J Real-Time Image Proc (2006) 1:163–170
DOI 10.1007/s11554-006-0005-x

ORIGINAL RESEARCH PAPER

FPGA-based architecture for hardware compression/decompression of wide format images

M. Akil · L. Perroton · T. Grandpierre

Received: 28 March 2006 / Accepted: 7 June 2006 / Published online: 21 September 2006

Abstract In this article, we present a popular lossless compression/decompression algorithm, GZIP, and the study to implement it on an FPGA-based architecture, the ADM-XRC board from ALPHA DATA parallel system ltd. The algorithm is lossless, and applied to ‘‘bi-level’’ images of large size (A0 format). It ensures a minimum compression rate for the images we are considering. It aims to decrease storage requirements and transfer times, which are critical for wide format printing systems. In a wide format document industry, raster data are most of time processed in an uncompressed format, in order to apply processing (P) before printing (p). An example of a copy chain is composed of scanner, set of processing operations, storage, link and printer. We propose to use a compressed format as the new data-flow representation to improve the performances of the printing system. For example, the compression (C) is applied as soon as the data are produced by the scanner, and decompression (D) is performed at the last stage, before printing. The set of processing is applied to compressed images. The proposed architecture for the compressor is based on a hash table and the decompressor is based on a parallel decoder of the Huffman codes. We implemented the proposed architecture for compression and decompression algorithms on FPGA Xilinx Virtex XCV 400.

M. Akil (✉) · L. Perroton · T. Grandpierre
Laboratoire A2SI, ESIEE, 2Bd Blaise Pascal, BP99,
93162 Noisy-le-Grand Cedex, France
e-mail: akilm@esiee.fr

L. Perroton
e-mail: perrotol@esiee.fr
URL: www.esiee.fr/~perrotol

Keywords Lossless large images compression · Wide format image · Printing systems · Hardware-based compression/decompression · GZIP · FPGA

1 Introduction

In the context of industrial applications, the digital systems for transmission and processing must mainly satisfy constraints of performances and costs. The context of our study is within the field of professional reproduction of documents. Some images to be printed may be very large (A0 format, and up to 15 m long, for example, handled by the Océ 9600 printer machine) and of different types (bi-level pictures, full-color, posters, maps, plans, etc.). The size of the portable bitmap files (PBM) may be very large (several hundreds of mega bytes) and usually requires important transfer times. The improvement of the productivity of the printer platforms requires, among other features, to decrease the transfer time of the files to be printed, hence, the importance of the data compression *(transfers and manipulation)* and the decompression *(before being printed)*. The compression must be lossless, with a minimum ensured rate for all of the image types previously mentioned. The minimum required compression rate is 2. There are mainly two types of lossless compression techniques: the one based on the redundancy of the encoding of data (Huffman coding, arithmetical and statistical) and the other based on the redundancy between pixels (RLC/RLE, LZW, JBIG, ...). We performed an algorithmic and a performance study *(compression rate and complexity)* and have selected two algorithms, namely, GZIP [1] and

JBIG [2]. A comparative study of these two algorithms has been performed: evaluation of the compression rate, compression rate for the type of images we are considering, compression time, resources used by the algorithm (memory, ...). This study showed that GZIP has a minimum compression rate of 2 (from 3.8:1 to 17.4:1) better than JBIG (from 2:1 to 28:1) for the type of images we are considering in our application (test performed on PC VL800, Pentium IV, 1.5 GHz, with 256 MB) [3].

While the compression stage of the data to be printed usually takes place on a computer, the decompression will take place on a peripheral device, for example, such as a printer. Hence, the need to develop a *"standalone"* decompression module which can take place within the target peripheral device. Another solution is to have a hardware compressor in order to minimize the transfer time between the different printing systems (a printing system including a printer and a computer). Depending on the type of applications and the platforms, one can use a hardware solution (hardware compression/decompression) or a hardware/software solution (software compression and hardware decompression).

