HARDWARE ACCELERATION OF SOFTWARE

LIBRARY STRING FUNCTIONS

Approved By:

____
Dr. Mitchell A. Thornton

____
Dr. Frank Coyle

____
Dr. Peggy Ping Gui

HARDWARE ACCELERATION OF SOFTWARE LIBRARY STRING FUNCTIONS

A Thesis Presented to the Graduate Faculty of

School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Computer Engineering

by

Pallavi Anil Kulkarni

(B.E., Cummins College of Engineering)

December 15, 2007

Copyright (2007)

Pallavi Anil Kulkarni

All Rights Reserved

ACKNOWLEDGEMENT

Firstly, I would like to express the deepest appreciation to my advisor, Professor Dr. Mitch Thornton, for giving me the opportunity to work in a very interesting area of CAD, and for his support throughout my graduate studies at Southern Methodist University. This thesis would not have been possible without his continuous encouragement and guidance.

Secondly, I would like to thank Dr. Coyle, for sharing with me his knowledge regarding the time measurement, which was the critical factor in my research. His insightful suggestions helped me perform the time measurements to the maximum accuracy. I would also like to thank Dr. Ping Gui for her invaluable help in suggesting a robust and exact method to measure the time in my research.

Thirdly, I would really like to appreciate Prof. Jingbo Ye's help in allowing me to work in the department of Physic's laboratory, to perform the time measurements on the hardware. I am also grateful to Mr. Tiankuan Liu for offering me his help with any technical difficulties during my lab visits.

My special thanks to Nikhil, my fiancé and Prajna, a good friend of mine, for their suggestions, which helped me tackle number of difficulties during my experimentation. I admire their ever lasting support and continuous encouragement throughout my research.

Kulkarni, Pallavi B.E., Cummins College of Engineering, 2004

<u>Hardware Acceleration of Software Library String Functions</u>

Advisor: Professor Dr. Mitch Thornton

Master of Science conferred December, 15, 2007

Thesis completed November, 1, 2007

Character string matching, which involves finding all the occurrences of a pattern of length '*m*' in a string of length '*n*', is a very well known problem in the field of information systems. The strings may represent names of persons, postal addresses, number plates of vehicles, DNA sequences, or music notes, depending upon applications, and accordingly, the string matching can either be exact or approximate. Numerous algorithms like KMP [30], Boyer Moore [29], Wagner and Fischer [5], Ukkonen's [3], and CASM [11] have been developed so far to solve this problem efficiently. As the string lengths increase, the overall algorithm complexity in time and space also increases. There has been a huge amount of efforts put forth in the research community to increase the efficiency of such algorithms.

This thesis presents a very simple, efficient and fast implementation method for performing different string operations. The proposed method of implementation is to map the string matching operations into digital hardware. In this thesis, we used a Field Programmable Gate Array (FPGA) to perform the string operations at the lowermost level by representing strings in terms of bits. Several string functions implemented using

the FPGA with this approach were found to produce the desired results within few nanoseconds. The efficiency of the aforesaid approach is evident from the experimental results obtained for a variety of string operations. Although we chose to implement the character string algorithms in FPGA technology, these results are equally applicable to dedicated Application Specific Integrated Circuits (ASIC) or other digital circuit implementation technology.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

ACKNOWLEDGEMENTS iv

CHAPTER

1. INTRODUCTION 1

    1.1 Motivation 1

    1.2 Basic Terminologies Used In String Matching 1

    1.3 Previous Work 4

        1.3.1 Exact String Matching Algorithms 5

        1.3.2 Approximate String Matching Algorithms 8

    1.4 Organization Of Thesis 20

2. SOFTWARE AND HARDWARE FUNDAMENTALS 21

    2.1 Basics Of Communication 21

    2.2 Fundamentals Of **`Verilog`** Programming 26

    2.3 Programmable Logic Devices 29

    2.4 Software String Library Functions 34

3. IMPLEMENTATION DETAILS 39

    3.1 DE2 Development And Education Board 39

    3.2 QuartusII 40

    3.3 Experimental Setup 42

    3.4 Serial Communication Through C Program 43

    3.5 Serial Communication Through **`Verilog`** Program 45

3.5.1 Transmitter Block In **Verilog** 45

3.5.2 Receiver Block In **Verilog** 46

3.6 **Verilog** Implementation Of String Functions 47

3.7 Basic Setup 61

3.8 Time Measurement 63

3.8.1 Time Measurement In Software 63

3.8.2 Time Measurement In Hardware Using Oscilloscope 64

4. EXPERIMENTAL RESULTS AND ANALYSIS 66

4.1 Experimental Results 66

4.2 Analysis 72

5. CONCLUSION AND FUTURE WORK 74

5.1 Conclusion 74

5.2 Future Work 75

APPENDIX

A C CODE FOR SERIAL COMMUNICATION 76

B C CODE FOR TIME MEASUREMENT 81

C VERILOG CODE FOR SERIAL COMMUNICATION 85

D VERILOG CODES FOR STRING OPERATIONS 113

E IMAGES OF MEASUREMENTS AND RESULTS 173

REFERENCES 176

LIST OF TABLES

Table

1.1: Editing Path 3

1.2: Performance Summary Of Algorithms 8

1.3: Edit Distance Table For Comparing ‘ONE ‘And ‘ON’ 10

1.4: Edit Distance Table For ‘TGGC’ And ‘ACTG’ 15

3.1: Truth Table Of X-OR Function 48

LIST OF FIGURES

Figure

1.1: Approximate String Matching 9

1.2: Trie Data Structure 13

1.3: Architecture Of Linear Systolic Array 16

1.4: Internal Architecture Of Processing Cell 18

2.1: Rs-232 Logic Waveform (8n1) 22

2.2: Serial Transmission Of Character ‘A’ 23

2.3: Structure Of A Module In **Verilog** 28

2.4: Internal Structure Of FPGA 31

2.5: Design Cycle With FPGA Devices 33

3.1: Altera DE2 Board 39

3.2: Design Flow On QuartusII Tool 41

3.3: Experimental Setup 43

3.4: Flowchart Of Serial Communication Through C 44

3.5: Transmitter Block In **Verilog** 46

3.6: Receiver Block In **Verilog** 47

3.7: ASM Chart Of ‘**strcmp**’ Function In **Verilog** 49

3.8: ASM Chart Of ‘**strcasecmp**’ Function In **Verilog** 50

3.9: ASM Chart Of ‘**strstr’** Function In **Verilog** 52

3.10: ASM Chart Of ‘**strchr’** Function In **Verilog** 53

3.11: ASM Chart Of ‘**strchr_pos’** Function In **Verilog** 55

3.12: ASM Chart Of ‘**strrchr’** Function In **Verilog** 57

3.13: ASM Chart Of ‘**strupr’** Function In **Verilog** 58

3.14: ASM Chart Of ‘**strlwr’** Function In **Verilog** 60

3.15: ASM Chart Of ‘**strlen’** Function In **Verilog** 61

3.16: Overall Implementation Methodology 63

3.17: Time Measurements On Oscilloscope 65

This thesis is dedicated to

my dearest family

and my friends

Chapter 1

INTRODUCTION

1.1 Motivation

String matching is a very well known research problem in the information systems area. This problem finds applications in the areas such as DNA sequence matching, directory searches for particular name, and information retrieval based on person's SSN or Name etc. Many algorithms have been suggested so far for computing string comparisons. These algorithms are designed to match the given strings either exactly or approximately depending on application. The complexity involved in string matching increases as the string size increases, reducing the speed of calculation achieved through various proposed algorithms. Hence, different approaches have been developed previously by researchers to make existing algorithms work efficiently in all possible scenarios.

1.2 Basic Terminologies Used In String Matching

Following is the discussion of basic terminologies used in string matching.

Edit distance [11, 4]:

Edit distance is measure of similarity between two strings. If strings are made of '*n*' characters, then Edit distance between two strings **`string1`** and **`string2`** is defined as number of operations which should be performed on these characters to convert **`string1`** to **`string2`**. The operations are called as Edit Operations which can be insertion, deletion, and substitution or others [3, 11]. These operations have specific weights associated with them.

The overall edit distance between two strings is the minimum sum of all the edit operations required to obtain **`string2`** from **`string1`**. The characters in strings can be English letters, lines of source code, music notes, or DNA base pairs etc.

- Insertion operation: This operation inserts a particular character in a string. For example string1 '**`ABC`**' can be converted to string2 '**`ABCD`**' by one insertion operation consisting of the insertion of '**`D`**'.
- Deletion operation: This operation deletes one or more characters from a string. For example string1 '**`ABCD`**' can be converted to string2 '**`ABC`**' by deleting the last character '**`D`**'.
- Substitution operation: This operation substitutes one or more characters from a string with one or more characters from another string. For example string1 '**`ABCD`**' can be converted to string2 '**`BBDD`**' by two substitution operations: substituting '**`A`**' with '**`B`**' and substituting '**`C`**' with '**`D`**'.
- Transposition operation: This operation copies one or more characters from a string to another string. For example string1 '**`ABCD`**' can be converted to

string2 ‘**`ABDD`**’ by two transposition operations: copying ‘**`A`**’ and ‘**`B`**’ from first string to second string and one substitution operation: substituting ‘**`C`**’ with ‘**`D`**’.

Editing path [8]:

Editing path is a path that graphically represents the sequence of edit operations to convert **`string1`** to **`string2`**. Table 1.1 shows editing path between two strings ‘**`ABC`**’ and ‘**`ABB`**’.

Table 1.1: Editing Path

		A	B	C
	0	1	2	3
A	1			
B	2			
B	3			

Normalized edit distance [8]:

The normalized edit distance between two strings, **`string1`** and **`string2`** is defined as the minimum quotient between the sum of weights of the edit operations required to transform **`string1`** having length ‘m’ into **`string2`** having length ‘n’, and the length of the editing path corresponding to these operations. This can be symbolically represented as follows:

$NED = \min\{W(P)/L(P)\}$

Where, P = Editing path between A and B

W (P) = sum of the weights of the elementary edit operations of P

L (P) = number of these operations (length of P)

The string matching problem typically involves computation of edit distance. Many algorithms have been proposed to calculate edit distance efficiently and at a faster rate. The faster the computation of edit distance, the greater the increase in algorithm speed. It is also observed that normalized edit distance is a better measure of similarity than edit distance and it consistently provides better results than both unnormalized and post-normalized edit distances. One important point to be noted is that it is not possible to compute the normalized edit distance by calculating the unnormalized edit distance first and then normalizing it by dividing by the length of edit distance path.

Post Normalized edit distance [29]:

The post normalized edit distance between two strings **`string1`** and **`string2`** is computed by first calculating the edit distance between these two strings and then dividing this distance by the number of edit operations used.

1.3 Previous Work

Several algorithms have been suggested for exact string matching. These algorithms preprocess the search string to make the string search operation faster.

1.3.1 Exact String Matching Algorithms

Wagner and Fischer algorithm [5]:

This algorithm determines edit distance between the two given strings by using a matrix. The edit distance is computed by defining a cost function on a graphical structure called '**trace**'. Trace is a description of how an edit sequence S converts **string1** to **string2**, ignoring the order of operations and redundancy in S. It is observed that for every trace 'T' from **string1** to **string2**, there exists an edit distance equal to cost of the trace, and for every edit distance required to convert **string1** to **string2**, the cost of trace is less than or equal to the edit distance. The cost of trace is computed by the addition of three operations; substitute, delete, and insert. The algorithm concentrates on finding the minimum cost trace from **string1** to **string2**. The computational complexity of this algorithm is proportional to the product of lengths of two strings.

Knuth–Morris–Pratt algorithm [30]:

This algorithm searches for a word W within a string S. The comparison is carried out from left to right in a word stream. This algorithm builds tables along the process of searching and draws appropriate conclusions when there is a mismatch. Also it provides an intelligent estimate that predicts approximately where the next match could begin. This prediction process avoids re-examination of previously matched characters and saves time enhancing the overall computation speed. The functionality of the algorithm is explained with the help of the following example:

String: **BEG BBEGIERE BEGIN CODE**

Word: **BEGIN**

The search starts with comparing the first character of the word and the string. Since there exists a match, the next characters are compared until the letter '**G**' is found to be a successful match. In the next search operation, there is a mismatch as the string has a 'space' and the word has an '**I'** character. Hence the search operation is halted. As '**I**' doesn't appear in the first 3 characters of string compared previously, the comparisons starts with '**B**' after the space in the string and the '**B**' of the word. Again, since there is a match, the next character is compared which is found to be a mismatch. At this point, the next character in the string is '**B**' and hence the comparison starts with that character as it could be the start of the actual word that the algorithm is searching for. The comparison continues in the similar fashion until the desired word is found. In this example, we saw that a wise decision is made while resuming the comparison after the mismatch is found thus eliminating an unnecessary comparison.

Efficiency / Performance:

Assuming the prior existence of the table 'T', the search portion of the Knuth-Morris-Pratt algorithm has complexity $O(n)$, where 'n' is the length of the string. The overall complexity along with the computation of the tables and search operation is $O(m + n)$, where 'm' is the length of pattern. But this algorithm suffers when a word needs to be searched in a string containing a large amount of repetitions.

Boyer Moore algorithm [29]:

This algorithm follows a different approach in preprocessing the strings. Instead of preprocessing the string to be searched it preprocesses the target string that is being searched for. It scans the characters of the pattern from right to left, beginning with the rightmost character, however the pattern is moved from left to right across the string. In case of a mismatch (or a complete match of the whole pattern) it uses two pre-computed functions to shift the pattern to the right. It doesn't need to actually check every character of the string to be searched but rather skips over some of them. The algorithm pre-computes two tables to process the information whenever it finds a mismatch. The first table calculates the subsequent positions from where the next search will start and the second table makes similar calculations depending upon the matched characters before the mismatch occurred. This information is very useful to decide upon the next point to start the search and rules out as many positions of the text as possible where the string cannot possibly be matched and thus saves computation time.

It attempts to check whether a match exists at a particular position and works in a backward fashion. For example, if the algorithm starts the search at the beginning of a pattern for the word '**ABCDEFGH**', it checks the eighth position of the text to see if it contains an '**H**'. If it finds the '**H**', it moves to the seventh position to see if that contains the last '**G**' of the word, and continues towards right until it checks the first position of the text for an '**A**'.

Suppose there is a character '**Z**' instead of an '**H**' in the eighth position and '**Z**' doesn't appear anywhere in the whole string, then there is no point to search the first through seventh string positions. Once the algorithm finds that there is no match in the

last character position, it skips an entire word length in the pattern and again compares the last character in the pattern. This approach can prevent many unnecessary comparisons from occurring.

Performance /efficiency:

The Boyer-Moore algorithm has sublinear execution time with searching complexity of $O(n)$.

Below is a table containing summary of complexities of algorithms explained so far in terms of lengths of strings (m, n).

Table 1.2: Performance Summary Of Algorithms

Sr. No.	**Name of Algorithm**	**Preprocessing Complexity**	**Searching Complexity**
1	Wagner and Fischer	$O(m)$	$O(mn)$
2	Knuth-Morris-Pratt algorithm	$O(m)$	$O(m + n)$
3	Boyer-Moore algorithm	$O(m)$	$O(n)$

1.3.2 Approximate String Matching (ASM) Algorithms

This is a technique in which, strings are said to be similar when the value of edit distance is less than some particular threshold say ‘k’. Approximate string matching is an important operation to reduce errors, occurring due to misspelled words, in text processing. Hence, any spell check program, which implements approximate string

matching, must be able to find the match for the incorrect word as close to the options available in the dictionary as possible.

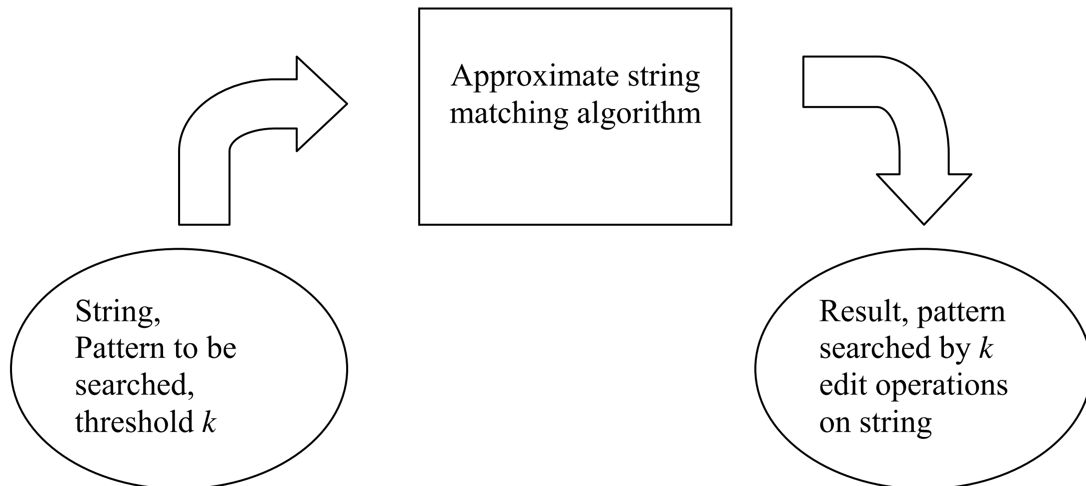

Figure 1.1: Approximate String Matching

Ukkonen's dynamic programming algorithm:

The dynamic programming algorithm can be explained with the help of the following example. The edit distance table for the comparison between strings '**ONE**' and '**ON**' is as follows:

Table 1.3: Edit Distance table for comparing '**ONE** 'and '**ON**'

		O	N
	0	1	2
O	1	0	1
N	2	1	0
E	3	2	**1**

Cost of insert operation: 1

Cost of delete operation: 1

Cost of substitute operation: 2

The difference between two strings is the value in the lower right corner of this matrix. Hence this result 1 means that the string '**ONE**' can be transformed to string '**ON**' by 1 deletion operations i.e. deleting character '**E**'.

The dynamic programming is accomplished using the matrix in the figure above. It builds the edit distance table and computes the edit distance between the two strings. Ukkonen has suggested an extension to this algorithm. The new algorithm has an asymptotic complexity similar to that of Ukkonen's algorithm, but it is significantly faster due to a decreased number of array cell calculations. Reported experimental results [1] indicate a 42 % increase in speed is achieved in applications involving name comparisons. This percentage increases by considerable amount when longer and dissimilar strings are compared. Although this speed is comparable to other fast ASM

algorithms, this particular approach has greater effectiveness in text processing applications.

The implementation of Ukkonen's algorithm to find the minimum edit distance between two strings is based on a dynamic programming approach. The complexity is O(*dist* x *length*) in both time and space, where '*dist*' is the distance between two strings, '*length*' being the smallest length of the two strings. This method is optimal, when the two strings are identical since only n comparisons are made for strings of length n. In Ukkonen's method, unnecessary calculations in the distance matrix are avoided in the threshold test of distance between the two strings. This process is repeated with successively larger threshold values until the test is successful, yielding the required string distance in $O(m \times d)$ time.

Finding approximate patterns in strings:

In this algorithm the approximate pattern matching problem is solved. Suppose there are two strings **`String1`** and **`String2`**. **`String2'`** is a string that has an edit distance of at the most'*t*' units from **`String2`**. **`String2'`** is searched in string **`String1`**. The definitions of edit distance and editing operations explained earlier are also applicable to the description of this algorithm.

The algorithm can be explained as follows:

String2 = $a_1, a_2, \ldots, a_m$

String1 = $b_1, b_2, \ldots, b_n$

The edit distance matrix is calculated row by row or column by column with the help of following equations:

$$ED_{0j} = 0 \qquad 0 \le j \le n$$

$$ED_{i0} = i \qquad 0 \le i \le m$$

$$ED_{ij} = \min \begin{Bmatrix} ED_{i-1, j-1_i} & \text{if } a_i = b_j \\ ED_{i-1, j-1} + 1, & \text{if } a_i \ne b_j \\ ED_{i, j-1} + 1, & \text{otherwise} \end{Bmatrix}$$

After the matrix is formed, the algorithm finds the value of the elements on the last row of ED_{ij}, that have an edit distance value of at the most'*t*' units. The execution time required for this algorithm is O(*m* x *n*).

An extension to this algorithm is suggested which divides the computation into two parts. In the first part, the pattern to be searched is preprocessed and in the second part the original string is scanned for the pattern. This method is less efficient in some applications because, after preprocessing, the scanning phase runs in time O (*n*).

Trie based algorithms [7]:

A trie is a tree data structure used to store strings. However, in keeping with the convention in the original paper [7], we will also use the term 'trie'.The text is represented as a Trie as shown in the figure below. In dynamic programming, an edit distance table of size '*m* x *n*' is required, but the trie representation of text drastically reduces the storage space as identical substrings in text are represented once.

A text consists of one or more sistrings. If sistrings start at word boundaries, then the text, "**`road rapid rolex rolling cupid castle`**“contains six sistrings. Figure 1.2 shows the sistrings and a corresponding index trie constructed over these sistrings.

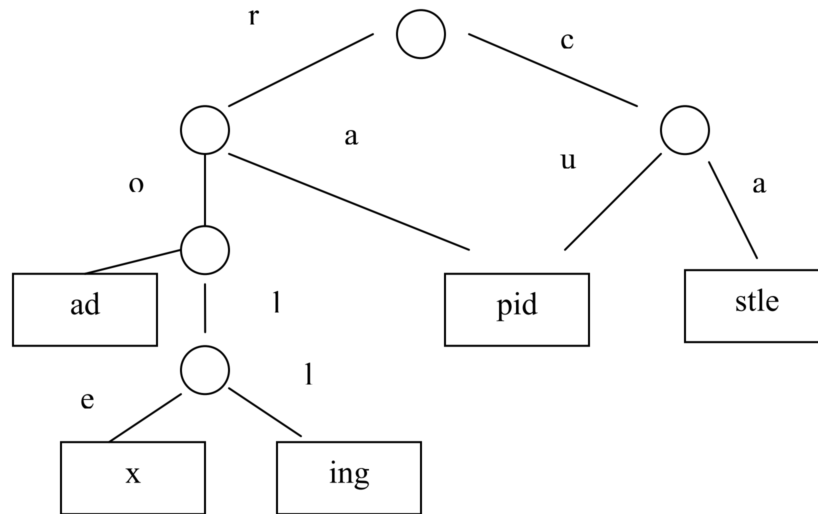

Figure 1.2: Trie Data Structure

Sistrings are indexed by their locations in the text. Edit distances between the sistring can be calculated by traversing the trie in depth-first manner, and simultaneously computing edit distance table. In case of searching a string, branching decisions at each node is made by each character of the string being sought. In the trie shown, while searching string ‘**`castle`**’, at first node both branches are checked for character ‘**`c`**’ and search proceeds in the direction of right branch. Again at the next node character branch containing character ‘**`a`**’ is selected. In the next step the pattern is found. This means that,

search time is proportional only to the length of the pattern string, and independent of the text size.

In trie representation the common prefixes like ‘**r**’,’**c**’,’**a**’ and ‘**u**’ of all sistrings are stored only once. This gives significant data compression, and turns out to be useful while indexing large texts. When the text size is huge, instead of having one huge tree, a set of trees pointing to smaller sub-trees can be constructed. As sharing of common prefixes is performed in a trie structure index space as well as search times are reduced. And hence many sub-tries are bypassed as it is not necessary to evaluate every sistring in a trie. As seen from example above the text searches performed with tries are independent of the size of the text being searched, and therefore they are preferred for large text size searches.

Linear systolic array for ASM / VLSI ASM [12]:

This algorithm can be explained with following example. Comparison between strings ‘**TGGC**’ and ‘**ACTG**’ can be represented with an edit distance table identical to that used in Ukkonen’s algorithm as follows:

Table 1.4: Edit Distance table for '**`TGGC`**' and '**`ACTG`**'

		T	G	G	C
	0	1	2	3	4
A	1	2	3	4	5
C	2	3	4	5	4
G	3	4	3	4	5
T	4	3	4	5	**6**

Cost of insert operation: 1

Cost of delete operation: 1

Cost of substitute operation: 2

The difference between two strings is the value in the lower right corner of this matrix. Hence this result 6 means that the string '**`ACGT`**' can be transformed to string '**`TGGC`**' by 3 substitution operations i.e. substitute '**`A`**' by '**`T`**', '**`C`**' by '**`T`**' and '**`T`**' by '**`C`**'.

