The Design of an Adaptive On-Line Binary Arithmetic-Coding Chip

Shiann-Rong Kuang, Jer-Min Jou, and Yuh-Lin Chen

Abstract— **In this paper, we present a very large scale integration (VLSI) design of the adaptive binary arithmetic coding for lossless data compression and decompression. The main modules of it consist of an adaptive probability estimation modeler (APEM), an arithmetic operation unit (AOU), and a normalization unit (NU). A new bit-stuffing technique, which simultaneously solves both the carry-over and source-termination problems efficiently, is proposed and designed in an NU. The APEM estimates the conditional probabilities of input symbols efficiently using a table lookup approach with 1.28-kbytes memory. A new formula which efficiently reflects the change of symbols' occurring probability is proposed, and a complete binary tree is used to set up the values in the probability table of an APEM. In an AOU, a simplified parallel multiplier, which requires approximately half of the area of a standard parallel multiplier while maintaining a good compression ratio, is proposed. Owing to these novel designs, the designed chip can compress any type of data with an efficient compression ratio. An asynchronous interface circuit with an 8-b first-in first-out (FIFO) buffer for input/output (I/O) communication of the chip is also designed. Thus, both I/O and compression operations in the chip can be done simultaneously. Moreover, the concept of design for testability is used and a scan path is implemented in the chip. A prototype 0.8-μm chip has been designed and fabricated in a reasonable die size. This chip can yield a processing rate of 3 Mb/s with a clock rate of 25 MHz.**

Index Terms— **Arithmetic coding, chip design, lossless data compression.**

I. Introduction

LOSSLESS data compression, which can recover compressed data without any distortion, is a useful strategy in many applications. One strategy is for compressing sources in which no loss of information is allowed, e.g., textual files, executable files, and medical images. Another strategy is for lossy image compression, in which lossless coding is a part of the whole coding algorithm, e.g., those algorithms specified by JPEG [1] and MPEG [2]. Various lossless compression techniques have been developed for sources of different characteristics. There are two types of general lossless coding schemes: dictionary and statistical coding [3]. Dictionary coding achieves compression by identifying repeated substrings and assigning a short code for them by references to other copies defined in a dictionary. Statistical coding makes the code table according to the probabilities that the symbols will occur. A shorter code is assigned to a frequent symbol and a longer code for a rare one. Although these two schemes are expert on different sources, it has been shown in [4] that any practical dictionary-coding scheme can be outperformed by a corresponding statistical-coding scheme.

The process of statistic coding can be split into two stages: *modeling*, which estimates the relative probability for each input symbol, and *coding*, which translates input symbols into a coded stream by the estimated probability. There are two strategies for the modeling: static or adaptive. Static models assume fixed probabilities for each input symbol throughout the coding procedure. In contrast, adaptive models represent the probabilities so far and change them with each new symbol. It has been demonstrated under general conditions that adaptive coding outperforms static [5]. Well-known statistical-coding techniques include Huffman [19] and arithmetic coding [6], [7]. Huffman requires that each symbol be represented by an integer number of bits. On the other hand, arithmetic coding represents the source data as a fraction that assumes a value between zero and one. It can achieve better compression ratios than Huffman coding as long as the statistics are accurate [3]. However, arithmetic coding tends to be slow because in its simplest form it requires at least one multiplication per input symbol. Moreover, if the adaptive coding scheme is applied, an extra division may be needed at every coding cycle. Therefore, algorithm modification and hardware realization for arithmetic coding to prompt the compression speed are critical in real-time applications.

Many fast adaptive arithmetic-coding algorithms have been proposed [8]–[10]. However, because the implementation of multialphabetic arithmetic coding is very complicated, few chips have been reported that use multialphabetic arithmetic-coding algorithm. To make the implementation of arithmetic coding easier and more practicable, the size of the alphabet needs to be reduced to binary so that the coding process can be correspondingly simplified. A faster and simpler implementation of arithmetic-coding algorithm is using the table lookup approach [23]. However, its memory size and compression ratio must make a tradeoff; and the memory size may be very large in order to get a high compression ratio. On the other hand, the Q-coder, an adaptive binary arithmetic-coding chip for the bilevel image compression, has been presented [11], [12]. Some architectural advances in the very large scale integration (VLSI) implementation of arithmetic coders has been made by [13] using the technique of loop unrolling and speculative execution. In addition, the QM-coder, a linear de-

Manuscript received April 21, 1997; revised November 10, 1997 and February 27, 1998. This work was supported by the National Science Council, R.O.C., under Grant NSC-86-2221-E-006-022. This paper was recommended by Associate Editor T. Noll.

The authors are with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan 70101, R.O.C.

Publisher Item Identifier S 1057-7122(98)05299-4.

1057–7122/98$10.00 © 1998 IEEE

scendent of the Q-coder, has been adopted by both JPEG and JBIG still-image-compression algorithms [21]. Fu and Parhi [14] proposed an algorithm which uses redundant arithmetic to obtain further speed-up for the QM-coder. However, all the Q-coder-based arithmetic-coding hardwares described above are designed to compress mainly bilevel image data and may be poor for other types of data. It would be nice to have a compression chip universal enough to quickly compress any type of data that could still achieve a good compression ratio.

In this paper, we present an adaptive division-free arithmetic-coding hardwared algorithm and its VLSI design for lossless compression of universal data. The basic binary arithmetic-coding scheme is first modified to adaptively estimate the occurring probabilities of input symbols, and then the important implementation problems of fixed-precision registers, source termination, and carry-over are solved, in order to make it apposite for hardware implementation. The modified hardwared algorithm iteratively codes input data by the three phases of: 1) probability estimation; 2) arithmetic operation; and 3) normalization. In the probability-estimation phase, a ten-order context (Markov) modeler is designed to gather the conditional probabilities of the input symbols, which are also updated by the modeler with each input bit in order to approach their accurate values and obtain a high compression ratio for different types of files. A new formula which efficiently reflects the change of symbols' occurring probability is proposed, and a complete binary tree is used to set up the values in the probability table. In the arithmetic-operation phase, a simplified parallel multiplier, which requires approximately half of the area of a standard parallel multiplier, is used to perform the multiplication operations of coding. In the normalization phase, a new bit-stuffing technique is applied to solve both the source-termination and carry-over problems simultaneously and efficiently. The proposed hardwared algorithm is then implemented into a VLSI chip. In it, an asynchronous interface circuit with a 1-byte first-in first-out (FIFO) buffer for input/output (I/O) communication is designed. Thus, both I/O and compression operations in the chip can be performed in parallel. Moreover, the concept of design for testability is used and a full scan is implemented in the chip. It is implemented by using the 0.8-μm CMOS technology, and it occupies a silicon area of $4.2 \cdot 4.5$ mm^2, yielding a compression rate of 3 Mb/s with a clock rate of 25 MHz. Since the chip can compress any type of data with a good average compression ratio, it is suitable for use in an environment which has much multimedia (e.g., text data, executing files, and audio and video data) to be stored and losslessly transmitted many times.

In Section II, we will introduce the basic adaptive division-free binary arithmetic-coding scheme and the proposed hardwared algorithm. In Section III, we will describe the system architecture of the chip to realize the proposed algorithm. The detailed hardware design of the key building modules in the chip is also addressed in Section III. The chip realization and analysis of the proposed adaptive binary arithmetic-coding chip are discussed in Section IV. Finally, concluding remarks are made in Section V.

II. Binary Arithmetic Coding

A. Basic Binary Arithmetic-Coding Scheme

Arithmetic coding is a statistical-coding technique that attempts to represent source data with minimal entropy [6], [7]. The encoding process begins with the open interval $[0,1)$ and subdivides it into k subintervals, where k is the number of unique symbols in the source data stream. Each subinterval represents a unique source symbol, and the size of the interval is proportional to that symbol's probability of occurrence. For a given source symbol, the encoder locates the corresponding subinterval, and then divides this interval into subintervals whose ratios are the same as the original cumulative probabilities. The encoder finds the appropriate subinterval for each successive symbol. Since this subinterval is located within the previous interval, it represents not only the present, but also the past symbols. This process continues recursively until the entire source data stream has been encoded, at which time the encoder transmits the final interval. The decoding process of arithmetic coding recovers the source symbols from the received interval using a procedure similar to that of the encoding process. Like the encoder, the decoder begins with the open interval $[0,1)$ subdivided into the same k subintervals. The decoder locates the subinterval in which the received interval resides, yielding the first symbol in the stream. This subinterval is further divided in the same manner to recover subsequent symbols. The procedure terminates when the current and received intervals are equivalent.

Binary arithmetic coding deals with only two input symbols: '1' and '0.' Therefore, the coding process will be simplified correspondingly and be easier to implement. Let A and C represent the width of a subinterval, and the starting point of the subinterval respectively, and let probability $P(\text{'0'} \mid X)$ denote the occurring probability of symbol '0' given the previous input string X. If symbol '0' is encoded, the new A becomes equal to $A \cdot P(\text{'0'} \mid X)$, and the C remains unchanged. If symbol '1' is encoded, then the new A being equal to $A \cdot (1 - P(\text{'0'} \mid X))$; the C is added $A \cdot P(\text{'0'} \mid X)$. If we approximate $A \cdot P(\text{'0'} \mid X)$ with a value of AP, the adaptive encoding algorithm may thus be written as

$C = 0$; $A = 1$;
for (each input symbol) {
  $AP \cong A \cdot P(\text{'0'} \mid X)$; (1)
  if (input symbol == '0') $A = AP$; (2)
  else $\{A = A - AP;\ C = C + AP;\}$ (3)
}
Output C as the encoding result.

Fig. 1 shows the interval subdivision example of binary arithmetic encoding when the following data string is "0 1 1 $\cdots$." The A_i and C_i in Fig. 1 denote the width and starting point of the subinterval of iteration i, respectively. Note $P(\text{'1'} \mid X) = 1 - P(\text{'0'} \mid X)$.

