Efficient encoding and decoding algorithms for variable-length entropy codes

G.R. Higgie and A.C.M. Fong

Abstract: Successive versions of the MPEG and JPEG coding standards have been widely adopted for compression of moving pictures and still images. Variable-length codes (VLC) such as Huffman codes are generally employed within the frameworks of these standards to achieve optimal coding efficiency in the entropy coding stage. Conventional table look-up encoding/decoding techniques for VLC are inefficient in memory usage. This is a particularly important consideration for hand-held devices, such as cellular phones with image reception and processing capabilities. The authors present a series of efficient and simple algorithms for VLC encoding and decoding based on a pointer look-up approach. They present both software and hardware implementations of these algorithms, and a comparison of memory requirements associated with the pointer look-up and conventional techniques.

1 Introduction

There is an increasing demand for efficient and reliable transport of multimedia data over band-limited and noisy communication channels such as the Internet and wireless networks. Video and image compression standards such as MPEG [1] and JPEG [2] have been developed for effective data compression. Each of these standards prescribes a framework in which there is some flexibility in the actual implementation while remaining standard-compliant [3]. In this research, we are interested in the implementation of the entropy coding stage within the compression standards' frameworks.

Traditionally, variable-to-fixed-length entropy coding or fixed-length codes (FLCs) [4] are employed to reduce the uncertainty associated with codeword synchronisation in situations where data integrity has a high likelihood of being compromised due to channel-induced errors. Indeed, there has recently been renewed interest in FLC research, some of which is specifically geared toward image/video applications, e.g. [5]. However, variable-length codes (VLCs) potentially offer coding efficiency unmatched by FLC. This is a particularly important consideration for image data, as the probabilities of different symbols occurring can vary much more greatly than, for example, in text messages. In fact, VLC is most often used in the lossless entropy coding stage for practical video and image data compression.

VLCs that are of practical use are referred to as prefix codes. Prefix codes are codes that are uniquely and instantaneously decodable. Thus, with prefix codes, the decoder can begin decoding as soon as the bit-stream arrives. From an implementation point of view, this implies that a decoding tree can be constructed. In addition, practically useful VLC sets must be exhaustive. A non-exhaustive code set is one where a decision node on the binary tree has only one output instead of the usual two. (We assume that we have a binary information source with alphabet symbols 0 and 1 only.) This results in an extra node without a corresponding extra codeword. Non-exhaustive code sets are of little practical significance, as they require extra bits to represent the same information as the corresponding exhaustive code set to which they can be converted. The conversion can be carried out by removing the superfluous node along with the corresponding bit of each codeword that passes through the node, as illustrated in Fig. 1.

Recently, advances in VLC development have resulted in two interesting variants: reversible (or bidirectional) VLC (RVLC), e.g. [6–8], and self-synchronising VLC (SSVLC), e.g. [9–12]. The chief advantage of RVLC is that decoding can be performed in the forward and backward directions. This sometimes provides a means of reconstruction of the signal from the last data received. However, major

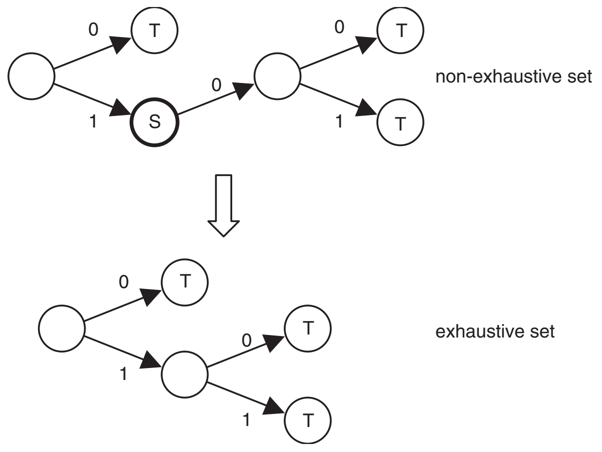

Fig. 1 *Conversion from non-exhaustive code set to exhaustive code set*
'T' denotes a terminal node and 'S' denotes the superfluous node

