Department of Electronics and Computer Engineering – Technical University of Crete

Microprocessor and Hardware Lab (MHL)

SENIOR THESIS

Christos Kozanitis

"Study and Implementation with Reconfigurable Logic of BLAST Algorithm for DNA Sequence Matching and Database Search"

Committee

Professor Apostolos Dollas (supervisor)

Associate Professor Dionisios Pnevmatikatos

Assistant Professor Ioannis Papaefstathiou

Acknowledgments

First of all, I would like to thank my family for their constant support not only during my thesis, but also during the entire period of my undergraduate studies.

In addition, I would like to thank my supervisor Prof. Apostolos Dollas not only because he gave me the opportunity to gain valuable knowledge and experience of his research area, but also because his remarkable advice is fundamental in my future as professional in computer engineering.

I would also like to thank Euripides Sotiriades, PhD student under Prof. Apostolos Dollas, whose constant support and creative collaboration played a dominant role in the successful development of this thesis.

I would also like to thank Prof. Dionisios Pnevmatikatos and Prof. Ioannis Papaefstathiou for their support during the sabbatical absence of my supervisor.

I would also like to aknowledge the support of Markos Kimionis in technical maters.

Finally I would like to thank my friends and collaborators, either graduate or undergraduate students, of Microprocessor and Hardware Lab for their support and advice during the development of this thesis.

Contents

Chapter 5 - The first generation of TUC-BLAST

Chapter 6 - The second generation of TUC-BLAST

Chapter 7 - Performance Comparisons

Chapter 1 - Introduction

1.1 Brief Description of the Needs of Molecular Biology

After the discovery of the structure of DNA by Crick and Watson in 1953, molecular biology has evolved tremendously. Being able to manipulate bio-molecular sequences, molecular biologists have produced and continue to produce large amounts of data. Scientists from other biological fields process this data in order to provide further useful information. However, the huge size and the complexity of such data make their use extremely difficult without the help of disciplines such as computer sciences and mathematics. This need has created a new scientific field named as *computational molecular biology*.

In [1] there is an introductory view of the main problems Computational Biology deals with. One of these problems is the Sequence Matching problem where one or more strings containing biological data are compared against a database or part of it, in order to find similar regions

To address the Sequence Matching problem a variety of algorithms have been introduced, which are divided into two categories. One category contains the algorithms which use dynamic programming and provide the highest accurate answers. The most known algorithms of this category are the Smith-Waterman [19] and the Needleman-Wunch [18]. The other category contains algorithms that make use of heuristics to find the answer. Although these answers are not as accurate as the first ones, these algorithms are preferable because they are faster. This category contains the BLAST[21] and the FAST[20] algorithm.

Since the early 1990's hardware designers have tried to provide architectures based on reconfigurable logic in order to enhance the Molecular Sequence matching algorithms. Most of the previous reconfigurable platforms for boosting DNA Sequence Matching and Database Search have primarily used the dynamic algorithms as these algorithms provide high parallelization which can be effectively exploited by a special purpose processor. On the other hand, only a few hardware approaches exist

using the heuristic algorithms. What is more, either these architectures do not provide a good speed up in comparison with the special purpose processors or the used heuristic algorithms are changed to a degree in order for the special purpose architectures to achieve high speed ups. The reason for which hardware implementations of the heuristic algorithms has been difficult is their I/O problem as well as the fact that these algorithms are not obviously parallelized.

In the Microprocessor and Hardware Lab together with Prof. Apostolos Dollas and PhD candidate Euripides Sotiriades, we use state of the art FPGA technology in order to develop a special purpose architecture enhancing significantly the performance of the BLAST algorithm. The purpose of this thesis, which is part of the afore-mentioned project, is the implementation and evaluation of the first generation of the architecture as well as the design and implementation of a part of the second generation. In the first generation Mr. Sotiriades introduces the use of the FPGA's embedded memory blocks as the matching scheme while he proposes the use of the embedded transceivers to overcome the I/O problem. In the second generation, we use the memories more effectively in order to increase the speed up and the reliability of the system while the embedded PowerPC is introduced to perform the non-computationally intensive parts of the algorithm.

1.2 Contribution of the current thesis

The contribution of this thesis is the following:

- Software implementation of a BLAST machine for a better understanding of the algorithm. This implementation also serves as the verification and profiling tool of the hardware implementation.
- VHDL coding and synthesis with Xilinx Ise 7.1 tools and post place and route simulation and verification with Modelsim 6.0 of the first generation of the architecture.
- Evaluation of the first generation of the system in terms of a reliable performance.
- Development, simulation and verification of the sequence matching part of the algorithm in the second generation.
- Evaluation of results

1.3 Thesis Overview

In chapter 2 the sequence comparison problem is described and the algorithms dealing with it are discussed. Then chapter 2 introduces the database search problem by presenting the different databases worldwide containing huge amounts of biological data.

Chapter 3 contains a brief overview of the hardware acceleration endeavors that exist in literature proposed to boost the performances of the algorithms shown in Chapter2. In this chapter, we additionally it is also discussed the reason for targetting a hardware implementation of the BLAST algorithm.

The contents of chapter 4 deal with BLAST software. After distinguishing the different software programs according to the form of the processed data, the blastn program of the popular NCBI software is presented. Finally in this chapter, we present the implementation of a software system, we developed from scratch, running the main blastn machine

Chapter 5 includes the first generation of the blastn hardware implementation. After a brief presentation of the architecture of this version, whose development was not within the scope of the current thesis, the implementation of this architecture in VHDL is described in detail. Afterwards, the performance of this architecture is compared against other known systems and its vulnerabilities are discussed.

Chapter 6 deals with the second generation of this architecture. There is a novel design of only one part of the BLAST algorithm, taking into account that the rest of the algorithm is executed by a special purpose processor. After the generation of multiple machines and verification of this part of the system, this generation is compared with its predecessor.

Finally chapter 7 describes all the performance comparisons between the second generation and other general or special purpose systems, while chapter 8 proposes future work to further evolve this system in order to increase its performance and its reliability.

Chapter 2 - Sequence Matching in Computational Biology

2.1 The Problem of Sequence Comparison

Sequence Comparison is one of the most fundamental operations in Computational Biology, as it is the basic primitive for performing a great variety of more complicated processes. Although it seems to be a simple procedure, there are a lot of distinct problems requiring totally different algorithms to provide efficient solutions. In [1] five categories of such problems are identified using Sequence Comparison, one of which involves the problem according to which a sequence consisting of some thousands of characters must be compared to thousands of others, so that similar substrings between the given sequence and any other sequence should be identified. Such problems occur when biologists search for local similarities against large biosequence databases.

Before diving into the rest of this work, it is essential that the most common terms should be explained. In the following of this section, terms from three different disciplines are discussed. First the fundamental biological terms are explained, based on [1]. The most common terms of the Language Theory, a hardware designer might ignores, are briefly introduced based on [4]. Finally, terms of Sequence Comparison are presented as in [1].

2.1.1 Biological Terms

Of course, the first thing we should explain is **DNA**. DNA is abbreviation of *deoxyribonucleic acid* and is one of the two kinds of nucleic acids contained in living organisms (The other one is the ribonucleic acid, abbreviated by RNA, but we will not deal with it in this work). DNA is a double chain of simpler molecules. The basic unit of each chain, which is also called strand, is the **nucleotide**, which consists of three

smaller molecules: a *sugar*, a *phosphate* and a *base*. Without adding more details about the structural molecules of a nucleotide, the sugar and the phosphate are the same in every nucleotide whereas there are four different kinds of bases: Adenine (A), Guanine (G), Thymine(T) and Cytosine(C). So, although it is clear that nucleotides are different than bases is is can be said that a *DNA molecule has 200 nucleotides or 200 bases*. But which is the relation of the nucleotides of the same positions in the double chain? Each base in one strand bonds to a base in the other strand. Base A is always paired with base T and base C is always paired with base G. So, it is clear that talking about the one strand of a DNA molecule is enough to describe the entire double chain molecule. As a result, DNA molecules can be easily represented as strings of the letters A, T, C, G, such as *ATTCGGGACT*.

Another important biological term is the **protein**. Human body mostly consists of proteins, of which there are different kinds. Some form tissue blocks, while others, known as enzymes, are useful in speeding up chemical reactions. A protein is a single chain of simpler molecules, called amino acids. There are 20 different amino acids which are shown in figure 2.1 with their one-letter and three-letter representation. Protein chains can be also mapped as strings of the twenty letters of the one letter codes of figure 2.1, such as *NTEFFFATTLKFM*.

One letter code	Three-letter code	Name
A	Ala	Alanine
C	Cys	Cysteine
D	Asp	Aspartic Acid
E	Glu	Glutamic Acid
F	Phe	Phenylalanine
G	Gly	Glycine
H	His	Histidine
I	Ile	Isoleucine
K	Lys	Lysine
L	Leu	Leucine
M	Met	Methionine
N	Asn	Asparagine
P	Pro	Proline
Q	Gln	Glutamine
R	Arg	Arginine
S	Ser	Serine
T	Thr	Threonine
V	Val	Valine
W	Trp	Tryptophan
Y	Tyr	Tyrosine

Figure 2.1 The twenty amino acids forming the proteins (from [48])

2.1.2 String Manipulation Terms

The term **alphabet** denotes any finite set of symbols. It is usually denoted by the letter Σ with usually a subscript. ASCII is an example of computer alphabet while the set {0,1} is the binary alphabet. In sequence comparison two alphabets are found. $\Sigma_{nucleotide}$ consists of the following four letters: {A, T, C, G} representing the four DNA bases. The other alphabet is the Σ_{amino} which consists of twenty letters representing the twenty amino acids, the basic elements of proteins.

The term **sequence** over some alphabet denotes a *finite set of symbols drawn from that alphabet*. This term is used as synonym for the term *string*. The **length** of a

string s, usually written |s|, is the number of occurrences of symbols in s. For example, *ATTCGGGACT* is a sequence of length ten.

Prefix of a string s is *a string obtained by removing zero or more trailing symbols of string s*. For example ATT is a prefix of *ATTCGGGACT*.

Suffix of a string s is *a string formed by deleting zero or more of the leading symbols of s*. For example, GACT is a suffix of *ATTCGGGACT*.

Substring of a string s is a string obtained by deleting a prefix and a suffix of s. Obviously every prefix and every suffix of a string is also substring of that string. For example, ATT, GACT, CGG are substrings of string *ATTCGGGACT*.

Subsequence of a string s is any string formed by deleting zero or more not necessarily contiguous symbols from s. For example, TCA is a subsequence of A*TTCGGGACT*.

2.1.3 Sequence Comparison terms

Alignment of two sequences *is the insertion of spaces in arbitrary locations along the strings so that they end up with the same size*. *Having the same size, the augmented sequences can now be placed one over the other, creating a correspondence between characters or spaces in the first sequence and characters or spaces in the second sequence*. In other words, the augmented strings are placed in such a way that every character in either sequence is placed against a unique character or a unique space and every space is placed opposite a unique character. For example, for the DNA sequences *GACGGATTAG* and *GATCGGAATAG*, a possible alignment is shown in figure 2.2, another one is in figure 2.3 and a third one is in figure 2.4.

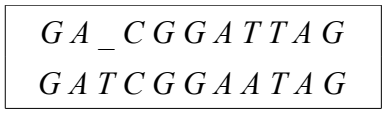

Figure 2.2 A possible alignment

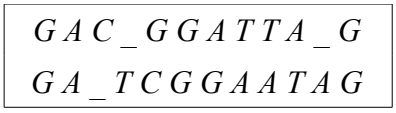

Figure 2.3 Another possible alignment

Figure 2.4 A third possible alignment

Given an alignment, a **score** can be assigned to it as follows. Every column of the alignment receives a score depending on its contents. The total score of the alignment is the sum of the values of all the columns of the alignment. For example, if a match between two characters receives 1, a mismatch between a pair of characters receives -1 and an existence of a gap in a pair of letters receives -2, the score of the alignments of figures (2.2, 2.3 and 2.4) are respectively the following:

$$1+1+(-2)+1+1+1+1+(-1)+1+1+1=6$$
$$1+1+(-2)+(-2)+(-1)+1+(-1)+(-1)+(-1)+(-1)+(-2)+1=-7$$
$$(-2)+(-2)+(-2)+(-2)+(-2)+1+1+(-1)+(-1)+(-1)+(-1)$$
$$+(-2)+(-2)+(-2)+(-2)=-20$$

The best alignment is the one with the maximum score. *This maximum total score of all possible alignments is called* ***similarity*** *between the two sequences*. This score is also called as optimal global alignment score. Generally, there could be multiple alignments of two sequences giving maximum total score.

In our previous example, the alignment between sequences *GACGGATTAG* and *GATCGGAATAG* giving maximum score is the one of figure 2.2 and the similarity of these sequences is six.

In the previous example it has been noted that each match receives 1, each mismatch receives -1 and each gap existence receives -2. These values are taken by a **scoring matrix** where all the pairwise scores between the characters of an alphabet are defined. Table 2.1 shows the scoring matrix of this example. This scoring matrix, as well most of the commonly used ones(PAM[22], BLOSUM[23]), emphasize matches by assigning positive values in matches, whereas they penalize mismatches and gap insertions by assigning zero or negative scores. Consequently, the alignment with as many matches as possible has the greatest possible score.

score	*A*	*T*	*C*	*G*	_
A	1	-1	-1	-1	-2
T	-1	1	-1	-1	-2
C	-1	-1	1	-1	-2
G	-1	-1	-1	1	-2
_	-2	-2	-2	-2	<none>

Table 2.1 The scoring matrix of the example

2.2 Global Comparison

The most obvious approach to computing the optimal global score between two sequences, could be the generation of all the possible alignments between these sequences and finally choosing the ones giving the greatest score. However, such an approach could produce a too slow algorithm as the number of the alignments grows exponentially with the length of the involved sequences. The solution to this problem came in 1970 by Needleman and Wunch in [18], where they involved dynamic programming in their proposed algorithm. Instead of determining the similarity of two sequences as a whole, the solution is built up by the similarities between arbitrary prefixes of the two sequences, starting with the ones with the smaller length and continuing with the larger prefixes.

The Needleman Wunch (NW)algorithm involves a two dimensional matrix, whose dimensions represent the two sequences to be aligned. So, all the possible pairs of nucleotides or amino-acids are represented as records of this table. Reference [16] shows an example of the algorithm, with the comparison of amino-acid sequence *MPRCLCQRJNCBA* with sequence *PBRCKCRNJCJA.* The NW algorithm has to fill the matrix of table 2.2 where each column represents a letter of the former sequence and each row represents a letter of the latter one. All the possible alignments are represented as pathways through this matrix.

	M	P	R	C	L	C	Q	R	J	N	C	B	A
P													
B													
R													
C													
K													
C													
R													
N													
J													
C													
J													
A													

Table 2.2 The table which has to be filled by the NW algorithm

The NW algorithm involves three steps. In the first step the matrix is initialized in such a way that every record has a value indicated by the score of the respective letters. For example, if every match receives 1 and every mismatch and gap receives 0, the entries of the matrix initialized with one are shown in table 2.3, whereas the rest are initialized with zeros.

	M	P	R	C	L	C	Q	R	J	N	C	B	A
P		1											
B												1	
R			1					1					
C				1		1					1		
K													
C				1		1					1		
R								1					
N										1			
J									1				
C				1		1					1		
J									1				
A													1

Table 2.3 The step1 of the NW algorithm (from [16])

In the second step of the algorithm the values of the matrix are updated according to the following pseudocode.

$$for\ i=1 \rightarrow |sequence_{horizontal}|$$
$$for\ j=1 \rightarrow |sequence_{vertical}|$$
$$A_{(i,j)} = S(i,j) + max\left\{A_{(i-1,j-1)}, A_{(i,j-1)}+d, A_{(i-1,j)}+d\right\}$$

where $A_{(i,j)}$ is the value of the (i,j) position of the matrix, S(i,j) is the score between the letters involved in the certain position and d is the score of a gap penalty (in our example it is zero).

An instance of this procedure is shown in table 2.4

	M	P	R	C	L	C	Q	R	J	N	C	B	A
P	0	1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1	1	1	1	2	1
R	0	0	2	1	1	1	1	2	1	1	1	1	2
C	0	0	1	3	2	3	2	2	2	2	3	2	2
K	0	0	1	2	3	3	3	3	3	3	3	3	3
C	0	0	1	3	3	4	3	3	3	3	4	3	3
R	0	0	2	2	3	3	4	?					
N													
J													
C													
J													
A													

Table 2.4 An instance of the step2 of NW algorithm of our example

For a better understanding of this procedure, let us compute the entry with the symbol '?', i.e. the value $A_{7,6}$. As it is clear from the table, there is a match between R characters. So the score S(7,6) is 1. Looking at the neighboring matrix entries, we have $A_{6,6}$=3, $A_{6,5}$=3 and $A_{7,5}$=4. As the gap penalty is 0, it is clear that the missing value is 5.

In the final step of the algorithm, when all the elements of the matrix have been assigned a value, there is a backtracking beginning from the highest scoring entry and ending back to the position (0,0) in order for the alignment producing the optimal global score being identified. In our example, table 2.5 has all the possible paths resulting from this backtracking, where every line has a direction from bottom to top and from right to left. If the sequence providing the i indexes is called s and the one providing the columns is called t, the alignments are found according to the following

algorithm. If a line leaving the entry (i,j) is horizontal, it corresponds to a gap in s matched with t[j]. If it is vertical, it corresponds to s[i] matching with a gap in t. Finally a diagonal arrow means s[i] paired with t[j]. Figure 2.5 shows one of the optimal global alignments of this example.

	M	P	R	C	L	C	Q	R	J	N	C	B	A
P	0	1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1	1	1	1	2	1
R	0	0	2	1	1	1	1	2	1	1	1	1	2
C	0	0	1	3	2	3	2	2	2	2	3	2	2
K	0	0	1	2	3	3	3	3	3	3	3	3	3
C	0	0	1	3	3	4	3	3	3	3	4	3	3
R	0	0	2	2	3	3	4	5	4	4	4	4	4
N	0	0	1	2	3	3	4	4	5	6	5	5	5
J	0	0	1	2	3	3	4	4	6	5	6	6	6
C	0	0	1	3	3	4	4	4	5	6	7	6	6
J	0	0	1	2	3	3	4	4	6	6	6	7	7
A	0	0	1	2	3	3	4	4	5	6	6	7	8

Table 2.5: Backtracking the paths

```
MP-RCLCQR-JNCBA
  | ||  |  |  |  |
-PBRCKC-RNJ-CJA
```

figure 2.5 An alignment produced by the NW algorithm on our example

It is obvious that both the time and space complexities of the algorithm computing the optimal global score of the alignment of sequences s and t with lengths m and n respectively is O(mn) as the worst case is the filling of the entire matrix. The time complexity of the backtracking algorithm is O(m+n), given the already filled matrix.

2.3 Local Comparison

According to Molecular Biologists, as it is discussed in [48], *duplications and modifications are common in proteins and DNA sequences*. In other words, there are

quite similar areas in the genome not only of an organism, but among different organisms. So, it is quite possible that two biosequences with poor global alignment, might have strong local similarities. In such cases, the use of the NW algorithm may give misleading results because only small regions of the sequences are related. Usually, it is **local alignment** which is the most appropriate method for comparing proteins of different families. In local alignment, the target is to find a pair of substrings, one of each of the compared sequences, which exhibit high similarity. In other words, the main task is to find an alignment that begins and ends at any position of the two sequences. As variety of algorithms have been developed performing exact local alignment, the most known of which is the one developed by Smith and Waterman. Later, other families of algorithms have been developed, such as FAST and BLAST, which use heuristics to find near optimal local alignments.

2.3.1 The Smith-Waterman algorithm

In 1981 T. Smith and M. Waterman proposed their algorithm for performing local alignment in [19], using dynamic programming. In fact, their algorithm is a variation of the Needleman-Wunch algorithm described in the previous section. The main difference of the NW algorithm is the fact that there are not any negative cells in the matrix of the algorithm[14]. On the contrary, cells giving negative values are set to zero. In this way, the local alignments are rendered visible. Another difference between the two algorithms is the fact Smith-Waterman (SW) algorithm requires that there is a negative gap penalty to work effectively, whereas the performance of NW was not affected by the absence of gap penalty. The backtracking starts from the matrix entry with the highest score, and proceeds until a score with zero value is encountered, *yielding the highest scoring local alignment*.

As with the NW algorithm, the space and time complexities are also quadratic O(mn), making its use difficult in real applications.

2.4 Heuristic Algorithms

Although the aforementioned algorithms produce optimal alignments, their quadratic complexities make their use difficult in database search, where the database files consist of billions of residues (i.e. nucleotides or amino acids) and the query sequences may involve thousands of hundreds of residues. In order to speed the database searching, faster methods have been introduced using heuristics. In this section the two most popular algorithms will be presented, the BLAST and the FAST.

2.4.1 BLAST algorithm

BLAST is the acronym of Basic Local Alignment Search Tool and it was firstly presented in [21] by S.F.Altchul et. al. in 1990. In order to remain consistent with the terminology found in the original paper, it is important for us to describe the basic terms.

A **segment** is a substring of a sequence. Given two sequences, a **segment pair** is a pair of substrings of the same length, one of each sequence. The subsequences of a segment pair can be **gaplessly** aligned as there are of the same length.

Given a scoring scheme such as PAM120 for protein sequences or in case of DNA sequences a +5 for every match and a penalty of -4 for every mismatch, a **Maximal Segment Pair (MSP)** is defined to be the highest scoring pair of identical length segments chosen from two sequences. An MSP may be of any length as its boundaries are chosen to maximize its score. This score provides a measure of local similarity for any pair of sequences. However, as *a molecular biologist may be interested in all conserved regions shared by two proteins, not only in their highest scoring pair*, a segment pair is defined to be **locally maximal** if its score cannot be improved either by extending or by shortening both segments. *BLAST can seek all locally maximal segment pairs with scores above some cutoff*[21].

We now have all the necessary information to describe precisely the performance of BLAST. Given a query sequence, BLAST returns all the segment pairs between the query and the database sequence with scores above a certain score

S. Most servers running the BLAST software provide a default value of S, but also a user may also define a value for S.

BLAST algorithm consists of 3 steps whose implementation depends on the form of the data processed, nucleotide sequences or amino acid sequences. In the following discussion, BLAST dealing with nucleotide data will be discussed in detail, whereas shorter explanations will be given regarding the manipulation of amino acid data.

