Reduced Complexity Compression Algorithms for Direct-Write Maskless Lithography Systems

Hsin-I Liu, Vito Dai, Avideh Zakhor, and Borivoje Nikolić
Department of Electrical Engineering and Computer Science
U.C. Berkeley
{hsil, vdai, avz, bora}@eecs.berkeley.edu

ABSTRACT

Achieving the throughput of one wafer layer per minute with a direct-write maskless lithography system, using 22 nm pixels for 45 nm feature sizes, requires data rates of about 12 Tb/s. In our previous work, we developed a novel lossless compression technique specifically tailored to flattened, rasterized, layout data called *Context-Copy-Combinatorial-Code (C4)* which exceeds the compression efficiency of all other existing techniques including BZIP2, 2D-LZ, and LZ77, especially under limited decoder buffer size, as required for hardware implementation. In this paper, we present two variations of the C4 algorithm. The first variation, Block C4, lowers the encoding time of C4 by several orders of magnitude, concurrently with lowering the decoder complexity. The second variation which involves replacing hierarchical combinatorial coding part of C4 with Golomb run-length coding, significantly reduces the decoder power and area as compared to Block C4. We refer to this algorithm as *Block Golomb Context Copy Code (Block GC3)*. We present the detailed functional block diagrams of Block C4 and Block GC3 decoders along with their hardware performance estimates as the first step of implementing the writer chip for maskless lithography.

Keywords: C4, maskless lithography, complexity, implementation, decoder, prediction, buffer, memory, segmentation

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 recently proposed[5] a data path architecture shown in Figure 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 micromirror array or E-beam writers.

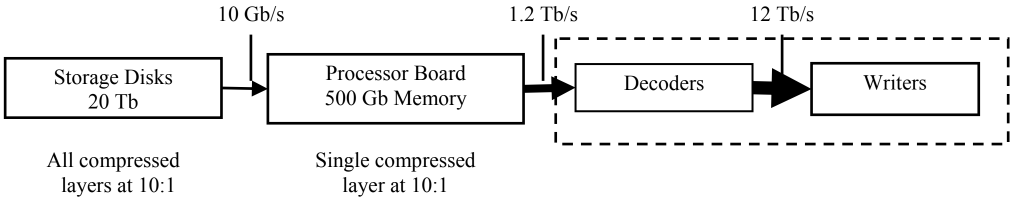


Figure 1. The data delivery path of maskless lithography.

Emerging Lithographic Technologies X, edited by Michael J. Lercel,
Proc. of SPIE Vol. 6151, 61512B, (2006) · 0277-786X/06/$15 · doi: 10.1117/12.656844

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. Since there are a large number of decoders operating in parallel on the writer chip, an important requirement for any compression algorithm is to have an extremely low decoder complexity. To this end, we have proposed a lossless layout compression algorithm for flattened, rasterized data called Context Copy Combinatorial Coding (C4) which 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, as required for hardware implementation. However, a major drawback of the C4 algorithm as presented in our previous work is its encode complexity[2]. For example, compressing one rasterized, 15-layer layout of a 10 mm × 10 mm chip would take 2 months on a 1000 processor machine. This can be attributed to the exhaustive search nature of the segmentation portion of the C4 algorithm.

In this paper, we present two variations of the basic C4 algorithm. In Section 2, we introduce Block C4 to improve the encode complexity of C4 by few orders of magnitude. As it turns out, Block C4 also results in lower decoder complexity. In Section 3, we replace the Hierarchical Combinatorial Coding (HCC) part of C4 with Golomb run-length coding, to obtain a lower decode complexity algorithm called Block Golomb Context Copy Coding (Block GC3). In Section 4, we discuss hardware implementation aspects of the decoder for Block C4 and Block GC3. In Section 5, we discuss area, power, and speed estimates of Block C4 and Block GC3 decoders for direct-write maskless lithography systems.

2. BLOCK C4

Block C4 is an improvement over the C4 compression algorithm[2]. Similar to C4, it is designed to compress flattened, rasterized layout data, and provides similar compression efficiency but at a tiny fraction of the encoding time. In Table 1, we compare the compression efficiency and encoding time for two 1024 × 1024 5-bit grayscale layout images, generated from two different sections of the poly layer of a layout. The flavor of C4 used here and throughout this paper is C4+LP, the variant of C4 with the lowest decoder implementation complexity. LP stands for Linear Prediction of a pixel based on its surrounding pixels as described in our previous work[1]. Encoding times are generated on an Athlon64 3200+ Windows XP desktop with 1 GB of memory. As seen, Block C4 is 115 times faster for the Poly-memory layout, and 865 times faster on the Poly-control layout with no noticeable loss in compression efficiency.

Table 1. Comparison of compression ratio and encode times of C4 vs. Block C4.

Layout	C4 Compression Ratio	C4 Encode Time	Block C4 Compression Ratio	Block C4 Encode Time
Poly-memory	7.60	1608 s (26.8 min)	7.63	14.0 s (115x speedup)
Poly-control	9.18	12113 s (3.4 hrs)	9.18	13.9 s (865x speedup)

The significance of a speedup of this magnitude in encoding time cannot be understated. Indeed, if we extrapolate from an average encoding time of 30 minutes per 1024 × 1024 layout image, a 20mm × 10mm chip die drawn on a 22 nm 5-bit grayscale grid would take over 22 CPU years to encode. Block C4 reduces this to number to 60 CPU days, still a large number, but manageable by today's multi-CPU compute systems.

Another benefit of Block C4, apparent in Table 1, is an approximately constant computation time of about 14 seconds per 1024 × 1024 layout image, independent of the layout data, as compared to widely varying computation times of C4, from 27 minutes to 3.4 hours. A predictable and consistent computation time is important to project planners managing the overall data processing flow, for example, when planning jobs to maximize tool usage.

The remainder of this section describes how Block C4 achieves this encoding speedup with no loss in compression efficiency. In Section 2.1, we introduce the segmentation algorithm of C4 and contrast it with Block C4. In Section 2.2, we examine the problem of choosing a block size for Block C4. In Section 2.3, we describe how the Block C4 segmentation is encoded for compression efficiency.

