9

JPEG BINARY ARITHMETIC CODING

This chapter describes the binary arithmetic-coding procedures that are used by both JPEG and JBIG. Our treatment here supplements the sections in the JPEG DIS (Annex D), and readers may find it instructive to refer occasionally to the DIS as they read this chapter. We assume the reader has read Chapter 8, and is already familiar with the basic concept of recursive interval subdivision, which is at the heart of arithmetic coding.

The JPEG/JBIG arithmetic-coding algorithm is also known as the QM-coder, a name that was coined to describe the coder's technical ancestry. The QM-coder is a lineal descendent of the Q-coder,[40-44] but is significantly enhanced by improvements in the interval subdivision[45] and probability estimation.[46]

The QM-coder also derives many technical elements from arithmetic coders such as the skew coder,[47] other early work by Langdon and Rissanen,[48] a paper by Jones,[49] and the first papers on arithmetic coding—by Rissanen,[50] Pasco,[51] Rubin,[52] and Rissanen and Langdon.[53] A discussion of this early work can be found in Langdon's tutorial on arithmetic coding.[54]

The QM-coder is the result of an effort by JPEG and JBIG to combine the best features of these various arithmetic coders. It uses the basic cod-

ing conventions and interval subdivision of the skew coder and Q-coder and the renormalization-driven probability estimation of the Q-coder. However, the approximation to the multiplication used for interval subdivision in the Q-coder is enhanced by the incorporation of conditional exchange,[45] and the state machine for the probability estimation is improved by the incorporation of rapid initial learning based on Bayesian estimation principles.[46]*

The QM-coder resolves the carry-over in the encoder in a manner similar to the technique of Jones[49] (see also Langdon[58]), thereby avoiding the need for the more complex bit-stuffing technique used in the Q-coder and skew coder.† The coding conventions and symbol ordering in the QM-coder are defined for optimal software performance, but, as with the Q-coder,[41] alternative coding conventions and inverted symbol ordering are easily implemented.

The QM-coder is a binary arithmetic coder, which means that there are only two symbols, 1 and 0, to code. When a descriptor can have many values, it is decomposed into a tree of binary decisions and each binary decision is assigned a particular context-index S, as described in Chapter 12. In that chapter we also list a number of changes that must be made to the idealized version of arithmetic coding discussed there to make it a practical coding procedure. These are:

1. Conventions for symbol ordering and positioning of the code stream in the interval must be defined.
2. An approximation is defined for the multiplication that is needed to scale the probability interval. To mitigate some of the impact on coding efficiency caused by this approximation, a symbol assignment interchange known as conditional exchange is defined.
3. A renormalization of the probability interval is defined so that coding can be done with fixed-precision integer arithmetic.
4. An adaptive probability estimation technique is integrated into the coding procedure.
5. A method for resolving carry-over in the encoder is defined that limits carry propagation when the code stream position in the interval is modified.
6. Zero bytes are stuffed after each X'FF' in the code stream in order to avoid accidental creation of markers.

* The concept of conditional exchange is related to the "over-half processing" of MELCODE,[55] a coding system that was originally a form of Golomb coding,[56] and that was extended later to a form of arithmetic coding. Bayesian estimation in arithmetic coders was first used by Chamzas and Duttweiler in their "Minimax" arithmetic coder, a variant of the Cleary, Witten, and Neal arithmetic coder.[57]

† The use of this method for resolving carry-over in the QM-coder was suggested by Chamzas.[59]

7. A termination procedure is defined that guarantees that the marker at the end of the entropy-coded segment is intercepted and interpreted before the decoding of the segment is complete.[60]

These topics will be discussed in detail in this chapter as we develop the flowcharts of the arithmetic-coding procedures. Although we cover the procedures for probability estimation here, we leave a thorough discussion of the principles of this estimation technique for Chapter 14. We also defer some of the practical details about arithmetic-coding implementations to Chapter 13.

9.1 The QM-encoder ◑

As described in Chapter 12, the QM-encoder uses four procedures, Initenc, Code_0(*S*), Code_1(*S*) and Flush. These four procedures initialize the encoder, code a 0 decision, code a 1 decision and empty the encoder register at the end of the entropy-coded segment. In this section we shall first discuss the principles and coding conventions for the Code_0 and Code_1 procedures; then we shall discuss the initialization and termination procedures.

9.1.1 Symbol ordering and code stream conventions ◑

When the QM-coder codes a 1 or 0 decision, it does not directly assign intervals to these symbols. Rather, it assigns intervals to the more probable symbol (MPS) and less probable symbol (LPS), and orders the symbols on the number line such that the LPS subinterval is above the MPS subinterval. If the interval is A and the LPS probability estimate is Qe, the MPS probability estimate should ideally be $(1 - Qe)$. The respective subintervals are then $A \times Qe$ and $A(1 - Qe)$. This ideal subdivision and symbol ordering are shown in Figure 9-1.

