Hardware Implementation of Block GC3 Lossless Compression Algorithm for Direct-Write Lithography Systems

Hsin-I Liu, Brian Richards, Avideh Zakhor, and Borivoje Nikolic

Dept. of Electrical Engineering and Computer Sciences,
University of California, Berkeley

ABSTRACT

Future lithography systems must produce chips with smaller feature sizes, while maintaining throughput comparable to today's optical lithography systems. This places stringent data handling requirements on the design of any direct-write maskless system. To achieve the throughput of one wafer layer per minute with a direct-write maskless lithography system, using 22 nm pixels for 45 nm technology, a data rate of 12 Tb/s is required. In recent years, we have developed a datapath architecture for direct-write lithography systems, and have shown that lossless compression plays a key role in reducing throughput requirements of such systems. Our approach integrates a low complexity hardware-based decoder with the writers, in order to decode a compressed data layer in real time on the fly. In doing so, we have developed a spectrum of lossless compression algorithms for integrated circuit rasterized layout data to provide a tradeoff between compression efficiency and hardware complexity, the most promising of which is Block Golomb Context Copy Coding (Block GC3). In this paper, we present the FPGA synthesis and emulation results for the Block GC3 decoder. For one Block GC3 decoder, 3233 slice flip-flops and 3086 4-input LUTs are utilized in a Xilinx Virtex II Pro 70 FPGA, which corresponds to 4% of its resources, along with 1.7 KB of internal memory. The system runs at 100 MHz clock rate, with the overall output rate of 495 Mb/s for a single decoder. In addition to the decoder implementation results, we discuss other hardware implementation issues for the writer system data path, including on-chip input/output buffering, error propagation control, and input data stream packaging. This hardware data path implementation is independent of the writer systems or data link types, and can be integrated with arbitrary direct-write lithography systems.

Keywords: Block GC3, lossless compression, hardware implementation, data path, direct-write lithography.

1. INTRODUCTION

Future lithography systems must produce chips with smaller feature sizes, while maintaining throughput comparable to today's optical lithography systems. This places stringent data handling requirements on the design of any direct-write maskless system. Optical projection systems use a mask to project the entire chip pattern in one flash. An entire wafer can then be written in a few hundreds of such flashes. To be competitive with today's optical lithography systems, direct-write maskless lithography needs to achieve throughput of one wafer layer per minute. In addition, to achieve the required 1nm edge placement with 22 nm pixels in 45 nm technology, a 5-bit per pixel data representation is needed. Combining these together, the data rate requirement for a maskless lithography system is about 12 Tb/s. To achieve such a data rate, we have proposed a data path architecture shown in Fig. 1[1]. In this architecture, rasterized, flattened layouts of an integrated circuit (IC) are compressed and stored in a mass storage system. The compressed layouts are then transferred to the processor board with enough memory to store one layer at a time. This board will then transfer the compressed layout to the writer chip, composed of a large number of decoders and actual writing elements. The outputs of the decoders correspond to uncompressed layout data, and are fed into D/A converters driving the writing elements such as a micro-mirror array or E-beam writers.

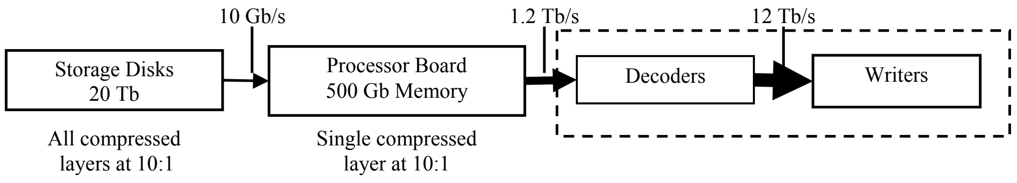

10 Gb/s
1.2 Tb/s
12 Tb/s
Storage Disks 20 Tb
Processor Board 500 Gb Memory
Decoders
Writers
All compressed layers at 10:1
Single compressed layer at 10:1

Fig. 1 The data-delivery path of the direct-write systems.

In the proposed data-delivery path, compression is needed to minimize the transfer rate between the processor board and the writer chip, and also to minimize the required disk space to store the layout data. Since there are a large number of decoders operating in parallel on the writer chip to achieve the projected output data rate, an important requirement for any compression algorithm is to have an extremely low decompression complexity. To this end, we have proposed a series of lossless layout compression algorithms for flattened, rasterized data. In particular, Block Golomb Context Copy Coding (Block GC3) has been shown to outperform all existing techniques such as BZIP2, 2D-LZ, and LZ77 in terms of compression efficiency, especially under limited decoder buffer size and hardware complexity, as required for hardware implementation[2][3].