2.1. Segmentation in C4 vs. Block C4

The basic concept underlying both C4 and Block C4 compression is exactly the same. Layout data is characterized by a heterogeneous mix of repetitive and non-repetitive structures, examples of which are shown in Figures 2(a) and 2(b) respectively. Repetitive structures are compressed efficiently using Lempel-Ziv (LZ) style copying, whereas non-repetitive structures are better compressed using localized context-prediction techniques[5]. The task of both the C4 and Block C4 encoder is to automatically partition the image into repetitive copy regions, and non-repetitive prediction regions, in a process called *segmentation*. The result is a *segmentation map*, which indicates whether copy or prediction should be used to compress each pixel of the image. Once the segmentation into prediction versus copy is complete, it is straightforward to encode each pixel according to this segmentation map. The segmentation map must also be encoded and included as part of the compressed data, so that the decoder knows which algorithm to apply to each pixel for decoding.

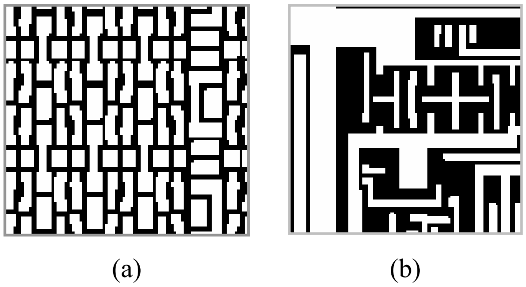


Figure 2. (a) Repetitive and (b) non-repetitive layouts.

The task of computing the segmentation map accounts for nearly all the computation time of the C4 encoder. Of the encode times reported in Table 1, the encode time excluding segmentation, is a constant 1.2 seconds, for both C4 and Block C4. In other words, over 99.9% of the encode time of C4 and 91% of the encode complexity of Block C4 is attributable to segmentation.

In C4, the segmentation is described as a list of rectangular copy regions. An example of a copy region is shown in Figure 3. Each copy region is a rectangle, enclosing a repetitive section of a layout, described by 6 attributes: the rectangle position (*x*,*y*), its width and height (*w*,*h*), the orthogonal direction of the copy (*dir* = *left* or *above*), and the distance to copy from (*d*) i.e. the period of the repetition.

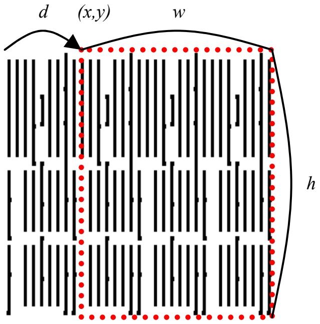


Figure 3. Illustration of a copy region.

What makes automated C4 segmentation such a complex task is that the "best" segmentation, or even a "good" segmentation is hardly obvious. Even in such a simple example shown in Figure 4, there are many potential copy regions, a few of which are illustrated in Figure 4 as dotted and dashed rectangles. The number of all possible copy regions is of the order of O(N^5) for N×N pixel layout, and choosing the best set of copy regions for a given layout is a combinatorial problem. Exhaustive search in this space is prohibitively complex, and C4 already adopts a number of greedy heuristics to make the problem tractable[2]. Clearly further complexity reduction of the segmentation algorithm is desirable.

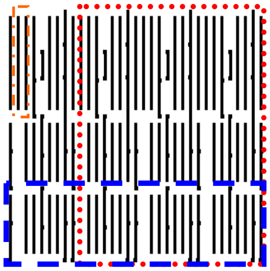
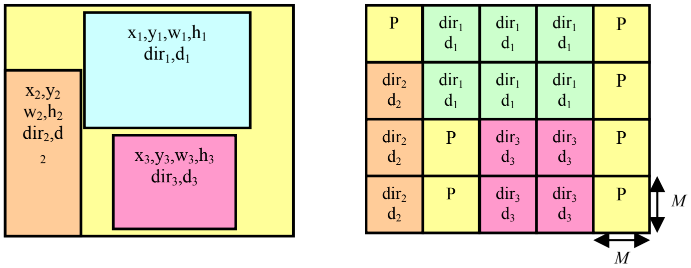


Figure 4. Illustration of a copy region.

Block C4 adopts a far more restrictive segmentation algorithm than C4, and as such, is much faster to compute. Specifically, Block C4 restricts both the position and sizes to fixed *M*×*M* blocks on a grid whereas C4 allows for copy regions to be placed in arbitrary (*x*,*y*) positions with arbitrary (*w*,*h*) sizes. Figure 5 illustrates the difference between Block C4 and C4 segmentation. In Figure 5(a), the segmentation for C4 is composed of 3 rectangular copy regions, with 6 attributes (*x*,*y*,*w*,*h*,*dir*,*d*) describing each copy region. In Figure 5(b), the segmentation for Block C4 is composed of 20 *M*×*M* tiles, with each tile marked as either prediction (*P)*, or the copy with direction and distance (*dir*, *d*). This simple change reduces the number of possible copy regions to

$$O(\frac{N^3}{M^2}) \sim \frac{N^2}{M^2} \times O(N),$$

a substantial N^2M^2 reduction in search space compared to C4. For the experiment in Table 1, we have $N = 1024$ and $M = 8$, so the copy region search space has been reduced by a factor of 64 million. However, this complexity reduction could potentially come at the expense of compression efficiency.



(a) (b)

Figure 5. Segmentation map of (a) C4 vs. (b) Block C4.

2.2. Choosing a block size for Block C4

The three large copy regions in C4 segmentation map in Figure 5(a) have been divided into 13 small square blocks in Block C4 in Figure 5(b) in this example. In general, a large repetitive *w*×*h* region is broken up into wh/M^2 tiles in Block C4. Each copy region tile in Block C4 is represented with only 2 attributes (*dir*,*d*) rather than the 6 per copy region (*x*,*y*,*w*,*h*,*dir*,*d*) in C4. If a sufficiently large tile is broken up, there may be a net increase in the amount of data needed to represent the segmentation information, which adversely affects the compression ratio of Block C4. Smaller values of *M* accentuate this effect, motivating the use of larger *M*.