Ideally, the code stream C needs only to point somewhere within the current interval, but for simplicity the QM-coder uses the convention that the code stream points to the bottom of the current interval. Then, we need only to add to the code stream when our coding decision requires us to select the upper (LPS) subinterval. Note that if the bottom of the interval is included within the interval, the top must be excluded.

If we follow this ideal scheme, coding a symbol changes the interval and code stream as follows:

After MPS:

C is unchanged
$A = A(1 - Qe) = A - A \times Qe$

After LPS:

$C = C + A(1 - Qe) = C + A - A \times Qe$
$A = A \times Qe$

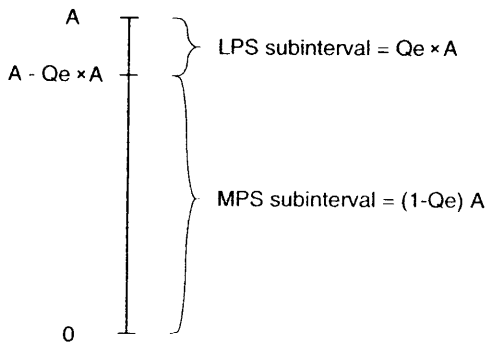

Figure 9-1. Illustration of symbol ordering and ideal interval subdivision. The LPS probability is Qe, and therefore the MPS probability is $(1 - Qe)$.

Note that the procedures for subdividing the interval and moving the code stream waste no code space (the MPS and LPS probabilities sum to 1); note also that the definition of the code stream convention allows no ambiguity about which interval the code stream points to.

9.1.2 Renormalization ◑

There are two problems with this ideal procedure as sketched above. First, we need a potentially unbounded precision for A, and second, we need a multiplication, $A \times Qe$, in the interval subdivision—a potentially costly operation in either hardware or software. These two problems are resolved by periodic renormalization of A, and a matching renormalization of C. The renormalization is done by doubling A each time it drops below some convenient minimum value, which we shall shortly require to be 0.75.[6] Each time A is doubled, C is also doubled in order to maintain its identity as a pointer to the subinterval.

The logic behind this renormalization is illustrated by the following example. Suppose we start our interval at 1.0 (the interval assigned to all possible sequences of symbols) and we code a sequence of LPS of probability 0.5. Then, we see the following behavior for A and C.

No. symbols coded	Decimal notation (Qe = 0.5)		Binary notation (Qe = 0.1)	
	A	C	A	C
0	1.0	0	1.0	0
1	0.5→ 1	0.5→ 1	0.1→ 1	0.1→ 1
2	0.5→ 1	1.5→ 3	0.1→ 1	1.1→ 11
3	0.5→ 1	3.5→ 7	0.1→ 1	11.1→ 111
4	0.5→ 1	7.5→ 15	0.1→ 1	111.1→ 1111

With each renormalization (indicated by →) we add a bit to the integer part of the code stream, leaving the fractional part for the coding of future symbols. Note that if we were to code an MPS, we would not add 0.5 to C, and would therefore get a zero bit after renormalization.

9.1.3 Approximating the multiplication ◑

The minimum value of 0.75 for the interval is motivated by the need to replace the multiplication by a simple approximation that requires A to be of order 1, i.e., in the range $1.5 > A \geq 0.75$. Then, $A \times Qe \approx Qe$ and we can use:

After MPS:

C is unchanged
$A = A(1 - Qe) \approx A - Qe$

After LPS:

$C = C + A(1 - Qe) \approx C + A - Qe$
$A \approx Qe$

With this approximation the multiplications are replaced by simple addition and subtraction.

9.1.4 Integer representation ◑

Renormalization allows the QM-coder to use fixed-precision integer arithmetic in the coding operations. For the representation chosen for the QM-coder, X'10000' is defined as the exclusive upper bound for the interval A (1.5), and X'8000' is defined as the inclusive lower bound for A (0.75). The encoder therefore is as follows:

$X'8000' \leq A < X'10000'$

After MPS:

```
C is unchanged          /* C points to base of MPS subinterval */
A = A - Qe              /* Calculate MPS subinterval           */
if A < X'8000'          /* If renormalization needed           */
   renormalize A and C
end
```

After LPS:

```
C = C + A - Qe          /* Point C at base of LPS subinterval  */
A = Qe                  /* Set interval to LPS subinterval     */
renormalize A and C     /* Renormalization always needed       */
```

The relationship between integer and decimal values we have been using is dictated by the approximation to the multiplication, and would suggest that a value of 1.0 be represented by (4/3)(X'8000') or X'AAAA'. However, this decimal equivalency is used only to provide an interpretation of the integer values. What counts is the accuracy of the interval subdivision, and this is determined by the relative values of Qe and A. Assuming Qe is matched to the statistics of the decision sequence, the average value of A defines the integer equivalent of decimal 1.0. Experimentally, we find that the average value of A is X'B55A' for JPEG compression, and this is only slightly higher than the optimal value of X'AAAA'.* As a convenience, we shall continue through the next section to use 0.75 as the decimal equivalent of X'8000'. However, later in this chapter we shall use the measured average value in computing the decimal equivalents of the integer Qe values in the state machine.

