A High-Throughput Memory-Based VLC Decoder with Codeword Boundary Prediction

Bai-Jue Shieh, Yew-San Lee, and Chen-Yi Lee

***Abstract*—In this paper, we present a high-throughput memory-based VLC decoder with codeword boundary prediction. The required information for prediction is added to the proposed branch models. Based on an efficient scheme, these branch models and the Huffman tree structure are mapped onto memory modules. Taking the prediction information, the decompression scheme can determine the codeword length before the decoding procedure is completed. Therefore, a parallel-processor architecture can be applied to the VLC decoder to enhance the system performance. With a clock rate of 100 MHz, a dual-processor decoding process can achieve decompression rate up to 72.5 Msymbols/s on the average. Consequently, the proposed VLC decompression scheme meets the requirements of current and advanced multimedia applications.**

***Index Terms*—Codeword boundary prediction, Huffman coding, memory-based, VLD.**

I. Introduction

With the progress of multimedia technologies, a large amount of data is used for representing video films and photographic images. To transmit and keep the information, high bandwidth communication systems and large-capacity storage devices are developed. Nevertheless, they cannot satisfy the requirements of many advanced applications. An efficient data-compression scheme is necessary for reducing the transmission costs and saving the storage space. A classical data-compression scheme is the Huffman code [1], also called the variable length code (VLC). It is the most popular lossless compression technique, which is recommended as the entropy coding method by many international standards, such as JPEG, MPEG, and H.263. Based on the predetermined weight of each symbol, the Huffman procedure assigns shorter codewords to the higher probability symbols and longer codewords to the less frequency symbols. Therefore, it exploits data redundancy, and the achieved compression ratio is very close to the source entropy.

Although the Huffman encoding procedure reduces a great amount of data, two cases make the realization of high-performance decompression schemes difficult. The first: codeword lengths are variable. The codeword boundary in a bit stream cannot be detected until the decoding procedures of previous codewords are completed. This recursive dependence results in an upper bound on iteration speed. The second: pipeline schemes are not very efficient to increase the throughput of VLC decoders. For most applications, pipeline techniques can improve the performance of systems by optimizing the clock rate. However, the VLC decompression scheme has to go through one level of the Huffman trees in each operation. The time that this operation takes limits the possible decoding throughput even though it is divided into several pipeline stages.

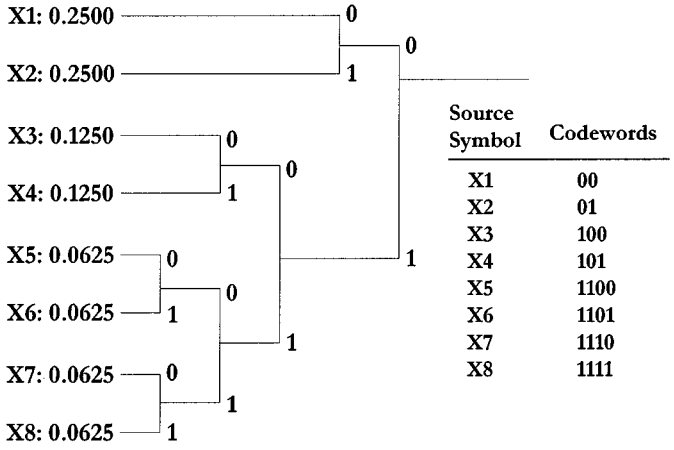

Fig. 1. An example of Huffman coding procedure.

Several VLC decoders and Huffman decompression schemes have been discussed. The PLA-based and ROM-based designs are presented in [2]–[6], [12], and [13]. Because their architectures are the direct mapping of coding tables, the VLSI implementations have to be redesigned when the tables are changed. Besides, the designs in [3] and [4] use the concurrent and parallel architectures to break the bottleneck of the decoding throughput. Nevertheless, they are designed for multiple independent bit streams. The iteration bound of a single bit stream remains unsolved. The memory-based VLC decoders are presented in [7]–[11]. Based on the memory-mapping schemes, the coding table information is loaded into on-chip memories to obtain flexibility. Therefore, the Huffman tables can be changed without redesign and the architectures can be used by various applications. In addition, the pipeline schemes in [8] and [9] optimize the operation clock rate. However, the total time that spends in going through one tree level is not reduced. The system performance is not improved significantly.

The motivation behind our research is developing a high throughput and flexible VLC decoder that can satisfy the requirements of current and advanced multimedia applications. According to the proposed branch models and the efficient memory-mapping scheme, the decoding procedure with codeword boundary prediction is presented. Because the recursive dependence of a single bit stream is broken by this procedure, a parallel-processor VLC decoder is proposed to

Manuscript received June 1, 1998; revised June 12, 2000. This work was supported by the National Science Council of Taiwan, R.O.C., under Grant NSC87-2215-E-009-035. This paper was recommended by Associate Editor N. Ranganathan.

