A Generalized Prediction Method for Modified Memory-Based High Throughput VLC Decoder Design

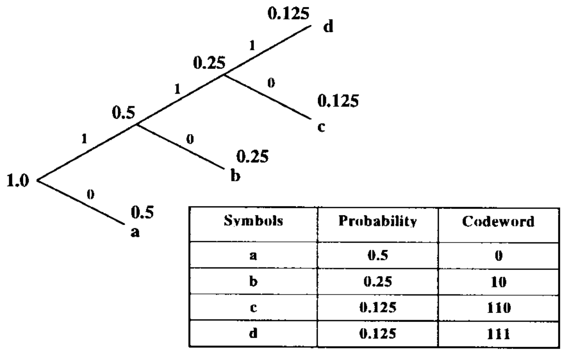
Yew-San Lee, Bai-Jue Shieh, and Chen-Yi Lee

Abstract— **Variable-length code (VLC) is the most popular data-compression technique which has been used in many data-compression standards, such as JPEG, MPEG-2, and H.263. In this paper, we present a new memory-based tree-search algorithm and very large scale integration architecture for VLC decoders which can achieve very high decoding throughput performance. Different coding tables can be implemented by simply changing the contents of the memory without changing the system hardware. The coding table is mapped onto a memory whose space requirement has been minimized by using a new tree data structure and efficient memory-mapping strategy. In addition, we break the recursive dependency of iterative searching operations by predicting method. The proposed algorithm and architecture can predict the searching node and perform parallel operations. As a result, the decoding throughput rate can be enhanced to about three to eight times more than previously announced architecture. The proposed architecture mainly consists of memory modules and simple arithmetic unit. Based on 0.6-μm single poly triple metal CMOS technology and MPEG-2 VLC table-15, the decoder system achieves average decoding throughput rate of 720 Mbits/s at 3 V and a 100-MHz clock rate.**

Index Terms— **Decoding throughput, FIFO, H.263, JPEG, MPEG, tree structure, VLC.**

I. Introduction

Recently, the development of multimedia and communication techniques has changed our lifestyle. It allows the use of pictorial information and photographic images in various scientific, industrial, medical, and consumer applications, such as video-on-demand, distance learning, video conferencing, etc. Hence, the transmitted information has been growing rapidly. In a real situation, the transmission channel has physical bandwidth limitation. As a result, the bottleneck of real-time applications is the transmitted bandwidth. There are two methods to meet such real-time requirements. One is to increase physical channel bandwidth, and the other is to use data-compression techniques. The cost of the first method is too high and not acceptable in practice. Numerous researches have shown that data-compression techniques offers an attractive approach to reduce the communication cost in transmitting a high volume of data over long-haul links via higher effective utilization of the available bandwidth. The number of applications that requires storage and transmission of large volumes of data is steadily increasing. In order to handle such a staggering amount of data, application-specific hardware algorithms and custom very large-scale integration (VLSI) architectures for data compression have to be developed as standard components for communication and image processing systems.



Symbols	Probability	Codeword
a	0.5	0
b	0.25	10
c	0.125	110
d	0.125	111

Fig. 1. An example of the VLC code.

Symbol	Code
a	11
b	000
c	101
d	1000
e	01
f	1001
g	001

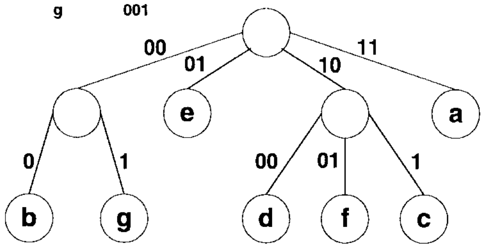


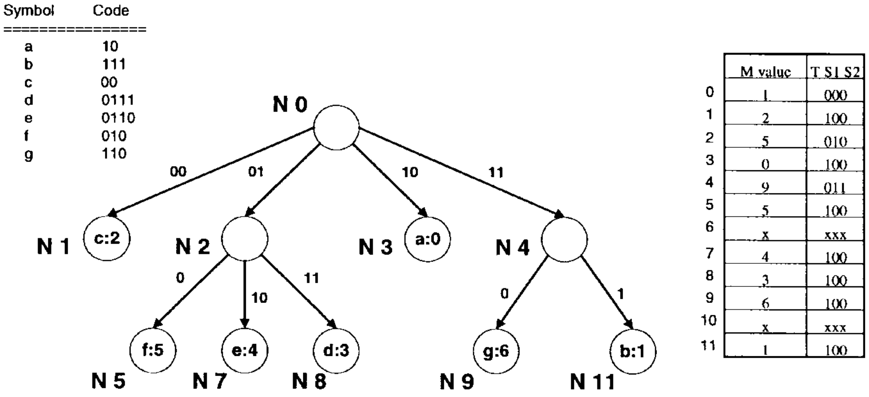
Fig. 2. Two-bit VLC code tree structure.

The term *data compression* refers to the process of reducing the amount of data required to represent a given quantity of information. The underlying basis of the reduction process is the removal of redundant data that either provide no relevant information or simply restate that is already known. Video data compression is a major key technology in the field of multimedia applications. Several international committees have concerned about the definition of technical standards like MPEG-2, JPEG, and H.263. The most popular lossless data-compression technique is variable-length code (VLC), also called the Huffman code [1]. It is an optimal code with the average codeword length approximating the source entropy.

Manuscript received November 10, 1997; revised February 2, 1999. This work was supported by the National Science Council of Taiwan, R.O.C., under Grant NSC86-2221-E-009-016. This paper was recommended by Associate Editor T. Fujii.

The authors are with the Institute of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan, R.O.C.

Publisher Item Identifier S 1057-7130(99)04887-9.



Symbol	Code
a	10
b	111
c	00
d	0111
e	0110
f	010
g	110

	M value	T S1 S2
0	1	000
1	2	100
2	5	010
3	0	100
4	9	011
5	5	100
6	x	xxx
7	4	100
8	3	100
9	6	100
10	x	xxx
11	1	100

Fig. 3. The original decoding tree structure.

The idea is to assign a variable length binary string to each fixed length input symbol such that the input symbols with higher frequency have shorter codewords and the less probable symbols are assigned with longer codewords. The procedure is shown in Fig. 1 with a codebook size of 4. With this coding scheme, the symbol sources can be encoded with the average bit rate very close to its entropy. However, the compression data amount varies due to the contents of image data, as well as picture compression mode. Processing bitstream at the peak data rate would be necessary to design a high-throughput rate decoder where any kind of bitstream could be decoded.

Up to now, several special-purpose VLSI architectures have been proposed to implement VLC decoder system. Two classes of architectures have been discussed in the literature, namely hardwired architecture [2]–[7] and memory-based architecture [8]–[12]. Although the VLC codeword length is variable, the decoding process can still be implemented with a hardwired look-up table. But hardwired architecture allows only one coding table to be implemented since all the VLC codes are hardwired permanently by a logic circuit (PLA or ROM) which forms the look-up table. The codes are fixed with implementation and cannot be changed later, which is the major disadvantage with this static scheme. It can increase throughput by using parallel and pipelined techniques. But this approach will not be very practical for large codebook sizes because the hardware complexity will increase rapidly and the speed will be degraded a lot by the nature of the logic circuit. In addition, this type of design needs to use multiple different size look-up table and parallel detection. The hardware cost is too high because of low hardware efficiency.

