academicJournals

Vol. 13(33), pp. 3344-3351, 13 August, 2014
DOI: 10.5897/AJB2014.13680
Article Number: 4C4439846615
ISSN 1684-5315
Copyright © 2014
Author(s) retain the copyright of this article
http://www.academicjournals.org/AJB

African Journal of Biotechnology

Full Length Research Paper

A parallel reconfigurable platform for efficient sequence alignment

A. Surendar[1]*, M. Arun[2] and P. S. Periasamy[3]

[1]Research Scholar, Anna University, Chennai-600025, India.
[2]School of Electronics Engineering, VIT University, Vellore-632014, India.
[3]Department of ECE, K.S.R. College of Engineering, Tiruchengode-637215, India.

Received 31 January, 2014; Accepted 4 July, 2014

Bioinformatics is one of the emerging trends in today's world. The major part of bioinformatics is dealing with DNA. Analysis of DNA requires more memory and high efficient computations to produce accurate outputs. Researchers use various bioinformatics algorithms for sequencing and pattern detection techniques, but still now it takes enormous amount of time for computations. In our method we are going to propose a time, memory and speed optimized algorithms for efficient repetitive finding in genomes and proteins. Then, another major aspect is the hardware implementation. It is a platform which reduces the complexity of process further. Therefore, we have proposed to implement the optimized algorithm in the reconfigurable and user friendly FPGA platform. Thus, our proposal mainly focuses on an efficient and optimized computation, analysis and sequencing of DNA pattern. The distinct feature is reducing the time consumption from several hours to few seconds.

Key words: DNA, sequencing, bioinformatics, efficient computations, repetitive finding, optimized sequencing.

INTRODUCTION

In the world of expanding set of biological species, finding repetitive structures in genomes and proteins is important to understand their biological functions. DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases Adenine, Guanine, Cytosine and Thymine in a strand of DNA (Surendar et al., 2013). The advent of rapid DNA sequencing methods has greatly accelerated biological and medical research and discovery. If the number of maximal repeat increases, then finding those structures becomes tedious. In existing method, Burrows Wheeler Transform and Wavelet Coding, the major disadvantage is time consumption. And it also needs huge computer space for processing the structures. One of the most important thing that decides our heredity is DNA. One of the well-known features of DNA is its repetitive structures. Many existing methods proposed different data compression formats to reduce the space consumption. Even though the method saves memory, time and speed efficiency cannot be obtained. To obtain optimization of the bioinformatics algorithms used: i) bloom filter; ii) content-addressable memory; iii) Aho-Corasick algorithm are used.

*Corresponding author. E-mail: surendararavindhan@gmail.com.

Author(s) agree that this article remain permanently open access under the terms of the Creative Commons Attribution License 4.0 International License

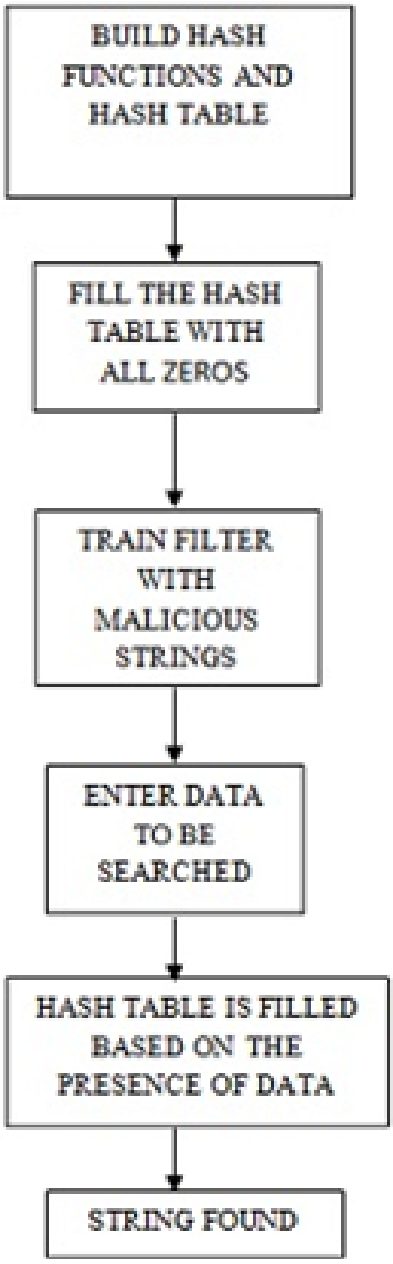

Figure 1. Flow of bloom filter.

