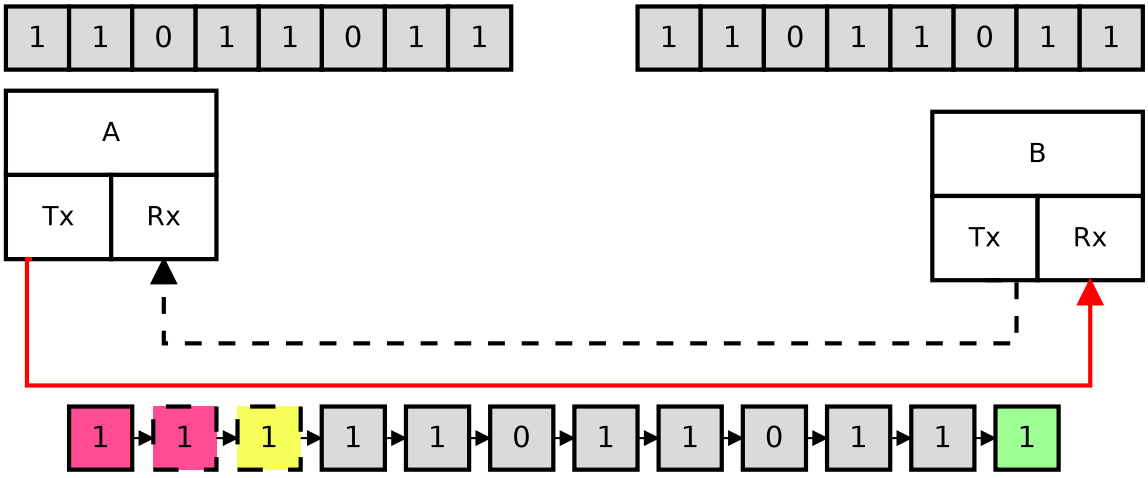

Figure 8: UART protocol

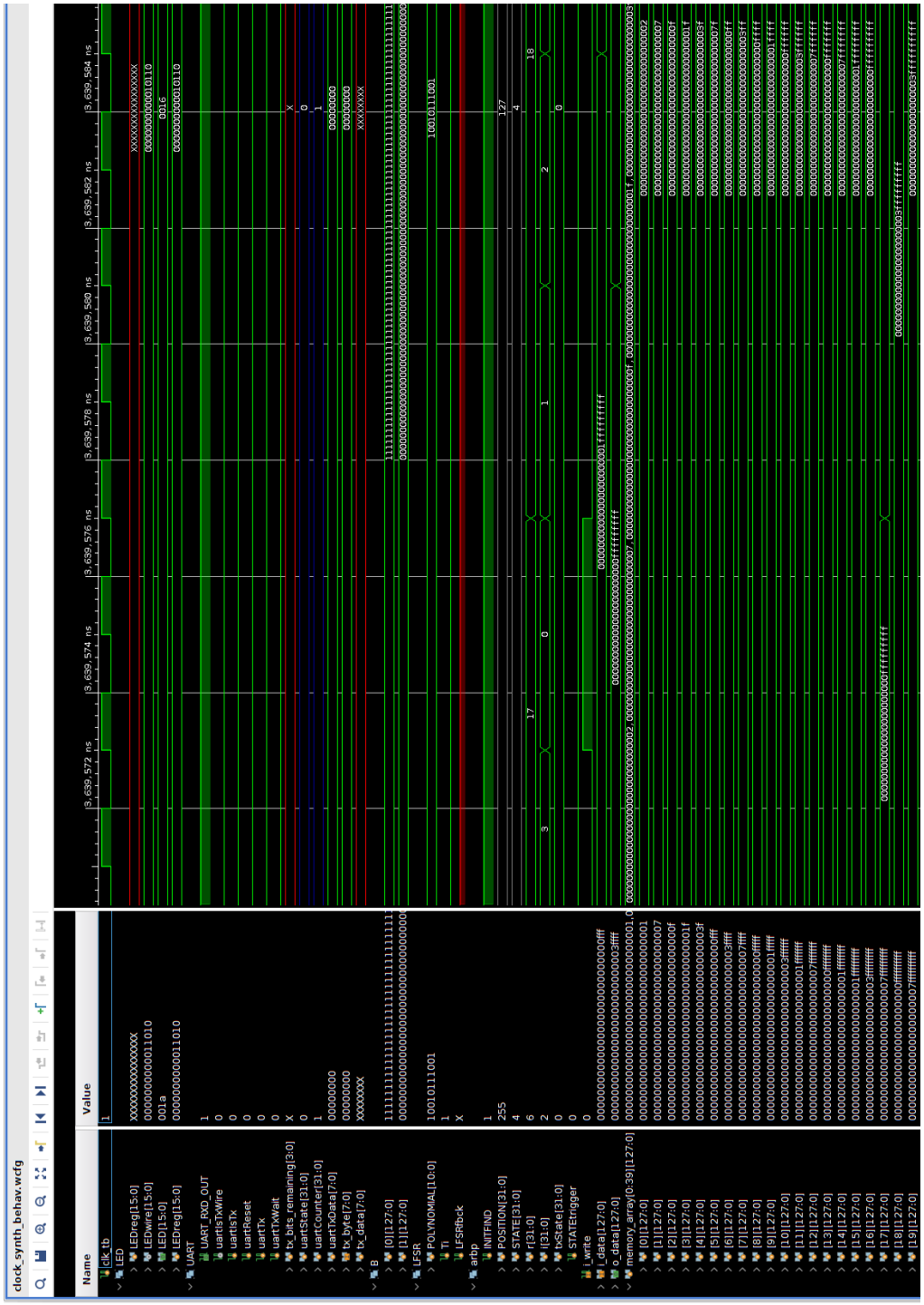

Figure 9: Vivado simulation view. Source: Magnus Øverbø

3 Methodology

3.1 Research methodology

For this thesis, the research methodology employed is a deductive approach[32], where quantitative methods[33] are used in the experimental design and for data analysis. The deductive research approach was used because the hypothesis and research questions are formulated on the basis of existing theory. The research method is then meant to confirm or disprove the hypothesis and provide the foundation for answering the research question through experimentation and subsequent data analysis.

With the experiment, we seek to asses how the subjects dependent variable, processing time, are affected by changes in the independent variables, polynomial, search word length and error threshold. With this in mind, we hypothesise that the Field-Programmable Gate Array (FPGA) implementation will perform better than the Central Processing Unit (CPU), given its ability to process arbitrary length search words without efficiency degradation.

The experimental design was based on a true-experimental design, within-subjects-design[33], where both subjects are given the same treatments and observed. The observations yield numerical data in interval form and are used for statistical analysis to obtain the mean processing time at a comparable level within and between subjects. The data is also used in line graphs to evaluate the collected data sets for each subject and as a method of comparative analysis.

3.2 Experiment

In this thesis, we perform the research using a within-subjects-design design to determine the cause-and-effect relationship between the independent variables and the dependent variable. Each subject, FPGA and CPU, is run through the same baseline of tests to obtain results that are comparable during analysis.

The within-subjects design depicted in table 1 represents the two subjects independent variables which affect the dependent variable, processing time, that is observed and measured. We structured the experiment as follows; for each polynomial, it tests the elements from the set of search word lengths, and for each search word length it test several error thresholds. Each test is then repeated to increase the internal validity of the obtained measurements.

In the design, the treatment issued is a change to any of the independent variables, which is observed by measuring the impact on the dependent variable.

Table 1: Within-Subject Experimental design

Subj	**Variables**			**Time →**		
	Pol	**Search word**	**Error**	**Test and observation**		
CPU	11	$M = \{16, ..1024\}$	$K = \{\frac{1}{4}M, .., , \frac{1}{2}M\}$	Tx → Obs	Obs	Obs
	16			Tx → Obs	Obs	Obs
FPGA	11	$M = \{64, ..2048\}$	$K = \{\frac{1}{4}M, .., , \frac{1}{2}M\}$	Tx → Obs	Obs	Obs
	16			Tx → Obs	Obs	Obs
	20			Tx → Obs	Obs	Obs

3.3 Data analysis

The quantitative data gathered from the experimental phase were analysed as individual subjects by using statistics and graph analysis[34]. The statistical analysis is limited to find the common factor within the subjects tests, and later for comparative analysis between subjects[34].

To assess the subjects individual behaviour and draw conclusions, we plotted the obtained data as line graphs for visual analysis. This method was later employed to perform a comparative analysis between subjects.

Although the number of R_{ops} is the generic base number used for comparison, it is based on three separate variables. This posed an issue since all of them vary throughout the data sets, but each variable may have a different impact on the dependent variable. The experiment design, table 1, allows us to organise the tests into a hierarchy of subclasses then used for plotting and interpreting the graphs. The subjects are represented as individual sets as a combination of the subject and polynomial degree. Further, the search word length plotted along the X-axis with multiple data points representing error thresholds resulting in a time-measurement along the Y-axis.

The goal with the intra-class analysis is to establish how fast the runtime is, based on the common factor mean time per R_{op}. This intra-class analysis is then further used in a comparative inter-class analysis, which is intended to show the overall difference between the two subjects.

3.4 Software Development

Two implementations had to be developed, for CPU and FPGA, ensuring only architectural differences were the contributor to differences in the measured processing time during the experiment phase. Since there was no implementations capable of performing a search of the entire search space existed, we had to implement both. The implementations were developed using an iterative development process with regular milestones of deliveries[35]. During each iteration, only small tasks were worked on to facilitate fast development with fewer errors and refactoring when needed.

The development order is shown in figure 10, where each task consisted of several iterations and milestones. Iterative development is a simple development method suited for small systems and fast development.

In this thesis, we could start with the single building block of implementing the search algorithm and keep adding functionality around it. Before beginning the development, several Proof of Concept (PoC) was created to learn about searching, and the algorithms were implemented and used

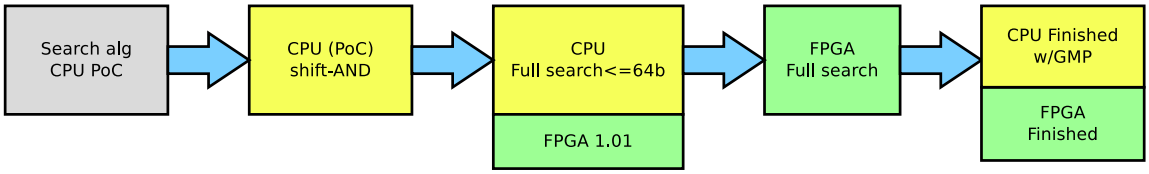

Figure 10: Software development progression

as the starting point for the CPU implementation.

The first building block was implementing the shift-AND algorithm[3]. Once basic functionality was verified, the surrounding system functionality was implemented. Eg. Data collection, looped search, and debug information. Simultaneously the task of learning FPGA design, and the Hardware Design Language (HDL) Verilog was begun via various open sources[36, 25, 37, 38].

Upon completing the CPU implementation, the development of the FPGA implementation began using the CPU as a template. All features were broken down into iterations, then redesigned and implemented for FPGA during iterations.

After the first major version was released, additional features were added to automate tasks, and large portions of the code were improved, e.g. through the implementation of GNU Multiple Precision Arithmetic Library (GMP) MPZ library[23] for CPU, and the implementation of Block-RAM (BRAM) for FPGA.

3.5 Problems and Limitations

3.5.1 Bias

An issue regarding this experimental design is the bias caused by both implementations being developed by the author. In the case of Borgund[6], the FPGA design was developed by the author and the CPU implementation was provided by Prof. Slobodan Petrovic. For this thesis it was not an option as existing systems did not meet the requirements.

With this in mind, the implementations have been created as equal as possible, considering the significant differences between the architectures. It is however imperative that for future use that the CPU and FPGA implementation is evaluated against a third party and analysed to discover any bias.

Even though there is a bias, It is also clearly shown that the results achieved in this thesis are congruent with the findings by previous work, see, for example, [6, 7, 8].

4 Implementation

The following chapters explain the two implementations created for this thesis to evaluate if there is a benefit to utilise Field-Programmable Gate Array (FPGA) for searches. It first describes the C implementation and choices made, followed by an explanation of the FPGA design. Lastly, in this chapter, it discusses the issues and corresponding mitigations employed regarding the implementations and how they compare to each other.

As a starting point and to figure out the requirements for this software, in terms of processing and data acquisition, Several Proof of Concept (PoC) programs were written in C to understand and create a basic implementation of different search algorithms. Both for exact and approximate searches explained by Navarro in [19].

For the implementation used in this thesis, an unconstrained Approximate Row-wise Bit-Parallel (ARBP) search using the shift-AND search algorithm was implemented, given by Baeza-Yates and Gonnet[3]. The choice for using shift-AND search, as opposed to the faster shift-OR, was because the example code provided by Navarro[20] used the shift-AND algorithm and therefore easier to implement. However, in the C implementation, a shift-OR would reduce the processing time because it removes an operation when updating the search state value.

By implementing a multiple precision arithmetic library, the implementation can process arbitrarily large numbers independent of w. The chosen library was the GNU Multiple Precision Arithmetic Library (GMP) MPZ[23], based on the software was written in C on a Linux system.

The FPGA design was designed and written in Verilog using the Vivado IDE, using the predefined board files[39] and FPGA constraint files[40] as provided by DigilentInc. The constraint file creates an abstraction layer by mapping physical pins, e.g. switches, Light Emitting Diodes (LED) and clocks, to variables for use in the source code. The FPGA implementation was designed based on the previously created C implementation but leveraged the hardware advantages that are provided by FPGA.

4.1 CPU Implementation

The logical flow of operations in this software is illustrated in figure 11, while the source code in its entirety is provided in appendix B.1.

In the preamble, this software initialises the variables before using an initial state, a plaintext of length M and a polynomial of degree L to simulate the encryption process, generating the M bit intercepted ciphertext(search word). From this, it creates the M bit prefix values, which ends the preamble phase.

Next, it will iterate through each of the $2^L - 1$ initial states of the clocked Linear Feedback Shift Register (LFSR). For each initial state of the clocked LFSR, it generates the undecimated bit

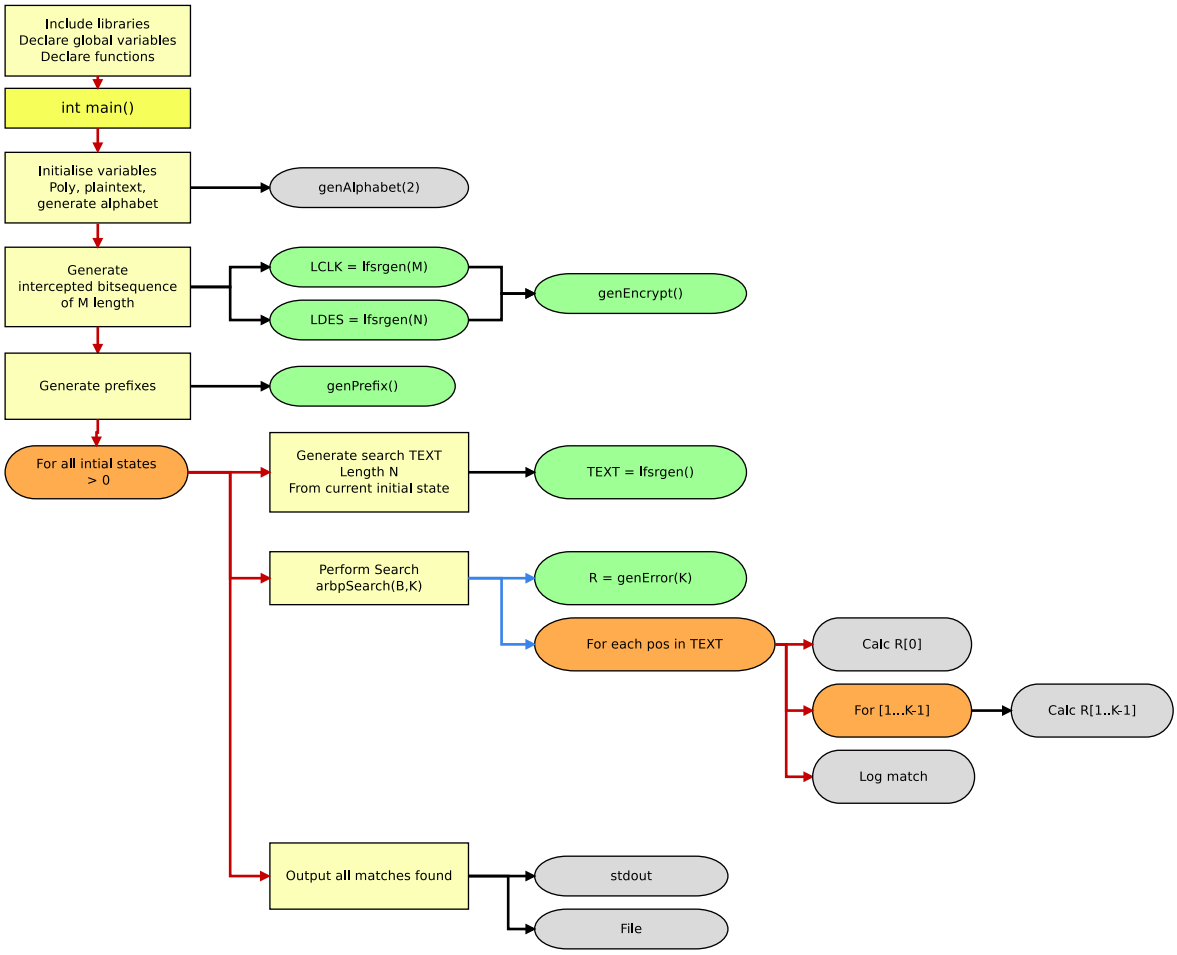

Figure 11: Logic flow diagram of Central Processing Unit (CPU) implementation, written in C

sequence which is N bit long. The length $N = 2M$ as the maximum undecimated length possible is $2M$ due to the 0/1 clocking Binary Rate Multiplier (BRM). Also, an arbitrary number of initial skips is allowed until the first bit is matched.

After generating the search text, the search function will create the initial representation of the search state table allowing K errors, represented by an array R consisting of $K + 1$, M bit representations of the search state value. For each position in the search text, all search state values, $R^K_{k=0}$, are updated before logging the match with least errors to stdout and file.

When it comes to the search algorithm, the most significant difference between FPGA and CPU lies in the implementation of how search state values are updated, see listing 4.1 and 4.2. Given the source code for updating the R_0 search state, several sequential steps have to be performed — each requiring multiple function calls and CPU operations when the search word is longer than w.

```
1     mpz_set( oldR, R[0]);                //Init oldR to cur R[0] (R[i])
2
3     mpz_set(tmp1, R[0]);
4     mpz_lshift( tmp1, m );               //lshift
5     mpz_setbit( tmp1, 0 );               //OR with 1
6     mpz_and( tmp1, tmp1, B[Ti] );        //AND with B[Ti]
7
8     mpz_set( newR, tmp1 );               //Set newR to tmp
9     mpz_set(R[0], newR);                 //Set R[0] to R'[i]
```

Listing 4.1: Calculation of R_0. Exact search

Given the source code 4.1 the number of steps required for calculating the initial R_0, oldR and newR values requires six GMP operations, along with a bit-wise left-shift operation that requires $L + 5$ steps where L is the size of the LFSRs binary representation. This calculation is however without taking into consideration how many actions each GMP function call requires, but a look at some of the primary functions, e.g. mpz_setbit function encompass a large number of operations due to error handling and calculations.

```
1     while( i < K ) {
2       mpz_clear( tmp1 );  mpz_clear(tmp2);
3       mpz_init(tmp1);     mpz_init(tmp2);//reset and initialise temp variables
4
5                                          //Substitute and deletion
6       mpz_ior(tmp2, oldR, newR);         //tmp2 = (oldR|newR)
7       mpz_lshift(tmp2, m);               //tmp2 = <tmp2> << 1
8       mpz_setbit( tmp2, 0 );             //tmp2 = <tmp2> | 1
9       #if !defined INC_INSERT
10                                         //Insertion
11      mpz_ior(tmp2, oldR, tmp2);         //tmp2 = oldR | <tmp2>
12      #endif
13
14      mpz_set(tmp1, R[i]);               //Copy value
15      mpz_lshift(tmp1, m);               //tmp1 = R[i]<<1
16      mpz_and(tmp1, tmp1, B[Ti]);        //tmp1 = <tmp1> & B[Ti]
17
18      mpz_ior(tmp1, tmp1, tmp2);         //tmp1 = <tmp1> | <tmp2>
19
20      mpz_set(newR, tmp1);               //newR = <tmp1>
21      mpz_set(oldR, R[i]);               //Store R[i] for next error
22      mpz_set(R[i], newR);               //R[i] == R'[i]
23
24      i++;                               //Next error
25    }
```

Listing 4.2: Calculation of R_1^K. Approximate search

Following this initial calculation is a set of repeating calculations for updating the remaining K search states values, code 4.2. It requires two left-shift operations and nine GMP operations. Each update of the search state will then require $K(2(L+5)+9)$ operations. So given the polynomial is of degree $L = 11$, $K = 16$ and $M = 64$, the number of operations for a CPU to perform for each position in the search text is.

$$\begin{aligned}(L+5)+(K(2(L+5)+9)) &= (11+5)+16\times((2(11+5))+9)) \\ &= 16+(16\times 41 = 672\end{aligned} \tag{4.1}$$

The result above is arbitrary for several reasons, but illustrates the drawback of CPU processing. First, it does not account for the number of actions performed by the functions in the GMP library, given based on the "mpz_setbit" function it would constitute upward of twenty separate operations for each operation call. Secondly, given that this is a CPU with $w = 64b$, each value larger than w, e.g. search state value, would be split and require several CPU operations to calculate an entire variable. Each of these two drawbacks adds to the processing time, which is why FPGAs perform searches more efficiently as opposed to a CPU.

The implementation does not utilise parallel processing, which would enable much faster run times. Given the assumption that multithreading does not result in unnecessary interrupts, one can assume that it would divide the total processing time by the number of allocated threads. Parallel processing has not been implemented and therefore not verified, but was implemented by Tran et.al[9] with excellent results and should be explored as mentioned in further work 7.2. However, by not implementing parallel processing, both systems in question are running the implemented search in serial, keeping the data comparable.

4.2 FPGA implementation

The logical flow of operations in the finished Field-Programmable Gate Array (FPGA) hardware design is illustrated in figure 12 and described in this section, while the source code is provided in its entirety in appendix B.2. For the development in this thesis, the Digilent Inc Nexys A7 FPGA development board with the Artix-7 100T CSG324 chip was used, as shown in figure 6-

The starting point for the FPGA design was the initial Central Processing Unit (CPU) implementation, but the major difference in architecture resulted in significant differences. These differences are how the search state values are updated and how data is transferred using Universal asynchronous receiver-transmitter (UART) for collection. The FPGA design made up of three separate Verilog modules. The UART module by Tim Goddard[31, 41], the Block-RAM (BRAM) module by, for example InventBoxTutorials[27] and TimeToExplore[28], and the custom module designed to perform the unconstrained Approximate Row-wise Bit-Parallel (ARBP) shift-AND search[3]. The modules are described separately in the following sections, followed by the explanation of the overall hardware design,

4.2.1 BRAM module

The Verilog code for the BRAM module is provided in appendix B.4. This module is a standard Hardware Design Language (HDL) design and is used to avoid storing data in Configurable Logic Block (CLB)s by utilising the dedicated built-in BRAM of the FPGA board. BRAM has a significant advantage because it is dedicated Random Access Memory (RAM) that does not use CLBs and has a larger storage capacity than CLBs, as explained in section 2.6. Dynamic Random Access Memory (DRAM) would be even better as it provides much larger storage space, but BRAM is much easier to implement and utilise. However, given requirements for even larger search words, DRAM must be implemented as the BRAM on Nexys A7 is expended given $M = 4096$ and $K = 1023$.