The dynamic programming is by building the edit distance table and computing the edit distance between the two strings. This process requires '*m* x *n*' processing cells to process strings of lengths '*m*' and '*n*', and in order to exploit parallelism '*m*+ *n*' inputs need to be provided in parallel during each clock cycle.

In this method it is observed that many elements of the edit distance table can be computed simultaneously which will reduce the string comparison time tremendously. It is possible to calculate the elements along the diagonals simultaneously. These calculations are performed through a systolic array. The systolic array consists of

processing cells which are involved in calculating the elements of the edit distance table. One processing cell calculates all elements of a diagonal. Hence, if the two strings have length '*m*' and '*n*' respectively, there will be '*m*+*n*-1' diagonals and hence total number of processing cells required is '*m*+*n*-1'. The overall systolic architecture to compute the edit distance table can be shown pictorially as follows:

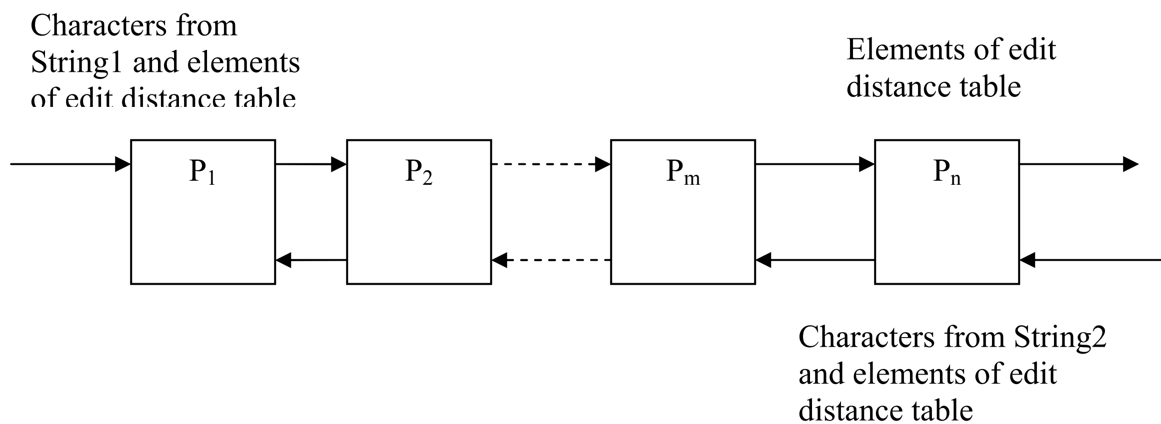

Figure 1.3: Architecture Of Linear Systolic Array

P_1, P_2,…, P_n: Processing cells

As seen from the architecture above, the two strings to be compared are shifted in from opposite ends. Along with these strings, alternately the values from first row and column are also sent. The processing cell calculates the new entry in the edit distance table by considering the value of two numerical shifted in and comparison of the characters and passes it to the next processing cell. This process continues until the last element in the edit distance is computed which is actually the result. The strings, while

being output from the array, carry the result of the comparison along with them. The straightforward implementation of this approach requires a bus width equal to the number of bits required to represent each character. For example, UNICODE characters would require a buswidth of 16 between each cell since each character is represented by 16 bits. It is observed that adjacent elements of edit distance table differ by small amounts, therefore instead of computing the element, its difference is computed with its left and top element [11]. This reduces the buswidth required between adjacent cells and the computations are also more efficient. An encoding scheme is proposed which computes differences in the elements of the distance table rather than computing the element [11]. The detailed structure of the processing cell is shown in figure 1.4:

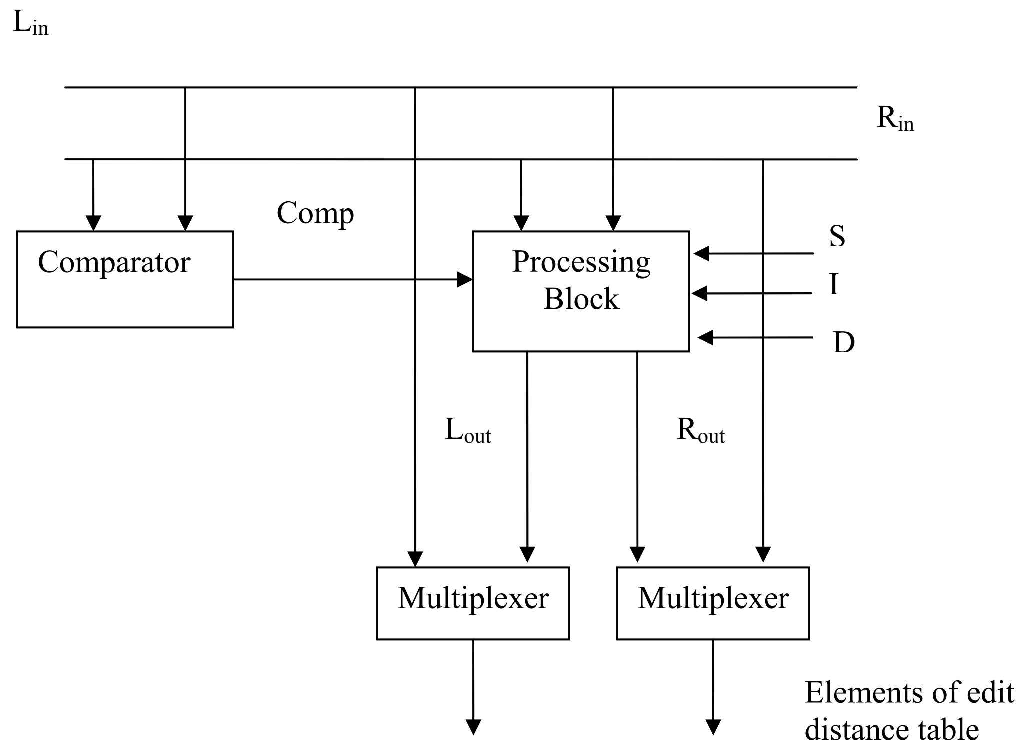

Figure 1.4: Internal Architecture Of Processing Cell

S: Cost of substitution.

I: Cost of insertion.

D: Cost of deletion.

$$R_{out} = \min \begin{Bmatrix} L_{in} + I \\ R_{in} + D \\ C \end{Bmatrix} - R_{in}$$

$$L_{out} = \min \begin{Bmatrix} L_{in} + I \\ R_{in} + D \\ C \end{Bmatrix} - L_{in}$$

$$C = \begin{Bmatrix} 0, comp = 1 \\ S, comp = 0 \end{Bmatrix}$$

In the first clock cycle, characters from both the strings enter the processing cell, which are then compared in the comparator. On the next clock cycle, values from the edit distance table i.e. L_{in} and R_{in} shift in and the existing characters shift out. As shown in the above equations, these values are modified based on the result of character comparison and values of S, I, and D. In order to compute the difference, values of R_{in} and L_{in} are subtracted from the respective calculated values. These values are passed to the next processing element which computes the differences in similar manner. This process continues until the element from last row and last column is computed. The strings shift out of the last processing cell followed by this element. Edit distance between two strings can be computed from this element.

For strings that are longer than the number of cells in the linear array, the comparison can be performed in multiple passes. During each pass the element values of the previous row and column are shifted in along with the string characters.

Performance:

The linear systolic array technique is a rapid way of performing string comparison due to hardware acceleration. The time required to compare two strings of same length is given by:

For a single pass, $Time = (Length + (No.of.PE)/2) * Clockperiod$

For multiple passes,

$$Time = ((m-1) * 2 * (No.of.PE)/2) + n + (No.ofPE/2)) * Clockperiod$$

1.4 Organization Of Thesis

This thesis is organized in 5 chapters. It begins with a brief background of hardware and software fundamentals. In chapter 2 previous contributions to string matching problems has been mentioned. Implementation details about the prototype are explained in the chapter3.Chapter 4 lists the experimental results followed by an analysis section. Chapter 5 concludes with future work suggestions to enhance the prototype system.

Chapter 2

SOFTWARE AND HARDWARE FUNDAMENTALS

2.1 Basics Of Communication

Digital data communication is the process of sending data represented in terms of bits that are modeled as streams of 1's and 0's, from one place to another .There are two means of digital communication:

- Serial communication
- Parallel communication

As the name suggests serial communication is the process of sending all data bits serially whereas in parallel communication multiple data bits are transmitted concurrently. Serial communication can be synchronous or asynchronous in nature. In synchronous communication, data transfer is dependant on a common clock signal at the transmitter and receiver indicating the beginning of each transfer. Whereas in asynchronous communication methods data transfer does not depend on a clock but on a predefined bit pattern or "start frame" that precedes the data transfer. One important advantage of asynchronous communication is that it requires fewer lines and hence is comparatively inexpensive. For synchronization purpose a lager number of bits are required per transmission to account for the start and stop frames.

EIA-232C Serial Protocol [28]:

This section provides a detailed description of the asynchronous serial communication protocol standard RS-232C (Recommended Standard 232) [28] which is also known as EIA-232C. This standard defines an asynchronous serial communication protocol over a physical channel that is typically implemented as a cable between from a DTE (Data Terminal Equipment) and a DCE (Data Circuit-terminating Equipment). Each bit of a data is transmitted independent of any clock or predefined time slot. Before the actual communication, the transmitter and the receiver agree upon the format of data to be transmitted and the baud rate, which is the rate at which the data will be transmitted. This agreement ensures that the receiver detects the start and the end of the data and receives the data correctly. Hence these initial steps are very important to avoid misinterpretation or loss of data.

To transmit any kind of data, initially a start bit is transmitted, followed by the actual data bits at the baud rate, and lastly one or more stop bits are transmitted. To enable error detection at the receiver side**,** a parity bit can also be transmitted before the stop bit. This can be pictorially represented as follows:

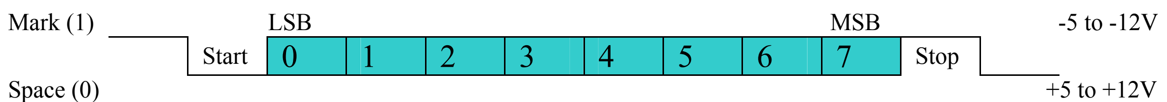

Figure 2.1: RS-232C Logic Waveform (8N1)

Start bit: This bit is transmitted to indicate the start of a new data frame at the receiver. This bit is logic 0 bit. In the idle state, the communication line is maintained at

logic 1 and hence being of opposite logic, the start bit informs the receiver about the start of a data-frame transmission.

Data bits: Data bits are transmitted immediately after transmitting the start bit. Each word can be represented by 5 to 8 bits. The least significant bit is always the first bit to be transmitted according to the EIA-232C standard. For example if we want to send an ASCII character '**A**' (65_d), the data will be transmitted as 'start bit –data with LSB first – stop bit'. This can be represented pictorially as follows:

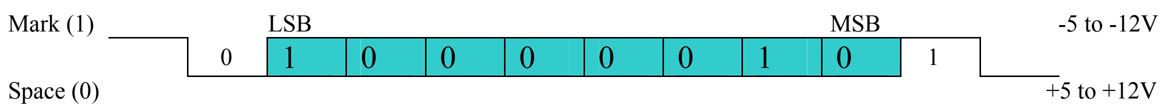

Figure 2.2: Serial Transmission Of Character '**A**'

Parity bits: To enable error detection at the receiver side, an extra bit referred to as the parity bit is transmitted after the data bits. It is optional to send this bit and it is a simple way to encode the data word for single-bit error detection. The value of this bit is calculated depending upon the data transmitted. The parity can be set as either odd or even. The same parity calculating procedures are followed by both the transmitter and the receiver for successful error detection. In the case of odd parity, the parity bit is adjusted to make sure that an odd number of 1's are transmitted to the receiver which means if the data to be transmitted has four 1's, then to construct an odd parity data frame, the parity bit '1' is transmitted, to make sure that an odd number of ones reach the receiver. To generalize, for odd parity, if the number of logic 1 data bits is even, a logic 1 is appended to the parity bit otherwise a logic 0 is appended. Similarly, in the case of an even parity,

the parity bit is adjusted so that the number of data bits transmitted will always contain an even number of logical 1's which means if the data bits contain five 1's then the parity bit is set to 1 so that the total number of logic 1's reaching the receiver are 6, which is an even number. So, for an even parity, if the number of logic 1 data bits is odd, the parity bit of logic 1 is appended otherwise a logic 0 is appended.

Stop bit: This bit indicates the end of the data frame. One or more stop bits can be transmitted depending upon the convention agreed upon between the transmitter and the receiver. When the data transmission is to be terminated, the stop bit is set to logic 1. Because the length of each data frame is fixed, the receiver knows when to expect a stop bit. If the receiver detects a value other than logic 1 when the stop bit is expected to be present on the line, some problem with the synchronization is detected. As an example a receiver might miss a start bit due to some noise in the transmission channel. The stop bit can have different lengths i.e. it can be 1, 1.5 or 2 bits. 1.5 bits are used only with the words of 5 bits length and 2 bits are used for longer words. A stop bit having a length of 1 bit is commonly used for all data word sizes.

Data frame format: 1 Start bit, 8 Data bits, No Parity, 1 Stop bit. The data transmission starts with a Start bit, followed by the data bits (LSB is sent first and MSB is sent last), and ends with a "Stop" bit and is known as 8N1. Other formats are possible such 7E1 indicating 7 data bits, an even parity bit, and a single stop bit.

Physical properties: In the EIA-232C standard, logic 1 and 0 are defined in terms of voltage levels. Logic 1 is defined as voltage -3 to -15 Volts. It is referred as a “mark”, and it has a functional significance of the OFF state. Logic 0 has voltage between 3 to 15

Volts. It is referred to as “space”, and it has a functional significance of an ON state. Therefore, valid signal ranges are ± 3 to ± 15 volts.

Commonly-used signals [28] are as follows:

- Transmitted Data (TxD): Data sent from DTE to DCE.
- Received Data (RxD): Data sent from DCE to DTE.
- Request To Send (RTS): Asserted (set to 0) by DTE to prepare DCE to receive the data.
- Clear To Send (CTS): Asserted by DCE to acknowledge RTS and allow DTE to transmit.
- Data Terminal Ready (DTR): Asserted by DTE to indicate that it is ready to be connected.
- Data Set Ready (DSR): Asserted by DCE to indicate an active connection.
- Data Carrier Detect (DCD): Asserted by DCE when a connection has been established with remote equipment.
- Ring Indicator (RI): Asserted by DCE when it detects a ring signal from the telephone line.

Connectors: RS-232 devices may be classified as Data Terminal Equipment (DTE) or Data Circuit termination Equipment (DCE).Terminals typically have male connectors with DTE pin functions, and modems typically have female connectors with DCE pin functions. Other devices may have any combination of the above two connectors and their respective pin functions.

Maximum cable lengths: This standard specifies the maximum cable length of 15 meters or less (total equivalent capacitance of 2500 pF). The cable length given by this standard allows for communication with the maximum baud rate specified. If a longer cable is required, the baud rate may need to be decreased .However, care must taken so that signal levels do not fall outside the ± 3 to ± 15 V range due to *IR* voltage drops depending upon the cable length.

Advantages of serial communication:

- Simple to use.
- Low cost.
- Availability of serial port on each PC.
- Long cable lengths possible (maximum cable length for EIA-232C is 15meters, whereas that for a USB cable is limited to 5 meters)

2.2 Fundamentals Of Verilog Programming

`Verilog` is a Hardware Description Language. It is used to design digital systems by describing their behavior via formal language. The **`Verilog`** description of any digital system can be transformed into gate level implementation through the use of automated circuit synthesis tools. The resulting gate level netlist can provide parameters for the analysis of design in terms of functional simulation, verification, and timing analysis.

Verilog is a case-sensitive language. It has a preprocessor like **C** and the syntax of **Verilog** and C exhibits some similarity. It has its own set of data types **wire** and **reg** and sets of logical, arithmetic and reduction operators along with a unique set of keywords.

There are two types of **Verilog** statements called procedural and concurrent statements. Procedural statements are placed inside '**always**' block and they are instantiated in the sequential order within the block. However, all concurrent statements are instantiated parallel. Hence, two or more '**always**' blocks are instantiated in parallel.

Verilog designs typically consist of a hierarchy of modules. Each module defines a set of inputs and outputs to communicate to the external world and a set of wires and/or registers, parameter definitions; and a set of sequential and concurrent statements to perform actual operations. Any module can either instantiate other modules or can be instantiated from other modules. A general structure of a module is shown below:

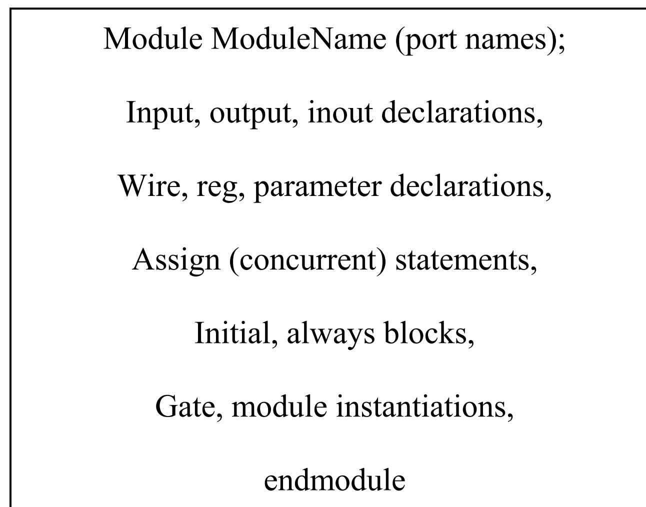

Figure 2.3: Structure Of A Module In **Verilog**

By the process of synthesis, Verilog description can be transformed into a netlist that describes the basic components and connections to be implemented in hardware. For example, an 'if-then-else' block may get transformed into a multiplexer.

A design can be described in different levels of abstraction which are as follows:

Netlist level model: This is the lowest level of abstraction where any design is directly expressed in terms of a set of connections of basic elements.

RTL level: The design is described in terms of Boolean expressions and registers. Such descriptions can be automatically synthesized to a netlist with the help of a CAD tool.

Behavioral level: This is a high level construct that specifies only functionality of the design.

Advantages of the **`Verilog`** language:

- It is easy and simple to learn.
- It also produces compact code which is easier both to write, and to read.
- It allows a design to be described in a number of styles and at various levels of abstraction namely behavioral, structural or lower-level implementations. Thus, the overall design with all these styles turns out to be very effective in the end.
- Many synthesis tools are available to allow very simple descriptions to be automatically translated into gate-level netlist.
- A design can be synthesized to actual hardware and can be simulated many times to achieve required functionality before actual fabrication. This detecting and removing bugs in the design reduces overall cost.

2.3 Programmable Logic Devices (PLD)

A PLD is a device which can be used to implement a digital circuit. The main feature of a PLD is that it contains resources that allow the internal circuitry to be configured into many different specific circuits, and hence can be "programmed" to realize a large amount of different circuits. There are many types of programmable logic devices such as Programmable Array logic (PAL), Programmable Logic Array (PLA), microprocessors, microcontrollers, CPLDs, and FPGAs. Each of these devices differs in terms of the number of gates, programmable resources and offer different design tradeoffs.

Advantages of PLD:

- Each PLD has many macrocells which ultimately consist of numerous combinational programming resources such as AND gates, OR gates, and Flip-Flops. Hence one or more Boolean equations can be easily realized. One chip can contain a design ranging from least complex to most complex, and hence it requires less board area, power, and wiring than several other chips.
- Design inside the chip is flexible, so a change in the logic doesn't require any rewiring of the board.

Field Programmable Gate Arrays (FPGAs):

This programmable device can be viewed as a sea of logic gates with programmable interconnects placed between them. These logic cells and interconnects can be programmed by the designer, after the FPGA is manufactured, to implement a logical function, hence the name is "field-programmable".

The basic building block of FPGA is called a logic cell. These logic cells can be arranged in the form of rows and columns with programmable interconnects between them. A logic cell must be classified as a functionally complete logic family with which any design can be built with them .It can be universal gates such as NAND gates or NOR gates or 2:1 multiplexers or lookup tables along with a D flip flop which can be built with any type of logic. In addition to these, other features such as dedicated memory elements, ALU units, phase locked loops for clock synchronization, might be included.

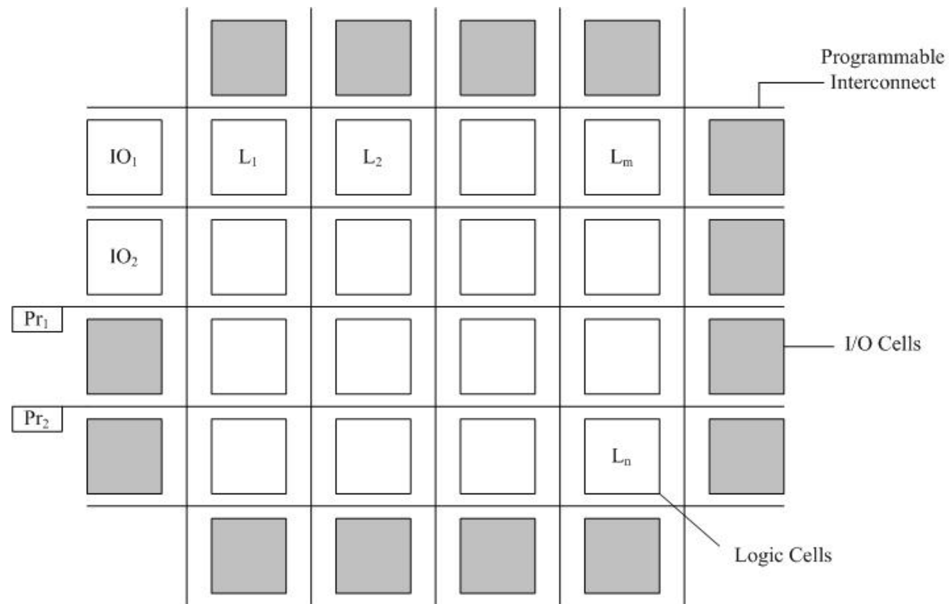

Figure 2.4: Internal Structure Of FPGA

L_1, L_2: Logic cell

IO_1, IO_2: I/O cells

Pr_1, Pr_2: Programmable interconnection resources

The figure above shows a simplified FPGA architecture. It consists of logic cells (L_1, L_2…), programmable interconnect resources (Pr_1, Pr_2…), and I/O elements (Io_1, Io_2…). Each I/O cells are programmable to configure the pins either as unidirectional (input / output) or as bidirectional. These I/O cells surround a set of logic blocks and programmable interconnect resources. These interconnects allow connection between different logic blocks as well as between logic block and I/O block.

Advantages of FPGA design methodologies:

- Rapid design development process and hence shorter time to market, ability to re-program in the field to fix bugs and lower non-recurring engineering costs.
- FPGA design flows support the use of CAD tools to perform tasks such as timing analysis, formal verification, and RTL and gate level simulation.

Basic Design process with FPGAs:

The figure 2.5 shows different steps involved in designing a digital system. The design process starts of with the specifications of the design which include the set of inputs, set of outputs, timing constraints and the functionality required. After the specifications are understood, the actual design process starts. A variety of CAD tools may be used to implement a design. These tools allow the initial design to be described in the form of block diagram or in the form of a code in **`Verilog`** / **`VHDL`** or set of connected gates and library components.

