5

Transform-Based Lossless Image Compression Algorithm for Electron Beam Direct Write Lithography Systems

Jeehong Yang[1] and Serap A. Savari[2]
[1]*University of Michigan, Ann Arbor*
[2]*Texas A&M University*
USA

1. Introduction

Conventional photolithography systems use physical masks which are expensive and difficult to create and cannot be used forever. Electron Beam Direct Write (EBDW) lithography systems are a noteworthy alternative which do not need physical masks [Chokshi et al. (1999)]. As shown in Figure 1 they rely on an array of lithography writers to directly write a mask image on a photo-resist coated wafer using electron beams. EBDW systems are attractive for a few reasons: First, their flexibility is advantageous in processes requiring the rapid prototyping of chips. Second, they are known to reduce fabrication costs [Lin (2009)]. Third, they are well suited for Next-Generation Lithography (NGL) because they are able to produce circuits with smaller features than state-of-the-art photolithography systems. Finally, since the mask images are electronically controlled EBDW systems could be improved by software. Our focus here will be on this last point.

EBDW is not at this time used in many circuit fabrication processes because it is much slower than physical mask lithography systems. One current focus of research to address the throughput problem is massively-parallel electron beam lithography. Some of the research groups/companies which are developing such systems include KLA-Tencor [Petric et al. (2009)], IMS [Klein et al. (2009)], and MAPPER [Wieland et al. (2009)].

Chokshi et al. (1999) proposed a maskless lithography system using a bank of 80,000 lithography writers running in parallel at 24 MHz. Dai & Zakhor (2006) pointed out that this lithography system can achieve the conventional photolithography throughput of one wafer layer per minute, but layout image data is often several hundred terabits per wafer and therefore data delivery becomes an important issue. Dai & Zakhor (2006) proposed using a data delivery system with a lossless image compression component which is illustrated in Figure 2. They hold compressed layout images in storage disks and transmit the compressed data to the processor memory board. This kind of EBDW lithography system can achieve higher throughput if the decoder embedded within the lithography writer can sufficiently rapidly recover the original images from the compressed files.

Dai (2008) discussed two constraints on this type of system: 1) the compression ratio should be at least (Transfer rate of Decoder to Writer / Transfer rate of Memory to Decoder), and 2) the decoding algorithm has to be simple enough to be implemented as a small add-on within the maskless lithography writer. Therefore the decoder must operate with little memory.

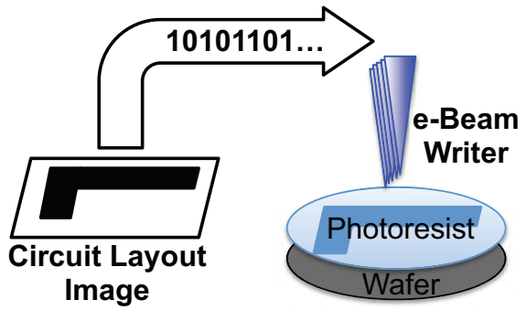


Fig. 1. EBDW Lithography

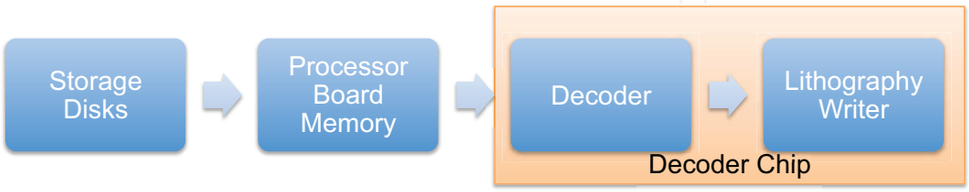


Fig. 2. Data Delivery for an EBDW Lithography System introduced in Dai & Zakhor (2006)

Dai & Zakhor (2006) reported that the layout images of control logic circuits are often irregular while the layout images of memory cells frequently contain repeated patterns. Their first algorithm *C4* attempts to handle the varying characteristics of layout images by using context prediction and finding repeated regions within an image. Liu et al. (2007) later proposed *Block C4*, which significantly reduces the encoding complexity.

Based on the framework of Dai & Zakhor (2006) and Liu et al. (2007), Yang & Savari (2010) improved the compression algorithm via a corner-based representation of the Manhattan polygons. Their initial algorithm *Corner* achieves higher compression rates than *Block C4* on an irregular circuit. Yang & Savari (2011) recently proposed an improvement *Corner2* which simplifies the corner transformation to deal with the irregular parts of the layout images and also uses a frequent pattern replacement scheme to deal with the repeated parts. Their experimental results indicate that their approach is often more efficient than the context prediction method used in *C4* and *Block C4*.

In this paper we extend the work of Yang & Savari (2011) to gray-level images to better address the issue of handling proximity correction for EBDW systems and show that we obtain better compression performance and faster encoding/decoding than *C4* and *Block C4*. Hence our work can be used to solve the data delivery problem of EBDW lithography systems with smaller features. Moreover, since our decoding speed is faster than *C4* and *Block C4* we can improve the throughput of the EBDW lithography system.

2. The compression algorithm

2.1 Overview

Layout image data is commonly cached in GDSII [Rubin (1987)] or OASIS [Chen et al. (2004)] formats. GDSII and OASIS describe circuit features such as polygons and lines by their corner points [see Rubin (1987) and Reich et al. (2003)]. GDSII and OASIS formatted data are far more compact than the uncompressed image of a circuit layer. Therefore GDSII and OASIS initially seem to be well-suited for this application, but the problem is that EBDW writers operate directly on pixel bit streams and GDSII and OASIS layout representations must therefore be converted into layout images before the process begins. The conversion process involves 1)