In this paper, we present FPGA implementation of the Block GC3 decoder. In Section 2, we review encoding/decoding algorithm of Block GC3. In Section 3, we present implementation of Block GC3 decoder design architecture, further modifications of the algorithm to reduce complexity, and its FPGA synthesis and emulation results. In Section 4, we discuss integration of the decoder into the writer system, data buffering, error propagation control, and data packaging. Conclusions and future work are presented in Section 5.

2. OVERVIEW OF BLOCK GC3

There are two prominent characteristics for rasterized, flatten layout images: Manhattan shape of the patterns, and repetitiveness of the patterns. To compress the images efficiently, we must utilize both characteristics by either predicting the pixel value from its neighborhood to preserve the horizontal and vertical edges of the patterns, or copying the patterns directly from the buffer to exploit the repetition of the data. The family of Context Copy Combinatorial Code (C4) compression algorithms combines those two techniques to achieve lossless compression for the rasterized layout images. Among those algorithms, Block GC3 has been shown to be a suitable candidate for hardware implementation [1]-[3].

Fig. 2 shows a high-level block diagram of the Block GC3 encoder and decoder for flattened, rasterized gray-level layout images. The detailed description of the algorithm can be found in [3]. First, "Find best copy distance" block finds the best matching pattern in the buffer for direct-copying on the macroblock basis. Then, for each macroblock, the copied pattern is compared with the 3-pixel based predict pattern, and the one with fewest mismatch is selected. This generates a segmentation map, indicating for each macroblock, whether copy or predict technique is applied, and where to copy the pattern from. Based on the segmentation map, the result is compared to the actual value in the layout image. Correct pixel values are assigned a "0" and incorrect values are assigned a "1". The pixel error location is compressed losslessly by the Golomb run-length code, and the corresponding pixel error value is compressed by the Huffman code. These compressed bit streams are transmitted to the decoder, along with the segmentation map.

The decoder mirrors the encoder, but skips the complex process necessary to determine the segmentation map, obtained from the encoder. The Golomb run-length decoder decompresses the error location bits from the encoder. As specified by the segmentation map, the Predict/Copy block estimates each pixel value, either by copying or by prediction. If the error location bit is "0", the pixel value is correct; otherwise, it is incorrect, and must be replaced by the actual pixel value decoded from Huffman decoder. There is no segmentation performed in the Block GC3 decoder, so it is considerably simpler to implement than the encoder. In practice, the encoding process is done offline, and the resulting compressed data is stored for writing process, whereas the decompression is done on-the-fly in hardware, integrated with the writer system. Therefore, the decoder must have low complexity implementation; Block GC3 decoder satisfies this criterion.

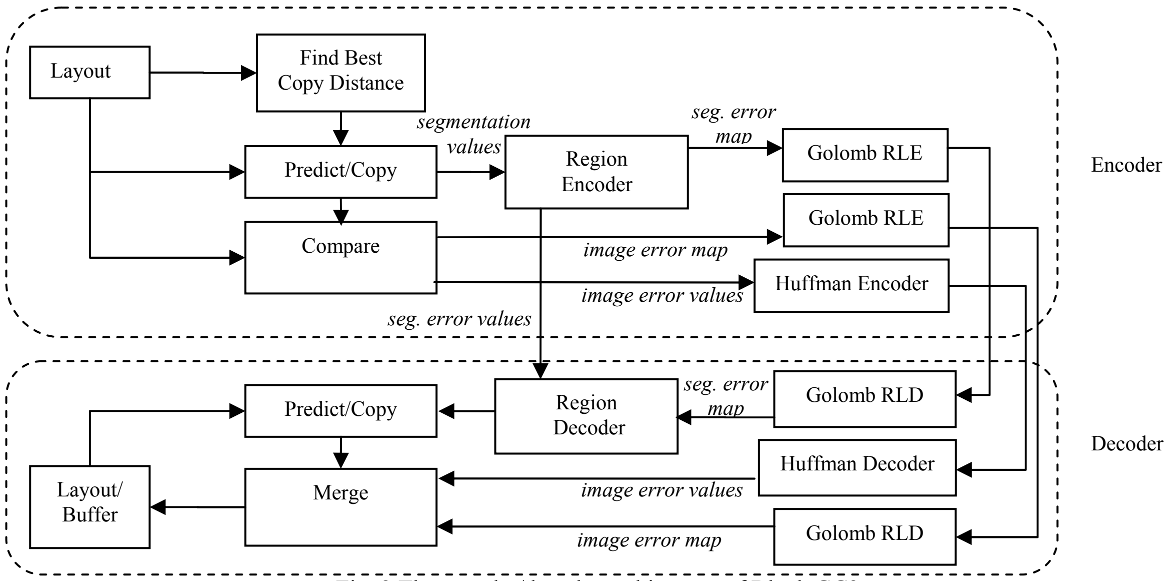


Fig. 2 The encoder/decoder architecture of Block GC3.

3. HARDWARE IMPLEMENTATION OF BLOCK GC3 DECODER

For the decoder to be used in a maskless lithography data path, it must be implemented as a custom digital circuit and included on the same chip with the writer array. In addition, to achieve a system with high level of parallelism, the decoder must have the data-flow architecture and high throughput. By analyzing the functional blocks of the Block GC3 algorithms, we devise the data-flow architecture for the decoder.

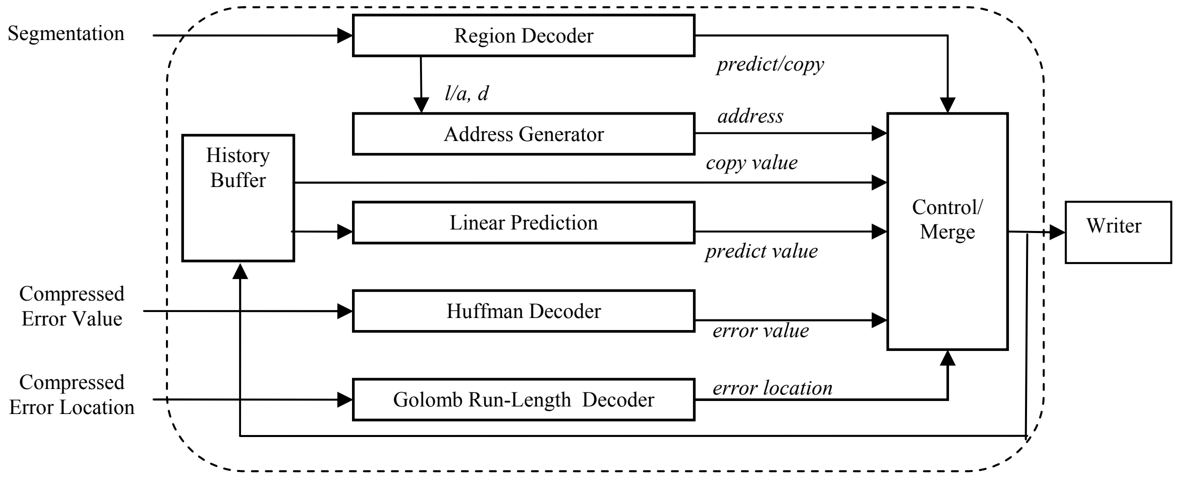


Fig. 3 Functional block diagram of Block GC3 decoder

The block diagram of Block GC3 decoder is shown in Fig. 3. There are three main inputs: segmentation, compressed error location, and compressed error value. The segmentation is fed into the Region Decoder, generating a segmentation map as needed by the decoding process. Using this map, the decoded predict/copy property of each pixel can be used to select between the predict value from Linear Prediction and the copy value from History Buffer in the Control/Merge stage by a multiplexer (MUX). The compressed pixel error location is decoded by Golomb run-length decoder, resulting in an error location map, which indicates the locations of invalid predict/copy pixels. In the decoder, this map contributes to another control signal in the Control/Merge stage to select the final output pixel value from either predict/copy value or the decompressed error value generated by Huffman decoder. The output data is written back to History Buffer for future usage, either for linear prediction or for copying, where the appropriate access position in the buffer is generated by the Address Generator. All the decoding operations are combinations of basic logic and arithmetic operations, such as selection, addition, and subtraction. By applying the tradeoffs described in [3], the total amount of memory needed in a single Block C4 decoder is about 1.7 KB; this can be implemented using hardware memory, either on-chip SRAM for ASIC implementation or Block memory for FPGA.

In this section, we present the FPGA implementation of Block GC3 decoder. Before doing so, however, we need to modify the algorithm in order to reduce its hardware complexity. In Section 3.1, the design of Golomb run-length decoder is refined. In Section 3.2, the Huffman code is modified to reduce the hardware complexity. The FPGA synthesis result is presented in Section 3.3.

3.1 Selectable bucket size for Golomb run-length decoder

In previous work, we have discussed the hardware design of Golomb run-length decoder. Due to varying image error rates of the layout images over different layers, the bucket size B of the Golomb run-length decoder needs to be varied from image to image in order to achieve the best compression efficiency [3]. In terms of the hardware design, this implies the width of the data bus has to match $\log_2 B_{\max}$, where $B_{\max}$ is the greatest bucket size, even though $B_{\max}$ may hardly be used. Fig. 4 shows the block diagram of the Golomb run-length decoder, which reads in the compressed binary Golomb code through a barrel shifter and generates the decompressed binary stream of error locations using the counter and comparators with various bucket sizes. With variable bucket sizes, the arrows inside the dashed box, indicating the data buses inside the Golomb run-length decoder, have to be the width of $\log_2 B_{\max}$ to achieve the correct decoding. As a result, in order to fit large bucket sizes, some bus wires are allocated but seldom used, resulting in a waste of resources. To minimize such a waste, we limit the bucket size to be smaller than 64, which corresponds to the 6-bit data buses in the hardware.

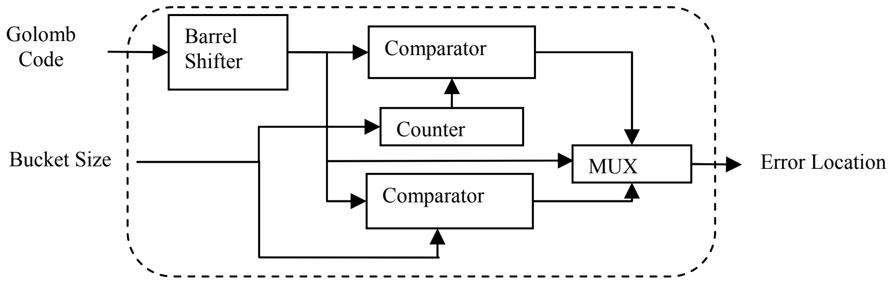


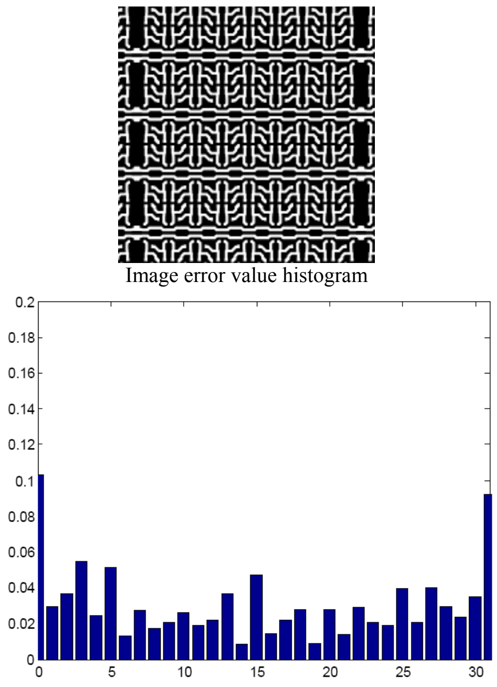
Fig. 4 Block diagram of the Golomb run-length decoder.

Such a design choice adversely affects the compression efficiency by lowering its upper bound. For example, the compression ratio for a black image goes from 1280 to 316.8, and other easily compressible images will also suffer from lower compression efficiencies. However, those images are not bottleneck of the data path; based on the compression ratio distribution reported in [4], changing the compression efficiency of those images does not significantly affect the overall compression performance of Block GC3. On the other hand, by limiting the bucket size of the Golomb run-length decoder, the hardware resources can be saved, and the routing complexity of the extremely wide data buses can be reduced.

3.2 Fixed codeword for Huffman decoder

Similar to other entropy codes, Huffman code adapts its codeword according to the statistics of the input data stream to achieve the highest compression efficiency. In general, the codeword is either derived from the input data itself, or by the training data with the same statistics. In both scenarios, the code table in the Huffman decoder has to be updated to reflect the statistic changes of the input data stream. For layout images, this corresponds to either different layers or different parts of the layout. However, the updating of the code table requires an additional data stream to be transmitted from encoder to the decoder. Moreover, the update of the code table has to be done in the background such that the current decoding is not affected. Consequently, more internal buffers are introduced, and additional data is transmitted over the data path.

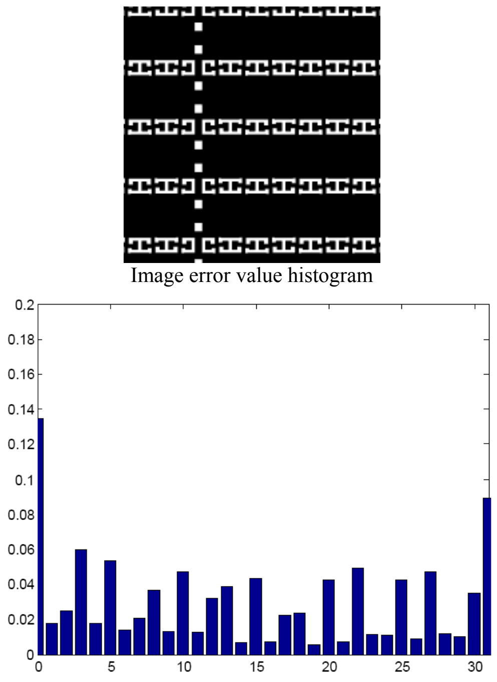
Close examination of the statistics of input data stream, namely, the image error values in Fig. 2, reveals that the update can be avoided. Fig. 5 shows two layout images with its image error value histogram and a selected numbers of Huffman codewords. The left side shows the poly layer and the right one the n-active layer. Although the layout images seem different, the histograms are somewhat similar, and so are the codewords. More specifically, the lengths of the codewords for the same error value are almost identical, except for those on the boundaries and those with low probability of occurrence. The similarity can be explained by the way we generate the error values: After copy and predict techniques are applied, the error pixels are mainly located at the edges of the features. As a result, the error values for different images are likely to have similar probability distribution, even though the total number of error values varies from image to image. Based on this observation, we can use a fixed Huffman codeword to compress all the images without losing too much compression efficiency, in exchange for no code table updating for the decoder. Table 1 shows the comparison of the compression efficiency between the fixed Huffman code table and adaptive Huffman code table over several 1024×1024 5-bit gray-level images. The compression loss of the fixed code table is less than 1%, and is lower for the low compression ratio images. Therefore, in hardware implementation, we opt to use a fixed Huffman code table to compress all the layout images.



Huffman Codeword

Value	Codeword
0	111
31	1101
3	1010
5	1011
15	1100
27	10000
25	01111

(a)



Huffman Codeword

Value	Codeword
0	111
31	1101
3	1000
5	1001
15	01010
27	1100
25	01110

(b)

Fig. 5 Image error value statistics and Huffman codeword comparison, for (a) poly layer and (b) n-active layer.

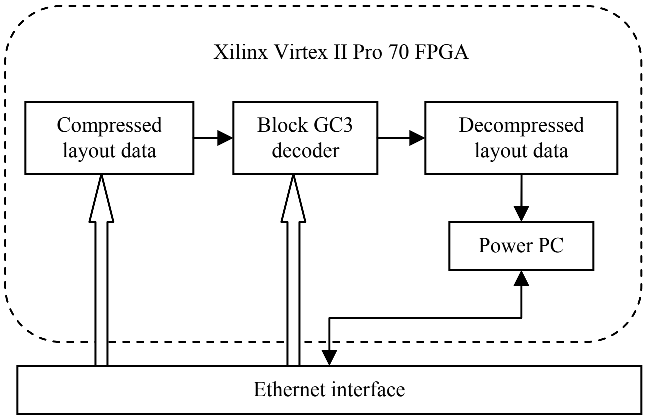
Table 1 Compression efficiency comparison between different Huffman code tables.

Layout image	Compression ratio		Efficiency loss (%)
	Adaptive Huffman code table	Fixed Huffman code table	
Metal 1	13.06	12.97	0.70
Metal 2	29.81	29.89	0.74
N-active	38.12	38.01	0.28
Poly	9.89	9.87	0.17

3.3 FPGA emulation results

After applying the aforementioned modifications of the algorithm, we implement the Block GC3 decoder in Simulink-based design flow, then synthesize and map onto the FPGA. We use Xilinx Virtex II Pro 70 FPGA, part of the Berkeley Emulation Engine 2 (BEE2), as our test platform [5]. Fig. 6(a) shows the photo of the BEE2 system, and Fig. 6(b) shows the schematics of Block GC3 emulation architecture. BEE2 system consists of five FPGAs and the peripheral circuitries, including Ethernet connections, which is the communication interface in our emulation. Since Block GC3 decoder is deliberately designed to be of low complexity, only one FPGA is utilized. Inside this FPGA, the design of Block GC3 decoder is synthesized and mapped, and the compressed layout data is stored into additional memory. After decoding, the decoded layout data is stored, and can be accessed by the user through the Power PC embedded in the FPGA, under the BORPH operating system [6]. Using this architecture, we can easily modify the design and verify its functionality.

(a)



(b)

Fig. 6 (a) The BEE2 system [7]. (b) FPGA emulation architecture of Block GC3 decoder.

