

ANALOG LAYOUT SYNTHESIZER FOR A PARASITIC AWARE DESIGN LOOP

by

Ahmet Unutulmaz

B.S., Electrical and Electronics Engineering, Boğaziçi University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University
2009

ANALOG LAYOUT SYNTHESIZER FOR A PARASITIC AWARE DESIGN LOOP

APPROVED BY:

Prof. Günhan Dündar
(Thesis Supervisor)

Prof. Levent Akın

Asst. Prof. Arda D. Yalçınkaya

DATE OF APPROVAL: 28.01.2009

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Günhan Dündar and Professor Francisco Fernandez for their invaluable guidance and help during the preparation of this dissertation. I would like to mention their patience, giving me inspiration and hope when I was stuck at dead-ends.

I also would like to thank to my family for their infinite patience, support and love. Besides, I thank them very much for everything they sacrificed.

Finally, I would like to thank to TÜBİTAK for its support through BİDEB-2210.

ABSTRACT

ANALOG LAYOUT SYNTHESIZER FOR A PARASITIC AWARE DESIGN LOOP

Analog design automation is being studied for a couple of decades and many researchers developed their own tools. However, there are no standards for these tools. The work done in this M.S. thesis aims to define standard interfaces for layout automation tools, making the integration of different tools possible. In addition to these interfaces, an interface for a layout database is defined. This interface is designed to hold any kind of layout structure and to cooperate with any kind of layout tool.

Additionally, implementations for these interfaces are done. A floor-planner tool, a device generator, a fast database and a simple router are implemented in Java. The implemented tools are combined and a template based layout synthesizer is constructed. This layout synthesizer requires templates files coded in Java. Implementing the floor-planner, a new floor-plan representation and a new inequality solver are developed and used.

Moreover, a novel synthesis loop is defined. In this synthesis strategy, contrary to the old synthesis approaches, the effects of the parasitic are considered in the synthesis loop. The layout synthesizer implemented in this thesis is preferable in this novel synthesis loop, due to the very short running time.

ÖZET

PARASİTİK ETKİLERİ İÇEREN BİR TASARIM DÖNGÜSÜ İÇİN ANALOG SERİM SENTEZLEYİCİ

Analog devre tasarımı son yirmi yıldır üzerinde çalışılan bir konudur ve birçok araştırmacı bu alanda kendi otomasyon aracını geliştirmiştir. Maalesef, bu araçlar için tasarlanmış standart bir ara-yüz yoktur. Bu yüksek lisans tezi, tasarım otomasyon araçları için ara-yüzler tanımlamayı hedefliyor. Tasarım araçları için tanımlanan ara-yüzlere ek olarak, serim veri tabanı için de bir ara-yüz tanımlanmıştır. Bu ara-yüz herhangi bir yapıyı tutacak ve herhangi bir serim aracı ile uyumlu olacak şekilde tasarlanmıştır.

Ek olarak, bahsedilen bu ara-yüzler gerçekleştirilmiştir. Yerleşim aracı, cihaz sentezleyici, hızlı bir serim veritabanı ve basit bir bağlayıcı Java kodlama dilinde gerçekleştirilmiştir. Gerçeklenen bu parçalar birleştirilerek, otomatik serim sentezleyici oluşturulmuştur. Bu sentezleyici Java dilinde yazılmış şablonlarla çalışmaktadır. Yerleşim aracını gerçeklemek için yeni bir yerleşim gösterimi ve yeni bir eşitsizlik çözücü ortaya konulmuştur.

Bütün bunlara ek olarak, yeni bir sentez döngüsü tanımlanmıştır. Bu yeni döngüde parazit etkiler sentez döngüsünün içinde değerlendirilmektedir. Bu tezde geliştirilen otomatik serim sentezleyici hızlı çalışması sebebi için tanımlanan yeni tasarım döngüsünde kullanılabilir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	ix
LIST OF TABLES	xii
LIST OF SYMBOLS/ABBREVIATIONS	xiv
1. INTRODUCTION	1
2. DESIGN TOOLS	3
2.1. Design Flow	3
2.2. Automation Tools	3
2.2.1. Numerical Simulators	4
2.2.2. Symbolic Analyzers of Analog Circuits	4
2.2.3. Analog Circuit Synthesizers and Optimizers	5
2.2.4. Analog Layout Synthesis	6
2.2.5. Yield Estimation and Optimization	7
3. LAYOUT DATABASE	9
3.1. Data Structures	9
3.1.1. List of Blocks Structure	10
3.1.2. Bin-Based Structure	10
3.1.3. Neighbor Pointers Structure	10
3.1.4. Corner Stitching Structure	10
3.2. Interface	12
3.2.1. Building Instances from a Master	13
3.2.2. Supporting Grouping and Hierarchy	13
3.2.3. Transforming Coordinates	13
3.3. Implementation	14
4. FLOOR-PLANNER	17
4.1. Constraints	18
4.1.1. Absolute Constraints	19
4.1.2. Relative Constraints	19

4.1.2.1.	Alignment Constraints	19
4.1.2.2.	Abutment Constraints	19
4.1.3.	Symmetry Constraints	20
4.2.	Floor Plan Representation	21
4.2.1.	Slicing Structures	21
4.2.2.	Non-Slicing Structures	22
4.2.2.1.	Sequence Pair	22
4.2.2.2.	O-Tree	24
4.2.2.3.	B*-Tree	26
4.2.2.4.	BSG	26
4.2.2.5.	TCG	26
4.2.2.6.	CBL	28
4.2.3.	P-Sequence	28
4.2.3.1.	Compaction of P-Sequence	29
4.2.3.2.	Analysis of P-Sequence	31
4.3.	Layout Description Script (LDS)	31
4.3.1.	Alignment Commands	31
4.3.2.	Abutment Commands	31
4.3.3.	Insertion Commands	32
4.3.4.	Command for General Constraints	32
4.3.5.	Update Commands	33
4.3.6.	Basic Commands	34
4.4.	Constraints and Inequalities	34
4.4.1.	Classification of Constraints	34
4.4.1.1.	Soft Constraints	34
4.4.1.2.	Hard Constraints	35
4.4.2.	Constraints and Inequalities	35
4.4.3.	Inequalities and Graphs	36
4.4.4.	Solving the Inequalities	36
4.4.5.	Loop Breaking	37
4.5.	Floor-planner Implementation	38
5.	DEVICE GENERATOR AND ROUTER	40
5.1.	Interface	40

5.1.1. Example	41
5.2. Devices	41
5.2.1. NMOS Transistor	41
5.2.2. PMOS Transistor	42
5.3. Routing	42
5.3.1. Classification of Routers	43
5.3.1.1. Area Routing	44
5.3.1.2. Channel Routing	44
5.3.2. Required Capabilities	45
5.3.3. Routing Styles	46
6. IMPORTER AND EXPORTER	47
6.1. Importer	47
6.2. Exporter	48
7. LAYOUT GENERATOR	49
8. CONCLUSION AND FUTURE WORK	53
8.1. Conclusion	53
8.2. Future Work	54
APPENDIX A: COMPUTATIONAL COMPLEXITY	55
APPENDIX B: ALGORITHMIC GRAPHS	57
APPENDIX C	60
REFERENCES	71

LIST OF FIGURES

Figure 2.1.	Design flow	3
Figure 3.1.	Linked list of blocks	10
Figure 3.2.	Bin-based representation	11
Figure 3.3.	Neighbor pointers structure	11
Figure 3.4.	Corner stitches	11
Figure 3.5.	Transformation is needed	13
Figure 3.6.	Transformations	14
Figure 3.7.	Fast Transformations	15
Figure 3.8.	Structure of the database	16
Figure 4.1.	Absolute constraint	19
Figure 4.2.	Relative constraint	19
Figure 4.3.	Alignment constraints (a) alignment of the left boundaries, (b) alignment of the centers,(c) alignment of the right boundaries	20
Figure 4.4.	Abutment constraint	20
Figure 4.5.	Symmetry constraint	20
Figure 4.6.	Slicing floor-plan structure	21

Figure 4.7.	Floor-plan for the sequence pair: $(\Gamma^+, \Gamma^-) = (abdecf, cbfaed)$	23
Figure 4.8.	An O-tree representation	25
Figure 4.9.	A B*-tree representation	26
Figure 4.10.	Bounded Slicing Grid	27
Figure 4.11.	Transitive Closure Graphs	27
Figure 4.12.	Floor-plan is represented as $(S, L, T) = (fcegbad, 001100, 001010010)$	28
Figure 4.13.	Modules are projected onto x and y axis	29
Figure 4.14.	Floor-plan is represented as $(H, V) = (\{baif, b, aie, g, e, h\}, \{bg, hg, ba, ai, e, i, ehf\})$	30
Figure 4.15.	Four modules separate the axis at most to 7 pieces.	31
Figure 4.16.	Constraints in the floor-planer	35
Figure 4.17.	(a) The plan and the constraints (b) The representation of constraints in a graph	37
Figure 4.18.	Block diagram of the loop breaker	37
Figure 4.19.	Sample signals (visited edges)	38
Figure 4.20.	Periodicity in the signal	38
Figure 5.1.	NMOS Transistor	42
Figure 5.2.	PMOS Transistor	43

Figure 5.3.	Template based routing	43
Figure 6.1.	Design loop	47
Figure 7.1.	Differential pair	49
Figure 7.2.	Code for the template	49
Figure 7.3.	Template based construction	50
Figure 7.4.	Block diagram of the layout generator	50
Figure 7.5.	SPICE code	51
Figure 7.6.	SPICE code	51
Figure 7.7.	Constructed layout (spice code in Figure 7.5)	51
Figure 7.8.	Constructed layout (spice code in Figure 7.6)	52

LIST OF TABLES

Table 4.1.	Decoding the Sequence Pair representation	23
Table 4.2:	Required types of perturbations for SP representation	24
Table 4.3.	Required types of perturbations for the O-tree representation	25
Table 4.4.	Perturbations inspired from TCG	27
Table 4.5.	Perturbations	28
Table 4.6.	Commands for alignment	32
Table 4.7.	Command for abutment	32
Table 4.8.	Module insertion	33
Table 4.9.	Adding general constraints	33
Table 4.10.	Command for updating	33
Table 4.11.	Basic commands	34
Table 5.1.	Interface for adding new devices	40

Table 5.2.	Device types for CMOS process	40
Table 5.3.	Interface to get the ports	41
Table 6.1.	Formats for exchange	48
Table 6.2.	Interface for the importer	48
Table 6.3.	Interface for the exporter	48

LIST OF SYMBOLS/ABBREVIATIONS

TOLAS	Tool Oriented Layout Automation System
EDA	Electronic Design Automation
Diff	Differential Pair

1. INTRODUCTION

Advances in technology have provided the opportunity of integrating more devices into a single chip. This opportunity is becoming more difficult to utilize with every technology generation. Fortunately, in the digital world, the designers can utilize this opportunity and cope with a large number of transistors, using design automation tools. Development of these tools requires extensive amount of research on the topic. Since 1970's many researchers have been trying to advance the electronic design automation (EDA) tools. Note also that, the systematic structures in the digital circuitry ease the development of digital EDA tools. Unfortunately, when analog design is considered, there is almost no mature automation tool in the field.

One can argue that analog circuitry contains at most a few hundred transistors and can be designed by a few engineers without the need of an automation tool. However, one should consider that the VLSI IC industry tries to maximize its profit and a main consideration is the time to market. A very fast response to the market will increment the profit considerably. Unfortunately, without a design automation tool, a single design cycle may not be adequate for the designer to complete the design, and time consuming trials may be required. Additionally the SOC designs (System on Chip) are very popular and more analog circuitry is being integrated.

Many researchers are trying to develop EDA tools to ease the design challenge of the analog world. Some EDA tools contain hard-coded synthesizers which construct the circuitry through the instructions in the code. Another group of tools try to construct templates and use them for further synthesis. Also, there are tools searching for the solution: heuristic and optimization based tools. However, there is not a complete solution for analog design automation. Unfortunately, due to the lack of a standard in the field, it is almost impossible to combine the available tools to obtain a complete solution.

An aim of this thesis is to define standard interfaces for the layout automation tools, making the integration of different tools possible. These interfaces are defined by reviewing literature about related tools. In addition to these interfaces, an interface for a

database is defined. This interface is designed to hold any kind of layout structure and to cooperate with any kind of tool (floor-planner, router, etc...).

During the thesis study, an implementation of a floor-planner and an implementation of a layout database were performed. These implementations were written in Java and they comply with the related standards defined. Further, the codes are combined into a template based layout synthesizer.

While implementing the floor-planner, a new floor-plan representation (*P-Sequence*) is developed and used. Also a new inequality solver is designed and used in the implemented floor-planner.

Moreover, a novel synthesis loop is defined. In this synthesis strategy, contrary to the old synthesis approaches, the effects of the parasitic are considered in the synthesis loop. The layout synthesizer implemented in this thesis is preferable in this novel synthesis loop, due to the very short running time (a few seconds).

In section 2, some of the available design tools and approaches are introduced. In section 3, the interface for the layout database and an implementation of this interface are presented. Section 4 defines the interface for the floor-planner and includes a floor-planner implementation. This section also reviews some common floor-plan representations and includes the description of P-Sequence representation. Additionally, an inequality solver is presented in this section. In section 5, a device generator and a template-based router are described. Section 6 contains information about the exchange formats between the available EDA tools. All of the implemented tools are combined and a layout synthesizer is constructed, this synthesizer is described in section 7. Section 8 concludes this work and proposes some future work.

2. DESIGN TOOLS

2.1. Design Flow

Analog design is a time consuming process; many iterations are required to finalize the design. In general, the design is separated into three parts. These parts are *behavioral description*, *structural description* and *physical description*.

A design is described behaviorally through a HDL (hardware description language), such as VHDL-AMS. This description is then converted to structural description, which includes the circuitry as blocks. These blocks are implemented with transistors and this implementation is called the physical description.

The design flow is schematically shown in Figure 2.1.

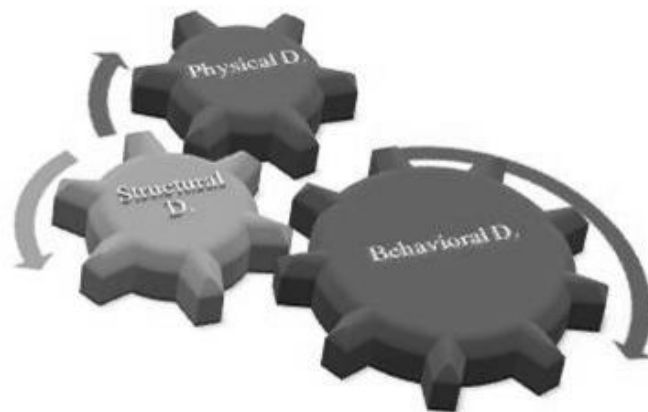


Figure 2.1. Design flow

2.2. Automation Tools

Researchers developed many tools for EDA and these tools are classified according to their functionality in [1], [2], [3] and [4]. Also a brief review for some of these tools is going to be presented in the following sub sections.

2.2.1. Numerical Simulators

Hand analysis of analog circuits is a complicated task and involves solving many differential equations simultaneously. As the circuit includes many devices, it becomes almost impossible to analyze the circuitry by hand. Fortunately, the early development of analog simulators helped the designer through the design of today's complicated analog circuits.

Analog simulation is a critical task for design verification. Verification is needed to avoid performance degradation of an analog circuit after fabrication. The degradation should be accounted by the designer during the design process and analog simulators with circuit extractors give this opportunity to the designer.

Circuit simulation began with the development of the SPICE tool and nowadays there are many commercially available variants of SPICE. These circuit simulators are used in the design and verification processes. The circuit simulations may require considerable amount of time. To speed up the simulation of a module, including many analog circuits, *circuit models* are derived. The VHDL-AMS or Verilog-A are the standardized analog description languages. These circuit models not only speed up the simulation processes but also make it possible to simulate a group of analog circuitry without an implementation at the transistor level.

There also exist dedicated simulators for special purposes; i.e. switched capacitor simulation tools, RF simulation tools, etc...

2.2.2. Symbolic Analyzers of Analog Circuits

Designing analog circuitry requires an insight into the circuitry. Thus, it is desirable to have the design equations of a circuit. For circuits of few transistors, one can derive the required equations (i.e. small signal gain, bandwidth, etc...) by hand. However, as more transistors are added to the circuit, it will not be practical to analyze the behavior of the circuit by hand. Symbolic analyzers, capable of handling circuits with around 10 transistors, emerged late 1980's. These tools were simplifying the expressions as done in

the hand analysis. However, these tools were not able to handle complicated circuits due to the extensive computational requirements. A new technique of *simplification before and during the symbolic expression generation* reduced the computation time considerably and enabled the symbolic analysis of practical circuits.

2.2.3. Analog Circuit Synthesizers and Optimizers

Topology selection is a very important problem in analog design. This problem arises when there are multiple topologies for an implementation. A good topology meets the specifications for the circuit at the minimal implementation cost (i.e. Power, Area, Robustness ...). Unfortunately, it is hard to formulate the solution of the topology selection problem quantitatively. Thus, the primary synthesis tools used rule sets for the selection of the best topology.

Later approaches constructed more quantitative ways to select a topology. Feasible performance space of each topology is defined. According to the specs, the best matching topology is selected and optimization techniques are also applied to the problem.

Once a topology is selected, the next step is the *performance translation*. According to the performance specifications of the overall circuit, the performance specifications of the sub blocks are determined. These sub-blocks may be composed of single devices, then the problem can be stated as the *device sizing* problem. Extracting the device sizes from the given performance specifications is an under constrained problem. The degrees of freedom in the problem can be handled by knowledge based and optimization based approaches.

In the knowledge based approaches, the design knowledge is coded into a computer executable form. This method uses the heuristic methods. Most of the knowledge based synthesis tools require design plans. Unfortunately, construction of these design plans is more time consuming than simply designing the circuit.

On the other hand, optimization approaches are based on numerical optimization techniques. These techniques try to solve the degrees of freedom of a design, under the constraint of the performance specifications. Optimization based approaches also try to

extract the required design knowledge using analysis techniques (i.e. symbolic analysis) or try to use equation-free simulation-oriented approaches. Running an optimization loop depends upon evaluating the performance of an instance circuit. This performance evaluation is done by evaluating the design equations or by simulating the circuit.

Although evaluation of the performance through circuit simulations does not require design plans and precise design equations, it requires high computational power and advanced numerical algorithms. In recent years, with the development of high speed computers, this approach started to become popular.

2.2.4. Analog Layout Synthesis

Due to the adaptation of the digital layout synthesis techniques to the analog counterpart, analog layout synthesis is the most mature part of the analog design automation. The goal of the layout synthesizer is to convert the schematic of a circuit into a producible layout. A fully automated layout synthesizer is expected to floor-plan, place and route the circuit elements by taking the performance considerations into account.

The earliest approaches rely on *procedural module generation*, which is based on pre-coded software. The software constructs the layout according to the input parameters. For this approach, development and maintenance of a module generator for each circuit topology is necessary. *Template driven module generation* is a similar approach and uses geometric templates for each circuit. The template fixes the position and the routing of the circuit. Template driven approach is a fast synthesis method. This approach is suitable, if the layout structure is given and changes from circuit to circuit are minor.

Due to the parasitic effects induced in analog layouts, the resulting circuit may not satisfy the performance specifications. Sometimes, big changes in the layout may be required for both satisfying the performance specs and having a compact layout. *Optimization-based macro-cell place and route layout generation* approach uses optimization techniques to optimally place and route to combine macro-cells. Macro-cell is defined to be a single device (i.e. transistor, capacitor) or a group of devices (matched pairs). This approach depends upon a pre-defined cost function and the optimizer which

tries to minimize the area and tries to satisfy the performance specs using this function. This approach requires large CPU times and the quality of the generated layout is sensitive to the introduced cost function. ILAC [5] and KOAN/ANAGRAM II [6] are samples for this approach which relies on macro-cells. The ILAC tool is inspired from digital automation tools; the problem with tool is that, it is not always possible to handle sensitive analog designs with the digital automation techniques. KOAN/ANAGRAM II tool set is a combination of KOAN which is a device generator and an optimization based placer, and ANAGRAM II which is a maze-style area router. Also tools like LADIES [7] and ALSYN [8] operate similarly.