The first step of the algorithm involves the compiling of the list of high scoring words. For DNA sequences this list contains all contiguous w-mers, i.e. words of length w, in the query sequence. For nucleotide sequences, the value of w is usually 12 and a typical range of this value is between 11 and 15. Obviously this list will contain n-w+1 w-mers where n is the length of the query sequence. To better illustrate the algorithm steps for DNA sequences, we will use a smaller value for w in our examples. Let *ACGTAAATGCAG* be the query sequence of length 12 and let w be equal to 3. The word list will contain 10 w-mers. As it is shown in figure 2.6, *ACG* will be the first one, CGT the second, GTA the third etc. and CAG will be the last one.

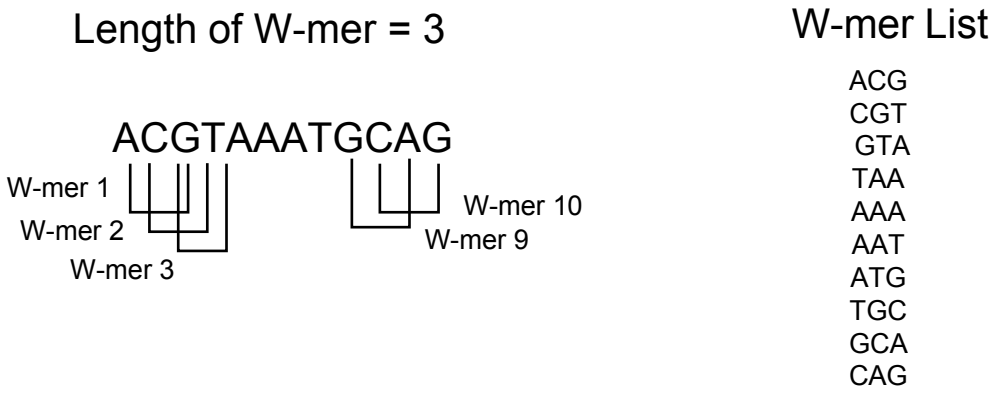

Figure 2.6 Step1 of BLAST (from [40])

For queries with protein sequences containing all words which score at least T when compared with some word in the query sequence. So, a query may be represented by no words in the list or by many.

The *Second step* is the search of the database for "hits". After the word list generation, the database sequences are searched for an **exact** match between any substring of the W-mer list and the database sequence. Every word of the word list found in the database is called hit and it is possible to be part of a High Scoring Pair (HSP). Figure 2.7 shows an instance of the execution of step2, when the incoming database stream matches with a word of the hit list.

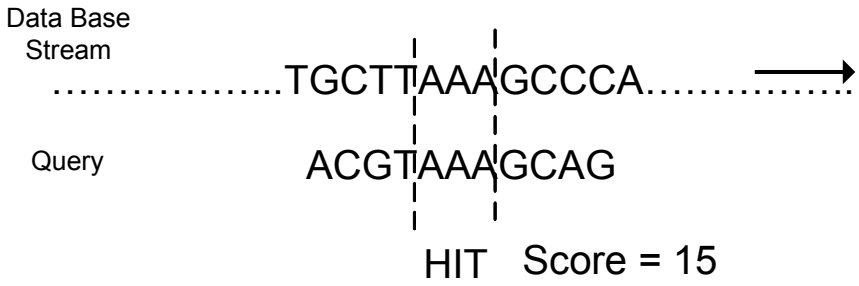

Figure 2.7: **Step2 of BLAST**

This step is different between DNA and the protein data regarding its programming implementation. Although the search for hits in protein data is complicated as the two methods used involved use either large hash tables or Deterministic Finite Automata (DFA), the hit finding for DNA data is easier because a programmer may take advantage of the small alphabet length (four letters may be represented by two bits) so that four nucleotides fit in a byte. Apart from the obvious space benefits, the time may also be reduced as if a byte is compared each time. In the original BLAST paper [21] it is also reported an extra filtering step removing from the initial list very common words from the database to avoid hits with little interest.

As soon as a hit is identified, in a straightforward process, not differing in case of nucleotide or amino acid data, it is extended by the step3 of BLAST for finding a locally MSP. In the original BLAST paper it is stated that for timing reasons the process of extending in one direction is terminated when *a segment pair whose score falls below a certain distance below the best score found for shorter extensions* is reached. According to this paper, the added inaccuracy is negligible.

Figure 2.8 shows step by step the extension of the hit found in figure 2.7. In the first iteration of the extension process there are matches in both extension directions, so the score increases by 10. In the second and third iterations there is a match only in the one extension direction so the score in both iterations is increased by one, as each

match yields 5 and each mismatch is penalized with -4. In the fourth iteration, there are mismatches in both directions and the score should be decreased by 8. As the score decreases in this iteration, the extension process stops without taking into account the last iteration.

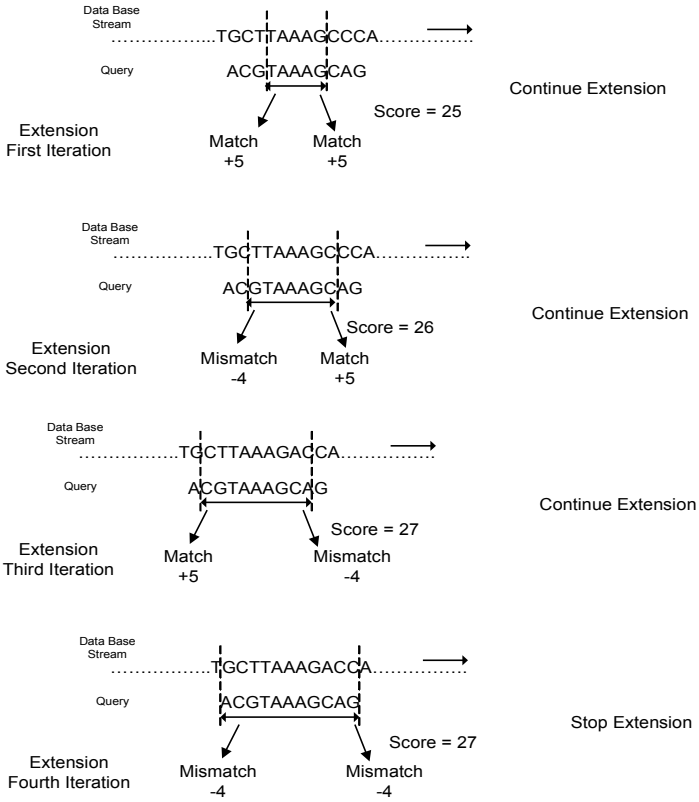

Figure 2.8 Step3 of the BLAST algorithm

BLAST tool is based on a strong statistics background which will be briefly explained as in [1]. The distribution of the MSP score for random sequences s, t with respective lengths m, n can be accurately approximated as follows.

Given a matrix of replacement costs $s_{i,j}$ for the pairs of characters in the alphabet, and the probability p_i of occurrence of each individual character in the sequences, the lamda λ value may be obtained by solving the equation.

$$\sum_i p_i p_j e^{\lambda s_{ij}} = 1$$

The parameter λ is the unique positive solution of this equation and may be obtained by numeric methods. Once λ is known, the expected number of distinct Segment Pairs between s and t with score above S is $Kmne^{-\lambda S}$, where K is a

constant. Actually this distribution is a Poisson distribution with a mean given by the above formula. From this, it is easy to derive expressions for useful quantities such as the large score.

2.4.2 FASTA algorithm

FAST family includes a set of tools performing local alignment of Biosequences. The first program of the family, the FASTP, introduced in 1985 in and it was designed for protein sequence similarity searching. FASTA is the most widely known tool of the family which was first presented in 1988 in [20] and it added to its predecessor the ability to perform searches between DNA sequences, translated protein with DNA sequences and provided a more sophisticated program for evaluating statistical significance, as it is described in [14].

Reference [17] has a brief description of this algorithm. According to it, FASTA is a two step algorithm. In the first step, a search for regions with high similarities takes place. *In this search a word with a specific word size is used to find regions in a two-dimensional matrix similar to that shown for the Smith-Waterman algorithm*. These substrings are located in the diagonal or in a few neighboring diagonals of the table which have a high number of identical word matches between the sequences. The second step of FASTA involves a Smith-Waterman alignment on regions of the aforementioned diagonals. These regions are bounded by a **window size** which limits the number of insertions or deletions one sequence may have with respect to the other sequence in the alignment. This is the main point of FASTA algorithm which makes it significantly faster than the SW algorithm. In other words, due to the prior restriction of the alignment space, the SW alignment will take place only in the necessary regions and not in the entire O(mn) space, not to mention that the SW step can be omitted when no regions are found with high similarities.

However, the weakness of the algorithm is also the window size, as it is explained in the following examples of [17]. *The first example would have two proteins that share 50% identity - but the proper alignment consists of alternating match and mismatches. With a word size of two, there would be no word matches along the main diagonal of the dot plot for the sequences (although there will*

potentially be random or spurious word matches on the off-diagonals) and the proper alignment would probably not be found. The second case consists of two proteins that are almost identical, except the second protein has a 20 residue insertion into the middle of the sequence. If the window size is 15, then the Smith-Waterman alignment phase of FASTA will not have enough alignment space to jump the 20 residue insertion. Thus, the resulting alignment will be either the sequence prior to or after the insertion (whichever had the higher diagonal scores) and the fact that the proteins were basically identical (with only one long insertion) will be missed.

2.5 Search in Genetic Databases

As it has been stated in the introduction of the sequence matching problem in the first section of this chapter, Database Search is one of the major problems involving string matching, as query sequences usually have to compared against large Databases containing billions of residues, i.e. nucleotides or amino acids.

At this point, it should be clarified that the term *database* simply refers to a large set of recorded sequences. Other known features from Database systems, such as fast access and data sharing, are excluded from biological databases. Usually, biological databases are found in FASTA format, a readable format. FASTA files consist of sequences of letters from the nucleotide or amino acid alphabet. Different sequences are separated by comment lines denoting the description of each sequence. These comment lines start with symbol '>'. In the following of this thesis, biological databases are considered to be in FASTA format, unless otherwise stated.

Biological Sequence databases are divided in two categories, depending on the type of data stored. So, there are DNA and amino acid sequence databases. **GenBank** is the National Institutes of Health (NIH) genetic sequence database, an annotated collection of all publicly available DNA sequences There are approximately 59,750,386,305 bases in 54,584,635 sequence records in the traditional GenBank divisions and 63,183,065,091 bases in 12,465,546 sequence records in the Whole Genome Shotgun (WGS) division as of February 2006 [24]. In other words, in this release there is a total of 122,933,451,396 bases in 67,050,181 nucleotide sequences,

while GenBank releases a new version every two months. Another major database is located in Europe. Through the European Bioinformatics Institute (EBI) European Molecular Biology Lab (EMBL) [10] is the central European repository and supplier of biology data services for both the academic and industrial R&D biosciences communities. The **EMBL Nucleotide Sequence Database** (also known as EMBL-Bank)[11] constitutes Europe's primary nucleotide sequence resource. Main sources for DNA and RNA sequences are direct submissions from individual researchers, genome sequencing projects and patent applications. The core databases are either unique or represent Europe's contributions to world-wide, irreplaceable and coordinated information resources for the life sciences community. According to the latest release of the database in March 2006 [25], there is a total of 126,401,347,060 DNA bases contained in 69,783,593 entries. Japan has also its Data Bank which collaborates with the two other Data Banks described above through exchanging data and information on Internet. DNA Data Bank of Japan(**DDBJ**)[9] began DNA data bank activities in 1986 at the Japanese National Institute of Genetics (NIG). In its latest release of March 2006, it announces a total of 60,564,721,635 DNA bases in 55,890,995 entries with an increase of 7.9 % from the previous release.

As far as it concerns protein data, Universal Protein Resource(**UniProt**)[12] is the world's most comprehensive catalog of information on proteins. It is a central repository of protein sequence and function created by joining the information contained in Swiss-Prot, TrEMBL, and PIR. UniProt Knowledgebase (**UniProtKB**)[26] is the central hub for the collection of functional information on proteins, with accurate, consistent, and rich annotation. In addition to capturing the core data mandatory for each UniProtKB entry (principally, the amino acid sequence, protein name or description, taxonomic data and citation information), as much annotation information as possible is added. This includes widely accepted biological ontologies, classifications and cross-references, and clear indications of the quality of annotation in the form of evidence attribution of experimental and computational data. Created by merging the data in Swiss-Prot, TrEMBL and PIR-PSD, individual UniProt Knowledgebase entries may contain more information than was available in any given separate source database. The UniProt Knowledgebase consists of two sections: a section containing manually-annotated records with information extracted from literature and curator-evaluated computational analysis, and a section with

computationally analyzed records that await full manual annotation. For the sake of continuity and name recognition, the two sections are referred to as "Swiss-Prot" and "TrEMBL", respectively.

2.6 Summary

In this chapter an introduction to the problem of Sequence Matching in Molecular Biology has been presented and the most useful terms have been introduced divided in three categories: biological terms, terms of Language Theory and terms of Sequence Comparison. Afterwards the dynamic algorithm performing global alignment has been analytically described. However, as it has been analyzed, in many cases local alignment is the preferable method for sequence matching. Although the dynamic algorithm shown for local alignment provides optimal alignments, its quadratic complexity makes it difficult for database search. So, other local alignment algorithms have been introduced, which use heuristics and although they provide near optimal alignments, their strong statistical theory strengthens their results. Such algorithms is FASTA and its successor BLAST. As BLAST is more popular than FASTA, due to the accuracy of its results and its stronger statistical theory, it has been presented in more detail than FASTA. Finally, the problem of database search has been introduced because there is a large amount of data contained in databases worldwide which need to be effectively searched in a reasonable time period.

Chapter 3 - Hardware efforts in Sequence Matching

As it has been presented at the end of the previous chapter, the most common biological databases contain currently more than a hundred billion DNA bases in their entries, while they grow exponentially as it is clearly shown in figure 3.1 published in the GenBank website. Although data shown in this figure have not been updated since August 2005, they show a representative view of this exponentially growth.

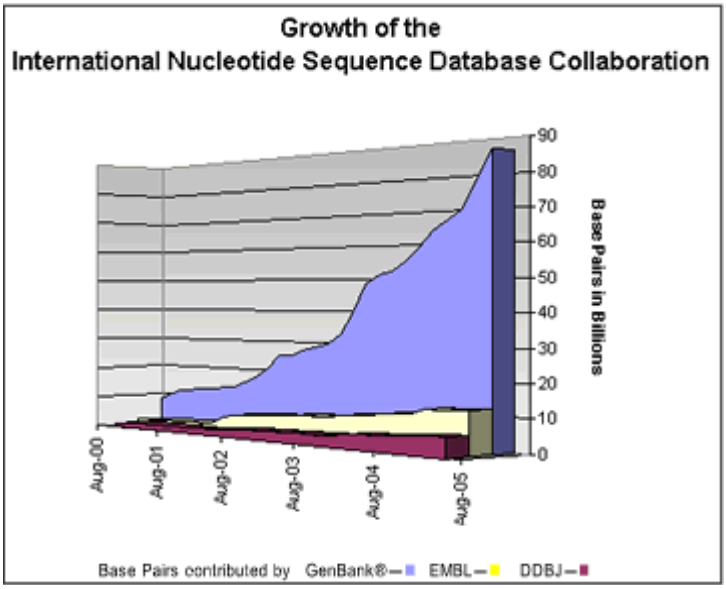

Figure 3.1: A comparison of the growth of each DB (from [8])

Furthermore, in [2] there is a comparison of the GenBank's rate of growth with the one of number of transistors in a personal computer chip. In figure 3.2 there is a curve showing the growth of the GenBank in thousands of base pairs against the time, until the year 2002. Noticing that the vertical axis is in logarithmic scale, the straight lines of the database growth show the database exponential growth. As it is calculated in this book, the GenBank doubles in size every 1.4 years. In figure 3.2 there is another famous increasing curve that goes by the name of the Moore's Law. As it is

shown in the curve, the chip size doubles every two years. This rate is substantially slower than the rate of increase of GenBank.

From the curves of figures 3.1 and 3.2 it is clear that, although current general purpose computing systems are still able to perform biological database searching, no matter how much time may be required for a search in billions of DNA bases, in the future they may not be capable for searching the entire GenBank due to hardware limitations. So, specific purpose architectures should be developed enhancing the database search. In this chapter, an overview of hardware designs speeding up the performance of the sequence comparison algorithms are presented. Almost all of these hardware

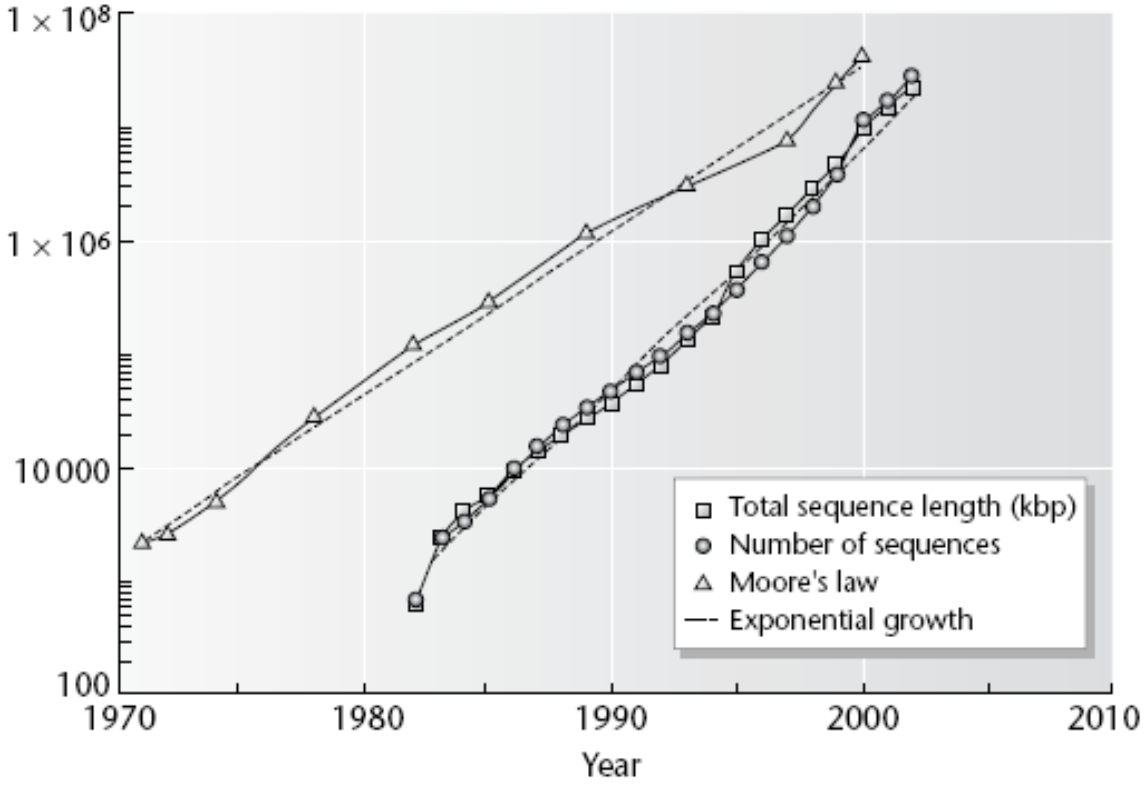

Figure 3.2 Comparison of the rate of growth of GenBank with the rate of growth of the number of transistors in personal computer chips (from [2])

implementations use FPGAs as the latter have been usually shown to accelerate a variety of applications from various disciplines such as in [28]

3.1 Implementations of Needleman and C. D. Wunsch*

As global alignment is not of common use for the reasons discussed in chapter 2, Needleman-Wunch algorithm is not expected to have been implemented in hardware. In our literature only one hardware implementation of this algorithm exist. In 1993 Hoang in [30] developed two systolic arrays, a unidirectional and a bidirectional one, computing the editing distance between two genetic sequences using the NW algorithm. Figure 3.3 shows the dataflow through the bidirectional systolic array.

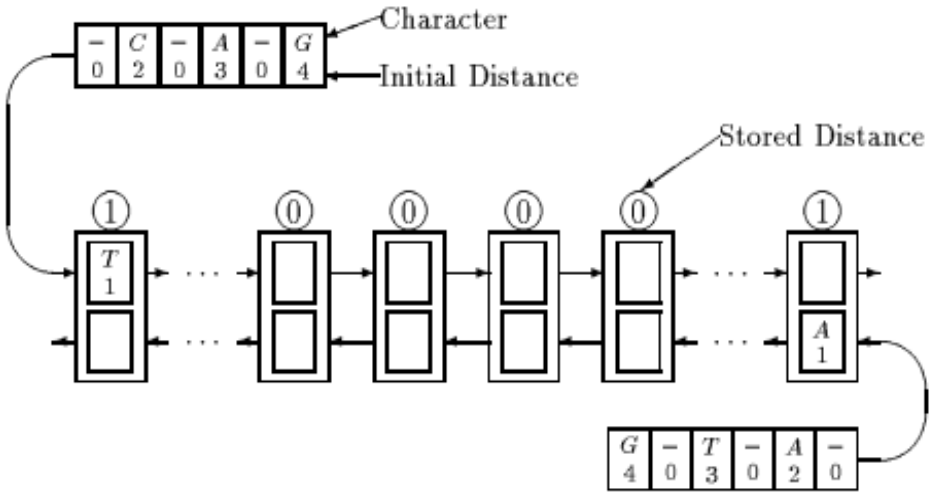

Figure 3.3 The dataflow in Hoang's bidirectional systolic array(from[30])

In the bidirectional systolic array, each processing element (PE) computes the values along a particular diagonal of the NW scoring matrix. The source and target sequences enter the array on opposite ends and flow at opposing directions at the same speed. At each cycle, the contents of the streams represent the characters to be compared and the scores along one of the anti-diagonals. At the end of computation, the resulting edit distance is extracted out of the array. For the comparison of two sequences with length m, n respectively they are required at least 2*max(m+1, n+1) processors and the time required for the computing of the maximum score is proportional to the length of the array.

