Optimal hardware and software arithmetic coding procedures for the Q-Coder

by J. L. Mitchell
W. B. Pennebaker

The Q-Coder is an important new development in arithmetic coding. It combines a simple but efficient arithmetic approximation for the multiply operation, a new formalism which yields optimally efficient hardware and software implementations, and a new form of probability estimation. This paper describes the concepts which allow different, yet compatible, optimal software and hardware implementations. In prior binary arithmetic coding algorithms, efficient hardware implementations favored ordering the more probable symbol (MPS) above the less probable symbol (LPS) in the current probability interval. Efficient software implementation required the inverse ordering convention. In this paper it is shown that optimal hardware and software encoders and decoders can be achieved with either symbol ordering. Although optimal implementation for a given symbol ordering requires the hardware and software code strings to point to opposite ends of the probability interval, either code string can be converted to match the other exactly. In addition, a code string generated using one symbol-ordering convention can be inverted so that it exactly matches the code string generated with the inverse convention. Even where bit stuffing is used to block carry propagation, the code strings can be kept identical.

1. Introduction

A new binary arithmetic coding system, the Q-Coder, has been developed as a joint effort by the authors of this paper and colleagues at the Almaden site of the IBM Research Division. This paper covers one key aspect of the Q-Coder system, compatible optimal software and hardware coding algorithms. An overview of the Q-Coder is found in [1]. Other aspects of the Q-Coder such as the probability-estimation technique and the software and hardware implementations are described in three companion papers [2–4]. The Q-Coder is part of a proposal submitted to the CCITT and ISO Joint Photographic Experts Group (JPEG) for color photographic image compression [5]. The overview [1] also contains a more comprehensive list of references to earlier work on arithmetic coding.

A brief review of arithmetic coding is given in [1]. A more extensive review has been published by Langdon [6]. As discussed in these references, arithmetic coding is closely related to Elias coding [7]. A sequence of symbols is coded

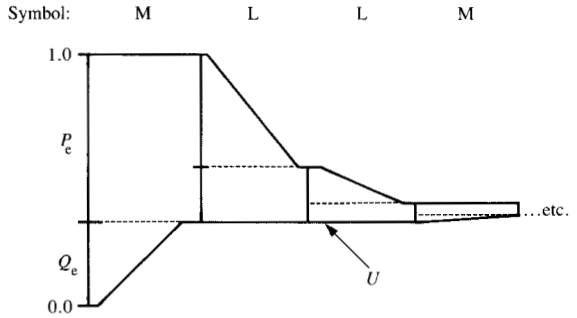

Figure 1

Elias coding recursion. Note that in this particular example of Elias coding the symbols are ordered in the interval such that the more probable symbol (M) is above the less probable symbol (L).

by sending a point on the probability number line between 0 and 1 with sufficient precision to uniquely identify the probability interval assigned to that particular sequence. More specifically, in Elias coding the interval for a particular sequence of binary symbols is developed by a recursive subdivision process in which each binary decision is coded by subdividing the interval. This recursive subdivision is sketched in **Figure 1**. Note that at the end of each recursion the unrenormalized code string U is left pointing at the subinterval for the symbol just coded. The code string grows in length as coding proceeds, because it must be of sufficient precision to uniquely identify the interval.

In Figure 1 the symbols are ordered in the probability interval such that the more probable symbol (MPS or M) occupies the upper subinterval and the less probable symbol (LPS or L) occupies the lower subinterval. As each symbol is coded, the interval is subdivided into two parts in proportion to the probability estimates, P_e and Q_e, for the MPS and LPS symbols respectively. A specific convention is followed in developing the unrenormalized code string U in Figure 1—the code string is defined to point to the *base* of the current interval. Because no renormalization is used, the coding sequence sketched in Figure 1 requires increasing arithmetic precision as more symbols are coded. When this coding system is modified to use fixed-precision arithmetic, it becomes an arithmetic coder.

Figure 1 shows one particular symbol-ordering convention (P/Q), and one convention where the string points to the base of the interval. However, the symbol-ordering convention can be inverted (Q/P), and other code-string conventions are possible. The interrelationships among symbol ordering, code-string conventions, and efficiency of implementation are a central topic of this paper.

In Section 2, the basic coding structures which allow optimal hardware and software coding implementations are explored. In Section 3, the techniques for obtaining compatible (and in fact, identical) code strings when following different code-string conventions are presented. In Section 4, a technique for inverting the code string so as to give compatible code strings for different symbol-ordering conventions is given. Section 5 discusses the code-register structure and shows that code space is opened up for some structures which can be used to "escape" from the arithmetic coding environment.

2. Optimal hardware and software structures

In this section the basic coding structures for optimal hardware and software implementations are explored. Unless specifically noted, the P/Q symbol-ordering convention followed in Figure 1 is used. The optimal hardware coding structure discussed in this section was developed by Langdon and Rissanen [8–10] for their Skew Coder [10]. The optimal software structure which uses inverted symbol ordering is also found in earlier work [8, 11].