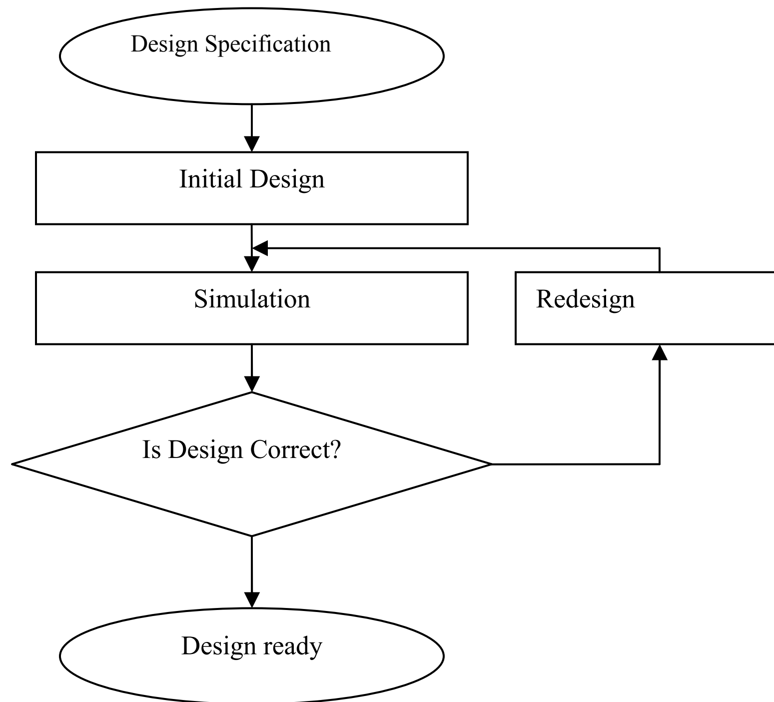

Figure 2.5: Design Cycle With FPGA Devices

After initial compilation, a series of simulations are usually carried out. The simulator uses user defined inputs, simulates the design, resulting in a set of output waveforms. These output waveforms are compared with the reference waveforms in the design specification to verify functionality of the design for the defined input stimulus. If there is a mismatch, the design is modified to rectify the logical errors, and then it is recompiled. The design is modified to remove the errors if any and then compiled again. The process is continued until the desired functionality is obtained.

Programmability options of FPGAs:

FPGAs can be programmed with the help of a separate piece of equipment, a device programmer or can be in-circuit programmable. The following programmability options are available for FPGAs:

- Antifuse - One-time programmable.
- UV-Erasable- Erased with ultraviolet (UV) light. Programmability limited to 1000 cycles.
- EEPROM - Electrically erasable as well as programmable.
- SRAM – Configuration bits are stored in SRAM. Unlimited programmability.
- Flash - Unlimited programmability.

Points to be considered in choosing a device:

A particular type of FPGA chip is selected based on the application which it is going to be used for. The important factors considered while making a particular choice are availability of gates, number of Input / Output pins, cost of the chip, availability of CAD tools for synthesis and simulation of design, expected performance, and the power consumption.

2.4 C String Library <string.h>

String.h is a built in standard library for performing different string operations in C. These functions include string comparison, string length calculation, string

concatenation, string copying, finding the index of substring and so on. The following list shows the detailed string operations along with example:

- **`int strcmp(const char* s1, const char* s2):`**

  `Input: s1 = "abc", s2 = "abc"`

  `Output: 0`

  The **`strcmp`** function compares two strings s1 and s2 and returns an integer indicating the result of comparison. If the return value is 0, then the strings match. If the return value is less than zero (-1), then s1 is lexically less than s2. If the return value is greater than zero (1), then s1 is lexically greater than s2.

- **`char* strcat(char*dest, const char* src):`**

  `Input: dest = "abc", src = "def"`

  `Output: "abcdef"`

  The **`strcat`** function concatenates one string to the other string and returns a pointer to the destination string.

- **`char* strcpy(char* dest, const char* src):`**

  `Input: dest = "abc", src = "def"`

  `Output: dest = "def"`

  The **`strcpy`** function copies one string to another.

- **`int strlen(const char* s):`**

  `Input: s = "abc"`

  `Output: 3`

  The **`strlen`** function calculates and returns the length of given string.

- **char* strstr(const char* str, const char* substr):**

  Input: str = "abcdef", substr = "bc"

  Output: "bcdef"

  This function finds the occurrences of a given substring within another string and returns a pointer to its first instance. If the sub-string can not be found, a NULL pointer is returned.

- **char *strncat(const char *string1, char *string2, size_t n):**

  Input: string1 = "abcd", string2 = "efg", n = 2

  Output: "abcdef"

  This function appends the specified 'n' characters from string2 to stringl.

- **int strncmp(const char *string1, char *string2, size_t n):**

  Input: string1 = "abcde", string2 = "abcfg", n = 3

  Output: 0

  This function compares first n characters of two strings. If n is zero, this function will always return zero – and no characters are checked, so no differences are found.

- **char *strncpy(const char *s1,const char *s2, size_t n):**

  Input: s1 = "abc", s2 = "efgh", n = 2

  Output: string1 = "ef"

  This function copies first n characters of s2 to sl.

- **`int strcasecmp(const char *s1, const char *s2):`**

  `Input: s1 = "aBc", s2 = "AbC"`

  `Output: 0`

  This function compares s2 and s1 but by ignoring the case of the characters. It is a case-insensitive version of **`strcmp()`**.

- **`int strncasecmp(const char *s1, const char *s2, int n):`**

  `Input: s1 = "aBcghy", s2 = "AbCdfde", n = 3`

  `Output: 0`

  This function is a case insensitive version of strncmp().

- **`size_t strspn(const char *s1, const char *s2):`**

  `Input: s1 = "abcdef", s2 = "abctyu"`

  `Output: 3`

  This function returns the number of characters at the begining of s1 that match s2.

- **`size_t strcspn(const char *s1, const char *s2):`**

  `Input: s1 = "abcdef", s2 = "ergdef"`

  `Output: 3`

  This function returns the number of characters at the begining of s1 that do not match s2.

- **char *strrchr(const char *string, int c):**

  Input: string = "This is a sample string", string2 = "s"

  Output: 18

  This function finds last occurrence of character‘s’ in string.

Chapter 3

IMPLEMENTATION DETAILS

3.1 Altera DE2 Development And Education Board

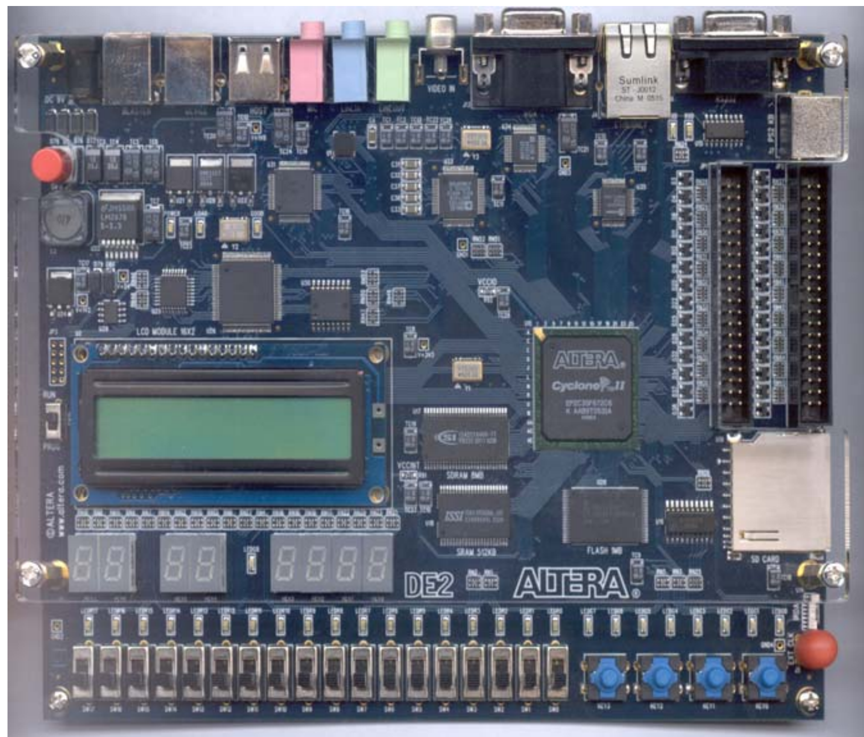

Figure 3.1: Altera DE2 board

The figure above shows the Altera DE2 board [22] used for implementing different string functions. This board contains a Cyclone II EP2C35F672C6 FPGA (672 pins), which is programmed based on a Verilog description of various string operations.

This board also provides support circuitry including toggle switches, push button switches, LED's, 7 segment displays, LCD, SRAM, SDRAM, Flash memory chips, RS-232 and PS/2 ports, and many more components which can be interfaced with the cyclone FPGA to provide inputs or to show the results.

This board is also equipped with a NIOS II processor, standard connectors for a microphone, Line-in, Line-out (24 bit codec), video-in (TV Decoder), and a VGA (10 bit DAC) for advanced applications.

3.2 QuartusII

This software tool is produced by the Altera programmable logic device company and provides complete support for the design of circuitry to be implemented in FPGA and CPLD device families available from Altera. It is compatible with a wide variety of device families like MAX II, Stratix II, Stratix III, Apex II, Flex 10K, Cyclone, CycloneII, and Cyclone III. Hence it can be used to program the FPGA, Cyclone II EP2C35F672C6, which is available on the DE2 board. In this work, we developed **`Verilog`** descriptions for various string operations and synthesized them into hardware netlists using the QuartusII CAD tool that were then used to configure the FPGA on the DE2 board.

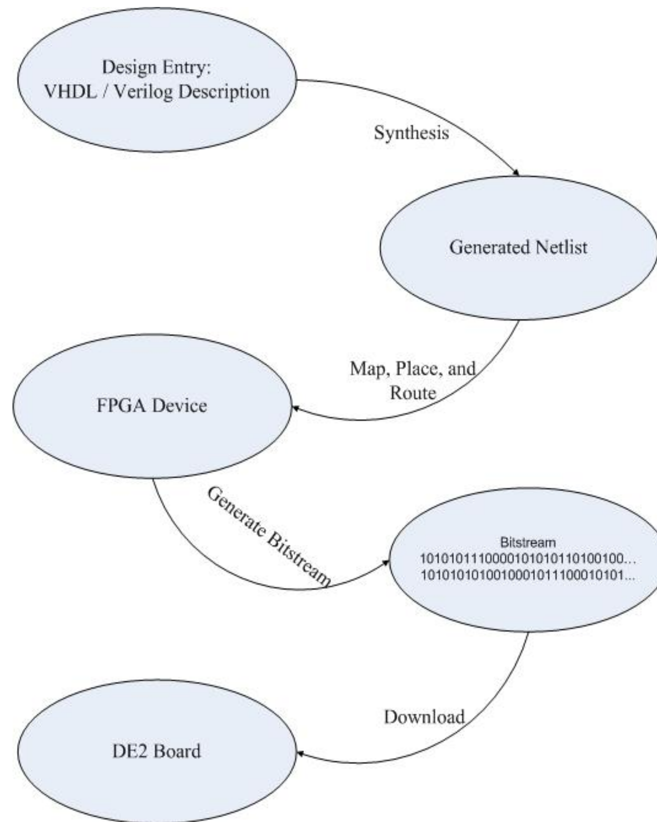

Figure 3.2: Design Flow On QuartusII Tool

QuartusII allows the initial design to be described in the form of block diagrams a HDL description in **`Verilog`** or **`VHDL`** and a set of connected gates and standard library components. The process outlined in Figure 3.2 involves the three steps given in the following section.

First, an intermediate representation of the hardware design is produced. This step is called synthesis and the result is a representation called 'netlist' which is a set of interconnected circuit elements, gates etc. Netlist is device independent; therefore its contents do not depend on the particulars of the FPGA or CPLD.

The second step is called place & route and involves mapping the logical structures described in the netlist onto the actual logic cells, interconnections, and input and output pins. The assignment editor allows assigning inputs and outputs of the Verilog code to the necessary pins of FPGA.

The result of the place & route process is a bitstream which is a stream of 1's and 0's. This format is device specific. The bitstream is then downloaded on the FPGA on DE2 board to configure the FPGA.

After initial compilation and before the actual downloading of the designs onto the FPGA, a series of simulations are carried out to remove logical errors in the design. The simulator receives user-defined inputs and simulates the design with the provided set of inputs and displays the output waveforms. As it's not possible to check the functionality for each and every possible input, a subset of inputs is applied to check the performance of the design. The design is modified to remove the errors if any and then recompiled again. This process continues until the design works as per the desired functionality.

3.3 Experimental Setup

Figure 3.3 shows the experimental setup for the string comparison operations we investigated. The DE2 board and the computer have 2 different sets of cables running between them. The USB blaster cable is used to download the bitstream corresponding to the design onto FPGA, whereas the RS-232 serial cable is used for the actual data transfer during the string operations.

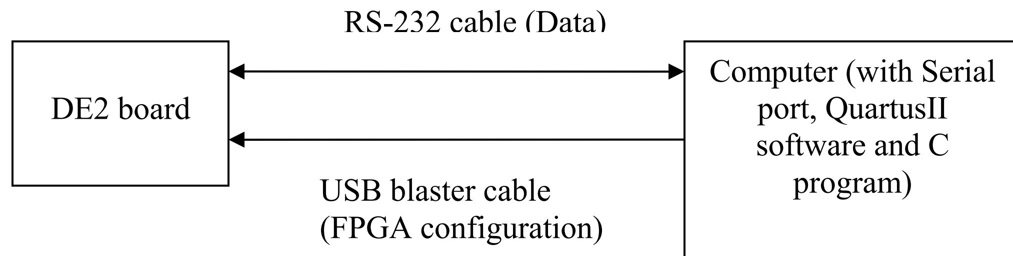

Figure 3.3: Experimental Setup

A serial cable is connected between the serial ports of DE2 board and the computer as per EIA-232C standard. This cable transmits strings from the computer to the DE2 board for performing the operations and receives the results of the string operations.

3.4 Serial Communication Through C Program

The following flow chart shows the working of C code for serial communication.

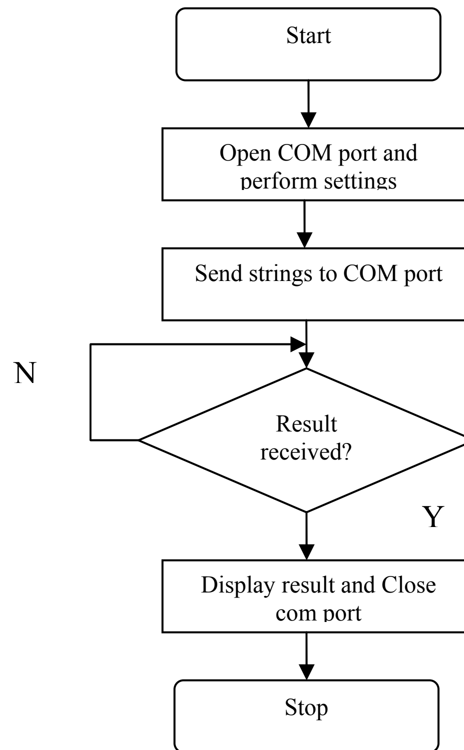

Figure 3.4: Flowchart Of Serial Communication Through C

This C code is basically used to send strings to the DE2 board for performing operations and then receiving the results back on the computer. This program initially opens the COM port with the help of CreateFile function. Then it sets the COM port for total number of bits, start bit, stop bits, and parity bits. It then sends two strings to the COM port through a function which calls Writefile function to send data to the port. The C code then waits for the result of string operation. ReadFile function is used to read the data from the port. Once the data is received, it is printed onto the console. The corresponding codes can be found in Appendix A.

3.5 Serial Communication Through **Verilog** Program [22]

3.5.1 Transmitter Block In **Verilog**

The **Verilog** Transmitter block transmits the result of string operations to the computer through serial port using EIA-232C standard. The result of string operations is buffered before it is transmitted. Transmitter block consists of two modules. One of them is to generate the clock as per the baud rate and other one is for instantiating scfifo (single clock FIFO) megafunction. On receiving the transmit enable signal the clock is generated according to baud rate. The data to be transmitted is saved in the internal memory block of FPGA through scfifo megafunction. Use of this megafunction ensures the transmission of data to be following the order in which they were received. After sending the logic 0 start bit, the data to be transmitted is shifted by one bit using a shift register which outputs one bit starting from LSB. This bit is transferred at the rising edge of baud rate clock. A separate bit counter keeps track of number of bits to be transmitted. When all the bits have been transmitted, a logic 1 stop bit is sent. The corresponding codes can be found in Appendix C.

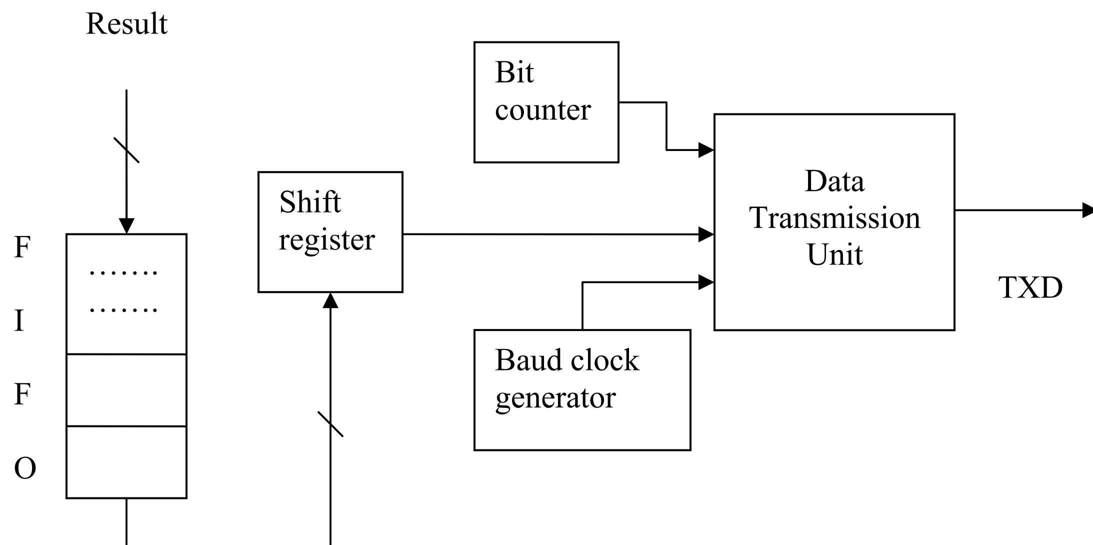

Figure 3.5: Transmitter Block In **Verilog**

3.5.2 Receiver Block In **Verilog**

The **Verilog** Receiver module receives strings for performing operations, through serial port, using EIA-232C standard. It consists of three modules, two of which are similar to the modules from transmitter block. When data is received, the data bits are accepted one by one based on the baud rate clock. The start and stop bits are removed, and the 8 bit data corresponding to the string is obtained with the help of shift register. This data, which is a character from one of the strings, is then written to the memory block using scfifo megafunction. The strings are still bounded by **STX** and **ETX**. Once a character is received the processing block starts checking for **STX** and **ETX** flags, and after removing these special characters the strings are ready to get operated. This operation can be pictorially represented as that in figure 3.6. The corresponding codes can be found in Appendix C.

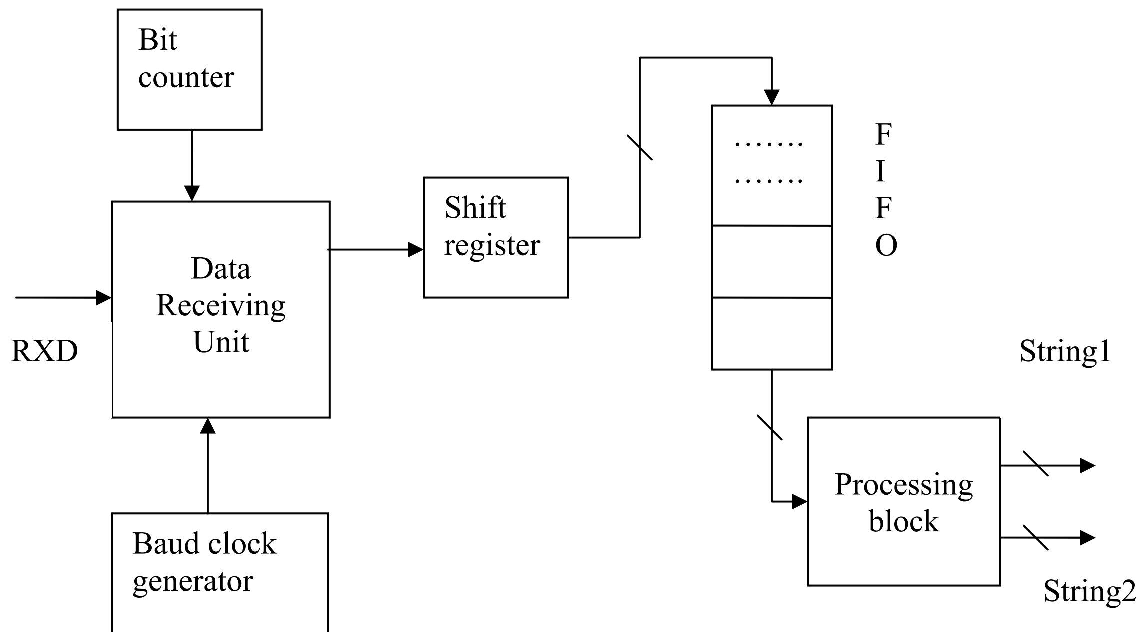

Figure 3.6: Receiver Block In **`Verilog`**

3.6 **`Verilog`** Implementation Of String Functions

1) **`strcmp:`**

`Input: String1, string2`

`Output: 0 if string1 is not equal to string2`

`1 if string1 is equal to string2`

This function receives **`string1`** and **`string2`** as inputs. It uses a simple X-OR function to decide if the strings are equal or not. The bitwise X-OR operation of two strings can be explained with the help of following truth table:

Table 3.1 Truth Table of X-OR Function

Input1	Input2	Output
0	0	0
0	1	1
1	0	1
1	1	0

When two strings are identical i.e. have exactly same bits, the result of X-OR operation yields logic 0 result otherwise the result is logic 1.The operation of function is case sensitive. Hence, if **`string1`** and **`string2`** are same but with different cases the result will be 0. These operations require a single clock cycle to finish. Figure 3.5 shows the functionality of **`strcmp`** function through ASM chart.

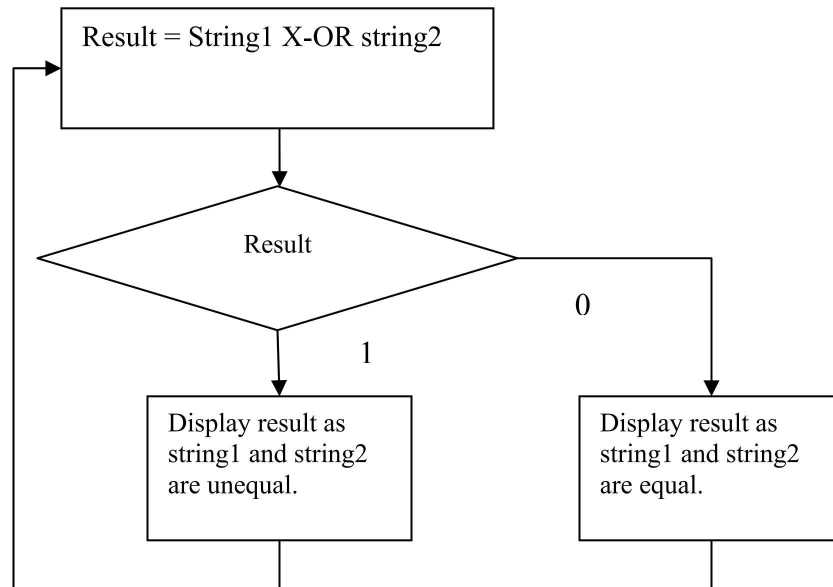

Figure 3.7: ASM Chart Of '**`strcmp`**' Function In **`Verilog`**

2) **`strcasecmp:`**

`Input: String1, string2`

`Output: 0 if string1 is not equal to string2`

`1 if string1 is equal to string2`

This is a case insensitive version of string comparison. It receives **`string1`** and **`string2`** as input arguments. The function checks the equality of strings character by character. Every time a character is compared to the other character, it is checked to see if it is a lower case letter or an uppercase letter. In ASCII representation lower case letters (a-z) have decimal values (97-122) and upper case letters A-Z have (65-90) values. A lowercase letter and uppercase letter differ by value 32 i.e. have opposite value of 6th bit. Hence, during each character comparison the 6th bit is ignored meaning that the case of the letter is ignored. Thus, if **`string1`** and **`string2`** are same but with different case

the result will be 1 otherwise result will be 0. Figure 3.6 shows the functionality of `strcasecmp` function through ASM chart.

