2016

A Genetic Algorithm for ASIC Floorplanning

Anvesh Kumar Perumalla
Wright State University

A GENETIC ALGORITHM FOR ASIC FLOORPLANNING

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

by

ANVESH KUMAR PERUMALLA
B.E., Osmania University, India, 2012

2016
Wright State University

WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

December 16, 2016

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Anvesh Kumar Perumalla ENTITLED A Genetic Algorithm for ASIC Floorplanning BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Electrical Engineering.

J. M. Emmert, Ph.D.
thesis Director

Brian D. Rigling, Ph.D.
Chair, Department of Electrical Engineering

Committee on
Final Examination

J. M. Emmert, Ph.D.

Saiyu Ren, Ph.D.

Raymond E. Siferd, Ph.D.

Robert E.W. Fyffe, Ph.D.
Vice President for Research and
Dean of the Graduate School

ABSTRACT

Perumalla, Anvesh Kumar. M.S.E.E., Department of Electrical Engineering, Wright State University, 2016. *A Genetic Algorithm for ASIC Floorplanning.*

Semiconductor integrated circuits (ICs) have become key components in almost every aspect of our daily lives. From simple home appliances to extremely sophisticated aerospace systems, we have become increasingly dependent on ICs. System-on-chip (SoC) is an IC methodology that includes multiple design technologies on a single IC chip. SoC was developed to further integrate and manage system complexity. Due to SoC and increasingly dense IC fabrication technologies, design time and thereby system time-to-market are becoming more critical drivers of the IC design cycle. In order to address issues related to design time and time-to-market, highly optimized semiconductor intellectual property (IP) is often leveraged. These IP blocks are predesigned, highly optimized sub-system components. To take full advantage of these and other sub-components, we have developed a Genetic Algorithm (Simulated Evolution) based floorplanning tool to quickly and efficiently solve the SoC and application specific integrated circuit (ASIC) floorplanning problem. Our tool takes advantage of both hard and soft macros to optimize IC area usage.

Contents

List of Figures

List of Tables

Acknowledgment

First and foremost, I would like to thank my advisor, Dr. J. M. Emmert, for inspiring me to pursue a career of research in VLSI. His knowledge of physical design automation and CAD tools further helped me with this research.

I would like to thank my committee, Dr. Saiyu Ren and Dr. Raymond Siferd, for giving me greater insights during my thesis presentation.

Additionally, I thank my colleagues, David Bowman, Robert Howells, and Kiran Jayarama. Without them, I would not have managed my time effectively to finish this thesis.

Lastly, I want to thank my parents, Amruthaiah and Sujatha Perumalla, for financially supporting my education, and my finacée, Deanna Brown, for her support and encouragement.

Dedicated to

Dr. J. M. Emmert and Amruthaiah Perumalla

INTRODUCTION

1.1 SoC and Intellectual Property

"The future of integrated electronics is the future of electronics itself" [1]. Semiconductor integrated circuits (ICs) have become key components in our daily lives. Almost every modern device from basic home appliances to highly complex aerospace systems take advantage of some IC [2]. With recent emphasis on the internet-of-things (IOT), usage of highly connected electronics has exploded. The electronic system era began with hand-soldered, printed circuit board (PCB) based electronic components. These components on the PCB were optimized based on the application of the system and the cost of manufacturing the PCB itself [3]. In 1958, Jack Kilby was the first to propose the possibility of producing integrated transistors, diodes, capacitors and resistors on a single semiconductor joined with wire bonds as interconnects. This idea made manufacturing very difficult, and it limited the number of components that would fit on a chip. Later, Robert Noyce suggested deposition of aluminum on planar structures as a replacement for the bonding wires [4]. Based on their ideas, Jack Kilby and Robert Noyce are together credited as the inventors of IC [4].

The transistor is arguably the most important component on an IC. In 1952, radar scientist Geoffery Dummer was the first to coin the idea of the stand-alone IC. He proposed that all electronic devices could be made into a "single block," but in 1956, he failed to produce a working prototype [2]. Jack Kilby was actually the first to make ICs a reality [2].