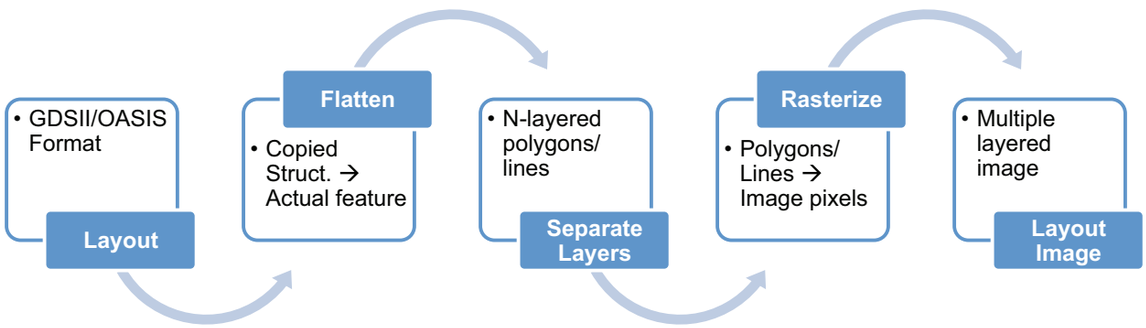


Fig. 3. Preparing Layout Images from a Circuit Layout - Rasterizing Process

removing hierarchical structures by replacing all of the copied parts with actual features, 2) placing the circuit features such as polygons and lines into the correct layers of the circuit, and 3) rasterizing (see Figure 3). This conversion process often lasts hours or even days using a complex computer system with large memory and cannot be executed by the decoder chip.
The final rasterizing step consists of two parts: a) it produces a binary image on a finer grid, and b) the binary image is processed in blocks to generate a gray-level image. In the second step the input binary string is partitioned into $m \times m$ pixel blocks. For each block the number of filled pixels are computed and normalized/quantized to the corresponding gray level. When this gray-level image is transmitted to the EBDW lithography system the lithography writer interprets the gray level (or pixel intensity) as an exposure dose which is controlled by exposing the corresponding region multiple times with an electron beam. Through this process the printed layout pattern becomes more robust to the electron beam proximity effect making better quality circuits.
Our approach is motivated by the compactness of the GDSII/OASIS format and uses corner representation. However, we bypass the complex flattening and rasterizing processes and instead work with a simple decoding process. Yang & Savari (2011) considered some of these ideas for binary images which handle the proximity correction by rasterizing the input binary image on a fine enough grid. Here we will extend these ideas to gray-level images on a coarser grid.
Figure 4 summarizes the components of the compression algorithm. We begin by applying a corner transformation to the image like the one in *Corner2* [Yang & Savari (2011)]. However, unlike *Corner2* this transformation outputs two streams: a "corner stream" and an "intensity stream". The corner stream is a binary stream which locates the polygon corners[1] and the intensity stream is a stream of pixel (corner/edge) intensities. Each stream is input to a separate entropy coding scheme which outputs a compressed bit stream. The corner stream is compressed using a combination of run length encoding [Golomb (1966)], end-of-block coding, and arithmetic coding [Moffat et al. (1998)]. The intensity stream is compressed using end-of-block coding and then compressed by LZ77 [Ziv & Lempel (1977)] and Huffman coding [Huffman (1952)].
In Section 2.2 we will first describe the corner transform process which outputs the corner stream and the intensity stream. In Section 2.3 we will describe the final entropy coding process of the corner stream, and in Section 2.4 we will describe how the intensity value is compressed.

[1] This is not actually a corner, but a horizontal/vertical transition point as explained in Subsection 2.2.

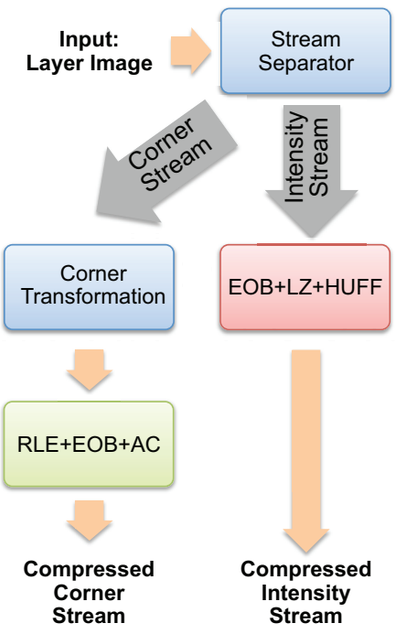


Fig. 4. Compression Process Overview

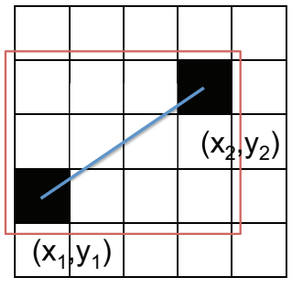


Fig. 5. Required decoder memory (red) to reconstruct a line (blue) from (x_1, y_1) to (x_2, y_2).

2.2 Corner transformation

The GDSII/OASIS representation of a structurally flattened single layer describes the layout polygons by their corner points. This representation requires large decoder memory since the decoder needs to access a memory block of size $(|x_1 - x_2| + 1) \times (|y_1 - y_2| + 1)$ for the encoder to connect an arbitrary pair of points (x_1, y_1) and (x_2, y_2) as in Figure 5. Therefore this representation is infeasible for our application.

However, the rasterizing process becomes much less complex if the angle of a contour line is constrained to a small set. Yang & Savari (2010) took advantage of horizontal and vertical contour lines and decomposed an arbitrary polygon into a collection of Manhattan polygons, i.e., polygons with right angle corners. This approach is effective because most components of circuit layouts are produced using CAD tools which design the circuit in a rectilinear space, and the non-Manhattan parts can also be described by Manhattan components.

In this framework, the decoder scans the image in raster order, i.e., each row in order from left to right. When the decoder processes a corner it must determine whether it should reconstruct a horizontal and/or a vertical line. Observe that a corner is either the beginning of a line going to the right and/or down or the end of a line. Yang & Savari (2010) assigned each pixel one of five possible values – 'not corner,' 'right,' 'right and down,' 'down,' and 'stop.'