The authors are with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, 300, Taiwan, R.O.C. (e-mail: titany@royals.ee.nctu.edu.tw).

Publisher Item Identifier S 1051-8215(00)10623-8.

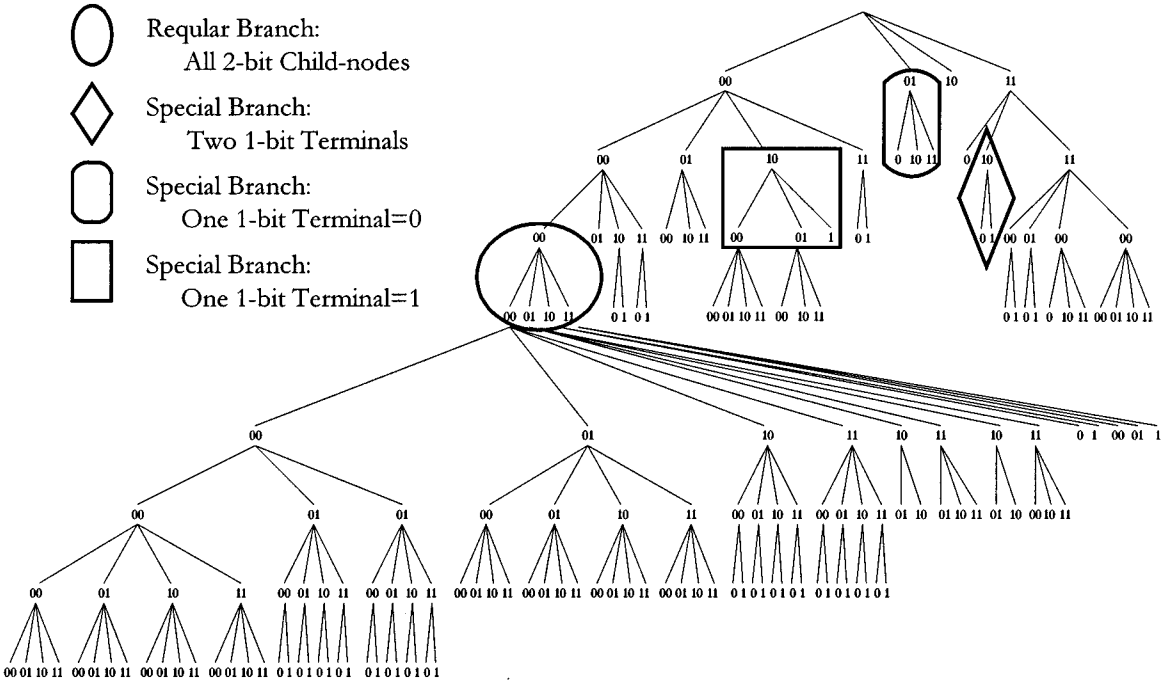

Fig. 2. Branch types and the 2-bit tree structure of MPEG-2 VLC table 15.

