A Q-CODER ALGORITHM WITH CARRY FREE ADDITION

Gianluca Cena Paolo Montuschi Luigi Ciminiera Andrea Sanna
Dipartimento di Automatica e Informatica,
Politecnico di Torino, corso Duca degli Abruzzi 24,
10129 Torino (Italy)

Abstract

The Q-Coder algorithm is a very efficient compression technique for bi-level images based on the arithmetic coding. This paper presents a new and fast version of the Q-Coder algorithm in which the carry-propagated adders have been replaced by carry-save adders. In this way, all the additions can be performed with a delay time of a single full adder, independently of the length of the operands. Our compression method is faster than the traditional Q-Coder algorithm with an almost unnoticeable increasing of the hardware requirements.

1 Introduction

Quite a lot of techniques are known in the literature for the information coding; among them, an important role is played by the arithmetic coding. Arithmetic coding is theoretically able to achieve code messages with minimum redundancy without any loss of information; moreover, it often allows better compression ratios than well-known algorithms such as Huffman coding [7]. The main features of arithmetic coding are its generality (it has not been designed bearing in mind a specific application or source model) and its suitability for adaptive implementations (it takes advantage of the changes in the statistics of the symbols to be encoded).

The original version of the general arithmetic compression method was presented by Rissanen and Langdon in [14], who introduced also a version for black and white (bi-level) images [9]. Langdon in [10] presented a survey on the arithmetic coding algorithms. Some implementation issues for applying arithmetic coding to data organized as a stream of bytes were presented in [17] and more recently in [8].

This class of arithmetic compression methods, however, is not well suited for real-time encoding and decoding, because they require a re-normalization of the intervals, and hence a multiplication for each symbol encoded or de-coded. In [11], Mitchell and Pennebaker substituted the multiplications with simpler additions. Variations of this method have been adopted for the final entropy encoder in JPEG [15] (in alternative to Huffman coding) and for bi-level images encoding (JBIG or ITU T.82) [16].

The encoder of this method, which is the most relevant for the purposes of this paper, may be thought as composed by three blocks:

- a model block that, on the basis of the value of the surrounding (binary) pels, selects the probability estimator (or context) to be used;
- an adapter block which, on the basis of the context and the pel, decides the new symbol probability, and the new state for the estimator itself;
- the coder, which performs the actual encoding, by using the probability selected by the adapter and the pel.

This paper focuses on the last of the above listed blocks; the goal is to substitute the carry-propagate addition used in previous algorithms with a carry-free one. In other words, the addition performed by the coder block will be implemented by using a carry-save adder, which produces a redundant but carry-free result; then, a group of the most significant digits are inspected, in order to make a decision about the re-normalization of the intermediate quantities. This inspection is equivalent to the assimilation of the most significant redundant digits of the result. While using this scheme, each addition requires an execution time shorter than for carry-propagate adders and independent of the length of the operands.

It will also be shown that it is possible to decode the string produced by our encoder by using the same type of decoder used in [11] with minor modifications to the blocks performing the arithmetic operations.

1063-6889/97 $10.00 © 1997 IEEE

An evaluation of the compression capabilities of the encoder proposed is included which is based on the CCITT test documents and a comparison is made with the results obtained with the Q-coder. We will show that the proposed architecture is up to 25% faster than that based on the traditional Q-Coder algorithm and that the compression ratios obtained decrease by less than 2%.

The content of the paper is the following: arithmetic coding and the traditional Q-Coder algorithm are briefly reviewed in section 2, together with the definitions and symbols used in the paper. In section 3 the proposed methodology is presented, whereas the architecture for a carry-free Q-Coder is shown in section 4. Finally, performance comparisons between the carry-free Q-Coder algorithm and the conventional approach are reported in section 5.

2 Arithmetic coding

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1 which covers the whole range of probabilities for the input data. The initial range is the entire half-open interval $[0, 1)$; each time a symbol is encoded, the range is narrowed to a portion of it, according to the cumulative probability interval of the encoded symbol. The more likely symbols reduce the range less than the unlikely symbols and therefore they add fewer bits to the encoded message. The output code is the string of bits used to represent a value in the final coding range; the lower end-point value is usually chosen. Practical implementations of arithmetic coding can be found in [8] and [17]. In this paper we address the problem of the binary arithmetic coding which is slightly different from the general problem of the arithmetic coding, since just two symbols are involved in the alphabet.