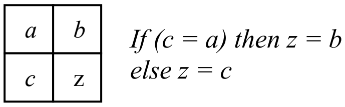
However, large values of M could also be disadvantageous. Comparing the segmentation map of C4 in Figure 5(a) to that of Block C4 in Figure 5(b), the rectangles are forced to snap to the coarse grid in Block C4. In C4, the rectangle boundaries are optimized to delineate repetitive regions from non-repetitive regions. In Block C4, the coarse grid causes this delineation to be sub-optimal. Consequently, at the boundary of the copy regions, repetitive regions are predicted, and non-repetitive regions are copied. This sub-optimal segmentation could potentially lower the compression efficiency. Of course, the smaller and finer the grid, the lower the occurrence of grid snapping, hence motivating the use of a smaller M.

These arguments would suggest that there is an optimal M value that trades off between grid snapping and the breakup of large copy regions. We have empirically found $M = 8$ to exhibit the best compression efficiency for nearly all test cases as compared to $M = 4$ or $M = 16$. In the remainder of this paper, we use $M = 8$ in all of our Block C4 experimental results, unless otherwise stated.

2.3. Context-based block prediction for encoding Block C4 segmentation

To further improve the compression efficiency of Block C4, we note that the segmentation shown in Figure 5(b) is highly structured. Indeed, the segmentation can be used to represent boundaries in a layout separating repetitive regions from non-repetitive regions, and that these repetitions are caused by design cell hierarchies, which are placed on an orthogonal grid. Consequently, Block C4 segmentation has an orthogonal structure, and C4 already employs a reasonably efficient method for compressing orthogonal structures placed on a grid, namely context-based prediction[2].

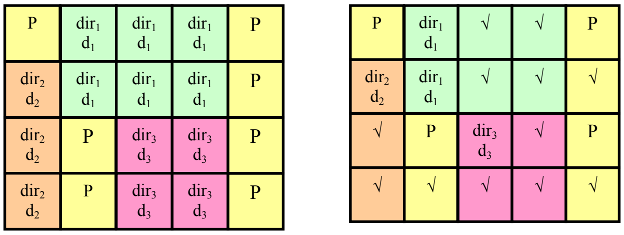
To encode the segmentation, blocks are treated as pixels, and the attribute (P,dir,d) as colors of each block. Each block is predicted from its 3-block neighborhood, as shown in Figure 6. For vertical edges corresponding to $c = a$, it is likely for z to be equal to b. Similarly for horizontal edges corresponding to $a = b$, it is likely for z to be equal to c. Consequently, the prediction shown in Figure 6 only fails around corner blocks, which are assumed to occur less frequently than horizontal or vertical edges. Applying context-based block prediction to the segmentation in Figure 7(a), we obtain Figure 7(b) where √ marks indicate correct predictions. The pattern of √ marks could be compressed using HCC[3] or any other binary coding techniques, and the remaining values of (P,dir,d) could be Huffman coded, exactly analogous to the method of coding copy/prediction error bits and values used in C4[2]. For Block C4, we choose to use Golomb run-length coder to compress segmentation error locations. This is because segmentation error location amounts to a very small percentage of the output bit stream, and as such, applying a complex scheme such as HCC is hard to justify.



a	b
c	z

If ($c = a$) then $z = b$
else $z = c$

Figure 6. Three-block prediction for encoding segmentation in Block C4.



P	dir_1 d_1	dir_1 d_1	dir_1 d_1	P
dir_2 d_2	dir_1 d_1	dir_1 d_1	dir_1 d_1	P
dir_2 d_2	P	dir_3 d_3	dir_3 d_3	P
dir_2 d_2	P	dir_3 d_3	dir_3 d_3	P

(a)

P	dir_1 d_1	√	√	P
dir_2 d_2	dir_1 d_1	√	√	√
√	P	dir_3 d_3	√	P
√	√	√	√	√

(b)

Figure 7. (a) Block C4 segmentation map (b) with context-based prediction.

3. BLOCK GC3－ALTERNATE WAY TO COMPRESS THE ERROR LOCATION

In both C4 and Block C4, the error location bits are compressed using HCC. While HCC is useful for encoding the highly-skewed binary data in a lossless fashion[3], when it comes down to hardware implementation, the hierarchical structure of HCC implies repetitive hardware blocks and inevitable decoding latency from the top level to the final output. Moreover, as we show in Section 5, the HCC block becomes the bottleneck of the entire system due to its long delay. To overcome this problem, we propose to replace HCC in Block C4 by a Golomb run-length coder[8], resulting in a new compression algorithm called BLOCK GC3. As such, Golomb run-length coder in Block GC3 is now used to encode error locations of both the pixels in the layout and the segmentation blocks in the segmentation map. Figure 8 shows the block diagram for Block GC3, which is more or less identical to that of C4[1] with the exception of the pixel error location encoding scheme.

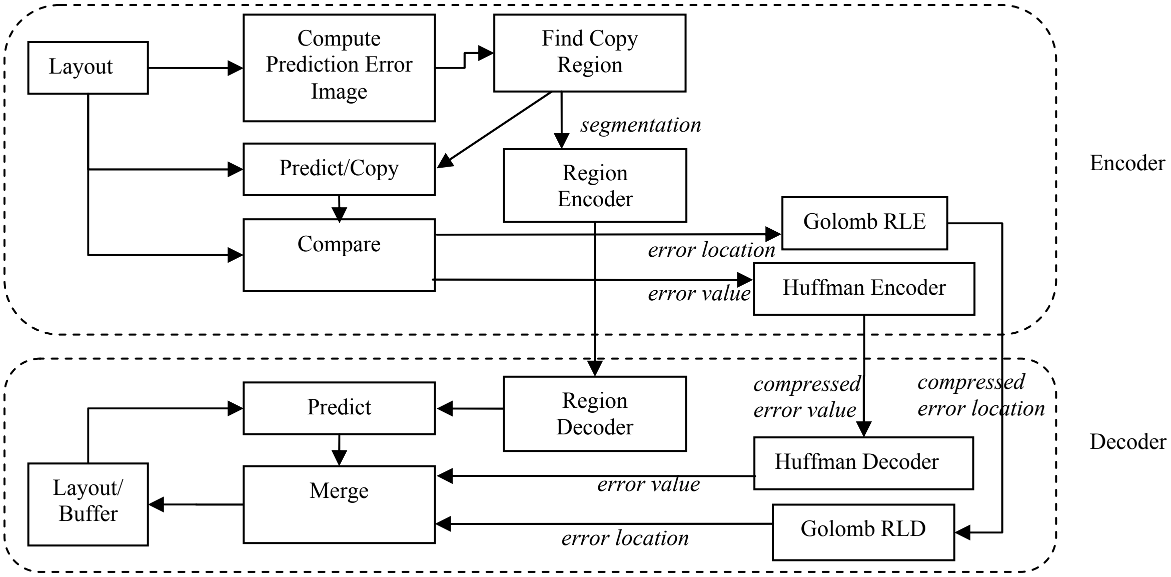


Figure 8. The encoding/decoding architecture of Block GC3.

