HIGH PERFORMANCE PARALLEL STREAMING BASED HYBRID HUFFMAN ENCODER

Namburi Sambamurthy

Assistant Professsor
Department of ECE
Seshadri Rao Gudlavalleru
Engineering College, Gudlavalleru

Talabattula Harish

UG Student
Department of ECE
Seshadri Rao Gudlavalleru
Engineering College,
Gudlavalleru

Yarramsetti Jagadeep

UG Student
Department of ECE
Seshadri Rao Gudlavalleru
Engineering College, Gudlavalleru

Yadala Prashanth

UG Student
Department of ECE
Seshadri Rao Gudlavalleru
Engineering College, Gudlavalleru

Dasari Satya Prabhas

UG Student
Department of ECE
Seshadri Rao Gudlavalleru
Engineering College, Gudlavalleru

---***---

Abstract—The Huffman encoder uses the Huffman encoding technique, which effectively reduces input strings of eight bits into far fewer bits often down to only one bit or more. The two primary parts of this method are a Huffman block and a frequency sorting block. The frequency sorting block sends the sorted data to the Huffman encoder by arranging the input strings in ascending order according to their frequency and ASCII values. Every byte in the Huffman block is encoded into less than 8 bits; in the worst situation, a single byte might be compressed into as low as 1 bit. In order to illustrate the compression of these values, this work concentrates on processing 64 bits of data, or 8 string values.

Keywords–Huffman encoding, Compression, Frequency Sorting, ASCII values, Data Processing.

I.INTRODUCTION

Huffman encoding is a well-known method for significantly reducing the quantity of input data while maintaining its frequency order that is used in many networking applications and routers. Sorting the input data right before feeding it into the Huffman block ensures a greater compression ratio and optimal data accuracy. In order to lessen the drawbacks of current devices, this work presents a high-throughput VLSI design for Huffman encoding that is based on the conventional/canonical Huffman encoder.

A mainstay of data compression, Huffman encoding provides a potent way to reduce the size of data payloads across a variety of networking platforms and devices. Huffman encoding becomes a vital technique for attaining ideal data compression while preserving critical information integrity as the need for effective data transmission and storage in contemporary communication systems grows. Huffman encoding reduces the total bitstream size by assigning shorter codes to symbols that appear more frequently by using the frequency distribution of symbols within a dataset.

The use of Huffman encoding becomes especially important in the context of networking technologies and routers, where bandwidth and memory resources are frequently constrained. Huffman encoding enables higher throughput and lower latency in data transfer by compressing data payloads without sacrificing important information. Additionally, Huffman encoding facilitates prioritized transmission by grouping data according to symbol frequency, guaranteeing that important data is given priority throughout data transfer procedures.

Huffman encoding, however, has major benefits in terms of data compression and transmission efficiency; nonetheless, its implementation can be difficult, especially in terms of throughput and hardware complexity. The usability of conventional Huffman encoder designs in contemporary networking contexts may be limited by their inability to fulfil the needs of high-speed data processing. Furthermore, inefficiencies in current implementations might lead to less-than-ideal compression ratios and more processing overhead.

In order to overcome these obstacles and improve Huffman encoding efficiency in networking applications, a unique high-throughput VLSI design for Huffman encoding is

proposed in this research. The suggested design seeks to address the limitations of current devices by providing increased throughput, less hardware complexity, and greater compression efficiency. It draws inspiration from both conventional and canonical Huffman encoding approaches. The suggested architecture's effectiveness and performance advantages are shown through in-depth study and experimental validation, underscoring its potential to completely transform networking technologies' use of data compression.

1.1Motivation

Effective data compression methods are desperately needed in networking technologies, which is what spurred this research. High-throughput Huffman encoder designs are essential given the exponential expansion of data transmission and the limitations of finite bandwidth and memory resources. Current implementations might not be able to match these needs, which would result in higher processing overhead and less than ideal compression ratios. The goal of this study is to improve throughput, lower hardware complexity, and increase compression efficiency by presenting a unique VLSI design for Huffman encoding. The ultimate goal of this effort is to push data compression for networking applications closer to the state-of-the-art.

1.2 Objectives:

1. Create a Huffman encoding high-throughput VLSI architecture that satisfies the demands of contemporary networking technologies.

2. Optimize the suggested design to reduce hardware complexity and get better compression efficiency.

3. To satisfy the demands of high-speed data processing in networking contexts, improve the Huffman encoder's throughput capabilities.

4. Use in-depth research and testing to verify the suggested architecture's performance against other Huffman encoder implementations already in use.

5. Describe the practical ramifications and possible uses of the suggested design to improve networking system transmission efficiency and data compression.