9.1.5 Conditional exchange ◑

One of the problems with the approximation to the multiplication is that when Qe is of order 0.5 (X'5555'), the size of the subinterval allocated to the MPS can be as small as 0.25 (X'2AAB'). To avoid this interval size inversion, the assignment of LPS and MPS to the two intervals is interchanged whenever the LPS subinterval becomes larger than the MPS subinterval. This is known as "conditional exchange," and the term "conditional" comes from the fact that the interval reassignment is only carried out when the LPS probability occupies more than half of the total interval available. A sketch of two possible cases, one without, the other with conditional exchange, is shown in Figure 9-2. Note that when conditional exchange occurs, $0.5 \geq Qe > (A - Qe)$. Both subintervals are clearly less than 0.75 (X'8000'), and renormalization must occur. Consequently, the test for conditional exchange is performed only after the encoder has determined that a renormalization is needed. The encoder is therefore:

* A slightly different average, X'B893', is given in the JBIG DIS.

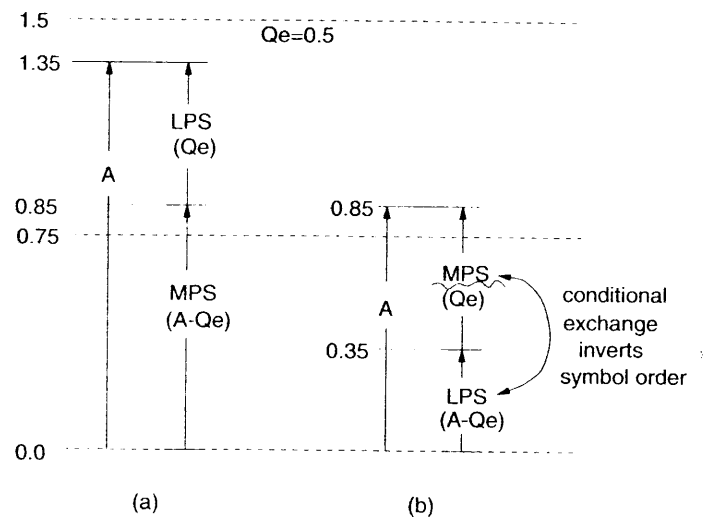

Figure 9-2. Illustration of interval subdivision. (a) without and (b) with conditional exchange.

After MPS:

```
C is unchanged
A = A - Qe
if A < X'8000'          /* Calculate MPS subinterval           */
                        /* If renormalization needed           */
   if A < Qe            /* If interval sizes inverted          */
      C = C + A         /* Point to LPS subinterval base       */
      A = Qe            /* Set interval to LPS subinterval     */
   end
   renormalize A and C
end
```

After LPS:

```
A = A - Qe              /* Calculate MPS subinterval           */
if A ≥ Qe               /* If interval sizes not inverted      */
   C = C + A            /* Point to LPS subinterval base       */
   A = Qe               /* Set interval to LPS subinterval     */
end
renormalize A and C
```

9.1.6 Probability estimation ◑

Adaptive probability estimation has been used in a number of arithmetic coders.[47, 48, 62] The probability estimation used in the QM-coder is based on the renormalization-driven estimation developed for the Q-coder.[42] However, it has been enhanced by a better estimator state machine structure based on the Bayesian estimation principles incorporated in the Minimax coder.[46]*

The estimation process is based on a form of approximate counting[64] in which the interval register renormalization is used to estimate the MPS and LPS symbol counts. Whenever either MPS or LPS renormalization occurs, the count of MPS is cleared (effectively), and a new estimate is obtained from a table that provides a bigger *Qe* value when the LPS renormalization occurs and a smaller *Qe* value when the MPS renormalization occurs.

Although the system is stochastic, the estimation state machine naturally tends to move toward the correct estimate. If the *Qe* value is too large, MPS renormalization is more probable than LPS renormalization and the *Qe* value is likely to decrease. Conversely, if the *Qe* value is too small, MPS renormalization is less probable than LPS renormalization and the *Qe* value is likely to increase. If the sense of the MPS is wrong, *Qe* will increase until it reaches approximately 0.5, at which point the sense of MPS and LPS are interchanged. Note that the estimation is done for whichever context is being coded at the time that the renormalization occurs.