© IEE, 2003
IEE Proceedings online no. 20030540
doi:10.1049/ip-com:20030540
Paper first received 5th August 2002 and in revised form 7th January 2003

G.R. Higgie is with the Cecam Consultancy, 2 Kari Street, Grafton, Auckland, New Zealand
A.C.M. Fong was with the Institute of Information and Mathematical Sciences, Massey University, Albany, Auckland, New Zealand and is now with the School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798

disadvantages with RVLC include decreased compression efficiency, increased error propagation and increased decoding complexity [13]. Further, in situations where multiple bits that are further apart are lost the entire segment in between them cannot be recovered. Then SSVLC would be more suitable [7].

In general, the purpose of applying SSVLC is to achieve the dual goal of optimal coding efficiency and providing enhanced data integrity. The former means selecting optimally efficient VLCs close to the information source entropy, whereas the latter implies a strict localisation of error propagation, typically to within two symbols of an error occurring.

Among the various SSVLCs reported in the literature, the families of T-codes [11, 12] appear to be some of the most promising in terms of their coding efficiency and self-synchronising properties. All T-codes exhibit exemplary automatic synchronisation properties by virtue of the T-augmentation algorithm, which spreads synchronisation information throughout the codes as a matter of course. It has also been reported [14] that the T-augmentation algorithm can be used to obtain codes that were previously obtained by some transformation techniques, e.g. [10], applied to improve codes derived from the standard Huffman algorithm.

A number of techniques have been reported on VLC encoding and decoding. Some of these are specific hardware approaches, e.g. [15]. However, software implementations offer many advantages, such as better scalability and flexibility, and lower costs. In this paper, we present fast, efficient and simple algorithms for encoding and decoding VLCs. Further, our algorithms can be implemented efficiently using both software and hardware solutions.

Without loss of generality, we illustrate our examples with T-codes mainly because of the ease of code identification and specification. The codes could equally well have been derived from, for example, the Huffman algorithm with or without transformation. In fact, the algorithms described in this paper apply to all instantaneously decodable and exhaustive codes. Further, we consider only binary information sources.

2 Conventional look-up table approach

The look-up table approach is conceptually very simple and is the conventional way of encoding VLCs [16]. In this method, the entire code set is stored in an array. Each element of the array contains a bit sequence and a length indicator for one codeword. The index of each element in the array is the source symbol.

2.1 Encoding

As illustrated in Fig. 2, the conventional look-up table approach gives very fast encoding as the bits for each codeword are simply read out from memory. However, it has the disadvantage that for a simple array structure each element in the array structure must be capable of storing the maximum possible number of bits of any codeword in the code set.

Thus, the major drawback of the conventional table look-up approach is that it requires a large storage capacity, with most of the storage remaining unused. Table 1 shows the look-up table for the T-code set $S_{(0,01,1,11)}^{(1,1,1,1)}$ and arbitrary source symbols $s_1, \ldots, s_{17}$.

```
for bit_position := 1 to length[symbol]
  output code[symbol].bit[bit_position]
```

Fig. 2 *Basic look-up table encoding algorithm*

2.2 Decoding

To perform decoding, the entire code set is again stored in an array, along with the length indicators. However, this time the codes have first to be sorted into order such that all codewords that start with a zero appear first, followed by all codewords that start with a '1'. Within each group, the codewords are sorted on their second bits, third bits, etc. in the same manner. The source symbols are also stored because the index of the array is no longer valid after the sort operation. Table 2 gives the sorted look-up table corresponding to Table 1, and the basic decoding algorithm is shown in Fig. 3.