The BRAM module is implemented with four inputs, one output and an internal two-dimensional

data structure for holding the addressable data. All actions are performed on the input clock rate and are synchronous. The input and output data registers contain the data to be written to and read from the BRAM. These registers are M bit large and hold one search state in each address space. The two remaining inputs are the address register and the action input register which sets if the current operation is to read data from a location in the BRAM or write data to a position in the BRAM.

4.2.2 UART module

The Verilog code for the UART communication module is shown in appendix B.5. Timothy Goddards[31, 41] originally wrote the UART protocol module and was reduced in this thesis to only perform transmission of data to simplify the design. The UART module runs synchronous based on the provided clock signal, one parameter, two inputs, and two outputs which control the data transfer.

The "CLOCK_DIVIDE" parameter specifies how many clock cycles a bit should be transmitted to generate the desired baud rate for UART transfer. The "CLOCK_DIVIDE" number, T_p, is calculated below[31]. An issue with the UART protocol is that it is slow, which is explained later in this section.

$$\begin{aligned} T_p &= \left\lfloor \frac{B_{rate} \times 4}{f} \right\rfloor \\ T_p &= \left\lfloor \frac{115200 \times 4}{\frac{1}{50E+6}} \right\rfloor \rightarrow \left\lfloor \frac{50E+6}{115200 \times 4} \right\rfloor = \lfloor 108.507 \rfloor = \underline{108} \end{aligned} \tag{4.2}$$

The two inputs, besides clock and reset signal, are "TRANSMIT" which signals the module to start transmitting the currently loaded 8b input value, "TX_BYTE". When the transmission has begun, the UART module will set the output register "IS_TRANSMITTING" to 1. Once the transfer has started, the input "TRANSMIT" is no longer needed and the "IS_TRANSMITTING" is set to zero upon finishing the entire 8b transmission. The last output variable "TX" is mapped to the Nexys A7 UART bridge pin which sends the communication across the combined JTAG and UART micro USB port.

Data transfer issue

Using UART for data transferring data is a slow method as it does not support high baud rates. There are also issues regarding transfer distance, but in this implementation it is not an issue as the range is maximum 50cm. However, using this protocol is a drawback as the design runs sequentially, and the data transfer halts further processing until it has finished. An example calculation of potential total transfer time is shown below.

Eg. with a mean of 20 matches per initial state, $f = 50MHz$, 96b transferred per match, and $B_{rate} = 115200$bps results in the following calculation.

$$\begin{aligned} R_{mean} &= 20 \\ T_p &= 108 \\ f &= 50MHz \\ b &= 96 \\ S &= (2^{16} - 1) \end{aligned} \qquad \begin{aligned} T_{tot} &= f \times (S \times b \times T_p \times R_{mean}) \\ T_{tot} &= \frac{1}{50E+6} \times ((2^{16} - 1) \times 96 \times 108 \times 20) \\ T_{tot} &= \frac{13589337600}{50E+6} = \underline{271s} \end{aligned} \tag{4.3}$$

The result is a data transfer time of 271 seconds, which given appendix A table 4, a search of $L = 16$, $M = 1024$ and $K = 511$ has a runtime of 5510s. Data transfer time would in this case add 4.9% additional processing time affecting the measurements considerably.

To minimise the impact transfer time has on the total measurement, the system is designed only to transfer the matches found for the correct undecimated initial state. Doing so provides the ability to measure the whole processing time without the transfer time, affecting the total transfer time significantly. This feature was also applied in the CPU implementation to keep the processing measurements comparable. The result of this is the reduced data transfer time of $0,0041s$ or 0.0001% of the total transfer time given the above example.

4.2.3 Ciphersearch Module

The Verilog code for the communication module is provided in full in appendix B.3 and illustrated in figure 12. This module is made up of three separate ALWAYS blocks and two imported modules where the ALWAYS blocks are designed for deriving a new clock rate, running the actual search and handling the orchestration regarding transferring data across UART. The two instantiated modules are the BRAM and UART module.

Before the search begins, the design is loaded onto the FPGA, which loads the design and sets the initial values of the variables and registers, which applies to all modules in the design before the FPGA starts to run.

First off the clock rate is set, by deriving a clock rate from the FPGAs clock or using the FPGA clock directly in the design. The clock is provided to each module and Verilog ALWAYS block. The second ALWAYS block is the primary system and acts as a state machine that controls the FPGA circuit, which in turns runs the Non-Deterministic Finite Automaton (NFA) based unconstrained ARBP shift-AND search. The final ALWAYS BLOCK is the UART orchestrator which handles the interaction with the UART module. All of these ALWAYS blocks and modules are driven concurrently by the same clock at each clock pulse, but the FPGA design simulates an Finite State Machine (FSM) based on current states triggering different operations accordingly.

Primary "ARBP" ALWAYS block

The main always block, represented by the ARBP block in figure 12, operates as a state machine which performs different tasks based on which state it is in, as described below. This ALWAYS block is responsible for running the central part of the system, controlling and interacting with other modules. This block will generate the M bit search ciphertext, N bit search text, prefixes, search state table for $R^K_{k=0}$, maintain its state and perform the search.

State 0 It starts in state zero, generating the intercepted M bit ciphertext and the prefixes. Which is simulated by generating a 0/1 clocking Binary Rate Multiplier (BRM) from two Linear Feedback Shift Register (LFSR), where the clocking LFSR is used to decimate the clocked LFSRs output sequence. Finally, the output bit is XOR-ed with the plaintext before updating the prefixes. This state is repeated until the desired length M of the ciphertext is achieved. For the remainder of the search, this state does not repeat since the ciphertext is constant.

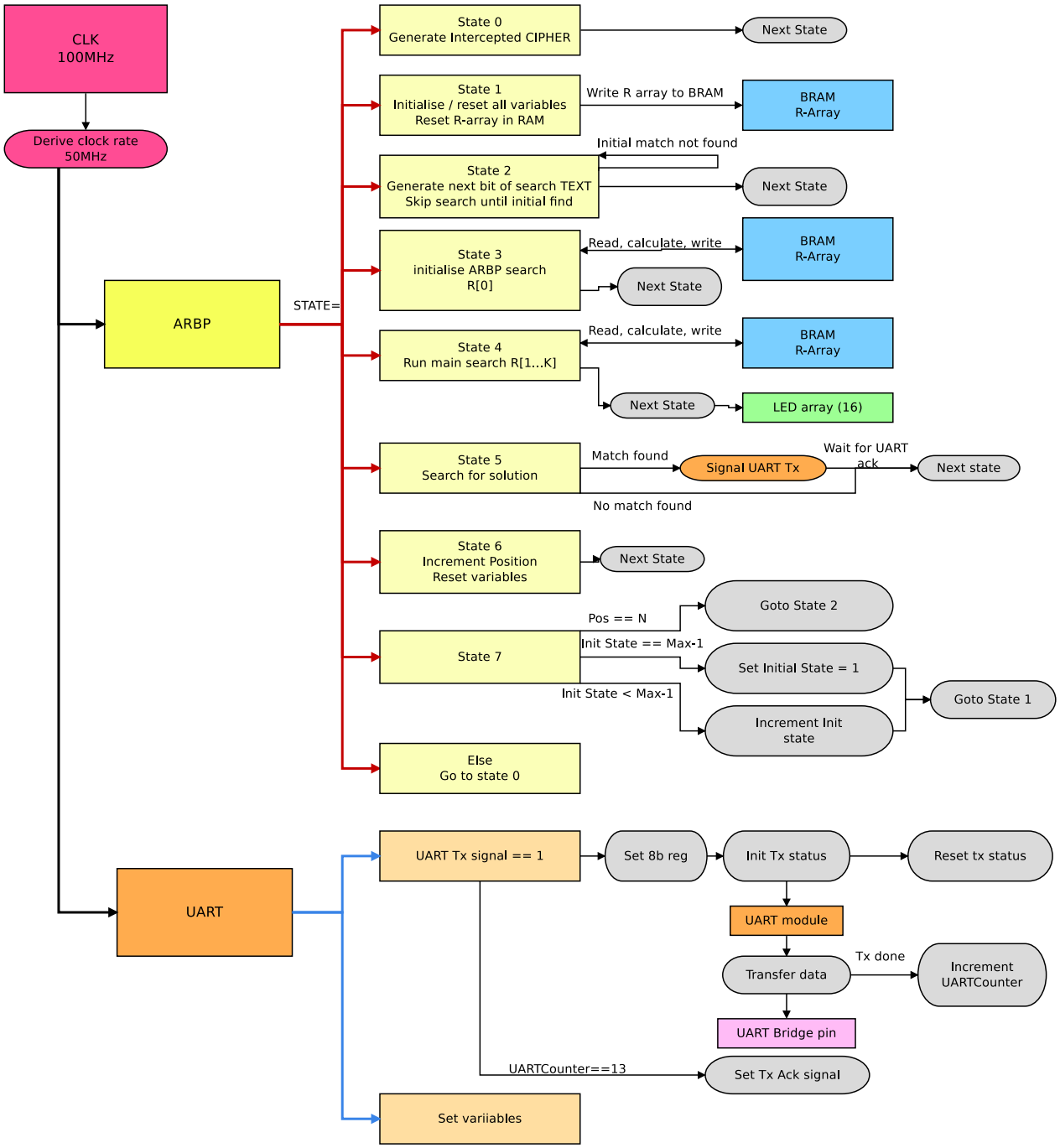

Figure 12: Logic flow diagram of FPGA design

State 1 State one initialises the variables to start a new search, which includes setting the initial state of the LFSR, generating the initial state of the search state table $R_{k=0}^{K}$ in BRAM. This state is triggered every time the initial search vector is changed after a search has finished.

State 2 State two generates the search text one bit at a time. However, during the first run of each search, an arbitrary number of skips is allowed until the bit matches the first prefix value.

State 3 State three performs the update of R_0, allowing only horizontal transitions which is the exact search method by Baeza-Yates and Gonnets[3]. The R_0 value is temporarily stored in R_{old}, as read from R_ODATA_WIRE giving the R_{k-1} value for the next search state.

After calculating the next search state R'_k based on the current search text bit $B[T_i]$, the new value is written to BRAM, and temporarily stored in R_{new}

```
1        Rold             <= R_odata_wire;  //
2                                           //R0 exact search
3        {R_idata, Rnew} <= {2{ (((R_odata_wire<< 1) | 1) & B[Ti]) }};
4        R_write          <= 1;             //
```

Listing 4.3: Calculation of R_0

State 4 State four runs a loop for $R_{k=1}^{K}$ to update the remaining search states. It starts each iteration by writing the current search state of R_k into temporary storage R_{old}, from BRAM. Then it calculates the new value R'_k based on Wu and Manber's[5] algorithm, and writes it into temporary storage for the next search R_{new}. During the calculation, it uses the R_{old} and R_{new} values from the previous search state, as it does not acquire the new values before the next clock pulse.

When all $K+1$ search states values are updated, it will jump to state five if it is at the solution state. Otherwise, it jumps to state six. Even though the cipher system does not utilise insertion in the ciphertext, the search used during experimentation allowed for this transition.

```
1          Rold             <= R_odata_wire;
2          `ifdef INC_INSERT
3          //Insert transtition allowed
4          {R_idata,Rnew}   <= {2{((R_odata_wire<<1)&B[Ti])|Rold|(((Rold|Rnew)<<1)|1)}};
5          `else
6          //Insert transition removed
7          {R_idata,Rnew}   <= {2{((R_odata_wire<<1)&B[Ti])|(((Rold|Rnew)<<1)|1)}};
8          `endif
```

Listing 4.4: Calculation of $R_{k=1}^{K}$

State 5 State five sequentially reads the search state table $R_{k=0}^{K}$ looking for matches, marked by a set Most Significant Bit (MSB). When found it signals the UART ALWAYS block to transfer data by setting the UARTTXWAIT register, Then it waits until the UARTTXWAITACK register is set by the UART ALWAYS block before progressing to the next state.

State 6 State six resets the needed variables and increments the position counter in the search text.

State 7 State 7 will, based on the position counter, jump to state two if it has not reached the end of the search text. Otherwise, it will set the next initial state for the LFSR and jump to state 1 for starting the next search.

4.2.3.1 UART ALWAYS block

This ALWAYS block has the task of orchestrating the transfer of data from FPGA to the host using UART. For each match that occurs, it has to transfer three sets of 32b values in the form of four separate 8b registers, totalling 96b representing the initial state, current position and the lowest error level represented as a 32b integer.

When the primary ALWAYS block raises the UARTTXWAIT register, this block begins loading data into the UART TX_BYTE register and raises the transfer signal. Then it waits for the UART module to raise its transmitting flag, at which point the transfer signal is lowered and waits for the UART modules "transmitting" flag to be lowered. When it is lowered, the counter is incremented, and the cycle repeats itself by loading the next 8b register and wait for its transfer.

When the counter reaches thirteen, the UARTTXWAITACK signal is raised allowing the primary ALWAYS block to continue to its next state. Once the UARTTXWAIT signal is lowered, the UART ALWAYS block is put into wait mode where all registers are reset.

4.3 Testing methodology

4.3.1 FPGA

The Python source code for measuring the FPGA processing time is given in appendix B.2 as source code listing B.6. It starts by reading the 96b output and reconstructs the three original variables; position in the search text, current initial LFSR state, and the minimum error level where the match occurred. Then it will control check the error level value to remove data which has been received incorrectly, due to possible shifts in the byte order when data is collected. If an error occurs, it drops the current data and restarts the search. The error level is used for this because it is the lowest value and therefore faster to error check.

When the first data point is registered in the data set, it records the current time as the start. All subsequent data points are registered, but when the search repeats, the first duplicate match is discarded, and the timestamp logged as the end of the search. After this, it closes the serial connection, calculates the total elapsed search time and writes the data to stdout and file.

During this process, it will not measure the preamble, as explained in section 4.2.3. The measurements start and end at state five of the FPGA design, which avoids the preamble generation in state zero.

A problem with this measurement is the time required to transfer the 96b per match is too high. UART data transfer is generally not recommended because it is slow, but its simplicity made it very easy to implement, and the assessments of the mitigations employed showed it would not affect the overall measurements. The mitigation was to skip the data transfer when it is not searching the solution state and is also done for the CPU implementation. The assessment is discussed in section 4.4.1.

The testing itself was performed by connecting the Nexys A7-100T FPGA board[42] to a Ubuntu laptop using the combined JTAG, power and UART USB port. The Ubuntu system was set up with Python3, PySerial (Python library[43]), and the Vivado suite installed for loading the generated bit files on to the FPGA. Once the bit file is loaded onto the FPGA, the design starts running. The python script is then run until it has finished collecting the entire search and outputs its data.

The Nexys A7-100T FPGA board used for testing and development include the Artix-7 XC7a100T-CSG324C chip, 100MHz clock rate, 4860kb BRAM and 15850 logic slices each with four 6-input Look-Up Table (LUT)s and eight flip-flops[42].

4.3.2 CPU

The measurements for the CPU implementation showed early that the time requirements would be quite different from the FPGA. Therefore required more reliable hardware, which is the reason for running the CPU implementation on server hardware where effects from outside forces were monitored, and the test could run for an arbitrary amount of time.

System specifications for the test system were the following. A Virtual Machine (VM) running Ubuntu 18.04.2 server with 10GB RAM, and 4 CPU cores allocated were set up with git for repository management, GNU Compiler Collection (GCC) for compiling the C program, and the GNU Multiple Precision Arithmetic Library (GMP) library. The physical virtualisation server it was set up on was a Dell R320 with 92GB DDR3 RAM, single 2.4GHz 6-core CPU (Intel Xeon CPU E5-2430L v2 @ 2.40GHz), which ran Ubuntu 18.04 using KVM/QEMU as the virtualisation platform.

Several other VMs was running on the virtualisation platform, which is why the system resources were monitored during runtime, to verify that the performance was unaffected by external factors. Since the CPU implementation only runs on a single thread, three tests could be done simultaneously accounting for any new processes to use the last core.

The processing time was measured by setting the parameters in the source code, compile it and run it inside a screen session. The compiled executable was executed with the time command[44] to measure the total runtime.

Using the time program does not affect the executables processing time as it forks off the executable as a child process and only holds a handle to it. This method enables it to run inquiries regarding the process and log the metadata. Since the executable was instantiated as a separate process, it is not affected by the time program, in addition to the VM having four cores available.

As mentioned in the previous section and described in section 4.4.1, the CPU measurements contains the entire runtime, including the preamble. This issue is discussed in section 4.4.2, where it shows that the time required to generate the preamble is insignificant compared to the total processing time.

4.4 Assessment of variables in data collection

As mentioned in the previous section, 4.3, this section explores specific issues which may affect the data set used for measurements. These issues involve how the FPGA and CPU have reduced the output to only the solution state and how the preamble affects the CPU processing time.

4.4.1 Data transfer / Output timing issue

As discussed in the section on UART in sections 2.7 and 4.2.2, the UART protocol is slow and do not have clock synchronisation. The first is also true for the CPU when it writes to stdout. To minimise the impact on the total processing time, given different polynomial degrees, search word lengths and number of errors allowed. Both implementations have minimised its output to only send the data for the solution state of the clocked LFSR, reducing the impact that this variable has on the total processing time, as depicted in section 4.2.2.

Given the data set in appendix A table 4 of $L = 16$, $M = 1024$ and $K = 511$, with an approximate number of matches per initial state rounded down to 1500. This minimisation of output reduces the processing time from 20384s(eq. 4.4) to 0.311s(eq. 4.5). The time used to transfer the reduced output is a minuscule variable in the data collection as it makes up 0.0056% of the total processing time.

$$\begin{aligned} R_{mean} &= 1500 \\ T_p &= 108 \\ f &= 50MHz \\ b &= 96 \\ S &= (2^{16} - 1) \end{aligned} \qquad \begin{aligned} T_{tot} &= f \times (S \times b \times T_p \times R_{mean}) \\ T_{tot} &= \frac{1}{50E+6} \times ((2^{16} - 1) \times 96 \times 108 \times 1500) \\ T_{tot} &= \frac{1.0192E+12}{50E+6} \\ T_{tot} &= \underline{20384.006s => 05:39:44} \end{aligned} \tag{4.4}$$

$$\begin{aligned} R_{mean} &= 1500 \\ T_p &= 108 \\ f &= 50MHz \\ b &= 96 \\ S &= 1 \end{aligned} \qquad \begin{aligned} T_{tot} &= f \times (S \times b \times T_p \times R_{mean}) \\ T_{tot} &= \frac{1}{50E+6} \times (1 \times 96 \times 108 \times 1500) \\ T_{tot} &= \frac{1 \times 15552000}{50E+6} \\ T_{tot} &= \underline{0.311s} \end{aligned} \tag{4.5}$$

4.4.2 Preamble generation

Preamble generation time is the time required to simulate the encryption process obtain the intercepted ciphertext, and then generate the alphabet prefixes, which encompasses the entire initialisation process described in section 2.4.5, regarding ciphertext generation.

For the FPGA, the preamble is disregarded as it is not part of the measurement, as explained in section 4.3. However, for the CPU implementation, this preamble generation is still minuscule. As shown in appendix A table 7, this epoch is extremely short even at very large search words.

Table 7 contains measurements for the preamble time for two separate polynomials $L = 11$ and $L = 16$, for $M = [64...32768]$. The highest comparable value is $M = 1024$ for the collected data set, but the preamble generation time was measured up to $M = 32768$. Figure 13 shows the graph of how long the preamble generation time takes based on the search word length.

At $L = 11$ and $M = 1024$ the preamble requires 0.01 seconds to generate, and given $L = 16$ the preamble requires 0.016 seconds to generate. Compared to the total processing runtime for the same values we get the percentage of time spent on preamble generation in equation 4.6 and 4.7. For the measured data in equation 4.6 we get the worst case preamble time for $L = 11$ is 0,0001%

of the total processing time, making the impact insignificant. Even at the lowest measurements, equation 4.8 with $M = 64$, is too low to have a significant impact on the data set with 0.044% given $L = 11$ and $M = 64$.

$$\begin{aligned} T_{11,1024} &= 15036 \\ T_{\%11,1024} &= \frac{0.01}{150.36} \\ T_{\%11,1024} &= \underline{0.0001\%} \end{aligned} \tag{4.6}$$

$$\begin{aligned} T_{16,1024} &= 541101 \\ T_{\%16,1024} &= \frac{0.016}{5411.01} \\ T_{\%16,1024} &= \underline{0.000003\%} \end{aligned} \tag{4.7}$$

$$\begin{aligned} T_{11,64} &= 4.57 \\ T_{\%11,64} &= \frac{0.002}{0.0457} \\ T_{\%11,64} &= \underline{0,044\%} \end{aligned} \tag{4.8}$$

$$\begin{aligned} T_{16,64} &= 140.86 \\ T_{\%16,64} &= \frac{0.003}{1.4086} \\ T_{\%16,64} &= \underline{0.0021\%} \end{aligned} \tag{4.9}$$

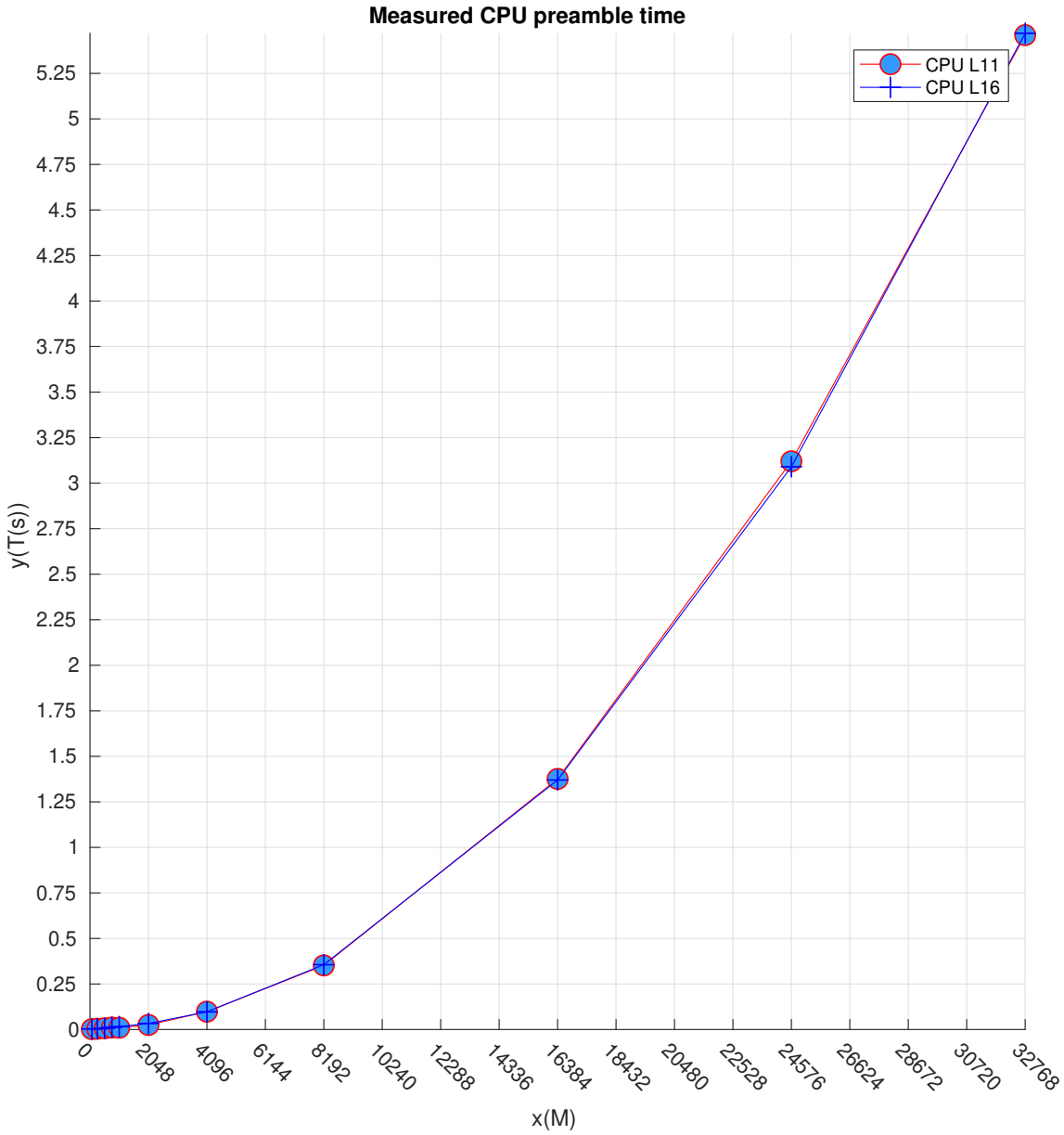

Figure 13: Time required for generating the preamble on CPU

5 Results

5.1 Data gathering

The data points are collected from a single measurement using the parameters described in section 3.2, but data points under half an hour was verified for consistency during testing. Data points of more than half and hour was only measured once in order to obtain a broader set of data points within the time frame.