We present in this article the results of the study of an architecture which implements a compressor/decompressor conforming to the GZIP standard. In Sect. 2, we describe the different encoding modes and compression modes of the GZIP format. Section 3 describes a study of a compressor in a "fixed" compression mode.The different architectures of the decoder, which is the main component of the decompressor for the three compression modes (i.e. stored, fixed and dynamic modes) are analyzed in Sect. 4. We have selected a parallel decoding approach because of its better performances. We present the complete architecture and some results of the implementation in Sect. 5. Conclusions are finally provided in Sect. 6.

2 Lossless algorithm: GZIP compression and decompression principle

The GZIP algorithm combines three compression methods. The first one is a run length encoding compression based on the *dictionary* approach. In this method, a string which has already been encountered in the data to be compressed will be substituted by a couple (L,D) where L is its length and D the distance of the previous occurrence of the string (only strings longer than three characters will be encoded this way due to the small over cost of the encoding of the couples). Hence, such a compressed file will contain *strings* of characters and *couples* (L,D).

In order to improve the compression, GZIP encodes the character flow with a Huffman code, whose principle is to associate a binary code with each character whose length is conversely proportional to the frequency of the character. Finally, the *alphabet* compression method is applied to the resulting Huffman codes.

2.1 Description of the different encoding methods

Concerning the **dictionary**-based compression, the method described in [1] uses the same technique to encodes the distances and the length: the code is subdivided in two fields, a base taken from a table, and additional *extra-bits* concatenated to the base to form the final data. The representation of the data *character/length* and *distance* is as follows: ASCII characters from 0 to 255 with no extra-bits, end of block is 256, from 257 to 287 (30 values) with 0 to 5 extra-bits are the length from 3 to 258. Then the distances with values 0 to 31 with 0 to 13 extra-bits represent the distances from 1 to 32,768. An entry in the table characters/length bases includes the number of extra-bits, together with the interval of the encoded lengths [V_{base},V_{limit}]. For example, to encode a length of L = 20, 20 corresponds to the entry i = 269, V_{base} = 19 and V_{limit} = 22. The 9-bit code of 269 is $(100001101)_2$, 20 is equal to 19+1 and is therefore coded with 2 extra-bits; $(01)_2$. The value 20 will be finally encoded as $(100001101)_2(01)_2$.

The application of the dictionary encoding to the string CCABDEFFFCCABDETFFFABCDEFIABC DEF will give the following string: CCABD EFFF(6,9)T(3,10)ABCDEFI(6,7). One can see that the second occurrence of the string CCABDE of length 6 is 9 characters away from its previous occurrence and is replaced by the couple (6,9). This method brings a compression rate of τ_c = 23.4% of the string.

The **Huffman code** uses a binary tree to determine the binary codes of the characters. For example, let us consider the string BBABAIL. The occurrences of the characters B, A, I and L are, respectively, 3, 2, 1 and 1. The binary code of the characters B, A, I and L are, respectively, 0, 10, 110 and 111. The binary code of each character is extracted from the binary Huffman tree (the length of the Huffman codes are variable, from 1 to 15 bits).

In the GZIP algorithm, the Huffman tree construction follows two more rules: (1) all the consecutive codes with the same number of bits are sorted in the *same order as the symbol* they represent, and (2) the value of an n bits code must be *smaller* than any of the ones of more than n bits. Therefore, the characters will be automatically sorted in the alphabetical order for a given level of the tree, starting from the left. The rule (2) enforces the following placement for a given level of the tree: the nodes are on the left and the leaves are on the right.

In order to still improve the compression rate, the GZIP algorithm builds the list of the *lengths* of the Huffman codes of the characters sorted in the alphabetical order. The corresponding Huffman codes can then be uniquely recovered from the series of their length instead of storing the codes themselves.

The list of length is in turn compressed by the **alphabet** method. Any length appearing between three and six times is represented by ⟨*this length, 16, the number of occurrences (coded with 2 extra-bits)*⟩. Any series of 3 to 10 times the length 0 (non existent code) is replaced by ⟨*18, the number of occurrence of 0 coded on 3 extra-bits*⟩.