The idea beyond the binary arithmetic coders (BAC) [10], [11] is to encode a sequence of binary symbols called Most Probable Symbol (MPS) and Less Probable Symbol (LPS) into a code string representing a fractional binary value. Such a value is obtained by successively partitioning a base interval in two subintervals, one related to the LPS symbol and the other to the MPS symbol, and then selecting the interval corresponding to the input binary symbol read. At each step, the interval is described by means of its base value and its size. The size of the interval is stored in a fixed length register A. By means of a normalization operation the representation of A is kept within a specified range. The base value of the interval (representing the encoded code string) is instead stored part by some unspecified external means (transmission channels or storage devices) and part (the less significant bits) in a fixed size register called C on which the actual arithmetic operations are performed.

```
C register:

zzzzzzzz, bbbbbbbb, ssss.xxxx, xxxxxxxx
^                   ^          ^
msb                 fractional point    lsb
```

Figure 1. Sample structure of the C register.

2.1 Definitions and symbols

In the Q-Coder algorithm (QCA) the role of the A register is related to the emission of encoded symbols: in fact, code symbols are output in correspondence with normalizations of A. The role of C is to store the less significant bits of the current code string, which is partially emitted when A is normalized. An example of the format of C [11] is provided in Fig. 1. The updating of the code string mainly concerns the bits labeled with x in Fig. 1. Bits b are instead used as a repository for a new byte of encoded data to be read out. Related to the bit stuffing technique [9] used to avoid carry propagation, a certain number of spacer bits has to be introduced in the format of the C register (bits s in Fig. 1). The bit stuffing technique will be analyzed in detail in section 3.3. Finally, z bits are used in the Q-Coder as a pad to a 32 bit register and are not involved in the arithmetic operations. In Fig. 1 the position of the fractional point is also shown. Let us denote with $A_i[j]$ and $C_i[j]$ the values of A and C respectively after j bits have been read in and encoded and i normalization shifts have been done (that is, after i bits have been written out of the encoder). Note that the values of i and j are in general uncorrelated and that no biunivocal relation exists between them. In fact, several input bits can be encoded without any bit is written out (the emission of any bit by the encoder is related to the normalization operation) while some other input bits (such as for example the LPSs) can cause the emission of more than one output bit. Moreover, let us denote with $\widehat{A}$ the estimate of A truncated to the t-th fractional bit and with s the number of spacer bits which have to be introduced in the representation of C to have the bit stuffing mechanism operating properly (see section 3.3).

We also denote with p the number of fractional bits in the representation of A, with u the number of shifts necessary for the normalization test (see sections 3.2),

and with Q_e the probability of the LPS at current step. Since we adopt the same probability estimation table used in [11], the probability is expressed on a 16 bit number in the range $0 < Q_e \leq 2753/4096$ (i.e., the representation varies in the range X'0.000'< $repres(Q_e)$ ≤X'0.AC1').

2.2 Analysis of the Q-Coder algorithm

The encoding operations of the Q-coder algorithm can be summarized as follows:

1. • if a MPS is encoded the addition $C_i[j+1] \leftarrow C_i[j] + Q_e$ and the subtraction $A_i[j+1] \leftarrow A_i[j] - Q_e$ have to be performed;
   • if a LPS is encoded C and A have to be updated as follows: $C_i[j+1] \leftarrow C_i[j]$ and $A_i[j+1] \leftarrow Q_e$;
2. normalization of $A_i[j+1]$: if $A_i[j+1] < K$ (where K is a predefined constant on a finite and small number of bits) then: $A_{i+1}[j+1] \leftarrow 2 * A_i[j+1]$ and $C_{i+1}[j+1] \leftarrow 2 * C_i[j+1]$. Every 8 normalization shifts (or 7 immediately after a stuff bit) the 8 (or 7) most significant bits of C are removed and a byte of encoded data is emitted from the Q-Coder. In this case, $C[j+1] \leftarrow C[j+1] - y_l \cdot 2^t$, where y_l is the value of the byte just emitted. If a normalization occurs, then Q_e is updated according to a predefined scheme.

Step (2) is repeated until $A[j+1] \geq K$. In [11] the value $K = 1$ was selected.

3 Basic idea

One of the bottlenecks of the Q-Coder algorithm is that the additions and subtractions are carry-propagated. An alternative strategy for the representation of the results can be considered in order to avoid the limitation of having long carry-propagated operations. The main idea is to represent the results in the *redundant* carry-sum form [3], [13], where the values involved are represented as pairs (carry and sum bits), whose sum equals the value of the result and where an addition has the delay of just one full adder.

3.1 Carry free Q-Coder algorithm

In the carry free Q-Coder algorithm (CFQCA), carry free adders have been used instead of the carry-propagated adders for carrying out the updating of $C[j]$ and $A[j]$, and this has mainly two advantages:

1. the updating of the contents of A and C is performed with carry-save adders which are faster than the carry-propagated ones;
2. the delay for updating A and C does not depend on the number of bits of A, C and Q_e. This implies that Q_e can be represented using a larger number of bits, without affecting the performance.