The data set collected is given in appendix A table 4 and 6. A subset of this data is plotted in figure 14, and the entire data set is illustrated in figure 16.

This data set contain measurements for Field-Programmable Gate Array (FPGA) using three separate polynomials of varying degrees on FPGA, but only two polynomials was used for the CPU implementation. All of these polynomials are shorter than commonly used in current encryption schemes, but larger polynomials would result in exponentially higher processing times and therefore avoided in this thesis.

To generate the search word, the cipher system, figure 2, was simulated as part of the preamble, see section 2.1. In this process several parameters was set for the test and are listed in table 2. The resulting M bit, decimated output was then XOR-ed with the plaintext, for which the collected data is a zero-noise plaintext. This poses an issue with regards to the match statistic only being based on delete actions, and not incorporating substitutions. The reason for this was to ensure the best possible match statistics when performing the search, in order for the correct initial state to be included in the findings. Implementing noise in the plaintext and performing a complete study of statistics regarding matches for this implementation of was outside the scope of the thesis.

Implementing noise into the plaintext, would not affect the processing time directly, but instead affect the number of matches which would occur. Which in return forces one to adjust the error threshold K which would impact the search, but with data collected for $K = \frac{1}{2}M$ the required error level should be captured.

The only changes made for each data collection, within the same polynomial degree, of the implementations was the desired search word length M and error threshold value K. During the tests, the error threshold was primarily set at $\frac{1}{4}M$ and $\frac{1}{2}M$ with a third measurement in between where possible.

The polynomials below was selected from a list of primitive polynomials for $GF(2^L)$, provided by [13], which is derived from [12].

Table 2: Parameters used for preamble and search

L	Polynomial	Clocking LFSR Init State	Clocked LFSR Init state	Plaintext
11	$x^{11}+x^8+x^6+x^5+x^4+x^1+1$	300	1024	$\emptyset$
16	$x^{16}+x^9+x^8+x^7+x^6+x^4+x^3+x^2+1$	300	1024	$\emptyset$
20	$x^{20}+x^9+x^5+x^3+1$	300	1024	$\emptyset$

5.2 Data set

A subset of the data set with run-times higher than $5E+4$ being removed, is illustrated in figure 14 and shows the captured data for six separate data sets. The data sets represented are the Central Processing Unit (CPU) implementation with two different polynomials and four data sets for the FPGA implementation, where the polynomial of degree twenty was run at two clock rates. Figure 14 shows that both FPGA and CPU are increasing exponentially in total processing time. However these are growing at vastly different rates which is due to the different implementations and hardware platform they are running on. This is explored further in chapter 6.

Figure 14 plots the search word length, with multiple threshold values K along the X-axis. This give multiple measurements per point on the X-axis for each polynomial. Then the total time used by a specific M, K combination is plotted along the Y-axis, which includes the preamble generation time for CPU and data transfer time for FPGA.

As clearly shown in figure 14 the CPU will grow at a much faster rate, which increases further given a polynomial of a higher degree. The same tendency is seen for the FPGA, but at a much slower rate.

The two polynomials tested on the CPU are of degree 11 and 16. $L = 11$ was only tested for search words up to 1024b, as the bit sequences generated would start repeating itself given longer sequences of the search text, given $N = 2M$. $L = 16$ was also tested for up to $M = 1024b$, but are omitted in figure 14.

The data sets was collected using regularly spaced intervals up to $M = 512b$ because the processing time began exceeding one hour for polynomials of degree eleven, and twenty hours for degree sixteen. Above 512b the increments are increasing between measurements up to the maximum of 2048b.

The number of data points captured for CPU is much greater than FPGA, because the CPU processing time is not constant, but increases depending on the search word length. Due to this it required more data points during analysis.

FPGA was tested with three polynomials, because the processing time allowed for this to be done for search words up to 512b. This polynomial of degree twenty was also used to test the FPGA implementation using a higher clock rate. This was in order to test the hypothesis that changes in the FPGA clock rate is directly affecting the processing time.

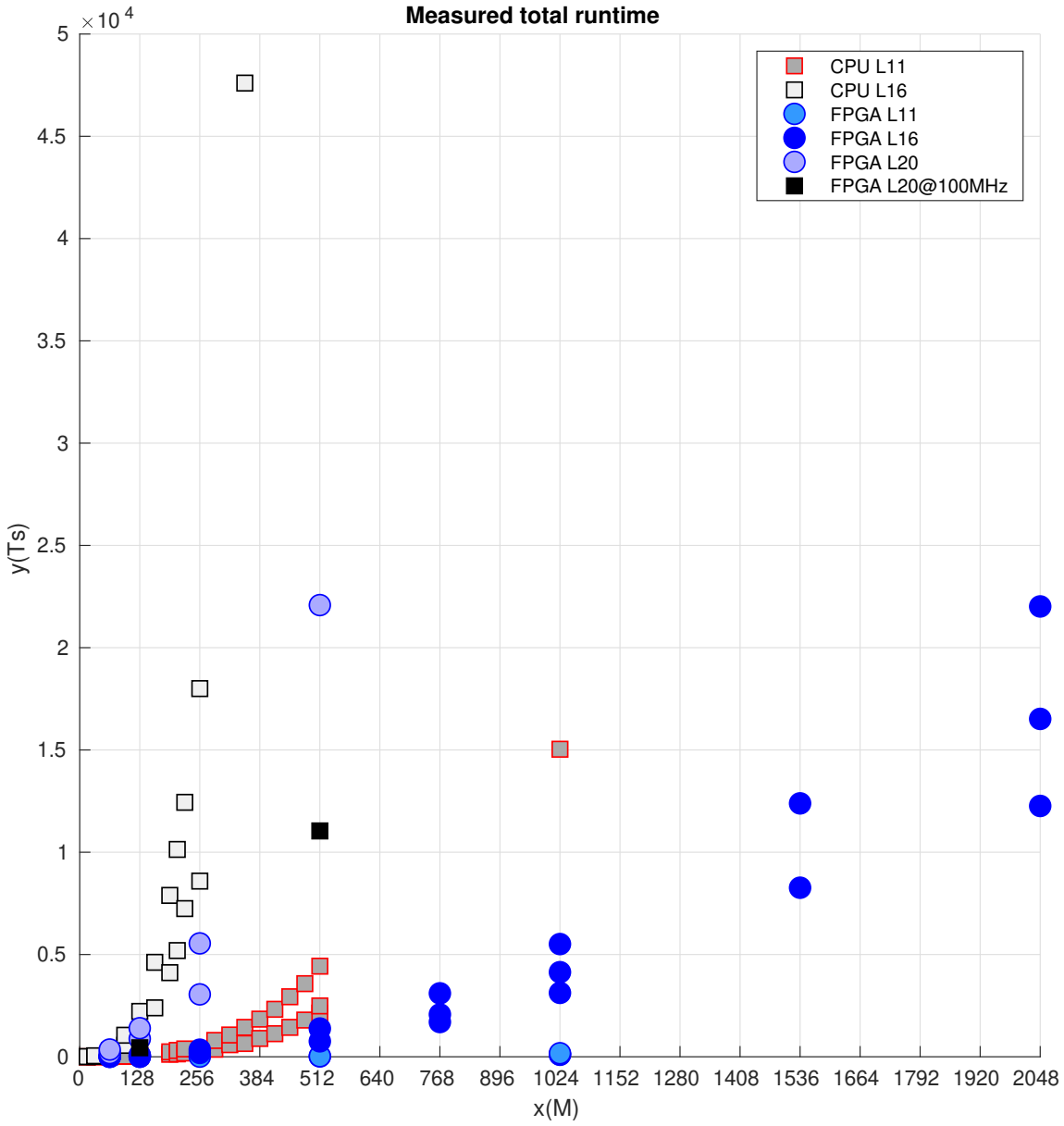

Figure 14: Total processing time plotted by architecture and polynomial degree. With search word length M on the X-axis, with multiple errors K.

5.2.1 Data set Internal validity

Preferably all observations is made multiple times in order to reduce errors within a set of measurements caused by reliability or errors of a single observations. However, because the time require-

ment for capturing even a single measurement on FPGA and CPU require several hours to complete, this was not possible. Several measurements regarding CPU with $M = 1024b$ would each require 150 hours. For CPU it is possible to capture this data given time since multiple data captures can be performed simultaneously. The same on FPGA would not be feasible, as it can only run one search at a time, which require up to 12 hours to complete. Even though the time frame did not allow for collecting a more comprehensive data set, possible future research should seek to capture an even larger data set.

6 Analysis

6.1 Foundation for analysis

In order to compare the measurements a common understanding of how many calculations has to be performed in total is fundamental. This is the foundation for how much work has to be performed for a specific search by the Central Processing Unit (CPU) or the Field-Programmable Gate Array (FPGA). This is discussed in section 2.4.5 which gives the calculation of R_{ops} and denoted below. The number of operations required to perform a search is illustrated in figure 15.

$$R_{ops} = (2^L - 1) \times N \times (K + 1)$$

The figure shows that the number of operations within a single polynomial increases linearly when either K, the number of errors allowed, or M, the search word length, is static. This is show as the linear plot which has a static K value, When both M and K increase the growth is exponential. This is shown with the two exponential plots, the lowest exponential plot is using a $K = \frac{1}{4}K$, and the other $K = \frac{1}{2}M$. The latter is the case for this thesis, as the number of errors must be increased along with the search word size in order to get matches.

This is because of the non-linear function used introduces between zero and M number of skips and this requires the error threshold to be increased in relation with the search word length. Otherwise one will not get any matches. During testing the minimum number of errors required looks to be approximately $\frac{1}{4}M$. Which is due to the testing being performed with a zero noise plaintext and a primitive polynomial. This causes the intercepted ciphertext to only contain deletes operations at a rate of approximately 0.5, and not any substitutions.

6.1.1 Further analysis

Given the R_{ops} number, further analysis will use this as the basis for interpreting the processing time as it will give the approximate time used to perform each separate calculation. Even though there are other task happening intermittently during the search, the vast majority of runtime time is spent on during these repetitive operations. As explained in section 4.4.2, the time used to generate preamble and transferring data is reduced and has an insignificant impact of the total runtime when the search word is large enough.

In addition to using the number of R_{ops} to compare the processing time, the clock rates of the CPU and FPGA is vastly different will be used for comparison

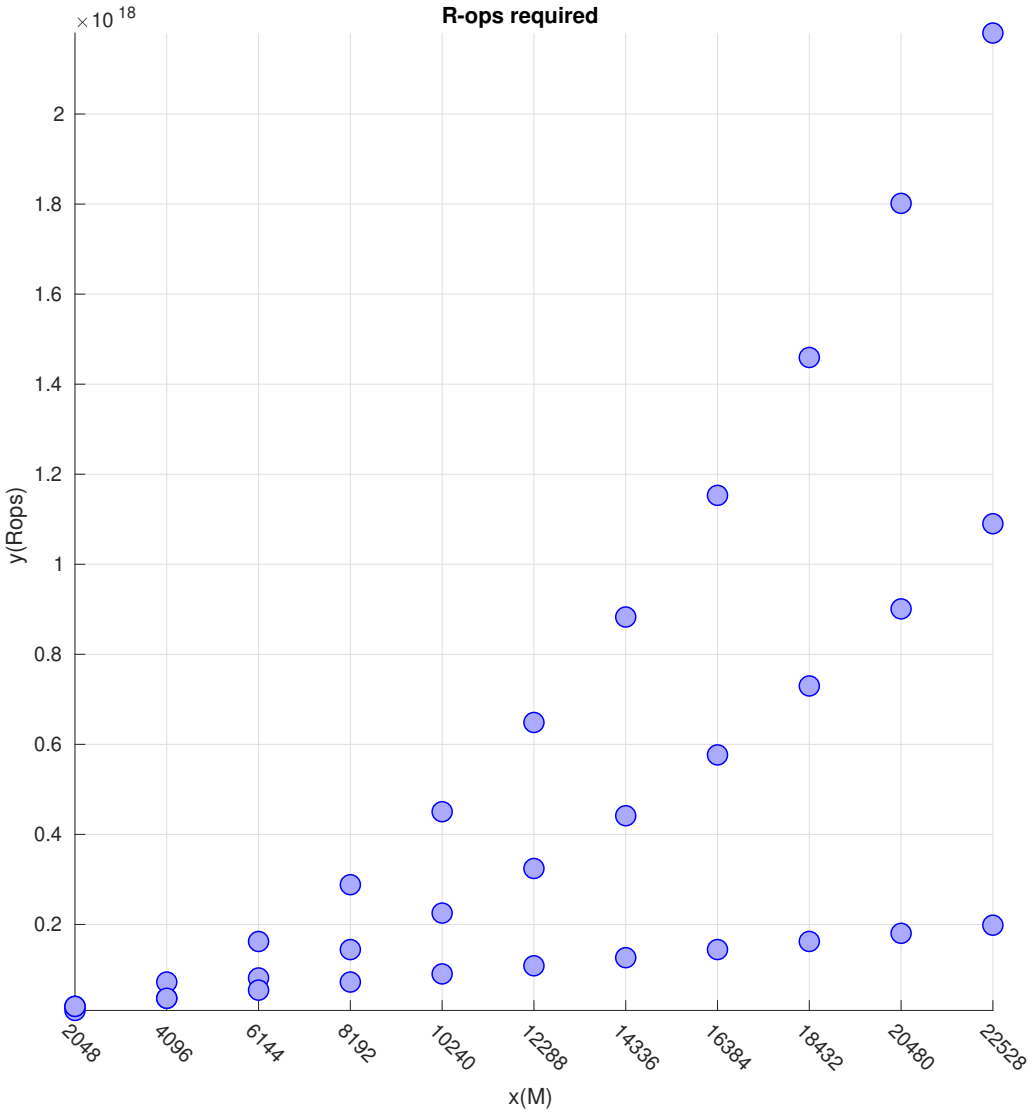

Figure 15: R_{ops} required for search w/M and $K \in \{1024, \frac{1}{4}M, \frac{1}{2}M\}$.

6.2 Initial analysis of data

Given figure 14 in section 5.2 shows a subset of the entire data set, with a threshold value on the Y-axis. It shows clearly that the CPU has quite an extreme exponential growth. Making it unsuitable in a practical application where any requirement for speed is needed, even at very low polynomial degrees with short search words. The figure also suggest that the FPGA has an exponential growth at larger polynomials of degree 20, as opposed to degree 11 where it is not visually recognisable in

the figure.

In both scenarios the data shows that it would require large amounts of time using the two implementations tested along with the equipment used during the thesis. However, there are several improvements and methods which could improve both implementations significantly and is discussed later in this chapter. This include multithreading and distributed computing for the CPU implementation and for FPGA investing in an enterprise level FPGA or even implementing the design in actual hardware.

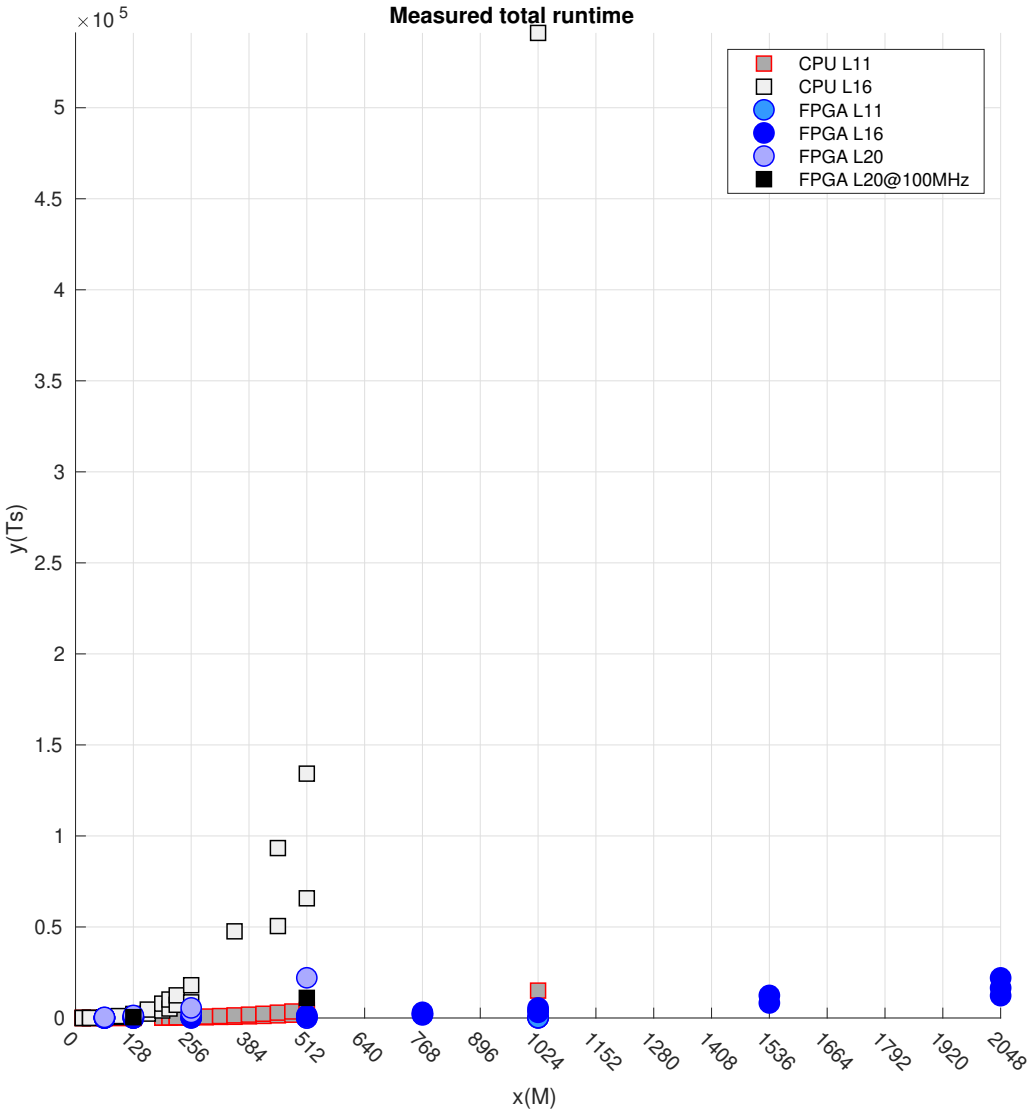

Figure 16: Entire runtime for FPGA and CPU with M bit search word and K bit allowed errors.

6.3 FPGA runtime analysis

This section analyses the FPGA processing time and how it is affected by the changes in the polynomial, search word, and error threshold.

When capturing the data, the time spent transmitting data across Universal asynchronous receiver-transmitter (UART) is not subtracted from the processing time. But as discussed in section 4.2.2 it has an insignificant impact on the overall processing time, when the search word becomes larger than 256 bit. This is shown in figure 17.

Figure 17 plots the data using the search word length along the X-axis with multiple data points illustrating the different error thresholds used in testing. On the Y-axis it plots the mean time used process a single search state value calculation (R_{op}). The R_{ops} number is explained in section 2.4.5 and represent the total number of calculations which has to be performed in order to complete an entire search.

The mean time used per R_{op} is extrapolated from the measured data total runtime. It illustrates how long a single calculation takes for the specific combination of polynomial degree, search word length and error threshold on that platform. Looking at figure 17 it appears to show a distribution following an exponential decay, but given certain issues which could affect and skew the relation between R_{ops} and other tasks could cause shorter search words to have inaccurate measurements. These issue relate to search text generation, Block-RAM (BRAM) IO operations, and UART data transfer.

A second part of this is that we know the timing for each set operations performed on the FPGA is equal in time because all actions has to be completed within a single clock pulse. This gives a specific timing for each operation.

Since the Nexys A7 has a clock rate of $f = 100$MHz and the implementation ran at $f = 50$MHz, means each FPGA operation specifically used $\frac{1}{50E+6}$ second to complete a single set of operations. Either it being generating a new bit in the search text, performing $\frac{1}{108}$ of a UART transfer, or calculate an entire R_{op} value.

Given that the R_{op} mean time in figure 17 stabilises at $80E - 9$ gives us the ability to infer that on average when the search word is larger than 256b, it will require approximately four operations to calculate a single R_{op}. Giving us a quotient to use for estimating FPGA runtime.

$$\frac{\text{mean(T}/R_{op})}{f} \rightarrow \frac{80E - 9}{\frac{1}{50E+6}} = 80E - 9 \times 50E + 6 = 4 \tag{6.1}$$

This quotient holds true independent of clock rate used to run the implementation, as discussed further in section 6.6 on FPGA runtime estimation. The reason for this is that FPGA does not change its operating requirements regardless of the variables in place. Because the design operates on basic bitwise operations, eg left-shift, OR, AND and XOR, it does not require complex Look-Up Table (LUT)s for calculating addition of large registers which could slow down the processing time.

The only operating requirement for this FPGA design is that it is able to complete each set of operations within the current clock pulse. If the design is unable to complete its current set of operations, the current operations will stop and leave registers in their current. Given the design

is able to complete the set of operations, the search word, error threshold and polynomial can be of arbitrary size. The FPGA will not require splitting a register in multiple sequential steps to be able to compute the result of an bitwise operation. Resulting in a consistent processing time which appears to be constant. Given the plotted data in figure 17 and table 4 in appendix A.

