

A MULTI-BIT BINARY ARITHMETIC CODING TECHNIQUE

Kishore Andra*, Tinku Acharya*⁺ and Chaitali Chakrabarti*

* Arizona State University, Tempe, Arizona, 85287, USA

⁺ Intel Corporation, Chandler, Arizona, 85226, USA

ABSTRACT

In this paper, we propose a new methodology for binary arithmetic coding which reduces the number of arithmetic operations significantly at the expense of a mild reduction in compression ratio. We achieve this by (i) considering a two symbol non-overlapping window and not coding the second symbol if both of them are Most Probable Symbols and (ii) moving the majority of computations to the Least Probable Symbol path. As a result, we reduce the additions/subtractions required by 60-70%, with a loss of compression ratio of about 1-3% compared to the Q-coder. This reduction in computational complexity makes the proposed technique particularly suitable for low-power VLSI implementation. In this paper, we have described the proposed algorithm and analyzed the results. We have also described a VLSI architecture capable of carrying out the algorithm.

1. INTRODUCTION

Arithmetic coding is a lossless data compression technique that can achieve entropy limit for reasonably long data sequences [1]. While its performance is superior to Huffman coding, it is more complex and not easily amenable to hardware implementation. Binary Arithmetic Coding (BAC) is a popular implementation of the 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. Of the various implementations of BAC, the Q-coder BAC [2,3,4,5] is the most popular one. In the Q-coder, the binary symbols are classified into the More Probable Symbol (MPS) and the Less Probable Symbol (LPS). These symbols are assigned to 0 or 1 dynamically based on the composition of the neighboring binary pixels. Our study of typical binary images show that the number of appearances of LPS is around 10%. Thus the number of computations can be significantly reduced if majority of the arithmetic operations are done only when an LPS is encountered and if we can skip coding as many MPS's as possible. In this paper, we propose an algorithm that has both these properties in order to reduce the number of arithmetic computations significantly (~60 - 70%) without incurring significant loss in compression ratio (compared to the Q-coder). The large reduction in the number of arithmetic operations makes this technique particularly suitable for low power implementation in both hardware and software.

In the next section we briefly describe the proposed coder and its similarities and differences with the Q-coder. In section 3 the new methodology for the model is explained. The pseudo code for the encoder, rules used in the decoder and a numerical example for encoding and decoding for a ACA1 coder are also

provided. Performance of the algorithm is discussed in section 4. In section 5, we propose a VLSI architecture to carry out the proposed algorithm.

2. ENCODING AND DECODING

In the Q-coder, the dynamic allocation of MPS or LPS is done based on the context (we have considered 10 previously encoded bits). The context is also used to calculate the probability of the next symbol being LPS. We propose changes in the original Q-coder in order to reduce the number of arithmetic operations. We call the proposed methodology the ACA coder.

The basic encoder and decoder for Q-coder BAC are described below. Here 'A' is the current interval, 'C' is the starting point of the interval and 'q' is the probability that a LPS occurs next. The product $A \cdot q$ is approximated to a value Q by maintaining A near unity.

Encoder :	Decoder :
If (sym = MPS)	If (C >= Q)
C = C + Q;	sym = MPS;
A = A - Q;	C = C - Q; A = A - Q;
else	else
A = Q;	sym = LPS; A = Q;

We propose to change C in the LPS path instead of the MPS path in the above basic encoder/decoder. This significantly reduces the number of arithmetic computations since the probability of appearance of LPS is significantly less than MPS.

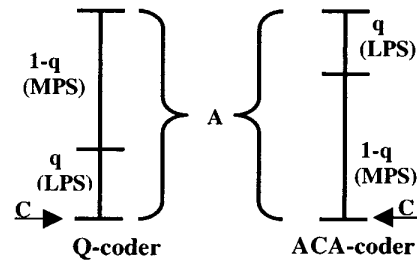


Fig.1: Interpretation of the parameters.

Based on the interpretation of the intervals described in [5] (shown in Fig.1), the encoder and decoder operations can be described as follows.

Encoder :	Decoder :
A = A - Q;	A = A - Q;
If (sym = LPS)	If (C >= A)
C = C + A;	sym = LPS;
A = Q;	C = C - A; A = Q;
	else sym = MPS;

This scheme could not be adopted in the Q-coder BAC, since the data dependencies prevent the operations from being performed in parallel. However, in the proposed method, the number of additions or subtractions needed to encode or decode the binary images is reduced by a factor of 2. As a result, the effect of the dependencies can be compensated and they do not negatively affect the throughput. It should be noted that, although the proposed methodology has been explained using the Q-coder BAC, it is applicable to any BAC algorithm that does not change 'C' in its MPS path.

3. NEW METHODOLOGY

We use a two symbol non-overlapping window for encoding/decoding operations. As the MPS's are coded predominantly, most of the time both the symbols in the window would be MPS's. This means that we can avoid coding the second MPS (and thus reducing a subtraction operation).

For ease of explanation, let us assume that 0 is the MPS and 1 is the LPS. So the sequence "00" would be coded as just '0'. If any of the other three combinations (01, 10, 11) of symbols occur in the window, then we code both the symbols. The four possible combinations for a two symbol non-overlapping window and their corresponding encoded symbols are

00 → 0; 01 → 01; 10 → 10; 11 → 11.

If this scheme is used, then at the decoder, a '0' not preceded by a '1' would imply "00". Thus, the code bit and the corresponding output would be

0 → 00; 01 → 001; 10 → 10; 11 → 11

In other words, if the code bit '0' or "01" is encountered, then the corresponding outputs would be "00" and "001". As observed from the above example, the "01" combination is an exception in that, '0' here means just '0' and not "00". This exception can possibly be handled by deciding whether a '0' implies a "00" or a '0' after the next symbol has been decoded. If we handle the exception in this manner, then the code bit and the corresponding output would be

0 → 00 01 → 01 10 → 10 11 → 11

This exception handling scheme is still not sufficient. Consider the sequence "0010(1)" which would be encoded as "010(1)". In this case the first '0' should still mean two "00". This implies that the sequence "01" at the decoder can be interpreted in two ways. To help the decoder differentiate, we need a flag bit after every "01" and "001" combination. So these two combinations would be encoded as - 01 → 011 & 0010(1) → 0100(1). In both of these encoded sequences the third bit is a flag bit. Thus, at the decoder, the code bits and the output would be 011 → 01; 0100(1) → 0010(1).

The above procedure can be summarized as follows - At the encoder if both the symbols in the window are MPS's we drop the second MPS. We code rest of the combinations without dropping the symbols. Whenever a MPS,LPS combination or a MPS,MPS,LPS,MPS(LPS) sequence occurs, we code a flag bit after the MPS,LPS sequence. At the decoder, if MPS occurs it implies a MPS,MPS sequence by default. But we wait for the next symbol to be decoded to decide whether the default assumption is true. If that symbol is MPS then we output the sequence. If, on the other hand the symbol is a LPS, then we decode the next symbol (which would be the flag bit) and decide whether the MPS implies the sequence or just MPS.

3.1. Handling the flag bits

The flag bits can be either embedded into the code or sent out in the header information, based on whether the application is on line or off line. The coder that codes the flag bits is called ACA1 and the one which sends flag bits in header is called ACA2. In the ACA1 coder, it is important that the statistics of the actual data not change due to coding of the flag bits. To achieve this, a separate one bit context has been used for the flag bits. We use the same A & C registers for coding both the actual data and the flag bits. The ACA1 decoder can output 0,1,2 or 3 bits. The ACA2 encoder (decoder) consumes (outputs) 2 bits each cycle.

3.2. Pseudo code for the encoder

```

loop_prev = 0
While (not EOF) {
  read (sym1, sym2); look up Q and MPS1;
  A = A-Q;
  if (sym1 != MPS1) {
    C = C + A; A = Q; Renormalize ();
    if (loop_prev = 1) { /* Flag bit = MPS sense*/
      lookup (Q and MPS_flag);
      A = A-Q;
      if (A < Amin) {
        Renormalize (); update (context_flag); } }
  }
  else {
    if (A < Amin) {
      Renormalize (); }
  }
  update (context); loop_prev = 1; lookup Q and MPS2;
  if ( sym1 != MPS1 or sym2 != MPS2) {
    A = A-Q;
    if (sym1 = MPS1) {
      C = C + A; A = Q; Renormalize ();
      lookup (Q and MPS_flag); /* Flag bit =LPS sense*/
      A = A-Q; C = C + A; A = Q;
      Renormalize (); update (context_flag);
    }
    else {
      if (sym2 != MPS2) {
        C = C + A; A = Q; }
      if (A < Amin) {
        Renormalize (); }
      update (context); loop_prev = 0; } }
}

```

We read in two symbols at each cycle. As explained earlier, we drop the second symbol if both the symbols are MPS's in the window. In the other three cases, we code both the bits. Further, in case of MPS, MPS, LPS, MPS (LPS), we code a flag bit with MPS sense after initial MPS, LPS symbols. We keep track of the sequence with the help of the loop_prev bit. In case of MPS, LPS, we code a flag bit with LPS sense. The renormalization procedure (both encoding and decoding) is adapted from [4].