The detailed analysis of the estimation technique will be described in Chapter 14. What is important here is to understand the implementation of the estimation procedure. Table D.2 of the JPEG DIS contains the complete estimator state machine. For each index in the table there are four columns: *Qe*, *Next_Index_LPS*, *Next_Index_MPS*, and *Switch_MPS*. *Qe* is the LPS probability estimate. *Next_Index_LPS* is the index to the new probability estimate following an LPS renormalization, and *Next_Index_MPS* is the index to the new probability estimate following an MPS renormalization. *Switch_MPS* controls the changing of the sense of the MPS decision; if *Switch_MPS* is not zero and the LPS renormalization is required, the sense of the MPS must be changed before moving to *Next_Index_LPS*. The steps needed to reach a new probability estimate after the MPS renormalization (see Figure D.5 of the JPEG DIS) are:

```
I = Index(S)                  /* Current index for context S          */
I = Next_Index_MPS(I)         /* New index for context S              */
Index(S) = I                  /* Save this index at context S         */
Qe(S) = Qe_Value(I)           /* New probability estimate for context S*/
```

* This merging of renormalization-driven estimation with the Bayesian estimation combines the coding performance of the Minimax coder with the computational simplicity of the Q-coder.[63]

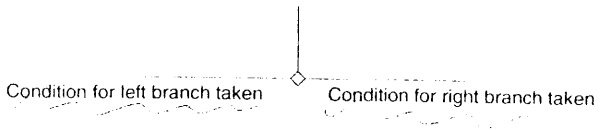

Figure 9-3. Decision tree conventions

For the LPS renormalization path (see Figure D.6 of the JPEG DIS):

```
I = Index(S)                  /* Current index for context S          */
if Switch_MPS(I) = 1          /* If changing MPS sense for context S  */
   MPS(S) = 1 - MPS(S)        /* Exchange MPS sense (1 to 0, 0 to 1)  */
end
I = Next_Index_LPS(I)         /* New index for context S              */
Index(S) = I                  /* Save this index at context S         */
Qe(S) = Qe_Value(I)           /* New probability estimate for context S*/
```

On the MPS path the state machine index for the current context is used to look up a new index, *Next_Index_MPS*, which is saved in the context store. Optionally, the *Qe* value for that index can also be saved in the context store, eliminating the need for an indirect storage access to get the value of *Qe*. The LPS path procedure is similar, except that *Switch_MPS* must be checked to see whether the sense of the MPS must be changed before getting the new estimate. A different organization of the estimation state machine that avoids explicit testing of the *Switch_MPS* bit will be described in Chapter 13.

9.1.7 Compact decision tree notation ◑

The compact decision tree notation illustrated in Figure 9-3 is used for the flowcharts in this chapter. An extended version of this notation is used in Chapter 12 and Chapter 13.

9.1.8 Decision tree for encoding ◑

The decision tree for the Code *D*(*S*) coding procedure is shown in Figure 9-4. If *D* = 0, this is the Code_0(*S*) procedure of the JPEG DIS, whereas if *D* = 1, it is the Code_1(*S*) procedure. Note that very few operations are required on the most likely path in which the MPS is coded without renormalization.

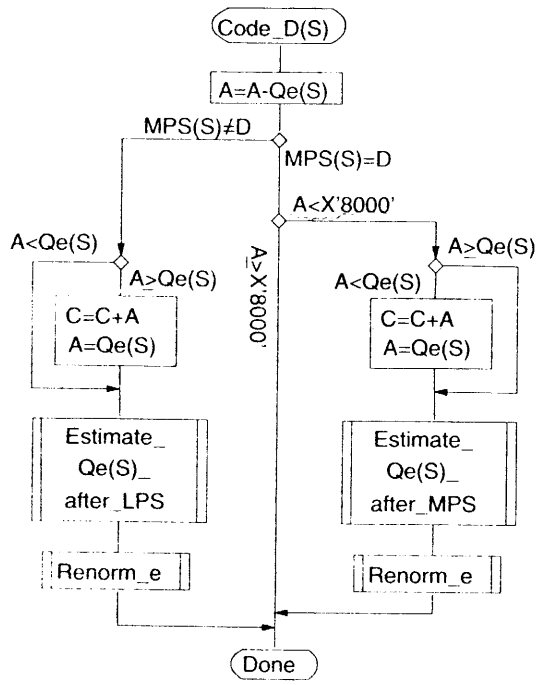

Figure 9-4. Encoder decision tree for coding a decision *D*

9.1.9 Output procedure for the encoder ◑

Output bits are created by the renormalization procedure in Figure 9-5 (see Figure D.7 of the JPEG DIS). A counter, *CT*, counts the number of times *A* and *C* are doubled by the shift-left-logical (SLL) operation, and thus counts the number of bits produced. When *CT* is zero, Byte_out is called to transfer a byte of compressed data from the code register to the code buffer. Byte_out, shown in Figure 9-6, contains the logic to resolve carry-overs so that they cannot propagate through more than one byte of data in the output buffer.

