Automatic Cell Layout in the 7nm Era

Pascal Cremer Stefan Hougardy Jan Schneider Jannik Silvanus
Research Institute for Discrete Mathematics, University of Bonn, Germany
{cremer, hougardy, silvanus}@or.uni-bonn.de, schneider.jan@gmail.com

ABSTRACT

Multi patterning technology used in 7nm technology and beyond imposes more and more complex design rules on the layout of cells. The often non local nature of these new design rules is a great challenge not only for human designers but also for existing algorithms. We present a new flow for the automatic cell layout that is able to deal with these challenges by globally optimizing several design objectives simultaneously. Our transistor placement algorithm not only minimizes the total cell area but simultaneously optimizes the routability of the cell and finds a best folding of the transistors. Our routing engine computes a detailed routing of all nets simultaneously. In a first step it computes an electrically correct routing using a mixed integer programming formulation. To improve yield and optimize DFM, additional constraints are added to this model.

We present experimental results on current 7nm designs. Our approach allows to compute optimized layouts within a few minutes, even for large complex cells. Our algorithms are currently used for the design of 7nm cells at a leading chip manufacturer where they improved manufacturability and led to reduced turnaround times.

1. INTRODUCTION

So far an experienced designer is able to craft cell layouts which are of higher quality than automatically generated layouts. However, with each new technology the need for high quality automatic cell layout generators increases. This is due to the fact that design rules and DFM (design for manufacturability) requirements become more and more complex and the number of different cells used in modern designs is growing steadily. Moreover, the manual layout of a complex cell can take several days making this process a severe bottleneck in turnaround time.

In this paper, we present a new flow for the automatic generation of cell layouts, both for placement and routing. Our approach provides solutions that are optimal in terms of area consumption and routability. In practice it also reduces

ISPD '17, March 19-22, 2017, Portland, OR, USA
© 2017 ACM. ISBN 978-1-4503-4696-2/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3036669.3036672

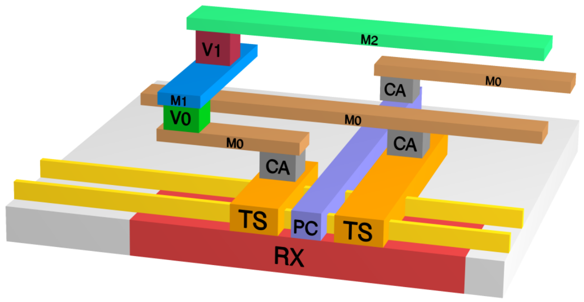
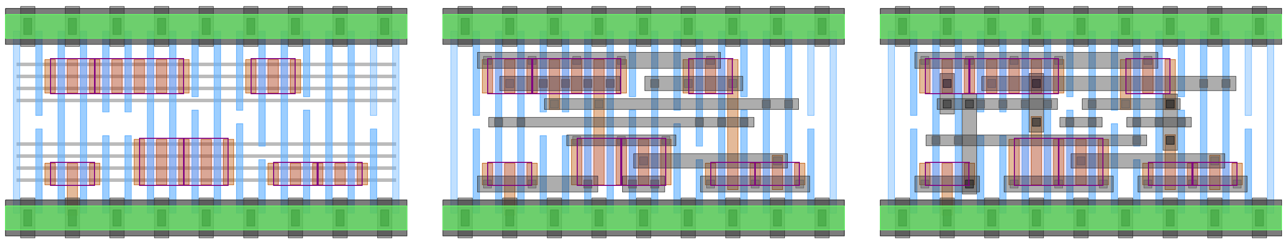


Figure 1: Schematic view of a 7nm layout showing a single finFET and some wiring. Fins (yellow), M0, and M2 are horizontal while PC, TS, and M1 are vertical layers. The diffusion area is denoted by RX. The via layers are CA, V0, and V1.

the need for manual interaction significantly. We consider many complex design rules and DFM requirements already during the placement and the routing phase. This is a crucial requirement for the current 7nm technology and beyond as design rule cleanness and DFM requirements can no longer be achieved by local post-processing operations alone.

In 7nm all layers are uni-directional (see Figure 1). PC and TS are used to contact gates and source/drain contacts of the finFETs. A finFET can have more than one gate. We refer to the number of gates as the number of *fingers* of the transistor. A transistor is called *folded* if it has more than one finger. Three additional wiring layers M0, M1, and M2 are available that alternately have horizontal and vertical orientation. As M2 is also used for inter cell connections, it should be used for internal cell wiring only if necessary. The wiring layers are connected by via layers CA, V0, and V1, where CA connects both PC and TS with M0, V0 connects M0 with M1, and V1 connects M1 with M2. Different multiple patterning techniques are applied for these layers. SAQP (self aligned quadruple patterning) is used for the fins, SADP (self aligned double patterning) for the metal layers, and up to LELELELE (four times litho-edge) for the via layers [19].

The *cell layout problem* can be described as follows. As input an image of the cell is given, i.e. an area with predefined horizontal power tracks at the top and bottom of the cell, equidistant vertical tracks for PC, TS, and M1, fin positions, and (not necessarily equidistant) horizontal tracks for M0 and M2. The finFETs, partitioned into p-FETs and


(a) Placement – containing power bus (green), fins (gray), PC (blue), TS (brown), FET boundaries (purple)

(b) Electrically correct routing, violating some DFM rules.

(c) Routing with optimal manufacturability and wire length.