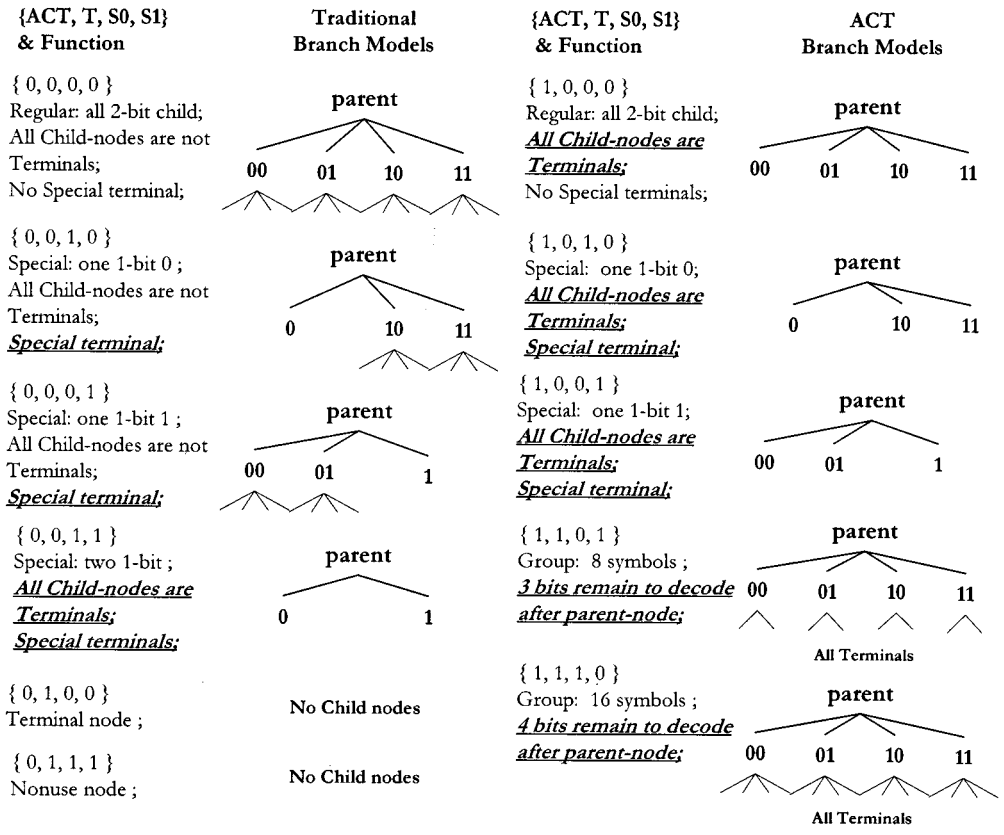

Fig. 3. Branch models and bit assignments.

increase the decoding throughput. Based on a dual-processor decoding process, simulation results show that the average decompression rate up to 72.5 Msymbols/s can be achieved at 100-MHz clock rate.

The organization of this paper is as follows. In Section II, the branch models and the memory-mapping scheme are proposed. Then the decoding procedure with codeword boundary prediction is described. A parallel-processor VLC decoder is

TABLE I
ANALYSES OF MEMORY REQUIREMENTS. (a) MEMORY LOCATIONS (LOC NODES) OF EACH METHOD. (b) WORDLENGTH FOR EACH MEMORY LOCATION. (c) TOTAL MEMORY REQUIREMENTS OF THE TABLES

Table	Ref[8]	Proposed		
Data Format	Data Field + Branch Typess	{ Loc, T, C }	{ R, T_R }	{ Symbol }
MPEG2 TB1	71	13	8	34
MPEG2 TB9	109	17	15	64
MPEG2 TB15	206	35	23	113
CCITT black	170	27	21	104
CCITT white	170	36	12	104
JPEG Cy	234	29	33	162
JPEG CbCr	232	31	30	162

(a)

Table	Ref[8]	Proposed		
Data Format	Data Field + Branch Types	{ Loc, T, C }	{ R, T_R }	{ Symbol }
MPEG2 TB1	7+3	16+4+5	4+6	6
MPEG2 TB9	7+3	16+5+5	4+6	6
MPEG2 TB15	11+3	16+6+6	4+7	11
CCITT black	12+3	16+7+6	4+7	12
CCITT white	12+3	16+7+6	4+7	12
JPEG Cy	8+3	16+6+6	4+8	8
JPEG CbCr	8+3	16+6+6	4+8	8

(b)

Table	Ref[8]	Proposed
MPEG2 TB1	710	609
MPEG2 TB9	1090	976
MPEG2 TB15	2884	2476
CCITT G3 black	2550	2262
CCITT G3 white	2550	2424
JPEG luminance	2574	2504
JPEG chrominance	2552	2524

(c)

TABLE II
PERFORMANCE COMPARISONS OF DIFFERENT VLC DECOMPRESSION SCHEMES. (a) INFORMATION OF THE RANDOM CODEWORD BIT STREAMS. (B) DECODING CYCLES OF EACH SCHEME. (c) COMPARISONS OF THE DECOMPRESSION SYMBOL RATE. (d) COMPARISONS OF THE DECODING THROUGHPUT

Table	# of codewords in the bit stream	# of bits in the bit stream
MPEG2 TB1	2025	5095
MPEG2 TB9	511	2591
MPEG2 TB15	65392	267072
CCITT black	8160	25644
CCITT white	4080	22842
JPEG Cy	131068	431806
JPEG CbCr	65535	222820

(a)

Table	Ref[8]	Ref[9]	Single-Processor	Dual-Processor
MPEG2 TB1	19240	7120	3401	2515
MPEG2 TB9	7726	3102	1442	815
MPEG2 TB15	733072	332464	149407	89186
CCITT black	78700	33804	14107	10519
CCITT white	64400	26922	12063	8277
JPEG Cy	1296358	562874	233056	161455
JPEG CbCr	669955	288355	120883	78596

(b)

