IFERP
connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

Modified Hash Function for Searching of DNA Sequence

Keka Mukhopadhyaya
Assistant Professor,
CMR Institute of Technology, Bangalore-37

Abstract: -- **The problem of detecting similarities between new gene sequence and genomic sequences by search operation is fundamental to many research pursuits in bioinformatics. There are a number of software tools which take as input, a query string and a database string and return a list of approximate matches between the two, each with an associated alignment and numeric score. BLAST is a kind of software tool used to search genomic sequence. In BLAST the first stage is most time consuming and critical step. To reduce the time in this stage, filtration method is applied to reduce amount of computation data. This is done by using Bloom filter. A Hash function is associated with Bloom filter to generate the address where a particular data will be stored in the programming phase. In querying phase, the same Hash function has been utilized to check whether the data has been stored. There are different methods to implement Hash function. In this paper a new method to implement Hash function by using LFSR has been proposed. The generator polynomials used to implement LFSR is 32-bit and the usual serial implementation has been converted into a parallel using the CRC-32 code to speed-up the address generation step in programming and querying phase of the Bloom filter.**

Keywords—**BLAST; Bloom Filter; HashFunction; LFSR; CRC32.**

I. INTRODUCTION

Searching provides the basic ideas about the functionality of the new gene sequence. Genomic sequence consists of DNA molecules. DNA molecules are composed of four basic nucleic acids called Adenine (A), Guanine (G), Cytosine (C) and Thymine (T). A and T, and C and G naturally bond together to form pairs. So the main objective of the search operation is to find similarities between the new gene sequence, which also termed as query sequence and a particular genome sequence and this has lots of importance in bioinformatics. There are software tools that take as inputs, a query string and a database string (usually much longer than the query), and returns a list of approximate matches between the two, each with an associated alignment and numeric score. This numeric score provides valuable information about functionality of newly found gene sequences. The software tools can be broadly categorized as dynamic programming based alignment algorithm and heuristic algorithm. Development logic of dynamic programming-based alignment algorithms guarantees to find the important similarities. The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (sub-problems), and then combine the solutions of the sub-problems to reach an overall solution. In this approach, many sub-problems are generated and solved man times. The dynamic programming approach solves each sub-problem only once, thus reducing the number of computations. Once the solution to a given sub-problem has been computed, it is stored. The next time the same solution is needed, it is simply looked up. However, as the search space is the product of the two sequences, which could be several billion bases in size, it is generally not feasible to use a direct implementation. A frequently used approach to speed up this time-consuming operation is to use heuristics in the search algorithm. Heuristic refers to experience-based techniques for problem solving, learning, and discovery that gives a solution which is not guaranteed to be optimal. When an exhaustive search is impractical, heuristic methods are used to speed up the process of finding a satisfactory solution. In more precise terms, heuristics are strategies using readily accessible information to control problem solving in human beings and machines.

II. ALGORITHM

A. *Dynamic Algorithm*

Dynamic programming assign scores to insertions, deletions and replacements, and compute an alignment of two sequences that corresponds to the least costly set of such mutations. The result obtained from such algorithm provides minimal evolutionary distance or maximum similarities between the two sequences compared. In either case, the cost of this alignment is a measure of similarity; the algorithm guarantees it is optimal, based on the given scores [1 - 4]. But one of the drawbacks of the dynamic algorithm based programming is its computational requirements for searching large databases without the use of a supercomputer [5] or other special purpose hardware [6]. While comparing two sequences their similarity measure can be classified as either global or local. Global similarity algorithms (Needleman-Wunsch algorithm) optimize the overall alignment of two sequences, which include large stretches of low similarity [1]. The Needleman–Wunsch algorithm is an algorithm used in bioinformatics to align protein or nucleotide sequences, and is the first application of dynamic programming to biological sequence comparison. It is referred to as the optimal matching algorithm. Fig. 1 highlights the Needleman-Wunsch pairwise alignment of two sequences GCATGCU and GATTACA. Local similarity algorithms seek only small sub-sequences, and a single comparison yield several distinct subsequence

		G	C	A	T	G	C	U
	0	−1	−2	−3	−4	−5	−6	−7
G	−1	1	0	−1	−2	−3	−4	−5
A	−2	0	0	1	0	1	−2	−3
T	−3	−1	−1	0	2	1	0	−1
T	−4	−2	−2	−1	1	1	0	−1
A	−5	−3	−3	−1	0	0	0	−1
C	−6	−4	−2	−2	−1	−1	1	0
A	−7	−5	−3	−1	−2	−2	0	0