For example, the following list of the Huffman code length 0005522222 will be encoded as ⟨18, 3⟩⟨55⟩⟨2,16,5⟩. For this list, the description alphabet consists in 2, 3, 5, 16 and 18. These characters are also themselves encoded with corresponding Huffman codes.

2.2 The different modes of compression

The GZIP algorithm has three different compression modes: **stored**, **fixed**, and **dynamic**. The header of a block includes two fields: the first one tells if it is the last block of the file, the other (2 bits) identifies the compression mode of the following data. Next comes the data. For example in the *stored* mode, the data block is as follows: [LEN (2 bytes)] which tells the number of bytes in the data block, [$\mathrm{NLEN} = \overline{\mathrm{LEN}}$] (if $\mathrm{LEN} \neq \overline{\mathrm{NLEN}}$ then the compressed file is corrupted), and then the data of the block. In the *stored* mode, the data are not compressed. In the *fixed* mode, the frequency of the characters is estimated in advance, and therefore the Huffman code for the characters/length and distances is predefined in two tables. However, in the dynamic mode, the Huffman codes are generated at compression time, in a function of the actual frequencies of the characters in the input block. The GZIP algorithm analyzes the input file to determine the best mode to use for each block. In the dynamic mode, the compression process involves the following steps: (1) dictionary encoding, (2) construction of the table of the frequency of each character, lengths and distances, (the Huffman code lengths can then be deduced as well as the codes themselves for the characters, lengths and distances), (3) encoding by the alphabet method previously described by the successive repetitions of the *lengths* of the Huffman codes of the characters, lengths and distances, (4) encoding of the list of the length of the codes by the Huffman method.

Let us consider the string ABCDEFABCD, the second occurrence of ABCD is replaced by ⟨4,5⟩. We then get ABCDEF⟨4,5⟩. The frequencies of the occurrence of the characters are A = B = C = D = 2 and E = F = 1. The length of the Huffman codes is 2 bits for A, B, C and D, respectively, 00, 01, 10 and 11. The Huffman codes of E and F are 000 and 001. The distance 5, occurring once, has a code of length 1 equal to 0. The code of the entire string is 00,01,10,11,000,001,⟨010,0⟩. The list of the lengths of the "characters/length" codes is A(2)B(2)C(2)D(2) E(3)F(3)G(0)...Z(0)3(0)4(3)5(0)...287(0).

The list of the lengths of the "distance codes" is 1(0)2(0)...5(1)...24577(0). Using the alphabet method, we encode the successive repetitions of the same lengths of code, so [2222330...0] for the characters is encoded as 2,16(4),3,16(2),18 (number of repetition of "0") and 4,18(29). We proceed as well with the distances lengths. Finally, the resulting lists of the length of the Huffman codes are again encoded with the Huffman method.

2.3 Redundancies search and hash table

In order to optimize the search for redundancies, GZIP performs a sorting of the data using a hash table. The insertion of a new element in the table requires the computation of its index in the table using the "hash function". In order to optimize memory usage, the hash function is implemented by using two tables (H1 and H2) which actually contain a linked list structure.

The hash function is

$$F(h,c) = ((h_{\mathrm{Shift}})\ \mathrm{XOR}\, c)\ \mathrm{AND}\ (H_{\mathrm{Mask}}),$$

where c is the last character of the current triplet, h is the hashing value of the previous triplet of characters, H_{Shift} is a left shift, for the updating of the function (in this case, H_{Shift} is equal to 6 in order to keep the information of the two previous triplet of characters) and H_{Mask} is a mask inhibiting the most significant bit of the resulting function. This function is computed on 16 bits, whereas the hash table has 32 K entries (hence, a 15 address bus).