The binary decoding process follows a similar procedure in reverse. The received C is compared at each cycle with the AP, and falls in one of the regions corresponding to the symbol '0' or '1' according to its magnitude. The corresponding symbol is thus decoded. A is then adjusted by the same method as employed in the encoding process. In the case in

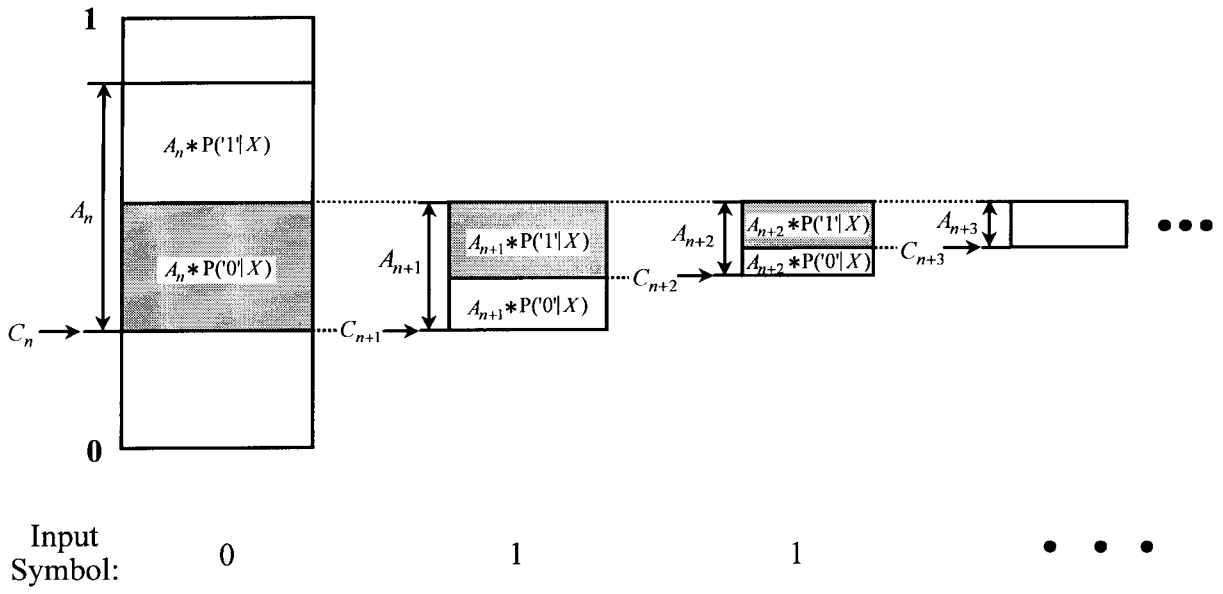


Fig. 1. Interval subdivision example of the binary arithmetic encoding.

which symbol '0' is decoded, C remains unchanged. In the case in which symbol '1' is decoded, C must be subtracted by AP. The adaptive decoding algorithm is as follows:

C is the input of encoding; $A = 1$;
while (the input encoding data does not terminate) {
$AP = A \cdot P(\text{'0'} \mid X)$;
if $(C < AP)$ {
'0' is decoded;
$A = AP$; (4)
}
else {
'1' is decoded;
$A = A - AP$; $C = C - AP$; (5)
}
}.

B. Implementation Issues

On a conceptual basis, hardware implementation of the above algorithms would seem quite simple. However, some practical considerations, which complicate the design of the desired on-line and adaptive binary arithmetic-coding chip, need to be addressed before the practical chip implementation is carried out. They are outlined as follows.

1) *Fixed-precision registers*: How can the arbitrarily long fixed point binary fractions A and C be calculated on a chip employing fixed length registers?
2) *Source termination*: Because we want the coder to work on-line, it cannot know the length of the encoded data stream in advance. How does it know when to terminate?
3) *Carry-over problem*: Since the encoder works on-line, once some symbols are encoded, the encoding of subsequent symbols may alter what has already been outputted due to the carry occurring in (3). This condition occurs when a carry generated in (3) propagates through all bits of register C to affect the encoded symbols that have been shifted out by the normalization operation on C, which solves the fixed-precision registers issue and is discussed later.
4) *Adaptive statistical modeling*: How is the probability of each symbol estimated, and how is their cumulative probability maintained?

The approaches to solve the first three issues will be discussed in this subsection, while the design of the adaptive modeler needed to deal with the fourth issue will be explained in the following subsection.

1) Fixed-Precision Registers: In the designed chip, two fixed-size registers, named register A and register C, are used to keep track of the subinterval width A and the starting point of the subinterval C, respectively. Since finite-length registers are used, the multiplication result in (1) has to be maintained to a fixed number of bits by normalizing registers A and C [6]. During the encoding process, register A is normalized by shifting some bits to the left so that its most significant bit (MSB) is one whenever the value of A is less than half of the initial subinterval width. Register C is also shifted to the left the same number of times, so that bits in the C register always have the same algebraic weight as the aligned bits in the A register. The bits shifted out from the most significant end of register C are buffered and transmitted out as the encoded output.

During the decoding process, registers A and C are required to go through the same normalization process in which the number of bits shifted left out from A and C are discarded; at the same time, the same number of bits from the input encoded string are shifted to the right end of C for the next cycle decoding.

2) Source Termination and Carry-Over: For the source-termination problem, Witten *et al.* [7] use an extra symbol 'EOF' to indicate the situation of source termination. However, the method loses compression ratio and increases the complexity of the coding process. To solve the carry-over problem, a technique called *bit stuffing* is proposed

in [6]. Here, we present a new efficient method to solve the two issues simultaneously. The proposed method uses a termination mark rather than the termination symbol to indicate source termination, thus avoiding the complexity of coding increasing. This is an improved technique of bit stuffing. In the following, we show how this termination mark works.

To solve the two issues together, a 16-b shift register called R is used as the output buffer in the encoder. During the normalization process of encoding, we check the contents of register R. If all its bits are '1''s, we add and shift two stuffing bits with value '0' into the least significant bit (LSB) of R to block carry-over propagating. We know the second stuffing bit may be changed to '1' if a carry is generated in (3) during encoding. Since no carry can propagate to the same bit twice, the first stuffing bit will always be '0' and this feature will be used to indicate the source-termination condition. When the process of encoding ends, the encoder sends an additional successive 17 '1''s out as the termination mark. In the decoding process, the register R is used as the input buffer. If the decoder receives 16 bits all with value '1,' i.e., $R = 0\text{xffff}$, it will check the next two input bits (stuffing bits). If these stuffing bits are "1x" (x: do not care), the decoder will end the decoding process. If the stuffing bits are "01," the decoder will add 1 to the decoded result to amend it. The decoder just ignores the two stuffing bits when they are "00."

However, sometimes the encoder may output 16 consecutive '1''s though the contents of output buffer R are never equal to 0xffff. This will cause the decoder to work incorrectly. The reason why this situation occurs is explained as follows. Assume output buffer R to be equal to 0xfffe. During the normalization, the MSB of R will shift left out and become an encoding output. If the bit shifted from C into R is '1,' the contents of R will become 0xfffd. Then, if the operation $C+AP$ produces a carry when encoding the next symbol, the contents of R must add one and become 0xfffe, which will form 16 consecutive '1''s with the MSB '1' that has been shifted out; however, the contents of R are not equal to 0xffff at that time. An additional up-count counter $C1num$ is used to overcome the problem. The counter $C1num$ is first set to 0 and then counts the number of consecutive '1''s outputted from the encoder. If the output of the encoder is '1,' counter, $C1num$ accumulates one. Otherwise $C1num$ is reset to zero. When $C1num = 16$, which denotes consecutive 16 '1''s outputted from the encoder, the encoder adds and shifts two stuffing bits "00" into R and resets $C1num$.

Note that the decoder reads the encoded string and judges whether register R is equal to 0xffff only during the normalization phase. In order to ensure that the termination mark works, one extra less probability symbol (LPS) must be encoded before the encoder sends consecutive 17 '1''s to terminate the encoding process. As a result, when the decoder decodes the extra LPS, the MSB of A after the arithmetic operation of (4) or (5) will be zero and, therefore, the operations of normalization will be executed so that the termination mark can be detected to end the decoding process.

We now compare our termination mark with the termination symbol used in [7]. Regarding the complexity of coding, the method in [7] using the special termination symbol 'EOF' will increase the coding complexity because it needs at least three input symbols. This makes the probability model work more difficult, and more steps will obviously be needed in the coding process. As for the loss of compression ratio, we estimate it by the two following methods.

When the termination symbol 'EOF' is used:

In the arithmetic-coding process, the subinterval of any symbol must be greater than zero. If A is an 8-b register, the value of A after normalizing will be

$$0\text{xff} > A \geq 0\text{x80}.$$

Moreover, the subinterval of 'EOF,' which is stored in an 8-b register, must be larger than or equal to 0x01. Thus,

$$A \cdot P(\text{'EOF'} \mid X) \geq 0\text{x01}.$$

By (4) and (5), the probability of 'EOF,' $P(\text{'EOF'} \mid X)$, will be

$$\begin{aligned} & A \cdot P(\text{'EOF'} \mid X) \geq 0\text{x01} \\ & \quad \Rightarrow 0\text{x80} \cdot P(\text{'EOF'} \mid X) \geq 0\text{x01} \\ & \quad \Rightarrow P(\text{'EOF'} \mid X) \geq 1/2^7 = 0.78\%. \end{aligned}$$

Therefore, the interval that we can use for all other symbols will reduce 0.78% at least. That is, the overall compression ratio will reduce 0.78% at least.

