Hardware Implementation of a Lossless Image Compression Algorithm Using a Field Programmable Gate Array

M. Klimesh,[1] V. Stanton,[1] and D. Watola[1]

We describe a hardware implementation of a state-of-the-art lossless image compression algorithm. The algorithm is based on the LOCO-I (low complexity lossless compression for images) algorithm developed by Weinberger, Seroussi, and Sapiro, with modifications to lower the implementation complexity. In this setup, the compression itself is performed entirely in hardware using a field programmable gate array and a small amount of random access memory. The compression speed achieved is 1.33 Mpixels/second. Our algorithm yields about 15 percent better compression than the Rice algorithm.

I. Introduction

Lossless image compression is well-established as a means of reducing the volume of image data from deep-space missions without compromising the data quality. Missions often desire hardware to perform such compression, in order to reduce the demand on spacecraft processors and to increase the speed at which images can be compressed. Currently, the only available space-qualified hardware designed for lossless compression is based on the Rice compression algorithm [5].[2]

Rice compression has limitations, however, and there are other algorithms that achieve better lossless image compression. An algorithm known as LOCO-I (low complexity lossless compression for images) [6,7] is an appealing choice. LOCO-I is at the core of JPEG-LS, the algorithm selected by the Joint Photographic Experts Group as the International Standards Organization/International Telecommunications Union (ISO/ITU) standard for lossless and near-lossless compression of continuous-tone images [3,7].

We have modified LOCO-I to reduce the complexity of implementation on a field programmable gate array (FPGA). We refer to this modified version as FPGA LOCO. Our implementation of FPGA LOCO in hardware represents a first step toward producing a space-qualified hardware version of the LOCO-I algorithm.

---

[1] Communications Systems and Research Section.

[2] The Consultative Committee for Space Data Systems (CCSDS) lossless data compression standard [1] is a particular implementation of the Rice algorithm.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

For natural images, the compression achieved by FPGA LOCO varies slightly from that of the more complex JPEG-LS, with a slight average performance edge going to JPEG-LS. Both are better than Rice compression typically by around 15 percent. Table 1 compares the compression achieved by FPGA LOCO, JPEG-LS, and the CCSDS version of Rice compression on a set of images representative of those acquired by spacecraft. Tests on other natural images yield similar results. As FPGA LOCO is designed for 8-bit (gray-scale) images, all images tested were of this type.

The remainder of this article is organized as follows. Section II contains a detailed description of the FPGA LOCO algorithm. Section III describes our implementation. We conclude with Section IV, which contains performance estimates.

Table 1. Comparison of the compression achieved by our algorithm (FPGA LOCO), JPEG-LS, and CCSDS Rice compression, on a test set of 8-bit images. Rate is defined as the average number of bits per pixel in the compressed image.

Image	Image description	Rate, bits/pixel		
		FPGA LOCO	JPEG-LS	Rice
1	Lunar	4.48	4.35	5.44
2	Mars (Pathfinder)	4.61	4.69	5.72
3	Mars (Viking Orbiter)	3.07	2.95	3.67
4	Mars (Viking Lander)	4.79	4.76	5.32
5	Venus (Magellan radar image)	4.16	4.17	4.67
6	Europa (Galileo)	5.41	5.44	6.48
Average		4.42	4.39	5.22

II. Algorithm

The algorithm described in this section is based on the LOCO-I algorithm of [6].

The FPGA LOCO algorithm takes as input a rectangular image with 8-bit pixel values (the pixel values are within the range 0 to 255). The compressed image produced is a sequence of bits from which the original image can be reconstructed. Let `wd` be the image width and `ht` be the image height. Pixels are identified by coordinates (x, y) with x in the range $[0, \texttt{wd} - 1]$ and y in the range $[0, \texttt{ht} - 1]$. In our illustrations, $(0, 0)$ corresponds to the upper left corner of the image.

The FPGA LOCO algorithm is based on predictive compression (see, e.g., [4]). During compression, the pixels of the image are processed in raster scan order. Specifically, y is incremented through the range $[0, \texttt{ht} - 1]$, and for each y value, x is incremented through the range $[0, \texttt{wd} - 1]$. (Thus, the y-dimension is the slowly varying dimension.)

The first two pixels, with coordinates $(0, 0)$ and $(1, 0)$, are simply put into the output bit stream uncoded.