The VLC code can be represented by tree structure. The decoding processes can be considered as traversing the tree where the route is determined by decoded bitstream. The memory-based architecture can exploit the natural of the tree structure. We can map the tree structure into a memory format useful for decoding process. It has good flexibility since the coding table mapping results can be stored in an on-chip memory. Various coding tables can be implemented by simply changing the contents of the memory dynamically. In addition, this approach can achieve a high decoding rate even for large coding table size.

The motivation behind our research is to develop a high-throughput algorithm and architecture for the VLC decoder system design. In this paper, we propose a new scheme for mapping VLC code tree onto memory, which leads to a memory-space-efficient solution. The proposed scheme will reduce about 20% of memory space requirement, compared to the traditional memory-mapping scheme [8]. We also break the recursive dependency among the iterative tree-search operations, which enhances decoding throughput rate effectively. In addition, we design a high-speed synchronous static random access memory (SRAM) and register file for improving the performance of the designed VLC decoder system.

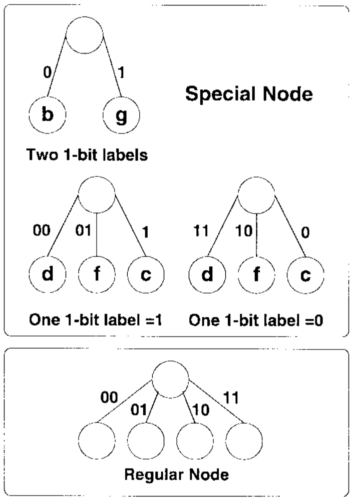


Fig. 4. The node types for decoding tree structure.

This paper is organized as follows. In Section II, we first discuss the tree-based code and tree structure. Then, we present the proposed tree structure and memory-mapping strategy. In Section III, we will describe the proposed decoding algorithm in detail. After that, we propose an efficient memory-based architecture for VLSI implementation of VLC decoder system in Section IV. In Section V, we provide performance evaluation to highlight the achieved improvement. Concluding remarks are made in Section VI.

II. Memory Mapping and Decoding Algorithm

A. Tree-Based Code and Tree Structure

Tree-based codes are codes that can be represented by tree structure. The sequence of zero's and one's on the unique path from the root of the tree to a leaf node represents the code for the symbol represented by that leaf node, as shown in Fig. 2. VLC code is a tree-based code. It can be represented by tree structure such as binary tree, two-bit tree, quad-tree, etc. Thus, the VLC decoding is inherently serial operations, since the tree must be traversed one edge at a time. In order to speed up the decoding process, a multibit tree was proposed, where each edge of the tree represents a maximum of k bits of input code. If the length of a codeword is n bits, then it is represented by n/k edges from the root of the tree to a leaf node of which only the last edge (the one terminating at the leaf node) could possibly have a label less than k bits long. Each node of the k-bit tree can have a maximum of 2^k children.

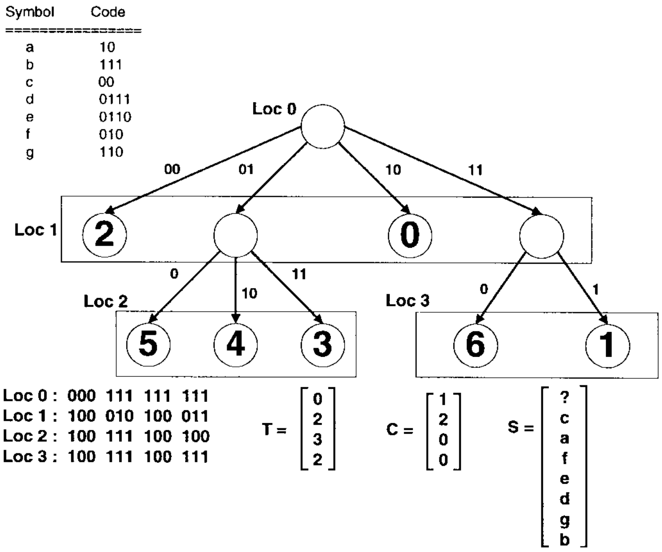
The VLC code is instantaneous and exhaustive, since no codewords can appear as the prefix of any other codeword. As a result, the VLC code tree is an unbalanced tree structure. If we use high-order tree structure to speed up the decoding process, more memory locations will be wasted to store the unused node information. Hence, we choose two-bit tree structure [11], [12] to map the VLC code onto a memory in our proposed decoder system. It can reduce about 30% of tree nodes compared with binary tree structure. Thus, the memory space requirement will be reduced effectively. Besides, the decoding process can be sped up to meet the high throughput requirement.

B. Decoding Tree Structure and Memory-Mapping Strategy

In VLC coding, the coded data is normally sent through a continuous stream of bits with no specific guard-bit(s) assigned to separate between two consecutive symbols. As a result, decoding procedure in this case must recognize the code length as well as the symbol itself. Using tree structure of VLC code, we can simply perform the decoding process by a tree traversal, starting from the root to the relative leaf nodes (symbol node). With input bitstream data, we can decide which child node is the searching destination. We are unable to find out the next searching node before getting relative input bitstream data. The searching operations continue until a terminal node is reached, which means that a complete decoded symbol has been found. The dependency between searching steps and the input bitstream data limits the decoding throughput. It becomes a challenge to break the recursive dependency between the searching steps. In this section, we will propose a new tree structure and memory-mapping scheme which breaks the decoding throughput bottleneck and can perform parallel operations. We address both issues of memory management and fast access to a source symbol.

In our decoding algorithm, we use two-bit tree structure for improving decoding speed, as shown in Fig. 3. Each node has "T S1 S2" three-bits used to indicate the situation of that node. The tree structure contains three types of nodes that need to be mapped onto the memory: 1) regular nodes; 2) special nodes; and 3) terminal nodes. The regular node is a nonleaf node, where all the edges from the node to its child nodes have labels of 2 bits (node N0 in Fig. 3). The special node is a nonleaf node, in which at least one of the edges connecting the node to its child node has a single bit label (node N2 in Fig. 3). There are three cases for this node type, as shown in Fig. 4. The terminal node is a leaf node that represents an output symbol (nodes N1, N3, N5, N7, N8, N9, and N11 in Fig. 3). Note that both N6 and N10 locations are unused. The data field for a nonleaf node (regular or special) contains the searching information M_i for that node, and the data field for a terminal node contains memory location of the symbol at that node. The three control bits T, S1, and S2 are used to identify the different types of nodes. The function of these control bits are listed as follows:

		T	S1	S2
Case 1:	Regular Node	0	0	0
Case 2:	Special Node			
i)	Two 1-bit labels	0	1	1
ii)	One 1-bit label = 0	0	1	0
iii)	One 1-bit label = 1	0	0	1
Case 3:	Terminal Node	1	0	0
Case 4:	Unused Node	1	1	1