The next generation of tools is based on *performance-driven or constraint-driven* approaches. These approaches try to quantitatively measure the performance degradations, considering the parasitic and secondary effect. The area routers ROAD [9] and ANAGRAM III [10], the placement tool PUPPY-A [11] and are examples of this approach. The LAYLA [4] tool also consists of performance-driven placer and router.

A new approach is to separate the device placement task into two separate tasks: device stacking followed by stack placement. By considering the circuit as a connected graph of sources and drains, the optimum merging is observed, called device stacking. In the stack placement phase, the devices are merged and placed accordingly. Also separate placing and routing phases result in non-optimal solutions. Thus, there are attempts to simultaneously place and route the circuits.

The performance degradation due to the layout parasitic is also considered. The affect of parasitic to analog layout synthesis is mentioned in [12], [13] and [14].

2.2.5. Yield Estimation and Optimization

Due to fluctuations in the fabrication process, the performance of fabricated circuits may vary. These variations may result in malfunctioning or badly performing circuits, which do not meet the performance specifications. The circuits need to be designed in such a way that maximizes the number of the functional circuits; this is also known as *design for*

manufacturability. As well as designing for manufacturability, the designs have to be robust and that is called *design for robustness or design for quality*.

It is desired to know the fluctuations and the acceptability region for the parameter space or for the performance space. Then, according to the statistical fluctuations, the statistics for the well-functioning circuits can easily be obtained. However, the statistical fluctuations in the design parameters in the performance space and the acceptability region in the parameter space are not known. A simple approach is the *worst-case analysis*; as the name implies, some combination of the device parameters are used to calculate the worst case performance. A statistical approach is *Monte-Carlo simulations*. In this approach, according to the statistics of the parameter space, many samples of the device are generated and the statistics for the performance space is obtained. This approach is a CPU intensive process; some approximations are present to speed up this process.

3. LAYOUT DATABASE

A database is defined to be the core of an analog layout tool. Thus, the implementation becomes very critical, if the speed of the layout tool is considered. To code layout automation tools compatible with the database, it is required to define a general interface for the database.

This chapter defines the necessary functionality for a layout database. After a detailed literature review, interface for the database is defined. Operations needed by the database and their corresponding definitions are presented. Also, an implementation for the defined interface is performed.

3.1. Data Structures

The database structure for an analog layout is in essence with a floor-plan structure. However, there are major differences. The database holds separate layers such as poly, diffusion, etc... On the other hand, the floor-plan structure holds the blocks; i.e. transistors, capacitors, matched pairs, etc...

The data structures for a database can be classified into two main groups; the *absolute coordinate based database* and *relative coordinate based database*. The absolute coordinate based database holds the absolute positions of the elements, as well as the layout properties. Thus, the database returns the coordinates immediately; however, the neighborhood information for the elements is not present in the database. On the other hand, the relative coordinate based database hold the neighborhood information (the local information). However, it cannot immediately return the coordinates of the blocks. Thus, layout manipulations (move, remove, etc...) cannot be immediately applied. In this database, the global information is first transformed into local information.

Some common structures are listed in [15]. The *List of Blocks Structure* holds the absolute coordinates of the elements in a list. Also, the *Bin Based Structure* holds absolute

coordinates in a more compact way. The *Neighbor Pointers Structure* and the *Corner Stitching Structure* hold relative positions of the elements.

3.1.1. List of Blocks Structure

This structure handles the components of a layout in a linked-list. The positions of the components are recorded into a structure and this structure is then added into the linked-list. The structure of the list is depicted in Figure 3.1.

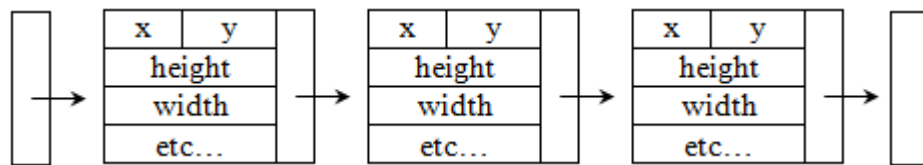


Figure 3.1. Linked list of blocks

3.1.2. Bin-Based Structure

Bin-based layout representation, superimposes a virtual grid over the layout. A bin $B(m,n)$ contains the list of elements it includes. In Figure 3.2, the bin $B(2,2)$ contains the components in the set S ($S = \{A, C, B\}$). These bins are held in a two-dimensional array.

3.1.3. Neighbor Pointers Structure

The list of blocks and the bin based methods do not hold the local information. In contrast, this structure holds pointers to keep local information. However, it is not easy to maintain this structure. A simple change in the layout may require all the pointers in the data structure be updated. Neighbor-pointers structure is symbolically depicted in Figure 3.3.

3.1.4. Corner Stitching Structure

Corner stitching data structures hold relative positions similar to the neighbor pointers data structure. However, they could be manipulated much faster than the neighbor

pointers data structure. Unlike the previous structure, this structure holds only four pointers. These point the components around the lower-left and upper-right corner.

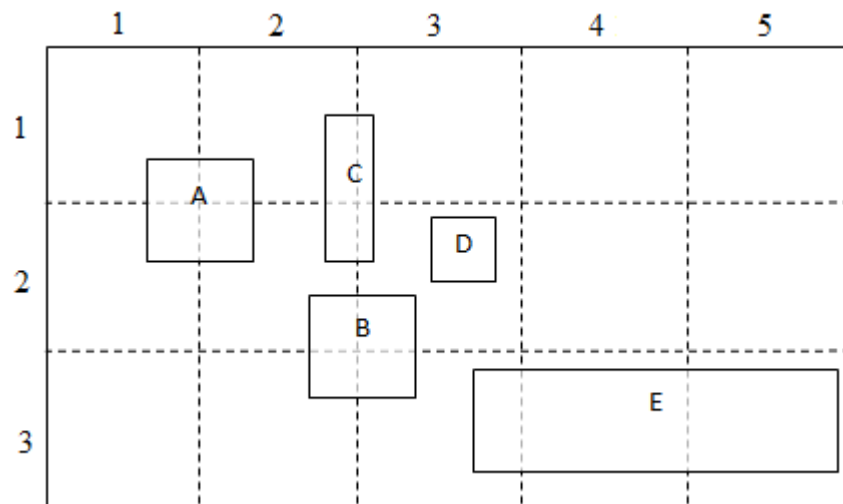


Figure 3.2. Bin-based representation

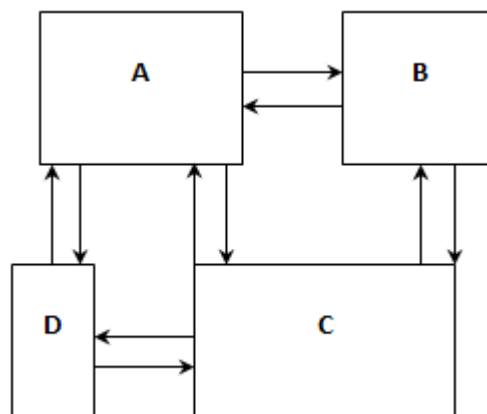


Figure 3.3. Neighbor pointers structure

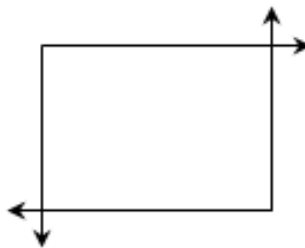


Figure 3.4. Corner stitches

3.2. Interface

Through the new framework for the layout automation; router, floor-planner, circuit extractor or any code manipulating the layout is called a tool. All the tools communicate with the layout through an interface; existence of this interface facilitates compatibility. Tool development will be much easier and any new tool will cooperate easily with the previously written codes. This will yield faster code development and code reusability.

Also, one can implement his/her own database, obeying the rules in the interface. If the new implementation is correctly coded, the new database will also work with the previously developed tools. The required functionalities for a database are stated below:

1. Add new element or remove old one,
2. Set or to get the properties of an element (i.e. layer, shape, etc...),
3. Rotate or flip an element.

In addition to the listed core functions the following functionality is added to the interface. These functionalities make the database compatible with digital automation tools

1. Build instances from a master
2. Support grouping and hierarchy
3. Transform coordinates from a frame to another

3.2.1. Building Instances from a Master

It may be desirable to clone an element, for instance to construct an array of transistors or to construct a symmetrical layout. The database interface supports generation of new elements from an initially generated element; the initial one is called a master. This functionality makes it possible to construct a layout from objects (like an object oriented language). The objects may be transistors, current mirrors or differential pairs; however unlike object oriented languages the instances are exact copies of the master.

3.2.2. Supporting Grouping and Hierarchy

The database interface can be used to group some elements; such as the transistors in a differential pair. This functionality contributes the concept of hierarchy into the database; the user may define a hierarchy in the layout. Moving a group will move all the components inside the group, which will speed up most of the floor-plan functionalities.

3.2.3. Transforming Coordinates

If a hierarchy is built in the layout, it will be required to calculate the positions of an element with respect to hierarchical frames. In Figure 3.5, the coordinate of the point in the outer frame can be calculated by summing the coordinate of the inner frame with the coordinate of the point. This is the simplest case and it may be more complicated if rotation and flip operations are included.

There are two types of transformations (except translation), these are rotation and flipping. These transformations are done by multiplying the coordinates of a point by the related matrix in Figure 3.6.

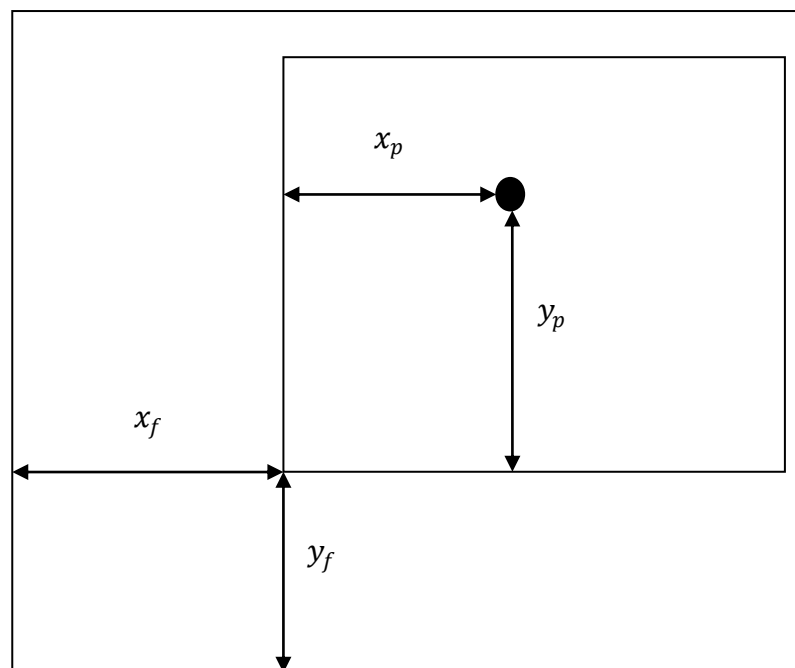


Figure 3.5. Transformation is needed

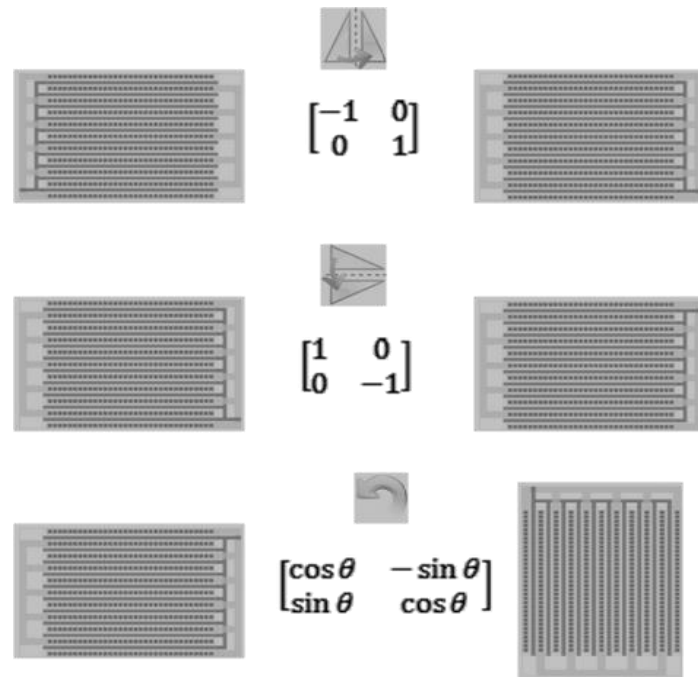


Figure 3.6. Transformations

3.3. Implementation

The implementation of the interface supports hierarchical structures. A group (module) is a parent and contains elements (transistors, capacitors, wires, etc...). An element may also be a parent, thus the implementation is capable of holding any degree of hierarchy.

A mathematical formulation of the implementation is given; the formulation is based on a previous work [16]. The layout is composed of groups (group of circuit elements or a single element). A set of groups is denoted by $G = \{g_1, g_2, \dots, g_n\}$. A group consists of many elements; for instance, in a CMOS process, the layout will be composed of elements such as: diffusion region, metal region, poly region, etc... The set of elements are denoted by $E(g) = \{e_1, e_2, \dots\}$. Similarly for a process, the set of available layers is denoted by $L = \{l_1, l_2, \dots, l_k\}$. Then the layer to which an element (e_i) belongs is denoted by $l(e_i) \in L$ and similarly the width and height of an element are denoted by $w(e_i)$ and $h(e_i)$, respectively. The absolute coordinates of an element (o_i) belonging to a group (g) is denoted by $(x(g) + x(e_i), y(g) + y(e_i))$; where $x(g)$ and $y(g)$ are the x and y coordinates of a group (g), respectively. The $x(e_i)$ and $y(e_i)$ are the relative coordinates of

an element (e_i) with respect to a group (g). In this representation, all the elements (circuit components, wires or any kind of structure) are denoted by groups (a collection of regions).

Generally, simple structures are involved in layouts; thus, the layout tool is designed to support rectangular shapes. Any elliptic shape is approximated by a polygon. The implemented tool also supports rectangular shapes and polygons. Due to the embedded objects, the code may be upgraded easily to elliptic shapes.

The implementation has the capability of rotating and flipping modules (groups). If a module is rotated or flipped, the entire sub modules are also affected; however, their coordinates are not modified. Thus, the database implementation runs extremely fast. The modifications are hierarchically stored. The coordinate of a module is calculated, when it is asked. This property speeds the floor-planner tremendously. Thus, a floor-planner may also run over the database, without holding a data structure.

The transformations may be calculated through the given transformation matrix. However, the sine and cosine functions -in the matrix- are time consuming and rarely required. Through the available tools, only simple rotations are involved. Thus to speed up the code, the sine and cosine are replaced by conditional code according to Figure 3.7.

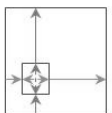
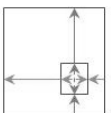
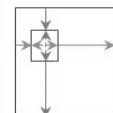
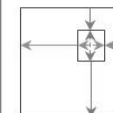
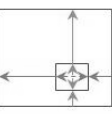
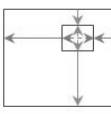
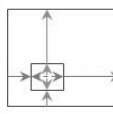
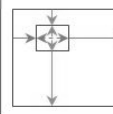
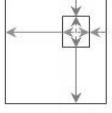
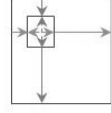
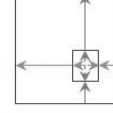
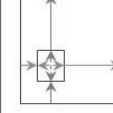
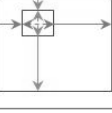
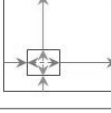
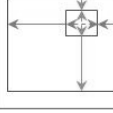
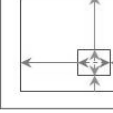
	No Flip	H. Flip	V. Flip	H. & V. Flip
0°				
90°				
180°				
270°				

Figure 3.7. Fast Transformations

The database is graphically illustrated in Figure 3.8. The elements are held in the database and when these elements are called through the interface, they are processed. The processing unit applies the related transformations to these elements.

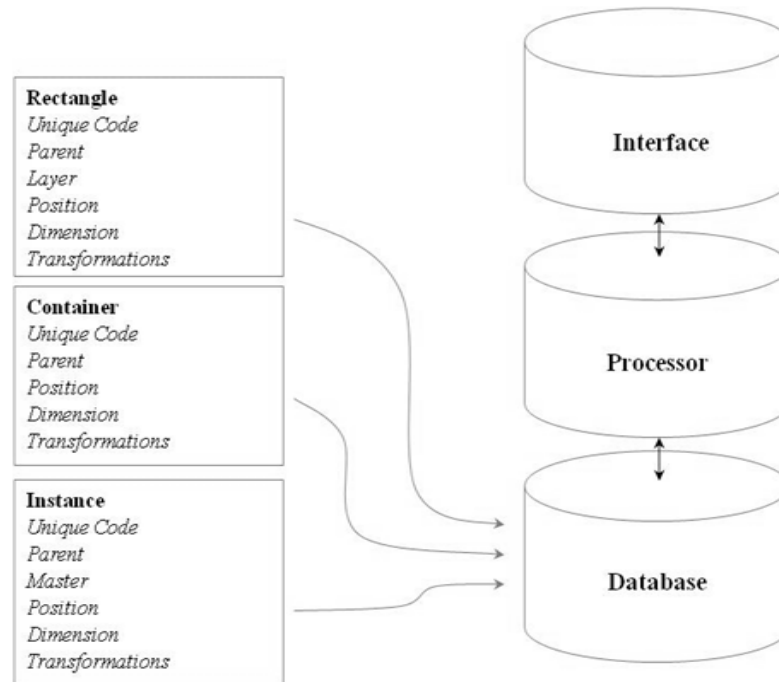


Figure 3.8. Structure of the database

4. FLOOR-PLANNER

The positions of the modules in a layout affect the performance of the circuit. Due to the omitted parasitic effects, the performance of the circuit degrades from the targeted value. These effects are the parasitic wire resistances and capacitances, potential variations in the bulk material, cross-talk effects etc... These parasitic wire resistances and capacitances can be minimized by carefully placing the modules in the layout. The placement of the modules in the layout is called floor-planning. Generally, floor-planners focus on the placement of the modules (transistors, macro-cells, capacitors, etc...), rather than placing each layer (poly regions, metal regions, and contacts). This speeds up the search for a solution by reducing the solution space. There are some approaches also trying to place the layers. The approach in [17] formulates the layout as equations. These equations and the cost functions are solved with a non-linear optimizer.

Some approaches consider the floor-planning problem as a *rectangular packing problem* (RP) [18]. With M being a set of rectangular modules, the RP is to place the modules in M without any overlaps into a minimum bounding rectangle. This approach only aims to minimize the chip area. Another approach considers the wire length, pre-place blocks and the density of the floor-plan [19]. The approaches in [20] and [21] optimize the wire length and the area. The approach in [21] also considers the density. Moreover, some approaches consider parasitic affects [17].

This chapter defines a general interface that can represent any floor-planner. The interface is also compatible with optimizers. The floor-plan can be modified not only by the internal optimizer but also by different optimizers, i.e. the optimizer of the router, etc...

The interface is constructed through a detailed literature review about the floor-planning structures and optimizers.

Additionally, an implementation for the floor-planner interface is presented. This implementation does not include an optimizer. However, it reads the placement from a template.

4.1. Constraints

It is important for the designer to control the absolute positions of some blocks, such as I-O pads. The designer also desires to control the relative positions of modules, i.e. symmetric modules. These constraints are needed to be passed through the floor-planner. Thus, the interface of the floor-planner should be capable of holding such constraints. In general, the constraints can be grouped as follows:

1. Absolute positions
2. Ranges for module positions
3. Positions with respect to the boundaries
4. Alignment of modules
5. Abutment of modules
6. Clustering of modules
7. Symmetric placement of modules

A method to handle these placement constraints is presented in [22]. Below, a short description of the idea, to hold these constraints, is stated.

