AN EFFICIENT ARCHITECTURE FOR CONTEXT-BASED ARITHMETIC CODING

Parvin Asadzadeh, Omid Fatemi
Department of Electrical & Computer Engineering
Tehran University
Tehran, Iran
plejades57@yahoo.com, omid@fatemi.net

Abstract

Significant progress has recently been made in loss-less image compression using discrete wavelet transforms. The overall performance of these schemes may be further improved by properly designing of efficient entropy coders. In this paper, we describe an efficient architecture for the context-based arithmetic coding in continuous-tone, color and multi-component digital still images. Optimizations have been made in our proposed architecture to reduce accesses to memories. Our Proposed architecture can be used for JPEG2000 image compression system.

Keywords: JPEG2000, context-based arithmetic coding, Coefficient Bit Modeler, EBCOT Binary Arithmetic Coder, BAC.

1. Introduction

With the continual expansion of multimedia and Internet applications, the needs and requirements of the technologies used, grew and evolved. With the increasing use of multimedia technologies, image compression requires higher performance as well as new features. To address this need in the specific area of still image encoding, a new standard is developed, the JPEG2000.

JPEG2000 is a new image-encoding standard that provides a feature set vital to many high end and emerging applications. JPEG2000 provides high compression with image quality superior to all existing standard encoding techniques. Its high compression capability is due to its sophisticated entropy coding mechanism used after wavelet transform. Error resilience, manipulation of images in compressed domain, region of interest coding, non-iterative rate control etc., are some of the important features of the JPEG2000 standard. All these features are possible due to adaptation of the Discrete Wavelet Transform (DWT), and intra-subband entropy coding along the bit planes using a combination of Bit Plane Coder (BPC) and a Binary Arithmetic Coder (BAC) in the core algorithm. All three core blocks, namely, the DWT, BPC and BAC blocks are computational and memory intensive blocks of the JPEG2000 standard. Of these blocks, both BAC and BPC are control intensive and have to be performed in a sequential fashion. Furthermore, memory accesses are substantial in EBCOT [4] implementation of BPC, which is used in JPEG2000 image compression system. We can conclude that specialized hardware implementation of BPC and BAC is required for a high performance JPEG2000 kernel.

This paper is organized as follows: Section 2 describes the context-based arithmetic coding. Section 3 describes the Coefficient Bit Modeling and Arithmetic Entropy Coding of JPEG2000 standard. Our proposed architecture is described in Section 4. Simulation results are given in Section 5 followed by conclusion in Section 6.

2. Context-based Arithmetic Coding

Arithmetic coding is a loss-less data compression technique that can achieve entropy limit for reasonably long data sequences. While its performance is superior to other entropy coders [1], it is more complex and not easily amenable to hardware implementation.

The arithmetic data compression technique encodes data by creating code string, which represents a fractional value on the number line between 0 and 1. On each recursion of the algorithm only one symbol is encoded. The algorithm successively partitions an interval of the number line between 0 and 1 in proportion to input symbol probabilities, and retains one of the partitions as a new interval.
Binary arithmetic coding (BAC) is a popular implementation of arithmetic coding that is applied to data sequences with only two symbols (0 and 1), thereby making it easier to implement both in hardware and software. Also, in most cases, the adaptive variant of arithmetic coding is used [2], [3], in order to take advantage from high order dependencies with the use of conditioning contexts.

The performance of arithmetic coders depends mainly on the estimation of the probability model, which the coder will use. The coder can achieve an average output code length very close to the entropy corresponding to the probability model it utilizes. Therefore, if the probability model accurately reflects the statistical properties of the input, arithmetic coding will approach the entropy of the source. Thus, a scheme employing adaptive calculation of the probability will be better than a non-adaptive scheme, as it will allow a better approximation to the true statistics of the data.

Context modeling provides estimates of conditional probabilities of the coding symbols. Utilizing suitable context models, given inter-symbol redundancy can be exploited by switching between different probability models according to already coded symbols in the neighborhood of the current symbol to encode. An example of an efficient context modeling is JPEG2000 coefficient bit modeling which uses 19 contexts to estimate probability of input symbols.