3 Architecture for hardware compression algorithm

The global architecture of the compressor is illustrated in Fig. 1 and is mainly composed of the block which computes the hash function for each new

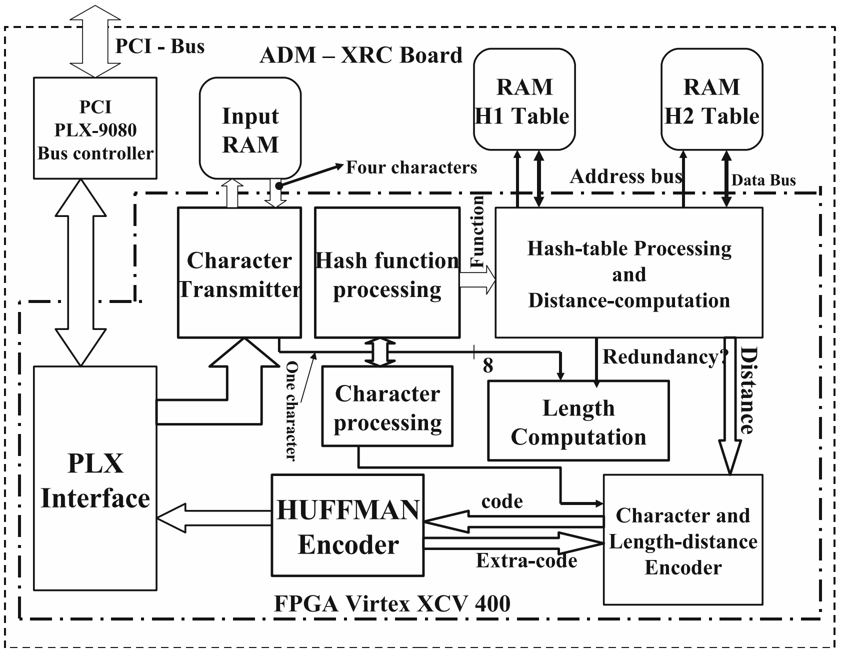

Fig. 1 Block diagram of the compressor

character, the blocks hash and distance (redundancies search and distance computation between the original string and the redundant one), the block length (length of the redundancies) and the block Huffman.

The management of the hash table uses two 32 K arrays. The *i*th entry of the first array holds the address of the last element having *i* has a hash value (the hash function has 32 K possible values) and the address, within the second array, of the next element having the same hash value. Then, the second array holds the address of the next element having also the same hash value, and so on, hence implementing a linked list. The search for redundancies is performed on a 32 K distances.

3.1 Implementation and validation of the compressor architecture

The evaluation of the main blocks of the architecture previously described leads to the following characteristics: hash function computation block: surface 11 CLBs, max freq 168 MHz, one cycle; hash table management and distance computation: surface 100 CLBs, max freq 49 MHz, this block performs a write access to the hash table (three memory access, plus three processing cycles) and one search operation (one memory access and two processing cycles); the Huffman encoding block: surface 20 CLBs, max freq 49 MHz, requires one clock cycle.

Globally, the architecture of the compressor in the fixed mode (i.e. Huffman codes are predefined) including the other blocks (character, transmitter, character processing, length computation and character, and length/distance encoder) requires 450 CLBs, 10% of Virtex FPGA XCV 400, for a frequency of 50 MHz.

The complete compressor architecture has been implemented on the ADM-XRC board [4]. This low-cost board houses a single Virtex XCV 4000 FPGA bg560 component and four banks of the 512 KB ZBT SSRAM. ADM-XRC board supports high performance PCI operations through the PLC PC9080 interface. The XCV 400 FPGA component has 4,800 slices, 164 Kb of block RAM and 154 Kb of distributed RAM.

The PCI data transfer and data compression are done in parallel (when data block N is compressed, the following data block is transfered through the PCI bus to the FPGA).

4 Architecture for the hardware decompression algorithm

The most important stage during the decompression is the reconstruction of the Huffman codes from the list of the lengths of these codes. In the dynamic mode, in addition to the two previously mentioned fields (indication of the last block of the file and the compression mode used), a compressed block includes the following fields: number of codes used for the "characters/ length", number of codes used for the distances, number of codes used for the alphabet, list of the lengths of the codes of the alphabet, list of the Huffman codes of the alphabet and finally the compressed data.