However, the use of a carry-sum representation for the $A[j]$ values leads to the introduction of uncertainty, which has to be reduced before the normalization test, since the decisions affect the compression capabilities of the algorithm. An estimate $\widehat{A}$ of A (starting from its most significant bits) is considered and the normalization test is based on the value of this estimate. This operation has costs both in terms of delay and hardware requirements (which depend on the bits number of the estimate), therefore a tradeoff between the number of bits required for the estimate and the performances of the hardware unit has to be found, as described in section 5.

The operating steps of the proposed CFQCA are:

1. • if MPS is encoded the carry save addition $C_i[j+1] \leftarrow C_i[j] + Q_e$ and the carry save subtraction $A_i[j+1] \leftarrow A_i[j] - Q_e$ have to be performed;
   • if LPS is encoded C and A have to be updated: $C_i[j+1] \leftarrow C_i[j]$ and $A_i[j+1] \leftarrow Q_e$;
2. an estimate $\widehat{A}$ of $A_i[j+1]$, on all the integer and t fractional bits has to be considered;
3. $\widehat{A}$ is compared with K, to determine the amount of the normalization:
   • if $\widehat{A} \geq K$ then no normalization is necessary;
   • if $\widehat{A} < K$ the amount u of normalization shifts has to be determined; then: $A_{i+u}[j+1] \leftarrow 2^u * A_i[j+1]$ and $C_{i+u}[j+1] \leftarrow 2^u * C_i[j+1]$. As in the QCA, every 8 normalization shifts (or 7 if a bit stuff has been inserted), the 8 (or 7) most significant bits of C are removed and one byte of encoded data is emitted. Finally, since a normalization occurred, the value Q_e is updated according to the same schemes used from the QCA.

As in the QCA, we set $K = 1$ for the normalization test. Since $Q_e < 1$ this implies that $A > 0$. Moreover, since a normalization may occur when $\widehat{A} \leq 1 - 2^{-t}$ and since $(A - \widehat{A}) < 2^{-(t-1)}$, it follows that the shifted (renormalized) A is

$$sh(A) < 2(1 - 2^{-t} + 2^{-(t-1)}) = 2 + 2^{-t+1} \quad (1)$$

Therefore, 2 integer bits are necessary and sufficient for the integer part of A. Since the estimate $\widehat{A}$ is obtained from truncation of A, the representation of $\widehat{A}$ too requires 2 integer bits (plus t fractional).

3.2 Remarks

The use of carry save adders for updating both A and C leads to carry free results. This implies that the normalization test has to be redefined in terms of the estimate $\widehat{A}$, otherwise, all the advantages of having a carry save adder would be lost. From its definition, $\widehat{A}_i[j+1] \leq A_i[j+1] \leq \widehat{A}_i[j+1] + 2^{-(t-1)} - 2^{-(p-1)}$, where p is the number of fractional bits of $A_i[j+1]$; therefore, the test $\widehat{A}_i[j+1] < K = 1$ does not guarantee that also $A_i[j+1] < 1$. This implies that in some cases the CFQCA could operate a normalization not activated from the QCA. This has two main effects:

1. since the normalization implies the emission of code bits and the "new tuning" of the compression parameters, it is important that the encoder and decoder both take the same decisions in correspondence of same values of $A_i[j+1]$ (i.e. they must be designed with similar architectures) since a carry assimilated decoder would not be able to decompress codewords encoded by a carry save encoder;
2. the "uncertainty" due to the term $2^{-(t-1)}$ could worsen and decrease the compression capabilities of the CFQCA with respect to the QCA, because a normalization could be carried out (together with the emission of some bits) even in cases when it is not strictly necessary.

The first issue is not a problem, because it can be solved by providing the same carry save mechanism for updating $A_i[j]$ in the encoder and in the decoder. The negative effects of the second issue can be addressed (or at least limited) by choosing a proper value for t, (related to the length of the estimate). However, as we will see in section 5, the CFQCA achieve almost the same compression ratios as the QCA.

Theorem 1 *The minimum value of t which is necessary for detecting (through the test on $\widehat{A}$) up to a maximum of u_{MAX} sufficient shifts (at one time) for obtaining a normalized A, is $t_{min} = u_{MAX} - 1$.*

Proof. Let us examine the estimate $\widehat{A}$ of A on t bits. By looking at $\widehat{A}$ we can detect the number u of shifts sufficient for the normalization of A by simply performing the test $2^{-u} \leq \widehat{A}_i < 2^{-(u-1)}$ with $u \leq t$. In fact from $\widehat{A}_i \leq A_i$ and $A_{i+u} = 2^u A_i$ we get $\widehat{A}_{i+u} = estim(2^u A_i) \geq 2^u \widehat{A}_i$; therefore, $\widehat{A}_{i+u} \geq 1$ which is the definition of normalization. With a similar analysis it can be seen that if $\widehat{A}_i < 2^{-t}$ then it follows $\widehat{A}_{i+t} < 1$ and hence at least $u = t + 1$ normalization shifts are necessary. □

Having fixed u_{MAX}, to consider values of t larger than t_{min}, may have the positive effect of further reducing the uncertainty between $\widehat{A}$ and A, and hence obtaining better compression factors, i.e. closer to the ones obtained with the QCA.