Symbol	Code
a	10
b	111
c	00
d	0111
e	0110
f	010
g	110

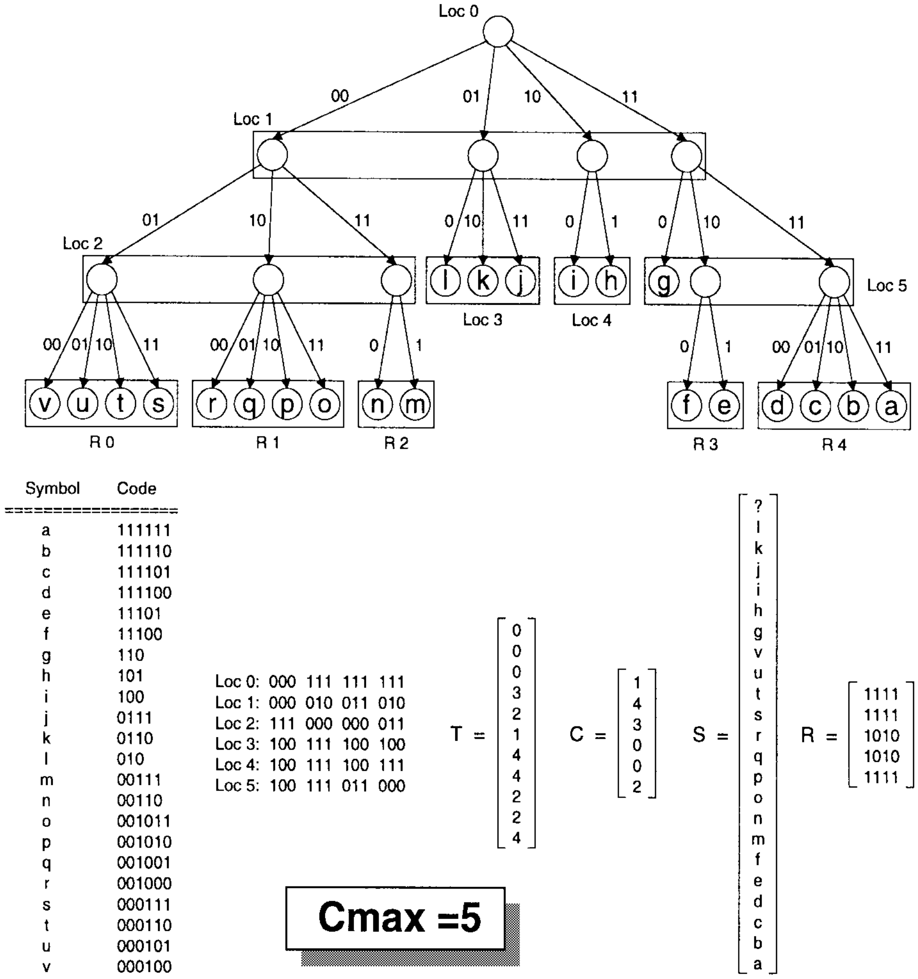
Loc 0 : 000 111 111 111
Loc 1 : 100 010 100 011
Loc 2 : 100 111 100 100
Loc 3 : 100 111 100 111

$$T = \begin{bmatrix} 0 \\ 2 \\ 3 \\ 2 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} \quad S = \begin{bmatrix} ? \\ c \\ a \\ f \\ e \\ d \\ g \\ b \end{bmatrix}$$

Fig. 5. Decoding tree structure with "Loc" nodes.

The searching dependency problem still exists in the tree structure of Fig. 3. To solve this problem, we merge the child nodes of the same parent node into a "Loc," as shown in Fig. 5. The stored values of each "Loc" node are four sets of "T S1 S2" data. With this approach, a greater amount of memory space can be saved, especially for the tree, which has many internal nodes (large coding table size). Besides, we construct two register files "T" and "C" for storing some additional data used to perform prediction. With these memory-mapping results, we can predict the next searching node before we get the input bitstream data. In other words, we can access next searching node information and input bitstream data concurrently.

The ith location of "T" register file stores the number of terminal nodes appeared in ith "Loc" node. In addition, the ith value of "C" register file indicates the number of nodes which has child nodes extension in ith "Loc" node. Finally, we can use current "Loc" node value and "C" value to predict next "Loc" node location and control the decoding steps. The recursive dependency of the iterative searching steps is broken by the proposed scheme. Thus, the decoding throughput rate can be enhanced effectively. With this decoding tree structure



Symbol	Code
a	111111
b	111110
c	111101
d	111100
e	11101
f	11100
g	110
h	101
i	100
j	0111
k	0110
l	010
m	00111
n	00110
o	001011
p	001010
q	001001
r	001000
s	000111
t	000110
u	000101
v	000100

```
Loc 0: 000 111 111 111
Loc 1: 000 010 011 010
Loc 2: 111 000 000 011
Loc 3: 100 111 100 100
Loc 4: 100 111 100 111
Loc 5: 100 111 011 000
```

$$T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 3 \\ 2 \\ 1 \\ 4 \\ 4 \\ 2 \\ 2 \\ 4 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 4 \\ 3 \\ 0 \\ 0 \\ 2 \end{bmatrix} \quad S = \begin{bmatrix} ? \\ l \\ k \\ j \\ i \\ h \\ g \\ v \\ u \\ t \\ s \\ r \\ q \\ p \\ o \\ n \\ m \\ f \\ e \\ d \\ c \\ b \\ a \end{bmatrix} \quad R = \begin{bmatrix} 1111 \\ 1111 \\ 1010 \\ 1010 \\ 1111 \end{bmatrix}$$

Cmax =5

Fig. 6. An example of VLC code tree with $C_{\max}$th "Loc" node.

and memory-mapping scheme, the memory space requirement can be reduced by about 20%, compared to [8].

Aside from the recursive dependency problem, the VLC code tree gets progressively sparse as it grows from the root. This sparsity in the tree may cause tremendous waste of memory space, unless a proper structure and mapping technique are adopted. For the parent nodes of terminal level, it is found that the number of its child nodes is often less than four, and all its child nodes are terminal nodes, such as "Loc-2" and "Loc-5" in Fig. 6. This situation will waste memory space to store unused node information. We therefore modify the memory-mapping strategy for these nodes in order to further reduce memory requirement. It is interesting to note that all "Loc" nodes labeled greater than $C_{\max}$th "Loc" node have only terminal nodes, as shown in Fig. 6 $(C_{\max} = 5)$. This situation happens frequently in VLC code tree structure. For the "Loc" nodes labeled greater than $C_{\max}$ value, we can use 4-bit data "R" to indicate valid terminal nodes, instead of using 12-bit data. With this modification, we can further reduce memory requirement up to 30%.

The discussion mentioned above shows that we have proposed a new tree structure and efficient memory-mapping strategy. However, it is not suitable for hardware implementation because "T" and "C" register files are used to store total number of terminal nodes and extension nodes appearing in ith "Loc" node, respectively. If we want to predict the next "Loc" location, it needs to find out the total number of extension nodes needed before the ith "Loc" node. As a result, we must accumulate the values of $C[0] + C[1] + \cdots + C[i]$. It is the same situation for using "T" values to predict the decoded symbol location. It is difficult to implement a variable number accumulator for our predicting requirement. The hardware cost is too high for implementing such a functional module. In addition, it will form a critical path in the system, resulting in slowing down the system operation rate, especially for a high order of i. Consequently, the decoding throughput will be limited by this critical path.

To cope with this problem, both "T" and "C" register files must be exploited to store accumulation value for avoiding the requirement of the accumulator. It is now recognized that they need to be stored with the total number of terminal nodes or extension nodes appearing from 0 to $(i-1)$th "Loc" node, respectively. In this mapping modification, the size of the register files will be increased because of storing accumulation values. However, real cases show that we still save up to a 20% memory space requirement compared to [8]. The final memory-mapping results of Fig. 5 are given in Fig. 7. A high-level description of the algorithm to generate decoding tree structure and memory-mapping results for "Loc" nodes, "T," "C," and "R" is given in Appendix I.

