IEICE TRANS. FUNDAMENTALS, VOL.E84–A, NO.4 APRIL 2001

PAPER

A Pipeline Chip for Quasi Arithmetic Coding

Yair WISEMAN[†a)], ***Nonmember***

SUMMARY A combination of a software and a systolic hardware implementation for the Quasi Arithmetic compression algorithm is presented. The hardware is implemented as a pipeline hardware implementation. The implementation doesn't change the the algorithm. It just split it into two parts. The combination of parallel software and pipeline hardware can give very fast compression without decline of the compression efficiency.

key words: *pipeline structure, compression, arithmetic coding, hardware-software combination*

1. Introduction

In order to increase the rate of computers, many manufacturers use the pipeline method [1]–[3]. This method enables performing several actions in a machine in parallel mode. Every action is in a different phase of its performing. The action is divided into some fundamental sub-actions which can be performed in one clock cycle. In every clock cycle, from every action, the machine will perform a new sub-action. A pipeline machine can perform different sub-actions in parallel. In every clock cycle, the machine performs sub-actions for different actions.

Nowadays, most of the CPU manufacturing is pipelining. For example, all Intel's Chips from 486 and on, are pipeline chips. Many articles describe improvements in pipeline machines for example [4]. In addition to this progress, there is a progress in manufacturing dedicated pipeline chips. For example, see in [5], [6] implementation of a Boltzmann Machine in a dedicated pipeline chip. Moreover, there is a progress in manufacturing dedicated chips for data compression or decompression [7].

This article introduces a combined implementation of hardware and software of the adaptive compression algorithm - Quasi Arithmetic Coding [8]–[10]. Quasi Arithmetic Coding is a version of the well-known Arithmetic compression algorithm [11], [12]. The use of Quasi Arithmetic Coding in embedded systems is suggested in [13], but the the compression there is done in software. This article suggests assigning part of the compression to a dedicated systolic hardware. The part of the hardware is developed as a pipeline hardware. The software part also can be implemented in parallel mode. It can be done as we suggested in [14] for decoding, or as was suggested in [15], or a combination of both of them.

There have been a lot of studies to reduce the implementation complexity of Arithmetic Coding. This complexity damaged the time performance of Arithmetic Coding. Some of those studies can be found in [16]–[19]. Quasi Arithmetic coding is one of the methods which approximate Arithmetic Coding. by using state tables. Such tables are very common in the chip designing field and can be easily adapted to a hardware implementation, which can obviously be much faster. So we achieve not only an improvement in time by replacing Arithmetic Coding by Quasi Arithmetic Coding, but even more so by implementing the tables of the Quasi Arithmetic Coding in hardware.

2. Quasi Arithmetic Coding

In [20] Shannon proved that an optimal coding encodes every event with a probability P, by $-\log_2 P$ bits. Huffman Coding rounds the sizes of the codewords to an integer number of bits. Hence Huffman Coding [21] is not optimal, unless $-\log_2 P$ is an integer number for every probability P for any codeword in the Alphabet. Such a distribution is called a Dyadic Distribution [22] and is not realistic in most cases. Arithmetic Coding tries to fix this disadvantage but can be complex and slow. Quasi Arithmetic Coding is a simpler and faster version of Arithmetic Coding.

Let $\mathcal{L}$ be a set of codewords. In Arithmetic Coding, every codeword $i \in \mathcal{L}$, gets a probability $P_i \in [0,1]$ when $\sum_{i\in\mathcal{L}} P_i = 1$. Every codeword is represented by the interval $[\sum_{j<i} P_j, \sum_{j\le i} P_j)$. Similarly, in Quasi Arithmetic Coding, the following steps are done:

- Constructing a code for the codewords "1" and "0." The construction of this code is explained below.
- Constructing a prefix coding for the codewords which need to be compressed. The well known prefix coding—Huffman [21] can be used but any other prefix coding is welcome. The probability of any codeword in this coding is actually the product

Manuscript received March 14, 2000.
Manuscript revised September 21, 2000.

[†]The author is with the Department of Math and Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel and with the Department of Computer Engineering, the Jerusalem College of Technology, P.O.B. 16031, Jerusalem 91160, Israel.
a) E-mail: wiseman@cs.biu.ac.il

of the probabilities of all the codewords' bits. Since every bit value is a binary value, the probability is known from the previous step.
- Writing the algorithm which implemented the prefix coding that was decided. This Algorithm creates the prefix code from the previous step, for every codeword it gets. Since this code contains just bits i.e. binary values, they can be compressed by the coding that was built in the first step.