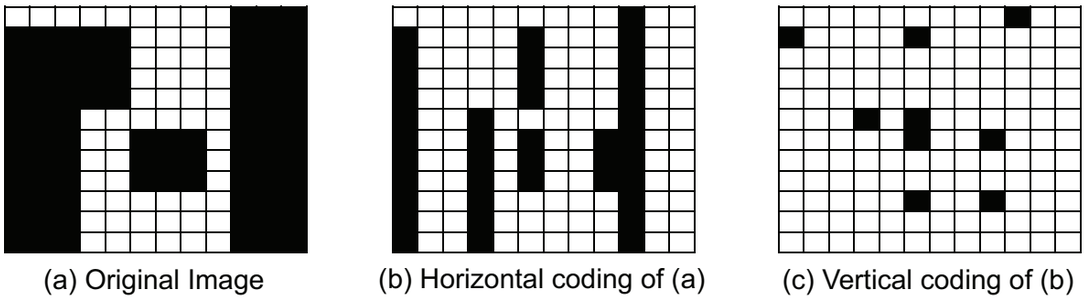


Fig. 6. 2-Symbol Corner Transformation

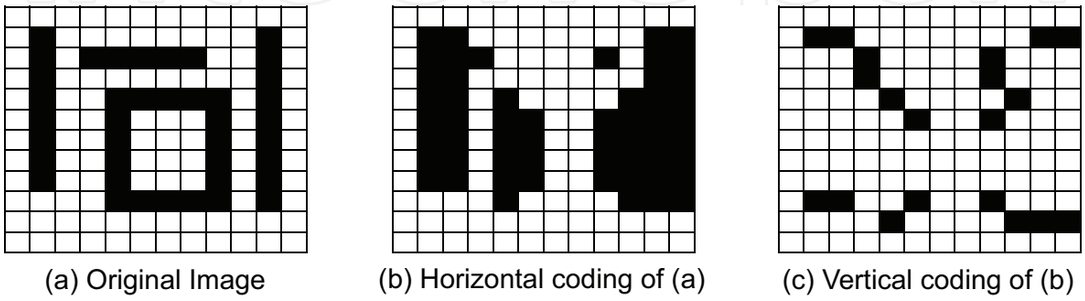


Fig. 7. Handling width-1 lines

Yang & Savari (2011) more recently observed that a row (or a column) of the original binary layout image consists of alternating runs of 1s (fill) and runs of 0s (empty). Therefore it is more efficient to encode pixels where there are transitions from 0 to 1 (or 1 to 0) using symbol "1" and to encode the other places using symbol "0." Observe, as in Figure 6(b), that after applying this encoding in the horizontal direction to a collection of Manhattan polygons that the output consists of alternating runs of 1s and 0s in the vertical direction. To increase the compression we repeat this encoding in the other direction to obtain the final corner image. In the binary corner transformation the final encoded image is binary and the "1"-pixels give information about the corners of the polygons. To describe the algorithm we begin with a two-step transformation process and then shorten it to a one-step procedure which requires less memory during the encoding process and is faster than the two-step transformation process.

The two-step transformation process begins with a horizontal encoding step in which we process each row from left to right. For each row, the encoder sets the (imaginary) pixel value to the left of the leftmost pixel to 0 (not filled). If the value of the current pixel differs from the preceding one we represent it with a "1" and otherwise with a "0." The second step inputs the intermediate encoded result to the vertical encoding process in which each column is processed from top to bottom. In the specification of the algorithms x denotes the column index $[1, \cdots, C]$ of the image and y represents the row index $[1, \cdots, R]$. Algorithm 1 illustrates this process.

Line 13 of Algorithm 1 constructs $\texttt{OUT}(x, y) = 1$ only if $\texttt{TEMP}(x, y) \neq \texttt{TEMP}(x, y-1)$. That is, $\texttt{OUT}(x, y) = 1$ only if $\texttt{TEMP}(x, y) = 1$ and $\texttt{TEMP}(x, y-1) = 0$ or if $\texttt{TEMP}(x, y) = 0$ and $\texttt{TEMP}(x, y-1) = 1$. Since $\texttt{TEMP}(x, y) = 1$ only if $\texttt{IN}(x-1, y) \neq \texttt{IN}(x, y)$ as in Line 5, we can shorten the corner transform process to Algorithm 2, which has no need for intermediate memory since pixel (x, y) is processed in terms of the input pixels $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$. Algorithm 2 is much faster than Algorithm 1. Finally, Figure 7 illustrates how the transformation handles width-1 lines.

Algorithm 1 Transformation : Two-Step Algorithm

Input: Binary layer image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Corner image $\texttt{OUT} \in \{0,1\}^{C \cdot R}$
Intermediate: Temporary image $\texttt{TEMP} \in \{0,1\}^{C \cdot R}$
{**Horizontal Encoding**}
Initialize $\texttt{TEMP}(x,y) = 0, \forall x, y$.
for $y = 1$ **to** R **do**
  for $x = 1$ **to** C **do**
    if $\texttt{IN}(x,y) \neq \texttt{IN}(x-1,y)$ **then**
      $\texttt{TEMP}(x,y) = 1$.
    end if
  end for
end for
{**Vertical Encoding**}
Initialize $\texttt{OUT}(x,y) = 0, \forall x, y$.
for $x = 1$ **to** C **do**
  for $y = 1$ **to** R **do**
    if $\texttt{TEMP}(x,y) \neq \texttt{TEMP}(x,y-1)$ **then**
      $\texttt{OUT}(x,y) = 1$.
    end if
  end for
end for

Algorithm 2 Transformation : One-Step Algorithm

Input: Binary layer image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Corner image $\texttt{OUT} \in \{0,1\}^{C \cdot R}$
Initialize $\texttt{OUT}(x,y) = 0, \forall x, y$.
for $y = 1$ **to** R **do**
  for $x = 1$ **to** C **do**
    if $\texttt{IN}(x-1,y-1) = \texttt{IN}(x,y-1)$ and $\texttt{IN}(x-1,y) \neq \texttt{IN}(x,y)$ **then**
      $\texttt{OUT}(x,y) = 1$
    end if
    if $\texttt{IN}(x-1,y-1) \neq \texttt{IN}(x,y-1)$ and $\texttt{IN}(x-1,y) = \texttt{IN}(x,y)$ **then**
      $\texttt{OUT}(x,y) = 1$
    end if
  end for
end for

Based on the experimental success in Yang & Savari (2010) and Yang & Savari (2011) for binary layout images it is natural to expect that a combination of the corner transformation for the outline of gray-level polygons and a separate representation for the intensity stream would outperform *Block C4*. Note that nLv-level gray images for this application have pixel intensity 0 (empty) outside the polygon outline, $\texttt{nLv} - 1$ (fully filled) inside the polygon outline, and an element of (0,nLv) along the polygon outline. Therefore we need only consider intensities along polygon corners and edges. Finally, in order to obtain the polygon outline using the corner transformation, we first have to map the gray-level image to a binary image. This is

easily done by mapping all of the nonzero intensities to 1 (fill) and leaving the zero intensities (not fill) unchanged.