Fig.1: Needleman-Wunsch Algorithm

alignments; large regions do not contribute to the measure of similarity [2, 4, 7] The Smith–Waterman algorithm performs local sequence alignment; that is, for determining similar regions between two strings or nucleotide or protein sequences. Instead of looking at the total sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. Like the Needleman–Wunsch algorithm, of which it is a variation, Smith–Waterman is a dynamic programming algorithm. It has the desirable property that it is guaranteed to find the optimal local alignment with respect to the scoring system being used (which includes the substitution matrix and the gap-scoring scheme). The main difference to the Needleman–Wunsch algorithm is that negative scoring matrix cells are set to zero, which renders the (thus positive scoring) local alignments visible. Backtracking starts at the highest scoring matrix cell and proceeds until a cell with score zero is encountered, yielding the highest scoring local alignment. The fig. 2 shows the pair-wise alignment of two sequences ACACACTA and AGCACACA.

B. *Heuristic Based Programming*

There have been several approaches to accelerate bio-sequence similarity searches. Some of these approaches use special hardware, while others attempt to solve this problem in software using heuristics. Software based heuristic approaches include, FASTA (FAST-All) [8], BLAST (Basic Local Alignment Search Tool) [9] and BLAT (BLAST Like Alignment Tool) [10].

H =

	–	A	C	A	C	A	C	T	A
–	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	12

Fig. 2: Smith-Waterman Algorithm

FASTA is a DNA and protein sequence alignment software package first described (as FASTP) by David J. Lipman and William R. Pearson [8]. The original FASTP program was designed for protein sequence similarity searching. FASTA added the ability to do DNA vs DNA searches, translated protein vs DNA searches, and also provided a more sophisticated shuffling program for evaluating statistical significance. The FASTA program follows a largely heuristic method

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

which contributes to the high speed of its execution. It initially observes the pattern of word hits, word-to-word matches of a given length and marks potential matches before performing a more time-consuming optimized search using a Smith-Waterman type of algorithm. The FASTA programs find regions of local or global similarity between protein or DNA sequences, either by searching Protein or DNA databases, or by identifying local duplications within a sequence. BLAT is a pairwise sequence alignment algorithm that was developed by Jim Kent at the University of California Santa Cruz (UCSC) to assist in the assembly and annotation of the human genome [10]. It was designed primarily to decrease the time needed to align millions of mouse genomic reads and expressed sequence tags against the human genome sequence.

III. DESIGN

BLAST (Basic Local Alignment Search Tools) [9], is an algorithm for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences. A BLAST search enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold. Different types of BLASTs are available according to the query sequences. For example, BLASTN to compare DNA vs. DNA, BLASTP compares protein with another protein, BLASTX compares DNA with a protein, TBLASTN compares protein with another DNA.

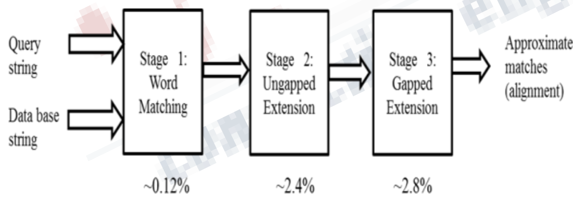

Fig. 3: The 3-stage pipeline structure of the BLAST algorithm.

Fig. 3 shows the basic architecture behind BLAST algorithm as reported by Ben Weintraub et. al. [11]. It consists of a three-stage pipeline. Each stage acts as a filter, selecting a small proportion of its input to be passed on to the next stage for further investigation. The approximate percentages of the inputs that are passed by each stage are shown in fig 3. Stage 1 consists of word matching, that is, finding short, fixed-length, exact matches (words) that are common to both the query sequence and the database sequence (referred to as w-mer). The length of these word matches varies between the different BLAST variants, and is also configurable, but in nucleotide BLAST, the default length of such words is 11 nucleotides. NCBI BLASTN [12] reduces storage and I/O bandwidth by storing the database using only 2 bits per letter (or base) for A, C, G, T. Using a longer words will result in fewer matches, and thus will decrease the running time of the search, but will also decrease the search's sensitivity, while using shorter words will increase the number of word matches,
and thus the running time while also increasing the sensitivity of the search.

In stage 2 of the algorithm, the ungapped extension, extends each seed in both directions allowing substitutions only and outputs the resulting the high scoring segment pairs (HSPs). AN HSP indicated two sequence segments with equal length, whose alignment score meets or exceeds an empirical set threshold (or cut-off score).

In stage 3, gapped extension, uses Smith – Waterman dynamic programming algorithm to extend HSPs allowing insertions and deletions. Among all these three stages, Word Matching stage is most time consuming. It takes almost 85% time of total searching operation. Different methods have been considered as a measure to reduce the time consumption at this particular stage. Yupeng Chen, et. al. [13] in their research paper on BLASTN, divided the word matching stage into three sub-stages, as shown in fig 4.

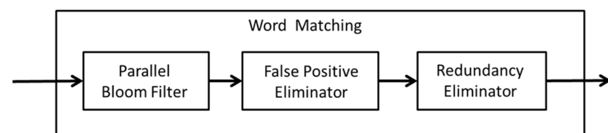

Fig 4 : Three sub-stages of FPGA-based accelerator for the word-matching stage of BLASTN.

In first sub-stage they have used Bloom filter. A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970

[14], that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not; i.e. a query returns either "inside set (may be wrong)" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter). The more elements that are added to the set, the larger the probability of false positives.

A Bloom filter is defined by a bit-vector of length m, denoted as BF[1, …, m]. A family of k hash functions hi : S → A, 1 ≤ i ≤ k, is associated to the Bloom filter, where S is the key space and A = {1, . . . ,m} is the address space. This filter is a simple space-efficient randomized hashing data structure suitable for quick membership tests on FPGA implementations.
A Bloom filter works in two steps.

1) Programming: For a given set I of n keys, I = {x1,…,xn}, I ⊆ S, the Bloom filter's programming process is described as follows. First of all, initialize the bit vector m with zeros, then, for each key xj ∈ I , compute its k hash values hi (x j), 1 ≤ i ≤ k, subsequently, set the bit vector to one according to the k hash values (i.e.,BF[hi(xj)]:= 1 for all 1 ≤ i ≤ k).

2) Querying: the querying process of the Bloom filter works the same as its programming process. For a given key x, compute k hash values hi (x), 1 ≤ i ≤ k. If any of the k bits BF[hi(x)], 1 ≤ i ≤ k, is zero, then, otherwise x is said to be a member of set I with a certain probability.

Though Bloom filter is space efficient, it is associated with an inherent false positive probability (FPP). This implies, a query returns either "inside set (may be wrong)" or "definitely not in set". Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives. The second sub-stage is false positive eliminator, which finds all false-positives matches generated by the Bloom filter and gets the corresponding position information in the query sequence for the true-positive w-mers. The third stage is the redundancy eliminator, avoids repeated generation of the same sequence alignment during the ungapped extension stage of BLAST.

In this paper, emphasis is given on optimally configuring Bloom filter. If we maintain a fixed false positive probability, the bit vector size needs to scale linearly with the inserted keyset. Table 1 [13] shows the Bloom filter size for false positive values of around 10-5 and around 10-3 and varying query length.

Table 1: Bloom Filter Memory size (in bits) for different Parameter Combinations

Query Size(n)	No. of Hash Functions(k)	Bit length of Bloom filter(m)	False positive Probability (FPP)
1k	16	23 084	1.53E-05
10k		230 832	
100k		2308 313	
1k	8	11 542	2.91E-03
10k		115 416	
100k		1154 157	

In this design the following parameters have been chosen,

- ♣ bit length of Bloom filter m = 23084 bits,
- ♣ query size n = 1k bits,
- ♣ kopt (no of Hash function) = 16

Conventional Bloom filter has been programmed by parsing the query sequence into overlapping sub-strings of length, w-mer_in, in the preprocessing step. To increase the processing speed conventional Bloom filter architecture changed to partitioned parallel Bloom filter (PPBF) architecture with each having 2 hash functions , 2 BRAMs and w-mer_in = 22bits, as shown in fig 5. In this design I have created eight groups of such PPBF to obtain total 16 Hash functions. Hence the total memory (m of size 23084 bits) is divided into 16 BRAMs of size 2 Kbit each.

Hash functions are an essential ingredient of the Bloom Filters. It is used in a variety of hardware applications and is critical for modern high performance computer architecture. The performance of a hashing scheme depends on two factors:

1) the collision/overflow handling method,
2) the hashing function chosen [15].

IFERP connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 3, Issue 11, November 2016

A hash function is an algorithm that maps data of variable length to data of a fixed length. The values returned by a hash function are called hash values, hash codes, hash sums,

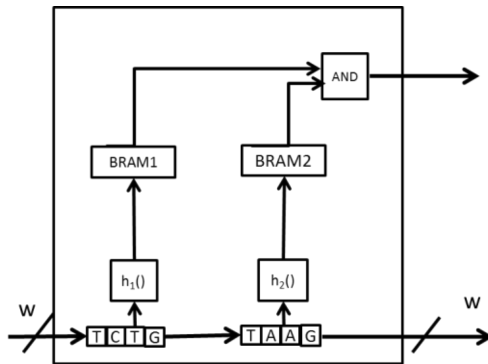

Fig.5: PPBF architecture with 2 Hash functions and 2 BRAMs.

checksums or simply hashes. Hash functions are used to accelerate table lookup or data comparison tasks such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on. Hash functions are primarily used in hash tables, to quickly locate a data record (e.g., a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

Hash function only hints at the record's location—it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries. Ramakrishna et.al [16] has discussed the performance of various Hash functions implemented in hardware's.

In this design, Hash function has been implemented using CRC (Cyclic Redundancy Check) codes which utilize Linear Feedback Shift Register (LFSR) to increase search speed [17]. Two codes that have wide use are CRC–16 and CRC–32. As the names imply, CRC–16 makes use of a 16-bit LFSR, while CRC–32 uses a 32-bit LFSR. The generator polynomials CRC-32 is used in this design and shown in Eq1,

$$G(x)=x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1 \quad \text{(Eq 1)}$$

In Eq1, the superscripts identify the tap location in the shift register. The order of the polynomial is identified by the highest order term, and specifies the number of flip-flops in the shift register. Since these polynomials are for modulo-2 arithmetic, each bit-shift is equivalent to a multiply by 2. Fig 6 shows the circuit diagram for CRC-32 code. It becomes quite difficult to implement the CRC calculation using a shift register. However, it is possible to convert a serial implementation into a parallel form that accumulates multiple bits in each clock cycle. The following paragraphs and tables describe how the CRC–32 polynomial is converted to calculate 16 bits at a time (i.e. on a half-word basis). The initial content and the final content for CRC–32 are presented in Table 2, 3, 4, 5 without the intermediate calculations.

Table 2: CRC-32 register contents (MSW) prior to any shift

R32	R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17
C32	C31	C30	C29	C28	C27	C26	C25	C24	C23	C22	C21	C20	C19	C18	C17

Table 3: CRC-32 register contents (LSW) prior to any shift

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1

Table 4: CRC-32 register contents (MSW) after twenty-two shifts

R32	R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17
X6	X5	X4	X3	X2	X1	X1	X1	X5	X4	X3	X2	X1	X3	X2	X1
X10	X6	X5	X4	X3	X2	X2	X6	X10	X9	X6	X5	X4	X4	X3	X2
X12	X9	X6	X5	X4	X3	X8	X7	X11	X10	X8	X6	X5	X6	X5	X4
X13	X10	X8	X7	X7	X9	X11	X12	X12	X11	X9	X7	X7	X8	X7	X6
X16	X11	X9	X8	X10	X12	X14	X14	X14	X13	X13	X9	X9	X9	X8	X7
X22	X13	X13	X12	X11	X15	X15	X15	X18	X17	X17	X10	X10	X12	X11	X10
	X15	X14	X13	X14	X16	X16	X17	X19	X18	X19	X13	X13	X15	X14	X13
	X16	X15	X14	X16	X17	X17	X19	X21	X20	X20	X18	X16	X16	X15	X14
	X21	X16	X15	X18	X18	X18	X20	X22	X21	X22	X19	X17	X17	X16	X15
	X22	X20	X19	X19	X19	X20	X22				X21	X18	X19	X18	X17
		X21	X20	X20	X21	X21					X22	X20	X20	X19	X18
		X22	X21	X22	X22							X21	X21	X20	X19
												X22			

Table5: CRC-32 register contents (LSW) after twenty-two shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
X1	X2	X1	X1	X1	X1	C32	C31	C30	C29	C28	C27	C26	C25	C24	C23
X3	X3	X2	X2	X2	X2	X1	X2	X1	X3	X2	X1	X1	X1	X2	X1
X4	X4	X3	X3	X3	X4	X3	X5	X4	X4	X3	X2	X2	X3	X8	X7
X5	X5	X4	X4	X5	X9	X6	X6	X5	X5	X4	X3	X4	X9	X12	X11
X6	X6	X5	X6	X10	X11	X8	X7	X6	X11	X6	X5	X10	X13	X14	X13
X7	X8	X7	X11	X12	X12	X11	X13	X12	X14	X12	X11	X14	X15	X15	X14
X9	X13	X12	X13	X13	X13	X13	X16	X15	X19	X16	X15	X16	X16	X18	X17
X14	X15	X14	X14	X14	X17	X22	X21	X20	X20	X18	X17	X17	X19		
X16	X16	X15	X15	X18			X22	X21		X19	X18	X20			
X17	X17	X16	X19							X22	X21				
X18	X21	X20													
X22															

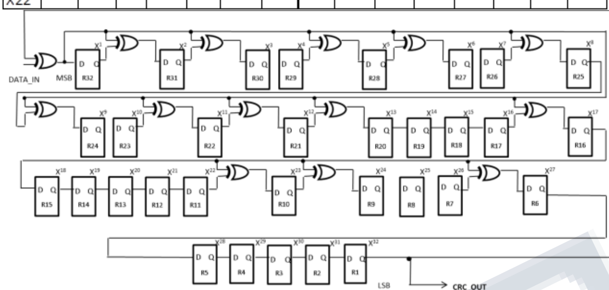

Fig 6: Linear Feedback Implementation of CRC–32

To obtain the results for parallel implementation of LFSR, a few important points are as mentioned below,

- Ri is the i th bit of the CRC register.
- Ci is the contents of i th bit of the initial CRC register, before any shifts have taken place.
- R1 is the least significant bit (LSB).
- The entries under each CRC register bit indicate the values to be XORed together to generate the content of that bit in the CRC register.
- Di is the data input, with LSB input first.
- D16 is the MSB of the input half-word, and D1 is the LSB.
- A substitution is made to reduce the table size, such that Xi = Di XOR Ci.
- The following properties were used to simplify the equations:
- Commutativity (A XOR B = B XOR A).
- Associativity (A XOR B XOR C = A XOR C XOR B).
- Involution (A XOR A = 0).

After formulation of the tables, we can generate the equations, which provide us the information about the address of BRAM to be made 1 if a w-mer of 22 bits enters to the crc32 circuit. As each BRAM is of size 2 kbit, so address of 11bits is sufficient to decode the address uniquely. LFSR can provide the information about the address after the entry of 22bits w-mer bit by bit, after 22 cycles, but by computing CRC-code, it is possible to generate the required address in the next cycle of clock only.

In the equations generated, Ci denotes the initial content of Ri register as curr_crc32[i]; the data input Di as w-mer_in[i]; next_crc[i] is content of register i after 22 shifts. The following equations are illustrating how the bit by bit expression is generated for the register content after 22 shifts.

next_crc32[1]=curr_crc32[6]^wmer_in[6]^curr_crc32[10]^ wmer_in[10] ^ curr_crc32[12] ^ wmer_in[12] ^ curr_crc32[13] ^wmer_in[13]^curr_crc32[16] ^ wmer_in[16] ^ curr_crc32[22] ^ wmer_in[22]; (Eq 2)

next_crc32[2] = curr_crc32[5] ^ wmer_in[5] ^ curr_crc32[6] ^ wmer_in[6] ^ curr_crc32[9] ^ wmer_in[9] ^ curr_crc32[10] ^wmer_in[10]^curr_crc32[11] ^ wmer_in[11] ^ curr_crc32[13] ^wmer_in[13]^curr_crc32[15]^ wmer_in[15] ^ curr_crc32[16] ^wmer_in[16]^curr_crc32[21] ^ wmer_in[21] ^ curr_crc32[22] ^ wmer_in[22]; (Eq 3)

This way we can get the content of register 32 as next_crc32[32] = curr_crc32[23] ^curr_crc32[1] ^ wmer_in[1] ^ curr_crc32[7] ^ wmer_in[7] ^ curr_crc32[11] ^ wmer_in[11] ^curr_crc32[13]^wmer_in[13] ^ curr_crc32[14] ^ wmer_in[14] ^ curr_crc32[17] ^ wmer_in[17] (Eq 4)

As already discussed the size of BRAM used is 2kbit, so the required address is of 11 bits. Out of the 32 registers only the contents of R1 to R10 are taken to generate the required address.

In current BLAST program, the required no of Hash functions is equal to the number of BRAMs used, which will point to the different memory locations of the BRAMs for the same data input. This can be easily done by using the same CRC-code, by changing the initial content of the register prior to any shift.

Module Hash_Pointer_Implementation – this module generates the address to which data 1 will be stored in BRAM corresponding to a particular w-mer_in input of size 22 bits.

IV. IMPLEMENTATION

To implement Bloom filter operation (programming and querying phase) following modules have been created: Module BRAM – To write and read data in BRAM (Block RAM) of memory size of 2K bits when write and read control signals are generated .It also resets the entire BRAM locations by storing 0 when reset signal is generated.

Module Controller – this is a finite state machine (FSM). It controls step by step the operations of module BRAM and module Hash_Pointer_Implementation depending upon the control signals which behaves as input ports or generated from inside logic of the modules.

The main objective of this state machine is to control the different functionality of module BRAM and module Hash_Pointer_Implementation in a meaningful manner.

The state machine is having following states:

- State INIT
- State RAM_ST_0
- State RAM_ST_1
- State START_HASH_POINTER
- State RAM_QUERY
- State RAM_RD_WAIT
- State RAM_MATCH

Transitions from one state to another state are triggered when change in state variables occur. The state diagram is shown in fig 7. Initial state is represented by state INIT. It will be in the state until reset signal (rst_n) is reasserted (as it is active low) and prog_in and query_in are low level. When rst_n is high, then if prog_in is equal to 1'b1 then it will move from INIT state to state RAM_ST_0. If query_in is equal to 1'b1 then it will move from INIT state to state RAM_QUERY.

In programming phase of Bloom filter prog_in signal should be asserted after resetting i.e in that case first entire memory locations should be initialized with a value 0 (as it is a bit array).This is done first by pointing to the first address of memory and then storing a value 0 through data_in signal. Then address counter is increased by 1, so that it points to the next address location, this keeps continuing till the address value becomes 11'h7ff- the last address of 2 Kbits memory size. Initialization of the BRAM, i.e. writing a value 0 in all memory locations is done with the help of state RAM_ST_0 and RAM_ST_1 by asserting and reasserting wr_n signal (as active low). After initialization of the entire BRAM, address becomes equal to 11'hff. When address becomes 11'hff transition to the START_HASH_POINTER state occurs. One output signal init_done is asserted by module controller, after this only the available w_mer_in can be considered as a valid input. Whether a particular w_mer_in is valid or not is decided by the status of w_mer_val signal (if it is high then w_mer_in is a valid one). A valid w_mer_in will trigger the functionality of module Hash_Pointer_Implementation and it will generate an address depending upon w_mer_in. The generation of a valid address is understood by assertion hash_addr_valid signal, this causes a write of 1'b1 in that particular memory location to indicate that that particular w_mer_in is already entered. This keeps on continuing till w_mer_val is 1'b0.

In querying phase, whether a particular w_mer_in already stored or not in BRAM is queried. From the INIT state transition to RAM_QUERY state occurs if rst_n is equal to 1'b1, prog_in is equal to 1'b0 and query_in is equal to 1'b1. Till the hash_addr_valid signal is not asserted after a valid w_mer-in input and corresponding address generation, it remains in same state only. Once hash_addr_valid is asserted, it changes rd_n to 1'b0 and wr_n to 1'b1 and reads the data from that particular address of BRAM and sends this data through data_out signal. This required a delay of clock cycle, done by the RAM_RD_WAIT state.

If the queried w_mer_in is already programmed then data_out should be equal to 1'b1 else 1'b0. In case of a matched w_mer_in an output of w_mer_ match is made 1'b1, else for mismatch w_mer_match is equal to 1'b0. This process is carried out in RAM_MATCH state.

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

If more w_mer_in needs to be queried, i.e. qurey_in is equal to1’b1, then from the RAM_MATCH state transition occurs to the RAM_QUERY state, otherwise to the INIT state.

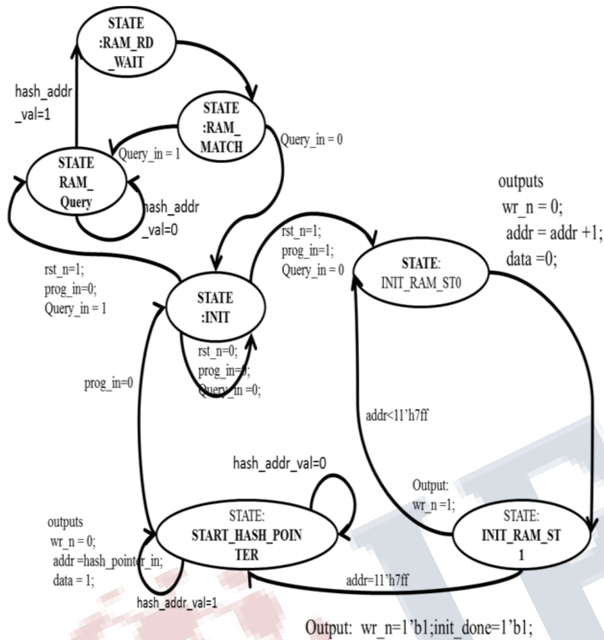

Fig 7: State diagrams for module controller

Module PPBF_macro – It is a top level module one step higher than previous modules, consists of two instances of module BRAMs. Each BRAM is again associated with separate instances of module Hash_Pointer_Implementation and module Controller. The inter-relation of inputs and outputs of different modules inside module PPBF_macro is shown in fig 8.

Module Bloom_filter – It is also a top level module of PPBF_macro. In this module, 8 instances of module PPBF_macro have been created. As already mentioned that each PPBF_macro consists of 2 instances of module BRAM, module Hash_Pointer_Implmentation and module Controller, so altogether 16 BRAM (BRAM0 – BRAM15) instances , 16 Hash functions (h0()- h15()) associated with each BRAMs have been created to point a particular address location in the respective BRAM for the same w_mer input.

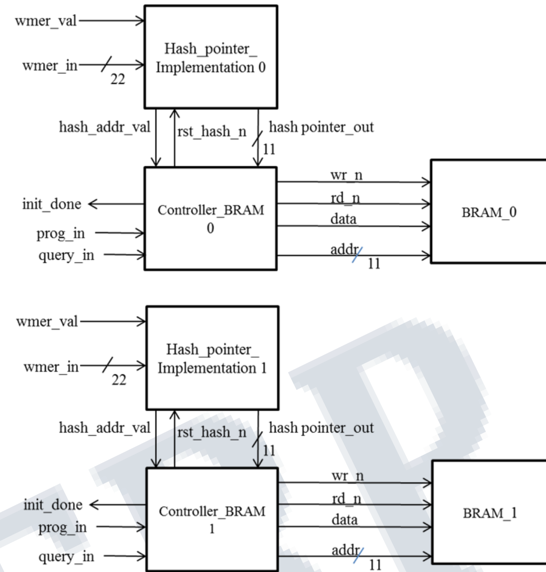

Fig 8: Port description of module PPBF macro

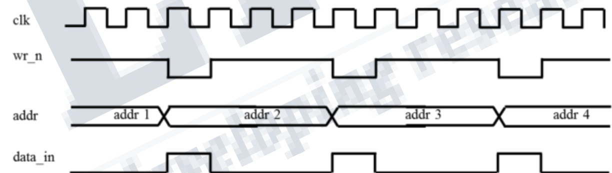

Fig 9: Timing diagram for memory write operation of BRAM

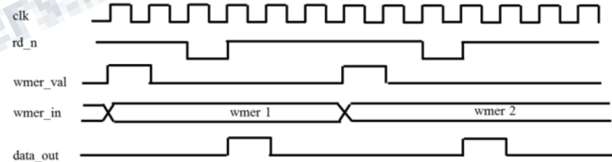

Fig 10: Timing diagram for memory read operation of BRAM

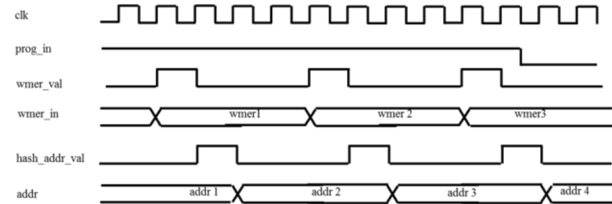

Fig 11: Timing diagram for programming phase of Bloom

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 3, Issue 11, November 2016

In fig 9 and 10 timing diagrams of write and read functionality of BRAM memory have been shown respectively. During a

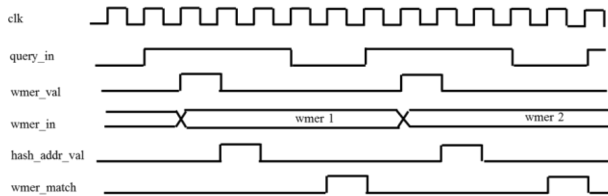

Fig 12: Timing diagram for querying phase of Bloom Filter

write operation, control signal wr_n is set to zero for one clock cycle and data_in is stored in a particular address of memory. During a read operation, a control signal rd_n is set to zero for one clock cycle and corresponding to the memory address generated by w_mer_in data is read and stored in data_out.

In fig 10 and 11timing diagrams of Programming and Querying phase of Bloom Filter are highlighted. Fig. 11 shows timing relationship of the important signals i.e after a valid w_mer_in it requires one cycle to generate a valid address. In the same manner fig. 12 shows in query phase after a valid w_mer_in, one cycle is required to generate a valid address and four cycles to indicate match or mismatch.

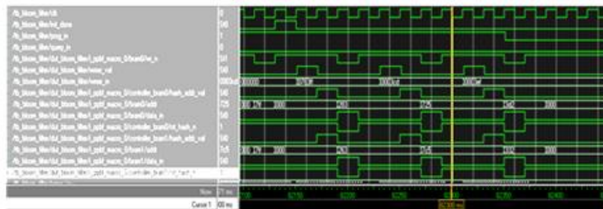

Fig. 13: Screen shot of signals showing valid wmer_in input and address generation of the BRAM

Fig.13 and 14 are highlighting simulations of the Verilog codes written to generate Bloom filter functionality for programming and querying phase respectively. Fig.13 shows after initialization of entire BRAM, when a valid w_mer_in comes as input, it generates an address corresponding to that input and a 1'b1 stores in that particular address. Fig.14 shows in querying phase that after a valid w_mer_in it requires 4 cycles to know whether it is a match or mismatch.

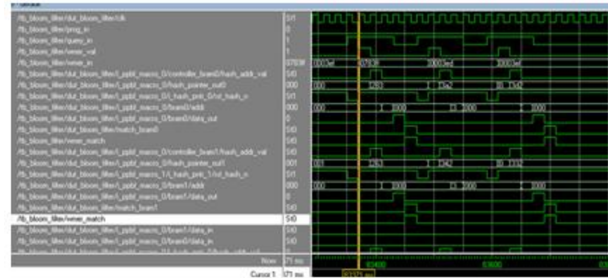

Fig. 14: Screen shot for showing the querying phase of Bloom Filter

V. CONCLUSION

In this paper, implementation of Bloom filter of word matching stage of BLAST is done. To do that, Hash function associated with Bloom Filter is implemented by using CRC-32 which is nothing but parallel implementation of LFSR. Simulation results of the inputs and outputs are observed thoroughly so that a better understanding of all the modules and their interdependency can be analyzed. Output signal w_mer_match bit is set to one when an already programmed w_mer_in is given as input in querying phase. A mismatch is indicated when w_mer_match bit is set to zero. Scope of improvement in reduction of searching time exists by transforming this sequential search of w_mer_in in eight PPBF_macros in parallel.

Acknowledgment

I take great pleasure in expressing my sincere thanks to Dr. Sanjay Chitnis, Principal, CMRIT and Dr. H.N. Shankar, Professor & Dean -- Academics & Research, CMRIT for their valuable support and encouragement.

REFERENCES

[1] Saul B. Needleman and Christian D. Wunsch, , “A general method applicable to the search for similarities in the amino acid sequence of two proteins”, Journal of Molecular Biology, 48 (3): 443–53, 1970.

[2] P.H. Sellers, “On the theory and computation of evolutionary distances” SIAM JInal on Applied Mathematics 26 (4): 787–793, 1974.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 3, Issue 11, November 2016

[3] D. Sankoff, “Matching sequences under deletion/insertion constraints". Proceedings of the National Academy of Sciences of the USA 69 (1): 4–6, 1972.

[4] Temple F. Smith, and Michael S. Waterman, (1981). "Identification of Common Molecular Subsequences". Journal of Molecular Biology 147: 195–197, 1981

[5] O. Gotoh, and Y. Tagashira, “Sequence search on a supercomputer” Nucl. Acid Res.,. 14, 57-64, 1986.

[6] A. F. W. Coulson, J. F. Collins, and A. Lyall, “Protein and nucleic acid sequence database searching: a suitable case for parallel processing” Comput. J . 30,420 – 424, 1987.

[7] W. B. Goad, and M.L Kanehisa, “Pattern recognition in nucleic acid sequences. I. A general method for finding local homologies and symmetries” Nucl. Acid Res.,. 10, 245-263, 1982

[8] DJ Lipman and WR Pearson, "Rapid and sensitive protein similarity searches". Science 227 (4693): 1435–41, 1985.

[9] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, & D.J. Lipman, "Basic local alignment search tool." J. Mol. Biol. 215:403-410, 1990.

[10] James W Kent, "BLAT--the BLAST-like alignment tool". Genome Research 12 (4): 656–664, 2002.