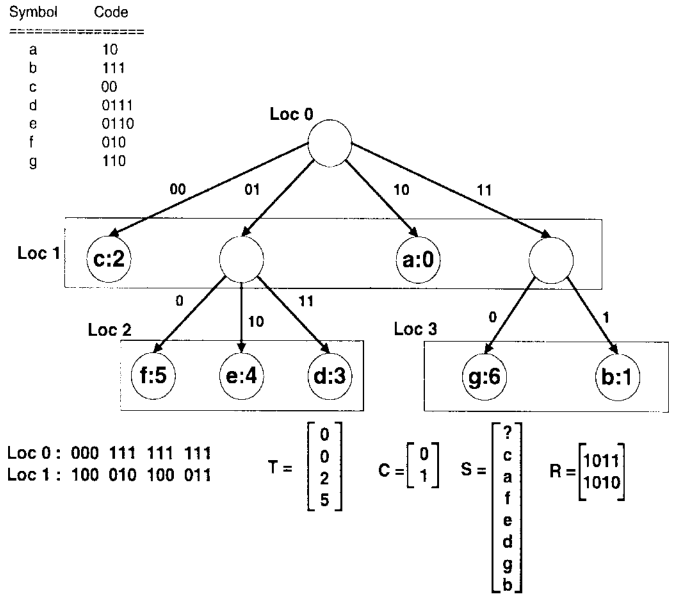


Fig. 7. The final memory-mapping result for the proposed decoding tree structure.

III. Decoding Algorithm

A. Basic Decoding Algorithm

Decoding of an input VLC bitstream is performed by a tree traversal, starting from the root to the terminal node which forms the decoded symbol. We give an example for explaining our decoding algorithm. Assume current location is at the fourth node of "Loc 1" in Fig. 7, which means the previous input bitstream are "11." We can calculate OFSC $=$ 2 from the information of "Loc 1" node. OFSC is the number of $T = 0$ that appears previously to the 4th field in "Loc 1." It indicates the total number of nodes which have child node extension appearing in first through fourth nodes of "Loc 1." As a result, we can simply add $C[1]$ with OFSC to predict the location of next searching node "Loc 3" before we get next input bitstream data "1." Since the label of predicting node "Loc 3" is larger than $C_{\max} = 1$, we will access $R[1]$ instead of "Loc 3." At the same time, we also access $T[3]$, that indicates the total number of terminal nodes appearing before "Loc 3" node. The $R[1], T[3]$ and next input bitstream data will be accessed concurrently. The bitstream data will be used as an index to decide which field of $R[1]$ is the searching destination. Then, we can calculate OFST $=$ 2 from $R[1]$, where the searching destination is the third field of $R[1]$. For the case of searching "Loc" node labeled smaller than $C_{\max}$ value, OFST will be calculated from "Loc" values. OFST will be added with $T[3] = 5$ to predict symbol memory location of decoded symbol "b." Finally, we will access decoded symbol "b" from symbol memory.

Simulation results show that our proposed decoding algorithm is better than traditional method and achieves high decoding throughput rate. The decoding steps are summarized in Appendix II and the decoding flow chart is given in Fig. 8.

B. Advanced Decoding Algorithm

According to the proposed basic decoding algorithm, "Loc0"and "Loc1" must be used for processing in the first decoding cycle of each codeword. In other words, the accesses of "Loc0" and "Loc1" do not depend on input VLC bitstream data. In addition, we define the first searching "Loc" node as "LocX," which depends on bitstream data. From the tree structure, we notice that the "LocX" will be one of "Loc2"–"Loc5" nodes. As a result, the decoder system needs to decide which "Loc" node information is correct for "LocX." Since the possible locations of "LocX" have been known, we can simply use input bitstream data to decide the correct node information for "LocX" in the first decoding cycle, instead of the second decoding cycle.

For an advanced decoding algorithm, we use individual registers to store "Loc0" and "Loc1" information. They will be accessed for processing in the first decoding cycle of each codeword. In addition, we modify the tree structure for simplifying the decision of actual "LocX" location. We consider the input bitstream data $\{\mathrm{bitstream}[1:0]\}$ as an offset value for accessing correct "LocX" information. The strategy is listed as follows:

$$\begin{aligned}&Case\ (bitstream[1:0])\\&\qquad 2'b00 : LocX = Loc2;\\&\qquad 2'b01 : LocX = Loc3;\\&\qquad 2'b10 : LocX = Loc4;\\&\qquad 2'b11 : LocX = Loc5;\\&endcase\end{aligned}$$

Since the bitstream data $\{\mathrm{bitstream}[1:0]\}$ is considered as an offset value, we need to add "Unused Loc" for dummy node in the tree structure, as "Loc4" in Fig. 9. Consequently, T and C register files will be added dummy value into "Unused Loc" location. The "Unused Loc" consists of four sets of unused "T S1 S2" data ("111") and the inserted dummy values of T and C are the same with previous ones, as shown in Fig. 9. The modified decoding operations in the first decoding cycle of each codeword are listed as follows.

1) If the $\mathrm{bitstream}[3:0]$ pattern includes a terminal node in Loc1 $(\mathrm{bitstream}[1:0] = 10)$, we use Loc0, Loc1, and input stream data $\{\mathrm{bitstream}[1:0]\}$ to determine decoded symbol memory location.
2) If the $\mathrm{bitstream}[3:0]$ pattern includes a terminal node in "LocX" (Loc2–Loc5), we use Loc1 and input stream data (bitstream[2, 1]) to determine decoded symbol memory location.
3) If the $\mathrm{bitstream}[3:0]$ pattern contains no terminal node, we use "LocX" and $C[X]$ to predict next searching "Loc" node location.

Because the VLC code is instantaneous and exhaustive, only one case will be detected. With this modified decoding algorithm, the system can search to level-3 of the tree structure in the second decoding cycle. For the basic decoding algorithm mentioned earlier, it needs three decoding cycles for searching to level-3. Thus, the modified decoding algorithm can reduce one decoding cycle time for each codeword. As a result, the decoding throughput rate can be enhanced.

C. Adaptive Decoding Algorithm

By observing VLC code, some long codewords will have the same long parent path. We can consider these long parent

