JPEG2000 Arithmetic Encoding on the StarCore® SC140

Application Note

freescale™
semiconductor

Abstract and Contents

This application note describes how the advanced features of the StarCore® processor can be used to implement the arithmetic coding algorithm employed in the JPEG2000 image compression standard. Both C code and optimized assembler listings are presented, as well as a review of the basic principles of arithmetic encoding.

Appendix A
Arithmetic Encoder: C Code

Appendix B
Arithmetic Encoder: Assembly Code

Appendix C
Excerpts from FDIS

1   Introduction

This application note is the second in a series of notes that describe how the advanced features of the StarCore processor can be used to implement the algorithms associated with the image compression standard JPEG2000. The first application note in this series [1] which described the wavelet transform included a brief overview of the JPEG2000 standard. In this application note, arithmetic coding is explained in conjunction with the specific arithmetic encoder used in the JPEG2000 standard. StarCore implementation in both C code and optimized assembler are also provided.

Although the StarCore processor is a general-purpose DSP, it has many features that make it possible to perform image compression algorithms quickly and efficiently. The performance of StarCore in processing the JPEG2000 arithmetic encoder is discussed with results of performance tests being provided. Although the arithmetic encoder is primarily a sequential engine, it is shown that some of StarCore's parallel features can still be used to good effect.

2 Background Theory

This section presents a look at some of the theory behind symbol coding, including the well-known Huffman coder, general arithmetic coding, and binary arithmetic coding, which is employed by the JPEG2000 standard.

Arithmetic coding is a form of statistical coding, which compresses data by encoding more probable symbols with shorter code words than less probable symbols. The ideas behind compressing the information associated with a sequence of random variables in order to transmit them over a communication link can be traced back to Shannon's ground breaking 1948 paper [2], in which he defined the entropy of a random variable, X, with a discrete alphabet, e.g. $\{x_0, x_1, \ldots, x_n\}$ as:

$$H(X) = -\sum_{i=0}^{n} f_X(x_i)\log_2 f_X(x_i) \qquad \text{Eqn. 1}$$

where $f_X(x_i)$ is the probability that x_i occurs. The entropy of a random variable can be thought of as how much information it holds. As an extreme example, when it is known in advance that one particular symbol will be received, e.g. an a, then its probability is 1 and its entropy is 0 because it holds no new information. At the other extreme, if all the symbols in a particular alphabet are equally likely, then this will give maximum entropy because each new received symbol is totally unpredictable. Shannon also proved that for a stationary process, entropy is equal to the minimum average number of bits per symbol required to represent the information source. The significance of arithmetic coding is that it can be made to approach this theoretical limit.

As an introduction to arithmetic encoding, let us examine the simpler approach of Huffman coding, which is used in the baseline JPEG standard.

NOTE:

The following discussions assume that a perfect communication channel is present so that there are no errors associated with the code stream.

2.1 Huffman Coding

One way of coding symbols in an information stream is to allocate a unique code word for each symbol. One such coding scheme, Huffman coding, is a form of prefix coding, meaning that each prefix in a given set of code words is unique. This fact simplifies the decoding process because the decoder simply continues to receive binary digits until a new code word has been obtained.

2.1.1 The Huffman Coding Algorithm

The significance of the Huffman coder is that it uses an algorithm which ensures that

$$\sum_{i=0}^{n} f_X(x_i)l_i \qquad \text{Eqn. 2}$$

is minimized, where l_i is the length of the code word associated with x_i. A simple example can serve to explain the algorithm. Consider an alphabet which consists of four possible symbols, {c, d, e, o}, with respective probabilities of 1/2, 1/4, 1/8, and 1/8. The encoding algorithm, illustrated in Figure 1, consists of the following steps:

- Sort the symbols from least probable symbol to most probable symbol.

- Group the two least probable symbols together to form a root to the two separate symbols, which become the leaves.

- Iterate the first two steps until the symbols are arranged in a binary tree in which only two branches extend from each root.

- Assign '1' to one branch from each root, and '0' to the other branch to obtain the code words. In the example, the left-hand branches are all 1's, and the right-hand branches are 0's.
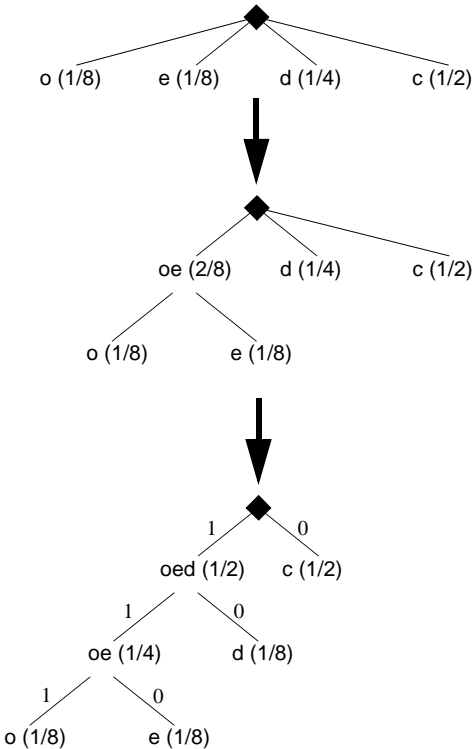
Figure 1. Binary Tree for Huffman Coding Example

It is evident in the final step that the code words are not unique; however, the length of the code word associated with each symbol is always the same.

Table 1 lists the code words obtained from the example in Figure 1, as well as the probability for each symbol expressed as a binary fraction. In binary fractions, 0.1 represent 2^{-1}, 0.01 represents 2^{-2}, etc. In other words, the exponent associated with the negative power of two is equal to the number of places after the binary point.

Table 1. Example of Huffman Coding

Symbol	Probability (in Binary)	Code Word
c	0.1	0
d	0.01	10
e	0.001	110
o	0.001	111

2.1.2 Limitations of Huffman Coding

In the above example, the code rate and the entropy are the same value, 1.75, because the probability for each symbol is equal to 2^{-l_i} where, as before, l_i is equal to the length of the ith symbol. In most cases, however, this is not true, and the limitations of Huffman coding become evident.

2.1.2.1 Minimum Obtainable Coding Rate

In Huffman coding, each code word must consist of at least one bit, which results in an average coding rate of at least one bit per symbol. However, the theoretical minimum or entropy of a system can be less than one bit per symbol, so in some cases it will be impossible for Huffman coding to attain the theoretical minimum.

2.1.2.2 Varying Probability

In the example, it has been assumed that the probabilities of the various symbols are fixed. In many cases, however, the probability varies, so the average number of bits per symbol in a Huffman coding scheme could be much larger than the entropy. For instance, if symbol e in the above example actually occurred with a probability of say 0.5, but still had 3 bits in its code word, the output bit stream would average 3 bits for half of the time rather than an eighth of the time as expected. Therefore, it would be preferable to be able to adapt the coding system to update the probabilities in real-time, which requires some sort of statistical estimation process. However, implementing the Huffman algorithm is a computationally intensive process, so modifying the code words adaptively could be prohibitively expensive from a computational point of view.

2.1.2.3 Practical Implementation

In practical implementations of the Huffman coder, a lookup table is usually used. The number of table entries must be 2^L, where L is the maximum length of the code words. In the above example, the maximum length is 3, so the lookup table must contain 8 entries to account for all unused combinations of 3 digits containing the relevant symbol with a shorter prefix. The bit stream in this case is examined 3 bits at a time, with these bits providing the addressing to the table. For example, the table entry for 100 would be c (treating the right hand bit as the next input, i.e. the code stream input is read from right to left). In this case, only one digit (the right-hand 0) is required to specify a particular code word, so only that digit is dropped, and the next bit in the bit stream is concatenated to the remaining 10 to form the next lookup table entry (which in this particular case, would be c again.) The problem here lies in the fact that because the Huffman algorithm does not limit the length of the code words, the table can grow very large.

These problems are partially addressed with the use of arithmetic coding.

2.2 Arithmetic Coding

For simplicity, the following discussion of arithmetic coding assumes a sequence of symbols in a stationary random process with independent elements. In this case, arithmetic coding produces a rate which approaches the entropy of the sequence, but also applies to correlated sequences if the process remains stationary. This discussion does not describe how arithmetic coding adapts to changing statistics in the bit stream because the adaptive nature of the coding is mainly determined by the coefficient bit modeler within the JPEG2000 standard. A good tutorial on arithmetic coding can be found in reference [3].

Whereas Huffman coding involves transmitting separate code words for each symbol in a sequence, arithmetic coding requires transmitting only the information needed to allow a decoder to determine the particular fractional interval between 0 and 1 to which the sequence is mapped. This information includes a fractional code word, C, which points to the lower bound of the interval, and the interval width, A, as illustrated in Figure 2.

Figure 2.   Mapping a Sequence to a Fractional Interval

NOTE:

The convention for expressing an interval employed in this discussion uses a bracket for an inclusive bound, and a parenthesis for an exclusive bound. Thus, [0,1) represents a range including zero but not including 1.

Each time a new symbol is received by the encoder, a smaller interval is chosen within the current interval. This interval represents the whole sequence of symbols up to and including the new symbol. The length of the updated interval is the probability associated with this sequence of symbols, as shown in Figure 3. The position of the probability interval associated with a particular symbol is always fixed in relation to the intervals of the other symbols. (Note that in arithmetic encoding, the position of the fractional interval is as important as the length.) In theory, this process can continue indefinitely until all symbols have been received. In practice, finite precision problems restrict the size of the shortest possible interval that can be represented. When the length of the probability interval falls below a certain minimum size, the interval must be renormalized to lengthen it above the minimum. The renormalization process is explained in more detail in Section 2.4.4 on page 18.
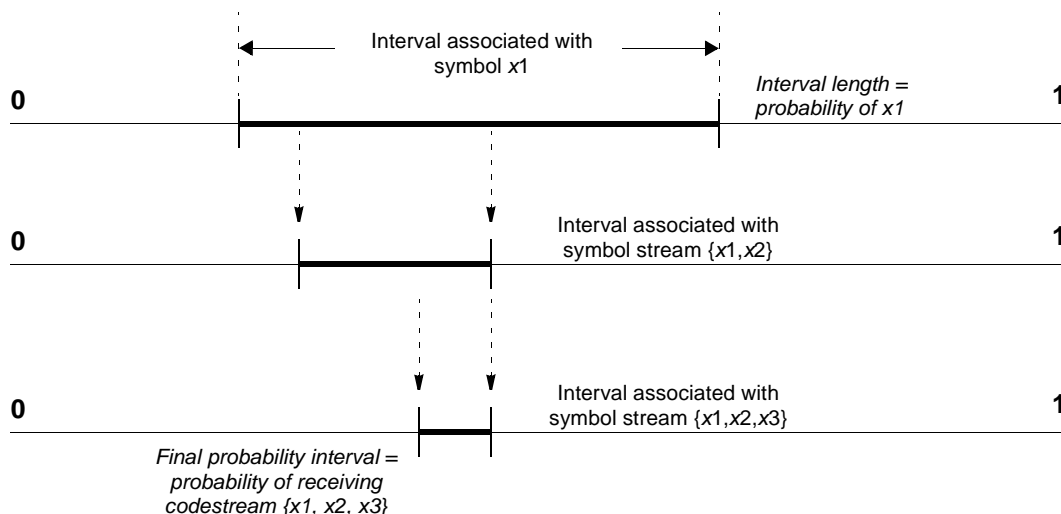
Figure 3.   Arithmetic Coding Process

Figure 4 illustrates how arithmetic coding is used to derive the fractional numbers which represent the position and width of the interval on [0,1) which can be decoded as the sequence `code`. This example uses the same symbol set and probabilities as in the previous example (see Table 1 on page 7). The top line in the diagram is the [0,1) interval. This line is subdivided into smaller intervals whose widths are directly

proportional to the probability associated with each symbol. For example, the symbol c is mapped to an interval of width 1/2 at position [3/8, 7/8).Note that for arithmetic coding the ordering of the intervals is arbitrary, whereas in Huffman coding the ordering depends on the probabilities of the symbols.
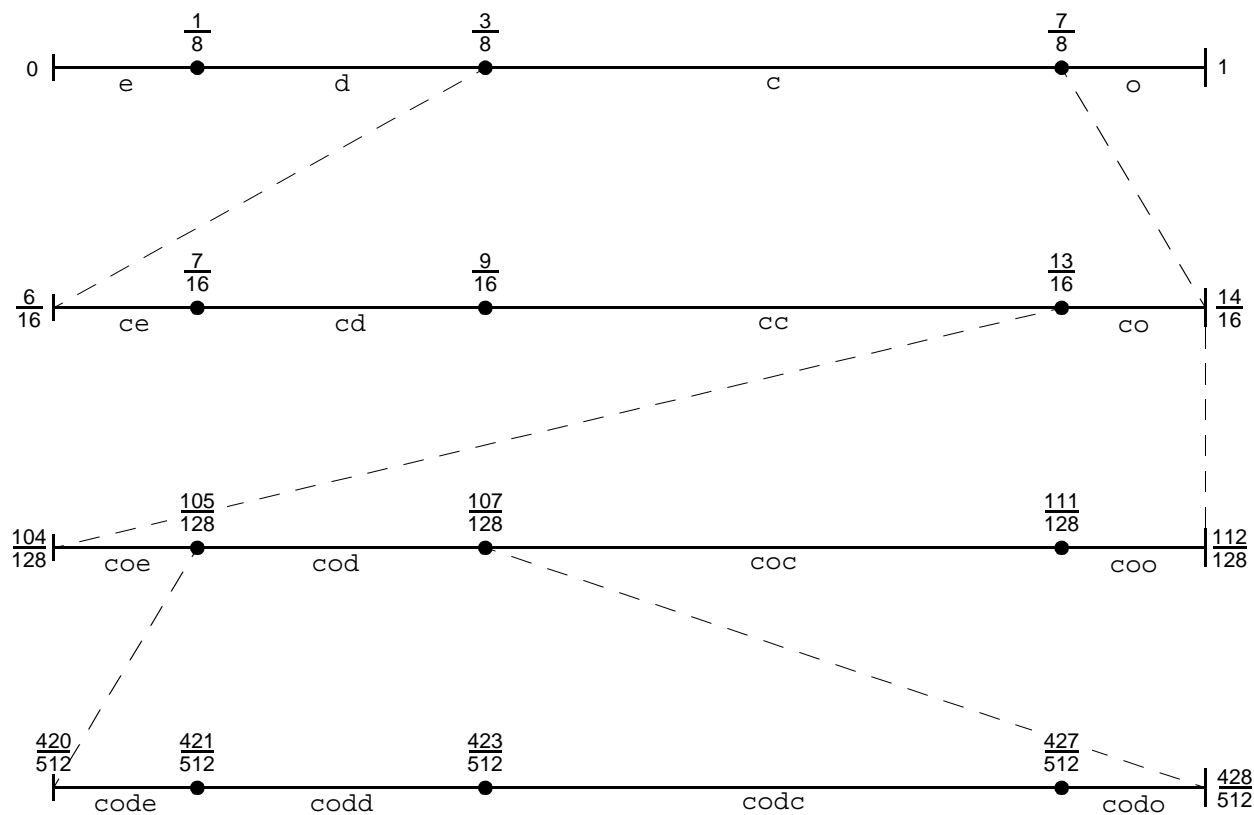
Figure 4. Arithmetic Encoding Example with Probabilities Restricted to 2^i