When our termination mark is used;

During the encoding process, two extra stuffing bits are added and outputted whenever 16 consecutive '1''s are generated. Assuming that the probability of each encoded bit '1' equals 1/2 (based on the assumption that the encoded result usually has the maximum entropy), we will add two stuffing bits every 2^{16} bits on average. This causes the compression ratio to be decreased by a figure of $2/2^{16} = 0.003\%$. Finally, one extra LPS and 17 consecutive '1''s as the termination mark are added fixedly to end the encoding process, only decreasing the compression ratio slightly.

C. Design of Adaptive Modeler

The main challenge in designing the adaptive coding chip is how to estimate the probability of each input symbol and how to maintain their cumulative probabilities. Many adaptive models have been proposed [8]–[10], their main problems being that they are not accurate enough or are too complex with more computation resources. Here, we design a new adaptive and division-free probability estimation modeler with less memory that still achieves a high compression ratio.

Our basic idea is to find the probability values with the most occurring frequencies for symbol '0' in the sources in advance, and to save them in a probability table called **Prob0**, which is used to approximate the conditional probability $P(\text{'0'} \mid X)$. We shall also construct two offset tables called **G0** and **G1**, which store the distances between the new and original probabilities in the probability table **Prob0** for the current input '0' or '1,' respectively. These tables will be used to get the new probability. An n-bit shift register named S is used to record the n previous input symbols, called conditional

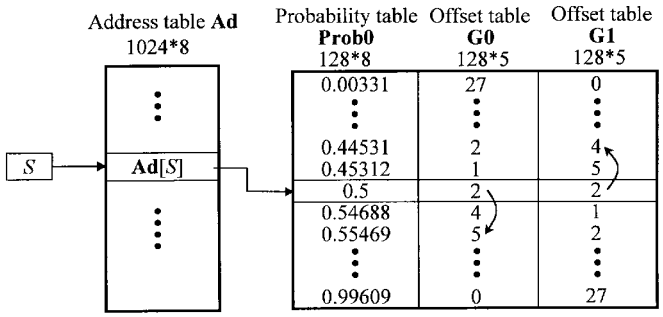


Fig. 2. The relation between table $\mathbf{Ad}$, $\mathbf{Prob0}$, $\mathbf{G0}$, and $\mathbf{G1}$.

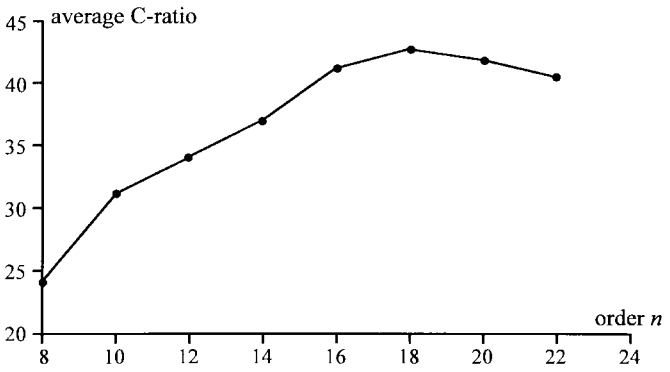


Fig. 3. The curve of average compression ratio versus order n.

states or contexts; this register helps the modeler to estimate the conditional probability $P(\text{`0'} \mid X = S)$ or simplify $P(\text{`0'} \mid S)$ for the next symbol. Because the n-bit S shift register is applied, we say we use an order-n context (Markov) model. Additionally, an address table, denoted as **Ad**, with 2^n pointers, which point to the entries of **Prob0** to find the conditional probabilities for 2^n states, is also constructed. Initially, all the entries in the **Ad** table point to the entry in the **Prob0** table with a probability value equal to 0.5. The value of the S register is used as an index for the $\mathbf{Ad}$ table. Based on the above mechanism, the formulas required to generate condition probability $P(\text{`0'} \mid S)$ of input symbol '0' given the previous input condition recorded in S, and to update the corresponding pointer in the **Ad** table for the next cycle adaptation are as follows:

$$P(\text{`0'} \mid S) = \mathbf{Prob0}[\mathbf{Ad}[S]] \tag{6}$$

$$\mathbf{Ad}[S] = \begin{cases} \mathbf{Ad}[S] + \mathbf{G0}[\mathbf{Ad}[S]], & \text{if symbol '0' is coded} \\ \mathbf{Ad}[S] - \mathbf{G1}[\mathbf{Ad}[S]], & \text{if symbol '1' is coded.} \end{cases} \tag{7}$$

The relation among tables **Ad**, **Prob0**, **G0**, and **G1**, as well as S, is shown in Fig. 2. How to determine the bit number n of S register and how to find the values filled in tables **Prob0**, $\mathbf{G0}$ and $\mathbf{G1}$ are described in the following paragraphs.

1) Determining the Bit Number of Register S: In general, the input data stream is a random process with some statistical relation. That is, there is a correlation between a particular symbol and its neighborhood in the input data stream. For example, the individual probability of character 'u' appearing in the text data is 2.4%, but the probability of character 'u' appearing after character 'q' is 99%. Using the Markov model in the coding will isolate the probability distribution of symbols and decrease the randomness. Therefore, using the high-order Markov model to estimate the probability of symbols occurring will increase the efficiency of arithmetic coding. However, it is not practical to use the high-order Markov model in the multialphabetic arithmetic coding due to memory limits because there exist m^n conditional states for the order-n context model if there are m different symbols in the alphabet. However, for binary arithmetic coding, the high-order Markov model is made possible by the fact that only two different symbols, '1' and '0' are involved. The problem is how to determine the n's value so that its high compression efficiency can be obtained without using a large memory.

In fact, the compression efficiency does not necessarily increase when order n increases. We find that the compression efficiency increases very little or even decreases by increasing order n when n is large. One of the reasons for this is that the order n becomes higher and that the condition states then increase much more; however, the length of the input data stream is finite and it may have been coded completely before the probabilities under the condition states become stable. Therefore, for short source input, the compression efficiency decreases earlier. Fig. 3 shows the average compression ratio of binary arithmetic coding versus the order n made by us for 26 different types of files (including text, binary, and image files). The compression ratio, denoted by C-ratio, is defined as

$$C\text{-ratio} = \left(1 - \frac{\text{the compressed data size}}{\text{the original data size}}\right) \cdot 100\%. \tag{8}$$

The results show that the compression efficiency decreases when order n is larger than 18. In our design, order n is selected as ten to save the required memory so that the whole circuit can be integrated into one chip and still have a good compression ratio. On-chip memory limitation can be overcome by adding external memories for higher order cases to increase the compression ratio.

2) Building the Probability Table **Prob0***:* The probability table **Prob0** contains the first 128 probabilities with the highest occurring frequencies for symbol '0.' To find these probability values, we use a complete binary tree and some counters to simulate the process of calculating conditional probabilities in coding (see Fig. 4). Every node in the tree represents a state of the probability calculating process and contains two data: one is the value of $P(\text{`0'} \mid S)$ at the state, the other is its weight. The weight of a node shown in parenthesis means the relative occurring frequency of the state compared to all other states. The left child of a node is the new state when input symbol '1' is inputted, and the right child is '0.' Initially, the value of the root node is "1/2" and its weight is set as 1000. The value of the right child of the root is "2/3," which means that the new $P(\text{`0'} \mid S)$ is "2/3" after an input symbol '0' is inputted, that the weight of the node becomes 500 because its parent node's weight is 1000, and that the probability of getting input symbol '0' is assumed to be 1/2. By this process, we build a complete binary tree of 255 levels. The partial tree and

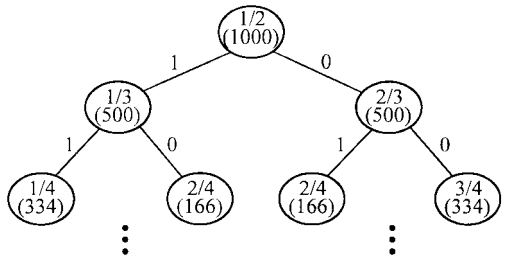


Fig. 4. Part of the probability estimation tree.

TABLE I
THE RESULT OF THE PARTIAL TREE OF FIG. 4

P('0'|S)	64/256 ("1/4")	85/256 ("1/3")	128/256 ("1/2")	171/256 ("2/3")	192/256 ("3/4")
occurring times	334	500	1332	500	334
percentage	11.1%	16.7%	44.4%	16.7%	11.1%

its corresponding results are shown in Fig. 4 and in Table I, respectively. After the complete binary tree of 255 levels is constructed, the first 128 probability values according to their corresponding weights are selected and put into the **Prob0** table, as shown in Fig. 2.

3) Building the Offset Tables **G0** *and* **G1***:* From observing the process of calculating conditional probabilities with counters, we know there are different variations of the conditional probability for symbol '0,' $P(\text{'0'} \mid S)$, during the encoding process. The conditional probability of symbol '0' will change faster (slower) when few (many) input data have been compressed. We use the following new formulas with a parameter *step* to approximate the complex variation of the conditional probability $P(\text{'0'} \mid S)$:

$$P(\text{'0'} \mid S) = \begin{cases} \dfrac{c0/_{step} + 1}{cT/_{step} + 1} \cdot 100\%, & \text{if input symbol is '0'} \\ \dfrac{c0/_{step}}{cT/_{step} + 1} \cdot 100\%, & \text{otherwise} \end{cases} \quad (9)$$

where $c0$ is the number of symbol '0' that has been encoded, cT is the aggregate number of all symbols that have been encoded, and *step* is a tuning parameter used to reflect the degree of the variation of the probability $P(\text{'0'} \mid S)$, $1 \leq step < 256$. The larger the value of *step* is, the more quickly the variation of $P(\text{'0'} \mid S)$ changes.

Using the simulation result of (9), we calculate the distance (offset) value in table **Prob0** between the current $P(\text{'0'} \mid S)$ and the next $P(\text{'0'} \mid S)$ for each state S, and store all offsets into tables **G0** and **G1**. However, how to choose the *step* value to tune $P(\text{'0'} \mid S)$ to approach the variation is still a difficult problem. We have chosen three *step* values: 16, 32, and 64, and use (9) and the proposed approach to simulate the variation of $P(\text{'0'} \mid S)$'s and to do compression. The corresponding compression results for different types of files are given in Table II. In this table, all compression results are obtained by adopting the order-16 context model. Each result in the "entropy" row is obtained by prescanning the whole input file to calculate the actual occurring probabilities of symbols '0' and '1' being used in the following coding process. The compression in the "division" row is achieved by using two counters and the division operation to directly calculate $P(\text{'0'} \mid S)$'s. We find that *step* set to 64 is good for text files since the data in the text files usually has more figure of local correlation, and that $step = 16$ is good for image files since the data in the image files usually has a uniform distribution. On average, the compression ratios of $step = 32$ are good for any type of files (see Table II). Because our coding chip is targeted to general use, such as lossless multimedia data communication applications, $step = 32$ is chosen and practically implemented.

TABLE II
THE COMPRESSION RATIO COMPARISON OF THREE STEP VALUES FOR DIFFERENT TYPES OF FILES USING THE ORDER-16 CONTEXT MODEL

Text Files						
file	t07	t18	t19	t20	t21	t22
entropy	67.78%	67.61%	71.74%	66.56%	70.49%	68.79%
division	64.05%	63.41%	67.37%	62.49%	67.26%	61.26%
step=16	60.65%	60.37%	62.81%	58.68%	64.81%	53.40%
step=32	62.60%	63.00%	65.31%	60.99%	66.97%	57.69%
step=64	62.85%	63.86%	66.02%	61.50%	67.69%	60.16%
Binary Files						
file	t03	t09	t10	t13	t14	t23
entropy	28.94%	42.74%	33.40%	40.45%	30.70%	49.09%
division	21.74%	37.53%	32.47%	38.86%	23.33%	42.72%
step=16	25.82%	39.41%	42.99%	48.68%	26.68%	44.21%
step=32	26.83%	40.90%	43.73%	49.87%	27.44%	46.57%
step=64	25.67%	40.51%	42.58%	49.29%	26.04%	47.31%
Image Files						
file	t15	t05	t12	t24	t16	t17
entropy	50.40%	27.27%	50.94%	14.65%	28.33%	40.70%
division	48.13%	21.99%	49.80%	9.53%	23.87%	37.70%
step=16	49.38%	19.96%	50.49%	10.42%	25.33%	36.12%
step=32	49.37%	19.69%	49.77%	9.41%	24.86%	35.90%
step=64	47.27%	16.36%	47.05%	5.39%	21.52%	33.13%

D. The Proposed Hardwared Algorithm and Comparison

Based on the above discussion, the pseudo-code description of the proposed hardwared algorithm for adaptive binary arithmetic coding is summarized in Fig. 5. Both the processes of encoding and decoding are classified into three phases: probability estimation, arithmetic operation, and normalization. In the probability-estimation phase, the adaptive modeler generates probability $P(\text{'0'} \mid S)$ by using (6). In the arithmetic operation phase, the new values of A and C are calculated by using (2) and (3) for encoding, or (4) and (5) for decoding. Moreover, the adaptive modeler is also updated by using (7) in the arithmetic operation phase. The normalization phase of encoding (decoding) normalizes register A and C, outputs (reads) the encoded string, and deals with the source-termination and carry-over problems by the approach described in Section II-B-2.

For comparison purpose, we considered some different coding methods for coding different types of data. Table III shows the comparison results of compression ratio for different

```
Encoding()
{
        C=0x00;   A=0xff;   R=0x0000;   S=0000000000;
        for (each input binary symbol) {
                phase1:  Generate P('0'|S) by Eq.(5);
                phase2:  AP=A* P('0'|S);
                         if (input symbol=='0')  A=AP;
                         else {
                                 A=A-AP;   C=C+AP;
                                 if (carry occurs)  R++;
                         }
                         Update the adaptive modeler by Eq.(6);
                         Shift the input symbol into S;
                phase3:  while (MSB of A==0)  normalization_of_encoding();
        }
        Encode LPS and then output 17 consecutive '1''s;
}

normalization_of_encoding()
{
        Shift MSB of R as output, shift MSB of C into R, and shift left A and C one bit;
        if (Output=='1') {
                if (C1num==15) {  Output two consecutive '0''s;   C1num++;  }
        }
        else  C1num=0;
        if (R==0xffff) {  Output two consecutive '1''s;  R=0xfffc;  }
}
```

(a)

```
Decoding()
{
        C and R is the encoded input;   A=0xff;   S=0000000000;
        while (true) {
                phase1:  Generate P('0'|S) by Eq.(5);
                phase2:  AP=A* P('0'|S);
                         if (C<AP) { A=AP;  '0' is decoded; }
                         else { A=A-AP;  C=C-AP;   '1' is decoded; }
                         Update the adaptive modeler by Eq.(6);
                         Shift the output symbol into S;
                phase3:  while (MSB of A==0)  normalization_of_decoding();
        }
}

normalization_of_decoding()
{
        Read input symbol;
        Shift left A and C one bit, shift MSB of R into C, and shift the input symbol into R;
        if (Intput=='1') {
                if (C1num==15)
                        Read two stuffing bits "b1b2";
                        if ("b1b2"=="01") { R=0;  C++ };
                        if ("b1b2"=="1x")  End decoding;
                }
                C1num++;
        }
        else  C1num=0;
}
```

(b)

Fig. 5. (a) The pseudo-code of the proposed adaptive binary arithmetic-encoding algorithm. (b) The pseudo-code of the proposed adaptive binary arithmetic-decoding algorithm.

coding methods by adopting the order-16 context model. In the table, ST_32 is the proposed hardwared algorithm with fixed tuning step 32. MF1 and MF2 are the binary arithmetic multiplication-free coding based on the algorithms proposed in [8] and [9], respectively. MDF is the binary arithmetic multiplication-free and division-free coding based on the algorithm proposed in [10]. The three software programs, MF1, MF2, and MDF are implemented by us using their original algorithms, except that an order-16 context model is added for comparison. In addition, Huffman is the adaptive Huffman method [19]. LZW is the compress utility on UNIX [25]. Comparison results show that the proposed hardwared algorithm can compress any type of data with a good compression ratio. The next problem is how to design and implement the hardwared algorithm with less hardware and a sufficient speed for real-time applications.

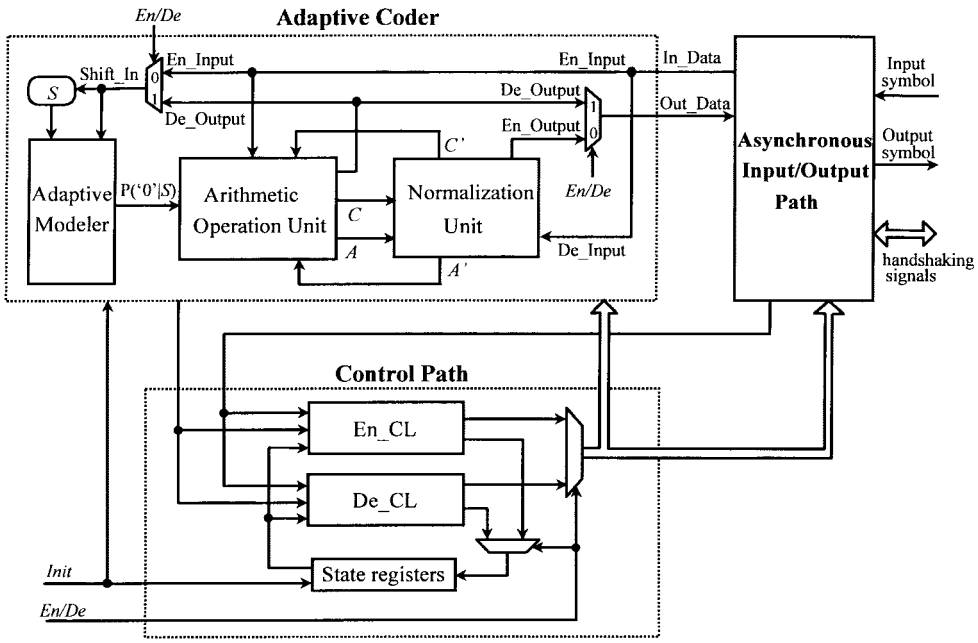


Fig. 6. The functional block diagram of the adaptive binary arithmetic-coding chip.

TABLE III
THE COMPARISON RESULTS OF COMPRESSION RATIO FOR DIFFERENT CODING METHODS

file		compression ratio of different coding methods					
type	name	ST_32	MF1	MF2	MDF	Huffman	LZW
Text	t07	62.60%	58.02%	61.27%	64.55%	47.13%	63.32%
	t18	63.00%	58.05%	60.57%	63.83%	44.70%	63.59%
	t19	65.31%	60.89%	63.86%	67.65%	48.39%	65.78%
	t20	60.99%	57.32%	59.37%	62.70%	43.52%	61.04%
	t21	66.97%	61.61%	63.46%	67.35%	46.57%	67.11%
	t22	57.69%	55.21%	58.15%	61.67%	45.25%	59.13%
average		63.57%	59.03%	61.51%	65.03%	45.91%	64.00%
Binary	t03	26.83%	18.59%	20.87%	21.42%	14.74%	14.41%
	t09	40.90%	32.74%	35.50%	37.72%	27.43%	38.55%
	t10	43.73%	29.68%	31.49%	32.38%	20.21%	42.06%
	t13	49.87%	35.16%	37.34%	38.87%	22.82%	47.99%
	t14	27.44%	20.89%	22.65%	23.06%	16.44%	18.62%
	t23	46.57%	36.82%	40.03%	43.39%	29.61%	44.71%
average		44.04%	30.65%	32.65%	33.79%	21.04%	41.58%
Image	t15	49.37%	45.01%	46.55%	48.12%	10.31%	38.82%
	t05	19.69%	20.15%	21.26%	21.46%	4.22%	6.40%
	t12	49.77%	47.14%	48.62%	49.82%	12.43%	39.84%
	t24	9.41%	8.44%	9.15%	8.69%	10.56%	0.00%
	t16	24.86%	22.66%	23.55%	23.46%	3.09%	10.23%
	t17	36.64%	35.58%	36.79%	37.48%	12.20%	26.18%
average		36.19%	34.12%	35.35%	36.04%	9.80%	25.22%
total average		42.32%	33.61%	35.36%	36.47%	18.39%	36.91%

III. SYSTEM ARCHITECTURE AND MODULE DESIGN

In this section, the system architecture of the proposed hardwared algorithm is first described and then the detailed hardware designs of its main building modules are introduced.

A. System Architecture

Fig. 6 shows the functional block diagram of the adaptive binary arithmetic-coding chip. The chip consists of three main modules: adaptive coder, asynchronous I/O path, and control path. The chip can perform both encoding and decoding functions. One external signal En/De is used to select the desired chip function (encoding or decoding). The functional description of each module is addressed as follows.

The adaptive coder, which contains an adaptive probability estimation modeler (APEM), an arithmetic operation unit (AOU), and a normalization unit (NU), performs the operations of encoding and decoding. The 10-b shift register S records the ten previous input symbols. The APEM generates condition probability $P(\text{'0'} \mid S)$ of symbol '0' given the previous input condition recorded in register S. Once the probability $P(\text{'0'} \mid S)$ is obtained from the adaptive modeler, the AOU uses it to calculate AP as well as the new values of A and C, which are successively sent to the NU to be normalized if necessary. Moreover, the NU will send the encoding result to the I/O path or receive the input symbols to be decoded from the I/O path.

All the above operations are governed by the control path. The control path receives the control statuses from the adaptive coder and the I/O path and generates the control signals for them. Since the adaptive coder can execute encoding or decoding operations, the control path consists of two distinct combinational logic circuits, as shown in Fig. 6. The combinational logic circuit En_CL and De_CL generate the control and next-state signals for encoding and decoding operations, respectively. Another external signal *Init* initializes all data and state registers in the chip.

The asynchronous I/O path has one 8-b input FIFO buffer and one 8-b FIFO output buffer, and generates appropriate handshaking signals for asynchronous I/O. The signals generated are used to control the data transfer between the I/O path and the external environment (device) or the internal adaptive coder. The I/O path can independently work to the other part of the chip, i.e., asynchronously. The adaptive coder may not be idle when data input or output is occurring in the I/O path. The I/O path will suspend the coder when it wants

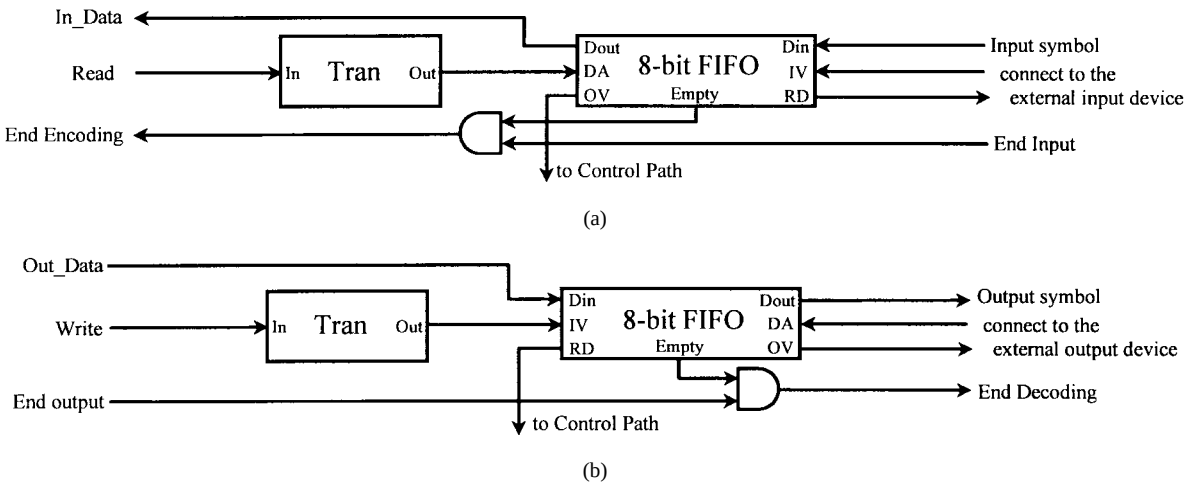


Fig. 7. The architecture of the asynchronous I/O path. (a) The architecture of the input path. (b) The architecture of the output path.

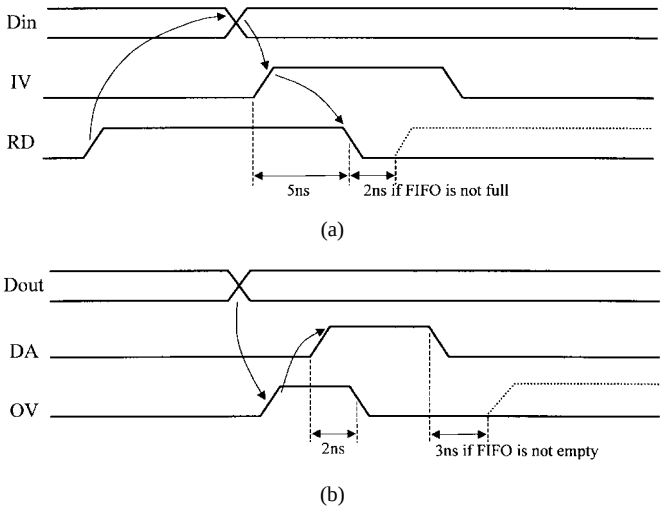


Fig. 8. The handshaking timing of the I/O path. (a) Input path timing. (b) Output path timing.

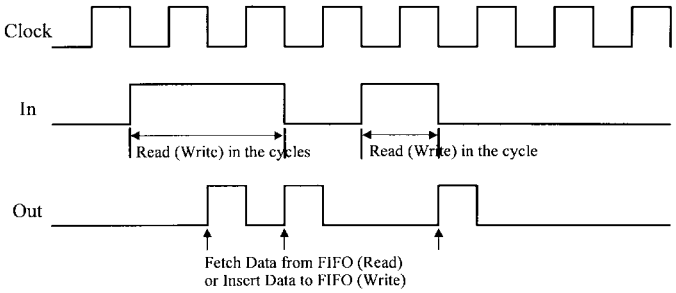


Fig. 9. The timing diagram of the Tran circuit.

to read data from the empty input buffer or to write data out into the jammed output buffer. The architecture of the input and output paths, which consist of an 8-b FIFO and a transfer circuit (Tran), is shown in Fig. 7(a) and (b), respectively. The input and output signals of the 8-b FIFO include empty (buffer empty), din (data input), dout (data output), request for data (RD), input data valid (IV), output data valid (OV), and data accepted (DA). Fig. 8(a) shows the timing diagram of input path's data transfer. The input path initiates the transfer by enabling the RD signal when the buffer is not full. The external input device places data on the din line after it receives the RD signal from the input path and enables the IV signal; the input path then disables the RD signal, which invalidates the data on the din line; the external device then disables the IV signal and the Input path can then initiate another input data transfer. Fig. 8(b) shows the timing diagram of the output path's data transfer. The output path initiates the transfer by placing the data on the dout line and enabling the OV signal. The DA signal is activated by the external output device after it accepts the output data from the dout line; the output path then disables the OV signal, and the external device then disables the DA signal. The output path does not initiate the next output data transfer until the external device shows its readiness for new data by deactivating the DA signal.

The Tran circuit in the I/O path converts the "Read" or "Write" control signal, denoted as In, from the control path into the internal handshaking signal, Out. Then signal Out activates the DA signal of the FIFO buffer. The Read (Write) control signal In will be set to one during the reading or writing cycles. The timing diagram of the Tran circuit is shown in Fig. 9.

B. Module Design

After designing the system architecture of the chip, the next problem is how to implement it with smaller hardware area and higher speed for real applications. The realization of the circuit employs the high-level synthesis concept [15] and the bottom-up approach. The detailed design of the key building modules in this chip, including the APEM, the AOU, and the NU is introduced as follows.

1) APEM: Fig. 10 shows two architectures of the proposed APEM. In them, the $1024 \cdot 8$ b SRAM forms the $\mathbf{Ad}$ table, and the $128 \cdot 18$ b ROM constitutes the **Prob0**, $\mathbf{G0}$, and $\mathbf{G1}$ tables. The contents of the 10-b shift register S are the memory address of the static random-access memory (SRAM). The contents of SRAM are used as the address of ROM. The input coding bit, represented by Shift_In in Fig. 10, is used to update the data stored in the $\mathbf{Ad}$ table. In architecture 1 of Fig. 10(a), the addition and subtraction operations in (7) are concurrently executed by one adder and one subtractor,

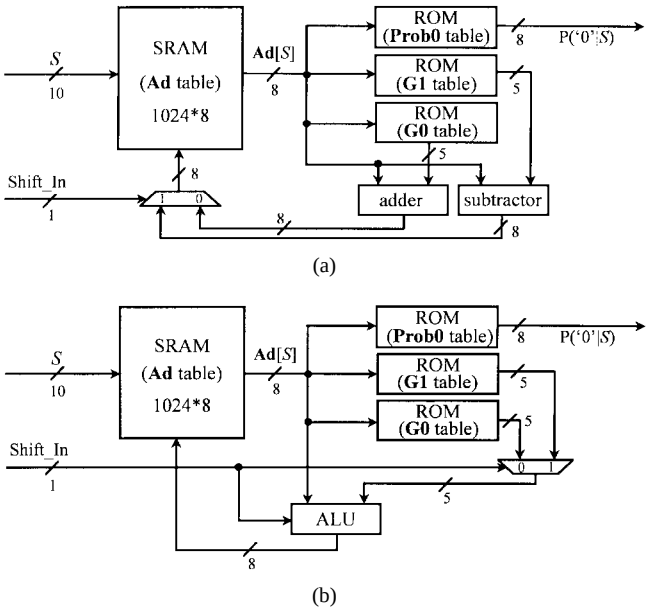


Fig. 10. The architecture of the proposed APEM. (a) Architecture 1. (b) Architecture 2.