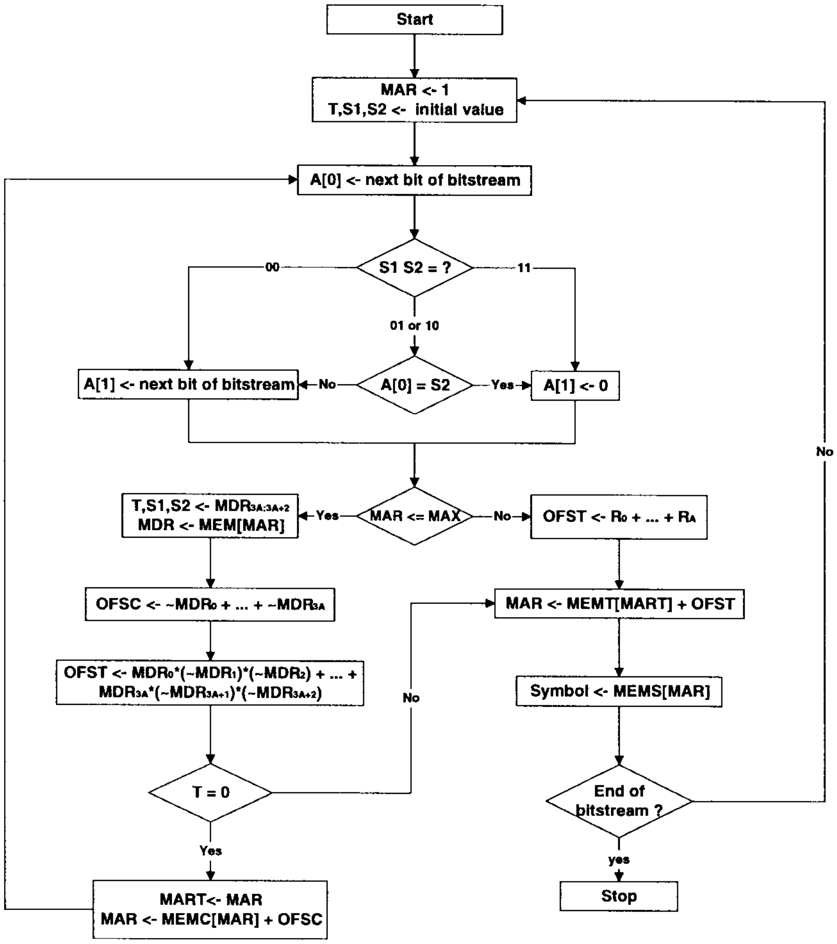


Fig. 8. The basic decoding algorithm flow.

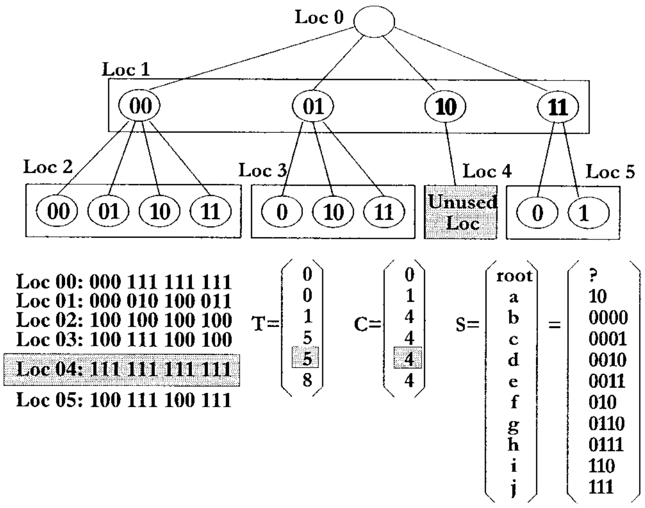


Fig. 9. Decoding tree structure with "Unused Loc" nodes.

paths as special case and detect it in the beginning of decoding steps. Thus, we can reduce more decoding cycles to decode these long codeword patterns. According to VLC algorithm, the probability of long codeword is smaller than for short codeword. However, one long parent path will be shared by many long codeword patterns, as shown in Fig. 10. The probability of a long parent path will gain the sum of probabilities of relative long codeword patterns. Hence, the detection of a long parent path is meaningful and can be exploited to improve decoding rate. We propose a strategy to choose long parent path and let the required detection hardware be low cost and low complexity.

The proposed strategy of choosing long parent path pattern is as follows.

1) The long parent path patterns need to be greater than 8-bit (≥level-4 of tree structure). The long parent path patterns smaller than 8-bit are meaningless.
2) All long codewords sharing the same long parent path must have the same codeword length. It can reduce the complexity of detecting and searching steps.
3) After extracting all the long parent paths from the tree structure, we select short "long parent path" patterns, because their probabilities are higher than long "long parent path" patterns.

We use additional registers to store data for each long parent path pattern. Besides, the searching of the long parent path patterns will be performed in parallel. For example, the maximum codeword length is 16-bit and the amount of "Loc" nodes is 63. The data format for long parent path detection is shown at the bottom of the page. The "parent path length" indicates the valid pattern length in the field of "parent path pattern." When the "Used" field is "1," the VLC input bitstream will be compared with relative "parent path patterns." If it is matched, the next searching "Loc" node location is "next_search_Loc." The "la-

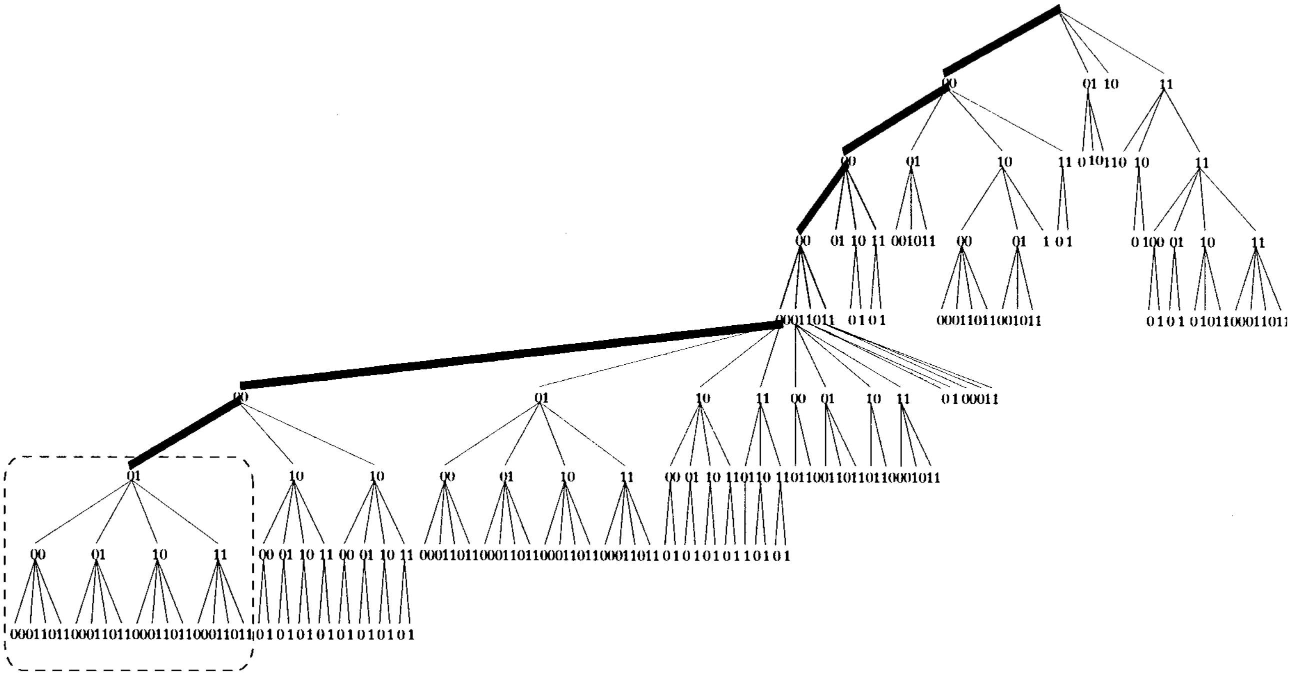


Fig. 10. Using MPEG-2 VLC table 15 as example of long parent path.

bel bit” field is used to indicate the label situation of edge following the long parent path. Note that “label bit” $= 0$ indicates one-bit and “label bit” $= 1$ is two-bit. In Appendix III, a high-level description is given to illustrate the strategy of long parent paths search and relative mapping data format generation.

With long parent paths detection, we can perform an adaptive searching operation. The VLC bitstream can be processed in an adaptive rate which is greater than two-bit/cycle (the rate of basic decoding algorithm). In this manner, more decoding cycles can be reduced for long codeword patterns. The average decoding throughput can be further improved. Finally, we give complete decoding steps in Appendix IV.

IV. Decoder System Architecture

The decoder system architecture is shown in Fig. 11. The major elements of the architecture are synchronous SRAM (SSRAM) modules, a set of combinational logic and register files. We use two memory modules to store “Loc” data and symbols table separately. With this architecture, we can perform parallel access operations. Thus, we can continuously decode next input bitstream and read previously decoded symbol from symbol memory in parallel. The architecture also includes three register files for storing prediction information $(C, T,$ and $R)$. It is important to use a pipeline technique and parallel operations in the decoder architecture for improving the operating speed [4], [5], [13], [14]. The long parent path detection circuit is not shown in Fig. 11 to simplify the data flow description.

An SSRAM was chosen over dynamic random access memory (DRAM) because the decoder system requires a fast and reliable memory module. The speed of memory is important, since the access time is a major performance factor of the decoder system. The SSRAM module is designed with a fully custom methodology that achieves an access time of 4.2 ns with 3-V power supply (size: 256 × 16). We develop basic cells used to compile different size of SSRAM in layout. With this compiled scheme, we can generate any size of memory modules to fit the system and application requirements. It will increase the memory efficiency and shorten the design cycle time.

We use an input (first in first out) FIFO for buffering the VLC input bitstream. In addition, a parallel-to-serial converter is used to extract valid data from input FIFO for decoding operations. When we get the current “Loc” data from tree-node memory (SSRAM-TN), it will be fed into terminal-node calculator and extension-node calculator. The computed results are OFST and OFSC. The sum of OFSC and “C” is the predicted searching “Loc” location. The predicting result will be used to access “T,” “R,” “C,” and SSRAM-TN in parallel. These accessed data are provided to perform next prediction. Meanwhile, the decoded symbol location is predicted by the sum of OFST and “T.” In the case of current “Loc” node’s label larger than $C_{\max}$ value, it is required to calculate OFSTR value by using “R” data. Note that “T” will be added with OFSTR instead of OFST to form the predicting memory location of decoded symbol. Thus, we need a multiplexer to select correct prediction memory location for accessing decoded symbol.

Used	parent path pattern	parent path length	label bit	next_search_Loc
1-bit	16-bit	4-bit	1-bit	5-bit

TABLE I
THROUGHPUT COMPARISON (MPEG-2 VLC TABLE-15)

Bench -mark Compare Items		*Random Pattern*	Symbol: 16660 BitStream: 68207		Image Size: 512 × 512 Symbol: 40398 BitStream: 230176		Image Size: 512 × 512 Symbol: 81697 BitStream: 420512
average decoding cycle per symbol	[8]	12.424		13.810		12.519	
	[9]	5.084		5.768		5.194	
	Ours	1.524		1.660		1.526	
Throughput (Mbps)	[8]	$\frac{F \times S}{12.424}$		$\frac{F \times S}{13.810}$		$\frac{F \times S}{12.519}$	
	[9]	$\frac{F \times S}{5.084}$		$\frac{F \times S}{5.768}$		$\frac{F \times S}{5.194}$	
	Ours	$\frac{F \times S}{1.524}$		$\frac{F \times S}{1.660}$		$\frac{F \times S}{1.526}$	
Throughput Ratio	[8]	1		1		1	
	[9]	2.444		2.394		2.410	
	Ours	8.152		8.319		8.203	

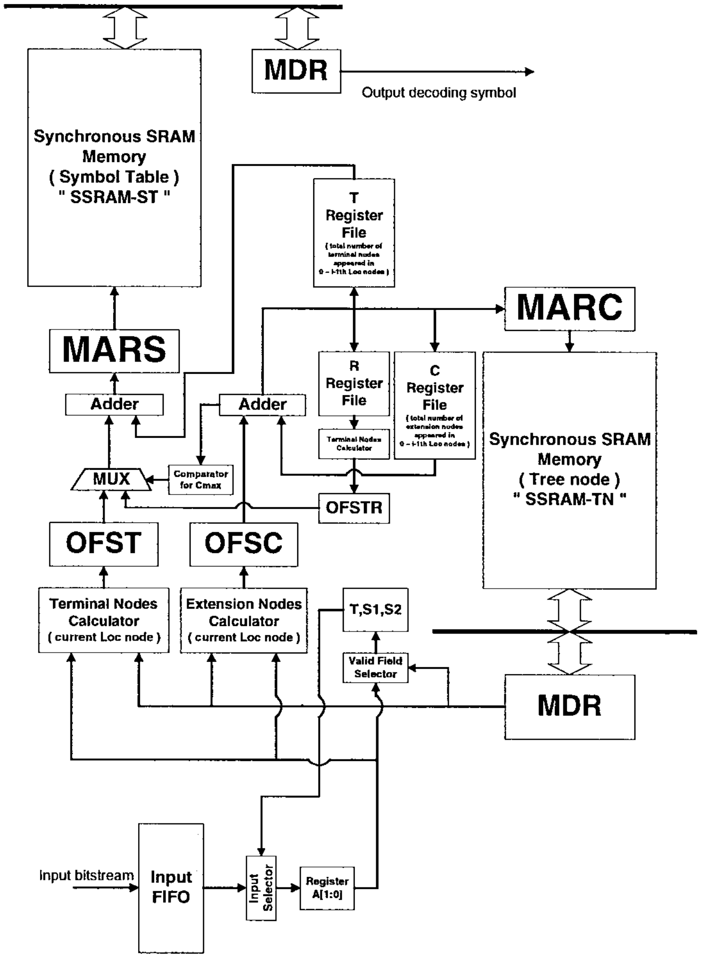

MDR
Output decoding symbol
Synchronous SRAM Memory (Symbol Table) " SSRAM-ST "
T Register File (total number of terminal nodes appeared in 0 ~ i-1th Loc nodes)
MARC
MARS
R Register File
C Register File (total number of extension nodes appeared in 0 ~ i-1th Loc nodes)
Adder
Adder
Terminal Nodes Calculator
Synchronous SRAM Memory (Tree node) " SSRAM-TN "
MUX
Comparator for Cmax
OFSTR
OFST
OFSC
Terminal Nodes Calculator (current Loc node)
Extension Nodes Calculator (current Loc node)
T,S1,S2
Valid Field Selector
MDR
Input bitstream
Input FIFO
Input Selector
Register A[1:0]

Fig. 11. The decoder system architecture.

The critical path of the decoder architecture is from MDR of SSRAM-TN, extension nodes calculator, adder, to MARC register. It forms a loop and these operations must be finished in a decoding cycle time. Consequently, it limits the decoder system clock rate and decoding throughput rate. Besides, the capacity of SSRAM modules and register files are designed large enough to decode 128-entry symbols and a 16-bit codeword length VLC table. The proposed decoder architecture can run up to 100 MHz in 0.6-μm single poly triple metal (SPTM) CMOS technology.

TABLE II
PERFORMANCE COMPARISON

	Process	Clock Rate	Power Supply	Throughput
Ref[8]	2.0um	83.3MHz	5V	95.2Mbps
Ref[9]	1.2um	40MHz	5V	40Mbps
Ours	0.6um	100MHz	3V	720Mbps

V. PERFORMANCE EVALUATION

Memory-based architecture provides the capability to implement different VLC codebooks dynamically by changing the contents of the memory. This flexibility becomes important in signal processing. The previous discussion has shown that the proposed algorithm and architecture can achieve a high decoding rate. In addition, the proposed memory-mapping scheme is efficient in reducing the memory space requirement and achieving high memory efficiency.

