System Level Design of the Adaptive Arithmetic Encoding Used in the JPEG 2000 Standard

Ali M. Reza

Department of Engineering, Electrical Engineering Section
United States Coast Guard Academy
New London, CT 06320, U.S.A.

Abstract: Hardware implementation of the JPEG 2000 image compression standard for real-time image compression requires implementation of the adaptive arithmetic encoder among other related algorithms. In this work, a system level design for the adaptive arithmetic encoder used in the JPEG 2000 standard is proposed. This design is based on the assumption that the bit-sequence and its corresponding context, obtained through bit modelling for the encoder, are input to the system in their proper order. The output bit-sequence is also used as input to the encoder bit modelling. The original bit-sequence provides the context based on the assumption that the first pass is the cleanup pass and the first context is the Run Length context. All other contexts are recursively evaluated as the arithmetic encoder encodes the bit sequence. The final encoding is lossless and results in perfect recovery of the original block image when proper decoding algorithm is used.

1. Introduction

Compression and decompression of the real-time image sequences require hardware implementation of compression as well as decompression algorithms in the corresponding hardware platforms. Coding in the JPEG 2000 standard is based on the adaptive arithmetic coding [1]. During the compression stage, the coding is called encoding and during the decompression stage, the coding is referred to as decoding. Wavelet coefficients will first go through what is referred to as bit modelling before the encoding process can take place. We have already presented a system level design for the wavelet coefficient bit modelling used in the JPEG 2000 standard [2]. In this work, we have developed a particular system level implementation design for the corresponding adaptive arithmetic encoder. We have also used a similar approach in designing a system level implementation of the JPEG 2000 decoding algorithm as described in [3].

In this design we assume that the compressed bit-sequence and its corresponding context, obtained through bit modelling for the encoder, are properly input to the system. The final bit-stream is the output bit-sequence that is resulted from this system. Our main reference in this work is the JPEG 2000 Part I Final Committee Draft Version 1.0 dated 16 March 2000 [4]. The relevant part of this reference is Annex C. For the sake of space and brevity of our presentation, we do not repeat the discussion that is presented in this reference. Interested reader is referred to pp. 71-84 of [4], namely Section C.1 and C.2 of Annex C. In the following, whenever an explanation of the method or procedure is given it is always with the aforementioned reference.

A system level architecture capable of encoding JPEG2000 algorithm is proposed in [5]. In this work, embedded block coding with optimized truncation (EBCOT), using column-based coding architecture of tier-1 block coding engine, is discussed. This implementation is based on hardware descriptive language (HDL) and Xilinx FPGAs. Another VLSI architecture for a generic block coder with concurrent symbol processing is proposed in [6]. This approach combines bit plane coder with arithmetic coder to achieve better throughput. A pipelined binary arithmetic coder architecture is proposed in [7] in order to achieve higher rate than the conventional architectures. In this work, we propose a system level implementation that takes advantage of parallelization and trades off hardware for speed to achieve the highest possible throughput for any VLSI implementation.
Efficient hardware implementation of the arithmetic encoding requires several registers, counters, and look-up-tables (LUTs). We introduce these items along with some circuitry, as we as we foresee appropriate for this design. At first we present the overall system input/output diagram and then proceed with explanation and design of each individual elements in the system. With reference to Figure 1, the inputs to this system consist of two input lines one with one-bit word that is the input bit-sequence coming from the bit modeller and the other one is a five-bit word that represents the context of the corresponding input bit. With reference to [4], there are nineteen different contexts that are always represented by unsigned integers between 0 and 18 inclusive. Initially the context for the first compressed bit is assumed to be 18. Contexts 0 through 8 are used for significance propagation and cleanup.

![Figure 1. Overall structure of the Input/Output relation for the Adaptive Arithmetic Encoder.](image)

In the following we discuss the structure and functionality of different sub-blocks that are needed for this design.

### 2. Registers, Memories, Counters, and Look-Up-Tables (LUTS)

There is a need for two sets of random-access memories (RAMs) in order to store the values for the most probable symbol (MPS) and index (I) associated to each context. Since we only have nineteen different contexts, the size of these RAMs should be 32 (assuming that their size should be power of 2). The MPS is a one-bit word but the index (I) is a six-bit word. It should be pointed out that the values used for the index are unsigned integers between 0 and 46. Initially these two RAMs are set as shown in Figure 2. The ‘Reset’ operation is a little complex and may take several clock cycles for its completion.