To implement the coder of Figure 1 in fixed-precision arithmetic, some constraints must be adopted on the range of probability estimates for the LPS. In addition, the code string and interval must be renormalized periodically, and something must be done to limit carry propagation in the code string. As discussed in [1], a good resolution of these issues has been provided by Langdon and Rissanen in the Skew Coder [10].

The Skew Coder uses a precision of 12 bits for the LPS probability, thereby limiting the minimum LPS probability to approximately 2^{-12}. It uses the symbol-ordering and code-string conventions of Figure 1. It also follows a renormalization rule which keeps the interval of order unity, thereby allowing the multiplicative scaling of the interval to be approximated by either subtraction or substitution. Finally, it uses a bit-stuffing procedure [9] to limit carry propagation. All of these concepts have, with modifications and extensions, been carried over to the Q-Coder.

Langdon and Rissanen's Skew Coder is a hardware-optimized arithmetic coder. Indeed, the major difference between the arithmetic coding procedures in the Skew Coder and the hardware-optimized Q-Coder is a change in the renormalization rule. The Skew Coder renormalization maintains the interval A between 1 and 2, while the Q-Coder renormalization maintains the interval between 0.75 and 1.5. Both renormalize by a shift-left-logical operation, but the rule used in the Q-Coder reduces the coding inefficiency caused by the approximation to the multiply. The renormalization rule used in the Q-Coder was suggested by J. Rissanen.[1]

[1] J. J. Rissanen, IBM Research Division, Almaden Research Center, San Jose, CA, private communication.

Following the conventions of Figure 1, the hardware-optimized coding process, with approximations for the multiplications such that $AQ_e \cong Q_e$ and $AP_e = A(1 - Q_e) \cong A - Q_e$, is

if MPS is encoded

$C \leftarrow C + Q_e$

$A \leftarrow P_e = A - Q_e$

renormalize A and C if necessary

else (LPS is encoded)

$A \leftarrow Q_e$

renormalize A and C

end

and the matching decoder is

if $C \geq Q_e$

(MPS is decoded)

$C \leftarrow C - Q_e$

$A \leftarrow A - Q_e$

renormalize A and C if necessary

else

(LPS is decoded)

$A \leftarrow Q_e$

renormalize A and C

end

Note that after decoding a symbol, the decoder subtracts from the code string any interval the encoder added.

A mathematical formulation is now developed which allows the interrelationships between the different symbol-ordering and code-string conventions to be expressed concisely. For the encoding process, define $U(j)$ as the unrenormalized code string, $C(j)$ as the renormalized code string, $A(j)$ as the current interval, and $R(j)$ as the renormalization factor which keeps $A(j)$ of order unity. Then,

$$U(j) = R(j)C(j). \tag{1}$$

The renormalizing factor has the form $R(j) = 2^{-k(j)}$, where $k(j)$ is the total number of renormalization shifts required to return $A(j)$ to the interval $1.5 > A(j) \geq 0.75$ *after* the jth symbol is coded. Analogous relationships hold for other code-string definitions used below.

Since the code string is designed to point to the bottom of the current interval, the unrenormalized code string *after* the jth symbol is coded is

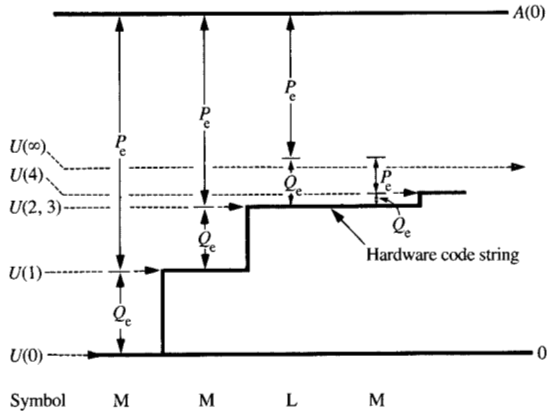

Figure 2

Code-string generation for P/Q symbol ordering in the interval, for the hardware coding convention. The unrenormalized sequence of values taken by the hardware code string is indicated on the left side. Note that renormalization must scale the size of the current interval and the code string identically.

$$U(j) = \sum_{i=1}^{j} R(i-1)Q_e(i)\delta_{S_i=\text{MPS}}, \tag{2}$$

where S_i is the ith symbol, $Q(i)$ is the LPS estimated probability for S_i, and $\delta_{\text{condition}}$ is the equivalent of the Kronecker delta function (1 if the condition is true, 0 if false).

After the jth symbol is decoded, the corresponding expression for the decoder code string $U_d(j)$ is

$$U_d(j) = U(k) - \sum_{i=1}^{j} R(i-1)Q_e(i)\delta_{S_i=\text{MPS}}, \tag{3}$$

where $U(k)$ is the code string generated by the encoder in coding k symbols, and $k > j$. When the MPS is decoded, the decoder subtracts the portion of the interval allocated to the LPS in order to keep the reference level for the decoder decision at zero for the next decoding operation. Consequently, if $C_d(j) < Q_e(j+1)$, the code string is pointing to the interval allocated to the LPS, and the next $(j+1)$ decision is decoded as an LPS.