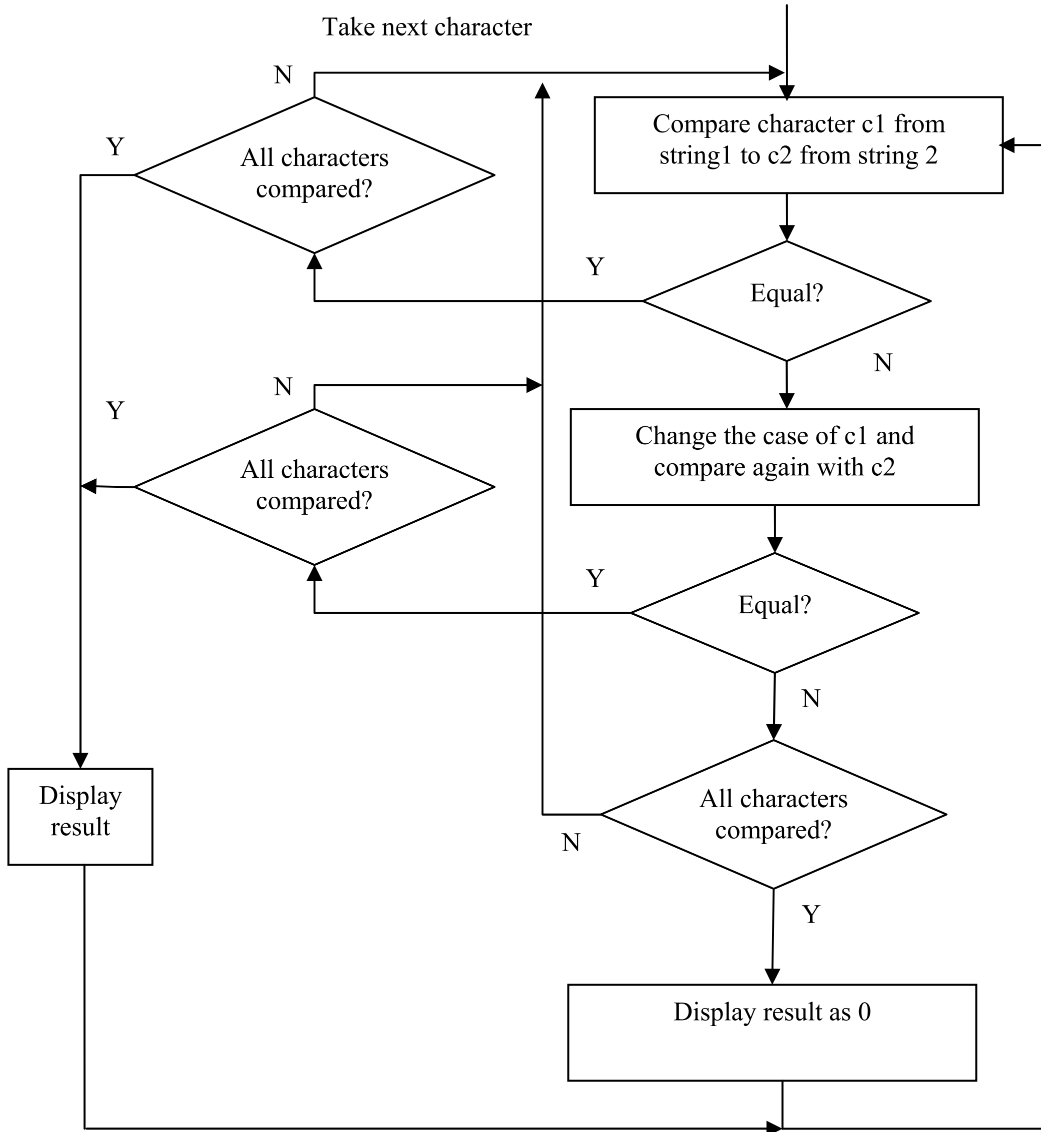

Figure 3.8: ASM Chart Of '**`strcasecmp`**' Function In **`Verilog`**

3) **`strstr:`**

`Input: String1, string2`

`Output: 0 if string2 is not found in string1`

`1 if string2 is found in string1`

This function searches for the first occurrence of **`string2`** in **`string1`** and when **`string2`** is found in **`string1`**, the result of **`string2`** is displayed as the output. This logic is implemented using state machine. The implementation is divided into two separate parts namely datapath and controller logic. The data path consists of shifter and comparator blocks which get control signals from the controller block. Controller is implemented as a state machine in which every state provides signals for shifting the **`string2`** and comparing it with **`string1`**.

Unless the strings are ready, logic keeps waiting in initial state state0.As soon as the strings are ready for comparison the state changes to state1 and signals to start comparison and shifting is provided. In state1, the output of initial comparison is checked. If it is logic 1, the position of **`string2`** in **`string1`** is displayed. Otherwise the control goes to the next state, state2 in which **`string2`** is shifted right by one character position. In each state the output of comparator is used to decide the signals to be given to the comparator and shifter blocks in next state. If the result of comparator is logic 1, then the shift and compare operations are stopped and the corresponding position of **`string2`** in **`string1`** is displayed. If the result of comparator is 0, these operations continue until the program finds a match or until the end of the **`string1`** and terminates when a match is found.

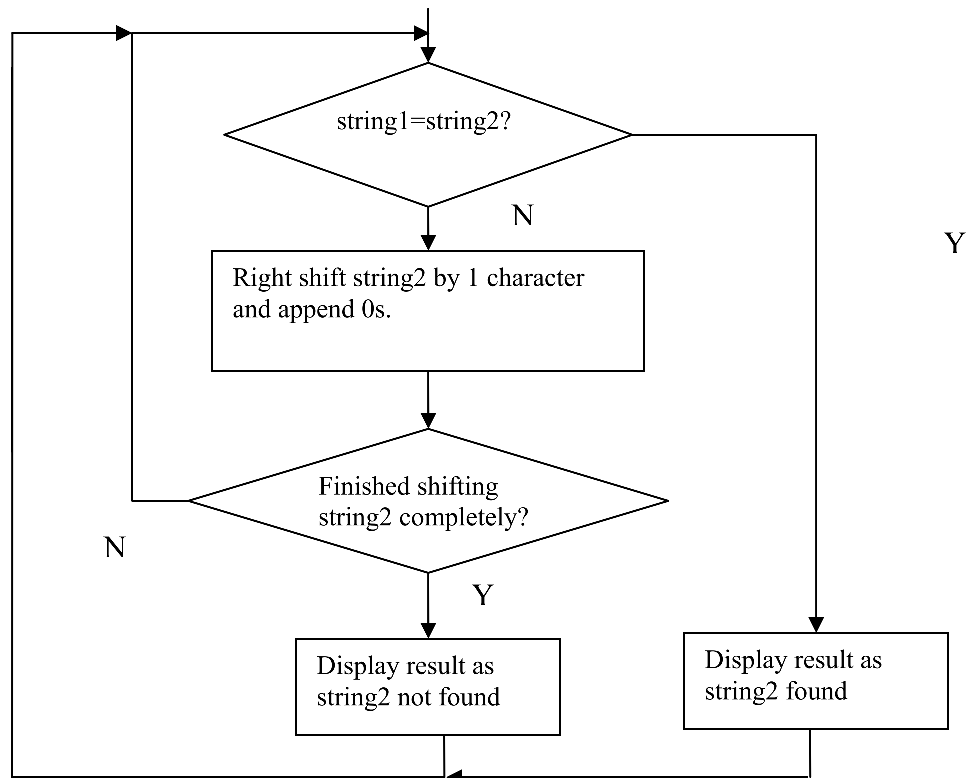

Figure 3.9: ASM Chart Of '**strstr**' Function In **Verilog**

4) **strchr: search a single character in string 1**

Input: String1, char1

Output: 0 if char1 is not found in string1

1 if char1 is found in string1

This function searches for the first occurrence of a particular character, **char1** in a given string, **string1**. For a string of length '*n*', this function is implemented with the help of '*n*' shifter and comparator blocks. Each shifter and comparator block right shifts a character by one position and perform a logical X-OR operation with the original string.

These '*n*' outputs are ORed together to give final result of search operation. For an OR logic when either of the input signals is logic 1 the result is logic 1.Hence whenever there is a match between the character in a string, one of the eight outputs will be high and hence the output of OR logic will be high indicating the occurrence of character in the given string. However, if the character is not found throughout the string, the results from all shifter and comparator block will be logic 0 and hence the ultimate result will be logic 0 indicating that character is not found in the string. The advantage of this implementation is that irrespective of the length of the string, the comparison is performed in 2 clock cycles.

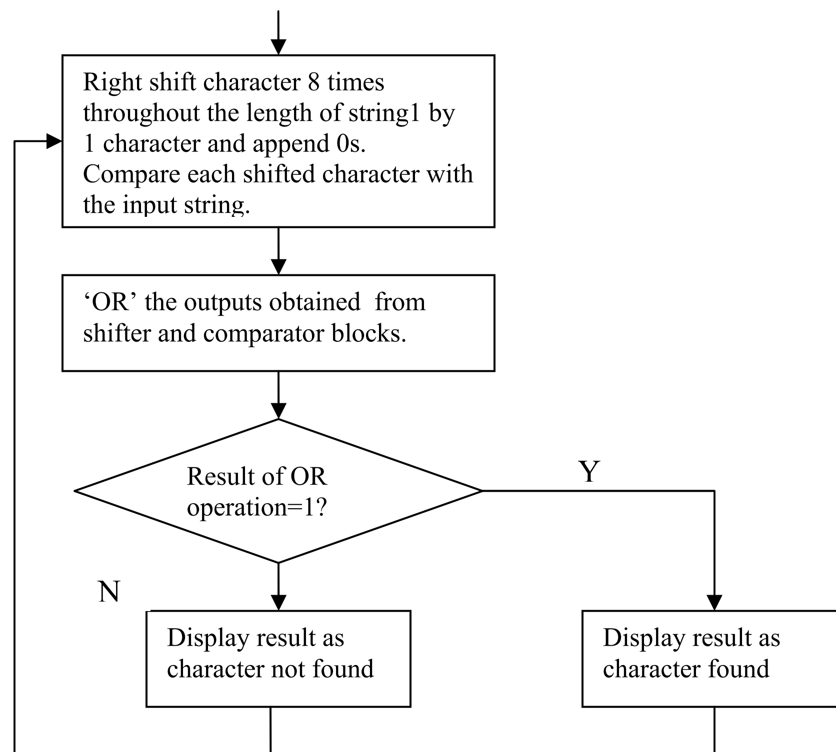

Figure 3.10: ASM Chart Of '`strchr`' Function In **`Verilog`**

5) **`strchr_pos (display the position)`**

`Input: String1, char1`

`Output: NULL if char1 is not found in string1`

`Position of char1 if char1 is found in string1`

This function searches for the first occurrence of a particular character, **`char1`** in a given string, **`string1`**. When the character is found in the given string, its position is determined otherwise it is set as NULL. For a string of length '*n*', this function is implemented with the help of '*n*' shifter and comparator blocks. Each shifter and comparator block right shifts a character by one position and perform a logical X-OR operation with the original string. These 'n' outputs are ORed together to give final result of search operation. For an OR logic when either of the input signals is logic 1 the result is logic 1.Therefore, whenever there is a match between the character in a string, one of the eight outputs will be high and the output of OR logic will be high indicating the occurrence of character in the given string. The position is determined by checking for logic 1 output of previous stage. However, if the character is not found throughout the string, the results from all shifter and comparator block will be of logic 0 and hence the ultimate result will be logic 0 indicating that character is not found in the string. The advantage of this implementation is that irrespective of the length of the string, the comparison and displaying position is performed in 2 clock cycles.

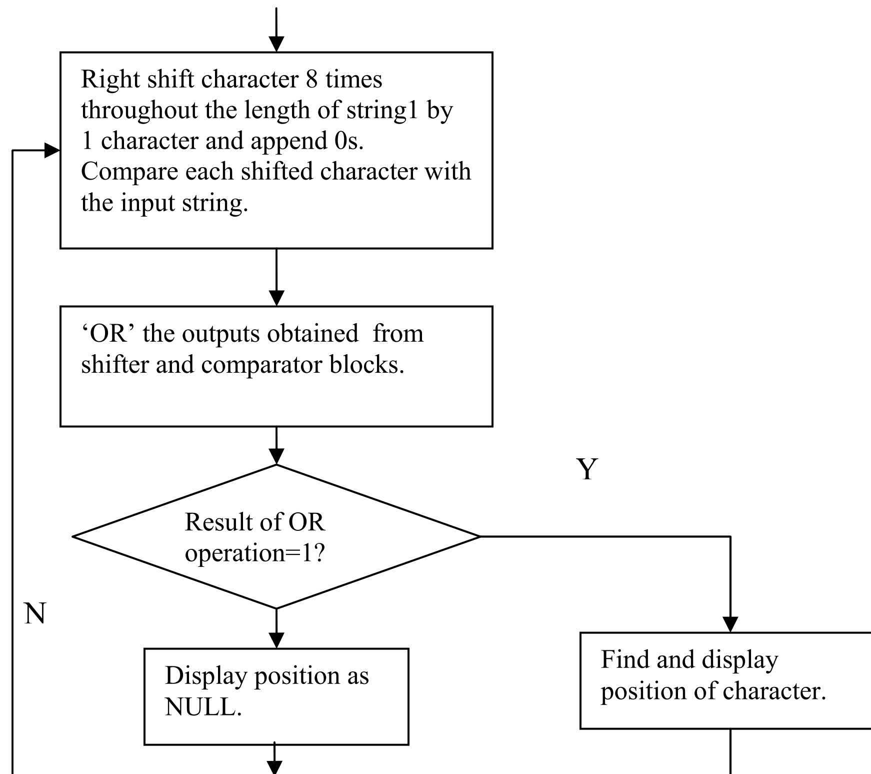

Figure 3.11: ASM Chart Of '**`strchr_pos`**' Function In **`Verilog`**

6) **`strrchr:`**

`Input: String1, char1`

`Output: Last occurrence of char1 in String1.`

This function searches for the last occurrence of a particular character, **`char1`** in a given string, **`string1`** and displays the position of **`char1`** if it is found in **`string1`**.For a string of length '*n*', this function is implemented with the help of '*n*' shifter and comparator blocks. Each shifter and comparator block right shifts a character by one position and perform a logical X-OR operation with the original string. These '*n*' outputs are ORed together to give final result of search operation. For an OR logic when

either of the input signals is logic 1 the result is logic 1.Therefore, whenever there is a match between the character in a string, one of the eight outputs will be high and the output of OR logic will be high indicating the occurrence of character in the given string. The position is determined by checking for logic 1 output of previous stage. In this case, the outputs of the previous stage are checked from last shifter and comparator block which allows to find the last occurrence of character in the string, However, if the character is not found throughout the string, the results from all shifter and comparator block will be 0 and hence the ultimate result will be 0 indicating that character is not found in the string and the position will be displayed as NULL. The advantage of this implementation is that irrespective of the length of the string, the comparison and displaying position is performed in 2 clock cycles.

For example is string1 is ‘`ABCDC`’ and char1 is ‘`C`’, strrchr function will output the position of char1 as 4 but strchr function will output the position as 2.

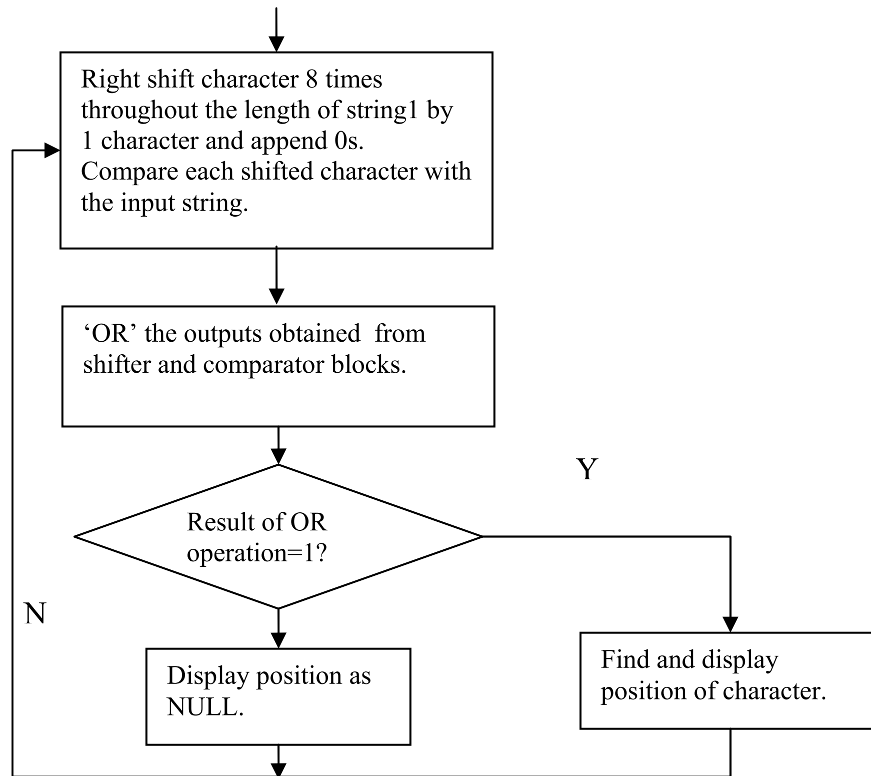

Figure 3.12: ASM Chart Of '**`strrchr`**' Function In **`Verilog`**

7) **`strupr:`**

`Input: String1`

`Output: String1 converted to uppercase`

This function converts all the characters in a given string **`string1`** to uppercase. It reads a character every clock cycle and checks if it is in upper case or lower case. The lower case letters from 'a' to 'z' fall in the range of ASCII equivalent 97 to 122 and upper case letters from 'A' to 'Z' fall in the range of ASCII equivalent 65 to 90. The only difference between a particular character's lowercase and uppercase is its 6th bit. All the lowercase letters have this bit as 1 and upper case letters have this bit 0. Hence to check

and change the case of a letter 6th bit is checked. If it is 1 it is converted to 0 to change the case to upper otherwise it is not modified. This process continues until the end of **string1** is reached.

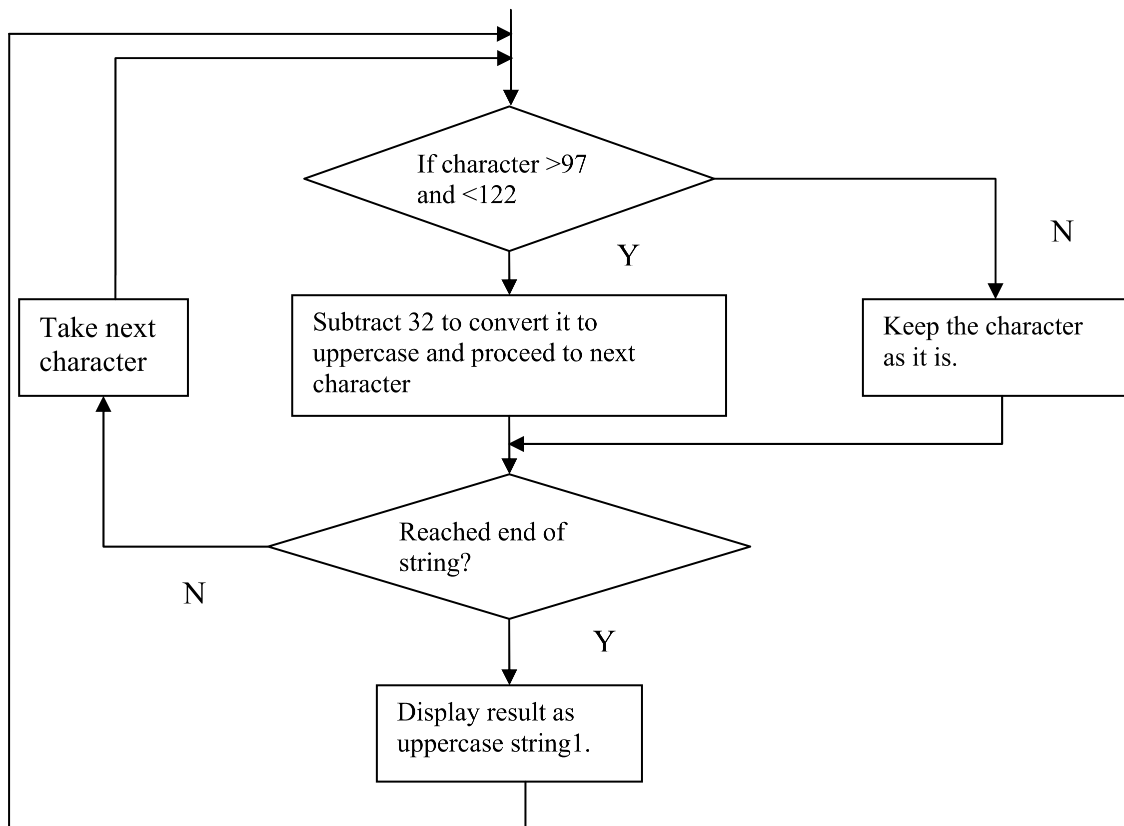

Figure 3.13: ASM Chart Of '**strupr**' Function In **Verilog**

8) **strlwr:**

Input: String1

Output: String1 converted to lowercase

This function converts all the characters in a string to lowercase. The characters which are already in lowercase are kept untouched. The upper case characters from A to Z fall in the range of ASCII equivalent 65 to 91. Hence every time the character is checked if it falls in this range, and it is converted to the lowercase by adding 32 to the original string.

This function converts all the characters in a given string **`string1`** to lower case. It reads a character every clock cycle and checks if it is in upper case or lower. The lower case letters from ‘a’ to ‘z’ fall in the range of ASCII equivalent 97 to 122 and upper case letters from ‘A’ to ‘Z’ fall in the range of ASCII equivalent 65 to 90. The only difference between a particular character’s lowercase and uppercase is its 6th bit. All the lowercase letters have this bit as 1 and upper case letters have this bit 0. Hence to check and change the case of a letter 6th bit is checked. If it is 0 it is converted to 1 to change the case to lower otherwise it is not modified. This process continues until the end of string is reached.