IC technology has evolved from designs with a few thousand transistors (small-scale integration) in 1960, to integration of millions of transistor designs on very-large-scale-integration (VLSI) chips today [5]. In the beginning, engineers were more concerned with wiring together small scale or large scale ICs. Now they find themselves facing the problem of programming processors, memories, and input/output (IO) controllers to function together [6]. Today, almost every single step in digital IC design is automated [7]. The state-of-the-art (SOA) microelectronics were greatly improved by automated computer-aided-design (CAD) tools [8]. Because the density of transistors on a single die has increased radically within a few decades, automated CAD tool usage has increased [9]. Figure 1.1 shows an example of how this increase has affected commercially available IC based processors.

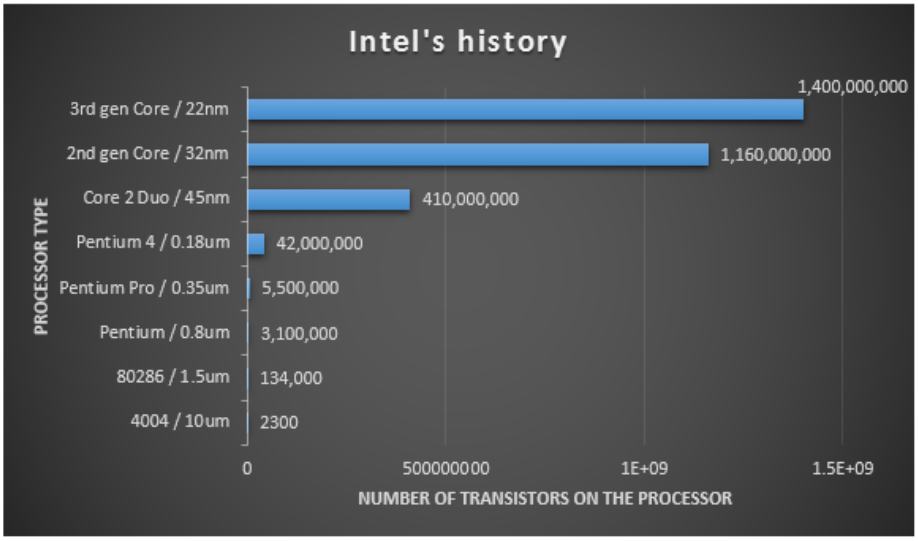


Figure 1.1: Growth of Transistor Density on Commercial Chips [9].

System on chip (SoC) is an IC methodology developed to manage the increased complexity. Intellectual property (IP) blocks, which are pre-designed, highly optimized and pre-verified blocks, are usually obtained from third parties or internal sources and are often combined on a single chip to quickly form very complex IC systems [10]. Semiconductor IP blocks, since they are pre-designed and pre-tested, are a huge innovative factor

for improving time-to-market for many IC chips [2]. Example categories of IC technologies that make use of re-usable IP blocks are SoC, application specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), and embedded systems [2] [10]. Additional examples of reusable IP cores include embedded processors, memory blocks, interface blocks, analog components, radio frequency components and application targeted functional units. Considering time-to-market, SoC designs take advantage of IP cores to integrate all subsystem blocks on a single die.

There are different versions of IP cores that are available in the market. Some of them are hard, fixed macros; some are firm, not completely routed macros; and some are soft, malleable macros [2].

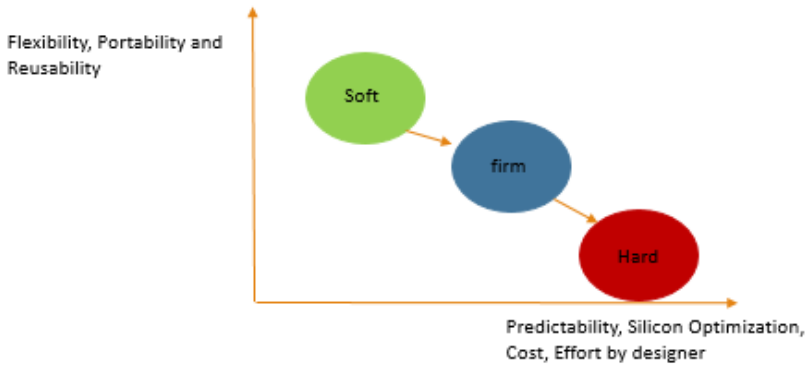


Figure 1.2: Different IP Cores and Their Characteristics [2].