For all other pixels of the image, the processing that occurs can be conceptually divided into four steps:

(1) Classify the pixel into one of several contexts according to the values of (usually 5) previously encoded pixels.

(2) Estimate the pixel value from (usually 3) previously encoded pixels, and add a correction (called the bias), which depends on the context.

(3) Map the difference between the estimate and the actual pixel value to a nonnegative integer, and encode this integer using Golomb's variable length codes [2].

(4) Update the statistics for the context based on the new pixel value.

These steps are explained in detail below.

A. Contexts

During the encoding loop, each pixel is classified into one of several contexts based on quantized values of differences between pairs of nearby pixels. The context is used to estimate the distribution on the prediction residuals and to determine a small estimated correction to the prediction. Both of these estimates are determined adaptively and on-the-fly, based solely on statistics for previous pixels with the same context.

The context of a pixel p is based on the values of 5 previous pixels a through e as shown in Fig. 1. Assume for now that p is sufficiently far from the image edges that a through e all exist. The context is determined by the values $Q_7(a-c)$, $Q_7(d-a)$, $Q_7(c-b)$, and $Q_3(b-e)$, where Q_7 and Q_3 are quantization functions taking on 7 and 3 possible values, respectively. We describe Q_7 by the 3 bits it produces and Q_3 by the 2 bits it produces. Specifically, we have chosen

$$Q_7(n) = \begin{cases} 011 & \text{if } n \geq 13 \\ 010 & \text{if } 5 \leq n \leq 12 \\ 001 & \text{if } 2 \leq n \leq 4 \\ 000 & \text{if } -1 \leq n \leq 1 \\ 101 & \text{if } -4 \leq n \leq -2 \\ 110 & \text{if } -12 \leq n \leq -5 \\ 111 & \text{if } n \leq -13 \end{cases}$$

and

$$Q_3(n) = \begin{cases} 01 & \text{if } n \geq 6 \\ 00 & \text{if } -5 \leq n \leq 5 \\ 11 & \text{if } n \leq -6 \end{cases}$$

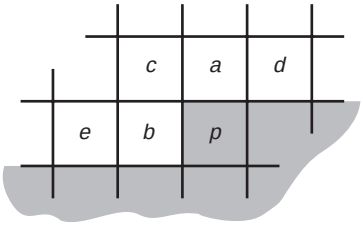

Fig. 1. The labeling of pixels relative to the current pixel, *p*. The shaded portion represents pixels not yet encoded.

We interpret the values taken on by Q_7 and Q_3 as integers in sign-magnitude form; e.g., 110 can be read as "negative 10_2" or -2. Note the number of quantization levels in Q_7 was chosen to allow a 3 bit representation; this differs from the 9 quantization levels for the corresponding quantizer in LOCO-I.

The context is based on the binary concatenation $(Q_7(a-c), Q_7(d-a), Q_7(c-b), Q_3(b-e))$. When we view $Q_7(a-c)$, $Q_7(d-a)$, $Q_7(c-b)$, and $Q_3(b-e)$ in sign-magnitude form, the context can be interpreted as an ordered quadruple of integers. In order to reduce the number of contexts, a context with some quadruple (c_1, c_2, c_3, c_4) is merged with the context with quadruple $(-c_1, -c_2, -c_3, -c_4)$. This is accomplished by inverting the signs of $Q_7(a-c)$, $Q_7(d-a)$, $Q_7(c-b)$, and $Q_3(b-e)$ if the first nonzero value of these is negative. For example, $(101, 010, 000, 11)$ is inverted to give $(001, 110, 000, 01)$, and $(000, 110, 011, 00)$ is inverted to give $(000, 010, 111, 00)$. When inversion occurs, an "invert" flag is set so that the prediction residual (the difference between the estimated and actual pixel values) and the bias can also be inverted. When contexts are combined, the leading bit in the binary concatenation will always be 0 and thus can be dropped, allowing contexts to be described with 10 bits.

This combining of contexts is based on symmetry assumptions and motivated primarily by the fact that fewer contexts allow quicker adaptation to image statistics.