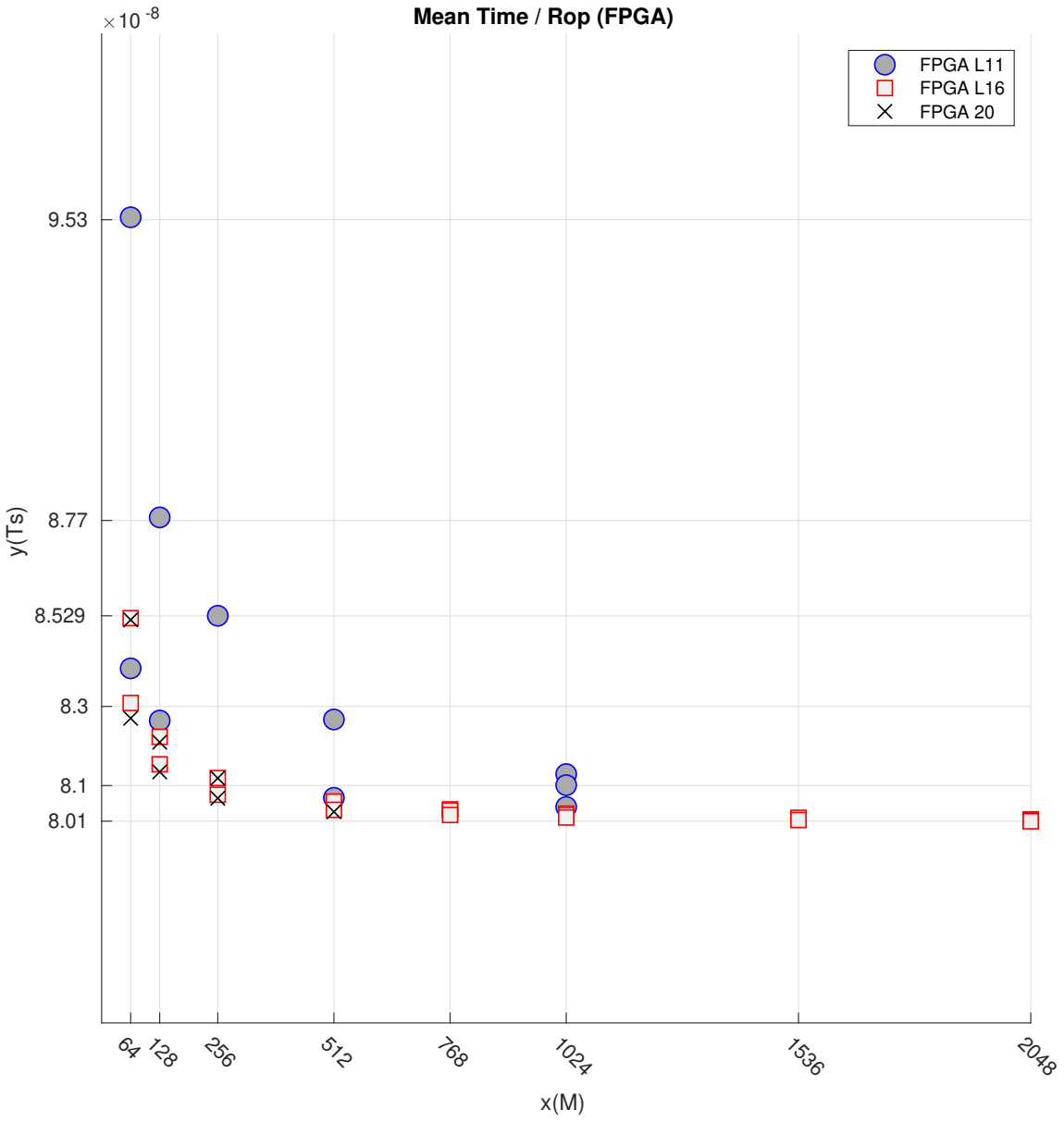

Figure 17: Mean time for FPGA to perform each R_{op} calculation based on an M bit search word

Given that the time requirement stays constant independent of search word length, the error threshold and the polynomial, the total processing time is only dependent on the total number of operations required to perform the entire search and the FPGAs clock rate. The result is that the total processing time will have an growth rate of $R_{ops} \times (4f)$. Where the R_{ops} grows linearly when the search word length or error threshold is consistent. When both is increased the growth rate is exponential. Given figure 14 we see that this is the case for FPGA when the polynomial degree is twenty and sixteen as the growth rate is exponential.

The last remarks in regards to processing time for FPGA is the comparison of data sets gathered for FPGA with a polynomial of degree twenty at 50MHz and 100MHz clock rate. Where we can see that two data sets shows that the doubling of the FPGAs clock rate reduces the processing time by half. Eg. with $L = 20$, $M = 512$ and $K = 255$ the total time measured is $881.5s$ for a clock rate of $f = 50MHz$ and $440.82s$ for $f = 100MHz$ clock rate.

6.4 CPU runtime analysis

This section analyses the CPU runtime and how it is affected by the changes in search word length and error threshold.

When capturing the data, the preamble generation time is not subtracted from the total processing time. As discussed in section 4.4.2, the preamble can be considered insignificant in comparison to the overall required. Even though it has an exponential growth rate, the worst case in the data set is 0.044% of the overall processing time with a polynomial of degree eleven, and search word length of 64b. Which in itself has too small values to be a reliable measurement, as shown in the analysis of FPGA, section 6.3.

Figure 18 and 19 is plotted in the same manner as for FPGA analysis in figure 17. Search word length along the X-axis with multiple measurements indicating different error thresholds, and the mean time used per search state value calculation (R_{op}) along the Y-axis. This gives the comparison on how processing on CPU differs from FPGA, where the result was a constant mean time when the search word was increased beyond 256b.

A major difference between the FPGA and CPU data sets is that the mean R_{op} time has a linear growth rate along the X-axis when calculating the mean directly. As shown in figure 18. This indicates that CPU measurements are affected severely by a phenomenon which occurs on the CPU and not on the FPGA. The theory in section 2.5 suggests that this is specifically due to the address space splitting the values into separate values. Thus increasing the number of operations needed to be performed by the search.

Given that the server was running a 64b operating system, the hypothesis for how the linear growth occurred was that the mean time increased based on the search word length. Meaning that it increased based on number of splits which occurs of the M-bit long search word. The goal is to have a constant mean time, if possible, for the search. This is because a search consists of a specific number of operations performed in order to complete.

Deducing the linear growth and how it occurs was done through testing and measuring the linear growth between data points, but did not yield results for the mean time even though it was

known $M < 256b$ provided less consistent data, none of the results reached a consistent mean time. Since the operating system, and GMP was built on 64b, the linear growth was contributed to this.

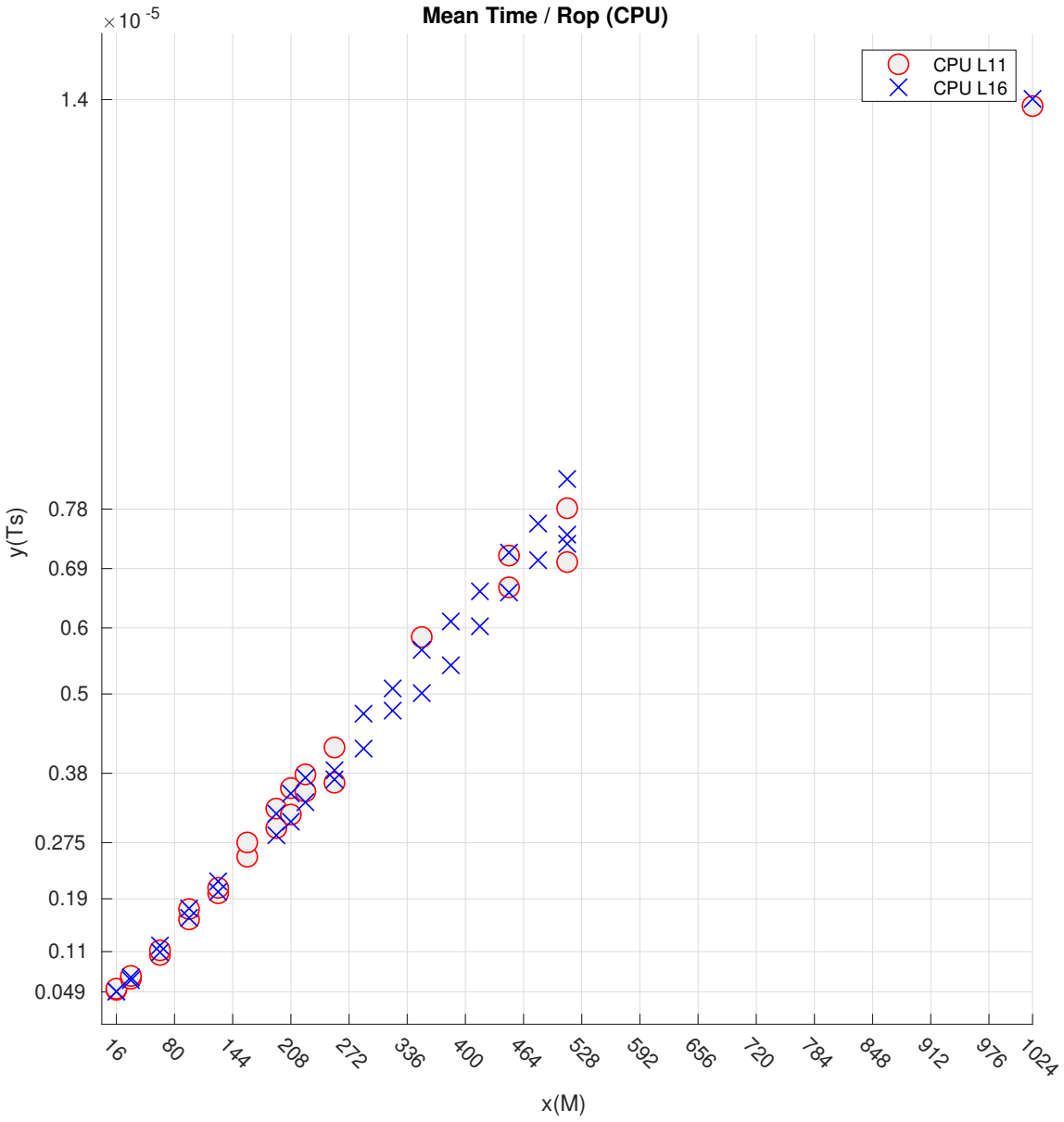

Figure 18: Mean time for CPU to perform each R_{op} calculation based on an M bit search word

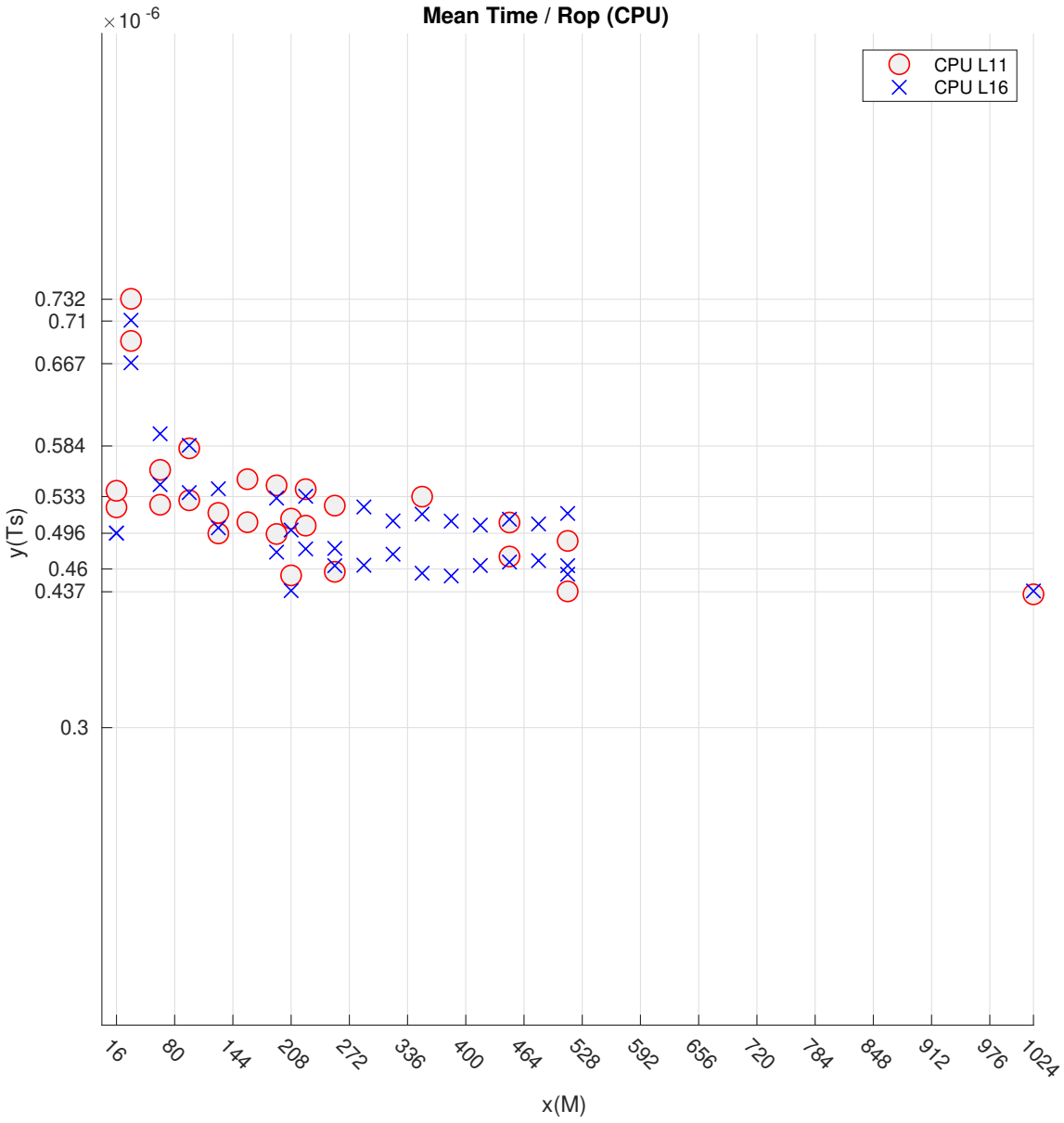

Figure 19: Adjusted mean time for CPU to perform each R_{op} calculation based on an M bit search word

Based on this several factors was tested, first $\lceil \frac{M}{64} \rceil$ was tested as a contributing factor for the mean time, but did not yield results. Further testing showed that $\lceil \frac{M}{32} \rceil$ was the most suitable factor because the GNU Multiple Precision Arithmetic Library (GMP) MPZ utilises the 32b UNSIGNED INT as their base value for representing data. The resulting estimation for processing time of the CPU is given below.

$$\frac{T_{tot}}{(2^L - 1) \times \lceil \frac{M}{32} \rceil \times 2M \times K} \tag{6.2}$$

The new mean time multiplies the number of R_{ops} by the factor $\lceil \frac{M}{32} \rceil$ which is the number that every M-bit value is split into. This yields the following graph, figure 19, following the same plotting as figure 17 and 18. Disregarding noisy data the distribution appears similar to figure 17. Where the initial short search words below 256b contains data affected by ratio between R_{ops} and other operations performed by the software. It has a negative exponential plot which stabilises at approximately $0.437E-6$ given the last data point at $M = 1024$.

However this data set have not captured data above 1024b because of the time requirement for collecting these samples. Given the data collected and presented it can be inferred that the mean time per R_{op} should be approximately $0.437E-6$ as shown by the data collected above a search word of 512b. Estimating processing times using this is discussed further in section 6.6.

For FPGA, we could stipulate that the system used four clock cycles to perform each search state value calculation. On CPU this is not directly relatable because there are many operations happening simultaneously causing interference. Given the same interpretation as for FPGA, we get 1044 CPU clock cycles per search state value (R_{op}) calculation at $M = 1024b$. However, this value has not been shown to be translatable between CPUs of different clock speeds. Estimation and how close this value is able to plot the total processing time is discussed in section 6.6.

$$\frac{T}{f} \rightarrow \frac{0.437E-6}{\frac{1}{2.39GHz}} = 0.437E-6 \times 2.39E+9 = 1044$$

The conclusion for the data collected and overall processing time is as follows. Meaning the processing time is constant, but it is directly affected by how the search word size is represented on the system. Given a 32b representation it will increase the overall processing time by the number of address spaces the search word is split across.

It is also discovered that GMPs MPZ library is not the most appropriate method of performing these calculations, given that speed is a critical factor and MPZ utilise 32b values and causes a lot of overhead which could be disregarded. Which is discussed in section 4.1. A far better library could be GMP MPN, which can process positive integers and are very low level as described by GMP[23].

6.5 Normalised clock comparison

In the previous sections the two implementations were analysed independently. This section explores how much better the FPGA is in relation to the CPU, both with and without a normalised clock rate. This comparison is illustrated in figure 20.

The following figure 20 plots the total total processing time as measured for the FPGA at $f = 50MHz$ and CPU at $f = 2.39GHz$ and last the FPGA using an estimated clock rate of $f = 2.39GHz$. The graph is plotted using logarithmic representation along both axis, with search word length along the X-axis and total processing time along the Y-axis.

As figure 20 shows and previously seen. The CPUs processing time is much slower at completing

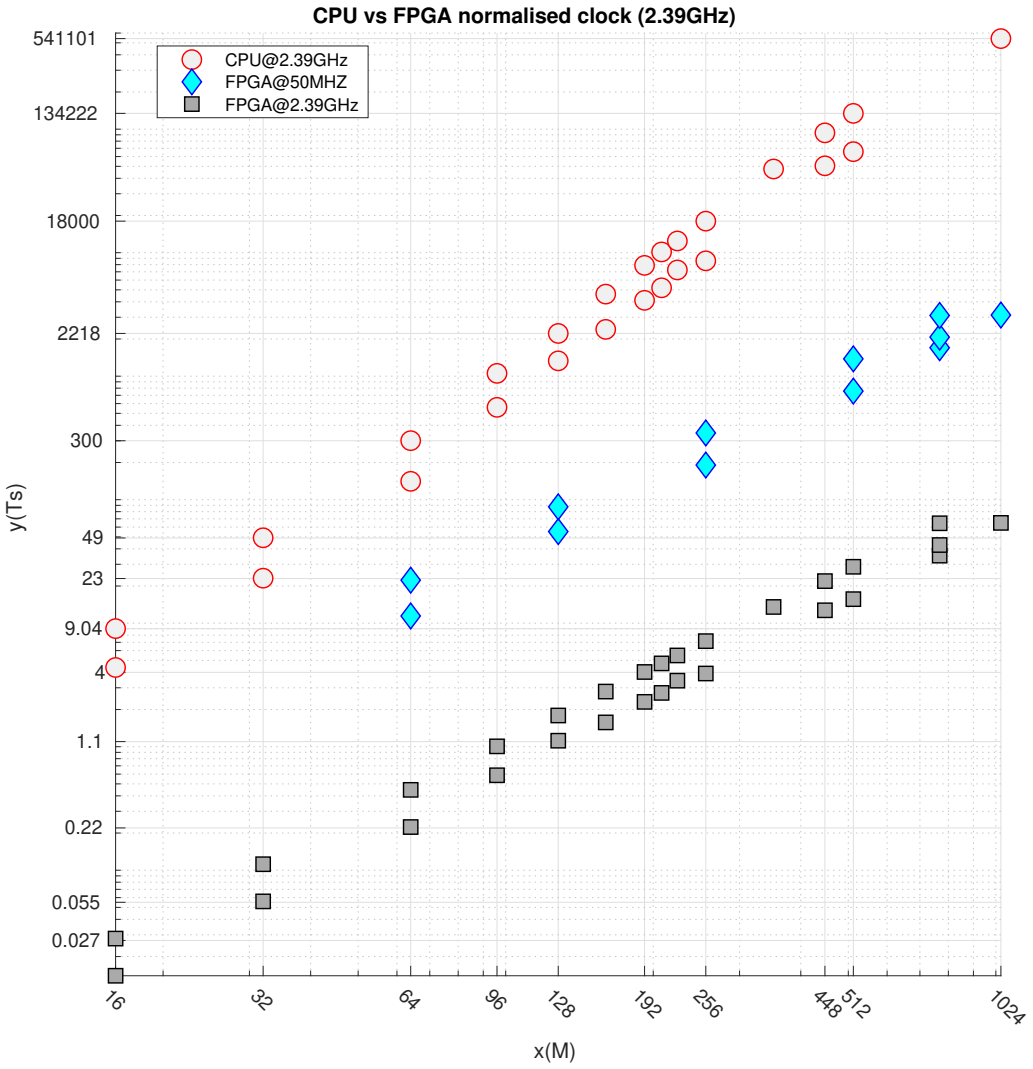

Figure 20: Normalised FPGA clock rate vs CPU (log,log)

a search, even in relation to the FPGA at $f = 50MHz$. For the lowest comparable data point with a search word of 64b and $\frac{1}{2}M$ error threshold, the FPGA is 13 times faster than the CPU at $f = 2.39GHz$. Given a normalised clock speed, this is increased to 670 times faster.

Building on the discoveries in the previous sections where CPU processing time increases linearly based on the search word length, while the FPGA processing time remains constant. Based on this the difference in processing time should increase quite severely. For FPGA at $f = 50MHz$ the difference is a 173 times faster, while at $f = 2.39GHz$ it becomes 8454 times faster.

The result is quite clear given the current data, but given a better implementation on CPU using multithreading and lower-level libraries with less overhead could reduce the CPU processing time. Given that each thread implemented reduces the processing time equally, on a standard 6 core CPU with 2 threads one could potentially reduce the processing time by 12. However, even so the resulting processing time is still far slower, as opposed to the normalised clock rate on an FPGA.

6.6 FPGA Estimated Runtime

Given the data gathered in section 6.3 and 6.4 it was derived the following equations for estimating the total runtime processing of both FPGA and CPU. The errors between estimation and measurements are plotted in figure 21, and as percentages. The FPGA total runtime is estimated as follows using the mean number of clock cycles per R_{op}.

$$\begin{aligned} T_{tot} &= R_{ops} \times 4f \\ T_{tot} &= R_{ops} \times \frac{4}{50E+6} \end{aligned} \tag{6.3}$$

The CPU total runtime is estimated as follows, using the number of CPU cycles based on the measurement at $M = 1024b$. This number was selected because the data for shorter search words can is unreliable.

$$\begin{aligned} T_{tot} &= R_{ops} \times \left\lceil \frac{M}{32} \right\rceil \times 1044f \\ T_{tot} &= R_{ops} \times \left\lceil \frac{M}{32} \right\rceil \times \frac{1044}{2.39E+9} \end{aligned} \tag{6.4}$$

Figure 21 illustrates the errors between the estimated processing times and actual measured data for the polynomial of degree 16. Given the results it shows that CPU processing times is less reliable when estimating processing time as opposed to the FPGA. This is due to the data set being shorter and only measured for search words up to 1024b. Given data points for longer search words, it could yield better estimation.

The worst estimation for FPGA is 6.1% at a 64b search word, whereas CPU has a much larger error rate of 21%. In conclusion the FPGA can be estimated reliably, with very little error rate. At M=2048b the error for FPGA is between 22.3 and 26.6s giving an error at approximately 0.2%. For CPU it can not be drawn any conclusions, because it is solely based on the value at M=1024 since this has the lowest error rate -0.5% however, this is however an error of -2967s. Given this data the conclusion in regards to CPU is that the processing time can not be estimated with good reliability. Even though the percentage is low, it still results in a large time difference.

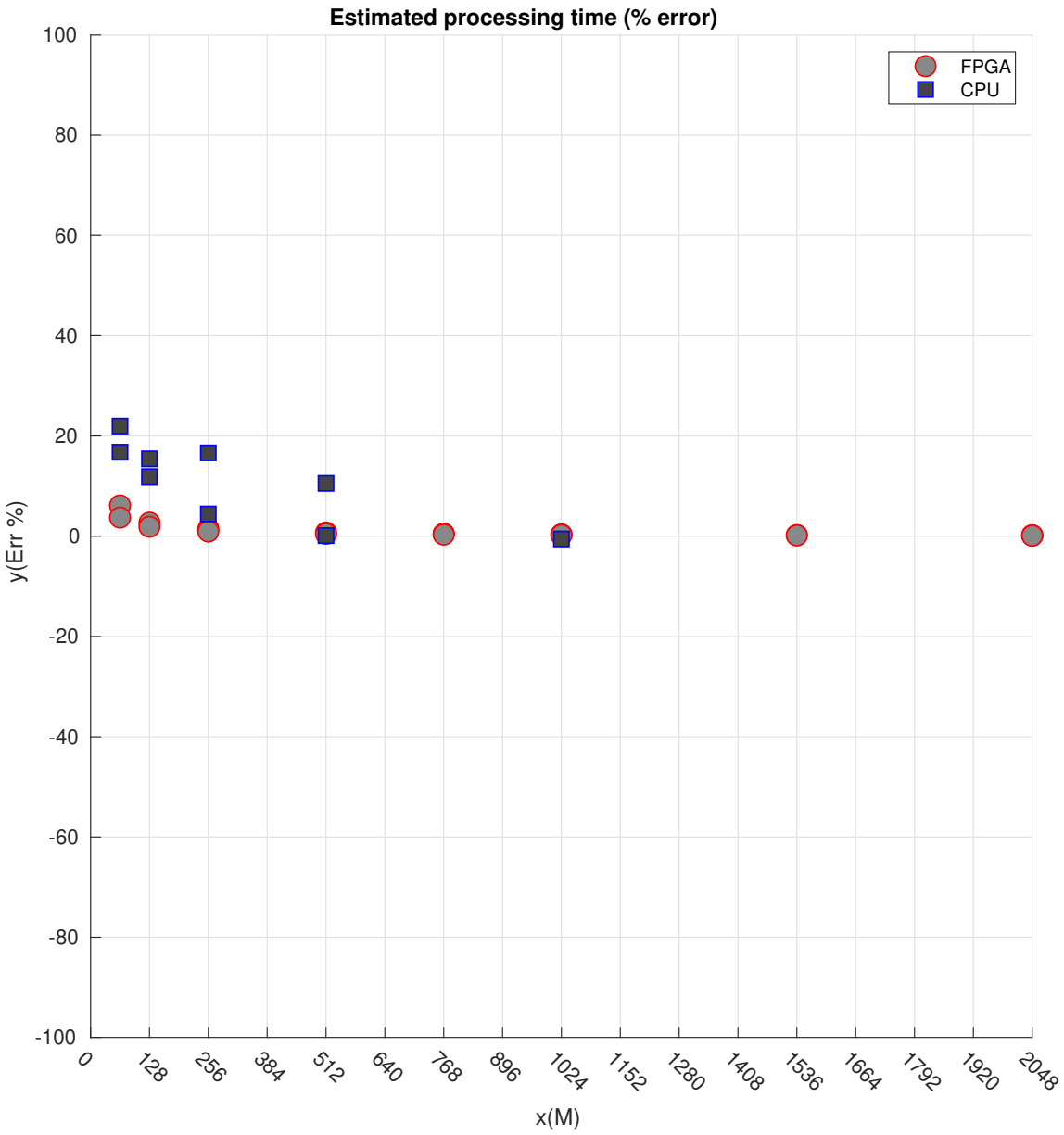

Figure 21: Estimated runtime error given in percentage.

6.7 Issues and Obstacles

During this thesis there were some obstacles to overcome. This section discusses the main issues and obstacles..

6.7.1 GMP

The final implementation was based on the GMP library, which provides several library implementations. The MPZ-library with supporting functions was selected because it is a commonly used library and was appropriate regarding the platform and language already used. It is not the best library, but the time requirement made us opt for the MPZ library as it would be fast to implement and the functional requirements was in place. The MPN library would be a better option, but its handling of type and conversion between types required more time for implementation than possible.

During the implementation of MPZ some of the functions source code was explored, and the implementations focus on error handling and verifying. Resulting in a large overhead, as noted in section 4.1. These additional steps, along with variables being split across multiple 64b registers, causes each operation to be less effective. Which is where the FPGA excels. Due to this future work should focus on testing the MPN library.

6.7.2 FPGA External communication

Microcontroller based programming languages can easily utilise complex system functionality, eg. Ethernet and WiFi. Even though there are libraries and finished IP designs available there are issues in regards to implementing them into a FPGA design, ranging from board differences, clock speed, port mappings, design requirements and complexity.

The main reason for utilising UART for communication was because the design was openly licensed, low impact on the FPGA and simple to implement, with very little configuration required.

6.7.3 FPGA utilisation

The designed FPGA implementation utilised Configurable Logic Block (CLB) in the beginning, which is of a finite amount on an FPGA. Initially before implementing BRAM to store the Non-Deterministic Finite Automaton (NFA) $(K+1) \times M$ number of search state values was stored in CLBs as registers, resulting in it maxing out on utilisation at $M = 1024$. This was because the number of CLBs was wasted on simply storing the values of search state registers.

Converting the design to utilise BRAM made it possible to perform search up to $M = 4096$ with $K = 1023$. After this the FPGA maxed out the BRAMs storage capacity, as the Nexys A7 has a capacity of 4,860kb BRAM it can maximum store the search state values for $K \leq 1186$ given $M = 4096$.

Given this, further development requires the implementation of DRAM, which is built into the FPGA and has a much larger capacity.

Summary

Resource	Utilization	Available	Utilization %
LUT	15920	63400	25.11
LUTRAM	2	19000	0.01
FF	10640	126800	8.39
BRAM	57	135	42.22
IO	19	210	9.05

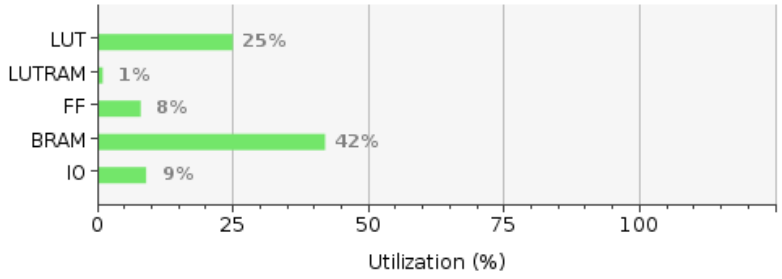

Figure 22: Device utilisation summary, L(16), M(2048), K(1024). Source: Magnus Øverbø

7 Conclusion

In this thesis introduction, we posed the following hypothesis.

> *Field-Programmable Gate Array (FPGA) will perform an unconstrained Approximate Row-wise Bit-Parallel (ARBP) search faster than a Central Processing Unit (CPU), given a normalised clock rate and serialised search.*

The experiment showed that the FPGA has a significantly better performance than the Central Processing Unit (CPU) when performing an unconstrained Approximate Row-wise Bit-Parallel (ARBP) search using the shift-AND algorithm[3], confirming our main hypothesis. Given the two implementations, the Field-Programmable Gate Array (FPGA) at $f = 50MHz$ is 13-173 times faster than the CPU. However, since the Nexys A7 FPGA could natively run at $f = 100MHz$, the FPGA is actually be 26-346 times faster. Utilising external clock or Phase-Locked Loop (PLL) the clock rate can be increased even further increasing the difference. Overall, this constitutes to FPGA being much faster than the CPU.

As it becomes time consuming to search the entire search space with one FPGA, but given an actor with enough resources it is possible to search very large search spaces. Although plausible, the requirements becomes increasingly high. E.g. given 100000 hypothesised hardware circuits, with $f = 5GHz$, mean $R_{op} = f$, each running 100 parallel searches would still only be effective up to $L \leq\approx 50$, see table 3. Even though this is only an example, it shows the requirements for such systems quickly becomes large and expensive, but is still plausible given enough resources. Although the cost is increasing significantly in relation to the search space.

Table 3: Time estimation for hypothesised circuit

L	M	K	R_{ops}	Estimated time
40	2048	1024	4.61E+018	92.32 sec
50	2048	1024	4.72E+021	26.26 hours
60	2048	1024	4.84E+024	3.07 years
70	2048	1024	4.95E+027	3143.45 years
90	2048	1024	5.19E+033	3296.14E+06 years
120	2048	1024	5.58E+042	3.54E+18 years

7.1 Answers to research questions

How long search words can practically be processed

Both CPU and FPGA can process $M \leq \infty$, given the ability to store and maintain the $M \times K$ table of search state values. In regards to practicality, the main factors are the feedback polynomial of

degree L, clock rate f, and number of parallel operators. Given 365 FPGAs, $f = 2.39GHz$ and $L = 32$ can easily be searched within one day.

For the CPU implementation this becomes impractical at $L = 16$, $M = 1024$ and $K = 289$ requiring 6.26 days, but with parallel processing this would be reduced significantly. CPUs are mainly affected by the factor $\lceil \frac{M}{w} \rceil$, and therefore needs special CPUs with w as large as possible.

How fast is the search performing

The FPGA has a consistent execution time of approximately $4f$ per search state value update (R_{op}). The total processing time is thereby calculated using

$$T_{tot} = R_{ops} \times 4f$$

The CPU do not have a consistent mean R_{op} time and results in unreliable estimates. Using the most stable value at $M = 1024$ as the mean R_{op} time, we get $R_{op} = 1044f$ yielding the estimation of total processing time.

$$T_{tot} = R_{ops} \times \left\lceil \frac{M}{w} \right\rceil \times 1044f$$

How do changes to polynomial, search word length and error threshold affect the performance

The polynomial sets the basis for the total number of R_{ops}. For FPGA, both K and M increase the time linearly given individual change, but since both increases when the search word increases the result is an exponential increase in total processing time.

The same goes for the CPU, but the search word length also causes the entire search to increase by a factor of $\lceil \frac{M}{w} \rceil$, resulting in a faster growth.

How do the FPGA and CPU compare, given a normalised CPU clock speed

Comparatively, FPGA is much faster given the implementations, even without normalised clock speed, but CPUs with many cores and $w = 512b$[9] could initially make CPU faster than an FPGA. Even so, investing the same effort into the FPGA would cause it outperform the CPU given the growth factor of $\lceil \frac{M}{w} \rceil$.

What are the limitations given each architecture

FPGAs has a limitation on the number of Configurable Logic Block (CLB)s and clock rate, and should be run at the fastest speed possible. The CLB limitation require better design to reduce the utilisation and allow for longer searches. The FPGA should also run multiple instances of the search in order to run multiple searches simultaneously and reduce the processing time.

CPUs main issue is regarding register size, which can be increased using better CPUs[9], but would only delay the effects of the issue. Another issue is that the processing should be multi-threaded throughout to increase the speed[9].

7.2 Future Work

Given the conclusion, there are several questions which pose a topic for further research. The main questions are listed below and encompass improving CPU implementation and whether other algorithms could yield faster processing times given polynomials of higher degree.

7.2.1 Improvement of CPU implementation

Several methods could be employed to improve upon the CPU implementation used in this thesis to provide the absolute best possible performance but involves several tasks. Multithreading is key when it comes to improving speed on a CPUs and should be implemented, but not by itself as it would not improve the search time given the $\lceil \frac{M}{w} \rceil$ factor increasing the search time every w bits. To combat this, either using better libraries or implementing the search on physical computer nodes utilising enterprise-grade CPUs which allowing for large w to be processed, as explored by Tran et.al[9]. It would not remove the factor but greatly reduce it.

7.2.2 Processing time for large polynomials

An interesting aspect is how to reduce the processing time for large polynomials. Currently, polynomials of degree $L = 20$ are easily searched, but searching a polynomials of degree $L \geq 32$. becomes increasingly difficult and costly with the current method. For CPU, search-OR will improve the search time, but an FPGA would not benefit in time, as the shift-AND executes within a single clock pulse. Methods to increase processing time could be by increasing the clock rate of the design, clustering FPGAs, or improving the design.

Bibliography

[1] Petrovic, S. 2018. Constrained approximate bit-parallel search with application in cryptanalysis. In *RECSI XV: Seión 6. Seguridad y Análisis de Datos*, 174–179. RECSI. URL: https://nesg.ugr.es/recsi2018/docs/ActasXVRECSI.pdf.

[2] Golic, J. D. & Mihaljevic, M. J. 1991. A generalized correlation attack on a class of stream ciphers based on the levenshtein distance. *J. Cryptology*, 3, 201–212. doi:10.1007/BF00196912.

[3] Baeza-Yates, R. & Gonnet, G. H. October 1992. A new approach to text searching. *Commun. ACM*, 35(10), 74–82. URL: http://doi.acm.org/10.1145/135239.135243, doi:10.1145/135239.135243.

[4] Siegenthaler. Jan 1985. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers*, C-34(1), 81–85. doi:10.1109/TC.1985.1676518.

[5] Wu, S. & Manber, U. October 1992. Fast text searching: Allowing errors. *Commun. ACM*, 35(10), 83–91. URL: http://doi.acm.org/10.1145/135239.135244, doi:10.1145/135239.135244.

[6] Borgund, H. M. Spam filtering with approximate search in fpga hardware. Master's thesis, NTNU, 2018. http://hdl.handle.net/11250/2502563.

[7] Irwin, S. G., Venkat, A. A., Winberg, S. L., & Mishra, A. K. Dec 2011. Fpga-based string matching. In *2011 International Conference on Energy, Automation and Signal*, 1–4. doi:10.1109/ICEAS.2011.6147137.

[8] Michailidis, P. D. & Margaritis, K. G. April 2006. Implementation of a programmable array processor architecture for approximate string matching algorithms on fpgas. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 4. doi:10.1109/IPDPS.2006.1639474.

[9] Tran, T. T., Schindel, S., Liu, Y., & Schmidt, B. Oct 2014. Bit-parallel approximate pattern matching on the xeon phi coprocessor. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, 81–88. doi:10.1109/SBAC-PAD.2014.37.

[10] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. 1997. *Handbook of Applied Cryptography*. CRC Press.

[11] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. *6.2 - Feedback Shift Registers*, chapter 6.2, 195–203. In *Handbook of Applied Cryptography*. CRC Press, 1997.

[12] Church, R. 1935. Tables of irreducible polynomials for the first four prime moduli. *Annals of Mathematics*, 36(1), 198–209. URL: http://www.jstor.org/stable/1968675.

[13] Partow, A. Primitive polynomial list. URL: http://partow.net/programming/polynomials/index.html (Visited 20190428).

[14] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. *6.3.3 - Clock-Controlled Generators*, chapter 6.3.3, 209–212. In *Handbook of Applied Cryptography*. C21RC Press, 1997.

[15] Chambers, W. G. & Jennings, S. M. November 1984. Linear equivalence of certain brm shift-register sequences. *Electronics Letters*, 20(24), 1018–1019. doi:10.1049/el:19840693.

[16] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. *Handbook of Applied Cryptography*, chapter 5.4.3-5.4.4, 195–203. In *Handbook of Applied Cryptography*. CRC Press, 1997.

[17] Knuth, D., Morris, Jr., J., & Pratt, V. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2), 323–350. URL: https://doi.org/10.1137/0206024, arXiv:https://doi.org/10.1137/0206024, doi:10.1137/0206024.

[18] Boyer, R. S. & Moore, J. S. October 1977. A fast string searching algorithm. *Commun. ACM*, 20(10), 762–772. URL: http://doi.acm.org/10.1145/359842.359859, doi:10.1145/359842.359859.

[19] Navarro, G. & Raffinot, M. 2002. *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 1 edition.

[20] Navarro, G. & Raffinot, M. *Pseudo code for Approximate RBP Search*, chapter 6, 153. In *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 1 edition, 2002.

[21] Tanenbaum, A. S. 2009. *Modern Operating Systems*. Pearson, 3 edition.

[22] Stallings, W. *Reduced Instruction Set Computers*, chapter 13, 498–539. In *Computer Organization and Architecture: Desiging for performance*. Pearson, 8 edition, 2010.

[23] GMP. 2016. The gnu multiple precision arithmetic library. https://gmplib.org/. Verified: 20190329.

[24] Xilinx. 3 2018. Reducing system power and cost with artix-7 fpgas (wp423). URL: https://www.xilinx.com/support/documentation/white_papers/wp423-Reducing-Sys-Power-Cost-28nm.pdf (Visited 20190512).

[25] nandland. Tutorial - how flip-flops work in fpgas. URL: https://www.nandland.com/articles/flip-flop-register-component-in-fpga.html (Visited 20190512).

[26] fpga4fun.com & Nicolle, J. P. fpga4fun.com - counters 4 - the carry chain. URL: https://www.fpga4fun.com/Counters4.html (Visited 20190512).

[27] Tutorials, I. B. October 2018. Learn fpga 20: Save resources!!! (distributed ram vs. block ram) - tutorial. URL: https://www.youtube.com/watch?v=T4khi8MmoVc (Visited 20190331).

[28] Green, W. 2018. Block ram in verilog with vivado. URL: https://timetoexplore.net/blog/block-ram-in-verilog-with-vivado (Visited 20190422).

[29] Xilinx. 2008. Fpga design flow overview. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm (Visited 20190512).

[30] JimBlom, S. 2012. Serial communication. URL: https://learn.sparkfun.com/tutorials/serial-communication/ (Visited 20190513).

[31] Goddard, T. 2010. Osdvu (open source documented verilog uart). https://github.com/cyrozap/osdvu/. Commit: d18488c41141cfb1c7b29f5a5840510e727ae5a2.

[32] Wilson, J. 2014. *Essentials of business research: a guide to doing your research project*. Sage Publishing, 2nd edition.

[33] Leedy, P. D. & Ormrod, J. E. 2014. *Practical Research Planning and Desgn*. Pearson, 10th edition.

[34] Cresswell, J. W. 2009. *Research Design: Qualitative, Quantitative and Mixed Methods Approaches*. Sage Publishing, 3rd edition.

[35] Larman, C. *Iterative and Evolutionary*, chapter 2, 9–24. In *Agile and Iterative Development: a manager's guide*, Cockburn, A. & Highsmith, J., eds. Agile Software Development Series. Pearson Education Inc, 1st edition, 2004.

[36] NANDLand. Nandland - fpga-101. URL: https://www.nandland.com/articles/fpga-101-fpgas-for-beginners.html (Visited 20190331).

[37] EM. 2018. Learning fpga and verilog - beginners guide. URL: https://hackaday.io/em31405 (Visited 20190331).

[38] Williams, A. 2018. Fpga boot camp series. URL: https://hackaday.io/projects/hacker/78538 (Visited 20190331).

[39] Digilent. November 2018. Vivado-boards. https://github.com/Digilent/vivado-boards. Commit: 1066a740f39d0513f00492ebf9e22f83c5344aaf.

[40] Digilent. November 2018. digilent-xdc. https://github.com/Digilent/digilent-xdc. Commit: 01e6bf6225fe890cc1419adc4ca70f646b469adf.

[41] Williams, A. 2018. How to add uart to your fpga projects. URL: https://hackaday.com/2018/09/06/fpga-pov-toy-gets-customized/ (Visited 20190331).

[42] DigilentInc. Nexys a7-100t. URL: https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/ (Visited 20190425).

[43] Liechti, C. Pyserial - short introduction. Revision a27715f3. URL: https://pyserial.readthedocs.io/en/latest/shortintro.html (Visited 20190425).

[44] MacKenzie, D. & Eddelbuettel, D. time - linux man page. URL: https://linux.die.net/man/1/time (Visited 20190425).

A Data sets

A.1 FPGA Measurements

Table 4: Data set collected from FPGA with 50MHz clock rate.

Variables				FPGA			
L	M	K	R-ops	T (50MHz)	Sec	Match	Mean T/R_{op}
11	64	15	4192256	00:00:00.352	0.35	1	83.96E-09
11	64	31	8384512	00:00:00.800	0.80	96	95.36E-09
11	128	33	17817088	00:00:01.472	1.47	8	82.64E-09
11	128	63	33538048	00:00:02.944	2.94	192	87.78E-09
11	256	79	83845120	00:00:07.151	7.15	308	85.29E-09
11	512	139	293457920	00:00:23.679	23.68	33	80.69E-09
11	512	159	335380480	00:00:27.726	27.73	628	82.67E-09
11	1024	289	1215754240	00:01:37.820	97.82	171	80.46E-09
11	1024	349	1467289600	00:01:59.276	119.28	1306	81.29E-09
11	1024	511	2146435072	00:02:53.883	173.88	1535	81.01E-09
16	64	15	134215680	00:00:11.440	11.44	1	85.23E-09
16	64	31	268431360	00:00:22.303	22.30	96	83.09E-09
16	128	39	671078400	00:00:55.183	55.18	92	82.23E-09
16	128	63	1073725440	00:01:27.550	87.55	192	81.54E-09
16	256	69	2348774400	00:03:10.699	190.70	12	81.19E-09
16	256	127	4294901760	00:05:46.887	346.89	384	80.77E-09
16	512	139	9395097600	00:12:37.196	757.20	17	80.59E-09
16	512	255	17179607040	00:23:00.986	1380.99	768	80.39E-09
16	768	209	21138969600	00:28:19.496	1699.50	25	80.40E-09
16	768	255	25769410560	00:34:31.098	2071.10	971	80.37E-09
16	768	383	38654115840	00:51:42.251	3102.25	1152	80.26E-09
16	1024	289	38922547200	00:52:05.070	3125.07	99	80.29E-09
16	1024	383	51538821120	01:08:55.982	4135.98	1369	80.25E-09
16	1024	511	68718428160	01:31:50.719	5510.72	1536	80.19E-09
16	1536	511	103077642240	02:17:45.337	8265.34	1941	80.19E-09
16	1536	767	154616463360	03:26:29.259	12389.26	2304	80.13E-09
16	2048	569	153005875200	03:24:22.798	12262.80	32	80.15E-09
16	2048	767	206155284480	04:35:18.288	16518.29	2743	80.13E-09
16	2048	1023	274873712640	06:06:56.513	22016.51	3072	80.10E-09
20	64	15	2147481600	00:03:02.939	182.94	11	85.19E-09
20	64	31	4294963200	00:05:55.191	355.19	96	82.70E-09
20	128	39	10737408000	00:14:41.497	881.50	108	82.10E-09
20	128	63	17179852800	00:23:17.498	1397.50	192	81.35E-09
20	256	69	37580928000	00:50:51.070	3051.07	71	81.19E-09
20	256	127	68719411200	01:32:24.214	5544.21	384	80.68E-09
20	512	255	274877644800	06:08:02.559	22082.56	768	80.34E-09

Table 5: Data set collected from FPGA with 100MHz clock rate.

Variables				FPGA			
L	M	K	R-ops	T (100MHz)	Sec	Match	Mean T/R_{op}
20	128	39	10737408000	00:07:20.821	440.82	108	41.05E-09
20	512	255	274877644800	03:04:01.732	11041.73	768	40.17E-09

A.2 CPU Measurements

Table 6: Data set collected from 2.39GHz CPU

Variables				CPU C w/GMP				
L	M	K	R-ops	T (2.39GHz)	Sec	Match	Mean T/R_{op}	Mean T/R_{op} $\lceil \frac{M}{32} \rceil$
11	16	3	262016	00:00:00.130	0.13	3	4.96E-07	4.96E-07
11	16	7	524032	00:00:00.260	0.26	24	4.96E-07	4.96E-07
11	32	7	1048064	00:00:00.700	0.70	13	6.68E-07	6.68E-07
11	32	15	2096128	00:00:01.490	1.49	48	7.11E-07	7.11E-07
11	64	15	4192256	00:00:04.570	4.57	7	1.09E-06	5.45E-07
11	64	31	8384512	00:00:10.000	10	96	1.19E-06	5.96E-07
11	96	27	11004672	00:00:17.730	17.73	53	1.61E-06	5.37E-07
11	96	47	18865152	00:00:33.090	33.09	144	1.75E-06	5.85E-07
11	128	33	17817088	00:00:35.740	35.74	8	2.01E-06	5.01E-07
11	128	63	33538048	00:01:12.560	72.56	192	2.16E-06	5.41E-07
11	192	54	43232640	00:02:03.720	123.72	39	2.86E-06	4.77E-07
11	192	95	75460608	00:04:00.620	240.62	288	3.19E-06	5.31E-07
11	208	54	46835360	00:02:23.640	143.64	4	3.07E-06	4.38E-07
11	208	103	88561408	00:05:09.500	309.50	311	3.49E-06	4.99E-07
11	224	59	55023360	00:03:04.970	184.97	0	3.36E-06	4.80E-07
11	224	111	102710272	00:06:23.430	383.43	335	3.73E-06	5.33E-07
11	256	63	67076096	00:04:08.610	248.61	0	3.71E-06	4.63E-07
11	256	79	83845120	00:05:22.470	322.47	308	3.85E-06	4.81E-07
11	288	74	88430400	00:06:09.110	369.11	432	4.17E-06	4.64E-07
11	288	143	169786368	00:13:18.430	798.43	3	4.70E-06	5.23E-07
11	320	94	124457600	00:09:51.010	591.01	286	4.75E-06	4.75E-07
11	320	159	209612800	00:17:45.750	1065.75	480	5.08E-06	5.08E-07
11	352	89	129697920	00:10:50.180	650.18	1	5.01E-06	4.56E-07
11	352	175	253631488	00:23:57.680	1437.68	528	5.67E-06	5.15E-07
11	384	104	165070080	00:14:57.170	897.17	38	5.44E-06	4.53E-07
11	384	191	301842432	00:30:40.910	1840.91	576	6.10E-06	5.08E-07
11	416	109	187341440	00:18:48.840	1128.84	9	6.03E-06	4.64E-07
11	416	207	354245632	00:38:41.920	2321.92	624	6.55E-06	5.04E-07
11	448	119	220093440	00:23:58.860	1438.86	18	6.54E-06	4.67E-07
11	448	223	410841088	00:48:54.250	2934.25	671	7.14E-06	5.10E-07
11	480	129	255465600	00:29:54.800	1794.80	25	7.03E-06	4.68E-07
11	480	239	471628800	00:59:35.240	3575.24	719	7.58E-06	5.05E-07
11	512	139	293457920	00:35:34.810	2134.81	33	7.27E-06	4.55E-07
11	512	159	335380480	00:41:26.330	2486.33	628	7.41E-06	4.63E-07
11	512	255	536608768	01:13:50.000	4430	768	8.26E-06	5.16E-07
11	1024	255	1073217536	04:10:36.000	15036	0	14.01E-06	4.38E-07
16	16	3	8388480	00:00:04.380	4.38	12	5.22E-07	5.22E-07

L	M	K	R-ops	T (2.39GHz)	Sec	Match	Mean T/R_{op}	Mean T/R_{op} $\lceil \frac{M}{32} \rceil$
16	16	7	16776960	00:00:09.040	9.04	24	5.39E-07	5.39E-07
16	32	7	33553920	00:00:23.150	23.15	12	6.90E-07	6.90E-07
16	32	15	67107840	00:00:49.150	49.15	48	7.32E-07	7.32E-07
16	64	15	134215680	00:02:20.860	140.86	1	1.05E-06	5.25E-07
16	64	31	268431360	00:05:00.590	300.59	96	1.12E-06	5.60E-07
16	96	27	352316160	00:09:19.410	559.41	24	1.59E-06	5.29E-07
16	96	47	603970560	00:17:33.910	1053.91	144	1.74E-06	5.82E-07
16	128	39	671078400	00:22:10.880	1330.88	92	1.98E-06	4.96E-07
16	128	63	1073725440	00:36:58.380	2218.38	192	2.07E-06	5.17E-07
16	160	44	943704000	00:39:52.910	2392.91	27	2.54E-06	5.07E-07
16	160	79	1677696000	01:16:58.000	4618	241	2.75E-06	5.51E-07
16	192	54	1384099200	01:08:33.000	4113	69	2.97E-06	4.95E-07
16	192	95	2415882240	02:11:30.000	7890	287	3.27E-06	5.44E-07
16	208	59	1635753600	01:26:33.000	5193	90	3.17E-06	4.54E-07
16	208	103	2835306240	02:48:57.000	10137	312	3.58E-06	5.11E-07
16	224	69	2055177600	02:00:46.000	7246	261	3.53E-06	5.04E-07
16	224	111	3288284160	03:27:20.000	12440	336	3.78E-06	5.40E-07
16	256	69	2348774400	02:23:09.000	8589	12	3.66E-06	4.57E-07
16	256	127	4294901760	05:00:00.000	18000	384	4.19E-06	5.24E-07
16	352	175	8120048640	13:13:22.000	47602	528	5.86E-06	5.33E-07
16	448	129	7633516800	14:01:39.000	50499	246	6.62E-06	4.73E-07
16	448	223	13153136640	25:55:36.000	93336	672	7.10E-06	5.07E-07
16	512	139	9395097600	18:15:47.000	65747	17	7.00E-06	4.37E-07
16	512	255	17179607040	37:17:02.000	134222	768	7.81E-06	4.88E-07
16	1024	289	38922547200	150:18:21.000	541101	99	13.90E-06	4.34E-07

Table 7: Timing of preamble generation on 2.39GHz CPU

Variables		Time	
L	M	Time (2.39GHz)	Sec
11	64	00:00:00	0.002
11	256	00:00:00	0.004
11	512	00:00:00.01	0.007
11	768	00:00:00.01	0.011
11	1024	00:00:00.01	0.010
11	2048	00:00:00.03	0.025
11	4096	00:00:00.10	0.098
11	8192	00:00:00.35	0.352
11	16384	00:00:01.38	1.376
11	24576	00:00:03.12	3.120
11	32768	00:00:05.46	5.460
16	64	00:00:00	0.003
16	256	00:00:00	0.004
16	512	00:00:00.01	0.007
16	768	00:00:00.01	0.012
16	1024	00:00:00.02	0.016
16	2048	00:00:00.03	0.033
16	4096	00:00:00.10	0.097
16	8192	00:00:00.36	0.356

L	M	Time (2.39GHz)	Sec
16	16384	00:00:01.37	1.370
16	24576	00:00:03.09	3.090
16	32768	00:00:05.47	5.470

A.3 Estimation

Table 8: Estimation of runtime processing for FPGA

Variables				Time		
L	M	K	R_{ops}	S (50MHz)	S (100MHz)	S (2.39GHz)
32	2048	512	9.02E+15	721983315.09	360991657.55	15104253.45
32	2048	1024	18.03E+15	1442559255.31	721279627.65	30179063.92
32	2048	1024	18.03E+15	1442559255.31	721279627.65	30179063.92
32	4096	1024	36.06E+15	2885118510.61	1442559255.31	60358127.84
32	4096	2048	72.09E+15	5767422271.46	2883711135.73	120657369.70
32	4096	1024	36.06E+15	2885118510.61	1442559255.31	60358127.84
32	6144	1536	81.12E+15	6489405586.55	3244702793.28	135761623.15
32	6144	3072	162.18E+15	12974589048.46	6487294524.23	271434917.33
32	6144	1024	54.10E+15	4327677765.92	2163838882.96	90537191.76
32	8192	2048	144.19E+15	11534844542.92	5767422271.46	241314739.39
32	8192	4096	288.30E+15	23064059586.30	11532029793.15	482511706.83
32	8192	1024	72.13E+15	5770237021.22	2885118510.61	120716255.67
32	10240	2560	225.27E+15	18021435379.70	9010717689.85	377017476.56
32	10240	5120	450.45E+15	36035833884.99	18017916942.50	753887738.18
32	10240	1024	90.16E+15	7212796276.53	3606398138.27	150895319.59
32	12288	3072	324.36E+15	25949178096.91	12974589048.46	542869834.66
32	12288	6144	648.62E+15	51889911944.53	25944955972.26	1085563011.39
32	12288	1024	108.19E+15	8655355531.84	4327677765.92	181074383.51
32	14336	3584	441.48E+15	35318072694.55	17659036347.27	738871813.69
32	14336	7168	882.83E+15	70626293764.91	35313146882.46	1477537526.46
32	14336	1024	126.22E+15	10097914787.14	5048957393.57	211253447.43
32	16384	4096	576.60E+15	46128119172.60	23064059586.30	965023413.65
32	16384	8192	1.15E+18	92244979346.14	46122489673.07	1929811283.39
32	16384	1024	144.26E+15	11540474042.45	5770237021.22	241432511.35
32	18432	4608	729.74E+15	58379317531.08	29189658765.54	1221324634.54
32	18432	9216	1.46E+18	116745968688.21	58372984344.11	2442384282.18
32	18432	1024	162.29E+15	12983033297.76	6491516648.88	271611575.27
32	20480	5120	900.90E+15	72071667769.98	36035833884.99	1507775476.36
32	20480	10240	1.80E+18	144129261791.13	72064630895.57	3015256522.83
32	20480	1024	180.32E+15	14425592553.06	7212796276.53	301790639.19
32	22528	5632	1.09E+18	87205169889.31	43602584944.65	1824375939.11
32	22528	11264	2.18E+18	174394858654.90	87197429327.45	3648428005.33
32	22528	1024	198.35E+15	15868151808.37	7934075904.18	331969703.10

Table 9: Estimation error FPGA

Variables				Time		
L	M	K	R_{ops}	Est (50MHz)	Measured (50MHz)	Diff S
16	64	15	134.22E+06	10.74	11.44	0.70
16	64	31	268.43E+06	21.47	22.30	0.83
16	128	39	671.08E+06	53.69	55.18	1.50
16	128	63	1.07E+09	85.90	87.55	1.65
16	256	69	2.35E+09	187.90	190.70	2.80
16	256	127	4.29E+09	343.59	346.89	3.29
16	512	139	9.40E+09	751.61	757.20	5.59
16	512	255	17.18E+09	1374.37	1380.99	6.62
16	768	209	21.14E+09	1691.12	1699.50	8.38
16	768	255	25.77E+09	2061.55	2071.10	9.55
16	768	383	38.65E+09	3092.33	3102.25	9.92
16	1024	289	38.92E+09	3113.80	3125.07	11.27
16	1024	383	51.54E+09	4123.11	4135.98	12.88
16	1024	511	68.72E+09	5497.47	5510.72	13.24
16	1536	511	103.08E+09	8246.21	8265.34	19.13
16	1536	767	154.62E+09	12369.32	12389.26	19.94
16	2048	569	153.01E+09	12240.47	12262.80	22.33
16	2048	767	206.16E+09	16492.42	16518.29	25.86
16	2048	1023	274.87E+09	21989.90	22016.51	26.62

Table 10: Estimation error CPU

Variables				Time		
L	M	K	R_{ops}	Est (2.39GHz)	Measured (2.39GHz)	Diff S
16	64	15	134.22E+06	117.26	140.86	23.60
16	64	31	268.43E+06	234.51	300.59	66.08
16	128	39	671.08E+06	1172.56	1330.88	158.32
16	128	63	1.07E+09	1876.10	2218.38	342.28
16	256	69	2.35E+09	8207.93	8589	381.07
16	256	127	4.29E+09	15008.79	18000	2991.21
16	512	139	9.40E+09	65663.48	65747	83.52
16	512	255	17.18E+09	120070.36	134222.00	14151.64
16	1024	289	38.92E+09	544068.81	541101	-2967.81

B Source Code

B.1 CPU Implementation in C

```
/*****************************************************************************
 ** TITLE:     CipherSearch
 ** AUTHOR:    Magnus Overbo
 ** ABOUT:     Ciphersearch utilise the GMP library to manage arbitrary sized
 **            numbers and perform an unconstrained approximate row-based bit
 **            parallell search.  It searches for an intercepted bitsequence,
 **            generated by a dessimated LFSR encrypted with a message, and
 **            tries to find this in all possible generated states which is
 **            not dessimated.
 **
 ** Release:  20190313 - 64b version
 ** Release:  20190328 - GMP library version for arbitrary bit size
 *****************************************************************************/

//---------------------------------------------------------------------------
// INCLUDES
//---------------------------------------------------------------------------
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>             //strlen
#include <stdint.h>             //64b Int
#include <inttypes.h>           //64b int
#include <gmp.h>                //arbitrary integer size

//---------------------------------------------------------------------------
// GLOBAL VARIABLES
//---------------------------------------------------------------------------
char*       FNAME;              //Output filename
const int   ALPHASIZE = 2;      //Alphabet size, 0 & 1
int         m         = 896;    //Size of search word
int         n         = 1738;   //Size of text
int         slen      = 448;    //K value
int         deg       = 11;     //Polynomial degree
int         SSTATE    = 1024;
int         CSTATE    = 1;
mpz_t       PLAINTEXT;          //Message to encipher
mpz_t       TEXT;               //Search text

//---------------------------------------------------------------------------
// LFSR STRUCT
//---------------------------------------------------------------------------
struct LFSR {
  mpz_t POLYNOMIAL;             //Polynomial definition
  mpz_t   STATE;                //LFSR state
  int     DEGREE;               //Polynomial degree
};