The operation of this type of encoder and decoder is illustrated in **Figures 2** and **3**. This encoder and decoder structure is appropriate for hardware implementation because the interval subdivision and the code-string addition can be done in parallel [1]. However, a software implementation following this logic is not as efficient, since two arithmetic operations are required on the MPS path, while only one arithmetic operation is required on the LPS path. The arithmetic coding used in the IBM Internal

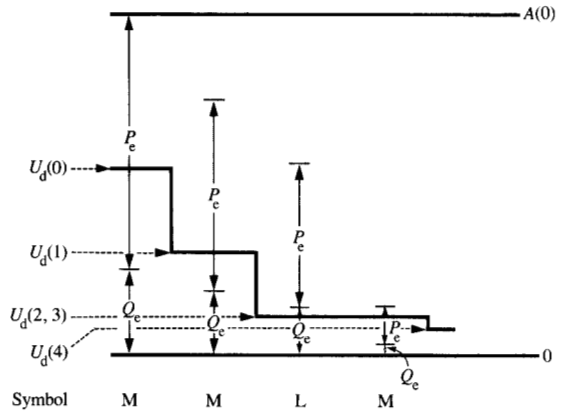

Figure 3

Hardware decoder operation, without renormalization.

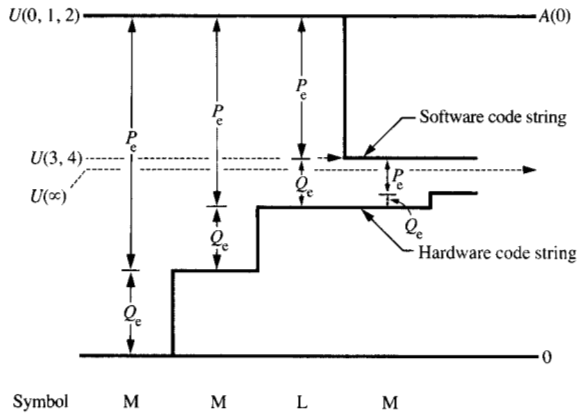

Figure 4

Code-string generation for P/Q symbol ordering in the interval, for the software coding convention. The unrenormalized sequence of values taken by the software code string is indicated on the left side. The code string generated with the hardware coding convention (Figure 2) is shown for comparison.

Teleconferencing System (Goertzel and Mitchell [11]) solved this problem by inverting the order of the symbols on the number line. Then, fewer operations are required on the MPS path.

An efficient software implementation of the encoder can be realized without reordering the symbols by pointing the code string C to the top of the current interval rather than the bottom. Then, the encoding process is as follows:

$A \leftarrow A - Q_e$

if MPS is encoded

    renormalize A and C if necessary

else (LPS is encoded)

    $C \leftarrow C - A$

    $A \leftarrow Q_e$

    renormalize A and C

end

C is initialized at $A(0)$, the starting interval, and always points to the top of the probability interval. With each coding operation the interval shrinks; if the LPS is coded, the MPS interval is subtracted from the code string. A sketch of code-string generation for the software implementation is shown in **Figure 4**; the hardware code string from Figure 2 is also shown for comparison. The interval between the two code strings is the unrenormalized probability interval, $R(i) \times A(i)$. Therefore, if the remaining interval is subtracted from the software code string after the last symbol is coded, the two code strings will—in principle—be identical. However, when finite-precision arithmetic is used, the two code strings are not necessarily the same, and may be incompatible. In Section 3 the effects of finite-precision arithmetic are treated, and a method for generating the hardware code string while following the software coding convention is described.

The mathematical description of the software encoder is

$$U(j) = A(0) - \sum_{i=1}^{j} R(i-1)P_e(i)\delta_{S_i=\text{LPS}}. \quad (4)$$

Since $A(i-1)$ is the interval at the start of coding of the ith symbol, $P_e(i) = A(i-1) - Q_e(i)$. Figure 4 illustrates the operation of this encoder.

Following the decoding of a given symbol, the decoder must add back to the code string any interval the encoder subtracted. Thus, for $k > j$, the decoder should follow the form

$$U_d(j) = U(k) + \sum_{i=1}^{j} R(i-1)P_e(i)\delta_{S_i=\text{LPS}}. \quad (5)$$

However, in this form the unrenormalized decoder code string converges to the top of the initial interval $A(0)$, and the test to decode the next symbol is rather awkward:

$$C_d(j) < [A(0)/R(j)] - [A(j) - Q_e(j+1)]. \quad (6)$$

As decoding proceeds, $R(j)$ gets smaller and smaller, and the precision required for the decoding grows correspondingly.