In the unidirectional array of figure 3.4 the source sequence is loaded once and stored in the array starting from the leftmost Processing Element (PE) while the target sequence is streamed through the array. In this configuration, every PE computes the distances in one row of the algorithm's matrix. At each time step PEs compute the values along an anti-diagonal of the NW matrix. A unidirectional array of length n can compare a source sequence up to n characters with a target sequence with m characters in O(m+n) steps. This property makes this array more effective in the database search as there is not any constraint regarding the length of the target sequence.

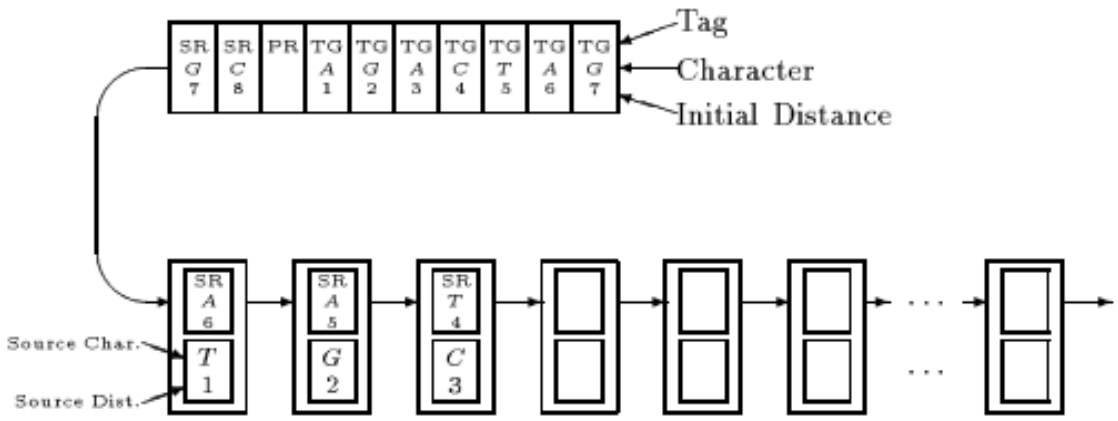

Figure 3.4 The dataflow in Hoang's unidirectional systolic array(from[30])

In this work, both systolic arrays mapped in the SPLASH2 system [44] consisting of an interface board containing two Xilinx 4010 FPGAs and up to 16 programmable logic blocks each containing 17 Xilinx 4010 FPGAs.

Reference[30] states that these architectures did not actually implemented but the performance evaluation was based on the results given by CAD tools. Table 3.2 shows the results taken by benchmarking these arrays on Splash2 system. The quantity CUPS of this table denotes the number of the scoring table entries updated per second. They are calculated by a run consisting of 1000 repetitions of a 1000 x 1000 comparison.

Specifics	*CUPS*
Unidirectional; 16 boards	43,000M
Bidirectional; 16 boards	34,000M
Unidirectional; 1 board	3,000M
Bidirectional; 1 board	2,100M

Table 3.1 Performance Comparison of the systolic arrays implementation mapped on Splash2(based on [30])

3.2 Implementations of Smith Waterman

Since mid 1990's a large number of architectures accelerating the SW algorithm has been reported [31]-[35]. Although it is difficult for one to present every one, two interesting approaches have been selected for a short description.

In Hokiegene [34] runtime reconfiguration is used for initializing the parameters of the processing elements of the architecture's systolic arrays. The computational core of the SW algorithm has been represented in hardware in figure 3.5.

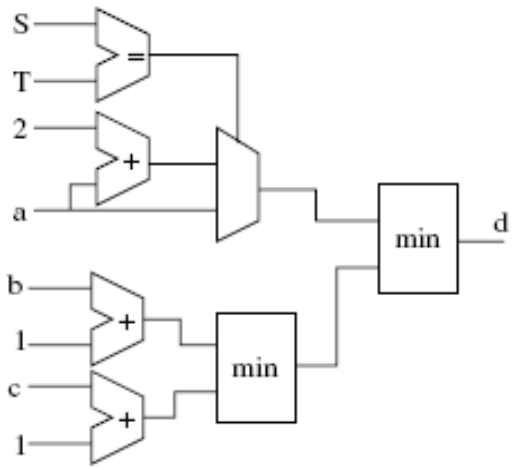

Figure 3.5 Smith Waterman algorithm implemented in hardware (from [34])

The unit of figure 3.5 is the main component of a Processing Element calculating the value of a single cell in the scoring matrix at each clock cycle. The constants for the gap and substitution penalties are embedded in the logic which can been set at run time. The query sequence characters are also embedded in the logic and they are used as a run time parameter to produce the customized circuit. The final output score is obtained by an up-down counter at the end of the systolic array. Figure 3.6 shows the systolic array of this architecture.

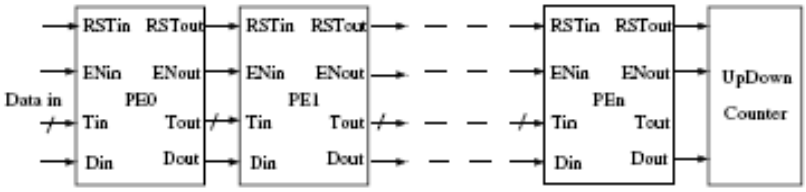

Figure 3.6 The systolic array structure (from [34])

This architecture has been implemented in the Osiris board which contains two Virtex II FPGAs. One FPGA, a VirtexII 1000, is used as an interface between the user FPGA and a host PC. The second FPGA, a VirtexII 6000, is a user programmable device available for programming designs. In such a device 7000 processing elements along with the glue logic fit. The processing element was found to run at 180MHz in this device of speed grade -4. Obviously, the Osiris board can hold 4 systolic arrays, containing 1750 elements each. The four arrays keep the same query.

Regarding the performance of the system, the 7000 processing elements running at 180MHz result in 1260 billion updates in the cells of the algorithm's matrix. Finally, table 3.2 gives the estimated time required for different databases of GenBank.

Database	*Size(in MB)*	*Time (in sec)*
GBUNA	0.159	0.002s
GBPHG	3.27	0.030s
GB14	11.2	0.103
GBGSS4	19.2	0.177
GB36	28.8	0.267
GB48	38.4	0.355
GBPRI20	47.5	0.438

Table 3.2 Performance of the Hokiegene System for the GenBank databases(based on [34])

Another interesting implementation of the Smith Algorithm is found in [33]. In this work the Xilinx JBits toolkit has been used, which is a *set of Java tools and API that permit direct impementation and reconfiguration of circuits for the XilinxVirtex family of FPGAs*. Figure 3.7 shows the combinatorial core of the SW algorithm, where each gray box represents a LUT/Flip-Flop pair. As runtime parameters can be set the constants a, b, c as well the sequence S. Based on the core logic of figure 3.7 a systolic array is formed similar to that of the Hokiegene system described above.

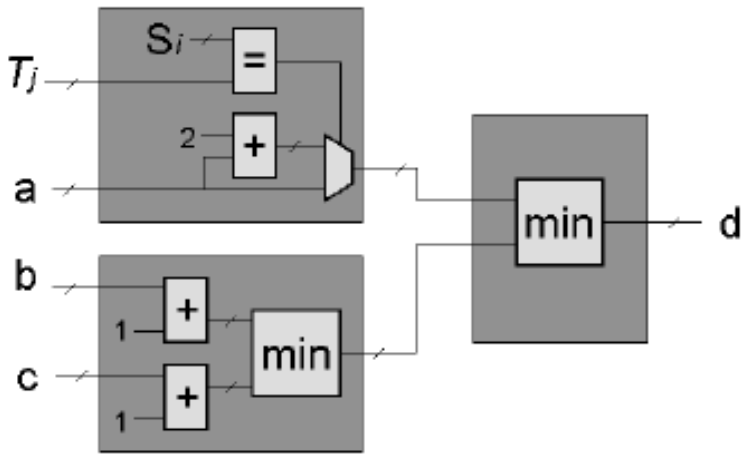

Figure 3.7. Smith Waterman algorithm implemented in hardware (from [33])

One advantage of the use of Jbits in this design is the space utilization. The developers of the system compared the space utilization of the circuit produced with Jbits with the one produced in Splash2 with standard VHDL and found that the number of the utilized LUTs decreased by a factor of 5.5.

Another benefit of the Jbits in this design of the run time reconfiguration is the effective folding of string S into the circuit. Although VHDL tools could embed the always constant values of the gap and substitution penalties, the periodically changing string data needed runtime reconfiguration.

Regarding the performance of this system with the device XC2V6000 of Virtex2 family, 11,000 processing elements were implemented at a clock speed of over 280 MHz. This implementation gave a matching rate of over 3.2 trillion elements per second.

3.3 Hardware accelerating BLAST

In contrast to the Smith Waterman algorithm, there are not a lot of hardware implementations available, either scientific or commercial, accelerating the BLAST algorithm. This section provides a brief overview of the current BLAST accelerators.

3.3.1 RC-BLAST

In [38], Muriki et al firstly profiled the performance of the BLAST software found on the NCBI website for a given query and a database. They found out that the 120 line long BlastNtWordFinder function of the file blast.c consumed about the 80% of the total execution time of the program and this amount of time increased with the size of the database. They further profiled this function and they isolated the computationally hard part of this function. Then they implemented this critical code in hardware and loaded it in an FPGA.

The application starts running from a host computer running the NCBI BLAST software. The query sequence is used to build a look up table for the w-mer

comparison of step2 and with this table an SRAM memory of the evaluation board is initialized. The software segment of the critical code is replaced by an interface stub with the FPGA board which helps with the initialization of the lookup table of the board, it sends the database subsequence as input to hardware and it reads the board's answers. When the host processor sends data to the FPGA, it waits until it receives an answer. In case the processor receives hit, the function implementing the hit extension is called to make the extension and finally the results are reported on the output file.

The card used for the implementation of this system is a PCI bus based card with two Xilinx XC4085XLA FPGAs and two SRAM blocks of 1 MB each, while the system used as a host computer and for comparison purposes was an Intel i386 machine.

However, due to the lack of advanced IO capabilities of the used card (the PCI protocol is rather slow as we will see in future sections), the pure software implementation ended to be faster than the system of RC BLAST. In fact, with the same test sequence as a query and the ecoli.nt as the subject database, the software scanned the database in 0.40 seconds whereas the RC BLAST needed 1.93 seconds. An improvement of the system was the fact that instead of sending all the subsequences to the FPGA for processing, only a fraction of them were sent to hardware and in the same time the personal computer performed the same task of the rest subsequences. With this approach, the execution of the above experiment took 1.85 seconds, still much more than the software execution time.

Although the results of this work are not impressive, they are encouraging for the field of BLAST acceleration as the main problem is identified to be the speed of communication between a hardware architecture and the rest personal computing structure.

3.3.2 Implementation on the BEE2 system

BEE2(Berkeley Emulation System)[47] is an FPGA reconfigurable platform consisting of three primary components: processing elements, memory elements, and interconnects. On the system level, processing elements are the FPGA chips; memory elements are the external DRAM modules locally attached to each of the FPGA;

interconnects consists of local connections, which links local FPGAs on the same PCB board, as well as global connections that link multiple boards into a unified system. The main difference of the BEE2 design from traditional parallel computer system design is that the processing elements are FPGA chips rather than microprocessors. In addition to the primary components, BEE2 also incorporate a range of secondary system components, including bootstrap, clock distribution, power regulation, and thermal regulation. They support and monitor the primary components to ensure proper operation of the overall system.

In the first step of this BLAST algorithm implementation an index of all w-mers for one of the input sequences is constructed, as shown in Figure 3.8. The index is a 1D array in which the address of each entry corresponds to a unique w-mer. The content of the index points to a packed array containing the locations that a particular w-mer appears in the input sequence.

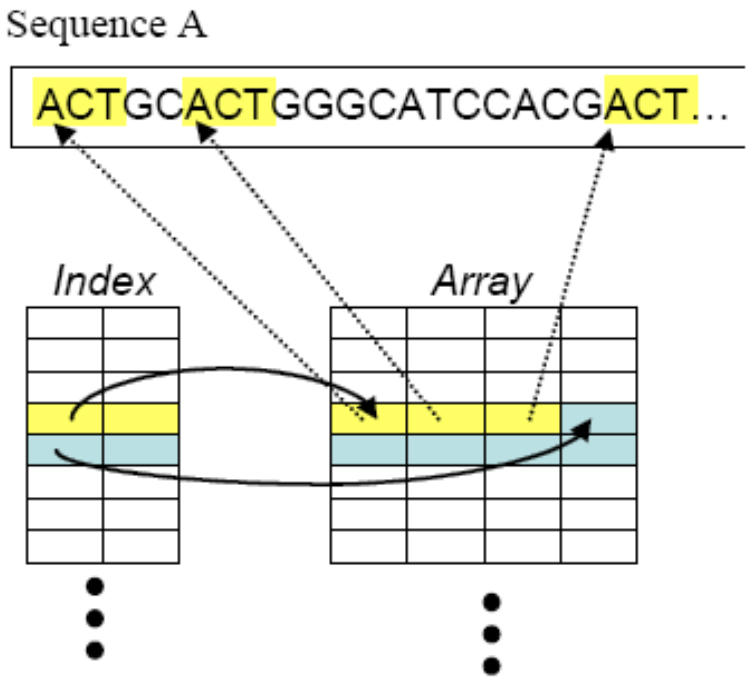

Figure 3.8 w-mer indexing of BEE2 implementation(from [47])

Second step takes for input the indexed array from step 1 and the second input sequence. The output of this step is also an index and a packed array. The resulting array for step 2 contains the locations of the hits (or complete K-mer match) from the

two input sequences. The key used to index and sort the array is the diagonal of 2D dynamic programming matrix on which a hit is located, as shown in figure 3.9.

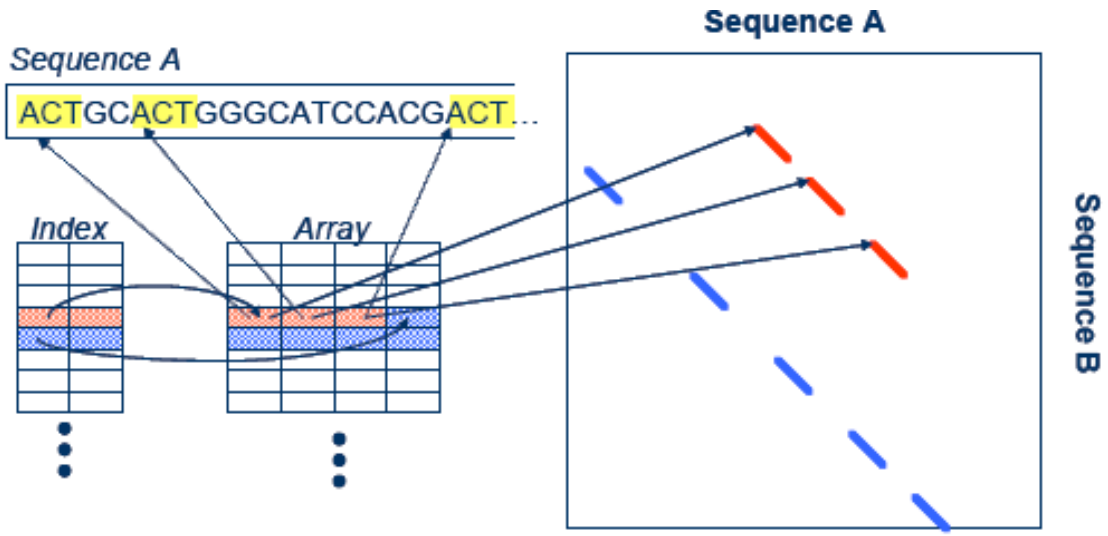

Figure 3.9 hit finding unit of BEE2(from [47])

In the last step of the implementation, the input is indexed array from the previous step, and the two input sequences. In this step the hits from step 2 are processed and expanded. Results with e-values above a certain threshold are returned. Since BLAST only does hit expansion along diagonals, each index entry representing a separate diagonal can be processed in parallel with no dependency from any of the other diagonals.

The overall system has been simulated cycle accurately in Matlab for a 300KB query against a 1.2 GB database. Assuming a clock frequency of 100 MHz, figure 3.10 shows the expected execution time of systems with different number of FPGAs.

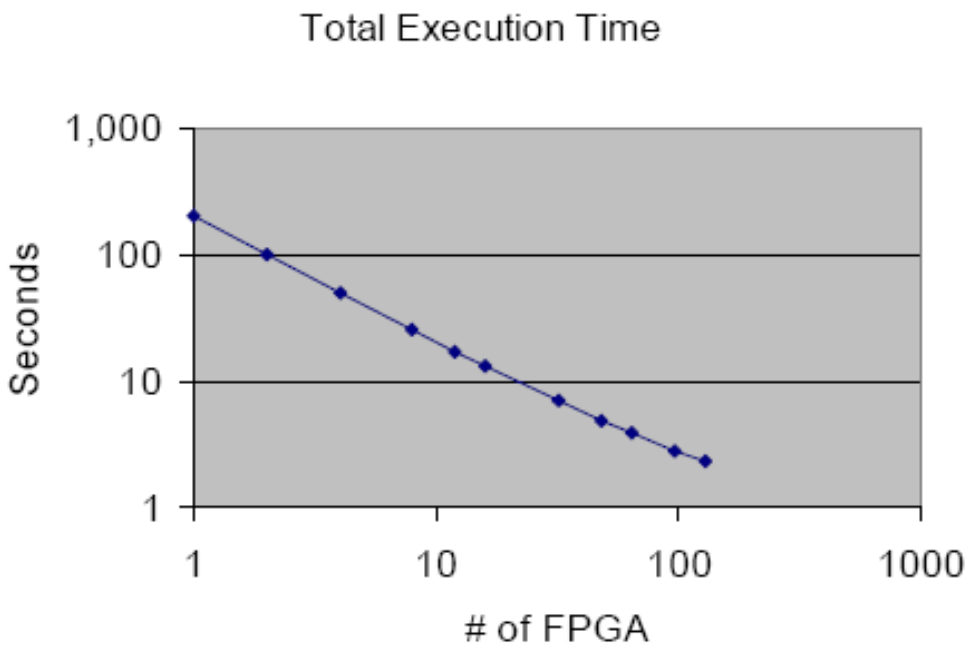

Figure 3.10 Execution time of BEE2 platform(from [47])

3.3.3 Decypher platform

Timelogic has developed a commercial BLAST accelerator, the DeCypherBLAST[15], which executes BLASTN, BLASTP, BLASTX, T-BLASTN and T-BLASTX searches using an accelerator card in a single server. However, lack of information about the architecture itself (number of chips, architecture type, etc.) as well as the details of that type of BLAST implemented by Decypher (it is reported that *Tera-BLAST* algorithms are used without further explanations of their functionality) do not allow for comparisons with our present work. Figure 3.11 shows the only reported performance calculation of this system and a comparison with an 8 CPU cluster. In this calculation all bacterial proteins (4,242 proteins sequences) were compared against 192 E. coli genomes (775 million symbols in 6- frames). However, with such parameters it is difficult to compare this system with the system developed in this thesis due to different performance parameters.

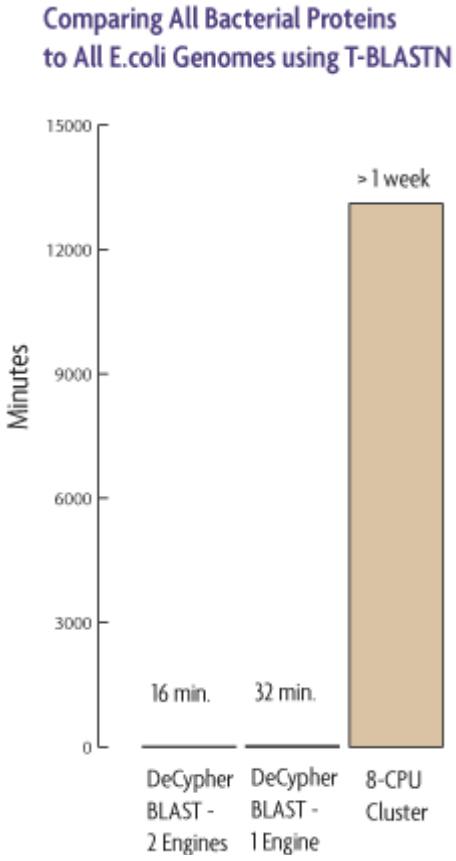

Figure3.11 Decypher performance evaluation (from [15])

3.4 Supercomputers running the BLAST software

3.4.1 IBM POWER4 pSeries 690 Model 681

BLAST software is usually used as a benchmarking tool for current computing systems. Although most systems do not report execution times rather than speed ups, IBM has a detailed performance evaluation on a pSeries 690 system with BLAST as a benchmarking tool. The pSeries 690 server is a major step that IBM has taken toward providing customers with shared memory machines and was introduced in October 2001. This new machine scales up to a 32-way POWER4 1.3 GHz or 1.1 GHz SMP machine with up to 256 GB of memory. The IBM pSeries 690 uses multi-chip modules (MCM), which are equivalent to a mainframe's central processing module. This approach optimizes chip-to-chip communications, boosting performance of the overall system.

Benchmarking this system with the BLASTn program with small single queries against large databases, IBM uses a query sequence of length 1998 and a large database (the ensembl.dna) with about 3.4 billion nucleotides. Table 3.3 shows the execution times obtained by the execution of the just described BLASTn query on a pSeries 690 Model 681 for a different number of processors.

Number of processors	*Elapsed time(sec)*
1	21.32
2	11.39
4	3.26
8	4.92
16	6.40

Table 3.3 BLASTn benchmarks for a single small query against a large database executed by a pSeries 690 Model 681

3.4.2 mpi BLAST

mpi BLAST[36] is an open-source parallelization of BLAST that achieves superlinear speed-up by segmenting a BLAST database and then having each node in a computational cluster search a unique portion of the database. Database segmentation permits each node to search a smaller portion of the database, eliminating disk I/O and vastly improving BLAST performance. Because database segmentation does not create heavy communication demands, BLAST users can take advantage of low-cost and efficient Linux cluster architectures.