Constructing Quasi Arithmetic Coding is done in an adaptive way. We save for each stage, the approximate ratio between the bits we read and their value was 0, to the bits we read and their value was 1. The ratio is saved as counters pair-counter for the "1"s and counter for the "0"s. The ratio is saved in an approximate way. The ratio is rounded to $\frac{i}{j}$ where $i, j \in [1, N]$ and N is an integer number. The counters are initialized to the value 1, although no bit was read. The reason is because a zero value can cause a division by zero problem [23]. If one of the counters exceeds the value N, we will try to find another approximate ratio which is near to the original ratio. In fact we defined N^2 states since the value of i and j, will be always between 1 to N. Let us denote these states as P states.

In addition we save the interval which represents the current state. In the original Arithmetic Coding the interval is always in the format $[i, j)$ with $0 \leq i \leq j \leq 1$. In Quasi Arithmetic Coding, the interval is in the format $[k, l)$ with $0 \leq k \leq l \leq M$ and k, l, M are integer numbers. In fact we defined $\frac{(M+1)M}{2}$ states. If l is the right edge of the interval, we can define $k < l$ as the left edge of the interval. Therefore we can have $\frac{(M+1)M}{2}$ states which is the sum of the sequence $1 \ldots M$. Let us denote these states as Q states. From all of these states $\frac{3M^2}{16}$ states are terminal states. When the state is a non-terminal state, this interval can be broaden by the known scaling algorithm of Arithmetic Coding. Sometimes this scaling algorithm also produces some output. When the state is a terminal state, the interval can't be broaden and there is no output.

The moving between P states is clear. According to the input which is always "0" or "1," we will increase the value of the appropriate counter. If we exceed the value N, we will try to find a new ratio that will be close to the required ratio. The state that will be defined with the two new counters, will be the new state.

The moving between Q states is a little bit complicated. According to the current Q, the interval size is computed. According to the new P state, we can know the ratio between the counters after reading the current bit. If the current bit is "0," we will look on the ratio as $\frac{Count_0}{Count_1}$. If the current bit is "1," we will look on the ratio as $\frac{Count_1}{Count_0}$. According to the interval size and ratio between the counters, the new interval size is set. If the current bit is "0," the left part of the interval is taken. If the current bit is "1," the right part of the interval is taken. The new interval can be a non-terminal interval. If the new interval is a non-terminal interval, the interval will be broaden until the interval is becoming a terminal interval. This broadening is done according to the scaling algorithm of Arithmetic Coding.

3. Parallel Quasi Arithmetic Coding

In [15] there is an algorithm for parallel Arithmetic Coding, done entirely in software. This Article will introduce a new approach. The algorithm will be implemented in a combination of software and hardware. Also the algorithm of this article is for Quasi Arithmetic Coding and not for Arithmetic Coding. The two major parts of the implementation are:

- The software will deal with building the prefix coding. There are some prefix codes which can be used. For example Huffman code [21], Golomb code [24], Elias code [25] or Rice code [26]. This part will not be introduced here since there are several ways to do it. One of them has been introduced by us in [14] for decoding. Also it can be done in another approach as introduced in [15]. One of these approaches (or a combination between them) can be taken for the software assignment.
- The hardware will deal with Quasi Arithmetic Coding for the binary values. After the software outputs its code, the hardware will take this stream of binary values and compress them according to the Quasi Arithmetic Coding algorithm. The prefix coding doesn't create an optimal coding as was mentioned above. This step will fix this non-optimal result.

Firstly we will build the hardware for handling the P and the Q states. This hardware is shown in Fig. 1. First of all into the register "New bit," a new bit is assigned. The register size is one bit. According to this bit, the value of the previous P state and according to Next_p() function, the new P state is calculated. This calculation is written into the register P. This calculation is done during one clock cycle. At the end of this clock cycle the calculated value is written into the register P.