Table	Ref[8]	Ref[9]	Single-Processor	Dual-Processor
MPEG2 TB1	10.52495	28.44101	59.54131	80.51690
MPEG2 TB9	6.61403	16.47324	35.43689	62.69939
MPEG2 TB15	8.92027	19.66890	43.76769	73.32092
CCITT black	10.36849	24.13916	57.84362	77.57391
CCITT white	6.33540	15.15489	33.82243	49.29322
JPEG Cy	10.11048	23.28550	56.23884	81.17928
JPEG CbCr	9.782	22.72719	54.21358	83.38211
Average	8.950804	21.41284	48.69891	72.56653

(c)

Table	Ref[8]		Ref[9]		Single-Processor		Dual-Processor	
	Mbps	Ratio	Mbps	Ratio	Mbps	Ratio	Mbps	Ratio
MPEG TB1	63	1	170	2.698	357	5.667	484	7.683
MPEG TB9	39	1	98	2.513	212	5.436	377	9.667
MPEG TB15	98	1	216	2.204	481	4.908	806	8.224
CCITT black	124	1	289	2.331	694	5.597	930	7.500
CCITT white	76	1	181	2.382	405	5.329	591	7.776
JPEG Cy	80	1	186	2.325	450	5.625	650	8.125
JPEG CbCr	78	1	182	2.333	434	5.564	667	8.551
Avg Ratio		1		2.398		5.447		8.218

(d)

presented, too. Based on a dual-processor decoding process, simulation results, and performance comparisons are given for reference. Finally, the conclusion is given in Section III.

II. THE VLC DECODER WITH CODEWORD BOUNDARY PREDICTION

A. Branch Models

To achieve high-performance decoding schemes, it is essential to analyze the characteristics of encoding procedures. An example of Huffman coding procedure is shown in Fig. 1. It combines two symbols having the lowest probabilities and generates a composite symbol having the probability equal to the sum of the combined symbols. By observing the result of this procedure, it is found that the codewords will have the same prefix and length if their source symbols are combined. For example, the codewords of the symbols, such as X5, X6, X7, and X8 in Fig. 1, have the same codeword length, 4-bit, and prefix, 2'b11. When this prefix is recognized, VLC decoders can determine the codeword length and boundary in the bit stream before the decoding procedure is completed. However, tree-based decoding schemes are performed by comparing the bit stream with the branch types which specify the conditions between the parent-nodes and child-nodes. The codeword boundary prediction must be realized by detecting the branch types rather than recognizing the codeword prefix.

The branch types that are presented in [8] and the 2-bit tree structure of MPEG-2 VLC table 15 are now depicted in Fig. 2. In addition to the information of these branch types, two messages are necessary for accomplishing the codeword boundary prediction. The first, called ACT, indicates whether All Child-nodes of a parent-node are Terminal-nodes. The second, denoted S, expresses that some child-nodes are Special terminal-nodes having single bit labels. According to the branch types and the required messages, branch models that can perform the codeword boundary prediction are generated as shown in Fig. 3.

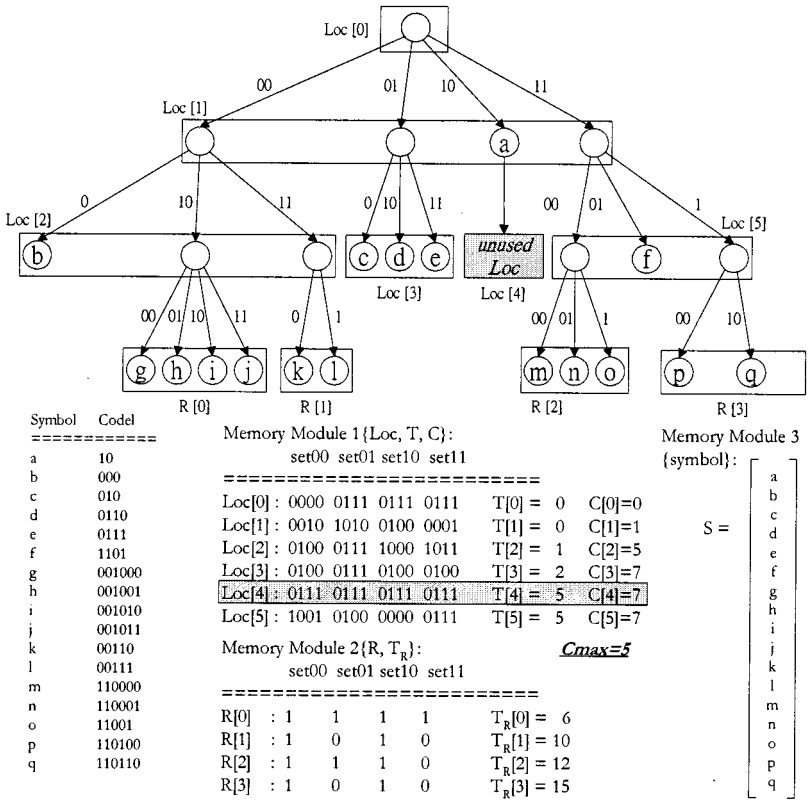

Fig. 4. An example of the proposed memory mapping scheme.

Furthermore, to enhance the prediction efficiency, two Group branch models that indicate all grandchild-nodes are terminal-nodes are created since their source symbols are combined and the same codeword prefix and length are received.

B. Memory-Mapping Scheme

An efficient memory-mapping scheme that can enhance the system performance and reduce the memory requirement is very important for memory-based VLC decoders. Based on the efficient scheme presented in [11], the decoding information, LOC, T, C, and R are mapped onto the memories. To save the memory space, the child-nodes of a parent-node are merged into a LOC. For 2-bit tree structure, each LOC contains 4-set bit assignments of the branch models. The information for calculating the decoded symbol address and the next LOC address is provided by T and C. The ith entry of T is the total number of the terminal-nodes from LOC[0] to LOC[i-1]. On the contrary, the ith entry of C indicates the total number of the nodes having child-nodes from LOC[0] to LOC[i-1]. In addition, the LOC behind the C_{max}th entry only consists of terminal-nodes and unused-nodes. To save the memory space, a 4-bit R instead of the 4-set bit assignments is used for indicating the terminal-nodes and C is eliminated because the next LOC is not required. Besides, T_R represents T in this condition. Based on this proposed scheme, three memories are requested to perform the memory-mapping. The first memory module stores LOC, T, and C. Both R and T_R are loaded into the second memory. The third memory stores the decoded symbols. Beside, {LOC[1], T[1], C[1]} are copied into individual registers to enhance the decoding throughput.

To access more decoding information in one operation, the distribution of LOC in the tree structure has to be fixed. Both LOC[0] and LOC[1] must be located in tree level 0 and level 1, respectively. Because some nodes in tree level 1 do not generate child-nodes, the LOC distribution is not regular in tree level 2. An unused-LOC which consists of 4 unused- nodes is introduced into the tree level 2 as the child-LOC of the unused-node or terminal-node of tree level 1. Consequently, the tree level 2 must be composed of LOC[2 : 5] and the LOC distribution is fixed from tree level 0 to level 2. Because the parent-node of the unused-LOC is treated as having child-nodes, the number of C_{max} has to be updated. An example of the proposed memory-mapping scheme is shown in Fig. 4 where C_{max} is 5. The analyses of memory requirements are given in Table I. Although the branch models need more memory space, the overall memory requirement of the proposed scheme is reduced about 5% $\sim$ 10% compared with [8].

C. Decoding Procedure with Codeword Boundary Prediction

The decoding procedure with codeword boundary prediction is performed by iterating the operation steps shown in Fig. 5, which is a high level description of this decoding procedure. Based on the memory-mapping results shown in Fig. 4, an example of a bit stream $(11001...)_2$ is given as follows for illustration.

Iteration 1, the initial cycle:

1.1) {LOC[1], T[1]} are loaded into registers, {dMDR, T_d}, since every codeword begins with tree level 1. According to the bit_stream[0 : 1] $= 11_2$, the branch

```
Decoding_processor(start, bit_stream, predict, code-length, finish, symbol_address)
input   start, bit_stream;
output  predict, code-length, finish, symbol_address;
{  // Start of procedure, receive the (start & bit_stream):
   if ( start )  { initial cycle = 1;  finish = 0;  L = 0; }   // L is tree level;
   while( finish == 0 ) {
      // Step 1: access the decoding information.
         A_d = bit_stream[ ( 2*L + 1 ) : ( 2*L ) ];   A_p = bit_stream[ ( 2*(L+1) + 1 ) : ( 2*(L+1) ) ];
         // Step 1.1: load the required LOC, T, and branch model for decoding operations.
         if ( initial cycle ) { {dMDR, T_d}_reg = {Loc[ 1 ], T[ 1 ] };   (branch model)_d = dMDR [ set(A_d) ]; }
         else                 { {dMDR, T_d}_reg = {pMDR,  T_p  };   (branch model)_d = (branch model )_p;  }
         // Step 1.2: load the required LOC, T, C, and branch model for prediction operations.
         if (initial cycle)
         {  {pMDR, T_p, C_p}_reg = {LOC[ 2+A_d ],   T[ 2+A_d ],   C[ 2+A_d ]   };
            branch_model_p = pMDR [ set(A_p) ];                                              }
         else if ( next_Loc ≤ C_max )
         {  {pMDR, T_p, C_p}_reg = {LOC[ next_Loc ],  T[ next_Loc ],  C[ next_Loc ]  };
            branch_model_p = pMDR [set(A_p)];                                                }
         else if ( next_Loc > C_max )
         {  {pMDR, T_p  }_reg = {R[new_addr],   T[new_addr ]                  };
            if  (R[A_p]==1 )  { (branch model)_p = terminal;  }
            else              { (branch model)_p = unused;  }                                }
      // Step 2: perform the decoding and prediction operations.
         // Step 2.1: compare the bit strem with the branch models.
         if    ( bit_stream[ 2*(L  )+1 : 2*(L  ) ] is terminal decoded by (branch model)_d      )
         {  finish = 1;  predict = 1;   determine code-length; }
         else if  ( bit_stream[ 2*(L+1)+1 : 2*(L+1) ] is terminal predicted by (branch model)_d  )
         {          predict = 1;   determine code-length; }
         else if  ( bit_stream[ 2*(L+2)+1 : 2*(L+2) ] is terminal predicted by (branch model)_p  )
         {          predict = 1;   determine code-length; }
         else if  ( bit_stream[ 2*(L+3)+1 : 2*(L+3) ] is group-terminal predicted by (branch model)_p  )
         {          predict = 1;   determine code-length; }
         // Step 2.2: calculate the next LOC address and the decoded symbol address.
         OFST = the # of terminal node before the set(A_d) of the LOC in dMDR
         OFSC = the # of non-terminal node before the set(A_p) of the LOC in pMDR;
         if  ( finish == 1 ) {  symbol_address = T_d + OFST-1;  }
         else                {  next_Loc = C_p + OFSC; L=L+1;  }
         if  ( next_Loc > C_max )  {  new_addr = next_Loc – C_max – 1;  }
         initial cycle = 0;
      // Step 3: return the results.
         return(predict & code-length) and (finish & symbol_address);
   }// End of while
} // End of procedure
```

Fig. 5. A high-level description of the decoding procedure with codeword boundary prediction.

model set 11 of LOC[1] in dMDR are selected for decoding operations.

1.2) Because LOC[2 : 5] distribute in tree level 2, the required LOC address = 5 is the sum of the constant = 2 and the bit_stream[0 : 1] = 11_2 = 3. {LOC[5], T[5], C[5]} are accessed from the memory module 1 and stored in registers, {pMDR, T_p, C_p}. According to the bit_stream[2 : 3] = 00_2, the prediction branch model is the set 00 of LOC[5] in pMDR.

2.1) Neither terminal-nodes nor prediction messages are detected form the set 11 of LOC[1]. With the bit_stream[0 : 1], the codeword cannot be decoded, nor can the codeword length be predicted. Based on the set 00 of LOC[5], the bit_stream [2 : 3] is in both ACT and S conditions. After comparing the bit_stream[4 : 5] with the prediction branch model, it is found that the codeword is the special terminal-node and one single bit, 1, remains to be decoded after the

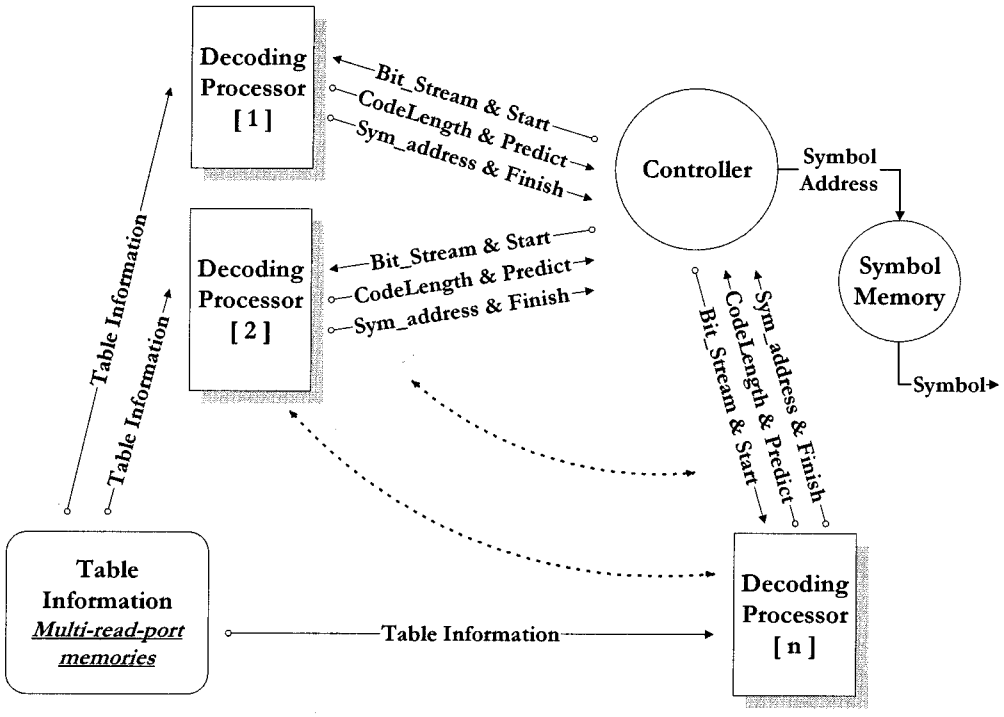

Fig. 6. Block diagram of a parallel-processor VLC decoder.

bit_stream [2 : 3]. Therefore, the 5-bit codeword length is predicted and the "predict" signal is set.