```

```
49
50  //------------------------------------------------------------------------------
51  // FUNCTION DECLARATIONs
52  //------------------------------------------------------------------------------
53  mpz_t*          genAlphabet( int );                 //Gen array of the alphabet
54  int             lfsr_iterate( struct LFSR*);        //Gen next state & output
55  void            lfsrgen(mpz_t, int, int, mpz_t,     //Gen an LFSR
56                          uint_least64_t, int, mpz_t*);
57  mpz_t*          arbp_search( mpz_t*, int );         //Main search function
58  mpz_t*          genError(int);                      //Gen init error table
59  void            genPrefixes( mpz_t*, mpz_t );       //Generate the prefixes
60  void            genEncrypt( mpz_t, mpz_t, mpz_t );  //Encrypt the plaintext
61  void            mpz_lshift( mpz_t, int );           //Left shift bin seq by 1
62  char*           pb( mpz_t, int );                   //Print prepending zeros
63
64  //------------------------------------------------------------------------------
65  //  MAIN FUNCTION
66  //------------------------------------------------------------------------------
67  int main(int argc, char *argv[]){
68    if( argc != 5 ){                          //Check required input parameters
69      printf("Too few arguments\n");
70      return 0;
71    }
72
73    deg = atoi( argv[1] );                    //Polynomial degree
74    m   = atoi( argv[2] );                    //Search word length
75    slen= atoi( argv[3] ) + 1 ;               //Allowed errors
76    n   = 2*m;                                //Serach text length 2m
77
78    printf("0: %s\n", argv[0]);               //Debug
79    printf("L:\t%d\n", deg);
80    printf("M:\t%d\n", m);
81    printf("N:\t%d\n", n);
82    printf("K:\t%d + 1\n", slen-1);
83
84    FNAME = malloc(60*sizeof(char));          //Filename allocation
85    sprintf(FNAME, "cipherserach_L%dM%dK%d_%s.log", deg, m, slen-1, argv[4]);
86
87    mpz_t max;    mpz_init( max );
88    mpz_setbit(max, deg);                     //Set max val, eg 2048 in 2^11
89
90    mpz_t pol;    mpz_init( pol );
91    if( deg == 11 ){                          //Set polynomial based on degree
92      mpz_set_ui(pol, 1209);                  //2^11 irreducible polynomial
93      printf( "POL: 1209\n" );
94    }
95    else if( deg == 16 ){
96      mpz_set_ui(pol, 33262);                 //2^16 irreducible polynomial
97      printf( "POL: 33262\n" );
98    }
99    else{
100     printf("Invaldig polynomial degree\n");
101     return 0;
102   }
103
104   printf("START:\t%s\n", argv[4]);              //Debug
105
106   mpz_init(PLAINTEXT);
```

```
107   mpz_set_ui(PLAINTEXT, 0);                      //Default value is 0
108
109   printf("START\n");
110   mpz_t*  B = genAlphabet( ALPHASIZE );          //Generate alphabet
111
112   mpz_t LCLK;   mpz_init(LCLK);                  //LFSR for dessimating
113   lfsrgen(LCLK, deg, m, pol,  300, 0, NULL);     //Clocking LFSR
114   mpz_t LDES;   mpz_init(LDES);                  //LFSR to be dessimated
115   lfsrgen(LDES, deg, n, pol, SSTATE, 0, NULL);   //Dessimated LFSR
116
117   mpz_t CIPHER;   mpz_init( CIPHER );            //Gen intercepted ciphertext
118   genEncrypt( CIPHER, LCLK, LDES );
119
120   genPrefixes( B, CIPHER);                       //Gen prefixes for the alphabet
121
122   char* t = pb(pol,deg);
123   //Output generated data
124   #if defined DEBUG
125   printf( "Setup:" );
126   printf( "\n\tPOLY:\t\t%s", t);     mpz_out_str(stdout, 2, pol);
127   free(t);  t = pb(PLAINTEXT, m);
128   printf( "\n\tPLAINTEXT:\t%s", t); mpz_out_str(stdout, 2, PLAINTEXT);
129   free(t);  t = pb(LCLK, m);
130   printf( "\n\tLFSR CLK:\t%s", t);  mpz_out_str(stdout, 2, LCLK);
131   free(t);  t = pb(LDES, n);
132   printf( "\n\tLFSR DES:\t%s", t);  mpz_out_str(stdout, 2, LDES);
133   free(t);  t = pb(CIPHER,m);
134   printf( "\n\tCIPHER:\t\t%s", t);  mpz_out_str(stdout, 2, CIPHER);
135   free(t);  t = pb(B[0], m);
136   printf( "\nAlphabet:\tB[0]:\t%s", t); mpz_out_str(stdout, 2, B[0]);
137   free(t);  t = pb(B[1], m);
138   printf( "\n\t\tB[1]:\t%s", t);  mpz_out_str(stdout, 2, B[1]);
139   free(t);
140   printf( "\n\n" );
141   #endif
142
143   mpz_clear( LCLK );                             //Cleanup LFSRs
144   mpz_clear( LDES );
145
146   mpz_t tmp;    mpz_init( tmp );                 //Geneate tmp variable
147
148   uint_least64_t i = 1;                          //initial state counter.
149   FILE*         fh = fopen(FNAME, "w");          //Open output file for writing
150
151   //While i less than 2^deg
152   while( mpz_cmp_ui( max, i ) > 0 ){
153     CSTATE = i;
154     mpz_set_ui( tmp, i);                         //For binary display
155     t = pb(tmp, deg);
156     #if defined DEBUG
157     printf("\nINITIAL STATE\t%"PRIu64"\t%s", i, t);
158     mpz_out_str( stdout, 2, tmp);   printf("\n");
159     #endif
160
161     lfsrgen( TEXT, deg, n, pol, i, 1, B );       //Undessimated bitseq iv=i
162     #if defined DEBUG
163     free(t);  t = pb(TEXT,n);
164     printf( "TEXT:\t\t%s", t );   mpz_out_str(stdout, 2, TEXT);
```

```
165       free(t);  t = pb(CIPHER,m);
166       printf( "\nCIPHER:\t\t%s", t); mpz_out_str(stdout, 2, CIPHER);
167       printf( "\n" );
168       #endif
169
170       #if defined DEBUG
171       printf( "Perform ARBP search\n" );
172       #endif
173       mpz_t*  MATCH = arbp_search(B, slen);
174
175       free(t);  t = pb(tmp, deg);
176       //Print initial state and all matches to screen and file
177       #if defined DEBUG
178       printf( "INITSTATE\t%"PRIu64"\n", i);
179       printf("\nMATCH\t" );
180       fprintf( fh, "INITSTATE\t%"PRIu64"\t%s", i, t );
181       mpz_out_str(fh, 2, tmp);
182       fprintf( fh, "\nMATCH\t" );
183       #else
184       if( CSTATE == SSTATE ){
185         printf( "INITSTATE\t%"PRIu64"\n", i);
186         fprintf( fh, "INITSTATE\t%"PRIu64"\t%s", i, t );
187         mpz_out_str(fh, 2, tmp);
188         fprintf( fh, "\nMATCH\t" );
189       }
190       #endif
191       free( t );
192
193       #if !defined DEBUG
194       if( CSTATE == SSTATE ){
195       #endif
196         int j     = 0;
197         int k     = 0;
198         while( j<n ){                                   //For each position
199           if( mpz_cmp_ui(MATCH[j], m) < 0){             //If match is less than m
200             if( k > 0 && (k%15)==0 ){                   //Decoration of stdout
201               printf("\n\t");
202             }
203             printf( "\t%d:", j); mpz_out_str(stdout, 10, MATCH[j] );
204             fprintf( fh, " %d:", j);  mpz_out_str( fh, 10, MATCH[j] );
205             mpz_clear( MATCH[j] );
206             k++;                                        //increment counter for print
207           }
208           j++;                                          //Next position
209         }
210         fprintf( fh, "\n");
211         printf( "\n\n" );
212         fprintf( fh, "\n" );
213         fflush( fh );
214       #if !defined DEBUG
215       }
216       #endif
217       i++;                                              //Next initial state
218       free(MATCH);
219     }
220     mpz_clear( TEXT );                                  //Clear variables
221     mpz_clear( PLAINTEXT );
222     mpz_clear( CIPHER );
```

```
223   mpz_clear( B[0] );
224   mpz_clear( B[1] );
225   free( B );                                    //Free up memory
226   fclose( fh );                                 //Close data file
227   printf("\n\nSoftware done\n");
228   return 0;
229 }
230
231
232
233 /*************************************************************************
234  **
235  ** FUNCTIONS
236  **
237  *************************************************************************/
238 /*-------------------------------------------------------------------------
239  * Prepend zeros to a binary representation of a number.
240  *  returns a char pointer to an array filled with missing zeros or nothing
241  *  if it is already full.
242 -------------------------------------------------------------------------*/
243 char* pb( mpz_t num, int len ){
244   size_t plen = len+1 - mpz_sizeinbase( num, 2 );
245   if( plen == 0 )
246     return "";
247   char* pre = malloc( plen * sizeof(char) );      //Allocate plen length array
248   int i = 0;
249   while( i < plen-1 ){
250     pre[i++] = '0';
251   }
252   pre[i]='\0';
253   return pre;
254 }
255
256 /*-------------------------------------------------------------------------
257  *  Left shift MPZ_T variable to the left
258  *    n number of shift
259  *    rop is mpz_t value to shift
260 -------------------------------------------------------------------------*/
261 void mpz_lshift( mpz_t rop, int len ) {
262   mpz_t tmp;  mpz_init( tmp );              //temp variable
263   int i   = len-1;
264   while( i > 0 ){
265     if(mpz_tstbit(rop, i-1) == 1){
266       mpz_setbit( tmp, i );                 //set bit if it
267     }
268     i--;
269   }
270   mpz_set(rop, tmp);                        //Set original var to tmp
271   mpz_clear(tmp);
272 }
273
274 /*-------------------------------------------------------------------------
275  * Generates the cipher which then becomes the prefix.
276  *  Y is the clocking LFSR
277  *  X is the encrypting LFSR
278 -------------------------------------------------------------------------*/
279 void genEncrypt(mpz_t rop, mpz_t CLK, mpz_t DES){
280   int   i       = 0;    int   j       = 0;
```

```
281   int   x, y;
282   mpz_t CIPHER;   mpz_init(CIPHER);
283
284   while( i < m ){                    //Counter for clocking lfsr
285     x = 0;    y = 0;
286     #if defined DEBUG
287     printf("CLK(%d)=%d  ", i, mpz_tstbit(CLK, i) );
288     #endif
289
290     if( mpz_tstbit(CLK, i) == 1 ){
291         j++;                         //Decimate LFSR by skipping a bit
292         #if defined DEBUG
293         printf("\tDES(%d)=%d --> \tDES(%d)=%d", j-1,
294                 mpz_tstbit(DES, j-1), j, mpz_tstbit(DES, j) );
295         #endif
296     }
297     #if defined DEBUG
298     else{
299       printf("\t\t\tDES(%d)=%d", j, mpz_tstbit(DES,j));
300     }
301     #endif
302
303     //Val of dessimated LFSR output
304     if( mpz_tstbit(DES, j) == 1 )          x = 1;  //grab value of bit
305     if( mpz_tstbit(PLAINTEXT, i) == 1 )    y = 1;  //Grab value of bit
306     if( (y^x) == 1 )  mpz_setbit(CIPHER, i);       //Xor to get CIPHER
307
308     #if defined DEBUG
309     printf(" ^ P(%d)=%d == C(%d)=%d\n", i, mpz_tstbit(PLAINTEXT, i),
310                                         i, mpz_tstbit(CIPHER, i) );
311     #endif
312     i++;    j++;                       //Increment counters
313   }
314   #if defined DEBUG
315     printf("\n");
316   #endif
317   mpz_set(rop, CIPHER);                //Set var to generated cipher
318   mpz_clear( CIPHER );
319 }
320
321
322 /*------------------------------------------------------------------------
323  *  Generate initial m-bitmasks for the alphabet of ALPHASIZE
324 ------------------------------------------------------------------------*/
325 mpz_t* genAlphabet( int ALPHASIZE ){
326   mpz_t* B = malloc((ALPHASIZE)*sizeof( mpz_t ));
327   #if defined DEBUG
328   printf( "Generating 0-masks for the alphabet\n" );
329   #endif
330
331   int i=0;
332   while( i < ALPHASIZE ){
333     mpz_init( B[i] );
334     mpz_set_ui( B[i], 0 );
335     #if defined DEBUG
336     printf("\tB[%d]",i);
337     mpz_out_str(stdout, 2, B[i]);
338     printf("\n");
```

```
339       #endif
340       i++;
341     }
342     #if defined DEBUG
343     printf( "\tDone\n\n" );
344     #endif
345     return B;
346   }
347
348
349
350   /*-------------------------------------------------------------------------
351    * LFSR iteration function
352    *    Calculates the next state of the LFSR by first grabbing the MSB as output
353    *    value.  Then it calculates the AND of cur state and the polynomial before
354    *    running an XOR on all bits that are set in the temp var to generate the
355    *    feedback value.
356    *
357    *    Finally it left shifts the entire original state and sets the LSB to the
358    *    value of the feedback polynomial.
359    *
360    *    The feedback value is then set
361   -------------------------------------------------------------------------*/
362   int lfsr_iterate( struct LFSR* lfsr) {
363     int     i       = 0;                    //Counter
364     int     fbck    = 0;                    //XOR calculated value
365     int     ret     = mpz_tstbit( lfsr->STATE, lfsr->DEGREE-1 );
366
367     mpz_t tmp;    mpz_init( tmp );
368     mpz_and(tmp, lfsr->STATE, lfsr->POLYNOMIAL);
369
370     while( i < lfsr->DEGREE ){              //Calc feedback
371       fbck = fbck + mpz_tstbit( tmp, i );
372       i++;
373     }
374     fbck = fbck % 2;
375     mpz_lshift(lfsr->STATE, lfsr->DEGREE);  //Left shift
376     if( fbck == 1 ) mpz_setbit( lfsr->STATE, 0 );
377
378     return ret;                             //Return output character
379   }
380
381
382   /*-------------------------------------------------------------------------
383    * Generate LFSR and output an n-length bitsequence
384    * With all arbitrary skips until first prefix is met
385   -------------------------------------------------------------------------*/
386   void lfsrgen(mpz_t rop, int psize, int olen, mpz_t p,
387                uint_least64_t iv, int skip, mpz_t* B){
388     int i;                                  //Counter var
389     int initmatch = 0;                      //Check if first prefix is found
390     struct LFSR lfsr;                       //Create struct variable
391     #if defined DEBUG
392     printf("Generating LFSR\n");
393     #endif
394
395     lfsr.DEGREE = psize;                    //Set polynomial degree
396
```

```
397   mpz_init( lfsr.POLYNOMIAL );
398   mpz_set(lfsr.POLYNOMIAL, p);          //Set polynomial
399
400   mpz_init( lfsr.STATE );
401   mpz_set_ui(lfsr.STATE, iv);           //Set initial vector (seed)
402
403   mpz_t OUTPUT;
404   mpz_init( OUTPUT );
405   int   tmpOUT  = 0;                    //temp char holder
406   char* t;
407   i             = 0;                    //Zero out counter
408   while( i < olen ){
409     tmpOUT = lfsr_iterate(&lfsr);
410     if( initmatch == 0 && skip == 1) {
411       if( mpz_tstbit(B[tmpOUT], 0) == 1 ){
412         initmatch = 1;                  //Set state to found
413         if( tmpOUT == 1 )
414           mpz_setbit(OUTPUT, i);        //Set output to tmpvar
415         i++;                            //inc counter
416
417         #if defined DEBUG
418         t = pb(B[0],m);
419         printf("\tinitmatch found\t\t%d |\t0: %s", tmpOUT, t);
420         mpz_out_str( stdout, 2, B[0] );
421         free(t); t = pb(B[1],m);
422         printf("\n\t\t\t\t\t\t1: %s", t);   mpz_out_str( stdout, 2, B[1] );
423         printf("\n");
424         free(t);
425         #endif
426       }
427       #if defined DEBUG
428       else{
429         t = pb(B[0],m);
430         printf("\tinitmatch not found\t%d |\t0: %s", tmpOUT, t);
431         mpz_out_str( stdout, 2, B[0] );
432         free(t); t = pb(B[1],m);
433         printf("\n\t\t\t\t\t\t1: %s", t);     mpz_out_str( stdout, 2, B[1]);
434         printf("\n");
435         free(t);
436       }
437       #endif
438     }
439     else {
440       if( tmpOUT == 1)  mpz_setbit(OUTPUT, i);
441       i++;                              //Next bit
442     }
443   }
444
445   //Print LFSR information
446   #if defined DEBUG
447   printf("\n");
448   printf("\tDEGREE:\t%d\n", psize);
449   printf("\tLENGTH:\t%d\n", olen);
450   printf("\tPOL:\t"); mpz_out_str( stdout, 2, lfsr.POLYNOMIAL );
451   printf("\n\tSTT:\t"); mpz_out_str( stdout, 2, lfsr.STATE );
452   printf("\n");
453   printf("\tDone\n");
454   #endif
```

```
455   mpz_set(rop, OUTPUT);
456   mpz_clear(OUTPUT);
457 }
458
459
460
461 /*-------------------------------------------------------------------------
462  * Creates the prefixes
463 -------------------------------------------------------------------------*/
464 void genPrefixes( mpz_t* B, mpz_t P ){
465   int  j = 0;
466   #if defined DEBUG
467   printf("Generating prefixes\n");
468   #endif
469   mpz_t tmp;  mpz_init(tmp); mpz_set_ui( tmp, 1 );
470   while( j<m ){
471     int ci = mpz_tstbit(P, j);        //Char value 0/1
472
473     #if defined DEBUG
474     printf( "j=%d\tPj=%d |\t%s", j, mpz_tstbit(P,j), pb(B[ci],m) );
475     mpz_out_str(stdout, 2, B[ci]);
476     #endif
477
478
479     mpz_ior( B[ci], B[ci], tmp );     //current value or-ed with 10^(j-1)
480     mpz_lshift( tmp, m );
481
482     #if defined DEBUG
483     printf( "\n\t\t%s", pb(B[ci],m));
484     mpz_out_str( stdout, 2, B[ci]);   printf( "\n" );
485     #endif
486
487     j++;                              //Next pattern character
488   }
489   mpz_clear( tmp );
490   #if defined DEBUG
491   printf("\n\tDone\n\n");
492   #endif
493 }
494
495
496
497 /*-------------------------------------------------------------------------
498  * Creates the error list of K-size
499 -------------------------------------------------------------------------*/
500 mpz_t* genError(int K) {
501   mpz_t*  R = malloc( K*sizeof(mpz_t) );    //Allocate memory for array
502   #if defined DEBUG
503   printf("Gen error R[%d..%d]\n", 0, K-1);
504   #endif
505   int k = 0;                                //Set counter
506   while( k<K ){
507     int i = 0;
508     mpz_init( R[k] );
509     while( i < k ){
510       mpz_setbit( R[k], i );
511       i++;
512     }
```

```
513       #if defined DEBUG
514       printf("\tR[%d]\t= ", k);
515       mpz_out_str( stdout, 2, R[i]);
516       printf( "\n" );
517       #endif
518       k++;
519     }
520     #if defined DEBUG
521     printf("\tDone\n");
522     #endif
523     return R;
524   }
525
526
527   /*-------------------------------------------------------------------------
528    * Perform search on TEXT and PREFIX
529    -------------------------------------------------------------------------*/
530   mpz_t* arbp_search(mpz_t* B, int K) {
531     mpz_t      tmp1;    mpz_init( tmp1 );    //Tmp variables
532     mpz_t      tmp2;    mpz_init( tmp2 );
533                                              //Match for each position in text
534     mpz_t*     R        = genError( K );     //Gen error-table
535     mpz_t*     MATCHES = malloc( n * sizeof(mpz_t) );
536
537     mpz_t oldR, newR;                        //Create and init variables
538     mpz_init( oldR );    mpz_init( newR );
539
540     #if defined DEBUG
541     printf( "Beginning search\n");           //Debug
542     #endif
543     uint_least64_t pos  = 0;                 //Start search from char 1
544     while( pos < n ){                        //Search entire string
545       int Ti  = mpz_tstbit(TEXT, pos);       //Grab current chars int value
546
547       mpz_clear( oldR );  mpz_init( oldR ); //Reset variables
548       mpz_clear( newR );  mpz_init( newR );
549
550       mpz_set( oldR, R[0]);                  //Init oldR to cur R[0] (R[i])
551
552       mpz_set(tmp1, R[0]);
553       mpz_lshift( tmp1, m );                 //lshift
554       mpz_setbit( tmp1, 0 );                 //OR with 1
555       mpz_and( tmp1, tmp1, B[Ti] );          //AND with B[Ti]
556
557       mpz_set( newR, tmp1 );                 //Set newR to tmp
558       mpz_set(R[0], newR);                   //Set R[0] to R'[i]
559
560       uint_least64_t i  = 1;                 //Calc matches with K allowed errors
561       while( i < K ) {
562         mpz_clear( tmp1 );  mpz_clear(tmp2);
563         mpz_init(tmp1);     mpz_init(tmp2);//reset and initialise temp variables
564
565                                              //Substitute and deletion
566         mpz_ior(tmp2, oldR, newR);           //tmp2 = (oldR|newR)
567         mpz_lshift(tmp2, m);                 //tmp2 = <tmp2> << 1
568         mpz_setbit( tmp2, 0 );               //tmp2 = <tmp2> | 1
569         #if !defined INC_INSERT
570                                              //Insertion
```

```
571         mpz_ior(tmp2, oldR, tmp2);          //tmp2 = oldR | <tmp2>
572         #endif
573
574         mpz_set(tmp1, R[i]);                //Copy value
575         mpz_lshift(tmp1, m);                //tmp1 = R[i]<<1
576         mpz_and(tmp1, tmp1, B[Ti]);         //tmp1 = <tmp1> & B[Ti]
577
578         mpz_ior(tmp1, tmp1, tmp2);          //tmp1 = <tmp1> | <tmp2>
579
580         mpz_set(newR, tmp1);                //newR = <tmp1>
581         mpz_set(oldR, R[i]);                //Store R[i] for next error
582         mpz_set(R[i], newR);                //R[i] == R'[i]
583
584         i++;                                //Next error
585       }
586
587       #if !defined DEBUG
588       if( CSTATE == SSTATE ){
589       #endif
590         mpz_init( MATCHES[pos] );
591         mpz_set_ui( MATCHES[pos], m );      //Init val of match at cur pos
592         int j = 0;                          //Init counter
593         if( mpz_tstbit(newR, m-1) == 1 ){   //Check if R-table has a match
594           while( j<K ){                     //Loop R-table for matches (MSB set)
595             if(mpz_tstbit(R[j], m-1) == 1){ //Check if MSB set
596               mpz_set_ui(MATCHES[pos], j);  //Set match to the R-level (0-K)
597               j = K;                        //Skip to end
598             }
599             j++;                            //Next error value
600           }
601         }
602       #if !defined DEBUG
603       }
604       #endif
605       pos += 1;                             //Next position in search text
606     }
607
608     #if defined DEBUG
609     printf("Search done.\n");
610     #endif
611     mpz_clear( oldR );
612     mpz_clear( newR );
613     free(R);
614
615     #if defined DEBUG
616     return MATCHES;
617     #else
618     if( CSTATE == SSTATE )  return MATCHES;
619     else                    return NULL;
620     #endif
621   }
```

