TECHNICAL UNIVERSITY OF CRETE

DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERS

Diploma Thesis

Study, analysis and implementation of gene alignment algorithm on a platform based on reconfigurable hardware

Orestis Xeni

Examining Board

Professor Dollas Apostolos (Supervisor)

Professor Pneumatikatos Dionysios

Associate Professor Papaeustathiou Ioannis

Chania, 2013

Acknowledgements

Firstly I would like to thank my family for sponsoring me and always being there when I needed them. Then I would like to thank my supervisor Professor Dollas Apostolos for supervising my diploma thesis and giving me the opportunity to work with him. Also I would like to thank Dr. Euripides Sotiriades for his guidance and his support through the whole process of the elaboration of my diploma thesis. As well, I would like to thank Mr. Gregory Chrysos for his technical support and the valuable time that he spent with me. I would like to thank the whole community of the MHL for their support and finally my friends for their psychological support.

Abstract

Computational Biology is one of the evolutionary scientific areas that the Electronics and Computer Engineers study. The Bioinformatics results can be used in biology, medicine and pharmaceutics and they can lead into new medicines and therapy methods. The Bioinformatics area consists of really high compute intensive and resource demanding problems.

The Bowtie algorithm is a sequence alignment algorithm that was introduced in 2009. This algorithm uses a different methodology and techniques from the other sequent alignment algorithms. In more details, the Bowtie algorithm uses the Burrows & Wheeler compression method and the FM – Index technique, which is a pattern matching method for very fast search for similar patterns. The Bowtie algorithm takes as input an organism's genetic database and genetic sequences-"queries" with high or low similarity to the input database. The algorithm's result shows the similarity between each input query and the genetic database.

This thesis presents a reconfigurable hardware-based implementation of the Bowtie algorithm. The search process of the algorithm, which is the most compute intensive part, is implemented on a multi-FPGA platform. The final system offers one order of magnitude execution speedup vs. the official software, as far as the most time consuming part of the Bowtie algorithm.

Contents

Chapter 1

Introduction

Computational Biology, sometimes referred to as bioinformatics, is the science of using biological data to develop algorithms and relations among various biological systems. Prior to the advent computational biology, biologists were unable to have access to a large amount of data. Researchers were able to develop analytical methods for interpreting biological information, but were unable to share them quickly among colleagues **[36]**.

Bioinformatics began to develop in the early 1970s. It was considered the science of analyzing informatics processes of various biological systems. At this time, research in artificial intelligence was using network models of the human brain in order to generate new algorithms. This use of biological data to develop other fields, pushed biological researchers to revisit the idea of using computers to evaluate and compare large data sets. By 1982, information was being shared amongst researchers through the use of punch cards. The amount of data being shared began to grow exponentially by the end of the 1980s. This required the development of new computational methods in order to quickly analyze and interpret relevant information **[36]**.

Since the late 1990s, computational biology has become an important part of developing emerging technologies for the field of biology. The terms computational biology and evolutionary computation have a similar name, but are not to be confused. Unlike computational biology, evolutionary computation is not concerned with modeling and analyzing biological data. It instead creates algorithms based on the ideas of evolution across species. Sometimes is referred as genetic algorithms, the research of this field can be applied to computational biology. While evolutionary computation is not inherently a part of computational biology, Computational evolutionary biology is a subfield of it **[37] [38]**.

Computational biology has been used to help sequence the human genome, create accurate models of the human brain, and assist in modeling biological systems.

1.1 The Problem

Computational biology executes very complex algorithms and in addition with the huge sized files that process, it needs computational recourses. Even if you have the most hi-tech computer running your algorithms the time that you need to reach a result it may varies from a few minutes to a few days, even months.

This is one of the most important problems in computational biology that costs enough drawbacks. For scientists to have results as fast as they can, they have to spend a significant amount of money every year for purchasing computers with very good capabilities so they can execute their algorithms. Another drawback is that they select to execute their algorithms sacrificing accuracy over performance. Not having results soon enough is a big problem enough to stall researches along with that science evolution and this is where FPGA's come to solve efficiently the puzzle saving money and time.

1.2 Motivation

Bioinformatics is a scientific field that always evolves and is in desperate need of computing resources so FPGA's can make a direct impact helping scientists with their work.

FPGA's have proved their capabilities through research helping lots of scientists achieving their goals and saving a lot of time doing that. This is what encouraged us to deal with this nature of problem, selecting a relatively new algorithm, and making all the procedures needed so it can be run on a FPGA.

Bowtie was introduced in 2009 using, for the first time in Bioinformatics, Burrows & Wheeler Transformation and FM – Index. This methods work together producing very fast and accurate results and they are extremely efficient in bioinformatics due to how good they handle pattern matching.

1.3 Diploma Thesis Structure

The diploma thesis consists of six chapters.

The next chapter talks about bioinformatics terminologies, gene finding and gene aligning algorithms that we have studied and finally the full analysis of the selected algorithm.

In the third chapter, we will talk about the modeling of the algorithm, preparation of input data and how the software communicates with the hardware.

After, in the fourth chapter, we will discuss about the data flow and the architecture design of the hardware.

In the fifth chapter we will talk about the results, comparing them with the software and with other relative works and finally in the sixth chapter we will talk about conclusions and further work.

Chapter 2

Gene Prediction (Gene Finding), Sequence Alignment and algorithms that implement it.

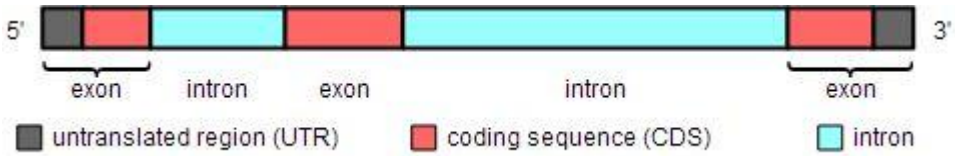
2.1 Gene Prediction or Gene Finding

In computational biology gene prediction or gene finding refers to the process of identifying the regions of genomic DNA that encode genes. This includes protein-coding genes as well as RNA genes, but may also include prediction of other functional elements such as regulatory regions. Gene finding is one of the first and most important steps in understanding the genome of a species once it has been sequenced. In its earliest days, "gene finding" was based on painstaking experimentation on living cells and organisms. Statistical analysis of the rates of homologous recombination of several different genes could determine their order on a certain chromosome, and information from many such experiments could be combined to create a genetic map specifying the rough location of known genes relative to each other. Today gene finding has been redefined as a largely computational problem due to resources restrictions.

Gene is a molecular unit of heredity of a living organism. It is a name given to some stretches of DNA and RNA that code for a polypeptide or for an RNA chain that has a function in the organism. Living beings depend on genes, as they specify all proteins and functional RNA chains. Genes hold the information to build and maintain an organism's cells and pass genetic traits to offspring, although some organelles (e.g. mitochondria) are self-replicating and are not coded for, by the organism's DNA **[41]**.

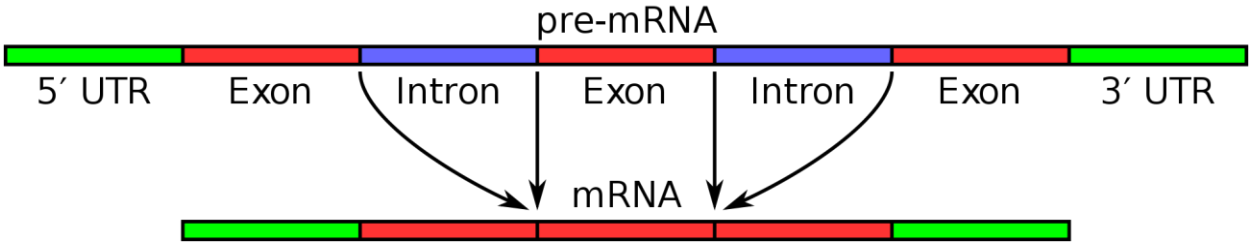
Exon is a sequence of DNA that is expressed (transcribed) into RNA and then often, but with many noteworthy exceptions, translated into protein. Adjacent exons may be separated by an intron, which is later removed from the RNA transcript via the splicing mechanism **[42]**.

Intron is any nucleotide sequence within a gene that is removed by RNA splicing while the final mature RNA product of a gene is being generated **[43]**.



Gene Structure **[39]**

Splicing is a modification of the nascent pre-mRNA taking place after or concurrently with its transcription, in which introns are removed and exons are joined. This is needed for the typical eukaryotic messenger RNA before it can be used to produce a correct protein through translation **[44]**.



mRNA before and after splicing **[40]**

2.2 Sequence alignment

Is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns.

```
GAATTCAG        GAATTCAG
|  |  ||  |     |  ||  |  |
GGA-TC-G        GCAT-C-G

GAATTC-A        GAATTC-A
|  |  ||  |     |  ||  |  |
GGA-TCGA        GCAT-CGA
```

2.3 Gene Finding or Gene Prediction Algorithms

Generally an algorithm is a step-by-step procedure for calculations. A Gene Finding or Gene Prediction algorithm uses methods (e.g. Interpolated Markov Models, Hidden Markov Models) so as to determine the beginning and end positions of genes in a genome.

2.3.1 Glimmer (Gene Locator and Interpolated Markov ModelER) uses interpolated Markov models (IMMs) to identify the coding regions and distinguish them from non coding DNA. Some of the advantages are low false positive rate, predicts many start sites correctly and high true positive rate **[1]**, **[2]**, **[3]**.

2.3.2 TWAIN is a new syntenic gene finder which employs a Generalized Pair Hidden Markov Model (GPHMM) to predict genes in two closely related eukaryotic genomes simultaneously. TWAIN performs very well on two related Aspergillus species, A.fumigatus and A.nidulans.Some disadvantages are that needs a better accuracy and wider area of organisms **[4]**.

2.3.3 GlimmerHMM is a gene finder based on a Generalized Hidden Markov Model (GHMM). It utilizes Interpolated Markov Models for the coding and noncoding models. Currently, GlimmerHMM's (GHMM) structure includes introns of each phase, intergenic regions, and four types of exons (initial, internal, final, and single). GlimmerHMM is very fast and memory efficient **[6]**, **[7]**.

2.3.4 GENSCAN is based on probabilistic model of gene structure similar to *Hidden Markov Models (HMMs).* GENSCAN uses a training set in order to estimate the *HMM parameters*, then the algorithm returns the exon structure using maximum likelihood approach standard to many HMM algorithms. It does not use similarity search to predict genes, it does not address alternative splicing and it could combine two exons from consecutive genes together **[5]**.

2.3.5 GeneZilla(former TigrScan) is a gene finder based on the Generalized Hidden Markov Model framework, similar to Genscan. The run time and memory requirements are linear in the sequence length. It utilizes Interpolated Markov Models (IMMs), Maximal Dependence Decomposition (MDD), and includes states for signal

peptides, branch points, TATA boxes, CAP sites, and will soon model CpG islands as well. GeneZilla is not so fast (compared to GlimmerHMM) but is extremely memory efficient **[6]**, **[7]**.

2.3.6 GeneSplicer is a fast, flexible system for detecting **splice sites** (the locations of the start codons, all the exons and introns and the stop codon for each gene) in the genomic DNA of various eukaryotes. The system has been trained and tested successfully on *Plasmodium falciparum* (malaria), *Arabidopsis thaliana*, human, *Drosophila*, and rice. GeneSplicer advantages are accuracy, memory efficiency and speed **[14]**.

2.3.7 ExAlt is a software program designed to predict alternatively spliced overlapping exons in genomic sequence. The program works in several ways depending on the available input. ExAlt can use information about existing gene structure as well as sequence conservation to improve the precision of its predictions. ExAlt can also make predictions when only a single genomic sequence is available. ExAlt has been extensively tested on *Drosophila melanogaster*, but can be adapted to run on other species.

The typical input to ExAlt is a known (or predicted) gene structure, which should be checked for alternative splicing. The core program takes as input a multiple sequence alignment and a phylogenetic tree and returns a GFF file containing the sequence coordinates of exon predictions. Wrapper scripts are provided to take a *Drosophila melanogaster* gene (using the CG identifier) and iterate through each exon, using blastn to find matches in closely releated species and muscle to generate multiple sequence alignments for input to ExAlt **[15]**.

2.3.8 JIGSAW is a program designed to use the output from gene finders, splice site prediction programs and sequence alignments to predict gene models. The program provides an automated way to take advantage of the many successful methods for computational gene prediction and can provide substantial improvements in accuracy over an individual gene prediction program. JIGSAW is available for all species **[7]**, **[16]**, **[17]**.

2.4 Gene Aligning & Short Reads Mapping Algorithms

Gene aligning & short reads mapping algorithms are algorithms that use various methods to align two or more short read gene sequences and come to conclusions for this genes, if they are related, or if they have evolve, or even if they have been mutated. Different algorithms exists using different methods, each one solving a different problem such as accuracy, performance, sequence size, memory foot print.

2.4.1 Maq can build assemblies by mapping shotgun short reads to a reference genome, using quality scores to derive genotype calls of the consensus sequence of a diploid genome, e.g., from a human sample. MAQ makes full use of mate-pair information and estimates the error probability of each read alignment using the Eland – like hashing technique. Error probabilities are also derived for the final genotype calls, using a Bayesian statistical model that incorporates the mapping qualities, error probabilities from the raw sequence quality scores, sampling of the two haplotypes, and an empirical model for correlated errors at a site. MAQ is accurate, efficient, versatile, and user-friendly **[8]**.

2.4.2 SOAP is designed to handle the huge amounts of short reads generated by parallel sequencing using the seed and hash look-up table algorithm. SOAP is compatible with numerous applications, including single-read or pair-end resequencing, small RNA discovery and mRNA tag sequence mapping. SOAP is a command-driven program, which supports multi-threaded parallel computing, and has a batch module for multiple query sets **[9]**, **[18]**.

2.4.3 RMAP can map reads having a wide range of lengths and allows base-call quality scores to determine which positions in each read are more important when mapping using the filtration method, a pattern matching method **[10]**.

2.4.4 SHRiMP algorithm draws upon three recent developments in the field of sequence alignment: q-gram filter approaches, spaced seeds and specialized vector computing hardware to speed up the Smith-Waterman Algorithm to rapidly find the likely locations for the reads on the genome. Once these locations are identified, Smith-Waterman based algorithm is conducted thoroughly to rigorously evaluate the alignments **[12]**, **[25]**, **[26]**, **[27]**, **[28]**, **[29]**.

2.5 Bowtie Algorithm, Burrows & Wheeler Transformation and FM Index

Bowtie is an algorithm that can align sequences of characters of genes with specified methods and determine if they are related, and if they are related if an evolution or a mutation exist **[13]**. It is used specifically in the section of computational biology from research and science centers. It is a new algorithm, released in 2009, ultra fast, very low memory foot print(approximately 1.3 GB) and has over 1200 citations. The pioneering of bowtie algorithm is the use for the first time in bioinformatics of the Burrows and Wheeler transformation (BWT) which was presented in 1994 by M. Burrows and D.J. Wheeler, **[19],** two mathematicians as a data compression method and FM Index an exact pattern matching method **[30]**, **[31]**, **[32]**. The FM Index is extremely fast in searching sequence patterns in very large text files and is already been used in other scientific fields and sections. One example is that it is used in LUT's of node routers in enormous networks.

Bowtie algorithm is divided into two sections. The first section is about building the database that you want to match your query pattern and the second one is doing the search **[20]**, **[22]**, **[23]**, **[24]**.

To build the database bowtie algorithm takes a genome sequence and uses the Burrows & Wheeler Transformation (BWT) to extract the transformed sequence. Given a text Q we denote by BWT(Q) its transform. The BWT of a string, or in our case sequence, is generated in five steps:

1. Terminate the text Q with a unique character: "$".
2. Generate all rotations of the text.
3. Sort all the rotations.
4. Extract the last characters of all the entries of the sorted list.
5. Join the characters in the same order they appeared in the sorted list.

The newly generated text is the BWT(Q).

For example lets set

Q= GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

After the first step Q will become

Q= GAACGATACCCACCCAACTATCGCCATTCCAGCAT**$.**

Then by fully rotating the sequence, executing the second step, our result is shown in Table 2.1 below.

Index	Rotated Sequence	Details
0	GAACGATACCCACCCAACTATCGCCATTCCAGCAT**$**	Initial Sequence
1	AACGATACCCACCCAACTATCGCCATTCCAGCAT**$**G	
2	ACGATACCCACCCAACTATCGCCATTCCAGCAT**$**GA	
3	CGATACCCACCCAACTATCGCCATTCCAGCAT**$**GAA	
4	GATACCCACCCAACTATCGCCATTCCAGCAT**$**GAAC	
5	ATACCCACCCAACTATCGCCATTCCAGCAT**$**GAACG	
6	TACCCACCCAACTATCGCCATTCCAGCAT**$**GAACGA	
7	ACCCACCCAACTATCGCCATTCCAGCAT**$**GAACGAT	
8	CCCACCCAACTATCGCCATTCCAGCAT**$**GAACGATA	
9	CCACCCAACTATCGCCATTCCAGCAT**$**GAACGATAC	
10	CACCCAACTATCGCCATTCCAGCAT**$**GAACGATACC	
11	ACCCAACTATCGCCATTCCAGCAT**$**GAACGATACCC	
12	CCCAACTATCGCCATTCCAGCAT**$**GAACGATACCCA	
13	CCAACTATCGCCATTCCAGCAT**$**GAACGATACCCAC	
14	CAACTATCGCCATTCCAGCAT**$**GAACGATACCCACC	
15	AACTATCGCCATTCCAGCAT**$**GAACGATACCCACCC	
16	ACTATCGCCATTCCAGCAT**$**GAACGATACCCACCCA	
17	CTATCGCCATTCCAGCAT**$**GAACGATACCCACCCAA	
18	TATCGCCATTCCAGCAT**$**GAACGATACCCACCCAAC	
19	ATCGCCATTCCAGCAT**$**GAACGATACCCACCCAACT	
20	TCGCCATTCCAGCAT**$**GAACGATACCCACCCAACTA	
21	CGCCATTCCAGCAT**$**GAACGATACCCACCCAACTAT	
22	GCCATTCCAGCAT**$**GAACGATACCCACCCAACTATC	
23	CCATTCCAGCAT**$**GAACGATACCCACCCAACTATCG	
24	CATTCCAGCAT**$**GAACGATACCCACCCAACTATCGC	
25	ATTCCAGCAT**$**GAACGATACCCACCCAACTATCGCC	
26	TTCCAGCAT**$**GAACGATACCCACCCAACTATCGCCA	
27	TCCAGCAT**$**GAACGATACCCACCCAACTATCGCCAT	
28	CCAGCAT**$**GAACGATACCCACCCAACTATCGCCATT	
29	CAGCAT**$**GAACGATACCCACCCAACTATCGCCATTC	
30	AGCAT**$**GAACGATACCCACCCAACTATCGCCATTCC	
31	GCAT**$**GAACGATACCCACCCAACTATCGCCATTCCA	
32	CAT**$**GAACGATACCCACCCAACTATCGCCATTCCAG	
33	AT**$**GAACGATACCCACCCAACTATCGCCATTCCAGC	
34	T**$**GAACGATACCCACCCAACTATCGCCATTCCAGCA	
35	**$**GAACGATACCCACCCAACTATCGCCATTCCAGCAT	

Table 2.1: Example of rotating a sequence in step 2 of Burrows & Wheeler Transformation.