2.2) Because the codeword has not been decoded, it is essential to find the decoding information of the next cycle. The address of the next required {LOC, T, C} is expressed by (C[5]=7) + (OFSC=1) = 8, where the OFSC is the number of nonterminal nodes before the set 00 of LOC[5] in pMDR. But this address = 8 is greater than $C_{\max} = 5$, {R[2], T_R[2]} instead of {LOC[8], T[8], C[8]} are accessed from the memory module 2, where the new address = 2 is the result of $(8 - 5 - 1)$.

3) Return the code-length = 5 and "predict" signal to a controller.

Iteration 2, the second cycle:

1.1) {LOC[5], T[5]} in{pMDR, T_p} are shifted into {dMDR, T_d}.

1.2) {R[2], T_R[2]} are loaded into {pMDR, T_p}.

2.1) The branch model set 00 of LOC[5] in dMDR is used for decoding the bit_stream [2 : 3] which is not the terminal- node. The codeword length needs not be predicted since it has been known in the previous cycle.

Iteration 3, the third cycle:

1.1) {R[2], T_R[2]} are shifted into {dMDR, T_d}.

2.1) The terminal of the codeword is detected by the set10 of R[2] in dMDR.

2.2) The decoded symbol address is ($T_R[2] = 12$) + (OFST= 3)$-1 = 14$ where OFST is the number of terminal nodes before the set 10 of R[2]. Besides, the "finish" signal is enabled.

3) Return the symbol_address $= 14$ and "finish" signal to the controller.

D. Parallel-Processor VLC Decoder

According to the proposed decoding procedure, the valid bit stream of the next codeword is available when the codeword length and boundary are determined. However, the VLC decoding processor has to complete the procedures for finding the decoded symbol address. To increase the decoding throughput, another processor is used for decoding the valid bit stream of the next codeword. A block diagram of a parallel-processor VLC decoder is depicted in Fig. 6. The processor starts the decoding procedure when the "Bit_Stream & Start" are available. Besides, it transmits the "Sym_address & Finish" to notify the controller that the decoding procedure is completed and the symbol address is found. On the other hand, the controller can determine the codeword boundary when the "CodeLength & Predict" are received. Since the codeword lengths are variable, the latter codeword in the bit stream can be decoded earlier than the former long codeword. The controller has to rearrange the decoded symbol addresses in order of the input codeword before accessing the symbol memory. Because the decoding information is identical for every processor, the multi-read-port memory modules are applied to save the memory requirement. As a result, the overhead of the parallel-processor VLC decoder is acceptable since only decoding processor needs to be duplicated.

The number of processors determines the system performance and hardware efficiency of the parallel-processor VLC decoder. If the valid bit stream is not available consecutively, the hardware efficiency will be degraded due to idle operations. On the other hand, the system performance will not be enhanced if the VLC decoder has no available processor to decode the valid bit stream continuously. Based on the coding table given by MPEG and JPEG, simulation results show that the triple-processor VLC decoder has the highest performance because the

```
Dual-processor decoder( )
{
   // Determine the valid bit stream:
   if ( predict[X] )      // X is 1 or 2;
   {  receive the codeword length; determine the codeword boundary and the valid bit stream; }
   // Assess the symbol memory:
   if ( finish[X] )              // X is 1 or 2
   { free the processor[X]; rearrange the decoded symbol address; access the symbol memory; }
   // Assign the bit stream to the processor:
   if ( bit stream is available)
   {
      if       (processor[1] is free)  { start[1] = 1; transmit the bit stream and set busy to processor[1]; }
      else if  (processor[2] is free)  { start[2] = 1; transmit the bit stream and set busy to processor[2]; }
      else                             { queue the bit stream and wait for free processor; }
   }
}
```

Fig. 7. The dual-processor decoding process.

bit stream can be decoded continuously. Nevertheless, several stalls are detected in the allocated processors. Compared to the triple-processor, the decoding throughput of the dual-processor becomes degraded a little bit, but the hardware efficiency is improved significantly. Therefore, the dual-processor decoder structure is selected for multimedia applications.