2.3 Entropy coding - corner stream

The corner stream typically contains long runs of zeroes and is therefore well-suited to compression algorithms like run length encoding [Golomb (1966)] and end-of-block (EOB) coding. Because the corner transformed image is a sparse binary image, if read in raster order (as we read) the string would consists of ones and runs of zeroes. During the compression process, the transitional corners (ones) of the transformed image are written unchanged, but each run of zeroes is described by its run length via an M-ary representation which we next describe. Define the new symbols "2", "3", $\cdots$, "M+1" to respectively represent the base-M symbols "0_M", "1_M", $\cdots$, "$(M-1)_M$". For example, if the transformed stream was "1 00000 00000 1 00000 0000 1 00000 00000 000" and $M = 3$, then the encoding of the stream is "1 323 1 322 1 333" because the run length are 10 (=101_3), 9 (=100_3), and 13 (=111_3), and 2/3/4 to respectively represent $0_3/1_3/2_3$.

We find that the addition of EOB coding helps represent the corner stream more efficiently. When the polygons are aligned and start/end at the same rows of the image the resulting runs of zeroes could be longer than a multiple of the row width. Although this could be handled by choosing M sufficiently large the memory requirements for the encoding and decoding of the final M-ary representation via arithmetic coding [Moffat et al. (1998)] for further compression requires a choice of M as small as possible in our restricted decoder memory setting.

We observe that it is effective to divide each line into k blocks of length L, and we define a new EOB symbol "X". If a run of zeroes appears at the end of a block we represent that run using an end-of-block symbol X instead of an M-ary representation. Hence the encoding for a line of zeroes is k X's instead of approximately $\log_M(kL)$ symbols. For the previous example, if M=2, $k = 5$, and $L = 7$, then the transformed stream "1000000 0000100 0000000 1000000 0000000" is described as "1X 3221X X 1X X," where 2/3 (=$0_2/1_2$) is used for the binary representations of runs of zeroes.

We find that EOB coding results in long runs of "X"s and it is useful to employ an N-ary run length encoding to these runs. For the previous example, if $M = N = 2$, $k = 5$, and $L = 7$, then the next description of the string is "1 4 3221 5 1 5," where 2/3 (or 4/5) handles the binary representation of runs of zeroes (or "X"s).

Finally, we compress the preceding stream using the version of arithmetic coding offered by Witten et al. (1987), and the decoder in this case requires four bytes per alphabet symbol. Since we used $M + N + 1$ symbols[2], $4(M + N + 1)$ bytes were used for arithmetic decoding.

2.4 Entropy coding - intensity stream

The corner stream contains no intensity information. Since we are applying row-by-row decompression (from left to right), the intensity values have to be given in that order. The intensity values that we require are for corner pixels and pixels on the edges. As we have mentioned earlier in Section 2.2, the pixels outside the polygons will have 0 intensity (empty) and pixels inside the polygon boundaries will have $\texttt{nLv} - 1$ intensity (fully filled).

To obtain better prediction we could apply linear prediction along the neighboring pixels as is done in *Block C4*. However, this approach requires the full information of the previous row which translates to decoder memory. Therefore we instead apply EOB encoding to the pixels corresponding to horizontal/vertical edges because the pixel intensity along an edge seldom

[2] M symbols are used for runs of zeroes, N symbols are used for runs of "X"s, and 1 is used for the transitional corners.

changes unless oblique lines are used. We encode the intensity stream as in Algorithm 3. Note that in the algorithm ρ is the length of the intensity stream which is determined at the end of the encoding process.

Algorithm 3 Intensity Stream Encoding

Input: Gray layer image $\texttt{IN} \in \{0, \cdots, \texttt{nLv} - 1\}^{C \cdot R}$
Input: Binary layer image $\texttt{BIN} \in \{0, 1\}^{C \cdot R}$
Output: Intensity stream $\texttt{OUT} \in \{0, \cdots \texttt{nLv} - 1\}^{\rho}$

1: Initialize $\rho = 0$.
2: **for** $y = 1$ **to** R **do**
3: **for** $x = 1$ **to** C **do**
4: **if** (x, y) is a corner pixel **then**
5: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(x, y)$.
6: **else if** (x, y) is a horizontal edge pixel with corners at $(x - 1, y)$ and $(x + \alpha, y)$ **then**
7: **if** $\texttt{IN}(i, y)$ has the same value for $i \in [x, x + \alpha)$ **then**
8: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(x, y)$.
9: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \epsilon$.
10: $x = x + \alpha - 1$.
11: **else**
12: **for** $i = x$ **to** $x + \alpha - 1$ **do**
13: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(i, y)$
14: **end for**
15: $x = i$.
16: **end if**
17: **else if** (x, y) is a vertical edge pixel with corners at $(x, y - 1)$ and $(x, y + \beta)$ **then**
18: **if** $\texttt{IN}(x, j)$ has the same value for $j \in [y, y + \beta)$ **then**
19: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(x, y)$.
20: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \epsilon$.
21: **for** $j = y$ **to** $y + \beta - 1$ **do**
22: $\texttt{IN}(x, j) = 0$
23: **end for**
24: **else**
25: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(x, y)$
26: **end if**
27: **else if** $\texttt{IN}(x, y) > 0$ and (x, y) is a vertical edge pixel **then**
28: $\rho = \rho + 1$, $\texttt{OUT}(\rho) = \texttt{IN}(x, y)$.
29: **end if**
30: **end for**
31: **end for**

If the current pixel corresponds to a corner (Lines 4-5), the intensity is represented as is. If the current pixel corresponds to a horizontal edge pixel (Lines 6-16) which starts from the left pixel, check the run of that intensity. If the horizontal edge pixel has constant pixel intensity throughout the entire edge, represent the intensity value followed by an end symbol ϵ and skip to the ending corner pixel (Lines 7-10). Otherwise, write the entire edge intensity as is (Lines 11-15). Similarly, if the current pixel corresponds to a vertical edge (Lines 17-29) which starts from the upper pixel determine whether or not the pixel intensities are fixed throughout the vertical edge. If they are constant then represent the intensity value followed by the end

symbol ϵ (Lines 18-20) and reset the intensity values for the following rows (Lines 21-23) so that they are not processed in Lines 27-29. Otherwise, write the intensity value as is and proceed (Lines 24-26). Finally, the remaining vertical edge pixel intensities are written in Lines 27-29.