Figure 2: Example results after placement and both routing phases. The green shapes on the top and bottom represent the power tracks of a circuit row.

n-FETs, have to be placed in two rows in between the power tracks (see Figure 2(a)). The electrical connectivity of the FETs and their size is described in a netlist. The task is to decide how many fingers a FET should use and to assign a location to each FET. Both choices are subject to the design rules and DFM constraints. Here, the width of the image is the most important optimization criterion as this determines the area of the cell on the chip. Given a placement of FETs, the goal in routing is to find an embedding of rectilinear Steiner trees which realizes the given netlist. This has to be done meeting the design rules and DFM constraints, as well. As overall goal in routing, we minimize weighted net length, with the topmost available layer M2 being more expensive than other layers. Other objectives (e.g. pin access [18], number of vias, or electromigration reliability [19, 12]) can easily be included as well.

A crucial point to obtain high quality cell layouts is for the placement algorithm to have a very good estimate of where and how much free area is needed in the routing step. If the placement is too pessimistic this will result in a waste of space, whereas a too optimistic placement is not routable. We have designed an objective function for the placement step that very accurately estimates the quality of a placement with respect to later routability (see Section 2.1). In many cases we can prove that our placement solution is *optimal* with respect to our objective function.

Simple sequential rip-up and reroute approaches turned out to fail for most of our placements. Instead we use an approach that allows to route all nets *simultaneously* and consider many design rules and DFM constraints already while building up the nets. The latter is required because only few design rule violations can be fixed after routing due to the limited space of our compact placements. We also have successfully extended our approach to *multi-bit* cells (see Section 2.5).

In Section 2, we discuss the placement algorithm, followed by the routing solution in Section 3. Section 4 reports the results of our implementation on cells at the 7nm technology node.

1.1 Related Work

Most previous work on cell layout only focuses on subproblems or restricted versions of the general cell layout problem and is not directly applicable to 7nm layouts. Moreover, design rules and DFM requirements are more restrictive in 7nm than in previous 14nm/15nm [9] and 10nm finFET technology nodes.

Many different approaches have been suggested for the transistor placement problem. In [2, 8] combinatorial algorithms are presented that optimize the cell area, but assume a given transistor folding. [4] use an integer programming approach that optimizes cell area and includes transistor folding but assumes the possibility to pair n-FETs and p-FETs. In [14] a branch-and-bound approach is used for transistor placement that allows to optimize additional objectives, but transistor folding is not considered.

The placement algorithm has to consider the complicated dependencies between positions of n-FETs and p-FETs that arise due to the design rules of the SADP cut mask for the PC layer. Several approaches to handle SADP in automated design have been suggested [17, 19]. Our approach differs in that we allow variable gate widths during cut mask generation and guarantee to find valid solutions (if existent) in polynomial runtime. To the best of our knowledge, our placement algorithm is the first to guarantee a legal SADP layer decomposition without wasting cell area (see Section 2.6).

For intra cell routing many different approaches are known. Traditional channel routing [15] and simple rip-up and reroute strategies [10] fail in recent technologies. More successful are SAT-based [13] and the closely related integer-programming based [18] approaches. In these approaches a set of candidate solutions is generated for each net. A SAT-formulation or an integer linear program is used to select one realization for each net so that all design rules are met.

Our routing approach is also based on an integer programming formulation. However, we do not need to pre-compute candidate solutions for each net but instead generate all possible routings for all nets simultaneously while packing them. Using this approach we do not have to restrict the candidate solutions in advance (e.g. by restricting them to lie within some bounding box around the terminals) but are able to consider all possible routings. While this makes the search space much larger, our well chosen integer programming formulation turns out to be quickly solvable by standard MIP solvers. Typically, cells with up to 15 FETs are solved within a few minutes.

2. PLACEMENT

The input of the cell placement problem is a set $\mathcal{F}$ of FETs, a set $\mathcal{N}$ of nets and a large number of technology-specific constraints. A *FET* is characterized by a tuple $(W, N_g, N_s, N_d, n_s, t)$, where

- $W \in \mathbb{N}$ is the total width of the gate measured in the number of fins it intersects,
- N_g, N_s, and N_d are the nets connected to gate, source, and drain, respectively,
- $t \in \{\text{n-FET}, \text{p-FET}\}$ is the FET's type.

We allow FETs to be folded, i.e. realized with different numbers of *fingers*. Therefore, solving the placement problem does not only include the assignment of locations to each transistor but also deciding how many fingers should be used. The total width W of a FET can be distributed to a number of fingers. Using only one finger, the FET is realized with one gate, intersecting W fins. Using a larger number of fingers, the FET is realized with several gates, located next to each other, which in total intersect W fins. If, for example, the width of a FET is 6 (measured in fins) it can be realized with 1, 2, 3, and 6 fingers, each covering 6, 3, 2, and 1 fin respectively. Additionally, the user can allow rounding, which means that the same FET can also be realized with 4 and 5 fingers, covering 2 and 1 fins respectively. Note that in this case the width of the implemented FET is only approximately the specified width. Depending on the used cell image, some fin configurations can be forbidden, e.g. most images do not allow fingers covering a single fin only. A FET realized with f fingers has f gates and $f + 1$ source and drain contacts. A FET with several fingers connects source and drain nets alternately. The placement algorithm is also allowed to swap FETs. In this case, the source and drain contacts of the FET exchange their places. Figure 3 shows the same FET realized in three different ways.

All gates are manufactured with self-aligned double patterning (SADP). In the first step, a regular pattern of unidirectional poly (PC) shapes is generated. In the second step, these shapes are cut off by the cut (CT) mask, leaving the desired gates. Not all placements admit a legal layer decomposition. Situations for which no legal cut mask exists are detected by our placement algorithm and excluded from the search tree as soon as possible.

The output of the placement algorithm consists of

- FET locations $(x, y) : \mathcal{F} \to \mathbb{R}^2$,
- finger numbers $\phi : \mathcal{F} \to \mathbb{N}_{>0}$, and
- swap status $s : \mathcal{F} \to \{\text{yes}, \text{no}\}$.

This information is then passed to the routing algorithm (see Section 3). The transistors are arranged in two *stacks*, one next to each power rail. One stack consists of the cell's n-FETs and is placed directly next to the lower power rail, whereas the other stack contains the p-FETs and is placed directly next to the upper power rail. Our program is also capable to realize multi-bit cells. These cells occupy multiple circuit rows and have several pairs of stacks placed upon each other. For the moment, we focus on single-bit instances with two stacks, more details on our multi-bit implementation will be given in Section 2.5.

The main design rules that have to be obeyed during placement are horizontal and vertical distance rules.

1. A function d specifies the minimum size of gaps between FETs in the following way: If F_2 is the right neighbor of F_1 on one of the stacks, then the gates must have a distance of at least $d := d(F_1, \phi(F_1), s(F_1), F_2, \phi(F_2), s(F_2)) \in \{1, 3\}$ tracks. For $d = 1$, the rightmost contact of F_1 may overlap the leftmost contact of F_2 and for $d = 3$ the FETs are separated by two empty PC tracks. In the first case the diffusion regions overlap and the contact is used simultaneously by both FETs.

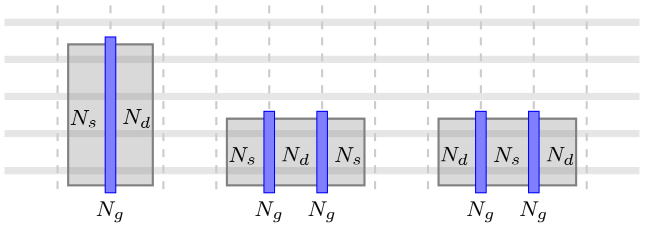


Figure 3: A FET of width 4 realized with 1 finger, 2 fingers, and 2 fingers swapped. Gates are shown in blue, Source and drain contacts in gray.

2. The SADP manufacturing process for the gates uses a cut mask (CT). The design rules for CT require that there is enough vertical space between FETs with different gate nets (see Section 2.6). Additionally, gates cannot be connected on the diffusion region but only in between both stacks. Therefore, enough vertical space between the FETs must be reserved for their connection as well.

2.1 Objective Function

We want to find placements which are small and as "routable" as possible. To do so, we use an efficiently computable model to measure the quality of a placement P involving the following two values:

- $W(P)$, defined as the *width* of the placement, i.e. the number of required PC tracks,
- The *gate-gate net length*, $\text{ggnl}(P) := \sum_{N \in \mathcal{N}} \text{ggnl}(N)$, where $\text{ggnl}(N)$ is the width of the bounding box of all gate contacts in N.

The algorithm returns a placement which respects all design rules (including vertical constraints) and additionally globally minimizes the placement width $W(P)$. If there are several such placements the algorithm chooses a placement among them with the minimum gate-gate net length. This does not guarantee routability but turns out to be a very good indicator for real-world 7nm cells. A very similar objective function was already used in [1]. In [17] the number of required M2 tracks is used as secondary objective.

2.2 Placement Algorithm

Our placement algorithm, as outlined in algorithms 1–3, implements at its core a recursive enumeration of all possible placements that backtracks as soon as the current (partial) solution cannot be part of a placement that is better than the best placement that has been found so far. We search for legal solutions with increasing cell width. The minimum width of a single stack, ignoring the constraints imposed by the other stack, can be calculated very efficiently. The minimum width of both stacks are lower bounds for the minimum width of the entire cell. Therefore, we take the maximum $W_{\max}$ of both lower bounds and start by looking for solutions

Algorithm 1 TwoStackPlacement

```
1: for all $S \in \{N, P\}$ do
2:     $W_{\max}^{S} \leftarrow 0$
3:     while EnumerateStack($\mathcal{F}_S, W_{\max}^{S}$) $= \emptyset$ do
4:         $W_{\max} \leftarrow W_{\max} + 1$
5:     end while
6: end for
7: $W_{\max}^{\text{init}} \leftarrow \max(W_{\max}^{P}, W_{\max}^{N})$
8: for $W_{\max} = W_{\max}^{\text{init}}, W_{\max}^{\text{init}} + 1, \ldots$ do
9:     Enumerate placements on both stacks with width at
       most $W_{\max}$.
10:    if found optimal placement $P$ then
11:        return $P$
12:    end if
13: end for
```

Algorithm 2 EnumerateStack($\mathcal{F}, W_{\max}$)

```
1: $\kappa \leftarrow 0$
2: $\mathcal{P} \leftarrow \emptyset$
3: while LowerBoundAddFingers($\kappa$) $\le W_{\max}$ do
4:     for all possibilities to add $\kappa$ fingers to $\mathcal{F}$ do
5:         StackRecursion($\mathcal{F}$)
6:     end for
7:     $\kappa \leftarrow \kappa + 1$
8: end while
9: return $\mathcal{P}$
```

whose width is at most $W_{\max}$. This describes lines 1–7 of Algorithm 1. If no solution can be found with this restriction, the cell width is increased and we repeat this procedure until a placement is found. This means that parts of the search tree are visited multiple times. However, this does not cause a significant performance penalty, since the running time of the last iteration usually dominates the running time of all previous iterations.

2.3 Placing a Single Stack

An important subroutine of our placement method is EnumerateStack (Algorithm 2). Its input is a set $\mathcal{F}$ of unplaced FETs and a width restriction $W_{\max}$. It outputs a list of all legal placements with width at most $W_{\max}$, or that no such placement exists.