The decoding procedure, for ACA1 coder is explained in section 2. Further, we need two state bits, mps_flag, lps_flag, and a two bit buffer (buf) to keep track of the flag bit and the output. The following rules are used to set and reset the flags and to generate the output.

Rules –

A) If symbol decoded is LPS then

1) If the mps_flag = 1 and lps_flag = 0 then next bit will be flag bit so we need to decode it.

a) If the flag bit is MPS then lps_flag = 1 and mps_flag = 0, output : buf[0], buf[1], LPS

b) If the flag bit is LPS then mps_flag = 0, output : buf[0], LPS

2) In all other cases, lps_flag = not (lps_flag), output : LPS.

B) If symbol decoded is MPS

1) If the lps_flag = 1 then lps_flag = 0 and output : MPS.

2) If the mps_flag = 0 then mps_flag = 1 and fill the buffer with MPS and MPS of the context formed with present MPS.

3) If the mps_flag = 1 then mps_flag = 1 and output : buf[0] buf[1]. Also fill the buffer with MPS and MPS of the context formed with present MPS.

It should be noted that the state bits are used to indicate what is the first symbol decoded in an uncompleted two bit window so that the mps_flag and lps_flag cannot simultaneously 1.

3.3. Example

Consider the following string {00 01 10 00 10 00}. If we assume 0 to be MPS and 1 to be LPS, then this string would be coded as {0 011 10 0 100 0}, where (1,0) are the flag bits. If we assume that A register is initialized and has a minimum value of 0x100, Q value for regular (flag) bits is 0x80 (0x85) and context is just a single bit, then the steps during the encoding are as follows (lp : loop_prev) -

step	sym	A	Q	C	lp
	-	100	80	.00	0
1	0 (sym1)	80	-	.00	1
		<i>100</i>	<i>80</i>	<i>0.00</i>	
2	0 (sym1)	80	-	0.00	1
		<i>100</i>	<i>80</i>	<i>0.00</i>	
3	1 (sym2)	80	-	0.80	0
		<i>100</i>	<i>85</i>	<i>1.00</i>	
4	1 (flag)	85	-	1.7B	0
		<i>10A</i>	<i>80</i>	<i>2.F6</i>	
5	1 (sym1)	80	-	3.80	1
		<i>100</i>	<i>80</i>	<i>31.00</i>	
6	0 (sym2)	80	-	31.00	0
		<i>100</i>	<i>80</i>	<i>32.00</i>	
7	0 (sym1)	80	-	32.00	1
		<i>100</i>	<i>80</i>	<i>34.00</i>	
8	1 (sym1)	80	-	34.80	1
		<i>100</i>	<i>85</i>	<i>39.00</i>	
9	0 (flag)	7B	-	39.00	1
		<i>F6</i>	<i>-</i>	<i>390.00</i>	
		<i>1EC</i>	<i>80</i>	<i>390.00</i>	
10	0 (sym2)	16C	80	390.00	0
11	0 (sym1)	EC	-	390.00	1
		<i>1d8</i>	<i>80</i>	<i>390.00</i>	

All the *italicized* rows indicate renormalizations. It should be noted that in step 5, the carry generated from the fractional side is propagated into code side. The decoding process is shown below (mf : mps_flag, lf: lps_flag) -

A	Q	C	mf	lf	buf	out	Rule
100	80	39	1	0	00	-	B.2
80		39					
<i>100</i>	<i>80</i>	<i>72</i>	<i>1</i>	<i>0</i>	<i>00</i>	<i>00</i>	<i>B.3</i>
80		72					
<i>100</i>	<i>80</i>	<i>E4</i>	<i>1</i>	<i>0</i>	<i>00</i>	<i>-</i>	<i>A.1</i>
80		64					
<i>100</i>	<i>85</i>	<i>C8</i>	<i>0</i>	<i>0</i>	<i>-</i>	<i>01</i>	<i>A.1.b</i>
85		4D					
<i>10A</i>	<i>80</i>	<i>9A</i>	<i>0</i>	<i>1</i>	<i>-</i>	<i>1</i>	<i>A.2</i>
80		10					
<i>100</i>	<i>80</i>	<i>20</i>	<i>0</i>	<i>0</i>	<i>-</i>	<i>0</i>	<i>B.1</i>
80		20					
<i>100</i>	<i>80</i>	<i>40</i>	<i>1</i>	<i>0</i>	<i>00</i>	<i>-</i>	<i>B.2</i>
80		40					
100	80	80	1	0	00	-	A.1
80		0					
<i>100</i>	<i>85</i>	<i>0</i>	<i>0</i>	<i>1</i>	<i>-</i>	<i>001</i>	<i>A.1.a</i>
<i>7B</i>	<i>-</i>	<i>0</i>					
<i>F6</i>	<i>-</i>	<i>0</i>					
<i>1EC</i>	<i>80</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>-</i>	<i>0</i>	<i>A.2</i>
<i>16C</i>	<i>80</i>	<i>0</i>	<i>1</i>	<i>0</i>	<i>00</i>	<i>-</i>	<i>B.2</i>
						<i>00</i>	

We initialize C register to 39 and as we renormalize, we move the 0 bits into it. It is evident, that we get back the original sequence.

4. PERFORMANCE

For comparison purposes we have used 8 CCITT images [3] of size 2376x1728 pixels and five thresholded binary images (baboon, barbara, fish, gold hill, lena) of size 512x512. The compressed code sizes, due to Q-coder, ACA1 coder and ACA2 coder are given in Table 1. The code size for the ACA2 coder is calculated as round[flag bits/8]+code size. The computation comparison is given in Table2. The number of additions/subtractions for the ACA1 coder include those required for the flag bits as well.

Table 1 Code size comparison

	Code Size			% differences	
	Q-coder	ACA1	ACA2	Q-ACA1	Q-ACA2
c1	26139	27494	27397	5.184	4.813
c2	22021	21916	22066	-0.477	0.204
c3	48442	49915	49784	3.041	2.770
c4	87100	96696	96250	11.017	10.505
c5	52487	55540	55372	5.817	5.497
c6	37508	38490	38536	2.618	2.741
c7	97252	98373	97914	1.153	0.681
c8	44162	43633	43855	-1.198	-0.695
Avg	51888.8	54007.1	53896.8	3.394	3.314
bab	17305	17511	17388	1.190	0.480
bar	11698	12716	12636	8.702	8.018
fish	2262	2227	2219	-1.547	-1.901
gold	8517	8488	8456	-0.340	-0.716
lena	8761	8635	8625	-1.438	-1.552
Avg.	9708.6	9915.4	9864.8	1.313	0.866

Table 2 Computation comparison

	Additions/Subtractions			% differences	
	Q-coder	ACA1	ACA2	Q-ACA1	Q-ACA2
c1	8120804	2217074	2148871	-72.699	-73.539
c2	8159508	2145256	2106290	-73.709	-74.186
c3	8054234	2327052	2215517	-71.108	-72.493
c4	7866744	2698058	2431776	-65.703	-69.088
c5	8032170	2376261	2244084	-70.416	-72.061
c6	8107316	2237510	2161064	-72.401	-73.344
c7	7890074	2591014	2378174	-67.161	-69.859
c8	8111102	2228592	2154549	-72.524	-73.437
Avg	8042744	2352602	2230041	-70.715	-72.251
bab	451156	236849	200303	-47.502	-55.602
bar	477420	205724	201907	-56.909	-57.709
fish	517874	140357	137076	-72.897	-73.531
gold	495184	173480	158781	-64.967	-67.935
lena	499774	169274	154795	-66.130	-69.027
Avg.	488281.6	185136.8	852862	-61.681	-64.761

From the tables, for CCITT images, we observe that on an average the loss of compression is about 3.4% but the reduction in the number of additions/subtractions is 70% for the ACA coders compared to the Q-coder. For the binary images, on an average the loss of compression is about 1.3% and the reduction in the number of additions/subtractions is 61%.