Soft IP: Soft IP blocks are often provided as register-transfer level (RTL) or high-level descriptions. These are the most flexible, often least optimized, and most convenient macros. They are often provided as synthesizable hardware description language (HDL) modules that can easily be adopted by almost any technology node. Sometimes this HDL is provided in an encrypted format, which cannot be modified or read. Encrypting prevents user modification and sometimes makes adoption more challenging. It should be noted that soft IP usually has a flexible shape and orientation when incorporated into design layout.

Hard IP: Hard IP blocks are provided as fixed layout (pre-placed and routed), (silicon) pre-verified, highly optimized blocks. Because of their lack of portability (from fabrica-

tion node to fabrication node), the application of these blocks is very limited. They must be completely redesigned for each technology node. The greatest advantage of hard IP is that it is silicon proven, and thus provides highest probability that it will work at its specified conditions. It should be noted that often hard fixed IPs have firm fixed shapes and orientations.

Firm IP: Firm IP blocks are distributed as parameterized circuit descriptions. These can be optimized for specific design needs. These are more flexible than hard IP and more predictable than soft IP. Some have a flexible shape and some do not.

1.2 ASIC Design Flow

The high volume semiconductor industry we know today started in the early 1970s and has exponentially increased productivity. Since then, chips have become denser and denser. The introduction of massive, high volume VLSI fabs in the1970s and 1980s did a lot to encourage the introduction of ICs into micro-electronic systems. The introduction of application specific integrated circuits (ASICs) has taken the semiconductor industry to next level [11]. These ASICs allow cheaper routes for many companies both large and small to introduce integrated circuits into their products.

Types of ASICs: Most ICs are made on thin silicon wafers, where each wafer is composed of hundreds of IC die. Each individual IC die usually consists of millions of transistors, which are connected by many layers of metal. Masks are designed to specify the layout and topology of each IC die. Below are some of the different categories of common digital IC die.

Full-Custom ASICs: For full custom ASICs, engineers design circuits and transistors in the ASIC layout. Designers do not use any pre-characterized cells. It is the most expensive relative to design and manufacture methods because of the cost involved in designing each piece within the IC. However, this technique, though expensive, results in the high-

est quality, highest performing ICs. The biggest drawback is that it is also the most time consuming type of IC design. It would usually only be used for very high volume circuits.

Semi-Custom ASICs: This is the most widely used ASIC design method in the industry. All the logic leaf cells are predesigned and some of the mask layers are customized. Building this type of IC is easier because of the use of automated tools and predesigned cells from a standard or macro cell library. Design time is much faster, and therefore it improves time-to-market.

There are two types of semicustom ASICs: standard cell based ASICs and mask programmable gate array based ASICs. Both use predefined logic and take advantage of automated CAD tools to complete the designs. Standard cell based ASICs offer higher performance but take longer (three to four times longer) to fabricate.

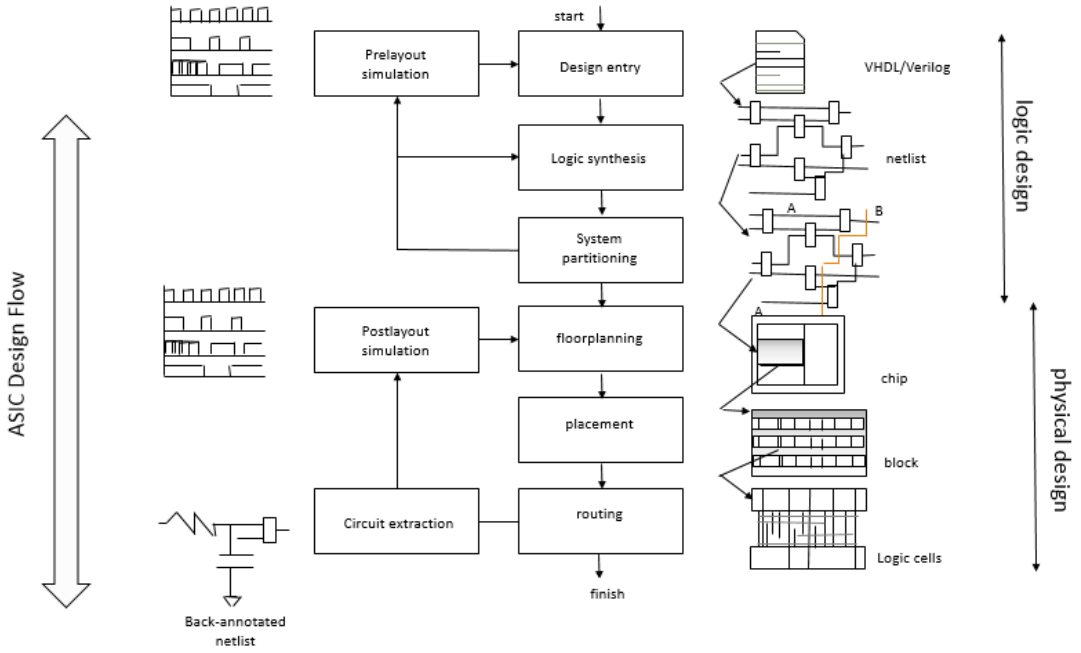


Figure 1.3: Typical ASIC Design Flow [11].

The basic automated ASIC design flow [11] [12] usually starts with a hardware description language (HDL) version of the design. VHDL and Verilog are the most widely used HDL languages. A simulated and synthesizable version with the desired functionality

of the IC is coded using an HDL language. Functionality is verified with what is referred to as pre-layout simulation. Once functionality is verified, the HDL IC code is passed to the high-level logic synthesis stage.

During high-level logic synthesis, the given high-level or behavioral netlist is converted into a gate level netlist using standard and macro cell libraries. Once the gate-level netlist is generated, functionality is verified again (via simulation) to check whether it still meets the original design specifications.

System partitioning is required if not enough pins or area are available. This step is completed after high level synthesis. Functionality is verified again via simulation.

The first steps in the automated design process are referred to as high-level synthesis. These include early front-end stages, behavioral stages, or high-level synthesis stages as the logic design phase.

The physical design stages come next. Physical design takes the front end result (or the gate level netlist) and generates a transistor level circuit layout using standard and macro cell technology libraries. Initially when we start the physical design, we start with zero-wire-load models. Then after routing, we have an option of extracting wire lengths and parasitic values to simulate more accurately the design. Post-layout simulation is performed using the back annotated netlist and is functionally verified again. Below we describe in more detail the physical design flow.

1.3 Physical Design Flow

Physical design [5] [13] [14] [15] is the procedure of converting synthesized gate level netlists into a transistor level circuit layout. Steps that are included in this flow are explained below:

Design Import: This is the beginning stage of the physical design cycle. In this stage, the gate-level netlist, which is generated during the logic synthesis stage, is loaded into the

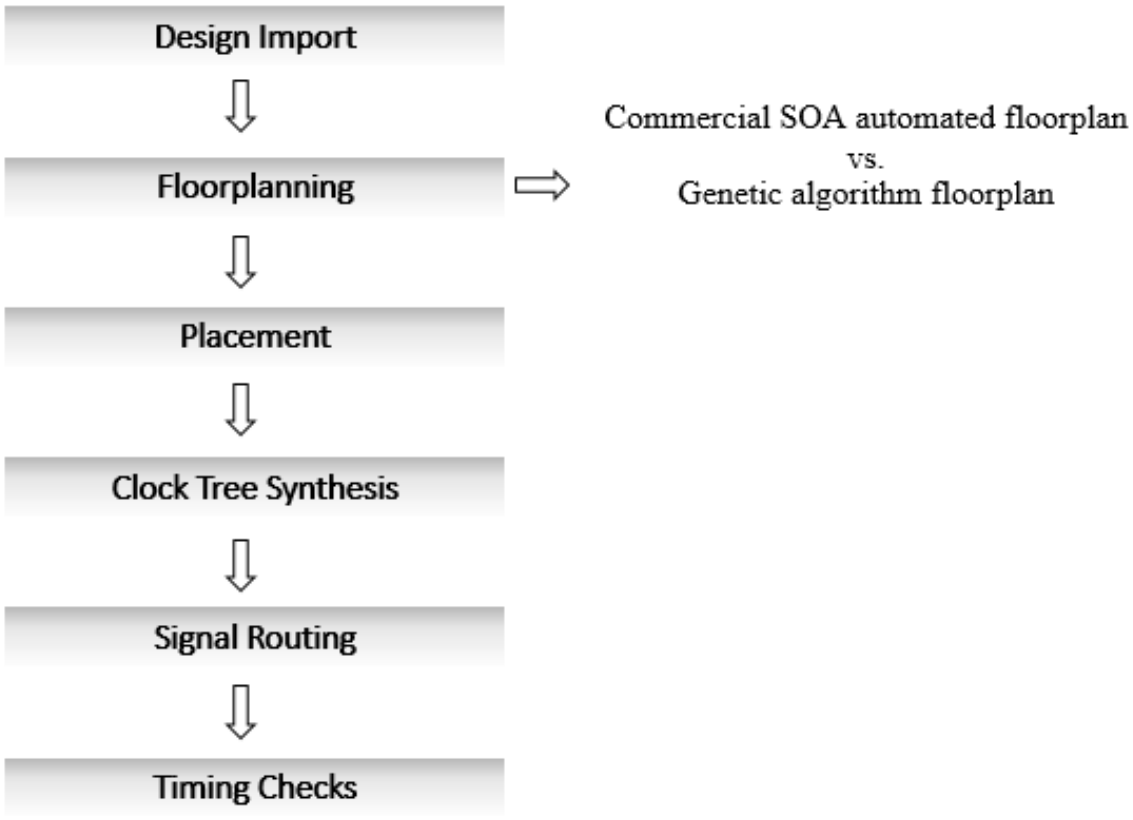


Figure 1.4: Typical IC Physical Design Flow [14].

tool along with the corresponding target technology libraries.

Floorplanning: The floorplan is the crucial step in the physical design. In this stage, the relative placement of macros or large circuit blocks are decided. Most of the time, these macros are available as hard and soft macros. Floorplanning is the focus of this thesis, and it is critical to obtaining good results in the remaining physical design steps.

Placement: During the placement stage, all the rest of the standard cells (glue logic) are placed on the die. After placement, every logic component of the design will have specific (x,y) locations on the circuit layout.

Clock Tree Synthesis: Clock signals are the most critical signals on the entire chip alongside power and ground nets. Usually there are thousands to tens of thousands of flip-flops in every IC circuit. Every flip-flop is triggered by a clock signal. Designing an efficient clock tree to minimize jitter and lag is very difficult.

Routing: Routing is the stage where standard cells and macros are connected. There are different metal layers available for the connections. Vias are used to connect the metals on different layers.

When macros are not placed properly, it effects the way the standard cells are placed. If the standard cells are not placed properly, it will be difficult to perform clock tree synthesis. Eventually, this will also affect signal routing. The floorplan step, then, is the key step in the IC design flow.

ICs in modern electronic systems are area driven. The implication is that with the advancement of fabrication technology nodes, the cost per area is rising. This leads to demand for designing ICs with as little area as possible. The floorplan plays a major role in deciding the area of the IC chip.

As an example, below (Figure 1.5 , Figure 1.6) is the floorplan of four hard macros in a netlist created by a commercial SOA tool. It is compared with a genetic algorithm floorplan. In both cases, the goal is to reduce the overall area of the circuit. The rest of the

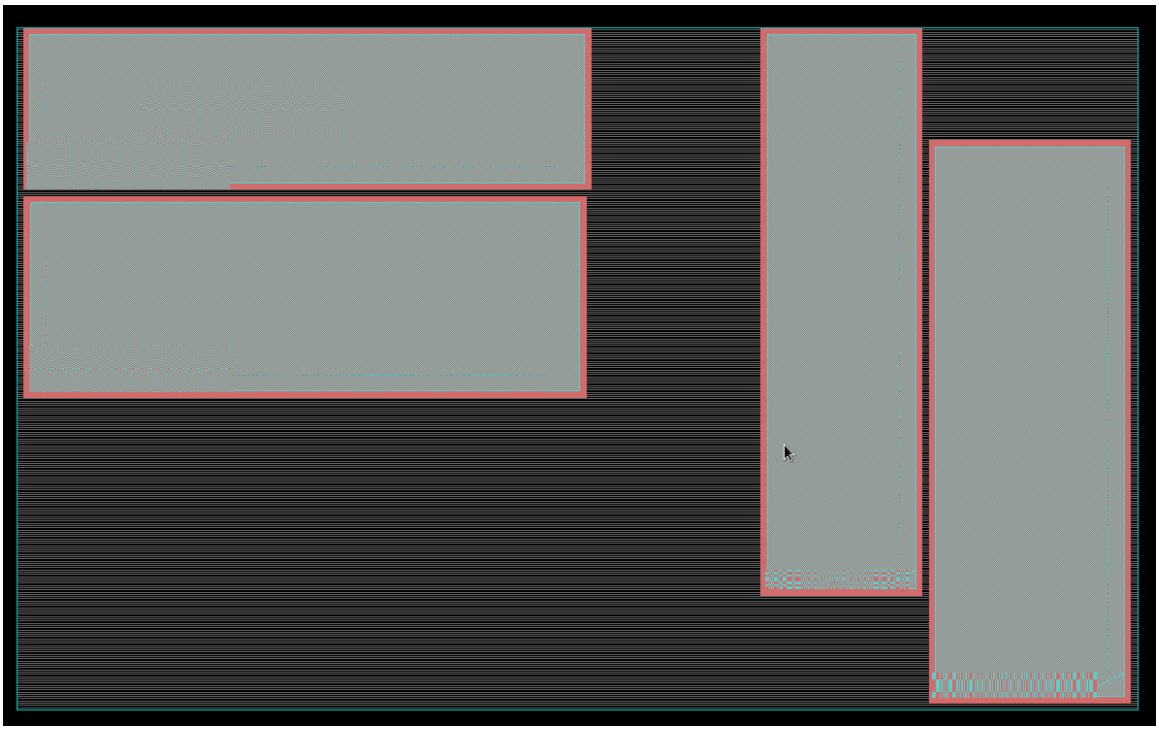


Figure 1.5: Floorplan of a Netlist Created by Commercial SOA Tool [16].

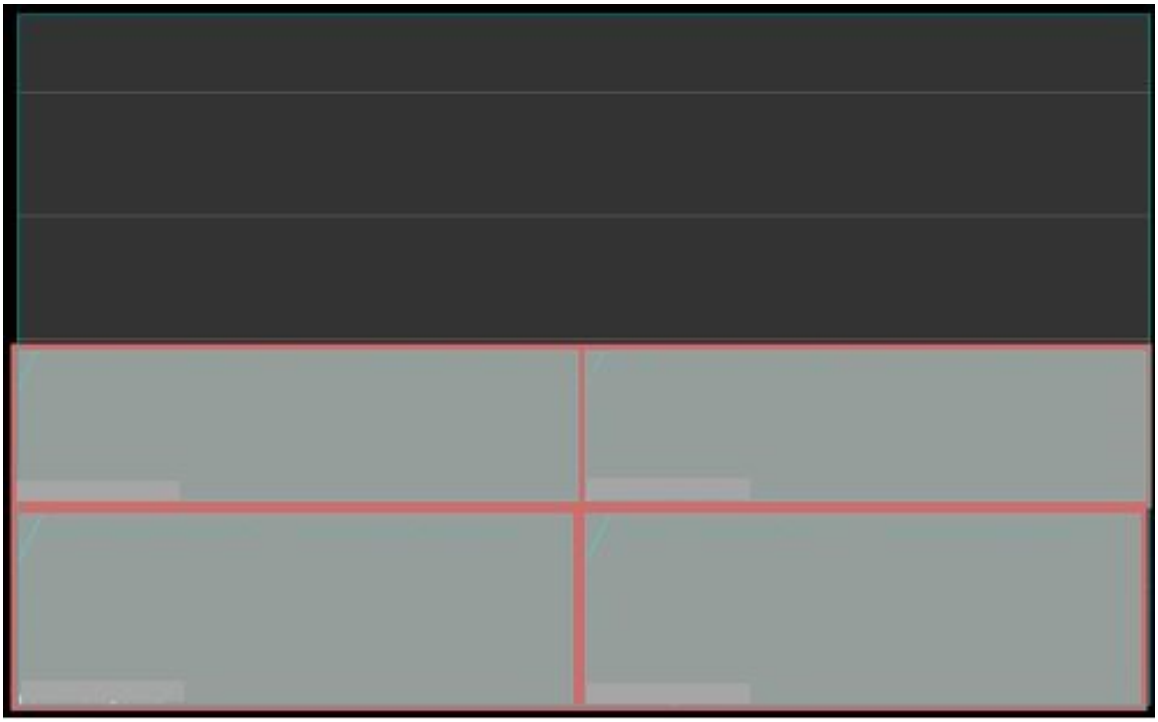


Figure 1.6: Floorplan Created by Applying Genetic Algorithm.

empty area is used by unplaced standard cells.