EnumerateStack is implemented by distributing fingers to the FETs. Initially, all FETs start with the minimum number of fingers they can have. Afterwards, an increasing number of fingers is distributed to all FETs, trying every distribution. Before assigning an additional finger to the FETs, we use a fast routine to calculate a lower bound for the width of placements with κ additional fingers. We can exit the loop as soon as this lower bound exceeds the width bound $W_{\max}$. Once all fingers are distributed, StackRecursion (Algorithm 3) is called.

Algorithm 3 is called recursively. In the beginning some of the FETs have already been placed. We keep the positions of these FETs fixed and compute a lower bound for the total cell width where all FETs are placed. The minimum width needed by a set of FETs with fixed number of fingers equals the total number of fingers plus 2 additional tracks for each gap that has to be inserted between two FETs. For our lower bound calculation we use that a gap has to be inserted if the nets of the rightmost contact of the left FET

Algorithm 3 StackRecursion($\mathcal{F}_u$)

```
1: if LowerBound($\mathcal{F}_u$) $> W_{\max}$ then
2:     return
3: else if $\mathcal{F}_u = \emptyset$ then
4:     $\mathcal{P} \leftarrow \mathcal{P} \cup \{$current placement$\}$
5: else
6:     for all $F \in \mathcal{F}_u$ do
7:         Place $F$ on the leftmost possible track
8:         StackRecursion($\mathcal{F}_u \setminus \{F\}$)
9:         Unplace $F$
10:        Swap $F$ and repeat lines 7 – 9
11:    end for
12: end if
```

and the leftmost contact of the right FET are not the same. We count the number of times a net connects a leftmost or rightmost FET contact and denote it by C^N. If C^N is odd, it is not possible to place all contacts of this net next to each other, avoiding gaps. However, the additional gap can be avoided by placing one contact of the net at the cell boundary. This can at most be done for two nets, one on the left and the other on the right cell boundary. This gives the following lower bound for the cell width.

$$W_{\text{lb}}(\mathcal{F}_u) = W(\mathcal{F} \setminus \mathcal{F}_u) + \sum_{F \in \mathcal{F}_u} \phi(F) + 2 \cdot \max\{0, n_{\text{odd}} - 2\},$$

where $W(\mathcal{F} \setminus \mathcal{F}_u)$ is the width of the already placed FETs, $\sum_{F \in \mathcal{F}_u} \phi(F)$ the number of fingers yet to place, and n_{odd} the number of nets with C^N odd.

If this bound does not exceed the width limit $W_{\max}$, we recursively place FETs in the stack from left to right, trying every permutation of FETs. Given a partial solution, it takes every yet unplaced FET and places it at the smallest legal horizontal coordinate on top of the partial solution – once unswapped and once swapped.

If only a single width-minimal placement (not necessarily optimal w.r.t. the entire objective function) is required, as in line 3 of Algorithm 1, a faster version of EnumerateStack is used that considers only partial solutions with a specific block structure.

2.4 Placing Both Stacks

The placement algorithm on an entire cell is done by placing one stack after the other. In contrast to single stack placement where only horizontal constraints have to be obeyed, vertical constraints are important for two stack placement as well.

We fix a cell width and enumerate placements for both stacks (Line 9 of Algorithm 1). This is done by enumerating all placements with width at most $W_{\max}$ on the first stack, and for each of these placements enumerating placements with width at most $W_{\max}$ on the second stack. For the second stack, an extended version of Algorithm 2 is used. This extended version checks the vertical constraints between the two stacks. Additionally, it is checked whether the space between the transistor rows is large enough to allow gate wiring and a legal PC cut mask. Quickly deciding whether a legal PC cut mask exists is a non-trivial problem and will later be discussed in some detail (see Section 2.6). To speed up the algorithm, this is already checked for a partial placement of

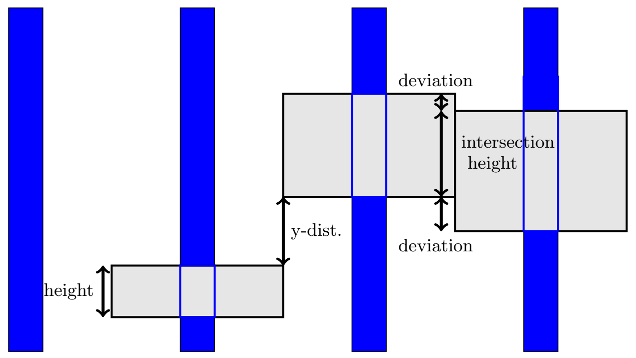


Figure 4: Cut mask rules. PC in blue, cut mask in gray. Height of each cut shape has to be at least d. y-distance between cut shapes on neighboring tracks has to be at least v or cut shapes have to touch. In the second case their deviation has to be at most s and their intersection height at least d.

the second stack, allowing to prune entire branches of the search tree.

Additional pruning can be accomplished since we are only interested in a single optimum solution. We calculate lower bounds for ggnl(.) and wnl(.) given a partial placement. If these values exceed the upper bound given by an already found placement we can prune all placements containing these partial placements.

2.5 Multi-Bit Cells

Very large cells are typically not implemented on one but several neighboring circuit rows. Such *multi-bit* cells can also be placed with our program. To place a cell with k bits, we first compute candidates for assignments of FETs to the bits. We then evaluate the quality of these assignments (by their number of bit-crossing connections, etc.) and solve the most promising candidates with a variation of Algorithm 1. The bits are placed one after the other with each new placement respecting constraints due to already placed bits.

2.6 Cut Shapes

Gates, i.e. poly conductors (PC) manufactured by a single exposure lithographic process suffer from major drawbacks in modern technology nodes [7]. One issue is that the spacing of printed line-ends becomes too large to allow dense packing of transistors. To overcome this effect, self-aligned double patterning (SADP) is used to produce lines and line-ends separately. This reduces line end roughness (LER), creates straighter PC lines [7] and doubles the density of PC stripes, allowing a smaller PC pitch.