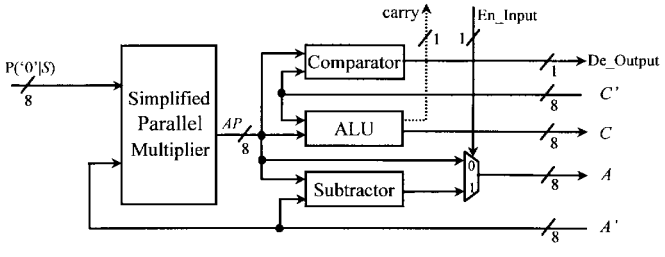


Fig. 11. The architecture of the AOU.

respectively. The produced results are selected by one two-input 8-b multiplexer to obtain the new $\mathbf{Ad}[S]$. However, the two operations in (7) are mutually exclusive. In other words, they will never be simultaneously executed. Therefore, they can share one ALU with addition and subtraction functions, as shown in architecture 2 of Fig. 10(b). The execution function of the ALU in architecture 2 is determined by the Shift_In signal, which is also used to select the correct offset values inputted to ALU from table **G0** or **G1** through a two-input 5-b multiplexer.

The advantages of architecture 2 are higher hardware utilization, simpler interconnection, and less hardware area. Its drawback is that the ALU must execute its operation after Shift_In is known so that more clock cycles are required for decoding. This drawback is avoided by architecture 1, in which the adder and subtractor can execute their operations before Shift_In is obtained. Moreover, the adder and subtractor in architecture 1 can be shared with the operations of (3) or (5) in the AOU (see Fig. 11) to further enhance the hardware utilization and save the hardware area since their active times are without conflict. Therefore, architecture 1 is better than architecture 2 in global view. As a result, architecture 1 is applied and implemented in our chip.

2) AOU: The AOU includes a multiplier, an ALU, a subtractor, and a comparator, as shown in Fig. 11. Once the probability $P(\text{`0'} \mid S)$ is obtained from the adaptive modeler, the multiplier calculates $A \cdot P(\text{`0'} \mid S)$. Subsequently, the subtractor executes the operation of $A - AP$ in (3) or (5). The ALU with only addition and subtraction functions executes the operation of $C + AP$ in (3) or $C - AP$ in (5). As mentioned before, the ALU and subtractor can also be shared to execute the addition and subtraction operations in (7) since their active times are without conflict. This sharing saves the area of one adder and one subtractor, as well as the expense of the additional multiplexer space.

The multiplier is a key component in the adaptive coder since it is located on the system's critical path and its area is the biggest in the unit. To get the faster performance, the parallel multiplier [16], [17] is always adopted at the expense of high area complexity. However, we find that the multiplication in (1) has a special feature in which the input operands A and $P(\text{`0'} \mid S)$ and the product AP are 8 bits. Moreover, the eight LSB's of the product are not used. As a result, a simplified parallel multiplier can be designed to reduce the area complexity without sacrificing any performance or compression efficiency.

Before discussing how to design the simplified multiplier, we first examine the operation of a standard multiplier. Assume that two 8-b numbers X and Y are to be multiplied, the standard multiplier performs the following operation to obtain the product P:

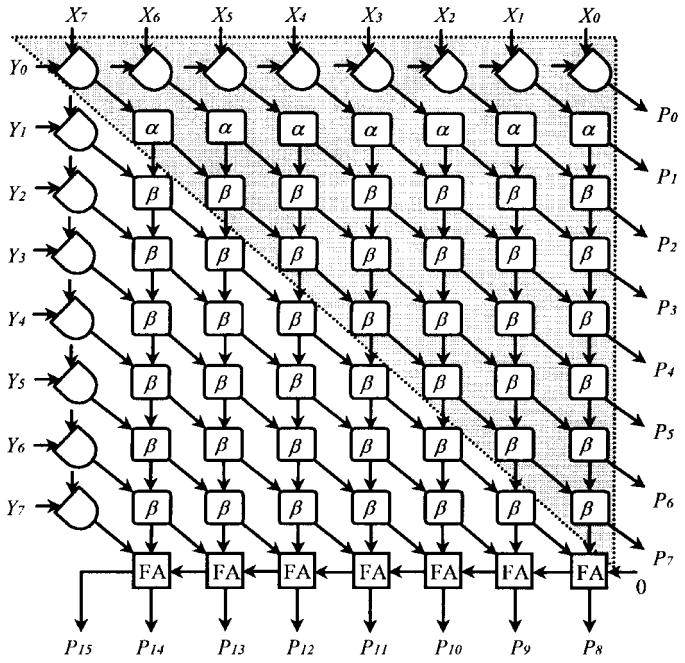
$$P = XY = \left(\sum_{i=0}^{7} X_i 2^i\right)\left(\sum_{i=0}^{7} Y_i 2^i\right) = \sum_{i=0}^{15} P_i 2^i \qquad (10)$$

where X_i, Y_i, and P_i denote the ith bit of X, Y, and P, respectively. For the standard $8 \cdot 8$ parallel multiplier in Fig. 12, the partial sums propagate diagonally in the southeast direction along lines of equal binary weight directions, and the carries propagate downwards along increasing binary weight directions. The delay in this operation is due to the carry propagation through the adder array and the carry–ripple adder, shown at the bottom of Fig. 12. The carry–ripple adder is usually replaced by a carry look-ahead adder to reduce the delay.

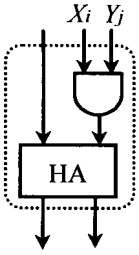
The product can be represented by the sum of two segments: the most and least significant segments mP and lP, i.e.,

$$\begin{aligned} P &= mP + lP \\ &= \sum_{i=8}^{15} P_i 2^i + \sum_{i=0}^{7} P_i 2^i \\ &= \left(\sum_{i=1}^{7} \sum_{j=7-i+1}^{7} (X_i 2^i)(Y_j 2^j)\right) + \left(\sum_{i=0}^{7} \sum_{j=0}^{7-i} (X_i 2^i)(Y_j 2^j)\right). \end{aligned} \qquad (11)$$

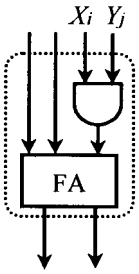
Fig. 12 also shows the various sections of the standard multiplier generating mP and lP. The shaded region contains cells that produce lP. If the eight LSB's of the product are truncated, segment lP is discarded, and approximately half of the adder cells in Fig. 12(a) are not used, but an error in the required product would be introduced. We find that the error involved is tolerable and does not degrade the compression



(a)



(b)



(c)

Fig. 12. (a) The block diagram of an $8 \cdot 8$ parallel multiplier where HA and FA are the half-adder and full-adder cells, respectively. (b) Details of cell α. (c) Details of cell β.

efficiency. The error and compression efficiency analysis will be discussed below. Therefore, we now design a simplified parallel multiplier which generates a product whose value is approximately equal to mP by modifying the left–bottom part of Fig. 12(a) and without using the cells in the shaded part. Fig. 13 shows the details of the simplified multiplier.

A comparison of Figs. 12(a) and 13 reveals that the simplified multiplier yields a product which is not exactly equal to mP. From Fig. 12, it can be seen that all the bit products are generated by performing an AND operation on the bits of the two operands. Let θ be the probability that any multiplicand or multiplier bit is one. The probability $p(i,j)$ that the bit

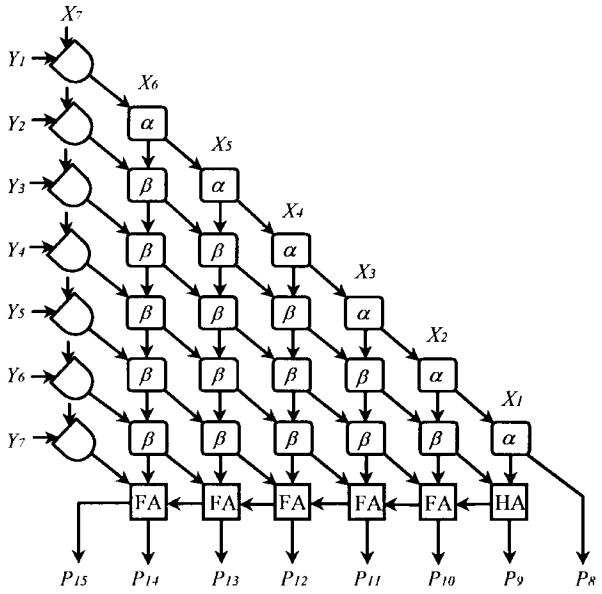


Fig. 13. The simplified $8 \cdot 8$ parallel multiplier.

product at column i and row j is one is equal to θ^2. Let $s(i,j)$ denote the probability that the sum bit of the adder at the ith column and jth row is one and let $c(i,j)$ denote the probability that the carry bit is one. Moreover, let $q(i,j) = 1 - p(i,j)$, $d(i,j) = 1 - c(i,j)$, and $t(i,j) = 1 - s(i,j)$. For a standard parallel multiplier we have the following.

