Internship - 4th Year

LZW Compression on FPGA

Student :

Sélim GHARZOUZI

Internship Supervisor :

Ing. Martin ROZKOVEC, Ph.D.

Academic Supervisor :

Antoine GOULLET

August 29, 2025

Abstract

This internship was carried out at the Technical University of Liberec (TUL), Faculty of Mechatronics, from June 9th to October 2nd, under the supervision of Ing. Martin Rozkovec. The project focused on the implementation of the Lempel–Ziv–Welch (LZW) lossless compression algorithm on Field Programmable Gate Arrays (FPGAs). The work was divided into several stages, starting with a software implementation of both compression and decompression in C, followed by the design of a hardware implementation using High-Level Synthesis (HLS) in Vitis Unified IDE. The target platform was primarily the ZedBoard (Xilinx Zynq-7000 SoC), programmed in bare-metal, and later extended to the Kria KV260 Vision AI board.

The main objectives were to evaluate the performance differences between software and hardware implementations and to explore potential optimizations of the compression algorithm for FPGA architectures. Special attention was given to parallelization strategies, including running multiple instances of LZW cores and testing different approaches to concurrent execution. The project involved the use of Vivado for hardware design and integration, AXI4 interfaces for communication, and user-level applications developed in C/C++ for performance testing.

Résumé

Ce stage a été réalisé à l'Université Technique de Liberec (TUL), Faculté de Mécatronique, du 9 juin au 2 octobre, sous la supervision de l'Ing. Martin Rozkovec. Le projet portait sur l'implémentation de l'algorithme de compression sans perte Lempel–Ziv–Welch (LZW) sur FPGA. Le travail a été divisé en plusieurs étapes, en commençant par une implémentation logicielle de la compression et de la décompression en C, suivie de la conception d'une implémentation matérielle à l'aide de la synthèse de haut niveau (HLS) dans l'environnement Vitis Unified IDE. La plateforme cible était principalement la ZedBoard (SoC Xilinx Zynq-7000), programmée en mode bare-metal, puis étendue à la carte Kria KV260 Vision AI.

Les principaux objectifs étaient d'évaluer les différences de performance entre les implémentations logicielles et matérielles, ainsi que d'explorer les optimisations possibles de l'algorithme de compression pour les architectures FPGA. Une attention particulière a été portée aux stratégies de parallélisation, incluant l'exécution simultanée de plusieurs cœurs LZW et l'étude de différentes approches d'exécution concurrente. Le projet a impliqué l'utilisation de Vivado pour la conception et l'intégration matérielles, des interfaces AXI4 pour la communication, ainsi que des applications utilisateur développées en C/C++ pour les tests de performance.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Ing. Martin Rozkovec, for his continuous support, guidance, and willingness to share his knowledge throughout the course of this internship. His advice and encouragement were invaluable in helping me learn and progress during the project.

I would also like to thank Ing. Simona Kuncová, the Coordinator of International Cooperation, for making this international internship possible and for her support in ensuring a smooth experience.

Finally, I would like to thank the Technical University of Liberec (TUL) for giving me the opportunity to carry out this internship and for providing the resources and environment necessary to complete this work.

Contents

List of Figures

List of Tables

Introduction

The fourth-year internship represents a first professional immersion into the engineering field, with the objective of bridging theoretical knowledge and practical application. My internship was conducted at the Technical University of Liberec (TUL) in the Czech Republic, within the Faculty of Mechatronics, from June to October 2025. The project was supervised by Ing. Martin Rozkovec and was dedicated to the study and implementation of the Lempel–Ziv–Welch (LZW) compression algorithm on FPGA platforms.

Data compression is a key challenge in modern information and communication systems, as it allows for reduced storage requirements and faster data transfer while maintaining data integrity in the case of lossless algorithms, like Lempel-Ziv-Welch.

The aim of this internship was not only to implement LZW but also to investigate its behavior across different environments: in pure software, in hardware using FPGA IP cores, and under parallelized configurations. This objective aligned with the broader goal of understanding hardware/software co-design and evaluating the trade-offs between flexibility, performance, and resource constraints.

This introduction concludes by presenting the structure of the report. The chapters that follow begin with an overview of the host institution, followed by the problem statement that frames the technical challenge. The main body details the carried-out work and methodologies, before moving into an analysis of the results and a discussion of project management aspects. Finally, the report closes with both technical conclusions and personal reflections drawn from this internship experience.

Chapter 1

Presentation of Technical University of Liberec

The Technical University of Liberec (TUL) is a public university located in the city of Liberec, in the northern region of the Czech Republic. Established in 1953, the university has grown into a recognized institution of higher education and research, particularly strong in the fields of engineering, textile sciences, and applied sciences. TUL is organized into several faculties, including the Faculty of Mechanical Engineering, the Faculty of Textile Engineering, the Faculty of Economics, the Faculty of Arts and Architecture, and the Faculty of Mechatronics, Informatics and Interdisciplinary Studies, among others.

The Faculty of Mechatronics, where this internship was carried out, is focused on interdisciplinary education and research, combining mechanics, electronics, computer science, and control engineering. The faculty's research activities span areas such as automation, robotics, embedded systems, and advanced digital technologies. This environment, with its strong expertise in digital systems, embedded design, and interdisciplinary research, provided a particularly suitable framework for my internship project on FPGA-based data compression, where both hardware and software perspectives needed to be combined.

TUL has a strong tradition of collaboration with both industrial partners and international institutions, offering students opportunities to apply theoretical knowledge to practical problems. Its facilities include modern laboratories and specialized research centers, which support experimental and applied research. Within this setting, I was able to conduct my internship and benefit from the expertise of my supervisor and the resources provided by the university.

Overall, the Technical University of Liberec offered an environment that was both academically rigorous and professionally relevant, making it an excellent host institution for the objectives of my internship.

Chapter 2

Context of the Internship

The objective of this internship was to study the LZW compression algorithm and to implement it on an FPGA using high-level synthesis (HLS) tools. The main focus was not on delivering an industrial-grade product, but rather on providing a hands-on learning opportunity. Through this project, the intention was to explore how a well-known algorithm can be translated from software into hardware, while gaining familiarity with FPGA design flows and toolchains such as Vivado and Vitis.

The broader motivation behind the internship was educational. By choosing a classic algorithm like LZW, which celebrated its 50th anniversary in 2024, the project provided a concrete and historically significant case study for understanding both compression techniques and hardware/software co-design.

The expected outcome was a complete implementation of the LZW compression algorithm, first in software and then in hardware, together with an analysis of the results. This included measuring performance, resource utilization, and understanding the trade-offs between the two implementations. Beyond the technical deliverables, the internship aimed to provide valuable learning outcomes: developing C/C++ programming skills and skills in HLS, understanding FPGA architectures, and gaining experience in hardware/software co-design.

Chapter 3

Study of the LZW Algorithm

In this chapter, I am going to cover the first stage of the internship, which consisted of a detailed study of the Lempel-Ziv-Welch (LZW) Algorithm.

3.1 History of the Algorithm

The origins of the LZW algorithm are closely tied to the broader development of dictionary-based compression methods. In the late 1970s, Abraham Lempel and Jacob Ziv introduced two seminal algorithms, LZ77 (1977) and LZ78 (1978), which pioneered adaptive dictionary techniques for data compression. These methods demonstrated that it was possible to replace repeated patterns in data with shorter references, thereby achieving significant reductions in size without losing information [1].

Building upon LZ78, Terry Welch introduced the LZW algorithm in 1984. Welch's refinement removed the need to transmit dictionary entries explicitly by ensuring that both compressor and decompressor could build identical dictionaries on the fly. This improvement simplified implementations and increased performance, making the technique suitable for practical, high-speed applications.

LZW rapidly gained popularity in commercial systems. It was adopted in hardware-based modems and storage devices where real-time compression was essential. In the software domain, it became the backbone of widely used file formats and standards, including GIF images, TIFF graphics, and PostScript printers. Its combination of efficiency and relative simplicity ensured broad adoption across the computing industry during the 1980s and 1990s.

Although more advanced algorithms such as DEFLATE, LZMA, and modern entropy coders have since surpassed it in compression ratios, LZW remains historically important as one of the first practical dictionary-based methods to achieve global adoption. Its conceptual clarity and deterministic dictionary construction also make it an excellent candidate for hardware implementation, as explored in this internship.

3.2 Algorithmic Principles

At its core, the LZW algorithm is a lossless, dictionary-based compression method [1]. It builds a mapping between sequences of input symbols and numerical codes, replacing repeated patterns with shorter representations. The key idea is that both the encoder and the decoder can construct identical dictionaries dynamically, without transmitting the dictionary itself.

A dictionary-based compression algorithm works by identifying repeated sequences in the input data and storing them in a growing table, or dictionary. Instead of writing the same sequence of symbols multiple times, the encoder outputs a reference (an index or code) to the dictionary entry. The decoder, which reconstructs the same dictionary while reading the codes, can then expand these references back to the original sequences. This approach reduces redundancy in the data, achieves compression without loss of information, and provides a flexible mechanism that adapts to the actual content being processed.

3.2.1 Compression Process

At the beginning, the dictionary is initialized with all possible values of a byte, i.e., the 256 ASCII single characters, and each symbol is associated with a fixed code. The algorithm then reads the input stream symbol by symbol while maintaining a current sequence. It extends the current sequence by appending the next symbol and checks if the resulting sequence exists in the dictionary. If the extended sequence exists, the algorithm continues building it. However, if the extended sequence does not exist, the algorithm outputs the code corresponding to the current sequence, adds the new extended sequence to the dictionary with a newly assigned code, and resets the current sequence to the last symbol read. This process is repeated until the input is fully processed, at which point the code for the final sequence is also output.

Figure 3.2.1 illustrates the steps of the compression process in a flow chart. K is a variable representing the next character, and ω is a variable containing the current sequence of symbols.

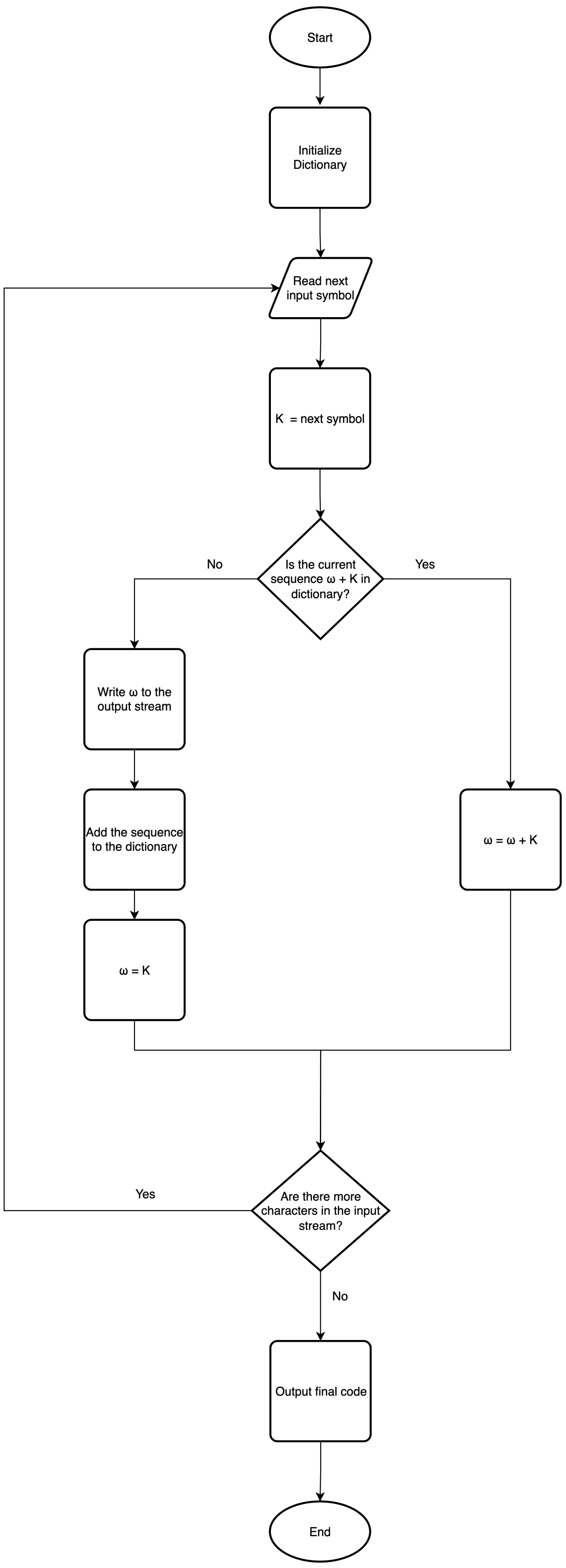

Start
Initialize Dictionary
Read next input symbol
K = next symbol
Is the current sequence ω + K in dictionary?
No
Yes
Write ω to the output stream
Add the sequence to the dictionary
ω = K
ω = ω + K
Are there more characters in the input stream?
Yes
No
Output final code
End

Figure 4.2.1: Compression Flow Chart.

