Parallel Hardware/Software Architecture for the BWT and LZ77 Lossless Data Compression Algorithms

Arquitectura Hardware/Software Paralela para los Algoritmos de Compresión de Datos sin Pérdida BWT y LZ77

Virgilio Zuñiga Grajeda, Claudia Feregrino Uribe and Rene Cumplido Parra
National Institute for Astrophysics, Optics and Electronics
Luis Enrique Erro No. 1. Tonantzintla, Puebla, Mexico. Postal Code: 72840
virgilio@inaoep.mx; cferegrino@inaoep.mx; rcumplido@inaoep.mx

Article received on july 25, 2006; accepted on october 2, 2006

Abstract.
Nowadays, the use of digital communication systems has increased in such a way that network bandwidth is affected. This problem can be solved by implementing data compression algorithms in communication devices to reduce the amount of data to be transmitted. However, the design of large hardware data compression models implies to consider an efficient use of the silicon area. This work proposes the conjunction of two different hardware lossless data compression approaches which share common hardware elements. The project also involves the design of a hardware/software architecture to exploit parallelism increasing execution speed while keeping flexibility. A custom coprocessor unit executes the compute-intense tasks of the Burrows-Wheeler Transform and the Lempel-Ziv lossless data compression schemes. This coprocessor unit is controlled by a SPARC V8 compatible general purpose microprocessor called LEON2.
Keywords: Data compression, Burrows-Wheeler Transform, Lempel-Ziv, Coprocessor, LEON2.

Resumen.
Hoy en día, el uso de sistemas de comunicación digitales ha aumentado de tal forma que el ancho de banda en las redes resulta afectado. Este problema puede solucionarse implementando algoritmos de compresión de datos en dispositivos de comunicación reduciendo la cantidad de datos a transmitir. Sin embargo, el diseño de modelos complejos de compresión de datos en hardware implica considerar el uso eficiente de la superficie de silicio. Este trabajo propone la combinación de dos esquemas diferentes de compresión de datos sin pérdida que compartan elementos comunes. Este proyecto también trata el diseño de una arquitectura hardware/software que explote el paralelismo e incremente la velocidad de ejecución manteniendo su flexibilidad. Un coprocesador ejecuta las tareas computacionalmente intensas de los esquemas de compresión Burrows-Wheeler Transform y Lempel-Ziv. El coprocesador es controlado por un microprocesador de propósito general compatible con la arquitectura SPARC V8 llamado LEON2.
Palabras clave: Compresión de Datos, Transformada de Burrows-Wheeler, Lempel-Ziv, Coprocesador, LEON2.

1 Introduction

In recent years, there has been an unprecedented explosion in the amount of digital data transmitted by communication systems. Millions of users access the World Wide Web every day to send and receive all kind of data. However, the amount of data transmitted tends to grow and the network bandwidth can be insufficient. To solve this problem, it is possible to reduce this amount of data without altering the information in the message. This procedure, known as *data compression*, may result in a reduction of the total amount of data to be transmitted. A wide variety of data can be transmitted through the network such as images, video, text, sound and software. Although some data types allow some loss of information when compressed, text and software must be decoded exactly as they were before compression, since the smallest change can compromise the meaning of the information. Also when choosing a data compression algorithm it is important to consider that some algorithms achieve better compression on certain kind of data than others. But when the kind of data to transmit is not known, lossless data compression algorithms are required. Hence, the present work focuses on lossless data compression and also explores the use of two different algorithms to suit the variety of the kind of data.

Many algorithms have been developed to compress data. As any other algorithm in computer science, basically there are two methods to execute them. One common method is to use a general purpose processor programmed to perform a set of tasks. A main microprocessor controls the computer resources and the software uses these resources to accomplish certain task. This method has the advantage of flexibility, if the algorithm must be modified, the software can easily be changed to execute another task. However the method has a disadvantage, its performance is poor and for some tasks the general purpose processor is unacceptably slow. This is because the approach lacks of hardware-optimized components. The other method is to implement the algorithm using an Application Specific Integrated Circuit (ASIC). The ASICs are circuits designed to solve a specific problem using optimized hardware to achieve better performance. However the drawback of this approach is the loss of flexibility, it is impossible to make any changes to the architecture when the circuit is already fabricated.

In the last 10 years, designers have been using reconfigurable computing to exploit the advantages of both methods [10], [9], [5]. Reconfigurable computing offers greater flexibility than an ASIC solution and increased speed over a software approach. These benefits are obtained with Field Programmable Gate Arrays (FPGAs). An FPGA is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates (such as AND, OR, XOR, NOT) or more complex combinatorial functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories. A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "Field Programmable") so that the FPGA can perform whatever logical function is needed. An FPGA can be used in conjunction with a general purpose processor. In this case, the functions that can not be speed up could be run simultaneously on a general purpose machine with only the compute-intense algorithms executed on the FPGA. Similarly, for problems requiring more resources than physically available on a single FPGA, multiple devices can be tied together and configured to solve even more complex problems. Other way to solve large problems when resources are limited is to reuse these resources.

In summary, a general purpose processor can execute an algorithm with an FPGA implementing the compute-intense tasks achieving higher processing speeds. Also it is possible to save FPGA resources by reusing elements. The goals of this work are twofold. First, to design a custom coprocessor to execute the compute-intense tasks of lossless data compression algorithms and to save resources by reusing hardware elements. Second, to test and validate the designed coprocessor by attaching it to a general purpose processor programmed with the lossless data compression algorithms. The execution of the algorithms using the coprocessor must be faster than a pure software implementation. The design of the hardware system is carried out using the bottom-up methodology, starting with simple components that are assembled and encapsulated to produce more complex components until the system is completed. The selection of the Lempel-Ziv (LZ77) and Burrows-Wheeler Transform (BWT) methods is the result of the analysis of several lossless data compression algorithms: The Arithmetic Coder [1], the Prediction by Partial Matching (PPM) [3] and the Huffman Coding [11] among others. The analysis of the corresponding hardware approach allows to compare the structures used by the algorithms and this study showed the feasibility to combine the presented algorithms. It is worth noting that the present work tackles the design of the compressor architecture and a decompressor scheme is part of the future work. In the next section, a brief introduction to data compression is given as well as a description of these methods.