The algorithm for coding the sequence 'code' is as follows:

1. Initialize $A_0 = 1$ and $C_0 = 0$

2. Update code word, C, using the equation

$$C_n = C_{n-1} + (A_{n-1} \times P_X(x_n))$$

Eqn. 3

where $P_X(x_n)$ is the cumulative probability of x_n and is equal to

$$P_X(x_n) = \sum_{i=1}^{n-1} f_X(x_i)$$

Eqn. 4

where, as before, $f_X(x_i)$ is the probability of symbol x_i occurring.

For example, for the first symbol, c, $n = 1$ and $P_X(c) = f_X(e) + f_X(d)$ because symbols e and d lie to the left of c on the interval [0,1). Therefore, the cumulative probability entails adding up the probabilities associated with these two symbols. This cumulative probability is then multiplied by A_0, the current interval for the first symbol, which is 1.

For c, the first symbol in Figure 4, the code word becomes

$$C_1 = C_0 + (A_0 \times (f_X(e) + f_X(d)))$$
$$= 0 + (1 \times (1/4 + 1/8)) = 3/8$$

In binary fraction format,

$C_1 = 0.01 + 0.001 = 0.011$

3. Update the interval, A, using the equation

$$A_n = A_{n-1} \times f_X(x_n)$$

For the first symbol in the example, $f_X(c) = 1/2$, so

$A_1 = 1 \times 1/2 = 1/2$

In binary fraction format,

$A_1 = 1 \times 0.1 = 0.1$

Thus, the 'c' symbol is mapped to an interval of width 1/2 at position [3/8, 7/8), as shown in the top line of Figure 4.

4. Repeat steps 2 and 3 until the entire symbol sequence is mapped to an interval.

The procedure for mapping the rest of the symbols in the sequence code as shown in Figure 4 follows. Binary fractional representations are included to show how a code word is derived in a practical application.

For sequence co

$C_2 = C_1 + (A_1 \times (f_X(e) + f_X(d) + f_X(c)))$

$\quad = \frac{3}{8} + \left(\frac{1}{2} \times \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \right) \right) = \frac{13}{16}$

$A_2 = A_1 \times f_X(o)$

$\quad = \frac{1}{2} \times \frac{1}{8} = \frac{1}{16}$

In binary fraction format,

$C_2 = 0.011 + 0.0111 = 0.1101$

$A_2 = 0.1 \times 0.001 = 0.0001$

For sequence cod

$C_3 = C_2 + (A_2 \times (f_X(e))$

$\quad = \frac{13}{16} + \left(\frac{1}{16} \times \frac{1}{8} \right) = \frac{105}{128}$

$A_3 = A_2 \times f_X(d)$

$\quad = \frac{1}{16} \times \frac{1}{4} = \frac{1}{64}$

In binary fraction format,

$C_3 = 0.1011 + 0.0000001 = 0.1011001$

$A_3 = 0.0001 \times 0.01 = 0.000001$

For sequence code

$C_4 = C_3 + (A_3 \times (f_X(0))$

$\quad = \frac{105}{128} + \left(\frac{1}{64} \times (0) \right) = C_3$

$$A_4 = A_3 \times (f_X(e)$$
$$= \frac{1}{64} \times \frac{1}{8} = \frac{1}{512}$$

In binary fraction format,

$$C_4 = 0.1011001$$

$$A_4 = 0.000001 \times 0.01 = 0.00000001$$

Therefore, any fraction received at the decoder between the interval of 420/512 and 421/512 will represent the sequence of transmitted symbols `code`.

In general, it turns out that it is only necessary to transmit the most significant $-\log_2 A_n$ bits in the binary fractional representation of C_n to uniquely define the encoded interval. This is because C_n will never reach the value of $C_n + A_n$ regardless of the number of 1s concatenated to C_n in the decoder. In the example, if there are $-\log_2 A_n$ bits representing n symbols, then the average code rate is

$$\frac{-\log_2 A_n}{n} = \frac{9}{4} = 2.25 \qquad \textit{Eqn. 6}$$

This shows that in the example, the average code rate for four symbols is higher than the entropy lower bound. However, as n tends to infinity, the average number of bits per symbol does converge to the entropy. This is because

$$\frac{-\log_2 A}{n} = \frac{-\sum_{i=1}^{n} \log_2 f_X(x_i)}{n} \xrightarrow[n \to \infty]{} -E[\log_2 f_X(X)] = H(X) \qquad \textit{Eqn. 7}$$

where the $E[x]$ operator denotes the expectation of x.

Again, this example is only intended to show the relationship between entropy and Huffman coding. Arithmetic encoding can be equally effective with symbols whose probabilities are not rational fractions of 2^i where i is an integer.

2.3 Binary Arithmetic Coding

In binary arithmetic coding (BAC), symbols in a code stream are classified as either Most Probable Symbol (MPS) or Least Probable Symbol (LPS). The interval A (see Figure 2 on page 9) has two divisions, one each for MPS and LPS.The width of each division is determined by the probability for each symbol. The interval associated with the LPS should always be less than that associated with the MPS. The events received by the encoder can either be MPS:True ('T') or MPS:False ('F').

The JPEG2000 literature has adopted the convention of referring to the probability for the LPS as Q_e and the corresponding probability for the MPS is $(1 - Q_e)$. In this document, the probability for the MPS is denoted by P_e.

Figure 5 illustrates the BAC process. In this example, the message 'TFTT' is coded where T denotes the MPS and F the LPS (MPS:False). The probabilities are $Q_e = \frac{1}{4}$ and $P_e = 1 - Q_e = \frac{3}{4}$ and are represented as two non-overlapping subintervals. The convention adopted here is that the Q_e subinterval always precedes the P_e subinterval. As before, the initial interval is [0, 1). Again, the notation has been given in both fractional and binary fraction format. When a symbol occurs, the subinterval associated with that symbol becomes the new interval. For example, the initial interval [0, 1) has two subintervals [0, 0.01) and [0.01, 1) associated with 'F and 'T' respectively. When 'T' occurs, the subinterval [0.01, 1.0) becomes the new interval. The code word C in this example always points to the left point (lower bound) of the interval and A denotes its width.
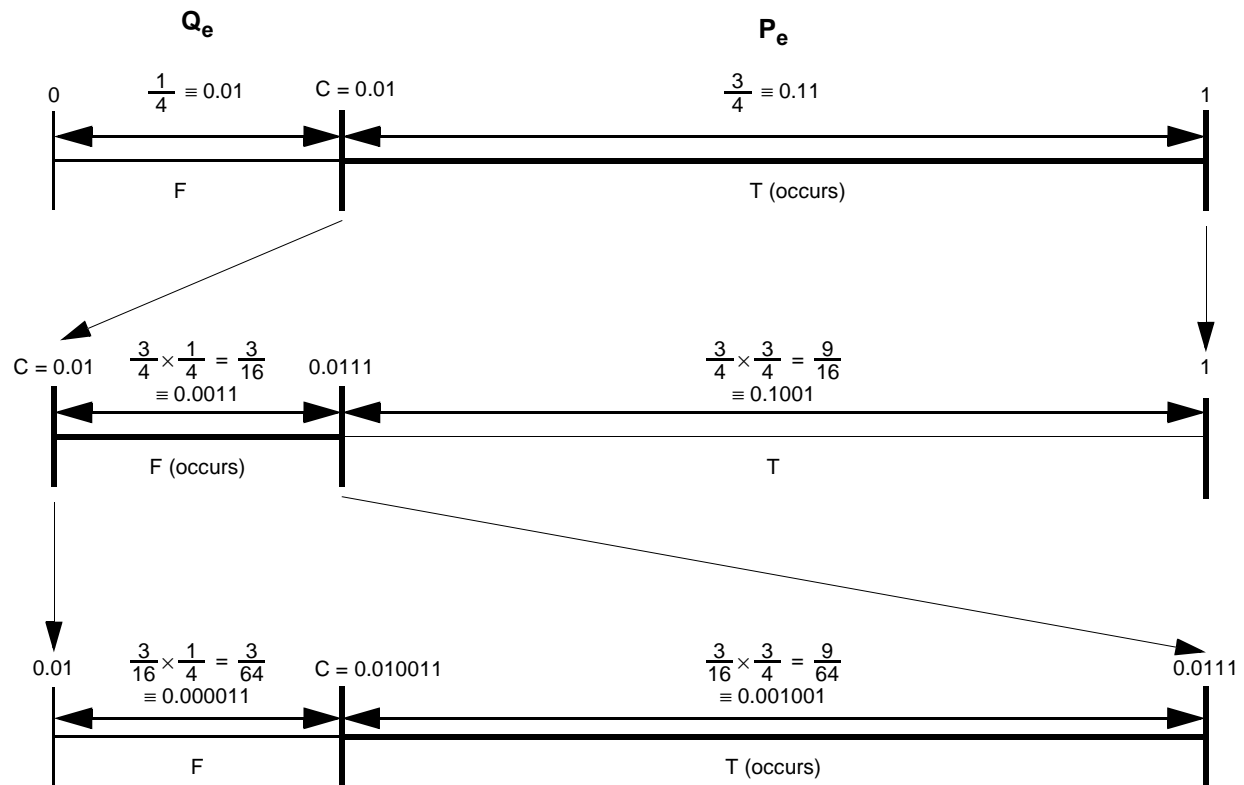
Figure 5. BAC Process

2.3.1   BAC Encoding

The encoding steps for the BAC can be summarized as follows:

1. Initialize $A_0 = 1$ and $C_0 = 0$

2. Determine the event values (MPS:True or MPS:False), code register C, Q_e, and P_e.

3. Update the code word and/or interval depending on the event values

    If MPS:True:T

    a) Update the code word to point at the lower bound of the interval
        $C = C + (Q_e \times A)$

    b) Update the interval width
        $A = A - (Q_e \times A)$

    If MPS:False: F Leave the code word as it is already pointing to the lower bound

    a) Update the interval width
        $A = Q_e \times A$

The new interval becomes $[C, C+A)$ with subintervals $[C, C+(A \times Q_e))$ and $[C+(A \times Q_e), C+A)$

The following paragraphs describe the application of these steps to the first three symbols in the example.

Initialization

C = 0 and A = 1.0.

Encoding the first symbol

Given: A = [0, 1) with subintervals
$[C, C+(A \times Q_e)) = [0, 0.01]$ and $[C+(A \times Q_e), C+A) = [0.01, 1)$:
'T' occurs
$Q_e \times A = 0.01 \times 1.0 = 0.01$
$C = C + (Q_e \times A) = 0 + 0.01 = 0.01$
$A = A - (Q_e \times A) = 1.0 - 0.01 = 0.11$

Subinterval $[C, C+A) = [0.01, 1.0)$ becomes the new interval.

Encoding the second symbol

Given: A = [0.01, 1) with subintervals [0.01, 0.0111) and [0.0111, 1)
'F' occurs
$Q_e \times A = 0.01 \times 0.11 = 0.0011$
$C = 0.01$ (unchanged)
$A = Q_e \times A = 0.0011$

Subinterval [0.01, 0.0111) becomes the new interval.

Encoding the third symbol

Given: A = [0.01, 0.0111) with subintervals [0.01, 0.010011) and [0.010011, 0.0111)
'T' occurs
$Q_e \times A = 0.01 \times 0.0011 = 0.000011$
$C = C + (Q_e \times A) = 0.01 + 0.000011 = 0.010011$
$A = A - (Q_e \times A) = 0.0011 - 0.000011 = 0.001001$

Subinterval [0.010011, 0.0111) becomes the new interval.

Encoding the fourth symbol continues …

Note that at any given stage, C is constrained within all preceding intervals. For example, when the third symbol is encoded, the preceding intervals are [0.01, 1), [0.01, 0.0111) and [0.010011, 0.0111). After encoding, the new value of C is 0.010011, which falls within all three intervals. Further, the next C code word will still be constrained within [0.010011, 0.0111) regardless of which symbol is encoded next. In other words, as the encoding continues, the dynamic range of C becomes progressively smaller while its precision gets progressively higher. Any value of C within the range [0.010011, 0.0111) will decode the symbol stream beginning with 'T F T.'

2.3.2   BAC Decoding

When the decoder receives a code word C, it simply reverses the encoding operation by keeping the same interval (i.e., A) as the encoder and using the same probability distribution.

