A multi-purpose VLSI chip for adaptive data compression of bilevel images

by R. B. Arps
T. K. Truong
D. J. Lu
R. C. Pasco
T. D. Friedman

A VLSI chip for data compression has been implemented based on a general-purpose adaptive binary arithmetic coding (ABAC) architecture. This architecture permits the reuse of *adapter* and arithmetic *coder* logic in a universal way, which together with application-specific *model* logic can create a variety of powerful compression systems. The specific version of the *adapter/coder* used herein is the "Q-Coder," described in various companion papers. The hardware implementation is in a single HCMOS chip, to maximize speed and minimize cost. The primary purpose of the chip is to provide superior data compression performance for bilevel image data by using conditional binary source models together with adaptive arithmetic coding. The coding scheme implemented is called the Adaptive Bilevel Image Compression (ABIC) algorithm. On business documents, it consistently outperforms such nonadaptive algorithms as the CCITT Group 4 (T.6) Standard and comes into its own when adapting to documents scanned at different resolutions or which include significantly different data such as digital halftones. The multi-purpose nature of the chip allows access to internal partition combinations such as the "Q" adapter/coder, which in combination with external logic can be used to realize hardware for other compression applications. On-chip memory limitations can also be overcome by the addition of external memory in special cases. Other options include the uploading and downloading of adaptive statistics and choices to encode or decode, with or without adaptation of these statistics.

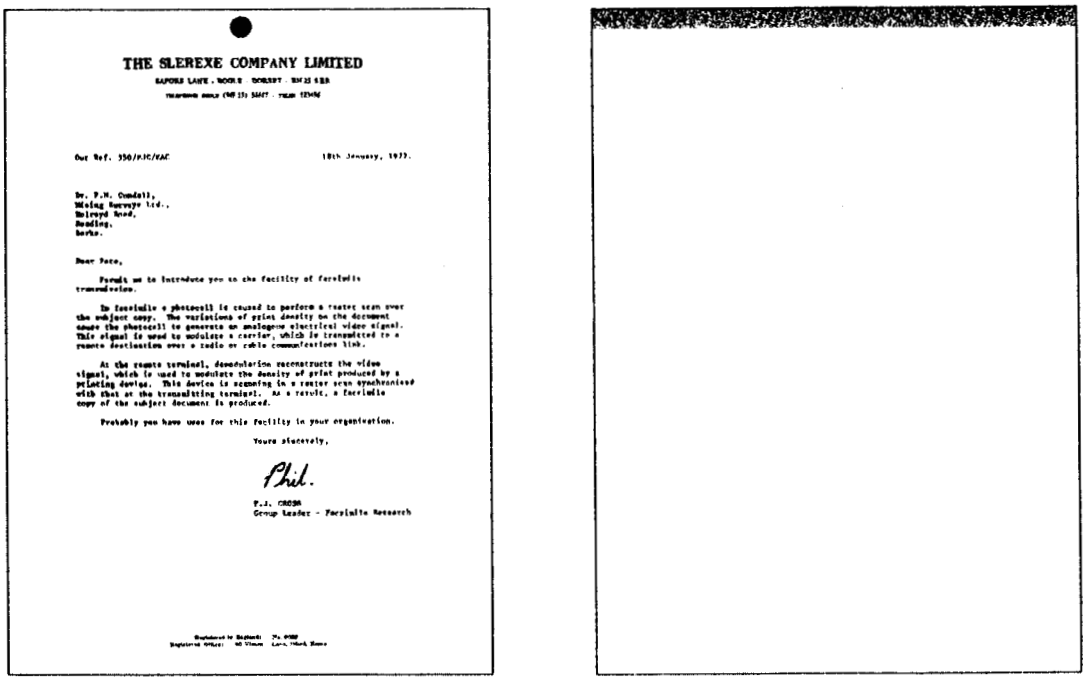
Introduction

• *Data compression algorithms*

Data compression algorithms are used to transform digital data into equivalent, but usually smaller, "compressed" representations. They are used in such applications as digital facsimile or image processing systems, to decrease the average amount of data to be transmitted or stored. Such algorithms are designed using the principles of information theory, which are introduced in a number of textbooks such as those written by Abramson [1] or Ash [2]. For *bilevel image* data compression, Arps has written a tutorial and detailed summary of the art in Chapter 7 of *Image Transmission Techniques* [3], as well as detailed bibliographies [4, 5].

Figure 1 illustrates these basic concepts for our Adaptive Bilevel Image Compression (ABIC) algorithm, designed for images digitized with only two levels of amplitude[1]. A

[1] Such images are binary in amplitude, with "white" or "black" picture elements (pels) typically represented by the bits 0 or 1, respectively.



(a) (b)

Figure 1

Compression of business document by binary-image compression algorithm: (a) CCITT-1 original image (513,216 bytes); (b) ABIC-encoded image (14,969 bytes).

Table 1 Compression ratio comparison of ABIC algorithm vs. G4 (T.6) algorithm.

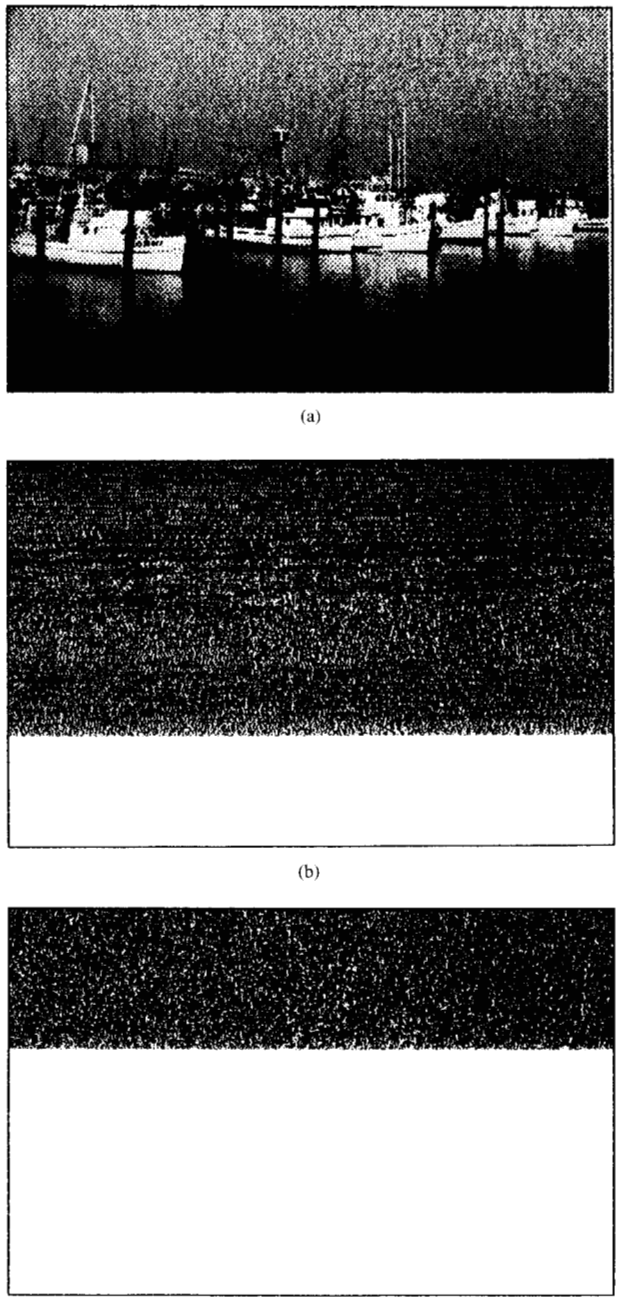
Test data		*G4 (T.6) algorithm*		*ABIC algorithm*		*Result*
File name	*Original size* (bytes)	*Compressed size* (bytes)	*Compression ratio*	*Compressed size* (bytes)	*Compression ratio*	*Change* (%)
Conventional documents						
CCITT1	513,216	18,103	28.35	14,969	34.29	+20.9
CCITT2	513,216	10,803	47.51	8,946	57.37	+20.8
CCITT3	513,216	28,706	17.88	23,466	21.87	+22.3
CCITT4	513,216	69,275	7.41	55,852	9.19	+24.0
CCITT5	513,216	32,222	15.93	26,986	19.02	+19.4
CCITT6	513,216	16,651	30.82	14,032	36.57	+18.7
CCITT7	513,216	69,282	7.41	58,529	8.77	+18.4
CCITT8	513,216	19,099	26.87	15,596	32.91	+22.5
Digital halftone documents						
BOAT2	46,848	33,418	1.40	17,187	2.73	+94.4
PANDAQ	46,800	23,685	1.98	10,948	4.27	+116.3
PANDA	46,800	24,043	1.95	11,163	4.19	+115.4
ERD	46,800	90,699	0.52	23,027	2.03	+293.9
DIT	46,800	70,334	0.67	15,766	2.97	+346.1
SUPC	46,800	49,485	0.95	16,653	2.81	+197.2

business letter, Figure 1(a), is shown along with a pseudo-image, Figure 1(b), displaying the bits resulting from its compression. The latter image was created by assembling compressed bits into the top of an area the same size as the original image, padding any unused area with white pels. The bit savings resulting from compression can be visualized by comparing the area (bits) of the original image with the "salt-and-pepper" area (bits) of compressed data in the pseudo-image. We use the term "compression ratio" to express the bit savings obtained, illustrated here as the ratio between the original and compressed image areas. In the example of Figure 1, the ABIC algorithm reduced the original 513K-byte source image to 15K bytes, yielding a compression ratio of 34.3.

The illustrated algorithm is "lossless," in that the original image can be reconstructed exactly by decompressing the data illustrated in its pseudo-image. "Lossy" algorithms are characterized by compression that is not reversible, wherein any loss is usually designed to be acceptable to the intended final user of the data. Compression for bilevel image data is typically lossless, since it is already preceded by the lossy step of forcing the original image pixels to two amplitudes. The latter step precludes much further lossy image processing without becoming unacceptable to the user and is similar to the *thresholding* which occurs in a typical copying machine.

Figure 2 illustrates the compression performance for a more difficult type of bilevel data—digital halftones. As can be seen in the pseudo-image of Figure 2(c), the ABIC algorithm now attains a compression ratio of only 2.7. In general, the compression process is highly data-dependent, and for bilevel data the most difficult images are those in which the percentages of black and white pels are about equal. The difficulty in compressing halftones is even more pronounced in Figure 2(b), which illustrates that a compression ratio of only 1.4 is achieved with the international standard Group 4 (T.6) algorithm [6] from the Consultative Committee for International Telephony and Telegraphy (CCITT). This dramatic difference in compression performance is due primarily to the fact that the ABIC algorithm is adaptive and the CCITT algorithm is nonadaptive ("static"). Notice also that the pseudo-image no longer appears to be completely random. This is a practical clue that there is potential for further compression than this algorithm has fully realized.

A more detailed comparison of these two algorithms is summarized in **Table 1**. Two categories of data are represented—business documents and digital halftones. The business document data are from the commonly used CCITT test set of documents [6]. The digital halftone data consist of the high-detail "BOAT2" image of Figure 2, plus lower-detail images for a set of different halftone algorithms applied to the same picture of a face [7]. Compression performance results are reported both as actual compressed data sizes and as the compression ratios achieved for each test document. A percent-change column has been included to facilitate comparison of the algorithms.