3. JPEG2000 Encoder

As depicted in Figure 7, the core structure of the JPEG2000 encoder follows a typical sequence of operations used in a transform coding scheme, which consists of transformation, quantization and entropy coding. The entropy coding process is grouped into two tiers. In the tier1 encoder, the quantized transform coefficients associated with each subband are arranged into rectangular blocks called code-blocks. Then, a bit-plane coding (BPC) technique with three coding passes is applied to each code-block, and the symbols that it produces are coded using an adaptive binary arithmetic coder (BAC). In the tier2 encoder, the inclusion and the order of appearance of bit-plane coding passes along with the actual coding pass data are assembled together to form the final compressed data. In this paper, we describe the tier-1 encoder functionality and propose an efficient hardware implementation.

3.1. Coefficient Bit Modeling

The bit-plane coding technique used by JPEG2000 codec (coder-decoder) is based on the algorithm that was proposed by David Taubman [4] called Embedded Block-based Coding with Optimized Truncation or EBCOT in short. The summary of this algorithm is given as follows.

The transform coefficients are arranged into rectangular blocks within each sub-band, called code-blocks. These code-blocks are then coded a bit-plane at a time starting from the most significant bit-plane with a non-zero element to the least significant bit-plane. For each bit-plane in a code-block, a special code-block scan pattern is used for each of three coding passes. The BPC works on stripes of four elements along the rows and the code block scan is carried from left to right. Two modes of coding are possible as shown in Figure 1.

![Figure 1 Scan Patterns](image)

Our proposed architecture implements the regular mode, which is generally used in image compression applications and outperforms the Vertical casual mode [5].

3.1.1. Coding Passes. Three different coding passes are performed on each bit plane. These passes are Significance propagation Pass (SPP), Magnitude Refinement Pass (MRP) and Cleanup Pass (CP). In each pass, only a part of the bit-plane is coded and each bit position is coded only once by one of the three passes. For each pass contexts are created which are provided to the arithmetic coder along with the bit stream. Each coefficient in a code-block has an associated binary state variable called its significance state. Significance states are initialized to 0 (coefficient is insignificant) and may become 1 (coefficient is significant) during the course of the coding of the code-block. Significance propagation pass includes all samples which are currently insignificant, but have at least one immediate neighbor that has been found to be significant. Magnitude refinement pass includes samples which are already significant. Finally Cleanup pass includes all samples skipped in the first two passes.

3.1.2. Primitives. Four kinds of codings are performed in Bit Plane Coding of JPEG2000 and contexts are created according to significant state values of eight neighboring (V0,V1,H0,H1,D0,D1,D2,D3) coefficients as shown in...
Zero Coding (ZC) – In ZC, the eight surrounding neighbor coefficients of a current coefficient (shown as X in Figure 2) are used to create 9 contexts (2-10) based on how many and which ones are significant. If a coefficient is significant then it is given a 1 value for the creation of the context, otherwise it is given a 0 value. The mapping to the contexts also depends on code-block sub-band.

Magnitude Refinement Coding (MRC) – The MRC uses 3 contexts (16-18) which are determined by the summation of the significance state of the horizontal, vertical, and diagonal neighbors. Further, it is dependent on whether this is the first refinement bit or not.

Run-Length Coding (RLC) – The RLC uses only 2 contexts (0-1). This coding process is invoked only at the beginning of a column if the significant state of all the 8 neighbors is 0 for all four bits of the column. If none of the bits in the column is significant, context 0 with data 0 is used. Otherwise, context 0 with data 1 is used. This is followed by MSB and LSB of Zero Index of the bit position (00-11), which contains the first 1 bit of the column. Context 1 is used for ZI bits. And the remaining bits are coded by Zero Coding described before.

Sign Coding (SC) – The SC uses the remaining 5 contexts (11-15), which are produced in a two-step process. In the first step, the signs and significant states of the horizontal and vertical neighbors are used to form the horizontal and vertical ‘h- and v-contributions’ and an ‘xor’ bit. In the second step, context is formed from the two contributions and data is formed by exclusive OR operation of the sign bit and the xor bit.

3.2. Arithmetic Entropy Coding

In JPEG2000, the decision (D) and context (CX) pairs produced by the bit-plane coding primitives discussed in the last section are processed together to produce the compressed data (CD). Coding is performed using the MQ coder [6]. The MQ coder is an adaptive, binary arithmetic coder (BAC), characterized by multiplier-free approximation, renormalization-driven probability estimation and bit-stuffing, all as in the Q-coder [7].

The recursive probability interval subdivision is the basis for the binary arithmetic coding process. With each binary decision the current probability interval is divided into two sub-intervals, and the code string is modified (if necessary) so that it points to the base of the probability sub-interval assigned to the symbol which occurred. In the partitioning of the current interval into two sub-intervals, the sub-interval for the more probable symbol (MPS) is ordered above the sub-interval for the less probable symbol (LPS). Therefore, when the MPS is coded, the LPS sub-interval is added to the code string. This coding convention requires that symbols be recognized as either MPS or LPS, rather than 0 or 1. Consequently, the size of the LPS interval and the sense of the MPS for each decision must be known in order to code that decision. Also, in order for the 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 coding operations are done using fixed precision integer arithmetic and using an integer representation of fractional values in which 0x8000 is equivalent to decimal 0.75. The interval register (A) is kept in the range 0.75<=A<=1.5 by doubling it (renormalization) whenever the integer value falls below 0x8000.

The code register (C) is also doubled each time A is doubled. Periodically, to keep C from overflowing, a byte of data is removed from the high order bits of the C register and placed in an external compressed data buffer. Carry-over into the external buffer is resolved by a bit stuffing procedure.

In JPEG2000, the coefficient bit modeler performs the statistical modeling by providing the BAC with context/data pairs. The context is used to index into LPS probability value (Qe) table. 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 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. 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 Qe probabilities. A portion of this table is presented in Table 1.

This table also includes associated next states for each type of renormalization. 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 if an MPS occurs, and the SWITCH value indicates if the sense of the MPS must be inverted.

4. Our Proposed Architecture

In this section our proposed architecture is presented. Two main parts of the architecture, i.e., coefficient bit modeler (EBCOT) and the Binary Arithmetic Coder (BAC) are explained in the following subsections.

4.1. Coefficient Bit Modeler (EBCOT) Architecture

Figure 8 shows our proposed architecture for the EBCOT encoder. The architecture consists of the following key building blocks:

4.1.1. Memory Building Blocks. Six memory building blocks each of size r*c, where r is the number of rows and c is the number of columns in the code-block are used.

- **Sign and Bit Memories** contain the sign of the current code block coefficients and the current magnitude bit of coefficients respectively.
- **State Memory** contains the current significant state of the coefficients. It is 1 when the first most significant 1 bit of the coefficient is met and 0 otherwise.
- **MRP Memory** contains the magnitude refinement primitive state of the coefficients. The corresponding bit of each coefficient in this memory is set the first time the magnitude refinement primitive is used.
- **MRC and CUC Memories** During the significance propagation pass, we have to check all bits of the whole code-block to decide which one must be coded in this pass. To avoid repeating the same for the other two passes, during the significance propagation pass, if the state of the current coefficient is 1, we set its corresponding bit in MRC memory to show that this bit must be coded in the coming Magnitude refinement pass, and if its state is zero and is not coded in significance propagation pass, we set its corresponding bit in CUC memory. ‘s’ flip-flops are associated with each of these memories, where s is the number of stripes in each code-block. The CUC flip-flops are initialized to 1 when the coding of a new code-block is started. And others all initialized to 0. During the Significance propagation pass, when data is written into these memories, the corresponding flip-flops are set to indicate in which stripe of the code-block there is a bit required to be coded in the desired pass. By applying this, we have avoided long delay of checking the rules for all bits of the code-block in order to find the appropriate bit to encode it. By using just one additional memory in our proposed architecture, higher speed compared to [5] has been achieved.

4.1.2. Context Combinational Block. The inputs to this block are 9-bit data signal, which has different meanings in different primitives and outputs are the 5-bit context in addition to the decision bit which both are input of the entropy coder block, for further compression. The contexts for different combinations of neighboring states can be extracted using specific tables, which are defined in standard [6]. In order to reduce the memory usage, we have implemented this part in combinational circuits. We propose logic equations for generating the context. Below are the example equations for LL sub-band in zero coding pass. These equations are extracted from defined tables of the JPEG2000 standard [6].

**Zero Coding**

\[
d(8:0) = (\text{bit}, D0, H0, D3, V0, V1, D1, H1, D2); \\
h2 = d(6)*d(1); h1 = d(6)*d(1)+d(1)*d(6); h0 = \\
/((d(6)+d(1)); \\
v0 = /(d(4)+d(3)); v>=1 = d(4) + d(3); \\
d0 = /d(7) */d(5) */d(2) */d(0); \\
d1=d(7)*d(5)*d(7)*d(5)*d(7)*d(7)*d(7)*d(7)*d(2)*d(7) \\
*/d(7)*d(7)*d(0)*d(7)*d(7)*d(7)*d(7)*d(7)*d(7); \\
d>=1 = d(7)+d(5)+d(2)+d(0); \\
context(4) = 0; \\
context (3) = h2 + h1*v>=1 + h1*v0*d>=1; \\
context (2) = h1*v0*d0 + h0*v2 + h0*v1 + h0*v0*d>=2; \\
context (1) = h2 + h1*v0*d0 + h0*v2 + h0*v0*d1 + h0*v0*d0; \\
context (0) = h1*v>=1 + h1*v0*d0 + h0*v1 + h0*v0*d1; \\
decision = d(8);
\]

4.1.3. Shift Registers. Figure 3 shows the order in which the bits of a middle stripe of a code-block are scanned (numbered from 1 to 16). The functionality of all the registers in the following sections is explained based on this order of coding.

```
<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Figure 3** Coding order for the middle stripe
**State and Sign Registers** - These registers are 18 bits wide and contain significant state variables and sign bits of three successive columns respectively, with data arranged as shown in Figure 4, Figure 5. It is assumed that the bit in position ‘1’ is being coded.

<table>
<thead>
<tr>
<th>D0</th>
<th>H0</th>
<th>D3</th>
<th>V0</th>
<th>V1</th>
<th>D1</th>
<th>H1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>a</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>e</td>
<td>b</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

*Figure 4 State Register*

<table>
<thead>
<tr>
<th>H0s</th>
<th>V0s</th>
<th>V1s</th>
<th>H1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>f</td>
</tr>
</tbody>
</table>

*Figure 5 Sign register*

Two zero-detectors are connected to selected outputs of state register, one for examining Run-Length Coding condition and the other is used for creating context during Magnitude refinement pass. And eight-bit selected outputs of this register are read and available to the combinational block to form the context when zero coding is used. The four selected outputs of Sign register are also read and are available to the Sign Context combinational block.

**4-bit Shift Registers (REG1-9)** – We use three 4-bit shift registers (REG1-3) to store data from State memory and three other ones (REG5-7) to store sign bits. In both cases, the target register is selected by the controller. And at each cycle a bit form one of these registers is shifted into the State or Sign Register described above. The serial input of another register (REG4) is the serial-out output of the State Register and the 4-bit data is read from this register and written to State memory. Two other 4-bit registers (REG8-9) are loaded during significant propagation pass and are written to CU and MR memories periodically.

**Code, Bit and MRP Registers** – These registers are 4-bit wide registers. Bit register stores current bit of each coefficient during all three passes and is loaded from bit memory. During Cleanup Pass, data from CUC memory is written into Code register. A 1-detector is attached to the Code Register which determines whether all 4 bits are going to be coded in this pass. If so, RLC condition is checked and if is satisfied then the zero detector attached to the output of the Bit Register determines if all bits of the column are zero or not. And an encoder generates the bit position of the first 1-value bit in the column, which is needed during RL-Coding as is described before. After coding the first 1-bit, other bits of the current column are zero coded as described before. In the case that RLC condition is not met, all 4 bits are coded as in Zero Coding. During Magnitude refinement pass, data from MRC memory is written to Code Register and the bits, which must be coded are detected and the required information is read from MRP and State Registers and sent to MR Context combinational block to create the context.

**4.1.4. Pass Detector.** This detector receives the output values of flip-flop arrays and decide which stripe in which pass must be scanned. The outputs of this detector goes directly to the ‘address generator’ to generate the memory addresses.

**4.1.5. Controller.** This state machine consists of 32 states. It produces memory read and write signals, register control signals, and multiplexers select signals.

**4.2. Binary Arithmetic Coder Architecture**

Figure 9 shows our proposed architecture for the Binary Arithmetic Coder (BAC). The architecture consists of the following key building blocks:

**4.2.1. Lookup Table.** For each possible 19 contexts, an index is stored. The index to this table is ‘context’ which is provided from Coefficient Bit Modeler. This index is then provided to other lookup tables to read LPS probability value (Qe) along with NMPS, NLPS and SWITCH values for each coming symbol.

**4.2.2. Registers.** Two 32-bit registers are used in this architecture. A register is the interval register and C register which is the code register. The registers structures are provided in Figure 6. The ‘a’ bits are the fractional bits in the A-register (the current interval value) and the ‘x’ bits are the fractional bits in the code register. The ‘s’ bits are spacer bits which provide useful constraints on carry-over, and the ‘b’ bits indicate the bit positions from which the compressed bytes of the data are removed from the C-register. the ‘c’ bit is a carry bit.

<table>
<thead>
<tr>
<th>MPS</th>
<th>LPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-register</td>
<td>0000 cbbb bbbb bsss</td>
</tr>
<tr>
<td>A-register</td>
<td>0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

*Figure 6 Register Structures*
4.2.3. Barrel Shifters. Barrel-shifters are used so that A and C are shifted in one clock cycle when renormalization is needed.

4.2.4. Encoder. A priority Encoder is used to determine the position of the first ‘1’ bit in the A-register. The result is provided to the controller so that the number of required shifts for both A and C-registers during renormalization process can be provided to the barrel shifters.

4.2.5. Adder/Subtractor. An Adder/Subtractor is used to update interval register (A) when a new symbol is coded. And update the code register (C) when an MPS occurs.

4.2.6. Output Buffer. Selects and outputs the desired byte from the code register. Also the bit stuffing procedure is done whenever required.

4.2.7. Controller. This state machine consists of 16 states. It produces lookup table read signals and registers, barrelshifters, output buffer control signals, and multiplexers select signals.

5. Simulation Results

We have implemented our proposed architecture using synthesizable part of VHDL and it is simulated in Modelsim environment. Figure 10 shows a part of Magnitude refinement pass results in a small period. The outputs are 1-bit decision ‘decision’ and 5-bit context ‘cx’ in an integer representation and the compressed data is shown as buffer outputs. Simulation results show that the proposed architecture is able to output the compressed bit stream in minimum time.

6. Conclusion

An efficient architecture for the JPEG2000 Entropy Coder has been proposed. The functionality of different blocks of this architecture is described. Our proposed architecture can be used for any code-block size even if the code-block height is not a factor of 4.

References


Figure 7 JPEG2000 Encoder Block Diagram
Table 1  A portion Table C-2 from the JPEG2000 Standard

<table>
<thead>
<tr>
<th>Index</th>
<th>Qe Value</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>NMPS</th>
<th>NLPS</th>
<th>SWITCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>0.099 634</td>
<td>0001 0001 0000 0001</td>
<td>0001 0001 0000 0001</td>
<td>0.099 634</td>
<td>30</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0.063 012</td>
<td>0000 1010 1100 0001</td>
<td>0000 1010 1100 0001</td>
<td>0.063 012</td>
<td>31</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0.057 153</td>
<td>0000 1001 1100 0001</td>
<td>0000 1001 1100 0001</td>
<td>0.057 153</td>
<td>32</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0.050 561</td>
<td>0000 1000 1010 0001</td>
<td>0000 1000 1010 0001</td>
<td>0.050 561</td>
<td>33</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>0.030 053</td>
<td>0000 0101 0010 0001</td>
<td>0000 0101 0010 0001</td>
<td>0.030 053</td>
<td>34</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8  Coefficient Bit Modeler Block Diagram

Figure 9  Binary Arithmetic Coder Architecture
Figure 10  Modelsim Simulation Results