The carry-over control in Byte_out works on the principle that only X'FF' bytes can propagate a carry and therefore, if they are produced, they are stacked—i.e. counted—until the carry can be resolved. The carry is resolved either when a carry-over explicitly occurs or when the next output byte is less than X'FF'. At that point stacked data can be transferred to the compressed data buffer.

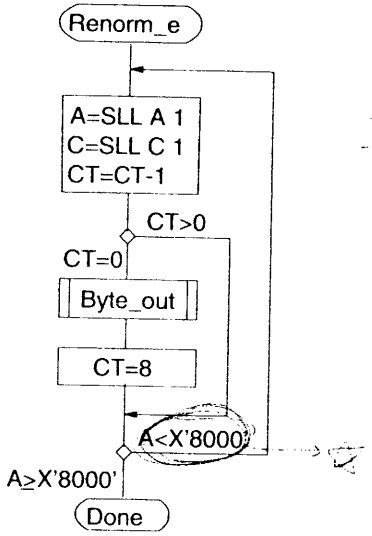

Figure 9-5. Renormalization in the encoder

The code register design determines how the output byte is removed. For the flowcharts given in the DIS, the register has the following bit assignment:

C register:	0000	cbbb	bbbb	bsss	xxxx	xxxx	xxxx	xxxx
A register:					0	aaaa	aaaa	aaaa	aaaa

The "x" bits are the fractional code register bits, "s" denotes three spacer bits, "b" bits are the bits of the byte that will be removed from the register after each 8 renormalizations, "c" is a carry bit that occasionally gets set because of carry-over in the register arithmetic, and "a" is an interval register bit. Normally bit 16 of the A register is zero, as shown—however, see the section on initialization below.

Spacer bits delay the output, thereby giving more chance for carries to be resolved before the byte of data is removed from the register. Note, however, that there is no way for carry-over to exceed the capacity of the register as defined here. The bounds on carry-over are dictated by the maximum possible sum of *C* and *A*. Immediately after a byte has been removed from the code register, *C* is less than X'80000' and *A* is less than X'FFFF'. The sum is therefore less than X'8FFFF', and after eight

renormalizations it is less than X'8FFFF00'. Therefore, the upper four bits of the *C* register can never be other than zero.

Referring to Figure 9-6, the branch to the left at the first test is taken if the carry bit is set. The carry bit must be added to the data written in earlier calls to Byte_out, including any stacked X'FF' bytes. The addition of the carry converts the stacked X'FF's to zeros, and the carry then propagates to the last byte *B* actually written to the code buffer. The carry bit is therefore added to the byte in the code buffer (which is always less than X'FF'), and if that addition produces X'FF', a zero byte is inserted (stuffed) immediately following. Any stacked bytes (now zero) are then placed in the compressed data buffer, followed by the new byte of compressed data. Note that when the carry occurs, the byte defined by 'bbbbbbbb' in the code register is X'1E' or less, and therefore can be written directly to the code buffer without testing for the X'FF' value.*

Again referring to Figure 9-6, the branch down from the first test is taken if the carry bit is not set. If the new byte is X'FF', the second branch down must be taken; the byte must be stacked until future carries can no longer propagate through it. The stack count *SC* is incremented, but no further action is taken except to clear the high order bits of the *C*-register.

If the new byte is less than X'FF', the branch to the right in Figure 9-6 is taken. At this point we can guarantee than no carry will propagate through the stacked X'FF' bytes, and they are therefore placed in the compressed data buffer with stuffed zero bytes appended. The new byte of data is then written immediately following.

If the compressed data is purely random, the probability of the stack count exceeding 3 or 4 is quite small. However, there are special conditions in which repeating patterns of coding decisions and contexts can cause *SC* to reach much larger values. The only rigorous upper bound on *SC* is the size of the compressed data set itself, and it would be dangerous to assume that *SC* can never approach this bound. We have observed stack counts of more than 80 in JBIG sequential compression of binary halftone images, and there is some reason to suspect that a worst-case stack count exceeding a 16-bit counter capacity is possible. However, a 32-bit counter represents such a vast amount of compressed data that it should be able to handle any situation that might reasonably be encountered in either JPEG or JBIG.

9.1.10 Initialization of the QM-encoder

The QM-encoder is initialized with *A* = X'10000' and *C* = 0. The starting value for *A* is the exclusive upper bound of the interval, and code values from X'0000000....00' to X'FFFFFFF...FF' are possible (ignoring stuffed

* This would not be true if there were no spacer bits. Without spacer bits the maximum value for both *C* and *A* is X'FFFF' when starting a new byte. Therefore, the sum is X'1FFFF', which after shifting by eight bits becomes X'1FFFF00'. The carry bit is set, and the next eight bits can be X'FF'.

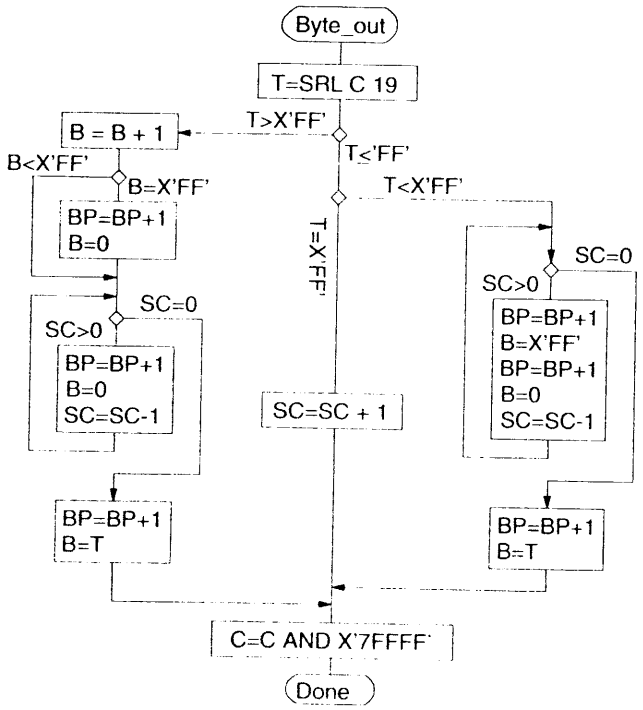

Figure 9-6. Outputting a byte in the encoder

zero bytes). Note that in 16-bit implementations *A* can be set to 0, and if normal underflow in the arithmetic can occur, it will have the correct value after the first symbol is coded. The Byte_out renormalization shift count, *CT*, is set to 11, reflecting the fact that the first byte of data will be complete when both the output byte and three spacer bits have valid data.

The pointer to the compressed data is initialized to point to a position one byte before the first byte of compressed data. Byte_out will increment this pointer before writing the first byte to the buffer.

The statistics areas are initialized to an estimation table index of zero. This starts each context in the correct rapid initial learning or "fast attack" state. The *Qe*(*S*) values are initialized to X'5A1D', the probability value for an index of zero (see Table D.2 of the JPEG DIS). *MPS*(*S*) values are set to zero.

There is an interesting question concerning the interpretation that should be given to the starting initial interval. If we use the relationship that decimal 1.0 is equivalent to X'B55A', we find that X'10000' is

equivalent to decimal 1.41. Yet clearly, we cannot have a probability interval of 1.41. The difficulty here lies in our attempting to assign rigorously a decimal equivalent to each integer value. These assignments have meaning on the average, but the initial startup condition is a special situation in which we can assign meaning relative only to the starting *Qe*. Since that first *Qe* value is X'5A1D', our choosing A=X'10000' simply means that in using a starting value for A that permits us the full range of compressed data output, we must use a less than optimal first interval subdivision. However, because we do not know what that first *Qe* should be anyhow, this less than optimal subdivision is of no practical consequence.

9.1.11 Termination of the entropy-coded segment ◑

After the last symbol has been coded, the information still in the code register must be transferred to the compressed data buffer. This is done by the Flush routine shown in Figure 9-7.

In order to understand the operation of Flush, we must first understand a convention adopted for the decoder. If the decoder encounters a marker, it must continue to supply zero bytes to the decoding procedures until the correct number of MCU have been decoded and decoding is complete. The strategy adopted in Flush, therefore, is to create as many trailing zero bits in the code register as possible and to write only two bytes to the compressed data buffer. The rest of the bits are guaranteed to be zero, and therefore will be regenerated in the decoder when it encounters the terminating marker. Furthermore, because the final zero bytes that were discarded are needed in order to have enough precision to decode the last few symbols, the decoder will try to read them and will therefore encounter the terminating marker before it completes decoding. This terminating marker can therefore provide information needed to terminate the decoding process.

Then, optionally, the encoder may discard as many more trailing zero bytes as possible from the code buffer, nibbling away until it encounters non-zero data, a stuffed zero, or the start of the buffer.* These trailing zeros can be discarded because the decoder will regenerate them when it encounters the marker at the end of the segment. This can be done in a number of ways, and the reader is referred to the JPEG DIS Figure D.15 for one possible procedure.