Using this double patterning technique means that design rules have to be obeyed for a cut (CT) mask which removes extraneous features. We assume that all gates are placed on a regular pattern, where all PC lines are parallel to each other and have equal distance. We call the possible positions of gates *tracks*. The CT mask is then used to remove undesired features, i.e. to remove a connection between two transistors. It is composed of several rectilinear cut shapes which have to obey certain rules. These rules are depicted in Figure 4.

Given some (partial) non-overlapping transistor placement and gate routing we need to decide if a legal CT mask exists. If two transistors on different stacks using the same gate track need to be disconnected, we need a cut shape in between them. The exact position of the cut shape is not important to obtain an LVS clean solution but matters regarding cut shape design rules. This leads to the following formal definition of the Cut Shapes problem.

Cut Shapes

Instance: Boundary intervals $[l_1, u_1], \dots, [l_n, u_n] \subseteq \mathbb{N}$, minimum height $d \in \mathbb{N}$, maximum deviation $s \in \mathbb{N}$, minimum y-distance $v \in \mathbb{N}$.

Task: Find cut shape intervals $[x_1, y_1], \dots, [x_n, y_n] \subseteq \mathbb{N}$, s.t. $[x_i, y_i] \subseteq [l_i, u_i], y_i - x_i \geq d$, for $i = 1, \dots, n$ and either

1. $\begin{matrix} |x_i - x_{i-1}| \leq s, |y_i - y_{i-1}| \leq s, \\ |x_i - y_{i-1}| \geq d, |y_i - x_{i-1}| \geq d \end{matrix}$, or
2. $x_i \geq y_{i-1} + v$, or
3. $y_i \leq x_{i-1} - v$

for $i = 2, \dots n$.

It can be proven that it is enough to deal with the special case that no y-distance has to be left between neighboring cut shapes i.e. $v = 0$:

Lemma 1. *An algorithm A which solves* Cut Shapes *instances for $v = 0$ in time T can be used to solve arbitrary* Cut Shapes *instances with runtime $T + \mathcal{O}(n)$.*

A fast algorithm for Cut Shapes is important as it is called many times during the placement algorithm and dominates its total running time. In the following, we present a polynomial time algorithm. The idea is to use a dynamic programming approach, going through the instance track by track. Since the number of coordinates inside an interval $[l_i, u_i]$ is in general exponential in the input size, we have an exponential number of states resulting in an exponential runtime. To reduce the number of states, we show that a polynomially sized subset of coordinates always contains a solution. We formally state this result after introducing some notation.

Definition 1. *The coordinate sum of a feasible solution $[x_i, y_i]_{i=1,\dots,n}$ is defined as $\sum_{i=1}^{n}(x_i + y_i)$. A solution is called uppermost optimal if no other solution with larger coordinate sum exists.*

Theorem 1. *For a given instance of the cut shape problem, let $B := \{l_1, \dots, l_n, u_1, \dots, u_n\}$ be the set of boundary coordinates and $[x_i, y_i]_{i=1,\dots,n}$ an uppermost optimal solution. Furthermore, using the Minkowski sum and product, let $B^\star := B + \{0, -d, d, -s, s\}^{2n}$. Then $x_i, y_i \in B^\star$ for $i = 1, \dots, n$.*

We will not give a formal proof here, but intuitively any legal solution can be "shifted upwards" until it becomes uppermost optimal. One can then show that the coordinates used by this new solution are a subset of $B^\star$.

Remark 1. *The size of $B^\star$ is polynomially bounded in the number of tracks n. We have $B^\star := B + \{0, -d, d, -s, s\}^{2n}$, $|B| = \mathcal{O}(n)$, and $|\{0, -d, d, -s, s\}^{2n}| = \mathcal{O}(n)$. Therefore, $|B^\star| = \mathcal{O}(n^2)$.*

Algorithm 4 Cut Shapes

```
1: for $x_1, y_1 \in B^\star \cap [l_1, u_1]$ with $y_1 - x_1 \geq d$ do
2:     Set $[x_1, y_1]$ as solution of $P_1(x_1, y_1)$
3: end for
4: for $i = 2, \dots, n$ do
5:     for $x_i, y_i \in B^\star \cap [l_i, u_i]$ with $y_i - x_i \geq d$ do
6:         for $[x_{i-1}, y_{i-1}]$ s.t. $P_{i-1}(x_{i-1}, y_{i-1})$ has a solution
           and $[x_{i-1}, y_{i-1}], [x_i, y_i]$ are legal neighbors do
7:             Set $[x_{i-1}, y_{i-1}], [x_i, y_i]$ as solution of $P_i(x_i, y_i)$
8:         end for
9:     end for
10: end for
11: Pick legal cut shape on track $n$ and use backtracking to
    obtain entire solution.
```

Definition 2. *Let $P_i(\bar{x}, \bar{y})$ be the problem instance restricted to tracks $1, \dots, i$, with the additional constraints $x_i = \bar{x}$, $y_i = \bar{y}$.*

As shown by Theorem 1, it suffices to search for optimal solutions on the coordinate set $B^\star$. This motivates Algorithm 4.

Theorem 2. *Algorithm 4 solves* Cut Shapes *optimally and can be implemented with runtime $\mathcal{O}(nk^4)$, where $k := |B^\star|$. Using Remark 1, this gives a runtime of $\mathcal{O}(n^9)$.*

Remark 2. *By iterating over $x_i, y_i, x_{i-1}, y_{i-1}$ in lines 5 to 9 more cleverly, the runtime of these lines can be improved to $\mathcal{O}(k^2)$. This gives a total runtime of $\mathcal{O}(nk^2) = \mathcal{O}(n^5)$.*

In practice the number of y-coordinates on which cut shapes can start or end is limited. In our application there are about 500 possible coordinates which gives a constant upper bound on k. Therefore, Algorithm 4 has a runtime of $\mathcal{O}(n)$ in practice.