Case 1: When $0 \leq i \leq 6$ and $j = 1$

$$s(i,j) = p(i,j)q(i+1,j-1) + p(i+1,j-1)q(i,j) \quad (12)$$

$$c(i,j) = p(i,j)p(i+1,j-1). \quad (13)$$

Case 2: When $0 \leq i \leq 6$ and $2 \leq i \leq 7$

$$\begin{aligned} s(i,j) = {} & p(i,j)s(i+1,j-1)c(i,j-1) \\ & + p(i,j)t(i+1,j-1)d(i,j-1) \\ & + q(i,j)t(i+1,j-1)c(i,j-1) \\ & + q(i,j)s(i+1,j-1)d(i,j-1) \end{aligned} \quad (14)$$

$$\begin{aligned} c(i,j) = {} & p(i,j)s(i+1,j-1)c(i,j-1) \\ & + p(i,j)s(i+1,j-1)d(i,j-1) \\ & + p(i,j)t(i+1,j-1)c(i,j-1) \\ & + q(i,j)s(i+1,j-1)c(i,j-1). \end{aligned} \quad (15)$$

It can be seen from Fig. 13 that the expected error in the product of the simplified multiplier is equal to the value of the output carry bits from the cells at position (i,j) where $i+j=7$, $0 \leq i \leq 6$, and $1 \leq j \leq 7$. The weight of these bits is equal to 2^{-8}. We call the cells of Fig. 13 at position (i,j) such that $i+j=7$ the diagonal cells. The probability that any output carry bit at the diagonal is one is equal to its expected value. Therefore, the excepted value of the error in the product of the simplified multiplier is given by

$$\varepsilon = 2^{-8} * \sum_{i,j} c(i,j) \quad (16)$$

for (i,j) belonging to the diagonal cells.

TABLE IV
THE COMPRESSION RATIOS OF THE CHIP USING THE STANDARD AND SIMPLIFIED MULTIPLIERS

files	file size (bytes)	standard multiplier	simplified multiplier
		C-ratio	C-ratio
Text1	73622	49.3%	49.1%
Text2	51740	53.2%	52.8%
Text3	108501	51.7%	51.5%
Text4	23680	49.9%	49.6%
Binary1	163840	22.9%	23.2%
Binary2	147456	34.3%	34.8%
Binary3	1064960	35.8%	36.3%
Binary4	98304	40.7%	40.9%
Image1	345600	41.3%	42.0%
Image2	245760	13.7%	13.9%
Image3	921856	41.3%	41.8%
Image4	345600	27.7%	28.1%
average	299243	36.0%	36.5%

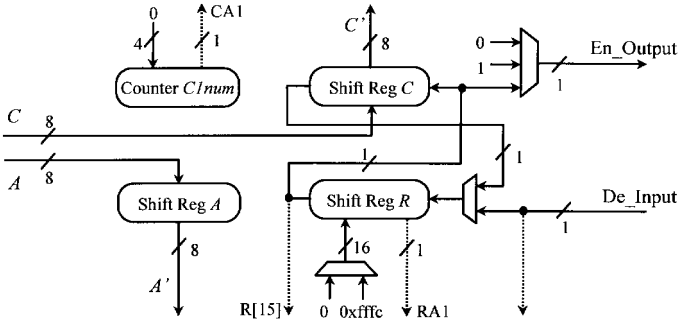


Fig. 14. The architecture of the NU.

Since $0 \le c(i,j) \le 1$, it is thus evident that the expected error ε lies in the range $0 \le \varepsilon \le 0.027$. The excepted error is tolerable in the coding process since one operand $P(\text{'0'} \mid S)$ of it is also an estimated value and the error will not significantly affect the compression efficiency of the chip. The compression ratios for the chip using the standard parallel multiplier and the simplified parallel multiplier obtained from the experiments are summarized in Table IV. From Table IV, it can be seen that a slight difference is incurred. On average, the compression ratios are about the same.

3) NU: In the NU, two 8-b shift registers A and C are used to keep track of the width of the subinterval and the starting point of the subinterval, and one up-count counter $C1num$ is used to count the number of consecutive '1''s coded. In addition, a 16-b shift register R is required to solve the source-termination and carry-over problems. The architecture of the NU is shown in Fig. 14. The dashed lines with arrows in it are the control statuses which will be sent to control path.

Observing the algorithms of Fig. 5, the bound of the contents of $C1num$ is 15 and, thus, a 4-b up-count counter is sufficient to implement the operation of $C1num$. An extra four-input AND gate is integrated into it to examine whether it is equal to 15 and to generate the status signal $CA1$. In addition, the contents of R must also be examined to determine whether they are equal to 0xffff when A and C shift one bit left during the normalization phase of encoding. Therefore, an extra simple circuit, which consists of AND gates, is integrated into the register R to examine whether all bits of register R are '1''s and to generate the status signal $RA1$. Moreover, in the arithmetic operation phase of encoding, register R is increased one when the operation $C + AP$ produces a carry. Although the operation can be executed in one 16-b adder, the hardware area overhead is significant. It is cheaper to use one up-count counter to execute the operation. Consequently, register R must be designed with the ability to perform up-count. On the other hand, register C may add one during the normalization phase of decoding. Similar to register R, shift register C must also be designed with the ability to perform up-count.

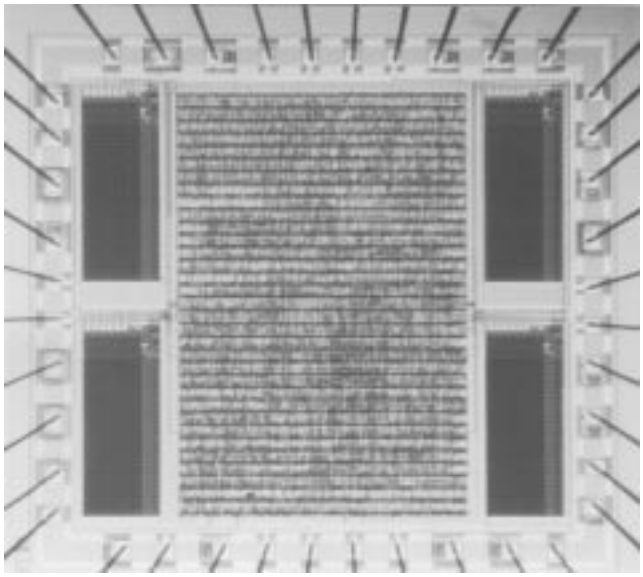


Fig. 15. The microphotograph of our binary arithmetic-coding chip.

IV. CHIP REALIZATION AND ANALYSIS

A prototype of the chip has been implemented and fabricated by using the standard cells of 0.8-μm single-poly double-metal (SPDM) technology [18]. The function of the proposed adaptive arithmetic-coding architecture was first verified by using the Verliog hardware-description language (HDL). We then employed the computer-aided design (CAD) tools OPUS [22] for the creation and verification of our chip layout. The brief design flow is: first, use the preview tool in OPUS to floor-plan blocks in the design; then, preview's block ensemble and cell ensemble are used to perform placement and routing; finally, use Dracula [22] to verify the layout. The microphotograph of the chip layout is shown in Fig. 15, which contains two parts: the layout of standard cells and four 0.25-kbyte SRAM's. Table V shows the characteristics of the adaptive binary arithmetic coding chip. The chip with 40 I/O pins occupies a silicon area of $4.2 \cdot 4.5$ mm^2. The die is mounted in a 40 LD S/B package. The I/O pins can be separated into five parts:

1) power supply pins;
2) chip control pins;
3) pins for data input;
4) pins for data output;
5) pins for testing.

TABLE V
THE CHARACTERISTICS OF THE ADAPTIVE BINARY ARITHMETIC-CODING CHIP

characteristic	proposed chip
technology	TSMC 0.8 μm SPDM
supply voltage	5V
package	40 LD S/B
operation frequency	25 MHz
chip area	4.2*4.5 mm^2
chip complexity	54k gates
scan-mode	yes

Since the coding nature and strategy, and even the integrated circuit (IC) fabrication technology are quite different for the respective chips, it is difficult to directly compare an arithmetic-coding chip with a Huffman one. Compared with the Huffman coding chip of [24], the proposed chip uses about three times the gate count and is seven times slower; note that [24] is bit-parallel and contains only the encoder, but the proposed chip is bit-serial and contains the encoder and decoder. However, the proposed chip achieves two times the compression ratio of [24] (see Tables III and VI). The benefit of a higher compression ratio may be dominated if there are larger quantity of data to be transmitted or stored many times, which is the common situation for today's multimedia Internet environment. The details of scan path and compression speed analysis of the proposed chip will be explained below.

In order to increase the testability of the chip, all registers in the adaptive coder are connected in serial to form a scan path. Moreover, besides the encoding and decoding modes, the test mode is build into the chip by increasing two extra external signals (pins) $T[1:0]$ and a testing control block is constructed. If $T[1:0] =$ "00," the chip works normally and pin En/De is used to determine whether the chip is operating in the encoding or decoding mode. Otherwise, the chip works in the test mode. When $T[1:0] =$ "01," the test data is shifted into the registers on the scan path and the original contents of the registers are shifted out in serial. When $T[1:0] =$ "10," the test data is sent to the functional units of the adaptive coder to calculate the results. When $T[1:0] =$ "11," load the calculated results of functional units into the registers on the scan path in parallel. To do pseudoexhaustive testing, 2^{16} test patterns are used and these test patterns take 0.22 s to finish the testing under a clock rate of 25 MHz. We also can use the same scan path to test the read-only memory (ROM) and SRAM's in the adaptive coder. The scan path in the chip provides 100% fault coverage for the adaptive coder which occupies about 90% of the total device count in the chip. Since the control and asynchronous I/O paths in the chip are not connected by the scan path, its fault coverage approximates to 90%.