We choose MPEG-2 VLC table-15 as test pattern for performance comparison, as shown in Tables I and II. We generate sequential codeword patterns randomly by exploiting the probabilities of relative source symbols. Moreover, two images, which are encoded by MPEG2 VLC codewords, are used to perform the comparison in actual applications. Note that both [8] and [9] are memory-based architectures. Assume, under the same working frequency, the throughput of our proposal is about 3.2 times of [9] and 8 times of [8].

VI. CONCLUSION

In this paper, we have presented a new memory-based tree-search algorithm and VLSI architecture for very-high-throughput VLC decoder system design. The architecture is based on an efficient scheme of mapping new two-bit tree structure onto memory format. Different tables can be updated by simply changing the contents of the memory dynamically. It can be used to implement lossless variable-length coding, as well as tree-base coding. In addition, we break the recursive dependency of iterative searching operations by prediction method. The proposed algorithm and architecture can predict the searching node and perform parallel operations. As a result, the decoding throughput rate can be enhanced up to 8 times, compared to available solutions. The approach is well suited for large coding tables and high throughput system design.

Base on 0.6-μm SPTM CMOS technology and a synchronous SRAM module with an access time of 4.2 ns, the decoder system can operate at 3-V power supply with clock rate up to 100 MHz. According to the MPEG-2 VLC table-15, simulation results show that the proposed algorithm and architecture can achieve average decoding throughput rate of 720 Mbit/s. It is sufficient for many high-throughput real-time applications.

APPENDIX I
ALGORITHM FOR DECODING TREE STRUCTURE AND MEMORY MAPPING STRATEGY

```
Begin

  Create root node;
  /* Create tree node */
  For(i=0; i<Symbol_number; i++)
      Begin
        pointer = root;
        current_level = 0;
        len = length of symbol_code[i];
        For(j=0; j<len; j=j+2)
          Begin
            current_level = current_level+1;
            s1 = symbol_code[i][j];
            IF ( j+ != len)
              Begin
                s2 = symbol_code[i][j+1];
                check = 0;
              End
            Else Begin
                s2 = 0;
                check = 1;
              End
            IF ( check == 1)
              Begin      /* 1-bit Label */
                Create pointer->child[s1] node;
                pointer->child[s1]->tt = 1;
                pointer->child[s1]->label1 = s1;
                pointer->child[s1]->label2 = 0;
                pointer->child[s1]->location = 'unknown;'
                pointer->child[s1]->level = current_level;
                pointer->child[s1]->T = 0;
                pointer = pointer->child[s1];
              End
            Else
              Begin
                Creater pointer->child[s1][s2] node;
                pointer->[s1][s2]->tt = 0;
                pointer->[s1][s2]->label1 = s1;
                pointer->[s1][s2]->label2 = s2;
                pointer->[s1][s2]->location = 'unknown;'
                pointer->child[s1][s2]->level = current_level;
                pointer->child[s1][s2]->T = 0;
                pointer = pointer->child[s1][s2];
              End
          End
        pointer->T = 1;           /* This is a Termaninal node */
        pointer->symbol = i;   /* Setup Symbol value for Terminal node */
      End
      /* Memory-mapping tree node for M and S */
      /* root node using this memory location: 0 */
      loc = 1;
      pointer = root;
      s_pointer = 0;
    For(i=0; i<=tree_level; i++)
    Begin
      current_level = i;
      do (search all tree node which has level == current_level,
          and setup pointer to indicate it)
        {
          do ( search all possible child node of pointer in this oder
              { child0, child00, child01, child1, child10, child11 },
              and all exists child node must do this same process in
              sequential)
          {
            IF ( pointer->child->T == 1 )
              Begin
                pointer->child->location = loc;
                S[s_pointer]= pointer->child->symbol;
                s_pointer = s_pointer+1;
              End
            Else
              Begin
                pointer->child->location = loc;
              End
          }
      IF ( child0 doesn't exist and child1 doesn't exist)
          Begin
              pointer->S1 = 0; pointer->S2 = 0; /* Regular node */
          End
      Else IF(child 0 exist and child1 doesn't exist)
              Begin
                pointer->S1 = 1; pointer->S2 = 0; /* one 1-bit Label=0 */
              End
          Else IF( child0 doesn't exist and child1 exist)
              Begin
                pointer->S1 = 0; pointer->S2 = 1; /* one 1-bit Label=1 */
              End
            Else
              Begin
                pointer->S1 = 1; pointer->S2 = 1; /* two 1-bit Label */
              End
      p= pointer->label1 * 2 + pointer->label2;
      M[loc][3p+0] = pointer->T;
      M[loc][3p+1] = pointer->S1;
      M[loc][3p+2] = pointer->S2;
      loc = loc +1; /* remember all child nodes of pointer has same memory location */
      continue do until no node has same level;
        }
End
  /* Memory-Mapping tree node for T and C */
  T[0] = 0;
  C[0] = 1;
  For( i=1; i<loc ; i++)
    Begin
      c_tmp = 0;
      IF ( M[i][0] == 0) c_tmp = c_tmp + 1;
      IF ( M[i][3] == 0) c_tmp = c_tmp + 1;
      IF ( M[i][6] == 0) c_tmp = c_tmp + 1;
      IF ( M[i][9] == 0) c_tmp = c_tmp + 1;
      C[i] = C[i−1] + c_tmp;
      IF ( c_tmp != 0) Cmax = i;   /* Record the last un-terminal node */
      t_tmp = 0;
      IF ( M[i][0] ==1 and M[i][1] ==0 and M[i][2] ==0) t_tmp = t_tmp +1;
      IF ( M[i][3] ==1 and M[i][4] ==0 and M[i][5] ==0) t_tmp = t_tmp +1;
      IF ( M[i][6] ==1 and M[i][7] ==0 and M[i][8] ==0) t_tmp = t_tmp +1;
      IF ( M[i][9] == and M[i][10] ==0 and M[i][11] ==0) t_tmp = t_tmp +1;
       T[i] = T[i−1] + t_tmp;
    End
       /* Memory-Mapping tree node for R */
       For( i= Cmax+1; i<loc; i++)
        Begin
          For( j=0 ; j < 4; j++)
```

```
              Begin
                 IF( mem[i][3j+0] ==1 and mem[i][3j+1] ==0 and mem[i][3j+2] ==0)
                    R[i][j]= 1;   /* This is Terminal node */
              End
           Else Begin
                 R[i][j]= 0;   /* This is not Terminal node */
              End
        End
     End
  End
```