The dual-processor decoding process is presented in Fig. 7. The controller will determine the codeword boundary and the valid bit stream when the "CodeLength & Predict" are received. When processor transmits "Sym_address & Finish," the controller frees the busy processor and rearranges the decoded symbol address. Then, the symbol can be accessed in order of the input codeword. Besides, the controller assigns "Bit_Stream & Start" to the free processor. If there is no free processor, the controller will queue the bit stream and wait for available processor.

E. Performance Estimation

Based on codeword boundary prediction, performance of the parallel-processor VLC decoder depends on whether the branch models can predict the lengths of the codewords in a given bit stream efficiently. Therefore, random codeword bit streams are generated to evaluate the performance, where the frequency of each codeword coincides with the probability of the related symbol. Performance comparisons of different VLC decompression schemes are given in Table II. It is found that the dual- processor VLC decoder achieves the average decompression rate of 72.5 Msymbol/s operating at 100 MHz. In other words, the decoding throughput of this decoder can be up to 810 Mbps for MPEG-2 DCT coefficient table 15 containing 11-bit symbols and with 60% compression ratio. Besides, with the same clock rate, the average decoding throughput of the proposal is about 1.5 times of a single-processor VLC decoder, 3.4 times of [9], and 8.2 times of [8].

III. Conclusion

In this paper, we present a VLC decompression scheme with codeword boundary prediction to break the iteration bound of a single bit stream. The required prediction messages are added to the coding table information by the proposed branch models. Based on an efficient memory-mapping scheme, the information is loaded into memory modules for both decoding and prediction operations. Hence, the codeword length can be determined before the decoding procedure is completed. To enhance decompression throughput, a parallel-processor VLC decoder is developed for bit stream decoding. Simulation results show that the dual-processor decoder structure is the optimal solution for video films and images applications. Therefore, the VLC decoder with codeword boundary prediction scheme is suitable for current and advanced multimedia systems, such as MPEG-2, H.263, and MPEG-4.

Acknowledgment

The authors would like to thank their colleagues within the SI2 group of NCTU for many fruitful discussions, especially T.-Y. Hsu and J.-J. Jong. The MPC support from NSC/CIC is also acknowledged.

References

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, Sept. 1952.

[2] K. K. Parhi, "High-speed Huffman decoder architectures," *Proc. 25th Asilomar Conf. Signals, Systems and Computers*, vol. 1, pp. 64–68, 1991.

[3] A. Mukherjee, H. Bheda, and T. Acharya, "Multibit decoding/encoding of binary codes using memory-based architectures," in *Proc. Data Compression Conf.*, Snowbird, UT, Apr. 1991, pp. 352–361.

[4] S.-F. Chang and D. G. Messerschmitt, "Designing a high-throughput VLC decoder Part I—Concurrent VLSI architectures," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 187–196, June 1992.

[5] H.-D. Lin and D. G. Messerschmitt, "Designing a high-throughput VLC decoder Part II—Parallel decoding methods," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 197–206, June 1992.
[6] K. K. Parhi, "High-speed VLSI architecture for Huffman and Viterbi decoders," *IEEE Trans. Circuits Syst. II*, vol. 39, pp. 385–391, June 1992.
[7] A. Mukherjee, N. Ranganathan, and M. Bassiouni, "Efficient VLSI design for data transformations of tree-based codes," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 306–314, Mar. 1991.
[8] A. Mukherjee, N. Ranganathan, J. W. Flieder, and T. Acharya, "MARVLE : A VLSI chip for data compression using tree-based codes," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 203–213, June 1993.
[9] H. Park and V. K. Prasanna, "Area efficient VLSI architectures for Huffman coding," *IEEE Trans. Circuits Syst.*, vol. 40, pp. 568–575, Sept. 1993.
[10] L.-Y. Liu, J.-F. Wang, and J.-Y. Lee, "Cam-based VLSI architecture for daynamic Huffman coding," *IEEE Trans. Consumer Electron.*, vol. 40, no. 3, pp. 282–289, Aug./Sept. 1994.
[11] Y.-S. Lee and C.-Y. Lee, "A memory-based architecture for very-high-throughput variable length codec system," in *Proc. ISCAS'97*, vol. 3, June 1997, pp. 2096–2099.
[12] M. K. Rudberg and L. Wanhammar, "Implementation of a fast MPEG-2 compliant Huffman decoder," *Proc. EUSIPCO'96*, vol. 3, pp. 1467–1470, Sept. 1996.
[13] J.-Y. Wu and L.-G. Chen, "A variable length decoder for MPEG-2," *Proc. 1996 HD-Media Technology and Applications Workshop*, no. A5, pp. 3/13–3/18.