After the entire intensity stream has been processed, compress the output stream using LZ77 and Huffman coding. The LZ77 algorithm by Ziv & Lempel (1977) compresses the stream by finding matches from the previously processed data. When a pattern is repeated within the search region, it could be encoded using a short codeword. Huffman coding is used at the end of LZ77 to represent the LZ77 stream more efficiently. The combination of LZ77 and Huffman coding is widely used in a number of compression algorithms such as *gzip*. We used *zlib* [zlib (2010)] to implement it. The compression rates depend on the size of the LZ77 search region and the dictionary for the Huffman code. Because of the decoder memory restrictions we chose an encoder needing only 2,048 bytes of memory for the dictionary. 2,048 bytes is slightly less than the memory used to describe an entire row of our benchmark circuit. However, since we were applying this only to the intensity stream we were able to match more rows than *Block C4*.

3. Decoder

The decoder consists of an intensity stream decoder and a corner stream decoder as in Figure 8. The intensity stream decoder is actually an entropy decoder which can be decomposed into a Huffman decoder and an LZ77 decoder. The corner stream decoder consists of an entropy decoder which consists of an arithmetic decoder, a run length decoder, an end-of-block decoder, and a corner transform decoder which reconstructs the polygons from the entropy decoder output. The corner transform decoder utilizes the output of corner stream entropy decoder to reconstruct the polygon outlines and uses the output of the intensity stream decoder to reconstruct the polygon pixel intensity.

The entire process works on a row-by-row fashion. Since each part of the decoding procedure (arithmetic decoding, run length decoding, end-of-block decoding, inverse corner transformation, LZ77 decoding, and Huffman decoding) is simple and works with restricted decoder memory, the entire decoder can be implemented in hardware. Note that the most complex part will be the arithmetic decoder which is widely implemented in microcircuits [Peon et al. (1997)], and the other parts are comprised of simple branch, copy, and computation operations as we will see in the following subsection.

3.1 Intensity stream decoder

Decompressing the intensity stream is straightforward. We apply LZ77 and Huffman decoding to obtain the ϵ-coded intensity stream. As we have mentioned in Section 2.4, the decoder requires 2,048 bytes of memory to decode the LZ77 and Huffman codes. The ϵ-coded intensity stream is passed on to the corner transform decoder for the final reconstruction. Note that the decoder does not decompress the entire compressed intensity stream at once but rather decompresses some number of ϵ-coded intensity symbols at the request of the corner transform decoder. The detailed decompression of the ϵ-coded intensity stream will be discussed at the end of the next subsection.

3.2 Corner stream decoder - corner transform decoder

As we have mentioned earlier, the corner stream decoder consists of an entropy decoder and a corner transform decoder. The entropy decoder reverses the procedure of the entropy encoder of Section 2.3. It first reconstructs the run length and end-of-block encoded stream using the

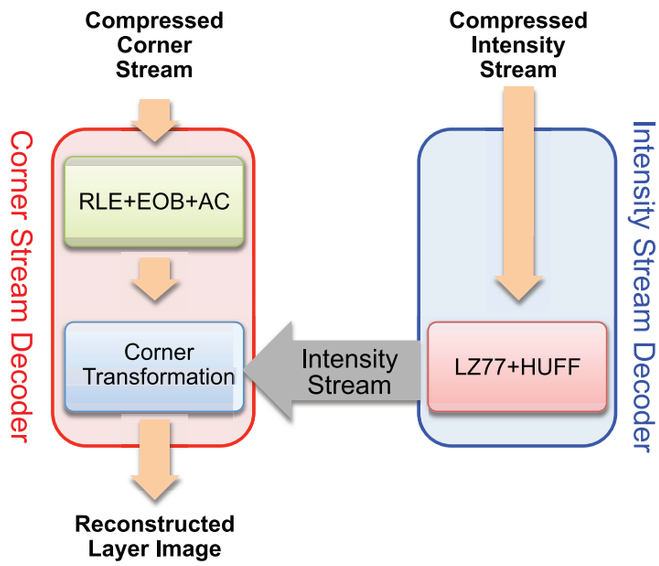


Fig. 8. Decoder Overview: Note that the decompressed corner stream is input into the image reconstructor in a row-by-row fashion and the intensity stream is actually ϵ-coded as in Section 2.4.

arithmetic decoder. Then, depending on the symbol, runs of zeroes (symbols $0_M, \cdots, (M-1)_M$), runs of EOBs (symbols $0_N, \cdots, (N-1)_N$), or the corners (symbol 1) are reconstructed. Finally, the output of the entropy decoder is a binary corner image. In this section we will focus on the operation of the corner transform decoder and why it can run in a row-by-row fashion. This feature makes our approach well-suited to the restricted memory available to an EBDW writer. In our corner transform decoder we use a row buffer `BUFF` to hold the status of the previous (decoded) row. It stores a binary representation of the status of each pixel and therefore consumes *width* bits of memory. "0" denotes 'no transition' while "1" denotes the 'transition' which delineates the starting/ending point of a vertical line. Moreover, since we need the polygon boundaries - corners and horizontal/vertical edge pixels - we use another row buffer `CNR` to hold the boundary status of the previous row. This also requires *width* bits of memory.

Since the algorithm is long, we have split its description into two parts, namely Algorithms 4 and 5. The input to the algorithm is the corner image, and the algorithm outputs the binary layer image and the corner map which shows whether a pixel in the binary layer image is outside all polygons (O), inside a polygon boundary (I), a corner (C), a horizontal edge pixel (H), or a vertical edge pixel (V) which will be used to reconstruct the pixel intensity along with the intensity stream decoder.

The first part of the algorithm, illustrated in Algorithm 4, shows how the buffers are used to pass previous row information to the current row so that the decoding process could be applied in a row-by-row fashion. Lines 5-7 process the binary image buffer `BUFF`. If the buffer is filled then the corresponding pixel is part of a vertical edge and it is filled. Lines 8-15 process the corner map buffer `CNR`. If the corresponding `CNR` pixel does not form a run, then the corresponding pixel above it was a vertical edge pixel (Line 10). The starting and ending points of the runs of 1s are interpreted as the corners (Line 12) and the other pixels in between are translated as horizontal edge pixels (Line 14).