9.2 The QM-decoder ◑

An arithmetic decoder decodes a binary decision by determining which subinterval is pointed to by the code stream. When fixed-precision integer arithmetic is used, however, the *A* register is not large enough to contain more than the current subinterval available for decoding new symbols. The code stream must therefore be kept relative to this subinterval by subtracting from it each subinterval that was added by the encoder. Then,

* This nibbling away of trailing zeros is nicknamed "Pacman" because of the similarity to the popular computer game of that name.

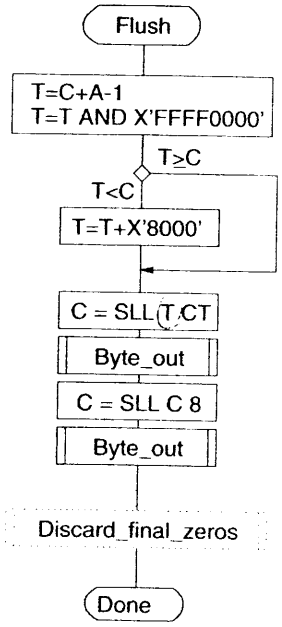

Figure 9-7. Termination of the entropy-coded segment

the code stream becomes a pointer into the current interval relative to the base of the current interval and is guaranteed to have a value within the current interval.

9.2.1 Register bit assignment in the decoder ◑

The decoder makes use of two registers, *A* and *C*. *A* is the interval register and serves exactly the same function as in the encoder. It also must always have exactly the same value before decoding a given symbol that the encoder had before encoding that symbol, and therefore must undergo the same arithmetic and renormalization procedures. *C* is segmented into two parts, *Cx* and *C-low*. *Cx* is the portion of the code register containing the offset or pointer to the subinterval and *C-low* contains up to eight bits of new data. Each time *A* is renormalized, *Cx* and *C-low* are shifted as a pair such that the MSB of *C-low* becomes the LSB of *Cx*. For 16-bit architectures where *Cx* and *C-low* are separate registers, the bit assignments for *A*, *Cx* and *C-low* are as follows:

	MSB			LSB
A	aaaa	aaaa	aaaa	aaaa
Cx	cccc	cccc	cccc	cccc
C-low	bbbb	bbbb	0000	0000

In 32-bit architectures *Cx* and *C-low* can be implemented as one 32-bit register:

	Cx				*C-low*			
Cx:*C-low*	cccc	cccc	cccc	cccc	bbbb	bbbb	0000	0000
A	aaaa	aaaa	aaaa	aaaa	0000	0000	0000	0000

If *C* is a 32-bit register, *A* should also be a 32-bit register, in order to facilitate the comparisons between *Cx* and *A*. Note that the bits in *C-low* cannot affect the comparison of *A* and *Cx*, because there are no tests for equality.

9.2.2 Decision tree for decoding ◑

The decision tree for the Decode(*S*) procedure is shown in Figure 9-8. The code register is, by definition, a pointer to the subinterval relative to the base of the current interval. Therefore, by comparing *Cx* to the size of the lower subinterval, $A - Qe$, the first decision determines whether the upper or the lower subinterval has been decoded. If the upper subinterval is decoded, the encoder must have added the lower subinterval to the code stream; the decoder must therefore subtract that amount. The value left in the code register, *Cx*—the offset into the interval—thus remains a pointer to the subinterval for the symbols that have not yet been decoded.

Decoding the MPS or LPS subinterval does not yet completely determine which symbol was decoded. If the MPS subinterval is decoded and no renormalization is needed, the result is known. However, whenever renormalization is needed, the interval assignments might have undergone conditional exchange. The decoder must then test for conditional exchange and interchange interval assignments if needed. It must then do the estimation in a manner appropriate for the symbol decoded (i.e., exactly as the encoder did it) and renormalize. As in the encoder, very few operations are required on the path in which the MPS is decoded without renormalization, and this is the most likely path.

If renormalization is needed, both the code register and the interval register are shifted left until the most significant bit of *A* is again set. This is shown in Figure 9-9. The counter *CT* determines when *C-low* is empty and Byte_in must be called to insert a new byte of data in *C-low*.

