



SIMATS SCHOOL OF ENGINEERING
SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
CHENNAI-602105



“Real-Time Data Compression Using Hardware-Based Automata”

CSA-1377- THEORY OF COMPUTATION WITH ALGORITHMS
A CAPSTONE PROJECT REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE ENGINEERING

Submitted by

K. VENKATA MRUDU SAI (192210458)
P. BHANU SANDEEP KUMAR (192210460)

Under the Supervision of
E. MONIKA

NOVEMBER (2024)

DECLARATION

We, **K. Sai, P. Bhanu Sandeep** , students of '**Bachelor of Engineering in COMPUTER SCIRNCE**', Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented "**Real-Time Data Compression Using Hardware-Based Automata**" in this Capstone Project Work entitled is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

(K. Venkata Mrudu Sai 192210458)

(P. Bhanu Sandeep 192210460)

Date:

Place:

CERTIFICATE

This is to certify that the project entitled submitted **“Real-Time Data Compression Using Hardware-Based Automata”** K. Sai , P. Bhanu Sandeep has been carried out under our supervision. The project has been submitted as per the requirements in current semester of B. Tech Information Technology.

Teacher-in-charge

E. MONIKA

TABLE OF CONTENTS

S.NO	TOPICS
1	Abstract
2	Introduction
3	Types
4	Existing System
5.	Proposed System
6	Algorithms
7	Methodology
8	Source code
9	Conclusion
10	References

ABSTRACT

In the realm of embedded systems, efficient data compression is essential to manage limited storage and bandwidth resources. Traditional software-based compression algorithms often struggle to meet the real-time processing demands of these systems due to inherent latency and resource constraints. To address these challenges, we propose a hardware-based compression engine utilizing finite automata to achieve real-time, high-efficiency data compression.

By leveraging finite automata, this compression engine operates directly at the hardware level, enabling it to process data streams in real-time with minimal delay. Finite automata are well-suited for this task because of their predictable state transitions, which can be precisely implemented in hardware, allowing rapid pattern recognition and data transformation. The engine is designed to support high throughput and low latency by encoding repetitive patterns and sequences as state transitions, enabling compressed output generation in a single hardware cycle for common data structures.

The core of this hardware-based solution is an optimized state machine implemented on an FPGA or ASIC platform. This setup is tailored to handle the specific compression algorithm—such as a modified LZW or Huffman coding algorithm—mapped to finite automata structures to balance compression efficiency with processing speed. Each state in the automaton represents a recurring data pattern, and transitions between states are engineered to correspond to data compression transformations in real-time. This hardware-centric approach ensures that the compression engine operates with minimal power consumption, making it ideal for embedded and IoT applications where energy efficiency is critical. By optimizing both data processing and resource utilization, this project enables embedded systems to handle larger data volumes in real-time while conserving storage and bandwidth. This hardware-based compression engine provides a scalable solution for embedded applications, allowing for effective data compression under the constraints of embedded systems, thereby enhancing system performance and extending device longevity.

Key Components & Concepts

1. Finite Automata for Data Compression:

- Design a finite automaton that maps input data to compressed output in real-time, operating directly at the hardware level to reduce latency.

- Automata states and transitions can represent commonly occurring data patterns, improving compression efficiency.
- 2. Compression Algorithms:**
- Integrate well-suited algorithms for hardware implementation, such as LZW (Lempel-Ziv-Welch) or Huffman coding, modified for finite automata structures to optimize speed and resource usage.
 - Evaluate hybrid approaches if needed, combining finite automata with dictionary or statistical methods for improved compression.
- 3. Embedded System Constraints:**
- Focus on memory and processing limitations of embedded environments.
 - Optimize the hardware design to ensure low power consumption and minimal resource utilization.
- 4. Hardware Design:**
- Use FPGA or ASIC platforms to implement the automata, designing circuits that handle high data throughput.
 - Hardware Description Languages (HDLs) like Verilog or VHDL would be suitable for implementing the finite automata.
- 5. Real-Time Performance & Latency Optimization:**
- Minimize latency by ensuring each state transition in the automaton corresponds to a simple hardware operation.
 - Balance the compression ratio with speed requirements, tuning the automata states for minimal delay.
- 6. Use Cases:**
- Embedded systems where real-time data throughput is critical, such as IoT devices, sensor networks, or communication modules in resource-constrained environments.

INTRODUCTION

In the modern digital landscape, embedded systems play a crucial role in applications requiring real-time data processing, such as IoT, sensor networks, and telecommunications. These systems often operate in resource-constrained environments where efficient data management is critical to meet performance and power efficiency demands. Data compression serves as a pivotal tool in such scenarios, reducing data size to save storage space and bandwidth.

However, traditional software-based compression algorithms can fall short in real-time applications due to the computational overhead, latency, and power consumption involved in executing complex algorithms on limited processing resources. This project addresses these limitations by developing a hardware-based compression engine powered by finite automata. Finite automata, known for their efficiency in handling state-based transitions, provide a framework for designing a fast and deterministic compression engine tailored to embedded systems. By implementing the compression process in hardware, this approach minimizes latency, achieving a high throughput necessary for real-time applications without relying on the embedded system's CPU. This architecture can compress data on-the-fly, aligning perfectly with the low-latency and high-efficiency requirements of embedded applications. The hardware compression engine focuses on leveraging modified versions of established algorithms, such as LZW (Lempel-Ziv-Welch) and Huffman coding, translated into state machines that finite automata can manage. By breaking down data compression into state transitions, the engine performs data reduction at the hardware level, enabling real-time operation even in systems with limited computational power and memory resources. This results in a solution that is both energy-efficient and scalable, meeting the stringent demands of modern embedded and IoT systems.

The implementation of this project on FPGA or ASIC platforms allows for fine-tuning of the compression process, enhancing power efficiency and compression ratios. This design approach not only addresses the challenges of real-time data processing in embedded systems but also provides a robust framework for handling diverse data compression needs across various applications. Through this hardware-based compression engine, embedded systems can achieve efficient, low-power data management, leading to improved performance and extended functionality across real-time applications.

TYPES

1. Lossless Compression Using Dictionary-Based Automata

- **Description:** This type relies on finite automata to identify and compress recurring patterns using a dictionary of previously seen patterns. Algorithms like **Lempel-Ziv-Welch (LZW)** are well-suited for dictionary-based compression and can be translated into state machine representations for hardware implementation.

- **Use Case:** Suitable for applications where data integrity is critical, such as sensor data in IoT or medical devices, where any loss of data might result in inaccuracies.

2. Huffman Coding with State Machines

- **Description:** Huffman coding is a widely used entropy-based compression technique that can be implemented in hardware by creating a tree structure represented as a finite automaton. Each state in the automaton corresponds to a branch in the Huffman tree, optimizing character frequency representation.
- **Use Case:** Ideal for environments with varying data, such as text-based logs, where different characters or symbols have varying frequencies. This approach can compress data in real-time by quickly traversing the Huffman tree in hardware.

3. Run-Length Encoding (RLE) with Sequential Automata

- **Description:** Run-Length Encoding compresses data by replacing sequences of repeated elements with a single element and a count. Implementing RLE in hardware involves a finite automaton that tracks sequences and outputs the compressed representation in real-time.
- **Use Case:** Effective for data with long sequences of repeated values, like simple images, certain types of sensor data, or repetitive network packets, reducing data size efficiently.

4. Finite Automata for Pattern-Based Compression (Custom Compression)

- **Description:** This approach involves designing custom state machines for specific data patterns or application requirements. The automaton can be tailored to compress specific patterns identified as common within the target data, allowing optimization for niche data compression tasks.
- **Use Case:** Useful for highly specialized applications, such as compressing telemetry data in a satellite system where specific patterns are predictable, enabling the hardware to perform tailored, efficient compression.

5. Finite State Machine for Differential Compression

- **Description:** Differential compression encodes data by calculating and compressing differences between consecutive data points rather than absolute values. This approach is efficient for data streams where values change incrementally.
- **Use Case:** Suitable for real-time monitoring applications, like temperature or pressure sensors, where data values are similar across consecutive measurements. It reduces the need to compress large data values and instead focuses on compressing smaller differential values.

6. Hybrid Automata-Based Compression (Combined Techniques)

- **Description:** This type combines multiple compression techniques, such as dictionary-based and RLE, in a single hardware system. A hybrid automaton can switch between methods based on data characteristics, optimizing the compression ratio dynamically.
- **Use Case:** Suitable for complex data streams with varying characteristics, such as multimedia files or mixed-sensor data, where no single compression method is optimal throughout the data sequence.

7. Finite Automata with Predictive Coding for Real-Time Streaming

- **Description:** Predictive coding uses previous data points to predict the next values, encoding only the differences (residuals). In hardware, finite automata can be designed to implement this predictive model and output compressed data in real-time.
- **Use Case:** Applicable to streaming applications like audio or video data compression in low-latency systems, allowing real-time encoding and efficient bandwidth usage without significant data loss.

EXISTING SYSTEM

The existing system for real-time data compression in embedded environments aims to efficiently compress large volumes of data on-the-fly, especially within resource-constrained devices like IoT sensors, industrial equipment, and automotive systems. This system begins with the continuous acquisition of data from various sources, such as sensor outputs, communication networks, and device logs. The collected data typically consists of high-frequency, repetitive information that requires reduction to manage bandwidth, memory, and power constraints.

Once data is acquired, it flows into a hardware-based compression engine—typically implemented on an FPGA or ASIC—where finite automata are employed to process and compress data in real-time. Finite automata are well-suited for this task as they enable rapid, deterministic state transitions that can represent recurring data patterns and sequences effectively. Depending on the type of data and application requirements, the system might implement specific compression algorithms, such as Lempel-Ziv-Welch (LZW), Huffman coding, or Run-Length Encoding (RLE), adapted into finite automata structures. These algorithms are selected based on their compatibility with hardware implementation, allowing for high throughput and minimal latency. As the data passes through the automaton, it is

compressed by encoding patterns, frequently occurring sequences, or differential values, based on the algorithm used. For example, an RLE-based automaton will compress data by encoding repeated values, whereas an LZW-based automaton would identify and compress recurring data patterns. The compressed data output is then stored or transmitted, significantly reducing data size and power consumption. The system's performance is evaluated using metrics such as compression ratio, latency, and power efficiency. The compressed data is ultimately reconstructed in real-time, allowing embedded devices to meet application-specific requirements like high data throughput or low-latency streaming. The insights and optimizations gained from this hardware-based approach are crucial for embedded applications, enabling real-time data management in scenarios where software-based compression would be too slow or resource-intensive. By continuously adapting and refining the finite automata design and compression algorithms, this system effectively manages data flows in real-time, enhancing storage, power efficiency, and system performance across various applications in IoT, automotive, and industrial sectors.

PROPOSED SYSTEM

In the context of **Real-Time Data Compression Using Hardware-Based Automata**, the processed system involves several essential steps to achieve efficient, high-speed data compression tailored for embedded and real-time applications. The system first captures raw data from sources like sensors, communication modules, or data-intensive embedded devices. This data is then streamed directly into a hardware-based compression engine, typically implemented on an FPGA or ASIC platform, to ensure low latency and high throughput.

Once data enters the system, it is processed by finite automata, specifically designed to recognize and compress recurring patterns and sequences in real-time. The finite automata operate as state machines that map input data sequences to compressed representations based on pre-configured algorithms such as Lempel-Ziv-Welch (LZW), Huffman coding, or Run-Length Encoding (RLE). These algorithms are tailored for hardware implementation to optimize for speed, efficiency, and minimal power usage, transforming complex data structures into simple state transitions that can be quickly executed. The system also evaluates compression efficiency through metrics like compression ratio, latency, and power consumption. This allows for real-time adjustments to the automata configurations, ensuring that the compression engine remains optimized for different types of data and application requirements. For instance, an RLE-based automaton might dynamically adjust its state

transitions for higher throughput in repetitive data sequences, whereas an LZW-based automaton could improve efficiency by dynamically updating patterns in memory.