The compression speed of the chip under 25 MHz clock rate is approximately 3 Mb/s (see Table VI) and is discussed as follows. During each iteration of the coding process, it takes eight clock cycles to complete the operation of the probability estimation and arithmetic operation phases. The normalization phase is a loop with variable iteration times. The execution delay of each iteration in it is very close to one clock cycle on average since the probability of register R being equal to 0xffff is very small. In each coding cycle, the iteration time of the normalization loop, denoted as *numb*, is equal to the number of left shifts occurring in register A such that its MSB is one. Although we have known that *numb* is bound by seven, it is difficult to predict the average iteration time of the normalization loop. To solve the problem, some sample files are provided for software simulation of the proposed algorithm to count the *numb* in the normalization phase. The percentage distribution of the *numb* is listed in Fig. 16. The results show that the most normalization loop executions are executed zero or one iteration time and that about 0.5 clock cycle is, therefore, needed to complete the normalization phase on average. As a result, the analysis speed is about $25/8.5 \cong 3$ Mb/s to which the chip speed conforms.

TABLE VI
THE COMPRESSION RATIO AND COMPRESSION SPEED OF OUR CHIP FOR DIFFERENT TYPES OF FILES

files	file size (bytes)	Huffman	MAAC	The chip		
		C-ratio	C-ratio	C-ratio	C-time (sec)	C-speed (Mbits/sec)
Text1	73622	44.7%	45.6%	49.1%	0.197	2.99
Text2	51740	48.4%	49.4%	52.8%	0.138	3.00
Text3	108501	46.6%	47.4%	51.5%	0.289	3.00
Text4	23680	45.2%	46.6%	49.6%	0.064	2.97
Binary1	163840	14.7%	20.8%	23.2%	0.458	2.86
Binary2	147456	27.4%	32.6%	34.8%	0.405	2.91
Binary3	1064960	22.8%	31.5%	36.3%	2.898	2.94
Binary4	98304	29.6%	36.1%	40.9%	0.268	2.93
Image1	345600	10.3%	22.2%	42.0%	0.925	2.99
Image2	245760	4.2%	7.4%	13.9%	0.690	2.85
Image3	921856	12.4%	17.7%	41.8%	2.508	2.94
Image4	345600	12.1%	16.7%	28.1%	0.934	2.96
average	299243	18.3%	24.8%	36.5%	0.814	2.95

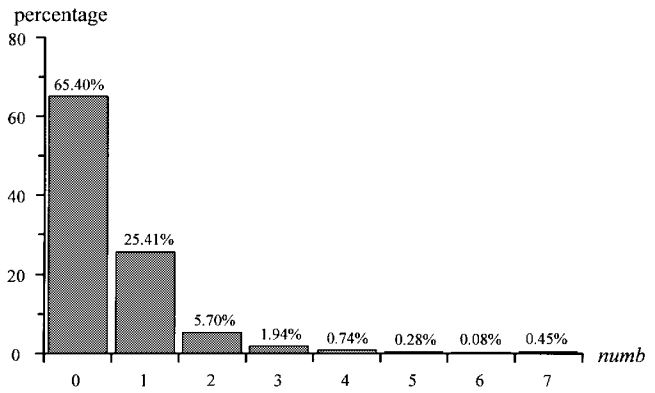


Fig. 16. The percentage distribution of *numb* during the normalization phase.

Besides the compression speed, the compression ratio is also an important evaluation item. Some different types of files were used to test the compression ratio. The results compared with two different schemes, including Huffman and the multialphabetic arithmetic-coding (MAAC) algorithm, are shown in Table VI. Huffman is the adaptive Huffman coding scheme [19]. MAAC was proposed by Jiang in [20]. Twelve sample files are provided for the experiments; their sizes are also listed in the second column under "file size." Compared with other schemes, the compression ratios of this chip are also good. Moreover, the actual compression time (C-time)

and the compression speed (C-speed) of our chip for these files also are shown in Table VI. The average C-speed of our chip is very close to 3 Mb/s.

V. Conclusion

In this paper, we have presented a VLSI design of the adaptive binary arithmetic coding for lossless data compression and decompression. The important implementation issues, including fixed-precision registers, source-termination, and the carry-over problem were all efficiently solved in the design. The key modules of this design include an APEM, an AOU, and a NU. A table lookup approach with 1-kbyte SRAM and 0.28-kbyte ROM were used in the APEM to achieve a good compression ratio. A simplified parallel multiplier, which requires approximately half of the area of a standard parallel multiplier, has been designed in the AOU to decrease the hardware area. We also designed an asynchronous I/O path in the chip, which enables the chip to release wait states when it is transmitting data. A prototype of the chip has been designed and fabricated. The performance of the chip is 3 Mb/s on average at 25 MHz. The performance seems to be improved by the pipelining technique, although there are some hard problems (e.g., the variable execution cycle numbers needed in each coding iteration) that need to be solved. That is our next goal.

Acknowledgment

The authors wish to thank Chip Implementation Center (CIC), R.O.C., for fabricating the arithmetic-coding chip and the Computer and Communication Laboratory, Industry Technology Research Institute, R.O.C, for providing the 0.8-μm SPDM cell library. The authors would also like to thank the anonymous reviewers, whose comments were useful in improving this paper's manuscript.

References

[1] G. K. Wallace, "The JPEG still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, Apr. 1991.
[2] *MPEG-2 Video*, Draft Int. Standard ISO/IEC DIS 13818-2.
[3] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[4] T. C. Bell, I. H. Written, and J. G. Cleary, "Modeling for text compression," *ACM Computing Survey*, vol. 21, no. 4, pp. 557–591, 1989.
[5] J. G. Cleary and I. H. Witten, "A comparison of enumerative and adaptive codes," *IEEE Trans. Inform. Theory*, vol. IT-30, pp. 306–315, Mar. 1984.
[6] G. G. Langdon and J. Rissanen, "Compression of black–white image with arithmetic coding," *IEEE Trans. Commun.*, vol. COM-29, pp. 858–867, June 1981.
[7] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, June 1987.
[8] D. Chevion, E. D. Karnin, and E. Walach, "High efficiency, multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Apr. 1991, pp. 43–52.
[9] G. Feygin, P. G. Gulak, and P. Chow, "Minimizing error and VLSI complexity in the multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1993, pp. 118–127.
[10] L. Huynh, "Multiplication and division free adaptive arithmetic coding techniques for bi-level images," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1994, pp. 264–273.
[11] R. Arps, T. Truong, D. Lu, R. Pasco, and T. Friedman, "A multi-purpose VLSI chip for adaptive data compression of bilevel images," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 775–794, Nov. 1988.
[12] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 717–725, Nov. 1988.
[13] G. Feygin, P. G. Gulak, and P. Chow, "Architectural advances in the VLSI implementation of arithmetic coding for binary image compression," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1994, pp. 254–263.
[14] B. Fu and K. K. Parhi, "Two VLSI design advances in arithmetic coding," in *Proc. ISCAS*, Seattle, WA, Apr. 1995, pp. 1440–1443.
[15] J. M. Jou and S. R. Kuang, "Library-adaptively integrated high level synthesis systems," in *Proc. NSC—Part A: Phys. Sci. Eng.*, vol. 19, no. 3, R.O.C., May 1995, pp. 220–234.
[16] M. Hatamian and G. L. Cash, "A 70-MHz 8 × 8-bit parallel pipelined multiplier in 2.5-μm CMOS," *IEEE J. Solid-State Circuits*, vol. SC-21, pp. 505–513, Aug. 1986.
[17] S. Nakamura and K. Y. Chu, "A single chip parallel multiplier by MOS technology," *IEEE Trans. Comput.*, vol. 37, pp. 274–282, Mar. 1988.
[18] *0.8 μm SPDM Technology Manual*, Comput. Commun. Lab., Industry Technol. Res. Inst., Taiwan, R.O.C., 1993.
[19] D. E. Knuth, "Dynamic Huffman coding," *J. Algorithms*, vol. 6, pp. 163–180, 1985.
[20] J. Jiang, "Novel design of arithmetic coding for data compression," *Proc. Inst. Elect. Eng.*, vol. 142, no. 6, pp. 419–424, Nov. 1995.
[21] W. Pennebaker and J. Mitchell, *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.
[22] *Design Framework II User Guide*. Cadence Design Syst., 1992.
[23] P. G. Howard and J. S. Vitter, "Arithmetic coding for data compression," *Proc. IEEE*, vol. 82, pp. 857–865, June 1994.
[24] L. Y. Liu, J. F. Wang, and J. Y. Lee, "CAM-based VLSI architectures for dynamic Huffman coding," *IEEE Trans. Consumer Electron.*, vol. 40, pp. 282–289, Aug. 1994.
[25] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol. 24, pp. 530–536, Sept. 1978.

Shiann-Rong Kuang received the B.S. degree in electrical engineering from National Center University, Taiwan, R.O.C., in 1990, the M.S. degree in electrical engineering from National Cheng Kuang University, Tainan, Taiwan, R.O.C., in 1992, and is currently working toward the Ph.D. degree in electrical engineering.

His research interests include high-level synthesis and VLSI chip design.

Jer-Min Jou received the Ph.D. degree in electrical engineering and computer science from National Cheng Kung University, Tainan, Taiwan, R.O.C., in 1987.

Since 1989, he has been an Associate Professor in the Department of Electrical Engineering and Computer Science, National Cheng Kung University. His current research interests include application-specific integrated circuit (ASIC) design/synthesis, hardware-software codesign, VLSI CAD, and asynchronous circuit design.

Dr. Jou was the recipient of a Distinguished Paper Citation at the 1987 IEEE ICCAD Conference, Santa Clara, CA.

Yuh-Lin Chen received the B.S. degree in electronic engineering from National Taiwan Institute of Technology, Taiwan, R.O.C., in 1993, and the M.S. degree in electrical engineering from National Cheng Kuang University, Tainan, Taiwan, R.O.C., in 1995.

He is currently an Engineer at Silicon Touch Technology Inc., Taiwan, R.O.C., where he works on development and testing of fan IC's, phase IC's, and hardware monitor IC's.