The decoding of the list of the Huffman codes of the alphabet allows the reconstruction of the lengths of the codes of the "character/length" and the "distances" in two lists. The decoding of these lists allows the construction of the tables of the Huffman codes of the "character/length" and the "distances". It will then be possible to process the "compressed data" using these two tables. We have studied different decoding methods.

4.1 Different approaches to decode the compressed data

Serial bit decoding: In this approach, we assume that we have three tables: the first one for the Huffman codes in the order of the lengths of the codes, **T1**; the second one contains the list of the characters in the order of the lengths of the Huffman codes, **T2**; and the third one is the list of the number of Huffman codes for a given number of bits, **T3**. The serial decoding consists in a decoding bit after bit of the compressed data (the minimum code being on 2 bits). This code is extended to 15 bits and is compared to the Huffman codes in **T1**, **T3** giving the index in the **T1** table. If there is an equality, the corresponding character is read in **T2**. The rate of the data is variable from 1 to 287 cycles per character, as for each character we have to perform between 1 and 287 comparisons in order to find its Huffman code.

Buffer approach decoding: In order to decrease the decoding time, the Huffman codes represent an address within an associative memory. This memory contains the characters and the lengths of the corresponding Huffman codes. The process requires at most 15 comparisons for the decoding of a character (as the lengths of the Huffman codes vary from 1 to 15 bits).

This approach requires a memory capacity of 16 bits $\times$ 2^{15} (for the character/lengths codes) + 16 bits $\times$ 2^{15} (for the distance codes) + 32 KB (buffer to copy the couples length/distance) = 256 KB.

Parallel decoding: The principle of this method is to determine the length of the Huffman code and then the address of the character/length or distance. Using properties described in [5, 6], the table of the "minimum codes" is computed. From this table, we deduce the length of the Huffman code to be decoded and for each code length the value V_a = (Address_code_Min) – Code_Min is computed and inserted into the table *"Base memory"*. Let C_{HD} be the Huffman code that we want to decode. The address e of the "character/length" or "distance" is $C_{HD} + V_a$. The determination of the length of the code is performed in parallel. Only one comparison is necessary between the length to be decoded (coded in Huffman) and the minimum Huffman codes (minimum table codes, 15 $\times$ 15), thanks to parallel comparators (2, 3, ...,15 bits) since the Huffman codes are variables from 1 to 15 bits. The minimum Huffman codes are stored in the "minimum codes" (15 $\times$ 15) according to the following rules: the codes are sorted by increasing order of the code lengths, the code of length n which does not exist are replaced either by the n most significant bits of the code of length $(n + 1)$ or by "1" if the code of length $(n + 1)$ does not exist.

From the length of the Huffman code, one can determine the address of the corresponding "character/ length" or "distance". This solution requires only two clock cycles per character and 64 KB of memory; furthermore, the output rate is constant. The decoder architecture includes a barrel shifter which selects a 16-bit fields out of a 32-bit word, a *"max length"* mask which selects a set of 1 to 15 bits according to the maximum length of the Huffman code and the *ONE-HOT length* block.

Figure 2 represents the block diagram of the architecture of the Huffman codes decoder.

The different steps of the decoding are (1) the computation of the length of the Huffman code, (2) determination of the address in the external RAM of the character/length or distance, (3) reading it from the RAM, (4) decoding of the extra-bits and write back the final uncompressed character.

The extra-bits decoder includes a table in an internal ROM containing the bases of the lengths and the distances. An "extra-bit" mask selects a field of 13 bits (maximum number of extra-bits) from the output of the barrel shifter, a comparator to 256 determine the nature of the symbol: character, length or end of block. In the case of a length or a distance, the corresponding value addresses the ROM. The extra-bits (extended to 15) are added to the base (length or distance) in order to obtain the final code.

The block diagram of the parallel decoder is given in Fig. 3 and includes two sub-blocks: the Huffman codes decoder (length computation, determination of the address of the character/length or distance) and the extra-bits decoding (in the case of a length or a distance).

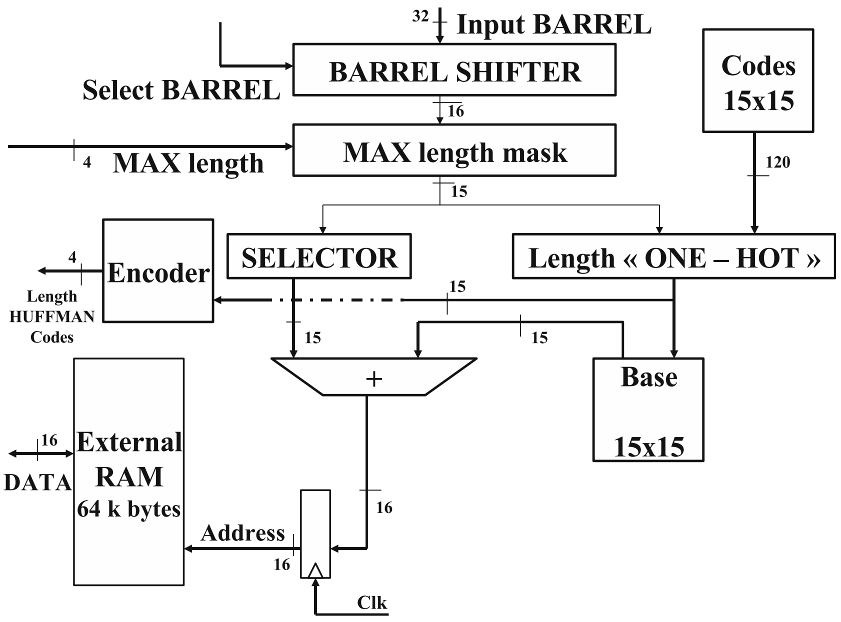

Fig. 2 Block diagram of the Huffman codes decoder

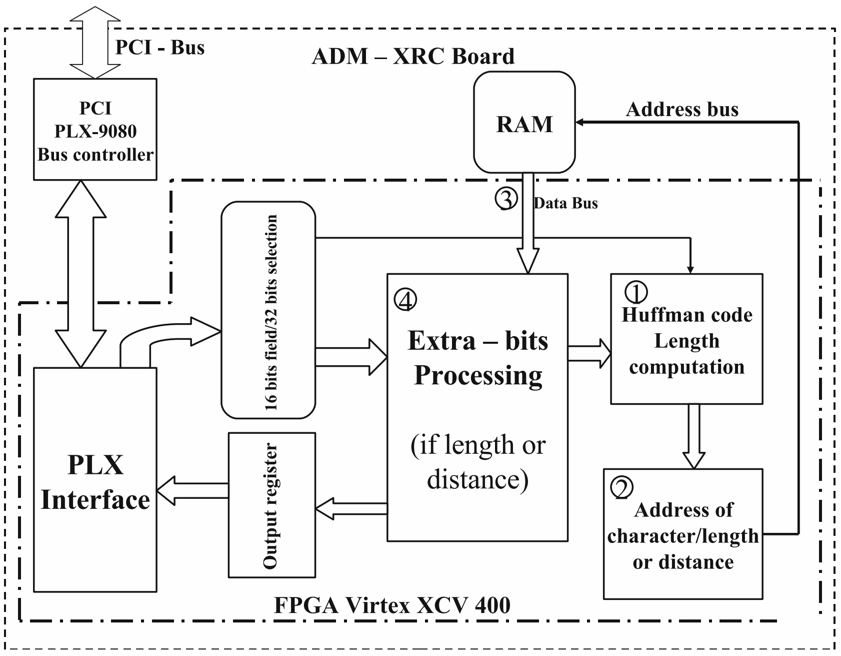
PCI - Bus
ADM – XRC Board
PCI PLX-9080 Bus controller
RAM
Address bus
③ Data Bus
16 bits field/32 bits selection
④ Extra – bits Processing (if length or distance)
① Huffman code Length computation
② Address of character/length or distance
PLX Interface
Output register
FPGA Virtex XCV 400

Fig. 3 Block diagram of the parallel decoder

The global architecture of the decompressor that we describe now is based on this parallel decoder.