Near the left, right, and top edges of the image, some of the values a through e do not exist. We handle these cases by forming separate contexts in each of these situations using the quantized difference values that are available. The resulting regions, with the number of different contexts for the region, are shown in Fig. 2. All of the border contexts formed can be put within the 10 bit context format without reusing any of the values taken by non-border contexts. This is accomplished by using 100 or 10 (the negative 0 unused by Q_7 or Q_3) for unavailable quantization values. Specifically, when $y = 0$, the context is $(00, 100, 100, Q_3(b-e))$; when $x = 0$, the context is $(00, Q_7(d-a), 100, 10)$; when $x = 1$, the context is $(Q_7(a-c), Q_7(d-a), Q_7(c-b), 10)$; and when $x = \texttt{wd}-1$, the context is $(Q_7(a-c), 100, Q_7(c-b), Q_3(b-e))$.

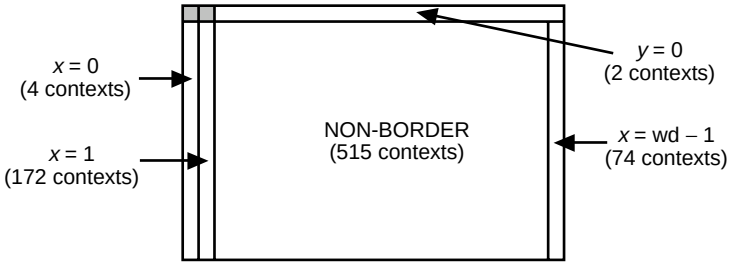

Fig. 2. Division of an image into regions with separate contexts.

We can enumerate the possible context values by representing them as "patterns" in ordered quadruple form, nominally corresponding to $(Q_7(a-c), Q_7(d-a), Q_7(c-b), Q_3(b-e))$, where we use "u" to denote an unavailable value, "+" to denote a positive value, and "$\cdot$" to denote any value. The result is given in Table 2. It can be seen that 767 of the 1024 possible 10 bit strings represent valid contexts. Since the 767 bit strings used as contexts are interspersed among the unused strings, we simply reserve all 1024 addresses for context data storage.

The benefit of having many contexts for edge regions of an image is necessarily small, since edge regions make up a small proportion of the image area; however, our context framework accommodates the extra contexts with little effort.

Associated with each context are 32 bits of memory to store data associated with the context. These data consist of 6 bits for a count of times the context occurs; 13 bits for the sum of the magnitude of the residuals encoded; 8 bits for the (signed) sum of the residuals encoded; and 5 bits for the bias associated with the context.

Table 2. Enumeration of contexts.

Location	Pattern	Number	Total
Non-border	$(+, \cdot, \cdot, \cdot)$	$3 \times 7 \times 7 \times 3 = 441$	
	$(0, +, \cdot, \cdot)$	$3 \times 7 \times 3 = 63$	
	$(0, 0, +, \cdot)$	$3 \times 3 = 9$	
	$(0, 0, 0, +)$	1	
	$(0, 0, 0, 0)$	1	515
$y = 0$	$(\mathrm{u}, \mathrm{u}, \mathrm{u}, +)$	1	
	$(\mathrm{u}, \mathrm{u}, \mathrm{u}, 0)$	1	2
$x = 0$	$(\mathrm{u}, +, \mathrm{u}, \mathrm{u})$	3	
	$(\mathrm{u}, 0, \mathrm{u}, \mathrm{u})$	1	4
$x = 1$	$(+, \cdot, \cdot, \mathrm{u})$	$3 \times 7 \times 7 = 147$	
	$(0, +, \cdot, \mathrm{u})$	$3 \times 7 = 21$	
	$(0, 0, +, \mathrm{u})$	3	
	$(0, 0, 0, \mathrm{u})$	1	172
$x = \mathrm{wd} - 1$	$(+, \mathrm{u}, \cdot, \cdot)$	$3 \times 7 \times 3 = 63$	
	$(0, \mathrm{u}, +, \cdot)$	$3 \times 3 = 9$	
	$(0, \mathrm{u}, 0, +)$	1	
	$(0, \mathrm{u}, 0, 0)$	1	74
Grand total			767

B. Estimation

The second step in the main encoding loop is to estimate the value of the pixel to be encoded. A more accurate estimate will produce a smaller, and thus more compressible, residual. In FPGA LOCO (and other versions of LOCO), a preliminary estimate is computed with a fixed (i.e., non-adaptive) estimator, and the adaptively computed bias is added to form the final estimate.

In this discussion, we let $\hat{p}$ denote the preliminary estimate of the current pixel p. As in [6], $\hat{p}$ is computed as the median of the three values a, b, and $a + b - c$ or, equivalently,

$$\hat{p} = \begin{cases} \min(a, b) & \text{if } c \geq \max(a, b) \\ \max(a, b) & \text{if } c \leq \min(a, b) \\ a + b - c & \text{otherwise} \end{cases}$$

Part of the motivation for this estimator is to serve as a primitive edge detector. Note that, for images without many sharp edges, a small improvement might be obtained by using an estimator (such as $a + b - c$) that is well-suited to smooth images. Similarly, for noisy images, it would be desirable to use a predictor that is better at averaging out noise values.

The final estimate of a pixel is obtained by adding the bias value (or its negative, if the invert flag is set) to the initial estimate. The bias value attempts to adaptively fine-tune the fixed predictor.

The addition of the bias could put the estimate outside the range of 0 to 255; therefore, the estimate is clipped to this range.

C. Encoding the Residual

After the encoder determines the estimate of the current pixel value, the difference between the actual pixel value and the estimate is losslessly encoded. (If the invert flag is set, the negative of this difference is encoded instead.) We denote this value by ϵ.

Although ϵ can take on values from -255 to 255, only 256 of these values are possible for a given pixel estimate. We eliminate the unused values by mapping ϵ to the range -128 to 127, accomplished by simply truncating the two's complement representation of ϵ to 8 bits, and interpreting the resulting number as a two's complement 8 bit integer, referred to as ϵ'.

The ϵ' values have a distribution that is usually approximately two-sided geometric. We map ϵ' to $M(\epsilon')$ to get a quantity with a distribution that is approximately (one-sided) geometric, with the range from 0 to 255. As in [6], this is accomplished with the transformation

$$M(\epsilon') = \begin{cases} 2\epsilon' & \text{if } \epsilon' \geq 0 \\ -2\epsilon' - 1 & \text{if } \epsilon' < 0 \end{cases}$$

Note that this transformation can be carried out efficiently with bit-manipulation operations (assuming two's complement form for ϵ'); with "$\ll$" as the left shift operation and "~" as the bitwise not operation, $2\epsilon'$ equals $\epsilon' \ll 1$ and $-2\epsilon' - 1$ equals $\sim(\epsilon' \ll 1)$.

The original motivations for this choice of mappings from ϵ to ϵ' to $M(\epsilon')$ are simplicity and a possible advantage for images with sharp transitions; however, the latter is not a significant factor with natural images. Mappings such as those suggested in [5] may give a small improvement.

Golomb codes [2] are simple entropy codes that are well-suited to encoding quantities with distributions that are approximately geometric and, thus, are a logical choice for encoding $M(\epsilon')$. The Golomb code with parameter k (corresponding to $m = 2^k$ in [2]) encodes a value v by putting the k low-order bits of v directly into the output bit stream, then encoding the remaining value $\lfloor v/2^k \rfloor$ with a unary encoding; that is, sending $\lfloor v/2^k \rfloor$ '0' bits followed by a '1' bit. Thus, v is encoded with $k + 1 + \lfloor v/2^k \rfloor$ bits.

The value of k for this encoding is determined from previous values of ϵ' occurring within the same context. Specifically, if A is the sum of the magnitudes of these previous values and N is the number of samples, then k is given by

$$k = \min\{i \mid 2^i N \geq A\}$$

This is from [6], which also contains reasons supporting this choice. The resulting k will be in the range 0 to 7.

D. Updating Context Data

After the encoding operation takes place, the data associated with the context are updated. The goal of this process is to ensure that later values with the same context are encoded efficiently.

As previously mentioned, there are 32 bits of data associated with each context:

(1) Occurrence count (`count`, 6 bit unsigned integer)

(2) Magnitude sum of residuals (`msum`, 13 bit unsigned integer)

(3) Sum of residuals (`rsum`, 8 bit signed integer)

(4) Bias value (`bias`, 5 bit signed integer)

For all contexts, the values of these data before compression are set identically: `count` is 2, `msum` is 12, `rsum` is 0, and `bias` is 0. A small compression improvement might be obtained by carefully choosing initial values to minimize the typical time to adapt to an image.

Note that during the update process `count`, `rsum`, and `bias` may take on values outside their usual ranges (e.g., we temporarily allow `count` to be 64 even though it is stored as a 6 bit integer).

The following steps occur during the update process:

(1) Increment `count` by 1.

(2) Add ϵ' to `rsum`. If the new `rsum` is greater than 0, `bias` is increased by 1 (unless it already equals its maximum value, 15) and `rsum` is decreased by `count`; if the new `rsum` is less than $-$`count`, `bias` is decreased by 1 (unless it equals its minimum value, -16) and `rsum` is increased by `count`.

(3) Clip `rsum` to the range $[-128, 127]$.

(4) Increase `msum` by $|\epsilon'|$.

(5) If `count` is 64, then `count`, `msum`, and `rsum` are all divided by 2 (accomplished with a right shift, with sign extension in the case of `rsum`).

Note that adjustment to `bias` that occurs in Step (2) attempts to keep the average ϵ' value between -1 and 0, rather than centered around 0. This is because the asymmetry of the mapping $M(\epsilon')$ results in an encoding that is better matched to a distribution on ϵ' that is centered around $-1/2$ when $k > 0$ [6]. When $k = 0$, the encoding is matched to a distribution centered at $-2/9$; however, we do not attempt to account for this case separately.

E. Run-Length Encoding

We did not implement the run-length encoding of [6] (referred to as embedded alphabet extension). The run-length encoding efficiently encodes regions of an image containing identical pixel values. Although this feature does not slow a software implementation, it would greatly complicate a hardware implementation. In any case, natural images usually do not contain such regions, although a possible exception is images with regions of uniformly black sky (and then only if the regions have *exactly* identical pixel values). Without the run-length encoding, our implementation cannot compress to less than 1 bit/pixel.

III. FPGA Implementation

A. Physical Setup

Our FPGA LOCO system is implemented using a board commercially available from Associated Professional Systems (APS). Specifically, the board is the APS-V240 with a Xilinx Virtex XCV50 FPGA. The APS-V240 is interfaced to a computer that we refer to as a "PC."

The physical setup is schematically represented in Fig. 3. The APS-V240 uses a PC104 interface, and so is connected to the PC's Industrial Standards Architecture (ISA) interface with an adaptor card. An optional 256K $\times$ 18 zero bus turnaround (ZBT) static random access memory (SRAM) is installed on the APS-V240.[3] It is convenient to be able to access all 32 bits of data for a context at once, so a second identical SRAM was installed on a daughter card. The daughter card connects to the APS-V240 with two ribbon cables: one to the original SRAM connector and a second to a connector with available XCV50 input/output (I/O) pins. These pins were designated for the extended 18 bit width SRAM. The onboard

[3] The notation 256K $\times$ 18 describes memory with 256 $\times$ 1024 addresses, each allowing storage of 18 bits.

SRAM and the daughter card SRAM together form 256K × 36 SRAM space available to the XCV50, of which 32 bits of width were used. Figure 4 contains photographs of the hardware.

B. Data Flow

Figure 5 illustrates the major data pathways of the implementation. The PC runs software that reads the image, transmits it to the FPGA (one pixel at a time), and receives the compressed image data (one byte at a time). Several registers in PC I/O space were implemented in the FPGA. These include a control register, an input byte register, and an output byte register.

The SRAM external to the FPGA is used to store the context data and to store pixel values. Because the algorithm transmits the output from a given pixel before receiving the next pixel, it is only necessary to store those pixels needed to recreate future pixel contexts. In this case, enough memory for one row of pixels is needed. Note that the total amount of SRAM needed by the algorithm is 1K × 8 for the pixel memory (assuming a maximum image width of 1024) and 1K × 32 for the context memory.

The transmission of pixels from the software to the FPGA, and of bytes from the FPGA to the software, is done one byte at a time using software–hardware handshaking.

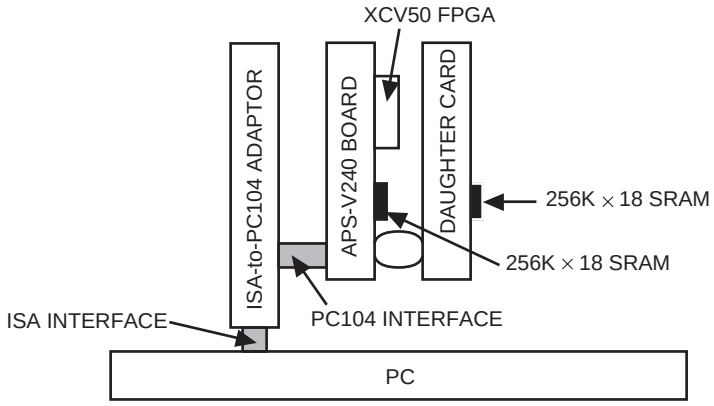

Fig. 3. Schematic representation of the physical setup.

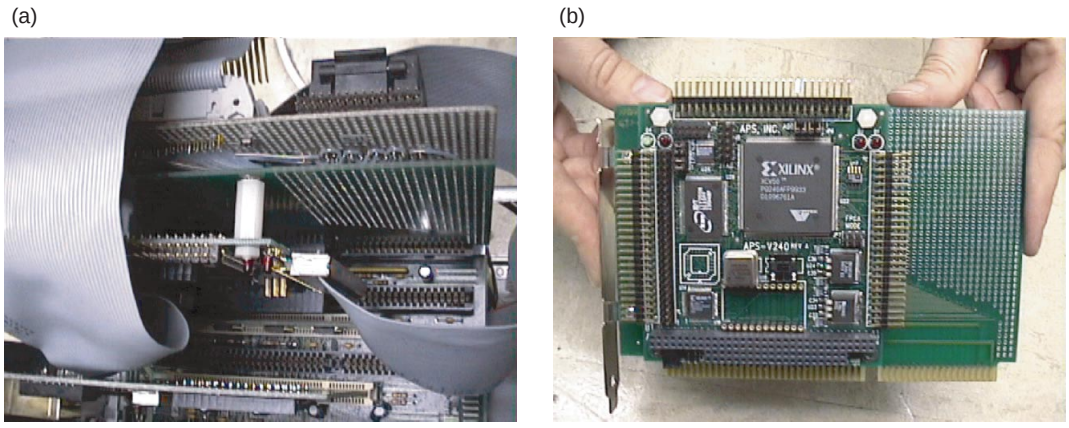

Fig. 4. Photographs of the hardware: (a) the inside of the PC containing the APS-V240 board, the shorter card near the center of the photo (above the APS-V240 are first the adaptor card and then the daughter card with the expansion SRAM), and (b) the APS-V240 board with the XCV50 FPGA with the ribbon cables removed.

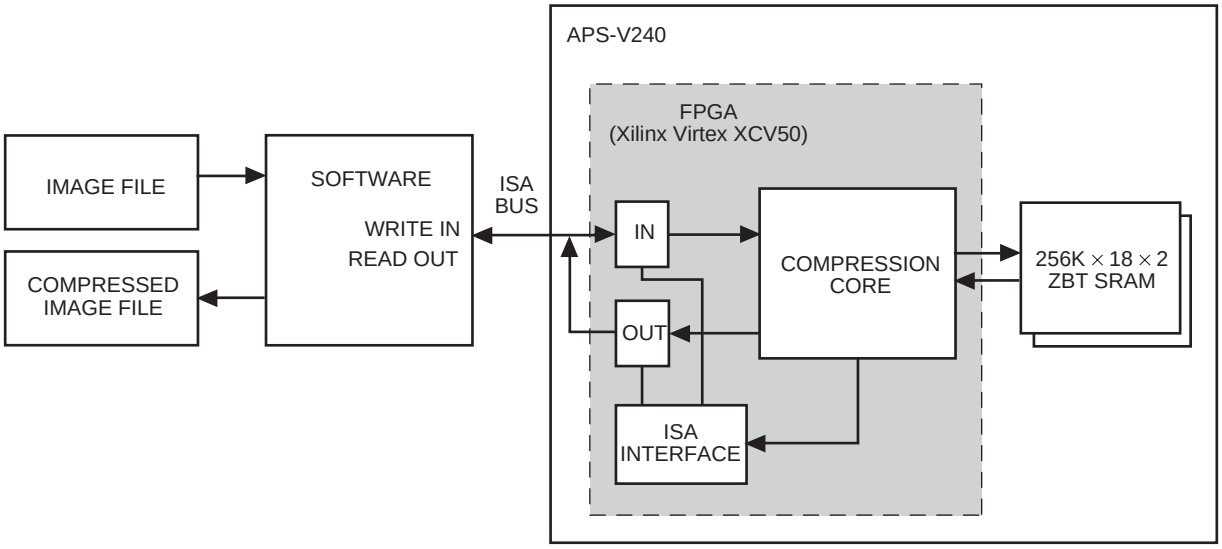

Fig. 5. The flow of data among components.