After this clock cycle, the Q state is calculated. According to the previous Q state, the size of the previous interval is known. Using the previous interval size and new P state, the new interval size is calculated. This calculation is done in the function defined in Compute_length(). If the input is "0," we can know the left edge of the interval. It will be the same as it was in the previous interval. If the input is "1," we can know the right edge of the interval. The previous state Q gives us the the left and the right edges. One of them is copied to the new interval. Now when we know the size of the new interval and one of its edges, we can find the new interval itself. This calculation is

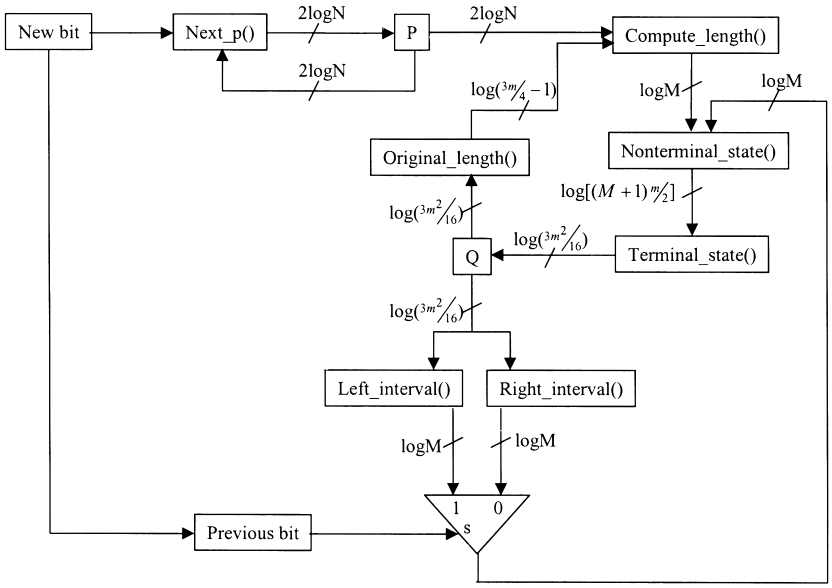

Fig. 1 Hardware for handling P and Q states.

done by the Nonterminal_state() function. The Terminal_state() function also determines the values of all the lines go out to the hardware which handles the memory. The lines which go out to the memory hardware are:

- The "Follow bits" count. (According to the scaling algorithm of Arithmetic Coding).
- The count of the bits written into memory. (Not including one leading bit).
- The bits that should be written into memory. (They are split to one leading bit and the other bits).
- Write enable for the memory.

As was mentioned, the function Nonterminal_state() is calculating a non-terminal state that can be broaden. This broadening is done by the scaling algorithm of Arithmetic Coding. In this hardware this broadening is done by the Terminal_state() function. In fact the functions Nonterminal_state() and Terminal_state(), can be implemented in one function. The split here is done just for making the illustration clearer. In a real implementation they should be implemented by one function.

The number of the lines in the hardware is varying according to the values of N and M. All the values in Fig. 1 will be ceiling values if there is a need in rounding. The number of the lines are always $\log_2$ of the possible values that should be represented in the lines.

The function Original_length() will output the size of the terminal interval. If we want to know how many values there are for the output, we should find the number of the possible sizes that can be for a terminal interval. Let us denote an interval as $[i, j)$. If this interval can be scaled, one of these conditions must be satisfied:

- $[i, j) \subseteq [0, \frac{1}{2}]$
- $[i, j) \subseteq [\frac{1}{4}, \frac{3}{4}]$
- $[i, j) \subseteq [\frac{1}{2}, 1]$

Any interval with size not larger than $\frac{M}{4}$, regardless of its edges, can be scaled. If M is divisible by 4, we can say that every interval with size not larger than $\frac{M}{4}+1$ can be scaled. We will generally use M such that $M = 2^n$ and $n > 2$, so the number of the possible sizes of the terminal intervals is $\frac{3M}{4}-1$

In Fig. 2 we can see a first attempt of implementation of the memory handling. The reading from the memory is simple. The read address register saves the value of the current address. In every clock cycle, the register's value is increased by one.

The writing to the memory is a little bit complicated. There is a need of writing two different values to the memory. Each one of these values' sizes is unknown. The first value is the leading bit and after it the following bits (if any at all). The second value is the other bits, if there are more bits for writing. Because of this reason, the first solution will be doubling the clock cycle time for the hardware which handles the P and the Q states. The clock cycle time of this hardware will be twice of the clock cycle time of the hardware which handles the memory. While the hardware which handles the P and the Q states is computing next states, the hardware which handles the memory will have two cycles. In the first clock cycle, the leading bit with the follow bits will be written. In the second clock cycle, the remaining bits will be written. Calculating the next Q state needs passing of three functions. If the number of lines is small, a truth table can be written and a circuit which implements sum of products can be done. If the number of lines is bigger, we can not implement