3.2.2 Demonstration of the compression process

To better illustrate the principles of LZW compression, let us consider a simple example. Let's take the following input string:

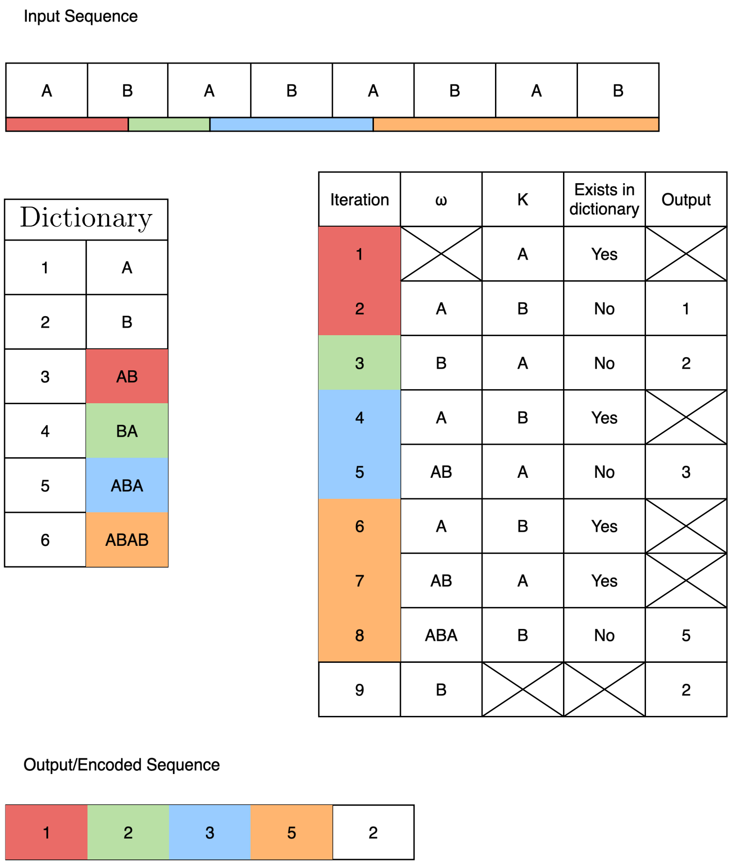
`ABABABAB`

This input string is a special case that was chosen because of the exception which arises in decompression. It's mandatory to present this exception case because, even though it is an uncommon case the algorithm could face, it is extremely frequent in streams with high repetition. Thus, the experimented string will be ABABABAB, i.e. 4 times AB.

FIGURE 3.2.2 presents a demonstration of the compression process. For simplicity, the dictionary is initialized with only two possible characters and their codes: $1 = \text{A}$ and $2 = \text{B}$. The left column of the dictionary shows the fixed code value, and the right column the associated symbol. The algorithm then proceeds as follows:

1. The algorithm starts by reading the first character of the input stream, here `A`, and assigns it to the next character variable K.
2. Since this character already exists in the dictionary, the algorithm sets the current sequence variable ω to K, then looks ahead to the next character and stores it again in K. In this case, $\text{K} = $ `B`.
3. The algorithm checks if the concatenated sequence $\omega + \text{K}$ (i.e., `AB`) exists in the dictionary.
4. Since `AB` is not yet in the dictionary, the algorithm outputs the code corresponding to ω (which is `A` $= 1$).
5. A new entry for `AB` is then added to the dictionary with the next available code (3).
6. The current string ω is reset to the value of K, i.e., $\omega =$ `B`, and the process repeats.

These steps are repeated until the input stream is fully consumed or an End-Of-File symbol is encountered. At that point, the code corresponding to the last current string ω is also written to the output.



Input Sequence

A	B	A	B	A	B	A	B

Dictionary	
1	A
2	B
3	🟥 AB
4	🟩 BA
5	🟦 ABA
6	🟧 ABAB

Iteration	ω	K	Exists in dictionary	Output
🟥 1	☒	A	Yes	☒
🟥 2	A	B	No	1
🟩 3	B	A	No	2
🟦 4	A	B	Yes	☒
🟦 5	AB	A	No	3
🟧 6	A	B	Yes	☒
🟧 7	AB	A	Yes	☒
🟧 8	ABA	B	No	5
9	B	☒	☒	2

Output/Encoded Sequence

🟥 1	🟩 2	🟦 3	🟧 5	2

Figure 3.2.2: An example to demonstrate the compression algorithm.

3.2.3 Decompression Process

The decompression algorithm reconstructs the original sequence from the compressed codes by rebuilding the dictionary in the same way as the compressor. The only information required is the encoded stream. At the beginning, the dictionary is initialized with all possible values of a byte, i.e., the 256 ASCII single characters, each associated with its fixed code.

The decoder then reads the compressed stream code by code, outputs the corresponding sequence, and updates the dictionary accordingly. Since the dictionary evolves deterministically, the decompressor always remains synchronized with the compressor without requiring explicit transmission of dictionary entries.

The process begins by outputting the first code directly, as it represents a single character already present in the dictionary. For each subsequent code, the decoder outputs the corresponding sequence from the dictionary. To ensure the dictionary is updated in the same way as during compression, a new entry is added at each step: the concatenation of the previous decoded sequence (ω) and the first character of the current sequence (K).

There is, however, a well-known exception case [1]. Because decompression always lags one step behind compression, it may encounter a codeword that has not yet been added to the dictionary. This is known as the $K\omega K\omega K$ case [1]. When this occurs, the decoder resolves it by outputting the previous sequence ω concatenated with its own first character. This ensures correct reconstruction of the original data and maintains synchronization with the encoder.

The demonstration of the compression process will lead to this exception during decompression. After presenting the flow chart, the occurrence of this special case will be explained along with the method used to handle it.

Figure 3.2.3 illustrates the steps of the decompression process in a flow chart. In this case, ω represents the previous sequence, while K denotes the current sequence.

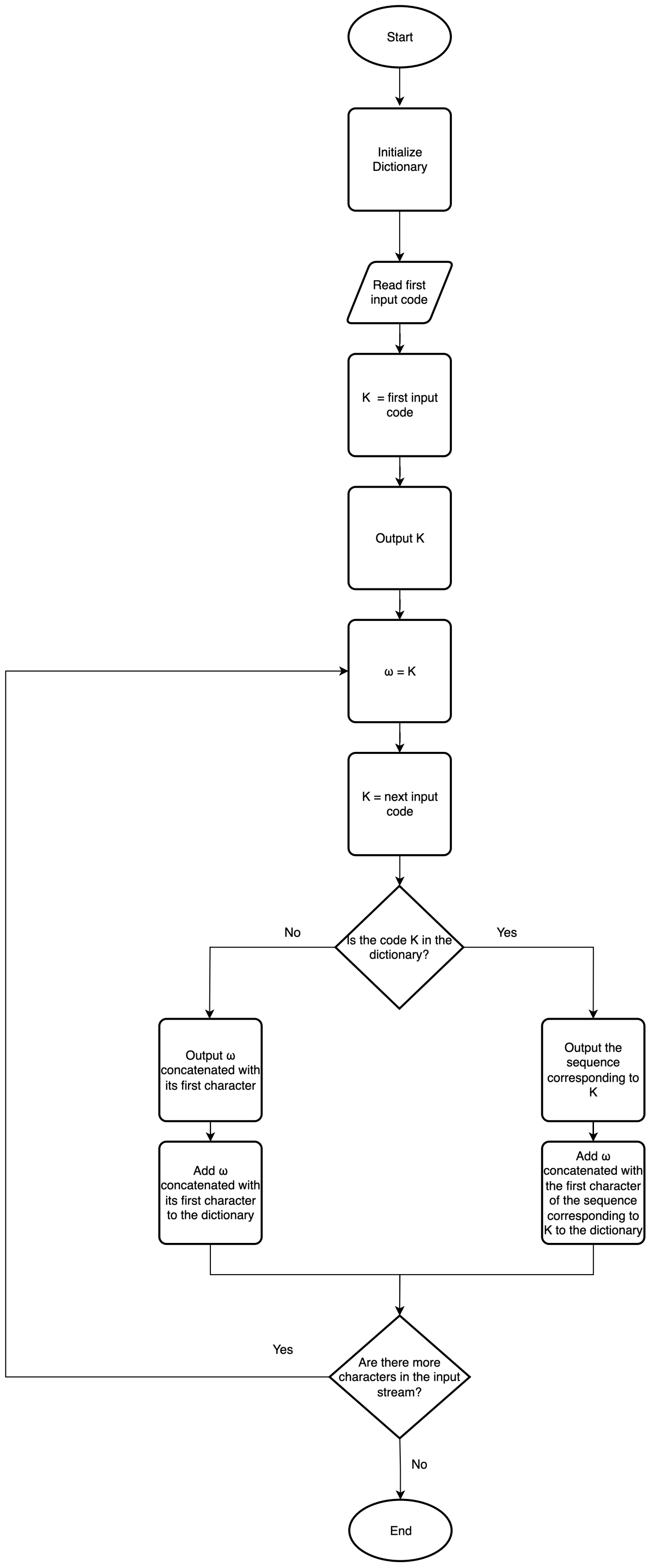

Start
Initialize Dictionary
Read first input code
K = first input code
Output K
ω = K
K = next input code
Is the code K in the dictionary?
No
Yes
Output ω concatenated with its first character
Add ω concatenated with its first character to the dictionary
Output the sequence corresponding to K
Add ω concatenated with the first character of the sequence corresponding to K to the dictionary
Are there more characters in the input stream?
Yes
No
End

Figure 3.2.3: Decompression Flow Chart.