One of the supercomputers ran the mpi BLAST software was the *Green Destiny*[37]. *Green Destiny* is a 240-processor supercomputer that operates at a peak rate of 240 billion floating-point operations per second (or 240 gigaflops) but fits in six square feet and sips as little as 3.2 kilowatts of power. Consequently, it does not require any special infrastructure to operate, i.e., no cooling, no raised floor, no air filtration, and no humidification control. Without giving a lot of details about the *Green Destiny* system, its performance comparison with a simple personal computer running the mpi BLAST software is impressive. Figure 3.12 shows the time needed

for the execution of mpi BLAST with a 300KB query against the 5.1-GB database. Overall, this query takes 1346 minutes (or 22.4 hours) on one compute node and less than 8 minutes on 128 nodes of Green Destiny. As it is shown in figure 3.12, the efficiency of mpiBLAST decreases as the number of nodes increase. If the "speed-up" column is divided by the "# nodes" column, the efficiencyof mpiBLAST across four nodes is 2.31 (9.23/4) and drops all the way down to 1.33 (170.41/128) when run across 128 nodes. The reason for this dropoff is due to the tradeoff that exists when segmenting the database into many small fragments. There is significant overhead in searching extra fragments; thus, the ideal database segment will typically be the largest fragment that can fit in memory and not cause any swapping to disk. Making fragments smaller than the available memory simply adds overhead.

# Nodes	Run Time (sec)	Speed-Up
1	80775	1.00
4	8752	9.23
8	4548	17.76
16	2437	33.15
32	1350	59.83
64	851	94.92
128	474	170.41

Figure 3.12 Runtime for a 300 KB query against the nt database performed by Green Destiny(from[37])

3.5 Algorithm selection

Although Smith-Waterman dynamic algorithm is widely implemented in hardware with systolic arrays which are ideal for mapping in FPGAs, BLAST algorithm cannot be parallelized easily as it uses heuristics. However, its execution can be parallelized as searching in distant database positions or the concurrent searching of multiple queries are independent procedure and can be parallelized in a hardware architecture. But the main problem due to which there are not many hardware implementations of the BLAST is its Input/ Output problem which will be

discussed in following chapters. As the current FPGA technology offers Gigabit transceiver solutions, this problem is feasible to be overcome in an implementation with reconfigurable logic. Taking advantage of the current technological features, in this thesis we deal with the hardware acceleration of the BLAST tool, popular not only for its accuracy but for the open source implementation available from the NCBI website.

3.6 Overview

In this chapter an overview of the hardware accelerators developed for the sequence matching algorithms. Obviously, the Needleman Wunch algorithm could not have many implementations as it is not popular any more. On the contrary Smith-Waterman algorithm offers a variety of hardware implementations as apart from popular for its sensitivity is also easily implemented with systolic arrays. Finally, the BLAST algorithm hardware approaches have been divided into two categories; the development of special purpose hardware, either in academic or in industrial efforts, implementing exclusively the algorithm, and the running of the open source versions on supercomputers.

Chapter 4 - BLAST software

4.1 The different BLAST programs.

The BLAST algorithm is employed by the programs blastp, blastx, blastn, tblastn, tblastx. Their differences are summarized in table 4.1.

program	description
blastp	Query: amino acid, database: amino acid
blastn	Query: nucleotide, database: nucleotide
blastx	Query: translated nucleotide sequence, database: amino acid
tblastn	Query: amino acid, database: translated nucleotide sequence
tblastx	Query: translated nucleotide sequence, database: translated nucleotide sequence

Table 4.1 The different BLAST programs

In the current thesis for simplicity, as it is explained later, it is the blastn program that is used and all the other programs are disregarded.

4.2 The NCBI implementation

Since 1988 the National Center for Biotechnology Information (NCBI) [8] creates public databases, conducts research in computational biology, develops software tools for analyzing genome data, and disseminates biomedical information - all for the better understanding of molecular processes affecting human health and disease.

In its website there is an open source implementation of the BLAST algorithm while from the ftp pages of the Center there is available the genetic database which

consists of numerous files containing biological data. In addition, there is available a web application which performs biological search in a certain application.

For benchmarking reasons we downloaded the executable files of version 2.2.10 of the implementation. In the following of this section we briefly describe the options for the execution.

Part1: Execution of the formatdb program.

The input databases to the BLAST program are not in the readable FASTA format, but in a compressed format. So, firstly the formatdb program has to be executed for each database file to be used.

For a detailed list of arguments of this program, the reader has to look at the relevant documentation[8]. However, we briefly explain the most useful arguments of this program with the following example:

Assuming that we could have a file named 'ecoli.nuc.txt' and format it as 'ecoli'. Then we should type the following command:

formatdb -i ecoli.nt -p F -o T -n ecoli

The -i flag denotes that one or more filename(s) of database data follow.

The -p flag is an optional flag which is followed by T if the type of the file is protein and F if the file consists of nucleotide sequences. The default value is set to T.

The -o flag is followed by T and parses seqID and creates indexes. When it is followed by F, it does not create any indexes.

The -n flag allows a user to create BLAST databases with a different names other than the original FASTA files. This can be used in situations where the original FASTA file is not required other than by formatdb. This can help in a situation where disk-space is tight.

Part2: Perform a search against the Database

Having created a database by using the aforementioned procedure, a user is ready to perform a search against this database. So a text file containing a query in the familiar FASTA format should be generated. It is a common sense that we do not have to be biologists in order to produce such queries. On the other hand, we could "cheat" in this stage and just extract a nucleotide sequence we already know exists in a file comprising our Database.

Supposing that our query file is the test.txt and the database file is the ecoli the command performing a search is the following:

blastall -p blastn -d ecoli.nt -i test.txt -o test.out

Blastall may be used to perform all five flavors of blast comparison. One may obtain the blastall options by executing 'blastall -' (note the dash). Here the most useful arguments of the program will be explained.

The -p flag denotes the choice of the program name. It must be followed by one of the strings "blastp", "blastn", "blastx", "tblastn", or "tblastx".

The -d flag is followed by the name of a formated database. Multiple database names (bracketed by quotations) are also accepted. An example would be *-d "ecoli.nt est"* , which will search both the ecoli and est databases, presenting the results as if one 'virtual' database consisting of all the entries from both were searched. The statistics are based on the 'virtual' database of ecoli.nt and est.

The -i flag is followed by the input file in FASTA format. This flag is optional as the default input file is the standard input. If multiple FASTA entries are in the input file, all queries will be searched.

The -o flag denotes the file in which the output will be stored. The default output file is the standard output.

4.3 Dimensioning

Before continuing with any implementation, it is essential to discuss the different sizes of the queries and the databases.

In [46] there is a concise description of the available sizes of the queries and the databases. As it is described in this paper, databases are classified by looking the size of a particular database, either in terms of characters or in terms of megabytes. There are three different cases; small, medium and large. Small consist of 400 sequences or 4.7MB, medium is between 400 and 6000 sequences or 5MB and 200MB and large is between 6000 and 200000 sequuences or 200MB and 4GB.

On the other hand, the type of query is classified by the number of sequences (single or multiple) and by the total number of characters involved in the query (small, medium, and large). In the case of a single sequence, small (small sequence) corresponds to less than or equal to 2000 characters, medium is between 2000 and 50000 characters, and large is between 50000 and 200000 characters. Multiple sequence queries were classified by the number of characters and by the total number of sequences per query. For multiple sequences; small corresponds to less than or equal to 2000 characters and a total of 20 sequences or less per query; medium is between 2000 and 50000 characters and between 20 and 200 sequences per query; and large is between 50000 and 200000 characters and between 200 and 2000 sequences per query.

Last but not least is the size of the w-mers we will produce, as it will play an important role in our computation endeavors. According to the NCBI manual, the most frequent values for the blastn program vary between 11 and 15, while the default value for this implementation is the 11.

In this thesis, we use small, single queries (size = 1024), w-mer size either 11 or 12 and large databases.

4.4 Software Implementation in this thesis

4.4.1 Database translation

In our software implementation the database input will be in FASTA format, which has been described earlier in this thesis. So, firstly all the comment lines of the database have to be removed. We remind the reader that such lines start with character '>' in the FASTA files. For this reason, the UNIX tool grep is used. Grep is a filter that searches a file and prints that content containing a certain pattern. This pattern could be a certain string or a regular expression. Of course there is also the opposite choice available, i.e printing the entire file except for that content containing a certain pattern. The idea of using this filter is that the first letter of a comment sequence is the character '>' and the last is the newline. Hence, the file should be printed without any sequence beginning with '>' and ending with '\n'. taking into account that the -v flag of grep prints everything not containing a pattern, the command used is the following.

grep -v ['>'.\n] <input file>*

We need to redirect the output of this script to a file in order to store the result.

Having removed all the comment lines, we are ready to convert all the database characters into small integers, as the alphabet size of a nucleotide database is 4 (we remind the user that Σ={A, T, C, G}). Hence, these characters can be represented by using only 2-bit numbers. The translation of this database in this software implementation follows the mapping of table 4.2.

Nucleotide Character	***Small number conversion***
A	0
T	1
C	2
G	3

Table 4.2 Mapping of the DNA base letters with a two bit number

This conversion is easy to be implemented if UNIX command tr is used. This command transforms the characters of the input file, taking into account that it does not change the file as it is also a filter so we have to redirect its output to store the result. The command as we used it is the following:

tr ATCG 0123 < <source_file> > <destination_file>

4.4.2 Database Compression.

As it is stated in the BLAST original paper, 2-bit letters are compressed to form a byte-long four letter word. In this way, not only space is saved, but also time as with one access to a memory byte we have 4 letters available for processing.

To compress the database files containing small integer sequences, which are produced with the method described in the above section, a C program has been created which reads integer numbers from a file containing sequences consisting of the numbers 0, 1, 2, 3 and fits four consecutive numbers in a single byte. The following pseudo code illustrates the process of the compression.

```
//Initialization
  char tmp = 0; //tmp is the byte-long variable used to form the four-character values.
  short rd = 0;

 //Read the file
  while(the end of the source file has not been reached):
     rd = read_source_file //read a number of the DB file
     tmp = (tmp<<2)+tmp //the tmp is shifted left 2 bits and the new number is appended
                                        //at its least significant region.
     if (four consecutive numbers have been appended to tmp):
              write tmp at the destination file
              reinitialize tmp
```

4.4.3 Description of the BLASTn machine

Having compressed the database of the nucleotide sequences in sequences of 8-bit values containing 4 letters, we are ready to develop our first BLASTn machine using C language. It takes as input a query file in FASTA format and a database file in a compressed format, as it is described in the previous section. The output is a list of

all HSPs produced by the BLAST process and their scores without considering any statistical, pre-filtering, or overlapping issues. The scoring matrix used is the very popular shown on table 4.3.

	A	***T***	***C***	***G***
A	5	-4	-4	-4
T	-4	5	-4	-4
C	-4	-4	5	-4
G	-4	-4	-4	5

Table 4.3 The scoring matrix of our implementations

According to table 4.3 the score for each match is 5 while the score for each mismatch is -4. The size W of the w-mers is fixed and equal to 12 while the threshold value T is 50. Obviously a seed (i.e. A portion of a database which produces a hit) is scored with 60, as there are 12 letters matching with a w-mer and scored with 5.

Now we will briefly describe implementation details of each step of the algorithm.

Step1: Initially the query file is scanned and its content is stored in an array as integers 0, 1, 2, 3. After that, we scan this array and create a linked list of the w-mers. We remind that this list consists of all possible words of length w that lie on the buffer. For the implementation of the list, dynamic memory allocation has been used even though an array implementation could be used as the length of the list is known for a fixed question length. Dynamic memory allocation was preferred because it was simpler and we are not interested in the performance of the algorithm in the current implementation. As soon as step1 finishes, we proceed to the database scanning.

Steps 2,3: This stage of the algorithm is repeated until the entire database file is scanned. For each iteration one byte is read from the compressed database file and it is decompressed in four integers which correspond to the initial alphabet. Then for each integer, a word with length w is created and it is compared against the list of the w-mers in order for hits being found. If a hit is found we immediately proceed to the left extension while the score is greater than a threshold (we selected 50). In order for the left extension being possible, some of the last database letters have to be kept in

the main memory. So, a buffer has been introduced with size as double as the size of our query (in our example the size of this buffer is 2000) where every new incoming letter is stored.

When the left extension of an HSP finishes, we proceed with the right extension. However, as data for this process are not yet available, this HSP is added into a waiting list (called as hit list) for the right extension. This hit list contains all the "active" HSPs, i.e. those HSPs whose extension has not finished yet. Obviously, HSPs are characterized by two integer numbers indicating the starting positions of the HSP in the query and the database, its length and its score.

The right extension is slightly different. First of all we disregard the score of the left extension and we consider that the score is the one obtained from the seed. Then for each database letter we extend by one letter all of the nodes located in the previously stated hit list. If a score of any node falls under the threshold the right extension procedure terminates for this node and the HSP is reported.

For a better understanding of the procedure described for steps 2 and 3, we add the flowchart in figure 4.1.

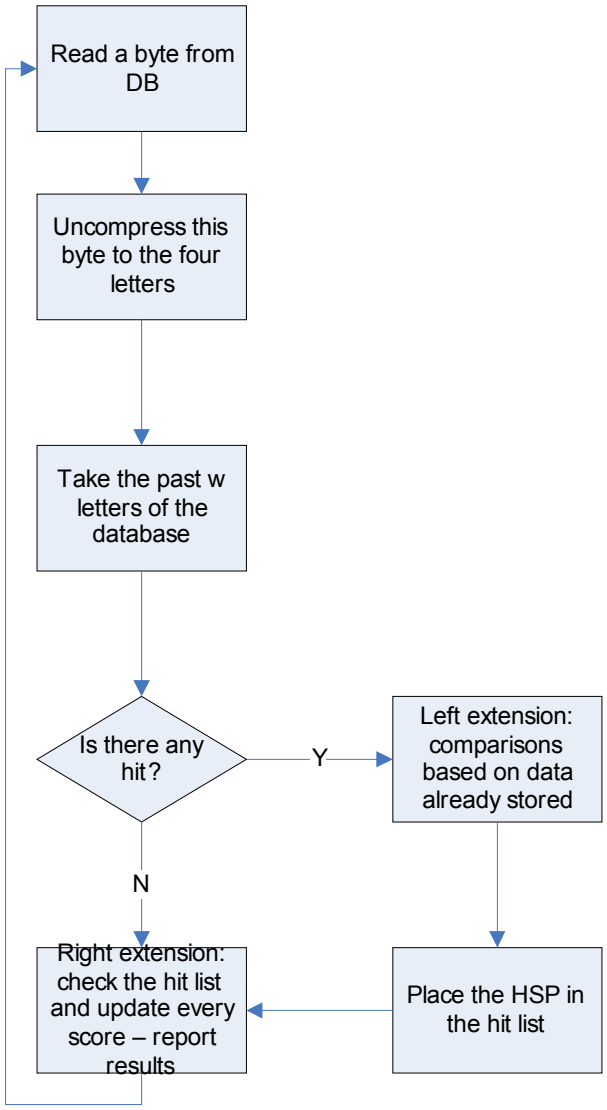

Figure 4.1 The flowchart of our software implementation

It should be admitted that this method is not really good because the extension process is very weak. First we extend left until the threshold is met and afterwards we disregard this score for the right extension.

Comparing the speed of this unit with the NCBI BLAST tool, it is obvious that the latter is much faster for the same inputs in the same computer. This is obvious because we did not care about the performance of our software.

Regarding the ouptuts of both software tools, there were a lot of incompatibilities. The reason is that the NCBI tool performs a lot of optimizations as well it applies statistical methods in the output of its results. However, the outputs of

the NCBI tool were included in the output of our implementation. For this reason we are confident for the functionality of this software.

4.5 Overview

In this chapter we dealt with the software implementation of the algorithm. After having discussed the six different BLAST programs, we dealt with the running of the implementation available of the NCBI web site. Then we standardized the parameters of our implementation after a careful view to all the different dimensions. Finally, based on the original paper of the BLASTn algorithm, we implemented our own BLASTn machine after having compressed the database so as each byte contains four consecutive letters.

Chapter 5: The first generation of TUC-BLAST

Section 5.1 describes the architecture developed by E. Sotiriades in [40] accelerating the execution of the BLASTn program. This architecture consists of N identical machines each of which executes the BLAST algorithm for different parts of the input database. In this thesis, a single machine of figure 5.1 consisting of the components of figures 5.2 and 5.3 has been developed and synthesized with Xilinx ISE 7.1 tool, and then it has been post place and route simulated with Modelsim 6.0. Finally, this machine is evaluated in terms of its performance and reliability.

5.1 A brief description of the architecture of the system

Figure 5.1 shows the architecture of the system. This architecture is divided into N identical computing machines, *each of which implements all three steps of the algorithm*. Input data have a width of 2N bits, and come from N different channels. Every channel drives one of the N computing machines. Every machine has two major subsystems, one for step 2 of the algorithm and one for step 3. The first step of the algorithm (the W-mer calculation) is precalculated before the algorithm is run. The precalculation results are the first inputs for the machine and they are stored in the memory, together with their position in the query. After this procedure the data stream of the database starts to be processed and if a match is found the second component of the architecture is activated and starts to extend the match, thus implementing the third step of the algorithm.

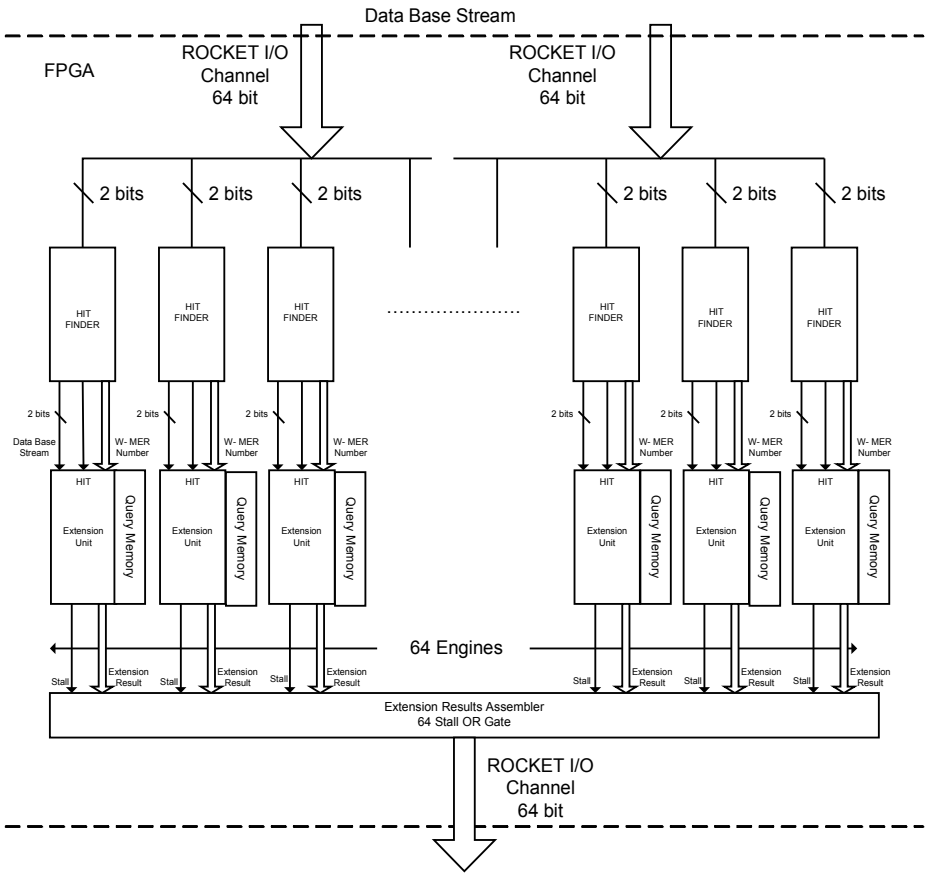

Figure 5.1 The BLAST acellerator architecture

In figure 5.2 there is the block diagram of the hit finder unit, which solves the second step of the BLAST algorithm. The input of the unit in normal mode (database search) is the database stream, one character for each machine. Only the 10 MSBs of W-mers are stored in the w-mer memory and at the address which corresponds to their 12 LSBs. The stored bits are called w-mer tags. The width of the memory is 23 bits, 10 for the W-mer tag, 1 for valid, and the remaining 12 to show the position of the corresponding w-mer in the input query. This memory is initialized before the database search begins, just when the w-mer list is available. This initialization process does not affect the performance of the algorithm significantly as it takes approximately 1000 cycles for a 1000 letter query. The Hit Finder Unit has also an input buffer which is 2 bits wide (1 character) and one thousand positions deep, called Future memory. This memory serves as a delay buffer in the flow of the input stream, as data following a hit should be also available in case they are needed during an extension.

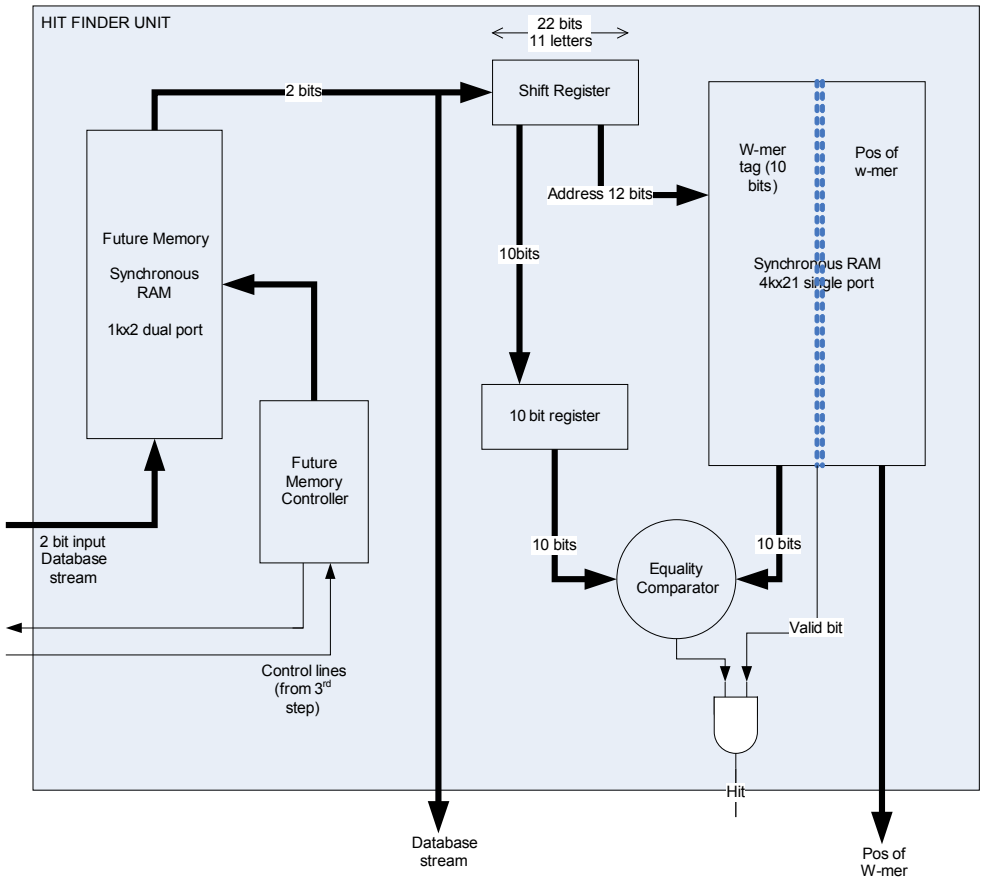

Figure 5.2 The Hit Finder Unit

The Extension Unit, shown in figure 5.3, executes two comparisons in every cycle, according to the algorithm. It extents HSPs in both sides and compares two pairs of letters. The first pair comes from the query memory and the history memory and the remaining couple comes from the Query memory and the Future memory. The data from the input are buffered in the History and Future memories. There are also counters and registers that keep several useful data, such as hit position for query and database, its length, and the score (which is the most important result to be calculated). Based on the score all the remaining useful data for biologists (e.g. e-value) can be calculated.