(a)

(b)

(c)

Figure 2

Compression of halftone image by static and adaptive algorithms: (a) "BOAT2" original image (46,848 bytes); (b) encoded by G4 MMR static algorithm (33,418 bytes); (c) encoded by ABIC adaptive algorithm (17,187 bytes).

The ABIC algorithm performs uniformly better than the CCITT algorithm for all of the data. It performs

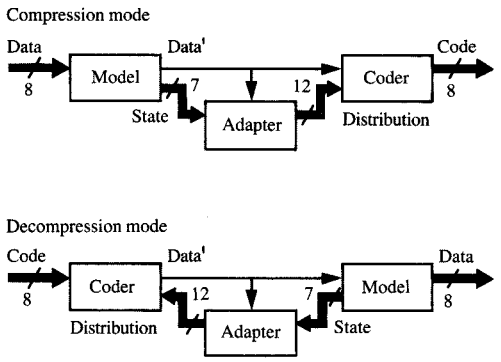


Figure 3

Block diagram for adaptive data compression.

approximately 20 percent better on the business documents, *for which the CCITT algorithm has been explicitly optimized.* Note that the results for the ABIC algorithm have been obtained without prior optimization on any test data set at all. Its adaptive process is initialized with random information. For the halftone data, the ABIC algorithm performs from 2.0 to 4.5 times better than the static CCITT algorithm. In some cases, the latter algorithm does not compress at all, but rather *expands* the data.

The ABIC algorithm is explained in detail in later sections, following discussion of the architectural concepts used to obtain adaptive data compression and implement it compactly.

• *VLSI implementations*

For systems that process bilevel images, data compression is imperative because of the large size of uncompressed image records and the large compression ratios that can be achieved. However, these systems also need very-high-speed compression/decompression (or "comdec") hardware implementations, in order to achieve split-second capture and retrieval of images. A comdec built with dedicated hardware offers the maximum in attainable speed, especially when built on one chip where interconnection delays can be minimized.

IBM built one of the first binary-image comdec chips in the early 1970s [8], using the LSI technology of that period. Early in the 1980s, the CCITT Standard Group 3 and 4 binary-image compression algorithms [6] were put into one chip by AMD [9]. Both of these chips implemented nonadaptive algorithms that were based on variations of run-length coding [3]. Building adaptive forms of such algorithms represented a formidable increase in complexity, tending to exceed the capacity of single-chip architectures.

With the development of "arithmetic coding" for data compression [10–17] and the increasing complexity possible with VLSI technology, the environment has changed dramatically. We are now able not only to integrate adaptive compression algorithms into single VLSI chips but, with arithmetic coding, also to create superior algorithms that were heretofore impractical to realize.

Using a general-purpose architecture for compression algorithms based on our adaptive binary arithmetic coding (ABAC) technology, we have been able not only to design such a chip for adaptive binary-image compression but also to make it a multi-purpose device. It offers separately accessible combinations of partitions, for use as building blocks in building hardware comdec prototypes for other applications. It also offers the ability to initialize or dump the statistics for adaptation, to select among adaptive and nonadaptive modes and other features, and to make the normal choices between encode and decode modes of operation. This paper reports the algorithmic and hardware design of two generations of such a chip, as well as the successful completion and fabrication of the second generation of our design by a "silicon foundry."

Overview of key concepts

• *Adaptive data compression*

The architecture we use for adaptive data compression or decompression uses three basic components: a *model*, an *adapter*, and a *coder*, as illustrated in **Figure 3**. The model and coder components can be defined in classical terms such as might be used to describe nonadaptive algorithms. For instance, the model for the CCITT Group 3 algorithm [6] assembles "outcomes" which are contiguous "runs" of black or white pels, for subsequent coding using corresponding code words of some appropriate length. Such code words are designed using a fundamental relationship from information theory,

$$l_i = -\log_2(p_i),$$

which relates the probability p_i of each possible output i from the model to an ideal code-word length l_i. Code words approximating this ideal length should be used to encode each outcome during compression or decode it during decompression.

The challenge, in making this methodology adaptive, is to find a practical method by which real-time changes in the probability distribution for model outcomes can be turned into appropriate changes in the lengths of code words in the coder. In Figure 3 the adapter component performs the function of tracking model outcomes, in order to continually estimate their probability distribution. It also passes probability-estimate information, for each model outcome to

be encoded or decoded, to an instantaneously reconfigurable coder. Note that the model/adapter combination must function identically during both of these modes. Additionally, the compression-mode adapter must use only that model information which the decompressor model at that point can already have reconstructed (i.e., the adapter must be causal).

A general-purpose reconfigurable coder should be able to adjust to all possible distributions over *codebooks* of a given size and still generate efficient code words dynamically. The difficult problem of designing such a coder is solved by breaking it into the following two steps:

Step 1 A reconfigurable coder for all possible *binary* (two-code-word) codebooks is used [12].

Such binary coders, based on arithmetic compression coding, are only described behaviorally here; the next section contains a more complete description of the specific coder we used.

Step 2 Any nonbinary model using an arbitrary but fixed codebook size is decomposed into a repeating sequence of *conditional binary* models [13].

In effect, this is a parallel-to-serial conversion of the bits in the outcomes of any desired nonbinary model. The resulting conditional binary models are then serially encoded by multiplexing their outcomes to the reconfigurable binary arithmetic coder.

The key concept in Step 2 comes from the *chain rule* for *entropies* in information theory [2]. Simply stated, one can serialize the outcomes of a nonbinary model into a sequence of outcomes from conditional binary models without any change in the potential for compression (entropy). Mathematically, this is formalized as

$$H(abc\cdots h) = H(a) + H(b \mid a)\cdots + H(h \mid abc\cdots g),$$

where $H(\cdot)$ is the classical entropy function. For this example, the set {*abcdefgh*} might represent the combination of bits *abcdefgh*, enumerating any outcome from a model representing the coding of arbitrary bytes of digital data.

• *Compressing binary models*

To accomplish Step 1 in our adaptive compression architecture, one must be able to compress *binary* models. The ideal lengths of the two code words for a binary model include one which is less than one bit long (if there is any potential for compression). When methods such as Huffman coding [1] are used to generate code words for these ideal lengths, the code words that result are again one bit in length, resulting in no compression.

The classical solution to this problem is to group together N outcomes from a binary model, estimate probabilities for the 2^N possibilities that arise, and treat them as if they were outcomes from an Nth *extension* of the original model.

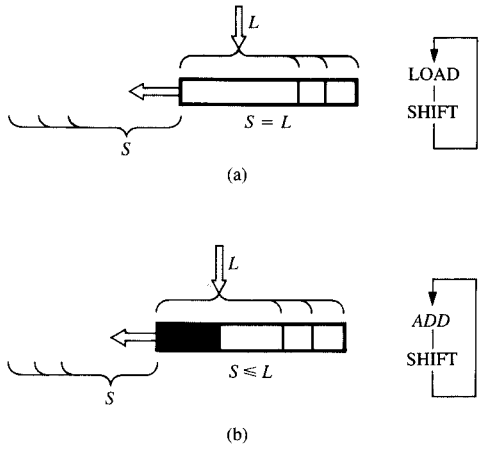


Figure 4

(a) Concatenation coding; (b) arithmetic coding.

Application of Huffman coding techniques to such a nonbinary model can then yield variable-length code words and the potential for compression.

Note that grouping of binary outcomes into an extension precludes the use of a reconfigurable *binary* coder and conditional binary models—as required by our general solution for adaptive compression coding. In contrast to this, *arithmetic* compression coding can encode binary models directly, without having to resort to grouping outcomes into an Nth extension! It accomplishes this by its unique ability to encode even fractional bits of information.

To better understand arithmetic coding, it is useful to compare its behavior with that of the classical approach, which we shall call "concatenation" coding. That is, *block* codes [1] such as Huffman coding typically assemble variable-length code words into a code string by using a register, as is illustrated in **Figure 4(a)**. This process consists of alternating between the *LOAD* of some number L of code-word bits into the register and the *SHIFT* of a corresponding number of bits, S, out of the register into the code string (i.e., $S = L$).

In contrast to this, arithmetic coding, as shown in **Figure 4(b)**, permits its code-string assembly register to shift out *less* than the size of the code word that was last loaded (i.e., $S \leq L$). There may be leftover data in the register when the next code word is to be assembled, so that the previous LOAD operation must be redefined. Its operation becomes binary arithmetic (i.e., ADD), hence the name *arithmetic* coding.

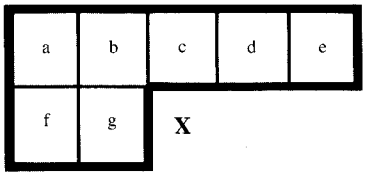


Figure 5

7-bit template.

The point, specifically, is that with arithmetic coding one can encode code words that ideally require fractional bits of length and, in particular, the fractional ideal code-word lengths required to be able to compress a simple binary model. Of course, it must be possible for a decompressor to undo the code from such a nonblock code. The theory of arithmetic coding specifies the extent to which its code words may be overlapped when assembling them into a code string and still retain "unique decodability" [1].

This powerful result includes the ability to instantaneously *change* the probability distributions used in creating code words, even though the code words are mixed into a nonblock code string. Decompression is possible so long as the binary decoder receives a sequence of distributions identical to that used during encoding. It is thereby possible to adapt after each new data bit, to readily encode *conditional* binary model data [16], and to realize the desired general-purpose coder design strategy.

ABIC algorithm

• *ABIC model*

The potential for compressing bilevel data with *conditional binary* models was investigated theoretically [18], in entropy studies, for some time before it became practical to actually realize implementations. One such model is illustrated in **Figure 5**, wherein each binary pel is encoded based on the conditional state or *context* of a set of 7 neighboring pels. The classical challenge was how to encode such a conditional binary source in the face of a conflicting need to take Nth extensions.

Preuss [19] developed the first algorithm that accomplished this, although it still had severe practical limitations. He created multiple Nth extensions for the separate states of a conditional binary Markov model, by demultiplexing the pels in a raster image into parallel state-dependent strings [3]. Each of these strings needed a separate parallel coder; the illustrated 7-pel model, for example, would have required *128* of them.

Langdon and Rissanen [16] elegantly simplified this problem by the application of ABAC technology. They first stored the conditional probability distributions for the 128 states illustrated. When the pels in a raster image were then applied to a reconfigurable binary arithmetic coder, they simultaneously applied probabilities from the appropriate distribution. This distribution address was determined as the state (or context) of the 7-pel model. They took maximal advantage of the instantaneously reconfigurable coder as well, by adapting the probability distributions. Note that although this approach uses additional storage for the 128 distributions, it requires the use of only *one* arithmetic coder.