Coding the pixel error location of layouts with Golomb run-length code could potentially lower the compression efficiency. Figure 9 shows a binary stream coded with both HCC and Golomb run-length coder. In the upper path, the stream is coded with Golomb run-length coder. In this case, the input stream is either coded as (0), denoting a stream of *B* "0"s where *B* denotes a predefined bucket size, or coded as (1,*n*), indicating a "1" occurs after *n* "0"s. These parameters are further converted into a bit stream, where parameter (0) is translated into a 1-bit codeword and (1,*n*) takes $1+\log_2 B$ bits to encode. Therefore, a stream with successive "1"s can potentially be encoded into a longer code than a stream with "1"s which are far apart from each other. On the other hand, in the lower path of Figure 9, HCC counts the number of "1"s within a fixed block size and codes it using enumerative code[2]. As long as the number of "1"s inside the block is fixed, HCC results in a fixed length bit stream regardless of the input distribution.

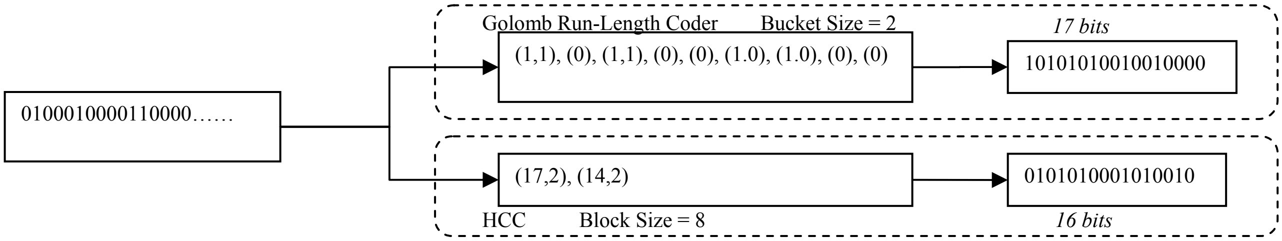


Figure 9. Golomb run-length encoding process.

Based on the above, Block GC3 can result in potential compression efficiency loss for certain class of images. Specifically, Figure 10 shows a typical layout with successive prediction errors occurring at the corner of Manhattan shapes due to the linear prediction property. Since error locations are not distributed in an independent-identically distributed (iid) fashion, there is potential compression efficiency loss due to Golomb run-length coder as compared to HCC. To alleviate this problem, we adapt the bucket size for Golomb run-length coder from layer to layer.

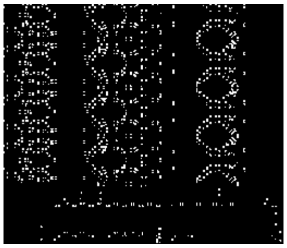


Figure 10. Visualization of pixel error location for a layout image.

As shown in Table 3, Block GC3 results in about 10 to 15 % lower compression efficiency than Block C4 over different process layers of layouts assuming decoder buffer size of 1.25 KB. The test images in Table 3 are 1024×1024 5-bit grayscale rasterized, flattened layouts. Similarly, Figure 11 compares the minimum compression efficiency of Block C4, Block GC3, and few other existing lossless compression schemes as a function of decoder buffer size[1]. The minimum is computed over ten 1024×1024 images manually selected among five layers of two IC layouts. In practice, we focus on 1.25 KB buffer size for hardware implementation purposes. While Block GC3 results in slightly lower compression efficiency than Block C4 for nearly all decoder buffer sizes, it outperforms all other existing lossless compression schemes such as LZ77, ZIP, BZIP2, Huffman, and RLE.

Table 3. Compression ratio comparison between Block C4 and Block GC3 for different layers of layout.

Layers	Compression Ratio (Block C4)	Compression Ratio (Block GC3)	Bucket size for Block GC3
Metal 1 mixed	14.21	12.67	16
Metal 2 mixed	33.81	28.83	64
N active mixed	43.10	36.51	64
P active mixed	66.17	59.24	128
Poly mixed	11.00	9.633	16

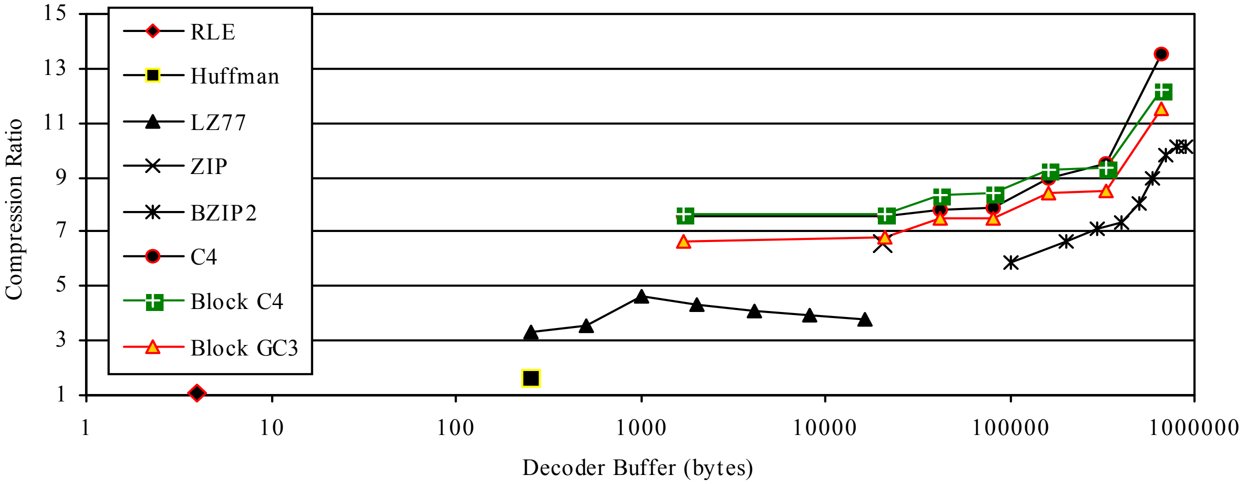


Figure 11. Compression efficiency and buffer size tradeoff for Block C4 and Block GC3.

4. DECODER ARCHITECTURE

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 C4 and Block GC3 algorithms, we devise the data-flow architecture for the decoder[1].

The block diagram of Block C4 decoder is shown in Figure 12. There are three main inputs: the segmentation, the compressed error location, and the 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 predicted value from Linear Prediction and the copied value from History Buffer in the Control/Merge stage, as shown in Figure 13. The compressed pixel error location is decoded by HCC, 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 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 Section 3, the total amount of needed memory inside a single Block C4 decoder is about 2 KB, which can be implemented using on-chip SRAM.

The block diagram of Block GC3 is almost identical to that of Block C4 shown in Figure 12, since it only replaces the HCC block of Block C4 by a Golomb run-length decoder.

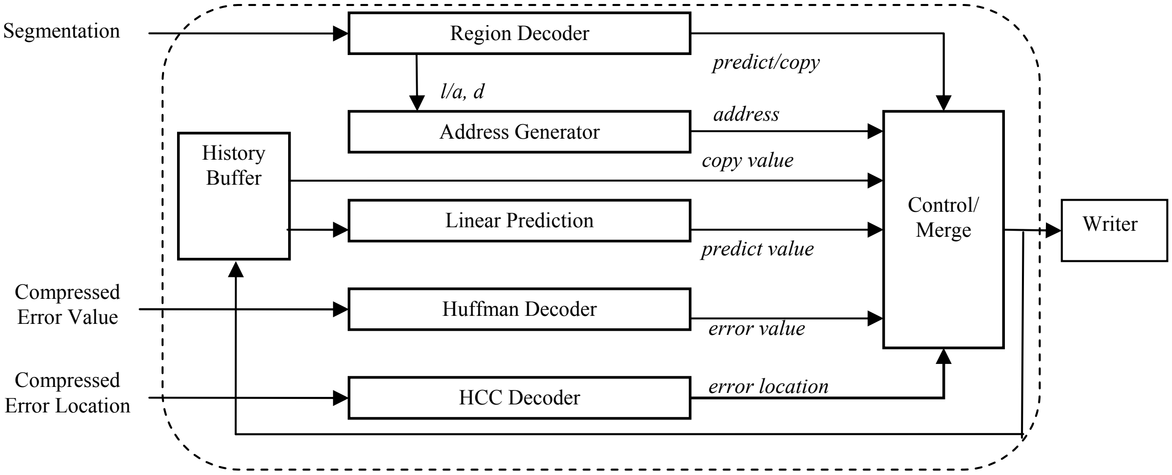


Figure 12. Functional block diagram of Block C4 decoder.

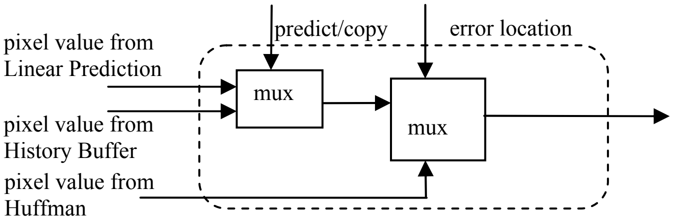

predict/copy
error location
pixel value from Linear Prediction
mux
pixel value from History Buffer
mux
pixel value from Huffman

Figure 13. Block diagram of Merge/Control block.

In the remainder of this section, we discuss the architecture for the Block C4 and Block GC3 decoders. Even though there are seven major blocks shown in Figure 12, we focus on the two most challenging blocks of the design, namely the Region Decoder and HCC. For Block GC3, we only discuss the design of Golomb run-length decoder, as its main distinctive feature.

4.1. Region decoder

In the description of the Block C4 algorithm in Section 2, a segmentation map is introduced to represent the predict/copy segmentation of the layout. Similar to an actual IC layout, the segmentation map is also Manhattan shaped, and can be compressed by prediction algorithms. However, since the segmentation map is an artificially generated image, there is no correlation between the values of adjacent segments. As a result, the segmentation predictor shown in Figure 6 is used in the Region Decoder rather than the linear predictor used for pixel predictions in a layout.

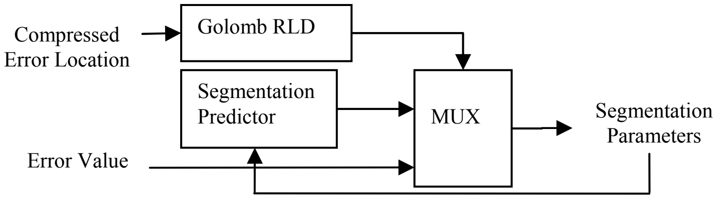


Figure 14. Block diagram of Block C4 Region Decoder.

Figure 14 shows the block diagram of the Region Decoder for Block C4. The segmentation input has been separated into two streams, the compressed error location and the error value. The core of the Region Decoder is the segmentation predictor. As shown in Figure 14, the output of the Region Decoder is selected to be either the error value or the output of the segmentation predictor, depending upon the segmentation error location provided by the Golomb run-length decoder. Similar to the linear predictor for the layout image, the segmentation predictor of Block C4 applies the three-block-based prediction: the segmentation of the current micro-block is determined by the three adjacent blocks from upper, left, and upper left micro-blocks under the conditions shown in Figure 6.

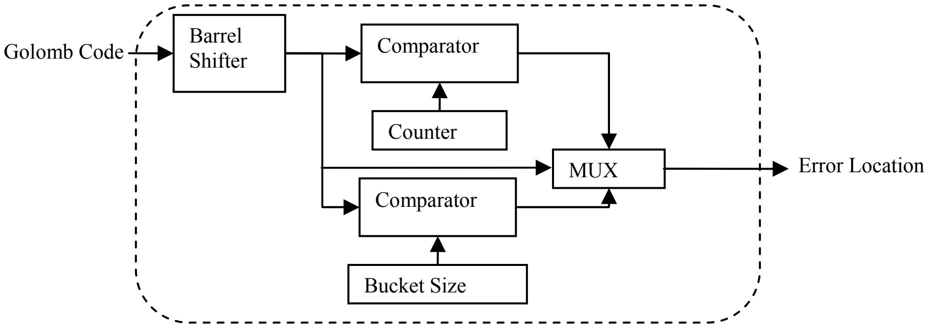


Figure 15. Block diagram of Golomb run-length decoder.

The design of Golomb run-length decoder is adapted from techniques in the existing literature[9], as shown in Figure 15. It is the combination of a barrel shifter and a conventional run-length decoder. The barrel shifter is used as a data buffer to compact the coded error location into an 8-bit data stream, and the decoded result is used as the input to a run-length decoder, resulting in a binary output stream. In contrast to the approach in the literature, we only use one barrel shifter in our design in order to reduce the hardware overhead.

The proposed Region Decoder has several architectural advantages over the original C4 Region Decoder[1]. First, it is implemented as a regular data path, in contrast to a linked-list structure, thus eliminating feedback and latency issues.

Second, the output of Block C4 Region Decoder is the control signal over an 8×8 micro-block, which lowers the output rate of Region Decoder by 64, and reduces the power consumption. Finally, the length of the segmentation parameter is reduced from 51 bits (*[x, y, w, h, dir, dist]*) to 19 bits, i.e. 8-bit error location and 11-bit error value, resulting in fewer I/O pins in the decoder.

4.2. HCC block design

Combinatorial coding (CC) is an algorithm for compressing a binary sequence of 0's and 1's. For Block C4, it represents a binary pixel error location map. A "0" represents a correctly predicted or copied pixel, and a "1" represents a prediction/copy error. CC encodes this data by dividing the bit sequence into blocks of fixed size H, e.g. $H = 4$, and computing k_{block} the number of "1"s in each block. If $k_i = 0$, this means block i has no ones, so it is encoded as 0000, with a single value k_i. If $k_i > 0$, e.g. $k_i = 1$, then it needs to be disambiguated between the list of possible 4-bit sequences with one "1": {1000, 0100, 0010, 0001}. This can be done with an integer representing an index into that list denoted $rank_{block}$. In this manner, any block i of H bits can be encoded as a pair of integers (k_i, $rank_i$). The theoretical details of how this achieves compression can be found in our previous work[3], but, intuitively it can be expressed as follows: if the data contains contiguous sequences of "0"s, and if the length of these all "0" sequences matches the block size H, each block of H "0"s can be concisely encoded as ($k_i = 0$) with no rank value, effectively compressing the data.

Computational complexity of CC grows as the factorial of the block size H. Hierarchical combinatorial coding (HCC) avoids this issue by limiting H to a small value, and recursively applying CC to create a hierarchy, as shown in Figure 16.

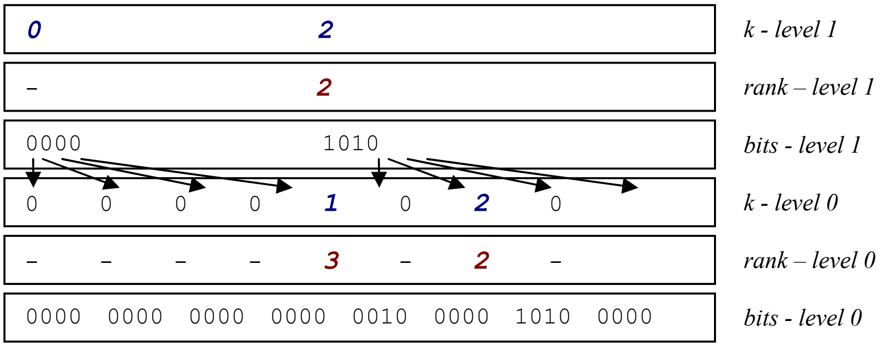


Figure 16. Two-level HCC with a block size H = 4 for each level.

In Figure 16, the original binary sequence is the lowest row of the hierarchy "bits – level 0". It has been encoded using CC as "k – level 0" and "rank – level 0" with a block size $H = 4$. We now recursively apply CC on top of CC by first converting the integers in "k – level 0" to binary "bits – level 1"as follows: 0 is represented as 0, and non-zero integers are represented as 1. Applying CC to "bits – level 1" results in "k – level 1" and "rank – level 1". The advantage of the hierarchical representation is that a single 0 in "k – level 1" now represents 16 zeros in "bits – level 0". In general, a single 0 in "k - level L" corresponds to H^{L+1} zeroes in "bits – level 0", compressing large blocks of 0's more efficiently. The disadvantage of decoding the HCC is that it requires multiple CC decoding steps, as we traverse the hierarchy from top to bottom.

The task of traversing the hierarchy of HCC decoding turns out to be the main throughput bottleneck of HCC decoder, which in turn is the throughput bottleneck of the entire Block C4 decoder. The block diagram of a sequential HCC decoder is shown in Figure 17(a). Block C4 uses a 3-level $H = 8$ HCC decoder. The dashed lines separate the HCC levels from top to bottom, and the data that moves between levels are the "bits – level L." Three CC blocks represent (k, *rank*) decoders for levels 2, 1, and 0, from top to bottom respectively. CC – level 2 decodes to bits – level 2. If bits – level 2 is a "0" bit, the multiplexer (MUX) selects the run-length decoder (RLD) block which generates 8 zeros for "bits – level 1". Otherwise, the MUX selects the CC – level 1 block to decode a (k, *rank*) pair. Likewise, "bits level – 1" controls the

MUX in level 0. A “0” causes the RLD block to generate 8 zeros, and a “1” causes CC – level 0 to decode a (*k*, *rank*) pair. In this sequential design, the output of a lower level must wait for the output of a higher level to be available before it can continue. Consequently, the control signal corresponding to when the output of the lowest level “bits level – 0” is ready resembles Figure 17(c). While levels 2 and 1 are decoding as indicated by the shaded boxes, the output of layer 0 must stall, reducing the effective overall throughput of the HCC block.