Table 1: Conventional look-up table

Source symbol	VLC	Codeword length
s_1	00	2
s_2	011	3
s_3	0100	4
s_4	0101	4
s_5	100	3
s_6	1011	4
s_7	10100	5
s_8	10101	5
s_9	1100	4
s_{10}	11011	5
s_{11}	110100	6
s_{12}	110101	6
s_{13}	1111	4
s_{14}	11100	5
s_{15}	111011	6
s_{16}	1110100	7
s_{17}	1110101	7

Table 2: Sorted look-up table

Source symbol	Sorted VLC	Codeword length
s_1	00	2
s_3	0100	4
s_4	0101	4
s_2	011	3
s_5	100	3
s_7	10100	5
s_8	10101	5
s_6	1011	4
s_9	1100	4
s_{11}	110100	6
s_{12}	110101	6
s_{10}	11011	5
s_{14}	11100	5
s_{16}	1110100	7
s_{17}	1110101	7
s_{15}	111011	6
s_{13}	1111	4

```
  position := 0; bits_used := 0;
  repeat
    inc(bits_used);
    if bit_stream[bits_used] = 1 then
      while code[position].bit[bits_used] <> 1
        inc(position);
  until bits_used = length[position];
symbol := source_symbol[position];
```

Fig. 3 *Basic look-up table decoding algorithm*

A number of attempts have been made to improve the memory efficiency of this conventional look-up approach, e.g. [17–19]. Sieminski [17] suggests breaking the variable-length format of the codewords into short, fixed-length bytes, but this is only applicable to certain system implementations involving microprocessors. In [18], a technique for generating self-synchronising code sets is presented, along with encoding/decoding techniques peculiar to the structure of those codes. An encoding/decoding technique based on the structure of Huffman codes has been proposed by Tanaka [19]. However, although the technique alleviates the storage requirements required by the conventional approach, it is achieved at the expense of increased complexity and reduced speed of operation. In this research, we propose pointer look-up table algorithms that boast high speed of operation, simplicity of structure and optimal memory efficiency.

3 Pointer look-up techniques

Practical applications generally require either a symmetric distribution of computational load between the encoder and decoder, or an asymmetric load distribution whereby encoding can be computationally intensive, as it can be performed off-line, while decoding should be efficient and fast. This is a particularly important consideration for hand-held receiving devices with limited memory and processing power.

Although the conventional look-up table technique can achieve reasonably fast encoding, decoding is not particularly fast. This is because every time it receives a ‘1’ it must scan down through the table to find the next ‘1’ in the current bit position. If the table position of the next ‘1’ is stored explicitly in the table, then the decoder can go directly to this position when required. This forms the basis for the pointer look-up approach.

3.1 Software decoder

For any instantaneously decodable code, the decoding process can be represented by a decoding tree by extending the exhaustive set in Fig. 1 to include all codewords in any given code set. Each terminal node ‘T’ (or leaf) of the decoding tree corresponds to one codeword. Non-terminal nodes are decision nodes. Decoding begins at the far left node with each received bit causing a transition to the next node along the ‘0’ or ‘1’ path until a terminal node is reached.

By arranging the decoding tree as shown in Fig. 4, the ‘0’ transitions are drawn horizontally and the ‘1’ transitions are drawn vertically. This results in having exactly one terminal node in each row. The rows can be pointed to by a vertical pointer (*VP*). The order of the terminal nodes follows the order of the codewords in the sorted list. Thus, *VP* points to the position in the sorted codeword list. Every decision node contains a next 1 pointer (*NP*) that indicates the position in the code list of the next ‘1’. This corresponds to the position to which *VP* must be set whenever a ‘1’ is received. The