5. VLSI ARCHITECTURE

The architecture consists of three blocks (Fig. 2) namely :
 1) Look up tables - To hold the MPS and Q index values for each of the context and one bit context of the flag bit.
 2) Coder - To generate the code (symbol) based on the input symbol (code) and the Look up tables for encoding(decoding).
 3) Context - To hold 10 previous encoded (decoded) symbols.
 In addition, there is a global controller for the three blocks

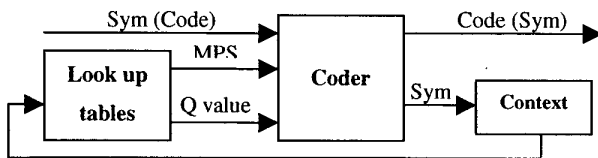


Fig. 2 Block diagram of the encoder (decoder)

The look up tables block consists of two memories of size 1024x2x6 (to hold MPS and Q-index for each of the 2¹⁰ contexts and 2 for the one bit context for the flag) and 30x12 (to hold the Q-values) and the logic for Q-index update from [4].

Coder (Fig. 3) consists of 5 registers A, B, C, Q and bufout, and an ALU, which consists of a 13 bit adder, a 12 bit incrementer, a 5 bit decremter and a 5 bit All 0's detector. There are two data busses, each 13 bits wide. Context is a 10 bit shift register.

A and C registers hold the range and initial point of the present range respectively. B register holds a byte of code bits to help in bit stuffing [4], Q register holds the Q value for the present context and the bufout register is used for memory interface purposes. A and C registers can carry out a one bit left shift. A register also has logic to find amount of shift required for

renormalization. Renormalization is triggered whenever bit A(13) = 0. The decremter and All 0's detector in ALU are used to keep track of the amount of shift left to complete the renormalization. To perform C = C + A operation, we need to add 24 bits of C to 13 bits of A. We obtain this sum by first using the 13 bit adder and then based on the carry out (cout) from the adder, C is incremented with the help of incrementer. The signal C(31) is used to indicate that a byte of code bits are ready and incr(12) is used to hold the carry of the previously decoded byte. An All 1's detector is present in the B register, to help in bit stuffing in case of the previous byte being 0xFF. Ack and buf_full signals are used for two signal handshaking scheme.

The controller is the most complex part of this architecture. We have developed a state machine with 58 states to carry out the ACA1 encoding process and a state machine with 30 states for the Q-coder. While the data path for both the coders is similar with the Q-coder having an extra ALU, the ACA1 controller is larger. We are currently synthesizing the architecture using Synopsys for better evaluation of their hardware complexities.

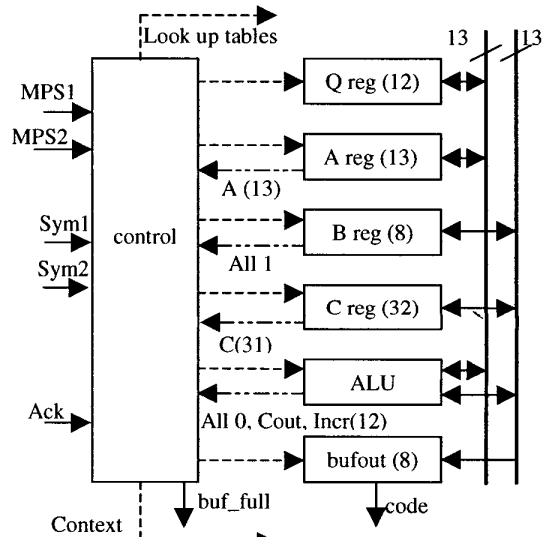


Fig. 3 Block diagram of the coder

7. REFERENCES

[1] I. H. Witten, R. Neal and J. G. Cleary, "Arithmetic coding for data compression," *Communication of the ACM*, Vol. 30, pp. 520-540, June 1987.
 [2] G. L. Langdon, Jr., and J. Rissanen, "Compression of black-white images with arithmetic coding," *IEEE Transactions on Communications*, Vol. Com-29, pp. 858- 867, June 1981.
 [3] R. B. Arps, T. K. Troung, D.J. Lu, R.C. Pasco and T.D. Friedman, "A multi-purpose VLSI chip for adaptive data compression of bi-level images," *IBM J. of Res. Develop*, Vol. 32, No. 6, pp. 775- 795, Nov. 1988.
 [4] J. L. Mitchell and W. B. Pennebaker, "Software implementations of the Q-coder," *IBM J. of Res. Develop*, Vol. 32, No. 6, pp. 753- 774, Nov. 1988.
 [5] J. L. Mitchell and W. B. Pennebaker, "JPEG still image data compression standard," Van Nostrand Reinhold, 1993.