Algorithm 4 Inverse Corner Transformation - Part I

Input: Corner image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Binary layer image $\texttt{BIN} \in \{0,1\}^{C \cdot R}$
Output: Corner map $\texttt{OUT} \in \{O, I, C, H, V\}^{C \cdot R}$
Intermediate: Buffer for image layer $\texttt{BUFF} \in \{0,1\}^{R}$
Intermediate: Buffer for corner map $\texttt{CNR} \in \{0,1\}^{R}$
1: Initialize $\texttt{BUFF}(x) = \texttt{CNR}(x) = 0, \forall x$.
2: Initialize $\texttt{BIN}(x,y) = 0$ and $\texttt{OUT}(x,y) = O, \forall x, y$.
3: **for** $y = 1$ **to** R **do**
4: **for** $x = 1$ **to** C **do**
5: **if** $\texttt{BUFF}(x) = 1$ **then**
6: $\texttt{BIN}(x,y) = 1$
7: **end if**
8: **if** $\texttt{CNR}(x) = 1$ **then**
9: **if** $\texttt{CNR}(x-1) = 0$ and $\texttt{CNR}(x+1) = 0$ **then**
10: $\texttt{OUT}(x,y-1) = V$.
11: **else if** $\texttt{CNR}(x-1) = 0$ or $\texttt{CNR}(x+1) = 0$ **then**
12: $\texttt{OUT}(x,y-1) = C$.
13: **else**
14: $\texttt{OUT}(x,y-1) = H$.
15: **end if**
16: **end if**
17: **end for**
18: **end for**

The second part of the algorithm is shown in Algorithm 5. Note that line 18 of Algorithm 4 and line 1 of Algorithm 5 should be removed when they are implemented; we inserted them to make the loops complete in each part. Lines 5-22 handle the processing of a transitional corner. If $\texttt{BUFF}(x) = 0$, then the transitional corner starts (or ends) a new horizontal edge at row y. Since the entire algorithm is designed to output the result of the previous row $y-1$, we store that horizontal edge in CNR to process it during Algorithm 4. If instead $\texttt{BUFF}(x) = 1$, then the transitional corner starts (or ends) the removal of the pre-existing horizontal edge from the buffer; i.e., the corresponding horizontal edge on the previous row was actually the horizontal boundary.
Similarly, lines 23-33 determine whether the corresponding empty corner pixel is a vertical edge pixel or an interior pixel of a polygon. Finally, note that the output OUT is always processed as a function of row $y-1$, the data that is stored in BUFF and CNR, and row y of IN. Since the buffers treat the inter-row dependencies the entire algorithm can be applied in a row-by-row fashion.
We next utilize the intensity stream to reconstruct the polygon pixel intensities. As we have mentioned earlier, the pixels in the interior of each polygon have full intensity and the pixels not within any polygon are empty (not filled). We use Algorithm 6 to reconstruct the intensity stream of the polygon boundaries. Note that the corner map CNR is the output of the inverse corner transform. In this algorithm, buffer BUFF is used to handle the vertical edge reconstruction.
Algorithm 6 is similar to the previous algorithm in that it utilizes a row buffer to handle the information that has been processed in the previous row. Lines 5-6 deal with the interior polygon pixels, lines 7-9 deal with corner pixels, lines 10-20 reconstruct the horizontal edges,

Algorithm 5 Inverse Corner Transformation - Part II
Run α means $\texttt{BIN}(x,y) = \texttt{BIN}(x,y) \bigoplus \texttt{Fill}$, $\texttt{BUFF}(x) = \texttt{BUFF}(x) \bigoplus \texttt{Fill}$, $x = x + 1$

1: **for** $y = 1$ **to** R **do**
2: $\texttt{Fill} = 0$.
3: Initialize $\texttt{CNR}(x) = 0, \forall x$.
4: **for** $x = 1$ **to** C **do**
5: **if** $\texttt{IN}(x,y) = 1$ **then**
6: $\texttt{Fill} = \texttt{Fill} \bigoplus 1$
7: **if** $\texttt{BUFF}(x) = 0$ **then**
8: Run α
9: **while** $\texttt{IN}(x,y) = 0$ **do**
10: $\texttt{CNR}(x) = 1$
11: Run α
12: **end while**
13: **else**
14: $\texttt{OUT}(x, y-1) = C$.
15: Run α
16: **while** $\texttt{IN}(x,y) = 0$ **do**
17: $\texttt{OUT}(x, y-1) = H$
18: Run α
19: **end while**
20: $\texttt{OUT}(x-1, y-1) = C$.
21: **end if**
22: $\texttt{Fill} = \texttt{Fill} \bigoplus 1$
23: **else**
24: **if** $\texttt{BUFF}(x) = 1$ **then**
25: $\texttt{OUT}(x, y-1) = V$
26: Run α
27: **while** $\texttt{BUFF}(x) = 1$ **do**
28: $\texttt{OUT}(x, y-1) = I$
29: Run α
30: **end while**
31: $\texttt{OUT}(x-1, y-1) = V$
32: **end if**
33: **end if**
34: **end for**
35: **end for**

and lines 21-33 reconstruct the vertical edges. Because the intensity buffer `BUFF` requires a full row, *width* $\times$ `bits per pixel` bits are required for this process.
Since the decoder requires two *width*-bit buffers for corner reconstruction and one *width* $\times$ `bits per pixel`-bit buffer for intensity decoding, the total decoder memory requirement is *width* $\times$ (2 + `bits per pixel`). However, when `bits per pixel` is at least 3 the decoder memory requirement is less than that of *Block C4* which is 2 $\times$ `bits per pixel` $\times$ *width*.