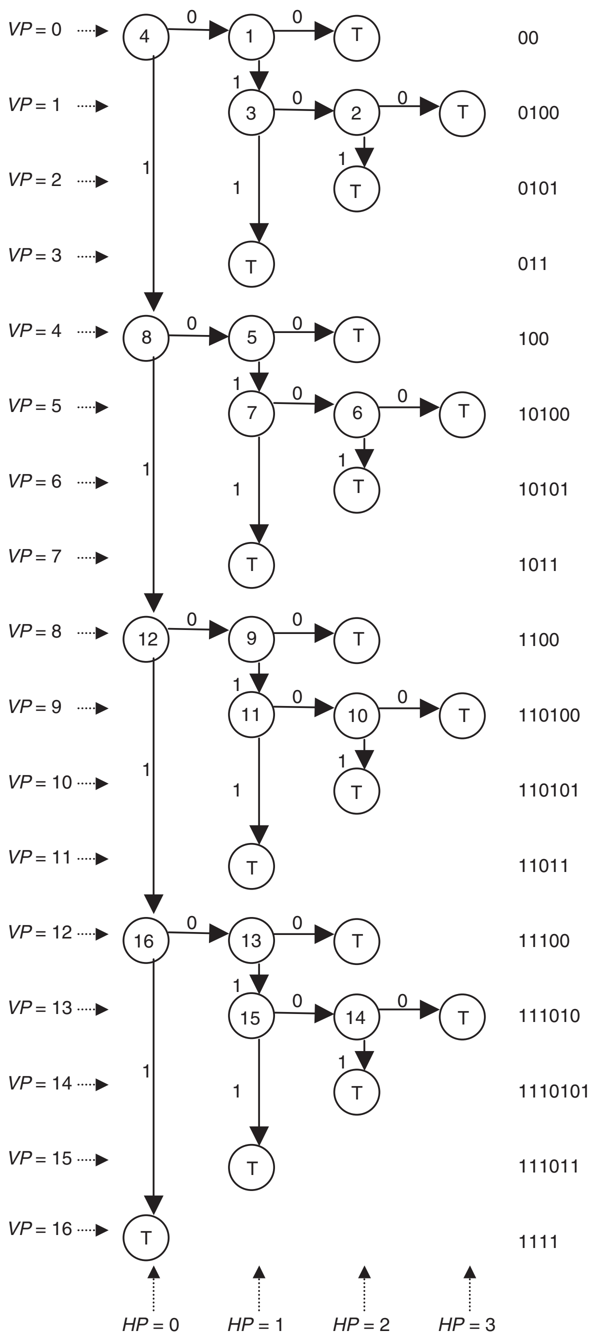

Fig. 4 *Decoding tree arranged for pointer lookup*

columns of the tree can be pointed to by a horizontal pointer (*HP*). *HP* is effectively the number of zeros that have been received and should be incremented every time a ‘0’ is received.

As shown in Table 3, the table of pointers can be derived directly from Fig. 4 by taking the *NP* values from the decision nodes and placing them in the corresponding positions of a two-dimensional (2-D) array. The indices of the array are the *VP* and *HP* positions of the decision nodes. It is important to note that once the table of pointers has been derived, it is no longer necessary to store the actual codewords in the array. The decoding algorithm is shown in Fig. 5.

The algorithm gives very fast decoding as each bit processed only requires three operations:

- decide whether the bit is ‘0’ or ‘1’
- increment or load a pointer
- check for the end of the codeword

Table 3: 2-D pointer table

Source symbol	Vertical position	Sorted VLC	*NP*			Codeword length
			*HP*0	*HP*1	*HP*2	
s_1	0	00	4	1		2
s_3	1	0100		3	2	4
s_4	2	0101				4
s_2	3	011				3
s_5	4	100	8	5		3
s_7	5	10100		7	6	5
s_8	6	10101				5
s_6	7	1011				4
s_9	8	1100	12	9		4
s_{11}	9	110100		11	10	6
s_{12}	10	110101				6
s_{10}	11	11011				5
s_{14}	12	11100	16	13		5
s_{16}	13	1110100		15	14	7
s_{17}	14	1110101				7
s_{15}	15	111011				6
s_{13}	16	1111				4

```
VP := 0; HP := 0; bits_used := 0;
repeat
  inc(bits_used);
  if bit_stream[bits_used] = 0 then inc(HP)
   else VP := NP[VP, HP];
 until length[VP] = bits_used;
 symbol := source_symbol[VP];
```

Fig. 5 *Two-dimensional pointer look-up decoding algorithm*

3.2 Reducing the 2-D array

One difficulty with the algorithm in Fig. 5 is that the array containing *NP* is two dimensional, with a row for every vertical position and a column for every horizontal position. In code sets that have codewords with long strings of zeros, there will be a large number of horizontal positions. However, there are no entries for rows that have only a terminal node.