Then, executing the third step of the transformation, sorting the rotated sequences lexicographically, our table changes as we can see in Table 2.2.

Index	Sequence
0	**$**GAACGATACCCACCCAACTATCGCCATTCCAGCAT
1	AACGATACCCACCCAACTATCGCCATTCCAGCAT**$**G
2	AACTATCGCCATTCCAGCAT**$**GAACGATACCCACCC
3	ACCCAACTATCGCCATTCCAGCAT**$**GAACGATACCC
4	ACCCACCCAACTATCGCCATTCCAGCAT**$**GAACGAT
5	ACGATACCCACCCAACTATCGCCATTCCAGCAT**$**GA
6	ACTATCGCCATTCCAGCAT**$**GAACGATACCCACCCA
7	AGCAT**$**GAACGATACCCACCCAACTATCGCCATTCC
8	AT**$**GAACGATACCCACCCAACTATCGCCATTCCAGC
9	ATACCCACCCAACTATCGCCATTCCAGCAT**$**GAACG
10	ATCGCCATTCCAGCAT**$**GAACGATACCCACCCAACT
11	ATTCCAGCAT**$**GAACGATACCCACCCAACTATCGCC
12	CAACTATCGCCATTCCAGCAT**$**GAACGATACCCACC
13	CACCCAACTATCGCCATTCCAGCAT**$**GAACGATACC
14	CAGCAT**$**GAACGATACCCACCCAACTATCGCCATTC
15	CAT**$**GAACGATACCCACCCAACTATCGCCATTCCAG
16	CATTCCAGCAT**$**GAACGATACCCACCCAACTATCGC
17	CCAACTATCGCCATTCCAGCAT**$**GAACGATACCCAC
18	CCACCCAACTATCGCCATTCCAGCAT**$**GAACGATAC
19	CCAGCAT**$**GAACGATACCCACCCAACTATCGCCATT
20	CCATTCCAGCAT**$**GAACGATACCCACCCAACTATCG
21	CCCAACTATCGCCATTCCAGCAT**$**GAACGATACCCA
22	CCCACCCAACTATCGCCATTCCAGCAT**$**GAACGATA
23	CGATACCCACCCAACTATCGCCATTCCAGCAT**$**GAA
24	CGCCATTCCAGCAT**$**GAACGATACCCACCCAACTAT
25	CTATCGCCATTCCAGCAT**$**GAACGATACCCACCCAA
26	GAACGATACCCACCCAACTATCGCCATTCCAGCAT**$**
27	GATACCCACCCAACTATCGCCATTCCAGCAT**$**GAAC
28	GCAT**$**GAACGATACCCACCCAACTATCGCCATTCCA
29	GCCATTCCAGCAT**$**GAACGATACCCACCCAACTATC
30	T**$**GAACGATACCCACCCAACTATCGCCATTCCAGCA
31	TACCCACCCAACTATCGCCATTCCAGCAT**$**GAACGA
32	TATCGCCATTCCAGCAT**$**GAACGATACCCACCCAAC
33	TCCAGCAT**$**GAACGATACCCACCCAACTATCGCCAT
34	TCGCCATTCCAGCAT**$**GAACGATACCCACCCAACTA
35	TTCCAGCAT**$**GAACGATACCCACCCAACTATCGCCA

Table 2.2: Example of sorting a sequence in step 3 of Burrows & Wheeler Transformation.

In step 4 we have to extract the last characters of all the entries of the sorted list, hence our table becomes as shown in Table 2.3.

Index	Sequence
0	**$**GAACGATACCCACCCAACTATCGCCATTCCAGCA - **T**
1	AACGATACCCACCCAACTATCGCCATTCCAGCAT**$** - **G**
2	AACTATCGCCATTCCAGCAT**$**GAACGATACCCACC - **C**
3	ACCCAACTATCGCCATTCCAGCAT**$**GAACGATACC - **C**
4	ACCCACCCAACTATCGCCATTCCAGCAT**$**GAACGA - **T**
5	ACGATACCCACCCAACTATCGCCATTCCAGCAT**$**G - **A**
6	ACTATCGCCATTCCAGCAT**$**GAACGATACCCACCC - **A**
7	AGCAT**$**GAACGATACCCACCCAACTATCGCCATTC - **C**
8	AT**$**GAACGATACCCACCCAACTATCGCCATTCCAG - **C**
9	ATACCCACCCAACTATCGCCATTCCAGCAT**$**GAAC - **G**
10	ATCGCCATTCCAGCAT**$**GAACGATACCCACCCAAC - **T**
11	ATTCCAGCAT**$**GAACGATACCCACCCAACTATCGC - **C**
12	CAACTATCGCCATTCCAGCAT**$**GAACGATACCCAC - **C**
13	CACCCAACTATCGCCATTCCAGCAT**$**GAACGATAC - **C**
14	CAGCAT**$**GAACGATACCCACCCAACTATCGCCATT - **C**
15	CAT**$**GAACGATACCCACCCAACTATCGCCATTCCA - **G**
16	CATTCCAGCAT**$**GAACGATACCCACCCAACTATCG - **C**
17	CCAACTATCGCCATTCCAGCAT**$**GAACGATACCCA - **C**
18	CCACCCAACTATCGCCATTCCAGCAT**$**GAACGATA - **C**
19	CCAGCAT**$**GAACGATACCCACCCAACTATCGCCAT - **T**
20	CCATTCCAGCAT**$**GAACGATACCCACCCAACTATC - **G**
21	CCCAACTATCGCCATTCCAGCAT**$**GAACGATACCC - **A**
22	CCCACCCAACTATCGCCATTCCAGCAT**$**GAACGAT - **A**
23	CGATACCCACCCAACTATCGCCATTCCAGCAT**$**GA - **A**
24	CGCCATTCCAGCAT**$**GAACGATACCCACCCAACTA - **T**
25	CTATCGCCATTCCAGCAT**$**GAACGATACCCACCCA - **A**
26	GAACGATACCCACCCAACTATCGCCATTCCAGCAT - **$**
27	GATACCCACCCAACTATCGCCATTCCAGCAT**$**GAA - **C**
28	GCAT**$**GAACGATACCCACCCAACTATCGCCATTCC - **A**
29	GCCATTCCAGCAT**$**GAACGATACCCACCCAACTAT - **C**
30	T**$**GAACGATACCCACCCAACTATCGCCATTCCAGC - **A**
31	TACCCACCCAACTATCGCCATTCCAGCAT**$**GAACG - **A**
32	TATCGCCATTCCAGCAT**$**GAACGATACCCACCCAA - **C**
33	TCCAGCAT**$**GAACGATACCCACCCAACTATCGCCA - **T**
34	TCGCCATTCCAGCAT**$**GAACGATACCCACCCAACT - **A**
35	TTCCAGCAT**$**GAACGATACCCACCCAACTATCGCC - **A**

Table 2.3: Example of step 4 of the Burrows & Wheeler Transformation.

Finally by joining the characters in the same order they appeared in the sorted list we generate the Burrows & Wheeler Transformation of sequence Q.

BWT (Q) = TGCCTAACCGTCCCCGCCCTGAAATA$CACAACTAA

After finishing with the transformation we use it to generate the sorted Burrows & Wheeler Transformation (SBWT) and the tables C and I which combined it together with the method of FM – Index help us to identify patterns in text.

Sorting BWT (Q) give us

SBWT (Q) = $AAAAAAAAAAACCCCCCCCCCCCCCCGGGGTTTTTT

I-table stores the first occurrence of each character on the sorted BWT(Q) as shown in Table 2.4.

Index	Character	Index	Character	Index	Character
0	$	**12**	C	**24**	C
1	A	**13**	C	**25**	C
2	A	**14**	C	**26**	G
3	A	**15**	C	**27**	G
4	A	**16**	C	**28**	G
5	A	**17**	C	**29**	G
6	A	**18**	C	**30**	T
7	A	**19**	C	**31**	T
8	A	**20**	C	**32**	T
9	A	**21**	C	**33**	T
10	A	**22**	C	**34**	T
11	A	**23**	C	**35**	T

Table 2.4: SBWT (Q)

As a result I – Table is shown in Table 2.5.

A	C	G	T
1	12	26	30

Table 2.5: I – Table

The C-table stores the count of each character on a previous location as shown in Table 2.6.

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

Table 2.6: C - Table

The FM-index is a pattern searching technique that operates on the BWT. The FM-index consists of two pointers: top and bottom **[21]**. The indices between the top and bottom pointers are all the suffix locations where a pattern occurs on the text. Top

points to an index of the suffix array element where a specific pattern is first located. The bottom pointer limits where the pattern can be last found. If bottom points to an index that is less than or equal to an index pointed by top, then the pattern does not occur on the text.

Pattern searching using the FM-index starts with initializing the top and bottom pointers to the first and last indices of the C - table respectively. To search for a pattern, we process one character at a time, beginning with the last character of the pattern. The top and bottom pointers move to different suffix array indices according to the current character processed and the current index where the top and bottom pointers are indexing. To compute the new location of the pointers, we follow Equation 1 for the top and bottom pointer respectively.

For example let's search the pattern ACCCACCC on the string Q using the FM – Index. Equation 1:

$Top_{new} = C – Table[n, Top_{current}] + I – Table[n]$

$Bottom_{new} = C – Table[n, Bottom_{current}] + I – Table[n]$

$Top_{cur} = 0$, $Bot_{cur} = 36$

1st Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 0 + 12 = 12$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 14 + 12 = 26$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3

17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

That means 26 – 12 = 14, we can find C fourteen times in the current sequence.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

$Top_{cur} = 12$, $Bot_{cur} = 26$

2nd Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 5 + 12 = 17$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 11 + 12 = 23$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2

11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

23 - 17 = 6, CC appears six times in the sequence.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

$Top_{cur} = 17$, $Bot_{cur} = 23$

3d Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 9 + 12 = 21$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 11 + 12 = 23$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1

5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

23 - 21 = 2, CCC appears only twice in the text.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

$Top_{cur} = 21$, $Bot_{cur} = 23$

4th Iteration: n=A

$Top_{new} = C_A(Top_{cur}) + I(A) = 2 + 1 = 3$

$Bot_{new} = C_A(Bot_{cur}) + I(A) = 4 + 1 = 5$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

5 - 3 = 2, ACCC appears twice.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

$Top_{cur} = 3$, $Bot_{cur} = 5$

5th Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 1 + 12 = 13$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 2 + 12 = 14$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5

32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

14 -13 = 1, CACCC appears only once.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

Top_{cur} = 13, Bot_{cur} = 14

6th Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 6 + 12 = 18$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 7 + 12 = 19$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4
21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5

27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

19 -18 = 1, CCACCC appears only once.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

$Top_{cur} = 18$, $Bot_{cur} = 19$

7th Iteration: n=C

$Top_{new} = C_C(Top_{cur}) + I(C) = 10 + 12 = 22$

$Bot_{new} = C_C(Bot_{cur}) + I(C) = 11 + 12 = 23$

Index	BWT(Q)	A	C	G	T
0	T	0	0	0	0
1	G	0	0	0	1
2	C	0	0	1	1
3	C	0	1	1	1
4	T	0	2	1	1
5	A	0	2	1	2
6	A	1	2	1	2
7	C	2	2	1	2
8	C	2	3	1	2
9	G	2	4	1	2
10	T	2	4	2	2
11	C	2	4	2	3
12	C	2	5	2	3
13	C	2	6	2	3
14	C	2	7	2	3
15	G	2	8	2	3
16	C	2	8	3	3
17	C	2	9	3	3
18	C	2	10	3	3
19	T	2	11	3	3
20	G	2	11	3	4

21	A	2	11	4	4
22	A	3	11	4	4
23	A	4	11	4	4
24	T	5	11	4	4
25	A	5	11	4	5
26	$	6	11	4	5
27	C	6	11	4	5
28	A	6	12	4	5
29	C	7	12	4	5
30	A	7	13	4	5
31	A	8	13	4	5
32	C	9	13	4	5
33	T	9	14	4	5
34	A	9	14	4	6
35	A	10	14	4	6
36	Total	11	14	4	6

23 -22 = 1, CCCACCC appears only once and we have successfully found the pattern we were searching.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

Now let's search in the same text but with a different text, so our next example will be with text Q and pattern ACCGT.

$Top_{cur} = 0$, $Bot_{cur} = 36$

1st Iteration: n=T

$Top_{new} = C_T(Top_{cur}) + I(T) = 0 + 30 = 30$

$Bot_{new} = C_T(Bot_{cur}) + I(T) = 6 + 30 = 36$

36-30=6, T appears six times in the text.

GAACGATACCCACCCAACTATCGCCATTCCAGCAT.

2nd Iteration: n=G

$Top_{new} = C_G(Top_{cur}) + I(G) = 4 + 26 = 34$

$Bot_{new} = C_G(Bot_{cur}) + I(G) = 4 + 26 = 34$

34-34=0, As we can see GT doesn't appear in our text so we stop searching.

2.6 Related Work

2.6.1 State of the Art on Hardware Acceleration on bioinformatics software tools

The reconfigurable hardware community used DNA sequence matching and database search as one of the first problems to show how computationally intensive problems can be solved using FPGAs. The venerable Splash 2 platform was used during the early 1990s by Hoang et. al. **[49][50]**. Later Guccione et. al. **[51]** used Jbits technology and both the Virginia Tech Configurable Computing Laboratory **[52]** and Nanyang Technological University **[53]** used run time reconfiguration for the same problem. Then, the **[54]** Technical University of Crete introduced their Hardware Acceleration design for BLAST Algorithm and later on in **[55]** again Technical University of Crete introduced their Hardware Acceleration design for GlimmerHMM. Other works on this area from Technical University of Crete are shown in **[56]**, **[61]**, **[62]**, **[63]**, **[64]**, **[65]**, **[66]**. Later, an implementation of K – means algorithm for bioinformatics was introduced by University of Edinburgh in **[57]**, then hardware acceleration of GASSST by Nanyang Technological University **[59]**. Finally we have Smith and Waterman algorithm hardware acceleration by Y. Yamaguchi et al. **[60]** and the hardware acceleration of BWA – SW algorithm from University of Science and Technology of China **[58]**.

2.6.2. Previous Work on Bowtie

Designing the Bowtie software tool on hardware and executing it on FPGA first appeared in **[21]**. This work appeared first time, early in 2011 and implements the search function of the algorithm and executing it on a Xilinx Virtex 6 (XC6VLX760) FPGA with promising results achieving significant speed ups but with limited functionality of the algorithm. Later, improving their designs, the next attempt appeared in **[20]** in 2012. This time the platform was the Convey HC – 1 with Intel Xeon L540B with 2 dual cores running at 2.13 GHz as the host processor, 192GB of RAM and four Virtex 5 FPGAs (XC5VLX330) as coprocessor. In this work they have implemented more functionalities of the algorithm and with the help of the Convey HC – 1 they parallelized their design helping them to reach speed ups up to 70x.

Chapter 3

System Modeling

In this chapter we will discuss about profiling and analyzing the algorithm and also the steps we follow to obtain input data for the system, the processing of this data and bringing it to the right form in order to connect them to our architecture design and finally how do we connect the data that we have prepared to the system.

3.1 Profiling

In order to understand better the algorithm, we use a couple tools to profile Bowtie. The tools that we use are GNU gprof and Intel Vtune. After collecting the results we have noticed that for different datasets the compute intensive function changed and if it was the same, the execution time changed, so we decided that we will continue our work by implementing the search functionality of the algorithm and not a specific function. Below, in Tables 3.1, 3.2, 3.3, 3.4 we can see the results from the experiments that we have performed.

Test 1		Test 2	
Execution Time	**Function**	**Execution Time**	**Function**
76%	Backtrack()	91%	Backtrack()
16%	mapLFEx()	7%	initFromRow()

Table 3.1 Functions with significant execution time from Test 1 and Test 2.

Test 3		Test 4	
Execution Time	**Function**	**Execution Time**	**Function**
31%	Backtrack()	40%	Backtrack()
29%	countUpToEx()	20%	countBWside()
18%	countBWSide()	20%	countBWSideEx()
18%	countFWSide()	20%	countFWSide()

Table 3.2 Functions with significant execution time from Test 3 and Test 4.