For two blocks A and B, the horizontal and vertical displacement between the lower left corners of the blocks are denoted by $h(A,B)$ and $v(A,B)$, respectively. The displacement may be positive or negative. If the lower left corner of A is to the left of that of B, the displacement $h(A,B)$ will be positive. Similarly the vertical displacement ($v(A,B)$) can take positive or negative values. The relative placement constraints (alignment, abutting, clustering and symmetry constraints) between these modules (A, B) can be written as $v(A,B) = [\alpha, \beta]$ where α, β are real numbers and $\alpha \leq \beta$, meaning that the vertical displacement between A and B is in the interval $[\alpha, \beta]$. Absolute placement constraints (absolute positions, position ranges with respect to the boundaries) are handled by replacing A or B with LL, RR, BB and TT. These correspond to left, right, bottom and top boundaries of the chip. For instance $h(A, RR)$ denotes the horizontal displacement of the module A from the right boundary.

4.1.1. Absolute Constraints

These constraints are used to impose the position of a module with respect to boundaries. A sample constraint is depicted in Figure 4.1.

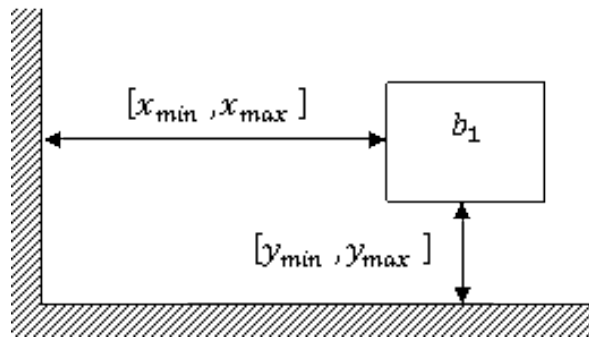


Figure 4.1. Absolute constraint

4.1.2. Relative Constraints

These constraints are used to describe the position of a module with respect to another module. The most general case -for relative constraints- is depicted in Figure 4.2. Special cases of the relative constraints are listed in the sub sections.

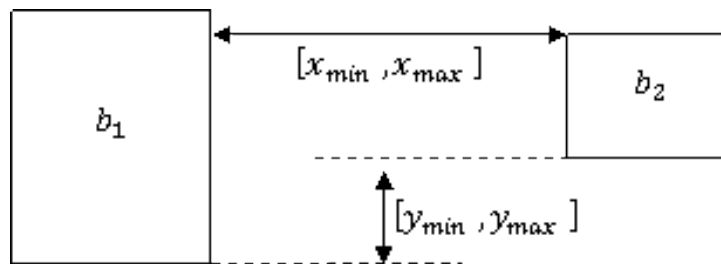


Figure 4.2. Relative constraint

4.1.2.1. Alignment Constraints. These are used to align the top, bottom, left or right boundaries of modules. A sample drawing is depicted in Figure 4.3.

4.1.2.2. Abutment Constraints. These types of constraints are used to bond two modules. It is possible to abut the left boundary of an element with the right boundary of another one

or top boundary of an element with the bottom boundary of another one. In Figure 4.4, the top of module b_1 is abutted with the bottom of module b_2 .

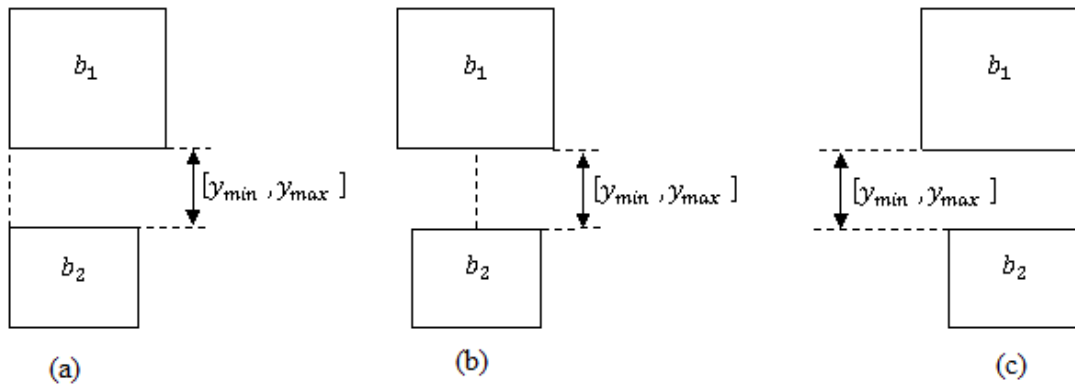


Figure 4.3. Alignment constraints (a) alignment of the left boundaries, (b) alignment of the centers, (c) alignment of the right boundaries

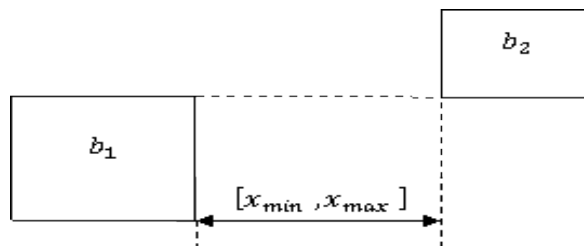


Figure 4.4. Abutment constraint

4.1.3. Symmetry Constraints

These constraints guarantee that two modules are symmetrically placed with respect to a reference point. A descriptive drawing is given in Figure 4.5.

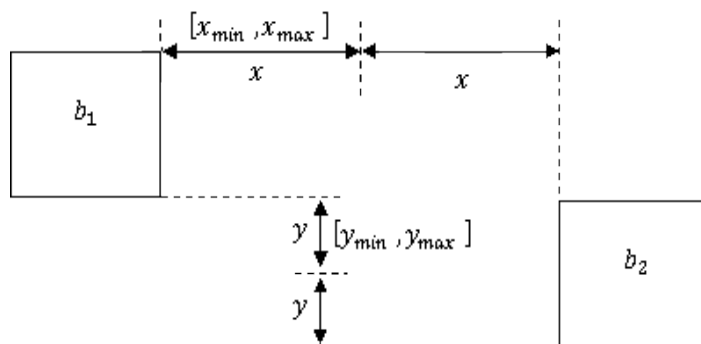


Figure 4.5. Symmetry constraint

4.2. Floor Plan Representation

There are many approaches to solve the problem of floor-planning. These approaches propose floor-plan structures to hold and manipulate the placement. In general floor-plans can be grouped into two; *slicing* and *non-slicing*.

4.2.1. Slicing Structures

Due to the complexity of the problem, hierarchical methods have been extensively used. For the slicing representation, Otten [23] proposed a binary tree representation. Then, a formal representation -normalized Polish expression- for this slicing structure was proposed by Wong and Liu [24]. A simple slicing floor-plan structure and the corresponding slicing tree representation are presented in Figure 4.6. The polish expression of the given floor-plan is (312*+654+**).

It is commonly believed that the slicing structure is not a complete representation. The structure seems not being capable of holding non-slicing floor-plan solutions. However, Lai and Wong [25] showed that applying many vertical and horizontal compactions, slicing tree representation can be converted into a non-slicing floor-plan having the minimum area.

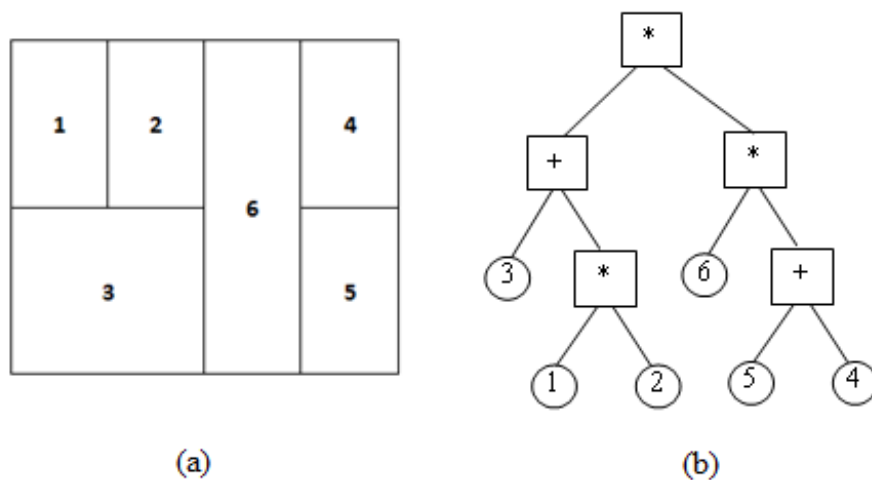


Figure 4.6. Slicing floor-plan structure

4.2.2. Non-Slicing Structures

Many non-slicing structures have been proposed, such as: *BSG* [26], *Sequence Pair* [18], *O-tree* [27], *B*-tree* [28], *TCG* [29], *CBL* [30] and *Q Sequence* [31], etc...

These structures are separated into sub categories in [32]. *BSG* and sequence pair can represent general floor-plans and describe relative positions between blocks, such as left-of, below, etc... The second category is composed of *O-tree* and *B*-tree*, and these describe relative positions of the blocks in 1-D. These two representations have smaller solution space however they do not directly hold the geometric relations between blocks. The last category contains the *CBL* and *Q-sequence*. These coding schemes are called *Mosaic* where the chip is divided into rectangular rooms. The mosaic structures cannot represent all kinds of floor-plans unless empty space is added.

4.2.2.1. Sequence Pair. A widely accepted and used floor-plan structure is the Sequence Pair [18]. This representation resembles the *above-below* and *left to-right to* representation in [33]. The difference is that each packing is represented by a pair of module name sequences (Γ_+ and Γ_-) in sequence pair representation

In the original paper, a set of requirements for the floor-plan representation are listed. First, the solution space -represented by the floor-plan structure- should be finite and every solution in the representation should be feasible. The paper lists more: every solution should be realizable in polynomial time. The last requirement is that the best solution evaluated in the representation, should be the optimum solution. The solution space satisfying these requirements is called *P-admissible*. It is also stated that the Sequence pair representation is *P-admissible*.

A floor-plan and the corresponding representation are presented in Figure 4.7. Γ_+ and Γ_- are ordered pair of module name sequences. A decoding scheme for the Sequence Pair is tabulated for any module x and y .

The packing (positions of the blocks) is calculated through the longest path algorithm on a vertex weighted directed acyclic graph. The vertices in this graph are the modules and the weights of the edges are the dimensions of the modules.

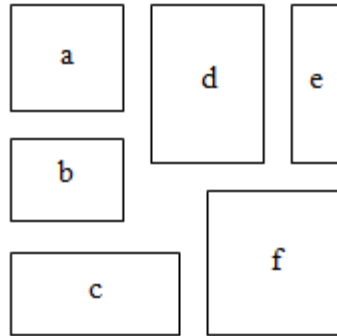


Figure 4.7. Floor-plan for the sequence pair: $(\Gamma_+, \Gamma_-) = (abdefc, cbfaed)$

Table 4.1. Decoding the Sequence Pair representation

Order in the sequences	Description
y is after x in both Γ_+ and Γ_-	y is right to x
x is after y in both Γ_+ and Γ_-	y is left to x
y is before x in Γ_+ and after x in Γ_-	y is above x
x is before y in Γ_+ and after y in Γ_-	y is below x

Optimization algorithms are used to search for the best floor-plan. These algorithms require perturbations in the layout to obtain the optimum placement. The original paper [18] presented two perturbations through searching the optimum layout. The applied perturbations to the layout are:

1. Changing the orientation either vertically or horizontally (this set includes 2^m perturbations)
2. Altering the order of the modules in the sequence pairs (this set includes $(m!)^2$ perturbations)

These perturbations have to be supported by the interface. The *iterate* function generates a new solution, the *isValid* function decides whether the solution is valid or not. The *getCost* function returns the cost of the floor-plan. All these functions have to be

defined in the implementation, however the perturb function should input the type of the perturbation. Required types for the perturbations for the given Sequence Pair implementation are tabulated.

Although just two perturbations are considered, finding the best placement is very time consuming. Two approaches are proposed to decrease the time complexity. The first approach is to speed up the calculations by special methods. The original packing algorithm has $O(n^2)$ complexity and a faster packing algorithm is presented in [34] with a complexity of $O(n \log n)$. The algorithm presented in [35] is based on *longest common subsequence* and obtains the coordinates of the modules and the total width and height of the floor-plan in $O(n \log \log n)$. The second approach, to speed up the search process, decreases the solution space, as done in [36].

These algorithms are directly related to the implementation. Solution space reduction should be implemented in the *autoPlace* function. This function runs the placement engine in the floor-planner. Note that the floor-plan implementation may not have a placement engine.

Table 4.2: Required types of perturbations for SP representation

Type of Perturbation	Description
PERTURB_ROTATE_90	Rotates the given module (equivalent to changing the orientation)
PERTURB_SWAP_P	Swaps the positions of two modules (there should not be constraints on the modules)

4.2.2.2. O-Tree. In [27] an ordered tree (O-tree) structure, to represent non-slicing floor-plans, is presented. In this paper, an admissible placement is defined; this placement is obtained through abutting the modules to the left and to the bottom. This process is called *compaction*. Also, in the paper, it is shown that the solution space of the O-tree representation is smaller than that of Sequence Pair.

A sample O-tree is given in Figure 4.8. The tree represents the horizontal placement; also, a tree for the vertical placement is required. Note that the O-tree represents compacted floor-plans.

In the original paper, a contour structure is used to reduce the required time during the conversion from the O-tree to constraint graph representation. Algorithms for conversion from the O-tree representation to the constraint graph representation and vice versa are also proposed.

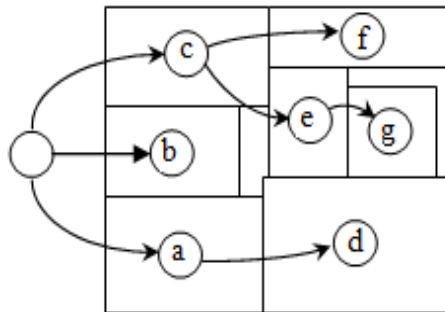


Figure 4.8. An O-tree representation

The perturbations used to search the solution space are deleting and insertion (see Table 4.3). However, these perturbations may not result in an admissible (compact) floor-plan. A post processing is needed to be applied to the resulting floor-plan. The post processing is done through the *postProcess* function. This function constructs an admissible tree and it is not included in the perturb function (This speeds up the algorithms including successive perturbations).

Table 4.3. Required types of perturbations for the O-tree representation

Type of Perturbation	Description
PERTURB_DELETE	Removes a module from the representation
PERTURB_INSERT	Inserts a module into the representation

4.2.2.3. B*-Tree. The B*-tree [28] is based on ordered binary trees. Similar to the O-tree, the B*-tree represents compact (admissible in [27]) floor-plans. The B*-tree is constructed according to *above* and *right to* relations (A sample B*-tree is given in Figure 4.9) and the operations search, insertion, and deletion are very fast compared to the O-tree representation.

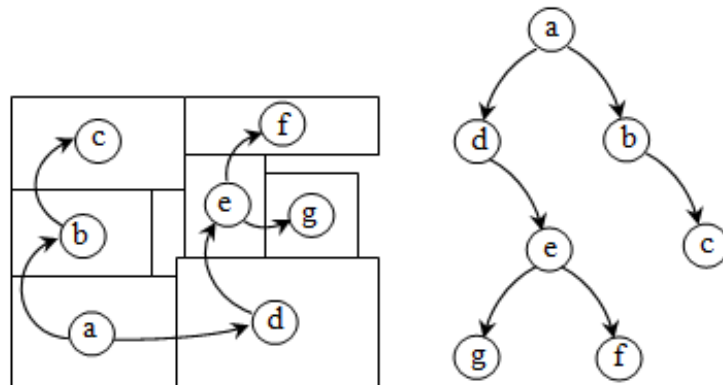


Figure 4.9. A B*-tree representation

The paper presents deleting and insertions as the floor-plan perturbations. The structure, in terms of the interface, is very similar to the O-tree representation. There are many B*-tree based floor-planners considering boundary constraints [37], pre-placed modules [38], etc...

4.2.2.4. BSG. Bounded slicing grid representation, presented in [26], cuts the plane into rooms with horizontal and vertical line segments. These rooms are represented by utilizing two directed graphs in [26]. The structure of the bounded slicing grid and a room are shown in Figure 4.10.

The BSG representation incurs redundancy [28], a packing may have multiple representations. The redundancy in the representation degrades the complexity of the search process. The BSG is extended to handle pre-placed modules and soft modules.

4.2.2.5. TCG. Transitive closure graph based representation is presented by Lin and Chang in [29]. TCG based representation holds the relative positions of the blocks in two TCG graphs (Figure 4.11). TCG based representation is reported to satisfy the four properties of

P-admissibility, defined in [18] (Also defined in section 4.2.2.1). The packing is calculated using the longest path algorithm.

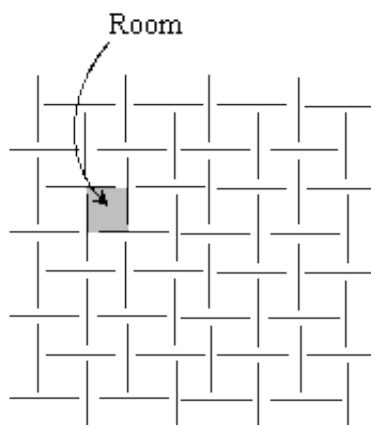


Figure 4.10. Bounded Slicing Grid

In addition to the perturbation defined (swap and rotate) the floor-plan is perturbed through reverse and move operations. These perturbations are added to the interface (Table 4.4).

Table 4.4. Perturbations inspired from TCG

Type of Perturbation	Description
PERTURB_MOVE_UP	Places a module above another module
PERTURB_MOVE_RIGHT	Places a module at right side of another

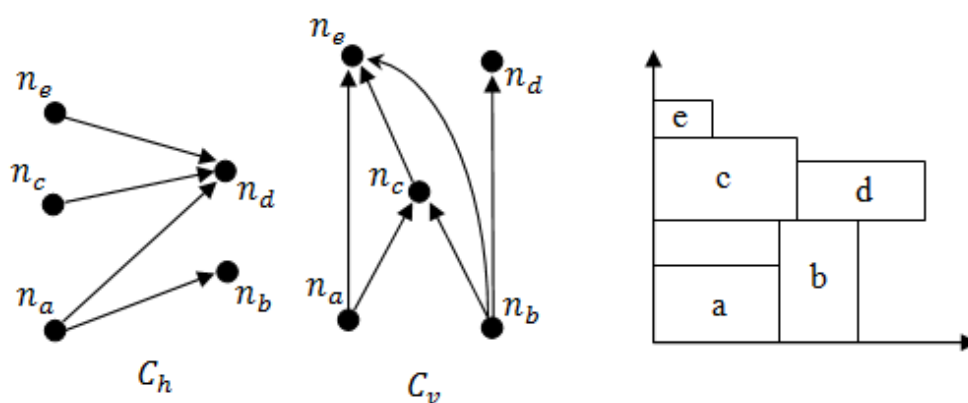


Figure 4.11. Transitive Closure Graphs

4.2.2.6. CBL. Corner block list representation is presented in [30]. The floor-plan structure is based on the mosaic structure. This structure is also defined in the paper.

Corner block is defined and the representation is constructed by deleting the corner blocks. The representation includes a sequence S of block names, a list of orientations L, and a list T of *T-junctions* (intersection of an edge of a module with corner of another module). The (S, L, T) sequence is called a corner block list. The corner block list of the floor-plan in Figure 4.12 is (S, L, T) = (fcegbad, 001100, 001010010). Note that, all CBL representations may not lead to a correct floor-plan, and this limits the representation.

The CBL representation is used to optimize the floor-plan in [30]. Rotation, reflection and perturbations of S, L, T lists are used to construct new solutions. The perturbations in the list are not general and should not be included in the interface.

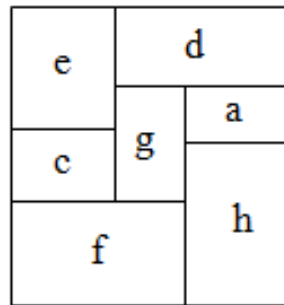


Figure 4.12. Floor-plan is represented as (S, L, T) = (fcegbad, 001100, 001010010)

Table 4.5. Perturbations

Type of Perturbation	Description
PERTURB_REFLECT_HORIZONTAL	Mirrors the modules horizontally
PERTURB_REFLECT_VERTICAL	Mirrors the modules vertically

4.2.3. P-Sequence

P-Sequence (Projection Sequence) is a new representation developed through the study of this thesis. This representation is used to hold the floor-plan of the implemented floor-planner.

Two sequences of module name are used; these sequences are held for the vertical and horizontal projections of the modules. These projections are constructed by projecting the modules onto x and y axis. The axis is divided into segments according to these projections. In Figure 4.13, the segments are regions between the two dashed lines.

The floor-plan may be represented as $(H, V) = (\{A, BC, C, D\}, \{ACD, ABD, B\})$. The groups of the module names (such as ACD) are the modules in the segments on the axis. ACD is a group of module names whose projections are in the first segment on the y axis.

P-Sequence is superior to other representations; the neighborhoods of a module are easily extracted from this representation. Also this representation does not require a compact layout; a layout with spaces is easily handled through this representation.

4.2.3.1. Compaction of P-Sequence. Most of the time the following segments include similar modules and it is not required to hold their names many times. Through the following this representation is more compacted.

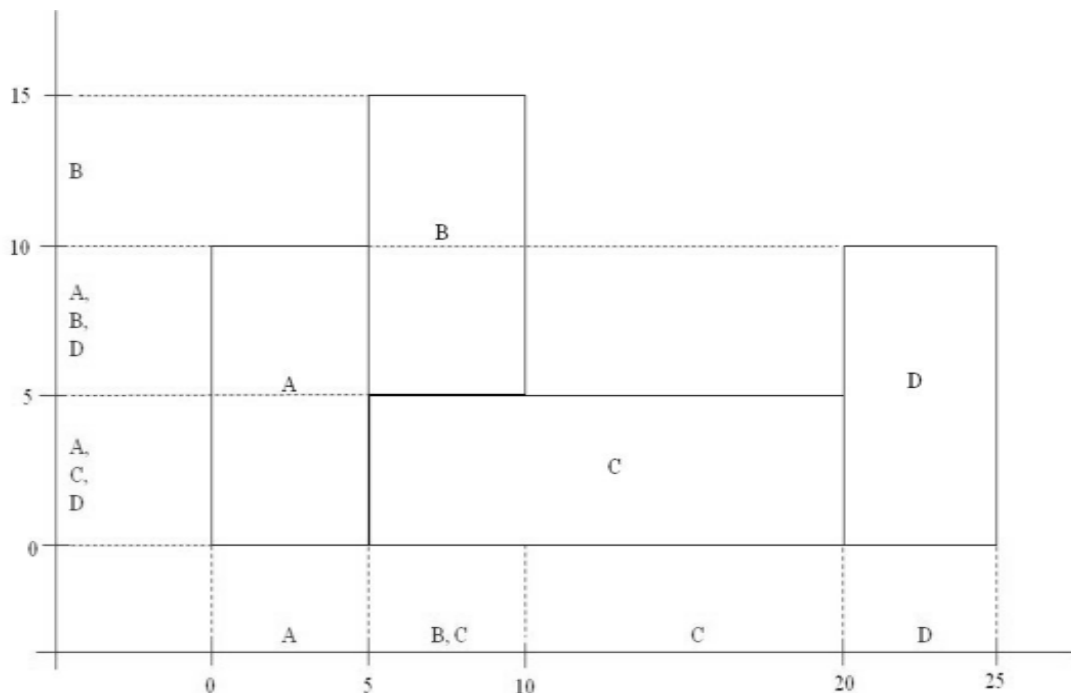


Figure 4.13. Modules are projected onto x and y axis

Process of compaction:

1. If the name of the module is recorded in segment i and is not present in segment $i + 1$ add the name to $i + 1$ th group.
2. If the name of the module is not present in segment i and is recorded in segment $i + 1$ add the name to $i + 1$ th group.
3. If the name of the module is recorded in segment i and it is also recorded in segment $i + 1$ do not change $i + 1$ th group.
4. If the name of the module is not present in segment i and it is also not present in segment $i + 1$ do not change $i + 1$ th group.

For instance, the representation for Figure 4.13 is found as $(H, V) = (\{A, BC, C, D\}, \{ACD, ABD, B\})$. After the compaction the representation is $(H, V) = (\{A, ABC, B, CD\}, \{ACD, CB, AD\})$.

For the floor-plan in Figure 4.14, normal representation is $(H, V) = (\{baif, aif, ef, gef, gf, ghf\}, \{bg, bh, ah, ih, ieh, eh, f\})$ and the compacted representation is $(H, V) = (\{baif, b, aie, g, e, h\}, \{bg, hg, ba, ai, e, i, ehf\})$. As the number of module increases the compact representation becomes much shorter than the normal representation.

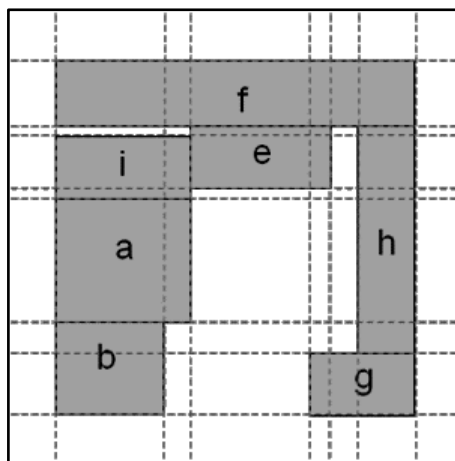


Figure 4.14. Floor-plan is represented as $(H, V) = (\{baif, b, aie, g, e, h\}, \{bg, hg, ba, ai, e, i, ehf\})$

4.2.3.2. Analysis of P-Sequence. The number of segments for the worst case is calculated. For the worst case, projection of a module separates the axis into three parts. An example is given for four modules in Figure 4.15. In general, the axis is separated into $2n - 1$ segments, for n modules and for the worst case.



Figure 4.15. Four modules separate the axis at most to 7 pieces.

Again, for the worst case segment at the center includes n modules and the first and last segments include a single module. Thus the total number pointers required to hold the modules are:

$$N = \frac{n(n-1)}{2} + n + \frac{n(n-1)}{2} = n^2$$

4.3. Layout Description Script (LDS)

This section presents the description of the floor-plan through java script. This description uniquely defines a placement for the chip.

The command in this description language describes an interface for the floor-planner. This description language is directly related to the layout constraints (See Section 4.1). The required commands for the script are stated under the subsections.

4.3.1. Alignment Commands

Alignment commands are used to align to modules. The predefined command handles aligning the top, bottom or middle (vertical) points of two different modules. It is also possible to align the left, right and center (horizontal) points of these modules.

4.3.2. Abutment Commands

The abutment commands are used to bond two different modules. It is mostly used to connect wires to modules. Due to electrical rules in the design all wires are required to touch the ports.

Table 4.6. Commands for alignment

Definition	Description
<code>int alignLeft(int reference, int e);</code>	Aligns left boundaries of modules
<code>int alignRight(int reference, int e);</code>	Aligns right boundaries of modules
<code>int alignCenter(int reference, int e);</code>	Aligns center points (horizontal) of modules
<code>int alignTop(int reference, int e);</code>	Aligns top boundaries of modules
<code>int alignBottom(int reference, int e);</code>	Aligns bottom boundaries of modules
<code>int alignMiddle(int reference, int e);</code>	Aligns center points (vertical) of modules

Table 4.7. Command for abutment

Definition	Description
<code>int abutRight(int reference, int e);</code>	Aligns right boundary of module reference module with the left boundary of input module
<code>int abutLeft(int reference, int e);</code>	Aligns left boundary of module reference module with the right boundary of input module
<code>int abutAbove(int reference, int e);</code>	Aligns top boundary of module reference module with the bottom boundary of input module
<code>int abutBelow(int reference, int e);</code>	Aligns bottom boundary of module reference module with the top boundary of input module

4.3.3. Insertion Commands

Previously defined commands are used to add constraints into the layout and they are converted to constraints. However, the following commands are not constraints and they are used to define the order of the modules.

4.3.4. Command for General Constraints

This command adds general constraints. Previously defined alignment and abutment commands are special cases for this command.

Table 4.8. Module insertion

Definition	Description
<code>int insertAt(int e, double x, double y);</code>	Adds a new module at the x and y coordinates
<code>int insertRight(int reference, int e);</code>	Adds a new module to the left of the reference module
<code>int insertLeft(int reference, int e);</code>	Adds a new module to the right of the reference module
<code>int insertAbove(int reference, int e);</code>	Adds a new module above the reference module
<code>int insertBelow(int reference, int e);</code>	Adds a new module below the reference module

4.3.5. Update Commands

These commands are used to update the layout database or update the floor-plan. An update to the database will load the coordinates of the modules to the database from the floor-planner.

Table 4.9. Adding general constraints

Definition	Description
<code>void setLocation(int e, int reference, double xMin, double xMax, double yMin, double yMax);</code>	Places two modules in such a way that the difference between their x coordinates is in the range $[x_{min}, x_{max}]$. Similarly the difference between their y coordinates is in the range $[y_{min}, y_{max}]$.

Table 4.10. Command for updating

Definition	Description
<code>void updatePlan(DataBase dB, int e);</code>	Reloads the properties of a module into the floor-planner
<code>void updateBase(DataBase dB);</code>	Updates the database according to the coordinates in the floor-planner

4.3.6. Basic Commands

Basic commands are used to delete/add modules, remove constraints, and undo the last modification.

Table 4.11. Basic commands

Definition	Description
<code>void iterate();</code>	Forces the floor-planner to find a valid solution
<code>int redo(int n);</code>	Redo
<code>int undo(int n);</code>	Undo
<code>void removeConstraint(int e);</code>	Removes the input constraint
<code>void delete(int e);</code>	Removes a module from the floor-planner

4.4. Constraints and Inequalities

In section 4.3, the commands for the layout description script are stated. Most of these commands add constraints to the floor-planner. In Figure 4.16, a simple floor-plan is shown. The arrows indicate the input constraints. In this example, left edge of module A is abutted with the left boundary of the floor-plan and the right edge of module D is abutted with the right boundary of the floor-plan. Also, there are constraints due to the static modules, the width of A is constant and this is another constraint. Moreover, module D cannot come closer to module B, because it cannot pass over module C. A classification for the constraints is stated in the next subsection.

4.4.1. Classification of Constraints

In general, constraints in a floor-plan may be grouped into two: soft and hard constraints.

4.4.1.1. Soft Constraints. Soft constraints are not static, and they can disappear. For instance, the module B in Figure 4.16 cannot touch the left boundary of the floor-plan, due to module A; however if module B moves up it can touch the left boundary. Thus, the constraint that B should be after A is a soft constraint and it can disappear.

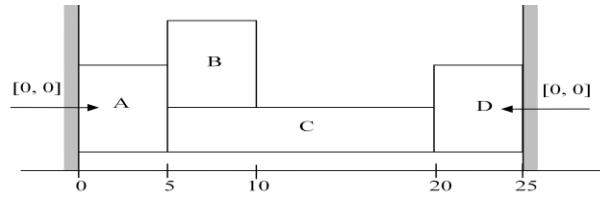


Figure 4.16. Constraints in the floor-planer

4.4.1.2. Hard Constraints. Hard constraints are static and they do not disappear unless the user removes it. In Figure 4.16, the width of module C is fixed to 15 and it will not change.

4.4.2. Constraints and Inequalities

The boundaries of the floor-plan are denoted by B and the subscript indicates which boundary it is. B_l represents the left boundary, B_r represents the right boundary and similarly B_b and B_t represents the bottom and the top boundaries, respectively.

The positions of the edges of the modules are represented like l_A , where the l represents the left edge and A represents the module name. The letter r represents the right edge, and similarly the b and t represents bottom and top edges, respectively.

The following equation is obtained through an observation of Figure 4.16 and states that the position of left boundary (B_l) equals to the position of the left edge of module A (l_A).

$$B_l \leq l_A \leq B_l \leftrightarrow l_A = B_l$$

Similarly the following equations are obtained from the floor-plan in Figure 4.16.

$$\begin{array}{lll}
 l_A = B_l & r_D = B_r & B_l \leq B_r \\
 B_l \leq l_A & r_A \leq l_B & B_l \leq l_B \\
 r_A \leq l_C & r_C \leq l_D & r_B \leq l_D \\
 r_D \leq B_r & r_B \leq B_r & r_A = l_A + 5 \\
 r_B = l_B + 5 & r_C = l_C + 15 & r_D = l_D + 5
 \end{array}$$

4.4.3. Inequalities and Graphs

It is shown in section 4.4.2 that the floor-plan problem is a set of inequalities. The inequalities have to be solved in order to obtain a valid solution.

In this thesis, a new algorithm is developed to solve a set of inequalities. The solution is found by making use of graph theory.

In section 4.4.2, inequalities for the floor-plan in Figure 4.17.(a) are extracted and in Figure 4.17.(b) these inequalities are converted to a directed graph. The presented graph holds ranges as well as values, and the nodes also hold values. The edges in the graph are allowed to be inverted. Inversion of an edge is done as follows:

- Value of the edge is negated.
- The range is negated. When the range $[\alpha, \beta]$ is inverted, it becomes $[-\beta, -\alpha]$.

4.4.4. Solving the Inequalities

The proposed solver deals with any set of constraints and detects if the solution does not exist. Starting from any node the algorithm runs and the procedure is as follows:

1. Change the directions of the surrounding edges as outgoing.
2. Start visiting the surrounding (directly connected) nodes.
3. Calculate the difference between the value of this node and the connected node.
4. Assign the difference to the edge and if the capacity of the edge is not enough to hold the difference, update the value of the connected node.
5. Add any updated or unvisited node to the queue.
6. If there is not any node to be visited, leave the loop. The node values are the solutions.
7. Use Breadth-first algorithm to visit the nodes.

As an example, the solution of the graph in Figure 4.17 is present in APPENDIX C.

4.4.5. Loop Breaking

It is not always possible to find a solution to a set of inequalities. In this case, the presented algorithm tries to find the solution, which is not present. It is expected that the algorithm locks. However, in that case, the presented algorithm realizes that the inequalities (constraints) conflict.

When a solution does not exist, the algorithm starts to visit the edges periodically. The order of the visited edges is considered as a signal and checked for replications. This signal (list of the visited edges) is processed with the system presented in Figure 4.18.

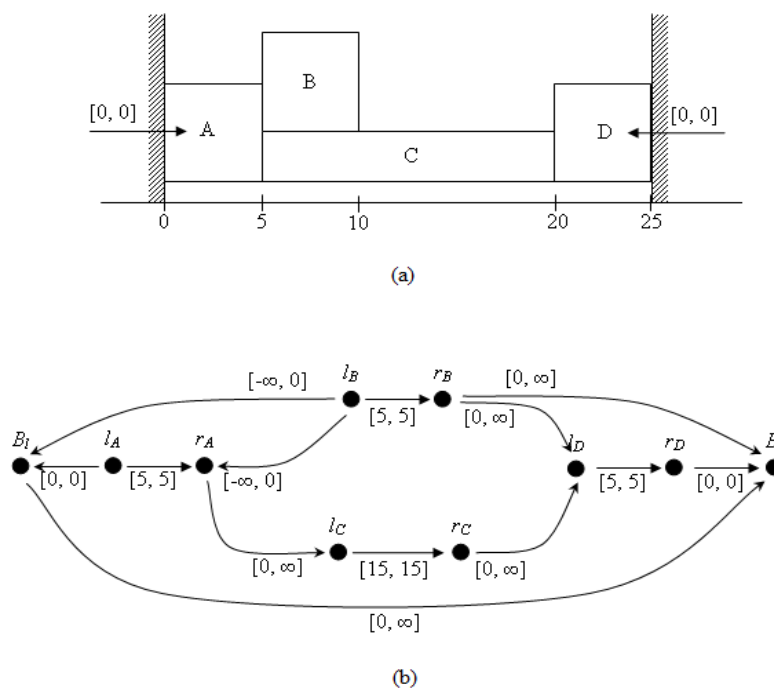


Figure 4.17. (a) The plan and the constraints (b) The representation of the constraints in a graph

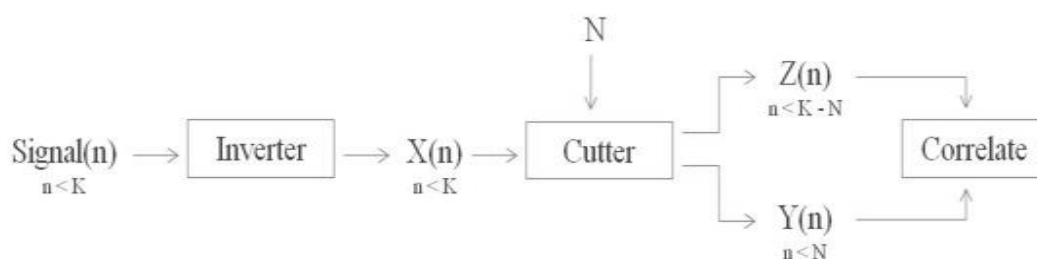


Figure 4.18. Block diagram of the loop breaker

Two sample signals (visited edges) are plotted in Figure 4.19. In this figure, the second signal is obtained from a set of solvable inequalities and the first signal is obtained from a set of unsolvable inequalities. As depicted in the plot, the visited edges in the first signal has some periodic components, these are pointed out in Figure 4.20. Loop breaking property of the solver prevents blocking of the software.

4.5. Floor-planner Implementation

In this thesis, the implemented floor-plan is based on the P-sequence representation in section 4.2.3 and the inequality solver in section 4.4.4. The implementation is compatible with the layout description script (LDS) described in section 4.3.

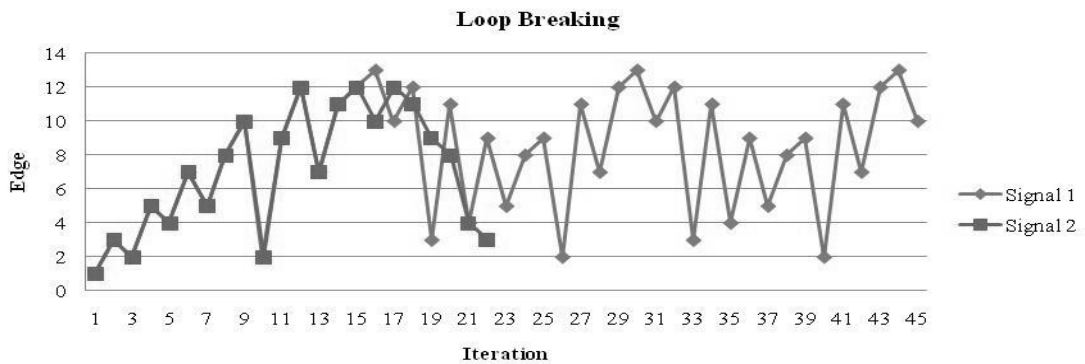


Figure 4.19. Sample signals (visited edges)

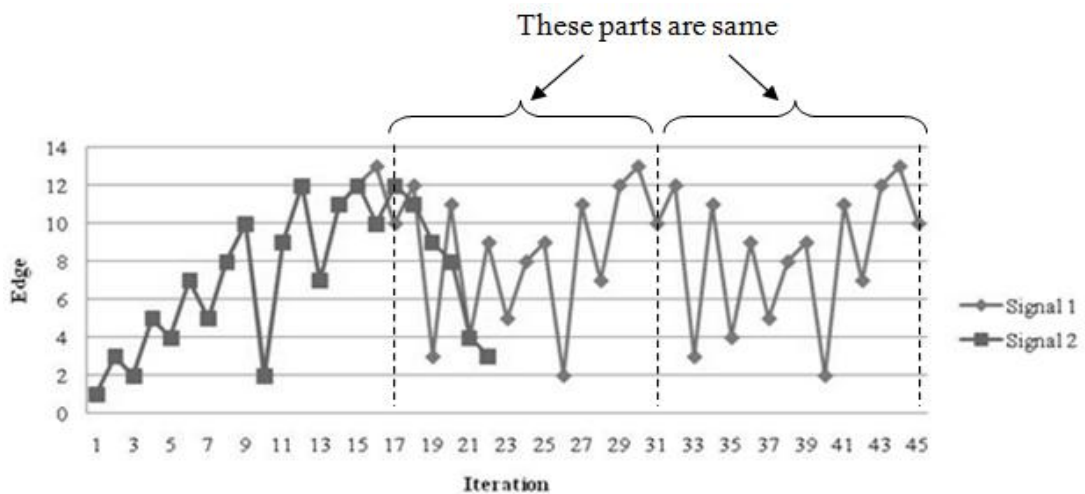


Figure 4.20. Periodicity in the signal

The P-Sequence representation is used to hold the ordering of the modules. This structure is used to extract the soft constraints described in section 4.4.1. The hard constraints are installed through the template (LDS). All these inequalities (constraints) are solved and the database is updated according to the solution.

5. DEVICE GENERATOR AND ROUTER

Device generator tool is one of the basic parts of a layout synthesizer. Similar to the other tools, this tool is connected to the data base and adds new structures to it. It is possible to construct any type of device through the device generators. These device generators are written in java and they depend on the available technology. It is required to update the device generator when the technology file updates. As an example, a CMOS transistor structure will be different that a bipolar one. To construct the bipolar device a new code is required. Thus it will be a good practice to define a interface for the device generator, just to make it general. In section 5.1, a general interface is defined.

5.1. Interface

The definition of the command, needed to construct a new device, is explained in Table 5.1.

Table 5.1. Interface for adding new devices

Definition	Description
<code>int newDevice(int type, double[] parameters, int[] nets);</code>	Adds a new device to the layout database

The *newDevice* command in Table 5.1 takes the device type as an input parameter. For a CMOS device technology, devices types are tabulated in Table 5.3.

Device generator constructs the requested devices, however the ports of the device are not known. Thus, the generator should somehow return the positions of the ports. This is done through the *getPorts* command, defined in Table 5.2.

Table 5.2. Interface to get the ports

Definition	Description
<code>int[] getPorts(int e);</code>	Returns references for the ports.

Table 5.3. Device types for CMOS process

Device	Value for the Type Parameter
NMOS	1
PMOS	2
RESISTOR	3
CAPACITOR	4
INDUCTOR	5

5.1.1. Example

The following code constructs a PMOS device.

```
int t = x.newDevice(PMOS, new double[] {w, l, m}, new int[] {d, g, s, b});
```

Where w is the width of the transistor, l is the length of the transistor; m is the number of folds, d is the reference of the drain node, g is the reference of the gate node, s is the reference of the source node and b is the reference of the bulk node.

5.2. Devices

In the implementation of this thesis, the device generator constructs NMOS and PMOS devices. The details for the constructed devices are explained in the sub sections.

5.2.1. NMOS Transistor

The NMOS device is constructed by calling the *newDevice* command. The width and the length of the transistor are not restricted and they can be adjusted independently. According to the length of transistor, the number of the metal-poly contact is updated.

Also a guard ring is added around the transistor. Figure 5.1 shows a sample NMOS transistor. The width of this transistor is $48\mu\text{m}$, the length of this transistor is $1\mu\text{m}$ and the number of folds is 14.

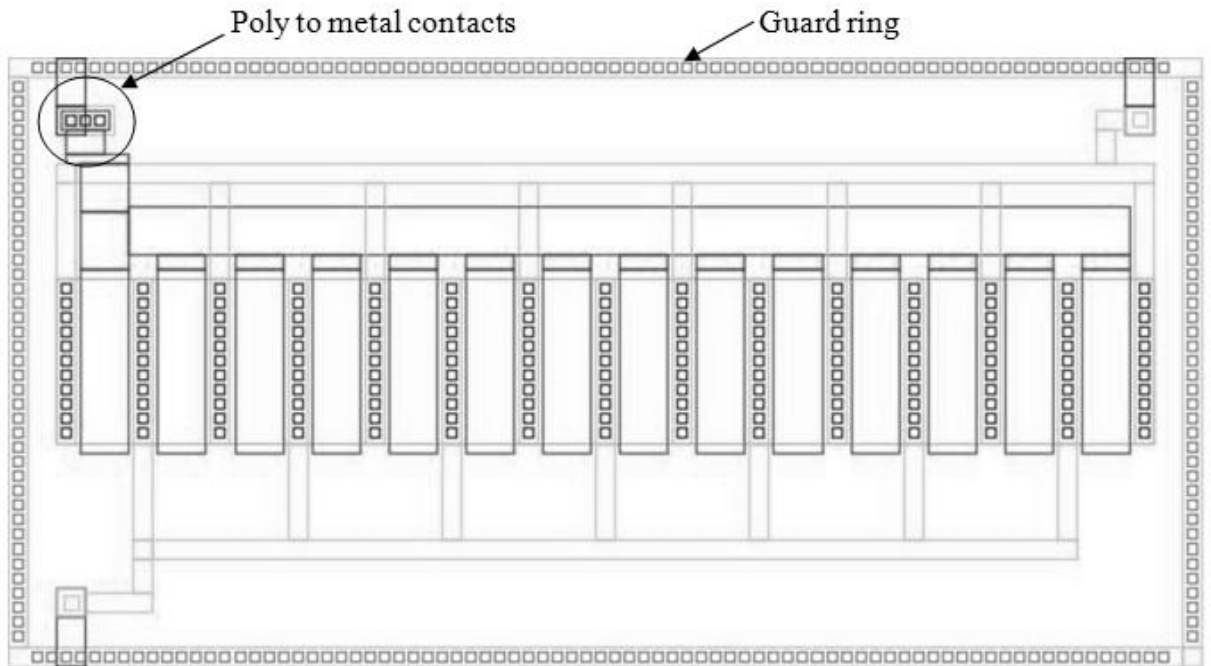


Figure 5.1. NMOS Transistor

5.2.2. PMOS Transistor

A PMOS device is constructed by the device generator and presented in Figure 5.2. The construction is similar to the NMOS transistor; additionally, NWELL layer is added around the transistor.

5.3. Routing

An interface for the routing has not been developed yet. However a template based router is developed. This routing scheme is based on a template that defines the wires as device and these devices are constructed by the device generator.

In Figure 5.3, a sample routing is shown. The wire segments are coded in the template. Due to the fact that the designer must consider the vias, the wires and the port, construction of the template is a time consuming process. Thus it will be a good idea to add an automatic router into the design environment. I have not implemented an automatic router but made a detailed literature review about the available routers. These are going to be detailed in the following subsections.

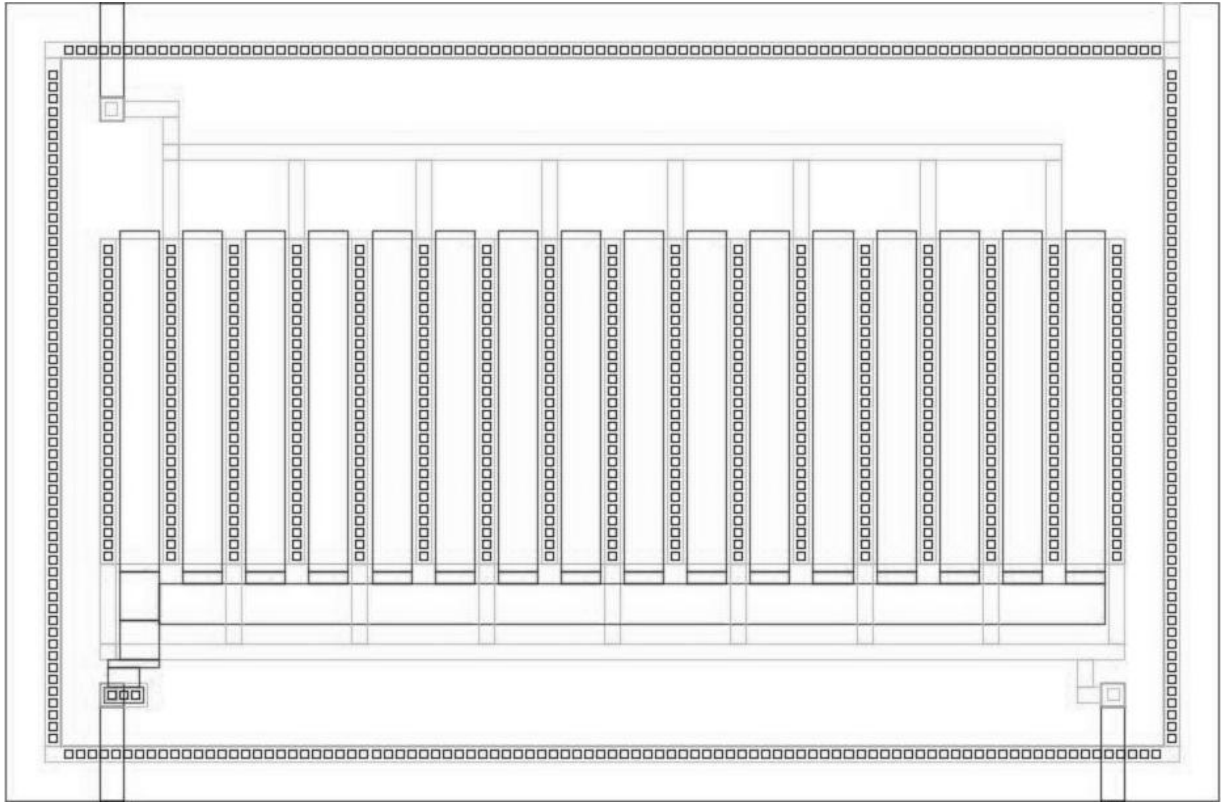


Figure 5.2. PMOS Transistor

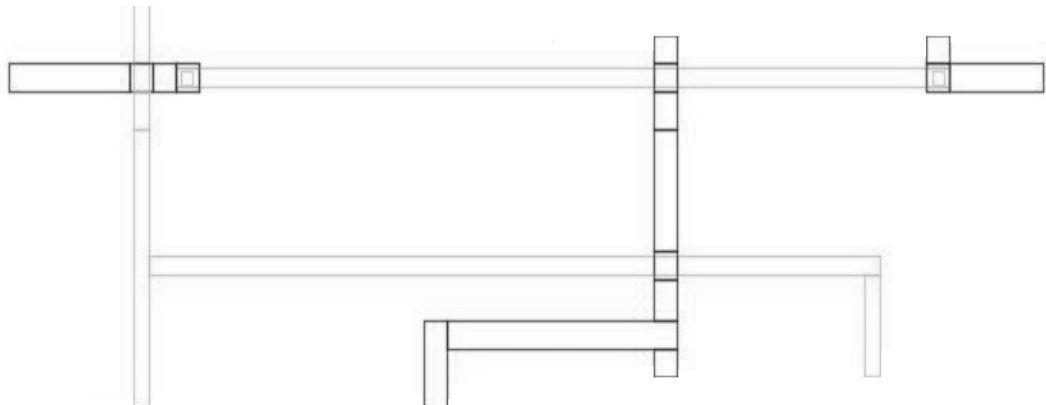


Figure 5.3. Template based routing

5.3.1. Classification of Routers

In general, routing can be classified into two groups. These are area routers and channel routers.

5.3.1.1. Area Routing. Area routing tries to utilize all the spaces in the layout. If multiple routing layers are available, the router should also consider the effects of coupling between the paths and the devices. Moreover the area routing is computationally less efficient than the channel routing.

In area routing, nets are considered individually and a global view of the interconnections is not present. On the other hand, area routing is general enough to be used with any class of circuits and geometric complexity.

A well known area routing algorithm is the maze routing approach. This approach is based on Lee-Moore Algorithm [39]. Also the Prim's and Kruskal's algorithms are used to find out the minimum spanning tree (MST). Another approach based on planning techniques is in [40]. Moreover, the area routing approach is generalized for analog layouts in [9].

As technology advances, the circuit sizes are getting larger. To cope with the increasing complexity, it proposed to use hierarchical approaches to handle the problem. The multilevel framework started to be popular in the literature recently. The approach uses a two stage technique, it is coarsening followed by un-coarsening. In the coarsening stage modules are groups according to pre-defined metric. Then the un-coarsening stage un-groups the grouped modules and refines the solution. A multilayer approach is presented in [41].

The area router is applied to analog layouts [9]. This tool uses a relative grid with and it allows over-device routing, although routing over sensitive modules can be prevented.

5.3.1.2. Channel Routing. The channel router is a special router designed for routing in an area with no inside obstructions and with terminals on two opposite sides. The dogleg router has been one of the most successful implementations of this router.

Channel routing is computationally more efficient than the area routing. In this approach *channels*, between the *rows*, are used for routing. Left-edge [42] and “Greedy” algorithms [43] are well known approaches.

The channel routing is not limited to digital circuits, the tool ALG [44] is an analog layout generator for CMOS circuits. This tool also uses a channel based routing algorithm. Moreover, channel based routing is applied to channel-less circuits, the approach in [45] aligns the terminals of the modules in each row and constructs channel like routing areas.

Routing between the rows is done through switchboxes. Thus, a switchbox router is also required and the switch-box problem is an NP hard problem. Some papers adapted a set of heuristic rules which reduces the possibility of search failure to avoid blocking.

To sum up, the channel routing approach is successfully applied to digital design automation and it may advance the analog layout automation.

5.3.2. Required Capabilities

Throughout the literature review, it is concluded that the following capabilities are required in a router.

1. Nets should be ordered according to their priority.
2. The layer and the widths of the wires should not be static.
3. The router should record the manipulation and when desired it should be able undo the modifications.
4. All the nets should be processed simultaneously. (It is not desired to have a net by net router.)
5. Physical constraints should be added between wires, such as minimum separation.
6. The router should efficiently allocate the routing space.
7. It should estimate the routing parasitic, such as coupling capacitance and wire resistance.
8. Detailed routing may be done through the device generator, thus the main functionality of the router should be to manage the routing.

9. Symmetric routing and bus routing should be supported.
10. Router should be able to add shielding around the wires, for preventing cross talk.
11. Manual and automated routing should be supported simultaneously. (Manually routed parts may be locked.)
12. Power nets should be defined separately.
13. Reporting of the missing connections should be present.
14. The layer transitions should be considered in the optimization loop. (Too many vias may influence the performance.)
15. Different paths of a net can have different current densities and widths. Net splitting is needed.

5.3.3. Routing Styles

Routing of an analog circuit requires different routing layer, wire widths. These are called styles for routing and these styles are added into the technology files.

A style contains the available routing layer, such as metal1 and metal2. Also a style contains information about the shielding around these layers. Moreover, in a style the spacing between wires, the widths of the wires are stored.

6. IMPORTER AND EXPORTER

The importer and the exporter tools are used to exchange information between different programs. In the layout generator; the importer tool is used to import devices, nets and device parameters from a spice file. On the other hand; the exporter tool is used to output the layout. The output format of the implemented tool is GDSII. This file format is commonly accepted.

In Figure 6.1, a design loop is defined and in Table 6.1, some formats are given. These formats are used to exchange information between different blocks in Figure 6.1.

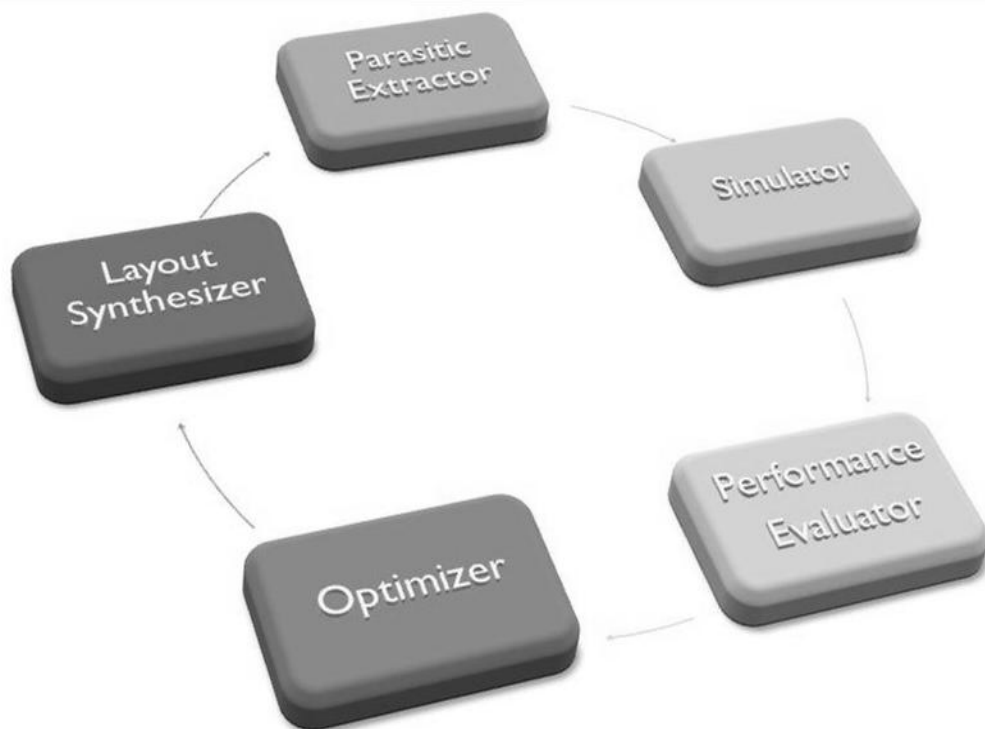


Figure 6.1. Design loop

6.1. Importer

This tool imports netlist and devices from the input spice file. An interface is define for the importer and tabulated in Table 6.2.

Table 6.1. Formats for exchange

Module	Input	Format	Output	Format
Parasitic Extractor	Layout Database	GDSII	Netlist	SPICE
Simulator	Netlist	SPICE	Output	ASCII
Performance Evaluator	Output of Simulation	ASCII	Output	ASCII
Optimizer	Objectives and Constraints	ASCII	Netlist	SPICE
	Netlist	SPICE		
	Output of Evaluator	ASCII		
Layout Synthesizer	Netlist	SPICE	Layout Database	GDSII
	Constraints	ASCII		

Table 6.2. Interface for the importer

Definition	Description
<code>void load(String fileName);</code>	Loads the netlist and components
<code>double[] getParameters(int e);</code>	Returns the parameter of a device
<code>int[] getNets(int e);</code>	Returns the nets of a device
<code>int[] getComponents();</code>	Returns the components from the spice file

6.2. Exporter

This tool is used to export the layout database. The output format of the layout generator is chosen to be GDS2. However, it is not restricted to be GDS and any exporter (i.e. OASIS [46]) implementing the interface in Table 6.3 can be used.

Table 6.3. Interface for the exporter

Definition	Description
<code>void connect(DataBase dB);</code>	Connects to a database
<code>void convert(String fileName);</code>	Exports the database into a file

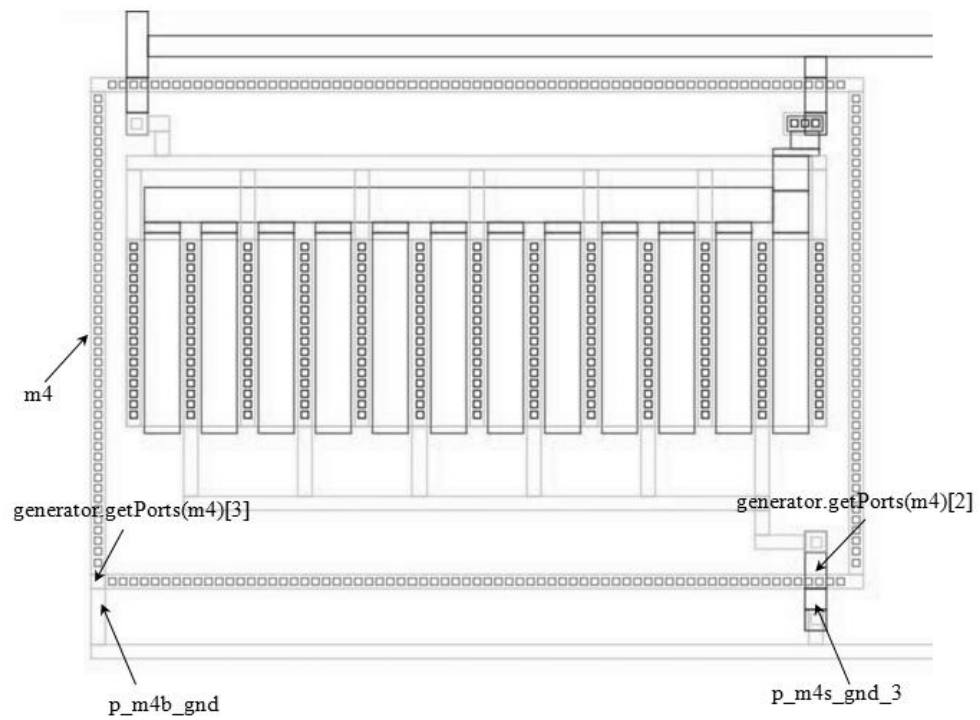


Figure 7.3. Template based construction

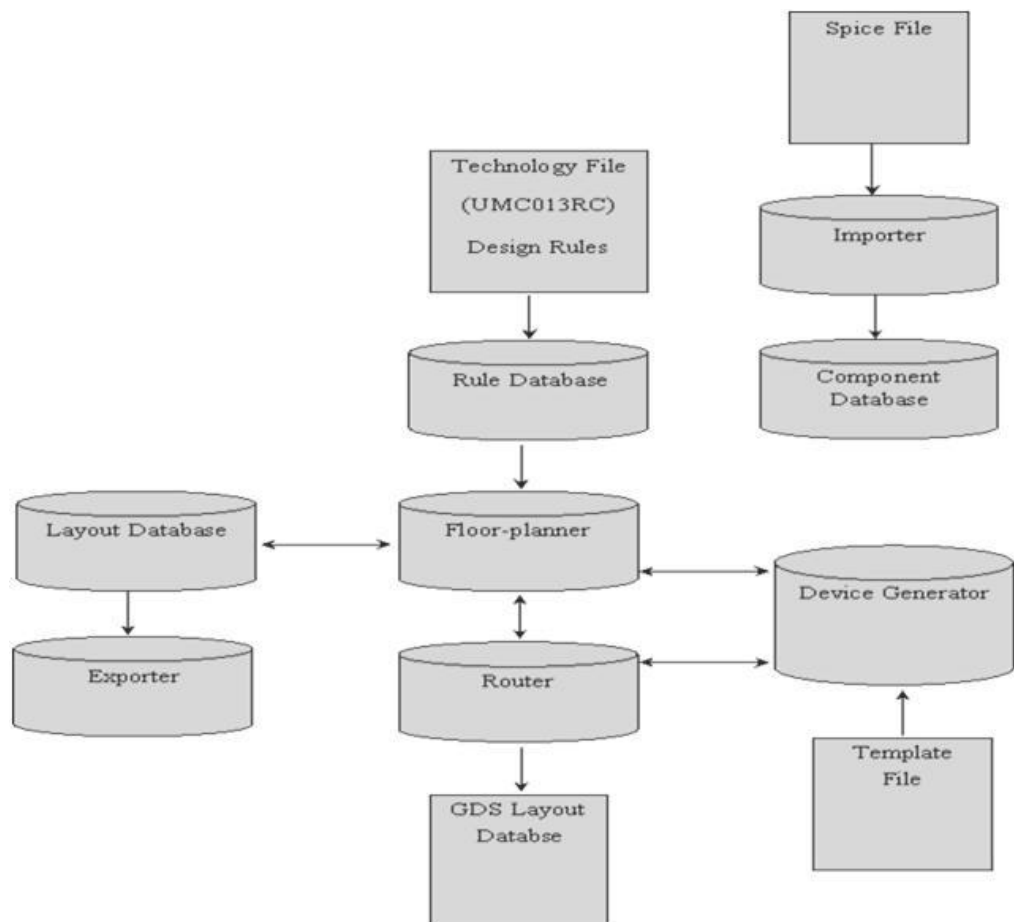


Figure 7.4. Block diagram of the layout generator

The layout generator is fed with the spice code in Figure 7.5 and the layout in Figure 7.7 is synthesized.

```
M1 2 7 3 1 pmos w=132e-6 l=1e-6 m=16
M2 2 8 4 1 pmos w=128e-6 l=0.8e-6 m=16
M3 4 3 6 6 nmos w=48e-6 l=1e-6 m=14
M4 3 3 6 6 nmos w=64e-6 l=1e-6 m=12
M7 1 9 2 1 pmos w=120e-6 l=1.2e-6 m=16
```

Figure 7.5. SPICE code

The layout generator is also fed with the spice code in Figure 7.6. In Figure 7.8, the synthesized layout is shown.

```
M1 2 7 3 1 pmos w=8e-6 l=0.2e-6 m=8
M2 2 8 4 1 pmos w=8e-6 l=0.2e-6 m=8
M3 4 3 6 6 nmos w=8e-6 l=0.2e-6 m=8
M4 3 3 6 6 nmos w=8e-6 l=0.2e-6 m=8
M7 1 9 2 1 pmos w=8e-6 l=0.2e-6 m=8
```

Figure 7.6. SPICE code

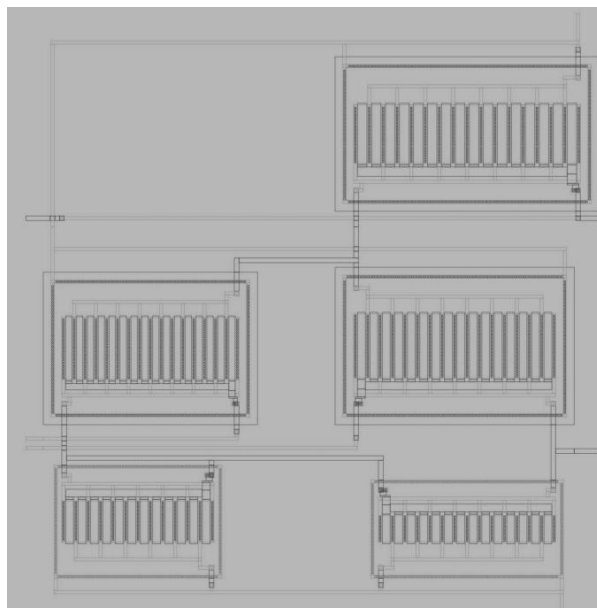


Figure 7.7. Constructed layout (spice code in Figure 7.5)

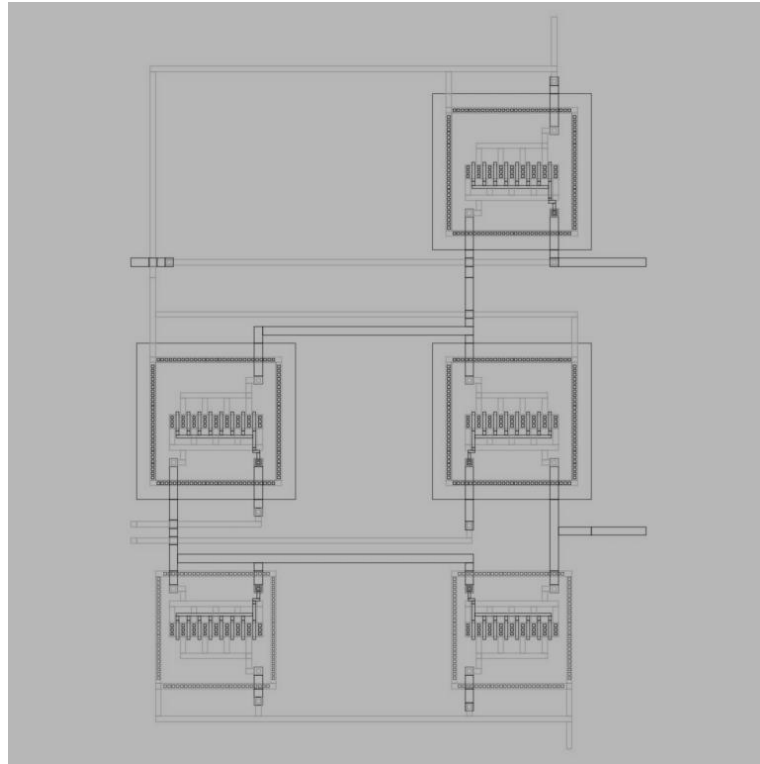


Figure 7.8. Constructed layout (spice code in Figure 7.6)

8. CONCLUSION AND FUTURE WORK

8.1. Conclusion

In this work, a template-based analog layout automation tool is developed. The templates are coded in java language and the functions, used in these templates, are pre-defined through the interfaces in sections 3, 4 and 5. This layout tool consists of six parts. These are: importer, layout database, device generator, floor-planner, router and exporter.

Through the importer tool, the circuit components and their dimensions are read. The implemented importer reads SPICE files. The components read from the input SPICE file are recorded into component database.

The layout database holds the drawings of the components. The implemented database considers the components as geometric shapes. Through this database; the components are easily rotate, flipped and moved. The database also supports hierarchy. Compared to conventional layout databases, this implementation runs faster.

Drawings of the components are synthesized by the device generator. The device generator reads design rules from a technology file and synthesizes NMOS and PMOS transistor. The implemented device generator is not restricted to synthesize transistors. However, only these devices are defined and new devices are required to be defined.

Floor-planner plans the placement of the layout components. The plans are read from template files. The template files include absolute and relative constraints. These constraints are mentioned in section 4.1. The constraints, read from the template files, are combined with physical constraints and they are solved by an equation solver. Algorithm of this solver is developed through the study of this thesis.

Routing between components is done by a template-based router. The implemented router does not decide on the routing; however reads it from a template file.

Finally, the exporter outputs the layout database into a GDSII file. This output format is a common layout exchange format.

Being a template-based tool, this work can synthesize layouts in a few seconds. Section 7 includes some of the resulting layouts. Due to the fact that template-based layout synthesizers run much faster than the optimization based synthesizers, they can be used in a parasitic aware design loops. Such a loop is defined in Figure 6.1. This design loop is going to be applied and this work will perform the layout synthesis.

8.2. Future Work

Given the template files, this work is capable of synthesizing any kind of circuit. However, construction of these templates takes considerable amount of time. These templates consist of tens of components and most of these components are wires and vias. Thus, a simpler routing scheme is needed.

A router is required to be implemented that will automatically add the wires and vias into the template file. Thus, the user will only add the modules; transistors, capacitors and etc... The required functionalities for a router are described in section 5.3.2.

Due to technology migration, template based placement may lead unoccupied die area. Thus, a template generator is required. This generator should synthesize a template for a given netlist.

APPENDIX A: COMPUTATIONAL COMPLEXITY

A main concern about an algorithm is how long it takes to end. Also the algorithm will utilize some of the memory resources of the machine on which it is running, another concern is about how much of the memory will it use. The *computational complexity* refers to the formulation of the time and the memory required by an algorithm in terms of the algorithm's input size (n). The time related computational complexity is referred to as *time complexity* where the memory related complexity is referred to as *space complexity*.

The time complexity is a measure for the time required to complete an algorithm.

Generally it refers to the upper bound for the time required by an algorithm. The upper bound for the time is referred to as *order* and indicated by (O) capital-O. Note also that, the formulation does not involve constants and is simplified by dropping non-dominant terms. For instance if the time required for some algorithm is formulated as:

$$f(n) = O(g(n)) = O(3n^3 + 3n^2 \log_2(n) + 5n)$$

It is said that the algorithm's time complexity ($f(n)$) is of order at most $g(n)$ function. This complexity can be simplified, by removing the constants and dropping non-dominant terms, as:

$$f(n) = O(g(n)) = O(n^3)$$

Other than the upper bound complexity, there exist other types of complexities such as the *average time complexity*. The average time complexity is a better figure of merit for the time complexity, however it is hard to formulate. Calculations require the *joint probability density functions* or *joint probability mass functions* for the inputs, which is hard to obtain. Due to this fact, the time complexity generally refers to the upper bound for the time complexity.

The space complexity is a measure for the amount of memory required to run an algorithm. Generally the space complexity is given less attention than the time complexity and is not mentioned. Note also that, an algorithm will not run at all if the memory requirements of the algorithm exceed the available resources of a machine.

A.1. Classification of Algorithms in terms of Order

Order is defined as the upper bound for the time required by an algorithm as detailed in the Computational Complexity part.

A.1.1. Exponential Order

As the name implies, these algorithms have an exponential order. The time required for the algorithm grows exponentially with the input size. In integrated design automation these algorithms are generally used to find exact solutions to optimization problems. One should be careful about the size of the input, if the input size exceeds some threshold, the algorithm will not finalize within a tolerable time interval.

A.1.2. Polynomial Order

These algorithms have polynomial order and they should be preferred over exponential algorithms. The algorithms with order in between to polynomial orders are also included in this class. For instance an algorithm with an order of $O(n^2 \log(n))$ is in this class.

In general one should prefer an algorithm over the other if the order is less. An algorithm with a linear order should be preferred over an algorithm with quadratic order. Also if available, algorithms with sub-linear order should be preferred over the others. For instance an algorithm of $O(\log(n))$ requires much less time than an algorithm of $O(n)$ as the input size n increases.

APPENDIX B: ALGORITHMIC GRAPHS

A graph can be defined as the description of the connections between the elements of a set. In general graphs ease the mathematical formulation and solution of a problem. Due to this fact, they are also in field of integrated design automation, and they are extensively used.

Algorithmic graph theory aims to design algorithms, running on graphs. This theory does not focus on the mathematical properties of the graphs; however it focuses on the application of these properties. A brief introduction to the graph theory, necessary to design algorithms, is stated in this part.

B.1. Terminology

A graph $G(V, E)$ is defined with two sets; the *vertex* set (V) and the *edge* set (E). An edge is defined to be the relation between two vertexes. If edge e is defined between vertexes u and v , the edge is formulated as $e = (u, v)$. See Figure B.1, for a graph $G(V, E)$ with $V = \{v_1, v_2, v_3\}$ and $E = \{e_1, e_2\}$, where edges e_1 and e_2 are defined as $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_3)$, respectively. Figure B.1

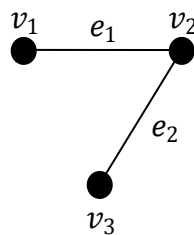


Figure B.1. A sample graph

An edge starting and ending at the same vertex is called a *self-loop*. If two edges $e_1 = (u, v)$, $e_2 = (u, v)$ start and end with the same vertexes, these edges are called *parallel edges*. A graph without self-loops or parallel edges is called a *simple-graph*. A graph with parallel edges but without self-loops is called a *multi-graph*.

A *path* is defined as an alternating sequence of vertexes and edges. The first and the last element in a path should be a vertex. Every edge in the sequence is surrounded by two

vertexes, and every edge should be defined by the surrounding two vertexes. The *length* of a path is defined as the number of edges in the path. If a path includes a vertex only once, this path is called a *simple path*. For instance in Figure B.1 the sequence v_1, e_1, v_2, e_2, v_3 is a path. Note also that, the vertexes u and v are called *connected*, if there exists a path between them.

A directed graph is sub-class of a graph, in which the edge $e = (u, v)$ is only defined from u to v . More formally the edge $e_1 = (u, v)$ is not the same with the edge $e_2 = (v, u)$. In Figure B.2, a directed graph is plotted. A path in a directed graph is defined as *directed path*, if all of the edges are directed in the same way as the path.

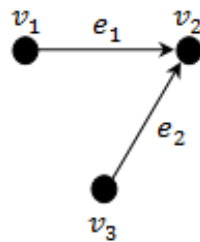


Figure B.2. A directed graph

The algorithmic graph theory focuses on developing algorithms on graphs. Thus if required one can extend the information, the graph holds. For instance weights to the edges or to the vertexes can be added. Assigning weights, the *weighted path lengths* are defined as the sum of the edge weights through a path.

B.2. Graph Algorithms

In this part, some of the common graph algorithms, utilized in the thesis are summarized.

B.2.1. Depth-First Search and Breadth-First Search

It is commonly required to visit the vertexes in a graph. One way to visit the vertexes is to use the depth first search (DFS). This algorithm recursively visits the vertexes. After a vertex is visited, the vertex connected to this vertex is visited.

Breadth-First Search (BFS) is another algorithm used to visit the vertices. The basic idea of BFS is to explore all vertices adjacent to a vertex before exploring any other vertex.

B.2.2. Longest-Path and Shortest-Path Algorithms

The longest path algorithm is used to find the longest path starting from vertex u , and ending at vertex v . As well as the longest path algorithm, there exist shortest path algorithms. Dijkstra's algorithm is one of the most popular shortest path algorithms.

APPENDIX C

In this section the floor-plan in Figure 4.17 is going to be solved using the algorithm in section 4.4.4. The legend for the Figure C.2 to Figure C.20 is presented in Figure C.1.

- Marked and previously not visited
- ⊗ Marked and previously visited
- Not marked and previously not visited
- ⊖ Not marked and previously visited
- ⊙ Selected

Figure C.1. Legend

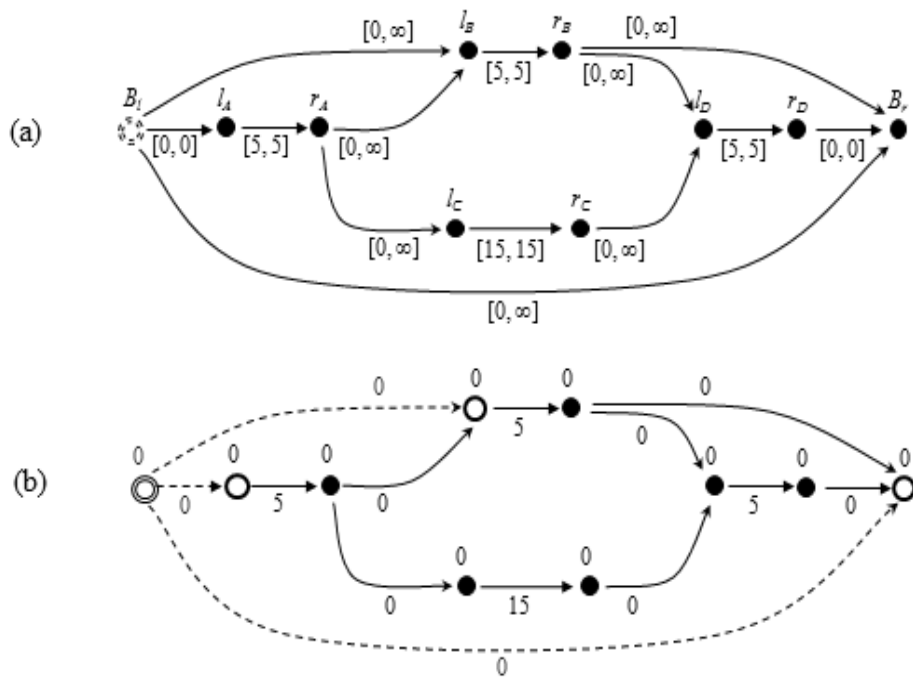


Figure C.2. Solution of the example: Step 1

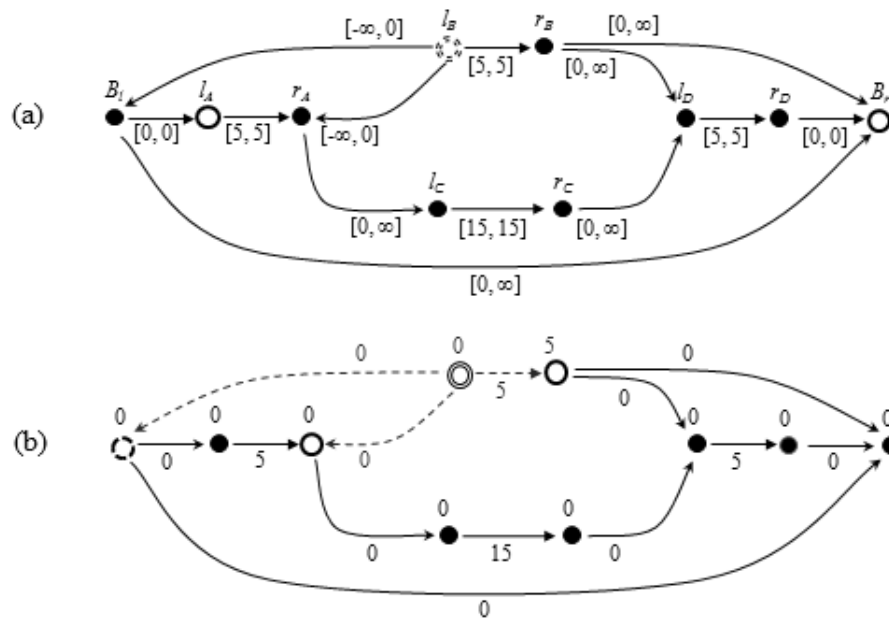


Figure C.3. Solution of the example: Step 2

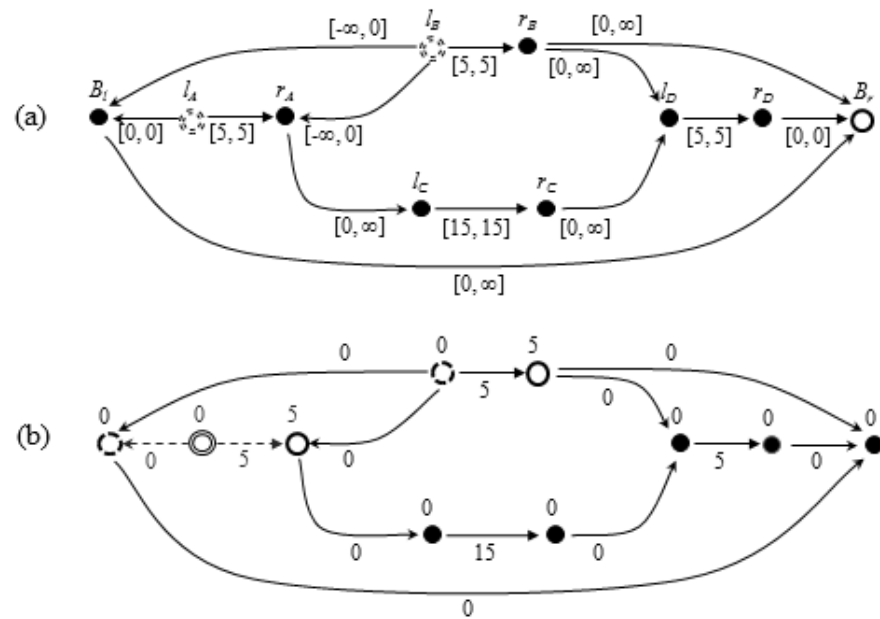


Figure C.4. Solution of the example: Step 3

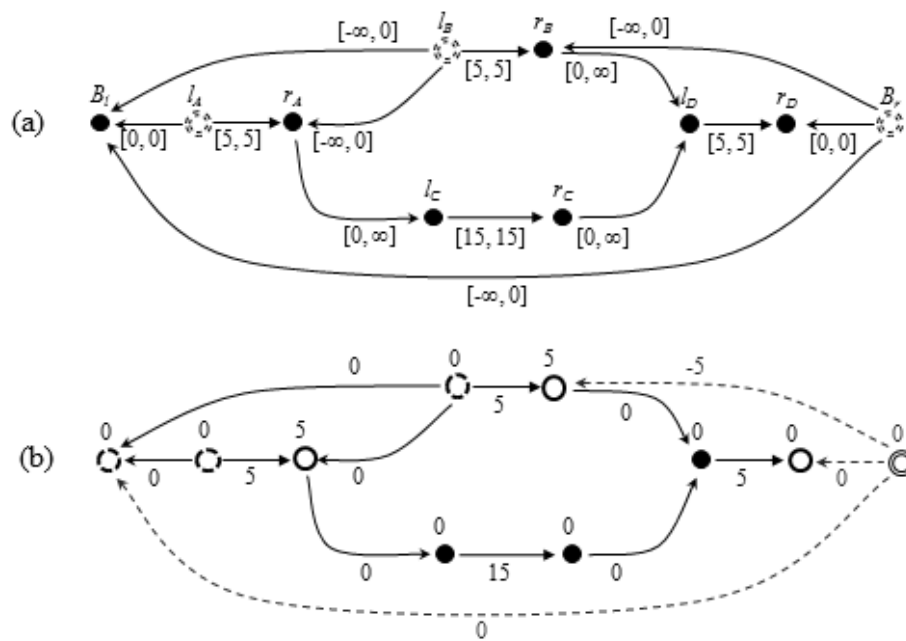


Figure C.5. Solution of the example: Step 4

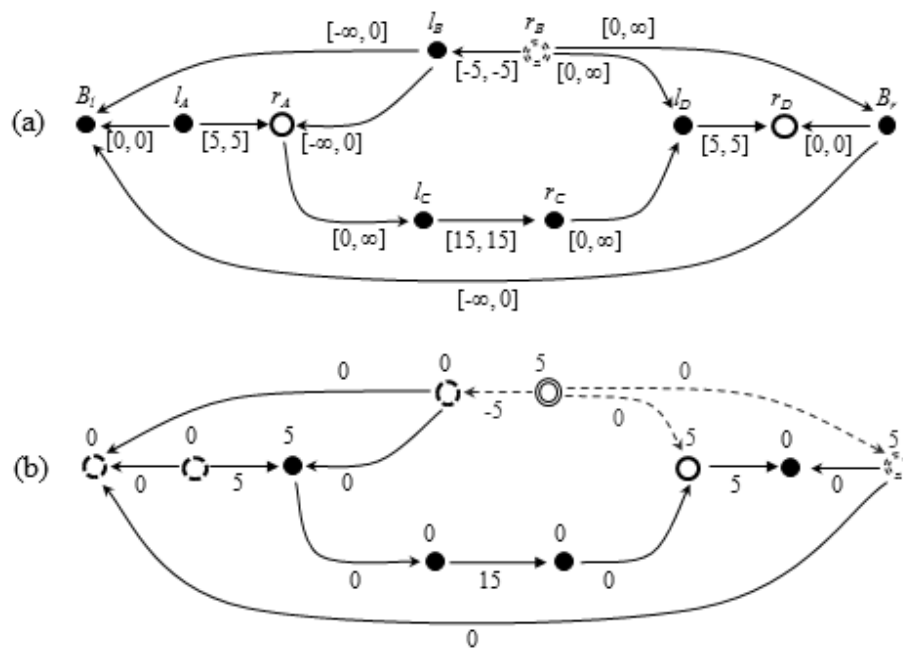


Figure C.6. Solution of the example: Step 5

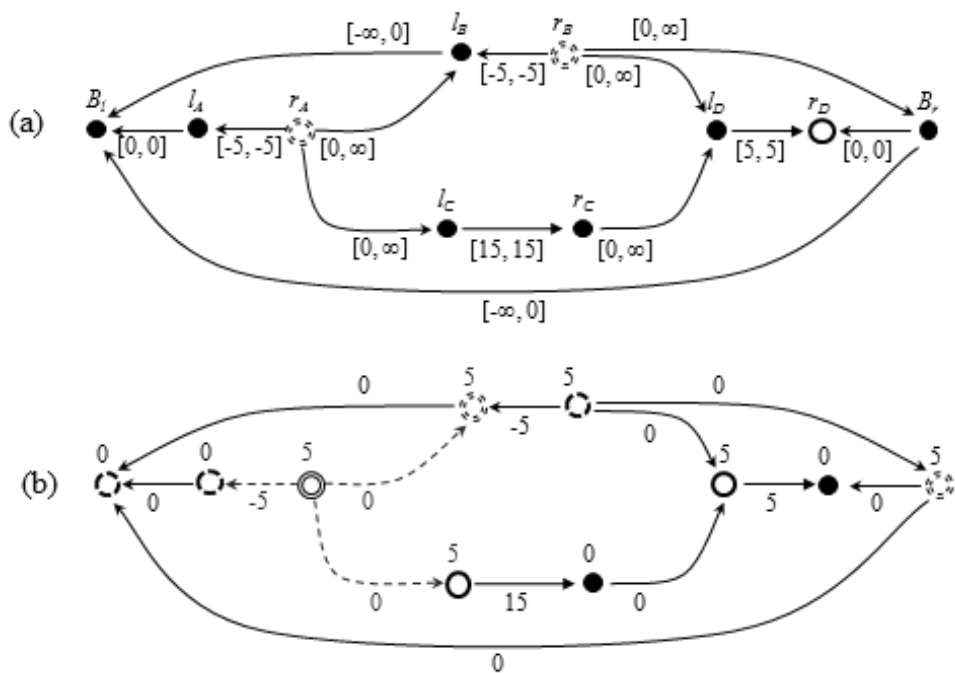


Figure C.7. Solution of the example: Step 6

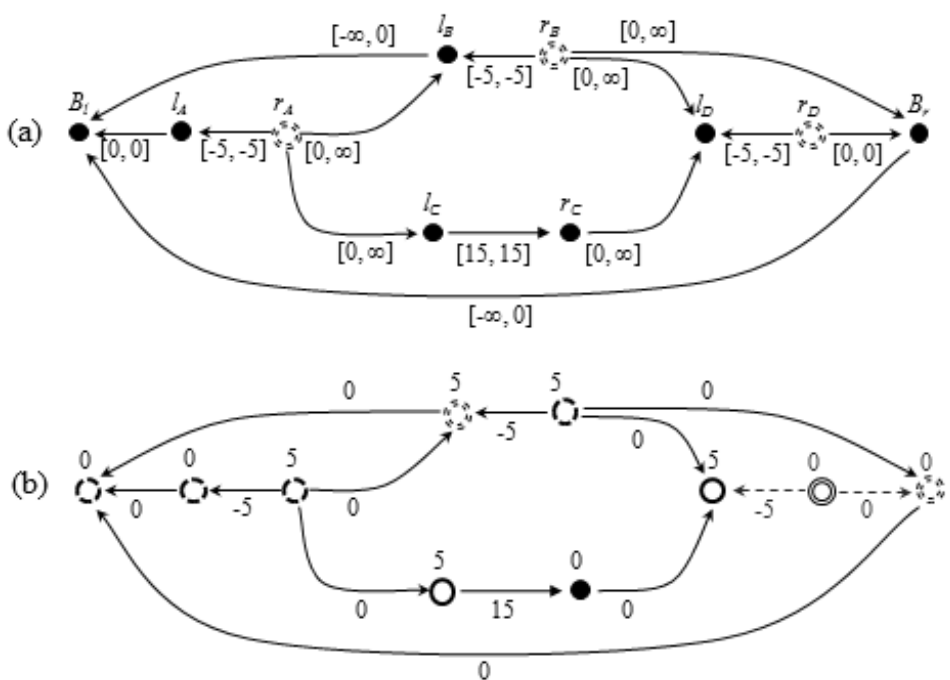


Figure C.8. Solution of the example: Step 7

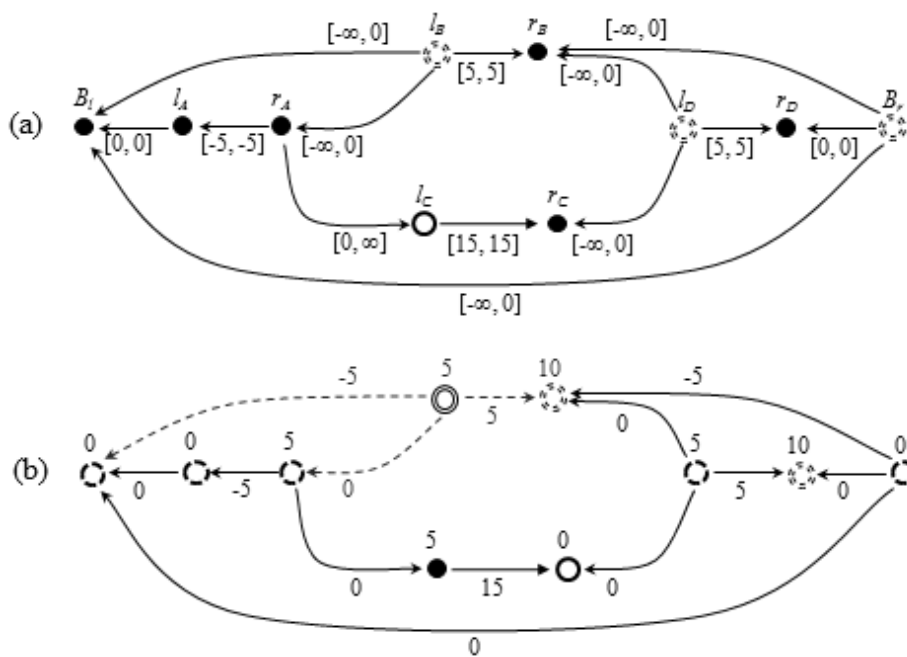


Figure C.11. Solution of the example: Step 10

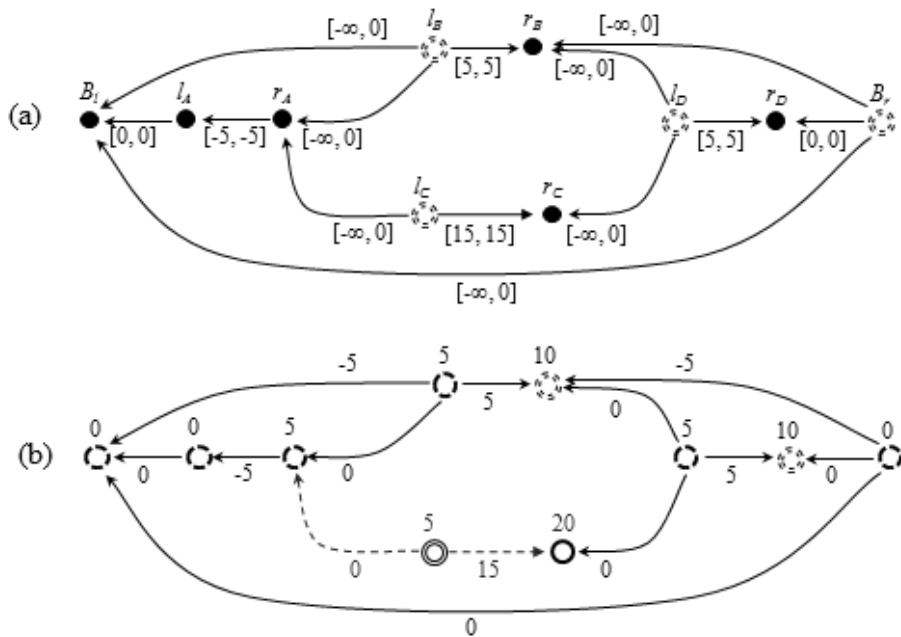


Figure C.12. Solution of the example: Step 11

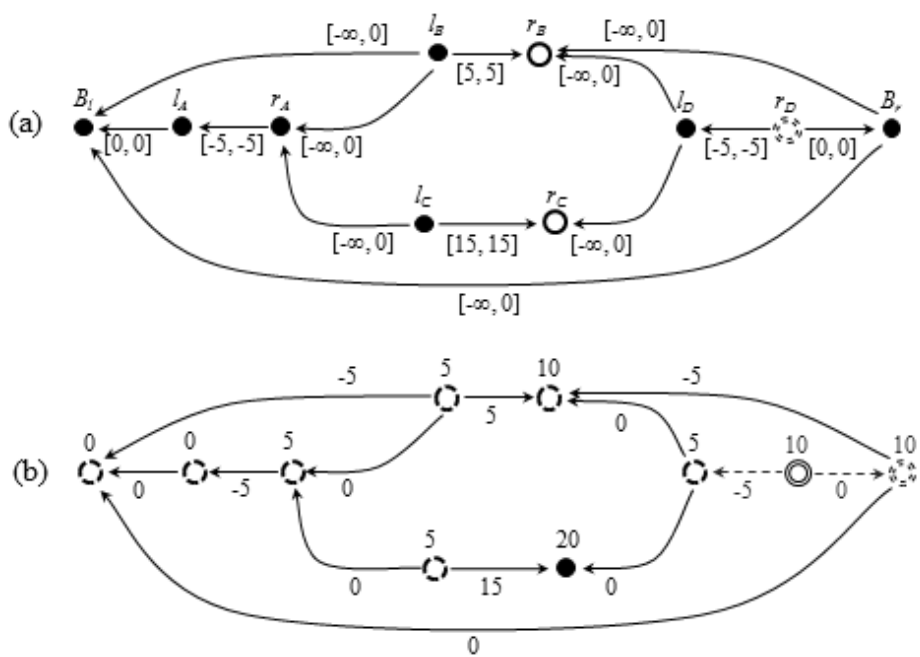


Figure C.13. Solution of the example: Step 12

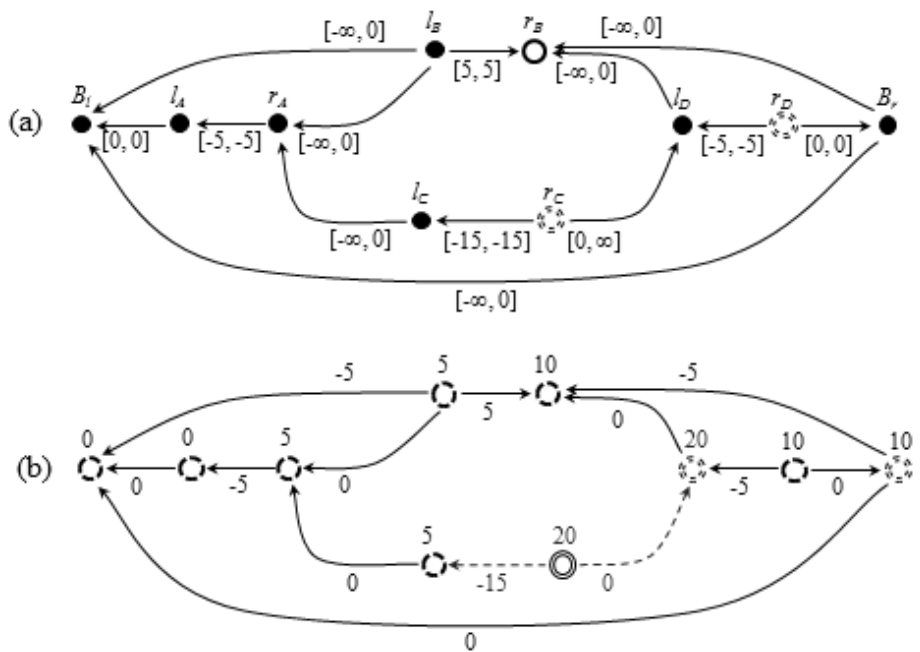


Figure C.14. Solution of the example: Step 13

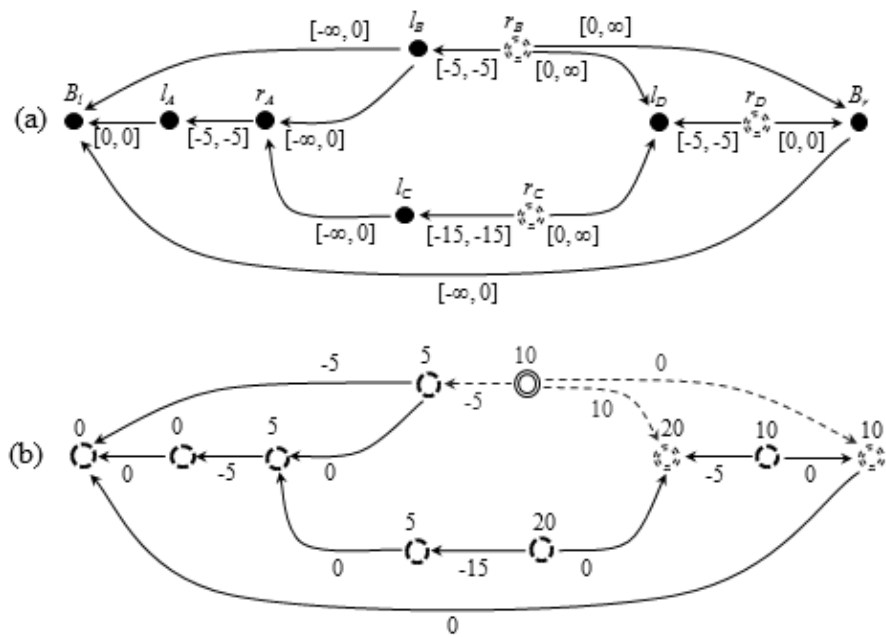


Figure C.15. Solution of the example: Step 14

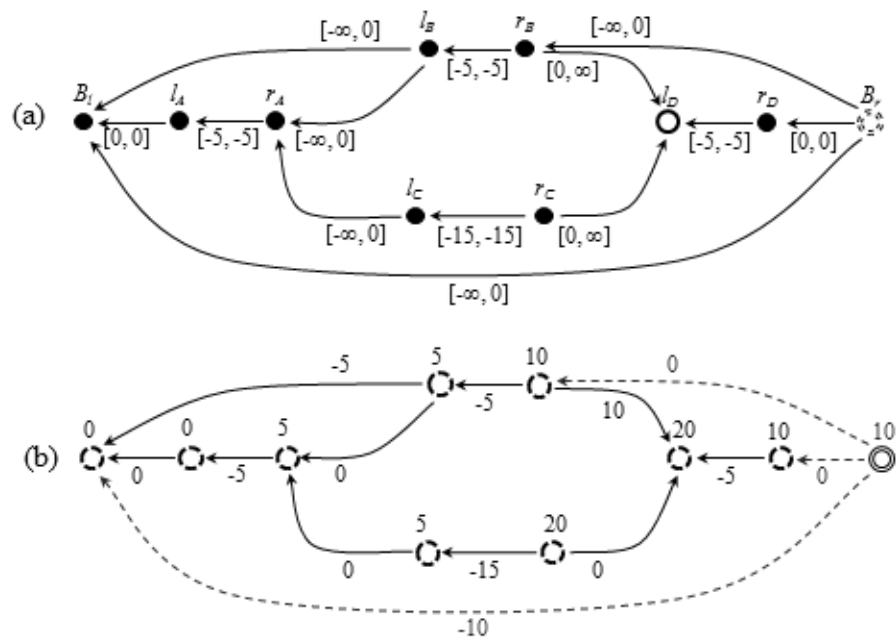


Figure C.16. Solution of the example: Step 15

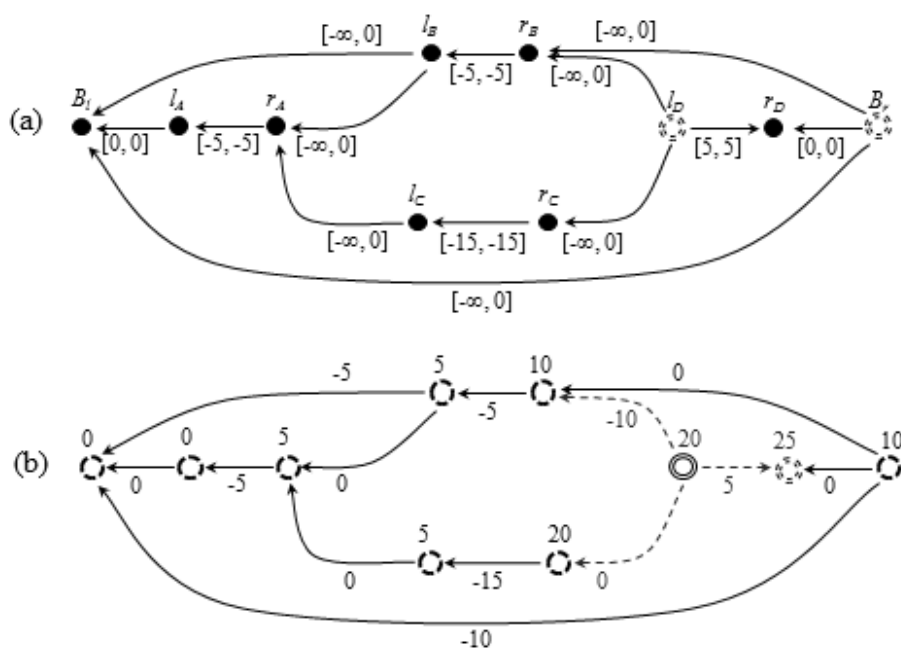


Figure C.17. Solution of the example: Step 16

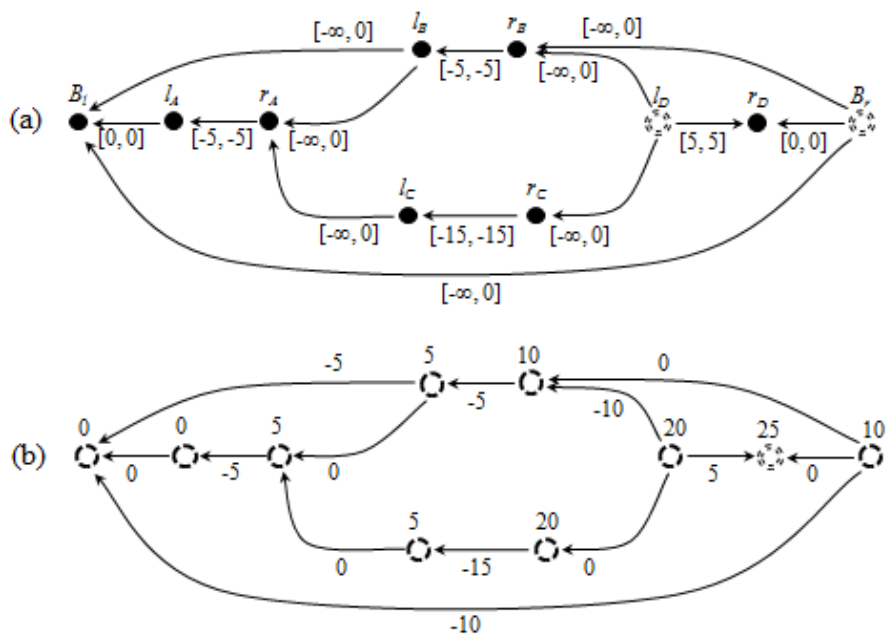


Figure C.18. Solution of the example: Step 17

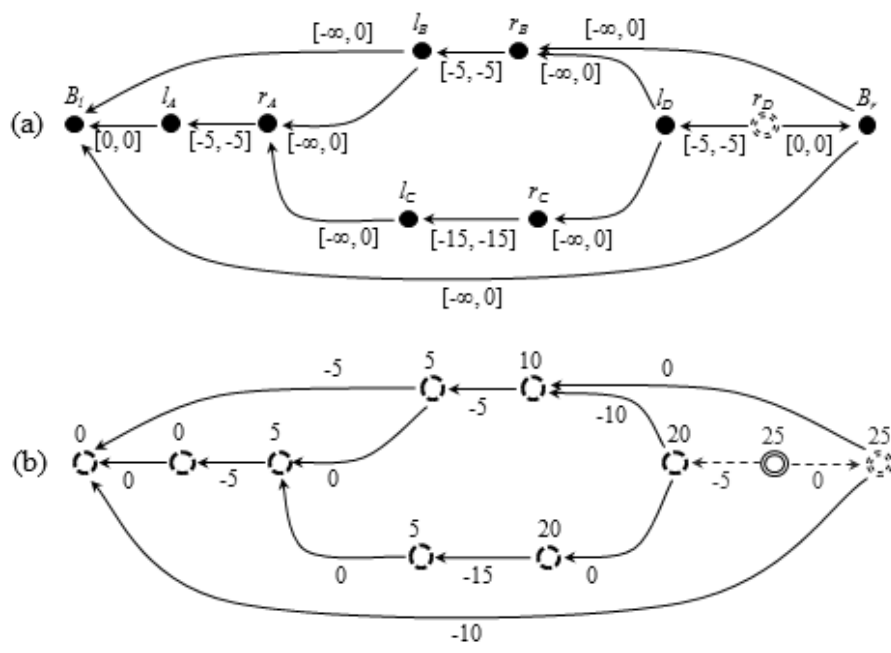


Figure C.19. Solution of the example: Step 18

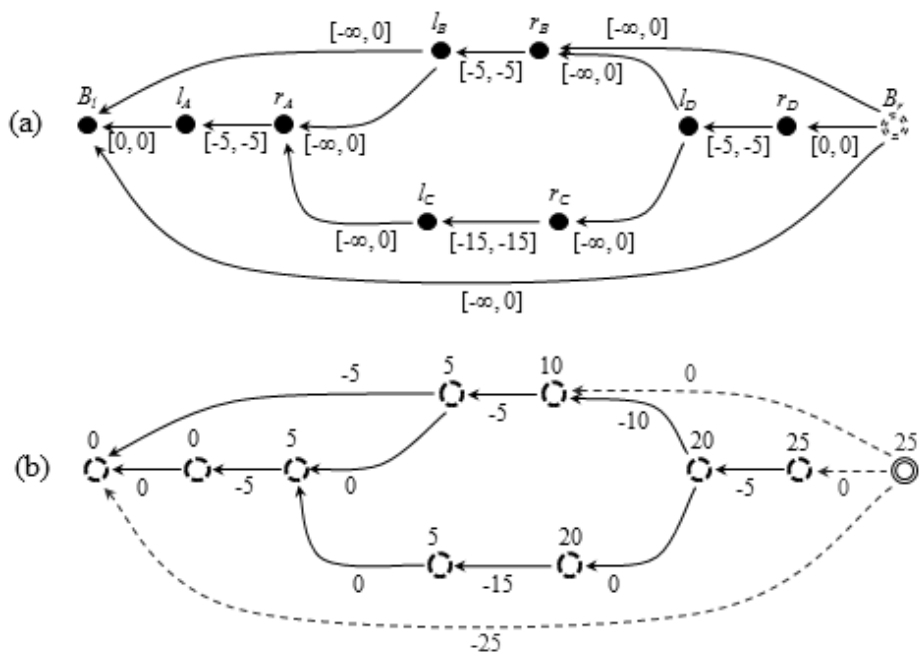


Figure C.20. Solution of the example: Step 19

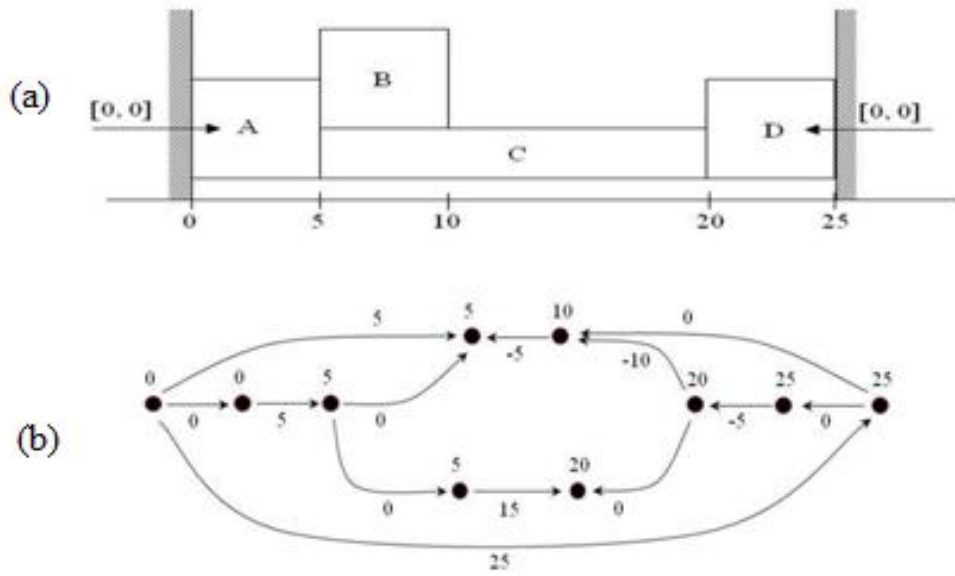


Figure C.21. Solution of the example (a) the floor-plan (solution) (b) graph showing the solution

REFERENCES

1. Gielen, G. G. E. and R. A. Rutenbar, "Computer aided design of analog and mixed-signal integrated circuits", *Proceedings of the IEEE*, pp. 1825-1849, 2000.
2. Antao, B. A. A. and J. A. Brodersen, "Techniques for synthesis of analog integrated circuits", *Design & Test of Computers*, pp. 8-18, 1992.
3. Gerez, S. H., *Algorithms for VLSI Design Automation*, John Wiley & Sons, Chichester, 1999.
4. Lampaert, K., G. Gielen and W. Sansen, *Analog Layout Generation For Performance and Manufacturability*, Kluwer Academic Publishers, Boston , 1999.
5. Rijmenants, J., et al, "ILAC: An automated layout tool for analog CMOS circuits", *IEEE Journal of Solid-State Circuits*, vol. 24, no. 2, April 1989.
6. Cohn, J., et al., "KOAN/ANAGRAM II: New tools for device-level analog placement and routing", *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, 1991.
7. Mogaki, M., et al., "LADIES: An automatic layout system for analog LSI's", *IEEE Int. Conf. CAD*, pp. 450-453, 1989.
8. Meyer, V., "ALASYN: Flexible rule-based layout synthesis for analog IC's", *IEEE Journal of Solid-State Circuits*, vol. 28, no. 3, 1993.
9. Malavasi, E. and A. Sangiovanni-Vincentelli, "Area routing for analog layout", *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1186-1197, 1993.
10. Basaran, B, R. Rutenbar and L. Carley, "Latchup-aware placement and parasitic-bounded routing of custom analog cells", *ICCAD-93*, 1993.

11. Charbon, E., et al., "A constraint-driven placement methodology for analog integrated circuits", *IEEE Custom Integrated Circuits (CICC)*, pp. 28.2.1–28.2.4, 1992.
12. Castro-Lopez, R., et al., "An Integrated Layout-Synthesis Approach for Analog ICs", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1179-1189, 2008.
13. Agarwal, A., et al., "Fast and accurate parasitic capacitance models for layout-aware synthesis of analog circuits", *DAC-2004*, pp. 145- 150, 2004.
14. Ranjan, M., et al, "Fast, layout-inclusive analog circuit synthesis using pre-compiled parasitic-aware symbolic performance models", *Design, Automation and Test in Europe Conference and Exhibition*, pp. 1530-1591, 2004.
15. Sherwani, N., *Algorithms for VLSI Physical Design Automation Second Edition*, Kluwer Academic Publisher, Massachusetts, 1997.
16. Kubo, Y., et al., "Explicit expression and simultaneous optimization of placement and routing for analog IC layouts", *DAC-2002*, pp. 467-472, 2002.
17. Zhang, L., et al., "Parasitic-Aware Optimization and Retargeting of Analog Layouts: A Symbolic-Template Approach", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 791 – 802, 2008.
18. Murata, H., et al., "VLSI module placement based on rectangle-packing by the sequence pair", *IEEE Trans. on CAD*, pp. 1518-1524, 1996
19. Chen, T., et al., "NTUplace3: An Analytical Placer for Large-Scale Mixed-Size Designs With Preplaced Blocks and Density Constraints", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1228-1240, 2008.

20. Chen, S. and T. Yoshimura, "Fixed-Outline Floor-planning: Block-Position Enumeration and a New Method for Calculating Area Costs", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 858 – 871, 2008.
21. Spindler, P., U. Schlichtmann and F. M. Johannes, "Kraftwerk2—A Fast Force-Directed Quadratic Placement Approach Using an Accurate Net Model", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1398 – 1411, 2008.
22. Young, E. F.Y., C. C. N. Chu and M. L. Ho, "Placement constraints in floor-plan design", *IEEE Transactions on VLSI Systems*, vol. 12, no.7, pp. 735 – 745, 2004.
23. Otten, R. H., "Automatic floor-plan design", *DAC-1982*, pp. 261-267, 1982.
24. Wong, D. F. and C. L. Liu, "A new algorithm for floor-plan design", *DAC-1986*, pp. 101-107, 1986.
25. Lai, M. and D. F. Wong, "Slicing tree is a complete floor-plan representation", *DATE-2001*, pp. 228-232, 2001.
26. Nakatake, S., et al., "Module placement on BSG structure and IC layout application", *ICCAD-1996*, pp. 484-491, 1996.
27. Guo, P. N., C. K. Cheng and T. Yoshimura, "An O-tree representation of non-slicing floor-plans and its application", *DAC-1999*, pp. 268-273, 1999.
28. Chang, Y. C., et al., "B*-trees: a new representation for non-slicing floor-plans", *DAC-2000*, pp. 458-463, 2000.
29. Jai-Ming, L. and, C. Yao-Wen, "TCG: a transitive closure graph-based representation for non-slicing floor-plans", *DAC-01*, pp. 764-769, 2001.
30. Hong, X., et al., "Corner block list: an effective and efficient topological representation of non-slicing floor-plan", *ICCAD-2000*, pp. 8-12, 2000.

31. Sakanushi, K. and Y. Kajitani, "The quarter-state sequence (Q-sequence) to represent the floor-plan and applications to layout optimization", *IEEE APCCAS*, pp. 829-832, 2000.
32. Xiaoping, T. and D. F. Wong, "Floor-planning with Alignment and Performance Constraints", *DAC-2002*, pp. 848-853, 2002.
33. Onodera, H., Y. Taniguchi and K. Tamm, "Branch and Bound Placement for Building Block Layout", *ACM*, pp. 433-439, 1991.
34. Takahashi, T., "An algorithm for finding a maximum-weight decreasing sequence in a permutation, motivated by rectangle packing problem", *IEICE Tech. Rep. VLSI Design Technol.*, vol. VLD96, pp. 31-35, 1996.
35. Tang, X., R. Tian and D. F. Wong, "Fast evaluation of sequence pair in block placement by longest common subsequence computation", *DATE-2000*, pp. 106-111, 2000.
36. Yano, Y. and M. Kaneko, "Solution Space Reduction of Sequence Pairs Using Model Placement", *CAS-2007*, pp. 1130 – 1133, 2007.
37. Lin, J. M., H. E. Yi and Y. W. Chang, "Module placement with boundary constraints using B*-trees", *Circuits, Devices and Systems*, pp. 251 – 256, 2002.
38. Jiang, Y., et al., "Module placement with pre-placed modules using the B*-tree representation", *ISCAS-2001*, pp. 347 – 350, 2001.
39. Lee, C., "An algorithm for path connections and applications", *IEE Trans. Electron Computer*, vols. EC-10, pp. 346-365. 1961.
40. Chen, S.-J., et al., "General area router based on planning techniques", *IEE Proceedings on Computers and Digital Techniques*, pp. 413-420, 1994.

41. Chang, Y. and S. Lin, "MR: a new framework for multilevel full-chip routing", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 793- 800, 2004.
42. Stevens, J., A. Hashimoto, "Wire routing by optimizing channel assignment within large apertures", ACM, pp. 155-169, 1971.
43. Rivest, R. L. and C. M. Fiduccia, "A greedy channel router", *DAC-1982*, pp. 418 – 424, 1982.
44. Yilmaz, E. and G. Dundar, "Analog Layout Generator for CMOS Circuits", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 32-45, 2009.
45. dos Santos, G.B.V., et al, Channel based routing in channel-less circuits, *ISCAS-2006*, pp. 337-340, 2006.
46. Chen, Y., et al., "Evaluation of the new OASIS format for layout fill compression", *ICECS-2004*, pp. 377- 382, 2004.