Lemma 1: If there are *n NP* and *q* codewords, then

$$n = q - 1 \qquad (1)$$

Proof: This is by induction, which applies to all instantaneously decodable and exhaustive codes. For the trivial code set that has only one codeword, the decoding tree will consist of only one terminal node and no decision nodes. This code set may be extended as follows. First, the terminal node can be converted into a decision node. Next, two new terminal nodes are added. The result is one extra codeword, one extra terminal node and one extra decision node. This happens every time the code set is extended, with every extra codeword requiring an extra terminal node and an extra decision node, starting from the trivial code set. Thus, (1) is true.

The consequence of lemma 1 is that the 2-D pointer look-up table can be reduced to 1-D. The corresponding 1-D pointer look-up decoding algorithm is shown in Fig. 6, and Table 4 shows the 1-D look-up table for the same code set as Table 3.

3.3 Tables without codeword lengths

A variation on the 1-D pointer look-up technique is not to use the array of codeword lengths to indicate when a codeword has been decoded. An additional pointer is needed that is initially positioned just beyond the end of the table. As before, whenever a '1' is processed *VP* is loaded with the pointer value. The variation occurs when a '0' is processed, in which case the end pointer (*EP*) is loaded with the pointer value in addition to *HP* being incremented. As each '0' or '1' is processed, *VP* and *EP* close in on each other to trap the decoded codeword between them. When these pointers point to adjacent locations, *VP* is positioned at the decoded codeword. The decoding algorithm is now modified to the one shown in Fig. 7.

```
VP := 0; HP := 0; bits_used := 0;
repeat
  inc(bits_used);
 if bit_stream[bits_used] = 0 then inc(HP)
   else VP:= NP[VP + HP];
 until length[VP] = bits_used;
 symbol := source_symbol[VP];
```

Fig. 6 *One-dimensional pointer look-up decoding algorithm*

Table 4: 1-D pointer table

Source symbol	Vertical position	Sorted VLC	*NP*	Codeword length
s_1	0	00	4	2
s_3	1	0100	1	4
s_4	2	0101	3	4
s_2	3	011	2	3
s_5	4	100	8	3
s_7	5	10100	5	5
s_8	6	10101	7	5
s_6	7	1011	6	4
s_9	8	1100	12	4
s_{11}	9	110100	9	6
s_{12}	10	110101	11	6
s_{10}	11	11011	10	5
s_{14}	12	11100	16	5
s_{16}	13	1110100	13	7
s_{17}	14	1110101	15	7
s_{15}	15	111011	14	6
s_{13}	16	1111		4

```
VP := 0; HP := 0; bits_used := 0;
EP := end_of_list + 1;
repeat
  inc(bits_used);
  if bit_stream[bits_used] = 0 then
     {EP := NP[VP + HP];  inc(HP)}
   else VP := NP[VP + HP];
 until(EP - VP) = 1;
            symbol := source_symbol[VP];
```

Fig. 7 *Variation on one-dimensional pointer look-up decoding algorithm (without codeword lengths)*

3.4 Software encoder

Having developed a technique for decoding using only a table of fixed-length pointers, it is advantageous to use the same table for encoding. The table is essentially a series of pointers that specify the location of the next '1' in the stored list. Hence, if the codeword for a particular position in

```
position = Translate[symbol];
VP := 0; HP := 0; bits_transmitted := 0;
repeat
  if position < NP[VP + HP] then
   {output a "0"; inc (HP)}
  else
   {output a "1"; VP := NP[VP + HP];}
  inc(bits_transmitted);
until code_length[VP] = bits_transmitted;
```

Fig. 8 *One-dimensional pointer look-up encoding algorithm*

the stored list is to be encoded, then the bit to be sent, b_{i+1}, will be:

$$b_{i+1} = \begin{cases} \text{`0'}, & \text{if (position} < NP[VP + HP]) \\ \text{`1'}, & \text{otherwise} \end{cases} \tag{2}$$

Thus, b_{i+1} is set to zero if the current position is before the next '1', and set to one otherwise. Figure 8 shows the encoding algorithm that corresponds to Fig. 6.