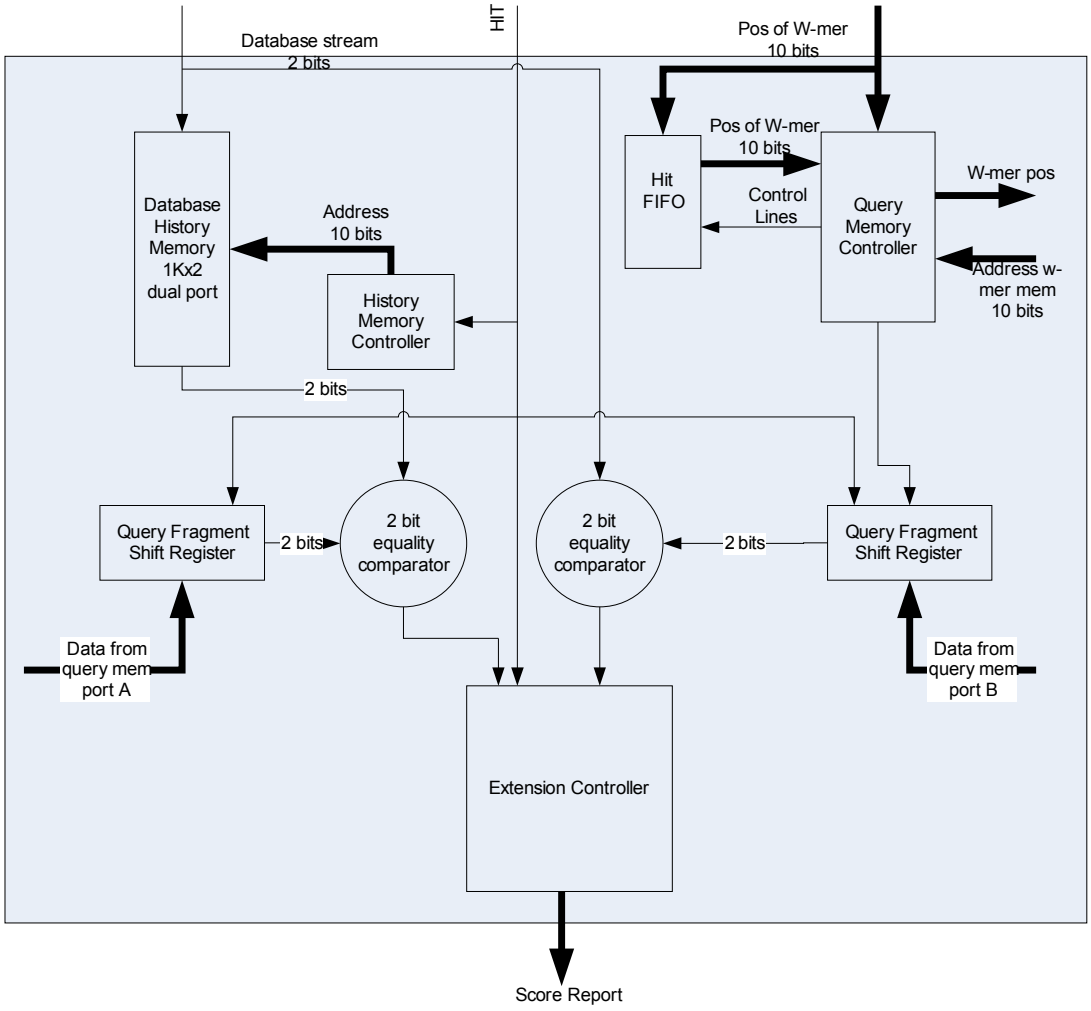

Figure 5.3 The Extension Unit

5.2 Implementation of a single machine

5.2.1 Virtex4 Block RAM

In addition to distributed RAM memory, i.e. the memory produced by the Configurable Logic Blocks (CLBs) of an FPGA, Virtex-4 devices feature a large number of 18 Kb block RAM memories. True Dual-Port™ RAM offers fast blocks of memory in the device. Block RAMs are placed in columns, and the total number of block RAM memory depends on the size of the Virtex-4 device. The 18 Kb blocks are

cascadable to enable a deeper and wider memory implementation, with a minimal timing penalty. Embedded dual- or single-port RAM modules, ROM modules, synchronous FIFOs, and data width converters are easily implemented using the Xilinx CORE Generator™ block memory modules. Asynchronous FIFOs can be generated using the CORE Generator FIFO Generator module. The synchronous or asynchronous FIFO implementation does not require additional CLB resources for the FIFO control logic since it uses dedicated hardware resources.

The 18 Kb block RAM dual-port memory consists of an 18 Kb storage area and two completely independent access ports, A and B. The structure is fully symmetrical, and both ports are interchangeable. Data can be written to either or both ports and can be read from either or both ports. Each write operation is synchronous, each port has its own address, data in, data out, clock, clock enable, and write enable. The read operation is synchronous and requires a clock edge. There is no dedicated monitor to arbitrate the effect of identical addresses on both ports. It is up to the user to time the two clocks appropriately. However, conflicting simultaneous writes to the same location never cause any physical damage, as the following three write modes are available:

The WRITE_FIRST mode, where the input data is simultaneously written into memory and stored in the data output (transparent write)

The READ_FIRST mode, where data previously stored at the write address appears on the output latches, while the input data is being stored in memory (read before write).

The NO_CHANGE mode, where the output latches remain unchanged during a write operation.

Mode selection is set by configuration. One of these three modes is set individually for each port by an attribute. The default mode is WRITE_FIRST.

5.2.2 Technology mapping

5.2.2.1 memories

In the following lines we describe the different memories we need for our design.

W-mer memory

This is a single port memory consisting of 4k entries of 21 bits each, and it is used in the hit finder unit (figure 5.2) to find if every string of 12 consecutive letters of the database matches with any of the query w-mers. It has been constructed with Block RAM memories ant it utilizes five Blocks of RAM.

Future and History memories

These memories are used as buffers so that the system can remember the database sequence recently passed (in the case of the history memory) as well the database letters will be processed afterwards. Both memories are dual port (with the one port read only and the other one write only) and have the same size (1024 entries of 2 bits) and each one consists of one Block RAM. Their write mode is READ_FIRST.

With this write mode, future memory can perform as a FIFO in the normal mode of the execution (as it is described in 5.1) if both read and write addresses are the same. In this case, the read operation reads the value stored 1024 cycles ago, while the write operation overwrites it with new data.

Query Memory

This is also a dual port block memory and is used to keep the query string. In our case it consists of 1024 entries of 2 bit words. Both ports are used by the extension unit (figure 5.3) to provide the query letters in certain positions. However, these ports cannot be configured as Read Only, because they are also used to initialize this memory during the pre-compilation stage.

Needless to say that query memory utilizes one Block of the embedded RAM, since it is comprised by only 2k bits.

5.2.2.2 Comparator

For the comparison of the 2 bit values in the extension unit, the comparator of figure 5.4 has been used. This comparator compares two values a and b and its output is high when both xor gates have zero output. This happens when the inputs of the xor gates are the same.

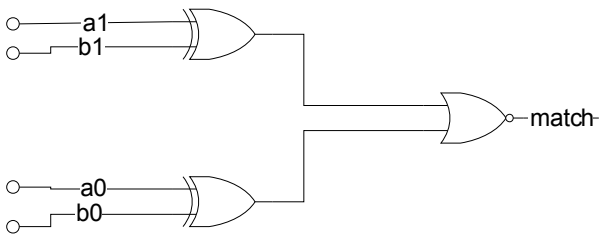

Figure 5.4 The two bit comparator

5.2.2.3The w-mer shift register

In figure 5.5 there is the 2 - bit shift register of depth 11 where words of 11 database letters are created in order for being compared against the query w-mers, as it is abstractly shown in figure 5.2. However, in this component one should be aware of an endianess issue that could lead us in false results if it is overseen. To illustrate this issue, we will use an example.

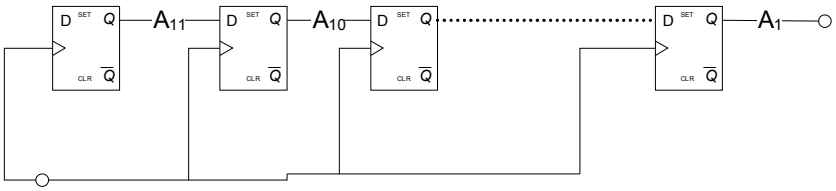

Figure 5.5 The w-mer shift register

Assuming that an eleven word is the following: $A_1A_2A_3...A_{11}$. As the comparison unit consists of a cache like memory structure, the substring $A_6...A_{11}$ is used for addressing the memory and the rest is used for the tag comparison. However, as figure 5.5 shows, the newly come data are stored in the most significant places of this vector (we use the convention that the leftmost bits of a vector are the most significant ones). So, the w-mer memory should be indexed by the most significant positions of this vector.

As concerns the tag region of this vector, no care could be taken of this issue provided that the tags of the memory containing the w-mers, are initialized with the correct endianess. In our implementation we consider this issue during the memory initialization.

5.2.2.4 Extension unit controller

The extension unit architecture, as shown in figure 5.3, contains a controller whose implementation is interesting to be discussed. The controller consists of a 13 bit counter and a Finite State Machine (FSM).

Extension counter

The size of the counter depends on the maximum score an extension may reach. For an 1000 letter query and adopting the scoring scheme of +5 in every match and -4 in every mismatch, the maximum score is achieved when the part of the extending database is the query itself. In this case there are 1000 matches, hence the maximum score is 5000. Since the integer number 5000 is represented with 13 bits, the size of this counter should be 13.

When an extension begins, the initial score is not zero. On the contrary, it is the score of w consecutive matches, because each extension process starts as soon as a part of the database matches with a w-mer. So, our counter should have a parallel load enable input which forces its output to a certain value. In our case, as w is 11, this value should be 55.

Finally, we should specify the way this counter counts up and down. As the extension process is involved in both sides of the input string, table 5.1 displays the different combinations of matches and mismatches that should be taken into account, in connection with the scoring scheme

Left extension	***Right extension***	***Action***
Match(+5)	Match(+5)	Increase 10
Match(+5)	Mismatch(-4)	Increase 1
Mismatch(-4)	Match(+5)	Increase 1
Mismatch(-4)	Mismatch(-4)	Decrease 8

Table 5.1 The scoring actions with the different combinations of matches and mismatches during the extension

From table 5.1 it is obvious that two input signals are required for the extension counter in order to control all the different cases. These signals are driven directly from the output of the comparators of figure 5.3.

Extension FSM

The function of the extension unit consists of two discrete states; when the unit is inactive, and the state when the unit is busy. It is implemented as a Mealy machine in order for the system not to have any cycles lost.

When the unit is inactive, it waits for a hit to come. During this time, the extension counter is reseted. As soon as the unit is notified that a hit waits for extension, it goes to busy state while the extension counter is initialized with the score of 11 consecutive matches. This is achieved by simply setting the parallel load enable input of the extension counter to 1.

In the busy state, the controller provides the addresses for accessing the right memory contents, so that the right comparisons are performed each time. In this state, the read addresses of all the relevant memories (i.e. future memory, history memory, query memory) are generated. If a hit is produced in the machine when the extension unit is busy, the latter ignores it as long as the unit is busy. As it has been described earlier in this chapter, all the relevant data of that hit is stored in a FIFO structure. The extension unit keeps to be in the busy state until one of the following happens: either a "finish" signal has been produced, or the length of the extension has already reached the length of the query. The finish signal is produced when the score begins to decrease from both directions. This happens when the comparisons of both directions produce mismatch.

5.2.3 Synchronization issues

In this architecture, there are a lot of issues which need careful treatment so that incoming data is used exactly when it is necessary. In the following lines we describe all these issues and we explain how we have dealt with them.

Issue1: In the architecture described above, during the extension procedure it is assumed that all database letters following the letter caused a hit are stored in the future memory. However, when the extension unit needs the first letters from the future memory, these letters have been already overwritten. Furthermore, history memory has not any free port to access them. So it is urgent to develop a unit where these letters are also temporarily stored outside the history memory.

Solution1: To address this problem, a shift register has been introduced to keep these values. The length of the shift register is equal to the difference of the cycle when data is used by the extension unit from the cycle when the data is erased from the future memory. This number is equal to four as there are 2 cycles until a hit is identified and other 2 cycles until the extension unit is triggered to work and asks for the first pair of letters.

Issue2: A major problem in this architecture was the future memory first read operation after stall. The problem was that the value of the address supposed to be read had been already overwritten before stall, and it has not been saved anywhere.

Solution2: As soon as a stall condition is detected and before the entire system stalls, a register is enabled to keep the data currently in the output of future memory. When the extension process ends, the output of the future memory in the first cycle after a stall is the value stored in this register.

Issue3: Due to the two cycle "gap" between a hit and the extension unit busy bit, it is possible that when stall conditions occur, stall may not been detected. This might occur when two hits occur sequentially. Then in the cycle after the first hit is identified, the extension unit is notified to begin the extension, which will happen in the following cycle. However, in the same cycle the second hit is identified and as the extension unit is not busy, there is not stall condition.

Solution3: For this reason, the stall signal is somewhat more complicated to be identified. Instead of a simple **and** operation of a hit and the busy bit of the extension unit, it should also check the cases when the extension unit has been notified although its busy bit is zero. For this reason a shift register of depth 2 has been added whose input is the trigger bit of step3 unit. For a stall detection, the system also checks the values of the registers of this unit.

5.2.4 Utilization

The unit has been implemented (until place and route phase) with Xilinx ISE 7.1 tool using as target FPGA the xcv4fx140 of the virtex4 family, package FF1517 with speed grade -11. Table 5.2 shows the design summary of a single machine of the first generation, while its speed is 121 MHz.

Number of Memory Blocks(out of 552)		*Number of 4 Input LUTs(out of 126,336)*	
8	1%	744	<1%

Table 5.2 Area utilization of a single machine in xcv4fx140 device

As table 5.2 shows, the critical aspect of this design is the block memory utilization, while the LUT utilization is negligible. Taking into account that the largest device of the virtex4 family provides 552 blocks of RAM, the maximum of 69 parallel machines could be produced in a single chip. Table 5.3 shows the area demands for an implementation of 60 and 69 machines, as well the clock frequency obtained from Xilinx tools.

Number of machines	*Number of Memory Blocks(out of 552)*		*Number of 4 Input LUTs(out of 126,336)*		*Speed(MHz)*
60	480	86%	46,522	36%	103
69	552	100%	53,836	42%	100

Table 5.3 Area and time constraints of an implementation of 60 and 69 machines

5.3Testing of the first generation

As it is discussed in the next section, the main drawback of the current architecture is the large number of collisions occurred in the w-mer memory, so it is difficult to perform simulations based on real questions and databases. Rather, we exhaustively simulated the performance of a single machine based on experiments we developed and tried to cover all the different cases. All the experiments described below successfully passed the simulation test in Modelsim.

Experiment1: This experiment involves a case where only one hit is found and it is extended in the following cycles. In this experiment, although the query memory was initialized with 1000 letter long data, only one entry of the comparison unit was

initialized with valid wmer data. The input database consisted of 3500 letters. Table 5.4 shows the interesting parts of the input strings

query	***A...AGAGGTT-CCCCCAAAAAC-GTACTAA...A***
database	***T.....TG..GCTGGTCCCCCCAAAAACGATCTT.....T***

Table 5.4 The input strings of experiment1

From the query and the database sequence of table 5.4 it is obvious that only one hit will be produced. This will happen when the database sequence will be compared with the wmer: *CCCCCAAAAAC*

It is expected that extension of both sides will be occurred and it will stop when the query and the database will be aligned as in table 5.5

query_part	***TT-CCCCCAAAAAC-GT***
database_part	***GT-CCCCCCAAAAAC-GA***

Table 5.5 The alignment produced for the experiment1

The score of the alignment of table 5.5 is 5*11+2*5-2*4=57. The reported HSP will not inlude the pairs of letters which produce the -8 alignment. So, the score of the reported HSP is expected to be 65.

Experiment2: This experiment has been designed to test the performance of a single BLAST machine when multiple hits arrive consecutively. In this case one has to examine if the machine stalls properly and if it recovers from a stall successfully, i.e. not any data losses occur to affect the functionality of the machine. In this experiment, the query memory was initialized with 1000 letter long data, but only three entries of the comparison unit were initialized with valid wmer data. The input database consisted of 3500 letters. As it is shown in Table 5.6 three consecutive hits are expected from the input strings.

query	***T...T-CCCCCAAAAAC-G-T.....***
database	***A...AT-CCCCCAAAAAC-G-CTTGCA..***

Table 5.6 The input strings of experiment2

When the first w-mer (the one beginning with T and ending with A) causes a hit, the extension unit immediately is notified to start to serve it. In the next cycle, another one hit is produced. As it is not possible for this hit to be immediately extended, data involving all the information necessary for the extension unit is stored to the FIFO, while the unit stalls and the hit searching stops. As soon as the extension of the first hit comes to an end, the extension is notified that a hit waits for extension in the FIFO and it proceeds with its extension. In the same time, the unit stops to be stalled and continues to search for hits, where a third hit is found. As in the previous case, the unit stalls again until the current extension of second HSP finishes, and the data of the third hit is stored to FIFO. Afterwards, the third hit starts its extension process and the unit stops to be stalled again.

Experiment3: The final experiment is the repetition of the experiment2 in two independent parts of the database. In more details, without changing the query sequence, we re-enter the same sequence as in experiment2 after some cycles of the completion of the process described above. This experiment assures us that no data is lost due to the consecutive stalls and all the stages of the architecture keep to be synchronized.

5.4 Evaluation of the first generation

5.4.1 Performance - Comparison

The NCBI software has been used as a benchmark from IBM for measurement of computing systems such as IBM 375 MHz POWER3-II multiprocessor (SMP) and the 1.1 GHz POWER4 pSeries 690 Model 681[46]. In this work several experiments

have been made for several sizes of databases, queries and BLAST version. Out of these results BLASTn results for small queries were selected to be compared with all the other experiments and are presented on Table 5.7. It can be shown that the fastest system throughput is achieved with the 16 processor Model 681 1.1 system, which has a throughput of 1,201.20 106 characters/sec. However, the fastest single chip system is the IBM Model 681 1.1 with 187.62 106 characters/sec.

Finally, our architecture performance is determined according to post place and route timing information of Xilinx software 7.1.03 which includes *Device speed data version: "ADVANCED 1.54 2005- 05-25"* for the specific device. Table 5.8 has speed measurements for the three experiments. Throughputs of all systems are presented at Table 5.9 and in Table 5.10 the speedup of TUC architecture against the other system is presented.

Number of Processors	**Type of Processors**	**Time (sec)**	**Database Size (characters)**	**Actual System Throughput (characters/sec)**	**Actual Throughput per Chip (characters/sec)**
1	POWER3	43.63	$4\ 10^9$	$91.68\ 10^6$	$91.68\ 10^6$
	Model 681 1.1	21.32	$4\ 10^9$	$187.62\ 10^6$	$187.62\ 10^6$
2	POWER3	24.09	$4\ 10^9$	$166.04\ 10^6$	$83.02\ 10^6$
	Model 681 1.1	11.39	$4\ 10^9$	$351.18\ 10^6$	$175.59\ 10^6$
4	POWER3	14.23	$4\ 10^9$	$281.10\ 10^6$	$70.27\ 10^6$
	Model 681 1.1	6.53	$4\ 10^9$	$612.56\ 10^6$	$153.14\ 10^6$
8	POWER3	9.25	$4\ 10^9$	$432.43\ 10^6$	$54.05\ 10^6$
	Model 681 1.1	4.33	$4\ 10^9$	$923.79\ 10^6$	$115.47\ 10^6$
16	POWER3	7.56	$4\ 10^9$	$529.10\ 10^6$	$33.07\ 10^6$
	Model 681 1.1	3.33	$4\ 10^9$	$1201.20\ 10^6$	$75.07\ 10^6$

Table 5.7 Performance of IBM systems

Number of Parallel Machines	**Speed (MHz)**	**Width of Data Stream (characters)**	**Predicted Throughput (characters/sec)**
1	121	1	$121.20\ 10^6$
60	103	60	$6{,}192.58\ 10^6$
69	100	69	$6{,}924.84\ 10^6$

Table 5.8 Speed measurements for various numbers of machines of TUC first generation

System	Predicted Throughput (10^6 characters/sec)
2GHz Xeon	319.25
1,7 GHz Intel M	256.26
2,66 GHz Intel P4	300.38
TUC Architecture N=1	121.20
TUC Architecture N=60	6,192.58
TUC Architecture N=69	6,924.84
IBM single chip	187.62
IBM System	1,201.20

Table 5.9 Predicted throughputs for a 1000 letter long query against a large database

	SpeedUp of TUC Architecture N=1	SpeedUp of TUC Architecture N=60	SpeedUp of TUC Architecture N=69
2GHz Xeon	3.76	192.37	215.12
IBM single chip	0.65	33.00	36.90
IBM System (16 chips)	0.10	5.15	5.76

Table 5.10 Predicted speedups of TUC architecture against the other systems

5.4.2 Memory limits

As it is mentioned in section 5.3, the number of the parallel machines generated in a specific FPGA device, depends on the number of the available memory blocks, as the logic utilization of a single machine is negligible. This situation motivates one to search for a more effective resource allocation, taking into account that entire memory blocks are allocated to produce memories of 2k bits only (future, history and query memory consist of 1024 2-bit entries). An improvement could be the implementation of such a memory with logic cells. A dual port 2k bit distributed memory utilizes 460 slices so that 69 machines need 31740 for this memory substitution (about 50% of the total number of slices available(63168) of the xcv4fx140 device). Considering also the logic utilization of the rest circuit (about 42% of the total LUTs as shown in table 5.3 for the 63 machines), the slice utilization increases dramatically so that about the

same number of parallel machine could be also generated with such substitution) so it is not worth the trouble to develop any of these memory units with distributed RAMs.