1.3 Existing System:

There are, in fact, a number of approaches and modifications in the field of Huffman encoding, each having advantages and disadvantages of its own. By mapping input symbols directly to variable-length codewords using hash functions, hash encoding, for instance, eliminates the requirement to build a Huffman tree based on symbol frequencies. Compared to classic Huffman encoding, hash encoding may be more difficult to obtain ideal compression ratios, especially for datasets with skewed symbol distributions.

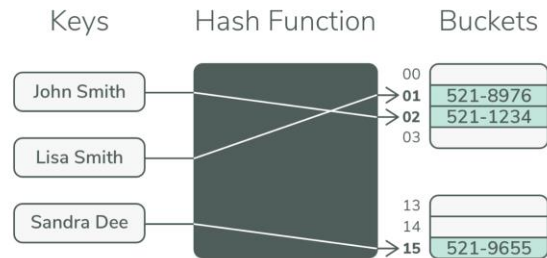


Fig1:Existing System

Nevertheless, it can provide simplicity and perhaps quicker encoding and decoding times.Some Huffman encoding implementations could build the Huffman tree straight from the input data by using techniques like adaptive algorithms or dynamic programming, doing away with the explicit sorting of symbols based on frequency. While these techniques could be beneficial in some situations, including real-time data processing or streaming applications, they may also introduce additional computational complexity and overhead.

Whether to use frequency-sorted Huffman encoding, hash encoding, or classic Huffman encoding relies on a number of criteria, such as the application or system's unique needs, the desired compression ratio, and the properties of the input data. Identifying and comprehending the trade-offs between these various techniques is crucial to creating effective and efficient data compression solutions that meet the requirements of various networking scenarios.

II.RELATED WORK

In their work, Sarkar, Sarkar, and Banerjee offer a unique method of data compression. They suggest using Huffman coding, a well-known compression method, to efficiently shrink the size of big data arrays[2]. They hope to obtain substantial storage requirements reductions while maintaining the integrity of the dataset by implementing Huffman coding. In the context of contemporary computing and communication systems, where big datasets are typical and effective data management is essential, this strategy is especially pertinent. The implementation specifics, performance assessment, and possible uses of their innovative technique are probably covered in the article,

which offers insightful information on the suitability and effectiveness of using Huffman coding for complex data compression jobs.

A unique VLSI architecture designed for high-throughput encoding of canonical Huffman codes is presented by Shao et al. focuses on the requirement for effective hardware Huffman encoding implementations, which is essential for many applications including data compression and communication systems[4]. The suggested architecture seeks to maximize throughput performance while reducing hardware complexity by concentrating on canonical Huffman codes, which provide streamlined encoding and decoding procedures. The authors most likely describe in depth the design process, optimisation strategies, and performance assessment of their architecture, showcasing how well it meets the demands of real-time data processing.

Canonical Huffman coding is investigated by Khaitu and Panday as a method of picture compression[5]. The work, which was published in the conference proceedings, probably explores the use of canonical Huffman coding, which provides a more straightforward encoding and decoding procedure than conventional Huffman coding methods. The authors want to show how canonical Huffman coding might reduce digital picture storage needs without sacrificing image quality by concentrating on image compression.

It is probable that Xilinx offers recommendations and best practices for designing protocol processing systems using Vivado High-Level Synthesis [7]. Given that Xilinx is a well-known supplier of FPGA technology, it is likely that this article provides information on how to effectively perform protocol processing tasks on FPGA platforms by utilizing Vivado HLS, a tool that transforms high-level language descriptions into RTL (Register Transfer Level) designs. This resource likely helps FPGA designers create protocol processing systems with better performance, flexibility, and time-to-market by providing methods, case studies, and optimization strategies. In order to successfully implement protocol processing functions, the paper may also provide useful examples and suggestions for integrating Vivado HLS into FPGA design workflows.

The field of VLSI design techniques is explored by Mukherjee, Ranganathan, and Bassiouni with the goal of improving the effectiveness of data transformation procedures for tree-based codes[16]. This study probably investigates novel strategies and optimisation techniques for effectively executing data transformations on VLSI hardware platforms. For encoding and decoding activities, tree-based codes which are often utilized in a variety of applications, including error correction and data compression need effective data transformation processes. It is probable that the article will include innovative VLSI designs, architectures, and algorithms designed to reduce hardware complexity and resource consumption and speed up data translation activities. Practical advice and experimental findings are included, which probably makes a significant addition to the field of VLSI design for data processing applications and offers direction to practitioners and researchers looking to increase the effectiveness and performance of tree-based code implementations on VLSI platforms.

2.1CANONICAL HUFFMAN CODING:

Canonical Huffman coding is a variation of Huffman coding, a well-liked data compression method with many applications because of its efficiency and simplicity. Canonical Huffman coding assigns a predetermined canonical code to each symbol as opposed to regular Huffman coding's assignment of variable-length codes depending on symbol frequencies. Based on a certain symbol ordering—typically arranged in ascending order by symbol value—these canonical codes are established. Compared to regular Huffman coding, canonical Huffman coding has a number of benefits, such as streamlined encoding and decoding procedures, lower memory needs, and improved error robustness.

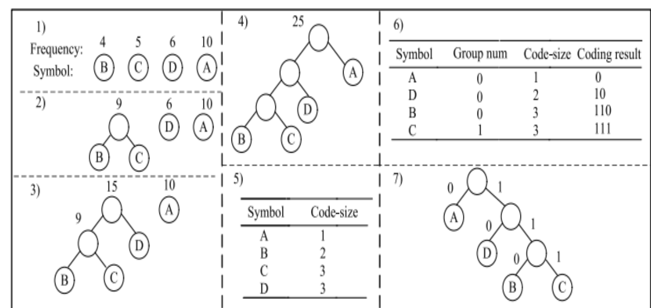


Fig2:Represents canonical Huffman encoding tree algorithm.

Every symbol that is used to build a Huffman tree is first represented by a parentless node that contains the symbol and its probability. The construction of the tree structure begins at these nodes. The method then successively chooses the two nodes with the lowest probabilities, joins them to form a new parent node, and gives this parent node a probability that is equal to the total of the probabilities of all of its offspring. This procedure keeps on until the Huffman tree's root—a single parentless node—remains. In the course of building the tree, greater probability symbols are positioned closer to the root, allowing for shorter codewords for more often recurring symbols.

After the Huffman tree is built, a path from the tree's root to each symbol must be traced in order to encode each symbol. A '1' is assigned for each branch taken in one direction and a '0' for each branch taken in the other direction as the traverse proceeds. Each symbol is uniquely identified by a binary pattern created by this method. For example, the symbol 'A' is encoded as '0', 'B' as '100', 'C' as '101', 'D' as '110', and 'E' as '111' in the given example. By guaranteeing that the most frequently occurring symbols receive the shortest codewords, this encoding approach maximizes the Huffman coding technique's overall compression efficiency.

Simple and effective encoding and decoding procedures characterize canonical Huffman coding. Constructing and storing a Huffman tree is not necessary during encoding or decoding since the codewords are assigned in a predefined canonical sequence. Canonical Huffman coding is a good fit for resource-constrained situations like embedded systems or hardware implementations since it greatly minimizes the computational cost and memory overhead associated with regular Huffman coding. When considering canonical Huffman coding against regular Huffman coding, the former has better error robustness. Small faults or variations in symbol frequencies are less likely to cause problems for the encoding and decoding operations since the codewords are allocated according to a predetermined sequence instead than being dynamically calculated depending on symbol frequencies.

2.2ASCII Values:

In digital computer equipment, characters are represented by numeric codes according to the ASCII character encoding standard. A 7-bit binary integer is used in ASCII to represent each character, for a total of 128 possible character combinations. These characters contain numerals, control characters, punctuation marks and special symbols in addition to capital and lowercase letters. The ASCII values for the capital and lowercase letters 'A' and 'a', respectively, are 65 and 97, respectively. ASCII values offer a standardized way to represent textual information across many platforms and programming languages. They are widely used in computer systems for text encoding, communication protocols, and data transfer.

2.3VLSI

The needs and goals for the electronic circuit that has to be created are described in the design specifications. Using symbols to create a visual depiction of the circuit and linking them to show the connections and functionality of the individual components is known as schematic capture. Custom symbols can be made to represent particular elements or operations that aren't easily found in standard libraries. Through the use of specialised software, simulation involves evaluating the circuit's performance and analysing its behaviour under various inputs and situations. Arranging the circuit's physical parts on a printed circuit board or integrated circuit while taking signal integrity, routing, and size into account is known as layout. To avoid mistakes or manufacturing problems, Design Rule Check verifies that the layout complies with specified design guidelines and limitations.

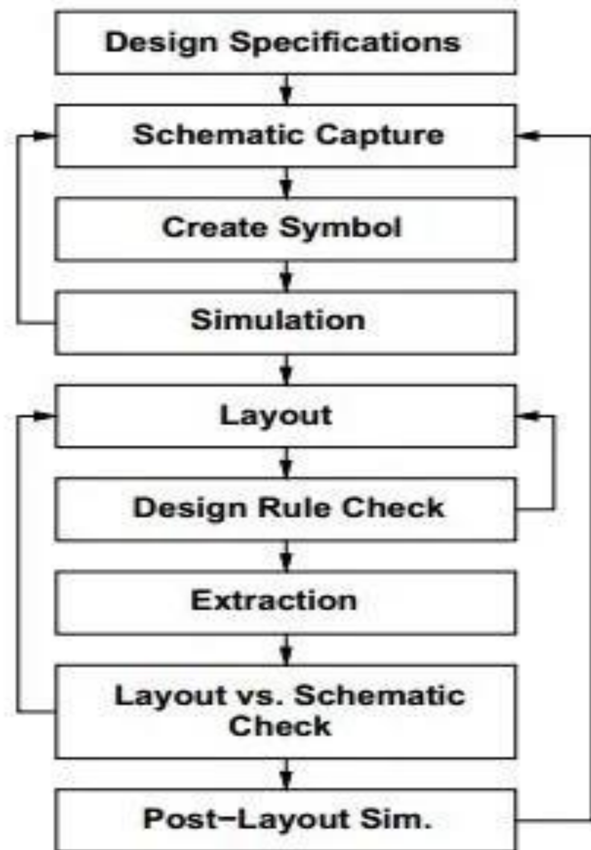


Fig3:VLSI Design

In order to get more realistic simulation results, extraction entails removing parasitic parts and features from the plan. The Layout vs. Schematic (LVS) check verifies that the schematic and layout match and finds any inconsistencies or mistakes. Post-layout simulation offers insights into the real-world behaviour and performance characteristics of the circuit by validating its performance based on the finalised layout. Together, these phases make up the design process for electrical circuits, which guarantees that their performance, dependability, and usefulness satisfy the criteria.

III.PROPOSED METHOD:

The process of implementing Huffman encoding starts with the input data, which is the unprocessed data that has to be compressed. The Huffman Buffer receives this input and processes it first there. In addition to the data, a Frequency Count is produced, indicating how frequently each letter or symbol that appears in the input occurs. Finding the best encoding method based on symbol probabilities requires this frequency count. Sorting comes next after the frequency count has been determined. Based on how frequently symbols appear, the Frequency Count is sorted. Sorting makes ensuring that throughout the encoding process, symbols with higher frequencies are given priority, enabling more effective compression. Following sorting, the information is arranged to make it easier to subsequent processing steps.

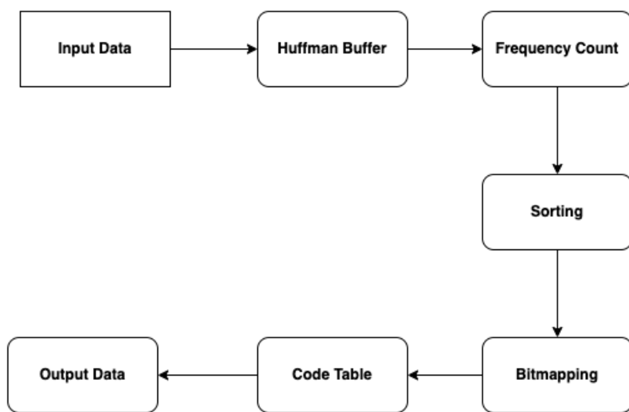


Fig4:Proposal Model

Sorting is followed by the Bitmapping procedure. Based on the frequency and location of each symbol in the sorted list, a binary code is assigned to it in this stage. Higher frequency symbols usually have shorter binary codes, which maximises the Huffman encoding scheme's compression effectiveness. By guaranteeing that every symbol is represented by a distinct binary pattern, bitmapping allows for accurate reconstruction to occur during decoding. The binary codes that are allocated to every symbol during the Bitmapping process are used to construct the Code Table as part of the implementation. The Code Table maps symbols to their corresponding binary representations and is used as a guide for encoding and decoding activities. When utilising the Huffman encoding technique to correctly compress and decompress data, this table is essential.

Ultimately, the compressed form of the input data is produced as the output data. Binary code sequences that are retrieved from the Code Table are commonly used to represent the compressed data. Due to effective symbol encoding based on frequencies, the output data has been compressed to a size substantially less than the original input data. Using the concepts of Huffman encoding, the implementation method makes sure that the output data retains data integrity while attaining the best compression ratios.

3.1Implementation:

The input data is a series of symbols with their corresponding frequencies shown: A, B, C, D, and E. In the input sequence, the symbols A, B, C, D, and E appear four times, seven times, two times, and four times, respectively, totaling 160 bits.

Sorting:

2	3	4	4	7
D	C	E	A	B

Bit-mapping involves allocating a binary code to every symbol according to its frequency and location in the sorted list. Higher frequency symbols usually have shorter binary codes, which maximises compression effectiveness. As an illustration:

B = 10 C = 110 D = 1110 E = 1111 A = 00

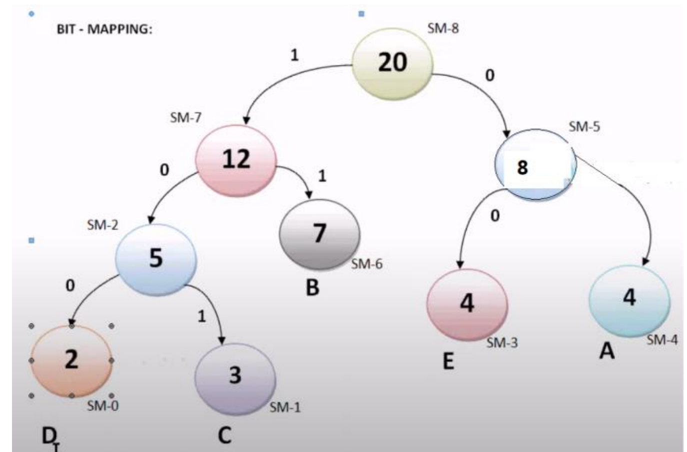


Fig5:BitMapping

During the encoding process, these binary codes are employed to represent the symbols, guaranteeing effective compression while maintaining data integrity.

A	01
B	11
C	101
D	100
E	00

Table1:Code table

This code table is used to represent the compressed data as follows:

An B C C D B C C D A, B, C, E, B, and E A

01 11 11 101 100 11 01 11 11 00 00 11 00 01 11

The original 160 bits of data have been reduced to 45 bits for this data.Furthermore, the compressor data code word form is supplied as follows, which indicates the frequency of occurrence and the associated code word: (2,01), (2,11), (2,11), (3,101), (3,100), (2,11), (3,101), (3,100), (2,01), (2,01), (2,11), (2,00), (2,00), (2,11), (2,01), (2,11)

Ninety bits are needed for this compressed data in word form compressor data code.

3.2 Software and Hardware Environment

Xilinx ISE (Integrated Synthesis Environment) is the synthesis tool used for the Huffman encoding project. It is a full-featured software tool for creating and assembling digital circuits that are intended to be used with Xilinx FPGAs (Field-Programmable Gate Arrays). With its array of synthesis, simulation, and verification functions, Xilinx ISE provides an intuitive user interface for creating and executing sophisticated digital systems. Designers may use Xilinx ISE to produce programming files for FPGA configuration, execute synthesis to transform RTL (Register Transfer Level) code into a gate-level netlist, and construct and optimise digital designs.

Furthermore, ModelSim is used in the undertaking for simulation needs. The popular HDL (Hardware Description Language) simulation programme ModelSim has sophisticated features for modelling digital designs defined in Verilog and VHDL (VHSIC Hardware Description Language). Before putting their digital circuits into hardware, designers may use ModelSim to mimic their circuits' behaviour, check for functioning, and troubleshoot any possible problems. ModelSim offers a powerful simulation environment and industry-standard language support, making it an effective platform for confirming the Huffman encoding design's functionality and efficiency.

By leveraging Xilinx ISE for synthesis and ModelSim for simulation, the project benefits from a comprehensive design flow that integrates synthesis and verification processes seamlessly. This integrated approach enables designers to develop and validate the Huffman encoding circuit efficiently, ensuring that the final implementation meets the project's requirements for performance, functionality, and reliability. With Xilinx ISE and ModelSim, designers have access to powerful tools and resources to facilitate the successful completion of the Huffman encoding project.

IV. RESULTS

The Huffman encoding process's output outcomes for the given input data are shown in the figure. It illustrates the binary codes assigned to each symbol and provides a visual representation of the compressed data, proving the effectiveness of the Huffman encoding approach in minimizing data size.

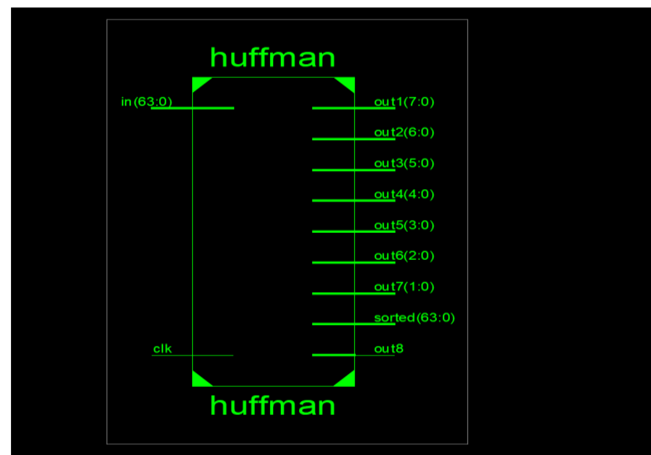


Fig6:Represent the output results for the input given.

The Huffman buffer module's simulated waveform shows how the buffer behaves during encoding. The waveform displays the processed and buffered input data together with signals denoting the beginning and ending of each symbol. The waveform also shows how each symbol's occurrence is tracked during the frequency count creation process. The Huffman buffer effectively handles the incoming data during the simulation, guaranteeing precise frequency counts and setting up the next encoding steps.

International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 11 Issue: 04 | Apr 2024 www.irjet.net p-ISSN: 2395-0072

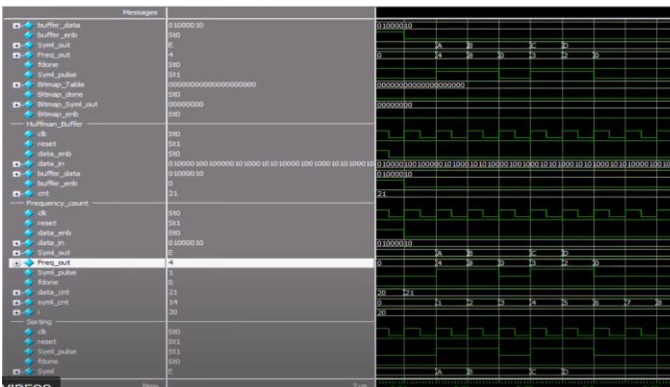


Fig7:Simulation waveform of Huffman buffer module

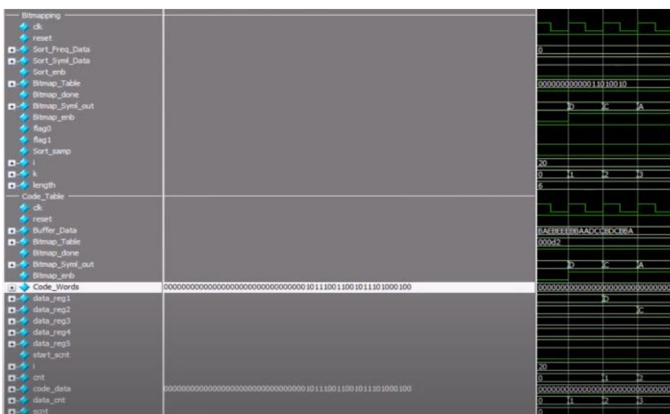
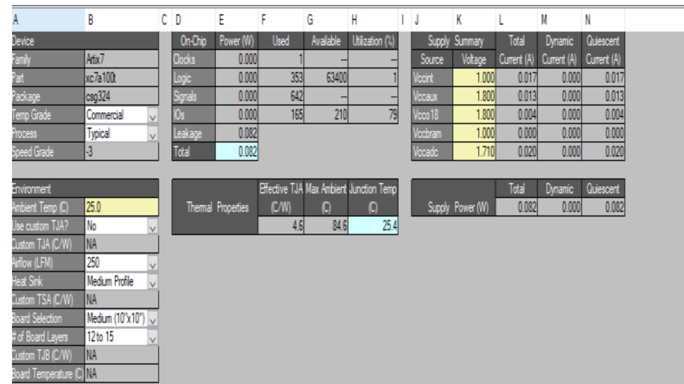


Fig8: simulation waveform of code table module

The code table module's simulated waveform shows how the module behaves dynamically during encoding. Based on the frequency and location of each symbol in the sorted list, it shows how binary codes are generated for each symbol. The waveform provides information on how the encoding process is being executed by displaying the interactions between the module's inputs, outputs, and internal signals.

By means of the simulation, it is noted that the code table module effectively ensures efficient compression of the input data by mapping symbols to their corresponding binary representations. The waveform illustrates how different symbols are converted into binary codes, emphasising the module's assistance with the encoding process.

The Huffman encoder's technical diagram is shown in the picture, which also shows the hardware parts and how they are connected throughout the encoding process. It displays the encoder's design, showcasing components including the sorting unit, bitmapping module, frequency counter, and Huffman buffer. The connections made between these modules show how data and control signals go through the encoder and illustrate the steps that the Huffman encoding method goes through in order.



Device	
Family	Artix7
Part	xc7a100t
Package	csg324
Temp Grade	Commercial
Process	Typical
Speed Grade	-3

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.000	1	---	---
Logic	0.000	353	63400	1
Signals	0.000	642	---	---
IOs	0.000	165	210	79
Leakage	0.082			
Total	0.082			

Supply Source	Summary Voltage	Total Current (A)	Dynamic Current (A)	Quiescent Current (A)
Vccint	1.000	0.017	0.000	0.017
Vccaux	1.800	0.013	0.000	0.013
Vcco18	1.800	0.004	0.000	0.004
Vccbram	1.000	0.000	0.000	0.000
Vccadc	1.710	0.020	0.000	0.020

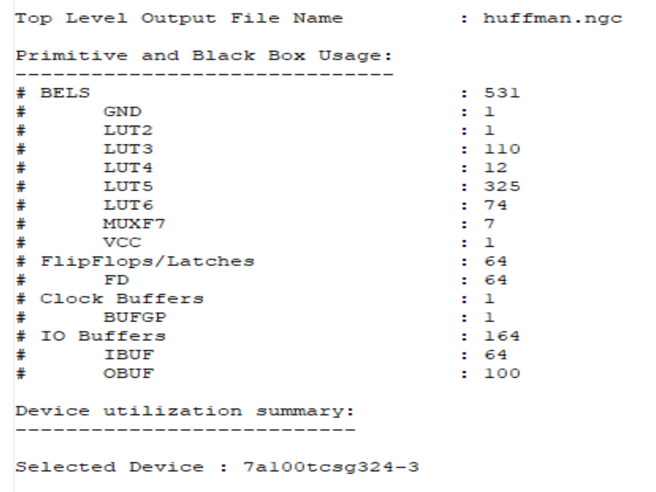
Environment	
Ambient Temp (C)	25.0
Use custom TJA?	No
Custom TJA (C/W)	NA
Airflow (LFM)	250
Heat Sink	Medium Profile
Custom TSA (C/W)	NA
Board Selection	Medium (10"x10")
# of Board Layers	12 to 15
Custom TJB (C/W)	NA
Board Temperature (C)	NA

Thermal Properties	Effective TJA (C/W)	Max Ambient (C)	Junction Temp (C)
	4.6	84.6	25.4

Supply Power (W)	Total	Dynamic	Quiescent
	0.082	0.000	0.082

Fig9:Technological diagram of the Huffman encoder

An exact breakdown of the hardware components used in the encoding process is given by the picture, which shows how many adders, comparators, and multiplexers the Huffman encoder uses. The distribution of computing components throughout the various encoding method stages is shown in this graphic representation, which provides insights into the complexity and resource usage of the encoder's architecture.Adders, comparators, and multiplexers serve as representations for the numerous arithmetic and logical operations that are needed throughout the encoding process. The amount of work required to perform the sorting, bitmapping, and symbol frequency counting phases of Huffman encoding is indicated by the number of each component. Designers may evaluate the computational needs and optimise the encoder's architecture for effectiveness and performance by visualising the distribution of these components.



```
Top Level Output File Name          : huffman.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 531
#      GND                         : 1
#      LUT2                        : 1
#      LUT3                        : 110
#      LUT4                        : 12
#      LUT5                        : 325
#      LUT6                        : 74
#      MUXF7                       : 7
#      VCC                         : 1
# FlipFlops/Latches                : 64
#      FD                          : 64
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 164
#      IBUF                        : 64
#      OBUF                        : 100

Device utilization summary:
---------------------------

Selected Device : 7a100tcsg324-3
```

Fig10: Number of adders and comparators and multiplexers used in Huffman encoder.

The Huffman encoder's implementation yielded findings that show notable differences in speed and resource use between modules. With 40 slice registers and 120 slice LUTs for the buffer and 385 slice registers and 554 slice LUTs for the frequency counting, the Huffman buffer and frequency count modules use comparatively less resources. In spite of this, they support crucial preprocessing stages in the encoding procedure.

	Huffman buffer	Frequency count	sorting	Bit Mapping	Code table	Top Module (Full work)
Number of slice Register	40	385	421	159	223	1529
Number of slice LUTs	120	554	2021	178	309	3097
Number of Occupied Slice	61	206	762	83	216	1294
Delay (ns)	3.701	4.69	8.026	3.879	5.192	8.016
power (w)	3.294	3.298	3.301	3.298	3.298	3.316

Table2:Results

On the other hand, the modules responsible for sorting and bit mapping demand larger resource allocations. Sorting requires 421 slice registers and 2021 slice LUTs, while bit mapping requires 159 slice registers and 178 slice LUTs. These modules are essential for organising the incoming data and producing binary codes that are based on the frequency of the symbols. As a result, their increased use of resources indicates their importance in the encoding process.

Significant resource consumption is shown by the code table generation and top module, which stand for the integration of all encoding components. The code table module's intricate process of converting symbols into binary codes is demonstrated by the use of 3097 slice LUTs and 1529 slice registers. Overall, our findings demonstrate how resource-intensive the Huffman encoding procedure is and stress how crucial effective hardware design is to achieving the best possible performance in terms of latency and power consumption.

V. Conclusion

The implementation of a canonical Huffman encoder combined with a frequency sorter is effectively accomplished by this work, in conclusion. Assigning shorter binary codes to more often occurring symbols, the encoder efficiently compresses data by sorting strings according to ASCII values and applying the principles of Huffman encoding. Effective data compression is fundamental in many applications, including Internet of Things and other data-intensive fields, and our approach establishes the groundwork for it. The created encoder is ideally suited for contexts with limited resources and communication channels with restricted bandwidth as it shows the ability to reduce data significantly while maintaining data integrity. Moreover, the flexible architecture and modular design of the encoder enable its integration into many applications, providing adaptability and scalability in data compression solutions.

VI. Future Scope

Future research will concentrate on improving the canonical Huffman encoder's scalability and efficiency by investigating optimisations such hardware acceleration and parallel processing. Furthermore, in order to handle changing data compression requirements across a variety of applications and further increase compression ratios, research will explore the integration of sophisticated compression algorithms and adaptive approaches.

VII. References:

1. D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proceedings of the IRE, vol. 40, no. 9, 1952, doi: 10.1109/JRPROC.1952.273898.

2. S. J. Sarkar, N. K. Sarkar, and A. Banerjee, “A novel Huffman coding based approach to reduce the size of large data array,” in Proceedings of IEEE International Conference on Circuit, Power and Computing Technologies, ICCPCT 2016, 2016. doi: 10.1109/ICCPCT.2016.7530355.

3. Y. Liu and L. Luo, “Lossless compression of full-surface solar magnetic field image based on huffman coding,” in Proceedings of the 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference, ITNEC 2017, 2017. doi: 10.1109/ITNEC.2017.8284866.

4. Z. Shao et al., “A High-Throughput VLSI Architecture Design of Canonical Huffman Encoder,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 69, no. 1, 2016, doi: 10.1109/TCSII.2016.3091611.

5. S. R. Khaitu and S. P. Panday, “Canonical Huffman Coding for Image Compression,” in Proceedings on 2017 IEEE 3rd International Conference on Computing, Communication and Security, ICCCS 2017, 2017. doi:10.1109/CCCS.2017.8586816.

6. Xilinx, “Vivado HLS Optimization Methodology Guide 2017.1,” Xilinx.Com, vol. 1270, 2017.

7. Xilinx, “Designing Protocol Processing Systems with Vivado High-Level Synthesis,” Xapp1209, vol. 1209, 2014.

8. Xilinx, "UltraFast Design Methodology Guide for the Vivado Design Suite," Ug949, vol. 949, 2016.

9. A. Pal, Low-power VLSI circuits and systems. 2015. doi: 10.1007/978-81-322-1937-8.

10. J. Oh and M. Pedram, "Power reduction in microprocessor chips by gated clock routing," Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 1998, doi: 10.1109/aspdac.1998.669478.

11. J. Oh and M. Pedram, "Gated clock routing for low-power microprocessor design," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 6, 2001, doi: 10.1109/43.924825.

12. Shunji Funasaka, Koji Nakano, and Yasuaki Ito. 2017. Adaptive lossless data compression method optimized for GPU decompression. Concurrency and Computation: Practice and Experience 29, 24 (2017), e4283.

13. W. M. Holt. 2016. 1.1 Moore's law: A path going forward. In 2016 IEEE International Solid-State Circuits Conference (ISSCC). 8–13.

14. Y. Kim, I. Hong, and H. Yoo. 2015. 18.3 A 0.5V 54W ultra-low-power recognition processor with 93.5% accuracy geometric vocabulary tree and 47.5% database compression. In 2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers. 1–3.

15. J. Matai, J. Kim, and R. Kastner. 2014. Energy efficient canonical huffman encoding. In 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors. 202–209.

16. A. Mukherjee, N. Ranganathan, and M. Bassiouni. 1991. Efficient VLSI designs for data transformation of tree-based codes. IEEE Transactions on Circuits and Systems 38, 3 (1991), 306–314.

17. A. Ozsoy and M. Swany. 2011. CULZSS: LZSS Lossless Data Compression on CUDA. In 2011 IEEE International Conference on Cluster Computing. 403–411.

18. R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. 2012. Parallel lossless data compression on the GPU. In 2012 Innovative Parallel Computing (InPar). 1–9.

19. Seong Hwan Cho, T. Xanthopoulos, and A. P. Chandrakasan. 1999. A low power variable length decoder for MPEG-2 based on nonuniform fine-grain table partitioning. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 7, 2 (1999), 249–257.

20. André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In Proceedings of the 47th International Conference on Parallel Processing (Eugene, OR, USA) (ICPP 2018). Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3225058.3225076