5 Implementation and validation of the parallel decompressor

The architecture of the decompressor is depicted in Fig. 4 and includes I/O interfaces, the parallel decoder, the Huffman table constructor and a block to write to the external RAM.

In the case of the fixed mode, the block for the construction of the Huffman codes will generate the necessary tables for the decoder from the predefined Huffman codes (indexation table, table of the lengths of the minimum code) and the addresses of the minimum codes. The dynamic mode includes the following steps: construction of the Huffman tables of the

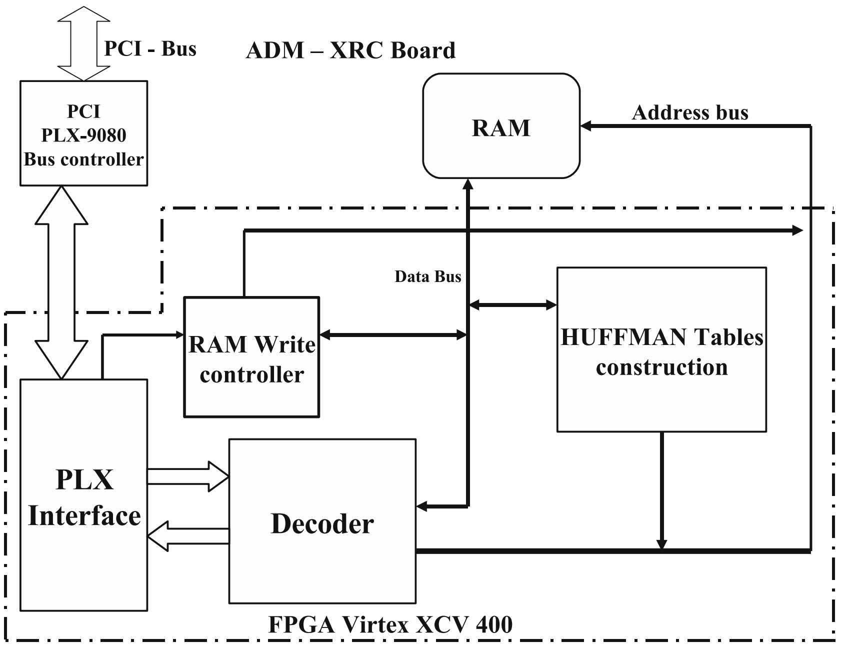

Fig. 4 General architecture of the decompressor

description alphabet, decoding of the lists of the length of the codes, construction of the Huffman table of the codes of the ‘‘character/length’’ and ‘‘distances’’ and finally the decoding of the compressed data. The indexation table is used to create the value table (alphabet, character/length or distances) and the one of the Huffman codes necessary to the construction of the tables: Base_Code-Min and Base_Memory.

The different stages of the ‘‘Huffman tables construction’’ block are initialization of the indexation table, computation of the occurrence of each code length, creation of the table of the addresses of the minimum codes for each code length, creation of the character/length or distances table and finally creation of the table of the Huffman codes.

Besides the reading of the header of the block (test of the ending block and compression mode used), the sequencer performs three steps: decompression in the fixed mode, decompression in the dynamic mode and decompression in the stored mode, in which it copies directly the input flow to the output.

The previously presented architecture has been designed and validated with the following CAD tools. The code has been written with VHDL, and functional simulation has been made with Qhsim (Mentor Graphics), synthesis made with ‘‘leonardo’’ and ‘‘Galiléo’’ (examplar) and the performance analysis (surface and speed). The implementation has been made with alliance (Xilinx), confirmation of the results of the synthesis, generation of the gates level VHDL files and of an SDF file (including propagation times of each elementary gate) and retro-annotated simulation with Qhsim. The RTL synthesis produces an EDIF file, associated with constraints (limit delay, surface) the routing will respect the proper criteria of the conception of the architecture. The routing stage gives several files including a bit stream (.bit), a VHDL retro-annotated for the simulations and a.sdf file specifying the associated delays to the retro-annotated VHDL.