5.4.3 Drawback

The main drawback of this architecture is the architecture of the comparison unit. This architecture is effective only if there are not any collisions between the w-mers when they are placed in the memory. Collisions occur when indexes of two different w-mers are the same (imagine how a cache memory works!!!). Table 5.11 shows an example of such a collision. In this table both w-mers have same indexes, but different "tags".

Wmer-1	***CCCCC-AAAAAA***
Wmer-2	***TTTTT-AAAAAA***

Table 5.11 A case of collision in the matching scheme of first generation

The size of this problem can be determined only by analyzing candidate queries to find the number of collisions. A lot of sample queries (i.e. parts of known proteins) have been examined on this issue but not a single collision free 1000 letter string of any database could be found. On the other hand all these strings had collisions in about 140 different addresses by average.

So, with this architecture it is impossible to run real examples of 1000 letter long queries.

5.4.4 Conclusion

In the first generation of this architecture the performance results are encouraging as a large throughput is achieved with the parallel machines in an FPGA. Although the comparison unit is unreliable, it gives us the incentive to develop more effective architectures using memories to perform comparisons. Furthermore, each parallel machine exploits a large number of memory blocks.

5.5 Summary

In this chapter we dealt with the following aspects of the first generation of the BLAST architecture developed in Technical University of Crete (TUC).

In section 5.2 there has been a detailed description of the implementation issues of the first architecture, which was has been briefly presented in section 5.1

In section 5.3 the experiments conducted to verify the implementation of 5.1 are described.

Finally in section 5.4. there is a general evaluation of the first architecture.

Chapter 6: The second generation of TUC-BLAST

In this chapter the second generation of the system is described. The contribution of this thesis involved the design of the exact matching unit of a hit finder module of figure 6.1, the design of the control units of this component, as well the VHDL coding, synthesis, simulation and verification of the entire step2 module of a single machine.

6.1 Improvements against the first design

Although the first generation of the architecture implemented in chapter 5 has significant performance drawbacks, the results concerning the speed report and the throughput were encouraging. So, we continued with the design of the second generation taking into account that this architecture should provide a reliable matching scheme. In addition, more parallel machines should fit in the design in order to exploit more bandwidth from the Rocket IO transceivers and increase the throughput. As it is discussed in chapter 5, this can happen only if a single machine uses less blocks of RAM. Finally, the different components of this generation should be synchronized easily in order for the system to be reliable. This can be achieved if the extension process, which rarely occurs in a database search, is performed independently of the comparison unit. This notion, in connection with the fact that the extension process is serial making the use of hardware useless, motivated the developers of this architecture to exploit the PowerPC embedded processor of virtex4 family and couple it with the reconfigurable implementation of the rest parts of the algorithm. The use of a general purpose processor is not new in the reconfigurable hardware design as it is widely used either in the controlling the reconfigurable logic or in the execution of program code that cannot be efficiently accelerated [27]. Figure 6.1 shows the second generation of the proposed architecture involving the PowerPC processors.

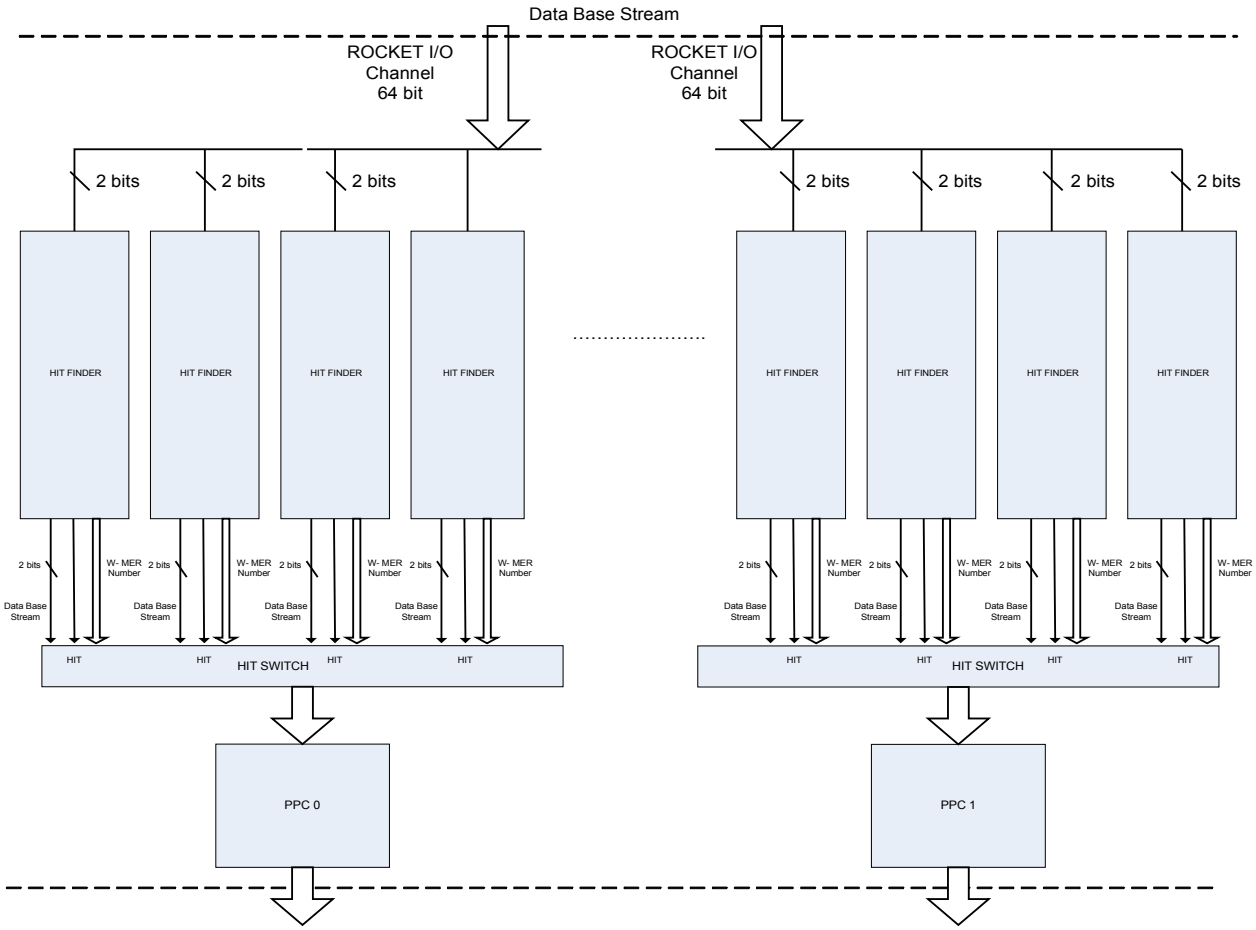

Figure 6.1 The generation2 architecture

6.2 Effective matching scheme

Figure 6.2 shows the architecture used to perform the exact matching of the w-mers to find a hit. Every w-mer, consisting of 12 letters or 24 bits, splits into two equal parts. Both of these parts are used as indexes to initialize the same places of both memories with the value '1', while the other places of the memory are initialized with zero values. The idea of using as narrow memories as possible as well the splitting of large strings into smaller was given by [49].

As the database stream enters into the structure shown in the left of figure 6.1 to form a length 12 word each time, half of this word indexes the one memory and the rest indexes the other one. In case of a hit, the output of both memories is 1 so that the logic AND of the memory outputs indicates a hit.

However, this matching architecture is not sound by itself. In other words, it is probable for this unit to produce "wrong hits". If, for example, a w-mer consists of the sequence: *ATCGCC-GATTGC,* the memory places indexed by the sequences *ATCGCC* and *GATTGC* contain the value 1. Apart from the above sequence which will obviously produce hit, hits will be also produced by any combination of these two subsequences, such as *ATCGCC-ATCGCC* (repetition of the same subsequence) and *GATTGC-ATCGCC* (both subsequences present in another turn). The process validating the appropriateness of an indicated hit happens during the third step of the algorithm inside the Power PC.

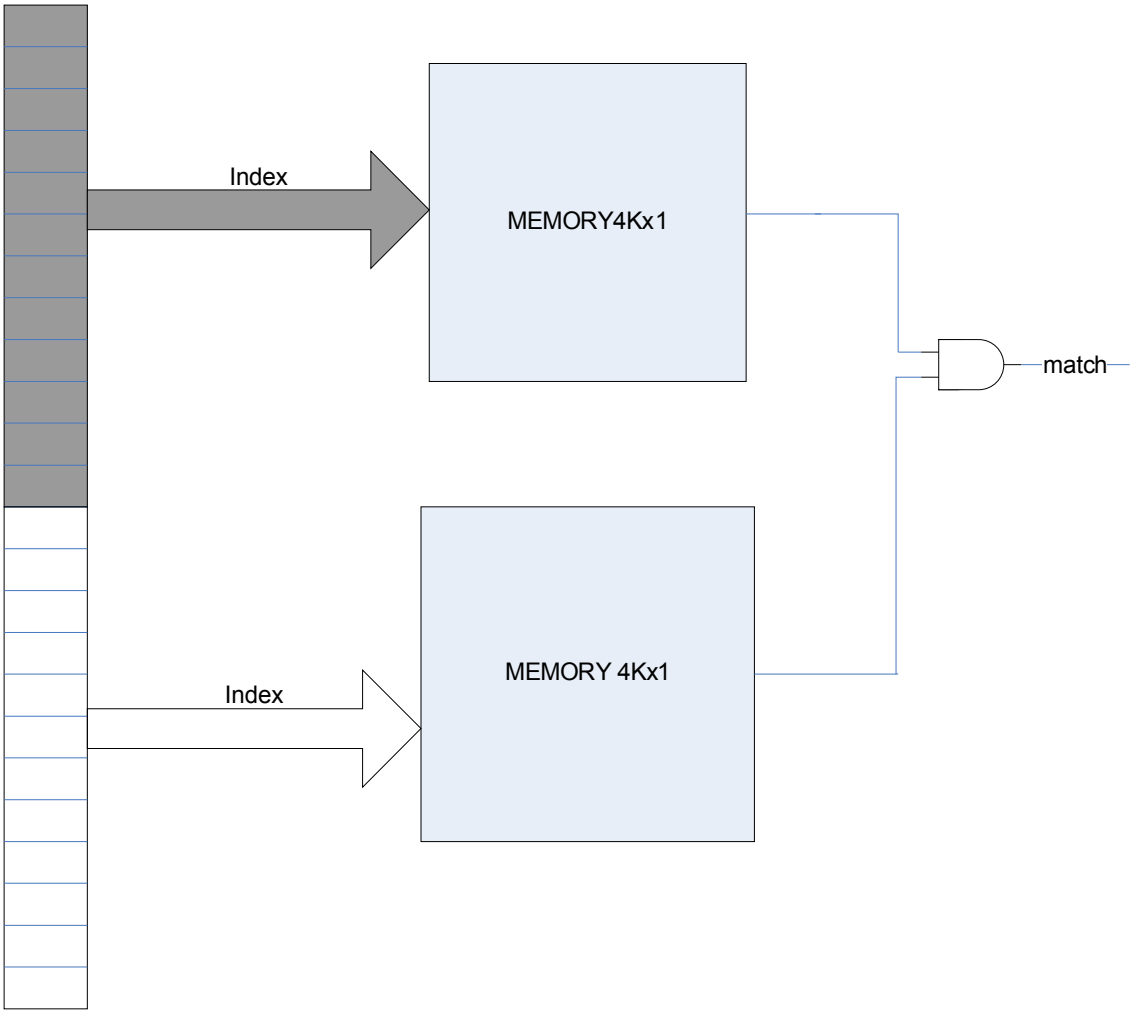

Figure 6.2 The matching scheme of generation2 architecture

So, it is clear that the proposed architecture is useless without benchmarking. For this reason, the performance of this unit has been emulated in Python scripting language. Two experiments have been contacted to measure the number of hits

produced in several executions of this emulation. In both cases a small query has been used from the database *month.nt*. As input databases have been used the *ecoli.nt,* a 4M letter database, and the *month.nt,* a 370M letter database. The results of the emulation are shown in table 6.1

Database size	***Right hits***	***False hits***	***Total hits***	***Hits to Database letters***
4M	***185***	***67,467***	***67,652***	***1.7 %***
370M	***24,858***	***5,986,112***	***6,010,970***	***1.6%***

Table 6.1 Hit profiling of a 1000 letter long query against a couple of databases

As it is shown in table 6.1, the number of false hits is much greater than the number of the number of right hits(about 300 times greater in both cases). However, as the total number of all hits is the 1.7% of the Database size and taking into account that the program in the Power PC decides in the first few cycles if the hit is right or false, this overload is negligible for the total program execution.

Of course there is an improvement to the proposed unit, if the memory indexes contain one more letter, without increasing the length of w-mers, so that there are regions overlapping with each other. For example the w-mer *ATCGCC-GATTGC,* successfully indexes the memory positions *TCGCCG, CGATTGC.* After this improvement, the new total number (6,048)of hits in the experiment with the four million letter database, is the 10 % of the respective number from table 6.1. This means that with this improvement the number of the false hits has been reduced by 90%.

6.3 Architecture of a single machine

Figure 6.2 shows the architecture of that part of a single machine used to perform the hit searching. In this figure, the components designed as a contribution of this thesis are in gray. Apart from the comparison unit described in the previous

section, the unit of figure 6.2 consists also of an input buffer, 2 bit wide and 1K entry long, from which the input database stream passes as soon as it enters into the system. Every cycle, new data are written to the memory while previously come data are read and provided to the comparison unit where a shift register, 2 bit wide and 12 bit long, creates database words of length 12 to be compared with the stored w-mers. In the same time, the data read from the Future Memory are also provided to the output of the unit after a relevant delay. Finally there is also a counter counting up one every cycle of the "normal mode", i.e. not in stall mode. Data are delayed in the output because it is necessary that in case of a hit the output has the letter caused that hit.

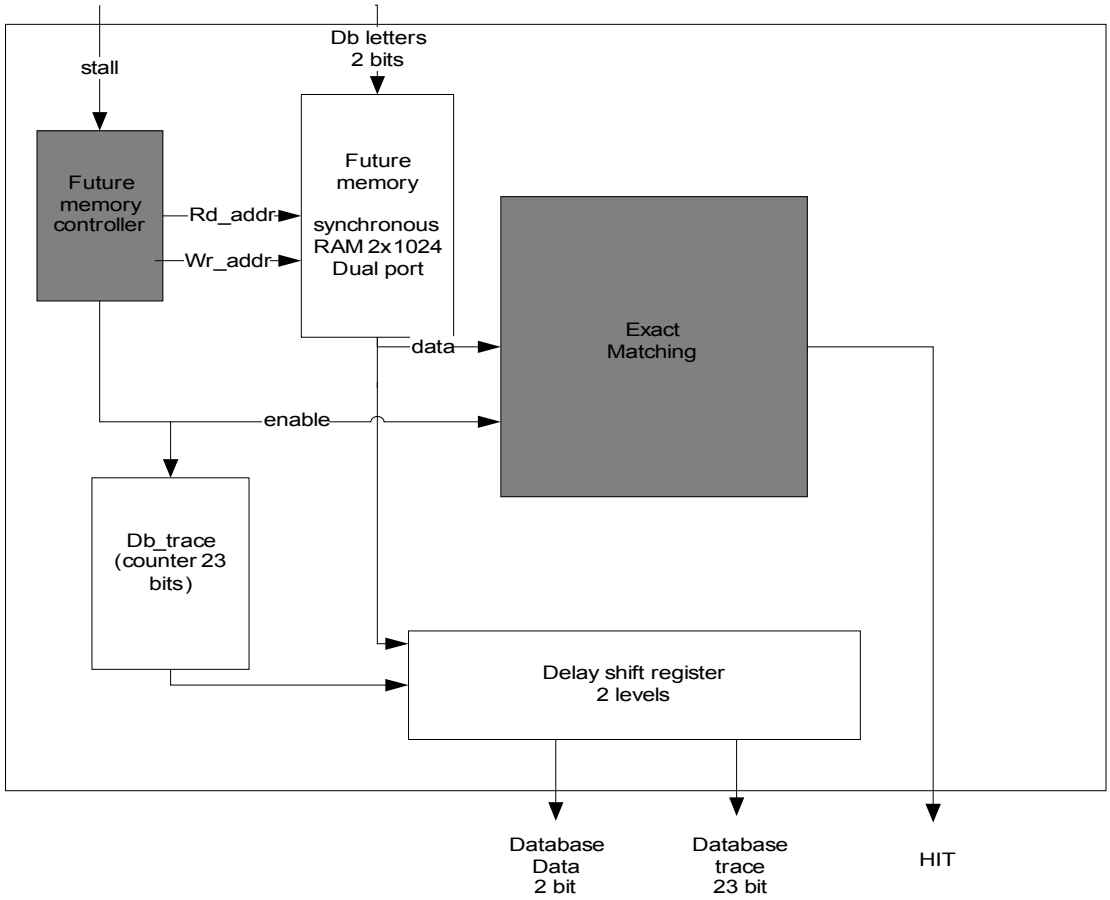

Figure 6.2 The step2 unit of second architecture

Middle Stage Unit, shown in figure 6.3, is the unit which controls that the correct data enter into step3 each time. In this figure, the control unit designed by the author of this thesis is in gray. The primary element of the Middle Stage Unit is a memory 16 bit(i.e. 8 letters) wide by 1K entry long, called history memory, which stores all the recently passed database letters. Its width helps Power PC to have faster access to the memory data.

Middle Stage Unit does not notify Power PC to start the extension of a hit unless the maximum necessary data are stored in the history memory. For a given query, the worst case is that a hit occurred either with the first or with the last w-mer of the query. In our case of a 1000 letter query, Middle Stage Unit guarantees that 1000 letters preceding a hit and other 1000 letters following that hit are located in the history when step3 is notified to perform its extension.

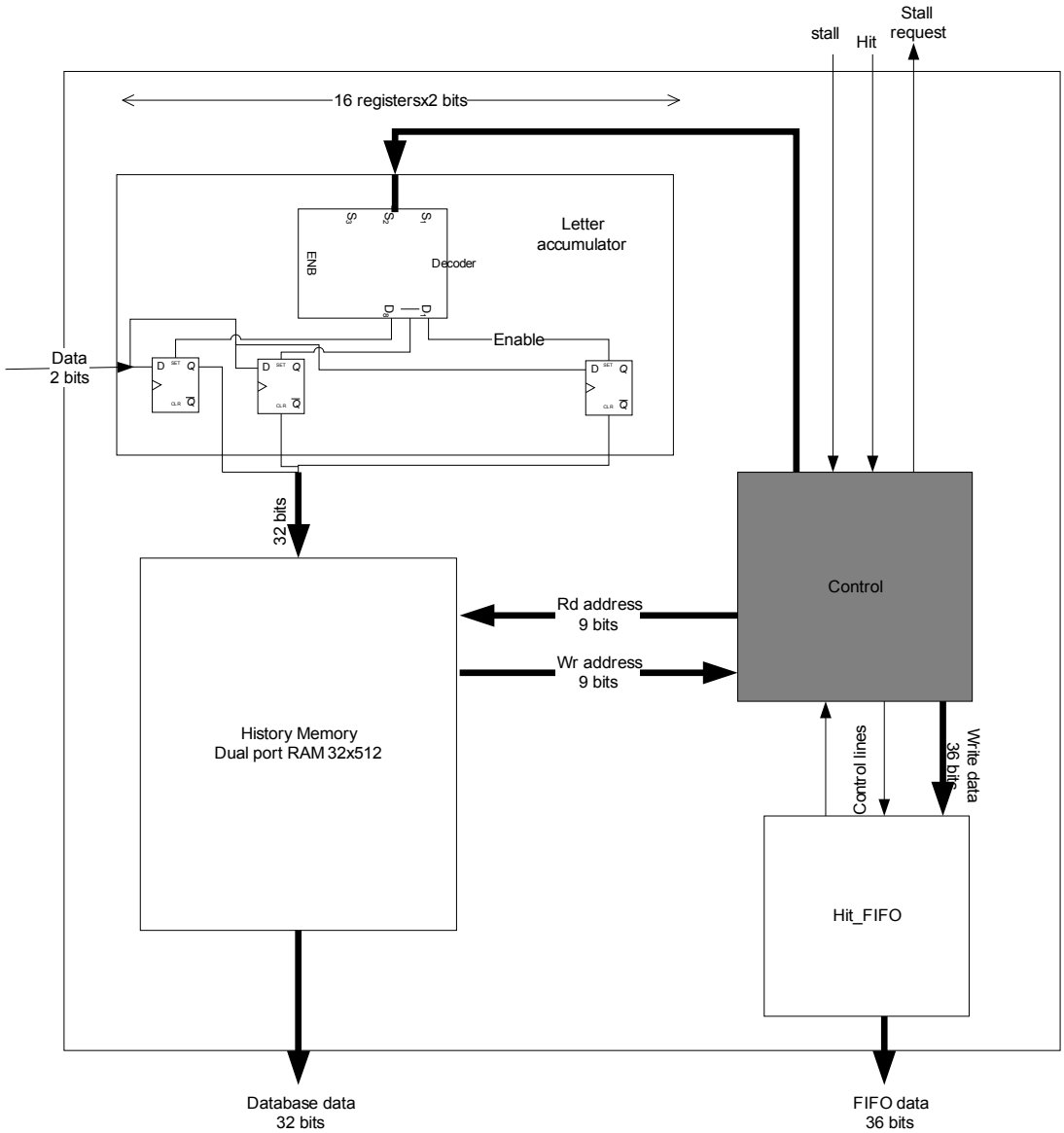

Figure 6.3 The middle stage unit

6.4 Control Units

In section 6.3 the data path of the design has been mainly discussed. In the current section, the control units which were developed in this thesis will be described.

6.4.1 Future Memory Controller

In the design of the control unit of Future Memory, one has to take into account the synchronization issues faced in the first generation of the architecture. In fact, one of the main problems comprised the fact that during the first cycle of a stall, a memory data entry was overwritten without previously read. The example of figure 6.4 illustrates better this issue.

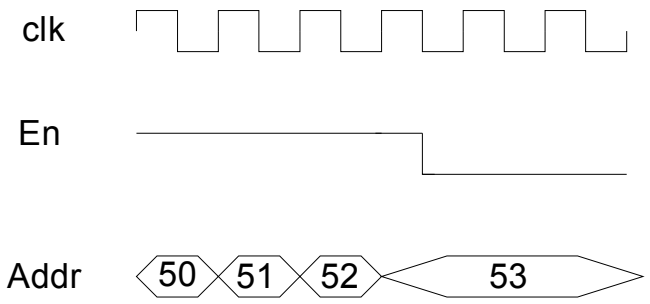

Figure 6.4 The Future Memory overwriting issue of first generation

In this example, Addr was the read and write memory address while the En signal was the memory write enable as well the enable signal of the rest units, such as the comparison unit. When En was high, the memory was written in the same address it was read. However, in the cycle which En transits to zero, address 52 has been already read and written, but the old value of address 52 gets lost as the entire system stalls. This happened because the memory's read and write addresses were the same. However, in this architecture we assume that the read address is always preceding the

write one by one place. So, in this way it is guaranteed that not a single memory value will be overwritten without previously read.

Another innovation of the Future Memory of the second generation is the way addresses are produced, as well the fact that it is the unit which provides the enable signal for the rest step2 components. In this architecture, when a machine stalls, the controller of the Future Memory continues to provide addresses, so that the memory continues to provide data, no matter if any other unit needs them. In fact, the middle stage unit may need these data but the controller of the future memory does not have to take care of it. Of course, at the end of a stall the addresses provided should be consistent with the ones before the stall. For this reason, the FSM of figure 6.5 has been introduced to provide the correct addresses in each time of execution.

In this FSM, there are two 10 bit variables, the tmp and the stored, which are responsible for the right memory addressing. The output address of the controller is the value of the variable tmp. During the state 0, both variables have the same value and increase in the same way. When a stall comes, i.e. when the en value falls, the tmp value continues to increase while the stored value does not. At the end of the stall, the value of the stored variable is copied to the tmp and the controller comes to the state 0 again.

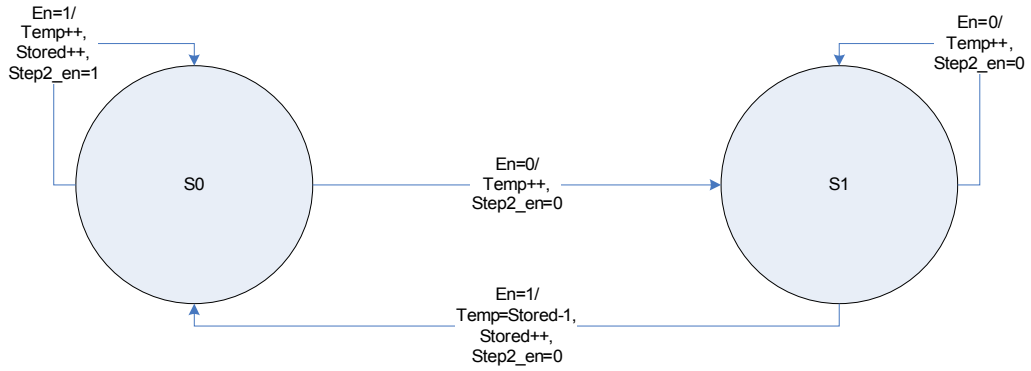

Figure 6.5 The FSM synchronizing the Future Memory address generation

6.4.2. History Memory Controller

Because of the completely different performance on read and write operations of the history memory, it has been selected to develop two different control modules, one for each operation.

6.4.2.1 History Memory Write Controller

This controller is responsible for providing history memory wright addresses as well for ensuring the correct data parallelization.

As it has been described in previous sections, the history memory consists of 16 bit data while the incoming data are 2 bit wide. For this reason, the incoming serial data stream needs to be transformed in 16 bit wide words. For readability purposes, figure 6.6 contains the module of figure 6.3 which parallelizes the incoming data stream. All the flip flops in this unit have the same input, and the write controller chooses via the decoder to which flip flop data will be written, taking into account that the older data is written to the most significant positions.

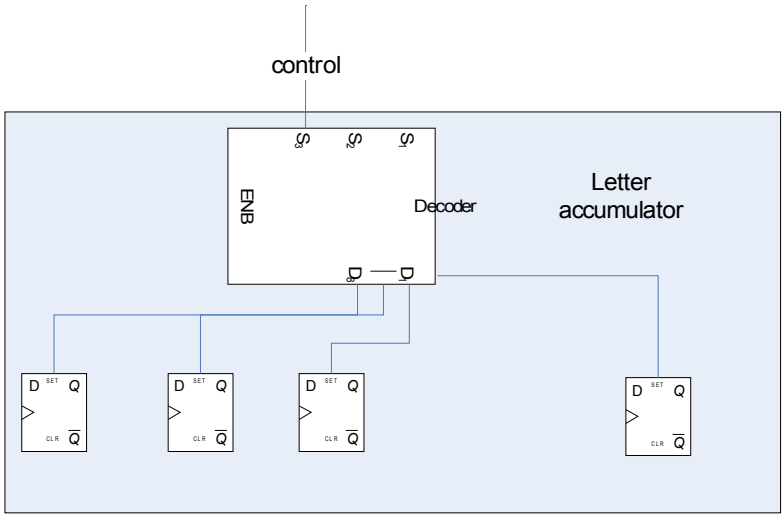

Figure 6.6 The unit parallelizing the incoming data

For the addressing of the history memory and the unit of figure 6.6, a module has been developed that provides addresses for both the history memory and the

parallelization unit as well write enable signals for the history memory and the entire unit of figure 6.3. This module consists of two variables, one 3 bit wide and another one 10 bit wide, which are the respective addresses for the parallelization unit and the history memory. The 3-bit variable constantly decreases by one, from 7 to 0. As soon as this variable underflows and starts again the counting-down from 7, the history memory write enable signal is set for a cycle while in the next cycle, the 10-bit variable increases and prepares the next write address of the history memory. The example of figure 6.7 further illustrates the functionality of this module. As it is clear, the 3-bit Parallelization_address variable decreases and when it starts again from 7, the History_memory_write_en signal is high for a cycle. This means that in the following cycle data will be read in the address 58 of the history memory. In the same cycle (i.e. when the value of the 3-bit variable is 6), the value of the 10-bit History_memory_address is updated and waits for the next memory write access.

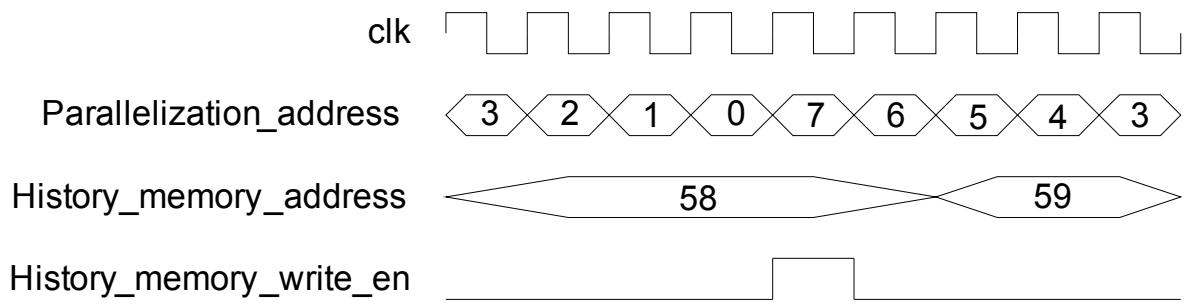

Figure 6.7 Performance of the module addressing the history memory

However, the module described above is not able to control successfully all the write operations. The reason is the stall mode when data continues to enter into the History Memory, but it has to be discarded in the first cycles after the stall when the incoming data are already in the memory. So, it is necessary the existence of another one unit with the same functionality with the module described above regarding the address generation, but without any write enable output. This unit, called backup counter, helps the FSM of figure 6.8 to keep track of the memory entries which do not need to be re-written during the first cycles after the stall.

The FSM of figure 6.8 contains four states. In state0, there is no stall and both counters count, with the generator counter providing normally addresses and enable

signals. When stall is set, the backup module stops counting while the generator counter continues its performance. In this way the system remembers the last address written during the normal mode of the execution. In case that the stall has a duration of more than 1000 cycles, the system stops writing data to the history memory and the generator counter stops providing write enable signals and new addresses. This happens because of the Future Memory size which holds up to 1024 letters following the currently compared word of size w. When the system is either in state1 or in state3 and the stall ends, it goes to state2 where the generator counter stops and the backup one starts again counting. The system remains in this state until both units have both their 10 bit and 3 bit addresses equal. Afterwards, it goes to state0 where it continues its normal execution. Obviously, if a stall comes in state2, the system goes again to state1.

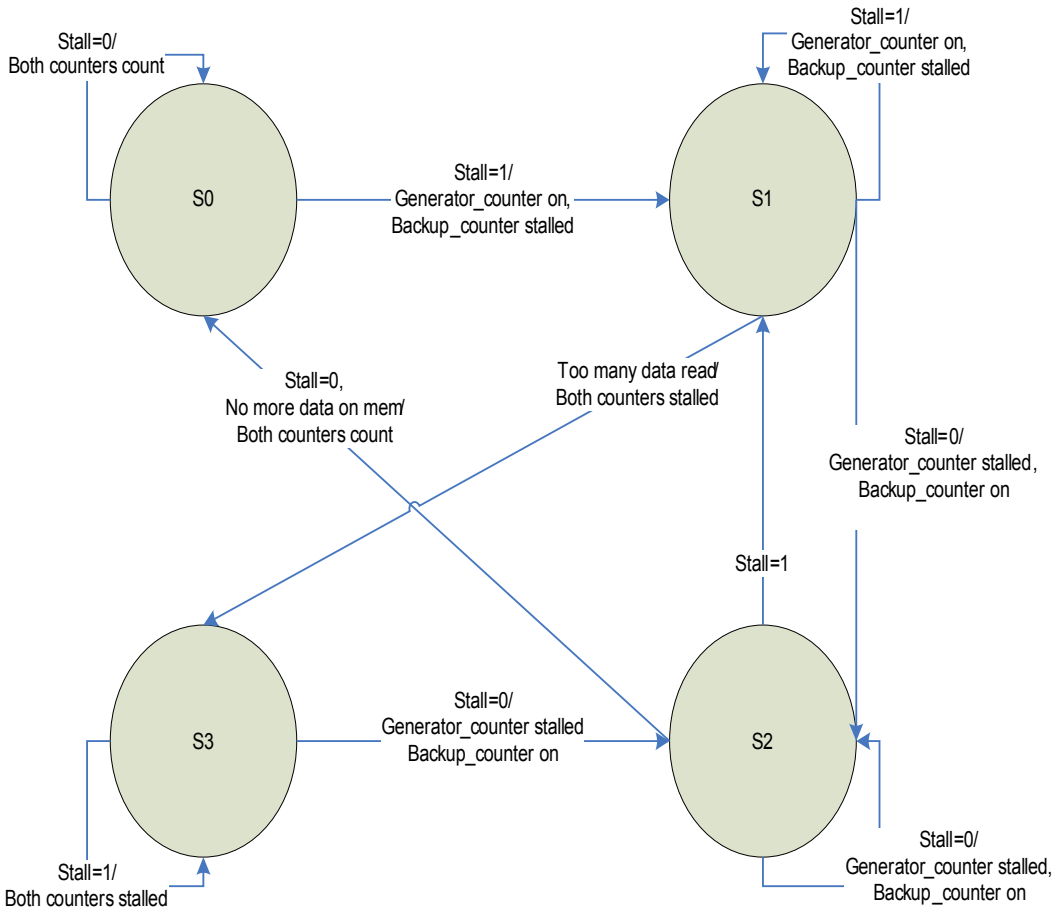

Figure 6.8 History memory write control FSM

6.4.2.2 History Memory Read Controller

In the design of the History Memory Read Controller, one has to take into account that the Middle Stage Unit starts sending to step3 History Memory values only when all the potentially necessary data for a hit extension are stored in the History Memory. In addition, the "incremental" data accumulation by step3 allows the designer to consider that the History Memory values may be discarded as soon as they are read.

So, in the History Memory it could be identified a frame of interest, which is the only region of the History Memory that the controller should deal with. Figure 6.9 represents a map of the History Memory where the aforementioned frame consists of the dark and the grey region and its borders are the pointers *start_address* and *end_address* respectively. Pointer *Hit_position* is the index value of the history memory where the last letter involved in a hit is stored. Finally the *Current Write Address* pointer is the most recently written memory address. The dark region of the frame denotes that this part of the memory contains all the relevant data, while the grey part denotes that these memory places have not been written by the appropriate data.

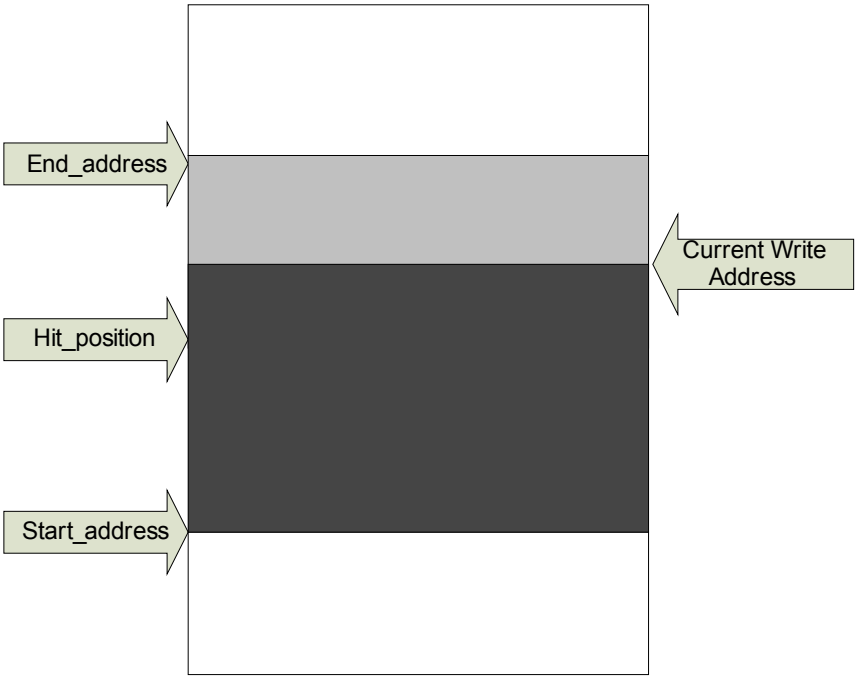

Figure 6.9 The interesting frames of History Memory

As the different regions of interest have been clearly identified in the History Memory, it is now easy for one to understand the performance of the read controller. When a hit is entered in the controller, the pointers are calculated. Then if there are "grey" regions, i.e. regions waiting for data, the controller waits until the entire frame of region gets "dark", i.e. all the necessary data are on the History Memory. Then, all the memory entries of the "dark" region are sequentially read, while this region decreases in size by changing the value of the *start_address* pointer as soon as the entry pointing to is read.

A nine-state Finite State Machine has been introduced to implement the control briefly described above. For the better communication with the step3, this module has also handshaking signals and an additional state in order to notify the step3 that data are ready to be transferred as soon as step3 asks for it. In the rest of the section, this FSM is described in detail.

For the reader's convenience this FSM will be presented in two parts. Figure 6.10a shows the first half of the machine. The FSM is initially in state0 and remains there until a hit comes. As all hits are recorded in the hit FIFO, the machine just reads the *empty* signal of the FIFO to determine whether a hit has been identified. When the FIFO is found to contain data of a hit, the machine changes state and sets the read enable signal of the FIFO for one cycle. In the next cycle the machine is in state6 waiting for the data of a hit to arrive from the FIFO. As soon as the data arrives, the start and end addresses of the frame of figure 6.9 are calculated so that the former is 126 entries less and the latter is 126 greater than the History Memory address where the last letter produced the hit is stored. Number 126 has been placed taking into account that the History Memory are 16 bit(8 letter) wide and the design of the system assumes that 1000 letters before and 1000 letters after the hit region should be read. Apart from these variables, some other variables are initialized helping to send data incrementally when necessary, and the machine goes to state2. The machine remains in this state as long as there are data missed from the History Memory, i.e. there are grey regions in the frame of figure 6.9. When all the relevant data are stored in the History Memory, the machine notifies the external environment that it is ready to begin the data transfer procedure and goes to state8. It remains to this state until the

external environment, i.e. the step3 unit, notifies that it is ready to read data from this machine.

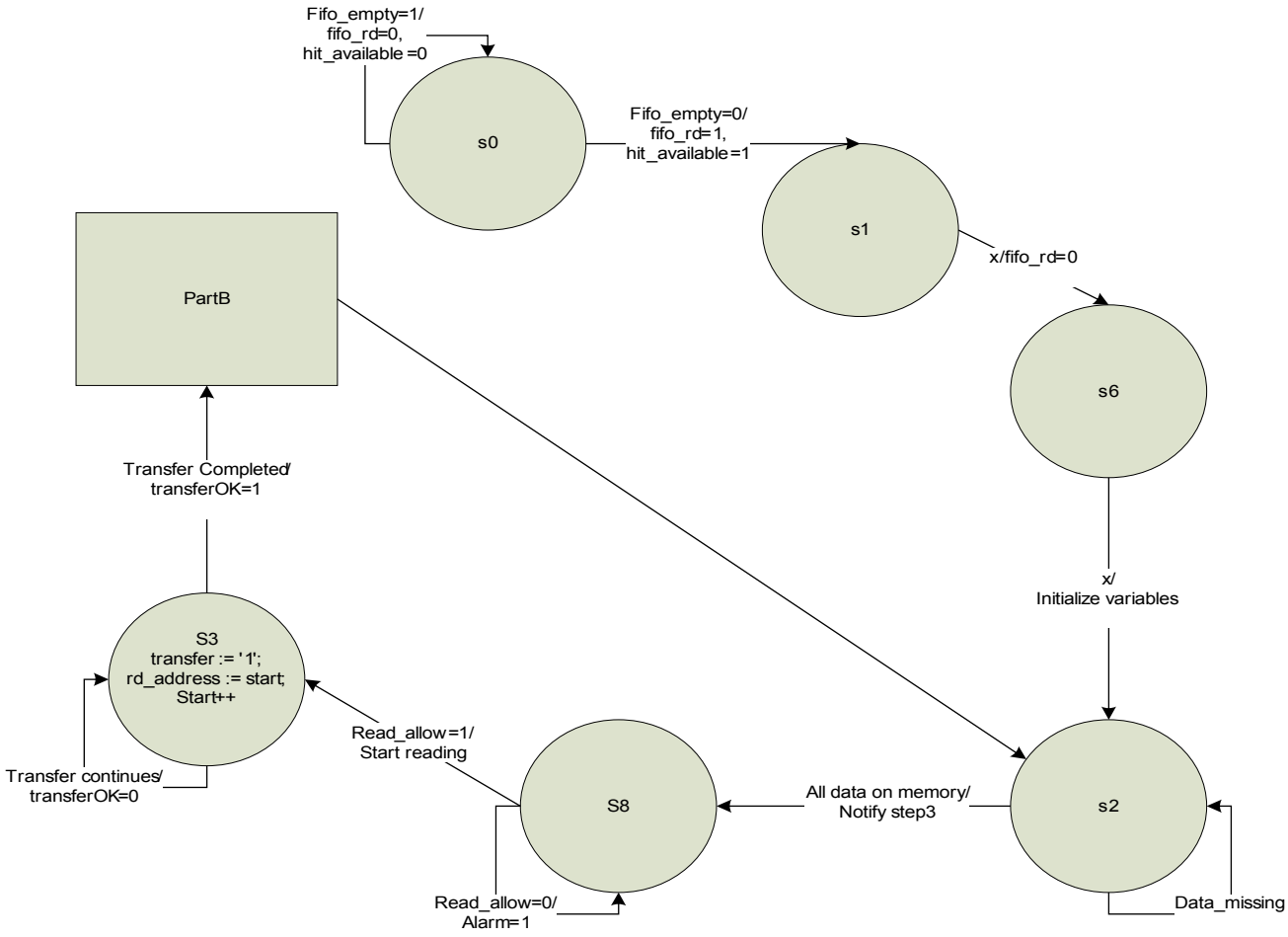

Figure 6.10a PartA of the FSM controlling the read operations of History Memory

When this happens, the machine goes to state3 and starts to provide sequential read addresses for the History Memory starting from the *start address* and ending when the *end address* is reached. In the same time, the *start address* pointer also increases so that the *frame* discussed earlier gets narrower. So, one could say that the read address of the History Memory is the *start_address* variable and the read addresses are provided for as long as the dark region of figure 6.9 does exist. When the data transfer ends, the signal *transferOK* is set for one cycle and the machine goes to state4 where it remains for as long as the FIFO containing the hit information is empty.