Observe, however, that Theorem 1 deals with a number of sufficient shifts, in order to obtain a normalized A. The actual number of necessary shifts should be obtained by performing a sequence of step-by-step normalizations (i.e. using single shifts & normalization tests), until the normalization is achieved. Since each normalization corresponds to the emission of one code bit, it is clear that it would be better to normalize only when it is strictly necessary. This statement has been checked with some simulations, which have confirmed that the step-by-step normalizations (i.e. with $u_{MAX} = 1$) lead to better compression ratios.

Concerning the value of t, it must be chosen in order to obtain a convenient tradeoff between the compression requirements, the execution speed and the hardware requirements, as explained; in section 5.3. As it will be shown, even for small values of t, the CFQCA achieves compressions sufficiently close to the QCA.

It is worth to observe that the proposed CFQCA is equal to the QCA in the part of the prediction hardware and that they differ only in the way A and C are updated and how the "normalizations" are carried out.

3.3 Bit stuffing

As in the QCA, also in the CFQCA a carry propagation may occur in an already emitted part of the code string. In the QCA, there is a carry propagation if in the part of the code string already emitted there is a sequence of consecutive 1 bits. This is surely a problem, because we want to use fixed length adders and, more important, part of the code string already generated could be no more available to the encoder. Since the above cited sequence can be arbitrarily long (theoretically up to the length of the whole code string), a technique known as bit stuffing has been introduced to limit carry propagation [9] [6].

According to this technique, sequences of 1 bits are periodically split by the insertion of 0 stuff bits. In particular the QCA forecasts that a 0 bit is stuffed in the high order bit of the byte immediately following an emitted byte whose value is X'FF', such to act as a repository for a possible subsequently generated carry.

This technique is proven to work correctly if only one stuff bit is sufficient to contain all the carries that can be generated by the algorithm from its insertion on.

In their paper [11] Mitchell and Pennebaker demonstrated that, by introducing 4 spacer bits in the structure of C, any byte containing a stuff bit cannot be followed by another byte with a stuff bit; this guarantees that no more than one stuff bit every 15 code bits is necessary and that any byte pattern in the range from X'FF90' to X'FFFF' is not legal in the code string output.

The bit stuffing technique of the QCA can be applied also in the CFQCA. In a very similar way to that explained in [11] it is possible to demonstrate that:

Theorem 2 *In the CFQCA no more than 3 spacer bits are necessary for the correct operation of the bit stuffing mechanism. If at least one more fractional bit beyond the minimum is considered in estimating the value of A (that is, $t \geq t_{min}+1$) then only two spacer bits are needed.*

Proof. For the representation we have chosen for the redundant register A with, $1 \leq A < 2(1+2^{-t})$. Let us consider the condition in which a byte of coded data has just been removed. Let C_i be the value of C at that time. C_i is such to satisfy the relation $C_i < 2^s + 1$.

After 8 normalization shifts (i.e. when a new byte is ready to be extracted) the value of $C_{i+8} + A_{i+8}$ is bounded by $C_{i+8} + A_{i+8} \leq 2^8(C_i + A_i)$ thus, since $A_{i+8} > 0$ we have $C_{i+8} < 2^8[2^s + 1 + 2(1+2^{-t})]$.

The worst case is when no fractional bit is considered in estimating A, that is, when $t = t_{min} = 0$. In this case the condition $C_{i+8} < 2^8(2^s + 5)$ holds.

To have no more than one carry bit we must impose $C_{i+8} < 2^{9+s}$, so the minimum number of spacer bits necessary that satisfies the expression $2^{9+s} \geq 2^8(2^s+5)$ is $s = 3$. In a similar way it is possible to demonstrate that if $t \geq t_{min} + 1$ the minimum number of spacer bits needed is $s = 2$. □

By using 4 spacer bits it is possible to show that in the CFQCA any two byte pattern in the range X'FFA8' to X'FFFF' is illegal in the code string output, thus these codes can be used as escape characters [11]. This confirms that the bit stuffing technique can be used unmodified also for the CFQCA.

In a real implementation of the CFQCA there is no reason to use more spacer bits than the minimum necessary, since this only increases the hardware complexity and does not lead to any advantages in terms of the compression ratio.