The encoding algorithm that corresponds to Fig. 7 can be deduced simply be modifying the algorithm in Fig. 8 by introducing *EP* in much the same way as the conversion from Fig. 6 to Fig. 7. At any rate, pointer-based encoding is as efficient as pointer-based decoding as it consists of essentially the same three operations:

- decide whether position $< NP$
- increment and/or load a pointer; output '0' or '1'
- check for the end of the codeword

3.5 Pointer table generation

The generation of pointers for the pointer look-up table is very simple. The variable-length codewords of the code set must first be generated and sorted. From Fig. 4, it is clear that there will be a pointer entry for each trailing zero of every codeword. A simple algorithm for pointer table generation is shown in Fig. 9.

For example, codeword 0100 has a pointer for each of its two trailing zeros and codeword 0101 has no pointers as it has no trailing zeros. The value of the pointer for the trailing zero at a particular bit position can be derived by searching down the codeword list until the next '1' is found at that bit position. When a sorted list of variable-length codewords is stored in an array code[ ] and their associated codeword length is stored in an array length[ ], the generating algorithm is as shown in Fig. 9. Once the pointers have been generated, the original code[ ] and length[ ] arrays can be discarded.

4 Hardware implementation

Hardware implementation of the pointer look-up encoding/decoding technique follows the same algorithms developed for software implementation. In this section, we present the hardware implementation of the algorithm, without codeword lengths, developed in Section 3.3. The decoder interprets incoming bits to trap the decoded position between *VP* and *EP*. Figure 10 shows a block diagram of hardware suitable for implementing the algorithm for up to 257 codewords.

The operation of the hardware shown in Fig. 10 is summarized in Fig. 11. This hardware implementation is considerably simpler than that required for conventional encoding and decoding techniques. Its speed of operation is also very high because only one operating cycle is required for each bit of encoded data, with no multiple processing of incoming encoded bit stream required.

5 Comparison of techniques

The pointer look-up table technique is generally very fast, especially when compared to the conventional look-up table technique. In essence, the convention approach wastes time looking for the 'next 1', while the pointer look-up approach eliminates this need. However, the main disadvantage

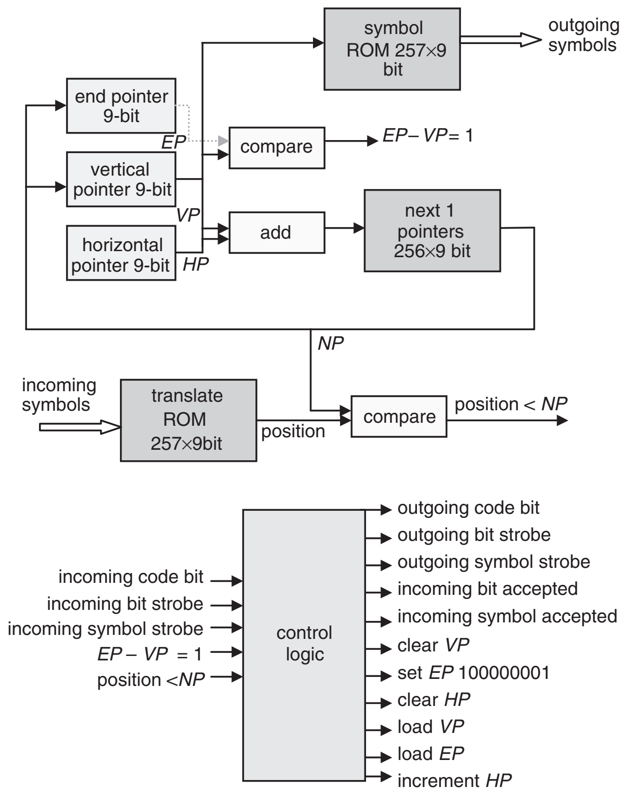

Fig. 10 *Hardware implementation for pointer look-up table encoding and decoding*

```
pointer_index = 0;
  for code_index = 0 to numer_codewords - 2
   {bit_position := length[code_index] + 1;
    while (code[code_index].bit[bit_position-1] = 0)
      and (bit_position>1)
       dec(bit_position);
     for bit_position = bit_position to
      length[code_index]
      {next_i_index := code_index + 1;
      while code[next_1_index].bit[bit_position]<> 1
          inc(next_1_index);
       NP[pointer_index] := next_1_index; inc(pointer_index);}}
```