3.2.4 Demonstration of the decompression process

To illustrate, let us consider the compressed version of the input string ABABABAB, used in the previous example, which is:

```
1 2 3 5 2
```

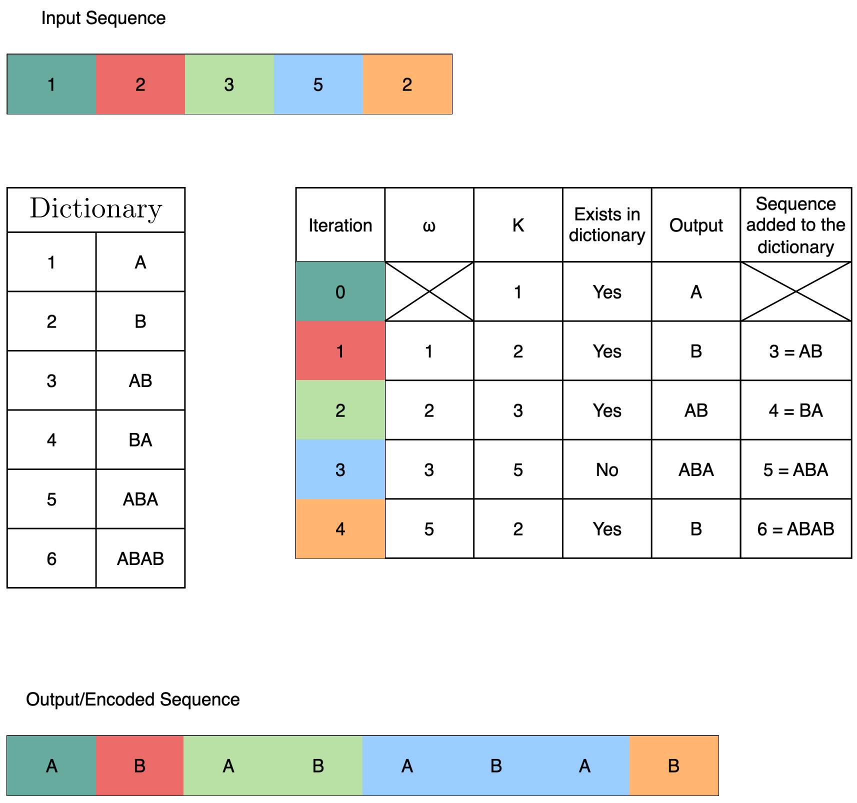
This sequence of codes will now be used to reconstruct the original string and demonstrate how LZW decompression works.

Figure 3.2.4 presents a demonstration of the decompression process. For simplicity, the dictionary is initialized with only two possible characters and their codes: 1 = A and 2 = B. In practice, it is essential that the decompressor starts with the same initial dictionary as the compressor; otherwise, decompression will fail.

The dictionary is represented with code values in the left column and their associated symbols in the right. The algorithm proceeds as follows:

1. The algorithm begins by reading the first code from the compressed stream and immediately writing the corresponding character to the output. Since the dictionary is initialized, the first element corresponds directly to a character, not to an extended codeword.
2. This first code is also stored in the variable ω, representing the previous sequence.
3. The next code from the stream is then read into K. If it exists in the dictionary, the corresponding sequence is output, and a new dictionary entry is created: the concatenation of ω and the first character of the string represented by K.
4. If the code is not yet in the dictionary, the algorithm applies the LZW exception rule (the so-called $K\omega K\omega K$ case). In this situation, the decoder outputs ω concatenated with its own first character. For example, when the decompressor reaches code 5 in the input stream, it is not yet present in the dictionary. The previous sequence ω is AB; concatenating it with its first character A gives ABA. This matches the entry that the compressor added earlier, thereby keeping both dictionaries synchronized. The sequence is then output and added to the dictionary.

These steps are repeated until the entire input stream is consumed. By the end of the process, the original string is fully reconstructed.



Dictionary	
1	A
2	B
3	AB
4	BA
5	ABA
6	ABAB

Iteration	ω	K	Exists in dictionary	Output	Sequence added to the dictionary
0	☒	1	Yes	A	☒
1	1	2	Yes	B	3 = AB
2	2	3	Yes	AB	4 = BA
3	3	5	No	ABA	5 = ABA
4	5	2	Yes	B	6 = ABAB

Figure 3.2.4: An example to demonstrate the decompression algorithm.

3.3 Strengths and Weaknesses

While LZW is straightforward in concept [1], its advantages and limitations deserve a more nuanced discussion. One of its main strengths lies in its elegant simplicity: both the encoder and decoder build their dictionaries in lockstep, which removes the need to transmit extra side information. This deterministic reconstruction ensures reliability and makes the algorithm appealing for hardware implementations. Furthermore, its execution speed is relatively high, since the main operations involve table lookups and simple concatenations.

On the other hand, the algorithm does not always achieve the same compression efficiency as more modern approaches. Its performance is heavily influenced by the statistical characteristics of the input data: highly repetitive streams compress well, while more random inputs yield modest gains. Another challenge is dictionary management. Because the dictionary grows continuously until a size limit is reached, an implementation must decide whether to freeze the dictionary or reset it, each option carrying trade-offs in terms of compression ratio and complexity.

Chapter 4

Software Implementation

4.1 Introduction

After understanding the fundamentals of LZW compression and decompression, the next step of the internship was to implement the algorithm in software. By coding both compression and decompression modules from scratch, without relying on existing optimized libraries, I was able to experiment with different design choices such as dictionary structure, codeword size, and input/output management.

This stage of the internship therefore served as a guideline to deepen my practical understanding of programming the LZW algorithm and to define a baseline set of functionalities aimed at improving both the compression rate and execution speed. During the FPGA implementation phase, the software version of the compressor was further optimized and adapted to fit the constraints and requirements of the hardware architecture.

4.2 Development Environment and Tools

4.2.1 Hardware Selection

The software implementation of the LZW algorithm was carried out on my personal computer, a MacBook Pro equipped with an M1 processor. This environment provided sufficient computational power for both development and testing.

4.2.2 Programming Language

For programming, I used the C language, chosen both for its suitability for later adaptation of the algorithm to FPGA implementation and because it is the language in which I am most proficient.

4.2.3 Integrated Development Environment

The development environment was **CLion**, an integrated development environment (IDE) developed by JetBrains.

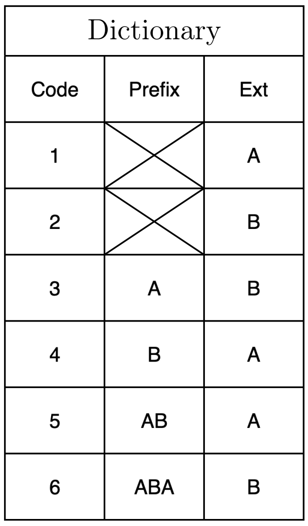
4.3 Compression Implementation

The software implementation of the LZW algorithm preserved the fundamental principles of dictionary-based compression while incorporating practical enhancements that have been introduced over the years. At its core, the algorithm still relies on dynamically building a dictionary of symbol sequences and replacing repeated patterns with shorter numerical codes. However, translating this into a real-world software application required refining several aspects: the management of dictionary size, the choice of data structures for efficient symbol lookup, and the handling of variable-length codewords. These improvements not only made the implementation more robust and adaptable but also ensured that the compressor could achieve better performance in terms of speed and compression ratio. By combining the theoretical foundations of LZW with these practical considerations, the software implementation served as a bridge between the algorithm's original design and the needs of modern applications, as well as a solid reference point for the subsequent FPGA-based development.

4.3.1 Maintaining the Dictionary

Each new string added to the dictionary is formed by extending an existing prefix with a single byte. As a result, each dictionary entry can be stored efficiently by keeping only the prefix code and the added byte.

Using the example shown in FIGURE 3.2.2, the dictionary can be represented as illustrated in FIGURE 4.3.1.



Dictionary		
Code	Prefix	Ext
1	✕	A
2	✕	B
3	A	B
4	B	A
5	AB	A
6	ABA	B

Figure 4.3.1: Trie Dictionary Example.

In C, the dictionary is represented as a structure.

Dictionary Structure

```c
typedef struct {
    uint16_t prefix_code;
    uint8_t ext_byte;
    uint16_t code;
} Dictionary;
```

The dictionary is then initialized with all possible values of a byte, the 256 ASCII single characters, each associated with its fixed code. In addition, I have created a new array called dictionary_used that has the same number of entries as our Dictionary. This new array is used to check if a certain entry is empty or not.

Dictionary Initialization

```
void Dictionary_init(void) {
    memset(dictionary_used, 0, sizeof(dictionary_used));
    for (uint16_t i = 0; i < 256 ; i++) {
        dictionary[i].code = i;
        dictionary[i].prefix_code = INVALID_CODE;
        dictionary[i].ext_byte = (uint8_t)i;
        dictionary_used[i] = true;
        dict_size_actual++;
    }
}
```

The dictionary must also support fast lookups to check whether a given sequence (a prefix combined with an extension byte) already exists, and insert it if it does not. To achieve this, the compressor's dictionary tree nodes are stored in a hash table, where the key is defined by the pair `(prefix code, extension byte)`.

A hash table is a data structure that allows very efficient storage and retrieval of key–value pairs. It uses a hash function to map each key to an index in an array, making the average time complexity for insertion and lookup close to constant time, $\mathcal{O}(1)$. This efficiency makes hash tables particularly well-suited for dictionary-based compression algorithms such as LZW, where thousands of lookups and insertions are performed during the processing of even small input streams. FIGURE 4.3.2 provides an example of how a hash table operates.

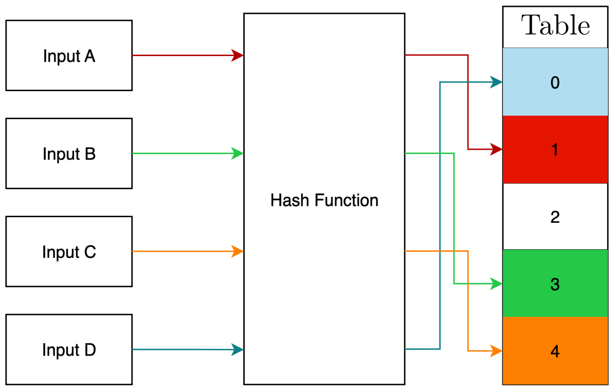


Figure 4.3.2: Hash table example.

The hash function that was chosen should be simple and fast to calculate to ensure fast access and distribution of dictionary entries while minimizing collisions.

First, the prefix code is shifted left by 5 bits. This operation spreads the binary representation of the prefix, helping to reduce clustering in the hash table when many similar prefixes occur.

Then, the shifted prefix is XORed with the extension byte. The XOR operation mixes the bits of both inputs, ensuring that even small differences in the extension byte result in different hash values.

Finally, the result is reduced modulo the maximum dictionary size, which ensures that the computed index always falls within the valid range of the hash table.

Hash function

```c
uint32_t hash(uint16_t prefix, uint8_t ext) {
    return ((prefix << 5) ^ ext) % MAX_DICT_SIZE;
}
```

The lookup function is responsible for determining whether a given sequence, defined by a prefix code and an extension byte, already exists in the dictionary. If the sequence is found, the function returns its associated code; otherwise, it signals absence by returning a special value `INVALID_CODE`.

The function begins by calculating a starting index with *hash(prefix, ext)*. This gives the initial position in the hash table where the sequence is most likely to be found.

If the slot at the computed index is occupied by another entry (a hash collision), the function resolves it by checking the following slots sequentially *((h + i) % MAX_DICT_SIZE)*. This process continues until either the sequence is found or all slots have been probed.

At each probed slot, the stored prefix_code and ext_byte are compared with the search values. If they match, the dictionary entry is found, and its code is returned.

If no match is discovered after checking the entire table, the function returns `INVALID_CODE`, indicating that the sequence does not exist yet and should be added during compression.

