Design and Implementation of a Data Compression Scheme: A Partial Matching Approach

F. Choong, M. B. I. Reaz, T. C. Chin, F. Mohd-Yasin
Faculty of Engineering, Multimedia University, 63100 Cyberjaya, Selangor, Malaysia
{florence@mmu.edu.my}

Abstract

Data compression is an essential process due to the need to reduce the average time required to send messages and reduce the data size for storage purposes. There is a vital need for lossless compression especially for text and binary compression because it is important to ensure that the restructured text is identical to the original text. The predictive by partial matching (PPM) data compression scheme has set the performance standard in lossless compression throughout the past decade. PPM is chosen as it is capable of very good compression on a variety of data. In this paper, we present the realization of data compression using PPM on Altera FLEX10K FPGA device that allows for efficient hardware implementation. The PPM algorithm for binary data compression was successfully written and modeled in VHDL. The design is followed by the timing analysis and circuit synthesis for the validation, functionality and performance of the designated circuit which supports the practicality, advantages and effectiveness of the proposed hardware realization for the application. The designed was verified using both 16-bit input and 32-bit input. The hardware prototype utilized 1164 logic cells with a maximum system frequency of 95.3MHz.

1. Introduction

Data compression is an essential process due to the need to reduce the average time required to send messages and reduce the data size for storage purposes. The ultimate goal of data compression is to represent an information source, e.g. a text file, binary information, an image or a video signal, as accurately as possible using the fewest possible number of bits [1]. There is a vital need for lossless compression especially for text and binary compression as it is important to ensure that the restructured text is identical to the original text, because a very small difference or variation in statement can lead to a totally different meaning.

Prediction by Partial Matching (PPM) is a "finite context" statistical modeling technique that can be viewed as blending together several "fixed-order context" models to predict the next character in the input sequence. The "Prediction by Partial Matching" data compression scheme is capable of very good compression on a wide variety of source data. The adaptive nature of the scheme, and the flexibility afforded by arithmetic coding, mean that an effective compression model will be built for any input file that is reasonably homogeneous [2]. The original algorithm was first published in 1984 by Cleary and Witten [1], and a series of improvements was described by Moffat, culminating in a careful implementation, called PPMC, which has become the benchmark version [2]. This still achieves results superior to virtually all other compression methods, despite many attempts to better it.

In 1999, Charles Bloom [3] developed PPMZ which uses an adaptive second level model to estimate the optimum value as a function of the order, the total character count, number of unique characters, and the last one or two bytes of context. In 2002, Dmitry Shkarin [4] developed PPMII which is similar to PPMZ as it also uses a secondary escape model. PPMII does not use statistics from the longest matching context. Instead, PPMII inherits the statistics of shorter contexts to set the initial estimate when a longer context is encountered for the first time. PPMONSTR and PPMD are based on PPMII. PPMONSTR is a variation of PPMD that trades compression rate for execution speed. This project will concentrate on the original version of PPM. Other methods such as those based on Ziv.Lempel coding [5, 6] are more commonly used in practice, but their attractiveness lies in their relative speed rather than any superiority in compression. Indeed, their compression performance generally falls distinctly below that of PPM in practical benchmark tests [7].

With the advent of hardware description language, VHDL, the design process of a PPM lossless data compression scheme is simplified tremendously. VHDL is the acronym for VHSIC Hardware Description Language where VHSIC stands for Very High Speed Integrated Circuit. Its strength lies in its ability to model a digital system at many levels of abstraction, varying from algorithmic level to gate level allowing models of varying levels of complexity to be modeled. Designing in VHDL enables faster time-to-market, exploration of design space and also enables designers to decide on architectural trade-offs that would lead to lower costs. VHDL is highly favored by electronics designers as it allows rapid prototyping and adopts a standardized design flow.

Field Programmable Gate Array (FPGA) offers a potential alternative to speed up the hardware realization. From the perspective of computer-aided design, FPGA comes with the merits of lower cost, higher density, and shorter design cycle [8]. It comprises a wide variety of building blocks. Each block consists of programmable look-up table and storage registers, where interconnections among these blocks programmed through the hardware description language [9, 10]. This programmability and simplicity of FPGA made it favorable for prototyping digital system.