The decoder also starts with the initial interval [0, 1.0), A = 1.0, $Q_e = 0.01$ and $P_e = 0.11$.

Decoding the first symbol

Given: interval [0, 1) with subintervals [0, 0.01) and [0.01, 1)
Since C = 0.010011 lies within [0.01, 1), the first symbol must be 'T'.

Subinterval [0.01, 1) becomes the new interval.

Decoding the second symbol

Given: interval [0.01, 1) with subintervals [0.01, 0.0111) and [0.0111, 1)
Since C = 0.010011 lies within [0.01, 0.0111), the second symbol must be 'F'.

Subinterval [0.01, 0.0111) becomes the new interval

Decoding the third symbol

> Given: interval [0.01, 0.0111) with subintervals [0.01, 0.010011) and [0.010011, 0.0111)
> Since C = 0.010011 lies within [0.010011, 0.0111), the third symbol must be 'T'.

Subinterval [0.010011, 0.0111) becomes the new interval.

When the lower bound of the interval is equal to C, the end of the symbol stream for the code word has been reached.

There are several practical issues which must be resolved before making use of arithmetic coding. All of these issues have been addressed by the JPEG2000 arithmetic coder.

2.4 JPEG2000 Arithmetic Coding

The JPEG2000 standard utilizes a specific type of efficient, statistical binary arithmetic coding (BAC) which has been adapted from the so called Q-Coder [5]. It deals with the following issues that arise when trying to implement a practical BAC:

- removing the multiplication operations required for each symbol encoding/decoding

- dealing with conditional exchange of the MPS sense, which arises when the resulting interval for an LPS exceeds that of the corresponding MPS

- making the arithmetic coder adaptive

- dealing with finite precision

- problems associated with the growing length of the code word and carry propagation

- software versus hardware implementation.

JPEG2000 has 19 possible contexts. The JPEG2000 BAC can be thought of as 19 independent coders that produce intermixed output. The contexts input to the BAC are used to choose between these coders by indexing different Q_e values in the state machine defined in Table C-2. The 19 coders use the same state machine but move among the various states in different ways.

As an aid to understanding the following discussion, certain figures from the JPEG 2000 Part 1 Final Draft International Standard [4] are reproduced in Appendix C. The diagrams in Appendix C describe the operation of the encoder. Note the figure labels correspond to those in the FDIS.

Figure C-1 shows the arithmetic coder inputs, the context (CX) and data (D) pairs which are provided by the coefficient bit modeler. In the example provided in Appendix A, these pairs are stored in an input array. The output from the arithmetic coder is a stream of compressed data (CD).

Figure C-2 is a top level flow diagram for the encoder. It includes an initialization and a flushing procedure, which are described in detail in Section 3.1.1 and Section 3.1.2 respectively.

The encoder has separate procedures for coding 0 and 1 data inputs. It must first be established whether the input data symbol is an MPS or an LPS. The data values for an MPS for each context are stored in an array which is simply looked up for each input. If the data is an MPS, then CODEMPS (Figure C-7) is used; otherwise CODELPS (Figure C-6) is used.

Figure C-8 shows the renormalization procedure, and Figure C-9 shows how a byte is output to the bit stream.

2.4.1 Removing Multiplication

The JPEG 2000 BAC maintains the interval A in the range [0.75, 1.5) by a process of renormalization (see Section 2.4.4). This means that the interval is always approximately equal to 1, if rounding to one significant figure. Therefore, $(Q_e \times A) \approx Q_e$, and the following approximations can be made to the encoding operations outlined in the BAC steps in Section 2.3:

If MPS:True:T

a) Update the code word to point to the lower bound of the interval
$C = C + Q_e$

b) Update the interval width
$A = A - Q_e$

If MPS:False:F

a) Leave the code word as is (it is already pointing to the lower bound)

b) Update the interval width
$A = Q_e$

Thus, all the multiplication operations have been removed, simplifying implementation.

2.4.2 Conditional Exchange of MPS Sense

If continuous MPS symbols are received, the interval length A is repeatedly updated as $A = A - Q_e$ (Section 2.4.1); eventually, A will be diminished to the point where $A - Q_e < Q_e$. At this point, the interval associated with the probability of an LPS exceed that of an MPS, so the sense of the MPS must be inverted. Similar circumstances arise when continuous LPS symbols are received.

The following discussion describes how the JPEG2000 arithmetic encoder ensures that the interval for an LPS is always less than that of an MPS. For convenience, the discussion assumes that '0' is the MPS and '1' the LPS for a particular context CX. In this context, the coder expects '0' to happen more often than '1'.

2.4.2.1 LPS Case

For an input of (CX, '1'), the ENCODER procedure (see Figure C-3 on page 46) selects the CODE1 procedure. Because '1' is the LPS, the CODE1 procedure (see Figure C-4 on page 46) interprets the value of MPS(CX) to be '0', so it selects the CODELPS procedure. In the CODELPS procedure (see Figure C-6 on page 47) it can be seen that when an LPS symbol occurs, the length of the interval (A) is updated to the value Q_e, while the code word (C) remains unchanged. It appears that if symbol '1' were to occur with the context CX many times, Q_e would become progressively larger (see Table C-2 on page 54) so that eventually $(A - Q_e)$ would become less than Q_e, i.e., the portion of the probability interval A which represents the LPS, ('1' in the case of context CX) would become greater than the portion allocated to the LPS, symbol '0'. However, this does not happen because the CODELPS procedure initially tests for this condition, and swaps the intervals associated with '1' and '0' if the condition is detected, so that '1' is still associated with the smaller portion of interval A. In addition, A is updated to $(A - Q_e)$, and C is updated to $(C + Q_e)$, pointing to the lower bound of the subinterval now associated with the LPS, '1'.

If LPS '1's continue to be received by the encoder with context CX, the NLPS field in Table C-2 causes the system to converge to one of two possible indices (6 or 14) which inverts the sense of the MPS. The MPS(CX) is now '1' instead of '0' and thus the encoder expects a '1' to occur more often than a '0' with context CX, so the next time a '1' is received with CX the procedure CODEMPS is called. This arrangement allows the decoder to detect the change and decode the symbol correctly. Note that the switch is controlled by the CODELPS loop only.

2.4.2.2 MPS Case

If an MPS ('0' for context CX) is received, the CODEMPS procedure (see Figure C-7 on page 46) is called. When symbol '0' (MPS for CX) occurs, normally the length of the interval A is updated to $(A - Q_e)$ and C is updated to $(C + Q_e)$. The interval A becomes progressively smaller as the encoding proceeds. The value of A is always checked after it has been updated to determine if it has fallen below 0.75, thus requiring a renormalization. If it does, it could also mean that stage $(A - Q_e)$ has fallen below the value of Q_e, meaning that the subinterval associated with symbol '0' (MPS for CX) is smaller than the subinterval associated with symbol '1' (LPS for CX). However, it should be noted that this does not indicate that symbol '1' is necessarily more likely to happen than symbol '0' on receipt of context CX, because the encoder is still receiving '0's with CX. It just means that the MPS must be assigned to the larger subinterval of A. When the subinterval representing MPS falls below that of the LPS, A is set to Q_e, the larger of the two subintervals, and C remains unchanged (it is already pointing to the lower bound of the LPS subinterval). A renormalization then occurs to ensure that A remains greater than 0.75. This arrangement allows the decoder to detect the change and decode the symbol correctly. Setting A to the larger subinterval generally reduces the number of renormalization operations required. This slows the growth of the encoded bit stream (i.e., improves data compression) because its growth varies with the frequency of renormalization.

2.4.3 An Adaptive BAC: Probability Estimation Process

Again, a primary advantage of the JPEG2000 BAC is that the probabilities associated with the LPS and MPS can be adapted. In order for a coder to be adaptive, a statistical model of the input data symbols is required to update the probabilities associated with the MPS and LPS. The model must also determine whether each incoming event is an MPS or an LPS.

In JPEG2000, the coefficient bit modeler performs the statistical modelling by providing the BAC with context/data pairs. The context is calculated from the properties of up to 8 of the wavelet coefficient's nearest neighbors. This context is used to index into Table C-2 (page 54), which contains the LPS probability values (Q_e). In addition, for each possible context, there is a 'sense' associated with the MPS, i.e., for each context the MPS has previously been declared as either a 1 or a 0. Thus, for example, if the data input from the coefficient bit modeler is a '1', and the MPS 'sense' is also a '1', then this input is treated as an MPS; otherwise, it is treated as an LPS.

The JPEG2000 arithmetic coder has adopted the practice of updating the probabilities associated with the MPS and LPS only when renormalization has occurred. This practice was first introduced in the Q-coder [5]. A probability model is needed for both the encoder and decoder. This probability model can be viewed as a finite-state machine. In practice, the various states are stored in the indexed table of Q_e probabilities presented in Table C-2. These probabilities have been derived through an extensive optimization procedure which includes both theoretical modelling and coding of actual data.

Table C-2 also includes associated next states (i.e., new table positions) for each type of renormalization. For convenience, a portion of this table is reproduced in Table 2. In this table, the index represents the current state, the NLPS (Next LPS) represents the next state to go to if an LPS occurs, the NMPS (Next MPS) represents the next state to go to if an MPS occurs, and the SWITCH value indicates if the sense of the MPS must be inverted.

Table 2. A Portion Table C-2 From the JPEG2000 FDIS

Index	Q_e Value			NMPS	NLPS	SWITCH
	Hexadecimal	Binary	Decimal			
29	$1101	0001 0001 0000 0001	0.099 634	30	27	0
30	$0AC1	0000 1010 1100 0001	0.063 012	31	28	0
31	$09C1	0000 1001 1100 0001	0.057 153	32	29	0
32	$08A1	0000 1000 1010 0001	0.050 561	33	30	0
33	$0521	0000 0101 0010 0001	0.030 053	34	31	0

In principle, the probability model counts the frequencies of LPS and MPS to adaptively estimate Q_e and P_e. The model always expects an MPS to occur. The state machine is designed so that when an LPS occurs, the machine transfers to a new state with a larger Q_e, and a renormalization is performed. When an MPS occurs and the interval length A becomes less than 0.75, the state machine transfers to a new state with a smaller Q_e. Thus, the change in state occurs only when the arithmetic coder interval register is renormalized. In other words, if an MPS occurs, then the probability associated with the MPS, $(1 - Q_e)$, is increased slightly because it has taken place once again, i.e. this means that the LPS is even less likely to occur so the Q_e is correspondingly reduced. Similarly, if an LPS occurs, then its probability of occurrence (i.e., Q_e) must be increased slightly.

For example, when the model is at state 31 (index 31), $Q_e = 0.057153$. If an LPS occurs, the model transfers to state 29 where $Q_e = 0.099634$, which is greater than 0.057153. Renormalization then follows. If an MPS then occurs, a state transition may or may not occur, depending on the value of interval length A. If A < 0.75, the model transfers to state 30 where $Q_e = 0.063012$ (smaller than 0.099634). Renormalization then follows. Otherwise, no state transition, hence no renormalization occurs.

2.4.4   Finite Precision

To solve the growing precision problem, coding operations are carried out using fixed precision integer arithmetic. The hexadecimal value $8000, which normally represents the decimal value 0.5, represents the decimal value 0.75 in the JPEG2000. With this representation, if the integer $8000 is left-shifted once, the decimal value becomes 1.5; if it is right-shifted once, the decimal value becomes 0.375. The interval length A is renormalized whenever the integer value falls below $8000, and is kept in the range [0.75, 1.5) by a left shift.

To maintain consistent scaling between C and A, C is left-shifted whenever A is renormalized. This scaling would require a register of unlimited length in which to store the value of C. To avoid this impossible requirement, an external data buffer or 'B register' is attached to the high order bits of the C register. A byte of data is shifted from the high order bits of the C register to this buffer whenever the C register is full. Each output byte effectively refines the code word, resulting in incremental code word transmission.

2.4.5   Carry Propagation

Whenever an MPS occurs, the C value is incremented by the value of the LPS interval, which can generate a carry. However, if the carry bit propagates into the data buffer, thus altering its contents, the incremental transmission is invalid. The B register described in Section 2.4.4 also serves to resolve the carry problem.

By buffering the completed bytes from the C register, the B register always keeps a byte just removed from the C register before the byte is sent to the output code stream. A shift counter, CT, counts the number of shifts in the C register. When CT counts down to zero, the byte currently stored in the B register is moved to the output code stream, then a byte from the C register is moved to the B register. This process is illustrated in Figure 6.

The C register in Figure 6 is partitioned as follows:

- The 'x' bits are the fractional bits which are directly incremented by the value of Q_e.

- The 'b' bits indicate the bit positions from which a complete byte of the data is removed from the C register.

- The 'c' bit is the carry bit.

- The 's' bits are spacer bits which provide a useful buffer for a carry, so that it takes longer for a carry to be propagated from the 'x' bits to the 'c' bit.
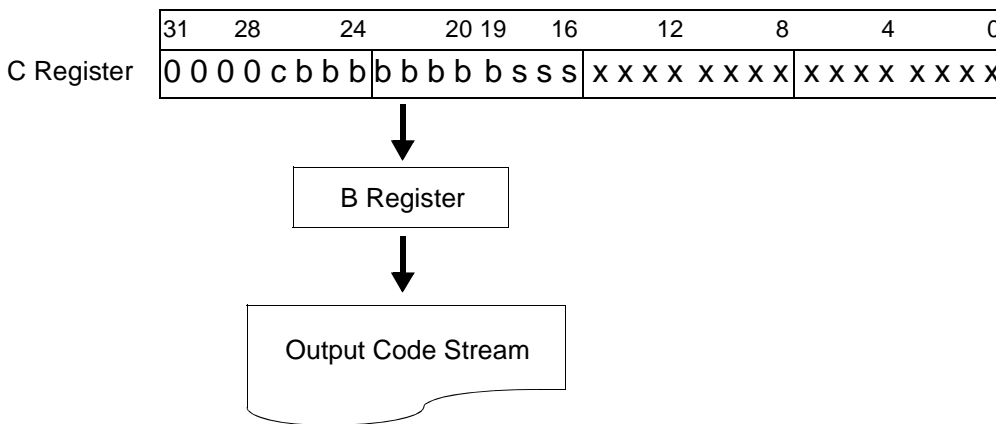
Figure 6.   C Register Partitions and Encoder Output

The operation of moving data to or from the B register must adhere to the following so-called 'bit-stuffing' rules (refer to the BYTEOUT procedure in Figure C-9 on page 50).

1. If B ≠ $FF and the carry bit ('c') is clear:

    a) the byte in the B register is moved directly into the output code stream

    b) bits 19 through 26 in the C register are moved into B

    c) CT is set to 8 so that the next byte is output when bit 18 is shifted to bit 26. The current byte output is complete.

2. If B = $FF, a bit must be 'stuffed':

    a) the byte in the B register is again moved directly to the output code stream.

    b) bits 20 to 27 in the C register are moved into B to include the carry bit.

    c) CT is set to 7 so that the next byte is output when bit 19, which has not yet been output, is shifted to bit 26. The current byte output is complete.

3. If B ≠ $FF and the carry bit ('c') is set, the byte in the B register cannot be moved directly into the output code stream; the carry bit must be propagated into B:

    a) increment B by 1

    b) clear bit 'c'.

    c) If B ≠ $FF, follow rule 1; otherwise, follow rule 2.

The decoder checks the bit following any $FF byte. If the bit is set, the decoder knows that a carry has occurred.

2.4.6 Software Versus Custom Hardware Implementation

In the binary arithmetic coding scheme described here, both the interval (A) and the code word (C) are altered whenever the encoder receives an MPS, which is most of the time. This is not of significant concern in hardware because the updates to A and C can be done in parallel. However, in a software implementation it is not generally desirable to perform more operations on the more probable path. A good discussion of this can be found in [6]. One remedy to this situation is to have C point to the right-hand end of the current interval instead of the left-hand end, as shown in Figure 7.

In this configuration, the code word is updated by subtracting P_e only when an LPS occurs. For comparison, the hardware-preferred method for updating C as described in the FDIS [4] is also shown (i.e., C points to the left side of the interval and is only updated when an MPS is received). By definition, an MPS occurs more often than an LPS, so the software version updates the code word less often than the hardware version because it only does so when an LPS is received. The resulting code words from the hardware and software conventions always result in code words that differ from one another by the current interval. It should also be noted that the process of subtraction for the software convention can result in a borrow propagation, as opposed to a carry propagation when using the hardware convention. The FDIS for JPEG2000 makes allowances for software implementations which are discussed in annex J of [4].
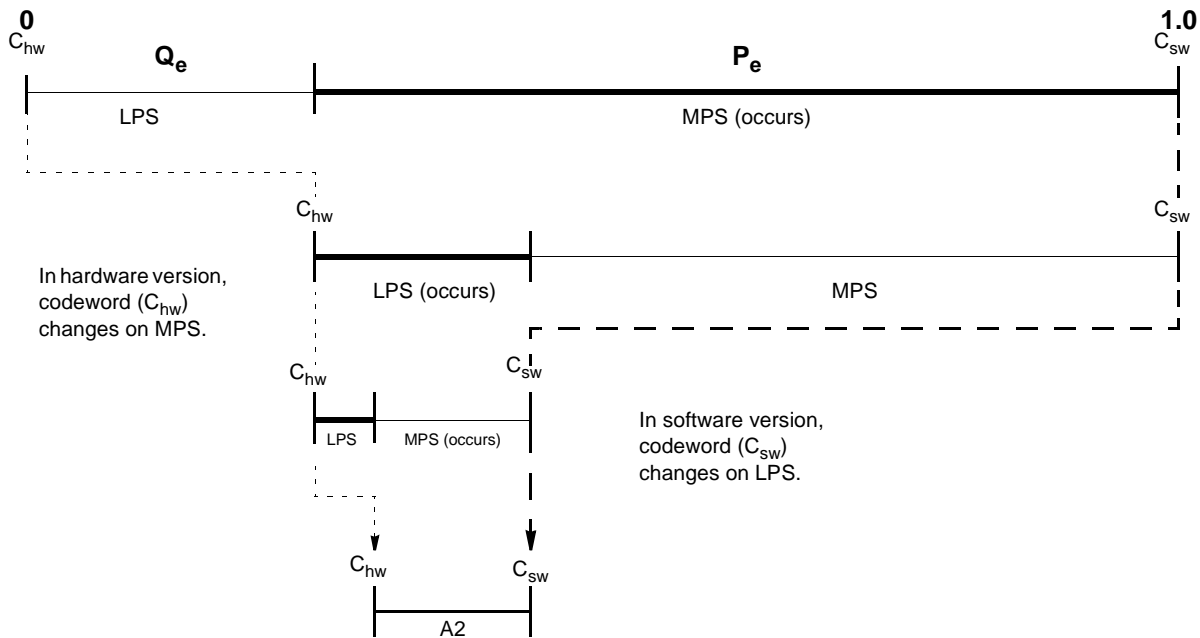
Figure 7. Codeword (C) Changes: Software vs. Hardware Implementation

Note however that the StarCore processor makes it possible for a software implementation to update A and C in parallel, just as in hardware. This is why the example provided in this application note follows the hardware convention for updating A and C.

3 Implementation

This section describes the software written for a StarCore implementation of the JPEG2000 arithmetic coder. Appendix A is a listing of the C code, and Appendix B lists the assembly code. All variables in the following descriptions refer to the actual variables used in the code.

3.1 StarCore Implementation in C Code

The code given in Appendix A is a direct implementation of the JPEG2000 arithmetic encoder as presented in Section 2.4, and is based on the flow diagrams provided in the FDIS [4] which have been produced for ease of reference in Appendix C. A brief description of these diagrams can be found on page 15.

The pseudo code for the encoder software in Appendix A is listed in Code Example 1.

Code Example 1.   JPEG2000 Arithmetic Encoder Pseudo Code

```
/* pre-set A to Pe interval
A = A – Qe
if symbol == MPS
        if A < 0.75
                if A < Qe
                        /* if Pe < Qe, encode the current symbol as LPS */
                        A = Qe
                else
                        /* encode the current symbol as MPS */
                        C = C + Qe
                end
                /* update Qe */
                transfer to new state
                /* scale A and C */
                renormalize
        else
                /* encode the current symbol as MPS */
                C = C + Qe
        end
else    /* symbol == LPS */
        IF a < qE
                /* IF pE < qE, encode the current symbol as MPS.
                the switch to invert the sense of MPS will be on */
                C = C + Qe
        else
                /* encode the current symbol as LPS */
                A = Qe
        end
        /* the switch is on when Qe > 0.5 in this state */
        if switch == 1
                /* if switch is on for this state,
                    invert the sense of MPS */
                MPS = 1 – MPS
        end
        /* update Qe */
        transfer to new state
                /* scale A and C */
                renormalize
        end
```

3.1.1 Encoder Initialization

The encoder initialization is illustrated in Figure C-10 on page 51. The coder is initialized to the following conditions:

- The interval length is set to A = $8000, equivalent to 0.75 in decimal.

- The code word points to the lower bound of the given interval, C = 0.

- The shift counter is set to CT = 12 (if B ≠ $FF) or CT = 13 (if B = $FF).

The shift counter is set to a larger number than 8 or 7 to accommodate the 3 spacer bits in the C register (refer to Figure 6), which initially do not contain valid data. The first byte usually requires 12 normalizations to shift the values updated by the addition of Q_e through to the b and c bits. After the first byte has been output, the spacer bits contain valid data, so only 7 or 8 renormalization shift lefts are required before outputting the next byte. The first byte requires 13 left-shifts if the previous byte was $FF because this causes an extra bit to be spuriously stuffed into the register when BYTEOUT is called for the first time. Thus, an extra renormalization (shift left) must occur to ensure that the leading bit is not lost during the BYTEOUT procedure, that is, when the next value of B becomes equal to C>>20.

3.1.2 Flushing the Encoder

When the encoding is complete, the bits in the C register must be moved to the B register and then to the output code stream before a terminating marker is generated. This task is performed by the FLUSH procedure shown in Figure C-11 on page 52.

The marker code has a prefix of $FF which can be made to overlap the final bits of the compressed data by ensuring that the final bits of the C register are also $FF. This means that any marker code at the end of the compressed image data is recognized and interpreted before decoding is complete because the marker and last image data for the arithmetic decoder are the same. Therefore, the final $FF in the C register can be discarded, thus reducing the length of the output stream. This overlapping of compressed data and marker code is achieved by setting as many bits in the C register as possible. This is determined as follows:

At the end of encoding, C points to the lower bound of the given interval. The upper bound of the interval is equal to C + A. The process of renormalization ensures that A ≥ $8000, hence (C + A) ≥ (C + $8000). Therefore, when A = $8000 the 15 least significant bits of the C register cannot increment the code word, C, beyond the interval even if they are all set. If A > $8000, this is true for the 16 least significant bits of the C register. Setting the correct number of bits is performed by the SETBITS procedure illustrated in Figure C-12 on page 53, which consists of the following steps:

1. Set bits 0 to 15 in the C register.
2. Compare the result to the upper bound of A to determine if C still points to the given interval.
3. If C is too large (i.e., A == $8000), clear bit 15.

This procedure guarantees that C points to the required final interval with the lowest 16 or 15 bits set, and the original symbol can be decoded correctly.

After the SETBITS procedure is called, the last of the compressed data bits in the C register are moved to the output code stream via the B register. The final $FF byte can be discarded.

3.1.3   StarCore Performance

Benchmark tests of several images were run on the StarCore processor with the C code listed in Appendix A, using level 2 optimization with 4 parallel multiply accumulator modules. Figure 8 shows a typical image used in the benchmark tests.
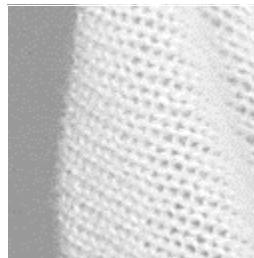
Figure 8.   Test Image for Arithmetic Encoder

This image is 128×128 pixels with 12 bits/pixel. After the coefficient bit modeler had produced the corresponding context/data pairs, the number of clock cycles required to encode this image was extrapolated to determine the performance figures in seconds for a 1 megapixel color image with 12 bits per pixel, assuming a 300MHz clock speed and multiplying by 2 to represent a YUV:422 format. For comparison purposes, the experiments were repeated for a constant grey scale image, which should yield the fastest possible execution times and the greatest compression. The results are summarized in Table 3 on page 24.

The results for the constant grey scale image are two orders of magnitude faster than the typical image. This is because the encoder receives almost constant MPS's, which yields a considerably more compressed bit stream. In addition to this, the coefficient bit modeller only produces context/data pairs for the LL subband at the lowest level of decomposition. Consequently, none of the other subbands have any context/data pairs associated with them.

Table 3.   Processing Time for C Version of Arithmetic Encoder

	Clock Cycles (128 × 128)	Time (ms) (1 Megapixel, Color)
Typical image	4.6 M	1960
Constant grey scale	31,787	14

For comparison, the arithmetic coder was tested on Motorola's DSP56307 with the constant grey scale image, where it was calculated that an equivalent 1M pixel color image with 12 bits/pixel would take 69ms to execute, about five times slower than StarCore. The code used on the DSP56307 was C code with optimized in-line assembler functions. In addition, it is assumed that the DSP56307 runs at 100 MHz rather than StarCore's 300 MHz.

3.2 StarCore Implementation in Assembler

This section explains how specific StarCore features improve the performance of the arithmetic encoder. An optimized assembly version of the arithmetic coder is provided in Appendix B.

Although the arithmetic coder is primarily a sequential operator, several parallel features of StarCore can be used to increase processing speed. In addition to the four MAC units to parallel process many instructions, StarCore offers numerous and flexible address registers, delayed jump instructions, and single-instruction if-then-else decisions.

3.2.1 Address Registers

StarCore's address registers can be used to great advantage to access the arithmetic encoder look-up tables. The assembly code in Appendix B shows two ways in which the look-up tables can be organized using StarCore.

There are two main look-up tables associated with the arithmetic encoder, one for Q_e values and one for the context data. The Q_e table in Appendix B, which is the same as Table C-2 in the FDIS, retains all values in one look-up table to highlight the use of address register arithmetic. The context table, on the other hand, has been separated into two look-up tables to illustrate the use of StarCore's multiple address registers.

3.2.1.1 Address Register Arithmetic

Address register arithmetic is used extensively to access the Q_e table. Features such as post increment and offset arithmetic allow access to the other 'columns' in the table without incurring many extra cycles.

Code Example 2.   Address Register Arithmetic

```
move.w (r3+n0),d9    cmpgt d0,d7
```

In Code Example 2, the Q_e table entry for the SWITCH variable is accessed without moving the pointer, r3, from the NMPS index. (A comparison between two data registers is performed simultaneously.)

NOTE:

> In order for this particular format of lookup table to be used, it was necessary to multiply all the original indices in FDIS Table C-2 by 8. This reflects the fact that there are four entries associated with each index and each entry is a word, not a byte.

3.2.1.2 Multiple Address Registers

The context table is split into two separate look-up tables in Appendix B—one for the indices and one for the MPS associated with each context— to highlight the advantage of StarCore's numerous address registers. Address register r2 is assigned the base address of the context table entries for the indices whereas r4 is assigned to the base address of the MPS values. Because there are up to 16 address registers available, there is no need to load the address registers repeatedly with the base addresses; they can retain the base addresses for the duration of the program. In addition, two different address registers hold the index offsets into the two separate context tables when a new context is received at the input. This allows the address arithmetic which calculates the new position for both the Q_e and context tables to be carried out in parallel, as shown in Code Example 3.

Code Example 3. Multiple Address Registers

```
                             ; r5 is index into CONTXT:MPS table
        adda r2,r6   adda r4,r5   ; r6 is index into CONTXT:index table
```

In fact, there are enough registers available to split the Q_e table into four separate tables, as is done to the context table. There is almost no difference in processing speed between the two approaches.

3.2.2 Change-of-Flow Instructions

The assembler implementation of the JPEG2000 arithmetic coder requires several change-of-flow instructions.These instructions generally require more cycles to execute than other instructions because they disrupt the pipeline. StarCore allows a delayed version of most jump instructions which enables the execution of an extra instruction set while the pipeline is filled, thereby saving one or more cycles in processing time. In addition, StarCore allows jump instructions to be combined with decisions based on whether or not the T bit is set in the status register, as illustrated in Code Example 4.

Code Example 4. Jump Instruction Using Delay and the T Bit

```
        jtd    CODEMPS            move.w (r3)+,d7
        sub    d7,d0,d0
```

In this example, a jump to the label CODEMPS is executed if the T bit is set. However, the `move.w` instruction occurs in parallel with this operation and the `sub` command executes in the delay cycle. Thus, although jump and branch instructions are normally costly in terms of cycle time, StarCore allows other instructions to be executed at the same time, increasing overall execution speed.

3.2.3 If-Then-Else Decisions

StarCore enables an 'if-then-else' decision, which depends on the state of the T bit, to be made in one instruction. This is extremely useful in the JPEG2000 arithmetic encoder, in which several such decisions are made. The T bit in the status register determines which value to load into a variable.

Code Example 5. T Bit Selects a Value

```
        ift    move.w #13,d4      iff    move.w #12,d4
```

In Code Example 5, if the T bit is set, d4 is loaded with #13; otherwise it is loaded with #12.

There are several instructions whose execution depends on the state of the T bit.

Code Example 6.   T Bit Selects an Instruction

```
tfrt  d7,d0              iff   add d7,d1,d1
```

In Code Example 6, if the T bit is set, d7 is transferred to d0; otherwise d7 is added to d1.

3.2.4   Results

Table 4 compares the results for the optimized assembler version of the arithmetic coder with the results for the C version for both a typical image and a constant grey scale image. The time required for a 1M pixel color image in assembler was derived by extrapolating the results obtained with a single 128×128 grey scale tile, as it was done for C code.

Table 4.   Comparison of Results for Assembly with C Versions of Arithmetic Encoder

Image	Assembly		C Code
	Clock Cycles (128×128)	Time (ms) (1 Megapixel, Color)	Time (ms) (1 Megapixel, Color)
Typical image	1.5 M	640	1960
Constant grey scale	9586	4	14

The optimized assembler completes the task in less than a third of the time required by the C code. For the constant grey scale image, the assembler version is about 17 times faster than the C version run on the DSP56307 with in-line optimized assembler functions (see Section 3.1.3). Note that this result assumes that the DSP56307 runs at 100 MHz while StarCore operates at 300 MHz.

4　　Summary

The arithmetic encoder C code given in Appendix A takes 4.6M cycles to complete one typical 128×128 pixel grey scale image, which corresponds to a processing time of 2 seconds to complete a color 1 megapixel image. This processing time has been measured with the assumption that the pixels are all 12 bits in size. In addition, it has been assumed that the StarCore processor is operating at a clock speed of 300 MHz. While the C code in Appendix A is ANSI C compliant, it was fully optimized by the StarCore compiler to produce the given performance figures.

An optimized assembler version of the encoder codes the same grey scale image in less than one third of the time taken by the C version. This increase in speed is due primarily to the arithmetic and multiplicity of the address registers. StarCore's parallel decision instructions and delayed change of flow instructions also contributed to the improvement.

The DSP56307 (running at 100 MHz) encoded a constant grey scale image about 17 times more slowly than the optimized assembler version using StarCore (running at 300 MHz).

Modifying the code to work with any image size should be fairly straightforward. All results were obtained using the Beta 1.1 version of the StarCore software.

5   References

[1] S. Twelves, A. White, M. Wu, *JPEG2000 Wavelet Transform Using StarCore*, Motorola application note, order number AN2089/D.

[2] C. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, Vol. 27, pp. 379–423, July 1948.

[3] G. G. Langdon Jr., *An Introduction to Arithmetic Coding*, IBM Journal of Research and Development, Vol.28, No.2, March 1984.

[4] *JPEG 2000 Image Coding System*, JPEG 2000 Part 1 Final Draft International Standard, FDIS 15444-1, 18th Aug. 2000.

[5] W.B. Pennebaker, J.L. Mitchell, *Probability Estimation for the Q-Coder*, IBM Journal of Research and Development, Vol. 32, p. 737, 1988

[6] J.L. Mitchell, W.B. Pennebaker, *Software Implementations of the Q-Coder*, IBM Journal of Research and Development, Vol. 32, No. 6, Nov. 1988

Acknowledgement

Appendix A
Arithmetic Encoder: C Code

Arithenc.c

```c
/*
 An implementation of the MQ arithmetic encoder described in
 Annex C of the JPEG2000 FDIS,[4].

 All data types converted to uint32.
 Context variables made global, rather than access via pointers.
 Manually inlined the CodeMPS and CodeLPS functions.
   author Matthew Leditschke
   version $Revision: 1.1 $
*/

#include <stdio.h>
#include <stdlib.h>
#include "arithEnc.h"
#include "fileIO.h"

/*
 * Set this to inline to make all internal functions inline.
 */

#define inline

/* The information associated with a context state */
typedef struct
{
        uint32 I;     /*< The index into Table C-2 (values in the range 0..46) */
        uint32 MPS;   /*< The most probable symbol (either 0 or 1) */
} ArithEncContext;


/*
 * The state information required by the arithmetic encoder.
 */

/* The lower bound of the current probability interval */
static uint32 C;

/* The size of the current probability interval */
static uint32 A;

/* Counts the number of shifts that have been performed */
static uint32 CT;

/* The byte that is currently being assembled */
static uint8 B;

/* 1 if the current byte is the first output byte, otherwise 0 */
static int firstByte;


/* The array of allowable context states */
#define MAXCONTEXT 18
static ArithEncContext contexts[MAXCONTEXT+1];
```

```c
/* ========================================================================== */
/* Define these inputs to have different data types */

static uint32 Qe[47] =
{
0x5601UL, 0x3401UL, 0x1801UL, 0x0ac1UL, 0x0521UL, 0x0221UL, 0x5601UL,
0x5401UL, 0x4801UL, 0x3801UL, 0x3001UL, 0x2401UL, 0x1c01UL, 0x1601UL,
0x5601UL, 0x5401UL, 0x5101UL, 0x4801UL, 0x3801UL, 0x3401UL, 0x3001UL,
0x2801UL, 0x2401UL, 0x2201UL, 0x1c01UL, 0x1801UL, 0x1601UL, 0x1401UL,
0x1201UL, 0x1101UL, 0x0ac1UL, 0x09c1UL, 0x08a1UL, 0x0521UL, 0x0441UL,
0x02a1UL, 0x0221UL, 0x0141UL, 0x0111UL, 0x0085UL, 0x0049UL, 0x0025UL,
0x0015UL, 0x0009UL, 0x0005UL, 0x0001UL, 0x5601UL
};

static uint32 SWITCH[47] =
{
    1,    0,    0,    0,    0,    0,
    1,    0,    0,    0,    0,    0,    0,    0,
    1,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0
};

static uint32 NMPS[47] =
{
    1  ,   2  ,   3  ,   4  ,   5  ,  38  ,
    7  ,   8  ,   9  ,  10  ,  11  ,  12  ,  13  ,  29  ,
   15  ,  16  ,  17  ,  18  ,  19  ,  20  ,  21  ,  22  ,
   23  ,  24  ,  25  ,  26  ,  27  ,  28  ,  29  ,  30  ,
   31  ,  32  ,  33  ,  34  ,  35  ,  36  ,  37  ,  38  ,
   39  ,  40  ,  41  ,  42  ,  43  ,  44  ,  45  ,  45  ,
   46
};

static uint32 NLPS[47] =
{
    1  ,   6  ,   9  ,  12  ,  29  ,  33  ,
    6  ,  14  ,  14  ,  14  ,  17  ,  18  ,  20  ,  21  ,
   14  ,  14  ,  15  ,  16  ,  17  ,  18  ,  19  ,  19  ,
   20  ,  21  ,  22  ,  23  ,  24  ,  25  ,  26  ,  27  ,
   28  ,  29  ,  30  ,  31  ,  32  ,  33  ,  34  ,  35  ,
   36  ,  37  ,  38  ,  39  ,  40  ,  41  ,  42  ,  43  ,
   46
};



/* ========================================================================== */
/*
 Initialize the arithmetic encoder.
  See Section C.2.8 and Figure C-10.
  param: file - The file to write the output bytes to.
*/

void ArithEncInit()
{
            int i;

            A = 0x8000u;
            C = 0;
            B = 0;   /* Assume that the byte prior to starting is 0 */
            CT = 12;
    firstByte = 1;

            /* Initialize all of the contexts (see Table C-6) */
```

```
                    for (i = 0; i <= MAXCONTEXT; i++)
                    {
                            contexts[i].I = 0;
                            contexts[i].MPS = 0;
                    }
                    contexts[UNIFORM_CX].I = 46;
                    contexts[RUNLENGTH_CX].I = 3;
                    contexts[0].I = 4;
} /* ArithEncInit */


/* ======================================================================= */
/*
 Perform the bottom right box of Figure C-9.
*/

static inline void ByteOutRight(void)
{
    if (!firstByte)
        TransmitByte(B);
    else
        firstByte = 0;
            B = (uint8)((C >> 20) & 0xFF);
            C = C & 0xFFFFF;
            CT = 7;
} /* ByteOutRight */


/* ======================================================================= */

/*
 Perform the bottom left box of Figure C-9.
*/

static inline void ByteOutLeft(void)
{
    if (!firstByte)
        TransmitByte(B);
    else
        firstByte = 0;
            B = (uint8)((C >> 19) & 0xFF);
            C = C & 0x7FFFF;
            CT = 8;
} /* ByteOutLeft */

/* ======================================================================= */
/*
 Write out bytes, allowing for 0xFF stuffing.
 See Section C.2.7 and Figure C-9.
*/

static inline void ByteOut(void)
{
            /* This test sequence might speed up process */
            if (B != 0xFF)
            {
                if (C < 0x8000000)
                    ByteOutLeft();
                else
            {
                B += 1;
                if (B != 0xFF)
                    ByteOutLeft();
                else
                {
                        C = C & 0x7FFFFFF;
```

```
                        ByteOutRight();
                }
        }

                else
                        ByteOutRight();
} /* ByteOut */

/* ======================================================================= */
/*
 Renormalize the lower bound and size of the probability range.
 See Section C.2.6 and Figure C-8.  This function ensures that the range (A)
 is at least 0.75
*/

static inline void Renorme(void)
{
        do
        {
                /* Check that there will be no overflow with these shifts */
                A <<= 1;
                C <<= 1;
                CT -= 1;

                if (CT == 0)
                        ByteOut();
        }
        while (A <= 0x8000);
} /* Renorme */
/* ======================================================================= */
/*
 Encode a bit using the given context.
 Output bits will be written to the file given to ArithEncInit().
 See Section C.2.2 and B.2.3
  param: D  -  The bit to encode, either 0 or 1.
  param: CX -  The number of the context to use.
*/

void ArithEncEncode(uint8 D, uint16 CX)
{
        ArithEncContext* pCX;
        uint32 I;
        uint32 QeI;
        uint32 MPS;

        pCX = &(contexts[CX]);

        /* Put the current context into globals, and write them back later */
        I = pCX->I;
        MPS = pCX->MPS;
        QeI = Qe[I];

        if (pCX->MPS == D)
        {
                /* CodeMPS */
                A -= QeI;

                /* Doing this test first might speed up process */
                if (A >= 0x8000u)
                /*if ((A & 0x8000) != 0)*/
                {
                        C += QeI;
                }
                else
                {
                        /* Doing this test first might speed up process */
```

```
                                if (A >= QeI)
                                        C += QeI;
                                else
                                        A = QeI;

                                I = NMPS[I];

                                Renorme();
                        } /**/
                }
                else
                {
                        /* CodeLPS */
                        A -= QeI;
                        /* Doing this test first might speed up process */
                        if (A >= QeI)
                                A = QeI;
                        else
                            C += QeI;
                                if (SWITCH[I])
                                MPS = 1 - MPS;

                        I = NLPS[I];

                        Renorme();
                }

                pCX->I = I;
                pCX->MPS = MPS;
} /* ArithEncEncode */


/* ========================================================================= */
/*
 Flush the output of the arithmetic encoding process.
 This must be called once all of the data bits have been sent through the
 encoder.
 See Section C.2.9 and Figure C-11.
*/

void ArithEncFlush(void)
{
                uint32 tempC;

                /* SETBITS (Figure C-12) */
                tempC = C + A;
                C |= 0xFFFF;
                if (C >= tempC)
                        C -= 0x8000;

                C <<= CT;
                ByteOut();

                C <<= CT;
                ByteOut();

                if (B != 0xFF)
                        TransmitByte(B);

                ArithEncInit();
} /* ArithEncFlush */

/* ========================================================================= */
```

Encoder.c

```c
/*
 A test harness for the MQ arithmetic encoder described in
 Annex C of JPEG2000.

  This main program performs arithmetic encoding, given context-bit
  pairs stored in arrayIn[].

  author Matthew Leditschke
  version $Revision: 1.3 $
*/

#include <stdio.h>
#include <stdlib.h>
#include "arithEnc.h"
#include "fileIO.h"


/* Create arrayIn and arrayOut for testing */
#include "tile5-symbols.h"

void main()
{
    int i,j,k;
    uint16 CX;
    uint8 D;
    int numBlocks, numPairs, blockIdx;

    ArithEncInit();

    numBlocks = arrayIn[0];
    blockIdx = 1;

    for (i=0; i<numBlocks; i++)
    {
        numPairs = arrayIn[blockIdx];
        blockIdx += (1+numPairs<<1);

        for (j=0, k=2; j<numPairs; j++,k=k+2)
        {
            CX = arrayIn[k];
            D = arrayIn[k+1];
            ArithEncEncode(D, CX);
        }
        ArithEncFlush();
    }
/*
        When including "tile5-symbols.h", the
        correct results in arrayOut[] should be
            11 50 54 af
*/
} /* main */

/* ======================================================================= */
```

FileIO.c

```c
/*
 Handle the input and output of bytes.

  author Matthew Leditschke
  version $Revision: 1.3 $
*/

#include <stdio.h>
```

```c
#include <stdlib.h>

#include "types.h"
#include "fileIO.h"

/* Store the encoded bytes for testing */
extern uint8 arrayOut[];

/* Keep track of the total number of bytes written */
static uint32 bytesWritten = 0;

/* This writes out bits as text, '0' for a 0 bit and '1' for a 1 bit */
#undef WRITE_BITS

/* This writes bytes out as binary bytes */
#define WRITE_BYTES

/* This doesn't write out anything - only counts */
#undef NO_WRITE


/* ========================================================================= */

/*  brief Write a byte of data.

   For this testing code, this function just prints out the binary pattern
   for the bytes (most significant bit first) to standard output.
 parameter: byte - The byte to write out.
*/

void TransmitByte(uint8 byte)
{
#ifdef WRITE_BITS
    int bit = 1 << 7;
    while (bit)
    {
        putc((byte & bit) ? '1' : '0', outFile);
        bit >>= 1;
    }
#endif

#ifdef WRITE_BYTES
    arrayOut[bytesWritten++] = byte;
#endif

} /* TransmitByte */


/* ========================================================================= */
/*
Read a byte of data.
*/

uint8 ReceiveByte(FILE* inFile)
{
    uint8 byte = 0;

#ifdef WRITE_BITS
    int i;

    for (i = 0; i < 8; i++)
    {
        byte <<= 1;
        if (getc(inFile) == '1')
            byte |= 0x01;
        /* If reading past the end of the file, read in 1 bits */
```

```
            if (feof(inFile))
                byte |= 0x01;
    }
#endif

#ifdef WRITE_BYTES
    byte = getc(inFile);
    if (feof(inFile))
        byte = 0xFF;
#endif

#ifdef NO_WRITE
    fprintf(stderr, "Nothing was written, so there is nothing to read.\n");
    exit(EXIT_FAILURE);
#endif
    return byte;
} /* TransmitByte */
/* ======================================================================= */
```

ArithEnc.h

```
/*
 The declaration of functions that perform MQ arithmetic encoding
 as defined in Annex C of JPEG2000.
  author Matthew Leditschke
  version $Revision: 1.1 $
*/

#ifndef _ARITH_ENC_H
#define _ARITH_ENC_H

#include "types.h"

void ArithEncInit();
void ArithEncEncode(uint8 D, uint16 CX);
void ArithEncFlush(void);

/* Special context labels */
#define RUNLENGTH_CX    17
#define UNIFORM_CX      18

#endif /* _ARITH_ENC_H */
/* ======================================================================= */
```

FileIO.h

```
/*
 Handle the input and output of bytes.
  author Matthew Leditschke
  version $Revision: 1.2 $
*/
#ifndef _FILE_IO_H
#define _FILE_IO_H
void TransmitByte(uint8 byte);
uint8 ReceiveByte(FILE* inFile);
#endif /* _FILE_IO_H */
/* ======================================================================= */
```

types.h

```
/*
 Define some data types that represent different precisions.
  author Matthew Leditschke
  version $Revision: 1.2 $
*/
#ifndef _TYPES_H
#define _TYPES_H
```

```c
/* An unsigned 32 bit integer */
typedef unsigned long int uint32;

/* An unsigned 24 bit integer */
typedef unsigned long int uint24;

/* An unsigned 16 bit integer */
typedef unsigned short int uint16;

/* An unsigned 8 bit integer */
typedef unsigned char uint8;
#endif /* _TYPES_H */
/* ======================================================================= */
```

tile5-symbols.h

```c
/* Store the output, expected number of bytes = 4 */
uint8 arrayOut[4];

/* Extracted from original tile5-symbols.h */
uint16 arrayIn[1030] = {1,514,17, 1,18, 0,18, 0,12, 1,3, 1,13, 0,3, 1,13, 0,3, 1,
13, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,6, 1,
15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,
16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,
16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,
15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,6, 1,15, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,3, 1,13, 0,3, 1,13, 0,3, 1,
13, 0,3, 1,13, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,16, 0,7, 1,
 .
  .
   .
16, 0};
```

Appendix B
Arithmetic Encoder: Assembly Code

```
;*******************************************************************************
;       Arithmetic Encoder - Assembly Code
;*******************************************************************************
; FILENAME      :     arith_enc.asm
; PREPARED BY   :     MOTOROLA Australia Research Centre, 8 January 2001
; AUTHOR        :            Sue Twelves
;*******************************************************************************
; ALGORITHM     :   JPEG2000 Arithmetic Encoder
;
;*******************************************************************************;
; Constants Used:
;           RENORM_THRESH:       When the interval, A, drops below the threshold of
;                                $8000,renormalisation occurs
;           OUTPUT:              Output to the bitstream is initally to location $950
;           CARRYOVER:           If the C register spills into location $8000000, then
;                                a carry has occurred requiring bitstuffing to take
;                                place
;
; Input/Output location:
;
;           Input initial address is at $2d0
;           Output initial address is at $950
;
; registers used:
;
;           r0 through to r7 and n0
;
;           r0 is the base address of the Qe, table C-2, FDIS
;           r1 is the base address of the input context/data pairs
;           r2 is the base address of the context table for the indices
;           r3 is the offset into the Qe table, i.e. it starts with I(CX)
;           r4 is the base address of the context table for the MPS values
;           r5 is the offset into the context table for the MPS values, i.e. it is CX
;              to start with
;           r6 is the offset into the context table for the Indices, i.e. it is CX
;              to start with
;           r7 is BP, i.e. the output bitstream pointer from the arithmetic coder
;           n0 is used as an offset into the Qe table to access the SWITCH entries.
;
; assumptions:
;
;           Only 1 block of input data is dealt with by this code. In this case, this
;           means that the output CX/D pairs from the coefficient bit modeller when
;           1 tile of 128x128 has been processed.
;
;*******************************************************************************
RENORM_THRESH    equ $8000
OUTPUT           equ $950
CARRYOVER        equ $8000000

; Input data
        org p:$0
        jmp $1000
  org p:$100

; Table C-2, FDIS - Qe values, NMPS,NLPS, switch

qe:          dcw     $5601,$0008,$0008,$0001,$3401,$0010,$0030,$0000
             dcw     $1801,$0018,$0048,$0000,$0ac1,$0020,$0060,$0000
             dcw     $0521,$0028,$00e8,$0000,$0221,$0130,$0108,$0000
             dcw     $5601,$0038,$0030,$0001,$5401,$0040,$0070,$0000
```

```
            dcw     $4801,$0048,$0070,$0000,$3801,$0050,$0070,$0000
            dcw     $3001,$0058,$0088,$0000,$2401,$0060,$0090,$0000
            dcw     $1c01,$0068,$00a0,$0000,$1601,$00e8,$00a8,$0000
            dcw     $5601,$0078,$0070,$0001,$5401,$0080,$0070,$0000
            dcw     $5101,$0088,$0078,$0000,$4801,$0090,$0080,$0000
            dcw     $3801,$0098,$0088,$0000,$3401,$00a0,$0090,$0000
            dcw     $3001,$00a8,$0098,$0000,$2801,$00b0,$0098,$0000
            dcw     $2401,$00b8,$00a0,$0000,$2201,$00c0,$00a8,$0000
            dcw     $1c01,$00c8,$00b0,$0000,$1801,$00d0,$00b8,$0000
            dcw     $1601,$00d8,$00c0,$0000,$1401,$00e0,$00c8,$0000
            dcw     $1201,$00e8,$00d0,$0000,$1101,$00f0,$00d8,$0000
            dcw     $0ac1,$00f8,$00e0,$0000,$09c1,$0100,$00e8,$0000
            dcw     $08a1,$0108,$00f0,$0000,$0521,$0110,$00f8,$0000
            dcw     $0441,$0118,$0100,$0000,$02a1,$0120,$0108,$0000
            dcw     $0221,$0128,$0110,$0000,$0141,$0130,$0118,$0000
            dcw     $0111,$0138,$0120,$0000,$0085,$0140,$0128,$0000
            dcw     $0049,$0148,$0130,$0000,$0025,$0150,$0138,$0000
            dcw     $0015,$0158,$0140,$0000,$0009,$0160,$0148,$0000
            dcw     $0005,$0168,$0150,$0000,$0001,$0168,$0158,$0000
            dcw     $5601,$0170,$0170,$0000


        org p:$280

; Context table: indices and MPS values

ctxt_idx:   dcw     $0020,$0000,$0000,$0000,$0000,$0000,$0000,$0000
            dcw     $0000,$0000,$0000,$0000,$0000,$0000,$0000,$0000
            dcw     $0018,$0170

ctxt_mps:   dcb     $00,$00,$00,$00,$00,$00,$00,$00
            dcb     $00,$00,$00,$00,$00,$00,$00,$00
            dcb     $00,$00


        org p:$2d0

; Input context/data pairs - this corresponds to tile5-symbols.h from the C
; version of the code
; The output using this input should be: 11 50 54 af, as before

ip:     dcw $0001,$0202,
        dcb $11,$01,$12,$00,$12,$00,$0c,$01,$03,$01,$0d,$00,$03,$01,$0d,$00
        dcb $03,$01,$0d,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$03,$01,$0d,$00,$03,$01,$0d,$00,$03,$01,$0d,$00
        dcb $03,$01,$0d,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
```

```
        dcb $07,$01,$10,$00,$03,$01,$0d,$00,$03,$01,$0d,$00,$03,$01,$0d,$00
        dcb $03,$01,$0d,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$06,$01,$0f,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$03,$01,$0d,$00,$03,$01,$0d,$00,$03,$01,$0d,$00
        dcb $03,$01,$0d,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00,$07,$01,$10,$00
        dcb $07,$01,$10,$00
```

```
        org p:$1000
```

```
; Initialize the address registers of the various look-up tables.
; Also set up clock

init:

        move.l #$80030003,PCTL0               ; Ensure clock speed is 300MHz
        move.l #ctxt_idx,r2                   ; r2 is the base address of the context
                                              ; table:index
        move.l #ctxt_mps,r4                   ; r4 is the base address of the context
                                              ; table:MPS
        move.l #ip,r1     deca r2             ; r1 points to the input bitstream of
                                              ; context/data pairs

; decrement r2 so when add context from bitstream, points to right index
        move.l #qe,r0                         ; r0 is the base address of qe table
        move.w #0,d1          adda #2,r1      ; C = 0, assume one block for now so
                                              ; skip the input block numbers
        move.w (r1)+,d3    moveu.w #RENORM_THRESH,d13.l  ; Number of context/data
                                              ; pairs. Renorm threshold used again
                                              ; in arithmetic later
        move.l #CARRYOVER,d10    deca r4      ; This is the constant to check whether
                        ; the c register has overflowed into carry bit.
```

```
                              ; Need CX-1 because context table:MPS starts at 0.

; Initialize the arithmetic encoder - Fig C-10
INITENC:
        moveu.w #RENORM_THRESH,d0.l    move.w #OUTPUT,r7  ; A = 0x8000 (in d0.l)
                             ; BP = BPST-1   i.e. r7 is BP
                             ; output compressed bitstream (BP in FDIS)
                             ; This is BPST-1, i.e. the first byte will be output to r7+1
                             ; Assign this to BP
        move.l #$00010000,PCTL1        adda #1,r1    ; Ensure clock speed is
                             ; 300MHz; rest of input data in bytes, so byte align r1
        move.b (r7)+,d2                move.w #2,n0       ; get previous byte from
                             ; output stream, i.e. B pointed to by BP
        bmtsts #$ff,d2.l               move.b (r1)+,r6        ; read CX, D pair into r6
                             ; and d5.
                             ; CX is an index into CONTXT table of indices; B = 0xFF?
        move.b (r1)+,d5                deca r2             ; data in d5, need to have
                             ;CX-1 because CONTXT:index table address starts at 0

; if B not equal 0xFF, CT = 12  else CT = 13
        ift      move.w #13,d4      iff   move.w #12,d4
        asla r6     tfra r6,r5   ; need to multiply CX by 2 because stored indices
                             ; as words in table
                             ; r5 points to CX:MPS of bytes
        adda r2,r6  adda r4,r5   ; r2 and r4 are the base addresses of the CONTXT
                   ; tables for indices and MPS values respectively, so to access
                   ; correct index and MPS need to offset by base addresses
        dosetup0 BLOCK
        doen0 d3                     ; set up loop through cx,d pairs in one block
                                     ; (assumed only 1 block in this example)


BLOCK

        loopstart0
        falign
        move.w (r6),r3    move.w (r5),d6        ; d6 = MPS  i.e. MPS(CX) ; r3 = index
                                                ; i.e. I(CX)
                cmpeq d5,d6                     ; D = MPS(CX)?
                adda r0,r3                      ;  r3 -> NMPS
                             ; now pointing at Qe in table entry for this index as r0 is
                             ; base address for Qe table
                             ; now code lps or mps

        jtd      CODEMPS  move.w (r3)+,d7     ; from D = MPS(CX)? test, d7 = Qe,
        sub d7,d0,d0            ; in either MPS or LPS case, first step is A = A - Qe

CODELPS:
        move.w (r3+n0),d9        cmpgt d0,d7   ; d9 = SWITCH; A < Qe(I(CX))?
        adda #2,r3                             ; r3 -> NLPS(I(CX))
         ift add d7,d1,d1                      ; if A<Qe true, C = C + Qe(I(CX))

        bmtsts #$0001,d9.l      tfrf d7,d0    ; SWITCH(I(CX)) = 1?
                                              ; if A>Qe then A = Qe(I(CX))
        move.w (r3),d8                        ; if A<Qe, I(CX) = NLPS(I(CX))
        ift bmchg #$0001,d6.l                 ; if SWITCH, MPS = 1 - MPS
                          ; i.e. if 0 change to 1 and vice versa
                          ; replace new sense of MPS to memory and d8 = NLPS(I(CX))
         ift move.w d6,(r5)                    ; if SWITCH,  MPS(CX) = 1 - MPS(CX)
                                              ;always renormalize on the CODELPS path
        jmp RENORME            move.w d8,(r6)     ; I(CX) = NLPS(I(CX))


CODEMPS:
        bmtsts #$8000,d0.l                        ; A AND 0x8000 = 1?
        move.w (r3),d8                            ;  d8 = NMPS(I(CX))
        jt norenorm           ift  add d7,d1,d1  ; if false (i.e. T set)
                                                 ; C = C + Qe(I(CX))
```

```
                  ; if A > 0.75 no renormalisation is required. i.e. A and 0x8000 not = 0
                  ; if A < 0.75, needs renormalisation
                  ; the most sigfig number is not set
                  ; therefore, A has fallen below 0.75 (i.e. 0x8000)

        cmpgt d0,d7         move.w d8,(r6)       ; A < Qe(I(CX))?, r6 -> I(CX),
                                                 ; I(CX) = NMPS(I(CX))
        tfrt d7,d0          iff add d7,d1,d1     ; if true, A = Qe(I(CX)),
                                                 ; else C = C + Qe(I(CX))

; now renormalize

RENORME:
                deceq d4                         ; CT = CT - 1, CT = 0?
        asl   d0,d0         asl   d1,d1          ; A = A << 1, C = C << 1,
        ift   jsr BYTEOUT                        ; if CT = 0, output byte

rentest:
        bmtsts  #$8000,d0.l      ; A and 0x8000 = 1?
        jf RENORME               ; if = 0 (i.e. T cleared) repeat renormalisation

norenorm:
        move.b (r1)+,r6          ; get next CX/D pair
        move.b (r1)+,d5          ; CX is an index into two CONTXT tables of
                                 ; indices and MPS data in d5,
                                 ; need to have CX-1 because
                                 ; CONTXT table address starts at 0
        asla r6            tfra r6,r5            ; need to multiply CX by 2 because
                                 ; stored indices as words in table r5 is index into
                                 ; CONTXT:MPS table of bytes
        adda r2,r6               ; r2 is the base address of the CONTXT:index table so to
                                 ; access correct index need to offset
        adda r4,r5               ; r4 is the base address of the CONTXT:MPS table so to
                                 ; access correct MPS need to offset

        loopend0

FLUSH:
; terminate the coded bitstream with the correct marker code
; setbits - set as many of the C register bits as possible

SETBITS:
        add  d1,d0,d12     ; TEMPC  = C + A
        or  #$ffff,d1.l    ; C = C OR 0xffff
        cmpeq d12,d1       ; if C >= TEMPC then C = C - 0x8000
        ift   sub d13,d1,d1
        cmpgt d12,d1
        ift   sub d13,d1,d1

; end SETBITS

        asll d4,d1         ; C = C << CT
        jsr BYTEOUT        ; BYTEOUT
        asll d4,d1         ; C = C << CT
        jsr BYTEOUT        ; BYTEOUT

        bmtsts #$ff,d2.l   ; if B = 0xFF then discard B
        nop
        ift deca r7        ; if B not = 0xFF then BP = BP + 1
                           ; BP already points to BP + 1, therefore, decrement r7

test:   stop               ; finished coding all pairs of input

BYTEOUT:
; byteout procedure

        bmtsts  #$ff,d2.l                        ; B = 0xFF?
```

```
        jfd nobitstuffyet

bitstuff:
        tfr d1,d11                            ; d1 = d11 = C
        asrr #20,d11      and #$ffff,d1.l     ; C = C AND 0xFFFFF
        jmpd endbyteout   move.w #7,d4        ; CT = 7
        and #$000f,d1.h

nobitstuffyet:
        cmpgt d1,d10                          ; C < 0x8000000?
        jt outputbyte                         ;    true
        inc d2                                ;    false: B = B+1
        bmtsts  #$ff,d2.l                     ; B = 0xFF?
        jf outputbyte                         ; false
        and #$ffff,d1.l                       ; true:  C = C AND 0x7FFFFFF
        jmpd bitstuff
        and #$07ff,d1.h

outputbyte:
        tfr d1,d11                            ; d1 = d11 = C
        asrr #19,d11      and #$ffff,d1.l     ; C = C AND 0x7FFFF
        and #$0007,d1.h   move.w #8,d4        ; CT = 8

endbyteout:
        rtsd    tfr d11,d2                    ; B = C >> 20 or B = C >> 19
        move.b d2,(r7)+                       ; output B to memory BP out
```