There are two ways to convert this decoder to a form which allows fixed-precision arithmetic in the decoding. The simplest is to start the decoder code string at $-A(0)$ rather than at zero, and to subtract the *same* amount from the starting interval. The top of the interval—the reference level for this decoder—is then at zero. If we define a new decoder code string by

$$U_{d'}(j) = U_d(j) - A(0), \tag{7}$$

the comparison of Equation (6) becomes

$$C_{d'}(j) < -[A(j) - Q_e(j+1)]. \tag{8}$$

Both $C_{d'}(j)$ and $-[A(j) - Q_e(j+1)]$ are negative, but always within $|A(j)|$ of 0. Therefore, the software decoder can be implemented using fixed-precision arithmetic. The software decoder is thus

$A \leftarrow A + Q_e$

if $C_{d'} \geq A$

(MPS is decoded)

renormalize A and $C_{d'}$ if necessary

else

(LPS is decoded)

$C_{d'} \leftarrow C_{d'} - A$

$A \leftarrow -Q_e$

renormalize A and $C_{d'}$

end

The operation of this decoder is illustrated in **Figure 5**.

The second form for the decoder can be achieved by inverting the code string (and symbol ordering on the number line). Then, the inverted code string $\check{C}(j)$ is

$$\check{C}(j) = [A(0)/R(j)] - C(j). \tag{9}$$

This inversion process appears to have a problem with arithmetic precision. However, a technique for resolving this problem is discussed in Section 4. From Equation (3) the decoder becomes ($k > j$)

$$\hat{U}_d(j) = \check{U}(k) - \sum_{i=1}^{j} R(i-1)P_e(i)\delta_{S_i=\mathrm{LPS}}, \tag{10}$$

and the decoder decodes an MPS if $\check{C}_d(j) < P_e(j+1)$.

In the software implementation of this decoder, the code-string inversion is done as each byte of the normal code string is read. The actual decoding process then becomes

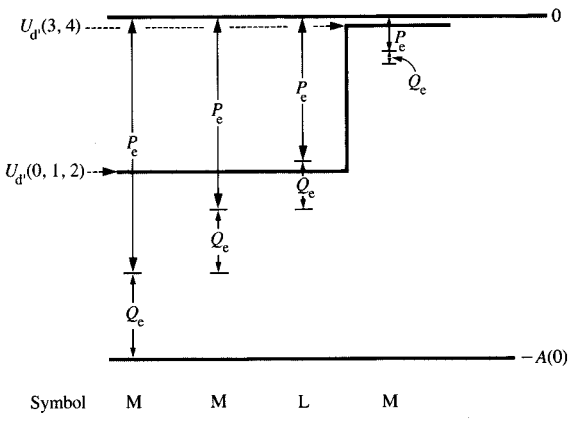

Figure 5

Software decoder operation, without renormalization.

$A \leftarrow A - Q_e$

if $\check{C} < A$

(MPS is decoded)

renormalize A and $\check{C}$ if necessary

else

(LPS is decoded)

$\check{C} \leftarrow \check{C} - A$

$A \leftarrow Q_e$

renormalize A and $\check{C}$

end

There is an entirely equivalent complementary set of encoder and decoder structures in which the symbol ordering in the interval is inverted. The second software decoder described above really belongs to this complementary set, and the code-string inversion allows translation to the code strings generated by the complementary form. **Figure 6** shows the full family of compatible encoders and decoders.

This section has explored the encoder and decoder structures which are derived for the different symbol-ordering and code-string conventions. The next two sections address problems in making the code strings compatible for these structures.

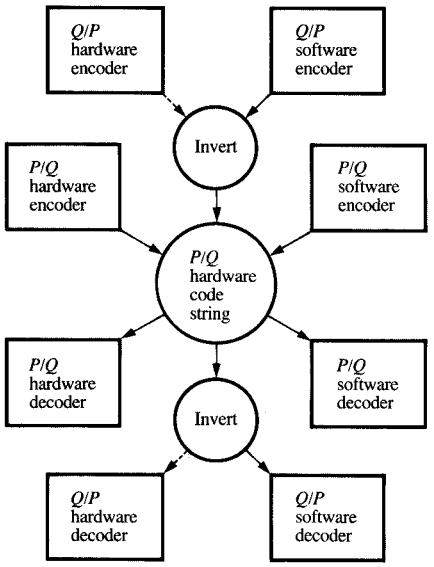

Figure 6

The complete set of compatible arithmetic encoders and decoders. If the box is connected by a dashed line, that particular variation was not implemented.

3. Obtaining compatible code strings for a given symbol ordering

The generation of compatible code strings for the two code-string conventions and a given symbol-ordering convention depends on a simple but important observation. The difference between code strings which point at the bottom and top of the probability interval is simply, by definition, the current probability interval. If after the ith symbol is coded, $C_T(i)$ is the renormalized code string pointing to the top of the interval and $C_B(i)$ is the renormalized code string pointing to the bottom of the interval,

$$C_T(i) = C_B(i) + A(i). \tag{11}$$

For any future symbol n, the lower and upper bounds $C_B(n)$ and $C_T(n)$ are given by

$$C_T(i) > C_B(n) \geq C_B(i), \tag{12}$$

$$C_T(i) \geq C_T(n) > C_B(i). \tag{13}$$

As n increases, the probability interval shrinks, and $C_B(n)$ and $C_T(n)$ converge to a common code string $C(n)$. Consequently, for $n \gg i$ the future code string can approach arbitrarily close to either bound. Thus, within the current precision of $C_T(i)$ and $C_B(i)$,

$$C_T(i) \geq C(n) \geq C_B(i). \tag{14}$$

Equation (14) is used in a number of ways in this section and the next section.

Consider implementation of the hardware coding convention with fixed-precision arithmetic. The Q-Coder code string is defined to follow the hardware coding convention for P/Q symbol ordering.

For convenience, the code string is separated into two parts, a code buffer which contains completed bytes of coded data, and a 32-bit code register which contains the low-order bits of the code string. At 8-bit intervals, completed bytes are taken from the code register and added to the code buffer. At that time, any carry-propagation problems are resolved by bit stuffing.

The basic concept of bit stuffing to block carry propagation is described in [9]. The carry propagation is blocked by detecting runs of 1-bits and inserting a 0-bit whenever the run length reaches a predefined length. The decoder, operating under the same rules, shifts the data so that any carry-over into the stuffed 0-bit is properly aligned when the new data are added to the code register.

The detection of patterns which would allow excessive carry propagation is greatly simplified in software if an additional constraint is introduced—namely, that the runs fill complete bytes (a 2-byte alignment with a stuffed byte was used in [11]). The minimum possible run length of binary ones is then a single coded byte with a value of hexadecimal 'FF' (X'FF'); bit stuffing after a single X'FF' byte is, in fact, the convention adopted for the Q-Coder. For simplicity, the rest of this discussion is restricted to this case. The Q-Coder also uses an integer representation suggested by G. G. Langdon,[2] in which hexadecimal 1000 is equivalent to the decimal fraction 0.75. With this representation the renormalized interval is a 13-bit binary fraction with the most significant bit set. Finally, the Q-Coder adopts the constraint that only the "current byte"—the byte most recently moved from the code register to the code buffer—can be modified by a carry from the code register.

In the software encoder the code string is generated by subtracting the MPS subinterval from the renormalized code string C whenever the LPS occurs. Consequently, borrow propagation rather than carry propagation must be prevented. Borrow propagation could be prevented by stuffing a 1-bit following any string of binary zeros of some given length. However, the code strings would then be incompatible with the code strings generated following the hardware conventions. Therefore, another approach must be taken.

To get compatible (in fact, identical) code strings, a simple conversion process is followed which converts the software code string to the hardware code string. This conversion

[2] G. G. Langdon, Jr., University of California at Santa Cruz, Santa Cruz, CA, private communication.

process also creates the same bit-stuffing structure, which is needed to block hardware carry propagation.

In the hardware encoder, the binary pattern in the code buffer triggering the bit stuffing is

$\cdots$, 11111111,

where the byte boundaries are denoted by commas. Any X'FF' in the code string must by definition be followed by at least one stuff bit. Thus, if the current byte is X'FF', the bits following have the pattern

$\cdots$, 11111111, *Snnnnnnn*, $\cdots$,

where *S* is the stuff bit containing any carry from bits *nnnnnnn*. In general, more than one stuff bit may be introduced. However, in Section 5 a proof is given that with proper constraints on the code-register structure, a single-bit stuff is sufficient and even provides for "illegal" (escape) codes in the byte following X'FF'. The presence of these escape codes also guarantees that an X'FF' byte cannot follow an X'FF' byte in the code string.

The decoder detects the X'FF' byte and shifts the new code byte left by one bit position before adding the byte to the code register. This properly aligns the carry bit with the data already in the code register.

In the software encoder described in the preceding section, the subtraction causes borrow propagation rather than carry propagation, and borrow propagation is triggered by runs of binary zeros:

$\cdots$, 00000000.

Since the goal is to create a code string which is compatible with the hardware convention for bit stuffing, the code string is created subject to two constraints: First, any X'FF' generated by the software coder must be followed by a stuffed bit. Otherwise, byte patterns which are illegal for the hardware decoder may be generated. Second, the code string is constructed such that whenever a borrow is needed from the current byte, it can, by definition, be taken without underflow occurring. In Section 5 a proof is given that, when following the hardware coding conventions, the register structure used in the Q-Coder allows no more than one carry to propagate as far as the current byte in the code buffer. Similarly for the software conventions, only one borrow can propagate as far as the current byte in the code buffer. Therefore, the only byte value which cannot be borrowed from is zero.