Algorithm 6 Intensity Stream Decoding

Input: Intensity stream $\texttt{IN} \in \{0, \cdots \texttt{nLv} - 1\}^{\rho}$
Input: Corner map $\texttt{CNR} \in \{O, I, C, H, V\}^{C \cdot R}$
Output: Gray layer image $\texttt{OUT} \in \{0, \cdots, \texttt{nLv} - 1\}^{C \cdot R}$
Intermediate: Intensity buffer $\texttt{BUFF} \in \{0, \cdots, \texttt{nLv} - 1\}^{C}$

Initialize $t = 1$.
Initialize $\texttt{OUT}(x, y) = 0, \forall x, y$.
for $y = 1$ **to** R **do**
  for $x = 1$ **to** C **do**
    if $\texttt{CNR}(x, y) = I$ **then**
      $\texttt{OUT}(x, y) = \texttt{nLv}$.
    else if $\texttt{CNR}(x, y) = C$ **then**
      $\texttt{OUT}(x, y) = \texttt{IN}(x, y), t = t + 1$.
      $\texttt{BUFF}(x) = 0$.
    else if $\texttt{CNR}(x, y) = H$ **then**
      if $\texttt{IN}(t + 1) = \epsilon$ **then**
        while $\texttt{CNR}(x, y) = H$ **do**
          $\texttt{OUT}(\rho) = \texttt{IN}(t), x = x + 1$.
        end while
        $t = t + 2$.
      else
        while $\texttt{CNR}(x, y) = H$ **do**
          $\texttt{OUT}(\rho) = \texttt{IN}(t), t = t + 1$.
        end while
      end if
    else if $\texttt{CNR}(x, y) = V$ **then**
      if $\texttt{BUFF}(x) = 0$ **then**
        if $\texttt{IN}(t + 1) = \epsilon$ **then**
          $\texttt{OUT}(x, y) = \texttt{IN}(t)$.
          Set $\texttt{BUFF}(x) = \texttt{IN}(t)$.
          $t = t + 2$.
        else
          $\texttt{OUT}(x, y) = \texttt{IN}(t), t = t + 1$.
        end if
      else
        $\texttt{OUT}(x, y) = \texttt{BUFF}(x)$.
      end if
    end if
  end for
end for

4. Experimental results

We tested the algorithm on a memory circuit which has 13 layers, uses 500 nm lithography technology, and has repeated memory cell structure. The circuit was rasterized on a 1nm grid and then regrouped with block size 250×250. The intensity for each block was quantized using 32 levels. The resulting 5-bit gray level image rasterized on a 250nm grid satisfies the minimum edge spacing of 8nm. For this circuit our algorithm *CornerGray* runs on the entire

layout image, but *Block C4* [Liu et al. (2007)] has a memory shortage/failure. We therefore divided the image in a way to enable the successful application of *Block C4*.
Our code for *CornerGray* and *Block C4* is in C/C++ for the former and in C# for the latter. The experiments were conducted on a laptop computer having a 2GHz Intel Core i7 CPU and 4GB RAM. The decoder memory requirement of *CornerGray* (with parameters $M = N = 64$) was $width \times (2 + \texttt{bits per pixel})/8 + 4(M + N + 1) + 2,048 = 8,514$ bytes while that of *Block C4* was $width \times \texttt{bits per pixel} \times 0.25 + 427 = 8,927$. We require 5% less decoder memory than *Block C4*. Tables 1 and 2 indicate that the compression performance of *CornerGray* is 6% better than *Block C4* and is 189 times faster in encoding and 24 times faster in decoding.

CornerGray Result				
Layer #	File Size (bytes)	Compression Ratio (x)	Encoding Time (s)	Decoding Time (s)
Layer1	2,826	10,720	1.27	0.28
Layer2	570,855	53	1.47	0.48
Layer3	2,826	10,720	1.28	0.28
Layer4	805,477	38	1.49	0.54
Layer5	116,516	260	1.29	0.33
Layer6	116,516	260	1.29	0.33
Layer7	531,770	57	1.44	0.44
Layer8	1,730,054	18	1.84	0.80
Layer9	255,919	118	1.29	0.37
Layer10	1,233,137	25	1.68	0.66
Layer11	107,631	281	1.21	0.31
Layer12	22,092	1,371	1.22	0.29
Layer13	116,516	260	1.27	0.33
Total	5,612,135	70	18.04	5.45

Table 1. Compression Result - *CornerGray*

Block C4 Result				
Layer #	File Size (bytes)	Compression Ratio (x)	Encoding Time (s)	Decoding Time (s)
Layer1	44,436	682	153.89	6.47
Layer2	356,424	85	170.48	7.14
Layer3	44,436	682	182.02	8.11
Layer4	595,116	51	181.68	7.32
Layer5	164,728	184	201.80	7.75
Layer6	164,728	184	300.37	11.21
Layer7	894,308	34	355.59	13.95
Layer8	1,426,824	21	356.40	11.47
Layer9	643,220	47	328.52	12.42
Layer10	1,021,240	30	302.27	9.70
Layer11	346,480	87	332.92	12.50
Layer12	115,040	263	311.24	10.70
Layer13	166,744	182	238.10	9.89
Total	5,983,724	66	3,415.28	128.63

Table 2. Compression Result - *Block C4*

The input image size was $6,800 \times 7,128$ (=30,294,000 bytes) and the *CornerGray* parameters were $N = M = 64$ for all layers. The compression ratio in Table 1 and Table 2 is defined as

$$\frac{\texttt{Input File Size}}{\texttt{Compressed File Size}}.$$

However, the results show that *CornerGray* is relatively weak for handling massively repeated patterns. Among the 13 layers, *CornerGray* did not perform well (compared to *Block C4*) for Layer2, Layer4, Layer8, and Layer10. These layers contained patterns which consist of a large array and Layer8 and Layer10 in particular had complex patterns which were scattered. For these parts the complex LZ-based copying part of *Block C4* resulted in better performance. Hence, more sophisticated pattern matching is required to improve *CornerGray*.

5. Conclusion

In the previous section we saw that the algorithm *CornerGray* outperforms *Block C4* and is considerably faster. The improvement in *CornerGray* over *Block C4* is a result of different techniques. Our corner location approach is simpler than the context prediction used by *Block C4* to handle the irregular parts of layer images. However, *CornerGray* needs a better pattern handling scheme for circuits which contain massively repeated patterns. We are currently trying to generalize the frequent pattern replacement component of *Corner2* [Yang & Savari (2011)] in order to handle frequent patterns within binary layout images and expect similar compression improvements for gray level images. The decoding operations for *CornerGray* include common decompression schemes which are widely implemented in hardware as well as simple branches, compares, and memory copies for the corner transformation part. Therefore our decoder can be deployed using hardware and is an approach to the data delivery problem of maskless lithography systems.

6. Acknowledgment

The second author was supported in part from NSF Grant CCF-1017303.

7. References

Artwork (2010) GDS_RIP. *http://www.artwork.com/gdsii/gds_rip/index.htm*

Chen, Y.; Kahng, A.;Robins, G.; Zelikovsky, A. & Zheng, Y. (2004) Evaluation of the new OASIS format for layout fill compression, *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems*, pp. 377- 382, Tel-Aviv, Israel, December 2004.