The technology used for the reconfigurable coder and its accompanying adapter evolved through a series of generations. Our chip design first used the "skew" form of coder [12] and "Monte Carlo" [20] form of adapter. We subsequently updated the ABIC algorithm and redesigned the chip to use an improved adapter/coder called the "Q-Coder" [21–24]. Developed jointly with colleagues at the IBM T. J. Watson Research Center, it permits the implementation of fast software as well as hardware. It has also improved the compression ratios achieved on the CCITT documents. The next sections describe our hardware Q-Coder implementation, after introducing more detailed arithmetic coding concepts.

• *ABIC coder*

Fundamentals: Intervals on the number line

Arithmetic coding is a coding technique used for *lossless data compression*, that is, an invertible mapping between any data file and a more compact representation of the same information. From an idea originally proposed by Peter Elias [1], arithmetic coding maps mutually exclusive outcomes of a probabilistic "event" into nonoverlapping intervals on a real number line. A particular outcome may be specified by giving the numerical value of any point in the corresponding interval. For data compression, each symbol from the source file is an event, the values the symbol may have are the possible outcomes of the event, and the widths of the intervals are chosen to be approximately proportional to the probabilities of the values. The compression occurs when more likely events correspond to larger intervals, because it takes fewer bits to specify some point in a large interval than in a small one.[2]

Iteration and strings

The benefit of arithmetic coding comes from encoding not just a single event or symbol but a sequence of events (string

[2] Of course, some small intervals could be specified with just a few bits, but these cases are so unusual that most arithmetic coding algorithms do not take advantage of them. The number of bits necessary to specify an arbitrary interval grows as the negative of the logarithm of the width of the interval.

of symbols). This is done iteratively as follows. Start with a given interval on the real number line. Any finite interval will do, but the ABIC uses the "unit interval" (numbers between zero and one). For each symbol, establish a correspondence between its possible values and nonoverlapping subintervals. The actual value of the symbol selects one of these subintervals. Further divide this new interval by subsequent symbols in the same manner. When the end of the source is reached, transmit a binary fraction to specify some point in the final subinterval. We will call the point so chosen the *code point.*

Decoding

The decoder starts with the same initial interval as the encoder, and establishes the same correspondence between possible values of each symbol and nonoverlapping subintervals. At each step it examines the binary fraction from the encoder to determine which subdivision was taken, and outputs the appropriate symbol. As in the encoder, this new interval is further subdivided until all of the original decisions have been reconstructed.

This technique uniquely reconstructs the original sequence regardless of how the subintervals are chosen, provided they are nonoverlapping and that both the encoder and decoder subdivide the intervals in the same way. When the goal is data compression, though, a wise use of code space must be made. It turns out that, on the average, the fewest bits are necessary to specify a point in the final interval if at each iteration the width of each subinterval is made proportional to the relative probability of its corresponding symbol.

Conditioning

The probability distribution for a given symbol, or even its alphabet, need not be fixed but may be conditioned on preceding symbols in the file. This is possible because at the time a given interval is to be subdivided, both the encoder and decoder know the value of all previous symbols, and hence can make identical assignments to possible subintervals. This fact allows arithmetic coding great flexibility in encoding Markov strings and in adapting to fluctuating source statistics. For example, in **Figure 6**, the alphabet and/or probability distribution for the second symbol might have been different if the first symbol had assumed value "A" or "C" instead of "B." When the first symbol was "B," its possible successors were "D," "E," and "F," but if the value of the first symbol had been "A," its possible successors might have been "TRUE" and "FALSE"; similarly, the possible successors of "C" might have been "RED," "YELLOW," and "GREEN." The system works as long as the encoder and decoder have prior agreement about the set of successors (and their probabilities) for each history.

Relationship of coder to model and adapter

Refining the definition of the "Model-Adapter-Coder" architecture described earlier, the *model* maps the incoming data into a series of symbols, or probabilistic events, to be encoded. Accompanying each symbol the model also delivers a summary of the previously processed data, expressed as a single number called a *context* [16]. Given this context, the *adapter* provides an estimate of the probability distribution of the symbol, based on the values which were actually seen accompanying previous occurrences of the same context. Next, the *coder* subdivides the interval on the number line according to this probability estimate and the actual symbol value. Finally, the adapter revises its state based on the symbol value just encoded.

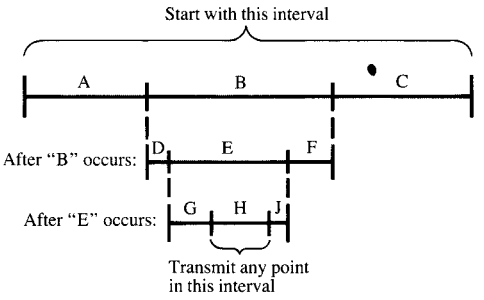


Figure 6

Encoding the string "BEH."

Finite-precision arithmetic

A direct implementation of the technique for subdividing intervals on the number line would require arbitrarily long word lengths for the calculations involved [25]. To avoid this, we use a kind of floating-point arithmetic which takes advantage of the fact that (usually) as the sequence progresses, the interval being considered becomes smaller in such a way that all points in it have the same leading (more significant) bits. Thus, it is *usually* possible to transmit these bits long before the subdivision process is complete, and focus attention on the less significant bits [11]. **Figure 7** illustrates this. The exception, where the interval persistently spans a roll-over of a more significant bit, is considered later as a special case.

Registers and variables

Because it takes two numbers to define an interval on the number line (either as two endpoints or as one endpoint and a width), every arithmetic coder needs two registers to keep track of the current interval [17]. In the Q-Coder used in the ABIC, they are called the A and C registers. For both encode and decode, the A register represents the width of the current interval. For encode, the C register represents the lower

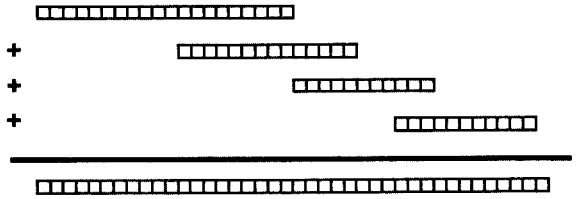


Figure 7

Finite-precision arithmetic.

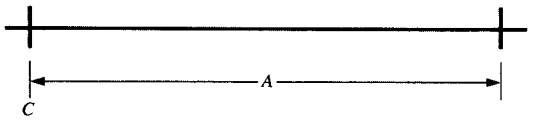


Figure 8

A and and C registers during encode.

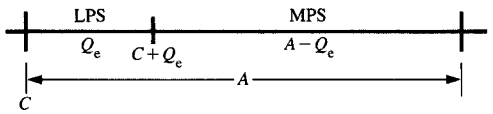


Figure 9

Partition by Q.

endpoint of the current interval, as shown in **Figure 8**. For decode, the C register represents the position of the code point relative to the lower endpoint of the current interval (i.e., the value of the code point minus the lower endpoint of the current interval).

Both of these registers have twelve bits to the right of an imaginary radix point. The A register has one bit left of the radix point, and the C register has four. The bits left of the radix point in the C register are called "spacer" bits. The internal function of "spacer" bits is discussed later under "Long carries and spacer and stuff bits."

Model initialization

The ABIC model memory is initialized such that the previous line (above the top edge of the image being encoded) is assumed to be all 0 (white).

Coder initialization

For both encoding and decoding, the A register is initialized to 1.0. For encoding, the C register is initialized to 0.0. Together, these values designate the initial interval to be the interval between zero and one. By convention, the first byte transmitted from the ABIC consists of the eight bits from the C register just right of its implied radix point (i.e., none of the four "spacer" bits mentioned under "Registers and Variables" are transmitted). In other words, the bits of the first byte have weights one-half, one-fourth, one-eighth, etc., beginning with the most significant bit (MSB).

For decoding, the four "spacer" bits to the left of the radix point of the C register are set to zero, and the bits to the right of the radix point are initialized to the first twelve bits of the code string.

Binary arithmetic coder

The coder portion of the ABIC is one of a class of arithmetic coders which consider only binary symbols. That is, each iteration considers an event with just two possible outcomes, and divides the interval into just two subintervals. The two possible values for each binary symbol (bit) to be encoded are called "more probable symbol" (MPS) and "less probable symbol" (LPS). By convention, the ABIC assigns the LPS at each iteration to the lower subdivision of the current interval (adjacent to C), and the MPS to the upper (greater than C). This is shown in **Figure 9**.

Q-Coder

The ABIC algorithm adapter and coder together are called a "Q-Coder." Each time an interval on the number line is to be divided, the width of the subinterval corresponding to an LPS is chosen from a fixed table called a Q-table. The remainder of the original interval corresponds to an MPS. The choice of which table entry to use is made by a finite-state machine in the adapter. Its state transitions are triggered by feedback from the coder, as discussed in the subsection on the ABIC adapter.

Ideally (for best compression), if A is the width of the current interval in the A register, and q is the probability that the next symbol is an LPS, then the width of the subinterval corresponding to an LPS should be approximately Aq and the width of the subinterval corresponding to an MPS should be approximately $A(1-q)$. The Q-table approximates Aq with value Q_e to avoid a multiplication, as is discussed in the subsection on the Q

approximation. Figure 9 shows how the Q-value partitions the interval into LPS and MPS subintervals. The encoding caculation for subdividing an interval is

```
if MPS then begin
  C := C + Qe;   (* move lower end *)
  A := A – Qe;   (* narrow width *)
end
else
  A := Qe;       (* narrow width *)
```

The corresponding decoding operation is

```
if C >= Qe then decode an MPS
else decode an LPS;
if MPS then begin
  C := C – Qe;   (* move lower end *)
  A := A – Qe;   (* narrow width *)
end
else
  A := Qe;       (* narrow width *)
```

Renormalization

As mentioned above, the ABIC avoids the need for infinite-precision registers by using a form of floating-point arithmetic. After each iteration, the A register, which defines the width of the subinterval chosen, is "renormalized," or shifted left such that its leading bit (the bit left of the radix point) is 1. Renormalization essentially "zooms in" or magnifies the current interval by a power of two, so that it fills at least half the capacity of the A register. The C register, which defines the lower endpoint of the interval, is also shifted left by the same number of places, so that bits in the C register always have the same algebraic weight as the aligned bits in the A register. During the encoding operation, bits shifted out of the left (most significant) end of the C register are buffered and, eventually, transmitted as the code string; we show later how this works. During the decoding operation, new bits from the code string fill the vacated least significant bit positions of the C register. Now the Q-Coder is ready to process the next symbol.

The Q approximation

For ideal compression performance, $Q_e = Aq$. However, the optimum is rather broad, and an approximate value for Q_e which depends only on q can avoid this multiplication and still provide reasonable performance. The Q-table need not be very large; in the ABIC it has just 30 entries for probabilities in the range from 1/2 down to 2^{-12}. Because of the renormalizations, A is nearly constant (within a factor of two). In other words, since the single bit left of the radix point in the A register is always 1, we note that

$$1 \leq A < 2.$$

Knowing this, we can approximate $Q_e = Aq$ as

$$Q_e \cong kq$$

for some constant k. Previous work [22] has assigned $k = 4/3$. To understand why this is reasonable, the section "Optimally filling the Q-table" finds for each qP the value of Q_e which minimizes the code space taken, and demonstrates that the ratio Q_e/q is only weakly dependent on q. Along with the definition of LPS (that $q < 0.5$), this implies $Q_e < 1$ and hence that Q-values need no places left of the radix point. In ABIC, the Q-values are expressed to 12 binary places to the right of the radix point.