The need for a borrow from the current byte is detected by setting a high-order "preborrow" bit in the code register at the start of a new byte. If this preborrow is used during the creation of the next byte, a borrow must be taken from the current byte. One convenient implementation places the preborrow bit in the bit position *P*, which becomes the register sign bit at the completion of the next byte. A flag bit *F* is also inserted one bit to the left of *P*. When the code register goes negative, the next renormalization shift completes the new code byte. Thus, when a new byte starts,

Code register: 0000000*F*, *P*0000000, *xxxxxxxx*, *xxxxxxxx*
A register : 000*aaaaa*, *aaaaaaaa*

where *F* and *P* are both set. When the next byte is complete,

Code register: *P*0000000, *nnnnnnnn*, *xxxxxxxx*, *xxxxxxxx*
A register : 000*aaaaa*, *aaaaaaaa*

If the code register is positive, the preborrow has been used and a borrow is therefore taken from the current byte before the new byte, *nnnnnnnn*, is transferred from the code register to the buffer. Note that when the preborrow is used, the value in the code register is always larger than the A register, and future borrows can be taken from the code-register contents. A borrow will never be needed from the new code byte, and it can therefore be written to the code buffer without further testing.

If the code register is negative, no borrow is needed from the current byte and the unused preborrow *P* is discarded. The A register is then compared to the code register. If the A register is larger than the code register, two things have been detected: First, the next byte to be written (*nnnnnnnn*) is zero. Second, the bounds on the software code string are such [Equation (14)] that a borrow from the current byte might be needed. A borrow is therefore immediately taken from the zero (*nnnnnnnn*) byte, converting it to X'FF', and propagated to the current byte in the code buffer. The X'FF' is then written to the code buffer. Since this byte is X'FF', bit stuffing is required when the next byte is written. Therefore, in addition to the normal preborrow bit *P*, the bit borrowed from the X'FF' is placed in bit position *C*, the bit position which becomes the carry bit for the next byte. Thus, if the code register is smaller than the A register, a borrow is taken from the current byte, an X'FF' is written to the code buffer, and the code register is modified to become

Code register: 0000000*F*, *P*000000*C*, *xxxxxxxx*, *xxxxxxxx*
A register : 000*aaaaa*, *aaaaaaaa*

where *F*, *P*, and *C* are all set to 1. When the next byte is complete,

Code register: *P*000000*C*, *nnnnnnnx*, *xxxxxxxx*, *xxxxxxxx*
A register : 000*aaaaa*, *aaaaaaaa*

The preborrow *P* will not be used when bit *C* is set. The current byte value of X'FF' will therefore trigger bit stuffing, and the next byte will be (*Cnnnnnnn*) rather than (*nnnnnnnn*). The bit in *C* goes into the stuff-bit position. If the *C* bit borrowed from the preceding byte remains unused, a hardware code-string carry would occur; if the *C* bit has been borrowed from, a hardware code-string carry would not occur.

Note that after a stuff bit, one extra valid x bit remains in the code register. Therefore, the preborrow bit and the flag bit for the next byte are inserted one bit to the left of the normal position.

If the code register is not smaller than the A register, the current contents of the code register are large enough to satisfy any borrow requirements. The current byte must then be checked; if it is X'FF', bit stuffing is triggered. In this case, since no preborrow was required, the stuffed bit will be zero.

If all zero bytes were simply converted to X'FF', a hardware decoder could decode the resulting code string. However, the look-ahead to see if a borrow might be needed when the byte to be written is zero makes the resulting code string identical to the hardware code string. In effect, this look-ahead detects the existence of the X'FF' that the equivalent hardware coder would have generated. It also resolves a problem with precision which otherwise would occur in coding long sequences of MPSs.

If it were desired, an entirely equivalent inverse bit stuffing following zero bytes could be effected, with hardware doing the look-ahead for the X'FF' pattern to see if an overflow to X'00' might be possible.

4. Code-string inversion

In this section we consider the process of code-string inversion. Only inversion to and from the hardware code string for P/Q symbol ordering is treated, since the Q-Coder is defined to use those conventions. Other code-string translations are direct extensions of this case.

First consider the code-string inversion process of Equation (9). In this case the inversion is carried out in the decoder, converting the hardware code string for P/Q symbol ordering to the software code string for Q/P symbol ordering.

If only LPSs are coded (a remote but real possibility), the code string remains zero. Subtraction of the code string from $A(0)$ as required for inversion then encounters precision problems unless a preborrow from $A(0)$ is propagated through all understood binary zeros in the binary fraction. In effect,

$$\text{binary } 1.0000000\cdots = \text{binary } 0.1111111\cdots + \varepsilon,$$

where $\varepsilon = R(i) \times 2^{-12}$ for a 12-bit integer representation of Q. ε is therefore the smallest change in A allowed by the integer representation. The preborrow of ε also has the desirable attribute of shifting the inverted code string slightly so that it is always within the correct interval for the software coding convention with Q/P symbol ordering. For Q/P symbol ordering, the software coding convention requires the code string to point to the bottom of the current interval. Then, the MPS interval is added to the code string each time the LPS symbol is coded.

Inversion in the encoder is a more complex process than inversion in the decoder, since the translation to the hardware bit-stuffing convention must be done at the same time. The encoding is done with Q/P symbol ordering, and the resulting code string is inverted to generate the hardware code string for P/Q ordering. Each time a new byte of coded data is completed in the code register, it is inverted by subtracting it from $A(0)$, and the difference is placed in the buffer. The inversion process requires the same preborrow as in the decoder, such that the subtraction of the new byte of code is always from X'FF' (unless bit stuffing is needed).