3. ROUTING

In order to solve the routing problem on a placed cell, we use a mixed integer programming (MIP) approach.

Next to the input already given for the placement problem (Section 2), the cell routing problem expects the location of each FET from the placement. Furthermore, external connections may be present for a net, together with a desired location for this external input and output. During routing, our goal is to minimize the wire length and number of vias in order to optimize the power, timing, and yield properties of the cell.

3.1 Grid Graph Construction

Since each wiring layer only allows either vertical or horizontal wires, we represent the cell routing space by a three-dimensional grid graph $G = (V, E)$ with edge costs. For each layer, we are given a set of routing tracks specifying feasible positions for wires which are not necessarily equidistant.

By intersecting routing tracks on adjacent layers, we obtain the vertex set V. The edge set E consists both of line segments connecting adjacent intersections on the same layer as well as vias between stacked vertices on adjacent layers.

Edge costs are given by multiplying their geometric length by a layer-specific constant. This allows to e.g. increase costs on M2 in order to leave more space for inter-cell routing, and to trade off wire length against the number of vias.

On this graph, we are seeking a minimum-cost Steiner tree packing that contains for each net a tree connecting its terminals, which are given by the gate, source, and drain contacts as well as its external pins. Furthermore, the packing is subject to additional constraints.

3.2 Design Rules and Coloring

For the wiring within a cell, design rules fall into two basic categories: *Diff-net rules* require a certain minimum distance between wires that belong to different nets. *Same-net rules* are in place to avoid geometric configurations with features below the lithographic capabilities and resolution, and to reserve space for optical proximity correction (OPC). While diff-net rules are most important, same-net rules have become more and more important with each new technology. Especially in routing, it is particularly important to obey all these rules already during the routing algorithm because it is not possible to fix errors in post-processing due to a lack of space.

All features are manufactured using multiple masks in order to increase packing density: Shapes on different masks are allowed to have a smaller distance than shapes on the same mask. Hence, a valid routing does not only consist of a disjoint Steiner tree packing, but also requires features on such layers to be assigned to masks such that certain design rules are met. We call this assignment *coloring*. However, in 7nm technology this only affects vias, since all wires are colored using an alternating track-based coloring scheme.

3.3 Mixed Integer Programming Formulation

First, we describe the core MIP we use to model the Steiner tree packing problem in graphs. Then, in Section 3.4, we explain how design rules are incorporated into the model.

We ensure connectivity by adding for each net a relaxation of the Steiner tree problem in graphs to the model. In [3], the undirected cut relaxation is used for that purpose: For each cut $X \subseteq V$ separating the terminal set, the undirected cut relaxation requires at least one edge in $E(X, V \setminus X)$ to be contained in the Steiner tree. However, the undirected cut relaxation has an integrality gap of 2, which is already asymptotically attained in the special case that G is a circuit, even if all vertices are terminals. One can strengthen the relaxation by using a bidirected auxiliary graph $G' = (V, A)$ with $A = \{(i,j) : \{i,j\} \in E\}$ which contains two opposing edges (i,j) and (j,i) for each original edge $\{i,j\} \in E(G)$. Let r be an arbitrary root terminal and add variables for all edges $e \in A$. Then, for each cut $r \in X \subseteq V$ that does not contain all terminals, require that at least one edge leaving the cut is used. Finally, lower bound the usage of each original edge $\{i,j\} \in E$ by the *sum* of the usages of both directed edges (i,j) and (j,i). This relaxation is called bidirected cut relaxation. The integrality gap of the bidirected cut relaxation is unknown, the worst known example due to Skutella (reported in [6]) has an integrality gap of $\frac{8}{7}$.

By introducing additional flow variables, one can eliminate the exponential number of cut constraints, resulting in the multicommodity flow relaxation, first introduced in [16]. This relaxation is equivalent to the bidirected cut relaxation [11] and was already used in [5] to solve Steiner tree packing problems. We will also use the multicommodity flow relaxation:

For each net $k \in \mathcal{N}$, we denote by $T_k \subseteq V$ the set of its terminals. A decision variable x_e^k is introduced for each net $k \in \mathcal{N}$ specifying whether k is using an edge $e \in E$ of the graph G. For each net k, we choose an arbitrary root terminal $r_k \in T_k$ and denote the set of sink terminals $T_k \setminus \{r_k\}$ by S_k. Then, the multicommodity flow relaxation introduces a commodity for each sink $t \in S_k$ and requires a flow of one unit of the commodity from r_k to t to be supported by x^k. More precisely, for each net $k \in \mathcal{N}$, terminal $t \in S_k$ and directed edge $(i,j) \in A$, a flow variable f_{ij}^t is introduced representing the flow of the commodity t along the directed edge (i,j). Then, we add flow conservation constraints at vertices in $V \setminus \{t, r_k\}$ and enforce that r_k sends one unit of flow and that t receives one unit of flow of commodity t. Furthermore, for each net k and each edge $(i,j) \in A$, we add directed edge usage variables $\vec{x}_{ij}^k$ that upper bound the directed flow variables f_{ij}^t and require $\vec{x}_{ij}^k + \vec{x}_{ji}^k \leq x_{\{i,j\}}^k$. For ease of notation, we combine the usage of each edge $e \in E(G)$ also to a single variable x_e. Also, we denote by $f^t(v) := f^t(\delta^+(v)) - f^t(\delta^-(v))$ the flow balance of sink t at a vertex $v \in V(G)$. The complete model is as follows:

$$
\begin{array}{llll}
\min & \sum\limits_{e \in E} c_e x_e & & \\
\text{s.t.} & x_e & = \sum\limits_{k \in \mathcal{N}} x_e^k & \forall\, e \in E \\
 & x_e & \in \{0,1\} & \forall\, e \in E \\
 & x_e^k & \in \{0,1\} & \forall\, e \in E, k \in \mathcal{N} \\
 & f^t(v) & = \begin{cases} 1 & \text{if } i = r_k \\ -1 & \text{if } i = t \\ 0 & \text{else} \end{cases} & \\
 & & & \forall\, v \in V, k \in \mathcal{N}, t \in S_k \\
0 & \leq f_{ij}^t & \leq \vec{x}_{ij}^k & \forall\, (i,j) \in A, k \in \mathcal{N}, t \in S_k \\
\vec{x}_{ij}^k + \vec{x}_{ji}^k & & \leq x_{\{i,j\}}^k & \forall\, \{i,j\} \in E, k \in \mathcal{N}
\end{array}
$$

In this basic formulation, no additional constraints, especially with respect to distances between wires resulting from the Steiner tree, are taken into consideration. Furthermore, two nets may actually share the same vertex, but not the same edge. Further constraints ensuring correctness in this sense, but also modeling coloring and the additional DRC constraints, are presented next.

3.4 Mapping DRC Constraints

To ensure vertex disjointness, for each $k \in \mathcal{N}$ and vertex $v \in V(G)$, we add a vertex usage variable $x_v^k \in \{0,1\}$. Then, for each net $k \in \mathcal{N}$ and $v \in e \in E(G)$, we add the constraint $x_e^k \leq x_v^k$, and add $\sum_{k \in \mathcal{N}} x_v^k \leq 1$ for all $v \in V(G)$.

On via layers, we need to assign colors to used edges. To that end, for each such edge $e \in E$, net $k \in \mathcal{N}$ and color $m \in M$, we add a binary variable ${}^m x_e^k$ and enforce

$$x_e^k = \sum_{m \in M} {}^m x_e^k.$$

Moreover, for each such edge e and color m, we add a binary variable ${}^m x_e = \sum_{k \in \mathcal{N}} {}^m x_e^k$, representing whether edge e is used with color m by any net.

The transistor placement already induces forbidden edges which are immediately mapped to the respective variables.

The basic distance rules are then mapped as follows. Suppose that if an edge $e \in E$ is used by the wiring of net $k \in \mathcal{N}$, then a nearby edge $e' \in E$ cannot be used by another net. The inequality

$$x_e^k + \sum_{i \in \mathcal{N} \setminus \{k\}} x_{e'}^i \leq 1$$

prohibits this situation. All basic diff-net distance rules can be modeled this way, including via distances and the inter-layer via rules that prescribe minimum distances between vias in adjacent via layers. Simple same-color and diff-color spacing constraints can be modeled as well, using the corresponding ${}^m x_e^k$ variables. We also add some of these distance constraints for segments of the same net, as there are also same-net rules for spacing between non-adjacent segments of the same net.

An important same-net rule requires that each wire must have a certain minimum length, depending on its layer. Let e, e' be adjacent edges and assume that e is used and e' is not used. In order to fulfill the required minimum length, a set F of edges must also be used, c.f. Figure 5. We model this implication by the constraint

$$\sum_{f \in F} x_f \geq |F| \cdot (x_e - x_{e'}).$$

Clearly, in the situation depicted above, this constraint requires $x_f = 1$ for all $f \in F$, and is non-binding otherwise.

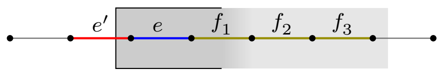


Figure 5: Minimum length rule implementation: If e is used and e' is not used, then f_1, f_2 and f_3 must also be used.

Other rules like minimum via overhangs are modeled in a similar way.

To route a cell, we first solve the model described in Section 3.3 together with vertex-disjointness constraints to obtain an electrically correct routing. Then, we add the additional variables and constraints described in Section 3.4 and re-solve, which yields an optimum routing obeying the design rules and DFM constraints.

4. EXPERIMENTAL RESULTS

The described algorithms are implemented and tested on real-world 7nm instances. Table 1 reports on runtime and quality of our results, split into placement and routing part, for several characteristic current 7nm cells. All experiments were done single-threaded on a 2.20GHz Intel Xeon E5-2699 v4 machine using CPLEX 12.6 as MIP-solver.

For the first 9 instances shown in Table 1 a placement with provably smallest possible area was found in a few seconds. Placements with optimal routability were found after at most 2 min and their optimality was proven after a maximum of 5 min. Cells 10 and 11 were placed with provably minimal area after less than 2 min. Here placements with improved routability were found after up to 7 min. While it could not be proven within a time limit of 60 min, it is still possible that the found placements were already globally optimal w.r.t. routability.

All cells were routed in two phases. First, an electrically correct routing was computed which took up to 30 s on the first 9 instances and 2 min and 11 min on cells 10 and 11, respectively. This solution can already be used for tasks

Table 1: Results on 7nm testbed: $|\mathcal{F}|$ number of fets, $|\mathcal{N}|$ number of nets, w cell width in tracks, t_1 time until an area optimal placement has been found, t_2 time until a placement with optimal routability has been found, t_3 time until optimal routability has been proven, t_4 time until electrically correct routing has been found, t_5 time until routing with optimal manufacturability and wire length has been found. M_2 number of used M2 tracks. All times in [mm:ss].