Table 2 shows the synthesis results of Block GC3 decoder. Only 3233 slice flip flops and 3086 4-input LUTs are used, which correspond to 4% of the overall FPGA resources. In addition, 36 Block RAMs are utilized, mainly to implement the 1.7 KB internal memory of Block GC3 decoder and I/O registers. The system is tested at 100 MHz clock rate, which acommedates the critical data path of the system after the synthesis.

Table 2 Synthesis summary of Block GC3 decoder

Device	Xilinx Virtex II Pro 70
Number of slice flip-flops	3,233 (4%)
Number of 4 input LUTs	3,086 (4%)
Number of block RAMs	36 (10%)
System clock rate	100 MHz
System throughput rate	0.99 (pixels/clock cycle)
System output data rate	495 Mb/s

In addition, through this empirical test, we find the internal buffer can be further reduced to 1 KB. Using the 2-row search range, the vertical copy can be fully replaced by prediction to achieve the same performance. In doing so, the data in the previous row is not needed, and the search range can be reduced to 1-row. This memory reduction may result in lower compression efficiency if the image is extremely hard to either predict or copy. However, for our test images, this never occurs.

By decompressing the actual layout images, we measure the actual throughput of the system. Unlike the previous estimate in [3], the actual system throughput is 0.99 pixels/clock cycle. The system only stalls in the transition from a copy region to a predict region, and in the practical scenarios, this only happens 1% of the time. Combining the 100 MHz clock rate, 0.99 system throughput, and 5 bit/pixel output data type, the system output rate is 495 Mb/s. In order to achieve the proposed 12 Tb/s data rate, 24,000 decoders are needed. By switching the implementation platform from FPGA to ASIC, the clock rate can be further improved, resulting in a higher output data rate for each decoder.

4. BLOCK GC3 DECODER DATAPATH

In order to integrate the Block GC3 decoder with the writer system, we also have to consider the datapath between the decoder and the rest of the system. This includes buffering the data from the I/O interface to the decoder, buffering the output of the decoder before it is fed into the writer, and packaging the input data stream so that multiple input data streams can share the same I/O interface. In addition, since the Block GC3 uses previous output data to either predict or generate the current pixel value, proper error control is needed to avoid the error propagation. These issues are discussed in this section.

4.1 Input/Output data buffering

In Block GC3 decoder, one bit in the input date stream can be decoded into multiple output pixel values, depending on the compression efficiency. In other words, the input data rate is potentially lower than the output data rate by the same compression ratio, resulting in a fewer number of input data links and lower power consumption by the I/O interface. In practice, this lower input data rate can only be achieved by buffering the input data on-chip before it is read by the decoder. However, this also implies additional internal buffers of the writer chip, which is what we are trying to avoid in the first place. In previous work, we have proposed the on-chip buffer to be of the size

$$\text{buffer size} = \frac{\text{image size}}{\text{comprssion ratio}} \tag{1}$$

which suggests the entire 1024×1024 compressed layout image to be stored in the memory before being decoded [4]. Assuming the compression ratio of 10, this corresponds to 64 KB of internal buffer. Even though this number is not substantial, considering hundreds of decoders running in parallel to achieve the projected output data rate, the scaled buffer size may not be affordable. In addition, this buffer size may be an overestimate since the writer system reads and writes the buffer simultaneously, in a first-in-first-out fashion. In this case, the buffer may only be completely full at the very beginning of the decoding process, resulting in a waste of the resources.

To circumvent the above problem, we propose to reduce the size of the input data buffer to

$$\text{buffer size} = a \times \text{image size} \times \left(\frac{1}{\text{compression ratio}} - \frac{1}{\text{input data rate}} \right) \tag{2}$$

Unlike (1), the buffer size in (2) is a function of both the input and output rate of the FIFO, and the size is reduced by taking the update speed into account. The constant *a*, which is slightly greater than 1, is introduced to ensure the FIFO will not be empty. For high compression ratio images, this buffer will always be almost full, since the input data rate is higher than the compression ratio, which corresponds to the output data rate. In this case, the input data link is on only when the FIFO is not full. On the other hand, for low compression ratio images, the FIFO is slowly drained to be empty; this is because its output data rate is higher than its input data rate, while the input data link is always on, running at the designed input data rate. In this architecture, after decomposing the rasterized layout into a series of images, we need to arrange the layout images so that not all low compression ratio images are clustered together, resulting in an empty input FIFO. The arrangement strategy for layout images was presented in [4], and can be performed at the encoding stage.