C. State Machine Implementation

The state machine implemented in the FPGA has five states that form the main loop, and a handful of additional states such as initialize states, wait states, and termination sequencing states. The overwhelming majority of active compression time is spent in the five states of the main loop.

The states S1 through S5 of the main loop are shown in Fig. 6, along with some of the data transfers that occur. Each state occurs for each pixel (except the first two pixels of the image). In state S1, a new pixel is received from the PC, and a pixel from the previous line (part of the context) is loaded from the SRAM. The read operation from SRAM requires three clock cycles, so state S1 is held for two additional clock cycles. In state S2, the new pixel is stored in the SRAM (the write is held for two additional clock cycles, but the main state machine does not pause for this). By state S3, the context has been computed so the context data can be retrieved from the SRAM. In state S4, the Golomb encoding is begun in a concurrent state machine. Finally, in state S5, the updated context data are stored in the SRAM, and a pause occurs until any bytes that were generated by the Golomb encoding are transmitted back to the PC.

Individual input pixels occasionally produce multiple output bytes (the worst case is 32 output bytes). The concurrent state machine that generates these bytes puts them into a first-in, first-out (FIFO) buffer. The contents of the FIFO are transmitted to the PC one byte at a time. The FIFO is implemented in the FPGA.

IV. Performance and Conclusions

Our implementation was tested with several images, and the output was verified to decompress successfully in each case.

The clock speed used in the implementation is 12 MHz. The main loop takes nine clock cycles to complete (unless the pixel produces multiple output bytes, which is rare), so this translates to a core speed of about 1.33 Mpixels/second. Our complete test system runs slower due to software delays during the transfer of bytes to and from the PC. (The speed actually realized was 260 kpixels/second; logic analyzer traces showed the compression core spent about 80 percent of its time waiting on the software-handled I/O.)