To overcome the problem of hierarchical HCC decoding, we can parallelize the operation by introducing a FIFO buffer between HCC levels, as indicated by the additional squares in Figure 17(b), and by dividing the input (*k*, *rank*) values for each HCC level into multiple sub-streams. The idea is that after an initial delay to fill the buffers of levels 2 and 1, level 0 can decode continually as long as the buffers are not empty. This is guaranteed because one level 2 output bit corresponds to 8 level 1 output bits, and 64 level 0 output bits. Level 2 and level 1 can continue to decode into these buffers while level 0 is working. Consequently, the output control signal of the parallel design resembles Figure 17(d), where only the initial delay is noticeable. The control mechanism of the parallel design is also considerably simpler than the sequential design, because each HCC level can now be controlled independently of the other HCC levels, halting only when its output buffer is full, or its input buffer is empty. Only a 2-byte FIFO is introduced between each level.

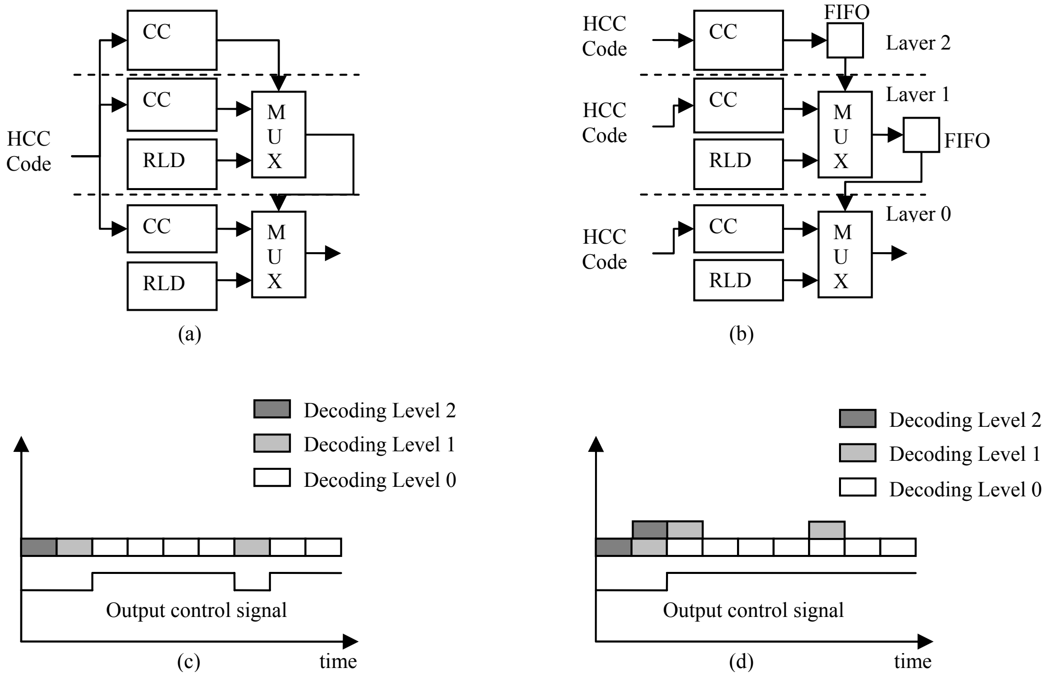


Figure 17. The decoding process of HCC in (a) top-to-bottom fashion and (b) parallel scheme. The timing analysis of (c) top-to-bottom fashion and (d) parallel scheme.

4.3. Golomb run-length decoder for Block GC3

Since the pixel error location in Block GC3 is encoded with Golomb run-length coder, the pixel error location decoder of Block GC3 resembles the Golomb run-length decoder for the segmentation map in the Region Decoder of Block C4. However, for pixel error locations, it is advantageous to use a variable bucket size in the Golomb run-length coder for different process layers in order to improve compression efficiency, as discussed in Section 3. The block diagram of Golomb run-length decoder for error location is shown in Figure 18. The only difference between Figures 15 and 18 is that the variable bucket size is introduced as an input signal to the decoder. The main advantage of this implementation over HCC is that it has zero latency, and does not require any stall cycles during the decoding process due to its regular data path structure.

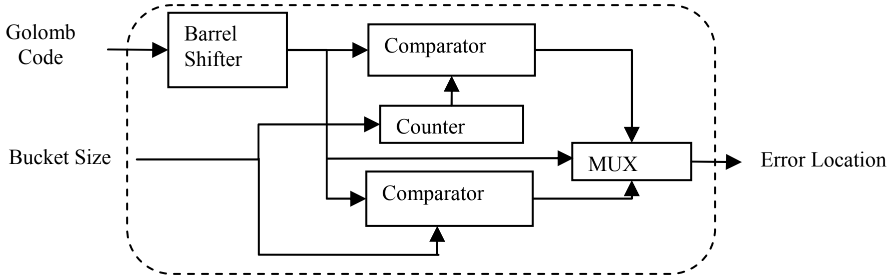


Figure 18. Block diagram of Golomb run-length pixel error location decoder in Block GC3.

5. DECODER PERFORMANCE

By applying the block diagram discussed in the last section, we have implemented Block C4 and Block GC3 decoders with logic synthesis tools in a general-purpose 90 nm bulk-CMOS technology. To accurately estimate the speed, power, and area, each of the building blocks has been translated from the hardware description language to the gate level. Table 4 shows the estimate of area, speed, throughput, and power for Block C4 and Block GC3. Comparing with the LZ77 decoder implemented in our previous work[7], Block C4 decoder uses half of the area, and results in twice as mush compression efficiency. Furthermore, the Block GC3 decoder is half the size of Block C4 decoder.