TOOLS FOR OPTIMIZATION

A field-programmable gate array (FPGA)

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer. The FPGA configuration is generally specified using a hardware description language (HDL). FPGAs are reprogrammable silicon chips. It provides hardware-timed speed and reliability. FPGAs are truly parallel in nature. Previously, a design may have included 6 to 10 ASICs, the same design can now be achieved using only one FPGA (Surendar et al., 2013).

Altium - 3000 nanoboard-XILINX variant

Altium - 3000 nanoboard-XILINX variant is a perfect entry-point to discover and explore the world of soft design. It is a programmable hardware platform. Rapid and interactive implementation and debugging of digital designs can be achieved. It has a fixed user FPGA's on the motherboard and so, speed of processing is increased. Circuit can be probed, analyzed and debugged interactively using an array of virtual instruments and JTAG-based monitoring features.

Bloom filter

Bloom filter (Arun and Krishnan, 2011) is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set (Figure 1). This compact representation is the payoff for allowing a small rate of false positives in membership queries; that is, queries might incorrectly recognize an element as member of the set which can be made negligible by the intensive design effort. It consists of number of hash tables and hash functions that easily store and handle the incoming strings. Each hash table entry stores only a single bit of data, thus a hash table of size M would be made up of M entries, each of size one bit. A bloom filter must be trained with a dictionary of malicious strings before it can be used in a system. All hash table entries are initialized to 0 before training begins. During the training phase malicious strings are fed one at a time to the bloom filter. Each of the k hash functions then acts on every incoming dictionary string (Table 1) and computes an output in the range 0 to M − 1. Entries corresponding to these k outputs are set to 1 in the hash table. The bloom filter reports the current string in its window as a member of the dictionary on which the bloom filter was trained. If a non dictionary input string (Table 2) is such that it hashes to k hash table entries, each of which was set to 1 by one or more dictionary elements during the training phase, the bloom filter will erroneously report this string to be a member of the dictionary on which the bloom filter was trained. The bloom filter thus reports a false positive in this case. Though a bloom filter may occasionally report false positives, it does not allow for false negatives. Even though a bloom filter may sometimes report a non member to be a part of the dictionary set, it will never happen that a true member goes unreported. Thus, the working of bloom filter can be explained in following steps: 1) training the bloom filter with various strings; 2) defining the hash functions and building hash testing the incoming strings for the finding of given string with the help of hash functions; 4) determination of result using the behavior of the filter.

The part of code to hash the values to the hash table to

Table 1. Dictionary.

Path	In dictionary	Suffix link	Dict. suffix link
()	-		
(a)	+	()	
(ab)	+	(b)	
(b)	-	()	
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	-	(a)	(a)
(caa)	+	(a)	(a)

Dictionary {a, ab, bc, bca, c, caa}

Table 2. Input string analysis.

Node	Remaining string	Output end position	Transition	Output
()	abccab		Start at root	
(a)	bccab	A:1	() to child (a)	Current node
(ab)	ccab	Ab:2	(a) to child (ab)	Current node
(bc)	cab	bc:3, c:3	(ab) to suffix (b) to child (bc)	Current node, Dict suffix node
(c)	ab	c:4	(bc) to suffix (c) to suffix () to child (c)	Current node
(ca)	b	a:5	(c) to child (ca)	Dict suffix node
(ab)		ab:6	(ca) to suffix (a) to child (ab)	Current node

Analysis of Input string analysis.

the filter is as:

hash1:=hash(present_state(1),d); hash2:=hash(present_state(2),d); a<=hash1; b<=hash2; y(hash1)<='1'; y(hash2)<='1'; if(hashd_bitvectr(hash1)='1'and hashd_bitvectr(hash2)='1')then state<="positive"; shft_out :='0'; match<=input(pos-1 to pos+4); -- elsif(hashd_bitvectr(hash1)='1' or hashd_bitvectr(hash2)='1')then --- state<="fals_pos"; -- shft_out :='1'; else state<="negative"; shft_out :='1'; end if The VHDL simulation of the filter yields the waveform shown in Figure 2.

Content addressable memory

Manuscript of content-addressable memory (CAM) was received on July 17, 1987 and revised October 5, 1987. A content-addressable memory (CAM) is a high speed matching unit because it has parallel matching capability (Yoshiki et al., 2002). It speeds up the data searching and pattern matching. CAMs are storage devices that allow its contents to be accessible on the basis of a match between a specified key and the contents, a process called "content addressing". CAM architectures fall between two extremes: the bit serial CAM and the fully parallel CAM. In the bit serial CAM, the matching logic is associated with one bit position, and shared among all the bits in a word, in effect matching one bit at-a-time simultaneously in all the CAM words. In the fully parallel CAM, each word has its own bit-parallel matching logic, allowing that match of all words to process. Here, initially the device is trained with certain 8 bit binary database. And then the input binary parameter is given. The number of 1's in the parameter is extracted by parameter extraction then it is stored in the parameter memory. Then 1's in the input parameter is compare with the trained database and produce a required result else next input parameter is given. Thus, the working of CAM filter can be expressed in following steps. Castelo et al., 2002

1. The device is trained with database,
2. Input is given,

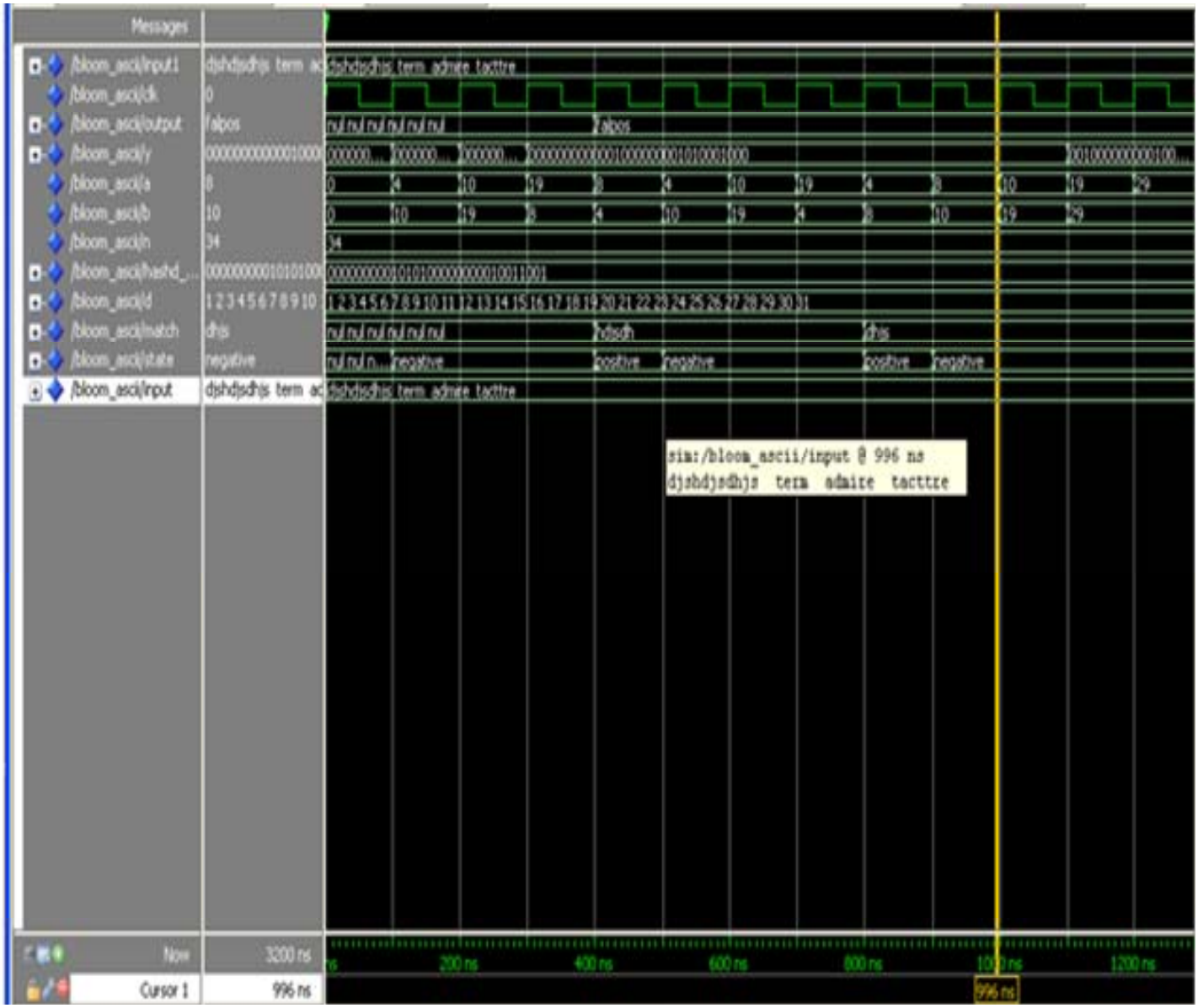

Figure 2. Output waveform of bloom filter.

3. The number of 1's and 0's is extracted by parameter extraction,
4. It is stored in memory,
5. Compare 1's in the given input with database,
6. Finally, it produced the required input if it is matched (Figure 3).

The code to train CAM is given as process (clk,fail,sram_out,ss)

begin; if rising_edge(clk) then; tcam_in.input<=input; tcam_in.current_state<=sram_out;
end if; end process; x2:tcam port map(tcam_in,tcam_out); x3:sram port map(tcam_out,sram_out,fail); process(sram_out); begin; if sram_out=2 then; output<=" pattern he matched "; elsif sram_out=9 then; output<="pattern hers matched"; elsif sram_out=7 then; output<="pattern his matched "; elsif sram_out=5 then; output<="pattern she matched "; end if; end process; The VHDL simulation of the filter yields the waveforms shown in (Figures 4 and 5).

Aho-Corasick

Aho-Corasick algorithm (Komodia, 2012) is a dictionary matching algorithm that searches for elements of a finite set of strings in the input text, developed by Alfred V. Aho

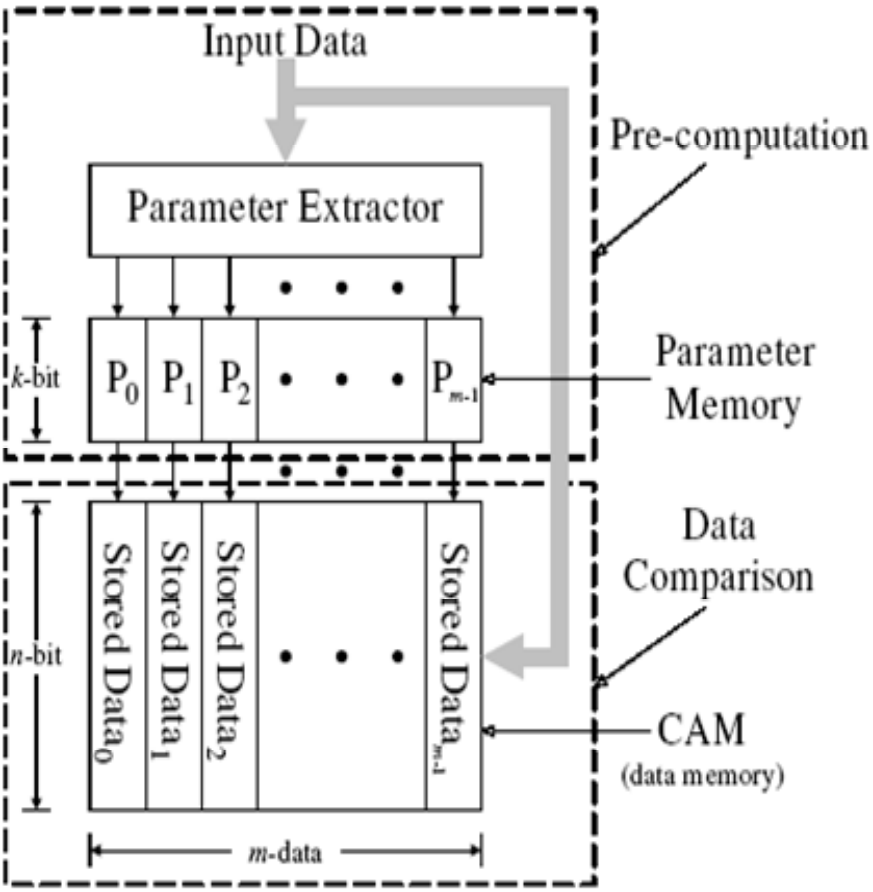

Figure 3. Pre-computation block of CAM.

and Margaret J. Corasick. Since, it locates all patterns in one time, the time complexity of the algorithm (Jung et al., 2006) is proportional to sum of the length of the patterns, length of the input text and the number of matches. In this algorithm, a trie with suffix tree-like set of links is established from each node representing a string to the node corresponding to the longest proper suffix. Since, it also consists of links from each node to the longest suffix node that connect to a match string; all of the matches can be traversed by going along the resulting linked list. The trie is utilized at runtime to keep track of the longest match and the suffix links are used to make sure the computation is proportional to the length of the input. For every link along the dictionary suffix linked list and every node in the dictionary located, a match is found. Since most of the time, the pattern database is known ahead, program can be created to build the trie, compile it and save it for later use. In this case, the computational complexity in the runtime is proportional to the sum of the length of the inputs and the number of matched entries. Figure 6 shows an example of data structure made up from a couple of strings. Each row represents a node in the trie while each column indicates the distinct order of characters from root to the node. In every step, the current node will try to find its child recursively if the suffix child does not exist until it reach the root node. Steps taken when scanning “abccab” are shown below.

Simulation on an input text

Since there may be two or more dictionary entries at a character location in the input text, more than one dictionary suffix link may need to be followed. The working of Aho-Corasick can be explained as follows.

1. The data pattern to be analyzed is built as a dictionary,
2. The pattern to be find is given as input,
3. The node built based on suffix matching,
4. The input for next string is taken from previous node.

Hence, all the pattern is matched at same time. The critical part of code to train the filter is given as, architecture behave of aho_Corasick is

type state_node is;
(state_0,state_1,state_2,state_3,state_4,state_5,state_6,

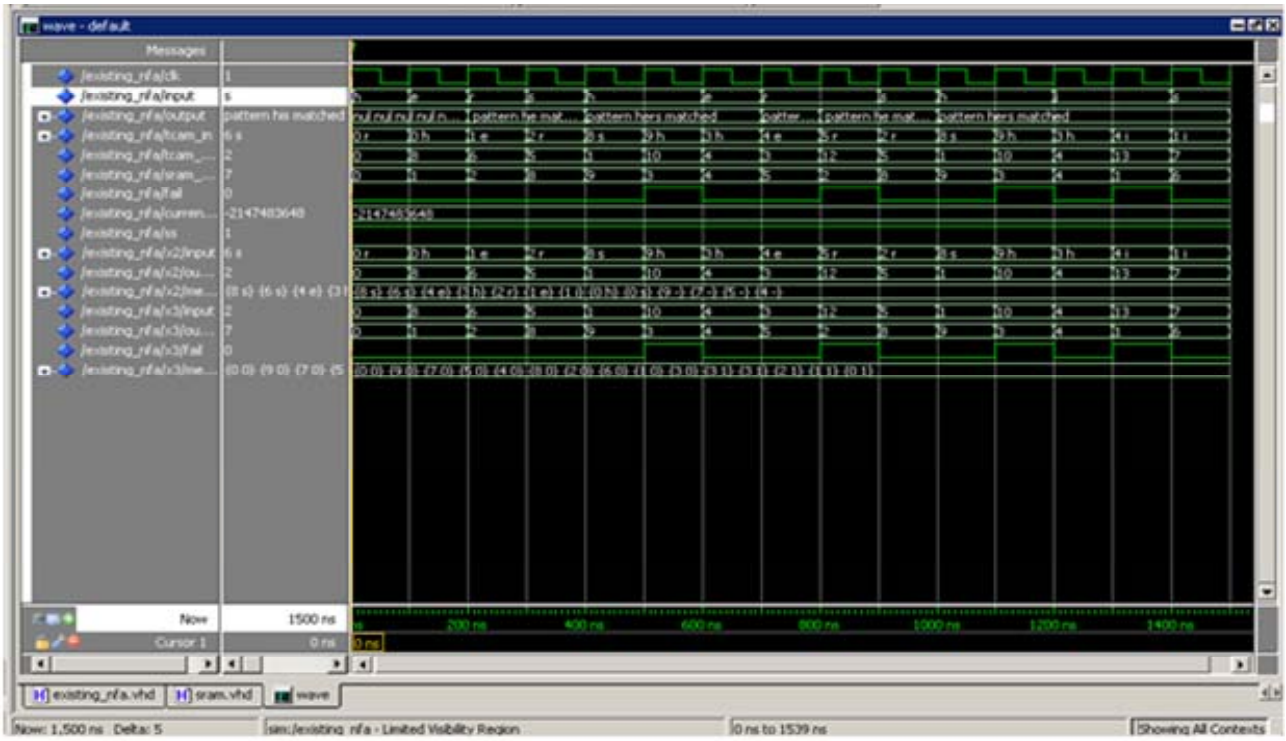

Figure 4. Output waveform of CAM.