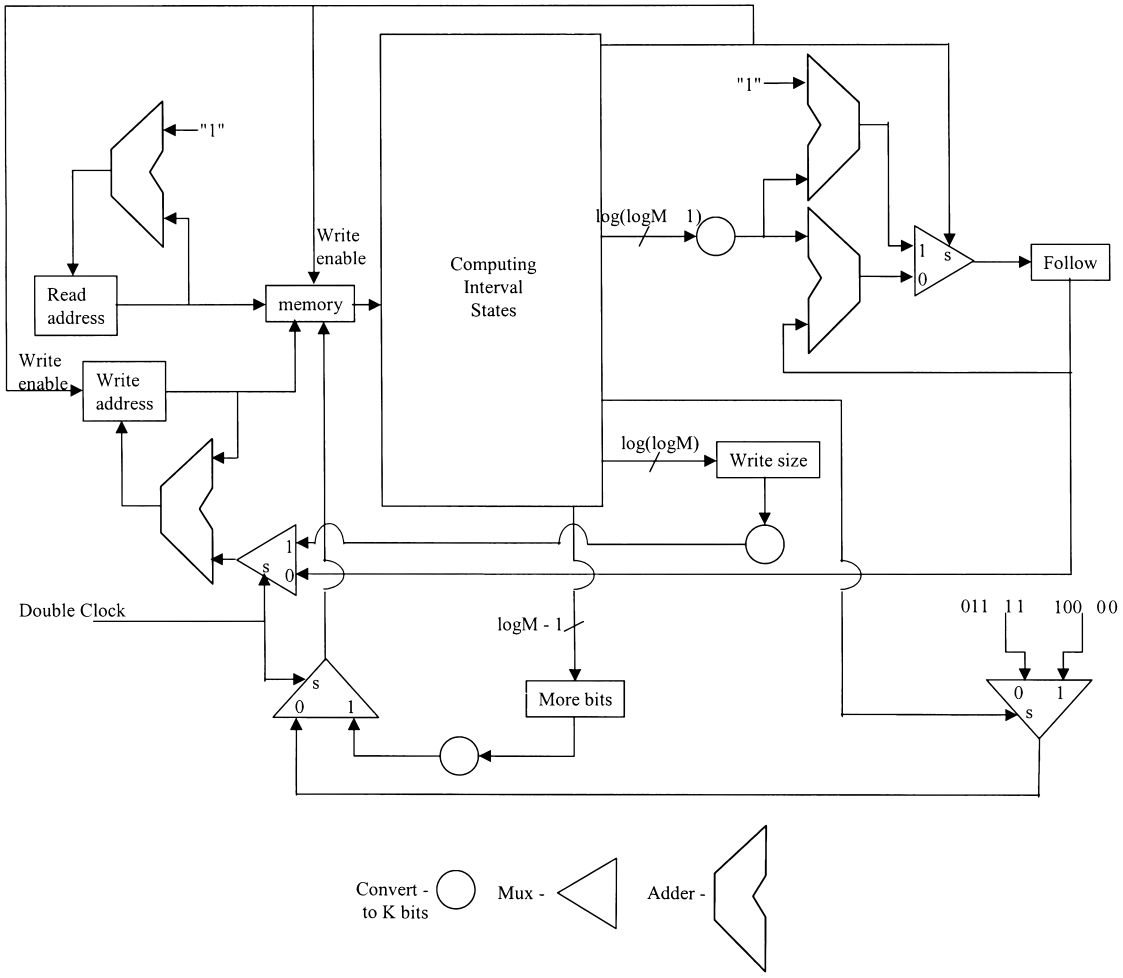

Fig. 2 Hardware for handling the memory.

this by sum of products, due to *Fan-In* of the gates [27]. We may need a *microcode* implementation [28], [29]. In such a case three functions in one cycle can be problematic. Doubling the clock cycle period can be reasonable in such a case.

In Fig. 2 we can see that the number of lines for the follow bits count is $\log(\log M - 1)$. Calculating the maximum count of follow bits that can be generated because of reading one bit is done in this way: We suppose M is even. Since M is even, an interval with size one, can not be scaled with follow bits. All of the interval will be in the right side of the interval or in the left size of the interval. It can't be in both sides of the middle of the interval $[0, M)$. The scaling for such an interval will be without follow bits. The minimal interval that can be scaled with follow bits is $[\frac{M}{2}-1, \frac{M}{2}+1)$. This interval will be scaled with follow bits $\log(\frac{M}{2})$ times. The number of lines that can introduce it is $\log(\log M - 1)$.

The maximum number of broadening for any interval is $\log M$. Since one bit is written as leading bit, the maximum number of remaining bits is $\log M - 1$. The number of lines required for introducing this value is $\log(\log M)$. We need $\log M$ and not $\log M - 1$, since the value zero is also meaningful here.

A rough estimation of hardware size (gates count) can be done by calculating the number of gates for each function in hardware. A formal expression can be derived, but in order to give a feeling of how many gates will be needed, we assume that N=4 and M=8. An explanation of how to generalize the results will be given too.

The hardware which handles the P and Q states has some functions which can be implemented in a simple truth table. On average, half of the entries in those truth tables are "1"s and the other half are "0"s.

- Next_p() has 5 input lines and 4 output lines. It has 32 options of inputs. In each line of output, about 16 input combinations will give true. In order to implement this function, the circuit should have about 64 NOT gates (4*16), 64 AND gates (4*16) and 4 OR gates which are 132 gates. The OR gates may have Fan-In problem, so a few more OR gates may be needed.

- Compute_length() has 9 input lines and 3 output lines. Similarly, the circuit will have about 771 gates.
- Nonterminal_state() has 6 input lines and 6 output lines, giving about 198 gates.
- Terminal_state() has 6 input lines and 4 output lines, giving about 132 gates.
- Original_length() has 4 input lines and 3 output lines, giving about 27 gates.
- Each of the functions Right_interval() and Left_interval() has 4 input lines and 3 output lines, giving about 27 gates.

The sum of the number of gates in these circuits is 1314.

There is also a need for flip-flops for registers in both hardwares (The hardware in Fig. 1 and the hardware in Fig. 2):

- New bit—1 flip-flop.
- Previous bit—1 flip-flop.
- P—4 flip-flops.
- Q—4 flip-flops.
- Write size—2 flip-flops.
- More bits—2 flip-flops.
- Follow—depends on how much follow bits are allowed to be.
- Read address, Write address—depends on what is the last address in memory.

The sum of the number of flip-flops in these circuits is a few dozens.

4 multiplexor and 4 adders will be needed too.

The estimation for the number of gates in hardware is some hundreds of gates. A larger N and M will give a circa linear increase in number of gates, since the number of lines which go into functions, is roughly logarithmic in N, M and the number of gates is roughly exponential in the number of lines which go into functions.

Figure 3 describes the progress of the machine by showing the pipeline chart. The names of the clock cycles are:

- New is the clock cycle when a new bit is read from memory into the New_bit flip-flop.
- P is the clock cycle when P state is computed.
- Q is the clock cycle when Q state is computed.
- F is the clock cycle when the new follow bits are computed and the leading bit with the old follow bits are written into memory.
- R is the clock cycle when the remaining bits are written into memory.

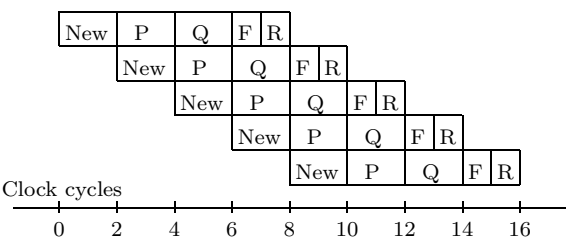

Fig. 3 Pipeline chart.

As was mentioned the clock cycle period of New, P and Q is twice of the clock cycle period of F and R. In Fig. 3 we can see the progress of this machine as function of time.

4. Improving Performance by Canceling the Clock Time Doubling

Doubling the clock time period causes a decline in performance. We shall try finding another solution to the problems that were solved by doubling the clock cycle period.

The first problem was that the computing time of Q was too long. Such a duration can decline the clock cycle rate. The second problem was that we had to write into memory two different values in variable lengths. The length was not known so we didn't know how much increase should be done to the address pointed by Write address register. The values were first the leading bit and the follow bits that should be written after it and second the remaining bits.

In order to solve the first problem of computing Q, we should split the computing into two stages. These stages will be done in two separated clock cycles. We should add a new register that will save the intermediate value.

In order to solve the second problem of writing two different values in variable sizes, we should add a multiplexor. The possible numbers of the follow bits are higher than the possible numbers of the remaining bits. Hence the remaining bits of the previous step will be joined to the follow bits of the current step. The multiplexor will select how many bits can be omitted from the remaining bits. In fact the *select* lines of the multiplexor will have exactly the same value as the lines that determine the number of bits which should be written in memory.

The writing of the remaining bits will be delayed by one cycle in order to enable the hardware to compute the follow bits of the next clock cycle. This delay can be done by connecting registers that will delay the writing of the remaining bits in one clock cycle. There will be two registers for this purpose. One register for delaying the adding of the number of bits to the address pointed by Write address register. One more register to delay the writing of the remaining bits into memory.

In Fig. 4 the new hardware is shown.

In Fig. 5 the hardware for computing P and Q states is shown. This hardware is very similar to the hardware in Fig. 1. The only difference is that in this

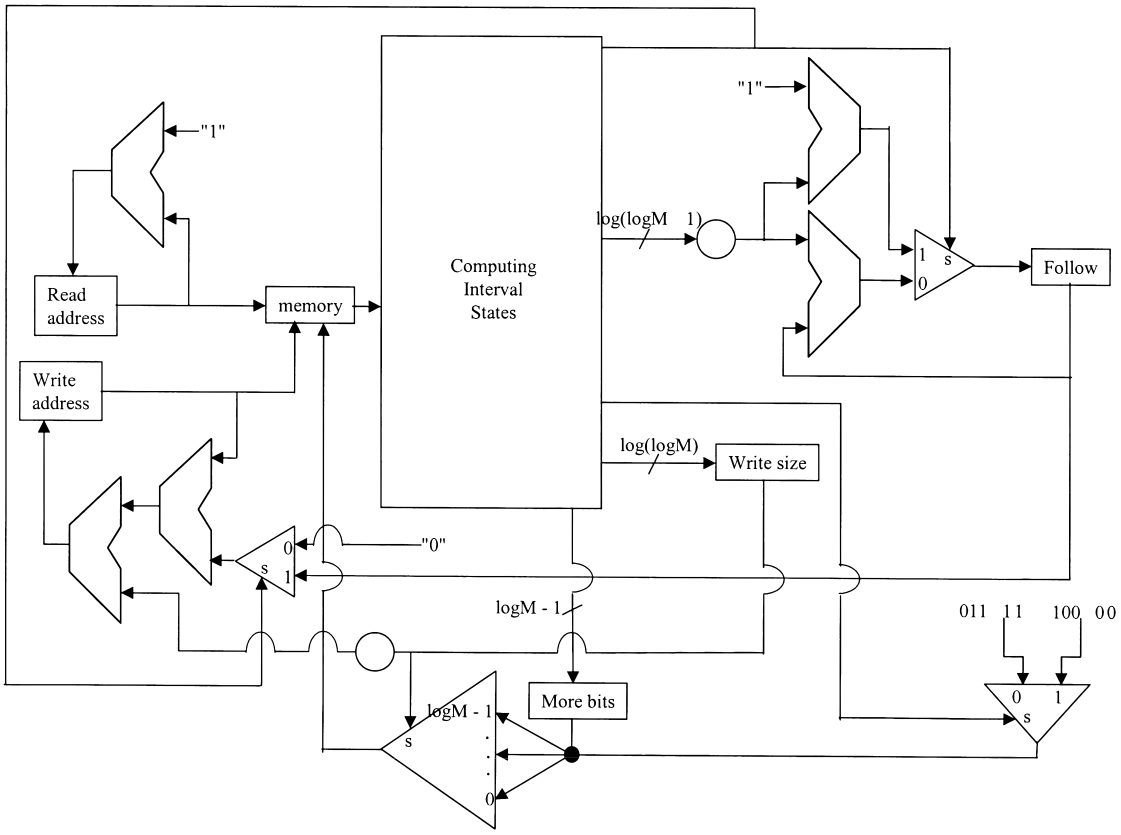

Fig. 4 New hardware for handling the memory.

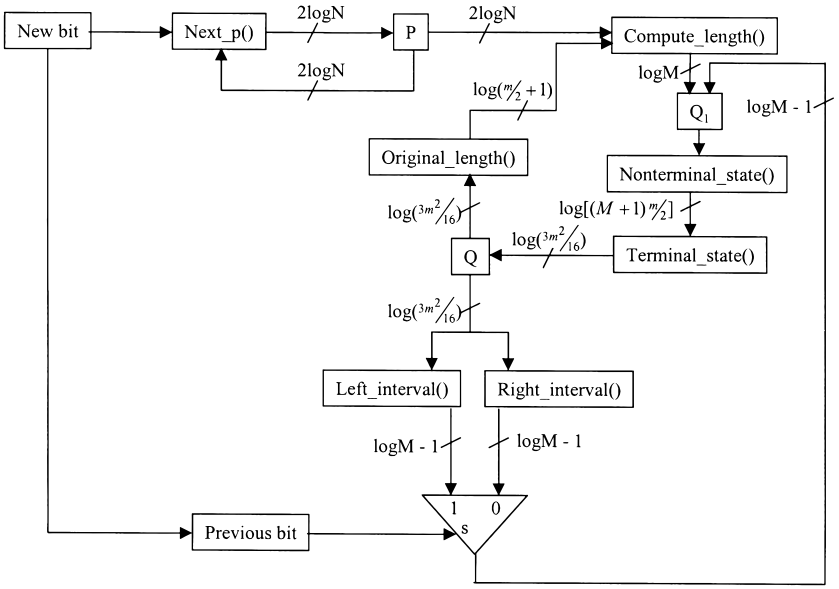

Fig. 5 New hardware for handling P and Q states.

hardware there is a new register Q_1, which saves the intermediate value of computing Q states.

Now we should give a new name to the new clock cycle that was added. The new clock cycle in which Q_1 is computed will be denoted Q_1. The new pipeline chart will be as described in Fig. 6.

The estimation for the number of gates in this implementation will be very close to that of the previous implementation, because of the similarity in circuits. However, the time savings are significant. The longest

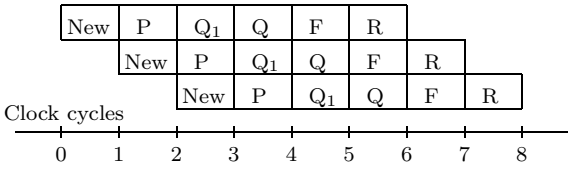

Fig. 6 Pipeline chart.

operation which has to be completed in one cycle is the memory access. According to [30], memory access time is decreasing over years, but the decrease is not linear in the number of years. The change in access time is getting smaller every year, but still access time itself is getting shorter every year. Every decrease can be helpful in reducing the clock cycle, so more bits can be compressed in a shorter time period. Nowadays, according to [31], such an access can take about 25–110 ns. In this implementation every bit will be compressed in one cycle, instead of two cycles. There is always an addition of five clock cycles to the total amount of cycles, in order to wait for the last bit, but if the amount of data to be compressed is large enough, these five cycles can be neglected. This means that if the caching system works well, some Megabits can be compressed in one second. An improvement is expected in coming years when memory access time will be reduced.

5. Conclusion

A systolic hardware for an implementation of a Quasi Arithmetic compression for binary values has been proposed. The implementation succeeds to compress $\mathcal{K}$ bits in $\mathcal{K}+5$ clock cycles. Such a chip should be connected to a very high speed computer which can do the software assignment in an appropriate time.

Acknowledgment

The author wants to thank Prof. Shmuel T. Klein from the Math and Computer Science department in Bar-Ilan University for his help.

References

[1] N.P. Jouppi and D.W. Wall, "Available instruction level parallelism for superscalar and superpipelined machines," Proc. Third Conf. On Architectural Support for Programming Languages and Operation System IEEE/ACM Boston, pp.82–272, 1989.

[2] P.M. Kogge, The Architecture of Pipelined Computers, McGraw-Hill, New-York, 1981.

[3] D.A. Patterson and J.L.Hennessy, Computer Organization and Design, pp.434–536, Morgan Kaufmann Publishers, INC. San Francisco, California, USA., 1997.

[4] W.K. Cheng and Y.L. Lin, "Code generation of nested loops for DSP processors with heterogeneous registers and structural pipelining," ACM Trans. Design Automation of Electronic Systems, pp.231–256, 1999.

[5] H. Zhu, M. Sasaki, and T. Inoue, "A pipeline structure for the sequential Boltzmann machine," IEICE Trans. Fundamentals, vol.E82-A, no.6, pp.920–926, June 1999.

[6] H. Zhu and M. Sasaki, "Introducing a parallel-transit-evaluation method to the sequential Boltzmann machine," IEICE Trans. Fundamentals, vol.E79-A, no.9, pp.1571–1581, Sept. 1996.

[7] D.M. Royals, T. Markas, N. Kanapoulos, J.H. Reif, and J.A. Storer, "On the design and implementation of a lossless data compression and decompression chip," IEEE J. Solid-State Circuits, vol.28, no.9, pp.953–984, 1993.

[8] P.G. Howard, "The design and analysis of efficient lossless data compression systems," Ph.D. Thesis, pp.7–34, Department of Computer Science, Brown University, Providence, Rhode Island, USA. 1993.

[9] P.G. Howard and J.S. Vitter, Practical Implementations of Arithmetic Coding Image and text compression, J.A. Storer ed., pp.85–112, Kluwer Academic Publishers, Norwell, MA, 1992.

[10] P.G. Howard and J.S. Vitter, "Design and analysis of fast text compression based on quasi-arithmetic coding," Proc. Data Compression Conference Snowbird Utah pp.98–107, 1993.

[11] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," Commun. ACM, vol.30, pp.520–540, 1987.

[12] P.G. Howard and J.S. Vitter, "Analysis of arithmetic coding for data compression," Information Processing and Management, vol.28, no.6, pp.749–763, 1992.

[13] H. Lekatsas, J. Henkel, and W. Wolf, "Arithmetic coding for low power embedded system design," Proc. Data Compression Conference Snowbird Utah, pp.430–439, 2000.

[14] S.T. Klein and Y. Wiseman, "Parallel Huffman decoding," Proc. Data Compression Conference Snowbird Utah, pp.383–392, 2000.

[15] P.G. Howard and J.S. Vitter, "Parallel lossless image compression using Huffman and arithmetic coding," Proc. Data Compression Conference Snowbird Utah, pp.299–308, 1992.

[16] R. Pasco, "Source coding algorithm for fast data compression," Ph.D. Thesis, Department of Computer Science, Stanford University, 1976.

[17] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon, and R.B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," IBM J. Res. Develop., vol.32, no.6, pp.717–726, 1988.

[18] J. Rissanen and K.M. Mohiuddin, "A multiplication-free multialphabet arithmetic code," IEEE Trans. Commun., vol.COM-37, no.2, pp.93–98, 1989.

[19] L. Huynh and A. Moffat, "A probability-ratio approach to approximate binary arithmetic coding," IEEE Trans. Inf. Theory, vol.IT-43, no.5, pp.1658–1662, 1997.

[20] C.E. Shannon, "A mathematical theory of communication," Bell System Tech. J., vol.27, pp.398–403, 1948.

[21] D.A. Huffman, "A method for the construction of minimum redundancy codes," Proc. Institute of Radio Engineers 40, pp.1098–1101, 1952.

[22] K.R. Stromberg, Probability for Analysts, Chapman and Hall Probability Series, pp.121–124, Chapman and Hall, NY., 1994.

[23] I.H. Witten and T.C. Bell, "The zero frequency problem; Estimating the probabilities of novel events in adaptive text compression," IEEE Trans. Inf. Theory, vol.IT-37, no.4, pp.1085–1094, 1991.

[24] S.W. Golomb, "Run length encoding," IEEE Trans. Inf.

Theory, vol.IT-12, no.4, pp.399–401, 1966.
[25] P. Elias, "Universal codeword sets and representations of integers," IEEE Trans. Inf. Theory, vol.IT-21, no.2, pp.194–203, 1975.
[26] R.F. Rice, Some Practical Universal Noiseless Coding Techniques, Jet Propulsion Laboratory, Passadena California, JPL Publication, pp.22–79, 1979.
[27] L.S. Bobrow, Fundamental of Electrical Engineering, pp.454–456, Oxford University Press., 1996.
[28] M.V. Wilkes and J.B. Stringer, "Microprogramming and the design of the control circuits in an electronic digital computer," Proc. Cambridge Philosophical Society, vol.49, pp.230–238, 1953.
[29] S.S. Husson, Microprogramming Principles and Practice, Prentice Hall, Engelwood Cliffs NJ, 1970.
[30] D.A. Patterson and J.L. Hennessy, Computer Organization and Design, Chapter 7, Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1997.
[31] UTMC Standard Products, http://www.utmc.com/products/radhard_mem.html, 2000.

Yair Wiseman received a B.Sc. in Mathematic and Computer Science in 1991 from Bar-Ilan University. Received a Summa Cum Laude M.Sc. in Computer Science in 1995 from Bar-Ilan University. He was an officer in the main computer unit of Israel Defense Force. He was a chief of staff of the Institute for Information Retrieval and Computational Linguistics in Bar-Ilan University. He is now about to receive a Ph.D. degree from Bar-Ilan University. His research interest is parallel compression in software and hardware. He is now with the Math and Computer Science Department in Bar-Ilan University and with the Computer Engineering Department in the Jerusalem College of Technology.