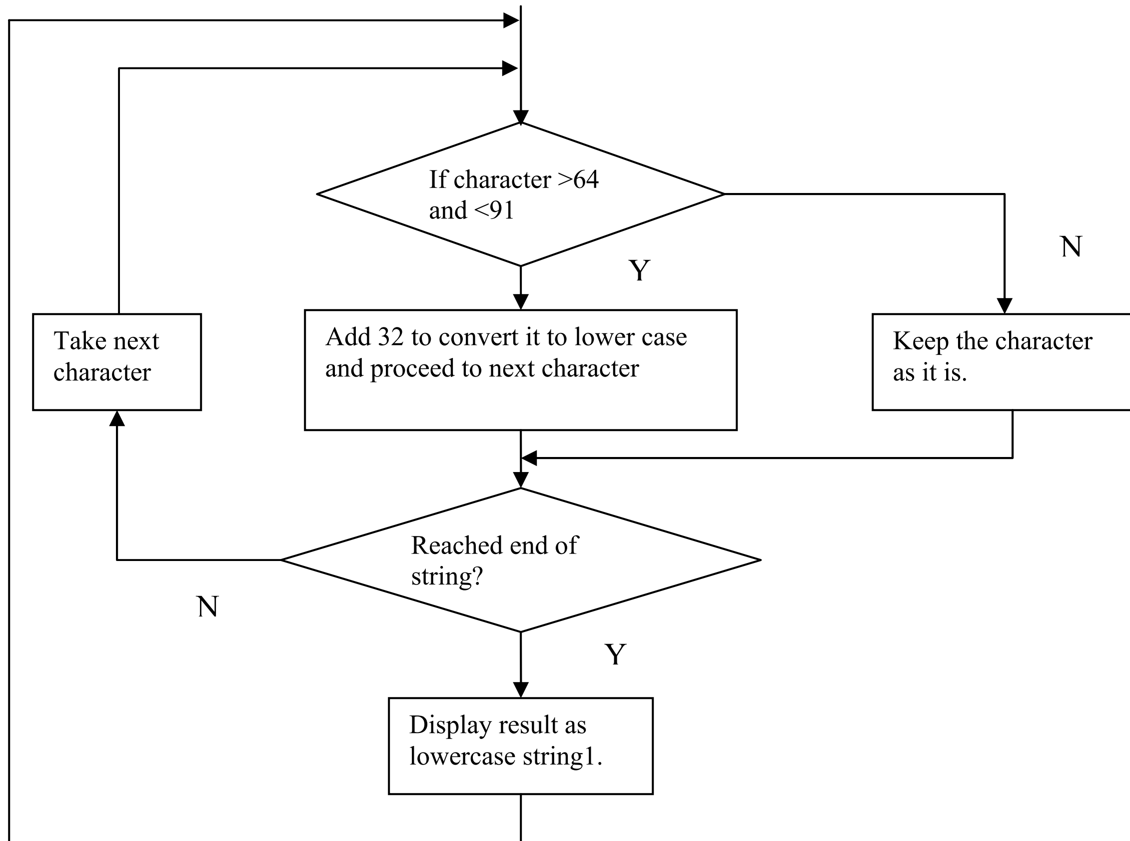

Figure 3.14: ASM Chart Of '**`strlwr`**' Function In **`Verilog`**

9) **`strlen:`**

`Input: String1`

`Output: Length of string1`

This function calculates the length of a given string, **`string1`**.On every clock cycle a character is read from the **`string1`** and a counter is incremented. This character is appended in a new string, **`string3`**.This **`string3`** is ORed with **`string1`** and the result is compared with **`string3`**.If its equal length is displayed as the counter value. If the result is not same as string3, next character from **`string1`** is appended to **`string3`**

and same procedure is followed until the end of string is reached. The counter value which is the actually the length of string1 is displayed when the end of **`string1`** is reached.

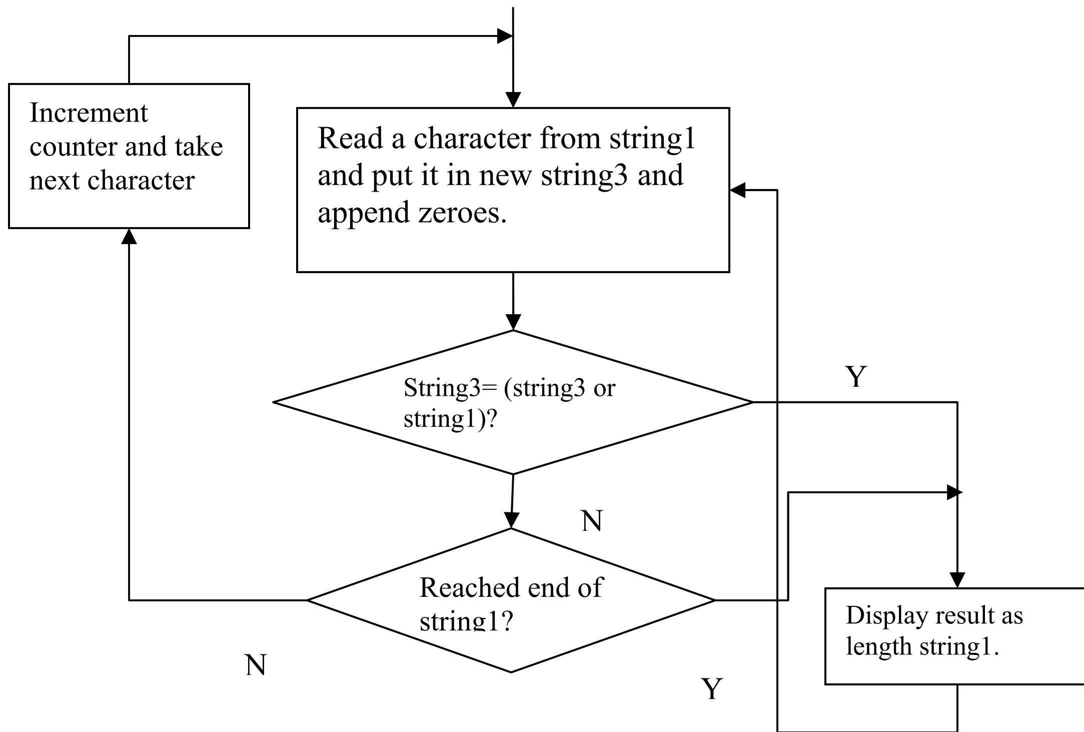

Figure 3.15: ASM Chart Of '**`strlen`**' Function In **`Verilog`**

3.7 Basic Setup

The following block diagram shows the prototype system we developed. The **`Verilog`** modules that are synthesized and used to configure the FPGA on the DE2 board consists of 3 different blocks. The actual string comparison block is coupled with

the sender and receiver blocks to allow communication with the computer through the RS-232 serial cable.

A program written in **C** runs on the computer and provides the strings or characters, depending on the operation required, to the board through serial cable and waits to receive the results. This data is sent in a specified format in order to distinguish between the two strings. The start of each string or character is characterized by sending **STX** and similarly end of string is characterized by sending **ETX**. Hence if we need to send two strings, **ABCDE** and **AB** then they are sent as **STX ABCDE ETX STX AB ETX.**

These strings are received by the receiver and formatting block both of which are implemented in the FPGA.This block converts the serial data into a parallel word through the implementation of a serial input register in the FPGA and allows the strings or character to be available for performing string operations. The string comparison block can perform any of the functions mentioned above. Once the result is ready the data is sent to the transmitter block also implemented on the FPGA which transmits data serially over the RS-232 cable to the computer. Once the data is transmitted, it is received by the C program running on the computer and the result is displayed on the console.

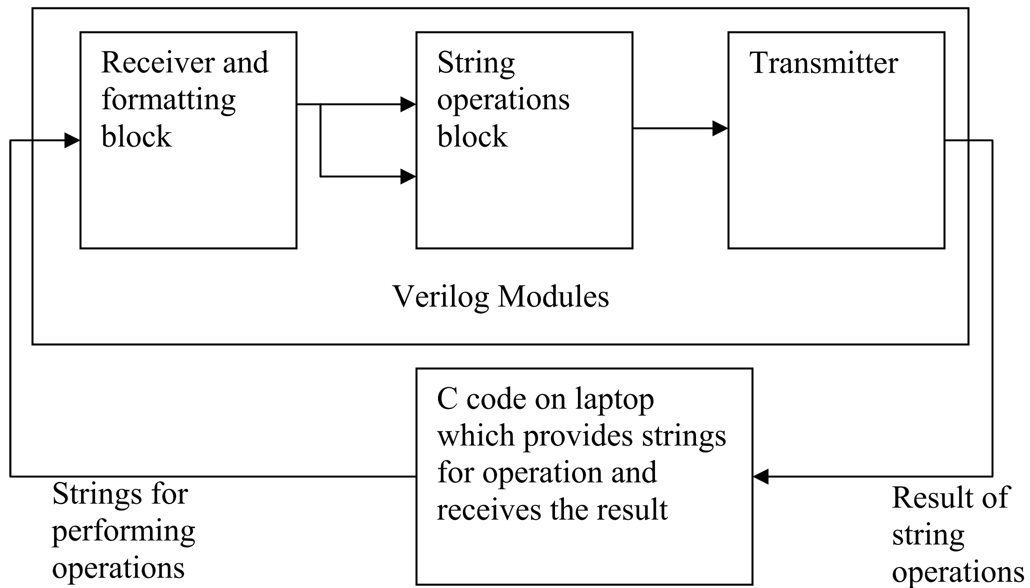

Figure 3.16: Overall Implementation Methodology

3.8 Timing Measurements

Various timing measurements are performed and computed in order to evaluate the effectiveness of hardware acceleration of string operations. This section describes these measurements.

3.8.1 Timing Measurement Of The Software String Functions

We used some of the C Library functions which return various time measures to determine computation time for our test cases. There are low as well as high resolution timers available to provide timing measurements. The Functions **`time()`** and **`clock()`** can be used together to calculate this parameter. This function returns the elapsed CPU time since the execution of the program commenced. This value is the total time of the

entire process. The function returns value in units of CPU clock cycles. A constant named CLOCKS_PER_SEC is defined as a data type of `float` in the C language header file **`time.h`** that contains the number of CPU clock cycles per second. To determine number of CPU seconds elapsed, the result is calculated by calling **`clock()`** and dividing the result by CLOCKS_PER_SEC. This generally gives the result in milliseconds, which is insufficient for our purposes since we are dealing with timing resolutions that are several orders of magnitude smaller. For this reason, we utilized the high resolution timer function described in [32].

The high resolution performance timer is called **`QueryPerformanceCounter()`**. This function measures the CPU time from the point where the execution of program starts. The returned parameter is divided by number of clock to calculate the actual CPU runtime. The number of CPU clock cycles is obtained by the function **`QueryPerformanceFrequency()`**. This calculated CPU runtime is the total time of the program including system overhead time.

3.8.2 Timing Measurement Of The Hardware String Functions Using Oscilloscope

To measure the function when it is implemented in hardware, for our case on FPGA, we added a new output signals to the FPGA circuit that outputs two pulses. One pulse transitions from logic-0 to logic-1 when the string operations are started and another pulse transitions from logic-0 to logic-1 when the string operations are completed. The delta time between these two pulses is measured using an oscilloscope.

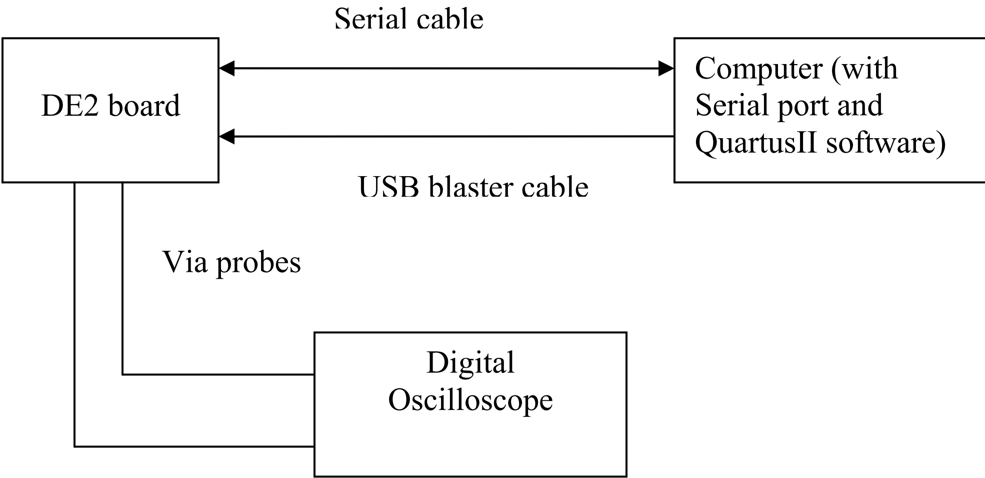

Figure 3.17: Timing Measurements On Oscilloscope

Chapter 4

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental Results

This chapter provides the results and the interpretation of the suggested hardware accelerated string functions. We compare these results to the amount of runtime required when the **`C`** library functions are used. In addition, theoretical calculations and circuit simulations using QuartusII tool are also included. The following tables contain the results obtained for the time measurements for different different string functions. Each string function is exercised with different data and corresponding timing values are reported.

Software Runtime Results:

These results are obtained by computing an average of 10,000 runs on a Windows machine using the **`Queryperformancetimer()`** function. The computer used to obtain these results is a Sony laptop with an Intel Pentium III processor containing 512 MB of RAM running the Windows XP Professional operating system at 1.2GHz. The detailed software program of the timer is included in Appendix B.

Theoretical Calculations:

The theoretical time for the actual string operations when implemented in hardware is listed in terms of number of clock cycles.

Two strings used for performing string operations are transmitted as **`STX string1 ETX STX string2 ETX`**. The time required to receive one byte utilizing the EIA-232C standard in 8N1 format at a baud rate of 115200, is given by:

$$Computatio\ ntime = (\text{Number of data bits} + \text{start bit} + \text{stop bit}) * \text{Time for transm ission of single bit}$$
$$= 10 * 8.68 \text{ us}$$
$$= 86.80\text{us}$$

Hence, the time to receive two strings of length '*m*' and '*n*' is:

$$4 * 86.80\text{us} + \text{m} * 86.60\text{us} + \text{n} * 86.60\text{us}$$

The transmission time to transmit the result:

10 x 8.68us =86.80 us.

Sample calculation:

String1: A

String2: A

Time to receive two strings = 4*86.80us + *1**86.60us+ *1**86.60us =520.8 us

Time to transmit result =10 x 8.68us =86.80 us.

Hence the total time for this implementation methodology is T_s+607.6us, where T_s is the time required for string operations. This T_s is mentioned in terms of number of clock cycles in the result table.

Simulation Results:

The string functions written in **Verilog** are compiled and simulated on the Cyclone FPGA present on the Altera DE2 board using the QuartusII timing simulator. The screenshots of the simulation results of **strcmp** function can be found in appendix as a sample. Some of these results are provided as screen captures in Appendix E.

Oscilloscope Measurements:

The FPGA board provides two 40-pin expansion headers, GPIO_A and GPIO_B .Each expansion header provides DC +5V, DC +3.3V and two ground pins, along with the pins for data transfer which are directly associated with the FPGA chip. Two signals GPIO_0[0] and GPIO_0[1] from FPGA are connected to the two channels of oscilloscope (Agilient Infiniium Oscilloscope,1.5GHz) with the help of probes. The ground of these probes is connected to the ground pin provided by the header. FPGA chip outputs pulses at the start and the end of string operations. The time difference between these two pulses is observed and measured on the oscilloscope to find out the actual time required for string operations on FPGA board.

`Strcmp` function:

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
A	A	1.060	1	20.21	20.25
B	A	1.067	1	19.68	20.34
ABCDEFGH	ABCDEFGH	1.136	1	20.03	20.6423
ABCDEFGH	AABBCEDF	1.134	1	20.15	20.6984
AB	ABCDE	1.119	1	20.19	20.6496
ABCD	ABCD	1.179	1	20	20.6635

`Strcasecmp` function:

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
A	A	1.061	1	21.36	20.963
A	a	1.058	1	18.71	20.9275
ABCDEFG	ABcDEFg	1.122	8	161.2	159.96
ABCDEFG	AABBCED	1.130	2	39.57	40.86
Abcde	AB	1.119	3	62.3	60.96
Abcd	abcd	1.179	4	80.34	80.123

`strupr` function:

String1	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
AbcD	1.131	4	80	80.16
A	1.121	1	19.88	20.192
Abcd	1.171	4	80.56	80.16
ABcDe	1.378	5	100.08	100.18
Ab	1.219	2	39.87	40.16

`strlwr` function:

String1	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
AbcD	1.15	4	80.23	80.23
A	1.129	1	20.07	20.07
Abcd	1.356	4	80.90	80.90
ABcDe	1.449	5	100.13	100.13
AB	1.121	2	39.95	39.95

`strlen` function:

String1	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
A	1.973	1	19.3	19.489
Abcd	1.150	4	79.44	79.5979
ABcDe	1.494	5	99.76	99.42
AB	1.122	2	42.1	39.4019
ABCDEFGH	1.774	8	158.99	159.42

`Strchr` function:

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
ABCD	A	1.141	2	40.41	39.805
ABCD	a	1.173	2	40.30	39.8314
ABCD	C	1.119	2	40.01	39.8415
ABCDB	B	1.163	2	40.28	39.8411
ABCDEF	D	1.119	2	40.41	39.8313
AbCd	d	1.170	2	40.42	39.8646

`Strchr` function (returns position):

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
ABCD	A	1.141	2	40.20	40.2509
ABCD	a	1.173	2	40.202	40.2846
ABCD	C	1.119	2	40.46	40.2202
ABCDB	B	1.163	2	39.87	40.2763
ABCDEF	D	1.119	2	40.147	40.2846
AbCd	d	1.170	2	40.468	40.3013

`Strstr` function (returns position):

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
ABCD	A	1.120	2	40.20	40.313
ABCD	a	1.128	5	100.354	100.13
ABCD	CD	1.447	4	79.752	81.3013
ABCD	BC	1.079	3	59.646	61.16
ABCDEF	DE	1.121	5	99.9597	101.13
AbCd	CD	1.355	4	100.242	100.18

`Strrchr` function:

String1	String2	Software Runtime Results (us)	Theoretical Results (No. of clock cycles)	Simulation Results (ns)	Oscilloscope Results (ns)
ABCD	A	1.123	2	41.896	40.2520
ABCDB	B	1.177	2	41.818	40.2668
ABCDCC	C	1.126	2	41.52	40.2622
ABCDCC	F	1.176	2	41.91	40.2944
AbCd	B	1.184	2	41.88	40.2453

4.2 Analysis

The theoretical results are the ideal expected results from the suggested implemented technology. These results are stated in terms of number of clock cycles required to obtain result from a particular string operations block. The Altera DE2 board has a 50 MHz clock cycle i.e. a clock period of 20 ns; hence all the results are roughly multiples of 20 ns.

The **`Verilog`** descriptions of the string operations are synthesized to the netlist and are mapped onto the actual device using the QuartusII tool. The simulated results are very close to the theoretical results stated earlier. These results differ in the range of 0.01 ns to 0.50 ns.

The measurements performed on the actual hardware give the results which validate the simulated results. A configured FPGA while performing string operations experiences delays from the realized circuitry such as gates, registers multiplexers etc. which affect the output by a small amount. These delays include clock to Q output delay of a register, and pure pin to pin combinational delay, and register to register delay which is given by addition of clock to Q delay ,longest path delay from Q output to next register input and setup time of register.

An important point to be noted here is that, the FPGA runs at a clock frequency of only 50 MHz whereas, the computer on which the string functions are executed in software, runs at a frequency of the order of GHz. So if we shift to an FPGA operating at a higher frequency we will definitely achieve a tremendous speed boost. In order to find the maximum frequency of the design, the designs were mapped to a fast device and QuartusII Timing Analyzer tool was run. The maximum frequency of various string

operations was found to be in the range of 220MHz - 270MHz, which proves that these **`Verilog`** descriptions are not limited by the clock frequency of 50MHz provided on the board.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

It can be observed from the results given in the chapter 4 that the theoretical results for string operations match almost exactly with the simulated as well actual experimental results, obtained from the oscilloscope. It can also be seen that the string functions yield the results ranging from few nanoseconds to few hundreds of nanoseconds. These results are minuscule compared to the results obtained from the timer in software.

Thus, the FPGA implementation of string functions considerably improved the computation time. The overall average reduction in computation time is observed to be close to 98%. This clearly supports the hypothesis behind this research that "The computation time required for string operations using hardware assisted acceleration should be less than that required on any software".

However, in this implementation technology the speedup from hardware acceleration is offset by the slow communication link to the hardware acceleration platform. The communication standard used in this experimental setup is EIA-232C which supports the maximum possible baud rate of 115200 bps, which is very slow.

Therefore, for any particular string application implemented in our prototype system, the majority of the time is spent in communication and the actual time required for the string operation is very small. Hence there is significant amount of scope for improving the communication time.

5.2 Future Work

This section describes how hardware accelerated string functions can be improved. Building a dedicated interface between the hardware acceleration circuitry and the computer will allow our approach to become feasible. This dedicated interface would ensure minimum communication time and hence faster overall computation of string operations. There are various ways to achieve this and some suggested approaches are listed below:

Recent Press Release [26] “AMD’s Opteron Processor-Based Systems having XD1000 FPGA coprocessor module make use of Altera® Stratix II EP2S180 FPGA and a HyperTransport bus, to increase the processing performance” is a perfect example of enhancing our prototype implementation.

A further step is to make this kind of FPGA dynamically configurable as per application. Defining an Application Programmer’s Interface (API) library which will be a first interface between the user and the hardware. This will also involve fixing the set of conventions to call these newly implemented functions.

APPENDIX A

C CODE FOR SERIALCOMMUNICATION [31]

```
#include <stdio.h>

#include <windows.h>


HANDLE handle1;


void ComPortClose()

{

 CloseHandle(handle1);

}


void ComPortOpen()

{

 COMMTIMEOUTS TimeOuts;

 DCB dcb;


handle1 = CreateFile("COM1:",GENERIC_READ | GENERIC_WRITE,

0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
if(handle1==INVALID_HANDLE_VALUE) { printf("\n error2 \n
"); exit(1); }
 if(!SetupComm(handle1, 4096, 4096)) { printf("\n error3 \n
");exit(1); }
 if(!GetCommState(handle1, &dcb)) { printf("\n error4 \n
"); exit(1); }
 dcb.BaudRate = 115200;
 ((DWORD*)(&dcb))[2] = 0x1001;  // Com port setting no
flow-control
 dcb.ByteSize = 8; // 8 data bits
 dcb.Parity = NOPARITY; //no parity
 dcb.StopBits = ONESTOPBIT;//1 stop bit
 if(!SetCommState(handle1, &dcb)) { printf("\n error5 \n
"); exit(1); }

 TimeOuts.ReadIntervalTimeout = MAXDWORD;
 TimeOuts.ReadTotalTimeoutMultiplier = 0;
 TimeOuts.ReadTotalTimeoutConstant = 0;
 TimeOuts.WriteTotalTimeoutMultiplier = 0;
 TimeOuts.WriteTotalTimeoutConstant = 0;
 if(!SetCommTimeouts(handle1, &TimeOuts)) { printf("\n
error6 \n "); exit(1);}
 }
```

```
DWORD WriteCom(char* Buffer, int Length)
{
 DWORD status;
 if(!WriteFile(handle1, Buffer, Length, &status, NULL))
exit(1);
 return status;
}

void WriteComChar(char byte)
{
 WriteCom(&byte, 1);
}

int ReadCom(char *Buffer, int Length)
{
 DWORD nRec;
 if(!ReadFile(handle1, Buffer, Length, &nRec, NULL))
exit(1);

 return (int)nRec;
    // printf("\n Reading from com port");
}
```

```
char ReadComChar()
{
 DWORD nRec;
 char c;
 if(!ReadFile(handle1, &c, 1, &nRec, NULL)) exit(1);


 return nRec ? c : 0;
   //  printf("\n Reading character from com port");
}
void main()
{
 char c, s[256],z,str1[256],str2[256];
 int len,tt,i,l1,l2,d;
printf("*************************START
*****************************");
printf("\n\n INTERPRETATION OF RESULTS ");
printf("\n\n RESULT 0 : string/character not found ");
printf("\n\n RESULT 1 : string/character found ");
printf("\n\n RESULT > 0   : length of string ");
printf("\n\n\n Please enter the string1 ");
gets(str1);printf("\n Please enter length of string1 ");
scanf("%d",&l1);
fflush(stdin);printf("\n Please enter the string2 ");
```

```
gets(str2);printf("\n Please enter the length of string2
");
scanf("%d",&l2);
ComPortOpen();
printf("\n Opened COM port\n ");
WriteComChar(0x02); WriteCom(str1,l1); WriteComChar(0x03);
printf("\n Transmitted string1 %s",str1);
WriteComChar(0x02); WriteCom(str2,l2); WriteComChar(0x03);
printf("\n Transmitted string2 %s",str2);
sleep(3); i=25;
printf("\n The result of operation is as follows \n RESULT
: " );
while(i>0)
{
c=ReadComChar(); printf("%c",c); i--;
}
ComPortClose(); printf("\n\n Closed COM port\n ");
printf("\n\n\n ********************* END
********************* ");
}
```

APPENDIX B

C CODE FOR TIME MEASUREMENT [32]

```
#include <stdio.h>

#include <windows.h>

#include <string.h>


void startTimer(Timer *t);

void stopTimer(Timer *t);

double Convert(LARGE_INTEGER * L);

double getElapsedTime(Timer *timer);


typedef struct {

    LARGE_INTEGER start_timer;

    LARGE_INTEGER stop_timer;

} Timer;


void startTimer(Timer *t) {

QueryPerformanceCounter(&t->start_timer);

}
```

```
void stopTimer(Timer *t) {
QueryPerformanceCounter(&t->stop_timer);
}

double Convert(LARGE_INTEGER * L) {
LARGE_INTEGER frequency;
QueryPerformanceFrequency(&frequency);
return ((double)L->QuadPart /(double)frequency.QuadPart);
}

double getElapsedTime(Timer *t) {
LARGE_INTEGER time;
time.QuadPart = t->stop_timer.QuadPart - t-
>start_timer.QuadPart;
return Convert(&time) ;
}

int main(void) {
    Timer tm;
     int i,j, len;
    long k=1;
    double difference, sum1=0, sum2=0;
     char *result;
```

```
     char *s1 = "Abcdefgh";
     char *s2= "CD";
    char s3[]="AB";
    printf("\n*************You are in the main
program***********\n");

     for(j=0;j<10;j++) {
          for(i=0;i<1000;i++) {
              difference=0;
               startTimer(&tm);
          //   len=strcmp(s1,s2);
               result=strupr(s3);
          //   result=strlwr(s3);
               len=strlen(s1);
          //   len=strrstr(s1,s2);
          //   result=strrchr(s1,'B');
          //   len=strrstr(s1,s2);
          //   result=strchr(s1,'d');
          //   result=strstr(s1,"A");

               stopTimer(&tm);
               difference=getElapsedTime(&tm);
               sum1=sum1+difference;
```

```
          }
          sum1=sum1/1000;
          sum2=sum2+sum1;
     }
     sum2=sum2/10;
    printf("\nTime required = %0.9lf",sum2);
    printf("\n");
//   printf("The result is %d",result-s1+1);
getch();
}
```

APPENDIX C

VERILOG CODE FOR SERIAL COMMUNICATION [22]

Sender module:

```
module my_sender
(clk,reset,transmit_data,transmit_data_en,p1_i,p2_i,p3_i,p4
_i,fifo_write_space,sop_data,p1_o,p2_o,p3_o,p4_o);
//parameters
parameter BAUD_COUNTER_WIDTH  = 9;
parameter BAUD_TICK_INCREMENT = 9'd1;
parameter BAUD_TICK_COUNT          = 9'd433;
parameter HALF_BAUD_TICK_COUNT     = 9'd216;
parameter TOTAL_DATA_WIDTH         = 10;//11;
parameter DATA_WIDTH               = 8;//9;

//Inputs
input clk,p1_i,p2_i,p3_i,p4_i;
input reset;
input [DATA_WIDTH:1]transmit_data;
input transmit_data_en;
```

```
//outputs
output reg[7:0]fifo_write_space;
output reg sop_data;
reg serial_data_out;
output reg p1_o,p2_o,p3_o,p4_o;

// Internal Wires
wire shift_data_reg_en;
wire all_bits_transmitted;
wire read_fifo_en;
wire fifo_is_empty;
wire fifo_is_full;
wire [6:0]     fifo_used;
wire [DATA_WIDTH:1] data_from_fifo;

// Internal Registers
reg transmitting_data;
reg [DATA_WIDTH:0] data_out_shift_reg;

always @(posedge clk)
begin
     if (reset == 1'b1)
          fifo_write_space <= 8'h00;
```

```
else
          fifo_write_space <= 8'h80 - {fifo_is_full,
fifo_used};
end

always @(posedge clk)
begin
     if (reset == 1'b1)
          serial_data_out <= 1'b1;
     else
          serial_data_out <= data_out_shift_reg[0];
end

always @(posedge clk)
begin
     if (reset == 1'b1)
          transmitting_data <= 1'b0;
     else if (all_bits_transmitted == 1'b1)
          begin transmitting_data <= 1'b0;  end
//output_ready=1;end PAK
     else if (fifo_is_empty == 1'b0)
          transmitting_data <= 1'b1;
```

```
end


always @(posedge clk)

begin

     if (reset == 1'b1)

          data_out_shift_reg  <= {(DATA_WIDTH + 1){1'b1}};

     else if (read_fifo_en)

          data_out_shift_reg  <= {data_from_fifo, 1'b0};

     else if (shift_data_reg_en)

          data_out_shift_reg  <=

               {1'b1, data_out_shift_reg[DATA_WIDTH:1]};

end


always @(posedge clk)

begin

if(all_bits_transmitted ==1)

begin p1_o=0; p2_o=0;p3_o=0; p4_o=0; end

else

begin p1_o=p1_i; p2_o=p2_i;p3_o=p3_i; p4_o=p4_i;

end

end
```

```
assign read_fifo_en = ~transmitting_data & ~fifo_is_empty &
~all_bits_transmitted;


//instantiation of other modules
Altera_UP_RS232_Counters_s RS232_Out_Counters_s (
	// Inputs
	.clk					(clk),
	.reset					(reset),


	.reset_counters
	(~transmitting_data),


	// Outputs
	.baud_clock_rising_edge		(shift_data_reg_en),
	.baud_clock_falling_edge (),
	.all_bits_transmitted		(all_bits_transmitted)
);
defparam
	RS232_Out_Counters_s.BAUD_COUNTER_WIDTH =
BAUD_COUNTER_WIDTH,
	RS232_Out_Counters_s.BAUD_TICK_INCREMENT	=
BAUD_TICK_INCREMENT,
```

```
	RS232_Out_Counters_s.BAUD_TICK_COUNT		=
BAUD_TICK_COUNT,
	RS232_Out_Counters_s.HALF_BAUD_TICK_COUNT	=
HALF_BAUD_TICK_COUNT,
	RS232_Out_Counters_s.TOTAL_DATA_WIDTH		=
TOTAL_DATA_WIDTH;


Altera_UP_SYNC_FIFO_s RS232_Out_FIFO_s (
	// Inputs
	.clk		(clk),
	.reset			(reset),
	.write_en	(transmit_data_en & ~fifo_is_full),
	.write_data		(transmit_data),
	.read_en	(read_fifo_en),


	// Outputs
	.fifo_is_empty	(fifo_is_empty),
	.fifo_is_full	(fifo_is_full),
	.words_used		(fifo_used),
	.read_data		(data_from_fifo)
);
defparam
	RS232_Out_FIFO_s.DATA_WIDTH	= DATA_WIDTH,
```

```
     RS232_Out_FIFO_s.DATA_DEPTH   = 128,
     RS232_Out_FIFO_s.ADDR_WIDTH   = 7;

endmodule
```

Bit Counter module:

```
module Altera_UP_RS232_Counters_s
(clk,reset,reset_counters,
     baud_clock_rising_edge,baud_clock_falling_edge,all_bit
s_transmitted);
parameter BAUD_COUNTER_WIDTH  = 9;
parameter BAUD_TICK_INCREMENT = 9'd1;
parameter BAUD_TICK_COUNT          = 9'd433;
parameter HALF_BAUD_TICK_COUNT     = 9'd216;
parameter TOTAL_DATA_WIDTH         = 11;

// Inputs
input clk,reset,reset_counters;

// Outputs
output reg     baud_clock_rising_edge;
```

```
output reg     baud_clock_falling_edge;
output reg all_bits_transmitted;

// Internal Registers
reg            [(BAUD_COUNTER_WIDTH - 1):0]  baud_counter;
reg            [3:0]     bit_counter;

always @(posedge clk)
begin
     if (reset == 1'b1)
          baud_counter <= {BAUD_COUNTER_WIDTH{1'b0}};
     else if (reset_counters)
          baud_counter <= {BAUD_COUNTER_WIDTH{1'b0}};
     else if (baud_counter == BAUD_TICK_COUNT)
          baud_counter <= {BAUD_COUNTER_WIDTH{1'b0}};
     else
          baud_counter <= baud_counter +
BAUD_TICK_INCREMENT;
end

always @(posedge clk)
begin
     if (reset == 1'b1)
```

```
            baud_clock_rising_edge <= 1'b0;
      else if (baud_counter == BAUD_TICK_COUNT)
            baud_clock_rising_edge <= 1'b1;
      else
            baud_clock_rising_edge <= 1'b0;
end

always @(posedge clk)
begin
      if (reset == 1'b1)
            baud_clock_falling_edge <= 1'b0;
      else if (baud_counter == HALF_BAUD_TICK_COUNT)
            baud_clock_falling_edge <= 1'b1;
      else
            baud_clock_falling_edge <= 1'b0;
end

always @(posedge clk)
begin
      if (reset == 1'b1)
            bit_counter <= 4'h0;
      else if (reset_counters)
            bit_counter <= 4'h0;
```

```
		else if (bit_counter == TOTAL_DATA_WIDTH)
			bit_counter <= 4'h0;
		else if (baud_counter == BAUD_TICK_COUNT)
			bit_counter <= bit_counter + 4'h1;
end

always @(posedge clk)
begin
	if (reset == 1'b1)
		all_bits_transmitted <= 1'b0;
	else if (bit_counter == TOTAL_DATA_WIDTH)
		all_bits_transmitted <= 1'b1;
	else
		all_bits_transmitted <= 1'b0;
end

endmodule



module Altera_UP_SYNC_FIFO_s
(clk,reset,write_en,write_data,read_en,
	fifo_is_empty,fifo_is_full,words_used,read_data);
parameter DATA_WIDTH     = 32;
```

```
parameter DATA_DEPTH     = 128;
parameter ADDR_WIDTH     = 7;

// Inputs
input clk, reset,write_en,read_en;
input [DATA_WIDTH:1]write_data;

// Outputs
output fifo_is_empty,    fifo_is_full;
output [ADDR_WIDTH:1] words_used;
output [DATA_WIDTH:1] read_data;

//lpm module instantiation
scfifo    Sync_FIFO (
     // Inputs
     .clock              (clk),
     .sclr               (reset),

     .data               (write_data),
     .wrreq              (write_en),

     .rdreq              (read_en),
```

```
    // Outputs
    .empty              (fifo_is_empty),
    .full               (fifo_is_full),
    .usedw              (words_used),
    .q                  (read_data)
    ,
    .aclr               (),
    .almost_empty  (),
    .almost_full   ()
);
defparam
    Sync_FIFO.add_ram_output_register  = "OFF",
    Sync_FIFO.intended_device_family   = "Cyclone II",
    Sync_FIFO.lpm_numwords             = DATA_DEPTH,
    Sync_FIFO.lpm_showahead            = "ON",
    Sync_FIFO.lpm_type                 = "scfifo",
    Sync_FIFO.lpm_width                = DATA_WIDTH,
    Sync_FIFO.lpm_widthu               = ADDR_WIDTH,
    Sync_FIFO.overflow_checking        = "OFF",
    Sync_FIFO.underflow_checking       = "OFF",
    Sync_FIFO.use_eab                  = "ON";

always @(posedge clk)
```

```
begin
case(tr_en)
0:sop_out=1;
1:sop_out=serial_data_out & 1;
default:sop_out=1;
endcase
end
endmodule
```

Receiver module:

```
module my_rx
(clk,reset,serial_data_in,receive_data_en,fifo_read_availab
le,output_ready,received_data,hex0,hex1,p1,p2);
parameter BAUD_COUNTER_WIDTH  = 9;
parameter BAUD_TICK_INCREMENT = 9'd1;
parameter BAUD_TICK_COUNT          = 9'd433;
parameter HALF_BAUD_TICK_COUNT     = 9'd216;
parameter TOTAL_DATA_WIDTH         = 10;//11;
parameter DATA_WIDTH               = 8;//9;
// Inputs
input clk,reset,serial_data_in,receive_data_en;
//outputs
output reg     [7:0] fifo_read_available;
```

```
output [(DATA_WIDTH - 1):0]   received_data;
output output_ready;
output reg  [6:0]hex0;
output reg [6:0]hex1;
output reg p2;
output wire p1;
//registers
reg [3:0] chek0;
reg [3:0] chek1;
reg receiving_data;
reg [(TOTAL_DATA_WIDTH - 1):0]     data_in_shift_reg;
reg output_ready;
reg [7:0] display;
assign p1=receive_data_en;
//  Wires
wire
shift_data_reg_en,all_bits_received,fifo_is_empty,fifo_is_f
ull;
wire [6:0]     fifo_used;
always @(posedge clk)
begin
     if (reset == 1'b1)
          fifo_read_available <= 8'h00;
```

```
     else
          fifo_read_available <= {fifo_is_full, fifo_used};
end

always @(posedge clk)
begin
     if (reset == 1'b1)
     begin     receiving_data <= 1'b0; end
     else if (all_bits_received == 1'b1)
          receiving_data <= 1'b0;
     else if (serial_data_in == 1'b0)
          begin receiving_data <= 1'b1; p2=1; end
end
always @(posedge clk)
begin
     if (reset == 1'b1)
          data_in_shift_reg   <= {TOTAL_DATA_WIDTH{1'b0}};
     else if (shift_data_reg_en)
          data_in_shift_reg   <=
               {serial_data_in,
data_in_shift_reg[(TOTAL_DATA_WIDTH - 1):1]};
end
Altera_UP_RS232_Counters_r RS232_In_Counters_r (
```

```
	// Inputs
	.clk				(clk),
	.reset					(reset),
	.reset_counters				(~receiving_data),
	// Outputs
	.baud_clock_rising_edge		(),
	.baud_clock_falling_edge (shift_data_reg_en),
	.all_bits_transmitted		(all_bits_received)
);
defparam
	RS232_In_Counters_r.BAUD_COUNTER_WIDTH  =
BAUD_COUNTER_WIDTH,
	RS232_In_Counters_r.BAUD_TICK_INCREMENT =
BAUD_TICK_INCREMENT,
	RS232_In_Counters_r.BAUD_TICK_COUNT		=
BAUD_TICK_COUNT,
	RS232_In_Counters_r.HALF_BAUD_TICK_COUNT	=
HALF_BAUD_TICK_COUNT,
	RS232_In_Counters_r.TOTAL_DATA_WIDTH		=
TOTAL_DATA_WIDTH;

Altera_UP_SYNC_FIFO_r RS232_In_FIFO_r (
	// Inputs
```

```
     .clk           (clk),
     .reset              (reset),


     .write_en      (all_bits_received & ~fifo_is_full),
     .write_data         (data_in_shift_reg[(DATA_WIDTH +
1):1]),
     .read_en       (receive_data_en & ~fifo_is_empty),
     // Outputs
     .fifo_is_empty (fifo_is_empty),
     .fifo_is_full  (fifo_is_full),
     .words_used         (fifo_used),
     .read_data          (received_data)
);
defparam
     RS232_In_FIFO_r.DATA_WIDTH    = DATA_WIDTH,
     RS232_In_FIFO_r.DATA_DEPTH    = 128,
     RS232_In_FIFO_r.ADDR_WIDTH    = 7;


always @(receive_data_en)
begin


if(received_data!=0)
```

```
begin display=received_data;  chek0=display[3:0];

chek1=display[7:4]; end

else

begin display=0; chek0=display[3:0];  chek1=display[7:4];

end


//if(chek!=0)

end



always @(posedge clk)

begin

case(chek0)

  4'h1: hex0 = 7'b100100100;

  4'h3: hex0 = 7'b0110000;

  4'h4: hex0 = 7'b0011001;

  4'h5: hex0 = 7'b0010010;

  4'h6: hex0 = 7'b0000010;

  4'h7: hex0 = 7'b1111000;

  4'h8: hex0 = 7'b0000000;

  4'h9: hex0 = 7'b0011000;

  4'ha: hex0 = 7'b0001000;

  4'hb: hex0 = 7'b0000011;
```

```
  4'hc: hex0 = 7'b1000110;

  4'hd: hex0 = 7'b0100001;

  4'he: hex0 = 7'b0000110;

  4'hf: hex0 = 7'b0001110;

  4'h0: hex0 = 7'b1000000; //displays 0

default : hex0 = 7'b1000000;

  endcase


end





always @(posedge clk)

begin

case(chek1)

  4'h1: hex1 = 7'b100100100;

  4'h3: hex1 = 7'b0110000;

  4'h4: hex1 = 7'b0011001;

  4'h5: hex1 = 7'b0010010;

  4'h6: hex1 = 7'b0000010;

  4'h7: hex1 = 7'b1111000;

  4'h8: hex1 = 7'b0000000;

  4'h9: hex1 = 7'b0011000;
```

```
  4'ha: hex1 = 7'b0001000;

  4'hb: hex1 = 7'b0000011;

  4'hc: hex1 = 7'b1000110;

  4'hd: hex1 = 7'b0100001;

  4'he: hex1 = 7'b0000110;

  4'hf: hex1 = 7'b0001110;

  4'h0: hex1 = 7'b1000000; //displays 0

default : hex1 = 7'b1000000;

  endcase


end




always @(received_data)

begin

if(received_data==0)

output_ready=0;

else

output_ready=1;

end

endmodule
```

Buffer module:

```
module buff (
clk,reset,in_data,enable,p1_i,p2_i,p3_i,p4_i,out_data,stop,
hex4,hex5,p1_o,p2_o,p3_o,p4_o);
input clk,reset,p1_i,p2_i,p3_i,p4_i;
input [7:0] in_data;
input enable;
output reg [7:0] out_data;
output reg stop;
reg [1:0] ctr;
reg [3:0] lsb;
reg [3:0] msb;
output reg [6:0]hex4;
output reg [6:0]hex5;
output wire p1_o;
output wire p2_o;
output wire p3_o;
output wire p4_o;
assign p1_o=p1_i;
assign p2_o=p2_i;
assign p3_o=p3_i;
assign p4_o=p4_i;
     always @(posedge clk)
```

```
    begin
    if(reset==0)
    begin
    if(in_data!=0 && enable==1)
    begin
    out_data=in_data;
    lsb=out_data[3:0];
    msb=out_data[7:4];
    case(ctr)
    0:   stop=1;//0
    1: stop=0; //1
    default:stop=0; //1
    endcase
    ctr=ctr+1;
    if(ctr==0)
    ctr=1;
    end
    end
    end

always @(posedge clk)
begin
case(lsb)
```

```
  4'h1: hex4 = 7'b1111001;

  4'h2: hex4=7'b0100100;

  4'h3: hex4 = 7'b0110000;

  4'h4: hex4 = 7'b0011001;

  4'h5: hex4 = 7'b0010010;

  4'h6: hex4 = 7'b0000010;

  4'h7: hex4 = 7'b1111000;

  4'h8: hex4 = 7'b0000000;

  4'h9: hex4 = 7'b0011000;

  4'ha: hex4 = 7'b0001000;

  4'hb: hex4 = 7'b0000011;

  4'hc: hex4 = 7'b1000110;

  4'hd: hex4 = 7'b0100001;

  4'he: hex4 = 7'b0000110;

  4'hf: hex4 = 7'b0001110;

  4'h0: hex4 = 7'b1000000; //displays 0

default : hex4 = 7'b1000000;

  endcase

end


always @(posedge clk)

begin

case(msb)
```

```
  4'h1: hex5 = 7'b1111001;

  4'h2: hex5=7'b0100100;

  4'h3: hex5 = 7'b0110000;

  4'h4: hex5 = 7'b0011001;

  4'h5: hex5 = 7'b0010010;

  4'h6: hex5 = 7'b0000010;

  4'h7: hex5 = 7'b1111000;

  4'h8: hex5 = 7'b0000000;

  4'h9: hex5 = 7'b0011000;

  4'ha: hex5 = 7'b0001000;

  4'hb: hex5 = 7'b0000011;

  4'hc: hex5 = 7'b1000110;

  4'hd: hex5 = 7'b0100001;

  4'he: hex5 = 7'b0000110;

  4'hf: hex5 = 7'b0001110;

  4'h0: hex5 = 7'b1000000; //displays 0

default : hex5 = 7'b1000000;

  endcase

end

endmodule
```

```
//This code takes the input received on RXD, removes
//delimiters and outputs 2 strings for operations.
```

Formatting Module:

```
module proc
(clk,reset,start,str_in,p1_i,p2_i,str11_o,str22,str_ready,p
1_o,p2_o,p3,num,pulse);
//inputs
input clk,p1_i,p2_i,start,reset;
input [7:0] str_in;
//outputs
output reg [63:0] str11_o;
output reg [63:0] str22 ;
output wire p1_o,p2_o;
output reg p3; output reg [3:0] num; output reg pulse;
output reg str_ready;
parameter stx=2;
parameter etx=3;
reg st1,see;
reg [63:0] str1;
reg [63:0] str11;
reg [4:0]counter;
reg go,other;
assign p1_o=p1_i;
```

```
assign p2_o=p2_i;

always @(posedge clk)

begin

if(reset==0 && start==1 && str_in!=0)

begin

case(counter)

0: begin see=1; end//stx

1: begin if (str_in != etx && str_in !=0 && str_in!=stx )

str1[63:56]=str_in;

   else begin str1=0; go=1; end  end

2: begin if (str_in != etx && str_in !=0 && str_in!=stx )

str1[55:48]=str_in;

     else begin str1[55:0]=0;go=1;  end  end

3: begin if (str_in != etx && str_in !=0 && str_in!=stx )

str1[47:40]=str_in;

   else begin str1[47:0]=0; go=1; end  end

4: begin if (str_in != etx && str_in !=0 && str_in!=stx )

str1[39:32]=str_in;

   else begin str1[39:0]=0; go=1;  end end

5: begin if (str_in != etx && str_in !=0 && str_in!=stx )

str1[31:24]=str_in;

   else begin str1[31:0]=0; go=1;   end end
```

```
6: begin if (str_in != etx && str_in !=0 && str_in!=stx )
str1[23:16]=str_in;
   else begin str1[23:0]=0; go=1; end  end
7: begin if (str_in != etx && str_in !=0 && str_in!=stx )
str1[15:8]=str_in;
   else begin str1[15:0]=0; go=1;  end  end
8: begin if (str_in != etx && str_in !=0 && str_in!=stx )
str1[7:0]=str_in;
   else begin str1[7:0]=0;go=1;   end end
9: begin if(other==0)begin str11=str1; other=1;str1=0; end
else begin str22=str1; str11_o=str11;other=0;
str1=0;str_ready=1;p3=1;end end //etx
10: begin counter=10;end
default: begin str11=0;
str22=0;go=0;counter=0;other=0;str1=0;str_ready=0;p3=0;end
endcase //inner case
counter=counter+1;
if(go==1) begin if(other==0) begin str11=str1;
other=1;go=0;str1=0;counter=0;end else begin if(other==1)
begin  str22=str1;
str11_o=str11;other=0;str_ready=1;go=0;str1=0;counter=0;p3=
1; end end end
if(counter==10)
```

```
counter=0;

end

if(reset==1) begin str_ready=0; str11_o=0; str22=0; end

end //always end

always@(posedge clk) begin

if(str_in==etx) begin num=num+1;

if(num!=0 && (num%2)==0) pulse=1;

else pulse=0;

end end

endmodule
```

APPENDIX D

VERILOG CODES FOR STRING OPERATIONS

Strcmp Function:

```
module
str_cmp(clk,reset,in1,in2,result,cmp_start,p1_i,p2_i,p3_i,s
t_tr,hex6,hex7,p1_o,p2_o,p3_o,p4);
//inputs
input [63:0] in1;
input [63:0] in2;
input clk,reset,cmp_start,p1_i,p2_i,p3_i;
//outputs
output [7:0]result;
output st_tr; //start transmission
output [6:0]hex6;
output [6:0]hex7;
output reg p4;
output wire p1_o,p2_o,p3_o;
//registers
reg st_tr;
```

```
reg [63:0] middle;
reg [7:0]result;
reg [3:0] lsb;
reg [3:0] msb;
reg [6:0]hex6;
reg [6:0]hex7;
assign p1_o=p1_i;
assign p2_o=p2_i;
assign p3_o=p3_i;
always @(posedge clk)
begin
if(reset==0 && cmp_start==1)
begin
     middle=in1 ^ in2;
      if (middle==0)
      begin result=49; st_tr=1; lsb[3:0]=result[3:0];
     msb[3:0]=result[7:4]; p4=1;end //result=1; ascii 1
     //result=49;
      else
     begin result=48;  st_tr=1;lsb[3:0]=result[3:0];
     msb[3:0]=result[7:4];p4=1;end//result=0; ascii 0
     //result=48;
end
```

```
else
begin
result=0;st_tr=0;lsb[3:0]=9; msb[3:0]=9;p4=0;
end
end

always @(posedge clk)
begin
case(lsb)
  4'h1: hex6 = 7'b1111001;
  4'h2: hex6=7'b0100100;
  4'h3: hex6 = 7'b0110000;
  4'h4: hex6 = 7'b0011001;
  4'h5: hex6 = 7'b0010010;
  4'h6: hex6 = 7'b0000010;
  4'h7: hex6 = 7'b1111000;
  4'h8: hex6 = 7'b0000000;
  4'h9: hex6 = 7'b0011000;
  4'ha: hex6 = 7'b0001000;
  4'hb: hex6 = 7'b0000011;
  4'hc: hex6 = 7'b1000110;
  4'hd: hex6 = 7'b0100001;
  4'he: hex6 = 7'b0000110;
```

```
  4'hf: hex6 = 7'b0001110;
  4'h0: hex6 = 7'b1000000; //displays 0
  default : hex6 = 7'b1000000;
  endcase
end

always @(posedge clk)
begin
case(msb)
  4'h1: hex7 = 7'b1111001;
  4'h2: hex7=7'b0100100;
  4'h3: hex7 = 7'b0110000;
  4'h4: hex7 = 7'b0011001;
  4'h5: hex7 = 7'b0010010;
  4'h6: hex7 = 7'b0000010;
  4'h7: hex7 = 7'b1111000;
  4'h8: hex7 = 7'b0000000;
  4'h9: hex7 = 7'b0011000;
  4'ha: hex7 = 7'b0001000;
  4'hb: hex7 = 7'b0000011;
  4'hc: hex7 = 7'b1000110;
  4'hd: hex7 = 7'b0100001;
  4'he: hex7 = 7'b0000110;
```

```
  4'hf: hex7 = 7'b0001110;
  4'h0: hex7 = 7'b1000000; //displays 0
  default : hex7 = 7'b1000000;
  endcase
end
endmodule
```

strcasecmp Function:

```
module
IC(clk,reset,str1,str2,st_ic,p1_i,p2_i,p3_i,result,done,p1_
o,p2_o,p3_o,hex6,hex7);
input reset,clk,st_ic,p1_i,p2_i,p3_i;
output [7:0] result;
input [63:0] str1;
input [63:0] str2;
reg [7:0] result;
reg [63:0] str_and;
reg [63:0] str_xor;
reg [7:0] counter;
reg [63:0] str22;
reg [3:0] msb;
output reg [6:0]hex6;
reg [3:0] lsb;
```

```
output reg [6:0]hex7;

output reg done;

output wire p1_o,p2_o,p3_o;

assign p1_o=p1_i;assign p2_o=p2_i;assign p3_o=p3_i;

always @(posedge clk)

begin

if(reset==1)

begin  str_and=0; str_xor=0; counter=0; str22=str2;

done=0; end

else begin

if(st_ic==1) begin

str_xor=str1 ^ str2;

str_and=str1 & str2;

if(str_xor==0 && str_and==str1 && str1!=0 && str2!=0)

begin result=49;  done=1;lsb[3:0]=result[3:0];end//string

found  result=1;

else

begin

counter=counter+1;

if(str1==0 && str2==0)

begin  result=48;counter=10;  done=0;

lsb[3:0]=result[3:0];end //result=0;

case (counter)
```

```
1:  begin
if(str2[63:56]==32 || (str2[63:56]==str1[63:56])) begin
str22[63:56]=str1[63:56];  end
if(str2[63:56]!=str1[63:56]) begin
if(str2[63:56]>96 && str2[63:56]<123 && str2[63:56]!=32)
     begin     str22[63:56]=str2[63:56]-32;  end
if(str2[63:56]>64 && str2[63:56]<91 && str2[63:56]!=32)
     begin     str22[63:56]=str2[63:56]+32; end
if(str22[63:56]!=str1[63:56]) begin  result=48; counter=10;
done=1; lsb[3:0]=result[3:0];end  else //begin
if(str2[55:0]==0) begin result=49; done=1; end end end
begin if(str2[55:0]==0 && str1[55:0]!=0) begin result=48;
done=1; lsb[3:0]=result[3:0];end if(str2[55:0]==0 &&
str1[55:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end
end
2:   begin  if(str2[55:48]==32 ||
(str2[55:48]==str1[55:48])) begin
str22[55:48]=str1[55:48];  end
if(str2[55:48]!=str1[55:48]) begin
if(str2[55:48]>96 && str2[55:48]<123 && str2[55:48]!=32)
     begin     str22[55:48]=str2[55:48]-32;  end
if(str2[55:48]>64 && str2[55:48]<91 && str2[55:48]!=32)
```

```
     begin     str22[55:48]=str2[55:48]+32; end
if(str22[55:48]!=str1[55:48]) begin  result=48; counter=10;
done=1; lsb[3:0]=result[3:0];end  else //begin
if(str2[47:0]==0) begin result=49; done=1; end end end
begin if(str2[47:0]==0 && str1[47:0]!=0) begin result=48;
done=1;lsb[3:0]=result[3:0]; end if(str2[47:0]==0 &&
str1[47:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end  end
3:   begin      if(str2[47:40]==32 ||
(str2[47:40]==str1[47:40])) begin
str22[47:40]=str1[47:40];  end
if(str2[47:40]!=str1[47:40]) begin
if(str2[47:40]>96 && str2[47:40]<123 && str2[47:40]!=32)
begin     str22[47:40]=str2[47:40]-32;  end
if(str2[47:40]>64 && str2[47:40]<91 && str2[47:40]!=32)
begin     str22[47:40]=str2[47:40]+32; end
if(str22[47:40]!=str1[47:40])
begin  result=48; counter=10;  done=1;
lsb[3:0]=result[3:0];end else
begin if(str2[39:0]==0 && str1[39:0]!=0) begin result=48;
done=1; lsb[3:0]=result[3:0];end if(str2[39:0]==0 &&
str1[39:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end end
```

```
4:  begin if(str2[39:32]==32 || (str2[39:32]==str1[39:32]))
begin  str22[39:32]=str1[39:32];  end
if(str2[39:32]!=str1[39:32])begin
if(str2[39:32]>96 && str2[39:32]<123 && str2[39:32]!=32)
str22[39:32]=str2[39:32]-32;
if(str2[39:32]>64 && str2[39:32]<91 && str2[39:32]!=32)
str22[39:32]=str2[39:32]+32;
if(str22[39:32]!=str1[39:32])
begin  result=48; counter=10;  done=1;
lsb[3:0]=result[3:0];end
else
begin if(str2[31:0]==0 && str1[31:0]!=0) begin result=48;
done=1; lsb[3:0]=result[3:0];end if(str2[31:0]==0 &&
str1[31:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end  end
5:  begin if(str2[31:24]==32 || (str2[31:24]==str1[31:24]))
begin  str22[31:24]=str1[31:24];  end
if(str2[31:24]!=str1[31:24])begin
if(str2[31:24]>96 && str2[31:24]<123 && str2[31:24]!=32)
str22[31:24]=str2[31:24]-32;
if(str2[31:24]>64 && str2[31:24]<91 && str2[31:24]!=32)
str22[31:24]=str2[31:24]+32;
```

```
if(str22[31:24]!=str1[31:24]) begin  result=48; counter=10;
done=1;lsb[3:0]=result[3:0];  end
else begin if(str2[23:0]==0 && str1[23:0]!=0) begin
result=48; done=1; lsb[3:0]=result[3:0];end
if(str2[23:0]==0 && str1[23:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end end
6: begin
if(str2[23:16]==32 || (str2[23:16]==str1[23:16])) begin
str22[23:16]=str1[23:16]; end
if(str2[23:16]!=str1[23:16])
begin
if(str2[23:16]>96 && str2[23:16]<123 && str2[23:16]!=32)
str22[23:16]=str2[23:16]-32;
if(str2[23:16]>64 && str2[23:16]<91 && str2[23:16]!=32)
str22[23:16]=str2[23:16]+32;
if(str22[23:16]!=str1[23:16])
begin  result=48; counter=10;   done=1; end else
begin if(str2[15:0]==0 && str1[15:0]!=0) begin result=48;
done=1; lsb[3:0]=result[3:0];end if(str2[15:0]==0 &&
str1[15:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end  end
7: begin
```

```
if(str2[15:8]==32 || (str2[15:8]==str1[15:8])) begin
str22[15:8]=str1[15:8]; end
if(str2[15:8]!=str1[15:8])
begin
if(str2[15:8]>96 && str2[15:8]<123 && str2[15:8]!=32)
str22[15:8]=str2[15:8]-32;
if(str2[15:8]>64 && str2[15:8]<91 && str2[15:8]!=32)
str22[15:8]=str2[15:8]+32;
if(str22[15:8]!=str1[15:8])
begin  result=48; counter=10;   done=1;
lsb[3:0]=result[3:0];end else
begin if(str2[7:0]==0 && str1[7:0]!=0) begin result=48;
done=1;lsb[3:0]=result[3:0]; end if(str2[7:0]==0 &&
str1[7:0]==0)begin
result=49;done=1;lsb[3:0]=result[3:0];end end end
end
 8: begin
if(str2[7:0]==32 || (str2[7:0]==str1[7:0])) begin
str22[7:0]=str1[7:0]; end
if(str2[7:0]!=str1[7:0])
begin
if(str2[7:0]>96 && str2[7:0]<123 && str2[7:0]!=32)
str22[7:0]=str2[7:0]-32;
```

```
if(str2[7:0]>64 && str2[7:0]<91 && str2[7:0]!=32)
str22[7:0]=str2[7:0]+32;
if(str22[7:0]!=str1[7:0])
begin  result=48; counter=10; lsb[3:0]=result[3:0];
done=1; end  else begin result=49;done=1;
lsb[3:0]=result[3:0];end end    //result=0;
end
9: begin result=49; lsb[3:0]=result[3:0];  done=1; end
10:   begin str22=str2; done=1; end
11: begin done=1; counter=11;end
default: counter=11; endcase end  end else begin
done=0;counter=0; end end //else end
end //always end
always @(posedge clk)
begin
msb=0;
case(msb)
  4'h1: hex7 = 7'b1111001;
  4'h2: hex7 = 7'b0100100;
  4'h3: hex7 = 7'b0110000;
  4'h4: hex7 = 7'b0011001;
  4'h5: hex7 = 7'b0010010;
  4'h6: hex7 = 7'b0000010;
```

```
  4'h7: hex7 = 7'b1111000;
  4'h8: hex7 = 7'b0000000;
  4'h9: hex7 = 7'b0011000;
  4'ha: hex7 = 7'b0001000;
  4'hb: hex7 = 7'b0000011;
  4'hc: hex7 = 7'b1000110;
  4'hd: hex7 = 7'b0100001;
  4'he: hex7 = 7'b0000110;
  4'hf: hex7 = 7'b0001110;
  4'h0: hex7 = 7'b1000000; //displays 0
default : hex7 = 7'b1000000;
  endcase
end
always @(posedge clk)
begin
case(lsb)
  4'h1: hex6 = 7'b1111001;
  4'h2: hex6=  7'b0100100;
  4'h3: hex6 = 7'b0110000;
  4'h4: hex6 = 7'b0011001;
  4'h5: hex6 = 7'b0010010;
  4'h6: hex6 = 7'b0000010;
  4'h7: hex6 = 7'b1111000;
```

```
  4'h8: hex6 = 7'b0000000;
  4'h9: hex6 = 7'b0011000;
  4'ha: hex6 = 7'b0001000;
  4'hb: hex6 = 7'b0000011;
  4'hc: hex6 = 7'b1000110;
  4'hd: hex6 = 7'b0100001;
  4'he: hex6 = 7'b0000110;
  4'hf: hex6 = 7'b0001110;
  4'h0: hex6 = 7'b1000000; //displays 0
default : hex6 = 7'b1000000;
 endcase end
endmodule
```

Strlen Function:

```
module
length(clk,reset,str1,start_l,,p1_i,p2_i,p3_i,length,done,p
1_o,p2_o,p3_o,p4,done1);
//inputs
input reset,clk,start_l,p1_i,p2_i,p3_i;
input [63:0] str1;
//outputs
output reg done,done1;
output reg [7:0] length;
```

```
output reg p4;
output wire p1_o,p2_o,p3_o;


//registers
reg [3:0] counter;
reg [63:0] str_im;
reg [63:0] str_or;
assign p1_o= p1_i;
assign p2_o= p2_i;
assign p3_o= p3_i;


always @(posedge clk)
begin
if(reset==1)
begin
str_im=0; counter=0;length=48;done=0;done1=0; //length=0;
end
else
begin
if(start_l==1) begin
counter=counter+1;
if(str1==0)
begin length=48; counter=9 ; done=0;done1=0;end //length=0;
```

```
case (counter)
1: begin
str_im[63:56]=str1[63:56];
str_im[55:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
     begin length=49; done =1; done1 =1;counter=9;end
else
done=0;
end
2: begin
if(done!=1) begin
str_im[63:48]=str1[63:48];
str_im[47:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
begin length=50; done =1;done1 =1;counter=9; end //length 2
else
done=0;
end
else
begin
counter=9;
```

```
end
end
3: begin
if(done!=1) begin
str_im[63:40]=str1[63:40];
str_im[39:0]=0;

str_or=str_im | str1;
if (str_or==str_im)  begin length=51; done =1;done1
=1;counter=9; end  //length 3
else begin done=0; end
end
else
begin
counter=9;
end
end
4: begin
if(done!=1) begin
str_im[63:32]=str1[63:32];
str_im[31:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
```

```
begin length=52; done =1; counter=9;done1 =1; end //length
//4
else
done=0;
end
else
begin
counter=9;
end
end
5: begin
if(done!=1) begin
str_im[63:24]=str1[63:24];
str_im[23:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
begin length=53; done =1; counter=9;done1 =1; end
//length 5
else
done=0;
end
else
begin
```

```
counter=9;
end


end
6: begin
if(done!=1) begin
str_im[63:16]=str1[63:16];
str_im[15:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
begin length=54; done =1; counter=9;done1 =1; end
//length 6
else
done=0;
end
else
begin
counter=9;
end
end
7: begin
if(done!=1) begin
str_im[63:8]=str1[63:8];
```

```
str_im[7:0]=0;
str_or=str_im | str1;
if (str_or==str_im)
begin length=55; done =1;counter=9; done1 =1;end //length 7
else
done=0;
end
else
begin
counter=9;
end
end
8: begin
if(done!=1) begin
str_im[63:0]=str1[63:0];
str_or=str_im | str1;
if (str_or==str_im)
begin length=56; done =1;counter=9;done1 =1; end //length 8
else
done=0;
end
else
begin
```

```
counter=9;
end
end
default: begin counter=9;done=0; done1 =1;end
endcase
end
else //else of start_l
begin
length=48;done1 =0;//length=0;
end
end  //else of reset
if(start_l==0)begin done1=0; counter=0; done=0; end
end //always end    endmodule
```

strupr Function:

```
module
to_uppercase(clk,reset,str1,start_conv,p1_i,p2_i,p3_i,str_o
ut1,done,start_u,p1_o,p2_o,p3_o,p4);

//inputs
input reset,clk,start_conv;
input [63:0] str1;
input p1_i,p2_i,p3_i;
```

```
//outputs
output reg done;
output reg start_u,p4;
output [7:0] str_out1;
output wire p1_o,p2_o,p3_o;
//registers
reg [7:0] str_out1;
reg [3:0] counter;
assign p1_o= p1_i;
assign p2_o= p2_i;
assign p3_o= p3_i;
always @(posedge clk)
begin
if(reset==1 )
begin
counter=0;
str_out1=0;
start_u=0;
end
else
begin
if(start_conv==1)begin
counter=counter+1;
```

```
case(counter)
1:
begin
if(str1[63:56]!=0) begin
     if(str1[63:56]>96 && str1[63:56]<123 &&
     str1[63:56]!=32)
     begin
     str_out1=str1[63:56]-32;  //make it capital
     end
     else
     str_out1=str1[63:56];
     done=1;
     start_u=1;p4=1;
     end
     else
     begin
     start_u=0;p4=0;
     done=1;
     end
end
2:
begin
     if(str1[55:48]!=0)
```

```
     begin
     if(str1[55:48]>96 && str1[55:48]<123 &&
     str1[55:48]!=32)
     begin
     str_out1=str1[55:48]-32;  //make it capital
     end
     else
     str_out1=str1[55:48];
     done=1;
     start_u=1;
     p4=1;
     end
else
begin
start_u=0;
done=1;
p4=0;
end
end
3:
begin
if(str1[47:40]!=0) begin
```

```
     if(str1[47:40]>96 && str1[47:40]<123 &&
str1[47:40]!=32)
     begin
     str_out1=str1[47:40]-32;  //make it capital
     end
else
str_out1=str1[47:40];
done=1;
start_u=1;
p4=1;
end
else
begin
start_u=0;
done=1;
p4=0;
end
end
4:
begin
if(str1[39:32]!=0) begin
     if(str1[39:32]>96 && str1[39:32]<123 &&
str1[39:32]!=32)
```

```
      begin
      str_out1=str1[39:32]-32;  //make it capital
      end
      else
      str_out1=str1[39:32];
      done=1;
      start_u=1;
      p4=1;
      end
else
begin
start_u=0;
done=1;
p4=0;
end
end
5:
begin
      if(str1[31:24]!=0) begin
      if(str1[31:24]>96 && str1[31:24]<123 &&
      str1[31:24]!=32)
      begin
      str_out1=str1[31:24]-32; // 32;  //make it capital
```

```
        end
        else
        str_out1=str1[31:24];
        done=1;
        start_u=1;
        p4=1;
        end
        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
end
6:
begin
if(str1[23:6]!=0) begin
        if(str1[23:16]>96 && str1[23:16]<123 &&
        str1[23:16]!=32)
        begin
        str_out1=str1[23:16]-32;  //make it capital
        end
        else
```

```
        str_out1=str1[23:16];
        done=1;
        start_u=1;
        p4=1;
        end

        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
    end
    7:
    begin
    if(str1[15:8]!=0) begin
        if(str1[15:8]>96 && str1[15:8]<123 && str1[15:8]!=32)
        begin
        str_out1=str1[15:8]-32;  //make it capital
        end
        else
        str_out1=str1[15:8];
        done=1;
```

```
        start_u=1;
        p4=1;
        end
        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
end
8:
begin
if(str1[7:0]!=0) begin
        if(str1[7:0]>96 && str1[7:0]<123 && str1[7:0]!=32)
        begin
        str_out1=str1[7:0]-32;  //make it capital
        end
else
str_out1=str1[7:0];
done=1;
start_u=1;
p4=1;
end
```

```
else
begin
start_u=0;
done=1;p4=0;
end
end
default: begin  done=1; start_u=0;counter=9;p4=0;end
endcase
end
else
begin done=0; counter=0; end
end  //else of reset
end //always end
endmodule
```

strlwr Function:

```
module
to_lowercase(clk,reset,str1,start_conv,p1_i,p2_i,p3_i,str_o
ut1,done,start_u,p1_o,p2_o,p3_o,p4);
//inputs
input reset,clk,start_conv;
input [63:0] str1;
input p1_i,p2_i,p3_i;
```

```
//outputs
output reg [7:0] str_out1;
output reg done;
output reg start_u,p4;
output wire p1_o,p2_o,p3_o;
reg [3:0] counter;
assign p1_o= p1_i;
assign p2_o= p2_i;
assign p3_o= p3_i;
always @(posedge clk)
begin
if(reset==1 )
begin
counter=0;
str_out1=0;
start_u=0;
end
else
begin
if(start_conv==1)begin


counter=counter+1;
case(counter)
```

```
1:
begin
if(str1[63:56]!=0) begin
     if(str1[63:56]>64 && str1[63:56]<91 &&
     str1[63:56]!=32)
     begin
     str_out1=str1[63:56]+32;  //make it capital
     end
     else
     str_out1=str1[63:56];
     done=1;
     start_u=1;p4=1;
     end
     else
     begin
     start_u=0;p4=0;
     done=1;
     end
end
2:
begin
     if(str1[55:48]!=0)
     begin
```

```
        if(str1[55:48]>64 && str1[55:48]<91 &&
        str1[55:48]!=32)
        begin
        str_out1=str1[55:48]+32;  //make it capital
        end
        else
        str_out1=str1[55:48];
        done=1;
        start_u=1;
        p4=1;
        end
        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
        end
3:
begin
        if(str1[47:40]!=0) begin
        if(str1[47:40]>64 && str1[47:40]<91 &&
        str1[47:40]!=32)
```

```
	begin
	str_out1=str1[47:40]+32;  //make it capital
	end
	else
	str_out1=str1[47:40];
	done=1;
	start_u=1;
	p4=1;
	end
	else
	begin
	start_u=0;
	done=1;
	p4=0;
	end
end
4:
begin
if(str1[39:32]!=0) begin
	if(str1[39:32]>64 && str1[39:32]<91 &&
	str1[39:32]!=32)
	begin
	str_out1=str1[39:32]+32;  //make it capital
```

```
      end
      else
      str_out1=str1[39:32];
      done=1;
      start_u=1;
      p4=1;
      end
      else
      begin
      start_u=0;
      done=1;
      p4=0;
      end
end
5:
begin
if(str1[31:24]!=0) begin
      if(str1[31:24]>64 && str1[31:24]<91 &&
      str1[31:24]!=32)
      begin
      str_out1=str1[31:24]+32; // 32;  //make it capital
      end
      else
```

```
        str_out1=str1[31:24];
        done=1;
        start_u=1;
        p4=1;
        end
        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
    end
    6:
    begin
    if(str1[23:6]!=0) begin
        if(str1[23:16]>64 && str1[23:16]<91 &&
        str1[23:16]!=32)
        begin
        str_out1=str1[23:16]+32;  //make it capital
        end
        else
        str_out1=str1[23:16];
        done=1;
```

```
     start_u=1;
     p4=1;
     end
     else
     begin
     start_u=0;
     done=1;
     p4=0;
     end
end
7:
begin
if(str1[15:8]!=0) begin
     if(str1[15:8]>64 && str1[15:8]<91 && str1[15:8]!=32)
     begin
     str_out1=str1[15:8]+32;  //make it capital
     end
     else
     str_out1=str1[15:8];
     done=1;
     start_u=1;
     p4=1;
     end
```

```
        else
        begin
        start_u=0;
        done=1;
        p4=0;
        end
        end
8:
begin
if(str1[7:0]!=0) begin
        if(str1[7:0]>64 && str1[7:0]<91 && str1[7:0]!=32)
        begin
        str_out1=str1[7:0]+32;  //make it capital
        end
        else
        str_out1=str1[7:0];
        done=1;
        start_u=1;
        p4=1;
        end
        else
        begin
        start_u=0;
```

```
     done=1;p4=0;
     end
end
default: begin  done=1; start_u=0;counter=9;p4=0;end
endcase
end
else
begin done=0; counter=0; end
end  //else of reset
end //always end
endmodule
```

strchr Function:

```
module shift_or(clk,reset,shift,in1,in2,out1,wait1);
input clk,reset,shift;
input [63:0]in1;
input [63:0]in2;
output reg out1;
reg [63:0]im_r;
output reg wait1;
always @(posedge clk)
begin
if(reset==0 && shift==1)
```

```
begin
wait1=1;
im_r= in1 | in2;
if(im_r==in1 && in1[63:56]==in2[63:56])  out1=1; else
out1=0;
end
else
begin wait1=0; end
end
endmodule
```

strchr_pos Function:

```
module shift_or(clk,reset,shift,in1,in2,out1,wait1);
input clk,reset,shift;
input [63:0]in1;
input [63:0]in2;
output reg out1;
reg [63:0]im_r;
output reg wait1;
always @(posedge clk)
begin
if(reset==0 && shift==1)
begin
```

```
wait1=1;
im_r= in1 | in2;
if(im_r==in1 && in1[63:56]==in2[63:56])  out1=1; else
out1=0;
end
else
begin wait1=0; end
end
endmodule


module
comp(clk,reset,in,in1,in2,in3,in4,in5,in6,in7,wait1,res_out
,pos,ready);
input clk,reset,in,in1,in2,in3,in4,in5,in6,in7,wait1;
output reg [7:0]res_out;
output reg [7:0] pos;
output reg ready;
always @(posedge clk)
begin
if(reset==0 && wait1==1)
begin
if ( in | in1 | in2 | in3 | in4 | in5 | in6 | in7 == 1)
begin res_out=49;  ready=1; end //ascii for 1
```

```
else
begin res_out=48; ready=1; end //ascii for 0
if (in==1) pos=48;//pos=0;
else
begin
if (in1==1) pos=49;//pos=1;
else
begin
if (in2==1) pos=50;//pos=2;
else
begin
if (in3==1) pos=51;//pos=3;
else
begin
if (in4==1) pos=52;//pos=4;
else
begin
if (in5==1) pos=53;//pos=5;
else
begin
if (in6==1) pos=54;//pos=6;
else
begin
```

```
if (in7==1) pos=55;//pos=7;
else
pos=33;//pos=255; ! mark will be displayed if character is
//not found
end
end
end
end
end
end
end
end
else
begin
ready=0; //pos=255;
end
end  endmodule
```

strrchr Function:

```
module shift_or(clk,reset,shift,in1,in2,out1,wait1);
input clk,reset,shift;
input [63:0]in1;
input [63:0]in2;
```

```
output reg out1;
reg [63:0]im_r;
output reg wait1;
always @(posedge clk)
begin
if(reset==0 && shift==1)
begin
wait1=1;
im_r= in1 | in2;
if(im_r==in1 && in1[63:56]==in2[63:56])  out1=1; else
out1=0;
end
else
begin wait1=0; end
end
endmodule
module
comp(clk,reset,in,in1,in2,in3,in4,in5,in6,in7,wait1,res_out
,pos,ready);
input clk,reset,in,in1,in2,in3,in4,in5,in6,in7,wait1;
output reg [7:0]res_out;
output reg [7:0] pos;
output reg ready;
```

```
always @(posedge clk)
begin
if(reset==0 && wait1==1)
begin
if ( in | in1 | in2 | in3 | in4 | in5 | in6 | in7 == 1)
begin res_out=49;  ready=1; end //ascii for 1
else
begin res_out=48; ready=1; end //ascii for 0
if (in7==1) pos=55;//pos=7;
else
begin
if (in6==1) pos=54;//pos=6;
else
begin
if (in5==1) pos=53;//pos=5;
else
begin
if (in4==1) pos=52;//pos=4;
else
begin
if (in3==1) pos=51;//pos=3;
else
begin
```

```
if (in2==1) pos=50;//pos=2;
else
begin
if (in1==1) pos=49;//pos=1;
else
begin
if (in==1) pos=48;//pos=0;
else
pos=33;//pos=255; ! mark will be displayed if character is
not found
end
end
end
end
end
end
end
end
else
begin
ready=0;pos=255;
end
end
```

```
endmodule
```

strstr Function:

Shifting Module

```
module my_shifting(clk,reset,shift,din,shifted);
//input
input clk,reset,shift;
input [63:0] din;
//output
output [63:0] shifted;
reg [4:0] counter1;
reg [4:0] counter2;
reg [4:0] counter3;
reg [4:0] counter4;
reg [4:0] counter5;
reg [4:0] counter6;
reg [4:0] counter7;
reg [63:0] shifted;
reg dont_shift;
reg transmit;
wire [4:0] length;
assign length=2;
always @ (posedge clk) //or posedge reset)
```

```
begin
if(shift==1)
begin
//shifted=din;
if(length==1)
begin
counter1=counter1+1;
case (counter1)
1: begin shifted=din; end
2: begin shifted[63:56]=0;shifted[55:48]=din[63:56];
shifted[47:0]=0; end
3: begin shifted[63:48]=0;shifted[47:40]=din[63:56];
shifted[39:0]=0; end
4: begin shifted[63:40]=0;shifted[39:32]=din[63:56];
shifted[31:0]=0; end
5: begin
shifted[63:32]=0;shifted[31:24]=din[63:56];shifted[23:0]=0;
end
6: begin
shifted[63:24]=0;shifted[23:16]=din[63:56];shifted[15:0]=0;
end
7: begin shifted[63:16]=0;shifted[15:8]=din[63:56];
shifted[7:0]=0;end
```

```
8: begin
shifted[63:8]=0;shifted[7:0]=din[63:56];counter1=0;end
endcase
end



if(length==2)
begin
counter2=counter2 + 1;
case (counter2)
1: begin shifted[63:48]=din[63:48]; shifted[47:0]=0;end
2: begin
shifted[63:56]=0;shifted[39:0]=0;shifted[55:40]=din[63:48];
end
3: begin
shifted[63:48]=0;shifted[31:0]=0;shifted[47:32]=din[63:48];
end
4: begin
shifted[63:40]=0;shifted[23:0]=0;shifted[39:24]=din[63:48];
end
5: begin
shifted[63:32]=0;shifted[15:0]=0;shifted[31:16]=din[63:48];
end
```

```
6: begin
shifted[63:24]=0;shifted[7:0]=0;shifted[23:8]=din[63:48];
end
7: begin shifted[63:16]=0;shifted[15:0]=din[63:48];
counter2=0;end
endcase
end
if(length==3)
begin
counter3=counter3+1;
case (counter3)
1: begin shifted[63:40]=din[63:40]; end
2: begin shifted=0;shifted[55:32]=din[63:40]; end
3: begin shifted=0;shifted[47:24]=din[63:40]; end
4: begin shifted=0;shifted[39:16]=din[63:40]; end
5: begin shifted=0;shifted[31:8]=din[63:40]; end
6: begin shifted=0;shifted[23:0]=din[63:40];counter3=0; end
endcase
end
if(length==4)
begin
counter4=counter4+1;
case (counter4)
```

```
1: begin shifted[63:32]=din[63:32]; end

2: begin shifted=0;shifted[55:24]=din[63:32]; end

3: begin shifted=0;shifted[47:16]=din[63:32]; end

4: begin shifted=0;shifted[39:8]=din[63:32]; end

5: begin shifted=0;shifted[31:0]=din[63:32]; counter4=0;end

endcase

end


if(length==5)

begin

counter5=counter5+1;

case (counter5)

1: begin shifted[63:24]=din[63:24]; end

2: begin shifted=0;shifted[55:16]=din[63:24]; end

3: begin shifted=0;shifted[47:8]=din[63:24]; end

4: begin shifted=0;shifted[39:0]=din[63:24]; counter5=0;end

endcase

end


if(length==6)

begin

counter6=counter6+1;

case (counter6)
```

```
1: begin shifted[63:16]=din[63:16]; end
2: begin shifted=0;shifted[55:8]=din[63:16]; end
3: begin shifted=0;shifted[47:0]=din[63:16]; counter6=0;end
endcase
end

if(length==7)
begin
counter7=counter7+1;
case (counter7)
1: begin shifted[63:8]=din[63:8]; end
2: begin shifted=0;shifted[55:0]=din[63:8]; counter7=0;end
endcase
end
if(length==8)
begin shifted=din; end
end  //shift end
else
begin
counter1=0; counter2=0; counter3=0; counter4=0; counter5=0;
counter6=0; counter7=0;
end  end //always end
endmodule
```

Controller Module

```
module
ctrl_signals(clk,reset,op,s_cs,shift_en,cmp_en,pstate,yes,p
os,done1);
input clk,reset,op,s_cs;
output reg shift_en,cmp_en,done1;
output reg [7:0]yes;
output reg [7:0]pos;
reg [4:0] nstate;
output reg [4:0] pstate;
reg done;
//state encoding (gray coding)
parameter s0=5'b00000,s1=5'b00001,s2=5'b00010;
parameter s3=5'b00011,s4=5'b00100,s5=5'b00101;
parameter s6=5'b00110,s7=5'b00111,s8=5'b01000,s9=5'b01001;
always @ (posedge clk or posedge reset)
```

```
begin
    if (reset == 1'b1) pstate = s0;
           else
              pstate = nstate;
     end
always @ (pstate)
      begin
nstate=pstate;
case  (pstate )
//reset state
s0:
begin
if(s_cs==1) begin
shift_en=1;
cmp_en=1;
done=0;
```

```
nstate=s1;

done1=1;

//pos=pstate-2;

end else begin shift_en=0;

cmp_en=0;

done=0;done1=0;end

 end

s1:

begin

if(op==1)

begin  shift_en=0; cmp_en=0;  nstate=s9; pos=0;
done=1;done1=1;end

else begin nstate=s2; shift_en=1; cmp_en=1; end

end

s2:

begin
```

```
if(op==1)

begin shift_en=0;cmp_en=0;   nstate=s9; pos=0; done=1;
done1=1;end

else

begin nstate=s3; shift_en=1; cmp_en=1; end

end

s3:

begin

if(op==1) begin shift_en=0; cmp_en=0;  pos=1+48;
nstate=s9;  done=1;done1=1; end

else

begin nstate=s4; shift_en=1; cmp_en=1; end

end

s4:

begin

if(op==1)
```

```
begin  shift_en=0; cmp_en=0; pos=2+48;   nstate=s9;
done=1; done1=1;end
else
begin nstate=s5; shift_en=1; cmp_en=1; end
end
s5:
begin
if(op==1)
begin   shift_en=0; cmp_en=0; pos=3+48;  nstate=s9; done=1;
done1=1;end
else
begin nstate=s6; shift_en=1;cmp_en=1; end
end
s6:
begin
if(op==1)
```

```
begin    shift_en=0; cmp_en=0;  pos=4+48; nstate=s9;
done=1; done1=1;end

else

begin nstate=s7; shift_en=1; cmp_en=1; end

end

s7:

begin

if(op==1)

begin    shift_en=0; cmp_en=0;pos=5+48;   nstate=s9;
done=1; done1=1;end

else

begin

nstate=s8; shift_en=1; cmp_en=1;end

end

s8:

begin

//display results
```

```
if(op==1) begin pos=6+48;

shift_en=0; cmp_en=0;done=1;done1=1;

nstate=s9;  //s0

yes=op+48;

end

else

begin pos=33;done=1;done1=1;nstate=s9;end

end

s9:begin

done=0; shift_en=0; cmp_en=0; done1=1;end

endcase

if(s_cs==0) begin done1=0; nstate=s0; end

end  endmodule
```

Compare module:

```
module my_comp(clk,reset,in1,in2,start_comp,op,done_comp);

input clk,reset;

input [63:0] in1;
```

```
input [63:0] in2;

input start_comp;

output reg op,done_comp;

reg [63:0] im;

always @ (posedge clk) //or posedge reset)

begin

if(reset==0 && start_comp==1 )

begin

im=in1 | in2; //or

if(in1==im && in1!=0 && in2!=0) begin op=1;done_comp=1;end

 else  begin op=0;  end

end

else

begin

done_comp=0;

end

end

endmodule
```

APPENDIX E

IMAGES OF MEASUREMENTS AND RESULTS

The following section shows screen shots of results obtained from oscilloscope and simulation measurements for **`strcmp`** function. The figure E.1 shows the calculation of execution time on oscilloscope, for **`Verilog`** implemented **`strcmp`** function. The two vertical lines represent the cursor positions for two pulses outputted by FPGA board and respective delta time 20.6425ns can be seen as the computation time for **`strcmp`**

Simulation results:

Figure E.2 and E.3 show the simulation results for **`strcmp`** function. Figure E.2 shows the result when string1 = string2, and figure E.3 shows the result when string1 $\neq$ string2.

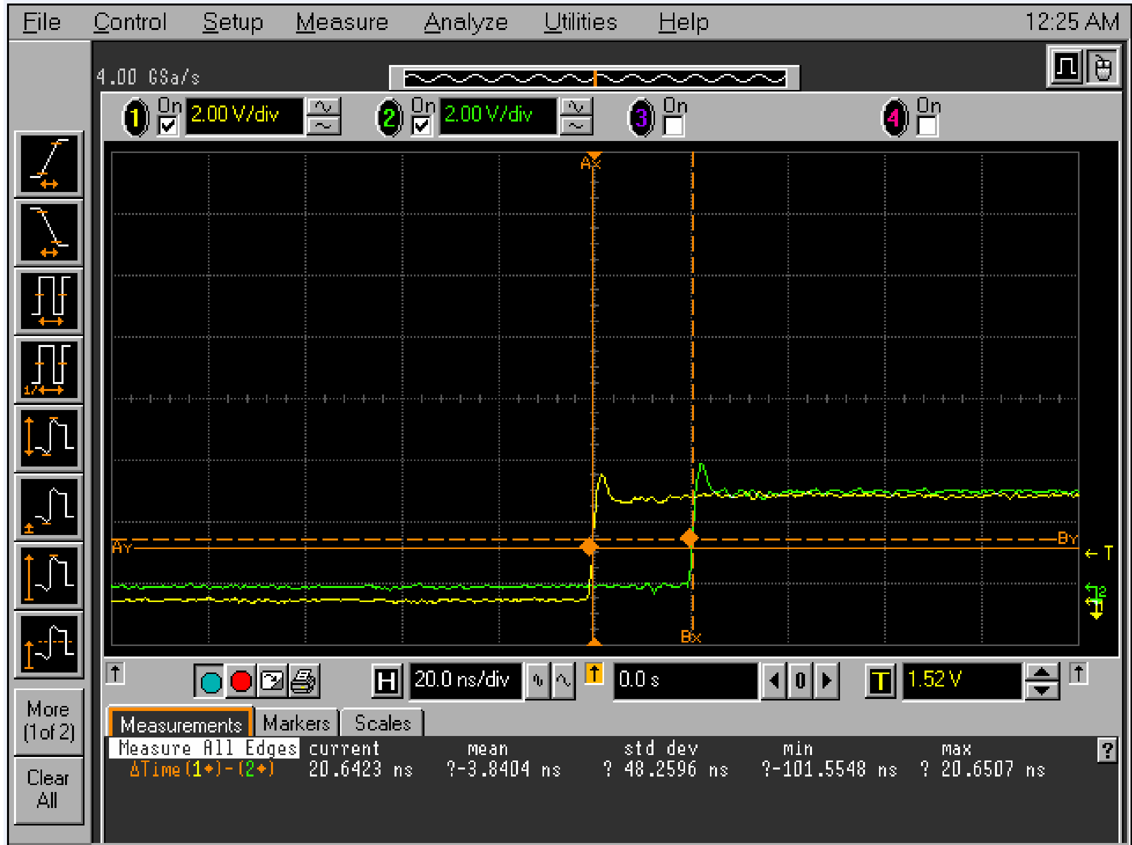

Figure E.1 Oscilloscope Screen For Measurement Of Computation Time

for **`strcmp`** function

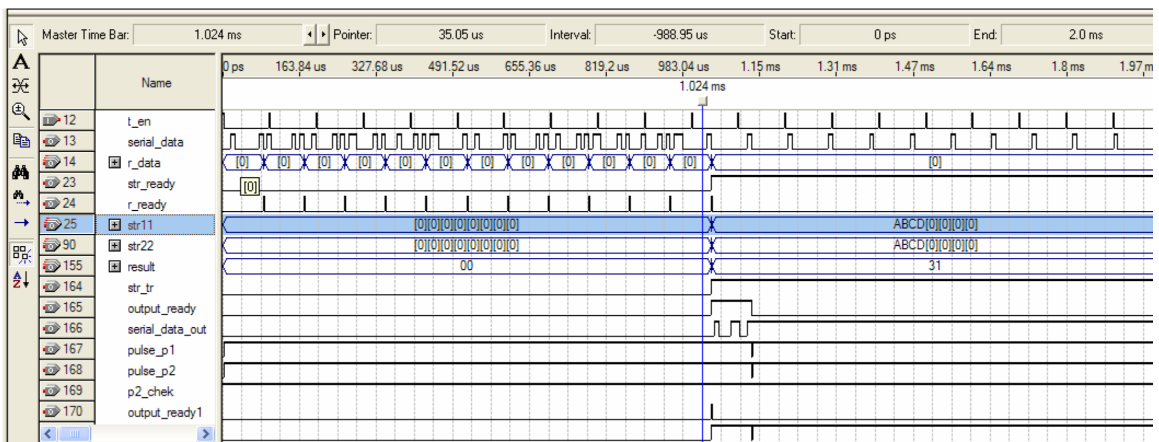

Figure E.2 Simulation Result For string1 = string2

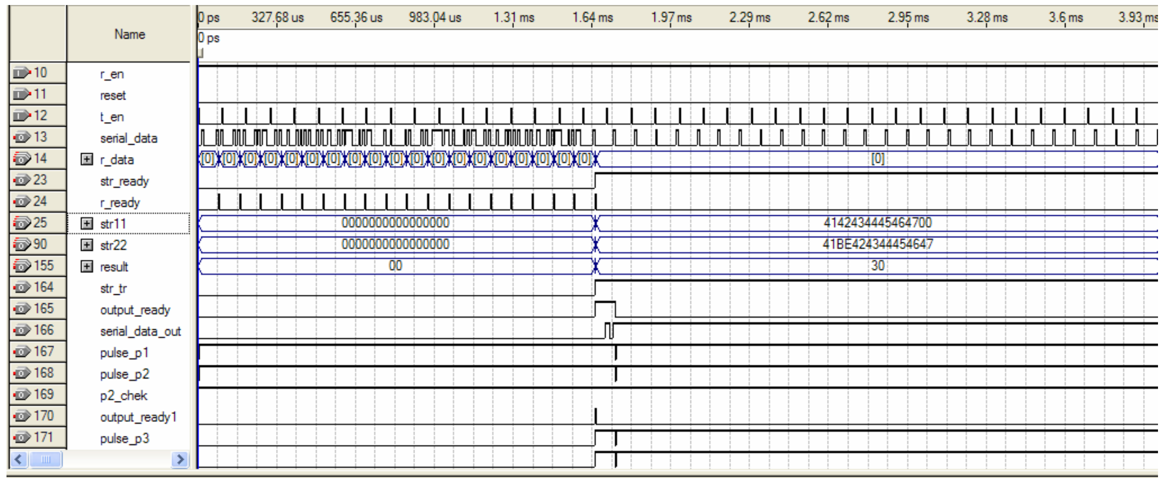

Figure E.3 Simulation Result For string1 ≠ string2

REFERENCES

[1] Hal Berghel and David Roach ,"An extension of Ukkonen's enhanced dynamic programming ASM algorithm", ACM Transactions on Information Systems (TOIS) archive, Volume 14 , Issue 1 (January 1996),Pages: 94 - 106,Year of Publication: 1996,ISSN:1046-8188

[2] Ricardo Baeza-Yates and Gaston H. Gonnet,"A new approach to text searching", Communications of the ACM archive Volume 35 , Issue 10 (October 1992),Pages: 74 - 82,Year of Publication: 1992,ISSN:0001-0782

[3] E. Ukkonen,"Finding approximate patterns in strings", J. Algorithms 6 (1985), 132-137.

[4] H. D. Cheng and K. S. Fu **, "**VLSI architectures for string matching and pattern matching" , Pattern Recognition archive ,Volume 20 , Issue 1 (1987) table of contents, Pages: 125 - 144,Year of Publication: 1987,ISSN:0031-3203.

[5] Robert A. Wagner and Michael J. Fischer, "The String-to-String Correction Problem", Journal of the ACM (JACM) archive Volume 21 , Issue 1 (January 1974),Pages: 168 - 173,Year of Publication: 1974,ISSN:0004-5411

[6] Patrick A. V. Hall and Geoff R. Dowling , "Approximate String Matching", ACM Computing Surveys (CSUR) archive, Volume 12 , Issue 4 (December 1980),Pages: 381 - 402 ,Year of Publication: 1980,ISSN:0360-0300

[7] H. Shang and T.H. Merrett ,"Tries for Approximate String Matching ",IEEE Trans. on Knowledge and Data Eng., Vol. 8, No. 4, pp. 540-547, Aug. 1996

[8] Enrique Vidal, Andres Marzal and Pablo Aibar ,"Fast Computation of Normalized Edit Distances ", IEEE Transactions on Pattern analysis and machine intelligence ,September 1995 (Vol. 17, No. 9) pp. 899-902

[9] G. M. Landau and U. Vishkin ,"Fast parallel and serial approximate string matching", Journal of Algorithms archive,Volume 10 , Issue 2 (June 1989),Pages: 157 - 169,Year of Publication: 1989,ISSN:0196-6774

[10] K. Zhang and D. Shasha,"Simple fast algorithms for the editing distance between trees and related problems", SIAM Journal on Computing archive, Volume 18, issue 6 (December 1989),Pages: 1245 - 1262,Year of Publication: 1989,ISSN:0097-5397

[11] Raghu Sastry,N. Ranganathan and Klinton Remedios ,"CASM: A VLSI Chip for Approximate String Matching", IEEE Transactions on Pattern analysis and machine intelligence ,August 1995 (Vol. 17, No. 8) pp. 824-830

[12] Richard J. Lipton and Daniel Lopresti,"A Systolic Array for Rapid String Comparison" 1985.

[13] Daniel P. Lopresti, "Rapid implementation of a genetic sequence comparator using field-programmable logic arrays", Proceedings of the 1991 University of California/Santa Cruz conference on Advanced research in VLSI,Pages: 138 - 152,Year of Publication: 1991,ISBN:0-262-19308-6

[14] Sun Wu and Udi Manber,"Fast text searching: allowing errors",Communications of the ACM archive Volume 35 , Issue 10 (October 1992),Pages: 83 - 91,Year of Publication: 1992,ISSN:0001-0782

[15] http://www.xess.com/fpgatut.htm

[16] String Handling by Dave Marshall
Available: http://www.cs.cf.ac.uk/Dave/C/node19.html

[17] String Matching- National Institute of Standards and Technology, By Paul E Black,Available: http://www.nist.gov/dads/HTML/stringMatching.html

[18] Top Coder Software- String Distance 1.0 Component Specification
Available: http://software.topcoder.com/catalog/document?id=8457494

[19] Exact String Matching Algorithms, Animations in Java by Christian Charras - Thierry Lecroq, Available:
http://wwwigm.univmlv.fr/~lecroq/string/

[20] Dr. Mitch Thornton, Southern Methodist University,
Available: http://engr.smu.edu/~mitch/

[21] QuartusII Web Edition Software, Altera Corporation.
Available: http://www.altera.com/support/software/sof-quartus.html

[22] Altera's Development and Education Board, Altera Corporation
Available: http://www.altera.com/education/univ/materials/boards/unv-de2-board.html

[23] Approximate String Matching, Wikipedia Encyclopedia
Available: http://en.wikipedia.org/wiki/Approximate_string_matching

[24] RS-232C (EIA 232 C) presentation, Available:
http://www.bridgewater.edu/~lwilliam/Chapter%2005/sld044.htm

[25] http://www.nullmodem.com/DB-9.htm

[26] Altera Press Release, Altera Corporation, Available:
http://www.altera.com/corporate/news_room/releases/releases_archive/2006/products/nr-xtremedata.html

[27] Egecioglu, O. and Ibel, M.,"Parallel algorithms for fast computation of normalized edit distance(Extended abstract)", Parallel and Distributed Processing, 1996.

[28] RS-232, Wikipedia Encyclopedia, Available: http://en.wikipedia.org/wiki/RS-232

[29] Boyer–Moore string search algorithm, Wikipedia Encyclopedia, Available:
http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm

[30] Design and Analysis of Algorithms, Knuth-Morris-Pratt string matching,Dept. Information & Computer Science -- UC Irvine, Available:
http://www.ics.uci.edu/~eppstein/161/960227.html

[31] Serial Communication in Win32,Microsoft Developer Network,
Available: http://msdn2.microsoft.com/en-us/library/ms810467.aspx

[32] High resolution timer for Windows C programs , Information Metrics Institute, Available: http://www.unb.ca/metrics/software/HRtime.html

[33] ASCII Table and description,Available: http://www.asciitable.com

[34] scfifo (Single-Clock FIFO) megafunction,
Available:http://www.pldworld.com/_altera/html/_sw/q2help/source/mega/mega_file_scfifo.htm