Dictionary lookup

```c
uint16_t Dictionary_find(uint16_t prefix, uint8_t ext) {
    uint32_t h = hash(prefix, ext);
    for (uint32_t i = 0; i < MAX_DICT_SIZE; i++) {
        uint32_t idx = (h + i) % MAX_DICT_SIZE;
        if (dictionary[idx].prefix_code == prefix && dictionary[idx].ext_byte
         ↪  == ext)
            return dictionary[idx].code;
    }
    return INVALID_CODE;
}
```

Whenever a new sequence (a prefix combined with an extension byte) is not found in the dictionary, the compressor must insert it as a new entry. This is handled by the function `Dictionary_add`, which assigns a new code to the sequence and stores it in the dictionary.

The function begins by checking whether the dictionary has already reached its maximum capacity `MAX_DICT_SIZE`. If so, no further entries can be added, and the function returns immediately.

Next, it verifies whether the current number of entries has exceeded the maximum value representable with the current code width (`bit_count`). If this is the case, the code width is incremented by one bit, provided it does not exceed the theoretical maximum of $\log_2$(`MAX_DICT_SIZE`). This mechanism ensures that as the dictionary grows, the compressor dynamically increases the number of bits used to represent codes. The variable-length codes and their implications will be expanded upon in the next section.

Once the dictionary growth policy is handled, the function computes the initial index in the hash table with the function `hash(prefix, ext)`. If the slot at the calculated index is already occupied, linear probing is used to find the next available slot, following the rule `(h + i) %`

MAX_DICT_SIZE. This guarantees that even in case of collisions, the new entry will eventually find a free location.

If a free slot is found, the new dictionary entry is created by storing the prefix_code, ext_byte, and the current value of dict_size_actual as the assigned code. At the same time, dictionary_used[idx] is set to true to mark the slot as occupied. Finally, dict_size_actual is incremented, ensuring that the next added entry receives a unique code, and the function returns immediately.

Dictionary add

```
void Dictionary_add(uint16_t prefix, uint8_t ext) {
    if (dict_size_actual >= MAX_DICT_SIZE) return;
    if(dict_size_actual >= (1u << bit_count) && bit_count<
     ↪  (int)log2(MAX_DICT_SIZE)) bit_count++;

    uint32_t h = hash(prefix, ext);
        for (uint32_t i = 0; i < MAX_DICT_SIZE; i++) {
        uint32_t idx = (h + i) % MAX_DICT_SIZE;
         if (!dictionary_used[idx]) {
            dictionary[idx].prefix_code = prefix;
            dictionary[idx].ext_byte = ext;
            dictionary[idx].code = dict_size_actual;
            dictionary_used[idx] = true;
            dict_size_actual++;
            return;
        }

    }
}
```

4.3.2 Variable-Length Codes and Writing Output

The input stream is composed of single bytes. When compressing 8-bit symbols, it is necessary to use at least 9-bit codes in order to represent sequences beyond the initial one-byte entries in the dictionary [1].

As the dictionary grows, the number of bits required to represent each code must increase accordingly. The standard schedule is given below:

$$\texttt{bit\_count} = \begin{cases} 9 & \text{for } \texttt{dict\_size\_actual} \in [256, 511], \\ 10 & \text{for } \texttt{dict\_size\_actual} \in [512, 1023], \\ 11 & \text{for } \texttt{dict\_size\_actual} \in [1024, 2047], \\ 12 & \text{for } \texttt{dict\_size\_actual} \in [2048, 4095]. \end{cases}$$

This schedule follows a predictable pattern that can be directly exploited in software. Whenever the dictionary size reaches a power of two (dict_size_actual == (1u « bit_count)), the number of bits used to encode each symbol is increased by one. In practice, this means that bit_count is dynamically adjusted during compression.

The value of bit_count therefore serves two purposes: it defines how many bits are used to represent each code, and it determines how many bits are written for each symbol in the output stream. Both `bit_count` and the write position pointer `bitstream_index` are maintained as global variables since they must be updated consistently throughout the compression process.

Once a code has been generated by the compressor, it must be written to the output buffer using the current number of bits defined by `bit_count`. This task is handled by the function `write_to_bitstream`, which packs variable-length codes into a continuous sequence of bytes.

The function receives as input a code of type `uint16_t`, which contains the numerical value to be written. Since the number of bits used to represent the code is not fixed but depends on the current dictionary size, the function writes exactly `bit_count` bits for each code.

The process begins by iterating over the code bits from the most significant down to the least significant. This is achieved through a loop that starts at `(bit_count - 1)` and decrements to zero. At each iteration:

1. The current bit is extracted by right-shifting the code and masking with 1: `bool bit = (code » i) & 1`.
2. This bit is then written to the correct position inside the output buffer `bitstream`. The destination byte is given by `bitstream[bitstream_index / 8]`, while the exact bit position inside that byte is determined by `(7 - (bitstream_index % 8))`. This ensures that bits are packed from the most significant to the least significant position inside each byte.
3. Finally, the global variable `bitstream_index` is incremented, so the next bit will be placed in the following position.

Write output

```
void write_to_bitstream(uint16_t code) {
    for (int i = (bit_count - 1); i >= 0; i--) {
        bool bit = (code >> i) & 1;
        bitstream[bitstream_index / 8] |= (bit << (7 - (bitstream_index %
        ↪  8)));
        bitstream_index++;
    }
}
```

This bit-by-bit approach is straightforward and guarantees correctness, but it is not the most efficient possible implementation. In particular, one could object that the loop could be replaced by masking and shifting multiple bits at once, thereby reducing overhead. While this is a valid concern for performance, the design choice here prioritizes clarity. Furthermore, this limitation is explicitly addressed in `Chapter` 5, where the output-writing mechanism is reworked to avoid the inefficiency of per-bit writes.

4.3.3 Main compression function

The core of the compressor operates as described and illustrated in the flow chart (FIGURE 3.2.1), with minor adjustments made to suit the software implementation.

First, the compressor works based on the length of the input stream. The function begins with two edge cases :

- If the input is empty (`data_len == 0`), nothing is produced.
- If the input has a single byte, that byte is written directly as a code.

After initialization, the loop scans the input from the second byte onward. For each byte, it probes the dictionary for the pair `<prefix, ext>` using `Dictionary_find`. On a hit, the prefix grows to the longer sequence (represented by the returned code). On a miss, the current prefix is emitted, the new pair is added via `Dictionary_add`, and the prefix resets to the single-byte code of `ext`. When the input is exhausted, the last pending `prefix` is written. Finally, the bitstream is padded with zeros up to the next byte boundary so that the output length is integral in bytes.

Main Compression Function

```
    void compress() {
    if (data_len == 0) return;

    if (data_len == 1) {
        write_to_bitstream(data[0]);
        return;
    }

    uint16_t prefix = data[0];
    uint8_t ext;

    for (size_t i = 1; i < data_len; i++) {
        ext = data[i];
        uint16_t code = Dictionary_find(prefix, ext);
        if (code != INVALID_CODE) {
            prefix = code;
        } else {
            write_to_bitstream(prefix);
            Dictionary_add(prefix, ext);
            prefix = ext;
        }
    }
    write_to_bitstream(prefix);

    while (bitstream_index % 8 != 0)
    {
        bitstream[bitstream_index / 8] |= (0 << (7 - (bitstream_index % 8)));
        bitstream_index++;
    }
}
```

4.4 Decompression Implementation

Unlike compression, the LZW decompression algorithm was not implemented on the FPGA. The software decompressor, although less optimized and less polished than its counterpart, proved valuable for deepening the understanding of the LZW process and guiding the design of the compressor. After all, compression without the ability to decompress would only result in permanent data loss.

4.4.1 Maintaining the Dictionary

The dictionary used by the decompressor is a simple structure, where each entry contains three fields:

1. A `code` variable that stores the code associated with the sequence,
2. A `sequence` array that stores the actual sequence of bytes,
3. The `length` of the sequence.

Dictionary Structure

```
typedef struct {
    uint16_t code;
    uint8_t sequence[MAX_SEQUENCE_LENGTH];
    uint16_t length;
} Dictionary;
```

The dictionary is initialized with all possible values of a single byte, i.e., the 256 ASCII characters, each associated with its fixed code.

Dictionary Initialization

```
void Dictionary_init(void) {
    for (uint16_t i = 0; i < 256; i++) {
        dictionary[i].code = i;
        dictionary[i].sequence[0] = (uint8_t)i;
        dictionary[i].length = 1;
        dict_size_actual++;
    }
}
```

Unlike the compressor, this dictionary does not use a hash table. Instead, lookup operations are performed by traversing the dictionary sequentially.

When the function `Dictionary_find` is called, it iterates from code 0 up to the current dictionary size, searching for a matching code. If a match is found, it returns a pointer to the dictionary entry, which contains both the stored sequence and its length.

If no match is found, the function returns `NULL`. This case only occurs when the requested code is greater than or equal to the current dictionary size, meaning that the code has not yet been assigned to any entry.

Dictionary Lookup

```
Dictionary* Dictionary_find(uint16_t code) {
    for (uint16_t i = 0; i < dict_size_actual; i++) {
        if (dictionary[i].code == code) {
            return &dictionary[i];
        }
    }
    return NULL;
}
```

Adding a new sequence to the dictionary is also straightforward. The function first checks whether the dictionary has already reached its maximum capacity (`MAX_DICT_SIZE`). If so, no further entries can be added. It also verifies that the sequence length does not exceed the maximum allowed length (`MAX_SEQUENCE_LENGTH`).

If both conditions are satisfied, a new entry is created at the index `dict_size_actual`. The code is set equal to the current dictionary size, the sequence is copied into the entry, its length is stored, and the dictionary size is incremented.

Dictionary Add

```
void Dictionary_add(const uint8_t *sequence, uint16_t seq_len) {
    if (dict_size_actual >= MAX_DICT_SIZE) return;
    if (seq_len > MAX_SEQUENCE_LENGTH) return;

    dictionary[dict_size_actual].code = dict_size_actual;
    memcpy(dictionary[dict_size_actual].sequence, sequence, seq_len);
    dictionary[dict_size_actual].length = seq_len;
    dict_size_actual++;
}
```

4.4.2 Variable-Length Codes and Reading Input

As in compression, the number of bits required to represent each code increases as the dictionary grows. The mechanism for updating `bit_count` is the same, but the key difference lies in the timing: during decompression, `bit_count` must be updated *before* reading from the input stream. This ensures that the decompressor always uses the correct number of bits to reconstruct each code word.

Unlike compression, which writes variable-length codes to the output stream, decompression must read variable-length codes from the input stream and produce fixed 8-bit bytes as output. Since the decompressor always lags one step behind the compressor, synchronizing the update of `bit_count` is critical to avoid misalignment.

The task of reading codes is handled by the function `ReadCode`, which reconstructs a code word by sequentially extracting `bit_count` bits from the input buffer. The function relies on two global variables, `byte_position` and `bit_position`, which track the current read position inside the compressed data stream.

The procedure works as follows:

1. A local variable `code` is initialized to zero.
2. The function iterates exactly `bit_count` times. At each step, the current bit is extracted from `data[byte_position]`:

   ```
   uint8_t bit = (data[byte_position] » (7 - bit_position)) & 1;
   ```

   This ensures that bits are read in most-significant-bit (MSB) order.
3. The extracted bit is appended to the code by shifting left and OR-ing:

   ```
   code = (code « 1) | bit;
   ```

4. The global variable `bit_position` is then incremented. If it reaches 8, the current byte has been fully consumed, so `bit_position` is reset to zero and `byte_position` is incremented to move to the next byte.

After completing `bit_count` iterations, the function returns the reconstructed code as a `uint16_t`.