Test 5		Test 6	
Execution Time	**Function**	**Execution Time**	**Function**
48%	Bowtie()	48%	Samples()
13%	setQuery()	27%	nextBlock()

Table 3.3 Functions with significant execution time from Test 5 and Test 6.

Test 7		Test 8	
Execution Time	**Function**	**Execution Time**	**Function**
45%	Samples()	41%	bucketSortSufDcU8()
30%	nextBlock()	17%	nextBlock()
		14%	Samples()

Table 3.4 Functions with significant execution time from Test 7 and Test 8.

3.2 Modeling the build process of Bowtie

3.2.1 Obtain Data

In order to find files that contain genome databases and short reads to map on the databases we go to the webpage of National Center for Biotechnology Information (NCBI). To find database files we go to the genome base **[33]**, and to find short reads files we change to the short reads base **[34]**. After we find and identify the files that we are going to use we download the files. The database files do not need any further processing but for the short read files we need to change the format from .sra to .fastq. To do that, we use the SRAtoolkit which is a tool offered from NCBI and is free for downloading and using **[35]**.

3.2.2 Processing Data

To be able to send data to our system it was necessary to model the algorithm's build function. We write six different scripts using MATLAB software, reading the .fasta and .fastq files that contains the database, finding its BWT, creating the tables C and I and finally writing the data to a .txt file **[A]**.

As we have mentioned before and continuing with our previous example in chapter 2.5, the first step is to read the database.

```
Reading Database...
Read Database succesfully!

Q =

GAACGATACCCACCCAACTATCGCCATTCCAGCAT
```

Picture 3.1 Reading database using Matlab snapshot

After we have read the database the next step is putting the $ character at the end of the sequence and generate all rotations of the text.

```
Starting...
Put Character $ at the end...
Rotating...

SuffixArray =

GAACGATACCCACCCAACTATCGCCATTCCAGCAT$
AACGATACCCACCCAACTATCGCCATTCCAGCAT$G
ACGATACCCACCCAACTATCGCCATTCCAGCAT$GA
CGATACCCACCCAACTATCGCCATTCCAGCAT$GAA
GATACCCACCCAACTATCGCCATTCCAGCAT$GAAC
ATACCCACCCAACTATCGCCATTCCAGCAT$GAACG
TACCCACCCAACTATCGCCATTCCAGCAT$GAACGA
ACCCACCCAACTATCGCCATTCCAGCAT$GAACGAT
CCCACCCAACTATCGCCATTCCAGCAT$GAACGATA
CCACCCAACTATCGCCATTCCAGCAT$GAACGATAC
CACCCAACTATCGCCATTCCAGCAT$GAACGATACC
ACCCAACTATCGCCATTCCAGCAT$GAACGATACCC
CCCAACTATCGCCATTCCAGCAT$GAACGATACCCA
CCAACTATCGCCATTCCAGCAT$GAACGATACCCAC
CAACTATCGCCATTCCAGCAT$GAACGATACCCACC
AACTATCGCCATTCCAGCAT$GAACGATACCCACCC
ACTATCGCCATTCCAGCAT$GAACGATACCCACCCA
CTATCGCCATTCCAGCAT$GAACGATACCCACCCAA
TATCGCCATTCCAGCAT$GAACGATACCCACCCAAC
ATCGCCATTCCAGCAT$GAACGATACCCACCCAACT
TCGCCATTCCAGCAT$GAACGATACCCACCCAACTA
CGCCATTCCAGCAT$GAACGATACCCACCCAACTAT
GCCATTCCAGCAT$GAACGATACCCACCCAACTATC
CCATTCCAGCAT$GAACGATACCCACCCAACTATCG
CATTCCAGCAT$GAACGATACCCACCCAACTATCGC
ATTCCAGCAT$GAACGATACCCACCCAACTATCGCC
TTCCAGCAT$GAACGATACCCACCCAACTATCGCCA
TCCAGCAT$GAACGATACCCACCCAACTATCGCCAT
CCAGCAT$GAACGATACCCACCCAACTATCGCCATT
CAGCAT$GAACGATACCCACCCAACTATCGCCATTC
AGCAT$GAACGATACCCACCCAACTATCGCCATTCC
GCAT$GAACGATACCCACCCAACTATCGCCATTCCA
CAT$GAACGATACCCACCCAACTATCGCCATTCCAG
AT$GAACGATACCCACCCAACTATCGCCATTCCAGC
T$GAACGATACCCACCCAACTATCGCCATTCCAGCA
$GAACGATACCCACCCAACTATCGCCATTCCAGCAT
```

Picture 3.2 Creating Suffix Array using Matlab snapshot.

The next step is to sort lexicographical the Suffix Array and extract the Burrow's and Wheeler's Transformation.

```
Sort lexicographicaly the rows of suffix array...
Exporting BWT...

BWT =

TGCCTAACCGTCCCCGCCCTGAAATA$CACAACTAA

BWT Created!
```

Picture 3.3 Extracting BWT using Matlab snapshot.

After we extract the Burrow's and Wheeler's Transformation, we move to the creating of tables SBWT, C and I.

```
Sorting...

SBWT =

$AAAAAAAAAAACCCCCCCCCCCCCCGGGGTTTTTT

Sorted!
```

Picture 3.4 Extracting SBWT using Matlab snapshot.

```
Creating Table I...

TableI =

     1    12    26    30

Table I Created!
```

Picture 3.5 Creating Table I using Matlab snapshot.

```
Creating Table C...

TableC =

     0     0     0     0
     0     0     0     1
     0     0     1     1
     0     1     1     1
     0     2     1     1
     0     2     1     2
     1     2     1     2
     2     2     1     2
     2     3     1     2
     2     4     1     2
     2     4     2     2
     2     4     2     3
     2     5     2     3
     2     6     2     3
     2     7     2     3
     2     8     2     3
     2     8     3     3
     2     9     3     3
     2    10     3     3
     2    11     3     3
     2    11     3     4
     2    11     4     4
     3    11     4     4
     4    11     4     4
     5    11     4     4
     5    11     4     5
     6    11     4     5
     6    11     4     5
     6    12     4     5
     7    12     4     5
     7    13     4     5
     8    13     4     5
     9    13     4     5
     9    14     4     5
     9    14     4     6
    10    14     4     6
    11    14     4     6

Table C Created!
```

Picture 3.6 Creating Table C using Matlab snapshot.

Finally, we write this data in text files in order to make the connection between software and hardware.

3.3 Connecting Input Data with Hardware

To connect our design with the input files we read the text files we created using C language and then, send them to the design **[B]**. Our design can accept files with the extension .FASTA and .FASTQ. For the database file in the first lines we write the minimum and maximum values of top and bottom pointers, and Table I's contexts. Then, we write the depth size of Table C and then each of its columns respectively. For the query file, the first line is filled with the amount of sequences containing the file and then the sequences are written respectively separating each sequence by writing a zero at the end, allowing us to send different length sequences in the same file.

Chapter 4

Implementation

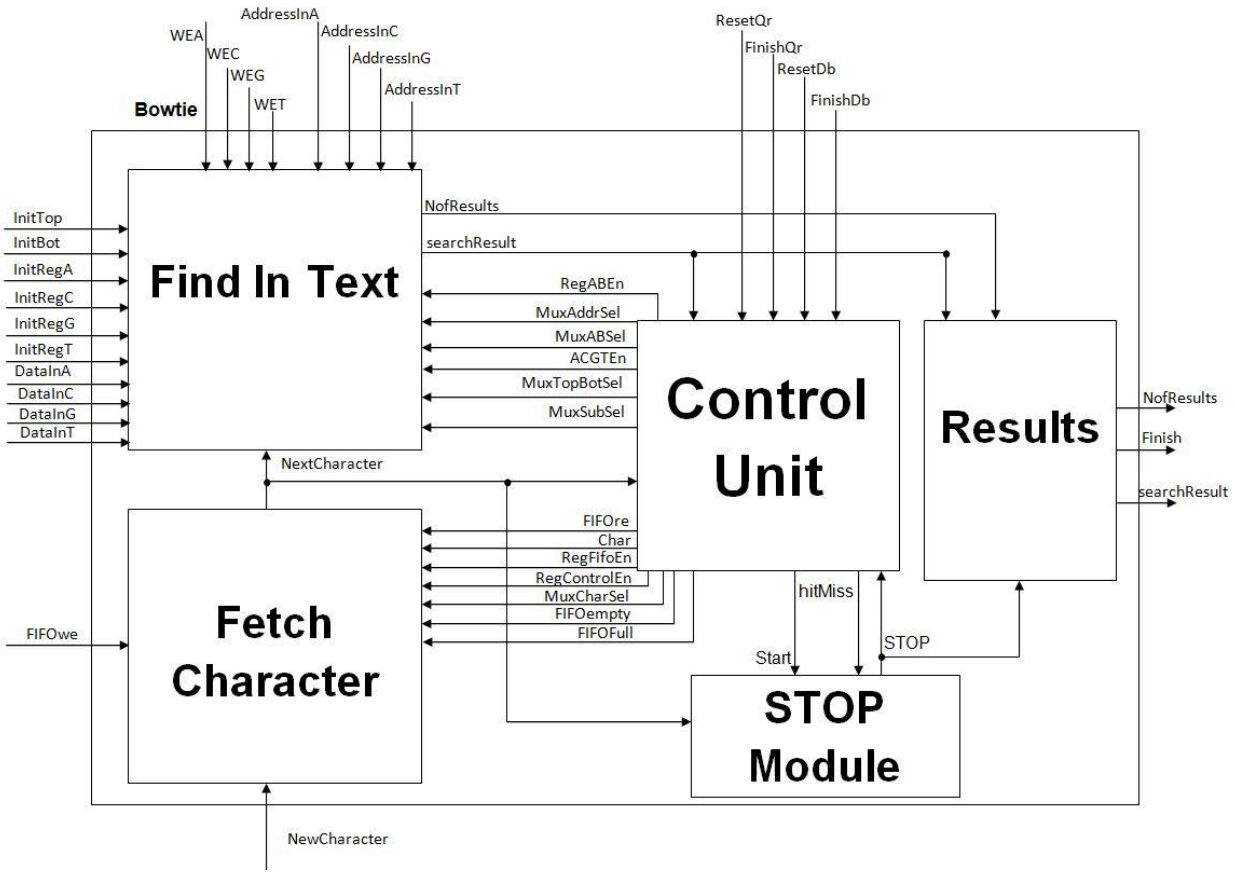
In this chapter we will discuss and analyze the architectural design, the basic structure is inspired by **[21]**, of the algorithm that we decided to implement. We will begin by explaining our first design which helped us confirm that it works and return correct output, determine the dataflow and identify how many of the FPGA's resources we are going to need. Our design it consists out of five basic components. The "Find In Text" component which is responsible for reporting if a character or sequence of characters exist or not in the database, the "Fetch Character" component which delivers a new character to be searched, the "Stop Module" component which decides when to stop searching, the "Results" which is responsible to deliver the output after the processing of a sequence is finished and finally the "Control Unit", the FSM, which is responsible for synchronizing and making the other four components working together.

4.1 Block Diagram

The design has totally twenty - five input signals and three output signals. Firstly, we have Clock and Reset which goes to every component. Secondly, we have the input that we import the query sequence into the design and then six inputs that we use initializing registers and signals in the design. Then, we have four which defines the address of each memory, another four for the input data and another four for the write enable of each memory. If we want to input a new query we use the ResetQr and after we finish writing, we switch on the FinishWrQr. Finally, ResetDb resets the memories and registers and prepares it for writing a new database and FinishWrDb indicates when the writing of the memories is finished. Moving on to the outputs, the first one if we have a hit or miss, the second one, how many hits we have if we have any and the third if the search is finished. The inputs and outputs are shown below fully detailed in Table 4.1 and the design in Picture 4.1.

Index	Description	Size	I/O
1	Clock	1 bit	Input
2	Reset	1 bit	Input
3	NewCharacter	3 bit	Input
4	InitTop	16 bit/17 bit	Input
5	InitBot	16 bit	Input
6	InitRegA	16 bit	Input
7	InitRegC	16 bit/17 bit	Input
8	InitRegG	16 bit/17 bit	Input
9	InitRegT	16 bit/17 bit	Input
10	AddressInA	16 bit/17 bit	Input
11	AddressInC	16 bit/17 bit	Input
12	AddressInG	16 bit/17 bit	Input
13	AddressInT	16 bit/17 bit	Input
14	DataInA	16 bit/17 bit	Input
15	DataInC	16 bit/17 bit	Input
16	DataInG	16 bit/17 bit	Input
17	DataInT	16 bit/17 bit	Input
18	WriteEnableA	1 bit	Input
19	WriteEnableC	1 bit	Input
20	WriteEnableG	1 bit	Input
21	WriteEnableT	1 bit	Input
22	FinishWrDb	1 bit	Input
23	FinishWrQr	1 bit	Input
24	ResetDb	1 bit	Input
25	ResetQr	1 bit	Input
26	FiFoWE	1 bit	Input
27	NofResults	16 bit/17 bit	Output
28	searchResult	1 bit	Output
29	Finish	1 bit	Output

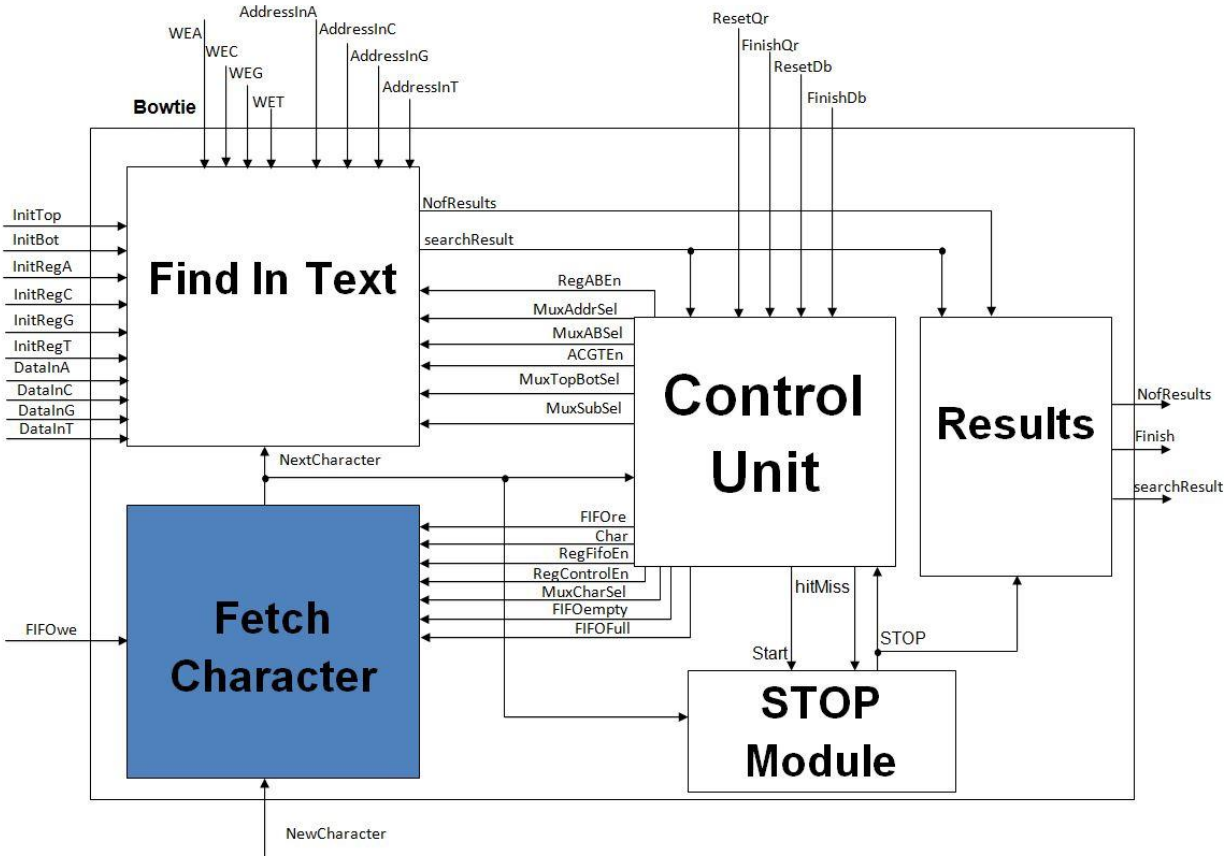
Table 4.1 Inputs and outputs of the design



Picture 4.1 Full Design Block Diagram

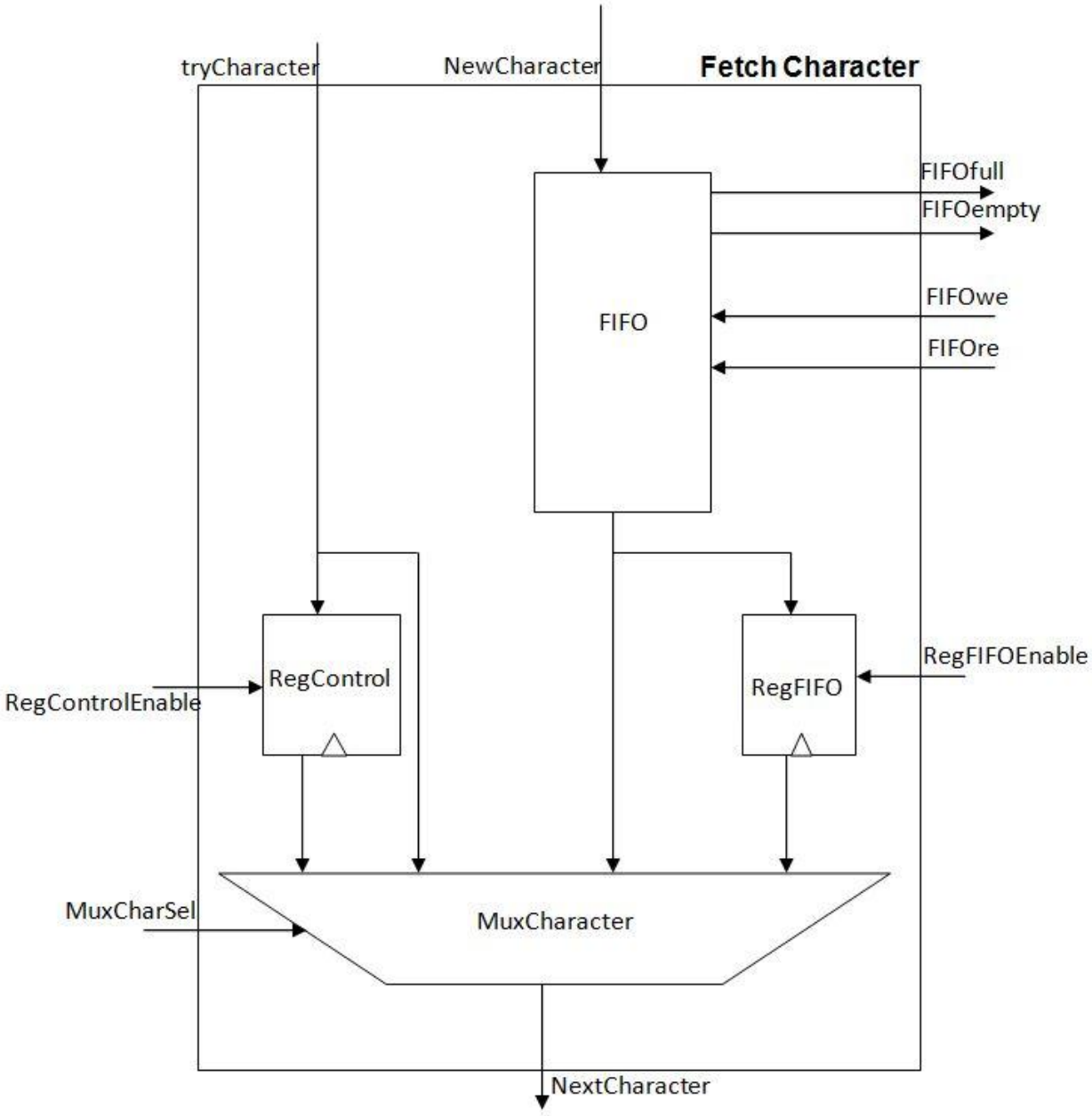
4.1.1 Fetch Character Component

Fetch Character component is the one which is marked with blue in Picture 4.2. Its basic operation is to deliver the next character to the other components.