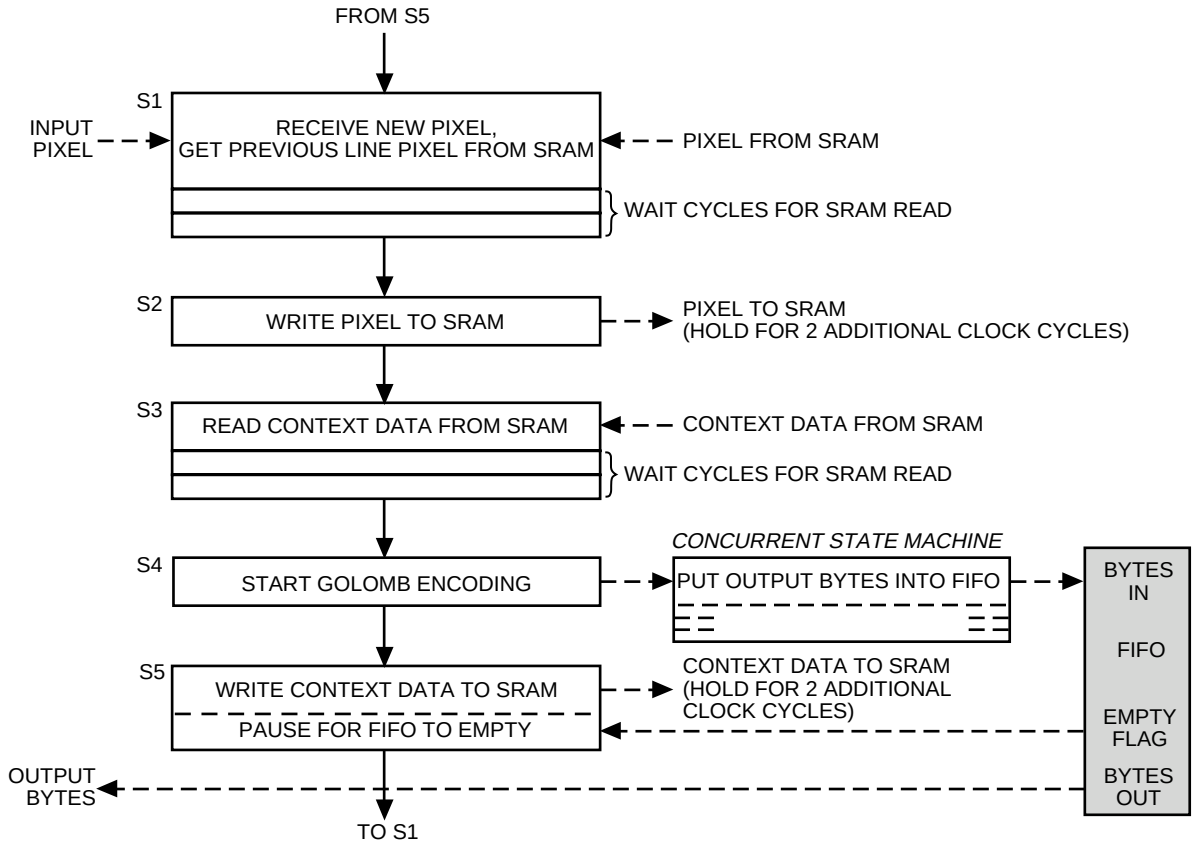

Fig. 6. States and major data transfers in the primary loop.

In a spacecraft implementation, it would be relatively straightforward to reduce or eliminate the I/O delays. For example, the spacecraft may have a bus controller that facilitates such data transfers; input and output buffers could be added to the compressor design; or a separate digital signal processor (DSP) could be used to handle data transfers. In any case, a throughput of approximately 1.33 Mpixels/second is realizable from our core design.

It is worth noting that a radiation-hardened version of the Xilinx Virtex XCV300 FPGA is available as the Xilinx Virtex XQVR300. The XCV300 has a component count about six times as large as that of the XCV50. Thus, conversion of the design to radiation-hardened hardware could be accomplished without redesigning the compression core. Since the amount of memory used by the algorithm is relatively low (about 5 kbytes), such memory probably could be provided internally in the FPGA containing the compression core, eliminating the need for external memory.

If still faster compression is desired, a pipelined architecture could be designed. Although difficulties arise from the sequential nature of the use of context statistics, a design achieving significantly improved speed (with the same clock rate as our current implementation) would be possible. Much of the current design would need to be redone to accomplish this, and an FPGA with a much higher component count (perhaps such as the Xilinx Virtex XCV1000) would be needed.

The algorithm and implementation can easily be adapted to handle modifications in various parameters. In particular, pixel bit depths greater than 8 could be accommodated. Some parameters of the design could be made controllable with the interface to allow greater flexibility. Depending on the downlink channel characteristics, it may be desirable to incorporate some method of limiting the effects of

channel errors. This would probably consist of dividing an image up into smaller images and need not be part of the hardware.

Overall, our implementation represents significant progress toward producing space-qualified lossless image-compression hardware with improved compression performance as compared with currently available hardware.

References

[1] Consultative Committee for Space Data Systems, "CCSDS Recommendation for Lossless Data Compression," CCSDS 121.0-B-1, Blue Book, issue 1, May 1997. http://www.ccsds.org/blue_books.html

[2] S. W. Golomb, "Run-Length Encodings," *IEEE Transactions on Information Theory*, vol. IT-12, no. 3, pp. 399–401, July 1966.

[3] *Information Technology—Lossless and Near-Lossless Compression of Continuous-Tone Still Images*, ISO/IEC 14495-1, ITU Recommendation T.87, 1999.

[4] M. Rabbani and P. Jones, *Digital Image Compression Techniques*, Bellingham, Washington: SPIE Publications, 1991.

[5] R. F. Rice, *Some Practical Universal Noiseless Coding Techniques, Part III, Module PSI14,K+*, JPL Publication 91-3, Jet Propulsion Laboratory, Pasadena, California, November 1991.

[6] M. J. Weinberger, G. Seroussi, and G. Sapiro, "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm," *Proc. of the 1996 Data Compression Conference (DCC '96)*, Snowbird, Utah, pp. 141–149, March 1996.

[7] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1309–1324, August 2000.