Table 4. Estimated hardware performance comparison of different data path for direct-write maskless lithography systems.

Block		Area (um^2)	Tp (ns)	Throughput (output/cycle)	Power (W)
Golomb		4,845.539	1.63	1	0.0018
HCC		43,135.68	1.43	0.71	0.0074
Huffman		2,745.10	0.83	1/codeword+2	0.0016
Linear Prediction		2,674.85	1.06	1	0.0011
Address Generator		1,897.75	0.6	0.94	0.0009
Region Decoder		5,965.46	1.39	1	0.0017
Control/Merge		4,166.49	0.69	1	0.0016
Memory		40,080	1.03	1	0.010
Block C4	Single decoder	100,665.3	1.43	0.71	0.0246
	Total (200)	20,254,592	1.43	0.71	6.375
Block GC3	Single decoder	62,375.18	1.63	0.94	0.0191
	Total (200)	11,550,960	1.63	0.94	5.192
Direct-write		--	--	--	10.8

Table 4 also shows that the critical path for Block C4 is in the HCC block, whereas for Block GC3, the critical path is in the Golomb run-length decoder. This timing analysis indicates that the single Block C4 decoder can operate at 700MHz clock rate, while Block GC3 can only run at 600MHz.

The throughput of the decoder is analytically generated, and summarized in Table 4. This is done by multiplying the latency for each block by its usage probability. The usage probability is determined by analyzing the encoder statistics of the test layouts. Although the latency of HCC in Block C4 has been smoothed out by applying parallel decoding, there are still some inevitable stall periods in HCC block, resulting in a throughput of 71% of the peak. In contrast, Block GC3 only has a few stalls in the predict/copy segment transition caused by the Address Generator. The throughput of Block GC3 is around 94% of the peak. Combining the clock rate, throughput, and the five-bit output pixel value we compute the output rate of Block C4 and Block GC3 decoders to be 2.48 Gb/s and 2.7 Gb/s respectively for a 1024×1024 image. As a result, 200 decoders are needed to achieve 500 Gb/s output rate, which corresponds to 3 wafer layers per hour. To be competitive with today's optical mask lithography systems which generate one wafer layer per minute, about 5000 decoders are needed to run in parallel.

Table 4 also shows the power consumption of different maskless lithography datapath approaches using gate-level simulations of Block C4 and Block GC3 decoders. As for the direct-write technique, it is possible to apply 80 high-speed I/O pins operating at 6.4 Gb/s to achieve 500 Gb/s data rate without data compression[6]. However, such a method would result in the total power of the writer to be 10.8 W. By applying Block C4 and Block GC3 to the data, the power consumption of the writer chip is substantially reduced by approximately a factor of 2. As shown in Table 4, Block C4 and Block GC3 achieve 37% and 51% power reduction respectively as compared to the direct-write technique. As compared to Block C4 decoder, Block GC3 decoder achieves 42% area reduction and 18% power reduction, making Block GC3 an attractive option to be implemented in actual maskless lithography systems.

6. SUMMARY AND FUTURE WORK

In this paper, we have discussed two variations of C4 to reduce its complexity overhead in both encoding and decoding processes. Block C4 can solve the encoding latency issue by changing the segmentation algorithm into a prediction-based scheme, reducing the encoding time by two orders of magnitude. Block GC3, which replaces HCC by Golomb run-length code for pixel error location coding, can further reduce the hardware decoder overhead of Block C4 by 42% in area and 18% in power.

We plan to fabricate the Block GC3 decoders in a near future. Subsequently, the integration of decoder and writer chip poses new challenges. One of them is the input balancing problem, where the input data of different decoders are transmitted from common input pins at constant rate from the storage device to the processor board. In such a case, the size of the on-board memory and the access mechanism has to be carefully designed. Also, another important topic to investigate is the mixed signal processing problem from the digital memory to the D/A converters.

ACKNOWLEDGEMENTS

This research was conducted under the Research Network for Advanced Lithography, supported jointly by the Semiconductor Research Corporation (2005-OC-460) and the Defense Advanced Research Project Agency (W911NF-04-1-0304). We would also like to acknowledge Brian Richards of U.C. Berkeley for his invaluable input on hardware design implementation.

REFERENCES

1. V. Dai and A. Zakhor, "Complexity Reduction for C4 Compression for Implementation in Maskless Lithography Datapath", in *Emerging Lithographic Technologies IX*, Proceedings of the SPIE, 5751, March 2005.
2. V. Dai and A. Zakhor, "Advanced Low-complexity Compression for Maskless Lithography Data", Emerging Lithographic Technologies VIII, Proc. of the SPIE, 5374, pp. 610-618, 2004.
3. V. Dai and A. Zakhor, "Binary Combinatorial Coding", Proc. of the Data Compression Conference 2003, p. 420, 2003.
4. V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data", Emerging Lithographic Technologies VI, Proc. of the SPIE, 4688, pp. 583-594, 2002.
5. V. Dai, A. Zakhor, "Lossless Layout Compression for Maskless Lithography" in *Emerging Lithographic Technologies IV*, Proceedings of the SPIE, 3997, March 2000.

6. K. Chang, S. Pamarti, K. Kaviani, E. Alon, X. Shi, T.J. Chin, J. Shen, G. Yip, C. Madden, R. Schmitt, C. Yuan, F. Assaderaghi, and M. Horowitz, “Clocking and Circuit Design for A Parallel I/O on A First-Generation CELL Processor,” in International Solid-State Circuit Conference, February 2005.
7. B. Nikolić, B. Wild, V. Dai, Y. Shroff, B. Warlick, A. Zakhor, and W. G. Oldham, "Layout Decompression Chip for Maskless Lithography" in *Emerging Lithographic Technologies VIII*, Proceedings of the SPIE, 5374, February 2004.
8. S. W. Golomb, “Run-length Encodings”, IEEE Transactions on Information Theory, IT-12 (3), pp. 399-401, 1966.
9. M.-T. Sun, “VLSI Architecture and Implementation of A High-Speed Entropy Decoder,” IEEE International Symposium on Circuits and Systems, pp. 200 – 203, 1991