![Figure 2. RAMs used for MPS and I index to be accessed dynamically and initialized as specified at the beginning of encoding of each block.](image)

In our approach, we take the maximum advantage of using look-up-tables (LUTs) for storage of previously calculated values. This design can benefit from four look-up-tables (LUTs) in order to store the next most probable symbol (NMPS or Next MPS), the next least probable symbol (NLPS), current estimate of the least probable symbol (LPS) probability,
i.e., $Q_e$ values, and the SWITCH bit. These LUTs should have a capacity of at least 47 (or 64 if the size of these LUTs are restricted to be in powers of 2). Word-lengths in these LUTs are as follows: $Q_e$ is 16-bit, NMPS and NLPS are 6-bit each, and SWITCH is 1-bit. The setup of these LUTs are explained in Figure 3.

Operation of the arithmetic encoder depends on two specific registers that are the core of the system. These registers, referred to as A-register and C-register (Code Register), are organized as shown in Figure 4. During the encoding operation we need to have a dedicated register, referred to as B-register, to hold the $b$ bits of the code register (C-register) for outputting the compressed code as needed. The B-register, not shown, is an 8-bit register that will be discussed with more details during the analysis of the C-register.

Details of these registers are decided and designed based on the kind of operations that are desired for each register. The A-register should accommodate operations like setting the register to the fixed value of $Q_e$ or 0x8000, subtracting $Q_e$ from the content of the register, comparing $Q_e$ with the content of the register, and shifting the content of the register to the left by one bit. These operations are designed in such a way that any of them can only take one clock cycle. This claim is based on the assumption that the value of $Q_e$ is not changed in the previous clock cycle and the comparison and the subtraction operations are readily

Figure 3. Structure of the four LUTs and their corresponding values.

Figure 4. Structures of the two main registers in the operation of the arithmetic encoder.
Figure 5. Structure of the A-register circuitry. The MUX (multiplexing operation block) is already registered and there is no need for a specific register for holding Data Out.

Figure 6. Structure of the C-register circuitry along with the two possible ways of getting the B-register values. Output MUXs are already registered and there is no need for a specific register to hold data.

Available at the moment that they are needed. The circuitry of the A-register is shown in Figure 5. Since in some occasions we need to know the state of the MSB of this register, the MSB value is also used as an output.

In the case of C-register, there are more operations and internal registers. The circuitry in this case is shown in Figure 6. In this case there are two registers a 32-bit register and a 8-bit register. The 32-bit register denoted as C Out represents the current value of the C-register. On the other hand, the 8-bit register, denoted as B Out represents the compressed byte one clock cycle after Get B is issued. In this design we assume that the ‘shift-left’ operation is foreseen before hand and is prepared for use right at the time that it is asked for. Therefore, our assumption is that all operations are calculated and ready for use at the input of the 8-input MUX for selection. If this is not true for any of the operations then we need to allow enough time for their preparation before accepting the content of the C-register as its final value. There are two simple operations on the B Input that is also included as part of the C-register circuitry.
There are two simple operations on the ‘B Out’ that can be carried out properly by using simple gates or digital comparators. In several situations we need to check if ‘B Out’ is equal to 0xFF. Also, in one case we need to increment B by 1 and then check if B equals to 0xFF. This last operation can be simplified by conducting a comparison between ‘B Out’ (without incrementing it) with 0xFE. These operations are explained in Figure 7.

Finally there is a need for a special counter to properly keep track of the CT parameter. This parameter keeps track of the compressed bit positions in the C-register. Operations needed for this counter are setup, decrement and increment. The setup operation works with two inputs. The ‘Set’ input, which is a 1-bit input, signals the setup operation and the ‘Set Value’ input, which is a 4-bit input, indicating the value that the counter should be set to. Block diagram of this counter is shown in Figure 8.

For registers, LUTs, RAMs and CT Counter each operation takes only one clock cycle with the assumptions stated earlier. As we proceed we address those assumptions for the more complex operations of the decoding process.