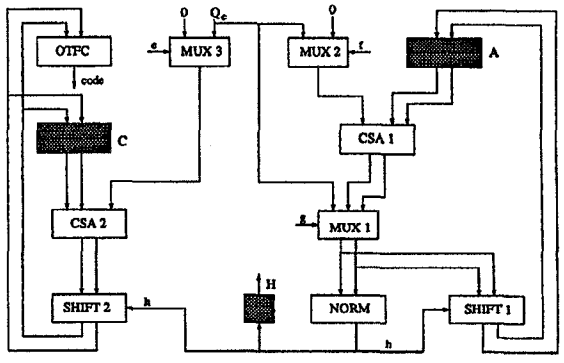

Figure 2. Proposed CFQCA encoder.

4 The proposed encoder

4.1 Proposed implementation

The architecture of the proposed CFQCA encoder (in its parts for the updating and test of A and for the updating of C) is depicted in Fig. 2, where the following blocks can be observed:

- blocks C and A: pairs of registers to store the carry save representations of C and A;
- blocks MUX: multiplexers;
- blocks CSA: carry save adder (or subtractor) to perform carry save the updating of C (or A);
- block NORM: hardware to consider the estimate $\widehat{A}$ in order to determine the amount of shifts required from the normalization, i.e., 0 or 1. This implies that, for the implementation of block NORM we have chosen $u_{MAX} = 1$ and $t_{min} = 0$. We have demonstrated (see [1]) that if $Q_e < \frac{3}{4}$, the normalization required in correspondence of a MPS can imply either 1 or 2 shifts. However, our simulations have shown that the normalizations in correspondence of MPS requiring 2 shifts occur very seldom. Therefore, in order not to increase the complexity of an architecture we have chosen to consider a maximum of one normalization shift per cycle;
- blocks SHIFT: shifter of 0 or 1 positions;
- block H: register, one bit long;
- block OTFC: hardware to convert on-the-fly into the conventional form the bytes of C as they are emitted; in addition, hardware for a correct management of the emission (alignment and bit stuffing);

- signal h: control signal carrying the information on the number of detected shifts (either 0 or 1);
- signals e, f, g: control signals for the MUXes.

Let us focus our attention on the part concerning the updating and normalizing of A, which is found in the rightmost part of Fig. 2. The operating principle is very simple: when a MPS is encountered, A is updated through CSA1 with the subtraction of Q_e (arriving from MUX2). The updated value of A (passing through MUX1) arrives to the block NORM, which produces in output the control signal h concerning the number of shifts required for the normalization (i.e., 0 or 1). Then, by means of the shifter and the control signal h, the correct A is produced and stored into the A register. During the cycle following a normalization, no symbol in input is accepted, but just the test for possible further normalizations is performed. Only when a normalized A has been obtained, the architecture is able to accept other symbols to be compressed.

When a LPS is encountered, the architecture must deal with the following events:

- the A register has to be loaded with Q_e (and not with $A - Q_e$);
- since the number of shifts which are necessary for the normalization of A could be larger than the maximum detectable from NORM and allowed from the blocks SHIFT (i.e., 1), in the proposed architecture of Fig. 2, the normalization of A could take more cycles. Observe that during these "extra" normalization cycles, the proposed CFQCA is not able to accept any symbol to encode until the normalization has been completed.

The extra cycles are accomplished in the following way: the block NORM produces in output the control signal h which is set if the number of detected shifts is 1, and which is stored into register H for the driving of the multiplexers during the next cycle. In such a case, during the current cycle, a shift of 1 position is obtained and loaded in A; then, during the next cycle the contents of A are not updated by presenting A and a 0 value (through MUX2) in input to CSA1. At this point, the unchanged A is enabled to pass also through MUX1 and another inspection by NORM is performed. This inspection again produces the signal h and the operations continue as previously explained. In order to obtain for the whole process to operate correctly, it is necessary for the control signal h to be forwarded to the circuitry for updating C, so that also C can be normalized correctly.

The architecture of the part for updating C is similar (see the leftmost part of Fig. 2); observe that the block NORM is not necessary, because C is shifted according to the values imposed by block NORM operating on A. The correctness of the operations in the case of normalization requiring "extra" cycles, is ensured from MUX3 which enables the presentation of the value 0 to CSA2, thus avoiding the modification of C (as well as for A during these "extra" iterations). It is worth to observe that the block OTFC is required so as to on-the-fly convert into the conventional form [2] the bytes of C as they are emitted. Moreover, the block OTFC incorporates also the same hardware for a correct management of the byte emission and of the bit stuffing mechanism of the QCA. The analysis of alternative implementations can be found in [1]. The simulations (as reported in section 5) have shown that it is suggested to choose $t \geq 2$ in order to have satisfactory compression ratios. In the following we present the implementation of NORM and the correspondent evaluation for $t = 2$.