Picture 4.2 Identifying Fetch Character component in the design

Fetch Character component it consists from a FIFO memory, two registers and a multiplexor as shown in Picture 4.3.



Picture 4.3 Block Diagram of Fetch Character Component.

Every new sequence that come it is saved in FIFO. FIFO has a size of 1024 places, which means that it can store up to 1024 characters which is the maximum size of sequence that it can be examined by the algorithm. RegFIFO is a register that holds the value that FIFO had in the previous cycle. RegControl is a register that holds the value that the character that comes from the Control Unit component had in the previous cycle. Finally the multiplexor MuxCharacter selects in of the four values available for output. If the ResetQr signal is switched on, then everything written in FIFO is erased.

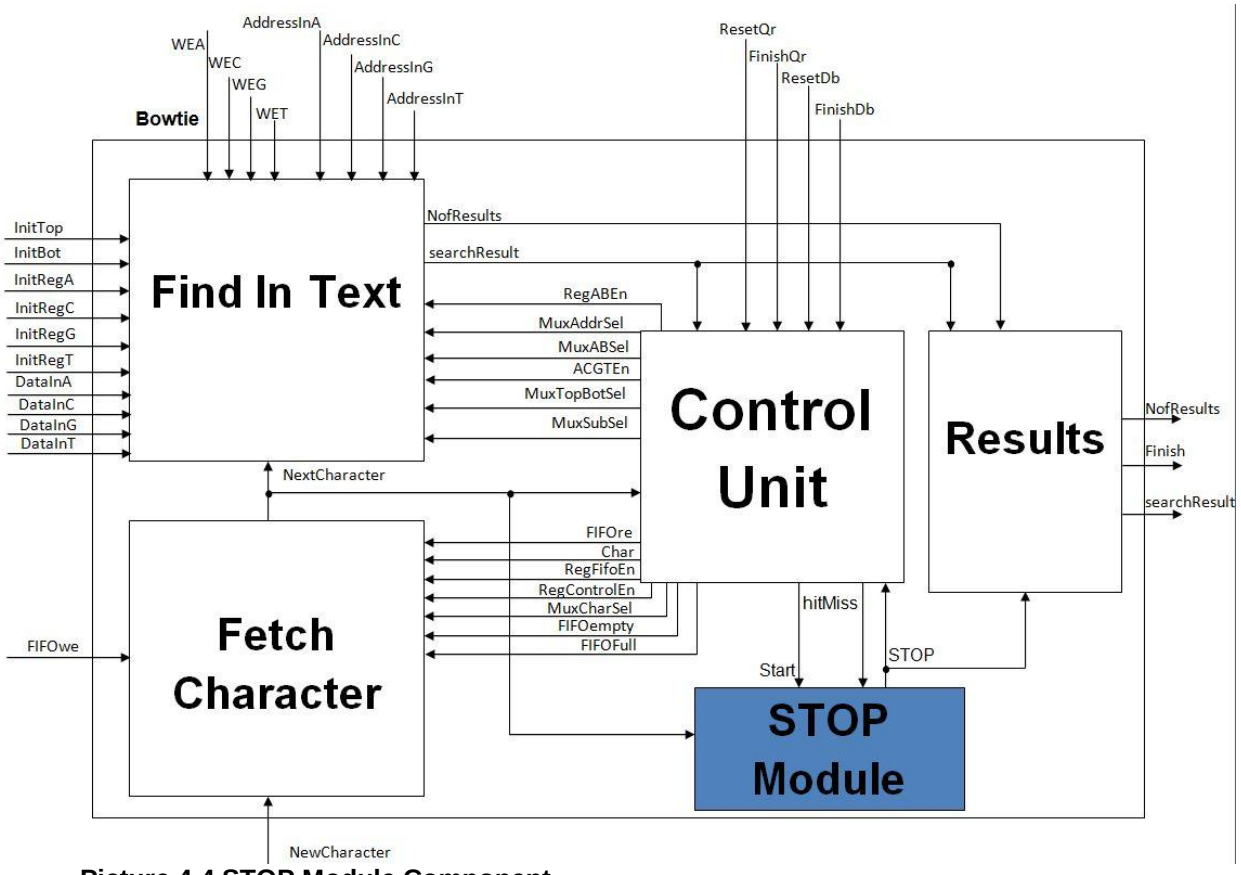
The inputs and outputs of Fetch Character component are shown in detailed in Table 4.2.

Index	Description	Size	I/O
1	NewCharacter	3 bit	Input
2	tryCharacter	3 bit	Input
3	RegControlEnable	1 bit	Input
4	RegFIFOEnable	1 bit	Input
5	MuxCharSel	2 bit	Input
6	FIFOwe	1 bit	Input
7	FIFOre	1 bit	Input
8	ResetQr	1 bit	Input
9	FIFOfull	1 bit	Output
10	FIFOempty	1 bit	Output
11	NextCharacter	3 bit	Output

Table 4.2 Inputs and Outputs of Fetch Character Component.

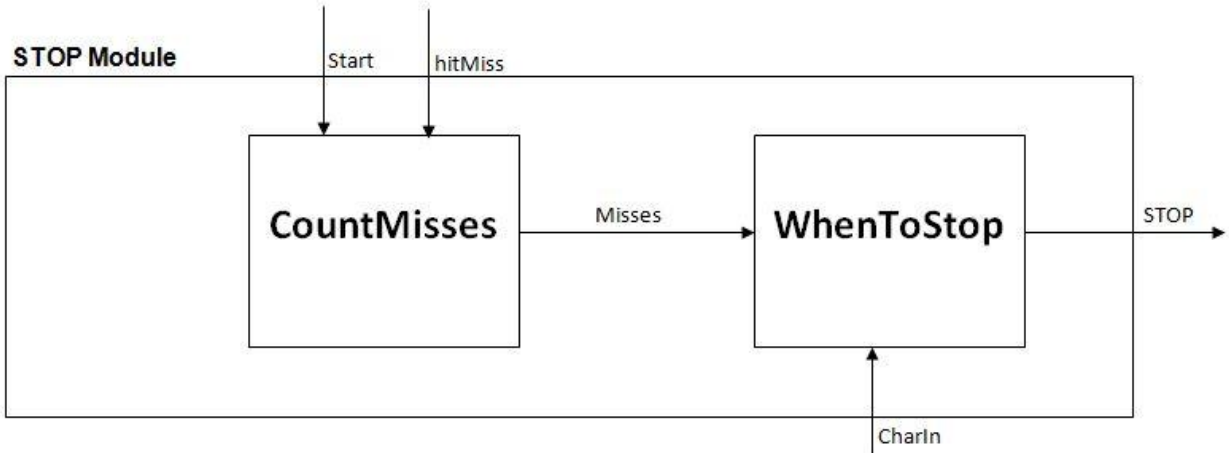
4.1.2 STOP Module

STOP Module component is shown below in Picture 4.4.



Picture 4.4 STOP Module Component.

STOP Module component is the component that is responsible to inform the control unit if it is necessary to stop, after we have completed three misses. Block diagram of the component is shown in Picture 4.5.



Picture 4.5 STOP Module component block diagram.

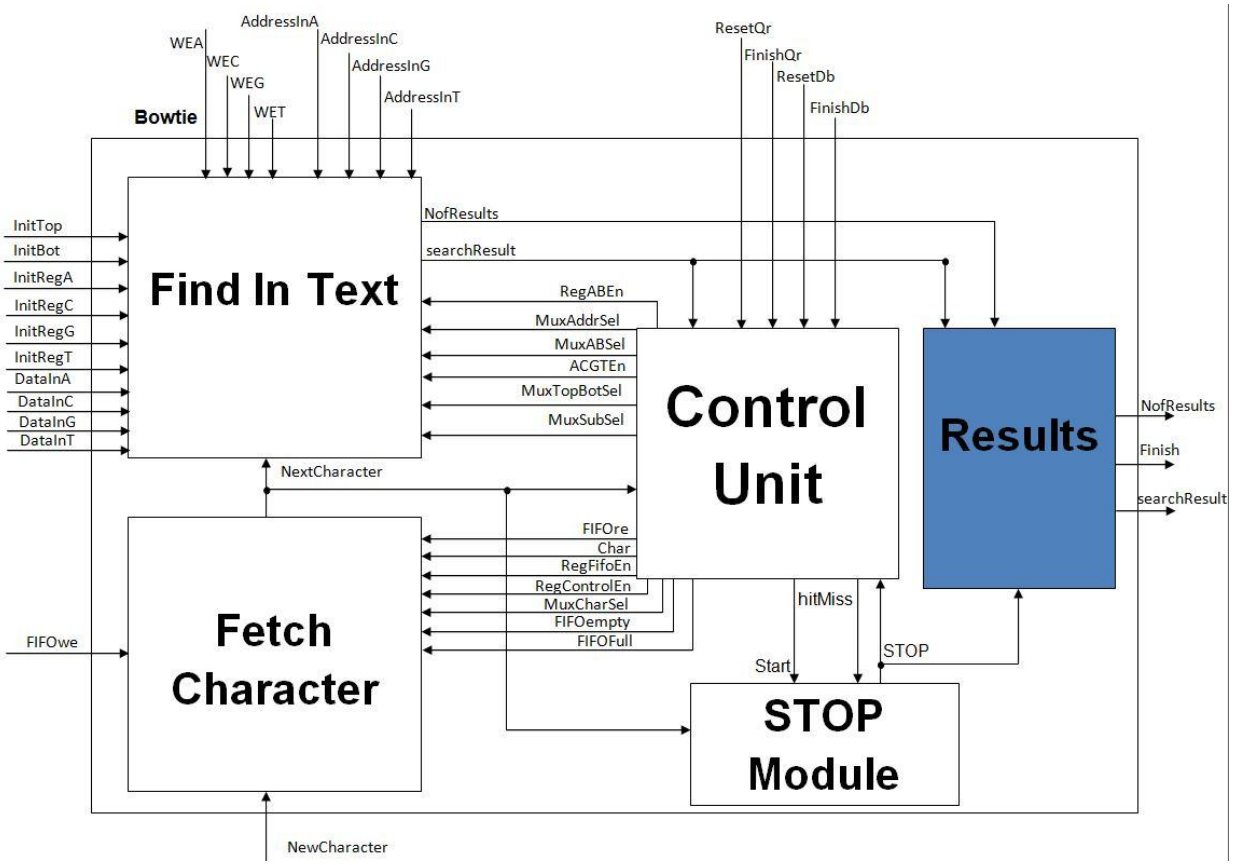
STOP Module component is consisted from two other components, CountMisses and WhenToStop. CountMisses holds the sum of misses that have arrived. If the amount of misses reach three before the sequence end, then WhenToStop component triggers the STOP signal. STOP signal is also triggered if CharIn is zero meaning that the sequence is finished. In Table 4.3 it is detailed shown inputs and outputs of the component.

Index	Description	Size	I/O
1	Start	1 bit	Input
2	hitMiss	1 bit	Input
3	CharIn	3 bit	Input
4	STOP	1 bit	Output

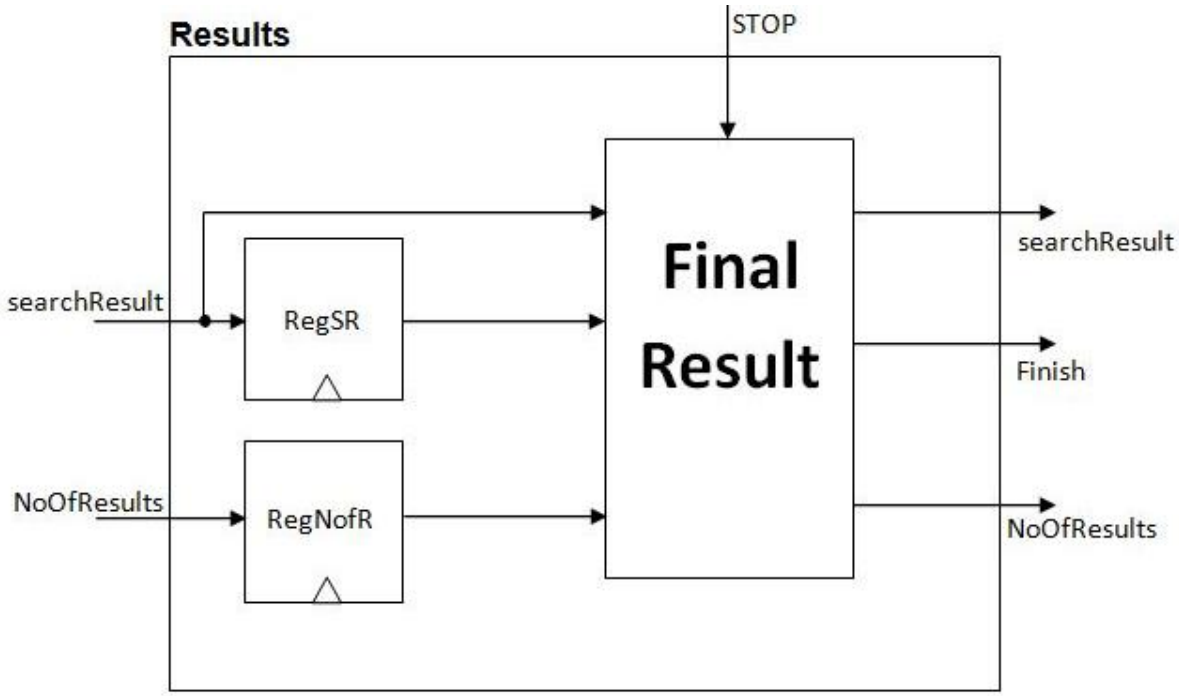
Table 4.3 STOP Module Component Inputs and Outputs.

4.1.3 Results component

This component is responsible for synchronizing the results. In Picture 4.6 we can see where the component is located and in Picture 4.7 the block diagram of the component.