Any carry created during the coding of the new byte is normally subtracted from the current byte as part of the inversion process. Consequently, inversion also requires a look-ahead (as in the software encoder for P/Q ordering) to see if a borrow might be needed from any X'00' byte created by the inversion. This look-ahead is done by summing the current code register with the A register—the sum provides an upper limit to the size of the code register as additional symbols are coded [Equation (14)]. If adding the A register to the code register causes a carry out of the newly completed code byte still in the register, that byte must be an X'FF'. Furthermore, a borrow might be required from the X'00' byte which results from the subsequent inversion of the X'FF' byte, and borrowing from it later is illegal. Therefore a borrow is taken immediately. It is conceptually taken from the X'00' byte resulting from the inversion (converting it to X'FF') and propagated to the current byte in the code buffer. A borrow flag is then set indicating that this borrow was taken, and the X'FF' is written to the code buffer. When the next byte is complete, the X'FF' triggers bit stuffing. The bit which was borrowed is put into the stuff-bit (carry-receiver) position before subtracting the new byte. If a carry has occurred in the new byte, it is subtracted from this bit as the new byte is inverted. Thus, if the current byte is X'FF', the next byte is subtracted from X'FF' when the borrow flag is set; otherwise it is subtracted from X'7F'.

For some definitions of the code-register bit assignments, the preborrow can propagate through an arbitrary number of X'FF' bytes in succession. However, with the code-register bit assignment discussed in the next section, an X'FF' in a byte with a stuff bit cannot happen.

5. Escaping from the arithmetic code string

In many coding environments it is highly desirable to provide an escape from the code string that can be located by a control system external to the arithmetic decoding system. Arps et al. [12] noted that following bit stuffing, certain code words were illegal, thus providing an escape, and that the key to the development of illegal code space is related to the introduction of "spacer bits" in the register structure. Langdon[3] also suggested this concept to limit the

[3] G. G. Langdon, Jr., private communication.

number of carries. In this section, that idea is developed to provide a rigorous bound on the code-register value when the X'FF' code byte is followed by bit stuffing. In particular, the 2-byte pattern X'FFFF' is shown to be illegal when more than one spacer bit is used. The hardware coding convention is followed in this section.

The bit pattern of the code register holding the partially completed next byte of the code string must be bit-aligned with the interval register A. Thus, for the integer representation chosen for the Q-Coder, one possible bit assignment in the encoder registers would be

Code register: 0000000*C nnnnnnnn ssssxxxx xxxxxxxx*
A register : 00000000 00000000 000*aaaaa aaaaaaaa*

where 0 indicates zero bits, C is a carry-receiver bit, n labels bits in the positions where the next code byte is generated, s indicates spacer bits needed to limit carry propagation, and the x bits contain the binary fraction still being developed in the code register. The number of spacer bits is chosen so that if the code register is memory-mapped, the completed byte will be byte-aligned. The a bits represent the value in the A register.

If the preceding code byte is X'FF', the byte removed from the register is shifted by one bit, such that the carry bit is the highest-order bit of that byte. In this case one of the n bits is left in the register, and the renormalization count to complete the next byte in the code register is reduced by 1.

Assuming that the byte taken from the code register does not contain a stuff bit, the rules for bit positioning and renormalization provide an upper bound on the values remaining in the two registers after removal of the byte. If we define a binary point between the s and x bits in the registers, then the upper bound on $C[0]$ (the part of the code string remaining in the code register after removal of a byte of coded data) is

$$C[0] < 2^s, \tag{15}$$

where s is the number of spacer bits. The code string cannot be increased by more than $A[0]$, the value of the A register after removal of a byte of coded data. Since for this discussion the binary point is positioned such that $1 \le A[0] < 2$, after eight renormalization shifts the maximum value of the sum $A[8] + C[8]$ is bounded by

$$A[8] + C[8] \le 2^8(A[0] + C[0]) \tag{16}$$

for any possible symbol sequence. Since $A[8] > 0$,

$$C[8] < 2^8(2 + 2^s), \tag{17}$$

where s is the number of spacer bits. Therefore, if the current byte is an X'FF', the upper bound on b, the next complete byte in the code register, is the sum of $C[0]$ and $A[0]$ after eight renormalizations:

$$b < 2^8(2 + 2^s)/2^{s+1}. \tag{18}$$

In this equation the numerator is the sum of the C and A registers multiplied by the renormalization factor; the denominator scales the value to the position in the C register where the byte is removed. Note that when the current byte is an X'FF', the register bit sequence from which the new byte is taken is shifted by one bit position to include the carry bit. For $s = 4$ (the convention chosen for the Q-Coder), this relationship shows that any 2-byte pattern in the range X'FF90' to X'FFFF' is illegal. Furthermore, any byte with a stuff bit cannot be X'FF'.