After decoding, output pixel values will be read by the writer devices. Depending on the writer systems, output data rate of Block GC3 may not be the same as the speed of the writer. Therefore, output buffer is also needed. Unlike input buffer, the size of the output buffer is determined by how many pixels the writer prints at the same time. Besides buffering output data of the decoder, the output buffer can also be used to synchronize multiple decoders in the data path. Even though each decoder outputs the data at a slightly different pace depending on the decompression process, once their output is buffered and read by the writer at the same time, the data from different decoders is synchronized properly.

4.2 Control of Error Propagation

In Block GC3 algorithm, both copy and predict schemes use the previous pixel values stored in the history buffer to generate the current pixel value. If the stored pixel value is altered during the read/write process of the history buffer, the error propagates to the remaining part of the image. To solve this problem, we have to consider error control strategies.

First, the history buffer has to be refreshed for every new image. Although Block GC3 algorithm is suitable for stream coding, i.e., using the history buffer of the previous image to code the current one, the error can also propagate from the previous image in the same fashion. Therefore, refreshing of the history buffer would confine the error to the image boundaries. By doing so, there would be some encoding overhead at the beginning of the images, and lower compression efficiency as compared to stream coding. However, considering the 1-row and 2-row buffer cases, the overhead and compression efficiency loss are negligible.

Besides setting up the boundaries of the images, we can further reduce error by applying error control code (ECC) to the history buffer. Hamming (7, 4) code is a simple error control code, and has been implemented in hardware in the literature [8][9]. In this code, 4 bits of data are coded with 3 extra parity bits to construct a 7-bit code. While decoding the Hamming code, one bit error in the code can be identified and corrected, and two bits error can be detected by the decoder. In the history buffer of Block GC3, we can apply the Hamming (7, 4) code to encode the four most significant bits of the 5-bit pixel value, resulting in a 8-bit code for every pixel, which can be stored into a typical memory design without wasting resources. While reading the pixel value from the history buffer, the single-bit error can be corrected. A schematic of possible history buffer design is shown in Fig. 7. Depending on the retention ability of the memory, this may effectively reduce the error propagation.

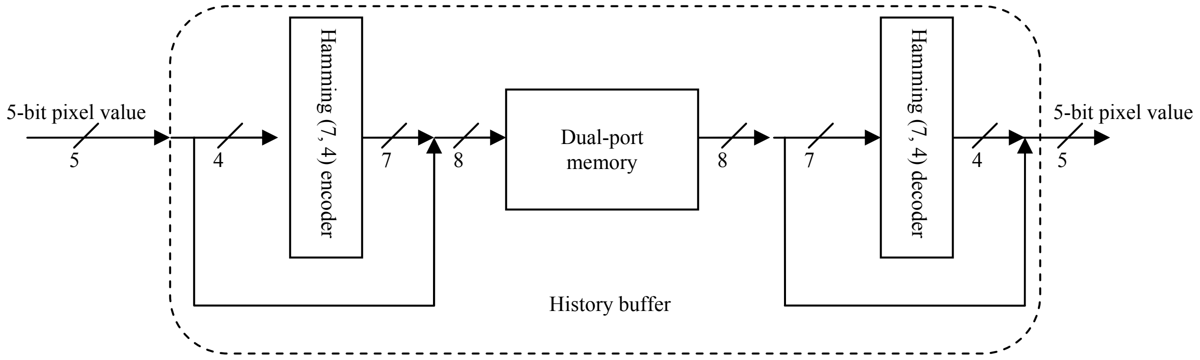


Fig. 7 The block diagram of the history buffer with ECC.

4.3 Data packing

In our FPGA implementation, all the compressed data streams are stored separately and sent to the region decoder, Golomb run-length decoder, and Huffman decoder of Block GC3 decoder simultaneously in order to demonstrate the data-flow type of decoding process, as shown in Fig. 3. In order to reduce the number of input data links, these data streams can be combined into one. However, not all the data streams are read at the same rate; for example, the segmentation information is needed at most per 64 pixels, whereas the compressed error location is read at most every 2 pixels. Therefore, in order to pack the input data stream, the Block GC3 encoder has to mimic the decoding process and arrange the data stream accordingly. This may introduce extra encoding overhead; however, since the decoding process is two to three orders of magnitude faster than the encoding, the impact is marginal. In addition, in the Block GC3 decoder, the input data stream for each block has to be further buffered to balance the data requests among different blocks, in case they send out the read requests at the same time. This extra buffer can be only several bytes and implemented in FIFO, but it is essential for each decoding block to unpack the input data stream in this architecture, as shown in Fig. 8.