- *ABIC adapter*

For the Q-Coder, it is the job of the *adapter* to provide the appropriate value of Q_e for each symbol (bit) to be encoded. The adapter is a finite-state machine multiplexed to maintain a separate state record for each context. The multiplexing is implemented by a random-access memory (RAM) whose address or index is the context and whose data are the state records for each context. The state record consists of

- A single bit, which records whether a 0 or a 1 is more likely to be emitted next by the source.
- A 5-bit number called the Q-index, which selects from a fixed table one of 30 possible Q-values to be provided to the coder.

As discussed above, these Q-values purport to be approximately 4/3 times the probability that the next bit will be an LPS. The complete table for the ABIC is shown in **Figure 10**.

The state record for each context can be thought of as marking a position on one of two ladders, each having rungs labeled with the various Q-values. There is one ladder for MPS = 0 and another for MPS = 1. When a given context occurs, after its state record provides an MPS-value and Q-value for the coder, a new state may be entered for that context (i.e., the state record may be modified). If an MPS occurred, the state may climb a step further up the ladder it is on, selecting for next time an even smaller probability that the next symbol will be an LPS, unless it is already on the top rung of the ladder. If an LPS occurred, the state descends a little on its ladder, increasing the probability that the next symbol will be an LPS, unless it is already at the bottom rung, in which case it steps over to the bottom rung of the other ladder, reversing the expected symbol. Figure 10 shows the state assignments and transition rules for all 60 states.

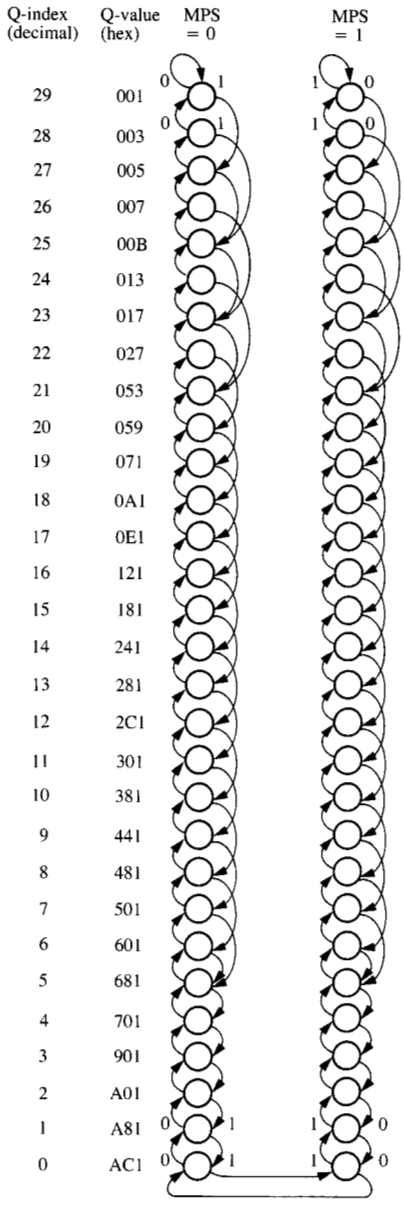


Figure 10

State-transition diagram for Adapter.

Adapter initialization

The adapter memory is initialized such that for each context the most probable symbol (MPS) is a 0 (white pixel) with Q-index 0 (the bottom rung of the ladder). Although the Q-value on this bottom rung corresponds approximately to probability 1/2, it is necessary that both the decoder and encoder agree as to which is the MPS, so that the sense of the first symbol is correct.

Renormalization-driven adaptation

We observe above that "the state *may* climb a step further up the ladder." Seeing an MPS is not surprising. It confirms our prediction and tends to reduce our expectation of seeing an LPS, but not so drastically that we would want to climb to a new rung on the ladder every time an MPS occurred. One previous adapter design [14] counted MPS occurrences and took a step after a large number had been seen, but this required a large-modulus counter for highly skewed cases (cases where the LPS was expected to be extremely rare). Another design [20], called *Monte Carlo*, flipped a skewed coin to decide whether it was time to take a step, but this required a coin flip at each iteration, when a step was only occasionally required. With a very clever insight [22], Pennebaker and Mitchell showed that the right amount of adaptation is provided when it is triggered by a renormalization of the A register: If the estimated probability of an LPS is small, an MPS is not very surprising. In this case, the Q-value subtracted from A when an MPS occurs is small and only occasionally results in a renormalization. Thus an MPS only occasionally causes a step up the ladder. Conversely, an LPS is always surprising. Since an LPS replaces A with some Q_e, and every Q_e is less than 1, each LPS automatically triggers a renormalization. Thus, an LPS always causes the adapter to descend the ladder. The effect is optimized by including on each rung of the Q-value ladders an additional entry, which says *how far* down the ladder to descend when an LPS is encoded. A very surprising LPS causes the adapter to take several steps down the ladder. This is shown in Figure 10.

In the ABIC, the state record for each context is initialized to expected MPS = 0 with Q-index = 0. A renormalization caused by encoding an MPS increments the Q-index for the current context, unless it is already at its maximum value. Encoding an LPS replaces the Q-index for the current context with the new Q-index from the table, unless it is at its minimum value, in which case the MPS value is reversed.

Long carries and spacer and stuff bits

The lower endpoint of the current number-line interval is represented by a long fixed-point binary fraction, whose less significant bits are still in the C register and whose more significant bits were shifted out of its left end during the renormalization process. The code point, or final point later chosen to designate the innermost subinterval, usually has the same more-significant bits. Thus, the bits shifted from the C register usually constitute the desired code string. The exception to this is when a subsequent MPS adds to C a Q-value large enough to cause a carry to propagate into the bits which have already left the C register. When this

happens, the carry must be resolved so that the bits sent to the decoder accurately represent the code point. This is accomplished by using a combination of *spacer* bits [26] and *stuff* bits [27].

Spacer bits are the bits to the left of the radix point in the C register. They provide a pipeline through which the bits shifted left must pass before leaving the register and being transmitted. Carries propagating from the active portion of the C register into these spacer bits are often resolved there. Only if the spacer bits are all 1 does a carry propagate through them and out of the C register. No matter how many spacer bits are provided, this will happen with nonzero probability. Inserting stuff bits provides a solution.

In viewing the code string as a binary fraction representing a point on the number line, each of its bits normally has half the algebraic weight of its predecessor. A stuff bit is an exception to this rule, in that it is an extra bit position with the same algebraic weight as its predecessor. The value of the fraction is the weighted sum of its bits, where the weights are negative powers of two. In computing the effective value of the fraction when a stuff bit is inserted, the two bits having the same weight may be added together.

This idea can prevent a carry from propagating into that part of the code string which has already been transmitted: Bits shifted out of the C register are buffered for a time before transmission. When the buffer contains a run (continuous sequence) of 1s, presenting a danger that a carry propagating into them would propagate through and out into the code string, a 0 stuff bit is inserted just below the run. Should a carry arrive from the C register, it sets the stuff bit to 1. This achieves the correct algebraic result but without propagating a carry through the run above it. Since the decoder can recognize the same run of 1s, it knows which bit is the stuff bit, and can assign it the correct algebraic weight.

In the ABIC, the stuff scheme is tied into the buffering that groups the bits shifting one at a time from the C register into eight-bit bytes. Each time a code byte with hexadecimal value 'FF' is transmitted, a stuff bit is inserted into the code string at the high-order bit of the following byte. Subsequent bits from shifting the C register are held back to make room. The stuff bit (which has the same weight as the low-order bit in the 'FF' byte just transmitted) catches any carry which would otherwise be added into the 'FF' and propagate out into the part of the code string which has already been transmitted.

The use of stuff bits implies that different code strings could have the same arithmetic value, depending on whether a carry settles naturally or propagates until it is "caught" in a stuff bit. By increasing the number of shift-register bits (called "spacer" bits) between the active C register and the serial-to-parallel conversion, the probability is increased that a carry settles in the registers before it gets to the stuff position. In order to replicate exactly the code string the ABIC produces, it is necessary to understand exactly what buffering and stuff bits are required. The following paragraphs detail what must be done.

Bits which are shifted out of the left end of the C register during renormalization are not transmitted immediately; instead, they are queued in an extension of the C register so that carries propagated out when a Q-value is added into C may be resolved. When this queue gets sufficiently long, its high-order bits are moved to a code-byte buffer and later transmitted into the code string.

The fixed-point C register has twelve fractional bits to the right of the radix point. To the left of the radix point are up to twelve additional "extension" bits, the exact number depending on how many renormalization shifts have occurred since the last transmission of a code byte. Initially there are no extension bits. Each renormalization shifts C left and increases by one the number of extension bits. There is a means of counting the number of extension bits which have been shifted from C.

When a carry propagates above the radix point of C, it propagates into the extension bits. This does not increase their number, but it does increment their value. If all the extension bits happen to have value 1, the carry propagates through all of them (and out the other side). A "carry flag" records when this has happened. This flag has the same arithmetic weight as the LSB of the code-byte buffer (double the weight of the leftmost extension bit).

The first time the count of extension bits reaches twelve, the highest eight of them are moved into the code-byte buffer, reducing the number of extension bits to four. The bit positions allocated for these four extension bits are also called "spacer" bits. The code byte in the buffer is not transmitted yet; it is held in the buffer for possible incrementation.

Each time thereafter that the count of extension bits again reaches twelve, it is time to transmit a code byte. If the code byte in the buffer is not X'FF', the code byte is incremented and the carry flag is cleared. Next, the code byte is transmitted. The value actually transmitted determines whether the following byte is to include a stuff bit.

If the value so transmitted was other than X'FF', then no stuff bit is called for. In this case, the code-byte buffer receives the eight highest extension bits, reducing the number of extension bits to four. But if the byte just transmitted was X'FF', then a stuff bit is called for. In this case, the code-byte buffer receives a stuff bit and seven data bits; its MSB receives the carry flag, and its low seven bits receive the high seven extension bits, leaving five extension bits behind. The stuff bit will be a 1 if and only if there was a carry from the bits below (following) it which could not propagate into the byte above because it was already X'FF'. Conversely, it should be 0 if either there was a carry from the bits below which did propagate into the byte above, turning it from X'FE' into X'FF', or the byte above was already X'FF' and there was no carry.

Unstuffing carries

When an X′FF′ is encountered in the data stream, the decoder knows that the MSB of the following byte is a stuff bit. This stuff bit has the same numeric weight as the LSB of the X′FF′ byte, and must be added into the latter in order that the carry which it "caught" may be released and allowed to complete its propagation. This action (called *resolving* the stuff) must take place early enough that decoding decisions based on the numeric weight of the code string are made correctly. We will see that correct decoding results if the stuff is resolved anytime prior to decoding the symbol whose encoding caused the carry-over into the stuff position.[3] In any case, the X′FF′ preceding the stuff bit can be completely shifted into the C register during decoding before the carry is propagated. This is because the code string is a sum of appropriately shifted Q-values corresponding to MPS events (recall that encoding an MPS results in some Q-value being added into the C register, whereas an LPS adds nothing to the C register). The X′FF′ was already part of the code string before the carry-causing MPS was encoded. Because the adding of the carry-causing MPS did not create the X′FF′, the Q-values pertaining to all prior MPSs are included in the code string through the X′FF′, and hence we can decode the code string completely through the X′FF′ before we need to propagate the carry.

Termination

Until now, we have stated that *any* point in the innermost subinterval is adequate for unique decoding. By convention, ABIC chooses the lower endpoint of the final interval to be the code point. This simplifies the hardware design because the C register already contains this value. After the last symbol from the source has been encoded, along with any concomitant renormalization, the C register is shifted left until its least significant bit has been shifted out into the code string, adding zeros if necessary to complete a byte. In the rare event that this last byte happens to take on the value X′FF′, an additional byte is transmitted containing X′00′, to satisfy any logic which automatically looks for a "stuff" bit following an X′FF′.[4] Decoding is terminated when the prearranged number of bits have been decoded.

Optimally filling the Q-table

This section seeks the best value to use for Q_e in a Q-Coder, given probability q that the next symbol to be encoded is an LPS. In a Q-Coder, the value A in the A register represents the width of the current interval on the number line. If an LPS occurs, the new interval has width Q_e; otherwise the new interval has width $A - Q_e$. Here "best" means minimizing the average code space taken to encode the next bit, where

$$\text{code space taken} = \log\left(\frac{\text{width of current interval}}{\text{width of new interval}}\right).$$

We want to select the value of Q_e that minimizes the expected value of this function of two independent random variables: (1) the current contents of the A register, and (2) whether an LPS or an MPS is to be encoded. We first derive an expression for the code space taken, then take its expected value over the two random variables. The resulting function of Q_e is then minimized by finding zeros of its derivative.

1. Since the A register has just been renormalized, $1 \le A \le 2$. If we assume that A is uniformly distributed on this interval, its probability density function is

$$p(A) = \begin{cases} 1 & \text{if } 1 \le a < 2, \\ 0 & \text{else.} \end{cases}$$

2. Suppose we have a good estimate q of the probability that the next symbol is an LPS.
   - LPS occurs with probability q. If LPS occurs,
     (a) Width of new interval is Q_e.
     (b) Code space taken is $\log(A/Q_e) = \log A - \log Q_e$.
   - MPS occurs with probability $1 - q$. If MPS occurs,
     (a) Width of new interval is $A - Q_e$.
     (b) Code space taken is $\log[A/(A - Q_e)] = \log A - \log(A - Q_e)$.

Because of the independence, we can nest the expectation in either order:

$$\text{average code space taken} = E_A[E_q(\text{code space taken})]$$

$$= \int_1^2 \{q[\log A - \log Q_e] + (1 - q)[\log A - \log (A - Q_e)]\}\, dA$$

$$= \int_1^2 [\log A - q \log Q_e - (1 - q) \log (A - Q_e)]\, dA$$

$$= \int_1^2 \log A\, dA - q \log Q_e \int_1^2 1\, dA - (1 - q) \int_1^2 \log (A - Q_e)\, dA,$$

and from the well-known [28] identity

$$\int \log x\, dx = x \log x - x,$$

[3] G. G. Langdon, Jr., University of California, Santa Cruz, CA, private communication.

[4] A proposed optimization is that any trailing zero bytes could be omitted from the code string, provided that the decoder can infer zeros if it needs additional bits after reaching the end of the code file. The present ABIC algorithm does not provide this feature.

average code space taken

$$= 2\log 2 - 1 - q\log Q_e - (1-q) \times [(2-Q_e)\log(2-Q_e) - (1-Q_e)\log(1-Q_e) - 1].$$

The minimum value for this function of Q_e can be found by setting its partial derivative with respect to Q_e equal to zero:

$$\frac{\delta(\text{average code space taken})}{\delta Q_e} = -\frac{q}{Q_e} + (1-q)[\log(2-Q_e) - \log(1-Q_e)] = 0.$$

Finally, to find the probability of q^* for which a given Q_e value minimizes the average code space taken, the above equation is solved for q:

$$q^* = \frac{Q_e[\ln(2-Q_e) - \ln(1-Q_e)]}{1 + Q_e[\ln(2-Q_e) - \ln(1-Q_e)]}.$$

The results are tabulated in **Table 2**.

ABIC implementation

• *Specifications*

We implemented the ABIC algorithm in a custom masterslice (Structured Array™)[5] silicon chip manufactured by LSI Logic Corporation [29]. The implementation of the ABIC algorithm in this VLSI technology is called the ABIC-1 chip, shown in **Figure 11**. The masterslice contains the desired combination of array logic and random-access memory (RAM) embedded in the diffusion layer, as shown in **Figure 12**. A large region of the masterslice is devoted to an array of transistors without pre-allocated routing channels (Compacted Array™)[5] for logic circuits and arithmetic logic units (ALUs). The RAMs and ALUs are provided by the supplier as large, customized design patterns called megacells. Since only metallization is needed to personalize the megacells and logic arrays, it was possible to overlap logic design and masterslice fabrication schedules. The logic array is based on 1.5-μm drawn-transistor geometries (size of smallest physical features) and a double-metal-layer high-performance complementary metal oxide semiconductor (HCMOS) process. This yields about 13,000 usable gates (two-input NAND equivalents) for data path and control logic. The whole chip is estimated to have about 194,000 transistors and a die size of 1 cm^2. Using this configuration, we enhanced a preliminary design based on previous 2.0-μm CMOS gate array and associated megacell technology.

Two fast on-chip static RAM megacells were configured by metallization of a prefabricated diffusion pattern to be 256 words by 36 bits each (18,432 bits total). The RAM megacells store previous-line data for the model and statistics

[5] Structured Array™ and Compacted Array™ are trademarks of LSI Logic Corporation, Milpitas, CA.

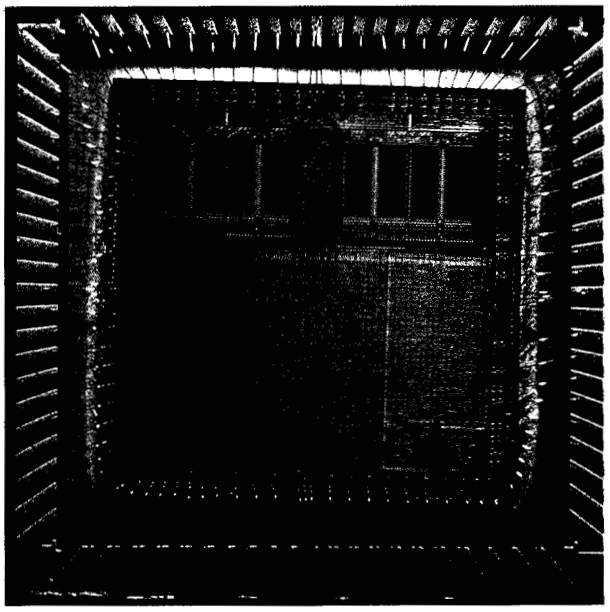


Figure 11

ABIC-1 die photograph.

Table 2 Values of q for which each given Q-value is optimal.

Q-index	*Q-value (Q_e)*		*Q_e/1.333*	*q^**	*Q_e/q^**
	Hex	*Decimal*	*Decimal*	*Decimal*	*Decimal*
0	0.AC1	0.67212	0.50409	0.48456	1.387
1	0.A81	0.65649	0.49237	0.47239	1.390
2	0.A01	0.62524	0.46893	0.44833	1.395
3	0.901	0.56274	0.42206	0.40107	1.403
4	0.701	0.43774	0.32831	0.30908	1.416
5	0.681	0.40649	0.30487	0.28646	1.419
6	0.601	0.37524	0.28143	0.26397	1.422
7	0.501	0.31274	0.23456	0.21929	1.426
8	0.481	0.28149	0.21112	0.19709	1.428
9	0.441	0.26587	0.19940	0.18602	1.429
10	0.381	0.21899	0.16425	0.15292	1.432
11	0.301	0.18774	0.14081	0.13094	1.434
12	0.2C1	0.17212	0.12909	0.11997	1.435
13	0.281	0.15649	0.11737	0.10902	1.435
14	0.241	0.14087	0.10565	0.09808	1.436
15	0.181	0.09399	0.07050	0.06534	1.439
16	0.121	0.07056	0.05292	0.04901	1.440
17	0.0E1	0.05493	0.04120	0.03814	1.440
18	0.0A1	0.03931	0.02948	0.02728	1.441
19	0.071	0.02759	0.02069	0.01914	1.442
20	0.059	0.02173	0.01630	0.01507	1.442
21	0.053	0.02026	0.01520	0.01405	1.442
22	0.027	0.00952	0.00714	0.00660	1.442
23	0.017	0.00562	0.00421	0.00389	1.442
24	0.013	0.00464	0.00348	0.00322	1.443
25	0.00B	0.00269	0.00201	0.00186	1.443
26	0.007	0.00171	0.00128	0.00118	1.443
27	0.005	0.00122	0.00092	0.00085	1.443
28	0.003	0.00073	0.00055	0.00051	1.443
29	0.001	0.00024	0.00018	0.00017	1.443

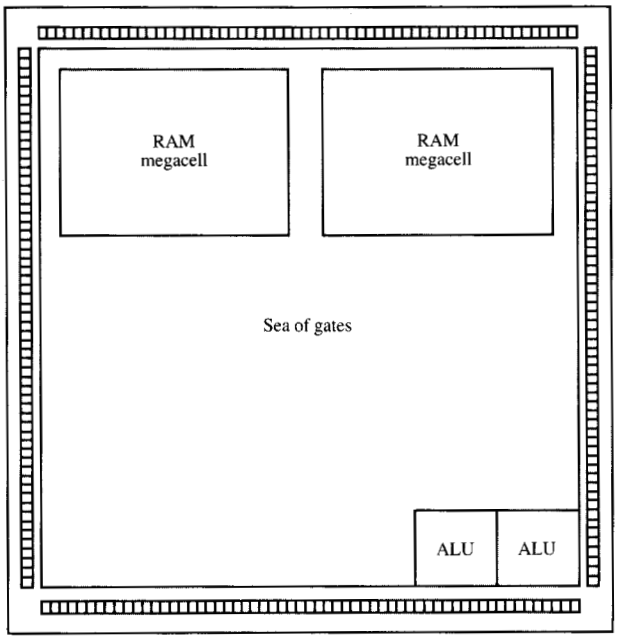


Figure 12

ABIC-1 "floor plan."