Appendix C
Excerpts from FDIS

This appendix reproduces several flow diagrams and a table from Appendix C of ISO/IEC FDIS 15444-1 : 2000 (18th August 2000).

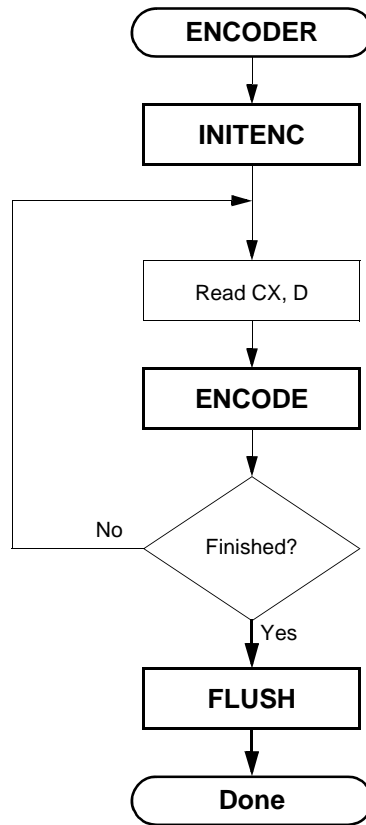**Figure C-1. Arithmetic Encoder Inputs and Outputs**
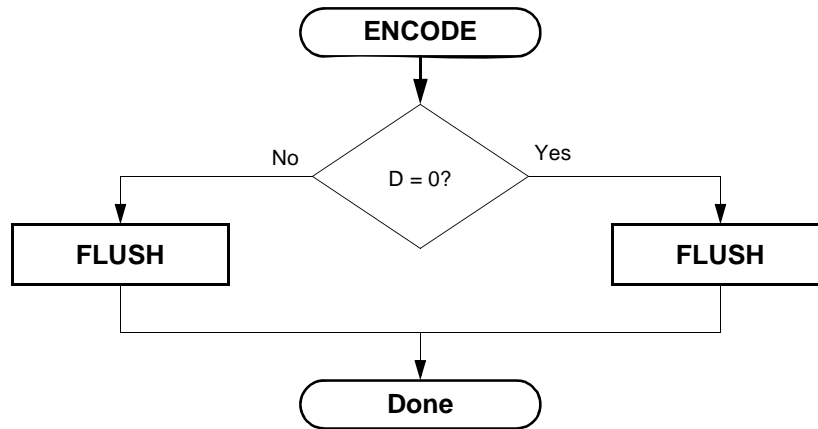

Figure C-2. Encoder for the MQ-Coder
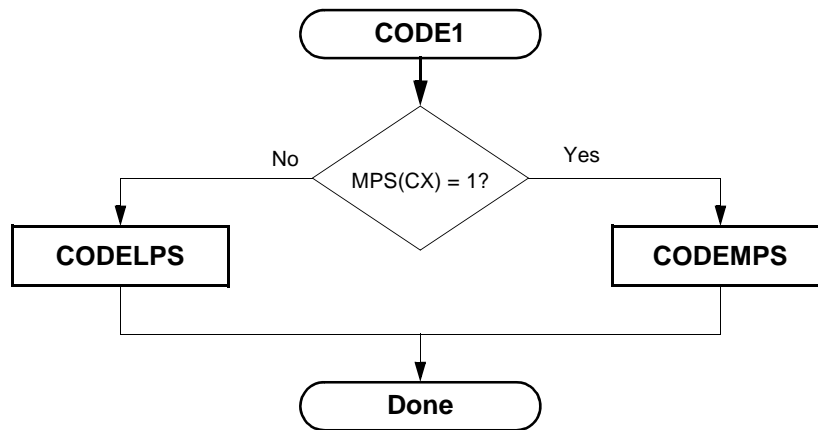
Figure C-3.   ENCODE Procedure

Figure C-4.   CODE1 Procedure
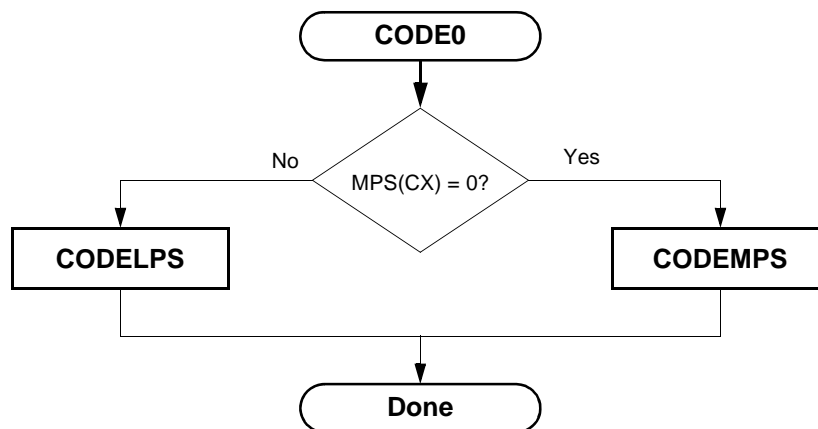
Figure C-5.   CODE0 Procedure
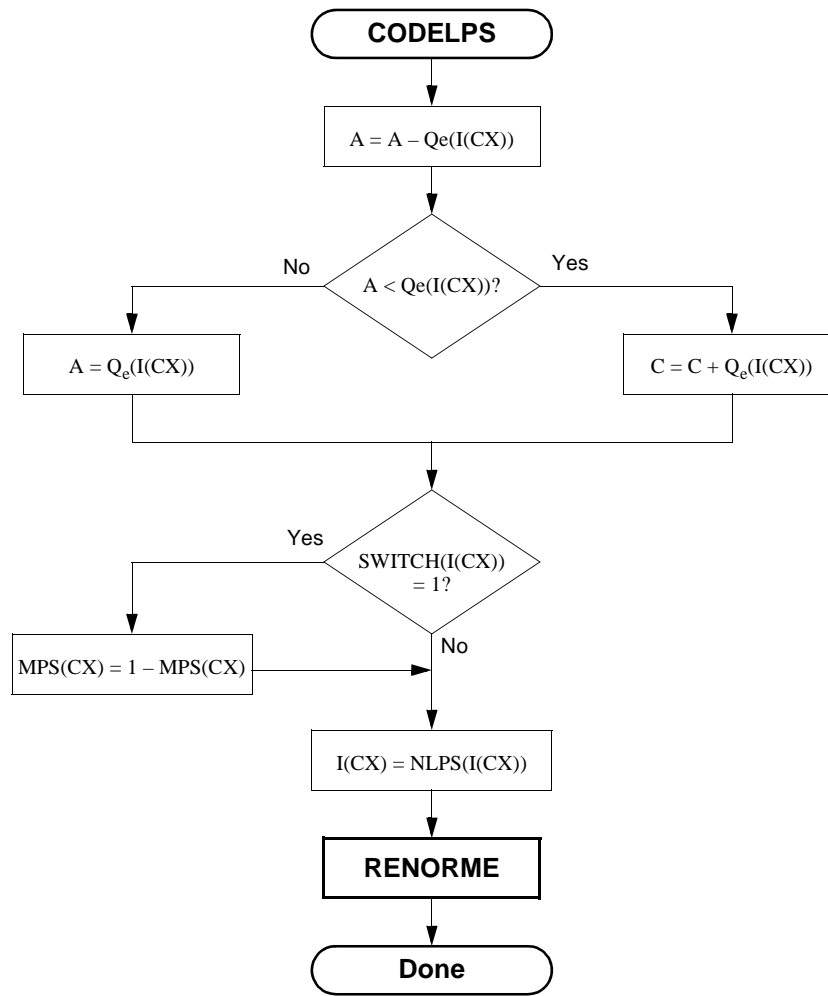
Figure C-6. CODELPS Procedure with Conditional MPS/LPS Exchange

CODEMPS

A = A − Qe(I(CX))

A AND \$8000 = 0?

No → C = C + Q$_e$(I(CX))

Yes → A < Qe(I(CX))?

No → C = C + Q$_e$(I(CX))

Yes → A = Qe(I(CX))
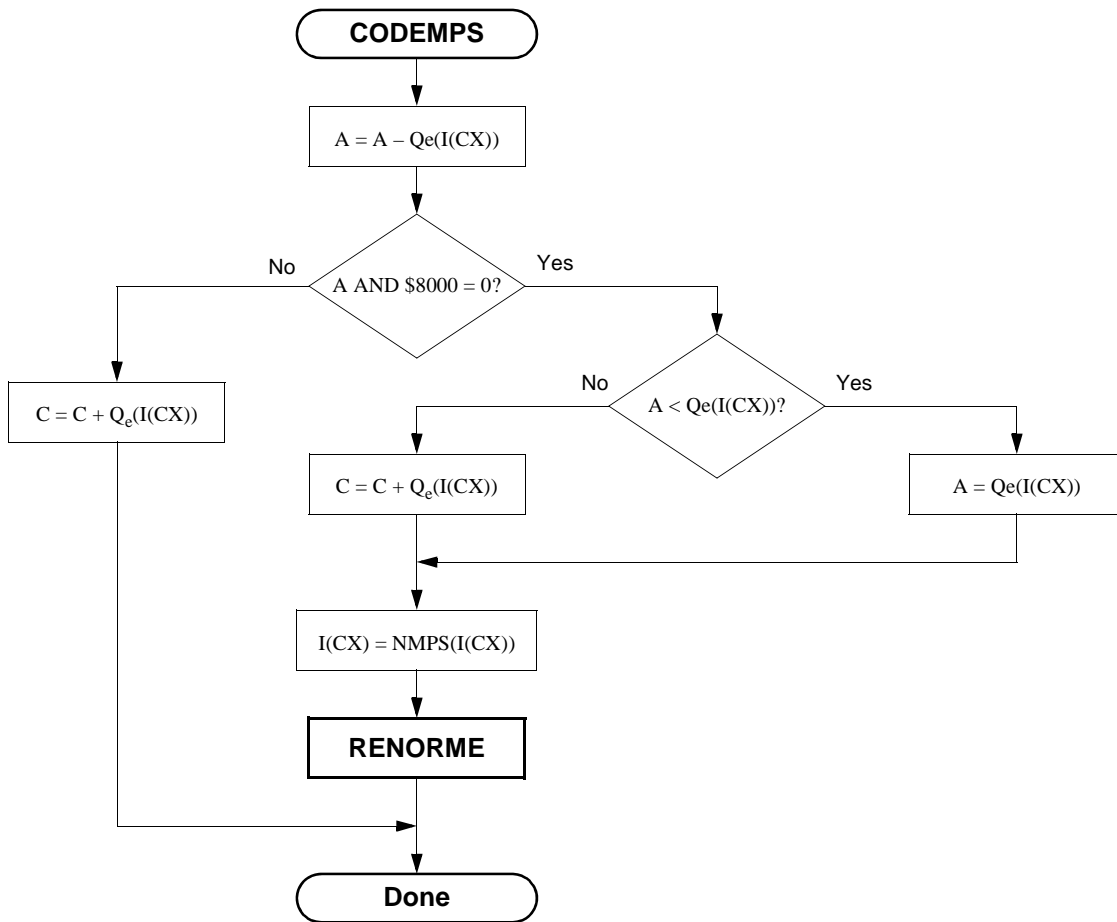
I(CX) = NMPS(I(CX))

RENORME

Done

Figure C-7.   CODEMPS Procedure with Conditional MPS/LPS Exchange
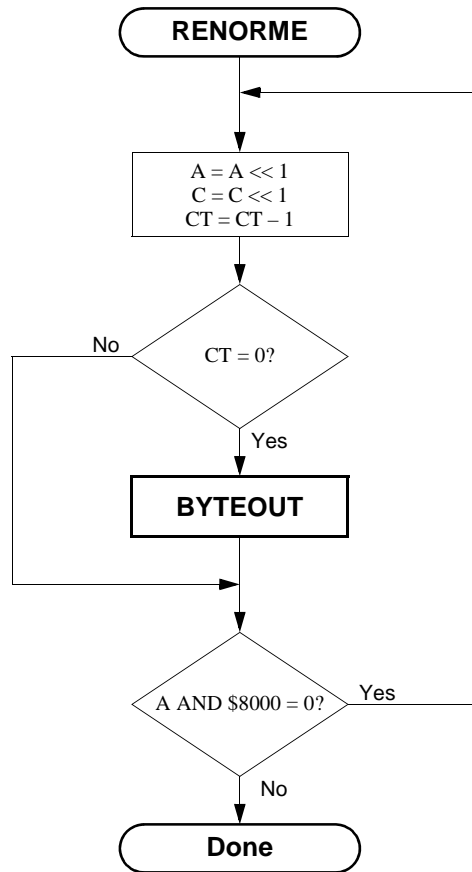
Figure C-8. Encoder Renormalization Procedure
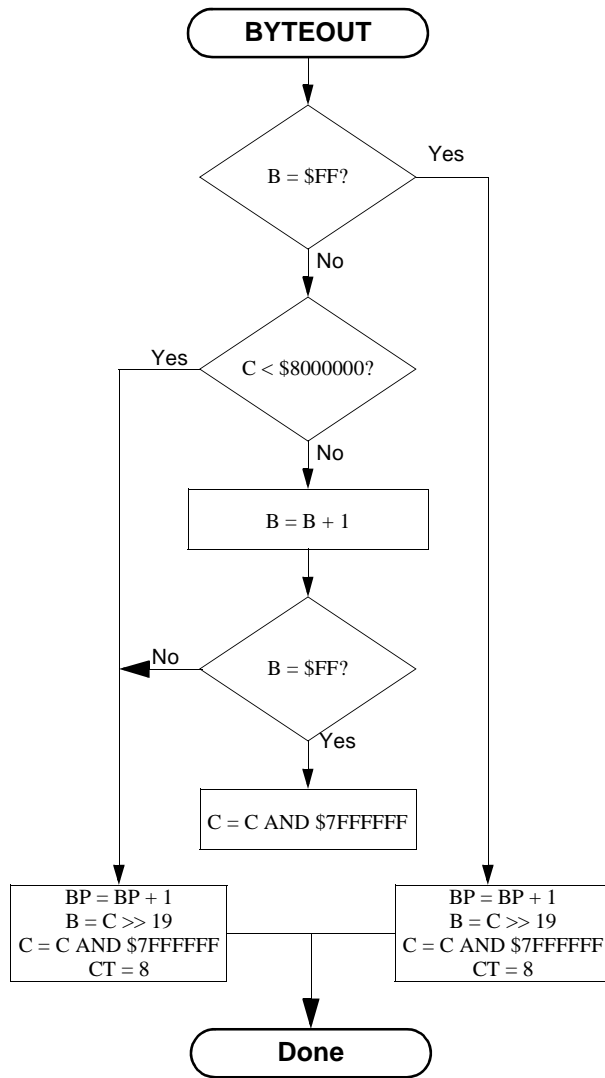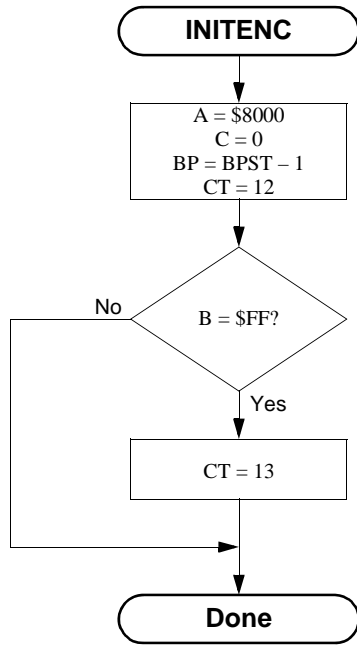
Figure C-9. Encoder BYTEOUT Procedure

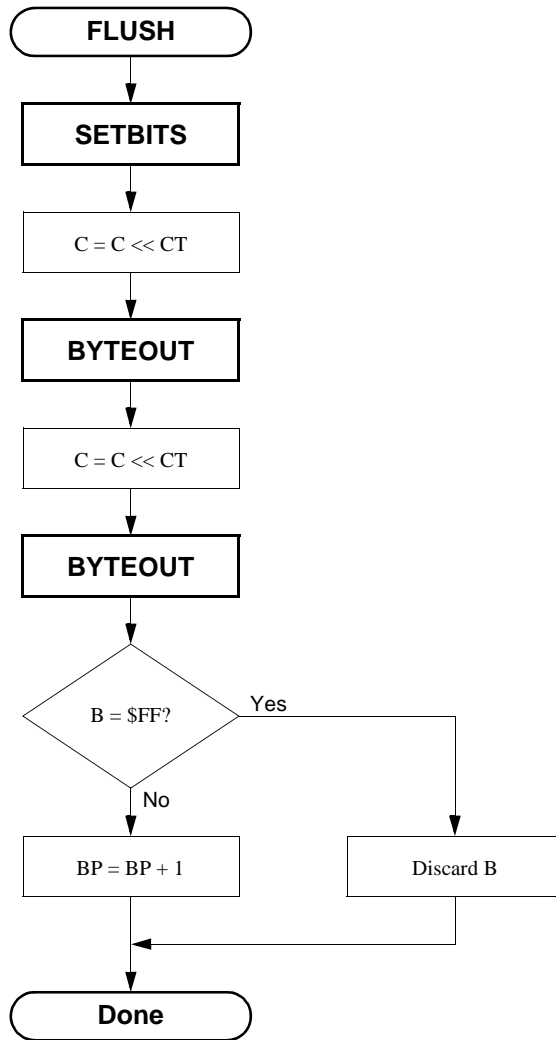**Figure C-10. Encoder Initialization**

Figure C-11. FLUSH Procedure
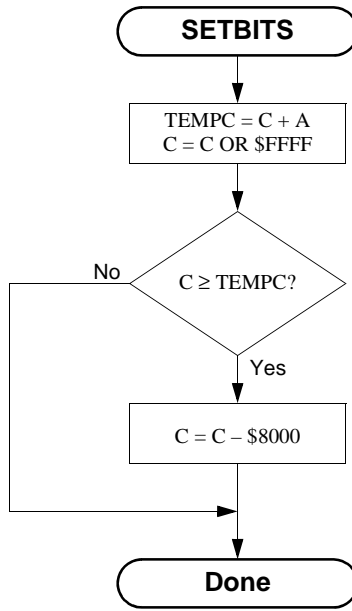
Figure C-12.   Setting the Final Bits in the C Register

Table C-2. Q_e Values and Probability Estimation

Index	Q_e_Value			NMPS	NLPS	SWITCH
	Hexadecimal	**Binary**	**Decimal**			
0	0x5601	0101 0110 0000 0001	0.503 937	1	1	1
1	0x3401	0011 0100 0000 0001	0.304 715	2	6	0
2	0x1801	0001 1000 0000 0001	0.140 650	3	9	0
3	0x0AC1	0000 1010 1100 0001	0.063 012	4	12	0
4	0x0521	0000 0101 0010 0001	0.030 053	5	29	0
5	0x0221	0000 0010 0010 0001	0.012 474	38	33	0
6	0x5601	0101 0110 0000 0001	0.503 937	7	6	1
7	0x5401	0101 0100 0000 0001	0.492 218	8	14	0
8	0x4801	0100 1000 0000 0001	0.421 904	9	14	0
9	0x3801	0011 1000 0000 0001	0.328 153	10	14	0
10	0x3001	0011 0000 0000 0001	0.281 277	11	17	0
11	0x2401	0010 0100 0000 0001	0.210 964	12	18	0
12	0x1C01	0001 1100 0000 0001	0.164 088	13	20	0
13	0x1601	0001 0110 0000 0001	0.128 931	29	21	0
14	0x5601	0101 0110 0000 0001	0.503 937	15	14	1
15	0x5401	0101 0100 0000 0001	0.492 218	16	14	0
16	0x5101	0101 0001 0000 0001	0.474 640	17	15	0
17	0x4801	0100 1000 0000 0001	0.421 904	18	16	0
18	0x3801	0011 1000 0000 0001	0.328 153	19	17	0
19	0x3401	0011 0100 0000 0001	0.304 715	20	18	0
20	0x3001	0011 0000 0000 0001	0.281 277	21	19	0
21	0x2801	0010 1000 0000 0001	0.234 401	22	19	0
22	0x2401	0010 0100 0000 0001	0.210 964	23	20	0
23	0x2201	0010 0010 0000 0001	0.199 245	24	21	0
24	0x1C01	0001 1100 0000 0001	0.164 088	25	22	0
25	0x1801	0001 1000 0000 0001	0.140 650	26	23	0
26	0x1601	0001 0110 0000 0001	0.128 931	27	24	0
27	0x1401	0001 0100 0000 0001	0.117 212	28	25	0

Table C-2. Q_e Values and Probability Estimation (Continued)

Index	Q_e_Value			NMPS	NLPS	SWITCH
	Hexadecimal	Binary	Decimal			
28	0x1201	0001 0010 0000 0001	0.105 493	29	26	0
29	0x1101	0001 0001 0000 0001	0.099 634	30	27	0
30	0x0AC1	0000 1010 1100 0001	0.063 012	31	28	0
31	0x09C1	0000 1001 1100 0001	0.057 153	32	29	0
32	0x08A1	0000 1000 1010 0001	0.050 561	33	30	0
33	0x0521	0000 0101 0010 0001	0.030 053	34	31	0
34	0x0441	0000 0100 0100 0001	0.024 926	35	32	0
35	0x02AI	0000 0010 1010 0001	0.015 404	36	33	0
36	0x0221	0000 0010 0010 0001	0.012 474	37	34	0
37	0x0141	0000 0001 0100 0001	0.007 347	38	35	0
38	0x0111	0000 0001 0001 0001	0.006 249	39	36	0
39	0x0085	0000 0000 1000 0101	0.003 044	40	37	0
40	0x0049	0000 0000 0100 1001	0.001 671	41	38	0
41	0x0025	0000 0000 0010 0101	0.000 847	42	39	0
42	0x0015	0000 0000 0001 0101	0.000 481	43	40	0
43	0x0009	0000 0000 0000 1001	0.000 206	44	41	0
44	0x0005	0000 0000 0000 0101	0.000 114	45	42	0
45	0x0001	0000 0000 0000 0001	0.000 023	45	43	0
46	0x5601	0101 0110 0000 0001	0.503 937	46	46	0