The rest of the machine is shown in figure 6.10b. From state4, when a hit is found in the FIFO, the machine goes to state7 and in the next cycle the variables helping with the incremental data transfer are updated while the machine goes to state5. The performance of state7 may look similar with the performance of state6 previously discussed. However, the difference is that in the state7 the *start* and *end* addresses are not calculated. Rather, it is state5 which takes care for producing the frame of figure 6.9. Initially, state5 checks if the current hit is not in the "neighbor" of the previous hit, i.e. it has been produced after more than 2000 letters from the previous one so that no part of the data concerning its extension has been already transferred. In this case both the *start* and *end* addresses are set to a distance of 126 History Memory entries each, such as in state6, of the raw containing the letters involved in the hit. Otherwise, it is examined if the newly come hit has its last letter in the same 8-letter word with the previous hit. In this case, not a single line should be read from the memory, the *transferOK* signal is set for a cycle and the machine goes again to state4. In any other case the *frame* of the data to be transferred is calculated taking into account that previously transferred data are not

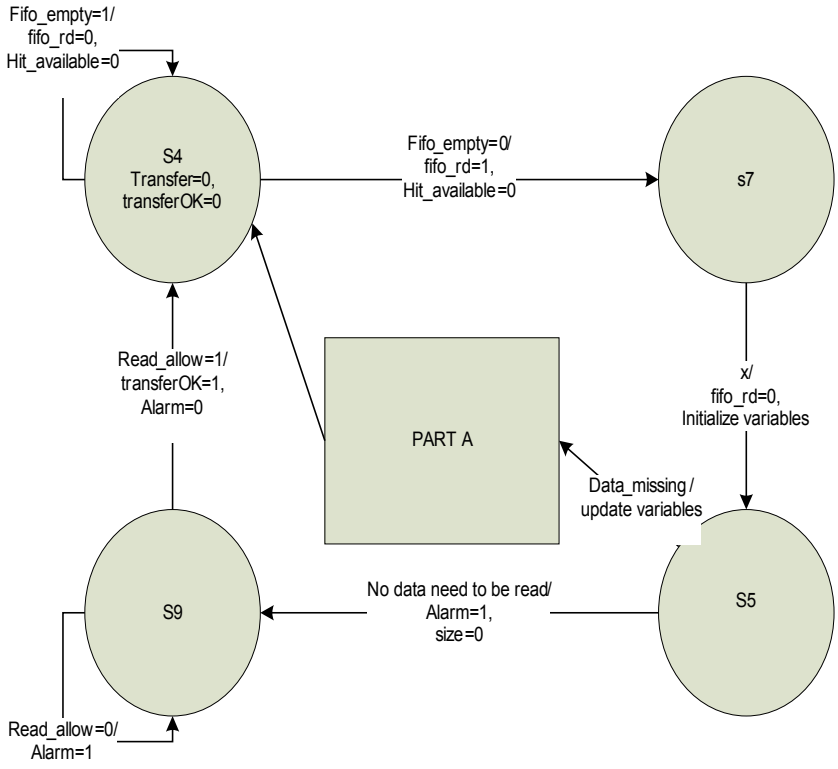

Figure 6.10b The rest part (part B) of the FSM controlling the read operations of History Memory

transferred again. So, the *start* address of the frame remains unchanged since the last time it has been updated in the step3 of the previous hit data transfer, while the *end* address is calculated to be 126 entries greater than the entry containing the last letter involved in the hit. Then the machine goes to step2 and the procedure from this point until the end of the data transfer has been already described.

6.5 Verification

For the simulation of a single hit finder unit of fig. 6.1, it has been used a database with length of 10,000 (part of the database month.nt) and a query of 1000 letters.

As a validator of this simulation, a python script has been developed which reports the places where a hit is found when database letters are compared against the w-mer data. Figure 6.11 contains the flowchart of this validator emulating the matching scheme of our architecture. Initially this script scans the query and creates the wmer list. Based on this list, another list is created containing all the words which are produced when every wmer is split in two equally sized substrings. Afterwards, the database is read and for every read character a word of size w is created. Then the system checks if both halfs of this word are contained in the previously created list. If so, the position of the word in the database is reported to the output.

A sample of the execution of this script is shown in table 6.5. where for readability purposes we present 8 reported hits.

Hit position
4471
4472
6463
6464
7483
7484
8487
8488

Table 6.5 A sample of the execution of the validator hit

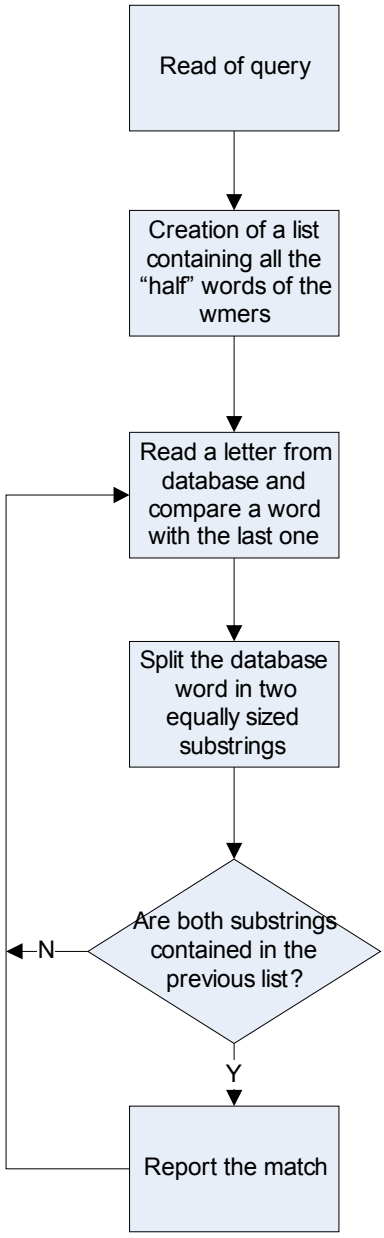

Figure 6.11 The flowchart of the validator software

In the simulation environment (Modelsim 6.0) we should check if hits occur in right positions (considering an offset due to initializations) and we should verify that all the appropriate data are on the memory. Table 6.6 shows the start and end addresses of the history memory frames that are read for every hit as well the number of the entries sent to the output. Also, all of these data are contained in the same data

db_position (validator)	*db_position (simulator)*	*Start read address*	*End read address*	*Number of entries read*
4471	7642	415	29	127
4472	7643	29	29	0
6463	9634	28	154	127
6464	9635	154	154	0
7483	10654	153	217	65
7484	10655	217	218	2
8487	11658	218	280	63

Table 6.6 Results from modelsim simulations

From the data shown on table 6.6 it is clear that different sizes of data are sent to the output for every hit. If hits occur in neighboring places, less memory entries are read as there are overlappings with the data from previous hits. At this point, the user is reminded that the history memory is 32 bits wide and 512 bits deep. When end address is less than the start address, that means that as the address generator counter overflowed, and it started from zero again.

6.6 Performance Comparison with the first generation

Apart from the reliable matching scheme of the second generation, - the reader is reminded that the first generation did not work properly in case of collisions-, there are also throughput advantages in the second generation. Table 6.5 shows the predicted throughputs of both generations and summarizes their features.

Generation	*Number of Parallel Machines*	*Speed (MHz)*	*Width of Data Stream (characters)*	*Predicted Throughput (characters per second)*
Generation I	69	100	69	$6{,}924\ 10^6$
Generation II	128	64	128	$8{,}192\ 10^6$

Table 6.7 Speed and Throughput of the two versions of the architecture.

From the results of table 6.7 it is shown that there is an 18% increase in the predicted throughput of generation2. However, this increase will grow in the future,

because it is the control between the reconfigurable design and the PowerPC that is responsible for the very low MHz speed of generalization2. In the future, it is estimated that this unit will be optimized and then the speed of generation2 system will be at least 100 MHz.

6.7 Summary

In this chapter the second generation of the BLAST machine has been presented. Initially, a new matching scheme has been proposed for step2. As PowerPC has been selected to implement all the extensions, only step2 has been implemented in hardware. In this chapter, we have not discussed in detail the development of the datapaths of the involved units, but we concetrated on the development of the several control units of this architecture. Finally the post-place and route simulation of this part of the system has been shown and the predicted throughput has been calculated, while a performance comparison with the first generation of the architecture has been made.

Chapter 7 - Performance Comparisons

In this chapter, a performance comparison will be made between the second generation of TUC BLAST system and the results published by other implementations, as well as runnings of the NCBI software in general purpose computers currently (May 2006) available at the Microprocessor and Hardware Lab.

7.1 NCBI runnings on different machines of MHL

Measurements of the NCBI software have been made on conventional computers and on the new machine, with identical queries. Runs of blast-2.2.12 were performed on a 2GHz Xeon with 2GB main memory running SUSE 9.1 Linux and the CPU usage was profiled. Searches of two queries of 1000, 5,000 and letters against five GenBank databases of several sizes were executed at the 2GHz Xeon and measured. The same experiment was repeated with a Intel Pentium M 1.7 GHz with 1 GB main memory running Windows XP professional and an Intel P4 2.66 GHz with 4 GB main memory running Windows XP. For Computers running Windows Intel VTune Performance Analyzer 7.2 was used and every measurement repeated 5 times. Results of these experiments are respectively on Tables 7.1, 7.2, and 7.3. The averages in the tables are arithmetic averages.

Data base name	***Database Size (characters)***	***Run Time (sec) Query 1000***	***Run Time (sec) Query 5000***	***Throughput Query 1000***	***Throughput Query 5000***
ecoli.nt	4,662,239	0.024	0.044	194.25	$105.95\ 10^6$
drosoph.nt	122,655,632	0.482	0.792	$258.33\ 10^6$	$154.86\ 10^6$
month.nt	386,242,580	1.753	2.962	$220.56\ 10^6$	$130.39\ 10^6$
env_nt	1,061,221,997	1.190	8.009	$891.63\ 10^6$	$132.50\ 10^6$
igSeqNt.ftptemp	44,419,359	1.397	0.325	$31.77\ 10^6$	$136.61\ 10^6$
Average	**323,840,361**	**0.968**	**2.426**	**$319.25\ 10^6$**	**$132.08\ 10^6$**

Table 7.1 Measurements on XEON 2 GHz / Linux

DataBase name	*Database Size (characters)*	*Run Time (sec) Query 1000*	*Run Time (sec) Query 5000*	*Throughput Query 1000*	*Throughput Query 5000*
ecoli.nt	4,662,239	0.045	0.132	102.85 10^6	35.32 10^6
drosoph.nt	122,655,632	0.364	0.56	337.32 10^6	219.02 10^6
month.nt	386,242,580	1,303	2.156	296.50 10^6	17914 $10^{6.}$
env_nt	1,061,221,997	3.670	10.57	289.19 10^6	100.39 10^6
igSeqNt.ftptemp	44,419,359	0.174	0.416	255.43 10^6	106.77 10^6
Average	**323,840,361**	**1.111**	**2.767**	**157.86 10^6**	**128.13 10^6**

Table 7.2 Measurements on Intel M 1,7 GHz / Windows XP

DataBase name	*Database Size (characters)*	*Run Time (sec) Query 1000*	*Run Time (sec) Query 5000*	*Throughput Query 1000*	*Throughput (characters/sec) Query 5000*
ecoli.nt	4,662,239	0,039	0.144	118.45 10^6	32.37 10^6
drosoph.nt	122,655,632	0.309	0.526	396.32 10^6	233.18 10^6
month.nt	386,242,580	1.022	1.75	378.10 10^6	220.71 10^6
env_nt	1,061,221,997	3.200	10.982	331.63 10^6	96.63 10^6
igSeqNt.ftptemp	44,419,359	0,160	0.406	277.40 10^6	109.40 10^6
Average	**323,840,361**	**0.946**	**2.761**	**300.38 10^6**	**138.46 10^6**

Table 7.3 Measurements on an Intel P4 2,66GHz / Windows XP

7.2 Performance Comparison

Table 7.4 shows the throughput of several system including the second generation of TUC architecture and figure 7.1 has these data in bar form for a better visualization of the results.

System	*Predicted Throughput (Mega characters/sec) Query 1000*	*Predicted Throughput (Mega characters/sec) Query 5000*
2GHz Xeon	319.25	132.08
1,7 GHz Intel M	256.26	157.86
2,66 GHz Intel P4	300.38	138.46
IBM single chip	187.62	14.23
IBM System	1,201.20	159.36
TUC Generation II	8,192.00	8,192.00

Table 7.4 Systems Throughput

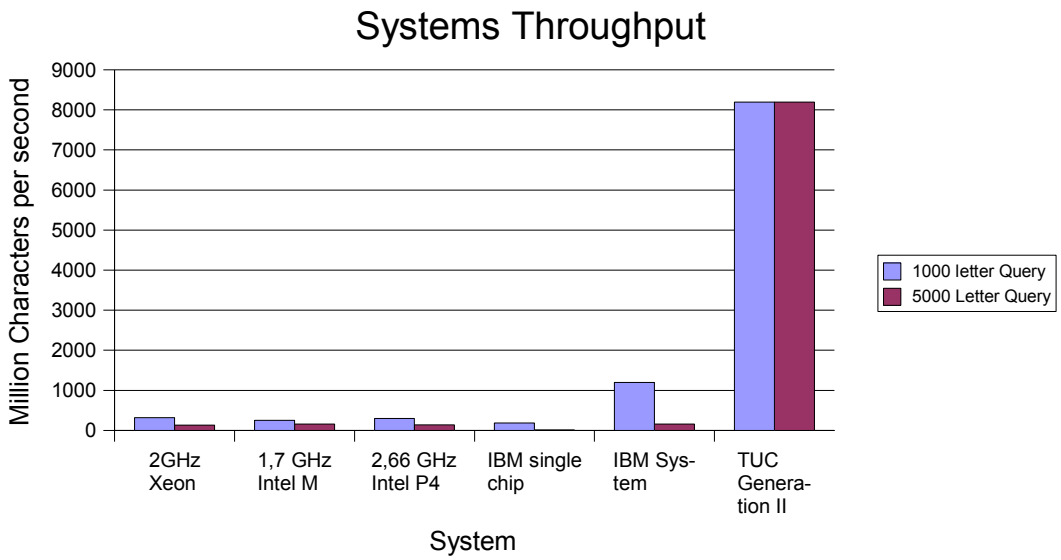

Figure 7.1 Systems Throughput Bar

Finally, table 7.5 displays the speedup of generation2 of TUC architecture when it is compared with the systems previously stated.

	SpeedUp of TUC Architecture Generation II Query 1000	*SpeedUp of TUC Architecture Generation II Query 5000*
2GHz Xeon	25.66	62.02
1,7 GHz Intel M	31.96	51.89
2,66 GHz Intel P4	27.27	59.16
IBM single chip	43.66	575.68
IBM System (16 chips)	6.81	51.40

Table 7.5 TUC Architecture Generation II SpeedUp

Chapter 8 : Conclusions – Future Work

The results obtaining from the implementation of the BLAST algorithm are encouraging regarding as they are compared with the ones reported by other systems running the BLAST software. However, there are some things necessary for the further improvement of the performance. Further improvements of this architecture can be achieved. The speed of the reconfigurable hardware can be increased through circuit-level optimizations (better placement, etc.). However, any speed improvement of the reconfigurable design will not improve system speed as the Power PC is at present the bottleneck. However, a good tradeoff can be achieved with computational load reduction with the creation of a hardware filter in the switch unit to reduce the Possible Hit Probability. The complete solution to the I/O problem is also due to be developed. Although from actual measurements of RocketIO throughput there is no I/O problem on the FPGA side, simple solutions such as PCI will not work and a special-purpose interface needs to be built. Finally, the query size should be extended in order for medium and large size queries being covered and then implementations of other programs of the BLAST family are worthwhile to be explored.

References

Book Chapters

[1] J. Meidanis and J.C. Setubal, *Introduction to Computational Molecular Biology*, PWS Publishing Company, 1997, ch. 3.

[2] P. G. Higgs and T. K. Attwood. *Bioinformatics And Molecular Evolution*. Blackwell Publishing, 2005, ch. 1.

[3] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*, 3rd ed., San Fransisco: Morgan Kaufmann Publishers, 1990, ch. 1-5

[4] A. Aho, R. Sethi and J. Ullman, *Compilers Principles, Techniques and Tools*, World Students Series Edition, Prentice Hall, ch:2, 3, 7

[5] S. Sjoholm and L. Lindh, *VHDL For Designers*, Prentice Hall, 1997, ch: 8-9

[6] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000, ch. 6-8

[7] B. Kernigham and R. Pike, *UNIX programming environment*, Prentice-Hall International INC, 1984, ch. 4

Web Locations

[8] http://www.ncbi.nih.gov

[9] http://www.ddbj.nig.ac.jp

[10] www.embl.org

[11] http://www.ebi.ac.uk/embl/index.html

[12] www.uniprot.ebi.org

[13] www.xilinx.com

[14] http://en.wikipedia.org

[15] http://www.timelogic.com/decypher_intro.html

[16] http://www.maths.tcd.ie/~lily/pres2/sld003.htm

[17] http://www.bioinfo.se/kurser/swell/fasta.html

Published Papers

[18] S. B. Needleman, and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins,", *J. Mol. Biol*., vol. 48, pp 443-453, 1970.

[19] T.F. Smith, and M.S. Waterman, "Identification Of Common Molecular Subsequences," *Elsevier J. Mol. Biol.*, vol. 147, pp 195-197, 1981

[20] W Pearson,. and D Lipman, “Improved tools for biological sequence analysis,” *Proceedings of the National Academic Science of the USA*, vol 85, pages 2444–2448, 1988

[21] S. Altschul, W. Gish, W. Miller, and E. Myers, “Basic Local Alignment Search Tool,” *Elsevier J. Mol. Biol.*, vol. 215, pp 403-410, 1990.

[22] W. Day, D. Johnson, D. Sankoff, “Computational Complexity of Inferring Phylogenies by Compatibility”, Systematic Zoology, 35(2),: 224-229, 1986.

[23] S. Henikoff, J. Henikoff, “Amino Acid Substitution Matrices from Protein Blocks”, Proceedings of the National Academy of Sciences of the USA, 89: 10915-10919,1992.

[24] D. A Benson et al, Nucleic Acids Res. 34(Database issue), D16-20 (2006)

[25] G. Cochrane et. al., “EMBL Nucleotide Sequence Database: developments in 2005”.*Nucleic Acids Research 34 (Database issue):D10-D5 (2006)*

[26] A. Bairoch et. al., “The Universal Protein Resource (UniProt)”, *Nucleic Acids Research 33*:D154-D159 (2005)

[27] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210. June 2002.

[28] Apostolos Dollas, Euripides Sotiriades, Apostolos Emmanouelides, "Architecture and Design of GE1, a FCCM for Golomb Ruler Derivation," IEEE Symposium on FPGAs for Custom Computing Machines, p. 48, 1998.

[29] D. Hoang, and D. Lopresti, “FPGA Implementation of Systolic Sequence Alignment,” in *Proc. of the 2nd International Workshop on Field-Programmable Logic and Applications*, Lecture Notes in Computer Science 705, 1992, pp 183-191.

[30] D. Hoang “Searching Genetic Databases on Splash 2,” in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, Napa, 1993, pp 185-191.

[31] J. Hirschberg, et. al. “Kestrel: A Programmable Array for Sequence Analysis”, in Proc. Int. Conf. Application-Specific Systems, Architectures and Processors, IEEE CS, August 19-21, 1996, pp: 25-34

[32] D. Lavenier, “Speeding up Genome Computations with a Systolic accelerator”, SIAM news, vol.31, No. 8, October 1998, pp: 1-7

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

[34] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, and P. Athanas, “A Run-Time Reconfigurable System for Gene-Sequence Searching,” In *Proc. 16th International Conference on VLSI Design*, New Delhi, 2003, pp 561 – 566.

[35] T. Oliver, B. Schmidt, and D. Maskell, “Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs”, in *Proc. of the 13th ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA)*, Monterey, 2005, pp 229 – 237.

[36] A. Darling, L. Carey, and W. Feng, “The Design, Implementation, and Evaluation of mpiBLAST”, *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference & Expo*, San Jose, CA, June 2003

[37] W. Feng, “Green Destiny + mpiBLAST = Bioinfomagic,” *10th International Conference on Parallel Computing 2003* (ParCo’03), Dresden, Germany, September 2003.

[38] K. Muriki, K. Underwood, and R. Sass, “RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation,” in *Proc. 19th IEEE International Symposium Parallel and Distributed Processing (IPDPS)*, Denver, 2005, pp 196b-196b

[39] R. Luethy, and C. Hoover, “Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms,” *BIOSILICO*, Vol. 2-1, pp 12-17, Jan. 2004.

[40] E. Sotiriades, C.Kozanitis, and A.Dollas, "FPGA based Architecture of DNA Sequence Comparison and Database Search", Reconfigurable Architectures Workshop(RAW2006), 20*th IEEE International Symposium Parallel and Distributed Processing (IPDPS),* 25-29 April 2006.

[41] E. Sotiriades, C.Kozanitis, and A.Dollas, "Some Initial Results on Hardware BLAST Acceleration with a Reconfigurable Architecture," *5th IEEE Workshop on High Performance Computational Biology(HiCOMB2006), 20th International Parallel and Distributed Processing Symposium*, *2006 (IPDPS 2006)*, 25-29 April 2006

[42] E. Sotiriades, C. Kozanitis, G. Chrysos, and A. Dollas, "Rapid Prototyping of a System-on-a-Chip for the BLAST Algorithm Implementation," *17th International Workshop on Rapid System Prototyping, 2006 (RSP 2006)*, 14-16 June 2006, Pages 223-229

[43] M.C. Herbordt, T. VanCourt, Y. Gu, J. Model, and B. Sukhwani, "Single Pass Approximate String Matching on FPGAs," in *Proc IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, 2006.

[44] J. M. Arnold, D. A Buell, and E.G. Davis, "Splash 2". In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 316–324, 1992.

Papers Available on the web

[45] R. Luethy, and C. Hoover, "Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms," *BIOSILICO*, Vol. 2-1, pp 12-17, Jan. 2004.

[46] C. Sosa, Z. Tu, and P. Fast, "Some Practical Suggestions for Performing NCBI BLAST Benchmarks on a pSeries ™ 690 System," [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp0437.html?Open.

[47] Chen Chang "BLAST Implementation on BEE2", [Online]. Available: http://www.cs.berkeley.edu/~ejr/GSI/cs267-s04/final-projects/chenzh/BLAST_implementation_on_BEE2.pdf

Dissertations

[48] G. Adamopoulou, "Information Alert in Biological Sequence Databases", Master Thesis, Dept. Electronics and Computer Engineering, Technical University of Crete, Chania.

[49] G. Papadopoulos, "On the Way of Hashing for Low Cost Exact Pattern Matching", Master Thesis, Dept. Electronics and Computer Engineering, Technical University of Crete, Chania, 2005