3. Circuits Operation

In this section we first present the sequence of operations in a form of a pseudo code and then we formulate their proper hardware implementation. Our assumption is that the first bit and its context is preceded by a ‘START’ signal and the last bit and its context is followed by an ‘END’ signal. Based on the JPEG 2000 standard, the sequence of encoding operations are as follows:

First, right at the time that the ‘START’ signal is issued the INITIALIZATION will be carried on. Then, the sequence of bits along with the corresponding sequence of contexts is delivered to the decoder. The speed of delivery of the bit-sequence depends on the overall speed of the encoder and is assumed to come at proper speed. This can be controlled by the way that the bits are read from memory or, if the bits are arriving at a lower speed, it is assumed that the bits enter the circuitry as soon as they become available. There might be a need for an input buffer to regulate the data delivery to the input of the decoder.

In table 1 the pseudo code of the initialization along with the main operation is presented. The circuitry for the initialization is shown in figure 9 and the circuitry for the main operation is shown in figure 10. These processes start by the ‘Start’ signal. The main process ends by the ‘End’ signal. The main process refers to other more detailed processes as discussed in the following.

One process that is referred to is the CODELPS operation. The pseudo code for this operation is shown in table 2. The corresponding circuitry for this operation is shown in
Table 1. Pseudo Code for The Main Operations

**INITIALIZATION:**
1- Set A-register to 0x8000: \( A = 0x8000 \)
2- Set C-register to 0: \( C = 0 \)
3- Reset All RAMs
4- If ‘B Out’ equals 0xFF (\( B \geq 0xFF \)),
   Set CT-counter to 13 (0xD): (CT = 0xD), Else
   Set CT-counter to 12 (0xC): (CT = 0xC)

**LOOP:**
1- Get Bit ‘D’ and its context ‘CX’
2- Use ‘CX’ as address into all RAMs to get ‘MPS’ and ‘I’
3- Use ‘I’ as address into all LUTs to get \( Q_e \), ‘NMPS’, ‘NLPS’, and ‘SWITCH’
4- Calculate \( C_{MPS} = D \cdot MPS + D \cdot MPS \)
5- If \( C_{MPS} = 1 \),
   USE CODELPS, Else
   USE CODEMPS
6- If NOT FINISHED,
   Go to LOOP, Else
   USE FLUSH operations
End LOOP

Figure 9. The circuitry for the initialization of the encoding process.

Figure 10. The circuitry of the main operation that includes the Initialization as the final ending of the encoding procedure.
Table 2. Pseudo Code for CodeLPS Operations

1- Subtract Q_e from A to get the new A: \( A \leftarrow (A - Q_e) \)
2- If \( A < Q_e \),
   Add Q_e to C: \( C \leftarrow (C + Q_e) \), Else
   Set A to Q_e: \( A \leftarrow Q_e \)
3- If ‘SWITCH’ = ‘1’ Change ‘MPS’ to its complement and store the new value into the corresponding RAM at the address given by ‘CX’.
4- Set ‘I’ to ‘NLPS’
5- CONDUCT RENORM OPERATIONS

Table 3. Pseudo Code for CODEMPS Operations

1- Subtract Q_e from A to get the new A: \( A \leftarrow (A - Q_e) \)
2- If MSB (Most Significant Bit) of A is ‘1’, Add Q_e to C: \( C \leftarrow (C + Q_e) \), Else Conduct the following:
   (a) If \( A < Q_e \),
       Set A to Q_e: \( A \leftarrow Q_e \), Else
       Add Q_e to C: \( C \leftarrow (C + Q_e) \)
   (b) Set ‘I’ to ‘NMPS’
   (c) CONDUCT RENORM OPERATIONS

The next process that is referred to in the Main process is the CODEMPS operation. The pseudo code for this operation is shown in table 3. The corresponding circuitry for this operation is shown in figure 12.

The pseudo code of the RENORM operation, which is referred to in both CODELPS and CODEMPS, is shown in table 4. The corresponding circuitry is shown in figure 13. This set of operations refers to another operation called BYTEOUT. It is also referring to itself under certain circumstances, which makes it a recursive operation under this condition.

The set of operations called BYTEOUT constitutes the stage in which compressed code is sent out whenever is ready. The pseudo code for this operation is given in table 5. The corresponding hardware circuitry for this stage is shown in figure 14.
Table 4. Pseudo Code for The Renorm Operations

**LOOP:**
1. Conduct the following:
   (a) Shift left A by 1 bit
   (b) Shift left C by 1 bit
   (c) Decrement ‘CT’ counter by 1
2. If ‘CT’ is ‘0’, CONDUCT BYTEOUT OPERATIONS
3. If MSB of A is ‘0’, go to LOOP

Table 5. Pseudo Code for BYTEOUT Operations

If $B$ equals to 0xFFFF ($B \not= 0xFFFF$),
   Conduct the RIGHT operations, Else
If ‘c’ (carry bit in the C-register) is ‘0’
   Conduct the LEFT operations, Else
   Increment $B$ by 1: $B \not= B + 1$
   If $B$ equals to 0xFFFF,
      Set ‘c’ bit to ‘0’ and conduct the RIGHT operations, Else
      Conduct the LEFT operations

RIGHT operations: All simultaneous:
   Get B1 (Use ‘c’ bit as part of the 8- bit ‘B Out’)
   Set 12 MSBs of the C-register to zeros
   Set counter ‘CT’ to 7

LEFT operations: All simultaneous:
   Get B2 (Use all the ‘b’ bits as the 8- bit ‘B Out’)
   Set 13 MSBs of the C-register to zeros
   Set counter ‘CT’ to 8

RIGHT operations: All simultaneous:
Finally, at the end of the code block, the procedure that is referred to as FLUSH will be conducted. The pseudo code for this operation is given in Table 6. The corresponding hardware circuitry is shown in Figure 15.

Table 6. Pseudo Code for FLUSH Operations

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set a temporary register T to A + C : T = A + C</td>
</tr>
<tr>
<td>2</td>
<td>Set 16 LSBs of the C register to ones</td>
</tr>
<tr>
<td>3</td>
<td>If C (\leq T), then subtract 0x8000 from C : C (\rightarrow) (C - 0x8000)</td>
</tr>
<tr>
<td>4</td>
<td>Shift Left C by ‘CT’ bits</td>
</tr>
<tr>
<td>5</td>
<td>Conduct BYTEOUT OPERATIONS</td>
</tr>
<tr>
<td>6</td>
<td>Shift Left C by ‘CT’ bits</td>
</tr>
<tr>
<td>7</td>
<td>Conduct BYTEOUT OPERATIONS</td>
</tr>
<tr>
<td>8</td>
<td>If B equals to 0xFF (B = 0xFF), discard B; else use B as the final output byte</td>
</tr>
</tbody>
</table>

4. Conclusion

System level design for the hardware implementation of the adaptive arithmetic encoder used in the JPEG 2000 image compression standard is presented. This implementation is one of the significant parts of the complete hardware implementation of the JPEG 2000 encoding standard. In this implementation, the bit-sequence and its corresponding context, obtained through the bit modelling for the encoder, are used as input to the system. The output of this implementation is the compressed bit-sequence used for storage or transmission of the image. The final encoding is lossless and results in perfect recovery of the original block image when used in conjunction with the corresponding JPEG 2000 decoder.
References

Disclaimer and Note
The views expressed herein are those of the author and are not to be construed as official or reflecting the views of the Commandant, the U.S. Coast Guard, the Department of Homeland Security, or any agency of the U.S. Government.

Ali M. Reza is a member of the Engineering Faculty at the U.S. Coast Guard Academy. He was formerly a tenured faculty member of the Electrical Engineering and Computer Science Department at the University of Wisconsin-Milwaukee where he is currently a Professor Emeritus. Dr. Reza's specialty is in the general area of Signal Processing. He has conducted research in array signal processing, signal detection, nonlinear filtering, image processing, and pattern recognition. In his research, he has applied new emerging technologies, including artificial neural networks, fuzzy systems, and genetic algorithms. Dr. Reza is also interested in hardware implementation of signal and image processing algorithms. Dr. Reza has more than eighty published articles in refereed journals and conference proceedings.