Listing B.1: C ARBP source code

B.2 FPGA Implementation and Design in Verilog

B.2.1 Testbench

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Engineer: Magnus Overbo
4  //
5  // Create Date:   20190212
6  // Revision Date: 20190324
7  // Design Name:   FPGA Ciphersearch
8  // Module Name:   testbench
9  // Project Name:
10 // Target Devices: FPGA Ciphersearch
11 //
12 // Revision:
13 // Revision 0.01 - File Created
14 // Revision 1.00 - Testbench simulating clock, switches and LED registers
15 //////////////////////////////////////////////////////////////////////////////////
16
17 module testbench();
18   reg           synclock;     //Create reg for synclock
19
20   reg    [15:0] SWreg=0;      //Simulate button state
21   wire   [15:0] SWwire;
22
23   reg    [15:0] LEDreg;       //Simulate LED pins
24   wire   [15:0] LEDwire;
25
26   reg           UARTreg;      //Simulate UART pin
27   wire          UARTwire;
28
29   //Modules
30   arbp s1( synclock, SWreg, LEDwire, UARTwire);
31
32   initial begin
33     synclock = 1'b0;          //Initialise simulated clock
34     #5000 $finish;            //Finish after 5k iterations
35   end
36
37   //Clock pulse simulation
38   always  #1 synclock = !synclock;
39
40 endmodule
```

Listing B.2: Verilog Testbench source code

B.2.2 ARBP search FPGA (main program)

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  //# Engineer:       Magnus Overbo
4  //#
5  //# Create Date:    20190219
6  //# Design Name:    FPGA Ciphersearch
7  //# Module Name:    arbp
8  //# Project Name:   FPGA Ciphersearch
9  //# Target Devices: nexys A7
10 //# Description:    Performs the ARBP search for a generated bitsequence.
11 //#                 Generated by the LFSR, starting at the "init" state
12 //#
13 //# Revision:
14 //# Revision 0.1 -  20190314 Initial version generated, with all as registers
15 //# Revision 0.2 -  20190322 Improved speed and refactoring M <= 256b
16 //# Revision 0.3 -  20190324
17 //////////////////////////////////////////////////////////////////////////////////
18
19 `define DEGREE11
20 //`define DEGREE16
21 //`define DEGREE20
22 //`define DEGREE30
23
24 //`define INC_INSERT
25
26
27 //////////////////////////////////////////////////////////////////////////////////
28 //# Approximate RBP Search
29 //# Generates an LFSR of length N and performs ARBP search after each bit is
30 //# generated by an LFSR
31 //////////////////////////////////////////////////////////////////////////////////
32 module arbp(input clk, input [15:0] SW, output [15:0] LED, output UART_RXD_OUT);
33 //-------------------------------------------------------------------------------
34 //--  INPUT VARIABLES
35 //-------------------------------------------------------------------------------
36 reg [15:0]  LEDreg  = 16'h0000; //Values sent to output LED
37 reg [15:0]  SWreg   = 16'h0000; //Switch registers
38 reg         runclock= 0;        //Halftime clock
39
40
41 //-------------------------------------------------------------------------------
42 //--  PARAMETER DEFINITIONS
43 //-------------------------------------------------------------------------------
44 `ifdef DEGREE11
45     parameter POL           = 11;         //Polynomial degree
46     parameter MAXSTATEVAL   = 2048;       //Value to mod LFSR state with
47 `elsif DEGREE16
48     parameter POL           = 16;         //Polynomial degree
49     parameter MAXSTATEVAL   = 65536;      //Value to mod LFSR state with
50 `elsif DEGREE20
51     parameter POL           = 20;         //Polynomial degree
52     parameter MAXSTATEVAL   = 1048576;    //Value to mod LFSR state with
53 `elsif DEGREE30
54     parameter POL           = 30;         //Polynomial degree
55     parameter MAXSTATEVAL   = 1073741824; //Value to mod LFSR state with
56 `endif
```

```
57
58
59  parameter M           = 256;  //Size of cipher/prefix/search word
60  parameter N           = 512;  //Size of search text
61  parameter K           = 69;   //Max error size
62  parameter Kbin        = 32;   //Lengt needed to represent address space
63  parameter Blen        = 8;    //Length of a byte for UART transmission
64
65
66  parameter SOLVESTATE  = 1024; //Init state to use for LFSR2 and as output state
67  parameter DESSTATE    = 300;  //Init state for clocking LFSR
68
69
70  //--------------------------------------------------------------------------------
71  //--  GLOBAL VARIABLES
72  //--------------------------------------------------------------------------------
73  //Prefixes
74  reg      [(M-1):0]  B[0:1];       //Alphabet prefixes
75
76  //ARBP Error calculation
77  reg        [M-1:0]  Rold;         //R[i]
78  reg        [M-1:0]  Rnew;         //R'[i]
79
80  //LFSR variables
81  `ifdef DEGREE11
82      //Feedback pol: x^11 + x^8 + x^6 + x^5 + x^4 + x^1 + 1
83      reg   [POL-1:0] POLYNOMIAL = 11'b10010111001;
84  `elsif DEGREE16
85      //Feedback pol: x^16 + x^9 + x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + 1
86      reg   [POL-1:0] POLYNOMIAL = 16'b1000000111101110;
87  `elsif DEGREE20
88      //Feedback pol: x^20 + x^9 + x^5 + x^3 + 1
89      reg   [POL-1:0] POLYNOMIAL = 20'b10000000000100010100;
90  `elsif DEGREE30
91      //Feedback pol: x^30 + x^23 + x^2 + x^1 + 1
92      reg   [POL-1:0] POLYNOMIAL = 30'b10000001000000000000000000000011;
93  `endif
94
95  reg     [POL-1:0] LFSRinitState = 1;//Initial state of LFSR
96  reg     [POL-1:0] LFSRstate;       //current state of LFSR
97  reg               LFSRfbck;        //Feedback variable
98
99  //Temp variables
100 integer       r           = 0;     //Counter for error calc loop
101 integer       i           = 0;     //genearal purpose counter
102 reg           STATEtrigger= 0;     //Triggers end of current state
103
104 //FSM variables
105 reg           Ti;                  //Current value
106 reg           INITFIND    = 0;     //Reg showing that first val is found
107 integer       POSITION    = 0;     //Position in text
108 integer       STATE       = 0;     //FSM state variable
109
110 //UART variables
111 wire          uartIsTxWire;        //is_transmitting wire where val is read
112 reg           uartIsTx;            //stored tx state
113 reg           uartReset   = 0;     //rst
114 reg           uartTx      = 0;     //transmit
```

```
115 integer       txState     = 0;      //State for UART transmission
116 integer       uartCounter = 0;      //Counter for uart
117 integer       uartState   = 0;      //State variable for uart
118 reg      [7:0]uartTxData  = 0;      //tx_byte
119 reg     [31:0]uartTxLFSRState;      //Stores current init states
120
121 //UART control variables
122 reg           uartTxWait    = 0;    //Wait / signal variable for arbp
123 reg           uartTxWaitAck = 0;    //Wait / signal variable for uart
124
125 //LFSR for generating cipher
126 integer       ENCSTATE      = 0;
127 `ifdef DEGREE11
128     //Polynomial deg 11
129     reg [POL-1:0] LFSR_CLK_POLY = 11'b10010111001;
130     reg [POL-1:0] LFSR_DES_POLY = 11'b10010111001;
131 `elsif DEGREE16
132     //Polynomial deg 16
133     reg [POL-1:0] LFSR_CLK_POLY = 16'b1000000111101110;
134     reg [POL-1:0] LFSR_DES_POLY = 16'b1000000111101110;
135 `elsif DEGREE20
136     //Polynomial deg 20
137     reg [POL-1:0] LFSR_CLK_POLY = 20'b10000000000100010100;
138     reg [POL-1:0] LFSR_DES_POLY = 20'b10000000000100010100;
139 `elsif DEGREE30
140     //Polynomial deg 30
141     reg [POL-1:0] LFSR_CLK_POLY = 30'b10000001000000000000000000000011;
142     reg [POL-1:0] LFSR_DES_POLY = 30'b10000001000000000000000000000011;
143 `endif
144
145 reg     [POL-1:0] LFSR_CLK_STATE= DESSTATE;
146 reg     [POL-1:0] LFSR_DES_STATE= SOLVESTATE;
147 reg               LFSR_CLK_BIT;
148 reg               LFSR_DES_BIT;
149
150 reg               PTXT          = 0;
151 reg               CIPHERBIT     = 0;
152
153 reg               R_write       = 0;
154 reg     [Kbin-1:0]R_addr        = 0;
155 reg        [M-1:0]R_idata       = 0;
156 wire       [M-1:0]R_odata_wire;
157
158
159
160 //----------------------------------------------------------------------------
161 //--  ASSIGN
162 //----------------------------------------------------------------------------
163 assign LED = LEDreg;
164
165
166 //----------------------------------------------------------------------------
167 //--  Modules
168 //----------------------------------------------------------------------------
169 uart #(.CLOCK_DIVIDE(217)) console(clk, uartReset, UART_RXD_OUT, uartTx,
170         uartTxData, uartIsTxWire);
171 //uart #(.CLOCK_DIVIDE(108)) console(runclock, uartReset, UART_RXD_OUT, uartTx,
172 //         uartTxData, uartIsTxWire);
```

```
173  sram #(.ADDR_WIDTH(Kbin), .DATA_WIDTH(M), .DEPTH(K)) R_array
174          (.clk(clk), .i_addr(R_addr), .i_write(R_write), .i_data(R_idata),
175           .o_data(R_odata_wire) );
176
177  //-----------------------------------------------------------------------------
178  //--  Independent update of Swith positions
179  //-----------------------------------------------------------------------------
180  always @(posedge clk) begin
181    SWreg <= SW;
182    runclock <= !runclock;
183  end
184
185  //-----------------------------------------------------------------------------
186  //--  FSM for running ARBP search
187  //-----------------------------------------------------------------------------
188  always @(posedge clk) begin
189    //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
190    //++  LFSR: Generate CIPHER text bit by bit
191    //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
192    if( STATE == 0 ) begin      //Calculate LFSR
193      if( r ==  M ) begin
194        ENCSTATE      <= 0;
195        STATE         <= 1;
196      end
197      else begin
198        if( ENCSTATE == 0 ) begin                                  //Gen clock bit
199          LFSR_CLK_BIT      <= LFSR_CLK_STATE[POL-1];              //MSB = output
200          LFSR_CLK_STATE    <= LFSR_CLK_STATE << 1;                //Left shift
201          LFSR_CLK_STATE[0] <= ^(LFSR_CLK_POLY & LFSR_CLK_STATE); //feedback val
202          ENCSTATE          <= 1;                                  //Next state
203        end
204        else if( ENCSTATE == 1 ) begin                             //Calculate bit
205          LFSR_DES_BIT      <= LFSR_DES_STATE[POL-1];              //MSB = output
206          LFSR_DES_STATE    <= LFSR_DES_STATE << 1;                //Left shift
207          LFSR_DES_STATE[0] <= ^(LFSR_DES_POLY & LFSR_DES_STATE); //feedback val
208          ENCSTATE          <= 2;                                  //Next state
209        end
210        else if( ENCSTATE == 2 ) begin              //Store as cipher bit
211          if( LFSR_CLK_BIT == 0 ) begin             //Output bit ready
212            CIPHERBIT   <= LFSR_DES_BIT ^ PTXT;     //XOR with plaintext
213            ENCSTATE    <= 3;                       //Next state
214          end
215          else begin
216            LFSR_CLK_BIT<= LFSR_CLK_BIT - 1;        //Decimate LFSR
217            ENCSTATE    <= 1;                       //Generate next LFSR bit
218          end
219        end
220        else if( ENCSTATE == 3 ) begin
221          B[0][r]       <= !CIPHERBIT;              //Generate 0 prefix
222          B[1][r]       <= CIPHERBIT;               //Generate 1 prefix
223          r             <= r + 1;                   //Increment counter <M
224          ENCSTATE      <= 0;                       //Go to initial state
225        end
226      end
227    end
228
229    //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
230    //++  Initial state
```

```
231   //++    Reset variables, error array and move on to next stage
232   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
233   else if( STATE == 1 ) begin
234     if( i == K ) begin                //Preliminary setup, done.
235       i             <= 0;             //Reset counter
236       r             <= 0;             //
237       STATEtrigger  <= 0;             //Unset reg
238       POSITION      <= 0;             //Zero counter for position in text
239       STATE         <= 2;             //Next stage in FSM
240       INITFIND      <= 0;             //Zero out first find
241       R_write       <= 0;
242       R_addr        <= 0;
243       LFSRstate     <= LFSRinitState; //Store value for later use
244     end
245     else begin
246       if( i == 0 ) begin
247         R_addr      <= i;
248         R_idata     <= 0;
249         R_write     <= 1;
250       end
251       else if( i < K) begin
252         R_idata     <= (R_idata << 1) | 1;
253         R_addr      <= i;
254       end
255       i    <= i + 1;
256     end
257   end
258
259
260   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
261   //++  LFSR: grab output bit and iterate the LFSR
262   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
263   else if( STATE == 2 ) begin           //Calculate LFSR
264     if( STATEtrigger ) begin            //LFSR gen done
265       i               <= 0;             //reset variables
266       r               <= 0;             //
267       STATEtrigger    <= 0;             //
268       R_addr          <= 0;             //
269       R_write         <= 0;             //
270       uartTxLFSRState <= LFSRinitState; //
271
272       //Trigger on initial match, and all subsequent values
273       if( B[Ti][0]==1 || INITFIND==1 ) begin
274         INITFIND      <= 1;             //Set INITFIND
275         STATE         <= 3;             //Next State
276       end
277       else begin                        //Initial loop until first match
278         STATE <= 2;                     //Don't count as position
279       end
280     end
281     else begin
282       Ti              <= LFSRstate[POL-1];          //grab MSB as output
283       LFSRstate       <= LFSRstate << 1;            //Left shift register
284       LFSRstate[0]    <= ^(POLYNOMIAL & LFSRstate); //AND pol with state
285       STATEtrigger    <= 1;                         //Trigger done stat
286     end
287   end
288
```

```
289
290   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
291   //++  Initialise ARBP search R0 Exact search
292   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
293   else if( STATE == 3 ) begin           //Primary ARBP search
294     if( R_write == 0 ) begin
295       Rold            <= R_odata_wire;  //
296                                         //R0 exact search
297       {R_idata, Rnew} <= {2{ (((R_odata_wire<< 1) | 1) & B[Ti]) }};
298       R_write         <= 1;             //
299     end
300     else begin
301       STATE           <= 4;             //
302       r               <= 0;             //
303       R_write         <= 0;             //
304     end
305   end
306
307
308   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
309   //++  Run ARBP algorithm for current position
310   //++  With continous output to LED array
311   //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
312   else if( STATE == 4 ) begin
313     if(r >= K ) begin             //Algorithm finished
314       r               <= 0;       //
315       STATEtrigger    <= 0;       //Reset trigger
316       R_write         <= 0;       //Stop write
317       R_addr          <= 0;       //Reset BRAM address
318       //Output data via UART
319       if( SW[0] == 0 ) begin      //Run output always
320         STATE         <= 5;       //Next stage in FSM
321       end
322                                   //Show solve-state matches only
323       else if( SW[0] == 1 && LFSRinitState == SOLVESTATE )  begin
324         STATE         <= 5;       //Next stage in FSM
325       end
326       else begin                  //Skip output
327         STATE         <= 6;       //Next stage in FSM
328       end
329       //Trigger LED indicators for LFSR state and matches
330       if(POL <= 16 )
331        LEDreg <= LFSRinitState;    //Current initial state
332       else
333        LEDreg <= LFSRinitState[POL-1:POL-16];    //Current initial state
334
335     end
336     else begin
337       case( i )
338       0:  begin
339         R_addr  <= R_addr + 1;
340         r       <= r + 1;
341         R_write <= 0;
342       end
343       2:  begin
344         Rold            <= R_odata_wire;
345         `ifdef INC_INSERT
346         //Insert transtition allowed
```

```
347           {R_idata,Rnew}  <= {2{((R_odata_wire<<1)&B[Ti])|Rold|((((Rold|Rnew)<<1)|1)}};
348           `else
349           //Insert transition removed
350           {R_idata,Rnew}  <= {2{((R_odata_wire<<1)&B[Ti])|((((Rold|Rnew)<<1)|1)}};
351           `endif
352         end
353         3:  begin
354           R_write  <= 1;
355         end
356         endcase
357         i = (i + 1) % 4;
358       end                                 //else
359     end                                   //State
360
361
362     //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
363     //++  Transfer match data
364     //++  Mark ready for transmission, wait until ack signal is recieved.
365     //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
366     else if( STATE == 5 ) begin
367       if( uartTxWait == 1 ) begin               //Wait until transmission done
368         if( uartTxWaitAck == 1 ) begin          //Reset values, go to next state
369           uartTxWait  <= 0;
370           STATE       <= 6;
371         end
372       end
373       else if( Rnew[M-1] == 0 ) begin         //NO matches
374         STATE         <= 6;
375       end
376       else if( R_odata_wire[M-1] == 1 ) begin //Match found
377         uartTxWait    <= 1;                     //
378       end
379       else if( R_addr == K-1 && r == 0 ) begin
380         STATE         <= 6;
381       end
382       else begin                                //
383         if( r==1 )  begin                       //
384           R_addr      <= R_addr + 1;            //
385         end
386         r <= (r + 1) % 4;                       //Iterate wait variable
387       end
388     end
389
390
391     //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
392     //++  Reset and increment POSITION variable
393     //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
394     else if( STATE == 6 ) begin
395       STATEtrigger  <= 0;                       //Reset statetrigger
396       POSITION      <= POSITION + 1;            //Inc position
397       R_addr        <= 0;
398       r             <= 0;                       //Reset remaining vars
399       i             <= 0;
400       Rold          <= {M{1'b0}};
401       Rnew          <= {M{1'b0}};
402       STATE         <= 7;
403     end
404
```

```
405
406   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
407   //++  Returns you to state 0 with next intial state if POSITION N is reached
408   //++    otherwise return you to STATE 1 for next value
409   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
410   else if( STATE == 7 ) begin
411     if( POSITION >= N ) begin               //If end of search
412       STATE       <= 1;                     //  go to 0 state
413       if( LFSRinitState == MAXSTATEVAL-1 ) begin
414         LFSRinitState <= 1;                 //Skip 0 state (loop of death)
415       end
416       else begin
417         LFSRinitState <= LFSRinitState + 1; //Next LFSR init state
418       end
419     end
420     else  STATE   <= 2;                     //Jump to next position
421   end
422
423
424   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
425   //++  Catch all state return to initial State
426   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
427   else begin
428     LFSR_DES_STATE  <= DESSTATE;      //
429     LFSR_CLK_STATE  <= SOLVESTATE;    //
430     STATE           <= 0;             //Something is wrong, go to zero
431   end
432 end
433
434
435
436
437
438
439 //-------------------------------------------------------------------------------
440 //--  UART transmission control
441 //-------------------------------------------------------------------------------
442 //--  uartTxWait == 0 - ARBP search does not need data transmitted
443 //--  uartTxWait == 1 - ARBP wants data transmitted and awaits ack signal
444 //--  uartTxWaitAck == 0 - Zero untill entire protocol has been transmitted
445 //--  uartTxWaitAck == 1 - 1 to signal ARBP that tx is complete
446 //-------------------------------------------------------------------------------
447 always @(posedge clk) begin
448   uartIsTx <= uartIsTxWire;             //Load current state value into reg
449
450   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
451   //++  Wait while ARBP state is signalling stop
452   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
453   if( uartTxWait == 0 ) begin
454     uartCounter   <= 1;
455     uartTxWaitAck <= 0;
456     uartReset     <= 0;
457     uartTxData    <= 8'h00;
458     uartState     <= 0;
459     uartTx        <= 0;
460   end
461
462   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
463   //++  Hinder state 0 while ack is being sent
464   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
465   else if( uartTxWaitAck == 1 ) begin
466   end
467
468   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
469   //++  SET transmission data
470   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
471   else if( uartState == 0 ) begin
472     case(uartCounter)
473       //Transfer POSITION as 32b number
474       1:  uartTxData    <= POSITION[ 7: 0];
475       2:  uartTxData    <= POSITION[15: 8];
476       3:  uartTxData    <= POSITION[23:16];
477       4:  uartTxData    <= POSITION[31:24];
478
479       //Transfer initial state  as a 32b number
480       5:  uartTxData    <= uartTxLFSRState[ 7: 0];
481       6:  uartTxData    <= uartTxLFSRState[15: 8];
482       7:  uartTxData    <= uartTxLFSRState[23:16];
483       8:  uartTxData    <= uartTxLFSRState[31:24];
484
485       //Transfer error level of match as a 32b number
486       9:  uartTxData    <= R_addr[ 7: 0];
487       10: uartTxData    <= R_addr[15: 8];
488       11: uartTxData    <= R_addr[23:16];
489       12: uartTxData    <= R_addr[31:24];
490
491         //Stop transmission
492       13: uartTxWaitAck <= 1;       //Signal transmission done
493     endcase
494     uartState   <= 2;               //Next state
495   end
496
497   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
498   //++  Signal transmission when transmission is "OFF"
499   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
500   else if( uartState == 2 ) begin
501     if( uartIsTx == 0 ) begin       // UART grabs the data
502       uartTx      <= 1;             // Signal transmission
503       uartState   <= 3;             // Next state
504     end
505   end
506
507   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
508   //++  Signal transmission stop when transmission has started
509   //++    Will not stop before it has transmitted 8b
510   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
511   else if( uartState == 3 ) begin
512     if( uartIsTx == 1 ) begin
513       uartState   <= 4;             //Next state
514       uartTx      <= 0;             //Pause transmission
515     end
516   end
517
518   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
519   //++  Increment counter and jump to state 0 when transmission has stopped
520   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
521     else if( uartState == 4 ) begin
522       if( uartIsTx == 0 ) begin
523         uartState   <= 0;             //Reset UART state
524         uartCounter <= uartCounter + 1;//Increment counter for next data
525       end
526     end
527   end
528
529   endmodule
```

Listing B.3: Verilog ARBP module source code

B.2.3 Block Ram Module (Verilog)

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Engineer: Magnus Overbo
4  //
5  // Create Date: 03/24/2019 12:22:01 PM
6  // Module Name: ram
7  // Revision   1.0 initial version
8  //////////////////////////////////////////////////////////////////////////////////
9
10 module sram #(parameter ADDR_WIDTH=8, DATA_WIDTH=8, DEPTH=256) (
11     input wire  clk,
12     input wire  [ADDR_WIDTH-1:0] i_addr,
13     input wire  i_write,
14     input wire  [DATA_WIDTH-1:0] i_data,
15     output reg  [DATA_WIDTH-1:0] o_data
16 );
17
18                                             //Create RAM array
19     reg [DATA_WIDTH-1:0] memory_array [0:DEPTH-1];
20
21     always @(posedge clk) begin
22         if( i_write ) begin                 //Write data to memory
23             memory_array[i_addr] <= i_data;
24         end
25         else begin                          //Store data to output reg
26             o_data <= memory_array[i_addr];
27         end
28     end
29
30 endmodule
```

Listing B.4: Verilog BRAM module

B.2.4 UART communication FPGA (shortened version Verilog)

```
1  `timescale 1ns / 1ps
2  // Documented Verilog UART
3  // Copyright (C) 2010 Timothy Goddard (tim@goddard.net.nz)
4  // Distributed under the MIT licence.
5  //
6  // Permission is hereby granted, free of charge, to any person obtaining a copy
7  // of this software and associated documentation files (the "Software"), to deal
8  // in the Software without restriction, including without limitation the rights
9  // to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10 // copies of the Software, and to permit persons to whom the Software is
11 // furnished to do so, subject to the following conditions:
12 //
13 // The above copyright notice and this permission notice shall be included in
14 // all copies or substantial portions of the Software.
15 //
16 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17 // IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18 // FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19 // AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20 // LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
21 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
22 // THE SOFTWARE.
23
24
25 module uart(
26     input clk,              // The master clock for this module
27     input rst,              // Synchronous reset.
28     output tx,              // Outgoing serial line
29     input transmit,         // Signal to transmit
30     input [7:0] tx_byte,    // Byte to transmit
31     output is_transmitting  // Low when transmit line is idle.
32 );
33
34 //100MHz clock which is derived to clock once every cycle == 50MHz
35 parameter CLOCK_DIVIDE = 217; // (100MHz / ( 115200*4)) = 217
36
37 // States for the transmitting state machine.
38 // Constants - do not override.
39 parameter TX_IDLE           = 0;
40 parameter TX_SENDING        = 1;
41 parameter TX_DELAY_RESTART  = 2;
42
43 reg [10:0] tx_clk_divider   = CLOCK_DIVIDE;
44
45 reg      tx_out             = 1'b1;
46 reg [1:0] tx_state          = TX_IDLE;
47 reg [5:0] tx_countdown;
48 reg [3:0] tx_bits_remaining;
49 reg [7:0] tx_data;
50
51 assign tx                  = tx_out;
52 assign is_transmitting     = tx_state != TX_IDLE;
53
54 always @(posedge clk) begin
55   if (rst)  tx_state <= TX_IDLE;
56
```

```
57    // The clk_divider counter counts down from the CLOCK_DIVIDE constant.
58    // Whenever it reaches 0, 1/16 of the bit period has elapsed. Countdown timers
59    // for the receiving and transmitting state machines are decremented.
60    tx_clk_divider <= tx_clk_divider - 1;
61    if (!tx_clk_divider) begin
62      tx_clk_divider  <= CLOCK_DIVIDE;
63      tx_countdown    <= tx_countdown - 1;
64    end
65
66    // Transmit state machine
67    case (tx_state)
68      TX_IDLE: begin
69        if (transmit) begin
70          // If the transmit flag is raised in the idle state, start
71          // transmitting the current content of the tx_byte input.
72          tx_data             <= tx_byte;
73          // Send the initial, low pulse of 1 bit period
74          // to signal the start, followed by the data
75          tx_clk_divider      <= CLOCK_DIVIDE;
76          tx_countdown        <= 4;
77          tx_out              <= 0;
78          tx_bits_remaining   <= 8;
79          tx_state            <= TX_SENDING;
80        end
81      end
82      TX_SENDING: begin
83        if (!tx_countdown) begin
84          if (tx_bits_remaining) begin
85            tx_bits_remaining <= tx_bits_remaining - 1;
86            tx_out            <= tx_data[0];
87            tx_data           <= {1'b0, tx_data[7:1]};
88            tx_countdown      <= 4;
89            tx_state          <= TX_SENDING;
90          end else begin
91            // Set delay to send out 2 stop bits.
92            tx_out            <= 1;
93            tx_countdown      <= 8;
94            tx_state          <= TX_DELAY_RESTART;
95          end
96        end
97      end
98      TX_DELAY_RESTART: begin
99        // Wait until tx_countdown reaches the end before we send another
100       // transmission. This covers the "stop bit" delay.
101       tx_state <= tx_countdown ? TX_DELAY_RESTART : TX_IDLE;
102     end
103   endcase
104 end
105
106 endmodule
```

Listing B.5: Verilog UART module

B.2.5 Data receiver script

```
#!/usr/bin/env python3
#-------------------------------------------------------------------------------
#   Author:     Magnus Overbo
#   Date:       20190324
#   Version:    1
#   Description:Opens RS232 communication to read 32b values sent from an FPGA.
#               It parses the data and performs synchronisation.  When the data
#               repeats itself, it will automatically stop receiving data and
#               print its data to file and screen.
#-------------------------------------------------------------------------------


#-------------------------------------------------------------------------------
#   Library import
#-------------------------------------------------------------------------------
import serial           #UART / RS232 library
import sys              #System
import datetime
from Crypto.Util.number import bytes_to_long   #Convert bytestring to long


#-------------------------------------------------------------------------------
#   Global variables
#-------------------------------------------------------------------------------
JSONDATA    = {}
i           = 0
POS         = None
STT         = None
ERR         = None
RUN         = True
DELAY       = False
LFSRSIZE    = int(sys.argv[1])
M           = int(sys.argv[2])
K           = int(sys.argv[3])
C           = int(sys.argv[4])
DSTART      = None
DEND        = None
LINIT       = -1
                    #Open COM port for UART communication at 115200 baud rate
serialConnection = serial.Serial("/dev/ttyUSB1", 115200, timeout=1/1000)


#-------------------------------------------------------------------------------
#   Helper function for printing binary representation of data
#-------------------------------------------------------------------------------
def binp( a, l ):
  A  = "{:b}".format( a )
  L  = l - len(A)
  return "{}{}".format("0"*L, A)


#-------------------------------------------------------------------------------
#   Main loop
#-------------------------------------------------------------------------------
while RUN:
  DATA = serialConnection.read(1)              #Read 1 byte
```

```
57    if len(DATA) == 0 or len(DATA) > 2 or DELAY: #Data synch (Skip one byte)
58      DELAY    = False
59      continue
60
61    #Load 32b data to generate long data
62    if i < 4:
63      POS = DATA if i == 0 else DATA + POS
64    elif i < 8:
65      STT = DATA if i == 4 else DATA + STT
66    elif i < 12:
67      ERR = DATA if i == 8 else DATA + ERR
68
69    if i == 11:                                  #Convert and store data
70      STT = bytes_to_long(STT)
71      POS = bytes_to_long(POS)
72      ERR = bytes_to_long(ERR)
73
74      if ERR > K:                                #if ERR > K, means byte mismatch
75        JSONDATA  = {}                           #  easiest to remove and redo
76        DELAY     = True                         #  Force delay to resynch
77        i         = 0
78        DSTART    = None
79        DEND      = None
80        sys.stdout.write("Error: {}\t{}\t{}\n".format( binp(STT,10), POS, ERR) )
81        sys.stdout.write("Resetting collected data, and skipping: 1B\n\n")
82        continue
83
84      if LINIT != STT:                           #New LFSR state logged
85        sys.stdout.write("STATE: {:>10}\t{}\n".format(STT, binp(STT,LFSRSIZE)))
86        LINIT     = STT
87
88      if DSTART == None:
89        DSTART = datetime.datetime.now()
90
91      if STT in JSONDATA.keys():                 #Add data to array
92        if POS in JSONDATA[STT].keys():
93          DEND    = datetime.datetime.now()
94          RUN     = False                        #End
95        else:
96          JSONDATA[STT].update( { POS:ERR } )    #new position
97      else:
98        JSONDATA.update( { STT : {POS:ERR} } )   #Entire new
99    i = (i + 1) % 12                             #Mod counter
100 serialConnection.close()                        #Close connection to UART RS232
101
102
103 #-------------------------------------------------------------------------------
104 #   Output data data to file and console
105 #-------------------------------------------------------------------------------
106 TDELTA = DEND-DSTART
107 PSTR="fpga_L{}M{}K{}C{}_{}.log"
108 fh = open(PSTR.format(LFSRSIZE, M, K, C, DSTART.strftime('%Y%m%d%H%M%S')), "w")
109 for key in sorted(JSONDATA.keys()):
110   sys.stdout.write( "{}\t{}\t".format( key, binp(key,LFSRSIZE) ) )
111
112   sys.stdout.write("\nMATCH_COUNT: {}\n".format( len(JSONDATA[key].keys()) ) )
113   fh.write("\nMATCH_COUNT: {}\n".format( len(JSONDATA[key].keys()) ) )
114
```

```
115     fh.write("INITSTATE {}\t{}\nMATCH\t".format( key, binp(key,LFSRSIZE) ) )
116     for skey in sorted(JSONDATA[key].keys()):
117       sys.stdout.write("{} ".format(skey))
118       fh.write( "{}:{} ".format(skey, JSONDATA[key][skey]))
119     sys.stdout.write( "\n\n" )
120     fh.write( "\n\n" )
121
122   fh.write("START:\t{}\nEND:\t{}\nTOT:\t{}\n".format(
123           DSTART.strftime('%Y %m %d %H:%M:%S.%f'),
124           DEND.strftime('%Y %m %d %H:%M:%S.%f'),
125           TDELTA)
126       )
127   sys.stdout.write("START:\t{}\nEND:\t{}\nTOT:\t{}\n".format(
128           DSTART.strftime('%Y %m %d %H:%M:%S.%f'),
129           DEND.strftime('%Y %m %d %H:%M:%S.%f'),
130           TDELTA)
131       )
132
133
134   fh.close()
```

Listing B.6: Python data logging script

Magnus Øverbø

Cryptanalysis of IC LFSR: ARBP search on FPGA

NTNU
Norwegian University of
Science and Technology