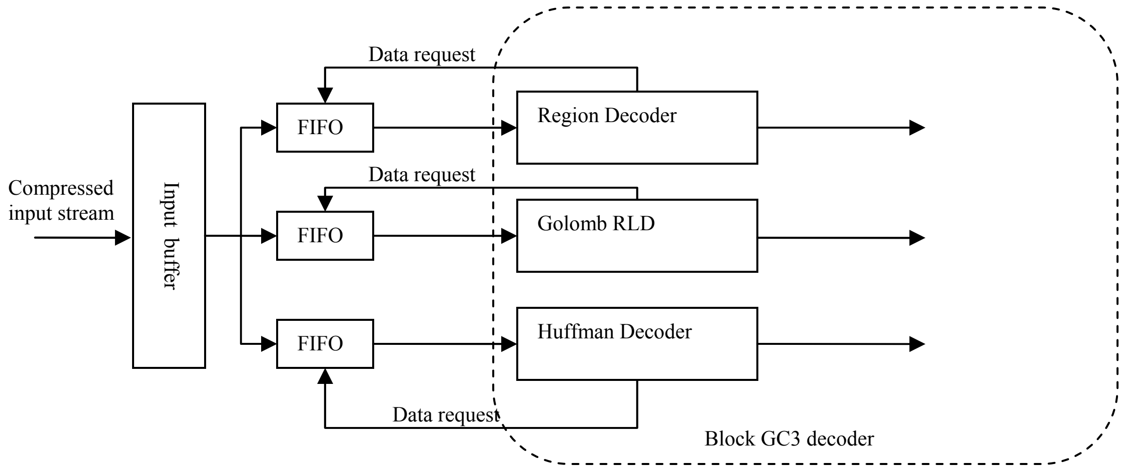


Fig. 8 Data distribution architecture of Block GC3 decoder.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we present the FPGA synthesis and emulation result for Block GC3 decoder. A single Block GC3 decoder only utilizes 4% of the resources of a Xilinx Virtex Pro II 70 FPGA. The system can run at 100 MHz, resulting in an output data rate of 495 Mb/s. With this synthesis result and data-flow architecture of the decoder, it is potentially feasible to run hundreds Block GC3 decoders in parallel to achieve the high-throughput needed for direct-write lithography systems. In order to integrate the decoder in the path, we also propose a number of data handling strategies in this paper.

The FPGA implementation is the first step towards implementing the Block GC3 decoder in hardware. For future work, we plan to implement the decoder in ASIC, in order to improve output throughput. In addition, since Block GC3 is a generic lossless compression algorithm for direct-write lithography systems, we may have to modify the algorithm to accommodate the specifics of different writer technologies and layout images while keeping the decoder complexity low.

ACKNOWLEDGEMENT

This research was supported by KLA Tencor. We would also like to acknowledge Daniel Burke and Henry Chen in Berkeley Wireless Research Center for their valuable input on BEE2 system.

REFERENCES

[1] V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data", Emerging Lithographic Technologies VI, Proc. of the SPIE Vol. 4688, pp. 583–594, 2002.
[2] V. Dai and A. Zakhor, "Advanced Low-complexity Compression for Maskless Lithography Data", Emerging Lithographic Technologies VIII, Proc. of the SPIE Vol. 5374, pp. 610–618, 2004.
[3] H. Liu, V. Dai, A. Zakhor, B. Nikolic, "Reduced Complexity Compression Algorithms for Direct-Write Maskless Lithography Systems," SPIE Journal of Microlithography, MEMS, and MOEMS (JM3), Vol. 6, 013007, Feb. 2, 2007.
[4] A. Zakhor, V. Dai, and G. Cramer. "Full Chip Characterization of Compression Algorithms for Direct Write Maskless Lithography Systems," SPIE Conference on Advanced Lithography, San Jose, California, February 2009.
[5] Chen Chang, John Wawrzynek, and Robert W. Brodersen, "BEE2: A High-End Reconfigurable Computing System," IEEE Design & Test of Computers, March 2005.

[6] H. K.-H. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH," ACM Transactions on Embedded Computing Systems (TECS), Volume 7, Issue 2, Feb, 2008.

[7] http://bee2.eecs.berkeley.edu/

[8] Lin, Shu, and Daniel J. Costello, "Error Control Coding: Fundamentals and Applications," Prentice-Hall computer applications in electrical engineering series, Prentice-Hall, 2001.

[9] C Winstead *et.* al., "CMOS analog MAP decoder for (8,4) Hamming code," IEEE Journal of Solid-State Circuits, Vol. 39, Issue 1, pp 122-131, Jan. 2004.