We got the following results: surface = 650 CLBs, temporal performances = 50 MHz, 64 KB of external RAM (working memory).

We have validated the complete compressor with this XDM-XRC card and have performed comparison tests between a software compression (standard GZIP utility running on a Pentium IV PC VL800, 1.5 GHz) and a hardware compression (50 MHz FPGA, data transfer using PCI bus). The results show that our architecture is about two to three times faster than the software version.

6 Conclusions

In this study that we have performed in collaboration with Océ Industry Corp., we have proposed and validated an hardware implementation of the well-known GZIP standard decompression algorithm as well as the fixed mode compression. The decoder of the Huffman codes of the character/length or distance is based on a parallel approach. This architecture has been validated with the CAD tools as well as by an implementation on a PCI board FPGA XCV 400. We also implemented a modified version of the GZIP algorithm which decreases the memory requirements (32 KB, 450 CLBs and 50 MHz). The limiting factor for the clock frequency is the hash table management and distance computation.

We have also studied and implemented an architecture for the hardware compression for the ‘‘fixed’’ mode. The described architecture have been validated with the CAD tools and then implemented on another board XDM-XRC. This board has a Virtex 400 FPGA XCV 400 and four memory banks of 512 Kb each. This PCI board is better suited to the industrial constraints of the application.

We have then evaluated the implementation of the decompressor with the Virtex XCV 400 family and the XDM-XRC board (surface 650 CLBs, max freq 50 MHz). In the decompressor case, the frequency is limited by the construction of the Huffman table.

Our encoding and decoding architecture based on the Huffman technique have higher throughput and efficiently implemented on the FPGA circuit.

Our objective is to have a hardware demonstration kit which implements others compression–decompression algorithms (JPEG2000) and some hardware operations (such as image rotation, scaling, translation) on compressed data.

Acknowledgments We wish to thank all of the people involved in this project, and mainly F. Bartier from OCE-PLT and E. Llorens from ESIEE.

References

1. Gailly JL, Adler: GZIP documentation and sources. http://www.gzip.org/index-f.html
2. JBIG specifications: technical report. http://www.datacompression.info/JBIG.shtml
3. Lossless compression: algorithm analysis and performance evaluation‘‘. Internal technical report, ESIEE, June (2005)
4. XDM-XRC Xilinx reconfigurable board documentation. http://www.alpha-data.com/adm-xrc.html
5. Larmore LL, Prytycka TM: Construction of Huffman trees in parallel. SIAM J. Comput. 24(6):1163–1169 (1995)
6. Wei BWY, Meng TH: A parallel decoder of programmable Huffman codes. IEEE Trans. Circuits Syst. Video Technol. 5:175–178 (1995)

Mohamed Akil received his Ph.D. degree from the Montpellier university (France) in 1981 and his doctorat d’état from the Pierre et Marie curie University (Paris, France) in 1985. He his currently teaching and doing research in the position of Professor at the Laboratoire Algorithmique et Architecture des Systémes Informatiques, ESIEE, Paris. His research interests are architecture for image processing, image compression, reconfigurable architecture and FPGA, high-level design methodology for multi-FPGA, mixed architecture (DSP/FPGA) and System on Chip (SoC). Dr. Akil has more than 60 research papers in the above areas.

Laurent Perroton is currently associate professor at the Laboratoire Algorithme et Architecture des Systémes Informatiques at ESIEE, an engineering school near Paris in the field of electronics and computer science. He got his Ph.D. thesis in 1995 on ‘‘Parallel 3D segmentation‘‘ at the LIP, ENS at Lyon, France. His main research topics are image processing and parallel computing.

Thierry Grandpierre received his Ph.D. degree in electronics and computer science from Orsay-Paris Sud University (France) in 2000. Currently, he is researcher and teacher in the A2SI (Algorithmique et Architecture des Systémes Informatiques) laboratory of ESIEE, Paris. His research interests include embedded system design and methodology, CAD design for real-time applications executed onto multi-DSP, FPGA and SoC.