4.2 Design of block NORM

In the proposed implementation we must be able to detect and perform single shifts; the block NORM receives in input the estimate $\widehat{A}$ on 2 integer and $t = 2$ fractional bits. The block NORM must operate the generation of the control signal h according to the following rules:

$$\begin{aligned} h &= 1 \quad if \quad \widehat{A} \geq 1 \\ h &= 0 \quad if \quad \widehat{A} < 1 \end{aligned} \tag{2}$$

From equation (1), we have that $A < 2 + 2^{-(t-1)} = 2 + 2^{-1}$, and hence the range of $\widehat{A}$ is $0 \leq \widehat{A} \leq 2 + 2^{-2}$.

It is worth to observe that the block NORM plays a similar role as the selection table in several digit-by-digit architectures [4].

4.3 CFQCA: the decoder

In order to perform a correct decoding, it is necessary for the decoder to operate the same choices of normalization carried out at the encoder level. This implies that the part for updating A depicted in Fig. 2 for the encoder is also used in the decoder (since the same decisions must be taken). On the other hand, the part on C cannot be carry save (as in the encoder), since the decisions on the decoding of the symbols must be taken on precise and full length comparisons of the code string (in C) and the probability Q_e. Therefore, the same hardware for the part concerning with the updating of C is used both for the CFQCA and for the QCA.

5 Evaluation and comparisons

5.1 Hardware requirements

The proposed encoder requires: two pairs of registers to store the carry-sum representations of C and A; one 1-bit register (H); one carry save adder and one carry save subtractor (CSA1 and CSA2, respectively); one block NORM; two shifters on 0 or 1 positions (SHIFT1, SHIFT2); one multiplexer (MUX1); two sets of AND gates (to implement MUX2 and MUX3).

5.2 Execution time in full-adder delays

The critical path of the proposed architecture passes through the updating and normalization of A. Let us denote with D_{PA} the duration of one cycle of the proposed architecture, with d_x the delay of block "x", and with d_{reg} the delay for register loading. The global cycle duration is:

$$D_{PA} = d_{MUX2} + d_{CSA1} + d_{MUX1} + d_{NORM} + \\ + d_{SHIFT1} + d_{reg} \quad (3)$$

On the other hand, the architecture based on the QCA, would have a similar layout as this depicted in Fig. 2, with the difference in the blocks CSA which must be substituted with Carry-Propagated Adders (CPA) and with a different block for testing the normalization (NORMAL). The cycle delay D_{NR} of the architecture based on the QCA, would therefore be:

$$D_{NR} = d_{MUX2} + d_{CPA1} + d_{MUX1} + d_{NORMAL} + \\ + d_{SHIFT1} + d_{reg} \quad (4)$$

For our comparisons we will consider the CPA as performing an addition on 12 bits [11].

To evaluate the execution time of the proposed architecture, we define the execution time in terms of full-adder delays (d_{FA}). This unit is chosen mainly because the main components are formed of full adders and because it is relatively simple to express the delay of other components using this unit. Although this makes the evaluation and comparisons relatively independent of the technology, the results are rough because they depend on the accuracy of the assumptions made. The evaluations are also rough since data routing delays are not included and no technology-dependent optimization is performed. In the cases in which there is no direct correspondence with the delay of a full adder, we have used the correspondence for a particular technology. In order to determine the delay of the components, assumptions have to be made about their implementation; we have used assumptions that seem reasonable, but some alternatives would also be possible. Specifically, we utilize the family of standard cells from the ES2-ECPD10 library [5]; moreover, for the delay of a carry-propagated adder we assume a logarithmic delay [12]. The delays of the components measured in this unit are as follows [4]: $d_{driver} = d_{FA}$, $d_{MUX} = 0.5d_{FA}$, $d_{CSA} = 0.5d_{FA}$, or $d_{CSA} = d_{FA}$, depending on the relative availability of the 3 operands [3], $d_{reg} = 1.5d_{FA}$. Since the shifter SHIFT is basically a multiplexer with a driver on its control lines, we have assumed $d_{SHIFT} = d_{driver} + d_{MUX} = (1 + 0.5)d_{FA} = 1.5d_{FA}$. For the implementation of NORM in the ES2-ECPD10 technology we assume (by looking at the boolean functions) to have a delay $d_{NORM} \leq 2.0d_{FA}$. Concerning the delay of the CPA, since it is on 12 bits, we assume a logarithmic delay [12] $d_{CPA} = 4d_{FA}$ whereas for the block NORMAL in the architecture of the QCA, since it can be implemented with just one level of logic, we assume $d_{NORMAL} \leq 0.5d_{FA}$. By substituting these values in (3) and (4), we get

$$D_{PA} = (.5+.5+.5+2.0+1.5+1.5)d_{FA} = 6.5d_{FA} \quad (5)$$

$$D_{NR} = (.5+4+.5+.5+1.5+1.5)d_{FA} = 8.5d_{FA} \quad (6)$$

A comparison of (5) with (6) indicates that the cycle duration of the proposed architecture is faster of about 25% than the corresponding architecture based on the QCA and using the same high level organization.