Picture 4.6 Results component



Picture 4.7 Results component block diagram

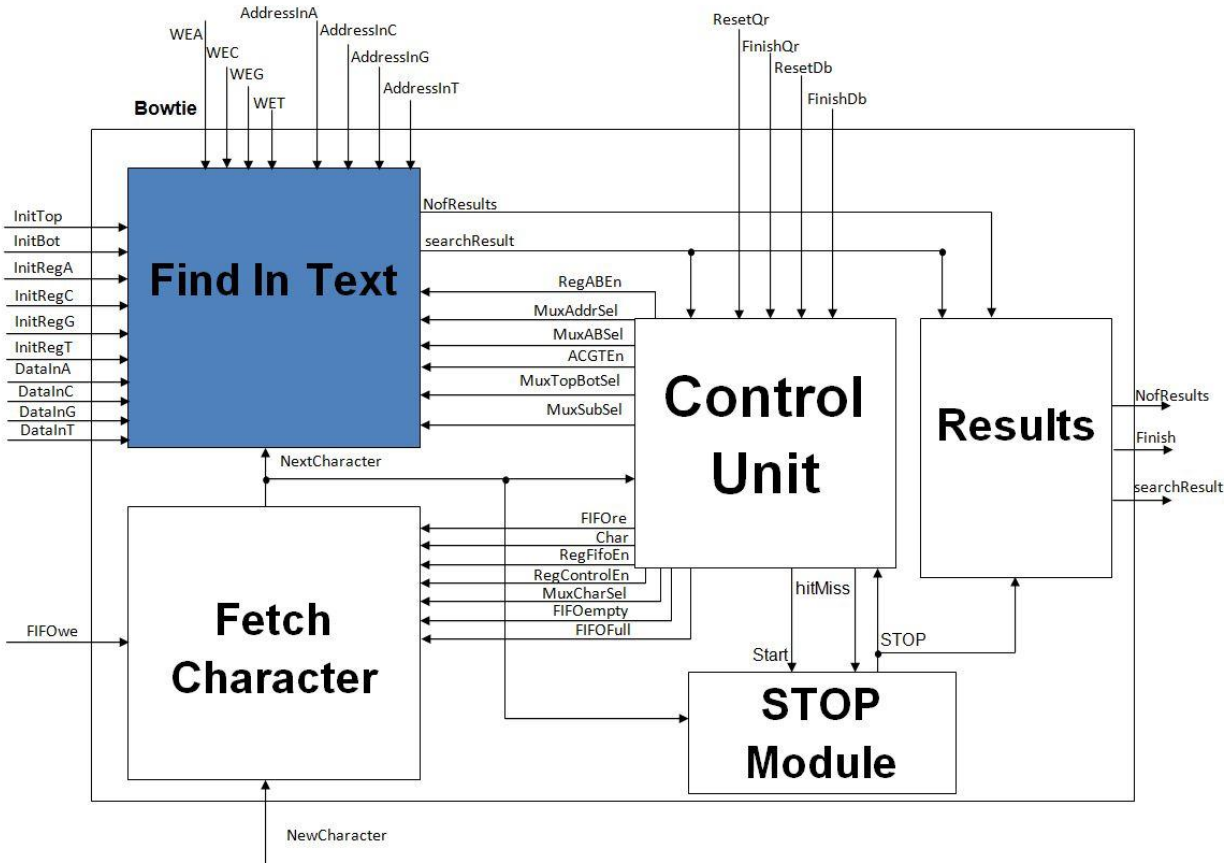
It consists from two registers and the component Final Result. When the STOP signal is switched on, then we check what the searchResult is and then give the right value to the output signals. In Table 4.4 it is detailed shown inputs and outputs of the component.

Index	Description	Size	I/O
1	searchResult	1 bit	Input
2	NoOfResults	16 bit/17 bit	Input
3	STOP	1 bit	Input
4	Finish	1 bit	Output
5	searchResult	1 bit	Output
6	NoOfResults	16 bit/17 bit	Output

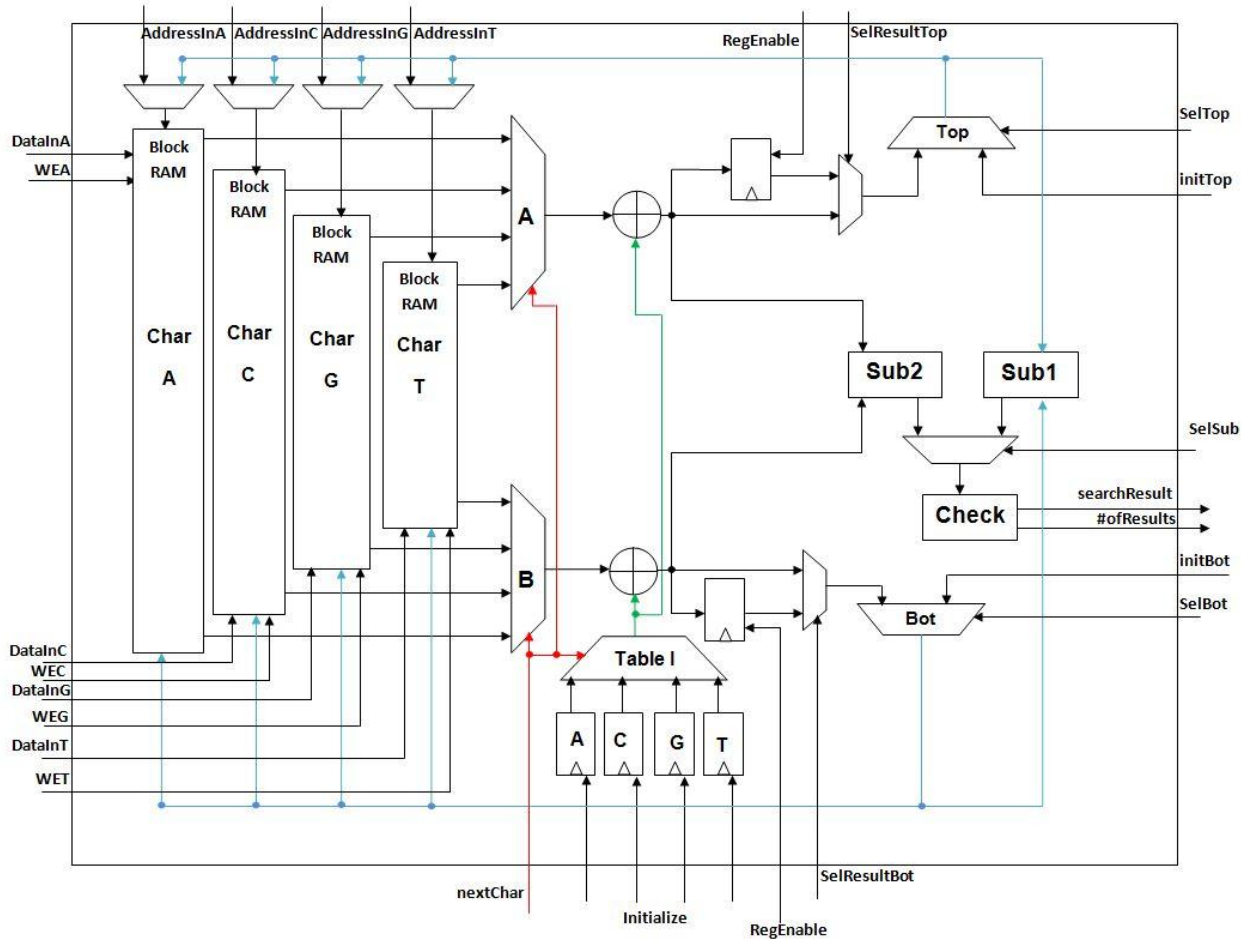
Table 4.4 Result component inputs and outputs.

4.1.4. FindInText component

FindInText component is the component responsible for finding if a character or a sequence of characters exists in the database. It is the largest component and it consists of four BRAMs, eight multiplexors, six registers, two adders, two subtracters, fourteen inputs and two outputs. In Picture 4.8 it is shown where the component is located in the design and in Picture 4.9 it is shown the block diagram of FindInText component in detail.



Picture 4.8 FindInText Component.



Picture 4.9. FindInText component datapath.

Firstly, BRAMs, A,C,G,T registers and the signals InitBot and InitTop are initialized. The value that passes through the multiplexors Top and Bot are the initTop and the initBot respectively. As a result, we have the initial address to start extracting values from the BRAMs. If we write data in the BRAMs we select the external address signals otherwise we select the signal that comes from multiplexor Top. Until a character arrives, the values are ready for multiplexors A,B and Table I to be selected from the character. After we have selected the correct value we pass it to the adders, and when the result is ready is sent to the multiplexor selecting the current value or the value the adder had in the previous cycle. After that we sent it through the multiplexors Top and Bot, this time selecting not the initialize value but the value that we have calculated and we sent it as an address to the BRAMs to select the next values and to the subtracter. If the result of the subtracter is greater than zero then we have a hit, otherwise we have a miss. Table 4.5 shows input and output signals of the component.

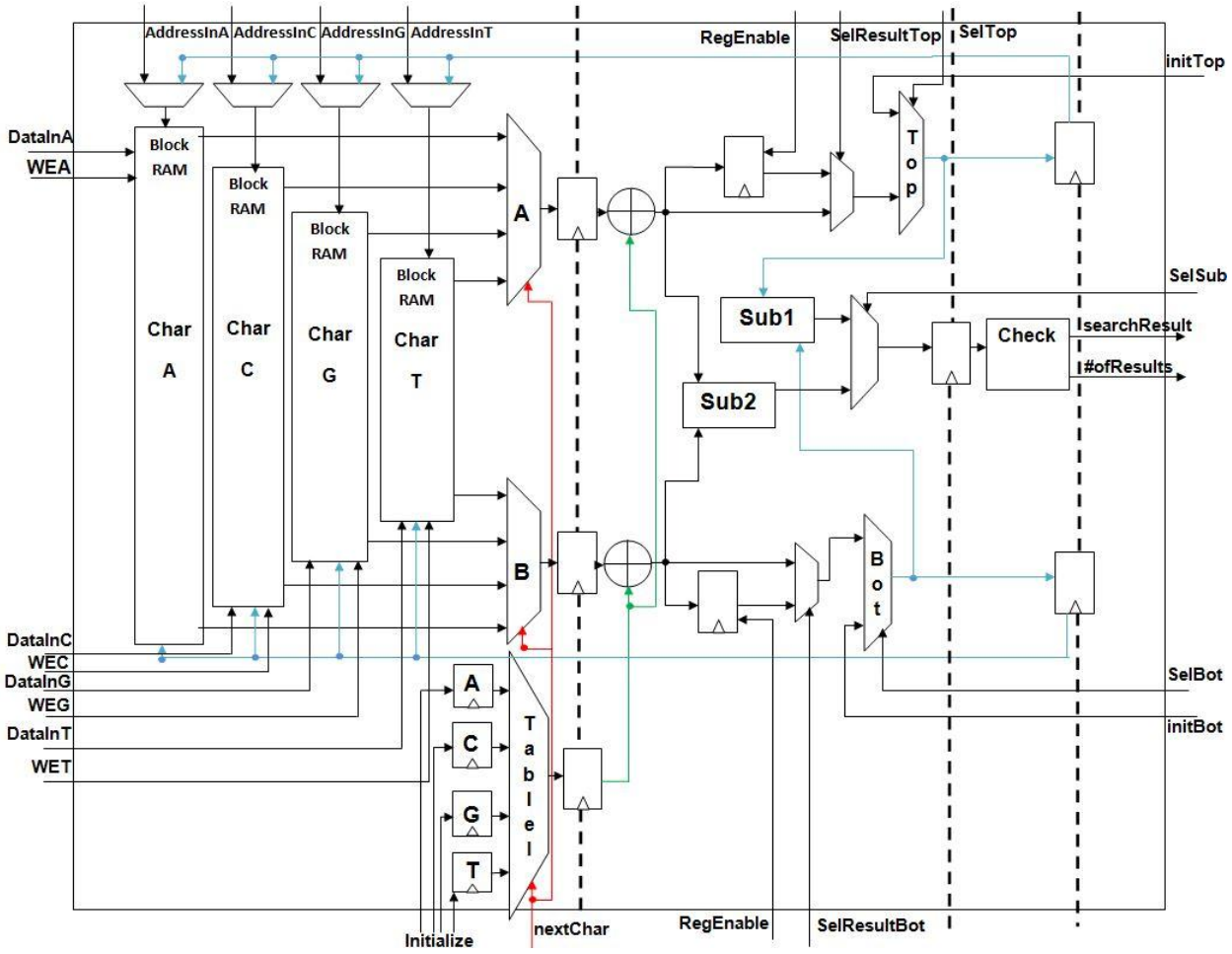
Index	Description	Length	I/O
1	nextChar	3 bit	Input
2	InitTop	16 bit/17 bit	Input
3	InitBot	16 bit/17 bit	Input
4	InitCharA	16 bit/17 bit	Input
5	InitCharC	16 bit/17 bit	Input
6	InitCharG	16 bit/17 bit	Input
7	InitCharT	16 bit/17 bit	Input
8	SelTopBot	1 bit	Input
9	SelTopTop	1 bit	Input
10	SelResultBot	1 bit	Input
11	SelResultTop	1 bit	Input
12	RegEnable	1 bit	Input
13	DataInA	16 bit/17 bit	Input
14	DataInC	16 bit/17 bit	Input
15	DataInG	16 bit/17 bit	Input
16	DataInT	16 bit/17 bit	Input
17	WEA	1 bit	Input
18	WEC	1 bit	Input
19	WEG	1 bit	Input
20	WET	1 bit	Input
21	searchResult	1 bit	Output
22	NoOfResults	16 bit/17 bit	Output

Table 4.5 FindInText Component Inputs and outputs.

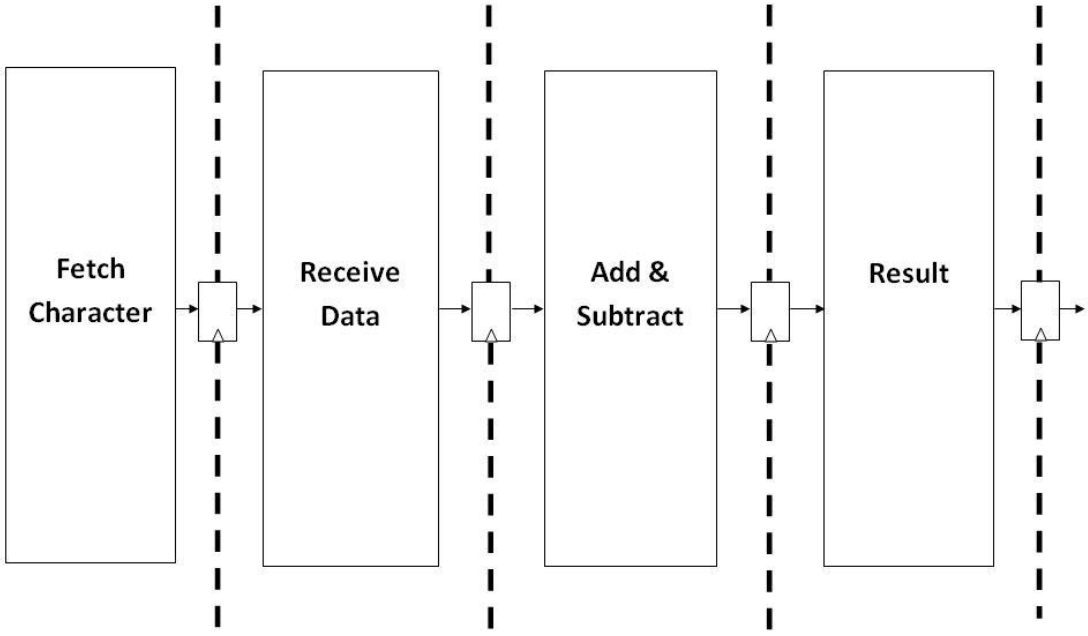
4.1.5 Design Evaluation

After evaluating the results and observed that they are correct, we notice that we use 1% of logic utilization, 42% of block memory utilization but having a clock frequency of 42MHz frequency.

Studying the results we decided to proceed with two designs, one with memories up to 50% but fitting it twice in the FPGA and the other one with memories up to 100% and fitting it only once in the FPGA. The trade offs of this choice is that with small memories it means that we can serve genomes with smaller database but executing 8 threads simultaneously and with the largest memories we can serve genomes with a bigger sequence but executing 4 threads simultaneously. The next thing to do was to modify our design due to timing constraints and the low clock frequency. We inserted five levels of pipeline overall in the whole design which the four of them are in the FindInText module. Picture 4.10 shows the pipelined design of this module and Picture 4.11 shows the overall pipelined design.



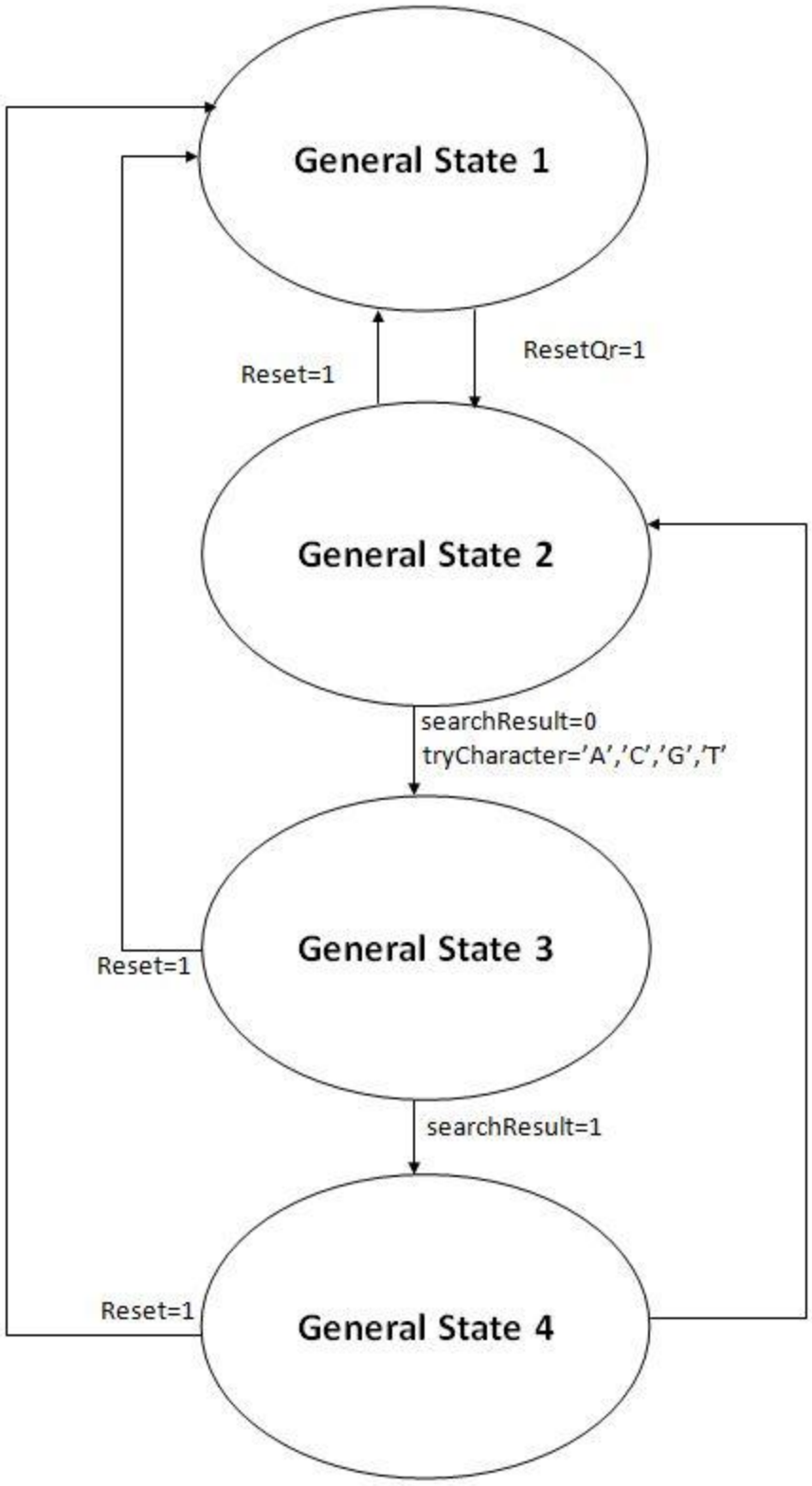
Picture 4.10 Pipelined block diagram of Find In Text module.



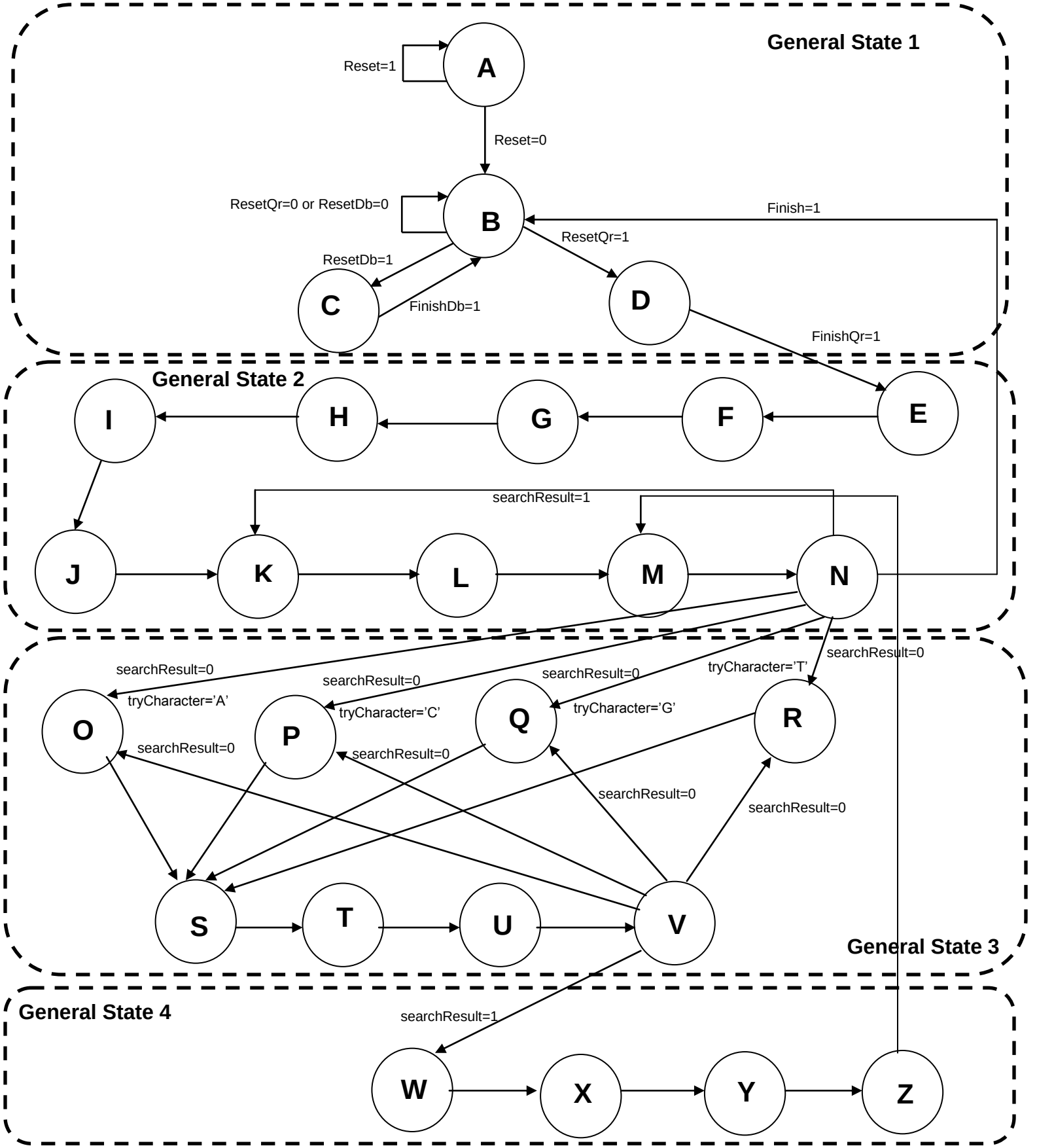
Picture 4.11 Pipelined block diagram of the overall design.

4.2 FSM

In this section the brains of the design is going to be discussed. The control unit component controls every signal at any given time. The FSM consists of twenty five states. The first state, State A, is the state where the system is in Reset mode for as long we have switch on the Reset. When the Reset signal is turned off we move to the next state, State B. We remain in State B until we receive ResetQr or ResetDb. If ResetDb is on, then we move to State C, which is responsible for the writing of the database to the BRams, and we stay there until the signal FinishDb is switched on. When is switched on indicating that the writing of the database in the memories is finished and that the registers took the correct value, we move back to state B. Now if ResetQr is on, this time we move to State D. State D is the state responsible controlling the signals for loading the queries. We remain in this state until the signal FinishQr is switched on indicating that the query is written in memory. After we exit State D we move to State E where all the Read Enable signals of the system are on (Memories, Registers), so data can start flowing through our system. We need 10 cycles until we fill up our pipeline and begin to receive results. The first result comes in State N so we are able to check the result if it is true or not, if search result is true then we move to State K and load a new character but if the result is false then we pause the system and we move to one of the following error states, State O, P, Q or R. The movement happens according to what was the last character. After we select character we wait until State V to get the result and check it. If the result is true then we move back to State M through States W, X, Y and Z, if not then we move to one of the remaining error states and try another character until we find the correct one. The process ends when the Finish signal is switched on and at the same time we store the result. When we are finished we move to State B and wait for a new database or a new query. When Reset is turned on then on whatever state the system is, it returns to State A. A general layout of the FSM is shown below in Picture 4.12 and a more detailed in Picture 4.13.



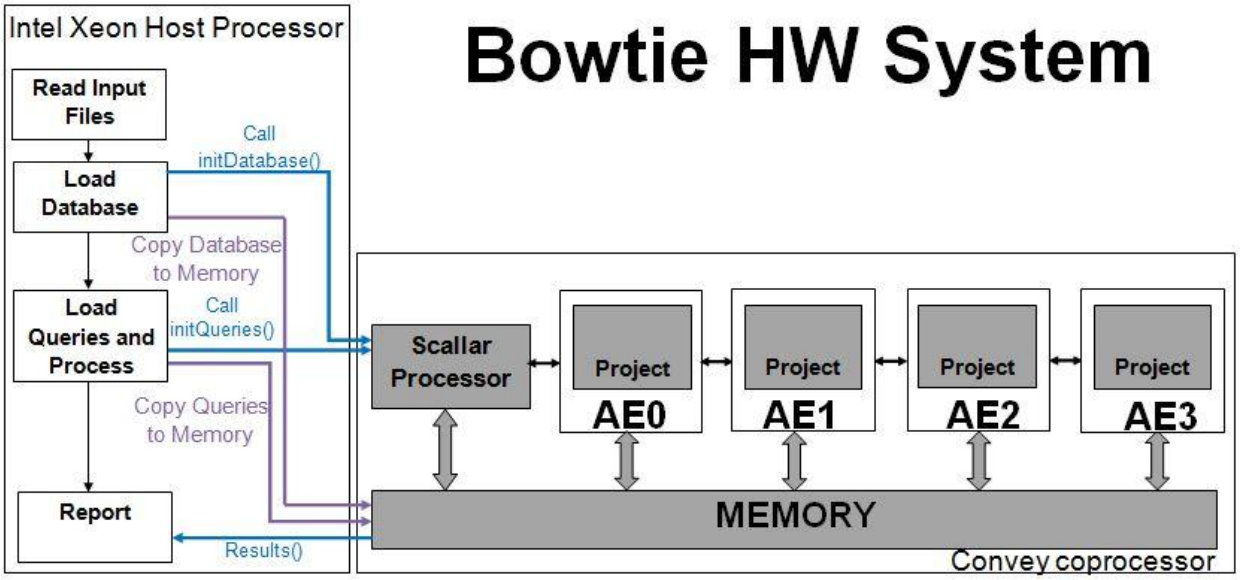
Picture 4.12 General FSM Layout



Picture 4.13 Detailed FSM layout

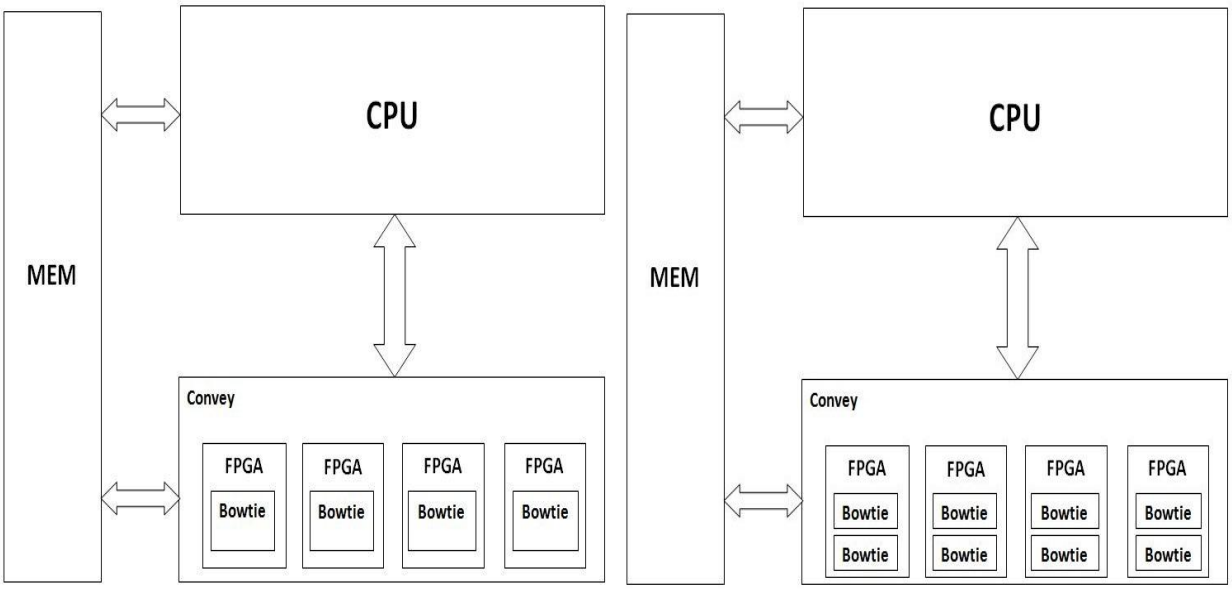
4.3 Hardware Implementation

For the hardware implementation we have use the Convey Computer HC – 1 a hybrid computer with FPGA's which it was introduced in 2009. The system specifications are dual – socket Intel Server motherboard, Intel 5400 memory controller hub chipset, 24 GBytes of RAM, 1066 MHz FSB, 2.13 GHz Xeon, a quad – core, low – voltage processor and four VIRTEX – 55 LX 330s. In the picture below we can see the configuration of HC – 1 **[46]**, **[47]**, **[48]**.



Picture 4.14 Convey Computer HC – 1 configuration.

To be able to connect our design with the Convey HC – 1, we wrote C and Verilog code (Wrapper), were we read the input files and then we send them to the design. The design clock frequency that we achieved is 227MHz for the small design and 198MHz for the big design and the actual clock frequency that we achieved is 128MHz for the small design and 116MHz for the big one. This is due mapping and routing on the FPGA. In picture 4.15 we can see the designs that we have implemented and connected on Convey HC – 1 platform.



Picture 4.15 Left: System with one design on every FPGA. Right: System with two designs on every FPGA.

Another important subject for discussion is the device utilization summary. Below, in Table 4.6, we can see the logic used by the two designs after we have synthesized them on a Virtex 5 XC5VLX330 FPGA, and after on the Convey HC – 1.

	Design for 60K Characters				Design for 100K Characters			
	XC5VLX330		Convey HC – 1		XC5VLX330		Convey HC – 1	
Number of occupied Slices	542/51840	1%	30103/521840	58%	850/51840	1%	25192/51840	48%
Number of BRAM/FIFO	121/288	42%	288/288	100%	197/288	68%	257/288	89%
Number of DSP's	0/192	0%	0/192	0%	0/192	0%	0/192	0%

Table 4.6 Device Utilization Summary.

After we have examined Table 4.6 we came to the conclusion that the Convey HC – 1 needs another 53% on average of Logic and 20% of BRAMs for the connection to our design. The reason that logic utilization of the small design is more than the big design is because the small design is placed twice on every FPGA. The extra memory and logic allocation is due to the extra verilog code, 1 FSM for every port we use, we wrote to read data from different ports of the memory controller. Totally we use 6 out of the 16 available ports for the big design and 12 out of 16 for the small design. We could use less memory controllers, meaning using less logic, but in this way we can transfer data much faster.

Chapter 5

Results

In this Chapter we are going to discuss about the results, how did they occur and their meaning.

5.1 Validation

To ensure the correct functionality of our design we aligned, firstly, some of our own examples and, afterwards, reads from the NCBI both to the software design and to our design. Once we got the results from both designs, extracting them in text files, we read them using a script written in MATLAB and compare the results.

For the execution of our experiments and the creation of the database files we used the NC_00978C.fna, a FASTA file containing the complete sequence of Escherichia coli E24377A plasmid pETEC_80, for the 100.000 characters design and the NC_017647.fna, a FASTA file containing the complete sequence of Escherichia coli 07:K1 str. CE10 plasmid PCE10A, for the 60.000 characters design. In Table 5.1 we can see the files for the queries that we used.

Number of queries	Files	
1.000	SRR000001.sra	SRR000135.sra
10.000	SRR000004.sra	SRR000920.sra
100.000	SRR000066.sra	SRR001316.sra
500.000	SRR005039.sra	SRR011186.sra
1.000.000	SRR029691.sra	SRR023975.sra
10.000.000	SRR029703.sra	SRR036755.sra
100.000.000	SRR036753.sra	SRR388772.sra

Table 5.1. List of SRA files that we used.

The sra files need processing with the SRAtoolkit, transforming them to FASTQ for running and on software and on hardware. All files are located in the NCBI database and are free for downloading and using.

5.2 Analysis and Comparison

We compare our results to the Bowtie software tool used for mapping DNA sequences. We executed the Bowtie software tool selecting full sensitivity, allowing three mismatches, using the Convey HC – 1. Table 5.2 shows the specifications of CPU running bowtie and our implementation.

	CPU
CPU Type	Xeon 5138
Number of Cores	Quad Core
Memory Size	24GB
Frequency	2.13 GHz

Table 5.2 Convey HC – 1 specifications.

We measured the execution time of searching 111 million reads in total with length 36 to 1024 base pairs on each design.

On each design we have tested seven query files, executing on hardware and on software. The files include a range from 1.000 to 100.000.000 respectively. The measurements that we took for the 60.000 Characters design are shown in Table 5.3 and for the 100.000 Characters design are shown in Table 5.4.

Number of Queries	60K Design Time (sec)	Bowtie Software Tool (sec)	Speed Up
1.000	0,00576	0,054128	9,40
10.000	0,025074	0,291998	11,65
100.000	0,217123	2,786437	12,83
500.000	1,089616	13,573387	12,46
1.000.000	2,113375	27,883339	13,19
10.000.000	21,247458	354,40833	16,68
100.000.000	212,990961	3281,239732	15,41

Table 5.3 Execution times comparing our 60.000 Characters Design versus Bowtie Software Tool.

Number of Queries	100K Design Time (sec)	Bowtie Software Tool (sec)	Speed Up
1.000	0,007977	0,054343	6,81
10.000	0,036598	0,278742	7,61
100.000	0,324073	2,722579	8,40
500.000	1,563452	18,095577	11,57
1.000.000	3,114806	27,729114	8,90
10.000.000	31,514343	302,813397	9,61
100.000.000	314,692125	3213,217386	10,21

Table 5.4 Execution times comparing our 100.000 Characters Design versus Bowtie Software Tool.

In the execution times above for our designs we aggregate and the time for loading the database which is 0,004218 sec for the small design and 0,005295 for the big one. Studying the results from the tables above we can see that our designs are faster from the Bowtie software tool achieving a 6x-16x speed up. In this speed up our design starts to make the difference processing files with content more than 500.000 queries and we can see that as long as the queries increases the speed up increases.

In figure 5.1 below it is shown the execution time of the bowtie software tool comparing to our design of 60.000 characters and figure 5.2 shows the execution time of the bowtie software tool comparing to our design of 100.000 characters.

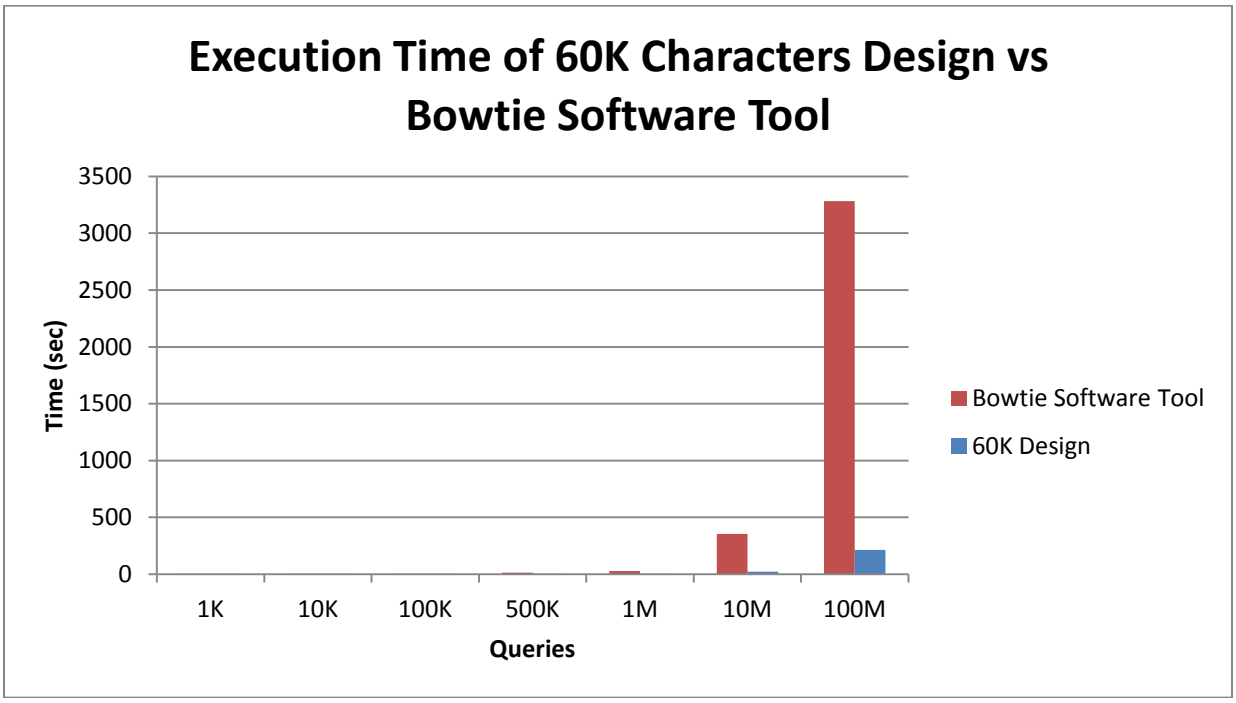


Figure 5.1. Execution time of 60K Characters design comparing to Bowtie Software Tool.

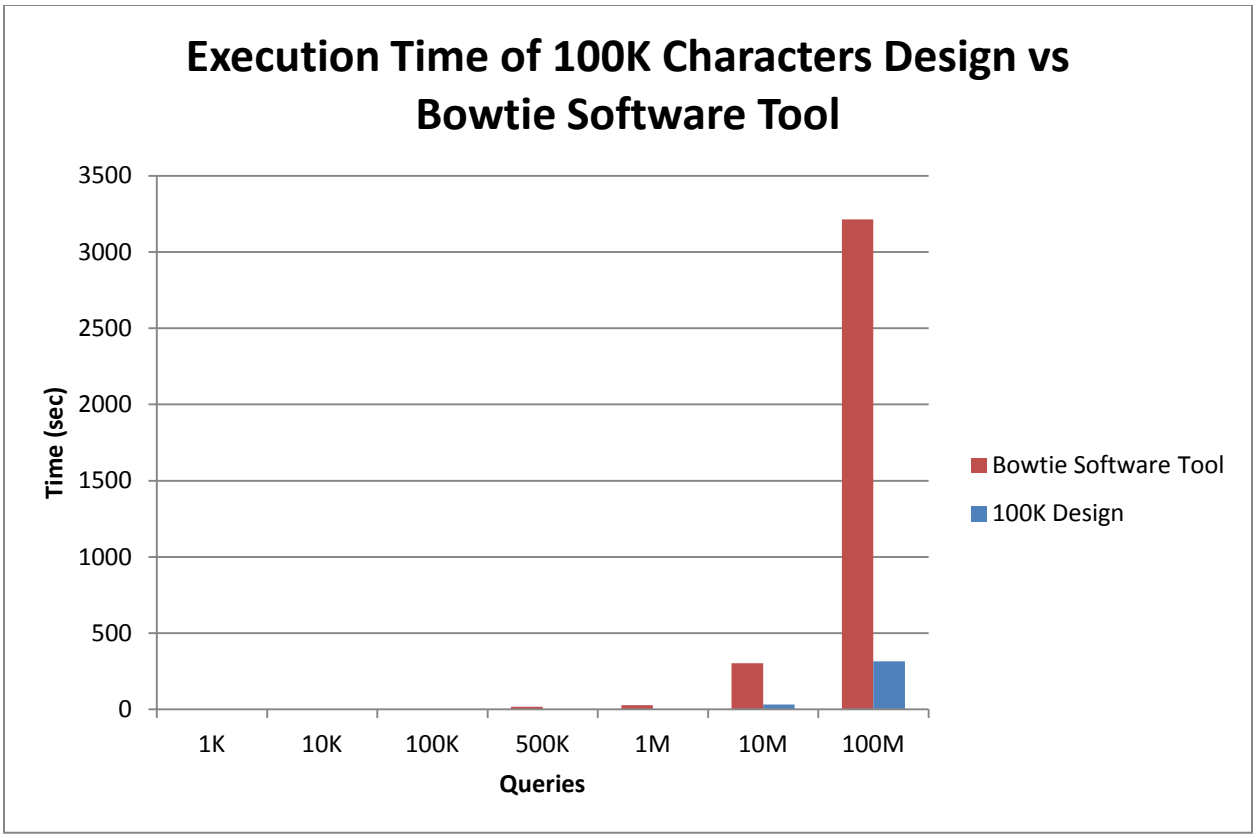


Figure 5.2. Execution time of 100K Characters design comparing to Bowtie Software Tool.

The 60.000 Characters design is faster than the 100.000 Characters design, as expected, because of larger clock frequency and for the reason that the design runs twice on every FPGA. Figure 5.3 shows the speed up that we achieved comparing our two designs.

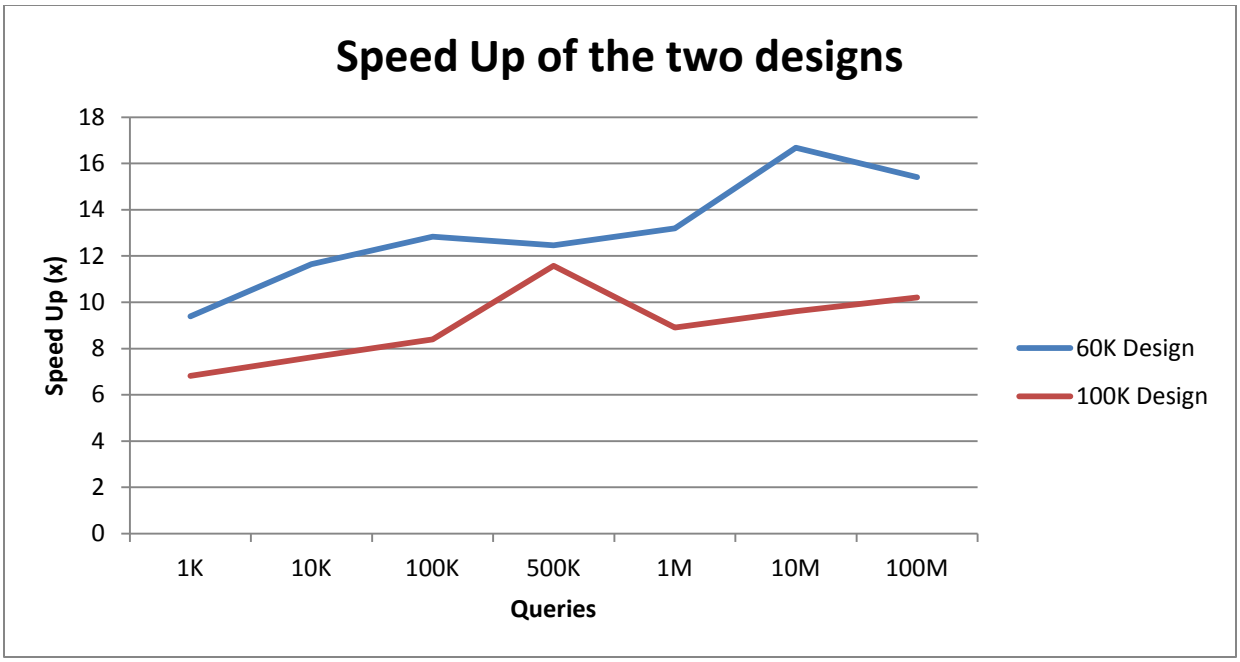


Figure 5.3. Comparing our two designs by the achieved Speed Up.

5.3 Related Work

We have compare our design with the designs appeared in **[21]**, Design 1, and **[20]**, Design 2. From the results we can see that we are slower from both implementations. Comparing our design to Design 1 we can see that our FPGA has half the number of Slices and one third of BRAMs from the FPGA used in Design 1. Also we both use the Escherichia coli database but we use a bigger range size on query length. Another difference is that our implementation supports three mismatches and Design 1 supports none. The differences between our design and Design 2 is that the memory on our Convey is a lot less, and again we use a bigger range size on query length. Table 5.5 shows the differences of the three designs in a brief.

	TUC Design	Riverside Design 1	Riverside Design 2
FPGA	XC5VLX330	XC6VLX760	XC5VLX330
Number of Slices	51480	118560	51480
Number of BRAM/FIFO	288	720	288
CPU Type	Xeon 5138	-	Xeon L540B
CPU Frequency	2.13GHz	-	2.13GHz
Memory	24GB	-	192GB
Algorithm functionality	Search	Limited Search	Search
Database	Escherichia coli	Escherichia coli	Chromosome 14
Queries Length	36-1024	36-108	101
Speed Ups	6x-16x	124x-196x	7x-70x

Table 5.5 Comparing our design with Design 1 and Design 2 of Riverside.

Chapter 6

Conclusions & Future Work

Chapter 6 talks about final conclusions and future work.

6.1 Conclusions

In this diploma thesis we have studied and analyzed the methods of Burrows and Wheeler Transformation, FM – Index and short read mapping algorithm Bowtie. Then we designed it on hardware and implemented it. Our design running on a Convey HC – 1 platform was faster than Bowtie on the same platform. Bowtie algorithm loses of speed when the sequence is not correct and tries to replace the wrong character with the correct one, when it finishes the processing of a sequence and tries to move to the next one and finally as long as the short reads size increases the performance of the algorithm decreases. We can align databases up to 100.000 characters long serving mostly bacteria such as Escherichia coli, Enterococcus, Enterobacter, Acetobaster pasteurianus etc. Furthermore, you can work easily from your office on the Convey HC – 1 platform, the output results come right out on the terminal on your desktop with no further equipment needed and finally, many problems can be solved easily and quick using the tools that come along with the platform.

6.2 Future Work

Some further work on this diploma thesis could be:

1) Further analyze of Bowtie so more functionalities of the algorithm can be implemented such as aligning colorspaces.

2) Using FM – Index and BWT on existing designs for improving their performance not only in Bioinformatics but also in Networking, Systems, Software etc.

3) Improving the architecture design in order to improve the clock frequency.

4) Improving the architecture design and the memory design in order to serve bigger databases.

5) Improving the code written in verilog in the wrapper of the Convey platform in order to decrease the logic and memory utilization.

6) Try to implement the resource demanding functions in hardware of the algorithm and not a whole functionality.

References

[1] S. L. Salzberg, A. L. Delcher, S. Kasif, O. White. "**Microbial gene identification using interpolated Markov models."** Nucleic Acids Redearch, 1998, Vol. 26, No. 2.

[2] A. L. Delcher, D. Harmon, S. Kasif, O. White, S. L. Salzberg. "**Improved microbial gene identification with GLIMMER."** Nucleic Acids Research, 1999, Vol. 27, No. 23.

[3] A. L. Delcher, K. A. Bratke, E. C. Powers, S. L. Salzberg. "**Identifying Bacterial genes and endosymbiont DNA with Glimmer**." Bioinformatics 2007.

[4] W.H. Majoros, M. Pertea, S. L. Salzberg. "**Efficient implementation of a generalized pair hidden Markov model for comparative gene finding**." Bioinformatics Vol. 21, no. 9, 2005, pages 1782 – 1788.

[5] C. Burge, S. Karlin. "**Prediction of Complete Gene Structures in Human Genomic DNA**." Journal of Molecular Biology (1997) 268, 79 – 94.

[6] W. H. Majoros, M. Pertea, S. L. Salzberg. "**TigrScan and GlimmerHMM: two open source ab initio eukaryotic gene-finders."** Bioinformatics, 2004 – Oxford Univ Press.

[7] J.E. Allen, W.H. Majoros, M. Pertea, S. L Salzberg. "**JIGSAW, GeneZilla, and GlimmerHMM: puzzling out the features of human genes in the ENCODE regions."** Genome Biology, 2006.

[8] H. Li, J. Ruan, R. Durbin. "**Mapping short DNA sequencing reads and calling variants using mapping quality scores."** Genome research, 2008.

[9] R. Li, Y. Li, K. Kristiansen, J. Wang. "**SOAP: short oligonucleotide alignment program."** Bioinformatics, 2008 – Oxford Univ. Press.

[10] A. D. Smith, Z. Xuan, M. Q. Zhang. "**Using quality scores and longer reads improves accuracy of Solexa read mapping**." BMC bioinformatics, 2008.

[11] H. Lin, Z. Zhang, M.Q. Zhang, B. Ma, M. Li. "**ZOOM! Zillions of oligos mapped**." Bioinformatics, 2008- Oxford Univ Press.

[12] S.M. Rumble, P. Lacroute, A.V. Dalca, M. Fiume, A. Sidow, M. Brudno. "**SHRiMP: Accurate Mapping of Short Color – space Reads."** PLoS computational Biology, 2009.

[13] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg. "**Ultrafast and memory – efficient alignment of short DNA sequences to the human genome**." Genome Biology, 2009.

[14] M. Pertea , X. Lin , S. L. Salzberg . "**GeneSplicer : a new computational method for splice site prediction** ." *Nucleic Acids Res* . 2001 Mar 1;29(5):1185-90 .

[15] J. E. Allen and S. L. Salzberg. **"A phylogenetic generalized hidden Markov model for predicting alternatively spliced exons.**" *Algorithms for Molecular Biology*, 1:14, 2006.

[16] J. E. Allen, M. Pertea and S. L. Salzberg. "**Computational gene prediction using multiple sources of evidence.**" *Genome Research*, 14(1), 2004.

[17] J. E. Allen and S. L. Salzberg. **"JIGSAW: integration of multiple sources of evidence for gene prediction.**" *Bioinformatics*21(18): 3596-3603, 2005.

[18] R. Li, C. Yu, Y. Li, T.W. Lam, S. M. Yiou, K. Kristiansen, J. Wang. "**SOAP2: an improved ultrafast tool for short read alignment**." Bioinformatics, 2009 – Oxford Univ. Press.

[19] M. Burrows, D. J. Wheeler. "**A Block – sorting Lossless Data Compression Algorithm**." 1994 – Citeseer.

[20] E. Fernandez, W. Najjar, S. Lonardi and J. Villarreal. "**Multithreaded FPGA Acceleration of DNA Sequence Mapping."** 2012 IEEE High Performance Extreme Computing Conference (HPEC '12), Waltham, MA USA, 2012.

[21] E. Fernandez, W. Najjar, and S. Lonardi, **"String Matching in Hardware using the FM-Index."** In Proc. IEEE Int. Symp. on Field-Programmable Custom Computing Machines, FCCM 2011, pages 218-225, Salt Lake City, UT, USA, 2011.

[22] E. Fernandez, W. Najjar, E. Harris and S. Lonardi, **Exploration of Short Reads Genome Mapping in Hardware,** In Proc. of 20th Int. Conf. on Field Programmable Logic and Application, 2010.

[23] H. Li and R. Durbin, **"Fast and Accurate Short Read Alignment with Burrows-Wheeler Transforms."** Bioinformatics, 2009.

[24] C. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, W. Ruzzo, **"Hardware Acceleration of Short Read Mapping."** in Proc. IEEE Int. Symp. on Field-Programmable Custom Computing Machines, FCCM 2012.

[25] T. F. Smith, M. S. Waterman. **"Identification of common molecular subsequences"**. Journal of Molecular Biology 147: 195–197, 1981.

[26] T. Rognes, E. Seeberg. **"Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors"**. Bioinformatics 16: 699–706, 2000.

[27] M. Farrar. **"Striped smith-waterman speeds database searches six times over other simd implementations"**. Bioinformatics 23: 156–161, 2007.

[28] A. Wozniak. **"Using video-oriented instructions to speed up sequence comparison"**. Comput Appl Biosci. pp 145–150, 1997.

[29] M. David, M. Dzamba, D. Lister, L. Ilie, M. Brudno. **"SHRiMP2: sensitive yet practical short read mapping"**. Bioinformatics, 27, 1011, 2011.

[30] P. Ferragina, G. Manzini, V. Makinen, G. Navarro. **"An Alphabet – Friendly FM - Index"**. In Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science, vol. 3246. Springer-Verlag, Berlin, Germany, 150–160, 2004.

[31] S. Grabowski, V. Makinen, G. Navarro. **"First Huffman, then Burrows-Wheeler: An alphabet independent FM-index"**. In Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science, vol. 3246. Springer-Verlag, Berlin, Germany, 210–211, 2004.

[32] S. Grabowski, G. Navarro, R. Przywarski, A. Salinger, V. Makinen. **"A simple alphabetindependent FM-index"**. Int. J. Found. Comput. Sci. 17, 6, 1365–1384, 2006.

[33] ftp://ftp.ncbi.nlm.nih.gov/genomes/

[34] http://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?view=download_reads

[35] http://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?view=software

[36] P. Hogeweg. **"The Roots of Bioinformatics in Theoretical Biology".** PLoS Comput Biol 7(3): e1002021. doi:10.1371/journal.pcbi.1002021, 2011

[37] C. A. Ouzounis. **"Rise and Demise of Bioinformatics? Promise and Progress".** PLoS Comput Biol 8(4): e1002487. doi:10.1371/journal.pcbi.1002487, 2012.

[38] J. A. Foster. **"Evolutionary Computation"**. Nature Reviews Genetics, 2001.

[39] http://en.wikipedia.org/wiki/File:Gene_structure.svg

[40] http://en.wikipedia.org/wiki/File:Pre-mRNA_to_mRNA.svg

[41] http://en.wikipedia.org/wiki/Gene#cite_note-1

[42] http://en.wikipedia.org/wiki/Exon

[43] http://en.wikipedia.org/wiki/Intron

[44] http://en.wikipedia.org/wiki/RNA_splicing

[45] J. D. Bakos. **"High- Performance Heterogeneous Computing with the Convey HC – 1"**. Scholar Commons, 2010.

[46] http://www.conveycomputer.com/technology/partners/

[47] Convey Computer. **"Convey Personality Development Kit Reference Manual"**. Version 5.2, April 2012.

[48] K. S. P. Pereira. **"Characterization of FPGA – based High Performance Computers"**. Blacksburg, Virginia, 2011.

[49] D. Hoang et. al. "**FPGA Implementation of Systolic Sequence Alignment"**, Proceedings of the 2nd International Workshop on Field-Programmable Logic and Applications, Lecture Notes in Computer Science 705, pp 183-191, 1992.

[50] D. Hoang. **"Searching Genetic Databases on Splash 2"**, Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), pp 185-191, 1993.

[51] S. Guccione, E. Keller. **"Gene Matching Using JBits"**, Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, Lecture Notes In Computer Science; Vol. 2438, pp 1168-1171, 2002.

[52] K. Puttegowda et. Al. **"A Run-Time Reconfigurable System for Gene-Sequence Searching"**, Proceedings, 16th International Conference on VLSI Design pp 561 – 566, New Delhi 2003.

[53] T. Oliver, B. Schmidt, D. Maskel. **"Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs"**, IEEE Transactions on Circuits and Systems II, Vol, 52, No, 12, pp, 851-855, 2005.

[54] E. Sotiriades, A. Dollas. **"A General Reconfigurable Architecture for the BLAST algorithm"**, The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Special Issue on Computing Architectures and Acceleration for Bioinformatics Algorithms, Kluwer Academic Publishers Volume 48, Issue 3 Pages: 189 – 208, September, 2007.

[55] G. Chrysos, E. Sotiriades, I. Papaefstathiou, A. Dollas. **" A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM)."** In *Field*

Programmable Logic and Applications, 2009. FPL 2009. International Conference on (pp. 683-686). IEEE, 2009
[56] N. Alachiotis, E. Sotiriades, A. Dollas, A. Stamatakis. **"Exploring FPGAs for accelerating the phylogenetic likelihood function."** In*Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (pp. 1-8). IEEE, May, 2009.

[57] H. M. Hussain, K. Benkrid, H. Seker, A. T. Erdogan. **"FPGA implementation of K-means algorithm for bioinformatics application: an accelerated approach to clustering Microarray data."** In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on* (pp. 248-255). IEEE, June, 2011

[58] P. Chen, C. Wang, X. Li, X. Zhou. **"Acceleration of the long read mapping on a PC-FPGA architecture."** In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (pp. 271-271). ACM, February, 2013.

[59] Y. Chen, B. Schmidt, D. L. Maskell. **"Accelerating short read mapping on an FPGA."** In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (pp. 265-265). ACM, February, 2012.

[60] Y. Yamaguchi, Y. Osana, M. Yoshimi, H. Amano. **"FPGA-Based HPRC for Bioinformatics Applications."** In *High-Performance Computing Using FPGAs* (pp. 137-175). Springer New York, 2013.

[61] P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, D. Pnevmatikatos. **"A rate-based prefiltering approach to BLAST acceleration."** In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* (pp. 631-634). IEEE, September, 2008.

[62] G. Chrysos, E. Sotiriades, I. Papaefstathiou, A. Dollas. **"A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM)"**. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* (pp. 683-686). IEEE, August, 2009.

[63] E. Sotiriades, C. Kozanitis, G. Chrysos, A. Dollas. **"Rapid Phototyping of a System-on-a-Chip for the BLAST Algorithm Implementation"**. In*Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*(pp. 223-229). IEEE, June, 2006.

[64] P. Afratis, C. Galanakis, E. Sotiriades, G. G. Mplemenos, G. Chrysos, I. Papaefstathiou, D. Pnevmatikatos. **" Design and implementation of a database filter for BLAST acceleration."** In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.* (pp. 166-171). IEEE, April, 2009.

[65] M. Smerdis, P. Dagritzikos, G. Chrysos, E. Sotiriades, A. Dollas. **"Reconfigurable Systems for the Zuker and Predator Algorithms for Secondary Structure**

Prediction of Genetic Data." In *Field Programmable Logic and Applications (FPL), 2010 International Conference on* (pp. 448-451). IEEE, August, 2010.
[66] N. Chrysanthou, G. Chrysos, E. Sotiriades, I. Papaefstathiou. **"Parallel accelerators for GlimmerHMM bioinformatics algorithm."** In*Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*(pp. 1-6). IEEE, March, 2011.

Appendix A

MATLAB code for processing input data.

A.1 Code for processing database files

```
%% Open and read file
disp('Reading Database...')
pause on
pause off
[fileID, message]=fopen('database1.txt');
A=fscanf(fileID,'%s');
fclose('all');
disp('Read Database succesfully!')
pause on
pause off

%% Procedure BWT
tic;
disp('Starting...')
pause on
pause off
BWT=CreateBWT(A);
disp('BWT Created!')
disp('Sorting...')
pause on
pause off
SBWT=CreateSBWT(BWT);
disp('Sorted!')
disp('Creating Table I...')
pause on
pause off
TableI=CreateTableI(SBWT);
%clear SBWT;
disp('Table I Created!')
disp('Creating Table C...')
pause on
pause off
TableC=CreateTableC(BWT);
%clear BWT;
disp('Table C Created!')
```

```
pause on
pause off
TotalTime=toc/60;
disp(['BWT Procedure Total Execution Time=',num2str(TotalTime),'mins'])
pause on
pause off

%% Export .txt file importing in C with all values

MemoryWidth=16;
largest=max(max(TableC));
bits=ceil(log2(largest));

MemoryDepth=16384;

tic;
disp('Creating BowtieValues.txt...')
pause on
pause off
fileID=fopen('BowtieValues.txt','w');
FilesOk=CreateTxtFile(fileID,TableC,MemoryWidth,MemoryDepth,TableI);

if FilesOk>0
    disp('BowtieValues.txt file was created succesfuly!')
    pause on
    pause off
else
    disp('There was an error creating BowtieValues.txt file!')
end

TotalTime=toc/60;
disp(['Creating .txt file Total Execution Time=',num2str(TotalTime),'mins'])
pause on
pause off


function BWT=CreateBWT(TextStream)

    %Copy the Stream into a temporary variable
    disp('Copy the Stream into a temporary variable...')
    pause on
    pause off
    for i=1:max(size(TextStream))
        temp(i)=TextStream(i);
    end


    %Put Character $ at the end
    disp('Put Character $ at the end...')
    pause on
    pause off
    temp(max(size(TextStream))+1)='$';

    %Initialize table SuffixArray
```

```
    disp('Initialize table SuffixArray...')
    pause on
    pause off
    for i=1:max(size(temp))
        for j=1:max(size(temp))
            SuffixArray(i,j)='A';
        end
    end

    %Copy on the first row of Suffix Array the Text Stream after we
    %inserted character $
    disp('Copy on the first row of Suffix Array the Text Stream after we
inserted character $...')
    pause on
    pause off
    for i=1:max(size(temp))
        SuffixArray(1,i)=temp(i);
    end

    %Fill the rest rows by rotating the last letter every time
    disp('Rotating...')
    pause on
    pause off
    for j=2:max(size(temp))
      SuffixArray(j,max(size(temp)))=SuffixArray(j-1,1);
      for i=max(size(temp)):-1:2
        SuffixArray(j,i-1)=SuffixArray(j-1,i);
      end
    end

    %Sort lexicographicaly the rows of suffix array
    disp('Sort lexicographicaly the rows of suffix array...')
    pause on
    pause off
    sortedSuffixArray=sortrows(SuffixArray);

    %Export BWT
    disp('Exporting BWT...')
    pause on
    pause off
    for i=1:max(size(temp))
        temp2(i)=sortedSuffixArray(i,max(size(temp)));
    end

    for i=1:max(size(temp))
        BWT(i)=temp2(i);
    end

end

function SBWT=CreateSBWT(BWT)

    %Sort BWT sequence and export SBWT
    SBWT=sort(BWT);
    SBWT
```

```
end

function TableI=CreateTableI(SBWT)

    sumA=0;
    sumC=0;
    sumG=0;
    sumT=0;
    sum=0;

    %Fill TableI
    %Check for the character and increase sum by 1
    for i=2:max(size(SBWT))
        if(SBWT(i)=='A')
            sumA=sumA+1;
        elseif (SBWT(i)=='C')
            sumC=sumC+1;
        elseif (SBWT(i)=='G')
            sumG=sumG+1;
        elseif (SBWT(i)=='T')
            sumT=sumT+1;
        else
            sum=sum+1;
        end
    end

    %Insert the calculated values into TableI
    TableI(1)=1;
    TableI(2)=TableI(1)+sumA;
    TableI(3)=TableI(2)+sumC;
    TableI(4)=TableI(3)+sumG;

End

function TableC=CreateTableC(BWT)

    sumA=0;
    sumC=0;
    sumG=0;
    sumT=0;
    sum=0;

    %Fill TableC
    for i=1:max(size(BWT))

        TableC(i,1)=sumA;
        TableC(i,2)=sumC;
        TableC(i,3)=sumG;
        TableC(i,4)=sumT;

        %Check the character and increase sum by 1
         if(BWT(i)=='A')
            sumA=sumA+1;
        elseif (BWT(i)=='C')
```

```
            sumC=sumC+1;
        elseif (BWT(i)=='G')
            sumG=sumG+1;
        elseif (BWT(i)=='T')
            sumT=sumT+1;
        else
            sum=sum+1;
         end
    end

    %Insert the calculated values into the next row of TableC
    TableC(i+1,1)=sumA;
    TableC(i+1,2)=sumC;
    TableC(i+1,3)=sumG;
    TableC(i+1,4)=sumT;
  end
```

A.2 Code for processing short read files

A.2.1 Code for processing fasta files

```
%% Read .fasta file and write it on .txt file
InfoStruct=fastainfo('e_coli_1000.fa');
FASTAData=fastaread('e_coli_1000.fa');
[header, sequence]=fastaread('e_coli_1000.fa');

fileID=fopen('FastaQuery.txt','w');

if fileID>0
    disp('FastaQuery.txt file was created succesfuly!')
    pause on
    pause off
else
    disp('There was an error creating FastaQuery.txt file!')
end

fprintf(fileID,'%s\r\n',FASTAData.Sequence);
fclose(fileID);

%% Read the .txt file we write above

disp('Reading Query...')
pause on
pause off
[fileID, message]=fopen('FastaQuery.txt');

tline=fgetl(fileID);
ActualquerySize=max(size(tline));
i=1;
```

```
while ischar(tline)

    for j=1:ActualquerySize
        queries(i,j)=tline(j);
    end
    tline = fgetl(fileID);
    i=i+1;
end

fclose('all');
disp('Read Query succesfully!')
pause on
pause off

%% Write to .txt in convey form

No_Of_Queries=InfoStruct.NumberOfEntries;
querySize=1024;
fileID=fopen('MyQueryFa.txt','w');
FilesOk=CreateQueryFileFa(fileID,queries,querySize,ActualquerySize,No_Of_Quer
ies);

if FilesOk>0
    disp('MyQueryFa.txt file was created succesfuly!')
    pause on
    pause off
else
    disp('There was an error creating MyQueryFa.txt file!')
end

function
fid=CreateQueryFileFa(fileID,queries,querySize,ActualquerySize,NoOfQuerries)

%Read and rewrite backwards every sequence and replace each character
%with is equivalent integer
for i=1:NoOfQuerries
    for j=1:ActualquerySize
        if queries(i,j)=='A'
            temp(i,ActualquerySize+1-j)=1;
        elseif queries(i,j)=='C'
            temp(i,ActualquerySize+1-j)=2;
        elseif queries(i,j)=='G'
            temp(i,ActualquerySize+1-j)=3;
        else
            temp(i,ActualquerySize+1-j)=4;
        end
    end
end

 %First line is the number of queries
 fprintf(fileID,'%d\r\n',NoOfQuerries);

 %Write the query
 for j=1:NoOfQuerries
     for i=1:ActualquerySize
```

```
        fprintf(fileID,'%d\r\n',temp(i));
    end

    fprintf(fileID,'%d\r\n',0);
end

fclose(fileID);
fid=fileID;
end
```

A.2.2 Code for processing fastq files

```
%% Read .fastq file and write it on .txt file
FASTQStruct = fastqread('DRR000019.fastq');
[Header, Sequence] = fastqread('DRR000019.fastq');

No_of_FastQ_Queries= max(size(FASTQStruct));

x=FastQFiles(FASTQStruct,No_of_FastQ_Queries);

if x>0
    disp('MyQueryFq.txt file was created succesfuly!')
    pause on
    pause off
else
    disp('There was an error creating MyQueryFq.txt file!')
end


function fid=FastQFiles(FASTQStruct,No_Of_Queries)

    %% Read .fastq file and write it on .txt file
    fileID=fopen('FastqQuery.txt','w');
    fprintf(fileID,'%s\r\n',FASTQStruct.Sequence);
    fclose(fileID);

    %% Read the .txt file we write above
    disp('Reading Query...')
    pause on
    pause off
    [fileID, message]=fopen('FastqQuery.txt');

    %Initialize index table with zeros
    index=zeros(max(size(FASTQStruct)),1)    ;
    tline=fgetl(fileID);
    ActualquerySize=max(size(tline));
    i=1;

    %Read the characters from the file and place them in a table, and also
    %save in index table how many characters are they. Some times queries
```

```
%inside fastq files are not the same length.
while ischar(tline)

    ActualquerySize=max(size(tline));
    index(i,1)=ActualquerySize;
    for j=1:ActualquerySize
        queries(i,j)=tline(j);
    end
    tline = fgetl(fileID);
    i=i+1;
end

fclose('all');
disp('Read Query succesfully!')
pause on
pause off
%% Writing .txt file in convey form
size(index); %[11052 1];
fileID=fopen('MyQueryFq.txt','w');

disp('Converting chars to integers and reversing...')
pause on
pause off
for i=1:No_Of_Queries
for j=1:index(i,1)
    if queries(i,j)=='A'
        temp(i,index(i,1)+1-j)=1;
    elseif queries(i,j)=='C'
        temp(i,index(i,1)+1-j)=2;
    elseif queries(i,j)=='G'
        temp(i,index(i,1)+1-j)=3;
    else
        temp(i,index(i,1)+1-j)=4;
    end
end
end
temp;
disp('Writing .txt file...')
pause on
pause off
%First line is the number of queries
 fprintf(fileID,'%d\r\n',No_Of_Queries);

 %Write the characters
 for j=1:No_Of_Queries
     for i=1:index(j,1)
        fprintf(fileID,'%d\r\n',temp(i));
     end

    %Put a zero at the end
     fprintf(fileID,'%d\r\n',0);
 end

disp('.txt file was created succesfully!!!!')
pause on
pause off
```

```
    fclose(fileID);
    fid=fileID;
end
```

Appendix B

C code for reading text files we created in MATLAB

```
#include <stdio.h>
int main ( void )
{
    //Open txt file for reading, and count the lines so we now how big
    //to make the arrays
      FILE* myfile = fopen("BowtieValues.txt", "r");
      int ch, number_of_lines = 0;


      do {
      ch = fgetc(myfile);
      if(ch == '\n')
            number_of_lines++;
      } while (ch != EOF);
      fclose(myfile);
/*************************************************************************
*/
//Reading DataBase
int i = 0, a=0;
 int values[7];
 int top,bot,CharA,CharC,CharG,CharT,depth;


int *numbers = (int*)malloc(sizeof(int)*number_of_lines);
```

```
//Read the file and store the elements in variables
static const char filename[] = "BowtieValues.txt";
    FILE *file = fopen ( filename, "r" );

    for (i=0;i<=number_of_lines;i++)
    {
      if(i<7){
            fscanf(file,"%d",&values[i]);
        }
        else{
                  fscanf(file,"%d",&numbers[a]);
                  a++;
        }
    }
    fclose(file);
/*********************************************************************/

      myfile = fopen("MyQuery.txt", "r");
      ch, number_of_lines = 0;
      do {
      ch = fgetc(myfile);
      if(ch == '\n')
            number_of_lines++;
      } while (ch != EOF);
      fclose(myfile);
/*********************************************************************/
  // Reading query
     i = 0, a=0;
    int querySize;
    int *query = (int*)malloc(sizeof(int)*(number_of_lines-1));
```

```
static const char filename2[] = "MyQuery.txt";
FILE *file2 = fopen ( filename2, "r" );

for (i=0;i<=number_of_lines;i++)
{
  if(i<1){
        fscanf(file2,"%d",&querySize);
    }
    else{
              fscanf(file2,"%d",&query[a]);
              a++;
    }
}
fclose(file);
```