Read Input

```c
uint16_t ReadCode(void) {
    uint16_t code = 0;

    for (size_t i = 0; i < bit_count; i++) {
        uint8_t bit = (data[byte_position] >> (7 - bit_position)) & 1;
        code = (code << 1) | bit;

        bit_position++;
        if (bit_position == 8) {
            bit_position = 0;
            byte_position++;
        }
    }

    return code;
}
```

As with the write function in compression, this implementation reads bits sequentially, which is clear but not the most efficient. A possible objection is that multi-bit masking and shifting could speed up the process. However, in this work, decompression was retained only as a software reference to verify correctness. Since the hardware design focused exclusively on compression, no further effort was invested in optimizing the software decompression routine, making this function mainly illustrative rather than performance-critical.

4.4.3 Main decompression function

The core of the decompressor operates as described and illustrated in the flow chart (Figure 3.2.3), with some minor adjustments made to adapt it to the software implementation.

First, the function begins by declaring two temporary buffers, `prev_sequence` and `new_sequence`, along with `prev_length`. The variable `len` tracks how many unread bits remain in the input buffer (initialized to `data_len * 8`).

The first code is then read using `ReadCode()`, and the corresponding dictionary entry is retrieved via `Dictionary_find`. If it exists, its sequence is copied into `prev_sequence`, written to the output, and `prev_length` is updated accordingly. After this seed step, `bit_count` is incremented to follow the growth schedule used during compression.

However, if it does not exist, this indicates that the compression was performed incorrectly, making correct decompression impossible. In such a case, the function terminates and returns immediately.

The main loop runs while there are still enough bits to read the next code:

1. First, it verifies wheter the current number of entries in the dictionary has exceeded the maximum value representatble with the current code width and adjust `bit_count` accordingly.
2. The function calls `ReadCode()` to obtain `curr_code`. It then looks up the entry using `Dictionary_find`.

3. Two cases are then possible :
    - If the entry is found, the current sequence and its length are taken directly from the dictionary.
    - If the entry is not found, the function will then handle this scenario by forming a new sequence and adding it to the dictionary (this is the famous $K\omega K\omega K$ case).
4. Then the current sequence is copied to `output` and `output_index` is advanced.
5. If there is room for one additional symbol, a new dictionary entry is added that concatenates the previous sequence with the first byte of the current sequence:

    ```
    Dictionary_add(prev_sequence + curr_sequence[0]).
    ```

6. Finally, `prev_sequence` and `prev_length` are updated to the current sequence, and the remaining bit counter `len` is decreased by `bit_count`.

After the loop, a final check trims a trailing zero byte from the output if present.

4.5 Summary

The software implementation successfully demonstrated the complete cycle of LZW compression and decompression. On the compression side, the algorithm was realized with a hash-based dictionary supporting fast lookups and insertions, dynamic adjustment of code width, and efficient bit-packing of variable-length codes into a continuous output stream. On the decompression side, the dictionary was implemented as a sequence-based structure, capable of reconstructing original data from variable-length codes while handling edge cases such as the $K\omega K\omega K$ scenario. Together, these modules validated the correctness of the approach: data compressed by the software compressor could be reliably decompressed back to its original form, establishing a solid foundation for the subsequent FPGA hardware implementation.

It is important to note that the presented implementation represents only one of many possible ways to realize LZW compression and decompression in software. The focus here was to create a functional and understandable version of the algorithm that could later serve as a reference for the FPGA-based hardware design. Naturally, there is still room for improvement. The code could be further optimized in terms of execution speed, memory management, and data structures, particularly for handling very large inputs. In addition, more extensive safety checks could be introduced to ensure robustness, such as protecting against buffer overflows, invalid input streams, or dictionary overgrowth. These refinements were not the primary goal of this internship stage but could be valuable directions for future development, especially if the software version were to be used in production environments.

Chapter 5

Hardware Implementation

5.1 Introduction

The final stage of the internship focused on the hardware realization of the LZW compression algorithm using an FPGA platform. After completing and validating the software implementation, the goal was to translate the algorithm into a hardware architecture capable of achieving higher performance and efficiency. This phase involved designing the LZW compressor as a custom hardware block, integrating it with the FPGA system, and ensuring that it could operate correctly on real data streams.

The implementation process included several key steps: adapting the software design to a hardware-friendly structure, managing memory and control logic within the constraints of the FPGA, and ensuring that the compression functionality was preserved. The design was then synthesized, deployed onto the FPGA, and thoroughly tested to verify its correctness.

Once functional validation was completed, the hardware compressor was benchmarked against its software counterpart. This comparison provided valuable insights into performance differences in terms of speed, resource utilization, and scalability.

First, we will review some key concepts that are essential for implementing the algorithm in hardware, followed by an overview of the development environment and tools used during this stage.

5.2 IP (Intellectual Property)

An IP (Intellectual Property) core is a block of logic or data that is used in making a field programmable gate array (FPGA) for a product. It can be thought of as a reusable design component, which may implement anything from a simple arithmetic unit to a complete communication interface. IP cores are commonly provided by FPGA vendors or developed in-house, allowing designers to accelerate development by integrating preverified modules instead of building every function from scratch.

5.3 Deployment Tools

5.3.1 Vivado

Vivado is the design environment provided by Xilinx for developing FPGA-based systems. It is a complete software suite that supports hardware description language (HDL) design, synthesis, simulation, and implementation. Vivado also provides tools for analyzing performance, debugging, and integrating custom IP cores, making it the central platform for FPGA development in this project.

5.3.2 Vitis Unified IDE

Vitis Unified IDE is the development environment provided by Xilinx for programming and testing FPGA-based systems. Unlike Vivado, which focuses on hardware design and IP integration, Vitis provides the software side of the workflow. It enables developers to create applications that run on the embedded processors of FPGA platforms, such as the ARM cores of a Zynq device.

A key feature of Vitis is its support for High-Level Synthesis (HLS), which allows developers to describe hardware functionality using high-level programming languages such as C or C++. The Vitis HLS tool then translates these descriptions into HDL, enabling the creation of custom IP cores without manually writing Verilog or VHDL. This capability was particularly important in this project, where the LZW compressor was implemented as a custom IP core generated through HLS.

In addition to HLS, Vitis facilitates the creation and development of applications that can be executed directly on the FPGA board, making it possible to test both the hardware IP cores and pure software implementations within the same environment.

5.4 Hardware Selection

In this internship, LZW compression was implemented on two different FPGA platforms. This provided the opportunity to evaluate the design under varying hardware constraints and to compare how the same algorithm behaves across devices with different resources and performance capabilities. By targeting more than one board, it was also possible to validate the portability of the implementation and ensure that the design was not tied to a single hardware configuration.

5.4.1 ZedBoard

The ZedBoard is an evaluation and development board based on the Xilinx Zynq-7000 SoC, which integrates a dual-core ARM Cortex-A9 processor with FPGA fabric. The ZedBoard was used as the primary development platform in this internship, providing a straightforward environment to design, synthesize, and test the LZW compressor [2].

5.4.2 Kria KV260 Vision AI

The Kria KV260 Vision AI Starter Kit is a more advanced FPGA platform based on the Xilinx Zynq UltraScale+ MPSoC. It is designed for edge AI and vision applications, featuring a quad-core ARM Cortex-A53 processor, GPU, real-time processors, and powerful FPGA fabric. Compared to the ZedBoard, the KV260 provides significantly more computational resources and higher performance, making it suitable for testing the scalability of the LZW compressor and evaluating its behavior in a modern heterogeneous computing environment [3].

5.5 HLS Implementation on ZedBoard

As mentioned earlier, the chosen approach for implementing and testing the LZW compression on FPGA was through High-Level Synthesis (HLS). The original software implementation of the algorithm was first adapted and optimized to meet the constraints of hardware design.

In this process, HLS directives played an essential role. Directives provide guidance to the compiler on how to map C source code into efficient hardware structures and primitives. By applying them strategically, it was possible to control aspects such as loop unrolling, pipelining, and memory access, which directly impact the performance and resource utilization of the final design [4].

The following subsections present the evolution of the code from its initial form to a synthesizable and optimized hardware design, while also highlighting the use of directives at each stage.

5.5.1 Maintaining the Dictionary

Both the dictionary and the procedure for its initialization were not modified from their original software counterparts. The dictionary itself is still implemented as a Trie-like structure, using both the prefix code and the extension byte to store sequences. As in the software version, the first 256 ASCII single-character entries are inserted at initialization. The auxiliary array `dictionary_used` is also present here, serving as a flag to determine whether a given slot in the dictionary is occupied.

For the initialization process, a performance optimization directive, `PIPELINE`, was applied. This directive instructs the HLS compiler to break down loop iterations into smaller stages that can execute concurrently [4]. This means that instead of completing one iteration of the loop before starting the next, multiple iterations overlap in time, allowing new iterations to begin each clock cycle. As a result, the throughput of the initialization loop is significantly improved, reducing the number of clock cycles required to load the base dictionary entries.

A new addition in the hardware implementation is a dedicated function responsible for resetting the dictionary. Unlike the software version, where reaching the maximum capacity of the dictionary simply prevents further insertions, the hardware version allows the dictionary to be fully reinitialized. This reset procedure restores the effective dictionary size back to 256 entries (the base ASCII set) and adjusts the number of bits required to represent codes accordingly. In doing so, the compressor can continue operating without requiring a complete restart of the system, ensuring better adaptability and robustness in long compression sessions, as well as faster compression time.

Dictionary Reset

```c
void Dictionary_reset(uint16_t *dictionary_size, uint8_t *bit_count) {
    #pragma HLS INLINE off
    (*dictionary_size) = 256;
    (*bit_count) = 8;
    init_dictionary();
}
```

The hash function was also retained in the hardware implementation. However, instead of relying on a single simple function as in the software version, the index calculation was redesigned to use two different hash functions. This dual-hash approach helps to ensure a more uniform distribution of dictionary entries across the available memory space, thereby reducing clustering and lowering the probability of collisions. As a result, lookup operations in the dictionary become more efficient, which is particularly important in hardware where timing and resource utilization are critical factors.

Hash Functions

```
uint32_t hash1(uint16_t prefix, uint8_t ext) {
    #pragma HLS INLINE
    return ((prefix << 8) ^ ext) & (MAX_DICTIONARY_SIZE - 1);
}

uint32_t hash2(uint16_t prefix, uint8_t ext) {
    #pragma HLS INLINE
    return (((prefix << 5) ^ (ext * 7)) & (MAX_DICTIONARY_SIZE - 1)) | 1;
}
```

Both the lookup function (`Dictionary_find`) and the insert function (`Dictionary_add`) remain largely identical to their software counterparts. The main distinction lies in their use of the dual-hash approach, which improves collision handling when searching for or inserting new entries in the dictionary.

In addition, a new condition was introduced in `Dictionary_find` to immediately stop the search if the probed slot does not contain the expected prefix and extension pair. This early-exit mechanism prevents unnecessary iterations through the dictionary, reducing latency and improving overall efficiency in hardware.

Dictionary lookup

```
    uint16_t Dictionary_find(uint16_t prefix, uint8_t ext) {
    #pragma HLS INLINE off
    uint32_t h1 = hash1(prefix, ext);
    uint32_t h2 = hash2(prefix, ext);
    for (uint32_t i = 0; i < MAX_DICTIONARY_SIZE; i++) {
        #pragma HLS PIPELINE II=1
        uint32_t idx = (h1 + i * h2) & (MAX_DICTIONARY_SIZE - 1);

        if (!dictionary_used[idx]) return INVALID_CODE;

        if (dictionary[idx].prefix_code == prefix && dictionary[idx].ext_byte
         ↪  == ext) return dictionary[idx].code;
    }
    return INVALID_CODE;
}
```

As for `Dictionary_add`, the function now handles the reset of the dictionary whenever its maximum capacity has been reached.

Dictionary Add

```
    void Dictionary_add(uint16_t prefix, uint8_t ext, uint16_t
     ↪  *dictionary_size, uint8_t *bit_count) {
    #pragma HLS INLINE off
    if (*dictionary_size >= MAX_DICTIONARY_SIZE)
     ↪  Dictionary_reset(dictionary_size, bit_count);
    if (*dictionary_size >= (1u << *bit_count)) (*bit_count)++;
    uint32_t h1 = hash1(prefix, ext);
    uint32_t h2 = hash2(prefix, ext);
    for (uint32_t i = 0; i < MAX_DICTIONARY_SIZE; i++) {
        uint32_t idx = (h1 + i * h2) & (MAX_DICTIONARY_SIZE - 1);

        if (!dictionary_used[idx]) {
            dictionary[idx].prefix_code = prefix;
            dictionary[idx].ext_byte = ext;
            dictionary[idx].code = *dictionary_size;
            dictionary_used[idx] = true;
            (*dictionary_size)++;
            return;
        }
    }
}
```

5.5.2 Writing the Output

The `write_output` function performs the same role as the software `write_to_bitstream`, namely packing variable-length codes into a byte stream. However, a few design choices were made specifically for hardware:

1. The function avoids global variables and instead uses explicit parameters (`output`, `bit_count`, `out_index`),
2. The loop is structured around filling bytes in chunks, with `#pragma HLS PIPELINE II=1` ensuring that one chunk can be processed per clock cycle.
3. Bytes are explicitly cleared before writing to avoid stale data, which is essential in a hardware context

Write Output

```
    void write_output(uint16_t code, uint8_t *output, uint8_t bit_count,
     ↪  uint32_t *out_index) {
    #pragma HLS INLINE off
    uint32_t idx = *out_index;
    uint32_t byte_index = idx / 8;
    uint32_t bit_offset = idx % 8;

    uint32_t bits_left = bit_count;
    while (bits_left > 0) {
        #pragma HLS PIPELINE II=1
        uint8_t space_in_byte = 8 - bit_offset;
        uint8_t bits_to_write = (bits_left < space_in_byte) ? bits_left :
         ↪  space_in_byte;
        uint8_t mask = ((code >> (bits_left - bits_to_write)) \& ((1U <<
         ↪  bits_to_write) - 1));

        if (bit_offset == 0) output[byte_index] = 0;
        output[byte_index] |= mask << (space_in_byte - bits_to_write);
        bit_offset += bits_to_write;
        if (bit_offset == 8) {
            bit_offset = 0;
            byte_index++;
        }
        bits_left -= bits_to_write;
    }
    *out_index += bit_count;
}
```

The hardware version of the function differs from the software one because the bit-by-bit, global-variable approach of `write_to_bitstream` is not efficient. In hardware, the routine was redesigned to use explicit parameters, chunk-based writes, and pipelining directives, which are necessary to achieve correct synthesis and high throughput on the FPGA.

5.5.3 Main Compression Function

Finally, the main compression function remains relatively unchanged in terms of its algorithmic structure. However, in the hardware implementation, it is defined as the *top function*. In the context of HLS, the top function is the entry point that will be synthesized into hardware. It represents the boundary of the IP core and defines how the module interacts with the outside world.

Within this function, the input and output ports, as well as the control interface, are explicitly defined. These ports establish how the IP core communicates with the Processing System (PS) of the FPGA. In this implementation, communication is achieved using AXI interfaces:

- `m_axi` interfaces provide high-throughput memory-mapped connections for reading the input data and writing the compressed output.
- `s_axilite` interfaces are used for lightweight control and configuration, such as passing parameters and retrieving status information.

These interfaces ensure seamless integration of the hardware accelerator into the overall system. The details of the AXI communication protocol and its role will be further discussed in the following section.

Top Function Interfaces

```
#pragma HLS INTERFACE m_axi depth=input_size port=input offset=slave
↪   bundle=AXIM_A
#pragma HLS INTERFACE m_axi depth=input_size port=output offset=slave
↪   bundle=AXIM_A
#pragma HLS INTERFACE s_axilite port=input   bundle=control
#pragma HLS INTERFACE s_axilite port=output  bundle=control
#pragma HLS INTERFACE s_axilite port=return  bundle=control
#pragma HLS INTERFACE s_axilite port=input_size    bundle=control
#pragma HLS INTERFACE s_axilite port=compression_size bundle=control
```

The next step is to package the IP core and integrate it into a custom hardware design in Vivado.

5.5.4 Improvement to the Algorithm

This version of LZW compression maintains the same advantages as the software implementation described in Chapter 4, while introducing several improvements and optimizations:

1. A reset policy is applied when the dictionary reaches its maximum number of entries.
2. A dual-hash function is used to make the lookup and insert operations faster and more efficient.
3. An overhaul of the output function eliminates certain programming choices from the software version, replacing them with solutions better suited to hardware design.
4. Finally, the strategic use of multiple performance optimization directives leverages hardware acceleration and the structure of the IP core to achieve more efficient compression in hardware.

5.5.5 Resource Utilization

Vitis HLS provides the user with a detailed report on how many hardware resources an IP core utilizes. This feedback is extremely valuable, as it allows the designer to optimize the implementation, correct inefficiencies, and guide the compiler with directives in order to better meet the requirements of the design. As shown in Table 5.5.1, the LZW compression IP core that was created is relatively lightweight compared to the available hardware resources.

This low utilization is a positive outcome for several reasons. First, it means that a large portion of the FPGA fabric remains free, allowing additional functionality to be implemented alongside compression without running into resource constraints. For instance, one could integrate pre-processing or post-processing modules on the same device.

Second, the lightweight nature of the IP makes it highly portable and easier to integrate into more resource-constrained FPGA boards, as well as into larger, resource-rich devices such as the Kria KV260.

Finally, if the focus were solely on LZW compression, the fact that the current design leaves much of the FPGA unused indicates that there is significant headroom to scale up. For example, parallelization techniques or deeper optimizations could be applied to push performance further, fully exploiting the available hardware resources when compression is the primary task.

Criteria	**Guideline**	**Actual**
LUT	70 %	5.17 %
FD	50 %	3.41 %
LUTRAM+SRL	25 %	1.68 %
MUXF7	15 %	0.05 %
DSP	80 %	0.45 %
RAMB/FIFO	80 %	4.29 %
DSP+RAMB+URAM (Avg)	70 %	2.37 %

Table 5.5.1: Resource Utilization Summary for the ZedBoard

5.6 Creation of the Hardware Design

Once the IP core is generated from the HLS project, it must be integrated into a complete hardware design. This stage involves creating a block design in Vivado where the custom IP core is connected to the Processing System (PS) and other necessary peripherals. The purpose of this step is to prepare a functional hardware platform that can later be used to run and test the compression algorithm directly on the FPGA.

5.6.1 Zynq Programmable System

The Zynq-7000 AP SoC present on the ZedBoard combines a Processing System (PS) with a tightly integrated Programmable Logic (PL). The major functional blocks are:

- **Processing System (PS)**
  - Dual-core ARM Cortex-A9 Application Processing Unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - High-bandwidth interconnect
- **Programmable Logic (PL)**

The PL extends the capabilities of the PS to meet specific application requirements. It includes configurable logic blocks (CLBs), block RAM (BRAM), and DSP slices. In this project, the LZW compression module is implemented as a hardware IP core within the PL.

Communication between the PS and the PL is achieved through the **AXI (Advanced eXtensible Interface)** protocol. AXI defines a standardized handshake-based mechanism for reliable data transfers, ensuring interoperability and design portability. The Zynq-7000 family supports the latest version of the protocol, AXI4.

There are three types of AXI4 interfaces:

- **AXI4** – for high-performance memory-mapped transfers.
- **AXI4-Lite** – for low-throughput memory-mapped control and status registers.
- **AXI4-Stream** – for high-speed streaming data transfers.

Each transaction is initiated by an **AXI Master**, while an **AXI Slave** responds to the request. In the case of the LZW compressor, the IP core instantiated in the PL operates as an AXI4 Master for data transfers and exposes an AXI4-Lite Slave interface for configuration and control by the PS.

5.6.2 Integration of the IP Core

The generated IP was imported into the Vivado IP catalog and then instantiated inside the block design along with the Zynq-7000 Processing System. Figure 5.6.1 shows the complete system block design.

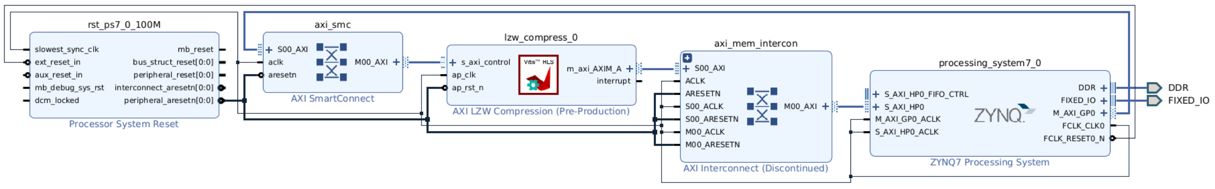


Figure 5.6.1: Vivado block design integrating the LZW Compression IP core with the Zynq-7000 PS.

As illustrated, the `lzw_compress` IP is connected to the ZYNQ Processing System (PS) through standard AXI interfaces:

- **AXI4-Lite (s_axi_control):** Used for control and configuration registers. This interface allows the ARM cores in the PS to start and stop the IP, configure parameters, and retrieve status signals.
- **AXI4 Master (m_axi_AXIM_A):** Used for high-throughput memory-mapped transfers between DDR memory and the compression IP. This interface streams input data to the IP core and stores back the compressed output.

These AXI connections are routed through the AXI SmartConnect and AXI Interconnect blocks, which handle address mapping and arbitration between the PS and PL modules.

5.6.3 Programmable Logic Clocking

The LZW compression IP was mapped into the Programmable Logic (PL) and driven by the `FCLK_CLK0` clock provided by the Zynq PS. For this design, the clock was configured at 250 MHz, which is the maximum frequency supported by this clock source.

Even at this relatively high operating frequency, the IP core was integrated successfully without timing violations. The test results, which will be detailed in the next section, also demonstrate that the compression IP can reliably perform compression tasks at this frequency.

The `Processor System Reset` block is used to synchronize resets across the design, ensuring that the IP core and all associated logic in the PL are properly initialized in coordination with the PS.

5.6.4 Vivado Resource Utilization

After creating the hardware design and implementing it in Vivado, a report is generated showing how many resources the hardware utilizes on the FPGA. Because of the way Vivado implements a block design, these numbers are not meant to be taken literally but rather provide an order of magnitude estimate to the user.

TABLE 5.6.1 shows the resource utilization of the hardware. This further reinforces the idea that the design is lightweight.

Resource	Usage
Slice LUTs	6.4 %
Slice Registers	3.91 %
Block RAM	4.29 %
DSPs	0.91 %
Slices	10.56 %

Table 5.6.1: Vivado Resource Utilization Summary

5.7 Tests and Results

After completing the hardware design, the system is exported as a bitstream. This bitstream is then used to create a platform in Vitis, which provides the interface needed to program and control the hardware from software.

The final system follows a hardware/software co-design approach. The compression algorithm's computationally intensive core (dictionary lookup, sequence extension, and bit-packing) is implemented in hardware as the IP block, while the PS handles tasks better suited to software:

- File management (reading input files and writing compressed output).
- Initializing and configuring the LZW IP via AXI4-Lite control registers.
- Managing data transfer buffers in DDR memory.

This division of labor between hardware and software allows the system to combine the flexibility of the ARM cores with the speed of the FPGA fabric. It also made it possible to directly compare the performance of a pure software implementation (running on the PS) with the hardware-accelerated implementation using the IP core, under the same development environment.

5.7.1 Test Setup

To evaluate the performance of the LZW compression system, a user-level application was created. The input data consisted of a 3.4 MB text file stored on the SD card connected to the Processing System (PS). Through this program, the IP core could be executed and controlled, managing the entire data flow from input to output. The two different compression modes evaluated were:

- **Software compression:** running entirely on the ARM Cortex-A9 cores of the PS. This version is functionally identical to the hardware implementation, but without the HLS optimization directives.
- **Hardware compression:** where the computationally intensive parts of the algorithm are offloaded to the custom LZW IP implemented in the Programmable Logic (PL).

In addition, dedicated functions were implemented to measure execution time using the global timer of the ZedBoard and calculate the compression ratio in order to validate correctness. The outputs of both approaches were compared bit-for-bit to ensure that the hardware accelerator produced identical results to the software reference.

5.7.2 Results

To recapitulate, the software implementation runs entirely on the ARM Cortex-A9 cores at a frequency of approximately 667 MHz, while the hardware implementation operates at 250 MHz, which is about 2.67 times slower in terms of clock speed. The results of compressing a 3.4 MB text file are as follows:

- Compiler optimization level: -O0
- Software compression time: 2.87 s
- Hardware compression time: 2.481 s
- Compressed size: 1.731 MB
- Compression ratio: 51.56%

Despite the lower operating frequency, the hardware implementation of the IP core achieved a speedup of 13.5% compared to the software implementation, while producing identical compressed output. This result highlights the efficiency of hardware acceleration: even when running at a significantly lower clock frequency, the FPGA-based design outperformed the ARM processor. It also suggests that with larger input files or additional optimizations, the performance gap in favor of the hardware solution would likely become even more pronounced.

This also demonstrates both the efficiency of the implemented design and the robustness of the generated hardware. Running the IP at 250 MHz also provides confidence that the design can sustain high-throughput compression, while still leaving significant FPGA resources available for other tasks in more complex system-level applications.

5.8 HLS Implementation on Kria KV260 Vision AI

The HLS implementation on the Kria KV260 Vision AI was carried out in the same way as on the ZedBoard. The code itself did not require any major modifications from the original IP core, demonstrating that the compression design can be easily ported to other FPGA platforms. This highlights the reusability of the implementation across different hardware targets.

The only addition was a `PIPELINE` directive in the `Dictionary_add` function. This was introduced to better exploit the higher logic resources available on the Kria platform and to improve parallelism in one of the more frequently executed functions of the algorithm.

5.8.1 Resource Utilization

As mentioned, the Kria platform offers significantly higher logic resources, including a much larger number of LUTs, BRAMs, and DSP slices compared to the ZedBoard.

As shown in Table 5.8.1, the LZW compression IP consumes only a small fraction of the total available resources. This further emphasizes the lightweight nature of the design: even on a resource-rich platform such as the Kria KV260, the compression core leaves the vast majority of hardware unoccupied.

Criteria	Guideline	Actual
LUT	70 %	2.35 %
FD	50 %	1.51 %
LUTRAM+SRL	25 %	0.51 %
MUXF7	15 %	0.02 %
DSP	80 %	0.08 %
RAMB/FIFO	80 %	4.17 %
DSP+RAMB+URAM (Avg)	70 %	2.12 %

Table 5.8.1: Resource Utilization Summary for the Kria KV260 Vision AI

5.9 Creation of the Hardware Design

The next step is, of course, to integrate the complete hardware design in Vivado.

5.9.1 Integration of the IP Core

The generated IP was imported into the Vivado IP catalog and then instantiated inside the block design along with the Zynq UltraScale+ Processing System. FIGURE 5.9.1 shows the complete system block design.

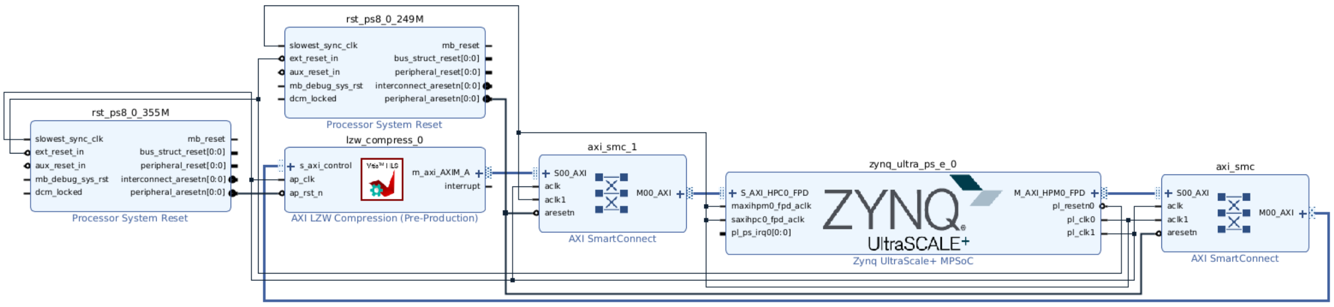


Figure 5.9.1: Vivado block design integrating the LZW Compression IP core with the Zynq UltraScale+ PS

The major difference compared to the ZedBoard is the use of two different clocks: `pl_clk0` and `pl_clk1`, as well as the addition of two AXI SmartConnect IP blocks. The Zynq UltraScale+ PS can provide multiple programmable clocks to the PL, which can theoretically reach a maximum frequency of about 1.6 GHz. This capability allows the IP core to potentially run at much higher speeds than on the ZedBoard.

However, the AXI interfaces, which operate in their own asynchronous clock domain, support maximum frequencies of approximately 333 MHz. This limitation motivated the decision to separate the clocks for the AXI interfaces and the PL logic, ensuring both reliable communication and high-performance operation of the compression core.

After extensive testing, the theoretical operating limit of the IP core was found to be around 600 MHz. Beyond this frequency, the core fails to compress data correctly or exhibits unstable behavior. For this reason, `pl_clk1` was configured at 600 MHz, while `pl_clk0` was set to 250 MHz.

5.9.2 Vivado Resource Utilization

Table 5.9.1 shows the resource utilization of the hardware.

Resource	Usage
CLB LUTs	4.19 %
CLB Registers	3.54 %
Block RAM	6.94 %
DSPs	0.08 %
CLB	7.77 %

Table 5.9.1: Vivado Resource Utilization Summary

5.10 Tests and Results

After exporting the bitstream and creating the platform in Vitis, a user-level application was developed. The only modification made to the code concerned the function used for measuring execution time: instead of relying on a global timer, the implementation now uses the ARM Generic Timer available on the Cortex-A cores.

The software implementation runs entirely on the ARM Cortex A-53 cores at a frequency of approximately 1.333 GHz, while the hardware implementation operates at 600 MHz, which is about 2.221 times slower in terms of clock speed. The results of compressing a 3.4 MB text file are as follows:

- Compiler optimization level: -O0
- Software compression time: 1.435 s
- Hardware compression time: 2.39 s
- Compressed size: 1.731 MB
- Compression ratio: 51.56%

In this case, the software implementation outperformed the hardware by approximately 39.52%. It is important to note that this performance gap cannot simply be attributed to the difference in clock frequencies between the Cortex-A53 cores and the PL logic. The slow-down of the hardware version can be explained by the overhead of communication between the Processing System (PS) and the Programmable Logic (PL), as well as the additional latency introduced by memory transfers across the AXI interconnect. While the compression core itself is efficient, these data transfer and synchronization costs can be significant when compared to the tightly integrated software execution running entirely within the PS.

Chapter 6

Parallel Compression

The idea behind parallel compression is to reduce execution time. Thanks to the lightweight nature of the IP core and the inherent ability of hardware to perform tasks in parallel, the decision was made to experiment with parallelizing LZW compression.

In the following tests, multiple instances of the LZW compressor are executed in parallel on the same input file. This approach involves dividing the initial input into several chunks and assigning each chunk to a separate compressor instance. Theoretically, this can reduce the execution time proportionally to the number of compressors. However, since each chunk is compressed independently, the overall compression ratio may be negatively affected. Additionally, running multiple compressors in parallel naturally requires more FPGA resources.

To achieve this, two possible solutions arise: the first is to update the hardware design by adding multiple IP cores that run in parallel, and the second is to create a single IP core capable of executing multiple compression functions in parallel.

6.1 Multiple IP Cores

6.1.1 Creation of the Hardware Design

The hardware design for parallel compression does not differ significantly from the single–IP core case. Each LZW compression core includes one master AXI interface and one slave AXI interface that must communicate with the Processing System (PS). To support multiple cores, the AXI Interconnect was customized by increasing the number of slave AXI ports. Each of these slave ports connects to the master AXI interface of an LZW core, while the master AXI interface of the interconnect connects back to the PS. A similar configuration was applied to the AXI SmartConnect, where the number of slave AXI interfaces was increased to accommodate all cores.

Figure 6.1.1 illustrates the Vivado block design with two LZW compression IP cores integrated to run in parallel.

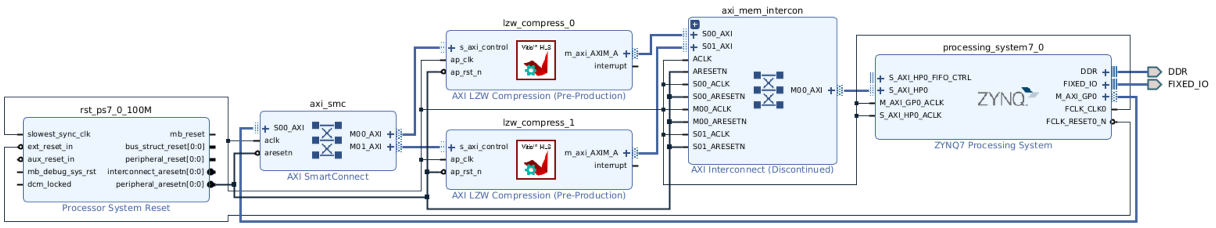


Figure 6.1.1: Vivado block design integrating two LZW Compression IP cores

Through extensive testing, the maximum number of LZW compression IP cores that could be run in parallel was found to be 12. It is important to note that starting from 8 IP cores, the operating clock frequency had to be reduced from 250 MHz to 200 MHz in order to achieve stable operation. This reduction was necessary because the increased logic utilization and routing complexity of multiple cores prevented the design from meeting timing closure at 250 MHz. Lowering the frequency provided sufficient timing margin for reliable communication across the interconnect and ensured correct operation of all cores.

Finally, Table 6.1.1 shows the resource utilization of the hardware for the 12 LZW compression cores.

Resource	**Usage**
Slice LUTs	79.10 %
Slice Registers	51.66 %
Block RAM	51.43 %
DSPs	5.45 %
Slices	99.57 %

Table 6.1.1: Vivado Resource Utilization Summary

6.1.2 Tests and Results

After exporting the bitstream and creating the platform in Vitis, a user-level application was developed. The program is responsible for managing data transfers, initiating and controlling the execution of all the LZW compression IP cores in parallel, and collecting the compressed output from each one. To achieve this, the input file is divided into chunks, with each chunk assigned to a different IP core. The application then coordinates the execution of all cores, waits for their completion, and merges the resulting compressed segments. In addition, it records execution time, validates correctness against the software reference, and evaluates the trade-off between faster execution and potential loss in compression efficiency caused by chunking the input data.

The compressed chunks generated by the parallel LZW IP cores are collected and written into a single `.bin` file. To ensure that the file can later be interpreted correctly, a simple format was defined.

The first line of the file contains metadata: it begins with the total number of chunks, followed by the size of each individual chunk, all separated by spaces. This information allows the decompressor (or any post-processing tool) to identify how many chunks are present and how many bytes to read for each one.

The second line of the file contains the concatenated compressed data, written sequentially as it was produced by the IP cores. Each segment corresponds to the compressed output of one chunk, in the same order as declared in the metadata line.

An example structure of the output file is shown below:

```
12  123  123  123  123  123  123  123  123  123  123  123  123
[chunk 1][chunk 2][chunk 3]...[chunk 12]
```

This structure ensures that the compressed output from multiple IP cores can be easily reassembled, while preserving enough information to enable correct decompression or further analysis.

FIGURE 6.1.2 shows the execution time of the LZW compression when increasing the number of IP cores running in parallel. As expected, the execution time decreases as more cores are added, since the input data is divided into smaller chunks that are processed simultaneously.

The results show a clear performance improvement up to 12 cores, where the execution time is reduced from approximately 2.482 seconds with a single core to around 0.29 seconds with 12 cores. However, the reduction in execution time is not perfectly linear: for example, doubling the number of cores from 6 to 12 does not halve the execution time. This observation is consistent with **Amdahl's Law**, which describes the theoretical speedup of a program as a function of the fraction of code that can be parallelized.

Amdahl's Law states that the maximum speedup $S(N)$ achievable with N parallel units is given by:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

where P is the proportion of the workload that can be parallelized, and $(1-P)$ is the inherently sequential part. In the case of LZW compression, most of the dictionary operations are parallelized across the IP cores, but certain tasks remain sequential: splitting the input, coordinating execution, handling AXI transfers, and merging the compressed chunks. These sequential components limit the maximum achievable speedup, even if more cores are added.

It is important to note that the loss of near-linear scaling at higher core counts (for example, when moving from 8 to 12 cores) is not primarily caused by large sequential portions in the user-level application, since the results consistently show speedup as cores are added. Instead, this behavior directly illustrates the principle highlighted by Amdahl's Law: even when most of the workload is parallelizable, the benefits of adding more parallel units diminish over time. Beyond a certain point, the speedup achieved by adding extra IP cores becomes so small that it is not worth the additional hardware resources required. In other words, while parallelization accelerates compression, Amdahl's Law shows that there is a practical upper limit to the performance gains.

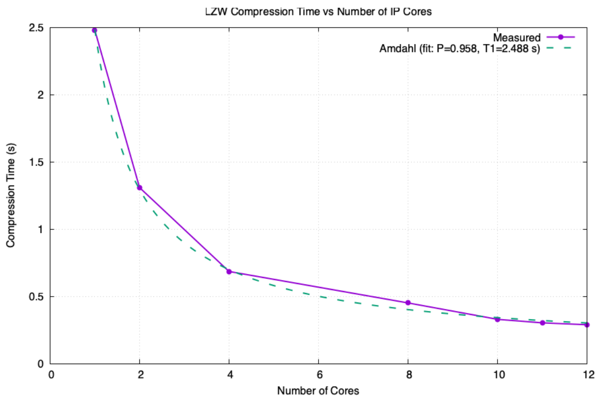


Figure 6.1.2: Measured execution time of parallel LZW compression compared with Amdahl's Law prediction ($P \approx 0.95$). The theoretical curve is obtained by converting Amdahl's speedup into execution time using the single-core runtime as baseline.

6.2 Multi-Function Single IP Core

The second explored solution was trying to execute a parallel compression in a single IP Core.

6.2.1 HLS Implementation

The first modification required was in the HLS implementation of the LZW Compression IP Core. Specifically, the function responsible for compression, `lzw_compress`, was no longer used as the top-level function. Instead, it was replaced by a new function called `top_parallel_lzw`. This function is responsible for receiving multiple chunks of input data and distributing each chunk to an instance of the `lzw_compress` function. The compressed outputs from each instance are then collected and written back.

This parallelism is enabled in HLS through the use of the `DATAFLOW` directive. The `DATAFLOW` directive allows different functions or loops within the design to operate concurrently, provided that their data dependencies are respected [4]. In practice, this means that as soon as one stage of the computation produces data, the next stage can immediately begin processing it, without waiting for the entire function to complete. This results in a task-level pipeline that significantly improves throughput and makes parallel execution possible.

As for memory interfacing, each input and output in this design uses the same master AXI interface, similar to the previous implementations of single-core and multi-core compression. Additional tests were also carried out by separating the AXI master interface, assigning a dedicated AXI interface to each input and output. However, the results showed no significant performance improvement. Since multiple AXI interfaces increase resource utilization and design complexity without yielding measurable gains, the approach with a single shared AXI interface was preferred.

6.2.2 Creation of the Hardware Design

Regarding the hardware design, the integration of this modified LZW Compression IP is similar to the previous tests with multiple IP cores. The main difference lies in the maximum degree of parallelism achievable: in this case, up to 10 compression processes can run in parallel, compared to 12 when using separate IP cores.

Table 6.2.1 shows the resource utilization of the hardware design containing the IP core with 10 parallel functions, using a shared master AXI interface for both input and output.

Resource	**Usage**
Slice LUTs	66.34 %
Slice Registers	43.26 %
Block RAM	42.86 %
DSPs	4.55 %
Slices	95.84 %

Table 6.2.1: Vivado Resource Utilization Summary for LZW Parallel Compression with 10 functions using shared MAXI interface.

Table 6.2.2 shows the resource utilization of the hardware design containing the IP core with 10 parallel functions, using a different master AXI interface for both input and output.

6.2.3 Tests and Results

After exporting the bitstream and creating the platform in Vitis, a user-level application was developed. As before, the program is responsible for managing data transfers, initiating and controlling the execution of the LZW compression IP core, and collecting the compressed outputs.

Resource	Usage
Slice LUTs	84.24 %
Slice Registers	56.00 %
Block RAM	46.43 %
DSPs	4.55 %
Slices	100.00 %

Table 6.2.2: Vivado Resource Utilization Summary for LZW Parallel Compression with 10 functions using different MAXI interface.

The input data stream is also divided into chunks in this program, and the compressed chunks generated by the LZW Compression IP Core are written into a single `.bin` file following the same format as before (see SECTION 6.1).

FIGURE 6.2.1 shows the execution time of the LZW compression as the number of functions running in parallel increases, using a shared master AXI interface. Similarly, FIGURE 6.2.2 presents the execution time when using separate master AXI interfaces for each function.

As expected, using a shared master AXI interface produces results comparable to those obtained with multiple independent IP cores connected through a shared interface. In both cases, the communication bottleneck through a single AXI channel limits the achievable speedup.

However, when the design is configured with separate AXI master interfaces for each function, the results show a noticeable improvement in performance. In this setup, input and output transfers can occur simultaneously, which reduces communication overhead and shortens the overall execution time. This demonstrates one of the advantages of consolidating multiple functions within a single IP core: while the maximum number of parallel functions (10) is slightly lower than the 12-core design, the ability to overlap memory transfers leads to faster compression times overall.

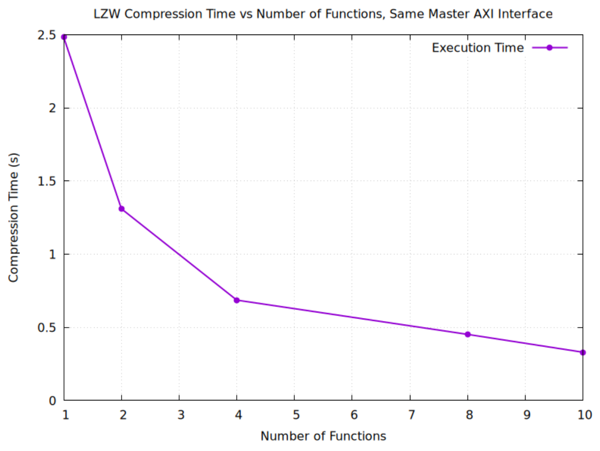


Figure 6.2.1: LZW Compression Time vs Number of Functions, Same Master AXI Interface for Input and Ouput

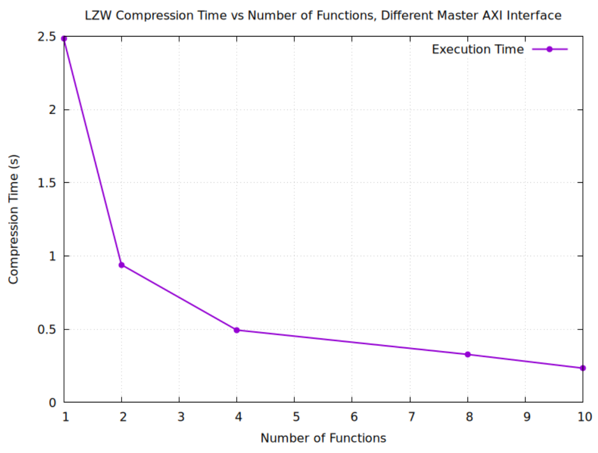


Figure 6.2.2: LZW Compression Time vs Number of Functions, Different Master AXI Interface for Input and Ouput

Chapter 7

Conclusion

This internship achieved its primary goal: studying the LZW compression algorithm and implementing it in both software and on FPGA using High-Level Synthesis (HLS). A complete software baseline (compression and decompression) was developed and validated, then translated into an HLS IP core, packaged and integrated in Vivado, and exercised from Vitis on real hardware. All initial objectives were met, and the work went further by exploring parallel compression and by porting and testing on two platforms (ZedBoard and Kria KV260 Vision AI). Briefly, the measurements confirmed that end-to-end performance depends not only on the accelerator itself but also on data movement and interface choices; nevertheless, the full flow from algorithm to accelerator was implemented and verified.

I personally found this internship both challenging and rewarding. It pushed me to go far beyond my initial knowledge: at the start, concepts such as HLS directives, AXI interconnects, PS–PL clocking, timing closure, and hardware debugging were completely new to me. Over time, I learned to navigate these topics and turn a pure software algorithm into a working, benchmarked hardware accelerator. I genuinely enjoyed the process of building reusable technical blocks — from synthesizable HLS code and packaged IP, to Vivado block designs, Vitis test applications, and validation tooling — and seeing them come together on real hardware was especially satisfying. This journey was not only about technical implementation, but also about learning how to approach problems systematically and persist through complexity.

What I do regret, however, is that the 20–50 page constraint forced me to skim over or completely omit many aspects of the work I carried out. I would have liked to show in greater detail the challenges, the trial-and-error, and the smaller victories that shaped the final result. That said, I take pride in having delivered a complete and validated workflow from algorithm to accelerator within these constraints. More importantly, I leave this internship with the confidence that the skills I acquired form a solid foundation for future projects and deeper explorations of this domain.

References

[1] H. Wennborg, *Zip file format: Lzw shrink decompression.* [Online]. Available: https://www.hanshq.net/zip2.html.

[2] D. Inc., *Zedboard: Zynq-7000 arm/fpga soc development board reference manual.* [Online]. Available: https://digilent.com/reference/programmable-logic/zedboard/start.

[3] A. Xilinx, *Kria kv260 vision ai starter kit: Product information and technical resources.* [Online]. Available: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html.

[4] Xilinx, *Vitis high-level synthesis user guide.* [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.