Fig. 9 *Algorithm for pointer generation*

Encoding:
```
 Clear VP
   Clear HP
   Set EP 100000001                    {EP to 1 past end of list}
 wait for Incoming symbol strobe
 Incoming symbol accepted            {input accepted for encoding}
 repeat
      if position < NP then                {bit to be output = 0}
    Outgoing code bit = 0
         Load EP                      {move EP to next 1 position}
    Increment HP
      else
         Outgoing code bit = 1
    Load VP                           {move VP to next 1 position}
   Outgoing bit strobe                     {output bit available}
 until EP - VP = 1                           {processed last bit}
```
Decoding:
```
 Clear VP
   Clear HP
   Set EP 100000001                    {EP to 1 past end of list}
 repeat
      wait for Incoming bit strobe
      Incoming bit accepted                        {bit accepted}
   if Incoming code bit = 0 then          {bit to be decoded = 0}
         Load EP                      {move EP to next 1 position}
    Increment HP
      else
         Load VP                      {move VP to next 1 position}
  until EP - VP = 1                          {processed last bit}
 Outgoing symbol strobe                          {output available}
```

Fig. 11 *Control logic algorithms for hardware encoding and decoding*

of all look-up tables is their memory requirements. The pointer look-up table technique requires the following arrays, with lengths up to the number of codewords in the code set q:

- Source symbols[q]
- Translate[q]
- Next 1 Pointer[$q-1$]

Source Symbols[q] is used by decode to extract the original symbol after ascertaining the decoded codeword position. Translate[q] is used by encode and is indexed on Source Symbol to point to the position in the list of the codeword for the source symbol. Next 1 Pointer[q–1] stores the pointers to the position of 'next 1' in the list. These arrays all contain fixed-length data entries. This arrangement is far more memory efficient than the conventional approach. The conventional look-up table technique uses the same fixed-length Source Symbols and Translate arrays, but also stores the length of each codeword in an array, along with the entire variable-length code set in an array where each element must be able to contain the longest possible codeword. The maximum codeword length of an exhaustive code set with q codewords is $q-1$. Therefore, the byte storage requirement M for each entry is given by

$$M = (q-1)/8 \qquad (3)$$

For example, for 8-bit codewords $M = 2^8/8 = 32$ bytes. It is possible to reduce this storage requirement by restricting the system to code sets with a lower maximum codeword length. However, the implication is that the system will not be able to handle all possible code sets, some of which may be important in practical situations.

To illustrate the advantage of the pointer look-up table technique over the conventional technique, the storage requirements for code sets with 256 codewords (suitable for encoding 8-bit source symbols) and code sets with 65536 codewords (suitable for encoding 16-bit source symbols) are given in Table 5.

For example, encoding using the 8-bit pointer approach requires 2^8 (for Translate) + 2^7 (for Next-1-Pointer) = 384 bytes, while decoding requires 2^8 (for Source Symbols) + 2^7 (for Next-1-Pointer) = 384 bytes. On the other hand, encoding and decoding using the 8-bit conventional approach each requires $2^7 \times 32 + 384 = 4480$ bytes. In our analysis, we included a restricted set for the 16-bit conventional approach. It is clear from Table 5 that the pointer look-up table technique offers significant saving in its memory requirements over the conventional technique.

6 Conclusion

Popular video and image compression standards, such as MPEG and JPEG, have been widely adopted. The entropy coding stage within each of the compression standards generally involves the use of variable-length codes. Conventional look-up table techniques employed for encoding VLCs is fast, but less so for decoding. Also, conventional techniques are inefficient in terms of memory requirements. This is a particularly important consideration for hand-held devices, such as cellular phones with image data reception and processing capabilities.

A number of attempts at reducing the conventional look-up table sizes have been proposed. However, these tend to be device and/or code specific, and often lead to reduced speed of operation and increased complexity. In this paper, we have proposed the use of pointer look-up tables for encoding and decoding VLCs. This technique achieves optimal usage of memory storage and can lead to very simple algorithms. Without loss of generality, we have illustrated our technique with reference to T-codes, but there is nothing in our discussion that precludes the application of our technique to other VLCs. In fact, the pointer look-up table technique can be applied to all instantaneously decodable and exhaustive code sets.