5.3 Compression

To study the compression capabilities of the CFQCA the size of the code strings obtained by varying parameters of the algorithm as t (the number of fractional bit used in estimating the value of A) and s (the number of spacer bits inserted in the representation of C) has been taken into account. For this analysis the 8 test documents of the CCITT study group XIV scanned with a resolution of 1728 × 2378 pels have been considered. As a predictor we use a simple 7 bit predictor very similar to the one described in [9] and used in [11] to analyze the performances of the QCA.

The size in bits of the code string output for the considered input documents and for several values of t has been reported in Table 1, together with the size of the code string obtained out of the QCA (the size of the original test documents is 4,105,728 bits). In this case s has been assumed to be equal to 4, as in the original QCA. However, if $t \geq 2$ the size of the obtained code string is almost equal to the case when $s = 2$ or $s = 3$. Observe that the compression ratios vary from a minimum of 9-to-1 (CCITT7) to a maximum of 57-to-1 (CCITT2), with an average of about

File	CFQCA (estimate of A on 2 integer and t fractional bits)							QCA	Comments
	t=0	t=1	t=2	t=3	t=4	t=5	t=12		
CCITT1	131048	124848	120984	120200	119704	119808	119768	119752	business letter
CCITT2	77896	74088	72024	71664	71336	71536	71568	71568	circuit drawing
CCITT3	207008	195832	190256	188872	187984	187912	187728	187728	French invoice
CCITT4	500368	472528	454288	449504	447864	446864	447008	447000	dense text
CCITT5	238384	225288	218632	216752	215712	215432	215888	215888	math book page
CCITT6	123112	115832	112896	112800	112392	112504	112256	112256	graph
CCITT7	525400	496000	477368	470928	469624	468576	468248	468232	Kanji
CCITT8	138560	132304	126472	125280	124664	124376	124392	124392	memo and sign

Table 1. Coding performances of the CFQCA varying the number t of fractional bits (with $s = 4$).

File	Number s of spacer bits		
	4	3	2
CCITT1	124848	124856	124856
CCITT2	74088	74088	74096
CCITT3	195832	195832	195848
CCITT4	472528	472536	472552
CCITT5	225288	225288	225304
CCITT6	115832	115832	115840
CCITT7	496000	496008	496040
CCITT8	132304	132304	132312

Table 2. Coding performances of the CFQCA varying the number s of spacer bits (with t=1).

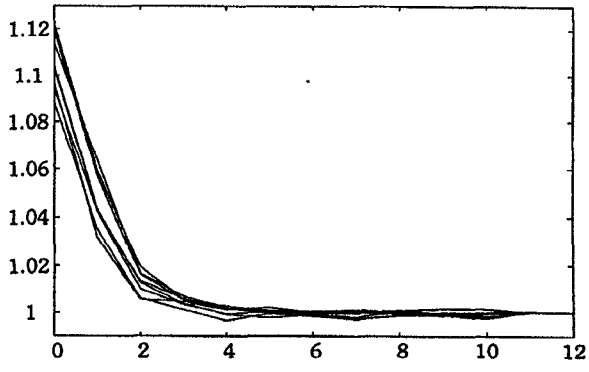

Figure 3. Size of the CFQCA coded string for the 8 CCITT test documents versus t.

30-to-1. This implies that, on the average, one normalization takes place every 30 symbols in input: it follows that the shifts are sufficiently low frequent. Therefore, our choice to limit the encoder to a single shift per cycle versus the alternative implementations in [1] is well justified. Table 2 shows the effect of the parameter s on the size of the code string, evaluated for $t = 1$. As can be easily seen the differences in term of the achieved compression are really negligible. In Fig. 3 the ratio between the size of the code string obtained by the CFQCA (assuming $s = 4$) and that obtained by the QCA is plotted versus the value of t for the 8 CCITT test documents. From these plots we can observe that:

1. the architecture depicted in Fig. 2 (where $t = 2$) achieves compression ratios which are from 0.5 to 2% worse than the corresponding QCA; the cycle speedup has been estimated to be about 25%;

2. higher compression rates, (less than 0.3% worse than the QCA) are obtained with a small increase of t ($t = 4$); the cycle speedup can be estimated to be about 15%;

3. for $t = 5$ the same compression ratio is achieved on the average: this implies that it is not worth considering $t > 5$; in this case, the cycle speedup has been estimated to be about 10%.