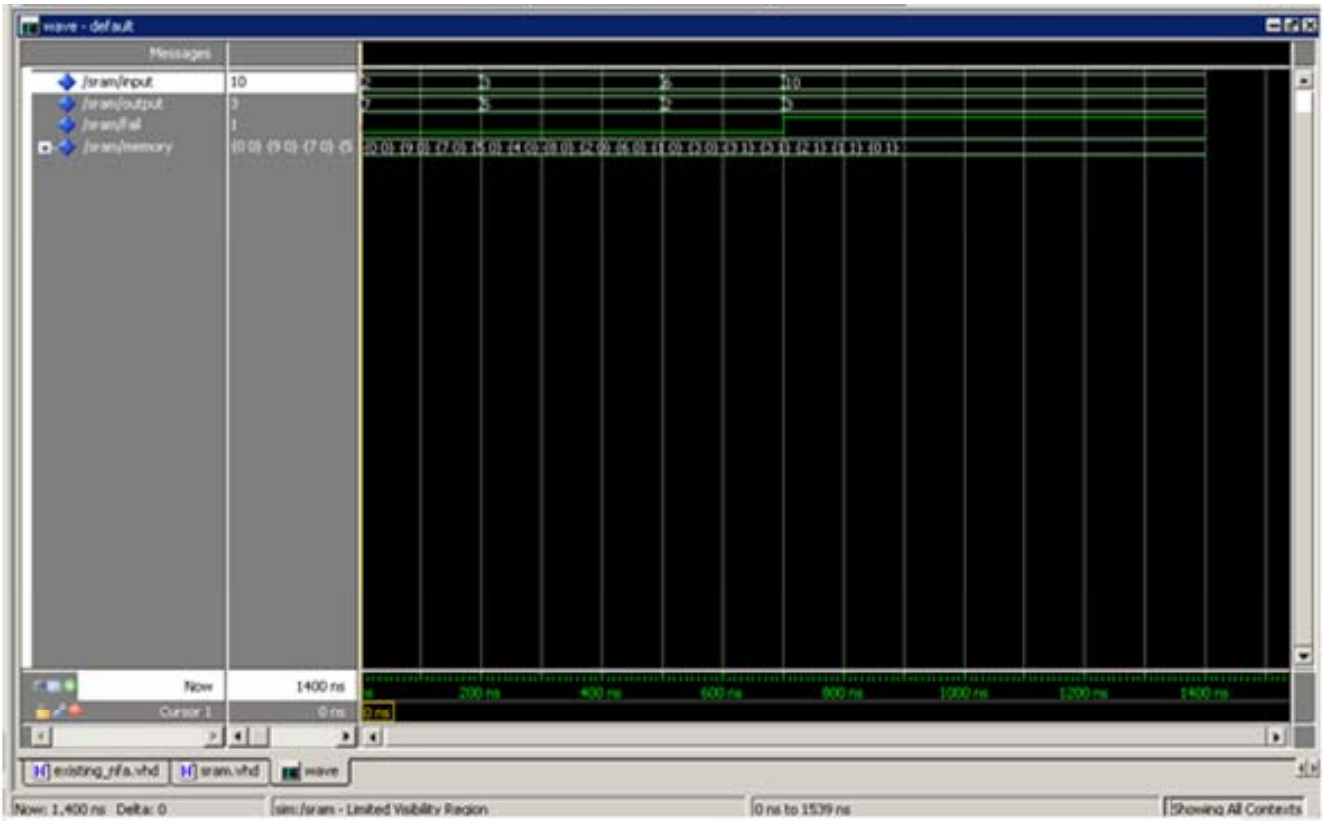

Figure 5. Output waveform of SRAM.

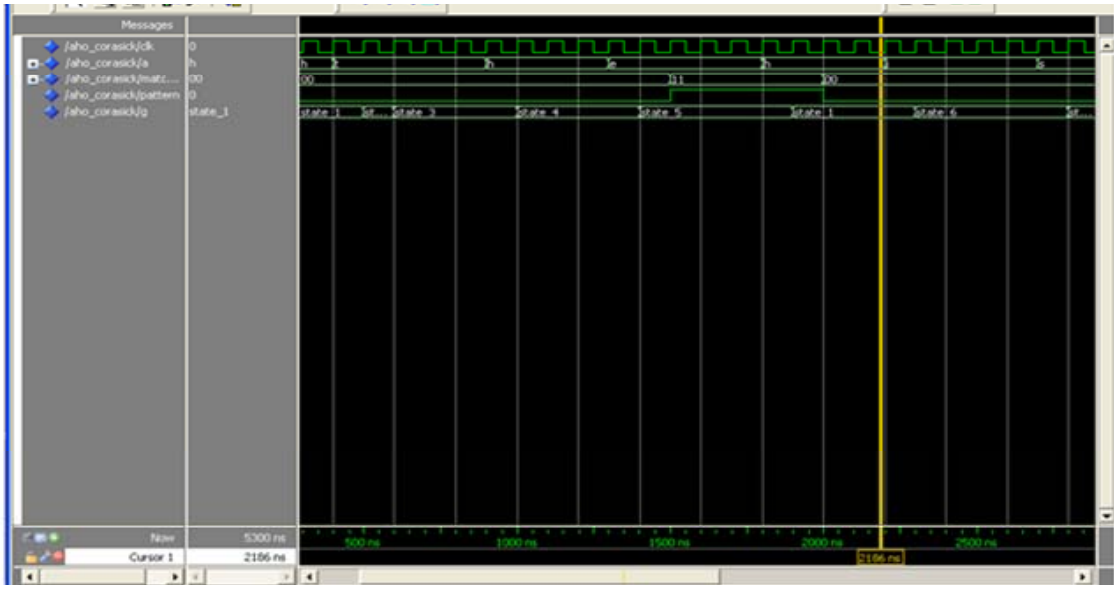

Figure 6. Output waveform of Aho-Corasick.

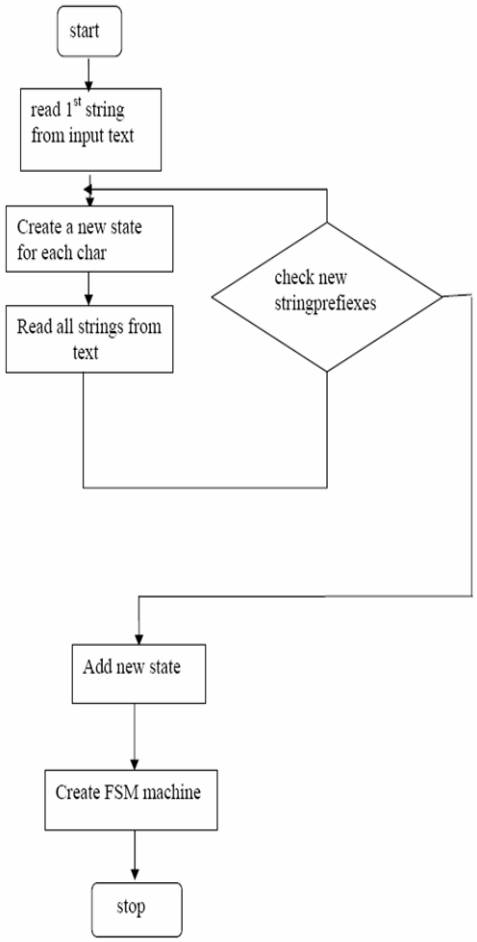

Figure 7. Flow of Aho-Corasick string matching.

PERFORMANCE RESULTS

PARAMETERS	AHO	BLOOM	CAM	TCAM	SRAM
IOs	12	321	169	65	72
CELL USAGE	111	9004	221	41	325
FF/LATCHES	13	104	20	-	-
IO BUFFERS	12	321	169	37	72
TOTAL REAL TIME	3s	10s	5s	3s	4s
TOTAL CPU TIME	3.23s	10.2s	5.4s	2.93s	4.28s
TOTAL MEMORY USED	234872Kb	26239Kb	24408Kb	232120Kb	233464Kb

Figure 8. Performance summary of filters.

state_7,state_8,state_9); signal g:state_node; begin; process(clk,g,a); begin; if rising_edge(clk) then; case g is; when state_0 =>; match_vector<="00"; pattern<='0'; if a(1)='h' then; g<=state_1; elsif a(1)='t' then; g<=state_3; else; -- g<=state_0; end if; The VHDL simulation of the filter yields the below waveform (Figure 7).

Conclusion

The filter used not only reduces the execution performance time but it stands out most in saving the memory. The flip flops and the latches used are triggered efficiently using perfect clocks. The total I/O ports used for this process are very less. The time of CPU processing is reduced very much and thus it enhances the output processing capability. The below table shows the requirements of optimized filters. The optimized algorithms are implemented in reconfigurable FPGA platform. The FPGA platform is in Altium Nanoboard 3000- Xilinx Spartan. The above proposed algorithms are analyzed to be efficient from their performance. When it is implemented in a reconfigurable platform it will work in more optimized way that produces accurate outputs. The Nano Board 3000 is a programmable design environment so it will be efficient for analysis.

Conflict of Interests

The author(s) have not declared any conflict of interests.

REFERENCES

Arun M, Krishnan A (2011). Functional Verification of Signature Detection Architectures for High Speed Network Applications. International Journal of Automation and Computing, Springer 9(4):395-402.

Arun M, Krishnan A (2011). Low Power Bloom Filter Architectures Using Multi Stage Lookup Techniques. Aust. J. Elect. Electronics Engineering, 8(3):1-10.

Castelo AT, Martins W, Gao GR (2002). Troll-Tandem Repeat Occurrence Locator. Bioinformatics. 18(4): 634-636.

Jung HJ, Baker ZK, Prasanna VK (2006). Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems.

Komodia (2012). Aho-Corasick source code. Available: http://www.komodia.com/aho-corasick

Surendar A, Arun M, Periasamy PS (2013). Hardware Based Algorithms for Bioinformatics Applications - A Survey. Int. J. Appl. Eng. Res. (6):745-754.

Surendar A, Arun M, Bagavathi C (2013). Evolution of Reconfigurable Based Algorithms for Bioinformatics Applications: An Investigation. Int. J. Life Sci. Bt & Pharm. Res. 2(4):17-27. Symposium on Biocomputing. 7: 271-282.

Yoshiki Y, Tsutomu M (2002). High Speed Homology Search with FPGAs. Pacific