IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Design And Implementation Of Similarity Detector Using Hashing On FPGA For Low End Iot Devices

VANITA AGARWAL
Department of Electronics and Telecommunication
COEP Technological University, Pune, INDIA

Abstract – String matching has been implemented in the past using various algorithms in the past like Bloom Filter, Hamming Distance, Levenshtein Distance etc.. These algorithms work efficiently for few bytes of data but are of little use when it comes to detect similarity between medium sized files. To overcome this limitation, hashing algorithms are used to compress the files. Additionally we can add fuzziness i.e. tolerance for detecting similarity. In the past several Researchers have implemented Fuzzy hashing on software but there are only few hardware implementations. This paper presents the design and implementation of Fuzzy Hashing for lower end devices for detecting similarity between two files on FPGA. This work can be extended to introduce Fuzzy hashing for improved malware clusters detection in low end IoT devices.

Keywords **– Rolling Hash, field programmable gate arrays (FPGA), Fuzzy hashing, context triggered piecewise hashing (CTPH), Levenshtein Distance.**

I. INTRODUCTION

Low end IoT devices should use sensor level crypto security and/or hardware level crypto processors to prevent attacks at the edge [1]. Computer forensic community use context triggered piecewise hashing (CTPH) [2], [3], [4] to enable examiners to compare files that previously were lost in vast quantities of data but can currently be a matter of investigation.

Study of Fuzzy Hashing for detecting similarity between two text files requires Context triggered piece-wise hashing for generating signatures. This requires strong hashing algorithm like MD5, SH1 etc. Parallel implementation of Edit Distance algorithm to compare signatures requires large hardware [5], [6], [7]. In this paper the authors target to implement the scaled down MD5 from original algorithm for accuracy in similarity detection which will work on 128 bits message and will produce 32 bits hash.

The Edit Distance for similarity matching technique to detect data similar to a given pattern from the input data plays important role in Fuzzy Hashing [2]. Similarity detector can also be used for fingerprint matching, text retrieval in database, observing of DNA, protein sequences in bioinformatics etc. Algorithms for similarity detection between two files have been extensively studied to shorten its computation time. Unlike the software implementations, there are only few implementations on the hardware level. Benefits of utilizing fuzzy hashing calculations, as sdhash, mvHash, ssdeep, and mrsh v2 were discussed in distinguishing likenesses in Malware area [2]. Few FPGA based implementations of MD5 hash and approximate string matching have been discussed in the past [7], [8], [9], [10], [11].

Low end IoT devices (8 bit to 32 bit edge processors) are restricted in memory and computational resources. Standard encryption algorithms like AES, DES are not light weight to secure these devices. To enhance hardware level security of these low end IoT devices, the authors in this paper present the design and implementation of Fuzzy hashing algorithm.

II. DESIGN OF ROLLING HASH FOR SEGMENTING

Fuzzy hashing technique works on two input files and gives the result in terms of similarity or dissimilarity of two input files. This technique first of all takes the input byte by byte and divides the whole file into many different sized segments depending on the calculations used in the window of rolling hash. These segments are then passed to any traditional hashing algorithm so that each segment is represented by its unique signature. Similarly the signatures are generated for another input file and finally these signatures from two different files are compared through Edit distance algorithm (Levenshtein Distance). Rolling hash is an efficient way for segmenting the input file. According to current context in window of rolling hash segmenting of file is done. MD5 algorithm can be used for converting each segment into a fixed size unique identity or signature or hash value.

The minimum edit distance algorithm works efficiently for comparing small sized files. For comparing medium-sized files, we need a more efficient approach. This is where signature generation from rolling hash algorithm comes into play (also called as LSH) [4]. The signatures are the hash values generated by the traditional hashing algorithm. In this paper, the authors used modified MD5 hashing algorithm. The input to this MD5 hashing algorithm is the segment or slice of the input file. These segments are generated with rolling hash. This will divide the complete input file into segments, the length of each segment depends on the context of the input file. This rolling hash use a sliding window which moves over complete input file taking a byte at every clock cycle. In this paper window size is decided as 3 bytes long for maximum accuracy. The movement of sliding window is shown below:

[“A”, “ ”, “ ”]
[“A”, “B”, “ ”]
[“A”, “B”, “C”]
[“B”, “C”, “D”]
[“C”, “D”, “E”]
[“D”, “E”, “F”]
[“E”, “F”, “G”]

Rolling hash for segmenting input file pays important role. This method is used to calculate reset points. These reset points or boundaries are used to determine the amount of a file need to segmented or sliced. It picks the boundaries based on input file content within the fixed windows size of the rolling hash. Rolling has is needed for this purpose as employed by spamsum and ssdeep, discussed in [2] and [3], to calculate values throughout the file and search for a fixed value. When this fixed value is reached, a boundary line is drawn and the content gets sliced or segmented.

Rolling window would continue through the file, adding a new byte at each clock cycle and removing the oldest byte, in a FIFO style. This will be continue until the trigger or reset point is reached, once reset point is reached a boundary line is drawn and till this reset point all the bytes of data will constitute a segment or sliced or a block and again window start moving segmenting the files till the end of the file. The window performed some mathematical calculations for defining the reset point.

RH = [P]*a + [Q]*b + [R]*c (1)

If RH% p == m (reset value).

Then slice the input fill.

P, Q and R are the 8-bit registers to store the contents of window but actually this will be store the ASCII value of the respective characters. The implementation in Verilog is shown in the below Fig.1

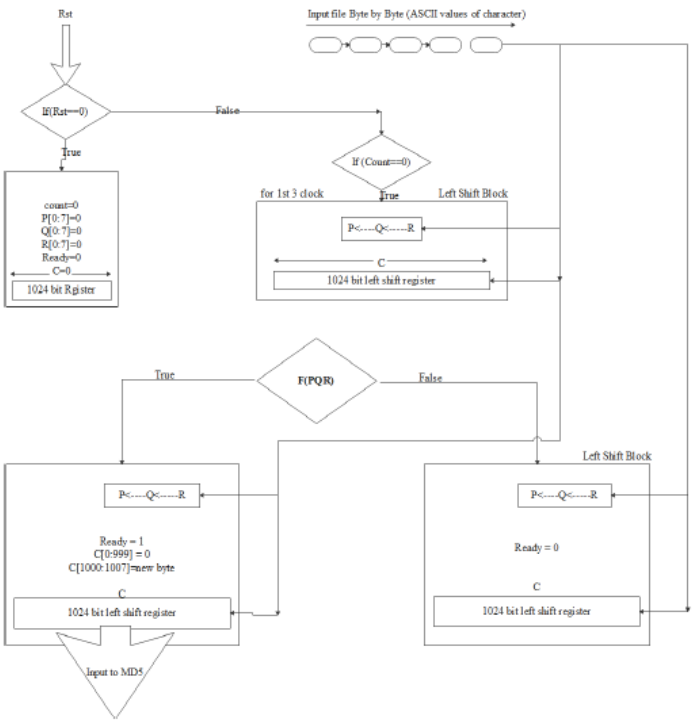

Fig. 1 Rolling Hash FSM

In the above Fig.1 Rst is reset condition which initializes the reset conditions, this circuit takes the Input text file byte by byte the ASCII value of character. After Rst =1 for 1st time for 1st 3 clock 1st left shift block executes, then at each positive edge of clock a certain condition on P, Q, R is checked and the respective block executes and produce the output, output will be sequence of input byte. The function which works on P, Q and R and the window sizing decides the efficiency and limit of the size of each segment or slice of the given input text file. The register C shown in above Fig.1 keeps on storing the content byte by byte coming from window by shifting its data to left till the window reaches reset point. Once the reset point is reached, the content stored in register C will be a segment of input file and this segment will be the input to our modified MD5 algorithm.

Each and every block in Fig. 1 executes on each positive edge of clock with respect to given condition.

For the text file given below the segments are shown in Fig.2. The File_1 is of 224 bytes including white spaces and newline character (\n). 224 bytes long file is converted to 7 segments, what every the size of segments, each segment is converted to 32 bit signature with modified MD5 algorithm. This is how compression is achieved.

For the specific values of a, b, c, p and m, the slicing input file result is analyzed below are the images of input file and its sliced segments

a = 13
b = 169
c = 255
p = 33
m = 32

File_1

To match the design needs with economic balance, hhrlhggvfmnbuiukjhiykmhtjiiuyrkndccxs,ptuyrtdis welding is used to achieve the high strength welding joints of dissimilar steel grades, tobe used in commercial vehicle.

```
['T', 'o', ' ', 'm', 'a', 't', 'c', 'h', ' ', 't', '
h', 'e', ' ', 'd', 'e', 's', 'i', 'g', 'n', ' ', 'n'
, 'e', 'e', 'd', 's', ' ', 'w', 'i', 't']
['h', ' ', 'e', 'c', 'o', 'n', 'o', 'm', 'i', 'c', '
 ', 'b', 'a', 'l', 'a', 'n', 'c', 'e', ',', '\n', 'h
', 'h', 'r', 'l', 'h', ' ', 'g', 'g', 'v', 'f', 'm',
 'n', 'b', 'u', 'i', 'u', ' ', 'k', 'j', 'h', 'i', '
y', 'k', 'm', 'h', 't', 'j', ' ', 'i', 'i', 'u', 'y'
, 'r', 'k', 'n', 'd', ' ', 'c', 'c', 'x', 's', ',',
'p', 't', 'u', 'y', 'r', 't', 'd', '\n', 'i']
['s', ' ', 'w', 'e', 'l', 'd', 'i', 'n', 'g', ' ', '
i', 's']
[' ', 'u', 's', 'e', 'd', ' ', 't', 'o', ' ', 'a', '
c', 'h', 'i']
['e', 'v', 'e', ' ', 't', 'h', 'e', ' ', 'h', 'i', '
g', 'h', ' ', 's', 't', 'r', 'e']
['n', 'g', 't', 'h', '\n', 'w', 'e', 'l', 'd', 'i',
'n', 'g', ' ', 'j', 'o', 'i', 'n', 't', 's', ' ', 'o
', 'f', ' ', 'd', 'i', 's', 's', 'i', 'm', 'i', 'l',
 'a', 'r', ' ', 's', 't', 'e', 'e', 'l', ' ', 'g', '
r', 'a', 'd', 'e', 's', ',', ' ', 't', 'o', ' ', '\n
', 'b', 'e', ' ', 'u', 's', 'e', 'd', ' ', 'i', 'n',
 ' ']
['c', 'o', 'm', 'm', 'e', 'r', 'c', 'i', 'a', 'l', '
 ', 'v', 'e', 'h', 'i', 'c', 'l', 'e', '.']
```

Fig.2 Segments for Input File_1

If one of the characters is altered, deleted, or inserted then there will be chance in one segment or at most two segments. This each segment is fed to modified MD5 which results in a 32 bit (4 byte) signature, so there will be 7 signatures each of 4 byte, this is how we convert 224 bytes of file to (4 * 7) = 28 bytes of data. Design for modified MD5 is shown in next section.

III. MODIFIED MD5 HASHING ALGORITHM

The objective here was to implement a 32-bit hash generator using the MD-5 algorithm in Verilog HDL. For this purpose, instead of understanding and building the core with dependencies from scratch, a preliminary study was made into how MD5 algorithm works. This provided a solid foundation for further changes as required to scale the 512 bit architecture to 128 bit architecture.

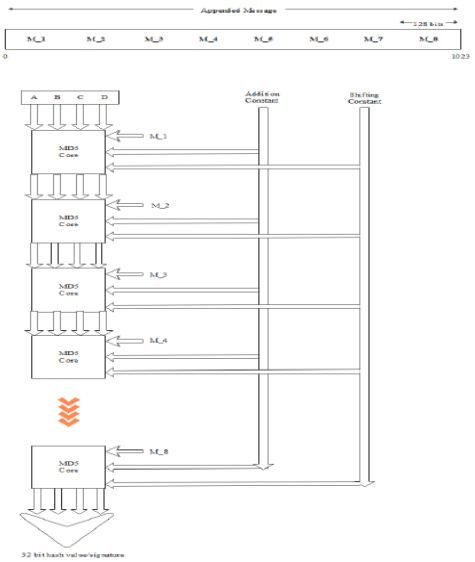

Fig.3 Combinational MD 5 block

For initial scaling down original architecture was required. This architecture was used as original MD5 core which was later scaled down. Now, the original algorithm computes in 4 blocks of 32 bit size each (A, B, C, D), giving a 128bit output. The original MD5 is very well explained in [5]. This was scaled down to 8bit so as to give a 32bit output. Similarly, the shifts and random additions also need to be down converted from 32bits to 8bits. This was achieved by taking modulo 4 of all previously given shifts and additions. This gave the values as required without much problems. However, the starting values for the registers A, B, C, D gave recurring values 0x01, 0x06, 0x01, 0x06, so the actual algorithm was looked at and the characteristics were studied and copied to 8bit values 0x56, 0xab, 0x65, 0xba. Thus all requirements were met, which then later led to a 32bit hash sum result.

The output obtained from the Rolling hash will be a stream of byte of random sized but with a certain maximum limit defines bye constants and the calculations used in rolling hash. With the certain constants define in section II, an approximate limit of 1000 bits is decided.

Appended message is a 1024 bit long stored in register from 0^{th} to 999^{th} will be the output from rolling hash and the bits from 1000th to 107th are the padded bits and the rest 16 bits use to represent the message size.

In Fig. 3, M_1 to M_8 are 128 bit register, A, B, C and D are 8 bit each.

1024 /128 = 8

In this modified MD5 algorithm 8 MD5 cores or stages are required; each will process 128 bits inputs of the given appended message.

Fig. 4 is showing the MD5 core, each MD5 core process the 128 bits in 4 rounds and each round uses a function.

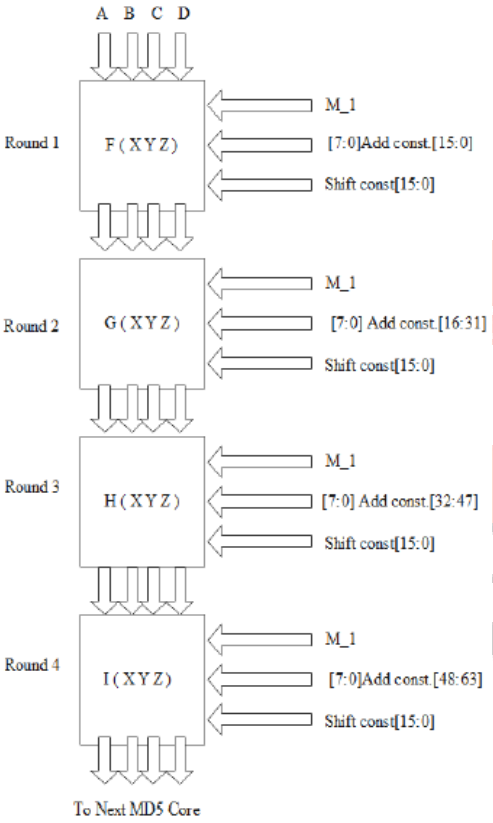

Fig. 4 MD 5 Core

F => { { [X] & [Y] } | { [~X] & [Z] } } (2)

G => { { [X] & [Z] } | { [Y] &[~Z] } } (3)

H => { [X] ^ [Y] ^ [Z] } (4)

I => { [Y] ^ [X] | [~Z] } (5)

Each round in MD5 core has 16 operations, each operation takes 8 bit input value of A, B, C and D. 8 bit stream out of 128 bit of total appended 1024 bit appended message, 8 bit add const. and 8 bit shift const.

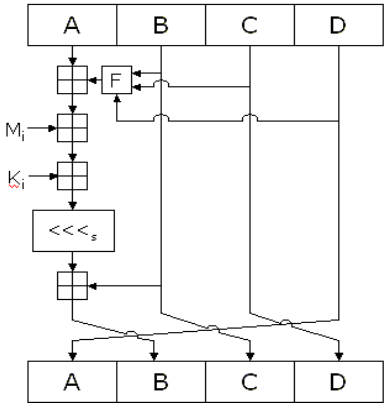

Fig. 5 MD-5 (1 of 16 operation in each round)

Note: M_i and K_i in the Fig. 5 are 8 bit input stream and add. Const. respectively, and <<<s is representing left shift by s number of bits.

Each operation produce four 8 bit output which is input to next operation. For the very 1st operation the values of A, B, C and are initialized to 0x56, 0xab, 0x65, 0xba.

IV. IMPLEMENTATION OF LEVENSHTEIN DISTANCE

Minimum edit distance for approximate string matching is the core of any similarity detector. Rolling hash and the MD5 process two file and store the respective hash values in memory. Let say the input Hash_1 represents all the memory content of A and input Hash_2 represents all the memory content of B. the Fig. 6 below shows the overall view of any Edit distance algorithm, where it need both the memory content as input, on comparing the hash values or signature from both memory A and B it will generate the output determining the difference between A and B.

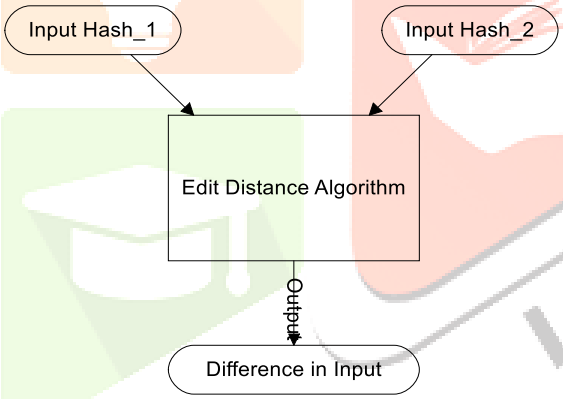

Fig. 6 Edit Distance

Depending on previous values, newvalue to each cell gets initialized. The formula for calculating the value of next cell is given as

$$r[i][j] = r[i-1][j-1] \quad (6) \quad \text{// if Hash\_1[I]=Hash\_2[j]} \quad (7)$$

$$r[i][j] = \text{minimum}(r[i-1][j-1], r[I][j-1], r[i-1][j]) \quad \text{//else}$$

The last cell will be output value which gives the dissimilarity between Input Hash_1 and Input Hash_2.

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Fig. 7 Edit Distance algorithm [Levenshtein.net]

The Fig. 8 shows the implementation of Edit Distance for difference detector. This is the sequential circuit, works on predefined memory size which stores the hash values or signatures. In this circuit each block executes at every positive edge of clock according to given condition. This requires three counters two for tracing the memory content from A and B, and one for storing the value in next cell. Out of the two counter use for tracing memory one is counting columns and other is rows.

Loading = 1 is the resetting condition, when Loading = 0 the actual working starts. When the Row counter reaches (N-1)th count, output is ready. For Row counter less than N-1, if column counter reaches N-1, the next Row operation gets start else it will be finding the each cell value in that Row. This continues till Row counter reaches N. Total number of clock cycles require are N^2.

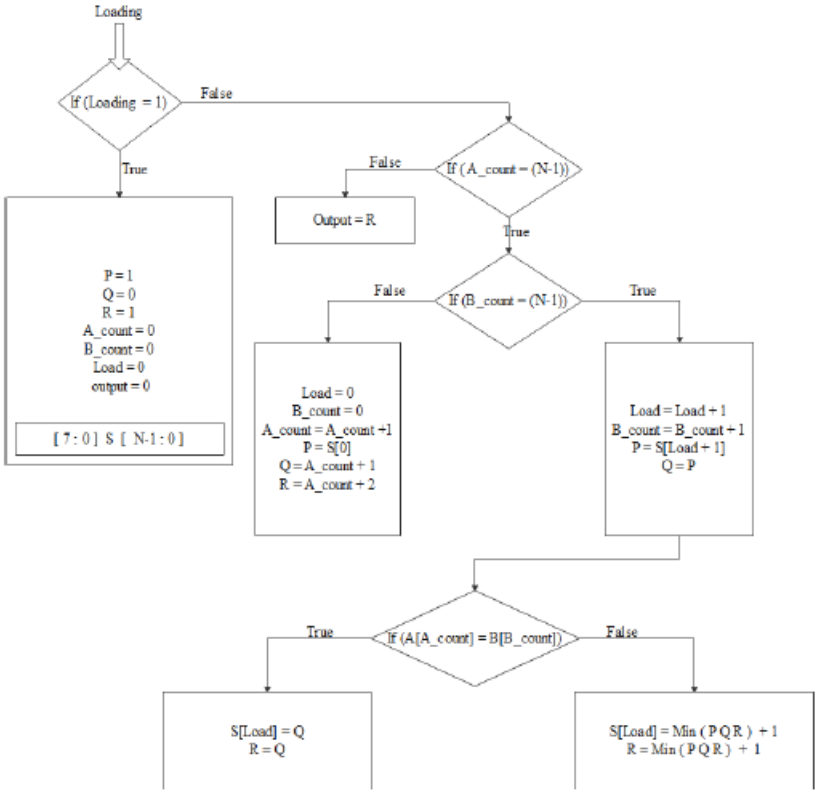

Fig 8 Edit Distance FSM

The Fig. 9 shows the interconnection of all three block.

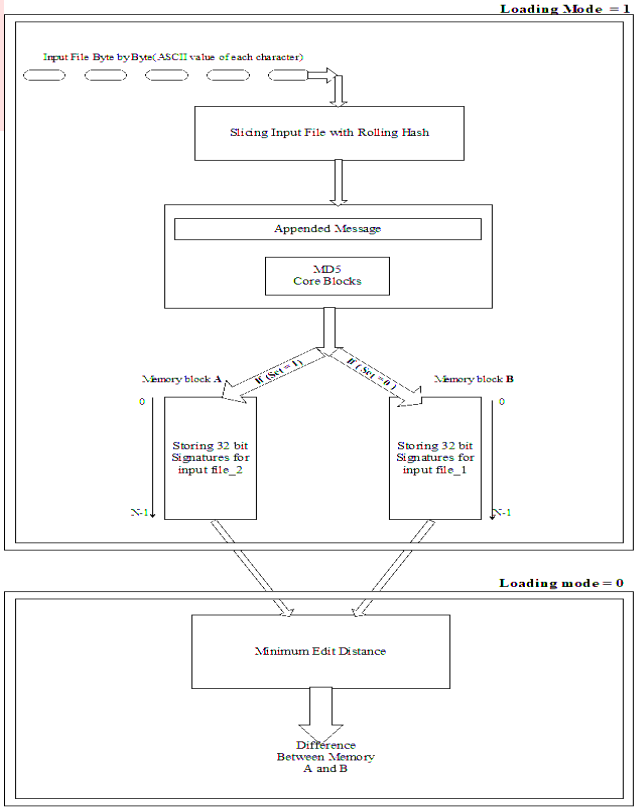

Fig. 9 Overview of complete fuzzy hashing

V. EXPERIMENTAL RESULTS

File_1:

Nowadays, with the rapid development of Internet, there are a huge number of documents in many different fields such as science, technology, medical, literature, businesses etc. Everyone can easily find documents they need. Checking Datasimilarity over a certainperiod of time is good.

File_2:

Nowadays, with the rapid development of Internet, there are a huge number of documents in many different fields such as science, technology, me corrupting the input data here etc.Everyone can easily find documents they need. Checking Data similarity over a certain period of time is good.

Table 1: 32 bit hash values – signatures for file_1 and file_2

File_1	File_2
77d7271e	77d7271e
9d64334c	9d64334c
1e7d787e	1e7d787e
36fb9670	36fb9670
87de5e5a	87de5e5a
036e871f	036e871f
89167112	89167112
35b04c33	bc223ef5
7dbe9256	f8e25f50
53c29bcf	bbefba4b
383db613	2d7cebff
188dc613	f8311745
70f62d41	383db613
41b4e620	188dc613
4f5b1281	70f62d41
0ebf48ea	41b4e620
d1b225c4	4f5b1281
	0ebf48ea
	d1b225c4

Table 2: Timing report

Minimum period	6.707ns (Max. Freq.: 149.096MHz)
Minimum input arrival time before clock	1.781ns
Maximum output required time after clock	0.654ns
Maximum combinational path delay	0.280ns

DEVICE UTILISATION SUMMARY:

For Selected Device: 7a100tcsg324-3

Table 3: Slice Logic Utilization

No. of Slice Registers	1035 out of 1268000%
No. of Slice LUTs	1021 out of 63400 1%
No. used as Logic	1021 out of 63400 1%

Table 4: Slice Logic Distribution

No. of LUT Flip Flop pairs used	1043
No. with an unused Flip Flop	8 out of 1043 0%
No. with an unused LUT	22 out of 1043 2%
No. of fully used LUT-FF pairs	1013 out of 1043 97%
No. of unique control sets	5

Table 5: IO Utilization

No. of IOs	1120
No. of bonded IOBs	1120 out of 2210 51%
IOB Flip Flops/Latches	8

Table 6: Specific Feature Utilization

No. of BUFG / BUFGCTRLs	1 out of 323%
No. of DSP48E1s	3 out of 240 1%

Table 1 shows the different signature generated for File_1 and File_2 input, Table 2 shows the Timing report generated, Table 3 shows slice logic utilization, Table 4 shows slice logic distribution, Table 5 shows IO utilization and Table 6 shows specific feature utilization for the selected FPGA device 7a100tcsg324-3. Several such files were given as input to check the reliability of the code. Finally we could successfully test the signatures and using the edit distance algorithm, we could calculate the dissimilarity distance. Fig. 10 shows the simulation snippet from Xilinx ISE. The distance i.e. difference between signatures of two files shown here is 14 for two sets of data. These values are used to generate trapezoidal Fuzzy hedges with hedges {less similarity, moderate similarity, high similarity}.

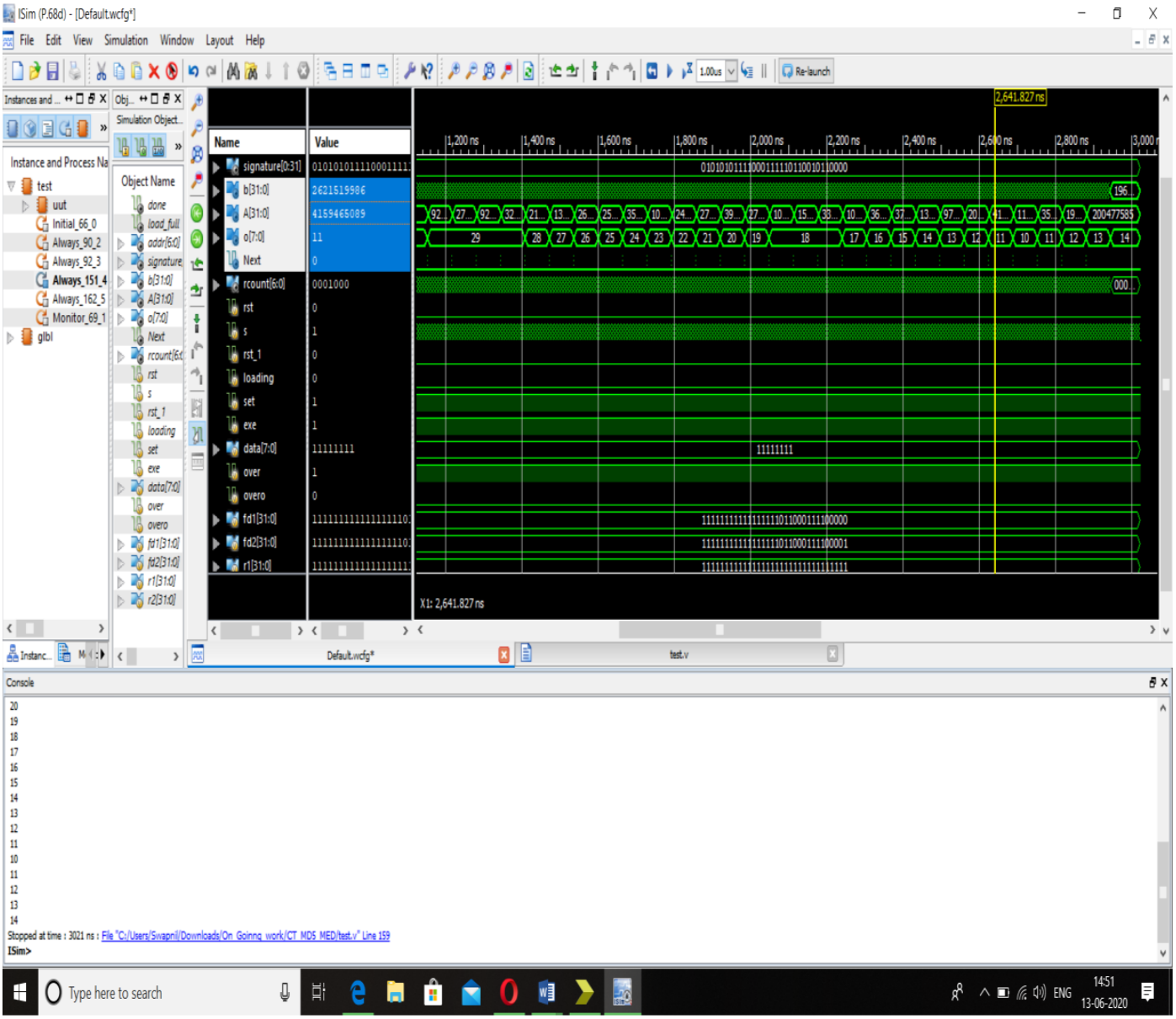

Fig. 10 Simulation snippet from xiling ISE

VI. CONCLUSION

Thus, the similarity detector is implemented successfully for low end IoT devices using Verilog HDL. This can further help in strengthening digital forensic of low end IoT devices with low computational resources and low power. The similarity detector has three major parts the rolling hash for segmenting the input text file, Hashing algorithm MD5 and the Edit Distance algorithm. All the major and minor parts of this design have been tested and appropriate results are verified. The rolling hash for segmenting input text file is the sequential circuit which takes ASCII value of single character as an input at every clock cycle, and generates the output upon reaching the reset point. The modified MD5 algorithm as a combinational circuit is successfully designed which will generate output once the rolling hash generates its output. The Edit Distance is implemented successfully as a sequential circuit and requires N*M clock cycles for final output. N represents number of 32 bit memory location for input file1. M represents number of 32 bit memory location for input file2. Output generated from Edit Distance is an integer value showing the difference between signatures of input file1 and input file2. With reference to the one of the file's number of signatures generated we can calculate similarity.

VII.ACKNOWLEDGEMENT

The Authors would like to thank E&TC Department, COEP Technological University for providing lab facilities for carrying out this work. Special thanks to Swapnil Daware for carrying out experimental work.

Conflict of Interest: On behalf of all authors, the corresponding author states that there is no conflict of interest.

Ethical Approval: This article does not contain any studies with human participants or animals performed by any of the authors.

REFERENCES

[1] Vanita Agarwal, R. A. Patil, A. B. Patki, "Architectural Considerations for Next generation IoT Processors", IEEE Systems Journal, Vol.13, No. 3, 2906-2917, ISSN: 1937-9234 c 2018 IEEE

[2] Nikolaos Sarantinos , Chafika Benzaıdy , Omar Arabiatz and Ameer Al-Nemrat, "Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities", 3rd International Conference on Devices, Circuits and Systems (ICDCS) 2016.

[3] Jesse Kornblum "Identifying almost identical files using context triggered piecewise hashing" The Digital Forensic Research Conference DFRWS 2006 USA Lafayette, IN (Aug 14th - 16th).

[4] Jonathan Oliver, Chun Cheng and Yanggui Chen, "TLSH- A locality sensitive Hash" 2013 Fourth Cybercrime and Trustworthy Computing Workshop.

[5] Jin Hwan Park, "Reconfigurable Parallel Approximate String Matching on FPGAs" 8th Euromicro conference on Digital System Design 2005.

[6] Takuma Wada, Shunji Funasaka, Koji Nakano and Yasuaki Ito, "A Hybrid Architecture for the Approximate String Matching on an FPGA", Fifth International Symposium on Computing and Networking 2017.

[7] Kenneth B. Kent, Ryan B. Proud foot, Yong Zhao Faculty of Computer Science, "Parameter Specific FPGA Implementation of Edit-Distance" Calculation.17th IEEE International Workshop on Rapid System Prototyping (RSP 2006), 14-16 June 2006.

[8] Janaka Deepakumara, Howard M. Heys and R. Venkatesan, "FPGA Implementation of MD5 Hash algorithm", 11th International Conference on Trust, Security and Privacy in Computing and Communications 2012 IEEE.

[9] Sadatoshi MIKAMI and Yosuke KAWANAKA, "Efficient FPGA based hardware algorithm for approximate string matching" The 23rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2008).

[10] Takuma Wada, Naoki Matsumura, Koji Nakano and Yasuaki Ito, "Efficient Byte Stream Pattern Test using Bloom Filter with Rolling Hash Functions on the FPGA" Sixth International Symposium on Computing and Networking (CANDAR) 2018.

[11] Panagiotis D. Michailidis, and Konstantinos G. Margaritis, "Implementation of a Programmable Array Processor Architecture for Approximate String Matching Algorithms on FPGAs", 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece.

[12] Guohua WU, Liuyang WANG and Ershuai FU, "Document Copy Detection Using the Improved Fuzzy Hashing" 2015 International Conference on Computer Science and Mechanical Automation.