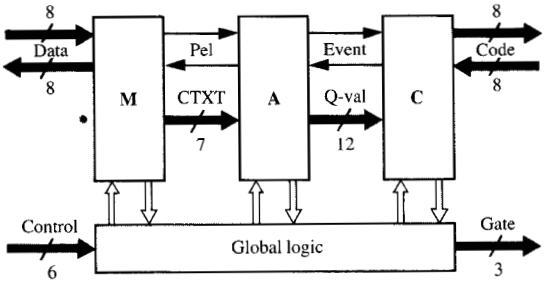


Figure 13

ABIC-1 block diagram.

for the adapter. Uniform RAM format was selected to allow memory pooling. This maximizes adapter RAM capacity for various ABIC-1 operating modes. On-chip embedded RAMs are used to eliminate I/O delays for speed, but off-chip RAMs also are supported for flexibility. For high performance, the design is pipelined with several stages and logic is performed during both phases of the two-phase, nonoverlapping clock cycle (two distinct, sequential clock waveforms). We specified two 16-bit ALU megacells to obtain the fast arithmetic operations needed by the ABIC algorithm. The ALUs perform addition and subtraction for the A register and C register of the coder. The vendor implemented their design of the ALUs as custom metallization patterns in the logic array.

These megacells and the fast array logic are the basic elements for this high-performance design. The original design was targeted for two-phase clocking at 20 MHz nominal (5.00 V, 25°C, nominal process variation), equivalent to 10 MHz worst case (4.75 V, 70°C, worst-case process variation). Timing analysis shows that the final design achieved this goal with a significant margin.

The high level of integration results in a low-cost multipurpose chip for compression and decompression of bilevel image data using adaptive binary arithmetic coding. The same functional core is used in both compression and decompression. For added flexibility and research purposes, the chip also can operate with external RAMs. Since this implementation of the algorithm fits in a single chip, the ABIC-1 has a relatively low external pin count, with only 37 inputs, 35 outputs, and 10 power/ground pins. Hence it can be packaged in an 84-pin ceramic pin grid array or in a surface-mount package.

• *Architecture*

To avoid clock skew problems, we followed a latch-based (level-sense) design discipline using two-phase nonoverlapping clocks. To enhance testability, most latches are scannable via multiplexers at their data inputs. This is similar in purpose to level-sensitive scan design (LSSD) [30], but does not conform strictly to the rules of that approach. More than one third of the ABIC-1 design logic components are latches, and use of LSSD latches would have increased the gate count significantly at the expense of functionality. Most ABIC-1 internal registers are accessible via 18 separate scan paths. Some registers also are parallel-accessed via input and output buses.

For uniformity and consistency, we chose to have all input and output signals clocked by the same phase. Thus all primary inputs and outputs are phase 1 (i.e., ".1") signals. The sources of all ABIC-1 inputs should be latched by phase 1 (ϕ_1), since they are transferred to ABIC-1 latches clocked by phase 2 (ϕ_2). Similarly, all ABIC-1 outputs are latched internally by ϕ_1 and therefore are stable during ϕ_2. In principle, then, the inputs and outputs of ABIC-1 are self-consistent with respect to clock phases. A machine cycle is thus defined to be a ϕ_1–ϕ_2 pair.

At a high level, the input and output data paths are organized into two groups. The first group, BUS1.IN and BUS1.OUT, is 8 bits wide. It is used to transfer compressed/decompressed data to and from the chip. The second group, BUS2.IN and BUS2.OUT, is 18 bits wide. It is used to transfer additional parameters needed by initialization and various operating modes. There is also a

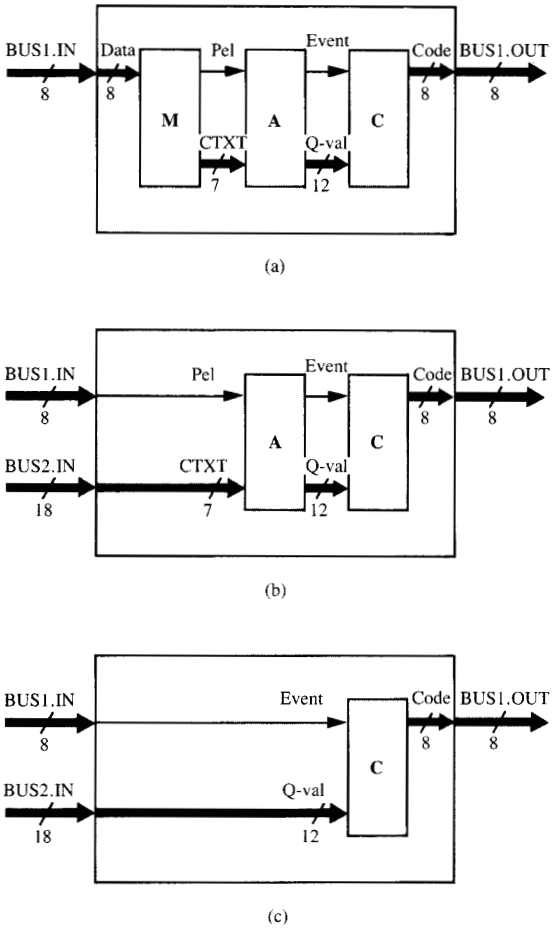


Figure 14

ABIC-1 Encode partitioning modes: (a) MAC, (b) AC, and (c) C.

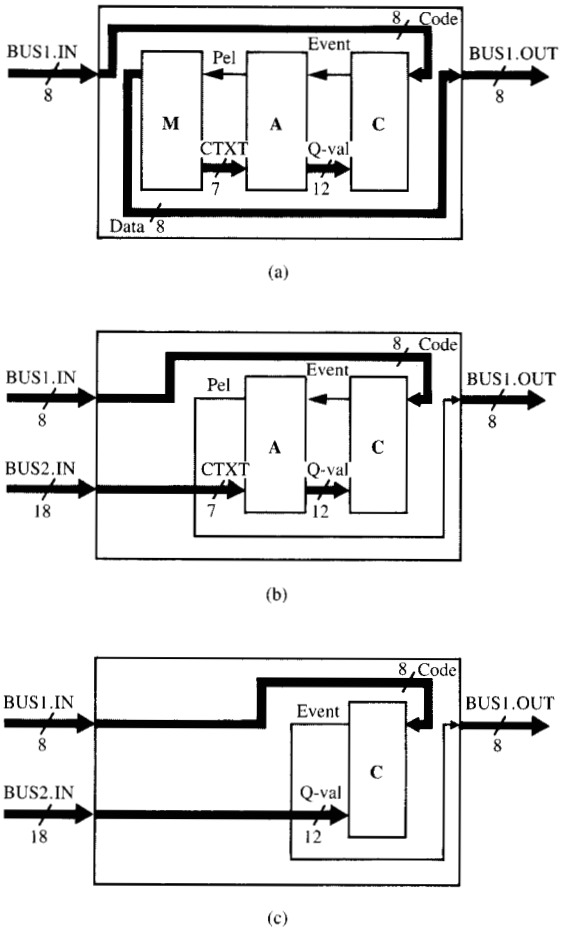


Figure 15

ABIC-1 Decode partitioning modes: (a) MAC, (b) AC, and (c) C.

group of control inputs that address registers for parallel access and select major operating states. Transfers on the data buses between the chip and external logic are supervised by a group of I/O "handshake" outputs from ABIC-1. For maximum performance, the handshake allows a transfer during each machine cycle. The data transfers are semisynchronous, as they are timed by the machine-cycle clocks.

In addition to normal encode/decode operation, ABIC-1 also supports loading and unloading of its internal RAMs. This is necessary for initialization and also allows a predetermined set of statistics to be stored in the statistics RAM before encoding or decoding. The data for the RAM are presented to the ABIC-1 via BUS2.IN as a burst transfer without handshaking. Since this bus is only 18 bits wide and the RAM is organized as 256 × 36 bits, two machine cycles are required to pack one full word for each RAM address. Since a RAM read or write cycle can be accomplished in one machine cycle, ABIC-1 takes advantage of this and multiplexes the loading and unloading cycle. In LOAD mode, the previous data in the RAM appear at BUS2.OUT (multiplexed 18 bits at a time) as the new data are presented at BUS2.IN. In DUMP mode, the contents of the RAM are read out without being overwritten. The previous-line RAM can be accessed similarly.

The ABIC-1 is partitioned into three functional blocks, each with its own local control logic, and one global "switchyard-interface" block, as shown in **Figure 13**. The three functional blocks are the model, the adapter, and the coder. This gives rise to the multi-partitioning modes of

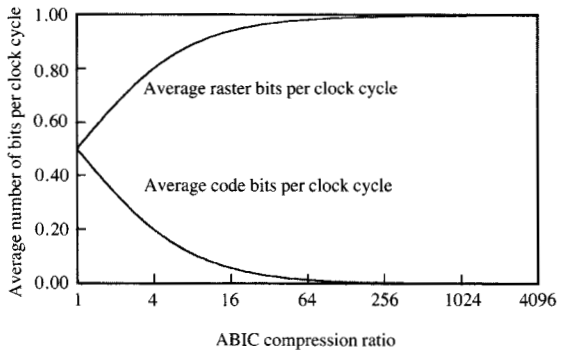


Figure 16

Effect of compression ratio on throughput.

operation using appropriate combinations of M, A, and C blocks. There are three primary partitioning modes, MAC, AC, and C, as shown for Encode in **Figure 14**, and for Decode in **Figure 15**. Compression (Encode) and decompression (Decode) are done using the same "core" logic blocks. In full-function (MAC) mode, the application is compression/decompression of bilevel images. In adapter/coder (AC) mode, the chip implements the Q-Coder function to be used with user-supplied, external model hardware. In coder-only (C) mode, the chip handles only the encoding or decoding operation, while external logic provides the necessary model and adapter functions. In addition, the use of internal on-chip RAMs and/or external off-chip RAMs is supported and provides two additional partitioning modes, M′AC and A′C. In M′AC mode, an external previous-line model RAM is used with the chip. The external RAM is connected to the ABIC-1 via BUS2.IN. In A′C mode, similar to M′AC, an external adapter RAM is used with the chip together with the internal Q-Coder adapter logic. The external RAM is interfaced to the ABIC-1 via BUS2.IN and BUS2.OUT. The internal-adapter-RAM write control can be disabled to provide nonadaptive operation. In addition to allowing access to a general-purpose building block such as the Q-Coder, this partitioning approach facilitated functional debugging in successive stages during logic design and prototype evaluation.

System integration

This section analyzes some issues of data flow rate which must be considered when building data compression into a real-time system.

• *Throughput characteristics: Compression ratio and clock multiplier*

The ratio between the number of raster bits in an image and the number of code bits representing the same image is known as the *compression ratio*. For the ABIC algorithm, the compression ratio depends on the image being encoded. An image consisting of purely random pixels will generally not compress at all, while a blank page compresses very well. The bounds are easily established. The lower bound on compression ratio is set by images which do not compress at all (and may even expand); in this case the raster image can be transmitted ahead.[6] Thus the lower bound is 1.0. The upper bound is determined by the finite-precision arithmetic of the ABIC algorithm. The most highly skewed symbols are encoded with a Q-value of 2^{-12}. It takes $2^{12} - 1 = 4095$ such Q-values to complete a renormalization shift cycle (decreasing the A register from 1.FFE to 0.FFF hexadecimal, when it is shifted to 1.FFE again). Hence the theoretical maximum compression ratio is 4095. A finite-dimension blank image does not quite achieve the bound, because the adapter takes time to reach its most highly skewed state. Practical images lie between these bounds. For example, the ABIC compression ratio for the CCITT test set ranges from 8.8 for No. 7 through 57.4 for No. 2.

Every ABIC clock cycle processes either one bit of raster data or one bit of code data (with a very few exceptions relating to bit-stuffing and un-stuffing logic). This is because each clock cycle is devoted to either an ALU cycle or a renormalization shift. During encode (or decode) operation, an ALU cycle encodes (or decodes) a raster data bit and a renormalization emits (or accepts) a code data bit. The process is symmetric; the time required to encode an image is very nearly equal to the time required to decode it.

It is useful to define the term *clock multiplier* as the average number of raster bits processed per clock cycle. If M is the clock multiplier, and R is the compression ratio, then the definition of compression ratio implies that the average number of code bits per clock cycle is M/R. Since each clock cycle handles either a raster bit or a code bit,

$$M + \frac{M}{R} = 1.$$

We may then solve for the clock multiplier and for the average number of code bits per clock cycle:

$$M = \frac{R}{R+1},$$

$$\frac{M}{R} = \frac{1}{R+1}.$$

These two expressions are plotted in **Figure 16**, as the compression ratio R ranges between its bounds. The figure is vertically symmetrical because the two rates sum to unity.

We conclude that the number of clock cycles required to process an image equals the sum of the raster file size plus the compressed file size (all sizes in bits). For example, the

[6] A single bit can prefix the code string, to provide an escape mechanism to indicate that the raster image is being literally transmitted.

CCITT test images are each 4,105,728 bits and compress with ratios between 8.8 and 57.4, so they require between 4.18 and 4.57 million cycles to compress or decompress. At a 10-MHz clock frequency this would take between 0.418 and 0.457 seconds. The relative constancy of processing time, despite wide variations in compression ratio, is a feature of the ABIC hardware implementation. The compressed data rate varies widely with the compression ratio, as is perhaps better illustrated by replotting the same data on a logarithmic scale, with an assumed 10-MHz clock, as shown in **Figure 17**.

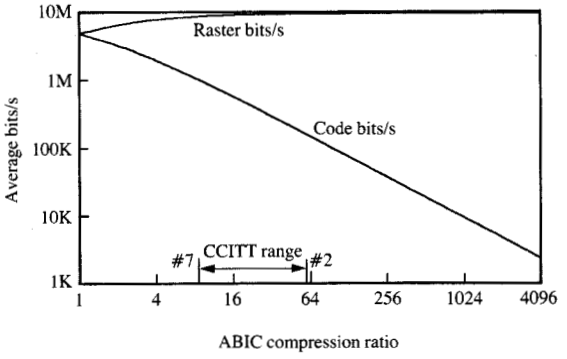


Figure 17

Effect of compression ratio on throughput (log scale; 10-MHz clock).

Application analysis

In a practical application, the ABIC will be installed between two rate-limited channels, one handling uncompressed raster data and the other handling compressed code data. Thus, there are three potential bottlenecks:

- Raster data channel capacity.
- Compressed (code) data channel capacity.
- ABIC speed, determined by clock frequency and clock multiplier.

To ensure that the ABIC is not a bottleneck in the system, its throughput must be at least as fast as that of the slower of the two channels with which it interfaces. However, the capacities of the raster and code data channels cannot be directly compared. Most system designs take advantage of the expected compression ratio to permit a slower rate for the code channel than for the raster channel. In order to determine which is the limiting factor, one channel is chosen as a reference point, and the rate of the other channel is reflected through the ABIC to that reference point. If the raster channel is chosen as the reference point, the capacity of the code channel is multiplied by the compression ratio to get its apparent capacity as seen from the point of view of the raster channel. Because the compression ratio depends on the image, a given system may have its bottleneck in the raster channel for some images and in the code channel for others. For example, suppose that a particular system can process raster data at 8 megabits per second and compressed data at 2 megabits per second. For images with a compression ratio greater than 4, the capacity of the compressed data channel is adequate but the raster data channel is the limiting factor. For images with a compression ratio less than 4, the situation is reversed.

The ABIC is introduced as the third potentially limiting factor, given its clock rate and the clock multiplier (a function of compression ratio, as discussed above). For example, on an image with a compression ratio of 9, an ABIC clocked at 10 MHz achieves a raster data rate of 9 megabits per second and a compressed data rate of 1 megabit per second. It is instructive to plot the throughput at the reference point as a function of compression ratio, for each of the three potential limiting factors, on the same set of axes. **Figure 18** exhibits this kind of plot for the present example. A system can operate anywhere in the region below all three limits.

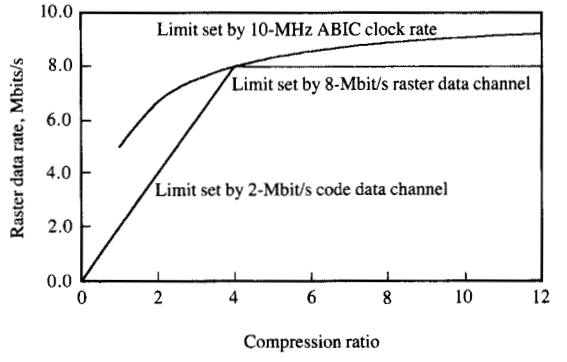


Figure 18

Example bottleneck analysis with all data rates normalized to raster data channel.

In most systems, data channels having the capacity to transmit raster images quickly are scarce and expensive. For this reason, a wise system designer will locate data compression at the extremities of the system, near the sources and sinks of image data (scanners, displays, and printers), so that only compressed data are handled on slow buses.

Summary

This paper describes the design and implementation of a multi-purpose, *adaptive* data compressor/decompressor in a single VLSI chip. The design has been optimized for speed, flexibility, and modularity, while maintaining a limited number of pins and employing an aggressive ASIC technology. Its development required insight and extensive trade-offs in the fields of both information theory and VLSI design. The final design exceeded its speed objective of 10-MHz (worst-case) operation, while also meeting throughput goals to compress and decompress binary image documents of all kinds at nearly uniform rates.

The ABIC algorithm for bilevel image data compression has been shown to consistently outperform the nonadaptive CCITT algorithm on the CCITT set of test documents. Even better comparative performance has been demonstrated for other kinds of data such as digital halftones. The architectural principles used in designing this algorithm have also been described in detail, to illustrate the use of adaptive binary arithmetic coding (ABAC) technology.

In summary, the ABIC chip offers competitive and robust compression performance in a low-pin-count, hence inexpensive, VLSI package, which should make it attractive for bilevel image systems of all kinds. Its adaptive arithmetic coding technology, combined with the modularity designed into the chip, also permits it to function as a research component—for use, e.g., in building future hardware implementations of more complex adaptive algorithms.

Glossary

A register	A register representing the width of the current interval on the number line. In ABIC, it is a 13-bit register whose highest-order bit position is always 1 or else renormalized to be 1, and whose 12 lower-order bit positions are aligned with the 12-bit Q-values.
ABIC	Adaptive Bi-level Image Comdec. A particular algorithm for compressing binary images. VLSI circuit chip from IBM Almaden Research Center. The subject of this paper.
Adapter	That portion of a data compression system which estimates the probability distribution of the next symbol from the source, and then revises the estimate for next time after seeing which symbol actually occurs. Typically the adapter is multiplexed over multiple *contexts*, in which case its estimate is based on the context, and revised for the next occurrence of the same context.
Alphabet	The set of possible values a given symbol can take on.
ALU	Arithmetic and logic unit. A circuit capable of adding, subtracting, etc.
Binary event	An event which can have one of two outcomes.
Bit	A quantity which can take on only the values 0 and 1.
Byte	A group of eight bits, taken as a unit.
C register	A register representing the lower endpoint of the current interval on the real-number line. In ABIC, it is a 16-bit register with an implied radix point above the lower-order 12 bits which are aligned with the 12-bit Q-values. The leading bits are called "spacers." Encoded output is created by left-shifting the C register into buffers which handle the carry-over and align the result to 8-bit bytes.
CCITT	Consultative Committee for International Telephony and Telegraphy. An international standards body.
Clock multiplier	The average number of raster bits processed per clock cycle.
Code point	A point on the real number line, chosen to designate the innermost subinterval in an arithmetic encoding. The value of the code string, taken as a binary fraction.
Code string	A sequence of bytes from a coder representing all the information from a source file, but in a compact way.
Coder	That portion of a data compression system which combines an estimate of the probability distribution of the next symbol from the source, with the actual symbol emitted by the source, into a code string.
Compression	Representing data in a more compact way, eliminating redundancy.
Compression ratio	The number of raster bits divided by the number of code bits.
Context	Information provided by the *model* which helps the *adapter* estimate the probability distribution of the next symbol. Typically the context is a subset or other function of the source data which has already been processed.
Entropy	A mathematical measure of uncertainty, given by the expected

	logarithm of inverse probability: $H(P) = \sum_i p_i \log (1/p_i)$.
Event	A happening with an outcome which is not known in advance.
HCMOS	High-performance complementary metal-oxide-semiconductor. An integrated-circuit technology featuring low static power and high-speed operation.
Lossless	Applied to data compression, means the source data can be reconstructed exactly.
LPS	Less probable symbol. With a binary source, the bit with the lesser probability.
LSB	Least significant bit. Rightmost bit in a binary register.
LSSD	Level-sensitive scan design. A logic design discipline featuring delay-independent (level-sensitive) operation and a capability to interconnect all latches into a shift-register chain for loading and unloading test data (scan design), usually implemented with two-phase nonoverlapping clocks.
Markov string	A string of symbols for which the probability distribution of each symbol depends on the value of the preceding symbol.
Megacell	A predesigned portion of very-large-scale integrated circuit, in which usually all layers have been customized to a specific function.
Model	That portion of a data compression system which establishes the *context*.
MPS	More probable symbol. With a binary source, the bit with the greater probability.
MSB	Most significant bit. Leftmost bit in a binary register.
Pel	Picture element. See *pixel*.
Phase	In clocked synchronous logic, a portion of a clock cycle where one of several clock signals is true.
Pixel	Picture element. A number representing the brightness of a single point in an image. The ABIC compresses bilevel images where pixels are constrained to have values 0 and 1.
Q-Coder	A particular adaptive arithmetic coder featuring a renormalization-driven adapter with sixty states per context.
Q-index	A small integer selecting one of the possible Q-values. In ABIC the Q-index is a 5-bit integer, representing a number in the interval $0 \cdots 29$.
Q-value	A binary fraction representing approximately 1.5 times the probability that the next symbol will be an LPS. In ABIC it is a 12-bit binary fraction, with 12 bits to the right of the radix point.
RAM	Random-Access Memory. In the ABIC, two RAMs are used—one to store the previous raster line and one to store the adapter state for each context.
Raster	A means of converting an image to a serial data stream, in which it is scanned line by line.
Renormalization	Scaling the content of a register by shifting it left so that the most significant bit has value 1.
Skew	The degree of extremity of probability, usually given by $-\log_2 [\min (p, 1-p)]$.
Skew coder	A coder where the Q-values are constrained to be powers of 1/2 (of the form 2^{-k} for integer k).
Source	The uncompressed data, taken as input to the coder.
Spacer	Extra bit(s) at the most significant end of a register, where carries may be resolved, before being propagated out.
State	In a finite-state machine, the content of the memory, giving the current condition of the machine.
State record	In a state machine multiplexed over several tasks, a record in memory giving the state of one of its tasks. See *state*.
Stuff	A bit (or bits) inserted into a binary number having the same algebraic weight as its predecessor(s), for the purpose of intercepting a carry which would otherwise propagate through a long run of 1s.
Symbol	An element of source data to be encoded in one iteration of the coding algorithm. Generalized arithmetic coding theory allows symbols to be taken from large alphabets (for example, a byte is a symbol from an alphabet with 256 elements). The Q-Coder in the ABIC can only encode binary symbols, which with the ABIC model represent one binary image pixel each.

Acknowledgments

We would like to acknowledge the extensive support of Behnam Tabrizi and Joe-Ming Cheng in simulation and testing of our ABIC chip, as well as consulting by our colleague Prof. Glen Langdon[7] on the chip design. We would also like to acknowledge the cooperation of fellow researchers Joan Mitchell and William Pennebaker from the IBM T. J. Watson Research Center, in working together to arrive at a hardware/software-optimized common form of adaptive arithmetic coding—the Q-Coder. Finally, we would like to thank Jeffrey Hibbard, Jean Chen, and Lee Bailey for helping us to prepare this paper on a tight schedule. Funding for the fabrication of our VLSI chip was provided by the IBM Development Laboratories in Yamato, Japan, and in Charlotte, North Carolina.

References

1. N. Abramson, *Information Theory and Coding*, McGraw-Hill Book Co., Inc., New York, 1963.
2. R. Ash, *Information Theory*, Interscience Publishers–John Wiley & Sons, Inc., New York, 1965.
3. R. B. Arps, "Binary Image Compression," Chapter 7 in *Image Transmission Techniques*, W. K. Pratt, Ed., Academic Press, Inc., San Francisco, 1979, pp. 219–276.
4. R. B. Arps, "Bibliography on Digital Graphic Image Compression and Quality," *IEEE Trans. Info. Theory* **IT-20,** 120–122 (1974).
5. R. B. Arps, "Bibliography on Binary Image Compression," *Proc. IEEE* **68,** No. 7, 922–924 (1980).
6. R. Hunter and A. Robinson, "International Digital Facsimile Coding Standards," *Proc. IEEE* **68,** No. 7 (1980).
7. Y. Chen, F. Mintzer, and K. Pennington, "PANDA: Processing Algorithm for Noncoded Document Acquisition," *IBM J. Res. Develop.* **31,** 32–43 (1987).
8. R. B. Arps, *COMDEC-1 Preliminary Functional Specification*, IBM-SDD, Los Gatos, CA, October 8, 1973, pp. 1–59.
9. J. Landau and J. Williamson, "System Unites Image, Text Processing," *Electronics Week* **57,** No. 25, 55–58 (October 1, 1984).
10. J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.* **20,** No. 3, 198–203 (1976).
11. R. C. Pasco, *Source Coding Algorithms for Fast Data Compression*, Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, May 1976.
12. G. G. Langdon, Jr., and J. J. Rissanen, "A Simple General Binary Source Code," *IEEE Trans. Info. Theory* **IT-28,** No. 5, 800–803 (1982).
13. G. G. Langdon, Jr., and J. J. Rissanen, "A Double-Adaptive File Compression Algorithm," *IEEE Trans. Commun.* **COM-31,** No. 11, 1253–1255 (1983).
14. G. G. Langdon, Jr., and J. J. Rissanen, "Method for Converting Counts to Coding Parameters," *IBM Tech. Disclosure Bull.* **22,** No. 7, 2880–2882 (December 1979).
15. G. G. Langdon, Jr., and J. J. Rissanen, "Universal Modelling and Coding," *IEEE Trans. Info. Theory* **IT-27,** No. 1, 12–23 (1981).
16. G. G. Langdon, Jr., and J. J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Commun.* **COM-29,** 858–867 (1981).
17. G. G. Langdon, Jr., "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.* **29,** No. 2, 135–149 (1984).
18. R. B. Arps, *The Entropy of Printed Matter at the Threshold of Legibility*, Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, June 1969.
19. D. Preuss, "Two-Dimensional Facsimile Source Encoding Based on a Markov Model," *Nachrichtentech. Z.* **28,** No. 10, 358–363 (1975).
20. D. R. Helman, G. G. Langdon, Jr., N. Martin, and S. J. P. Todd, "Statistics Collection for Compression Coding with Randomizing Feature," *IBM Tech. Disclosure Bull.* **24,** No. 10, 4919 (March 1982).
21. W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr., and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.* **32,** 717–726 (1988, this issue).
22. W. B. Pennebaker and J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* **32,** 737–752 (1988, this issue).
23. J. L. Mitchell and W. B. Pennebaker, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* **32,** 753–774 (1988, this issue).
24. J. L. Mitchell and W. B. Pennebaker, "Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder," *IBM J. Res. Develop.* **32,** 727–736 (1988, this issue).
25. Frederick Jelinek, *Probabilistic Information Theory*, McGraw-Hill Book Co., Inc., New York, 1968.
26. R. B. Arps, J. M. Cheng, and G. G. Langdon, Jr., "Method for Carry-Over Control in a FIFO Arithmetic Code String," *IBM Tech. Disclosure Bull.* **25,** No. 4, 2051 (September 1982).
27. G. G. Langdon, Jr., and J. J. Rissanen, "A Method and Means for Carry-Over Control in a High Order to Low Order Combining of Digits of a Decodable Set of Relatively Shifted Finite Number Strings," U.S. Patent 4,463,342, July 31, 1984.
28. Formula 485, *CRC Math Tables*, 22nd Edition, Chemical Rubber Company, Inc., Cleveland, OH, 1974, p. 450.
29. *LSA1500 1.5-Micron HCMOS Structured Array™ Series*, LSI Logic Corporation, Milpitas, CA, November 1986.
30. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference*, June 1977, New Orleans, LA, pp. 462–468.

Received May 2, 1988; accepted for publication September 2, 1988

[7] Professor Langdon, formerly located at the IBM Almaden Research Center, is now on the faculty of the University of California at Santa Cruz.

Ronald B. Arps *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Dr. Arps received the B.S., M.S., and Ph.D. degrees in electrical engineering from the California Institute of Technology, Pasadena, in 1960; Oregon State University, Corvallis, in 1963; and Stanford University, Stanford, in 1969, respectively. From 1960 to 1962 he was with the Electrodata Division, Burroughs Co., Pasadena. He has been with IBM since 1963, starting in its Advanced Systems Development Division at Los Gatos and currently in its Research Division at San Jose. His assignments have included exploratory studies on processing and compressing binary images, advanced development of computer peripherals and systems, and research into hardware-optimized adaptive compression and implementation of algorithms in VLSI microsystems. Dr. Arps received a Resident Study Award to Stanford University in 1967–69 and taught as an IBM Visiting Scientist at the Swiss Federal Institute of Technology, Zurich, during 1970–71. During 1977–78, he was on leave as a visiting associate professor at Linkoping University in Sweden. In 1979, Dr. Arps published a chapter entitled "Binary Image Compression" in *Image Transmission Techniques* (Academic Press, New York, 1979). Dr. Arps is currently manager of the VLSI-Oriented Algorithms project at the IBM Almaden Research Center and architect of the ABIC VLSI chip for Adaptive Binary Image Compression. His research interests include adaptive data compression algorithms as well as image processing, office automation, and computer-aided design of LSI.

Thomas K. Truong *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Mr. Truong is a Staff Engineer and has been with the VLSI-Oriented Algorithms Project since 1983. He is the co-designer of the ABIC-1 chip and is also responsible for the design and implementation of various adaptive data compression prototypes. His research interests range from algorithms to implementations in VLSI. Mr. Truong has written numerous demonstration and application programs for the IBM host mainframe and PC workstation environments. He received his B.S. in electrical engineering and computer science from the University of California at Berkeley in 1983, and his M.S. in electrical engineering from Stanford University through the IBM Honors Co-Operative program in 1987. Mr. Truong is a member of Eta Kappa Nu, the Institute of Electrical and Electronics Engineers, and Tau Beta Pi.

David J. Lu *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Dr. Lu received the B.A. in engineering and applied physics, with the Tau Beta Pi Prize for excellence in engineering sciences, from Harvard University in 1973. He received the M.S. in electrical engineering in 1975, and the Ph.D. in electrical engineering with a minor in computer science in 1981, from Stanford University. Dr. Lu was a Research Associate in the Center for Reliable Computing and the Computer Systems Laboratory at Stanford University, and a consultant in computer engineering, from 1981 to 1984. He joined IBM as a Research Staff Member in 1985. Dr. Lu is a co-designer of the ABIC-1 VLSI chip; his research interests center on the design and testing of application-specific integrated systems. He is a member of the Institute of Electrical and Electronics Engineers and an associate member of Sigma Xi.

Richard C. Pasco *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* At the IBM Almaden Research Center, as a contributor to the VLSI-Oriented Algorithms Project, Dr. Pasco has contributed to the design of the ABIC chip as well as a VLSI display controller for fabrication by a gate-array vendor. His research interests include special-purpose architectures, information theory, user-friendly systems, and computer-aided engineering. Dr. Pasco received his B.S. in electrical engineering from Rose-Hulman Institute of Technology, Terre Haute, Indiana, in 1972, and his M.S. and Ph.D. in electrical engineering from Stanford University in 1973 and 1976. He served on the technical staff at Bell Laboratories, Stanford Telecommunications, and TRW Vidar. At the Xerox Palo Alto Research Center, Dr. Pasco designed several experimental full-custom LSI devices (including a digital filter and an Ethernet controller), and helped develop a course in user-designed full-custom VLSI. As Manager of VLSI Design at Atari, he led development of several vendor gate-array chips (including a RAM controller and memory management unit). Dr. Pasco is a member of the Association for Computing Machinery, Eta Kappa Nu, the Institute of Electrical and Electronics Engineers, and Tau Beta Pi.

Theodore David Friedman *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* A Research Staff Member at the Almaden Research Center, in 1963 Dr. Friedman joined the IBM T. J. Watson Research Center, where he managed the Machine Assisted Design Project and developed "ALERT," the first behavioral logic design synthesizer. In 1970 he transferred to IBM's San Jose Research Laboratory, where he designed analysis tools for digital networks and devised an access control system for shared data. He also worked in image enhancement and font design, as well as data compression. Prior to joining IBM, Dr. Friedman was a member of the scientific staff at Technical Research Group, Inc., where he participated in the development of space-vehicle guidance systems. He received a B.A. degree from the University of Michigan in 1958, and M.S. and Ph.D. degrees from the University of California, Berkeley, in 1973 and 1976, respectively. Dr. Friedman has received IBM awards for his work on logic synthesis and font rescaling.