Cell			Placement				Routing		
#	$\|\mathcal{F}\|$	$\|\mathcal{N}\|$	w	t_1	t_2	t_3	t_4	t_5	M_2
1	8	11	6	0:00	0:00	0:00	0:03	0:06	0
2	8	10	6	0:00	0:00	0:00	0:05	0:08	0
3	8	11	12	0:00	0:00	0:13	0:08	0:16	0
4	8	11	12	0:00	0:00	0:26	0:11	0:26	0
5	14	16	12	0:00	1:27	4:28	0:09	0:39	1
6	8	11	16	0:01	1:38	2:37	0:24	2:42	0
7	17	22	12	0:00	1:00	2:33	0:09	0:17	0
8	11	15	12	0:03	0:19	0:36	0:09	0:20	0
9	8	11	16	0:08	1:03	1:04	0:28	2:08	0
10	14	18	30	0:38	6:47	–	1:48	18:36	1
11	8	11	44	1:09	1:13	–	11:04	33:28	0

that do not require optimal manufacturability, for example timing analysis. This allows a fast prototyping flow where changes in the input and their consequences on the layout can be tested quickly.

In the second phase additional rules were incorporated into the model to improve DFM. From the set of all routings fulfilling these additional constraints our router then found the solution with minimal net length. For the first 9 instances this took up to 3 min, cell 10 and 11 needed 19 min and 33 min, respectively. The final generated layout of cell 10 can be seen in Figure 6.

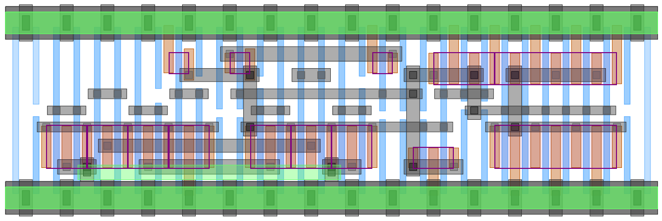


Figure 6: Generated layout of cell 10 with 30 tracks width. Only a single M2 wire (bottom left, green) has been used.

5. CONCLUSION

We have presented a new flow for the automatic cell layout that is able to deal with the challenges arising in 7nm technology. The main features are the global optimization of several design objectives, full integration of cut shape computation into the placement algorithm, and an efficiently solvable two stage MIP formulation in routing.

Acknowledgment

The authors would like to thank our cooperation partners at IBM, in particular Gerhard Hellner, Iris Leefken, and Tobias Werner.

6. REFERENCES

[1] R. Bar-Yehuda, J. A. Feldman, R. Y. Pinter, and S. Wimer. Depth-first-search and dynamic programming algorithms for efficient CMOS cell generation. *IEEE Trans. CAD*, 8:737–743, 1989.

[2] B. Basaran and R. A. Rutenbar. An $O(n)$ algorithm for transistor stacking with performance constraints. In *Proc. DAC'96*, pages 221–226, 1996.

[3] M. Grötschel, A. Martin, and R. Weismantel. The Steiner tree packing problem in VLSI design. *Mathematical Programming*, 78:265–281, 1997.

[4] A. Gupta and J. P. Hayes. Optimal 2-D cell layout with integrated transistor folding. In *ICCAD'98*, pages 128–135. IEEE, 1998.

[5] N.-D. Hoàng and T. Koch. Steiner tree packing revisited. *Math Meth Oper Res*, 76(1):95–123, 2012.

[6] J. Könemann, D. Pritchard, and K. Tan. A partition-based relaxation for Steiner trees. *Mathematical Programming*, 127(2):345–370, 2011.

[7] K. Lai et al. 32 nm logic patterning options with immersion lithography. In *Proc. SPIE 6924, Optical Microlithography XXI, 69243C*, 2008.

[8] C. Lazzari, C. Santos, and R. Reis. A new transistor-level layout generation strategy for static CMOS circuits. In *ICECS'06*, pages 660–663, 2006.

[9] M. Martins et al. Open cell library in 15nm freePDK technology. In *ISPD'15*, pages 171–178. ACM, 2015.

[10] C. J. Poirier. Excellerator: Custom CMOS leaf cell layout generator. *IEEE Trans. CAD*, 8(7):744–755, 1989.

[11] T. Polzin. *Algorithms for the Steiner problem in networks*. PhD thesis, MPII Saarbrücken, 2003.

[12] G. Posser, V. Mishra, P. Jain, R. Reis, and S. S. Sapatnekar. Cell-internal electromigration: Analysis and pin placement based optimization. *IEEE Trans. CAD*, 35(2):220–231, 2016.

[13] N. Ryzhenko and S. Burns. Standard cell routing via boolean satisfiability. In *DAC'12*, pages 603–612, 2012.

[14] B. Taylor and L. Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *DAC'07*, pages 344–349. ACM, 2007.

[15] S. Wimer, R. Y. Pinter, and J. A. Feldman. Optimal chaining of CMOS transistors in a functional cell. *IEEE Trans. CAD*, 6(5):795–801, 1987.

[16] R. T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28(3):271–287, 1984.

[17] P.-H. Wu, M. P.-H. Lin, T.-C. Chen, T.-Y. Ho, Y.-C. Chen, S.-R. Siao, and S.-H. Lin. 1-D cell generation with printability enhancement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):419–432, 2013.

[18] W. Ye, B. Yu, Y.-C. Ban, L. Liebmann, and D. Z. Pan. Standard cell layout regularity and pin access optimization considering middle-of-line. In *GLSVLSI'15*, pages 289–294. ACM, 2015.

[19] B. Yu, X. Xu, S. Roy, Y. Lin, J. Ou, and D. Z. Pan. Design for manufacturability and reliability in extreme-scaling VLSI. *Science China Information Sciences*, 59(6):1–23, 2016.