Based on the pointer look-up technique, we have implemented software and hardware solutions that are

Table 5: Comparison of storage requirements

	Fixed length entries size (bytes)	Variable length entries size (bytes)	Storage requirements for encode (bytes)	Storage requirements for decode (bytes)
8-bit conventional	1	32	4480	4480
8-bit pointer	1		384	384
16-bit conventional	2	8192	268632064	268632064
(restricted max length = 128 bits)	2	16	720896	720896
16-bit pointer	2		196608	196608

extremely simple, versatile, memory-efficient and fast in operation.

7 References

1 The MPEG standard. http://www.mpeg.org/MPEG/index.html and http://mpeg.telecomitalialab.com/
2 The JPEG standard. http://www.jpeg.org/
3 Ortega, A., and Ramchandran, K.: 'Rate-distortion methods for image and video compression', *IEEE Signal Process. Mag.*, 1998, pp. 23–50
4 Tunstall, A.: 'Synthesis of noiseless compression codes'. PhD dissertation, Georgia Inst. Tech., Atlanta, 1968
5 Llados-Beranus, R., and Stevenson, R.L.: 'Fixed-length coding for robust video compression', *IEEE Trans. Circuits Syst. Video Technol.*, 1998, **8**, pp. 745–755
6 Takishima, Y., Wada, M., and Murakami, H.: 'Reversible variable length codes', *IEEE Trans. Commun*, 1995, **43**, (2), pp. 158–162
7 Girod, B.: 'Bidirectionally decodable streams of prefix code-words', *IEEE Commun. Lett.*, 1999, **3**, (8), pp. 245–247
8 Tsai, C.-W., and Wu, J.-L.: 'On constructing the Huffman-code-based reversible variable-length codes', *IEEE Trans. Commun.*, 2001, **49**, (9), pp. 1506–1509
9 Neumann, P.G.: 'Efficient error-limiting variable length codes', *IRE Trans. Inf. Theory*, 1962, **8**, pp. 292–304
10 Ferguson, T.J., and Rabinowitz, J.H.: 'Self-synchronising Huffman codes', *IEEE Trans. Inf. Theory*, 1984, **30**, pp. 687–693
11 Titchener, M.R.: 'Digital encoding by means of new T-codes to provide improved data synchronisation and message integrity', *IEE Proc., Comput. Digit. Tech.*, 1984, **131**, pp. 151–153
12 Titchener, M.R.: 'The synchronisation of variable-length codes', *IEEE Trans. Inf. Theory*, 1997, **43**, (2), pp. 683–691
13 Webb, J.L.H.: 'Efficient table access for reversible variable-length decoding', *IEEE Trans. Circuits Syst. Video Technol.*, 2001, **11**, pp. 981–985
14 Gunther, U., and Titchener, M.R.: 'Calculating the expected synchronisation delay for T-code sets', *IEE Proc., Commun.*, 1997, **144**, pp. 121–128
15 Yang, J.-Y., Lee, Y., Lee, H., and Kim, J.: 'A variable length coding ASIC chip for HDTV video encoders', *IEEE Trans. Consum. Electron.*, 1997, **43**, (3), pp. 633–638
16 Jayant, N.S., and Noll, P.: 'Digital coding of waveforms- principles and applications to speech and video' (Prentice-Hall, 1984)
17 Sieminski, A.: 'Fast decoding of Huffman codes', *Inf. Process. Lett.*, 1988, **26**, pp. 237–241
18 Mirkovic, M.D.: 'Algorithm for obtaining self-synchronising M-ary code enabling data compression', *IEE Proc., Comput Digit. Tech.*, 1987, **134**, pp. 112–117
19 Tanaka, H.: 'Data structure of Huffman codes and its application to efficient encoding and decoding', *IEEE Trans. Inf. Theory*, 1987, **33**, pp. 154–156