9.2.3 Input procedure for the decoder ◑

The input procedure for the decoder is greatly simplified relative to the output procedure (Byte_out) of the encoder by the carry resolution in the

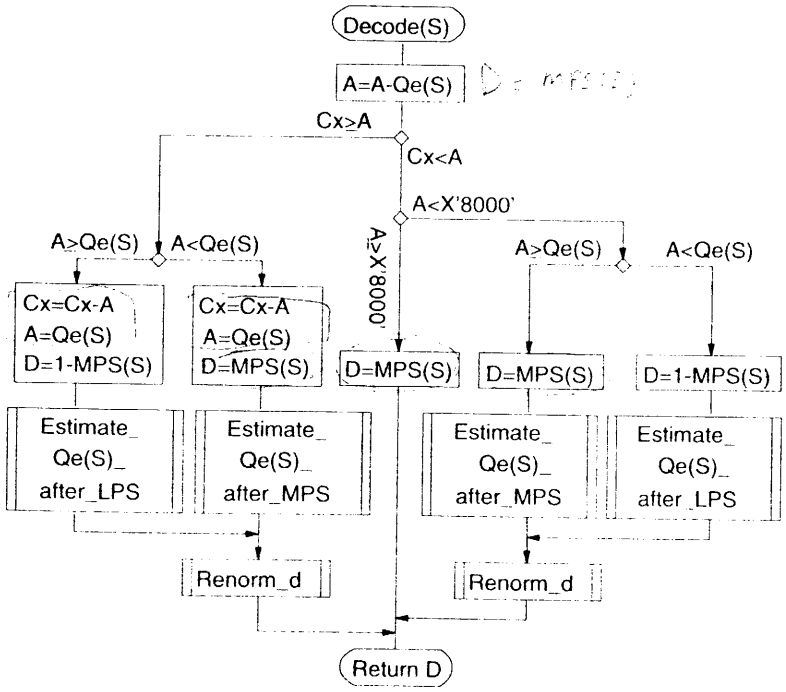

Figure 9-8. Decoder decision tree for decoding a decision *D*

encoder. As shown in the flowchart of Figure 9-10, the only complication is caused by the need to check for X'FF' bytes and to "unstuff" (i.e., discard) the zero byte that often follows that byte. As part of doing that, however, the decoder must check to make sure the byte following the X'FF' is actually zero. If the byte following an X'FF' is not zero, the marker code that terminates the entropy-coded segment has been encountered. The decoder must then, by definition, be provided with zero bytes until decoding is complete. Note that several zero bytes may be needed because of the "Pacman" termination allowed in the encoder.

Because the decoder may request as many zero bytes as it needs, decoding is not self-terminating. The decoder must determine by explicit testing when it has decoded the correct amount of data. Note that the decoder will always need at least one zero byte after encountering the marker. It therefore is guaranteed to intercept and interpret the JPEG DNL marker (which defines the number of lines in the image) in time to terminate decoding correctly.

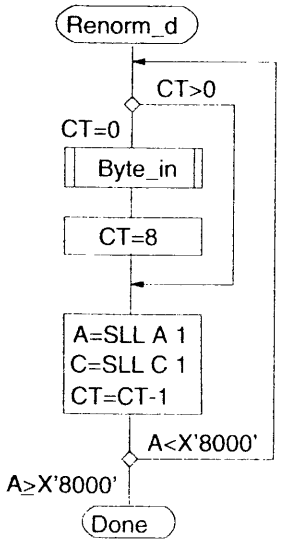

Figure 9-9. Decoder renormalization

9.2.4 Initialization of the decoder ◑

Initialization of the interval register is the same for encoder and decoder. As in the encoder, a 16-bit precision A register should be initialized to zero; the first decoding operation will cause it to underflow to the correct 16-bit value. The high-order part of the code register, Cx, is simply loaded with the first two bytes of compressed data and the count CT is cleared so the first renormalization will cause a new byte of data to be loaded into C-low.

9.3 More about the QM-coder ◑

In this chapter we have presented the basic structure of the QM-coder. Further details for the QM-coder that allow for efficient software and hardware are given in Chapter 13; Chapter 14 contains a discussion of renormalization-driven probability estimation.

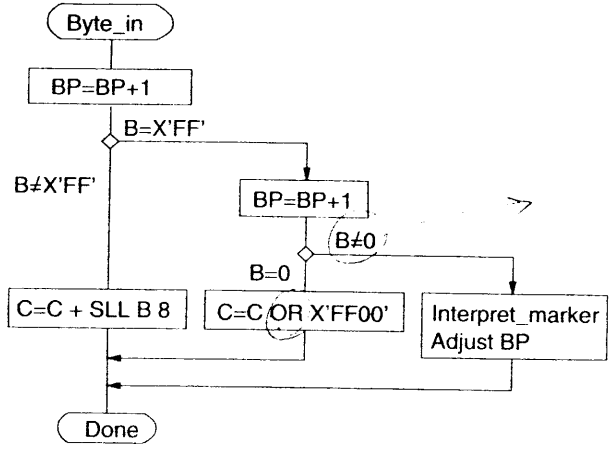

Figure 9-10. Decoder input procedure