2 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, are 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, let us 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 do not have to be stored perfectly. Much information can be simply thrown away from images, video data or sound, and when uncompressed such data will still be of acceptable quality. Lossless compression, in contrast, is used when data have 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 make, in the worst case, the text dangerously misleading.

2.1 LZ77 algorithm

In 1977 Lempel and Ziv [21] 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 1(a).

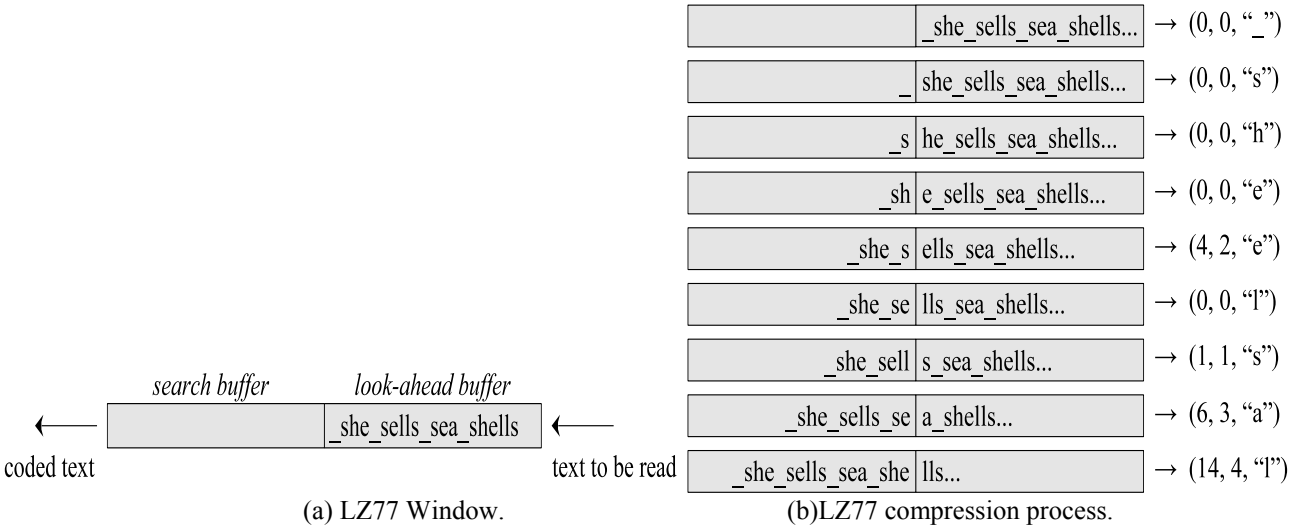

Fig. 1. LZ77 compression process example.

The part on the left is called the *search buffer*. This is the 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 dictionary, and a new matching process begins and proceeds in the same way until all the data are completely encoded. Starting

from the idea that recent data patterns are expected to appear in the near future, and the latest string 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 1(b). 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 yet. In the next step, the symbol "e" is found but it 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. A token of the form (0,0,...) which encodes a single symbol does not provide good compression but these kinds of tokens appear only at the beginning of the process. In Figure 1, it can be noticed that the more data contains the search buffer, the less tokens are needed to represent a string. The LZ77 algorithm laid the basis for compressed graphics formats such us GIF, TIFF and JPEG.

2.2 Burrows-Wheeler Transform

Burrows and Wheeler [2] presented in 1994 a block-sorting lossless data compression algorithm whose speed was comparable to algorithms based on LZ77 techniques and compression ratios were close to the best known compression ratios. 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 i^{th} character of *L* is the last character of the i^{th} 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 Move-To-Front (MTF) followed by a Run-Length Encoding (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 which number 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* x *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. Then the matrix is sorted lexicographically by rows as seen in Figure 2. 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 2 (row and column numbering starts from 0). This number is stored in *I*. It can be seen why *L* contains concentrations of identical symbols. It is worth noting that the larger *n*, the longer the concentrations of symbols and the better the compression. As this work only deals with the encoding process, the decoder phase is not described. The interested reader is referred to [15]. The most popular implementation of the BWT is the open-source program bzip2 [16] that uses the Huffman coding method.

_ s h e _ s e l l s _ s e a _ s h 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 e l l s _	_ s e l l s _ s e a _ s h e l l s _ s h e
h e _ s e l l s _ s e a _ s h e l l s _ s	_ s h e _ s e l l s _ s e a _ s h e l l s
e _ s e l l s _ s e a _ s h e l l s _ s h	_ s h e l l s _ s h e _ s e l l s _ s e a
_ 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
s e l l s _ s e a _ s h e l l s _ s h e _	e _ s e l l s _ s e a _ s h e l l s _ s h
e l l s _ s e a _ s h e l l s _ s h e _ s	e a _ s h e l l s _ s h e _ s e l l s _ s
l l s _ s e a _ s h e l l s _ s h e _ s e	e l l s _ s e a _ s h e l l s _ s h e _ s
l s _ s e a _ s h e l l s _ s h e _ s e l	e l l s _ s h e _ s e l l s _ s e a _ s h
s _ s e a _ s h e l l s _ s h e _ s e l l	h e _ s e l l s _ s e a _ s h e l l s _ s
_ s e a _ s h e l l s _ s h e _ s e l l s	h e l l s _ s h e _ s e l l s _ s e a _ s
s e a _ s h e l l s _ s h e _ s e l l s _	l l s _ s e a _ s h e l l s _ s h e _ s e
e a _ s h e l l s _ s h e _ s e l l s _ s	l l s _ s h e _ s e l l s _ s e a _ s h e
a _ s h e l l s _ s h e _ s e l l s _ s e	l s _ s e a _ s h e l l s _ s h e _ s e l
_ s h e l l s _ s h e _ s e l l s _ s e a	l s _ s h e _ s e l l s _ s e a _ s h e l
s h e l l s _ s h e _ s e l l s _ s e a _	s _ s e a _ s h e l l s _ s h e _ s e l l
h e l l s _ s h e _ s e l l s _ s e a _ s	s _ s h e _ s e l l s _ s e a _ s h e l l
e l l s _ s h e _ s e l l s _ s e a _ s h	s e a _ s h e l l s _ s h e _ s e l l s _
l l s _ s h e _ s e l l s _ s e a _ s h e	s e l l s _ s e a _ s h e l l s _ s h e _
l s _ s h e _ s e l l s _ s e a _ s h e l	s h e _ s e l l s _ s e a _ s h e l l s _
s _ s h e _ s e l l s _ s e a _ s h e l l	s h e l l s _ s h e _ s e l l s _ s e a _
Matrix of cyclic shifts of *S*	Matrix sorted lexicographically

Fig. 2. BWT matrix.