From Fig. 3 we can see that the plots of the 8 test documents are very similar: this implies that our conclusions are very general and are not affected by the choice made of the test documents.

6 Conclusions

In this paper we have studied carry a free Q-Coder algorithm (CFQCA). We have also presented a possible implementation of our CFQCA. A comparison between the evaluation of cycle delays of the the proposed architecture with the corresponding architecture based on the QCA has indicated that the proposed architecture is faster from about 25% to 10% with compression ratios less than 2% worse.

The price that the proposed architecture has to pay to achieve such speedup, is mainly identified in increased hardware requirements and (slightly) lower compression rates. Concerning the first issue, we can observe that the increase in required hardware for our implementation is very small since it is basically identified in the two pairs of registers instead of just the two registers for holding A and C, the block OTFC and the slightly more complex implementation of block NORM. On the other hand, we decrease the complexity of hardware for additions by using carry free adders instead of carry propagated ones. Concerning the second issue, the experimental results (section 5.3), have indicated that our algorithm achieves compression ratios which are anyway less than 2% worse than the QCA for $t = 2$ and even better for higher values of t. A small increase in the number t of fractional bits of the estimate $\widehat{A}$ improves the compression ratios to values which don't differ in a significant way from those obtained with the QCA.

Actually, an increase of t implies an increase of hardware complexity of NORM, and hence also an increase of the cycle delay. Therefore, for each specific application, a tradeoff among the parameters (compression factor, area, delay) must be found, which permits the design of an efficient encoder.

Anyway, the proposed algorithm and architecture offer an interesting alternative when a fast and efficient encoder must be designed, implemented and used.

References

[1] G. Cena, P. Montuschi and L. Ciminiera, "A Q-Coder Algorithm with Carry Free Addition," Politecnico di Torino, I.R. DAI/ARC 5-94, 1994.

[2] M. D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," IEEE Trans. Comput., Vol.C-36, pp.895-897, July 1987.

[3] M. D. Ercegovac and T. Lang, "Division and Square Root: Digit-Recurrence Algorithms and Implementations," Kluwer Academic Publishers, U.S.A., 1994.

[4] M.D. Ercegovac, T. Lang and P. Montuschi, "Very-High Radix Division with Selection by Rounding and Prescaling," IEEE Transactions on Computers, Vol.43, No.8, August 1994, pp.909-918.

[5] European Silicon Structures, ES2 ECPD10 Library Databook, April 1991.

[6] G. Goertzel and J. L. Mitchell, "Symmetrical Optimized Adaptive Data Compression/ Transfer/Decompression System," U.S. Patent 4,633,490, December 30, 1986.

[7] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proc. Inst. Electr. Radio Eng., 40, 9, September 1952, pp. 1098-1101.

[8] J. Jiang,"Novel Design of Arithmetic Coding for Data compression," IEE Proceedings, Vol. 142, Pt. E, No. 6, pp. 419-424, November 1995.

[9] G. G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," IEEE Transactions on Communications, Vol. COM-29, No. 6, June 1981, pp. 858-867.

[10] G. G. Langdon, "An Introduction to Arithmetic Coding," IBM J. Res. Develop., Vol. 28, No. 2, March 1984, pp. 135-149.

[11] J. L. Mitchell and W. B. Pennebaker, "Software Implementations of the Q-Coder," IBM J. Res. Develop., Vol. 32, No. 6, November 1988, pp. 752-774.

[12] R.K. Montoye, E. Hokenek and L. Runyon, "Design of the IBM Risc System/6000 Floating-Point Execution Unit," IBM J. Res. Develop. Vol. 34, No. 1, pp. 59-70, January 1990.

[13] P. Montuschi and M. Mezzalama, "Survey of square rooting algorithms," IEE PROCEEDINGS, Vol. 137, Pt. E, No. 1, pp. 31-40, January 1990.

[14] J. Rissanen and G. G. Langdon, "Universal Modeling and Coding," IEEE Transactions on Information Theory, Vol. IT-27, No. 1, January 1981, pp. 12-23.

[15] ITU-T Recommendation T.81, "Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines," September 1992.

[16] ITU-T Recommendation T.82, "Information Technology - Coded Representation of Picture and Audio Information - Progressive Bi-Level Image Compression," March 1993.

[17] I. H. Witten, R. M. Neal and J. G. Clearly, "Arithmetic Coding for Data Compression," Communications of the ACM, Vol. 30, No. 6, June 1987, pp. 520-540.