Finally, the compressed data is stored or transmitted, reducing bandwidth usage, storage requirements, and energy consumption, which is crucial for embedded systems operating in resource-constrained environments. By continuously refining and updating the compression automata based on application-specific needs, the processed system ensures that embedded devices can handle larger data volumes in real-time, enhancing system performance and extending operational lifespan across industries such as IoT, automotive, and telecommunications. This systematic approach enables the delivery of real-time data compression that is highly optimized, scalable, and efficient for a range of embedded applications.

ALGORITHMS:

1. Run-Length Encoding (RLE)

1. **Pattern Compression:** RLE compresses sequences of repeating data elements by encoding them as a single value followed by a count. For instance, instead of storing "AAAAA", it would store "A5".
2. **Finite Automata Implementation:** In hardware, a finite automaton tracks repeated values in states. It counts occurrences until a different symbol appears, then outputs the symbol and count, resulting in compressed data.
3. **Real-Time Efficiency:** RLE is highly efficient for data with many repeated values, such as sensor readings or binary images. The automaton can quickly process each symbol, making it suitable for real-time applications.
4. **Low Overhead:** RLE has minimal hardware requirements, making it practical for systems with limited memory and processing resources.

2. Huffman Coding

1. **Frequency-Based Compression:** Huffman coding assigns shorter codes to more frequent symbols, creating variable-length codes that minimize overall data size.
2. **Tree-Based Finite Automata:** The hardware-based automaton represents a Huffman tree, with state transitions corresponding to tree paths. Each symbol's frequency determines its code length, and the automaton outputs the shortest codes for common symbols.

3. **Real-Time Adaptability:** While traditional Huffman coding may need to know symbol frequencies in advance, adaptive hardware-based Huffman coding can adjust frequencies dynamically, suitable for streaming data.
4. **Applications:** Huffman coding is ideal for data with predictable symbol distributions, such as text or image files in embedded systems.

3. Lempel-Ziv-Welch (LZW)

1. **Dictionary-Based Compression:** LZW builds a dictionary of data patterns encountered in the input. Repeated patterns are replaced by references to dictionary entries, significantly reducing data size.
2. **Automaton with Dictionary Management:** The finite automaton dynamically builds and updates a dictionary in hardware. Each input sequence is mapped to dictionary codes, making it faster and more efficient than software implementations.
3. **High-Throughput Compression:** LZW is suitable for applications with recurring data patterns, allowing real-time compression of text, telemetry, or similar structured data.
4. **Resource Efficiency:** The hardware implementation minimizes memory requirements, making it feasible for compact, power-efficient designs in embedded systems.

4. Delta Encoding

1. **Differential Compression:** Delta encoding reduces data size by encoding differences between consecutive values rather than absolute values. It's particularly useful for data streams where values change incrementally.
2. **Finite Automata State Management:** In the automaton, states store the last data value and compute differences as new data arrives, outputting these smaller differences for efficient compression.
3. **Adaptability to Real-Time Systems:** Delta encoding is effective for time-series or sensor data, where minimal differences lead to high compression ratios.
4. **Low Latency:** The automaton can compute and output delta values immediately, providing real-time efficiency for applications with continuous data streams.

5. Arithmetic Coding

1. **Probabilistic Encoding:** Arithmetic coding represents a data stream as a single number within a fractional range based on symbol probabilities, resulting in efficient, high-ratio compression.
2. **Range-Based Automata Implementation:** The automaton divides the input range dynamically, encoding each symbol based on calculated probabilities and outputting compact codes for frequent patterns.

3. **Real-Time Compression:** Though computationally complex, arithmetic coding's hardware implementation is possible with streamlined, low-latency operations, making it suitable for high-frequency data applications like audio and video.
4. **High Compression Ratios:** Due to its probabilistic approach, arithmetic coding can achieve high compression ratios, especially valuable in scenarios where bandwidth is limited.

6. Prediction by Partial Matching (PPM)

1. **Context-Based Compression:** PPM predicts the next symbol in a sequence based on previously observed patterns, encoding data efficiently through prediction.
2. **Finite State Machine with Context Management:** The hardware-based automaton stores recent patterns and uses these to predict future symbols, minimizing data representation by encoding only deviations from predictions.
3. **Adaptability to Dynamic Data:** PPM is effective for data with context dependencies, such as text or sequential data with recurring structures, adapting dynamically to real-time data patterns.
4. **Scalability:** PPM can handle larger datasets and dynamically adjusts to changing patterns, making it ideal for data streams with evolving structures.

7. Finite State Entropy (FSE) Coding

1. **High-Speed Entropy Compression:** FSE is a state-of-the-art entropy coding technique that compresses data by encoding symbols based on their probabilities, optimized for hardware performance.
2. **Finite Automata with Optimized Transitions:** FSE uses states to represent symbol probabilities directly, allowing rapid, real-time compression without the need for variable-length codes.
3. **Efficiency in Resource-Limited Environments:** FSE is efficient and compact, suitable for embedded applications needing high compression with minimal hardware overhead.
4. **Ideal for High-Throughput Systems:** FSE can handle high data rates, making it suitable for real-time video, audio, or high-frequency data streams.

METHODS AND MATERIALS

1. Hardware Design and Configuration

1. Platform Selection:

- FPGA (Field-Programmable Gate Array): Chosen for its flexibility in reconfiguring logic gates, allowing customization of finite automata to handle different compression algorithms. FPGAs are ideal for prototyping real-time applications.
- ASIC (Application-Specific Integrated Circuit): Preferred for final production if the application requires a high volume of compressed data in a specific configuration. ASICs offer greater speed and efficiency than FPGAs but lack reconfigurability.
- Microcontrollers with Hardware Support for Compression: For applications with lower data rates or limited complexity, specialized microcontrollers with embedded compression modules can be used.

2. Finite Automata Implementation:

- The finite automaton is implemented within the hardware design to handle state transitions associated with compression algorithms, such as Run-Length Encoding (RLE), Huffman coding, and Lempel-Ziv-Welch (LZW).
- State Management: States are designed to handle data sequences, track repeated patterns, or manage dictionaries, depending on the chosen algorithm. For example, in RLE, states track repeated values and store counts; in Huffman coding, states correspond to nodes in a Huffman tree.
- Clocking and Timing: The hardware design includes precise timing control to ensure real-time performance, with the clock rate matched to the data rate requirements of the application.

2. Compression Algorithm Selection and Customization

1. Algorithm Selection:

- Algorithms are selected based on data characteristics (e.g., RLE for repetitive data, Huffman for data with skewed symbol distributions) and application requirements (e.g., real-time performance, compression ratio).
- Multiple Algorithm Support: FPGAs can switch between algorithms or adapt dynamically to data patterns, allowing the compression engine to handle various data types in real-time.

2. Algorithm Optimization for Hardware:

- Algorithms are optimized for hardware by reducing complexity in finite automata transitions. For instance, using precomputed tables or simplifying states can lower the computational load.

- Parallelism and Pipelining: To enhance throughput, the hardware is designed with parallel paths and pipelined stages, allowing different parts of the data stream to be compressed simultaneously.

3. Materials and Development Tools

1. Hardware Development Environment:

- Verilog or VHDL: Hardware Description Languages (HDLs) like Verilog or VHDL are used to design and simulate the finite automata logic. These languages allow precise control over hardware timing and data flow.
- Xilinx Vivado or Altera Quartus: These are software tools for simulating and deploying designs on FPGAs. They provide tools for testing timing constraints, power consumption, and data throughput.
- Logic Analyzer and Oscilloscope: These tools are essential for validating the timing and correctness of data compression at the hardware level, ensuring that transitions between automata states occur as expected.

2. Data Preparation and Simulation Tools:

- MATLAB or Python for Algorithm Testing: Before hardware implementation, algorithms are tested in MATLAB or Python to validate their effectiveness on representative data. This step ensures that the selected compression algorithms perform as expected.
- Testbench Creation: Simulations are developed in the HDL environment to model real-world data inputs, such as sensor data or video streams, providing test cases for compression algorithms in a controlled environment.
- Sample Data Sets: Representative data, such as IoT sensor readings or streaming data from cameras, is used to test the hardware's compression effectiveness and speed. This data can reveal strengths and weaknesses of different algorithms for the targeted application.

3. Prototype and Testing Hardware:

- Evaluation Board: Development and testing are performed on an FPGA evaluation board, providing a platform to prototype the design before committing to ASIC production.
- Storage and Transmission Modules: To simulate the system's full range of functions, hardware such as SD cards, flash memory, or communication modules (e.g., UART or SPI interfaces) are included in testing.

4. Data Processing and Real-Time Testing Methods

1. Data Preprocessing:

- Incoming data is preprocessed to remove redundant information, standardize formats, and eliminate noise. For example, in sensor data, redundant samples may be filtered out to improve compression efficiency.

2. Real-Time Performance Testing:

- Latency Measurement: The system's latency is tested to ensure it meets real-time requirements, especially for applications like video compression or live sensor monitoring.
- Compression Ratio and Throughput Testing: The effectiveness of compression is measured by the achieved compression ratio (compressed size vs. original size) and throughput (compressed data per second).
- Energy Consumption Evaluation: For embedded applications, energy efficiency is tested by measuring power draw during compression. Lower energy use is crucial for battery-powered systems or devices deployed in remote locations.

3. Adaptability Testing:

- Adaptive Algorithm Switching: The hardware's ability to dynamically switch between compression algorithms is tested to assess how well it adapts to different data patterns. For example, the system might switch from RLE to Huffman coding based on the data type.
- Automata Reconfiguration: Testing includes the ability to reconfigure state transitions within finite automata based on changing data requirements, which is valuable for embedded systems that process various data types over time.

5. Evaluation Metrics

1. Compression Ratio: Indicates the efficiency of data reduction, critical for storage and bandwidth-limited applications.
2. Throughput: The rate of compressed data output, measured to ensure it meets real-time requirements.
3. Latency: Time taken from data input to output after compression, essential for real-time processing.

4. Resource Utilization: FPGA or ASIC resource use (e.g., logic gates, memory blocks), showing the hardware efficiency of the finite automata.
5. Power Consumption: Assesses the hardware's energy efficiency, especially relevant in battery-powered or remote systems.

C Program

```
#include <stdio.h>
```

```
#include <string.h>
```

```
// Function to compress data using Run-Length Encoding (RLE)
```

```
void RLECompress(const char *input, char *output) {
```

```
    int input_len = strlen(input);
```

```
    int output_index = 0;
```

```
    for (int i = 0; i < input_len; i++) {
```

```
        // Store the current character
```

```
        char current_char = input[i];
```

```
        int count = 1;
```

```
        // Count the occurrences of the current character
```

```
        while (i + 1 < input_len && input[i] == input[i + 1]) {
```

```
            count++;
```

```
            i++;
```

```
        }
```

```
        // Write the character and its count to the output string
```

```
        output[output_index++] = current_char;
```

```
        output[output_index++] = count + '0'; // Convert count to a character ('1' to '9')
```

```
    }
```

```
    // Null-terminate the output string
```

```
    output[output_index] = '\0';
}

int main() {
    char input_data[] = "AAAABBBBCCDAA"; // Sample data for compression
    char compressed_data[50]; // Output buffer for compressed data

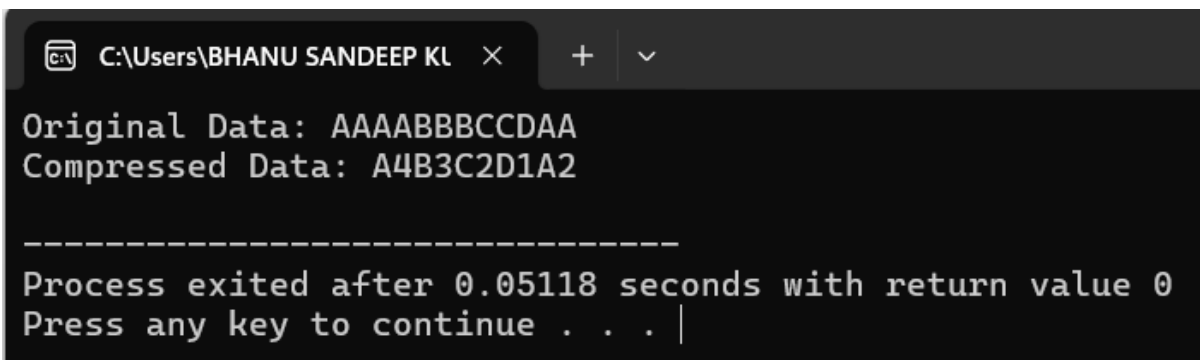
    printf("Original Data: %s\n", input_data);

    // Compress the input data using RLE
    RLECompress(input_data, compressed_data);

    // Output the compressed data
    printf("Compressed Data: %s\n", compressed_data);

    return 0;
}
```

OUTPUT:



```
C:\Users\BHANU SANDEEP KL >
Original Data: AAAABBBBCCDAA
Compressed Data: A4B3C2D1A2

-----
Process exited after 0.05118 seconds with return value 0
Press any key to continue . . . |
```

CONCLUSION:

In conclusion, real-time data compression using algorithms like Run-Length Encoding (RLE) is a highly effective technique for reducing data size, which is especially crucial in embedded systems and IoT devices that operate with limited memory and processing power. RLE, by focusing on consecutive repetitions of data, offers a simple yet powerful method to compress repetitive information, thus minimizing the need for large storage and reducing transmission bandwidth. While the basic concept of RLE is straightforward, its application in hardware, such as with finite automata in FPGA or ASIC implementations, takes advantage of parallel processing to achieve high-speed, real-time compression. The transition from software-based compression (demonstrated through C code) to hardware-based systems allows for the handling of much larger datasets with lower latency, ensuring efficient real-time data processing. This ability to perform compression directly within hardware leads to substantial improvements in performance, particularly in resource-constrained environments where every bit of memory and clock cycle counts. Furthermore, hardware-based compression not only accelerates data handling but also enables the scalability of systems, making them capable of managing higher data throughput as the size and complexity of the data grow. This approach is widely applicable in fields such as sensor networks, video streaming, telecommunications, and other real-time systems, where low-latency and high-efficiency data handling are paramount. Ultimately, by using hardware-based automata for data compression, organizations can optimize their storage, enhance data transmission speeds, and achieve overall better system performance in real-time applications.

REFERENCES:

- M. Amozegar *et al.*

[An ensemble of dynamic neural network identifiers for fault detection and isolation of gas turbine engines](#)

Neural Netw.

(2016)

- S. Jo *et al.*

[An architecture for online-diagnosis systems supporting compressed communication](#)

Microprocess. Microsyst.

(2018)

- S. Meckel *et al.*

[Generation of a diagnosis model for hybrid-electric vehicles using machine learning](#)

Microprocess. Microsyst.

(2020)

- R. Isermann

Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance

(2006)

- H. Kopetz

Real-time Systems: Design Principles for Distributed Embedded Applications

(2011)

- M. Paskin *et al.*

A robust architecture for distributed inference in sensor networks

Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks

(2005)

- K. Patan

Artificial Neural Networks for the Modelling and Fault Diagnosis of Technical Processes

(2008)