Chokshi, N.; Shroff, Y. & Oldham, W. (1999) Maskless EUV Lithography, *International Conference on Electron, Ion, and Photon Beam Technology and Nanofabrication*, Macro Island, FL, June 1999.

Dai, V. & Zakhor, A. (2006) Lossless compression of VLSI layout image data. *IEEE Trans. Image Processing*, Vol. 15, No. 9, pp. 2522-2530.

Dai, V. (2008) *Data Compression for Maskless Lithography Systems: Architecture, Algorithms, and Implementation*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Golomb, S. (1966) Run Length Encodings. *IEEE Trans. Information Theory*, Vol. 12, No.3, pp. 399-401.

Gu, A. & Zakhor, A. (2008) Lossless Compression Algorithms for Hierarchical IC Layout. *IEEE Trans. Semiconductor Manufacturing*, Vol. 21, No. 2, pp. 285-296.

Huffman, D.A. (1952) A Method for the Construction of Minimum-Redundancy Codes, *Proceedings of the I.R.E.*, pp. 1098-1102.

Klein, C.; Klikovitsa, J.; Szikszaia, L.; Platzgummera, E. & Loeschner, H. (2009), Projection Mask-Less Lithography (PML2). *Microelectronic Engineering*, Vol. 87, No. 5-8, pp. 1154-1158.

Petric, P.; Bevis, C.; Brodie, A.; Carroll, A.; Cheung, A.; Grella, L.; McCord, M.; Percy, H.; Standiford, K. & Zywno, M. (2009) REBL nanowriter: Reflective Electron Beam Lithography, *Proceedings of the SPIE 7271 - Alternative Lithographic Technologies*, pp. 727107-1 - 727107-15, San Jose, CA, March, 2009.

Kuhn, M. (2008) JBIG-KIT. *http://www.cl.cam.ac.uk/~mgk25/jbigkit/*

LibTIFF (2010) LibTIFF. *http://www.libtiff.org*

Lin, B. (2009) NGL comparable to 193-nm lithography in cost, footprint, and power consumption. *Microelectronic Engineering*, Vol 86, pp. 442-447.

Liu, H.; Dai, V.; Zakhor, A. & Nikolic, B. (2007) Reduced Complexity Compression Algorithms for Direct-Write Maskless Lithography Systems. *SPIE Journal of Microlithography, MEMS, and MOEMS*, pp. 1-12.

Wieland, M.; de Boer, G.; Ten Berge, G.; Jager, R.; van de Peut, T.; Peijster, J.; Slot, E.; Steenbrink, S.; Teepen, T.; van Veen, A. & Kampherbeek, B. (2009) MAPPER: high-throughput maskless lithography, *Proceedings of the SPIE 7271 - Alternative Lithographic Technologies*, pp. 72710O-1 - 72710O-8, San Jose, CA, March, 2009.

Moffat, A.; Neal, R. & Witten, I. (1998) Arithmetic coding revisited. *ACM Transactions on Information Systems*, Vol. 16, No. 3, pp. 256-294.

Pennebaker, W.; Mitchell, J.; Langdon, G. & Arps, R. (1988) An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of research and development*, Vol. 32, No. 6, pp. 771-726.

Peon, M.; Osorio, R. & Bruguera, J. (1997) A VLSI implementation of an arithmetic coder for image compression, *Proceedings of the 23rd EUROMICRO Conference '97 New Frontiers of Information Technology*, pp. 591-598.

Reich, A.; Nakagawa, K. & Boone, R. (2003) OASIS vs. GDSII stream format efficiency, *Proceedings of SPIE 23rd Annual BACUS Symposium on Photomask Technology*, pp. 163-173, Monterey, CA, September 2003.

Rubin, S. (1987) *Computer Aids for VLSI Design, 2nd ed.*, Addison-Wesley, Boston, MA.

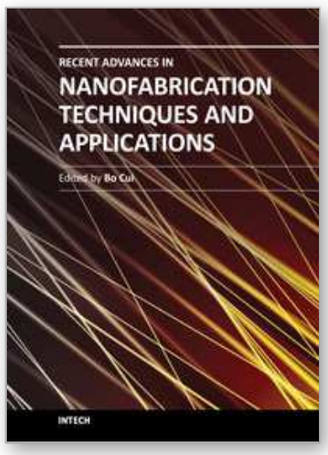
Witten, I.; Neal, R. & Cleary, J. (1987) Arithmetic Coding for Data Compression, *Communications of the ACM*, Vol. 30, pp. 520-540.

Yang, J. & Savari, S. (2010) A Lossless Circuit Layout Image Compression Algorithm for Maskless Lithography Systems, *Proceedings of the 2010 Data Compression Conference*, pp. 109-118, Snowbird, UT, March 2010.

Yang, J. & Savari, S. (2011) A Lossless Circuit Layout Image Compression Algorithm for Electron Beam Direct Write Lithography Systems, *Proceedings of SPIE 7970 - Advanced Lithographic Technologies III*, pp.79701U-1 - 79701U-12, San Jose, CA, March 2011.

Ziv, J. & Lempel, A. (1977) A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337-343.

zlib (2010) zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. *http://zlib.net*



Recent Advances in Nanofabrication Techniques and Applications
Edited by Prof. Bo Cui

ISBN 978-953-307-602-7
Hard cover, 614 pages
Publisher InTech
Published online 02, December, 2011
Published in print edition December, 2011

Nanotechnology has experienced a rapid growth in the past decade, largely owing to the rapid advances in nanofabrication techniques employed to fabricate nano-devices. Nanofabrication can be divided into two categories: "bottom up" approach using chemical synthesis or self assembly, and "top down" approach using nanolithography, thin film deposition and etching techniques. Both topics are covered, though with a focus on the second category. This book contains twenty nine chapters and aims to provide the fundamentals and recent advances of nanofabrication techniques, as well as its device applications. Most chapters focus on in-depth studies of a particular research field, and are thus targeted for researchers, though some chapters focus on the basics of lithographic techniques accessible for upper year undergraduate students. Divided into five parts, this book covers electron beam, focused ion beam, nanoimprint, deep and extreme UV, X-ray, scanning probe, interference, two-photon, and nanosphere lithography.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen