Hardware/Software Processing Architecture for Statistical Data Compression Algorithms

Student
Virgilio Zúñiga Grajeda
Supervisors
Dra. Claudia Feregrino Uribe and Dr. René A. Cumplido Parra

Computer Science Department
Instituto Nacional de Astrofísica, Óptica y Electrónica
Tonantzintla, Puebla. México

November, 2005

Contents

Chapter 1

Introduction

In recent years, mobile and wireless communications have gained interest between users who need to maintain contact with their coworkers and relatives. The use of cell phones, Personal Digital Assistants (PDAs), Global Positioning Systems (GPSs) and other mobile devices has grown since they become more handy, useful and popular.

But the amount of data transfered tends to grow and the network bandwidth can be affected.

The nature of the information that flows throughout the communication networks has diversified, not only voice conversations can be hold, but data...

1.1 Data compression in electronic communications

The need of transmit large amounts of data with electronic devices. Real time. Throughput. Trade-off between performance, silicon area and power consumption.

1.2 Platform for algorithm implementation

Application Specific Integrated Circuit (ASIC). General-purpose processors. FPGA

1.2.1 Field Programmable Gate Array (FPGA)

FPGA allow rapid prototyping using custom logic structures.

1.3 Objectives

Design a custom coprocessor to execute the data compression algorithms reusing hardware resources. Define a set of instructions for a processor to control the coprocessor.

1.4 Hardware or Software?

Part of the research is the design of efficient implementations. When an implementation consumes a lot of resources or takes a lot of time to execute is not very helpful. There are there main requirements when the algorithms are build for hardware: Small silicon area, high throughput and low power consumption. Commonly, there are two alternatives to implement an algorithm. The first choice is a pure software solution, this is done by coding the operations as a set of arranged instructions defined for a general purpose microprocessor. The second alternative consists of designing a custom coprocessor that is controlled by a main processor...

1.5 Methodology

Implement data compression algorithms in Matlab. Study different hardware architectures to find similar structures. Design the coprocessor where different compression algorithms share a common structure. Attach the coprocessor to a General-purpose processor and define the instructions. The system is implemented in VHDL and simulated with Modelsim.

1.6 Dissertation overview

This dissertation is organized as follows: The next chapter presents this and the last chapter presents that.

Chapter 2

A review of data compression

This chapter provides the reader with the background information needed to understand the dissertation. An introduction to data compression is given including their applications, concepts like compression ratio, compression models as well as a the terminology of this dissertation. Also a brief description of the methods considered in this work is reviewed.

2.1 Introduction to data compression

Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output or the compressed stream) that has a smaller size. A stream is either a file or a buffer in memory. There are many known methods for data compression. They are based on different ideas, area suitable for different types of data, and produce different results, but they are all based on the same principle, namely they compress data by removing *redundancy* from the original data in the source file.

Data compression has important application in the areas of data transmission and data storage. Many data processing applications require storage of large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of wireless communication networks is resulting in massive transfer of data over communication links.

Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium.

The essential figure of merit for data compression is the *compression ratio*, or ratio of the size of a compressed file to the original uncompressed file. For example, suppose a data file takes up 100 Kilobytes (KB). Using data compression software, that file could be reduced in size to, say, 50 KB, making it easier to store on disk and faster to transmit over a communication channel. In this specific case, the data compression software reduces the size of the data file by a factor of two, or results in a compression ratio of 2:1.

There are two kinds of data compression models, *lossy* and *lossless*. Lossy data compression, works on the assumption that the data does not have to be stored perfectly. Much information can be simply thrown away from images, video data, sound and when uncompressed such data will still be of acceptable quality. Lossless compression, in contrast, is used when the data has to be uncompressed exactly as it was before compression. Text files (specially files containing computer programs) are stored using lossless techniques, since losing a single character can in the worst case make the text dangerously misleading. Archival storage of master sources for images, video data, and audio data generally needs to be lossless as well. However, there are strict limits to the amount of compression that can be obtained with lossless compression. Lossless compression ratios are generally in the range of 2:1 to 8:1. Lossy compression ratios can be an order of magnitude greater than those available from lossless methods.

The question of which is "better", lossless or lossy techniques, is pointless. Each has its own uses, with lossless techniques better in some cases and lossy techniques better in others. Even given a specific type of file, the contents of the file, particularly the orderliness and redundancy of the data, can strongly influence the compression ratio. In some cases, using a particular data compression technique on a data file where there is not a good match between the two can actually result in a bigger file. For this dissertation, the lossless techniques are reviewed.

Some comments on terminology for the dissertation:

- Since most data compression techniques can work on different types of digital data, such as characters in texts or bytes in images, data compression literature speaks in general terms of compressing *symbols*.
- Many of the examples in the dissertation refer to compressing *characters*, simply because a text file is very familiar to most readers. However, in general, compression algorithms are not restricted to compressing text files. Data bytes are data bytes, regardless of whether they define text characters, graphics data or measurement data being returned from a space probe.
- Similarly, most of the examples talk about compressing data in *files*, just because most readers are familiar with that idea. However, in practice, data compression applies just as much to data transmitted over a modem or other data communications link as it does to data stored in a file. There is no strong distinction between the two as far as data compression is concerned, and the term *stream* can be used to cover them all.
- Data compression literature also often refers to data compression as data *encoding*, and of course that means data decompression is often called *decoding*. This document tends to use the two sets of terms interchangeably.

In the next sections of the chapter four data compression algorithms are explained, the first two are presented to understand the goal of the last one. The last two algorithms are the goal of this dissertation as mentioned in Chapter 1.

2.2 Run-Length encoding

Run Length Encoding [1] (often referred as RLE) is one of the simplest data compression techniques, taking advantage of repetitive data. Some sources have concentrations of characters, these repeating characters are called runs. In this instance, sequences of elements $X_1, X_2, \ldots, X_n$ are mapped to pairs $(c_1, l_1), (c_2, l_2), \ldots, (c_n, l_n)$ where c_i represents the repeated character and l_i the length of the ith run of that character. For example, a source string of

"AAAABBBBBCCCCCCCCDEEEE"

could be represented with

(A,4),(B,5),(C,8),(D,1),(E,4)

Four "A"s are represented as A4. Five "B"s are represented as B5 and so forth. The code represents 22 bytes of data with 10 bytes, achieving a compression ratio of

22 bytes / 10 bytes = 2.2.

This method works fine when there are many runs in the source data, but most of the data sources (like text) have different neighbor characters. For example

"MyCatHasFleas"

It would be encoded

(M,1),(y,1),(C,1),(a,1),(t,1),(H,1),(a,1),(s,1),(F,1),(l,1),(e,1),(a,1),(s,1)

Here 13 bytes are represented with 26 bytes achieving a compression ratio of 0.5. The original data is actually expanded by a factor of two. This problem can be solved representing unique strings of data as the original strings and run length encode only repetitive data. This is done with a special prefix character to flags runs. Runs are then represented as the special character followed by the data and the count.

2.3 Move-to-Front coding

The basic idea of the Move-to-Front (MTF) method [2] is to maintain the alphabet A of symbols as a list where frequently occurring symbols are located near the front. A symbol "s" is encoded as the number of symbols that precede it in this list. Thus if A=("t", "h", "e", "s",...) and the next symbol in the input stream to be encoded is "e", it will be encoded as 2, since it is preceded by two symbols. There are several possible variants to this method, the most basic of them adds one more step: After the symbol "s" is encoded, it is moved to the front of list A. Thus, after encoding "e", the alphabet

Figure 2.1: LZ77 Window.

is modified to A=("e", "t", "h", "s",...). This move-to-front step reflects the hope that once "e" has been read from the input stream, it will be read many more times and will, at least for a while, be a common symbol. The MTF method is *locally adaptive*, since it adapts itself to the frequencies of symbols in local areas of the input stream.

The method thus produces good results if the input stream contains concentrations of identical symbols (if the local frequency of symbols changes significantly from area to area in the input stream). This is called "the concentration property".

2.4 LZ77 method

In 1977 Lempel and Ziv [3] presented an adaptive dictionary-based algorithm for sequential data compression called LZ77. The fundamental concept of this algorithm is to replace variable-length phrases in the source data with pointers to a dictionary. The algorithm constructs the dictionary dynamically by shifting the input stream into a window that is divided in two parts, see Figure 2.1.

The part on the left is called the *search buffer*. This is de current dictionary, and it always includes symbols that have recently been input and encoded. The part on the right is the *look-ahead buffer*, containing text yet to be encoded. While shifting the input stream into the search buffer it is compared with the existing data in the dictionary to find the maximum-length-matching phrase. Once such a matching phrase is found, an output codeword or token $T = (T_o, T_l, T_n)$ is generated. Each token contains three elements: The offset T_o that points to the starting position of the matching phrase in the dictionary, the length T_l of the matching string and the next source data symbol T_n immediately following the matching string. In the next cycle following the generation of the token, a new source data string enters the system, and a new matching process begins and proceeds in the same way until the source data is completely encoded. Starting from the idea that recent data patterns are expected to appear in the near future, and

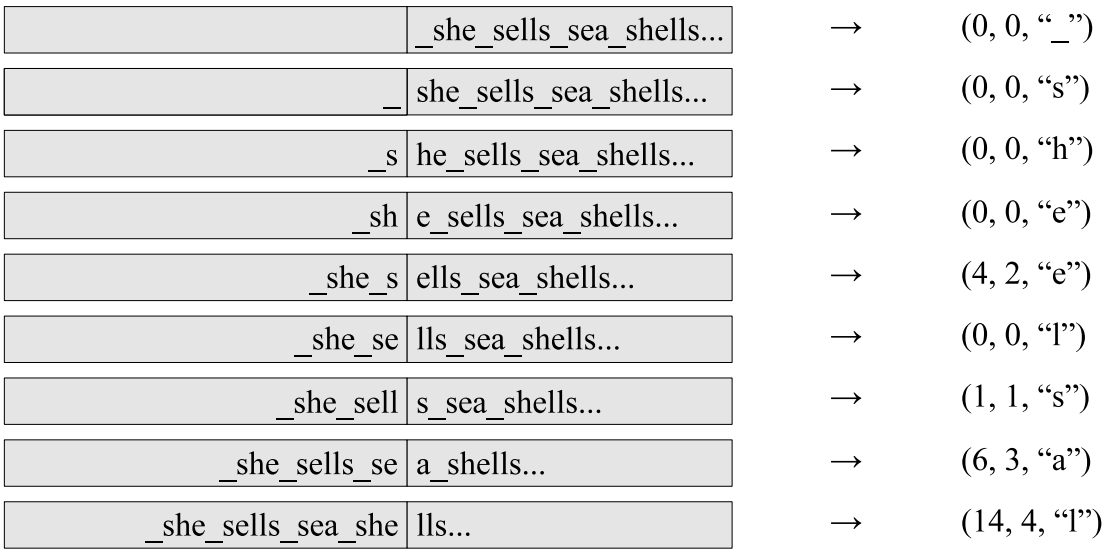


Figure 2.2: LZ77 compression process.

the latest data is contained in the dynamic dictionary, LZ77 can often replace a long and frequently encountered string with a shorter code. An example of the LZ77 encoding process is shown in Figure 2.2.

In the first four steps the search buffer is empty, thus the symbols, "_", "s" and "h" are encoded with a token with zero offset, zero length and the unmatched symbol. In the next two steps the "_" and "s" symbols are found, no token is generated still. In the next step, the symbol "e" is found but is not part of the string "_s" so the token (4,2,"e") is constructed. The process continues until all the input string has been coded. Clearly, a token of the form (0,0,...) which encodes a single symbol does not provide good compression but this kind of tokens appear only at the beginning of the process. In the Figure 2.2, it can be noticed that the more data contains the search buffer, the less tokens are needed to represent a string.

The decoder is much simpler than the encoder. It has to maintain a buffer, equal in size to the search buffer. The decoder inputs a token, finds the match in its buffer, writes the match and the third token field on the output stream. Then shifts the matched string and the third field into the buffer.

2.5 Burrows-Wheeler method

Burrows and Wheeler [4] presented in 1994 a block-sorting lossless data compression algorithm which speed was comparable to algorithms based on LZ77 techniques and compression rates were close to the best known compression rates.

The algorithm transforms a string S of n characters or symbols by forming the n rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last character of each rotation. A string L is formed from these characters, where the ith character of L is the last character of the ith sorted rotation. In addition to L, the algorithm computes the index I of the original string S in the sorted list of rotations. This operation is called Burrows-Wheeler Transform.

The transformation does not itself compress the data, but reorders the characters to make them easier to compress with simple algorithms such as MTF followed by a RLE. String L can be efficiently compressed because it contains concentrations of symbols and it is possible to reconstruct the original string S from L and the index I.

To understand how string L is created from S, and what information has to be stored in I for later reconstruction, a running example is given:

Let S = "_ s h e _ s e l l s _ s e a _ s h e l l s"

The encoder constructs an $n \times n$ matrix where it stores string S in the top row, followed by n - 1 copies of S, each cyclically shifted (rotated) one symbol to the left (Figure 2.3).

The matrix is then sorted lexicographically by rows, producing the sorted matrix of Figure 2.4. Notice that every row and every column of each of the two matrices is a permutation of S and thus contains all n symbols of S. The permutation L selected by the encoder is the last column of the sorted matrix, in this example "s e s a e h s s h s s e e l l l l _ _ _ _". The only other information needed to eventually reconstruct S from L is the row number of the original string in the sorted matrix, which in this case is 3 (row and column numbering starts from 0). This number is stored in I.

It is easy to see why L contains concentrations of identical symbols. Assume that the words *bail*, *fail*, *hail*, *jail*, *mail*, *nail*, *pail*, *rail*, *sail*, *trail* and *wail* appear somewhere

in S. After sorting, all the permutations that start with *il* will appear together. All of them contribute an a to L, so L will have a concentration of a's. Also, all the permutations starting with *ail* will end up together, contributing to a concentration of the letters *bfhjmnprstw* in one region of L. It is worth noting that the larger n, the longer the concentrations of symbols and the better the compression. For lack of space, the decoder phase is not described. The interested reader is referred to [1].

_ s h e _ s e l l s _ s e a _ s h e l l s
s h e _ s e l l s _ s e a _ s h e l l s _
h e _ s e l l s _ s e a _ s h e l l s _ s
e _ s e l l s _ s e a _ s h e l l s _ s h
_ s e l l s _ s e a _ s h e l l s _ s h e
s e l l s _ s e a _ s h e l l s _ s h e _
e l l s _ s e a _ s h e l l s _ s h e _ s
l l s _ s e a _ s h e l l s _ s h e _ s e
l s _ s e a _ s h e l l s _ s h e _ s e l
s _ s e a _ s h e l l s _ s h e _ s e l l
_ s e a _ s h e l l s _ s h e _ s e l l s
s e a _ s h e l l s _ s h e _ s e l l s _
e a _ s h e l l s _ s h e _ s e l l s _ s
a _ s h e l l s _ s h e _ s e l l s _ s e
_ s h e l l s _ s h e _ s e l l s _ s e a
s h e l l s _ s h e _ s e l l s _ s e a _
h e l l s _ s h e _ s e l l s _ s e a _ s
e l l s _ s h e _ s e l l s _ s e a _ s h
l l s _ s h e _ s e l l s _ s e a _ s h e
l s _ s h e _ s e l l s _ s e a _ s h e l
s _ s h e _ s e l l s _ s e a _ s h e l l

Figure 2.3: Matrix of cyclic shifts of S.

_ s e a _ s h e l l s _ s h e _ s e l l s
_ s e l l s _ s e a _ s h e l l s _ s h e
_ s h e _ s e l l s _ s e a _ s h e l l s
_ s h e l l s _ s h e _ s e l l s _ s e a
a _ s h e l l s _ s h e _ s e l l s _ s e
e _ s e l l s _ s e a _ s h e l l s _ s h
e a _ s h e l l s _ s h e _ s e l l s _ s
e l l s _ s e a _ s h e l l s _ s h e _ s
e l l s _ s h e _ s e l l s _ s e a _ s h
h e _ s e l l s _ s e a _ s h e l l s _ s
h e l l s _ s h e _ s e l l s _ s e a _ s
l l s _ s e a _ s h e l l s _ s h e _ s e
l l s _ s h e _ s e l l s _ s e a _ s h e
l s _ s e a _ s h e l l s _ s h e _ s e l
l s _ s h e _ s e l l s _ s e a _ s h e l
s _ s e a _ s h e l l s _ s h e _ s e l l
s _ s h e _ s e l l s _ s e a _ s h e l l
s e a _ s h e l l s _ s h e _ s e l l s _
s e l l s _ s e a _ s h e l l s _ s h e _
s h e _ s e l l s _ s e a _ s h e l l s _
s h e l l s _ s h e _ s e l l s _ s e a _

Figure 2.4: Matrix sorted lexicographically.

Chapter 3

Hardware approaches

This chapter describes the strategies proposed in the past to design hardware implementations of the LZ77 and Burrows-Wheeler algorithms.

3.1 LZ77 method

To speed up the massive comparisons on general-purpose processors, traditional software algorithms, including hash table lookup and tree-structured searching, have been developed and applied in storage systems [5]. However, for real-time applications dedicated parallel architectures are used to support higher throughputs. Two major hardware implementation methods are the Content-Addressable Memory (CAM)-based approach [6] [7] and the systolic array approach [8] [9]. In the next subsections, both methods are explained.

3.1.1 Content-Addressable Memory approach

As mentioned in the previous section, the main component of this approach is the Content-Addressable Memory (CAM). The CAMs are hardware search engines that are much faster than algorithmic approaches for search-intensive applications. A CAM is composed of a group of cells, each cell has a storage device (register) and a comparator, see Figure 3.1.

Unlike standard computer memory (Random Access Memory) a data word is read

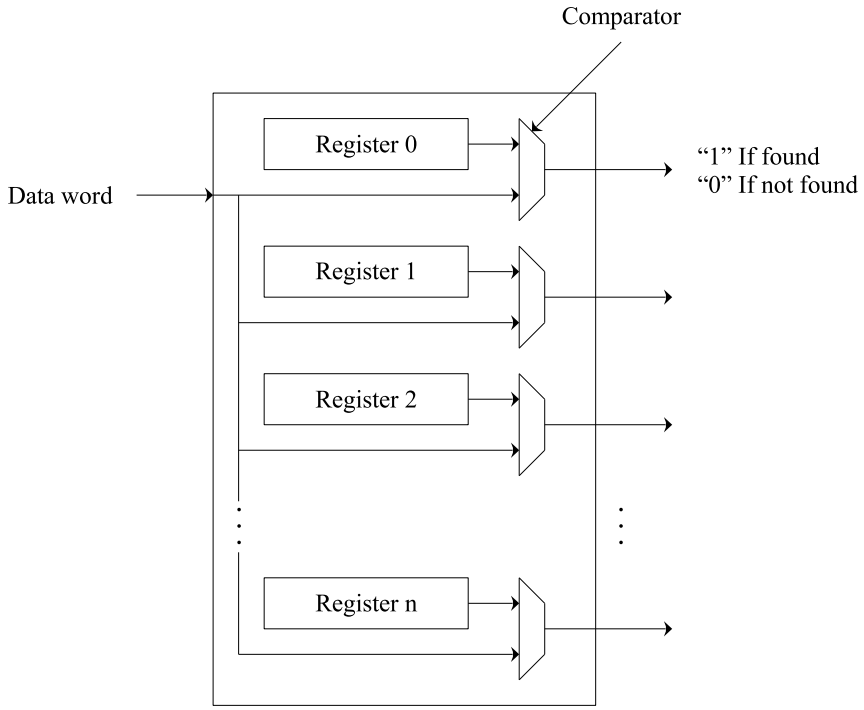


Figure 3.1: Content-Addressable Memory (CAM).

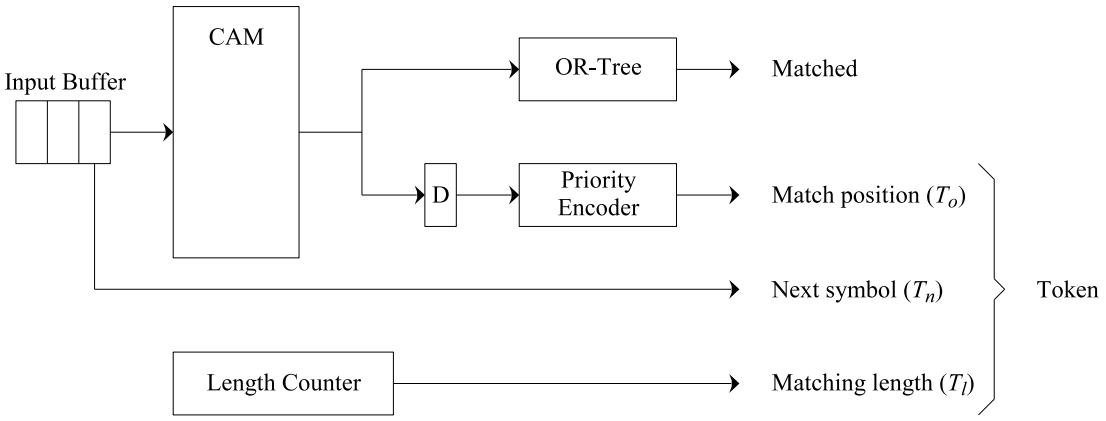


Figure 3.2: CAM-based LZ77 encoder.

instead of an address, the CAM searches in every cell to see if that data word is stored. If the data word is found, the CAM returns a list of one or more storage addresses where the word was found. As all the comparisons are in parallel, the search operation can be completed in a single clock cycle. The dictionary of the LZ77 encoder can be implemented using a CAM to improve the searching process. Figure 3.2 shows the architecture of the CAM-based LZ77 encoder.

The matching process for each source symbol can be pipelined into two stages. In the first stage, a source data symbol is fed into the CAM array to be compared with all the dictionary components. Each CAM cell generates its own match result. These match results are collectively encoded to a matching position pointer and a global *Matched* signal can be realized in the same cycle as the comparison operation using a simple $(log_2 N)$-stage OR-tree. If a match occurs, the corresponding matching position address can be resolved by a $(log_2 N)$-stage Position Encoder in the second stage. The Position Encoder in the second stage is a priority encoder such that when several inputs are logically-1 in the same cycle, only the one with the highest priority is selected as the output.

Priorities of the inputs do not affect the compression performance. They can be assigned in an ascending or descending order according to the indexes of inputs to minimize the encoder complexity. In a pipelined fashion, the compared data also shifts into the end of the CAM array to update the dictionary, and the next source symbol is

	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Symbol 1	Matching	Position Encoding		
Symbol 2		Matching	Position Encoding	
Symbol 3			Matching	Position Encoding

Table 3.1: Pipelines.

injected into the system to start the next comparison operation. Note that in this two-stage pipeline scheme, the output stage follows immediately after the Matched signal is disabled because the T_o and T_l elements of the output codeword are available at the same clock edge.

The sliding dictionary in this scheme can be implemented using a sliding pointer representing the current writing address in the CAM. The pointer also provides an offset for the match position encoder, Note that since the goal of the comparisons is to find the maximum-length-matching string, the matching result of the source symbol in each cell as to propagate to the comparison processes of the next source symbol. Hence, the actual match result in each cell is obtained by ANDing its own match result in the current cycle and the delayed match result of the previous cell in the earlier cycle. In addition, the complement of the *Matched* signal is used to set all the string match results to 1 in order to start the next string matching operation.

The advantage of CAM approach is the high throughput brought by parallelism. The processing rate of source data is 1 symbol/cycle. Compared to the software approaches where it takes at least $O(log_2 N)$ cycles to complete searching in an N-entry dictionary for one single source symbol, the throughput of CAM approach can outperform the C-programs running on fast and powerful CPUs even with slower clock rates. Therefore, CAM-based LZ77 compression hardware is popular in many application specific integrated circuit (ASIC) implementations, such as IBM ALDC compressor core [10] [11]. For ACS implementation, recently released Altera APEX 20KE programmable logic device (PLD) has built-in CAM to achieve better performance over discrete off-chip CAM approach [12]. However, in general PLDs without on-chip or off-chip CAM, realizing large CAM array is expensive because the synthesis of specialized memory de-

vice on current FPGA structure is not area-efficient. This makes the dictionary size a critical balancing factor since it hast to be large enough (normally 512 to 4096 entries) to obtain good compression ratio for LZ-based data compression. Another drawback for the CAM approach is the low resource utilization. Normally, remaining CAM cells are idle during latter cycles of a long string matching process.

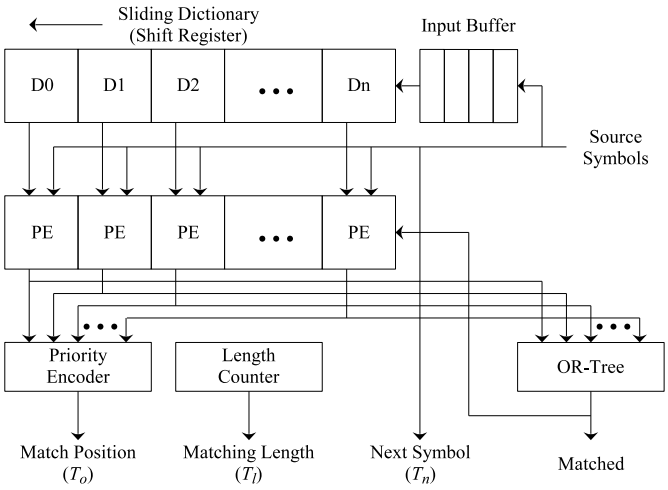
3.1.2 Systolic array approach

In searching for the maximum-length-matching phrase in the LZ77 data compression, source symbols have to be processed in the original order to ensure the correct sequential relationships after reconstruction. This leads to the data dependencies among comparisons of successive source symbols and adjacent dictionary contents. As a result, systolic arrays of processing elements can be applied to achieve better area and power efficiency [9] The idea is to separate the comparators from the CAM-based dictionary cells, and to balance the trade-off between throughput and the number of array components. A special high-performance case proposed in [9] with the same throughput as the CAM-approach is shown in Figure 3.3(a). In this architecture, the sliding dictionary is implemented in shift registers, and each processing element (PE) executes the string comparison function. The detailed structure of a PE is shown in Figure 3.3(b), with the match result of each PE activated by its previous output to account for string matching

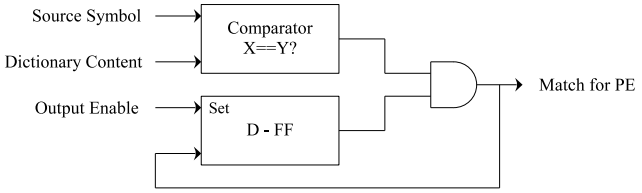
The timing diagram of this high-performance systolic array LZ77 encoder is the same as that of the CAM approach in Figure 3.1. The major advantage over the CAM approach is the flexibility of the processing elements. Since the PEs are now detached form the memory cells, we can schedule idle PEs for other purposes such as error detection and thus increase the hardware utilization.

3.2 Burrows-Wheeler method

The most complex task of the BWT algorithm is its lexicographic sorting of n cyclic rotations of a given string of n characters. In the next subsections the Suffix sorting approach and the architecture that calculates the suffixes are explained.



(a) Architecture



(b) PE structure

Figure 3.3: Systolic Array LZ77 encoder.

3.2.1 Suffix sorting approach

Suffix sorting is the problem of lexicographically ordering all the suffixes of a string. The suffixes are represented by integers denoting their starting positions. Suffix sorting has at least two important applications. One is construction of a suffix array and another is in data compression with the Burrows-Wheeler transform where sorting is a computational bottleneck and an efficient sorting method is crucial for any implementation of this compression scheme. In the next subsection the Weavesorter algorithm is presented. It is a hardware architecture used to calculate the suffixes of a string solving part of the Burrows-Wheeler Transform.

3.2.2 The Weavesorter machine

The Weavesorter algorithm was developed by Amar Mukherjee et al. and is described in [13]. The Weavesorter consists of a bidirectional shift register with a comparator for each pair of registers (or cells), it is capable to do the following operations: *shift-left*, *shift-right* and *compare/swap*. The idea of the weavesorting algorithm is to shift the input string character by character into the Weavesorter starting from the left edge and do a *compare/swap* operation after each shift step. This operation compares each pair of the characters and swaps them if the left character is larger than the right one. After the string is completely inserted, the smallest character of the whole string will be in the leftmost cell of the Weavesorter. The rest of the string is not yet sorted but presorted. Now, the shift direction is changed to shift left and the string is shifted out of the Weavesorter. While shifting out, each shift operation is followed by a *compare/swap* operation. This guarantees that always the smallest character of the substring which is still in the Weavesorter is read out of the Weavesorter. So the output of the Weavesorter is a sorted version of the original input string.

While outputting the first string, another string can be inputted from the other side. The largest character of this string will always be in the rightmost cell of the Weavesorter. After changing the shift direction this string will be outputted on the right side of the Weavesorter, beginning with the largest character of this second string. Thereby, a hardware utilization of almost 100 percent can be achieved. In Figure 3.4, a Weavesorter with eight basic cells is shown.

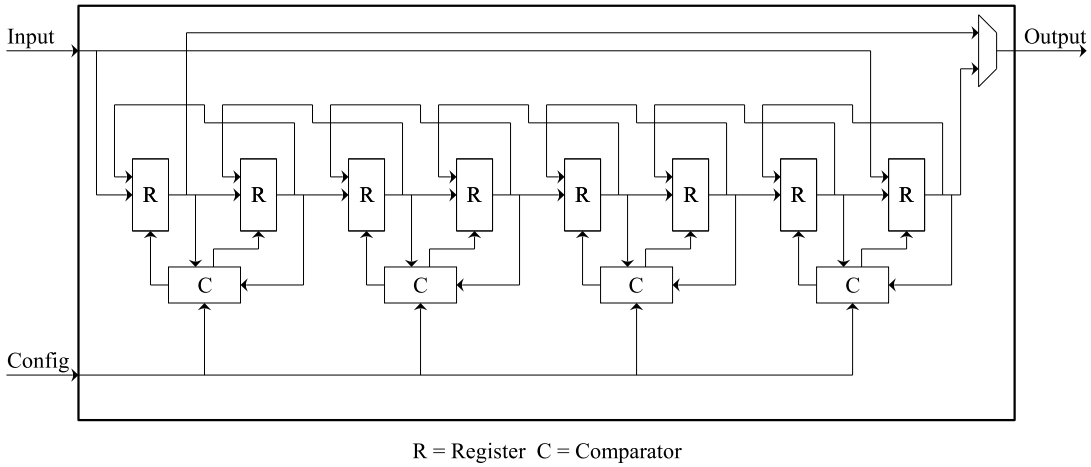


Figure 3.4: Weavesorter with 8 cells.

It is not sufficient to sort according to the first character of a cyclic rotation. This is only the first step. After sorting according to the first character there might be rows starting with the same character. These rows have to be sorted according to the second character of the cyclic rotation. This has to be done until all the sorting characters are different. To use the Weavesorter for BWT certain requirements have to be created. The input string must be stored in a memory. By doing this each character gets its own address. With this address the successor of a character in the original string can be easily accessed by incrementing the address. The character and his address are stored in the Weavesorter. The address is only carried along with the character but does not influence the sorting. After the first iteration of the Weavesorter, the output string might contain groups of the same characters which have to be sorted again according to their successor. Therefore the successors of the characters in the output string are inserted into the Weavesorter. This can be done from the other side of the Weavesorter. To prevent mixing of characters from different columns a control bit is inserted which blocks the *compare/swap* operation. Note that as soon as the first character drops out of the Weavesorter, its successor in the original string is read out of the memory and inserted from the other side. This is done until all characters are separated by a control bit. Then no further sorting is necessary. Actually what is stored in the Weavesorter at the end of the sorting is the first column of the sorted matrix. Only that as in each

iteration the successor of each character was processed the first column is represented by the X^{th} successor of each character, where X represents the number of iterations. To get the last column, the number of iterations minus one must be subtracted from the addresses stored within the Weavesorter.

Chapter 4

Problem statement

This dissertation engages in the implementation of lossless data compression algorithms sharing hardware structures to increase resource utilization. Also the interconnection of these architecture to a general purpose processor is studied. The details of the tackled problems as well as the proposed solution are described in this chapter.

4.1 The resource utilization

As described in Chapter 3, the popularity of the LZ77 CAM approach is due to its high data throughput achieved by parallelism. However, large CAM arrays on PLDs are expensive because memories are note area-efficient on these devices. Also this approach has a low resource utilization. On the other hand, the systolic array approach separates the cells of a CAM array and rearranges them into a shift register achieving better area, but still the number of cells must be large to get acceptable compression ratios. As data communication systems demand high throughput compression architectures, these kind of parallel approaches are indispensable and a way to raise the resource utilization must be found. In addition to these LZ77 architectures, there is also the BWT Weavesorter approach. The Weavesorter sorts the block string using a shift register and as said in Chapter 3, the size of the Weavesorter is directly proportional to the compression ratio just as the LZ77 dictionary. The study of the LZ77 hardware approaches, as well as the BWT existing architectures, leads to the design of a combined module that uses

the same memory unit for both algorithms and thus a high reuse of the resources can be reached. This dual architecture must be able to execute any of these algorithms separately, but each one shares its dictionary (LZ77) or the original string (BWT) at the same time.

4.2 The software and hardware approaches

When executing an algorithm on a general purpose microprocessor, the instruction set specified by its architecture must be used exclusively. Each one of this instructions have a fixed number of clock cycles to perform the task it is designed for. This leads to the following drawbacks:

- There is an overhead caused by the characteristic pipeline of a microprocessor, in other words, only one instruction can be executed by the system and thus, only one task is performed each lapse of time.
- Generally, the width of the processor's registers is 32 bits. If every symbol in a data compression scheme is represented by 8 bits (ASCII code), the comparison operations needed by the LZ77 and BWT algorithms would waste the rest 24 bits of the registers.
- Both compression algorithms operate using large dictionaries (LZ77) or registers (BWT) and they should be stored in external memory units. It causes several accesses and bus requests affecting the performance of the system.

Although the mentioned disadvantages of a software approach, the inherent flexibility of a general purpose microprocessor can be of great importance, particularly for wireless communication devices. Nowadays, the combination of hardware and software approaches are becoming attractive because they link most of the advantages of the two proposals. As stated in Chapter 3, the main benefit of the hardware architectures is the acceleration of the process using parallel structures. This advantages can be exploited if the parallel architecture is implemented as an external unit controlled by the microprocessor. This microprocessor can execute a data compression software program

when the processes are dependent from each other. Meanwhile, when the tasks can be executed in parallel, they are entrusted to the external unit. Thus the performance of the system is increased while the software flexibility is maintained. This external unit is often called *coprocessor*.

4.3 Solution proposal

This dissertation proposes a scheme consisting in adding an external unit or coprocessor to a general purpose processor core that carries out the most computationally expensive operations for the LZ77 and BWT data compression algorithms. The first main operation is the construction of the token for the LZ77 algorithm by a parallel searching process. The second main operation is the sorting of the original block and the obtaining of the transformed block for the BWT algorithm. These operations are included in the microprocessor's architecture by extending its Instruction Set Architecture (ISA) with instructions that control the added hardware. By the execution of these software operations, the microprocessor would employ fewer clock cycles speeding up the compression process. In addition, the coprocessor is designed to share structures between algorithms to save resources on a PLD implementation, particularly an FPGA.

4.4 Selecting the processor core

To design the proposed scheme mentioned in the previous section, a suitable processor core must be selected. The following list presents the specific requirements:

- **Availability of the source code.** As this project includes the integration process of the system, it is necessary to have the full source code of the core. In addition, due to budget limitations, a commercial IP can not be used, therefore the core must be open source.
- **Coprocessor interface.** For this project, it is important that the selected core is equipped with a control unit to handle a coprocessor. The integration of an external unit to any processor is feasible, but complicated. Therefore, a core already designed to operate a coprocessor is helpful.

- **Implementation platform.** The selected core must be designed for full implementation on FPGA PLDs since they are the main testing platform for this dissertation.

The commercial and open source processor cores are classified into *extensibles* and *configurables*. When a processor is extensible, the hardware description language source code is available. This allows the designer to add custom functionalities to the system and in some cases there is specialized software to define some parameters and constants in the source code. Some examples of this approach are: The MIPS family and the LEON2 core. The configurable cores let the designer to remove unneeded processing blocks and to set specific characteristics of the used functionalities, also using software tools. Some examples are the ARC^{TM}700 from ARC International and the Xtensa processor from Tensilica Inc. Both approaches the user can design the most suitable core for a specific application. At the end of the designing phase, the source code that describes the system is generated and it can be simulated and debugged.

The core chosen for this project is the LEON2 platform because it meets the mentioned requirements. The full source code is available in the VHDL hardware description language and it is open source. The LEON2 processor model provides an interface for external blocks like Floating-Point Units (FPU). The newest versions of the LEON2 are designed to be implemented in Xilinx FPGAs, which is the platform used for this project. In spite of its complexity, the core is not only extensible but also configurable, allowing the removal of unneeded blocks.

Chapter 5

Design of the BWT/LZ77 IP core

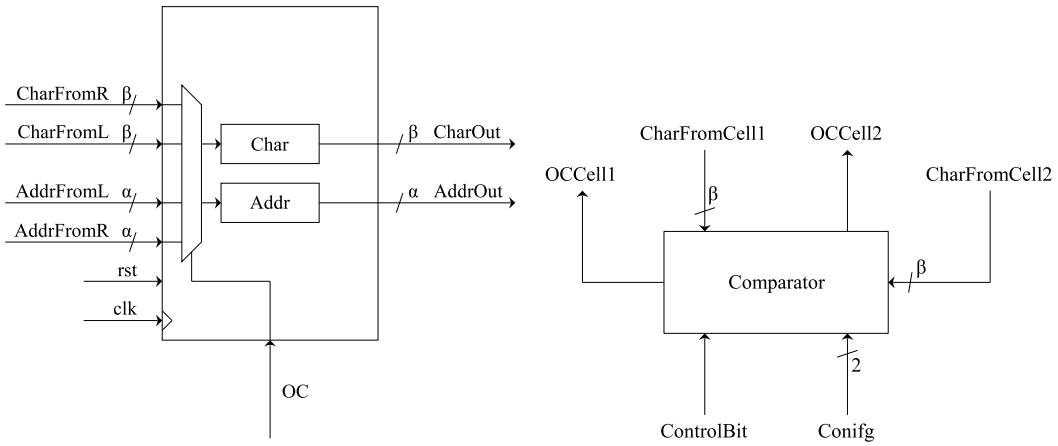
This chapter describes the design of the dual architecture for the BWT and the LZ77 algorithms reusing hardware resources. The first section presents a complete BWT architecture based on the proposal explained in Chapter 3. The second section describes the modifications made to the BWT architecture to reuse memory units and execute the LZ77 algorithm. The IP core is designed in VHDL language and validated with simulation tools.

5.1 The BWT architecture

The lexicographic sorting of n cyclic rotations of an n-character string is the most complex task of the BWT algorithm. As the Weavesorter machine has been proposed to solve this problem, the architecture of this dissertation is based on it. In [13] the design of the machine is shown but a complete functional model is not defined. In this section a module to operate a Weavesorter machine is proposed.

5.1.1 The Weavesorter machine

As stated in Chapter 3, the Weavesorter machine is designed as a bidirectional shift register with comparators for each pair of cells. Then, the basic modules of the Weavesorter are the cells and the comparators and they are wired in such a way that a shift register is conformed. Both modules are shown in Figure 5.1.



(a) Cell (b) Comparator

Figure 5.1: Basic elements of the Weavesorter machine.

Figure 5.1(a) shows the synchronous basic cell. It has two registers, the "Char" register stores the character or symbol and the "Addr" register stores the address of that symbol. By the *CharFromR* (Character From Right), *CharFromL* (Character From Left), *AddrFromR* (Address From Right) and *AddrFromL* (Address From Left) signals the ell can store the data from its right neighbor when shifting left and from its left neighbor when shifting right. The *OC* (Operation Code) signal comes from the Comparator and is used to choose the data form right or from left for the shift operation. Figure 5.1(b) shows the Comparator unit. The Comparator controls the behavior of a pair of cells through the *OCCell1* (Operation Code for Cell1) and *OCCell2* (Operation Code for Cell2) signals by reading the *Config* (Configuration) input. The configuration of the Comparator unit can be one of four codes:

- *Shift-Right.* The Comparator sets the two cells to take the data from the left neighbor to shift right the register.
- *Shift-Left.* The Comparator sets the two cells to take the data from the right neighbor to shift left the register.
- *Compare/Swap.* The Comparator compares the characters of the two cells and if the content of Cell1 is grater than the content of Cell2 the Swap operation is

performed. This is done by setting the Cell1 to take the data from Cell2 and setting the Cell2 to take the data from Cell1.

- *Idle*. The Comparator sets the two cells to hold their values.

The Comparator also has the *ControlBit* input, this is used to block the *Compare/Swap* operation if the characters of the two cells belong to different columns in the matrix and this way prevents the mixing of information. This control bit for every cell is stored in a bidirectional shift register called *ControlShiftRegister* and it shifts right and left together with the data on the cells as explained in Chapter 3. The width of the data signals is represented by the constants α and β. The α constant is the number of bits needed to represent a symbol (8 for ASCII code). The β constant depends on the total number of cells (n) in the system, it must be at least (log_{2n}) to represent the address of n symbols. The interconnection of two cells and a comparator is shown in Figure 5.2.

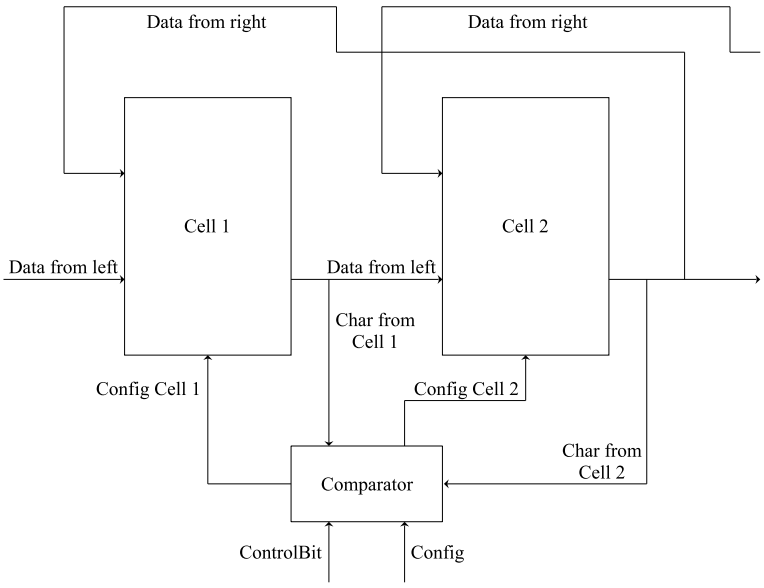


Figure 5.2: Interconnection between two cells and a comparator.

The Weavesorter machine (shown in Figure 5.3) is constructed by instantiating n cells, $n/2$ comparators, a bidirectional shift register of n bits and an n-input AND logic

gate. This logic gate allows to detect when all the bits in the *ControlShiftRegister* are set in "1". The output of the AND logic gate is a flag called *Done* and it indicates when all the columns of the matrix are sorted. The *ControlShiftRegister* has its input and output signals called *CtrlIn* and *CtrlOut* respectively. All the pairs of cells are identical but the first and the last ones. The inputs *CharFromL* and *AddrFromL* of the first cell are connected to the main input signal together with the inputs *CharFromR* and *AddrFromR* of the last cell. Another difference is that the outputs *CharOut* and *AddrOut* of the first and last cells are connected to a multiplexer. This configuration allows to have the general input signal *Input* connected to the left of the first cell when shifting right and connected to the right of the last cell when shifting left. Also the outputs of both cells are available in the general output signal *Output* depending on the actual shifting direction. The *Input* and *Output* signals are used to send and receive the characters and the addresses. It is worth noting that n must be even because every comparator has two cells.

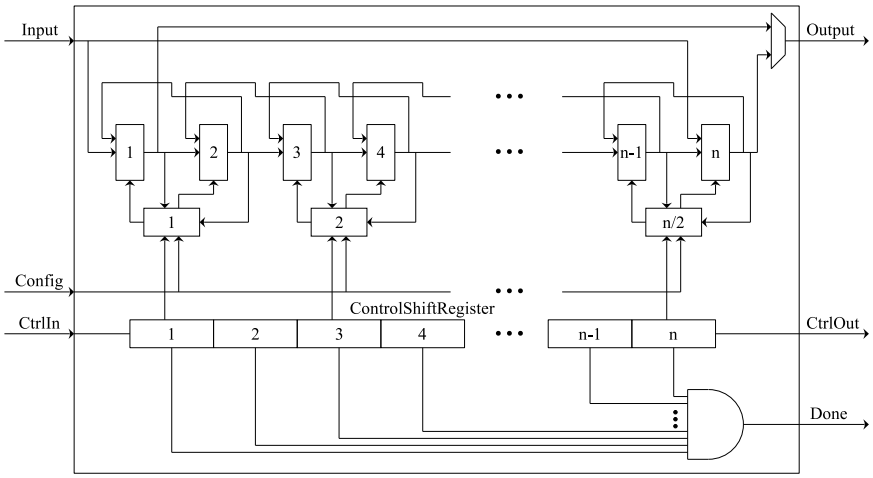


Figure 5.3: Complete Weavesorter machine of n cells.

5.1.2 The Control unit

When the shifting process of the Weavesorter is done the first column of the matrix is sorted. But if there is at least two similar symbols in the string they can not be

sorted and their respective next symbols must be compared. It means that the shifting process have to be repeated with the next columns until all the symbols are different from each other. If the BWT block is stored in a register of n elements, the reading of the next symbol is done by adding 1 to the address of the current symbol. Then, when the shifting process is completed, all the symbols stored in the next register of the block are inserted into the Weavesorter. To manage this register and to reinsert the next symbols an additional Control unit is needed. This unit is built as a Finite State Machine (FSM). A diagram is shown in Figure 5.4.

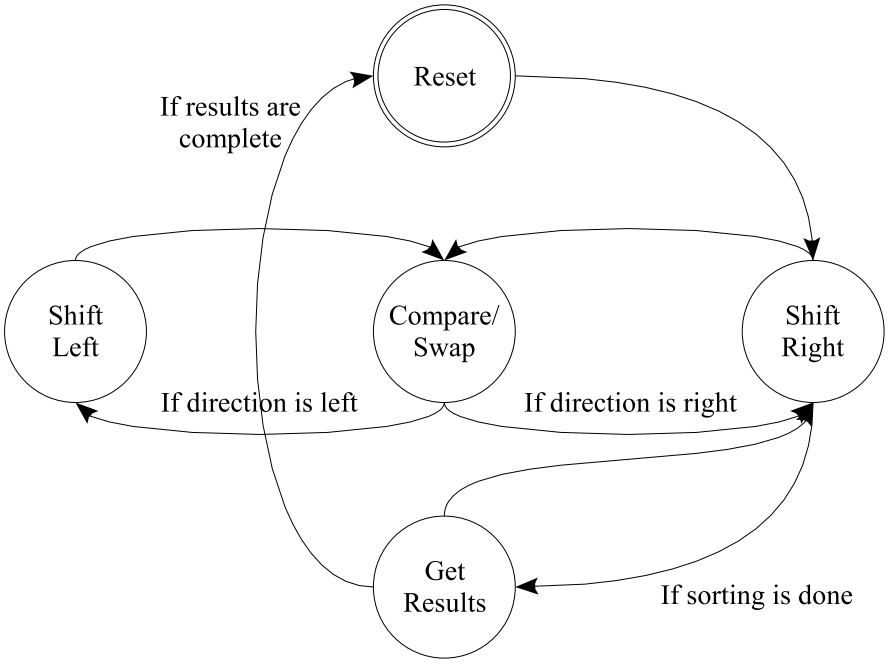


Figure 5.4: Finite State Machine to operate the Weavesorter machine.

The FSM is started in the *Reset* state and the next clock cycle it changes to the ShiftRight state. In this state the first symbol is shifted right into the Weavesorter and the next clock cycle the Compare/Swap state is reached. The two states follow each other until the Weavesorter is filled with symbols, thereafter is the turn of the ShiftLeft state. When all the rows are sorted, the FSM changes to the GetResults state where the addresses from the Weavesorter are obtained by shifting right while the last column

is calculated together with the position of the original string. A detailed description of the FSM is explained in the next subsection. This Control unit should also establish the communication with the coprocessor interface as will be explained in the next chapter.

5.1.3 The BWT module

The BWT module interconnects the Weavesorter machine with the Control unit. This module is fully functional once the original string is available to the FSM. Figure 5.5 shows its diagram.

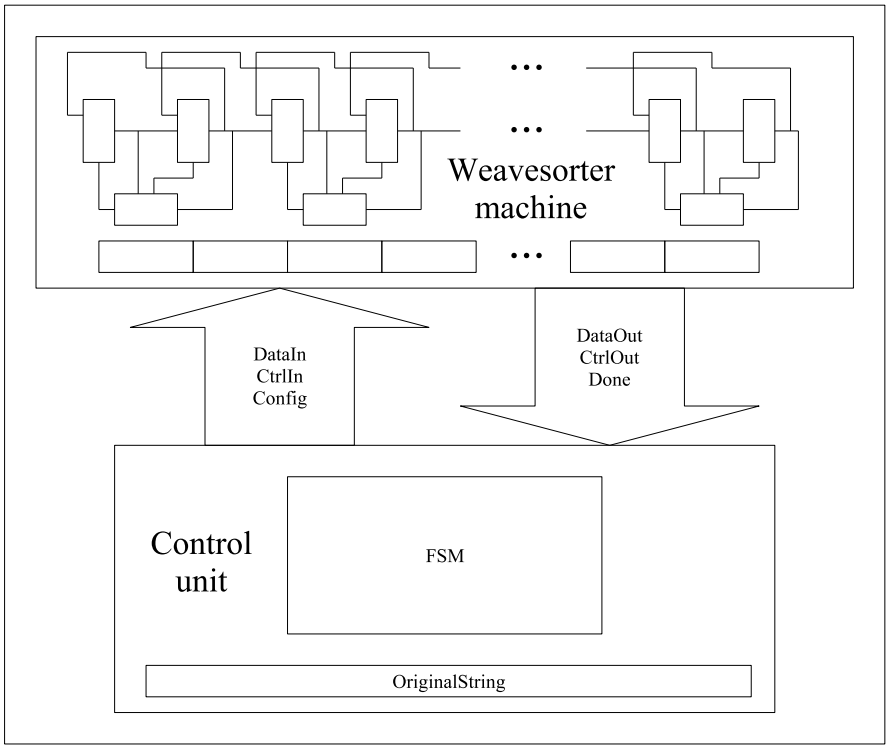


Figure 5.5: The BWT module.

The Control unit sends the information using the signal *DataIn* for the symbol and its address, the signal *CtrlIn* for the control bit and the signal *Config* for the configuration code. The signals *DataOut*, *CtrlOut* and *Done* are used to receive the character and its address, the control bit and the flag respectively. Once the Weavesorter

machine is connected with the Control unit the details of the operations performed in the states of the FSM can be described. Table 5.1 shows the signals used by the FSM.

Name	Type	Size
OriginalString	array	$n \times \beta$
SortedString	array	$n + 1 \times \beta$
Counter	counter	-1 to n
Iterations	counter	0 to n
TempData	register	$\alpha + \beta$
Direction	register	$1bit$
NextAddr	register	$1bit$
Original	register	β

Table 5.1: Signals used by the FSM.

This signals are used as following:

- *OriginalString*. To store the original string. The Control unit reads each symbol from this array and insert it to the Weavesorter machine.
- *SortedString*. To store the sorted string and the position in the matrix of the original string (index I). This is the output of the system.
- *Counter*. To count the shifting processes. The FSM must detect when to change the shifting direction. The counter is incremented for each *Shift-Right* or S*Shift-Left* operation.
- *Iterations*. Count the number of iterations. As explained in Chapter 3, the number of iterations is needed to obtain the index *I*.
- *TempData*. Store the previous character and its address.
- *Direction*. Set the current direction of the shifting process.
- *NextAddr*. Set the address of the next symbol.

- *Original*. Store the position in the matrix of the original string.

The following 5 figures describe the corresponding pseudocode for each FSM state. The Reset state is shown in Algorithm 1. Note that all the statements are executed in parallel.

Algorithm 1 The Reset state.

1: $Config \Leftarrow Idle$
2: $CtrlIn \Leftarrow 0$
3: $DataIn \Leftarrow 0$
4: $TempData \Leftarrow 0$
5: $Counter \Leftarrow 0$
6: $Iterations \Leftarrow 0$
7: $Direction \Leftarrow ToRight$
8: $NextState \Leftarrow ShiftRight$

In line 1: of Algorithm 1 the Weavesorter machine is configured with the *Idle* code. In lines 2: to 6: the signals are set to 0. In line 7: the shifting direction is set to right. And in line 8: the FSM moves to the ShiftRight state to begin the shifting process. The ShiftRight state is shown in Algorithm 2.

In line 1: of Algorithm 2 the Weavesorter machine is configured with the *Shift-Right* code to read the data from the Control unit. Only in the first iteration (line 3:) the symbols are read from the *OriginalString* array (line 4:), for the rest of the iterations the data is read from the Weavesorter machine itself (lines 6: and 7:). The ShiftRight state is executed until all the symbols are inside the Weavesorter machine, this is controlled by the *Counter* (line 9:). When the *Counter* reaches n it is reseted (line 12:), the *Direction* signal is set to left (line 13:) and the *Iterations* signal is incremented by one (line 14:). As stated in line 2:, if the symbols are still unsorted the FSM moves to the Compare/Swap state (line 16:). If the sorting process is done, the FSM moves to the GetResults state (line 19:). The Compare/Swap state is shown in Algorithm 3.

In line 1: of Algorithm 3 the code *Compare/Swap* is sent to the Weavesorter machine and if it is not full (line 2:) no data is prepared to be inserted (line 3:). In line 5: begins the process of calculating the symbol of the next column in the matrix. The address

Algorithm 2 The ShiftRight state.

1: $Config \Leftarrow Shift - Right$
2: **if** $Done = NO$ **then**
3: **if** $Iteration = 0$ **then**
4: $DataIn \Leftarrow OriginalString(Counter)$
5: **else**
6: $DataIn \Leftarrow DataOut$
7: $CtrlIn \Leftarrow CtrlOut$
8: **end if**
9: **if** $Counter < n - 1$ **then**
10: $Counter \Leftarrow Counter + 1$
11: **else**
12: $Counter \Leftarrow -1$
13: $Direction \Leftarrow ToLeft$
14: $Iterations \Leftarrow Iterations + 1$
15: **end if**
16: $NextState \Leftarrow Compare/Swap$
17: **else**
18: $Counter \Leftarrow Counter + 1$
19: $NextState \Leftarrow GetResults$
20: **end if**

Algorithm 3 The Compare/Swap state.

1: $Config \Leftarrow Compare/Swap$
2: **if** $Iteration = 0$ **then**
3: $DataIn \Leftarrow 0$
4: **else**
5: $NextAddr \Leftarrow DataOut(Addr) + 1$
6: **if** $NextAddr < n$ **then**
7: $DataIn \Leftarrow OriginalString(NextAddr)$
8: **else**
9: $DataIn \Leftarrow OriginalString(NextAddr - n)$
10: **end if**
11: $DataIn(Addr) \Leftarrow DataOut(Addr + 1)$
12: **end if**
13: **if** $Direction = ToRight$ **then**
14: $NextState \Leftarrow ShiftRight$
15: **else if** $Direction = ToLeft$ **then**
16: **if** $TempData \neq DataOut$ **then**
17: $CtrlIn \Leftarrow YES$
18: **else**
19: $CtrlIn \Leftarrow CtrlOut$
20: **end if**
21: $NextState \Leftarrow ShiftLeft$
22: **end if**

plus 1 of this symbol is written in the *NextAddr* signal. Lines 6: to 10: validate the reading of the next symbol from the *OriginalString* and it is put in the *DataIn* signal to be send to the Weavesorter machine in the next shifting operation. In line 11: the address plus 1 of the previous symbol is also presented in the *DataIn* signal. If the current shifting direction is right (line 13:), the FSM moves to the ShiftRight state. The process of detecting if a symbol coming from the Weavesorter machine belongs to the same group of the next symbol is described in lines 16: to 21:. If the symbol stored in *TempData* is different from the symbol on *DataOut* signal, they should not be compared by the Weavesorter machine and the control bit must be set to "1" (line 17:). If both symbols are equal, they belong to the same group and their next column symbols should be sorted, for this, the control bit keeps its value (line 19:). In line 21: the FSM moves to the ShiftLeft state which is described in Algorithm 4.

Algorithm 4 The ShiftLeft state.

1: $Config \Leftarrow Shift - Left$
2: $TempData \Leftarrow DataOut$
3: **if** $Counter < n - 1$ **then**
4: $\quad Counter \Leftarrow Counter + 1$
5: **else**
6: $\quad Direction \Leftarrow ToRight$
7: $\quad Counter \Leftarrow -1$
8: **end if**
9: $NextState \Leftarrow Compare/Swap$

In Algorithm 4, the Weavesorter machine is configure to shift left in line 1:. To be compared in the Compare/Swap state, the data from the *DataOut* signal is written in the *TempData* register (line 3:). From 4: to 9: lines, the *Counter* is incremented if the shifting process is not yet completed, else the direction is changed to right. In line 10: the FSM moves to the Compare/Swap state. GetResults is the last state and is shown in Algorithm 5.

In Algorithm 5, the Weavesorter machine is stopped by sending the *Idle* code. The GetResults state takes out all the addresses from the Weavesorter calling the ShiftRight

Algorithm 5 The GetResults state.

```
1: Config ⇐ Idle
2: if Counter < n then
3:    if DataOut(Addr − Iterations) = 0 then
4:       SortedString(Counter) ⇐ OriginalString(n − 1)
5:       Original ⇐ n − Counter − 1
6:    else
7:       SortedString(Counter) ⇐ OriginalString(DataOut(Addr) − Iterations − 1)
8:    end if
9:    NextState ⇐ ShiftRight
10: else
11:   SortedString(n) ⇐ Original
12:   NextState ⇐ Reset
13: end if
```

state (lines 2: and 9:). In line 3: the index I is detected if subtracting the total number of iterations from the address of of the *DataOut* symbols is equal to 0 and it is written in the *Original* signal. The addresses currently stored in the Weavesorter machine represent the suffix array that is used to get the last column of the matrix (line 7:). The index I is stored in the extra register of the *SortedString* array (line 11:) and the FSM moves back to the Reset state and the system is ready to sort the next block.

5.2 The LZ77 architecture

As one of the goals of this dissertation is the reuse of hardware, the LZ77 architecture must be designed in such a way that some modules of the BWT architecture are part of an LZ77 implementation. Throughout the study of these data compression algorithms, it was noticed that the LZ77 algorithm can be described using a shift register to store the dictionary of the model. The size of this shift register must be large if acceptable compression ratios are to be reached and the same requirement applies for the BWT Weavesorter approach. Thus if this structure is shared between the two algorithms a remarkable reuse of hardware can be achieved. In this section the adaptation of the

Weavesorter machine to execute the LZ77 algorithm is described and a mechanism to generate the LZ77 tokens is proposed.

5.2.1 The modified Weavesorter machine

The core of the Weavesorter machine is the bidirectional shift register that stores one symbol in every cell. The LZ77 scheme can use a similar shift register to build the dictionary by shifting left the input string. In a parallel approach, every symbol in the dictionary is compared with the incoming character. This comparison can be implemented by adding a comparator in every cell. In the Weavesorter machine, the modified cell is implemented as shown in Figure 5.6.

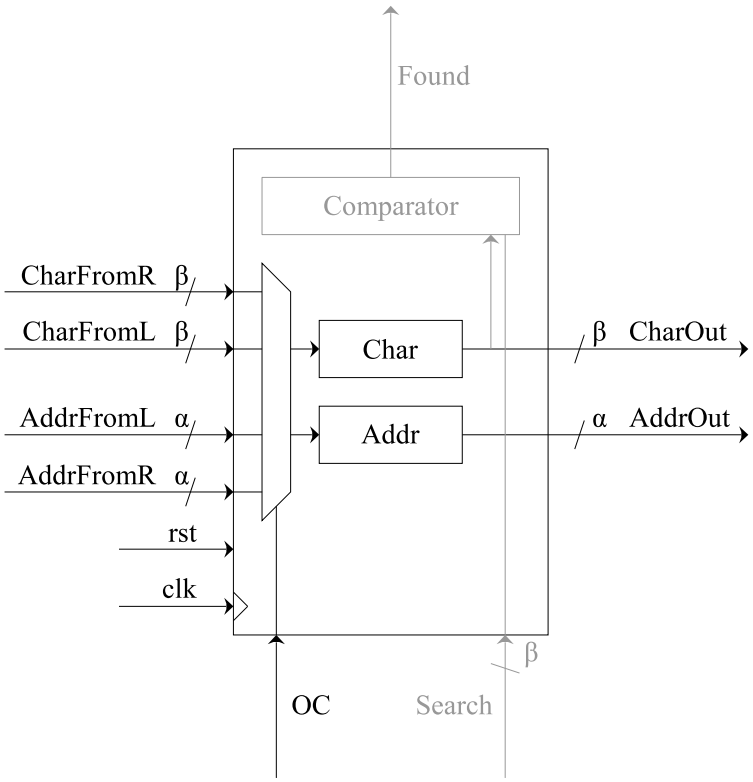


Figure 5.6: Modified Cell.

The new component of the cell is a comparator with the current symbol and a

searching symbol (*Search*) as inputs, and an output (*Found*) that indicates with a logic "1" if the searching symbol is equal to the current symbol and with a logic "0" if it is not. This change allows to search the dictionary in parallel. The modified Weavesorter machine is shown in Figure 5.7

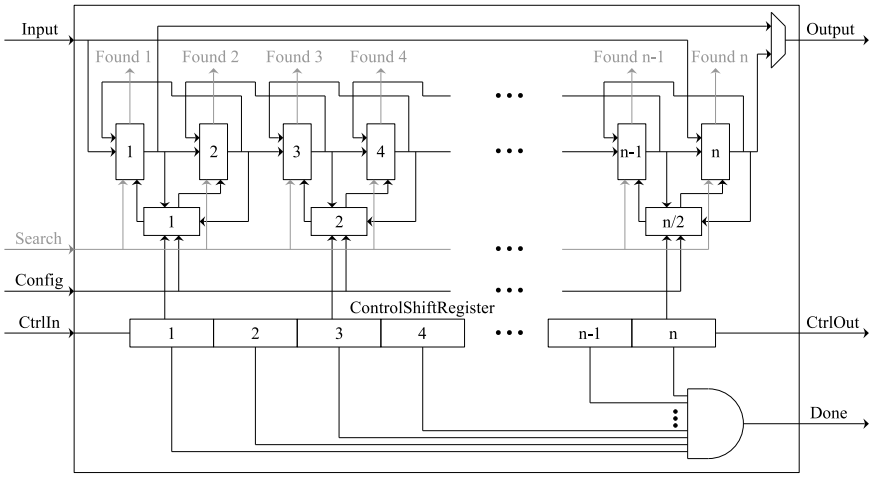


Figure 5.7: Modified Weavesorter machine.

The input string is shifted left into the Weavesorter machine while comparing the incoming symbol with the ones stored. This is done every clock cycle and the comparisons are asynchronous. The Weavesorter machine is ready to represent the LZ77 dictionary but still a mechanism to generate the token is needed. This mechanism is described in the next subsection.

5.2.2 The proposed LZ77 mechanism

To generate the LZ77 token an additional mechanism attached to the Weavesorter machine must be designed. This mechanism have to identify if a symbol is found in any of the cells, find the largest matching string in the dictionary and calculate its position and length. To detect if there is a matching symbol, all the *Found* outputs of every cell are connected to an OR-Tree gate, if a symbol is found the output of the gate will be a logic "1". It is a little more complicated to find a matching string. When a symbol is found in the dictionary, its position is stored in a record and the mechanism waits for

the next symbol. If the next symbol is found in the same position, it is part of the string already found (because of the shifting operation) and that position remains stored. The process is repeated until the incoming symbol is not found in the stored position and the match position (T_o) is the value in the record. But when several symbols are found there are several matching strings and the mechanism must find the largest. All the positions of the found symbols are stored in the record and if the next symbol is not found in a certain position it is erased. The position of the largest matching string is the last erased record. If two or more positions are the last ones on the record, the mechanism takes the corresponding to the right-most cell. (This is useful when an additional data compression algorithm like Huffman coding [14] will compress the tokens. With smaller values, the compression ratio is improved [1].) The length (T_l) is obtained from an ascending counter module that is enabled when there is a match (the OR-Tree is active). When no more symbols are found the counter contains the length of the string found and it is reseted to count the length of the next string. The next symbol (T_n) is taken from the buffer that is the input of the Weavesorter machine. The schematic diagram of the mechanism proposed in this dissertation is shown in Figure 5.8.

The cells of the Weavesorter machine are in the bottom of the figure and the remaining components are part of the mechanism that calculates the LZ77 token. A description of every component is listed:

- *History.* A register of size n. It is the record where the found positions for every cell are stored. All the bits can be set in "1" by the *set* signal.
- *OR-Tree.* An OR gate of n inputs. It detects if there is a matching in any cell.
- *AND-Group(a).* A group of AND gates. They store a copy of the *Found* signals in the *History* register every cycle.
- *AND-Group(b).* A group of AND gates. They activate the OR-Tree and are the input of the *Priority encoder*.
- *Priority encoder.* It finds the largest match position.

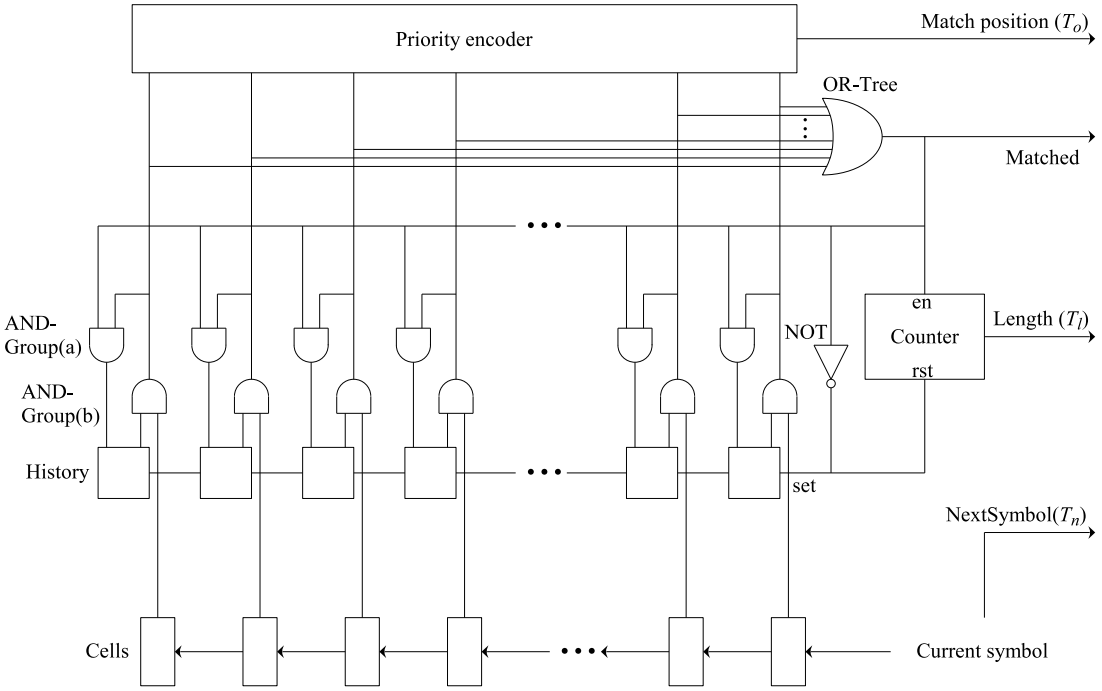


Figure 5.8: The proposed LZ77 mechanism.

- *Counter*. An ascending decimal counter. It calculates the length of a found string by counting when a match symbol is found.
- *NOT*. A NOT gate. Resets the *Counter* and sets in "1" the bits of the *History* register.

To explain the operation of the mechanism an example is given:

Let the string $S = A_1\ B_2\ R_3\ A_4\ C_5\ A_6\ D_7\ A_8\ B_9\ R_{10}\ A_{11}\ S_{12}$

Initially, as all the *Found* signals are "0", the OR-Tree and the AND gates are "0". The *Counter* is reseted to "0" and the *Priority encoder* has nothing to encode and its output is also"0". The *History* bits are set to "1" to detect the first found symbol. In cycle 1, the symbol A_1 is searched in the cells but is not found. All the *Found*, *AND-Group(a)*, *AND-Group(b)* and *OR-Tree* signals remain "0". The *Matched* signal

is "0" meaning that a token must be constructed. T_o and T_l are "0", T_n is the current symbol A_1, then the token is $T = (0, 0, A_1)$. In cycles 2 and 3, the symbols B_2 and R_3 are not found either, the tokens $T = (0, 0, B_2)$ and $T = (0, 0, R_3)$ are generated as well. In cycle 4, the symbol A_4 is found in the cell number 3 (from right to left), as its corresponding *History* bit is "1" the AND gate of *AND-Group(b)* is activated. The *Priority encoder* shows the match position 3 and the *Counter* is enabled by the output of the *OR-Tree* and starts counting. The *Matched* signal is "1" and it means no token should be generated yet. Note that the *AND-Group(a)* stores a copy of the *Found* signals in the *History* register, this is to remember where a symbol was found in the past. In cycle 5, the symbol C_5 is not found in the cells, this disables the *AND-Group(b)* and the *OR-Tree* shows "0". The *Matched* signal constructs the token $T = (3, 1, C_5)$ with the values of *Match position*, *Length* and the current symbol. The *Counter* is reseted by the *NOT* gate and the *History* register is set in "1"s. In cycle 6 the symbol A_6 is searched, this time is found in two cells (2 and 5). The *Priority encoder* selects the smallest match position (2). In cycle 7, the new symbol D_7 is not found and the token $T = (2, 1, D_7)$ is generated. In cycle 8 the symbol A_8 is found in the cells 2, 4 and 7, one of this will be the largest string. In cycle 9 the symbol B_9 is not found neither in cell 2 nor in cell 4, their *History* bit is set to "0" and they can not be a matching string. In cycles 10 and 11 the symbols R_{10} and A_{11} are found in cell 7, but in cycle 11 the symbol S_{12} is not found there. The remaining AND gate of *AND-Group(b)* is disabled and the token $T = (7, 4, S_{12})$ is constructed. As there are no more symbols to encode, the shifting operation stops and the process is finished.

5.2.3 The modified Control unit

The Control unit must also be modified to execute the LZ77 algorithm. As the LZ77 mechanism constructs the token, the FSM of the Control unit deals only with the process of send and receive the data and the modifications needed are few. A new state, called *LZ77* is added as shown in Figure 5.9.

5.2.4 The LZ77 module

A fully functional LZ77 module is shown in Figure 5.10.

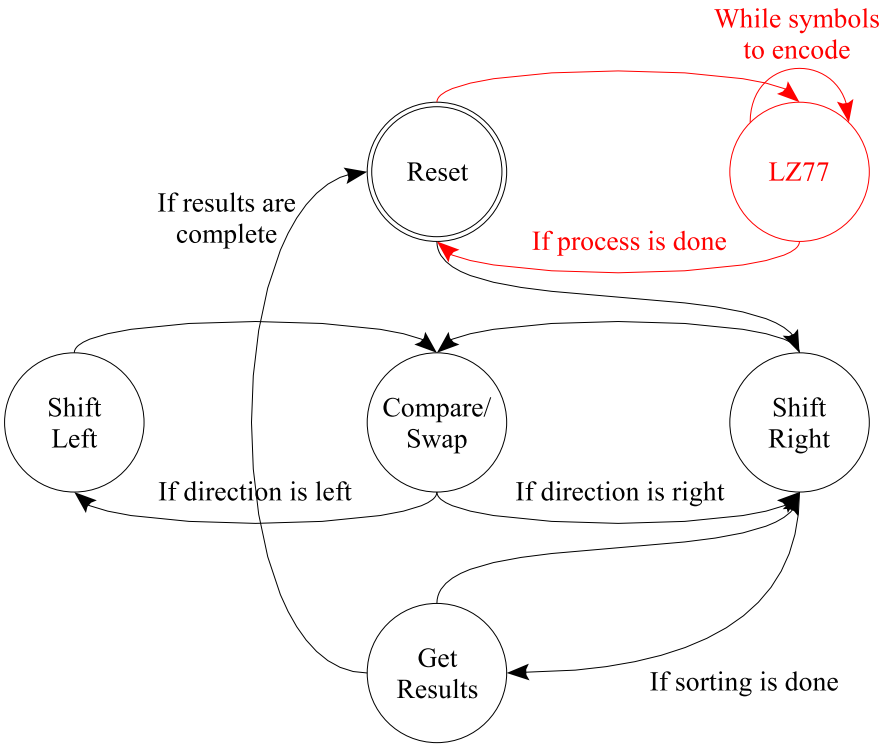


Figure 5.9: New *LZ77* state added to the Finite State Machine.

The signals needed to execute the LZ77 algorithm are added to the communication bus. The *Matched*, *MatchPosition*, *Length* and *NextSymbol* are the outputs from the modified Weavesorter machine. The *DataIn* and *Config* signals remain. New signals are needed to operate the *LZ77* state. Table 5.2 shows the added signals.

Name	Type	Size
SortedString	array	$3n \times \beta$
LZ77Counter	counter	0 to $3n$

Table 5.2: New signals for the *LZ77* state.

The *SortedString* used for the BWT operation can be used to store the generated tokens, but its size must be modified. To hold three values instead of one, the size is

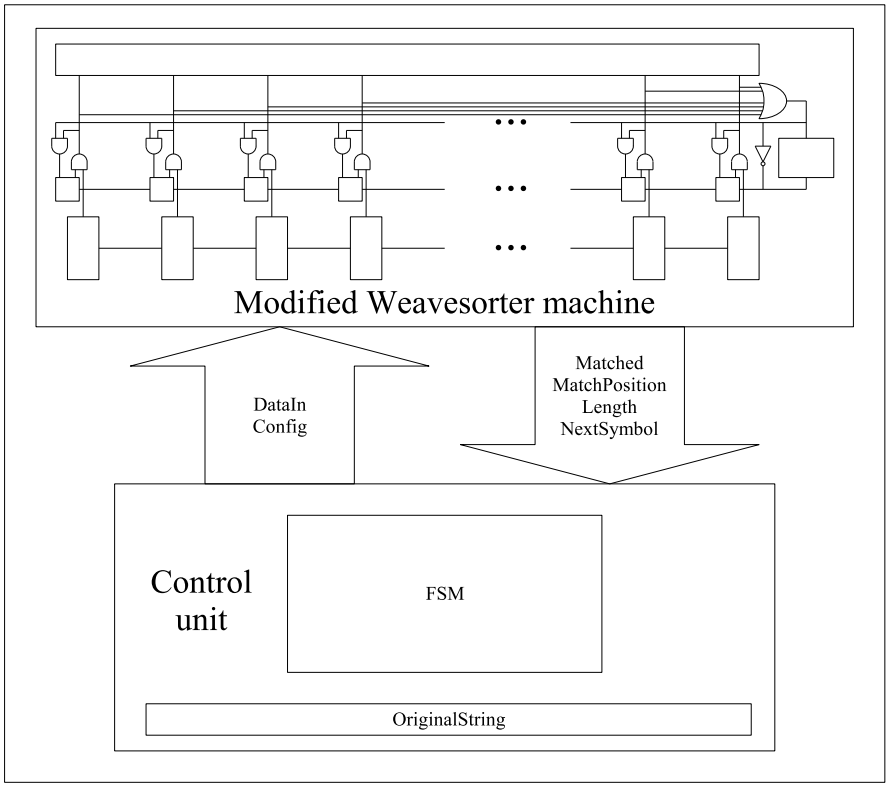


Figure 5.10: The LZ77 module.

increased to $3n$. The *LZ77Counter* is used to control the *SortedString* addresses where each value of the tokens are written. The previous BWT *Counter* can not be used because a three by three counting is needed. The pseudocode for the *LZ77* state is shown in Algorithm 6.

Line 1: of Algorithm 6 shows that all the operations in this state are performed every clock cycle. In lines 3: and 4: the Weavesorter machine is configured with the *Shift-Left* code and the symbols are read from the *OriginalString* array. The lines from 8: to 11: store the three components of the token in the *SortedString* array if the *Matched* signal is a logic "0" (line 7:). Note that these operations are executed according to the Table 3.1 by the conditions on lines 2: and 6:. It is worth noting that the Reset state is also modified but the only difference is that the *LZ77Counter* is set to "0".

In the previous sections the design of a dual architecture that executes both BWT

Algorithm 6 The LZ77 state.

1: **if** $RisingEdge(clk)$ **then**
2: **if** $Counter < n$ **then**
3: $Config \Leftarrow Shift - Left$
4: $DataIn \Leftarrow OriginalString(Counter)$
5: **end if**
6: **if** $Counter < n + 1$ **then**
7: **if** $(Counter > 0)$ *and* $(Matched = NO)$ **then**
8: $SortedString(LZ77Counter) \Leftarrow MatchPosition$
9: $SortedString(LZ77Counter + 1) \Leftarrow Length$
10: $SortedString(LZ77Counter + 2) \Leftarrow NextSymbol$
11: $LZ77Counter \Leftarrow LZ77Counter + 3$
12: **end if**
13: $Counter \Leftarrow Counter + 1$
14: **else**
15: $NextState \Leftarrow Reset$
16: **end if**
17: **end if**

and LZ77 algorithms is described. This architecture is called BWT/LZ77 IP core and is the main component of the coprocessor designed in this dissertation. In the next chapter the LEON2 processor is described as well as the coprocessor interface. To work with the interface a few modifications on the BWT/LZ77 core are still needed. The software program of the LEON2 to control the coprocessor and execute any of the algorithms is explained.

Chapter 6

Interface with the LEON2 platform

6.1 The LEON2 processor

LEON2 is a System on Chip (SoC) platform with a 32-bit SPARC V8 [15] compatible embedded processor, an Advanced Microcontroller Bus Architecture (AMBA) [16], I/O cores like a UART or PCI interface, etc. It was developed by the European Space Agency (ESA) and is available freely [17] with full source code in VHDL under LGPL (GNU Lesser General Public License) [18]. The LEON2 platform is extensively configurable and may be efficiently implemented on both FPGAs and ASIC technologies. A LEON2 diagram is shown in Figure 6.1.

The SPARC V8 architecture defines one (optional) Floating-Point Unit (FPU), the LEON2 pipeline provides one interface port for this unit. Three different FPUs can be interfaced: Gaisler Researchs GRFPU [17], the Meiko FPU from Sun Microsystems [19] and the LTH FPU. To integrate the coprocessor to the LEON2 processor, the Floating-Point Unit interface is used. Because GRFPU and Meiko FPU are not distributed with the open source LEON2 model (they must be obtained separately from Gaisler Research and Sun Microsystems respectively) the LTH FPU is considered in this dissertation. The LTH FPU is designed by Martin Kasprzyk, a student at Lund Technical University, and it uses the same serial interface as the Meiko FPU. The LTH FPU is replaced by the

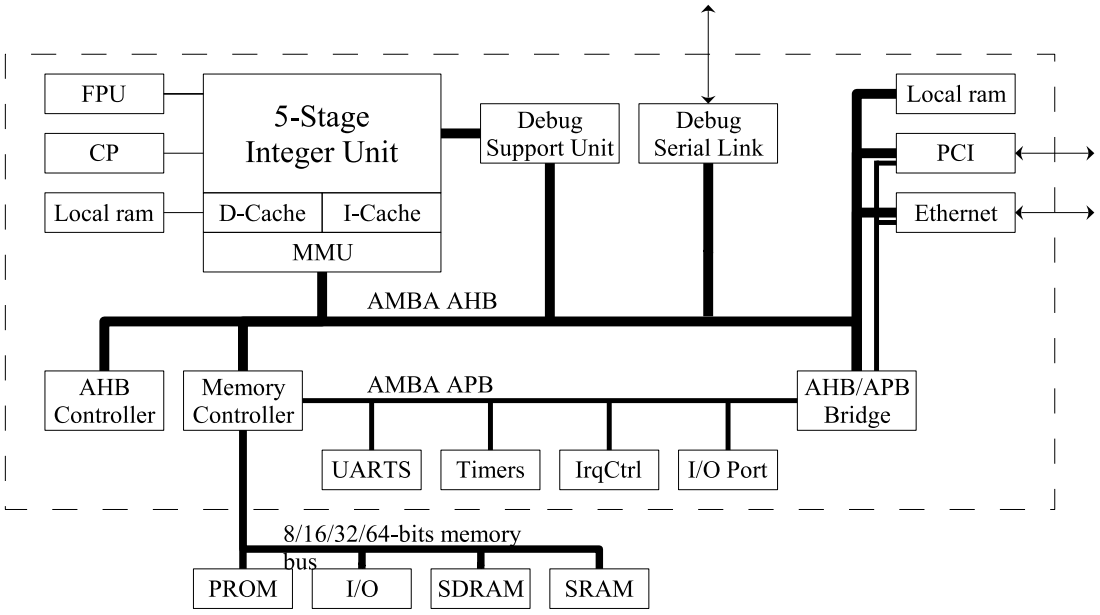


Figure 6.1: The LEON2 processor.

coprocessor core. In the next subsection, the details of the FPU interface are presented.

6.1.1 Generic FPU interface

The LEON2 model includes a module called *fpu_core* that is a wrapper around Meiko compatible FPU cores (Figure 6.2(a). This module uses records to connect with the Integer Unit (IU) of the LEON processor. These records are shown in Figure 6.2(b).

The FPU is started by asserting the FpOp signal together with a valid instruction in the FPInst signal. The operands are driven on the following cycle together with the FpLd signal. If the instruction will take more than one cycle to complete, the execution unit must drive FpBusy from the cycle after the FpOp signal was asserted, until the cycle before the result is valid. The result, condition codes and exception information are valid from the cycle after the de-assertion of FpBusy, and until the next assertion of FpOp.

Algorithm 7 The ReadInst state.

1: **if** $Start = YES$ **then**
2: $Instruction \Leftarrow Inst$
3: $NextState \Leftarrow ExecInst$
4: **end if**
5: $Busy \Leftarrow NO$

Algorithm 8 The ExecInst state.

1: **if** $Load = YES$ **then**
2: $ReadInstruction$
3: **end if**

Algorithm 9 The FADDD instruction.

1: **for** i from 8 downto 1 do **do**
2: $OriginalString(ReadCounter + 8 - i) \Leftarrow Op1(\beta * i - 1 downto \beta * (i - 1))$
3: $OriginalString(ReadCounter + 16 - i) \Leftarrow Op1(\beta * i - 1 downto \beta * (i - 1))$
4: **end for**
5: $Busy \Leftarrow YES$
6: $ReadCounter \Leftarrow ReadCounter + 16$
7: $NextState \Leftarrow ReadInst$

Algorithm 10 The FSQRTD instruction.

1: $Busy \Leftarrow YES$
2: $NextState \Leftarrow ShiftRight$

Algorithm 11 The FSQRTS instruction.

1: $Busy \Leftarrow YES$
2: $NextState \Leftarrow LZ77$

Algorithm 12 The FSUBD instruction.

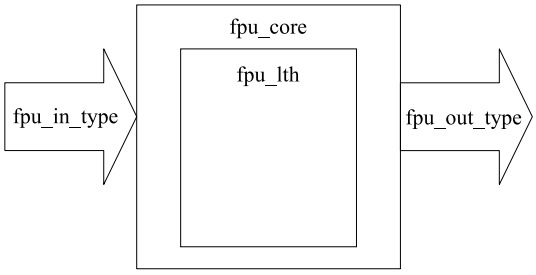
1: **if** $WriteCounter < n$ **then**
2: $FracResult(7 downto 0) \Leftarrow SortedString(WriteCounter + 0)$
3: $FracResult(15 downto 8) \Leftarrow SortedString(WriteCounter + 1)$
4: $FracResult(23 downto 16) \Leftarrow SortedString(WriteCounter + 2)$
5: $FracResult(31 downto 24) \Leftarrow SortedString(WriteCounter + 3)$
6: $FracResult(39 downto 32) \Leftarrow SortedString(WriteCounter + 4)$
7: $FracResult(47 downto 40) \Leftarrow SortedString(WriteCounter + 5)$
8: $FracResult(51 downto 48) \Leftarrow SortedString(WriteCounter + 6)(3 downto 0)$
9: $ExpResult(3 downto 0) \Leftarrow SortedString(WriteCounter + 6)(7 downto 4)$
10: $ExpResult(10 downto 4) \Leftarrow SortedString(WriteCounter + 7)(6 downto 0)$
11: $SignResult \Leftarrow SortedString(WriteCounter + 7)(7)$
12: **else**
13: $FrackResult(7 downto 0) \Leftarrow SortedString(WriteCounter)$
14: **end if**
15: $Busy \Leftarrow YES$
16: $WriteCounter \Leftarrow WriteCounter + 8$
17: $NextState \Leftarrow ReadInst$

Algorithm 13 The FSMULD instruction.

1: $NextState \Leftarrow Reset$

6.1.2 Configuration

6.2 The complete BWT/LZ77 coprocessor



(a) FPU core module

```
type fpu_in_type is record
        FpInst              : std_logic_vector(9 downto 0);
        FpOp                : std_logic;
        FpLd                : std_logic;
        Reset               : std_logic;
        fprf_dout1          : std_logic_vector(63 downto 0);
        fprf_dout2          : std_logic_vector(63 downto 0);
        RoundingMode        : std_logic_vector(1 downto 0);
        ss_scan_mode        : std_logic;
        fp_ctl_scan_in      : std_logic;
        fpuholdn            : std_logic;
end record;

type fpu_out_type is record
        FpBusy              : std_logic;
        FracResult          : std_logic_vector(54 downto 3);
        ExpResult           : std_logic_vector(10 downto 0);
        SignResult          : std_logic;
        SnnotDB             : std_logic;
        Excep               : std_logic_vector(5 downto 0);
        ConditionCodes      : std_logic_vector(1 downto 0);
        fp_ctl_scan_out     : std_logic;
end record;
```

(b) FPU core records

Figure 6.2: The fpu_core wrapping module and its I/O records.

Bibliography

[1] D. Salomon, *Data Compression: The Complete Reference*, 3rd ed. Springer-Verlag, 2004.

[2] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, April 1986.

[3] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[4] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," SRC (digital, Palo Alto), Tech. Rep. 124, May 1994.

[5] M. Nelson and J.-L. Gailly, *The Data Compression Book*, 2nd ed. M&T Books, 1996.

[6] S. Jones, "100 mbit/s adaptive data compressor design using selectively shiftable content-addressable memory," *IEE Proceedings-G*, vol. 139, no. 4, pp. 498–502, August 1992.

[7] R.-Y. Yang and C.-Y. Lee, "high-throughput data compressor designs using content addressable memory," in *ISCAS*, 1994, pp. 147–150.

[8] R. Ranganathan and S. Henriques, "High-speed VLSI design for lempel-ziv-based data compression," *IEEE Transactions on Circuits and Systems*, vol. 40, no. 2, pp. 96–106, February 1993.

[9] B. Jung and W. P. Burleson, "Efficient VLSI for lempel-ziv compression in wireless data communication networks," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 3, pp. 475–483, September 1998.

[10] J.-M. Cheng, L. M. Duyanovich, and D. J. Craft, "A fast highly reliable data compression chip and algorithm for storage systems," *IBM Journal of Research and Development*, vol. 40, no. 6, pp. 603–613, November 1996.

[11] D. J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions," *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 733–745, November 1998.

[12] A. Corporation, "Application note: An 119 (implementing high-speed search applications with apex cam) ver. 1.01," available: http://www.altera.com/. [Accessed September 20th, 2005].

[13] A. Mukherjee, N. Motgi, J. Becker, A. Friebe, C. Habermann, and M. Glesner, "Prototyping of efficient hardware algorithms for data compression in future communication systems," in *IEEE International Workshop on Rapid System Prototyping*, June 2001, pp. 58–63.

[14] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the Institute of Electronics and Radio Engineers*, vol. 40, no. 9, September 1952, pp. 1098–1101.

[15] *The SPARC Architecture Manual, Version 8*, Revision sav080si9308 ed., SPARC International Inc., 1992, available: http://www.sparc.org/. [Accessed September 20th, 2005].

[16] *AMBA Specification 2.0*, ARM Limited, May 1999, available: http://www.arm.com/. [Accessed September 20th, 2005].

[17] G. Research, available: http://www.gaisler.com/. [Accessed September 20th, 2005].

[18]

[19] S. Microsystems, available: http://www.sun.com/. [Accessed September 20th, 2005].