If the current byte is not an X'FF' and does not contain a stuff bit (that is, does not follow X'FF'),

$$b < 2^8(2 + 2^s)/2^s. \tag{19}$$

If the current byte does contain a stuff bit, the renormalization shift count for the next byte is reduced to 7. The trailing n bit left in the code register acts like an additional spacer bit, giving

$$b < 2^8(2 + 2^{s+1})/2^s. \tag{20}$$

Equations (19) and (20) prove that for spacer-bit counts greater than zero, a carry can never propagate more than one bit beyond the completed byte in the code register.

6. Summary

In the Q-Coder, P/Q symbol ordering is used and the convention is adopted that the code string points to the bottom of the current interval. However, compatible, optimal software and hardware implementations of binary arithmetic encoders and decoders can be achieved using either symbol-ordering convention and either code-string convention. Even though optimal implementation requires that the code string point to the bottom of the interval for one coder and the top of the interval for the other, the code strings can be made identical. Identity of the code strings can be guaranteed even when fixed-precision arithmetic is used in the coding process. In addition, a code string can be translated from one symbol-ordering convention to another by inverting it. The inversion can be done using fixed-precision arithmetic in either the encoder or the decoder. An analysis of the role of spacer bits in limiting carry propagation provides a rigorous bound for escape codes from the arithmetic code string.

Acknowledgments

This paper is part of a larger collaborative effort between IBM researchers at Almaden and Yorktown Heights to develop an arithmetic coding implementation which provides excellent compression and which is structured for efficiency in both software and hardware. We have benefited greatly from the continuing interaction with G. Langdon, J. Rissanen, R. Arps, and R. Pasco of the Almaden facility. We have also benefited from interactions with C. Gonzales and G. Goertzel at the Yorktown facility, and from the

continuing support and encouragement of our manager, K. Pennington.

References

1. W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr., and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.* **32,** 717 (1988, this issue).
2. W. B. Pennebaker and J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* **32,** 737 (1988, this issue).
3. J. L. Mitchell and W. B. Pennebaker, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* **32,** 753 (1988, this issue).
4. R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM J. Res. Develop.* **32,** 775 (1988, this issue).
5. W. B. Pennebaker, J. L. Mitchell, C. A. Gonzales, and C. F. Touchton, "Predictive Image Coder/Decoder Using Adaptive Binary Arithmetic Coding," *Research Report RC-13423*, IBM T. J. Watson Research Center, Yorktown Heights, NY, January 1988.
6. G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.* **28,** 135 (1984).
7. P. Elias, in N. Abramson, *Information Theory and Coding*, McGraw-Hill Book Co., Inc., New York, 1963.
8. J. Rissanen and G. G. Langdon, "Universal Modeling and Coding," *IEEE Trans. Info. Theory* **IT-27,** 12 (1981).
9. G. G. Langdon and J. J. Rissanen, "Compression of Black–White Images with Arithmetic Coding," *IEEE Trans. Commun.* **COM-29,** 858 (1981).
10. G. G. Langdon and J. J. Rissanen, "A Simple General Binary Source Code," *IEEE Trans. Info. Theory* **IT-28,** 800 (1982).
11. G. Goertzel and J. L. Mitchell, "Symmetrical Optimized Adaptive Data Compression/Transfer/Decompression System," U.S. Patent 4,633,490, December 30, 1986.
12. R. B. Arps, J. M. Cheng, and G. G. Langdon, "Control Character Insertion into Arithmetically Encoded Strings," *IBM Tech. Disclosure Bull.* **25,** 2051 (1982).

Received August 11, 1987; accepted for publication July 27, 1988

Joan L. Mitchell *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Mitchell graduated from Stanford University with a B.S. in physics in 1969. She received her M.S. and Ph.D. degrees in physics from the University of Illinois at Champaign-Urbana in 1971 and 1974, respectively. She joined the Exploratory Printing Technologies group at the IBM T. J. Watson Research Center immediately after completing her Ph.D., and since 1976 has worked in the field of data compression. Dr. Mitchell received IBM Outstanding Innovation Awards for two-dimensional data compression in 1978, for teleconferencing in 1982, and for the Image View Facility and resistive ribbon thermal transfer printing technology in 1985. She is co-inventor on fifteen patents.

William B. Pennebaker *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Pennebaker is a Research Staff Member at the IBM T. J. Watson Research Center and currently manages a group doing research in areas related to image processing and compression. He joined IBM's Research Division in 1962 and has worked in areas related to low-temperature physics, thin films, display technology, printing technology, and image processing. Dr. Pennebaker has received an Outstanding Contribution Award for work on strontium titanate films, an Outstanding Invention Award for work on silicon nitride films, and an Outstanding Innovation Award for work on image processing and compression. He has received ten IBM Invention Achievement Awards. Dr. Pennebaker received his B.S. in engineering physics from Lehigh University, Bethlehem, Pennsylvania, in 1957, and his Ph.D. in physics from Rutgers University, New Brunswick, New Jersey, in 1962. He is a member of the American Association for the Advancement of Science, the American Institute of Physics, the Institute of Electrical and Electronics Engineers, and the Society for Information Display.