3 Related work

This section describes the strategies proposed in the past to design hardware architectures of the LZ77 and BWT algorithms. For the LZ77 algorithm, two main hardware proposals were studied: The systolic array and the Content-Addressable Memory (CAM) approaches. Whereas only one architecture was found for the BWT algorithm: The Weavesorter machine. As one of the goals of this work is to combine an LZ77 architecture with a BWT one, the Content-Addressable Memory approach was chosen because it uses similar hardware components of the Weavesorter machine architecture. This allows to combine both architectures reusing hardware resources. A description of these two strategies is presented in this section.

3.1 LZ77 CAM Approach

To speed up the massive comparisons needed to search in the dictionary on general purpose processors, algorithms implemented in software including hash lookup tables and tree-structured searching, have been developed and applied in storage systems [14]. However, for real-time applications, dedicated parallel architectures are used to support higher throughputs. One main hardware implementation of the LZ77 method is the CAM approach [12], [20]. The CAMs are hardware search engines for search-intensive applications. A CAM is composed of a group of cells, each cell has a storage device (register) and a comparator. Unlike standard computer memory (Random Access Memory) a data word is read 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 the address 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 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 obtained in the same cycle as the comparison operation using a simple (log_2N)-stage OR-tree, where N is the size of the CAM. If a match occurs, the corresponding

matching position address can be resolved by a (log_2N)-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 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 has 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.

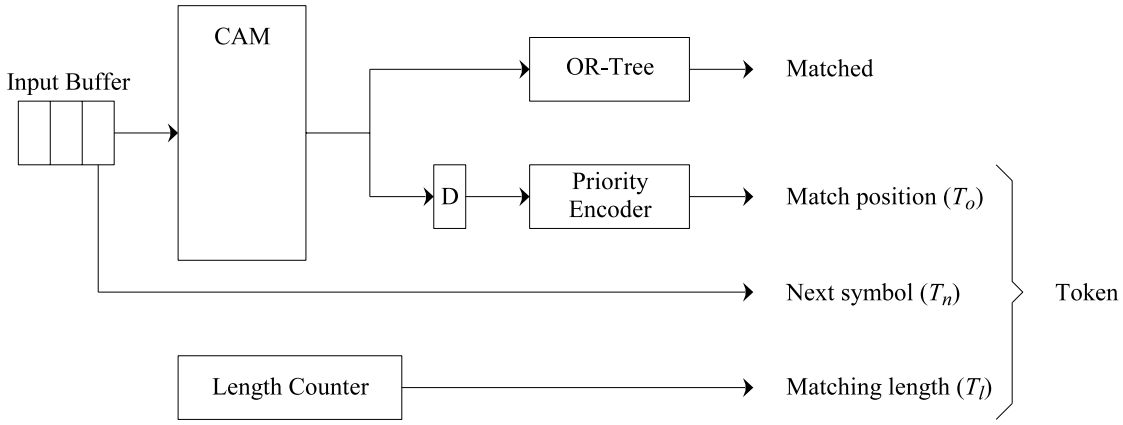

Fig. 3. CAM-based LZ77 encoder.

3.2 BWT Weavesorter Approach

The most complex task of the BWT algorithm is its lexicographic sorting of *n* cyclic rotations of a given string of *n* characters. To sort the cyclic rotations, a hardware architecture called *Weavesorter machine* has been proposed by Amar Mukherjee *et al* and it 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 the first string is put out, another string can be put in 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 out 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 4, a Weavesorter with eight basic cells is shown.

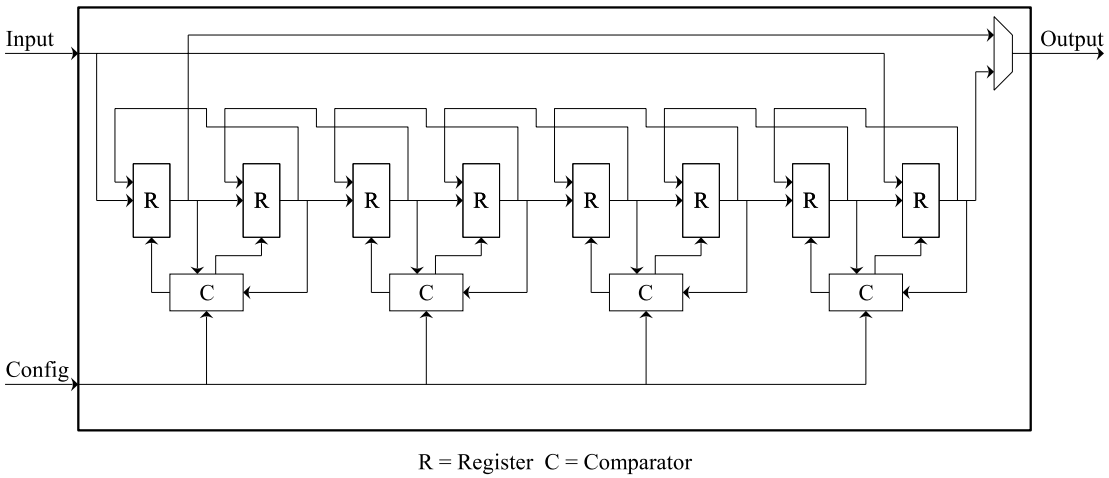

Fig. 4. Weavesorter with 8 cells.

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 met. 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 the 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. 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.

4 The BWT/LZ77 IP core

This section describes the design of an architecture that executes the BWT and the LZ77 algorithms. The first subsection presents a complete BWT architecture based on the Weavesorter machine. The second subsection describes the modifications made to the BWT architecture to reuse memory units and execute the LZ77 algorithm based on the CAM approach. The IP core is designed using the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) language and is validated with simulation tools.

4.1 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 work is based on it. In [13] the design of the machine is shown but a complete functional model is not defined. In this subsection a module to operate a Weavesorter machine is proposed. Two elements are added to the Weavesorter machine: A shift register called *ControlShiftRegister* to store the control bits which blocks the *compare/swap* operation. An *n*-input AND logic gate to detect when all the bits in the *ControlShiftRegister* are set to "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. See Figure 5.

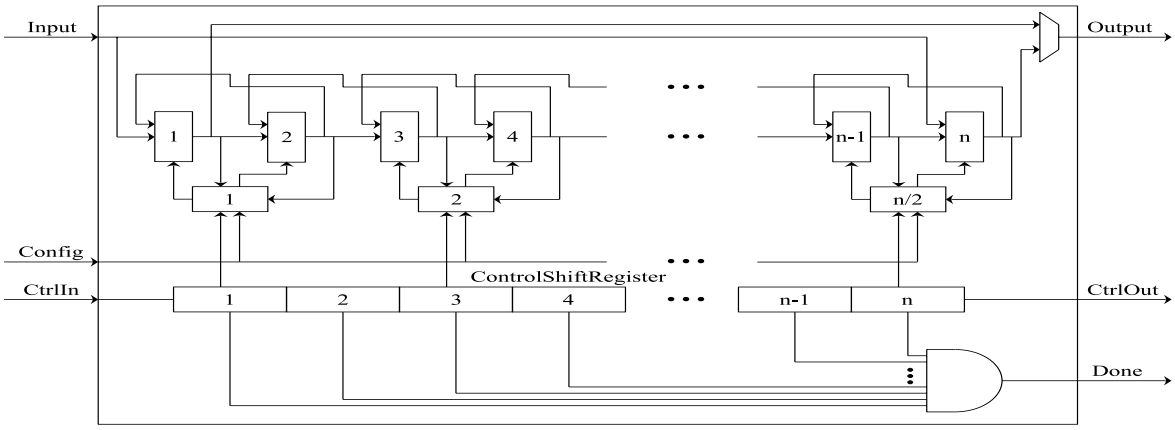

Fig. 5. Complete Weavesorter machine of *n* cells.

When the shifting process of the Weavesorter is done, the first column of the matrix is sorted. But if there are two or more 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. As 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) whose diagram is shown in Figure 6. 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 to the right while the last column is calculated together with the position of the original string.

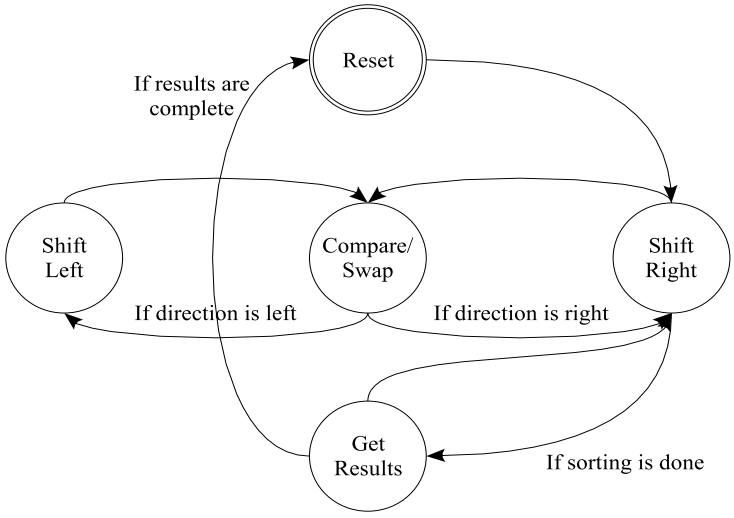

Fig. 6. FSM to operate the Weavesorter machine.

4.2 The LZ77 architecture

As one of the goals of this work 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 achieved and the same requirement applies for the BWT Weavesorter approach. Thus if this structure is shared between the two algorithms, a reuse of hardware can be achieved. It means that the larger the shift register, the better the compression ratio for both algorithms and also the more resources are reused. In this subsection the adaptation of the Weavesorter machine to execute the LZ77 algorithm is described and a mechanism to generate the LZ77 tokens is proposed.

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 to the 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. The modified cell has an input called *Search* to read the searching symbol. If this symbol is stored in the cell, the output signal *Found* is a logical "1", else a logical "0". This change allows searching the dictionary in parallel. A modified Weavesorter machine is shown in Figure 7. 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 called LZ77 token generator and is conformed by additional elements attached to the Weavesorter machine. These elements must 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". To find the largest matching string, the position of a symbol found in the dictionary is stored in a record and the token generator 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. When several symbols are found there are several matching strings and the token generator must find the largest one. 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 deleted. The position of the largest matching string is the last deleted record. If two or more positions are the last ones to be erased, the token generator takes the right-most position of the record that corresponds to the right-most cell. This is useful when an additional data compression algorithm like Huffman coding [11] will compress the tokens, with smaller values of (T_o) the compression ratio is improved [15]. 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 set to "0" 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 LZ77 token generator proposed in this work is shown in Figure 8. The cells of the Weavesorter machine are in the bottom of the figure.

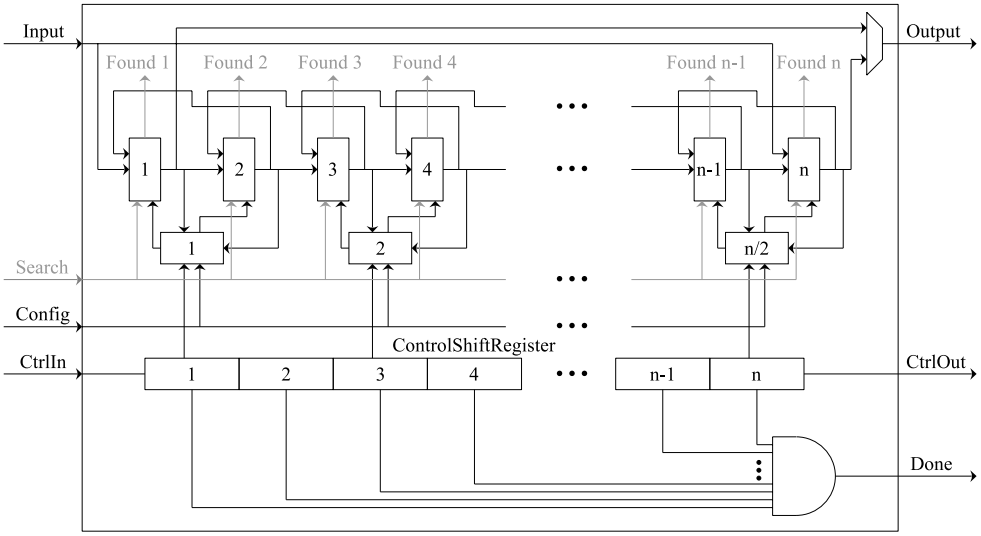

Fig. 7. Modified Weavesorter machine.

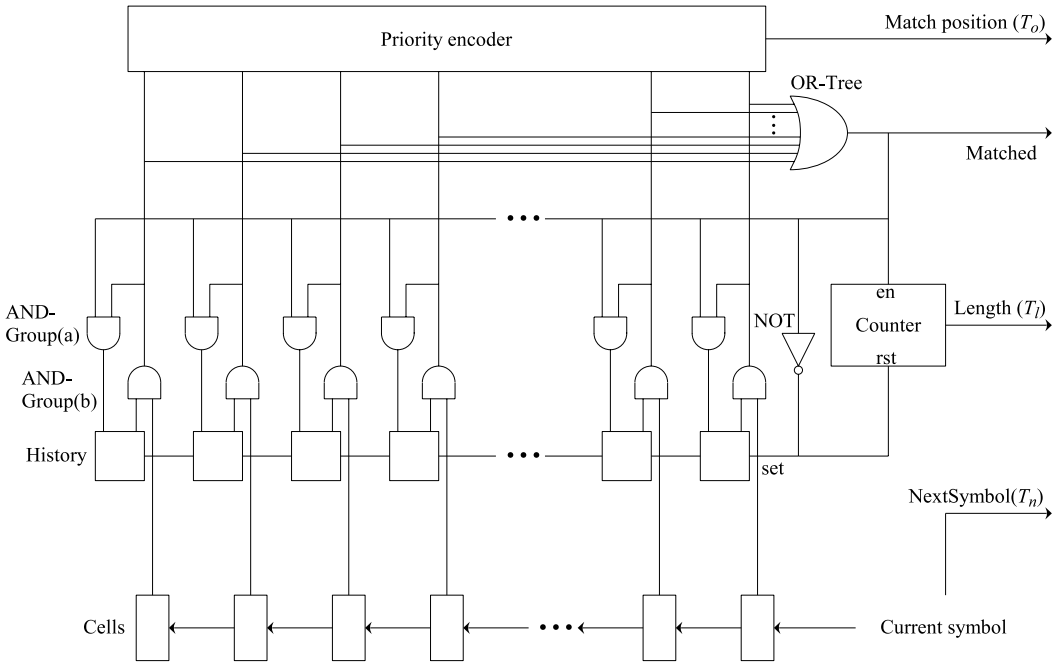

Fig. 8. The proposed LZ77 token generator.

The Control unit must also be modified to execute the LZ77 algorithm. As the LZ77 token generator constructs the token, the FSM of the Control unit deals only with the process of sending and receiving the data and the modifications needed are few. A new state, called LZ77 is connected to the Reset state as shown in Figure 9.

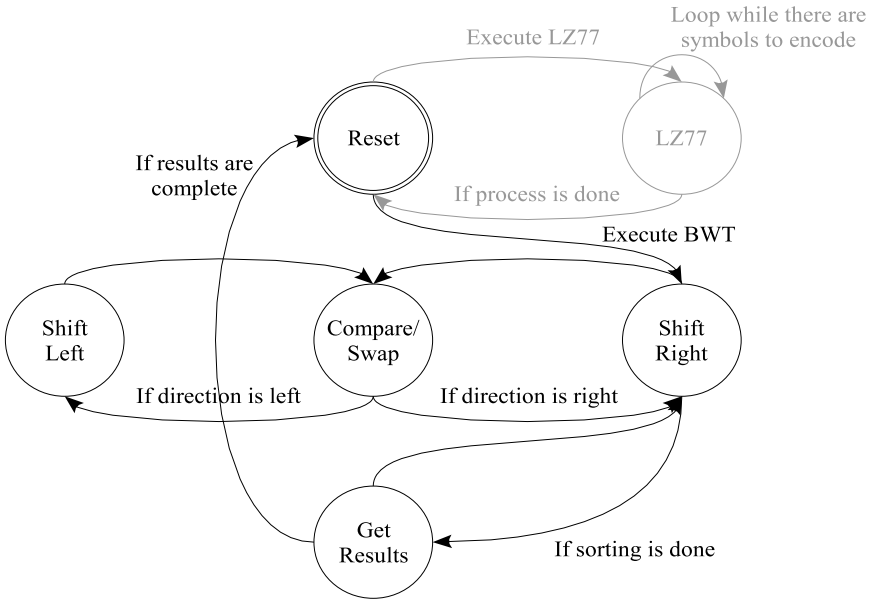

Fig. 9. New LZ77 state added to the FSM.

5 Interface with the LEON2 platform

In this section the LEON2 platform is introduced and its interface with the BWT/LZ77 IP core is explained. The LEON2 platform has a Floating Point Unit that uses an interface called Meiko. The core is modified to be compatible with this interface.

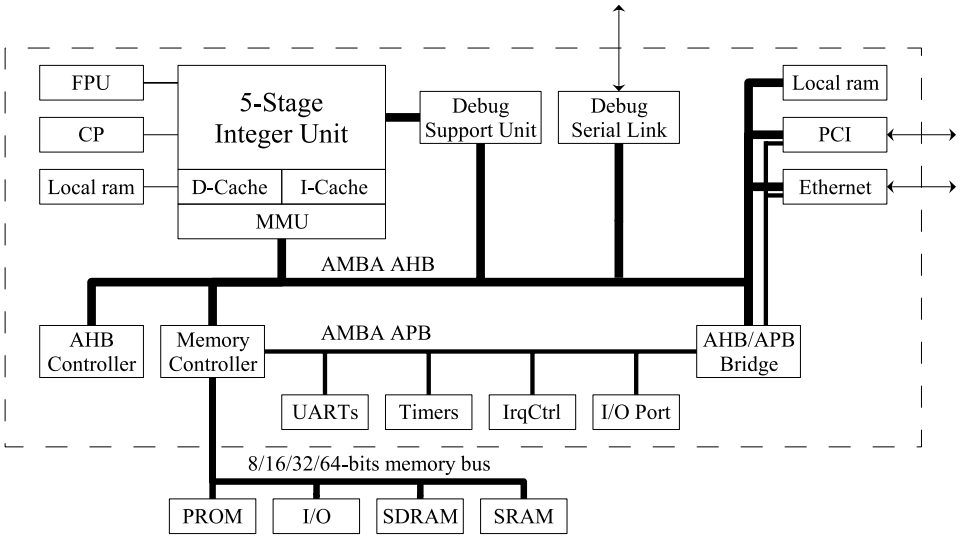

Fig. 10. The LEON2 processor.

LEON2 [7] is a System on Chip (SoC) platform with a 32-bit Scalar Processor Architecture (SPARC) V8 [17] compatible embedded processor. It was developed by the European Space Agency (ESA) and is freely available [6] with full source code in VHDL under GNU Lesser General Public License (LGPL) [8]. The LEON2 platform is extensively configurable and may be efficiently implemented on both FPGA and ASIC technologies. A LEON2 diagram is shown in Figure 10. The SPARC V8 architecture defines two (optional) coprocessors: one Floating-Point Unit (FPU) and one custom-defined coprocessor. The LEON2 pipeline provides one interface port for each of these units. Three different FPUs can be interfaced: Gaisler Research Floating-Point Unit (GRFPU) (available from Gaisler Research [6]), the Meiko FPU core (available from Sun Microsystems [18]) and the incomplete LTH FPU. A generic coprocessor interface is provided to allow interfacing of custom coprocessors. The LTH FPU is a partial implementation of an IEEE-754 compatible FPU contributed by Martin Kasprzyk from the Lund Technical University and is used in this work because of its free-source nature. This FPU uses the Meiko compatible interface and is replaced by the BWT/LZ77 IP core. The interface is a wrapper around compatible Meiko FPU cores and is described using VHDL records to connect with the Integer Unit (IU) of the LEON2 processor. Figure 11 shows the interface connected to the LEON2 IU.

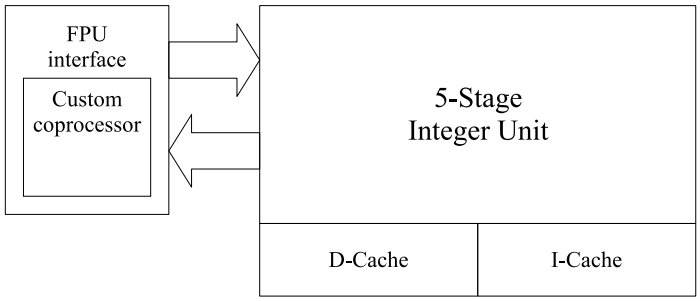

Fig. 11. The Coprocessor connected to the LEON2.

6 LEON2 platform configuration

The LEON2 model is highly configurable, allowing the model to be customized for a certain application or target technology. The model is configured through a number of constant records to configure a specific module/function. The declaration of the configuration record types is in the TARGET package, while the specification of the configuration records is defined in the DEVICE package. A graphical configuration tool based on the linux kernel *tkconfig* scripts is used to configure the model. This tool, in fact, modifies the "device.vhd" file that is the package to select current device configuration. To enable an FPU module for the IU, the `fpuen` record is set to `1`. To select the LTH FPU module, the `core` record is set to `lth`, the `interface` record is set to `serial` and the `fregs` record is set to `32`. To read the data from a Synchronous Synchronous Dynamic RAM (SDRAM), the `sdramen` record is set to `true`. All the VHDL files of the coprocessor are copied to the "leon" folder. The "fpu_core.vhd" file is modified to instantiate the designed coprocessor: The component of the coprocessor is written in the architecture definition and the generation of the LTH FPU is replaced by the generation of the coprocessor.

6.1 The extended instruction set

SPARC is an Instruction Set Architecture (ISA) with 32-bit integer and 32-, 64-, and 128-bit IEEE Standard 754 floating-point as its principal data types. It defines general-purpose integer, floating-point, and special state/status registers and 72 basic instruction operations, all encoded in 32-bit wide instruction formats. The BWT/LZ77 core replaces the FPU in the LEON2 and uses its FPU interface, then the operations of the core must be decoded using SPARC floating-point instructions. Table 1 shows the five instructions selected according to the needs of the coprocessor operations.

Table 1 Operations performed by the coprocessor.

SPARC floating-point instruction	Operation	Function
FADDd	ReadData	Reads 16 symbols from input and writes them in the coprocessor.
FSQRTd	ExecuteBWT	Moves to the ShiftRight state to begin the BWT process.
FSQRTs	ExecuteLZ77	Moves to the LZ77 state to begin the LZ77 process.
FSUBd	WriteData	Writes 8 symbols from the coprocessor.
FMULd	ResetCoprocessor	Moves to the Reset state.

6.2 Programming the LEON2 processor

Once the LEON2 platform is configured to handle the BWT/LZ77 IP core, the LEON2 processor must be programmed to execute the data compression algorithms. Being SPARC V8 compliant, compilers for SPARC V8 can be used with LEON2. In this work, the RCC cross-compiler system is used for the software development. RCC is a multi-platform development system based on the GNU family of freely available tools. RCC allows cross-compilation of C language applications for the LEON2 platform. To test the operation of the coprocessor two programs are written in C language including procedures in assembler language for SPARC standard [19]. The pseudocode for the BWT and LZ77 execution is shown in Figure 12. The procedures of both algorithms are written in assembler language to have access to the SPARC V8 instructions.

```
1: SetReadAddr()                    1: SetReadAddr()
2: SetWriteAddr()                   2: SetWriteAddr()
3: while notEOF do                  3: while notEOF do
4:      ReadString()                4:      ReadString()
5:      BWT()                       5:      LZ77()
6:      WriteColumns()              6:      WriteTokens()
7:      ResetCoprocessor()          7:      ResetCoprocessor()
8:end while                         8:end while
```

Fig. 12. Main BWT and LZ77 programs for the LEON2 processor.

In line 1:, the *SetReadAddr()* procedure sets the first memory allocation where the processing file is stored. In line 2:, the *SetWriteAddr()* procedure sets the first memory allocation where the results of the process are written. The *ReadData*() procedure in line 4: calculates the addresses to be read from the memory and executes the FADDd instruction to send the data to the coprocessor. In line 5: the *BWT()* procedure executes the FSQRTd instruction and the BWT process is started. Once the BWT operation is done, the results are read from the coprocessor and stored in memory by the *WriteColumns()* procedure that executes the FSUBd instruction (line 6:). Finally, the coprocessor is set to the Reset state by the *ResetCoprocessor*() procedure in line 7: executing the FMULd instruction. The process is repeated until the end of file (line 3:) is reached. The pseudocode for the LZ77 execution is very similar to the BWT one, the only changes are the *LZ77()* and *WriteTokens()* procedures in lines 5: and 6: respectively. The *LZ77()* procedure executes the FSQRTs instruction to perform the LZ77 algorithm and the *WriteTokens()* procedure reads the LZ77 tokens from the coprocessor and writes them into memory using the FSUBd instruction.

7 Implementation and results

To test the operation of the system the ModelSim SE PLUS 6.0d simulation tool was used and the synthesis results were obtained with the ISE Foundation 6.1i software. A comparison of the performance between Quicksort and Heapsort software implementations [4] and the coprocessor is also presented. These software approaches are two of the fastest sorting algorithms.

7.1 Simulation

The LEON2 provides a generic test bench ("tbgen.vhd") that allows to generate a model of a LEON2 system with various memory sizes/types by setting the appropriate generics. The file "tbleon.vhd" contains a number of alternative configurations using the generic test bench. To test the coprocessor the TB_FUNC_SDRAM test bench is selected because it allows to operate a SDRAM module where the original data can be stored. The test bench loads a test program in the LEON2 processor that among other operations executes a checking routine for the FPU. The C code for this test is provided with the model in the "fpu.c" file and the code to execute the BWT and LZ77 algorithms is written there. The test program must be recompiled with the RCC cross-compiler to generate a new assembler code for the TB_FUNC_SDRAM test bench. The compiler writes the assembler code in the "sdram.rec" file as the Motorola S-record format that simulates the SDRAM memory. Motorola S-records are an industry-standard format for transmitting binary files to target systems and PROM programmers. When the LEON2 model is simulated, the instructions are read from this file. To validate the operation of the system, an extra module is added to the output of the coprocessor, this module is for testing purposes only and it writes the results into a text file. The BWT and LZ77 algorithms are prototyped in Matlab and the results are also written to a text file. Both resulting files are identical.

7.2 Timing

The throughput of the coprocessor is estimated assuming that the LEON2 processor has already read the data from the SDRAM and they are stored in the floating-point registers. This is because the SDRAM read/write operations depend on the Memory Controller connected to the Advanced Microcontroller Bus Architecture (AMBA) bus (Figure 10) and the specific model of the SDRAM module. It is worth noting that these operations must be performed also in a software implementation. The number of clock cycles needed for the communication between the LEON2 and the coprocessor is fixed but it is not in the sorting operation for the BWT algorithm. This is because the Weavesorter performs a *shift-left* and *shift-right* iteration until all the characters are sorted, then the number of iterations depends directly on the kind of data. The Weavesorter takes n x 2 x 2 clock cycles to shift in and out the first column of the matrix. However in the next iteration, the second column is inserted while the first one is shifted out, it means that every extra iteration takes only n x 2 clock cycles. The number of iterations is equal to the length of the largest string segment repeated in the block, this number is represented by ι. Therefore, the total clock cycles requested by the Weavesorter is $2n\ (\iota + 1)$. On the other hand, the throughput for the LZ77 algorithm is 1 symbol per clock cycle, the same as the CAM-based approach. Table 2 shows a comparison between the coprocessor and software implementations of the Quicksort and Heapsort algorithms. A block of 128 bits is sorted for every file of the Canterbury Corpus. On average, the coprocessor is 3.6 times faster than the Quicksort software implementation and 1.6 times faster than the Heapsort one.

Table 2 Comparison with software sorting algorithms.

Files	Quicksort (ms)	Heapsort (ms)	Coprocessor (ms)
alice29.txt	4.597	3.542	0.679
asyoulik.txt	4.430	3.621	0.844
cp.html	3.668	3.706	0.741
fields.c	4.356	3.575	1.010
grammar.lsp	4.981	3.422	1.783
kennedy.xls	5.900	3.329	0.689
lcet10.txt	5.486	3.417	0.782
plrabn12.txt	5.026	3.504	0.669
ptt5	21.211	1.307	10.597
sum	14.353	1.915	2.867
xargs.1	4.469	3.575	1.081
Average	7.134	3.173	1.976

7.3 Synthesis

The coprocessor architecture is synthesized with a Weavesorter of 64 cells. The selected device is a Virtex-II 2v2000bg575-4. According to the synthesis report, the maximum clock frequency is 101.309 MHz. A device utilization summary is presented in Table 3.

Table 3 Device utilization summary.

Slices	7870 out of 10752	73%
Slice Flip Flops	3438 out of 21504	15%
4 input LUTs	14416 out of 21504	67%
bonded IOBs	205 out of 408	50%
TBUFs	128 out of 5376	6%
GCLKs	2 out of 16	12%

To find which components are shared between algorithms, the coprocessor is synthesized only with the components of the BWT algorithm. On the other hand, a synthesis with only the LZ77 algorithm components is also performed. Comparing with a synthesis of the complete coprocessor, the elements found in both algorithms are shared. From the data obtained it is possible to find the common elements shown in Table 4.

Table 4 Common Elements between the BWT and LZ77 models.

FSMs	1
Registers	65
8-bit register	65
Multiplexers	57
8-bit 64-to-1 multiplexer	1
2-to-1 multiplexer	56
Tristates	64
2-bit tristate buffer	64
Adders/Subtractors	3
32-bit adder	1
29-bit adder	1
28-bit adder	1
Comparators	3
32-bit comparator less	2
32-bit comparator greatequal	1

To obtain a percentage of the reused hardware, a summary of the device utilization for the BWT and LZ77 schemes is obtained. The total number of slices needed to design both algorithms separately is 3680 + 5707 = 9387. As seen in Table 3 the coprocessor designed in this work uses 7870 saving 1517 slices on the FPGA, in other words, 19.2% of the resources is shared.

8 Conclusions

This work proposed a hardware/software architecture with a general purpose microprocessor (LEON2) and a custom coprocessor. The microprocessor maintains the flexibility of the system while the coprocessor takes advantage of parallelism. Thus, the architecture combines the hardware and software approaches to make the most of the two solutions.

The architecture was designed to combine two different hardware lossless data compression algorithms in such a way that they had common elements, increasing resource utilization. The analysis of the synthesis reports leads to calculate a resource utilization of 19.2% of the total FPGA slices used by the coprocessor, this reduces the implementation costs and allows to increase the number of cells, improving the compression ratio.
Some advantages of the presented work are: The architecture can execute two different lossless data compression schemes. This flexibility is useful since both algorithms have different compression ratios and execution speed. The architecture favours compression ratio using the BWT capabilities and also can achieve high speeds with the LZ77 scheme. The design of the coprocessor does not introduce delays to the state of the art data compression schemes, this means that once the coprocessor has the data input available, the processing rate is the same as the original Weavesorter machine ($2n$ (ι + 1)) and CAM-based (1 symbol per clock cycle) architectures. The coprocessor is closely attached to the IU of the LEON2 processor, this reduces the clock cycles needed for communication and data transfer unlike a universal bus scheme. Since a universal bus is designed to handle more than one processor, the communication with the IU is arbitrated and this introduces delays.

Acknowledgments

The authors acknowledge the financial support from the Mexican National Council for Science and Technology (CONACyT), grant number 181512.

References

1. N. Abramson. *Information Theory and Coding*. McGraw-Hill, New York, 1963.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.
3. J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, Apr. 1984.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, second edition, 2001.
5. C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In R. Hartenstein and M. Glesner, editors, *th International Workshop on Field-Programmable Logic and Compilers*, pages 126–135. Springer-Verlag, Apr. 1996.
6. Gaisler Research. http://www.gaisler.com/. [Accessed September, 2005].
7. Gaisler Research. *LEON2 Processor User's Manual*, 1.0.30 edition, July 2005. http://www.gaisler.com/, [Accessed September, 2005].
8. GNU Lesser General Public License. http://www.gnu.org/copyleft/, Feb. 1999. [Accessed September, 2005].
9. S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, and R. Laufer. PipeRench: A co-processor for streaming multimedia acceleration. In D. DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Computer Architecture News, pages 28–41, New York, N.Y., May 1999. ACM Press.
10. J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In K. L. C. S. Press, editor, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, Apr. 1997. IEEE Computer Society Press.
11. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Electronics and Radio Engineers*, 4(9):1098–1101, Sept. 1952.
12. S. Jones. 100 Mbit/s adaptive data compressor design using selectively shiftable content-addressable memory. *IEE Proceedings-G*, 139(4):498–502, Aug. 1992.
    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*, pages 58–63, June 2001.
13. M. Nelson and J.-L. Gailly. *The Data Compression Book*. M&T Books, second edition, 1996.
14. D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, third edition, 2004.

15. J. Seward. http://www.bzip.org/. [Accessed September, 2005].
16. SPARC International Inc. *The SPARC Architecture Manual, Version 8*, 1992. http://www.sparc.org/, [Accessed September, 2005].
17. Sun Microsystems. http://www.sun.com/. [Accessed September, 2005].
18. Sun Microsystems, Inc. *SPARC Assembly Language Reference Manual*, May 2002. http://docs.sun.com/, [Accessed September, 2005].
19. R.-Y. Yang and C.-Y. Lee. High-throughput data compressor designs using content addressable memory. In *International Symposium on Circuits and Systems*, pages 147–150, 1994.
20. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

Virgilio Zuñiga *was born in 1979 in Guadalajara Mexico. In 2003, he obtained a B.Eng. degree on Electronics and Communications from the Exact Sciences and Engineering Centre, University of Guadalajara. In 2006, at the National Institute for Astrophysics, Optics and Electronics he received a M.Sc. degree in Computer Science. Nowadays, he is a PhD student in the System Level Integration research group at the University of Edinburgh, Scotland.*

Claudia Feregrino Uribe *received the M.Sc. degree from CINVESTAV Guadalajara, Mexico in 1997 and the PhD degree from Loughborough University, U.K, in 2001. Currently she is a researcher at Computer Science Department at INAOE. Her research interests cover the use of FPGA technologies, data compression and cryptography, steganography and software/hardware development for medical applications. She is a founding member of the ReConFig (Reconfigurable Computing and FPGAs) conference, which is one of the leading international forums for FPGAs. She is the chair of the Puebla IEEE Computer Chapter.*

René Armando Cumplido Parra *received the B.Eng. from the Instituto Tecnologico de Queretaro, Mexico, in 1995. He received the M.Sc. degree from CINVESTAV Guadalajara, Mexico, in 1997 and the Ph.D. degree from Loughborough University, UK in 2001. Since 2002 he is a member of the Computer Science Department at the National Institute for Astrophysics, Optics, and Electronics in Puebla, Mexico. His research interests include the use of FPGA technologies, digital design, computer architecture, reconfigurable computing, DSP, radar signal processing, software radio and digital communications. He is a founding member of ReConFig international conference.*