In this paper, the framework of FPGA-based hardware realization of data compression using PPM is proposed. With this approach, both the speed and compression performance are preserved without the need to trade-off between these two important criteria in data compression. In this method, VHDL is selected as the hardware description language to realize the data compression system. In the computation of method, the problem is first divided into small pieces; each can be seen as a sub- module in VHDL [4, 11, 12]. Following the software verification of each sub-module, the synthesis is then activated that performs the translations of hardware description language code into an equivalent netlist of digital cells. The synthesis helps integrate the design work and provides a higher feasibility to explore a far wider range of architectural alternative [10]. The method provides a systematic approach for hardware realization, facilitating the rapid prototyping of data compression system.

2. Design Methodology

For effective data compression, the compression algorithm must be able to predict future data accurately in order to build a good probabilistic model for compression [7]. The PPM compression scheme would operate on binary data as computer-based data is represented and transmitted using binary digits. The PPM involves two steps, the generation of the adaptive model and the compression/decompression using arithmetic coding [1]. The generation of the adaptive model uses Markov modeling to build a probabilistic distribution of binary digits. The Markov predictor of an order j predicts the next bit based on the j preceding bits. Adaptive coding allows the model to be constructed dynamically by both encoder and decoder during the course of the transmission, and has been shown to incur a smaller coding overhead than explicit transmission of the model's statistics [1]. A block diagram of a complete data compression system is shown in Figure 1.

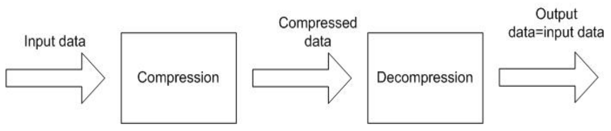


Figure 1: A basic compression system

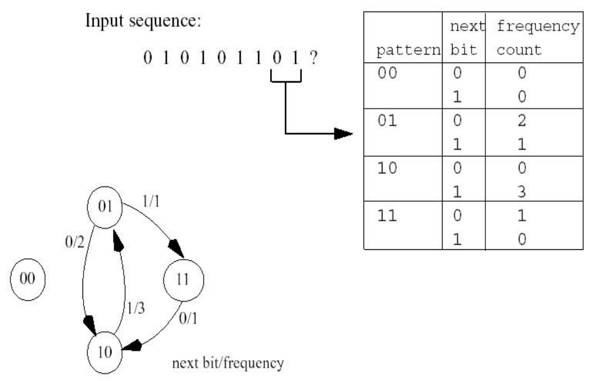
Data that has been compressed need to be decompressed to return it to its original form. Therefore a decompressor comes hand-in-hand with a compressor. Compression is dependent upon the fact that data is redundant and that its generation was based on a fixed set of rules. If those rules are known, we can accurately predict the data. The data compression can be viewed as a branch of information theory whose primary objective is to minimize the sum of data to be transmitted [13].

3. PPM Implementation

As mentioned earlier, PPM involves two steps, the generation of an adaptive model (predictor stage) and the compression or coding stage [1]. Both the encoder and decoder of a PPM system, adapts the model of coding dynamically to the message statistics as the transmission proceeds [14]. Adaptive coding is effective because what is happening is counted and used as the basis of subsequent coding so that the counts only needs to be incremented in the event that a character is correctly predicted [2]. It should also be noted that text statistics in reality are not homogeneous and so well behaved, thus requiring a need for the adaptive coding method [1]. In PPM, the adaptive model is generated using a Markov predictor.

The bases of the PPM algorithm of order m are set of (m + 1) Markov predictors. A Markov predictor of order j predicts the next bit based upon the j immediately preceding bits. It is just a very simple Markov chain. There will be 2j possible patterns if there is j number of bits. The transition frequency is built by the predictor in proportion to the observed frequencies of a '1' or a '0' that occur, given that the predictor has seen the bit pattern associated with that state. The predictor builds the transition frequency just by recording the number of times a '1' or a '0' occurs in the (j + 1)th bit following the preceding j bits. A Markov chain is built at the same time that it is used for prediction and often the chain is incomplete. Figure 2 shows an example of the 4-state Markov chain. Let there be an input sequence of '010101101' bits and the order of the Markov predictor is to be 2. The next bit is to be predicted based on the two immediately preceding bits of '01'. From observation, it can be noted that the pattern '01' occurs three times throughout the current input sequence. The frequency counts of the bit following '01' are as such: '0' follows '01' twice and '1' follows '01' once. Therefore, the predictor should predict the next bit to be '0' with a probability of 2/3 [15].

A 0th order Markov predictor simply predicts the next bit based on the relative frequency in the input sequence. For simplicity, the 0th order Markov predictor is adopted for this project and it assumed that each bit encountered by the Markov predictor is novel. Figure 3 shows the flowchart for a Markov predictor.



pattern	next bit	frequency count
00	0	0
	1	0
01	0	2
	1	1
10	0	0
	1	3
11	0	1
	1	0

Figure 2: The (incomplete) 4-state Markov chain

4. Arithmetic Encoder and Decoder

Arithmetic coding is one approach to generate variable length codes and is one of the best algorithms that can be used in lossless data compression. For the case of PPM, where the modeling and coding stages of the lossless data compression have to be kept separate, arithmetic coding is a particularly well-suited method to adopt.

Arithmetic coding replaces a sequence of symbols with a coding range of real numbers between 0 and 1. The range accorded to a symbol will depend on the probability of that particular symbol, the higher the probability, the higher the range, which assigns to it. The gist of arithmetic coding is the generation of a tag from that range for a sequence encoded. The tag is a floating-point value that corresponds to a binary fraction. Eventually this binary fraction will become the binary code for the sequence. In practice, the generation of the tag and the binary code are one in the same process. The arithmetic coding is divided into two phases. The first phase generates a unique identifier or tag for a given sequence of symbols and the second phase gives a unique binary code to the tag [16].

The arithmetic encoder is used to generate a tag. The tag generated by the arithmetic encoder has to fall within the probability line of 0 to 1. The symbols are given a range based on their probability. The higher the probability, the higher is the range that is given to it. Once the ranges and the probability line are defined, the symbols are then encoded where each symbol defines where the output floating point number lands [17].

The algorithm for arithmetic encoding is:

```
    Low=0
    High=1
Loop for all symbols.
    Range = High – Low
    High = Low + Range * Q(x)
    Low = Low + Range * Q(x – 1)
where,
    Q(x) = high range of symbol being encoded
    Q(x- 1) = low range of symbol being encoded
```

The tag can actually be any real number between 0.47265625 and 0.578125. For the purpose of this project, the tag is taken as the value of the low range, 0.47265625. The tag is useless if it cannot be deciphered. The decoder is used to recover the original data.

The algorithm for decoding is:

```
Loop for all symbols.
    Range = Q(x) – Q(x – 1)
    Tag = Tag – Q(x – 1)
    Tag = Tag / Range
Where,
    Q(x)=high range of symbol being decoded
    Q(x-1)=low range of symbol being decoded
```

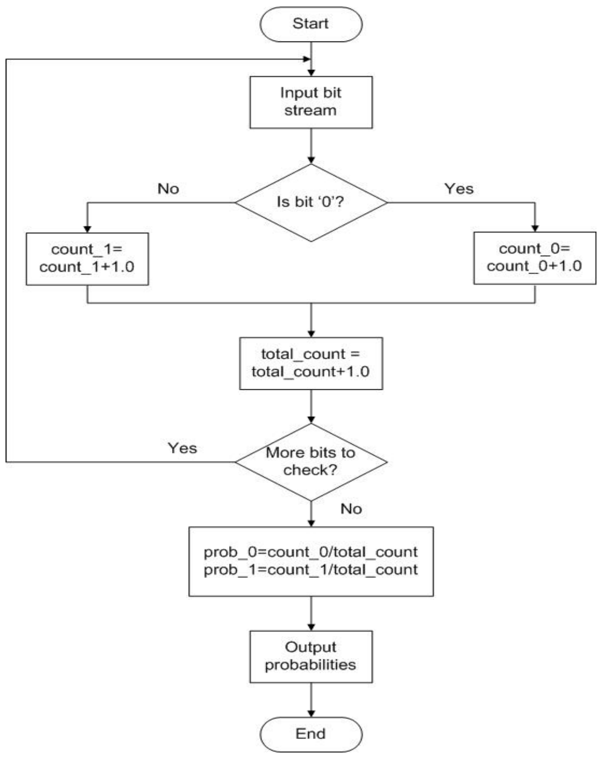


Figure 3: Flowchart of the Markov predictor

5. Software Implementation

The whole design is described using IEEE-compliant VHDL language. Optimization to the VHDL code was performed to reach an even higher speed. As described earlier, the PPM module carries out probability distribution using Markov predictor followed by arithmetic encoding and decoding. A bottom-up approach is adopted here, whereby the PPM module is split into sub-modules with each sub-module being coded, verified and validated separately before being combined to produce the final design. The three sub-modules are the Markov predictor, arithmetic encoder and arithmetic decoder. However, for the design of this project, the Markov predictor would be combined with the arithmetic encoder and referred to as the Markov model. Thus, the total number of sub-modules is two. Each distinct module performs a specific function as stated below:

Markov Model: The Markov predictor builds the probability distribution from the input symbols seen and the arithmetic encoder generates a unique identifier or tag based on the probability distribution of symbols. Figure 4 shows the block diagram of a Markov model.

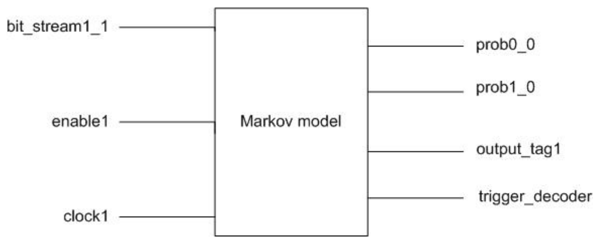


Figure 4: Block diagram of Markov model

The input to the Markov model sub-module is bit_stream1_1, where a string of binary bits would be read in accordance with the clock cycle, clock1. The control signal enable1 is used to select between the Markov model sub-module and the arithmetic decoder sub-module. The probability of '1' and '0' occurring will then be calculated and the probability of both will be outputted separately in real values (floating-point values) through two output ports, prob0_0 and prob1_0. The values of these probabilities are universal to all sub-modules. The Markov predictor needs to calculate all probabilities before the arithmetic encoder can start encoding.

After the probabilities have completely been calculated and the values have been outputted, the arithmetic encoding would be carried out to produce the tag in real value. The string of bits is then encoded based on the probability according to the clock cycle, clock1. The outcome is a series of ranges in real values signifying the high and low range of each bit occurring at the input. For these ranges of highs and lows, the tag is generated and outputted at output_tag1. After the tag has successfully been generated, the sub-module would output a high bit to port trigger_decoder for the decoder to start decoding.

Arithmetic Decoder: The arithmetic decoder takes the tag generated by the encoder, decodes it and output the results as a string of bits. The output string of bits for the decoder should be similar to the input string of bits for the Markov model. This sub-module consists of four input ports, one clock signal, one control signal and one output port. Figure 5 shows a block diagram of an arithmetic decoder.

The input to the arithmetic decoder is the tag in real value together with the probability of '1' and '0' (prob0_1 and prob1_1) also in real values and the trigger signal, decode. The tag is then decoded with reference to clock2. Once again, the control signal enable2 is used to select between the two sub-modules. The output is a string of bits, output_bits similar to that of the input string of bits of the Markov model.

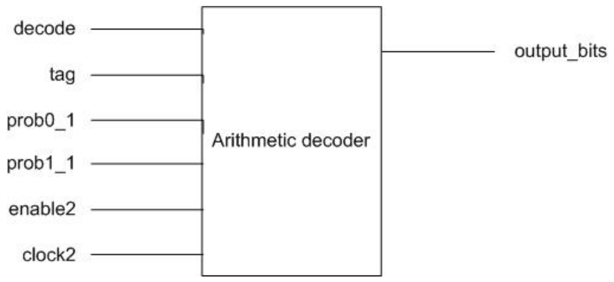


Figure 5: Block diagram of arithmetic decoder

3. PPM Model: The PPM main module consists of the two sub-modules; Markov model and arithmetic decoder. It does not have any input ports corresponding to clock signals or control signals but has one output port. The output is a value that needs to be checked to ensure correct operation of the module. The output is a string of bits, output_bit. Figure 6 below shows the complete top-level view of a PPM module.

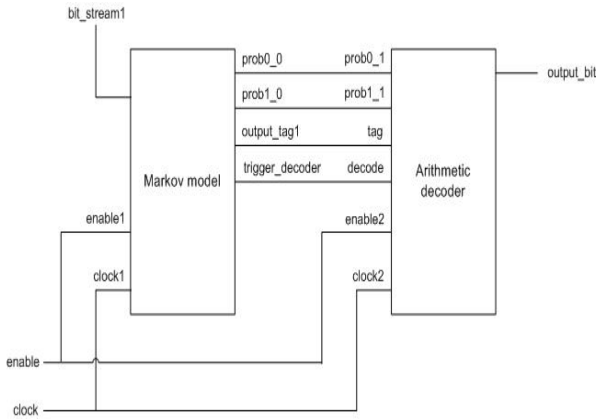


Figure 6: Top-level view of PPM model

6. Hardware Implementation

Hardware implementation is the unique abstract of this work, it is not sufficient by only performing software simulations. The physical hardware layout is generated using the synthesis tool Quartus II version 5.0. The compilation process is repeated with different synthesis options in order to properly trade-of between area and speed. The project was successfully configured and downloaded to the FLEX10K EPF10K30BC356-3 FPGA, tested and validated. The FLEX10K family provides the density, speed, and features to integrate entire systems, including multiple 32-bit buses into a single chip. The implementation results are shown in the next section.

7. Experimental Study and Discussions of Results

7.1 Simulation Results

An extensive experimental investigation was conducted in order to demonstrate the efficiency and feasibility of the proposed method. Functional simulation is performed to test the logic function of the hardware design and it is presented to verify the correctness of all the modules involved. The results of classification used different 16-bit inputs and later expanded to using a few different 32-bit inputs.

7.2 16-bit Input

The test vectors have been predetermined before the simulation was carried out. Therefore the first 16-bit input to the PPM module to test its functionality is “1101011011011000” in binary or “D6D8” in hexadecimal. The simulation results are shown in Fig7, using the following input/output:

bit_stream1=16-bit input
s01=probability of ‘0’
s02=probability of ‘1’
s03=output tag
output_bit=16-bit output

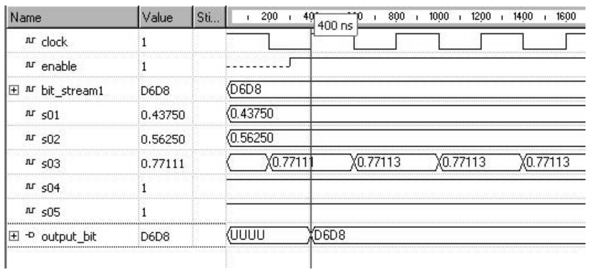
At the start of the clock, the probability of ‘0’ is 0.43750 and the probability of ‘1’ is 0.56250. From observation, the number of occurrences of ‘0’ in the 16-bit input stream is 7 and the number of occurrences of ‘1’ is 9. The total number of bits is 16. Thus,

P(‘0’) = 7/16 = 0.43750

P(‘1’) = 9/16 = 0.56250

Since both simulated and computed results are similar, it can be concluded that the Markov predictor portion of the PPM module is functioning correctly.

At time 200ns, the tag value is 0.77111 as shown in Figure 7. The subsequent values that appear at s03 are due to the fact that the operation of the arithmetic encoding runs based on the clock event. Therefore, since the clock continues to run there will still be values at signal s03. The only way to prevent values from appearing at s03 after the first value of the tag has been outputted is to find a way to stop the clock from running after that. At time 400ns, the 16-bit output “D6D8” is available at the port output_bit. By comparing, the initial 16-bit input with the current 16-bit output, it can be observed that both of them are similar. Therefore, it can be concluded that the arithmetic encoding portion of the PPM module is functioning correctly.

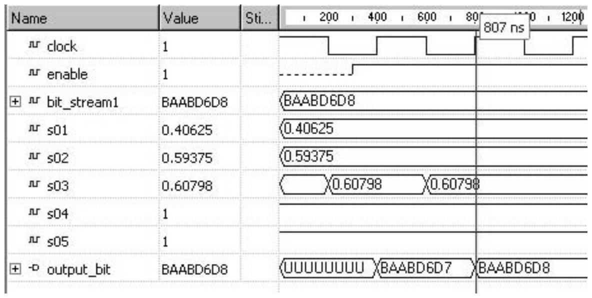


Name	Value
clock	1
enable	1
bit_stream1	D6D8
s01	0.43750
s02	0.56250
s03	0.77111
s04	1
s05	1
output_bit	D6D8

Figure 7: Simulation results for16-bit PPM module

7.3 32-bit Input

The PPM module is expanded to accept 32-bit inputs. The first 32-bit input to the PPM module to test its functionality is “10111010101010111101011011011000” in binary or “BAABD6D8” in hexadecimal. The simulation result is shown in Fig8.



Name	Value
clock	1
enable	1
bit_stream1	BAABD6D8
s01	0.40625
s02	0.59375
s03	0.60798
s04	1
s05	1
output_bit	BAABD6D8

Figure 8: Simulation results for 32-bit PPM module

At the start of the clock, the probability of ‘0’ is 0.40625 (13/32) and the probability of ‘1’ is 0.59375 (19/32). The values are correct since ‘0’ has a count of 13 and ‘1’ has a count of 19. At 200ns, the tag is outputted as 0.60798. Finally at time 800ns, the 32-bit output appearing on the output ports is “BAABD6D8. Comparing that with the 32-bit input shows that they are similar. Therefore, the PPM module functions properly with a 32-bit input.

8. Synthesis Results

A comparatively low critical path frequency was achieved which was 25.1 MHz. The design took a minimum resource i.e. 1164 logic cells, which is 67.36% of the EPF10K30BC356-3 device. A maximum frequency of 95.3MHz was achieved. Table 1 shows a detailed report of the usage of resources.

Table 1: The Usage Of Resources

Logic Resources (EPF10K30BC356-3)	Logic resources: 1164 LCs of 1728 (67.36%)
	Number of Nets: 386
	Number of Inputs: 1066
	I/O cells: 57
	Cells in logic mode: 312
	Cells in cascade mode: 45

Conclusions

In this research project, the FPGA prototyping of data compression using partial matching algorithm that allows for efficient hardware implementation had been implemented. It was successfully simulated and generated acceptable results for a 16-bit input as well as for a 32-bit input. The modules were successfully compiled and simulated. The hardware implementation demonstrated complete, correct functionality and met all the initial system requirements. The performance of the hardware prototype is encouraging. The results reveal that the proposed approach is computationally simple, accurate and exhibits a good balance of flexibility, speed, size and design cycle time. Comparison and results presented validate the successful compression of data using partial matching.

References

[1] Cleary, J. G. and Witten, I. H., "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions on Communications, vol. COM-32, April, 1984, pp. 396-402.

[2] Moffat, A., "Implementing the PPM Data Compression Scheme", IEEE Transactions on Communications, vol. COM-38, November, 1990, pp.1917-1921.

[3] Bloom, C., http://www.cbloom.com/src/ppmz.html (15th July 2003)

[4] Shkarin, D., "PPM: One step to practicality", Proceedings of Data Compression Conference, April 2002, pp. 202-211

[5] Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory, IT-23, 337.343.

[6] Ziv, J. and Lempel, A. (1978) Compression of individual sequences via variable rate coding. IEEE Trans. Inform. Theory, IT-24, 530.536.

[7] Bell, T. C., Cleary, J. G. and Witten, I. H., "Text Compression", Englewood Cliffs, NJ, Prentice Hall, 1990, 318 pages

[8] C. E. Cummings, "Verilog Simulation Xilinx Designs," Proc. Int. Verilog HDL Conf., Santa Clara, CA, 1994, pp. 93-100.

[9] A. Rushton, VHDL for Logic Synthesis, Wiley, New York, 1998.

[10] B. K. Fawcett, "Tools to Speed FPGA Development," IEEE Spectrum, November 1994, vol. 31, pp. 88-94.

[11] Alexandre Schmid, Yusuf Leblebici, and Daniel Mlynek, "Hardware Realization of A Hamming Neural Network with On-Chip Learning," IEEE International Symposium on Circuits and Systems, Monterrey CA, 1998, vol. III, pp. 191-194.

[12] Peter J. Ashenden. 1996. The Designer's Guide to VHDL. Morgan Kaufmann Publishers Inc., San Francisco.

[13] Hirschberg, D. http://www.ics.uci.edu/~dan/pubsDataCompression, November 2003

[14] Langdon, G. G and Rissanen, J., "Compression of Black-White Images, with Arithmetic Coding", IEEE Transactions on Communications, vol. COM-29, June 1981, pp. 858-867.

[15] Mudge, T. N., Chen, I-Cheng K. and Coffey, J. T., "Limits to Branch Prediction", 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 128-137.

[16] Sayood, K., "Introduction to Data Compression", 2nd Edition, San Francisco, Morgan Kaufmann Publishers, 1996, 636 pages.

[17] Campos, A., http://www.arturocampos.com/, November 2003