APPENDIX II
THE BASIC DECODING STEPS

```
           T1: Mar ← 1;    T,S1,S2 ← root node initial value
                A[0] ← next bit of bitstream
                if (S1=S2=0)                  {A[1] ← next bit of bitstream}
           else if (S1=S2=1)                  {A[1]      ← 0}
                     else   if (A[0]=S2)      {A[1] ← 0}
T2: if (MAR ≦ MAX)
        { MDR       ← MEM[MAR]
        T,S1,S2     ← MDR3A:(3A+2)
        OFSC        ← ∼MDR0 + ··· + ∼MDR3A
        OFST        ← MDR0*(∼MDR1)*(∼MDR2)+···+MDR3A*(∼MDR3A+1)*(∼MDR3A+2)
        if (T=0) { A[0] ← next bit of bitstrea
           if (S1=S2=0)                 {A[1] ← next bit of bitstream}
              else   if (S1=S2=1)       {A[1] ← 0}
                     else    if (A[0]=S2)    {A[1] ← 0}
                          else if (A[0]≠S2)  {A[1] ← next bit of bitstream}
           Mar ← MEMC[MAR] + OFSC
           repeat T2 }
        else   {MAR <- MEMT[MAR] + OFST}
        }
else    if (MAR > MAX) { R      ← MEMR[MAR]
                         OFST  ← R0 + R1 + ··· + RA
                         MAR   ← MEMT[MAR] + OFST}
T3: Symbol <- MEMS[MAR]
repeat T1
```

APPENDIX III
ALGORITHM OF SEARCHING LONG PARENT PATH

```
The steps for searching n long parent path:
i=0;
for(layer=4;layer<max layer; layer++){
   if(children of (layer*2)bit path are all 1 bit label){
      Long parent path[i]= (layer*2) bit path;
      parent_lenght[i]= (layer*2)
      label_bit[i]=0; /* 1 bit label */
      next_Loc_addr[i]=child Loc addr of (layer*2)bit
         path;
      i=i+1; }
   else if(children of (layer*2)bit path are all 2 bit
         label){
      Long parent path[i]= (layer*2) bit path;
      label_bit[i]=1; /* 2 bit label */
      next_Loc_addr[i]=child Loc addr of (layer*2)bit
         path;
      i=i+1; }
      if(i>n){break;}
}
```

APPENDIX IV
THE ADVANCED DECODING STEPS

```
/* Define: OFSC= ~MDR[0]+ ··· + ~MDR[ini_A1];
           OFST1=Loc1[0]*~Loc1[1]*~Loc1[2]+ ···
                 +Loc1[ini_A0]*~Loc1[ini_A0+1]*
                 ~Loc1[ini_A0+2];
           OFST =MDR[0]*~MDR[1]*~MDR[2]+ ···
                 +MDR[ini_A1]*~MDR[ini_A1+1]*
                 ~MDR[ini_A1+2];

*/
 T1:
    Loc Memory access: Loc0, Loc1, LocX;
          MDR ← LocX;
    T,S1,S2 ← LocX[3ini_A1:3ini_A1+2];
 if(used[i]=1 && bit stream = parent path[i]){
    Z=parent length[i];
    if(label bit=0) { A[0:1]= { bit[Z],bit[Z+1]}; }
    else            { A[0:1]= { bit[Z], 0};       }
    MAR = next_Loc_addr;   }
 else if(Loc1[3*ini_A0]=1){  /* ini_A0[1:0]
       ← {bit0,bit1} */
    According "Loc0[0:2]" to decide ini_A0[0:1]
          = {bit0,bit1 } or {bit0,0};
    Symbol address = T[1] + OFST1; }
 else if(T=1){    /* ini_A0[1:0] ← {bit0,bit1} */
    Accord "Loc1[3*ini_A0 : 3*ini_A0+2]" to decide
             ini_A1[0:1]= {bit2,bit3}or{bit2,0};
    Symbol address = T[ini_A0+2] + OFST;   }
 else{
    Accord "T S1 S2" to decide A[0:1]
          = {bit4,bit5}or{bit4,0};
    MAR = C[ini_A0+2] + OFSC;   }

 T2:
    Loc Memory access: Loc[MAR];
          MDR ← Loc [MAR] ;
    T,S1,S2 ← MDR[3A:3A+2];
    OFSC=~MDR[0]+ ··· + ~MDR[A]
    OFST=MDR[0]*~MDR[1]*~MDR[2]+ ···
          +MDR[A]*~MDR[A+1]*~MDR[A+2];
 if(T=0){
    Accord "T,S1,S2" to decide A[0:1]=
          {bit[2*cycle],bit[2*cycle+1]}or{bit[2*cycle],0};
    MAR = C[X] + OFSC;
    Repeat T2;   }
 else{ Symbol addr=T[MAR]+OFST; }
T3:
Symbol memory access: Symbol←MEMS[Symbol addr];
Repeat T1;
```

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," in *Proc. 25th Asilomar Conf. on Signals, Systems, and Computers*, vol. 1, pp. 64–68, 1991.

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

[4] 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.

[5] M. T. Sun and S. M. Lei, "A parallel variable-length-code decoder for advanced television application," presented at 3rd Int. Workshop on HDTV, Torino, Italy, Aug. 1989.

[6] M. K. Rudberg and L. Wanhammar, "Implementation of a fast MPEG-2 compliant Huffman decoder," in *Proc. EUSIPCO-96*, vol. 3, pp. 1467–1470, Sept. 1996.

[7] J.-Y. Wu and L.-G. Chen, "A variable length decoder for MPEG-2," in *1996 HD-Media Technology and Applications Workshop*, no. A5, pp. 3/13–3/18.

[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.

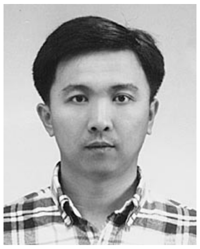
[10] A. Mukherjee, N. Ranganathan, and M. Bassiouni, "Efficient VLSI design for data transformations of tree-based codes," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 306–314, Mar. 1991.

[11] A. Bassiouni and A. Mukherjee, "Multibit and multigroup techniques for data compression," Univ. Central Florida, Aug. 1992, private communication.

[12] 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.

[13] N. Ranganathan and S. Henriques, "A parallel architecture for data compression," presented at IEEE Int. Symp. Parallel and Distributed Processing, Dallas, TX, Dec. 1990.

[14] J. A. Storer, J. H. Reif, and T. Markas, "A massively parallel VLSI design for data compression," in *Proc. IEEE Workshop on VLSI Signal Processing*, 1990.



Yew-San Mountain Lee was born in Muar City, Johore, Malaysia, on July 4, 1971. He received the B.S. and M.S. degrees from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in June 1995 and 1997, respectively. Since September 1997, he has been working toward the Ph.D. degree with research group SI2 at the Institute of Electronics Engineering, National Chiao Tung University.

His research interests include advanced VLSI design for video signal processing, high-performance cell library and memory circuit design, digital phase-locked loops, and related CAD design.

Bai-Jue Shieh was born in Taipei City, Taiwan, R.O.C., on August 9, 1974. He received the B.S. and M.S. degrees from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 1998, respectively. Since September 1998, he has been working toward the Ph.D. degree with research group SI2 an the Institute of Electronics Engineering, National Chiao Tung University.

His research interests include VLSI design for video signal processing, memory circuit design, and mixed-mode IC design.

Chen-Yi Lee received the B.S. degree from National Chiao Tung University, Hsinchu, Taiwan, in 1982, and the M.S. and Ph.D. degrees from Katholieke University Leuven (KUL), Belgium, in 1986 and 1990, respectively, all in electrical engineering.

From 1986 to 1990, he was with IMEC/VSDM, Leuven, Belgium, working in the area of architecture synthesis for DSP. In February 1991, he joined the faculty of the Electronics Engineering Department, National Chiao Tung University, where he is currently a Professor. His research interests mainly include VLSI algorithms and architectures for high-throughput DSP applications. He is also active in various aspects of high-speed networking, system-on-chip design technology, very low bit-rate coding, and multimedia signal processing.