For our research, we are not limited to just four blocks. We show a netlist (Figure 1.7) which has 29 macro blocks below:

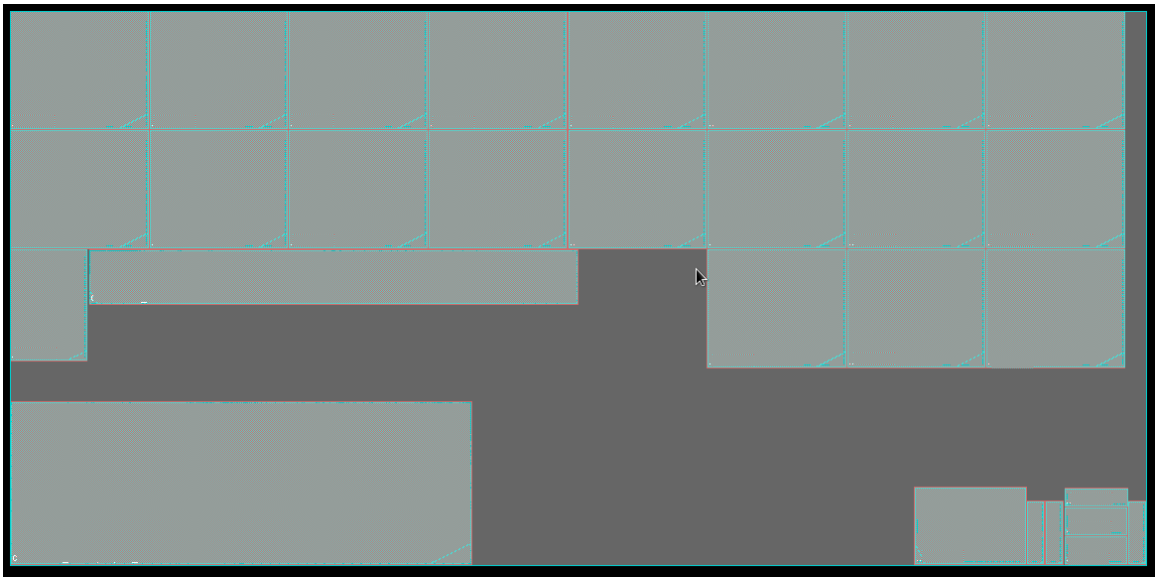


Figure 1.7: Commercial SOA Tool Floorplan for a Netlist of 29 Macros [16].

Our goal is to improve further the area used by the IC. In all these cases, the commercial SOA tool uses high effort to place the macros.

1.4 Graph Theory and CAD

Most electronic design automation (EDA) problems are represented using graphs and data structures. Algorithms to address EDA problems operate on the graphs using common data structures [5] [13] [14]. Designing an efficient algorithm for solving EDA problems is very challenging. To improve the probability of success when solving EDA problems, one efficient approach is to model the given problem using graph data structures and then to apply efficient algorithms [17].

Figure 1.8 is an example circuit and one of many graphical representations of it.

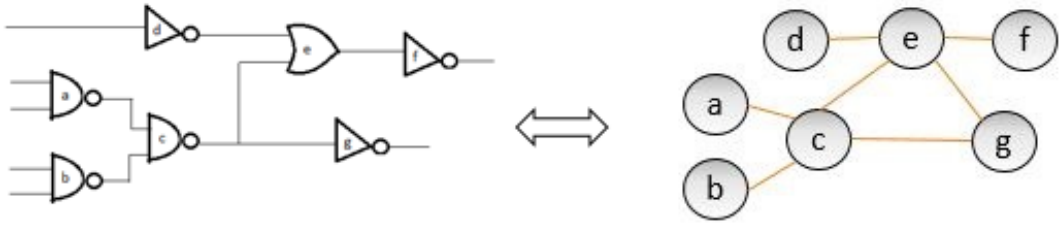


Figure 1.8: Sample Circuit and its Graph Representation.

In a similar way, we can represent a layout floorplan as a graph (Figure 1.9):

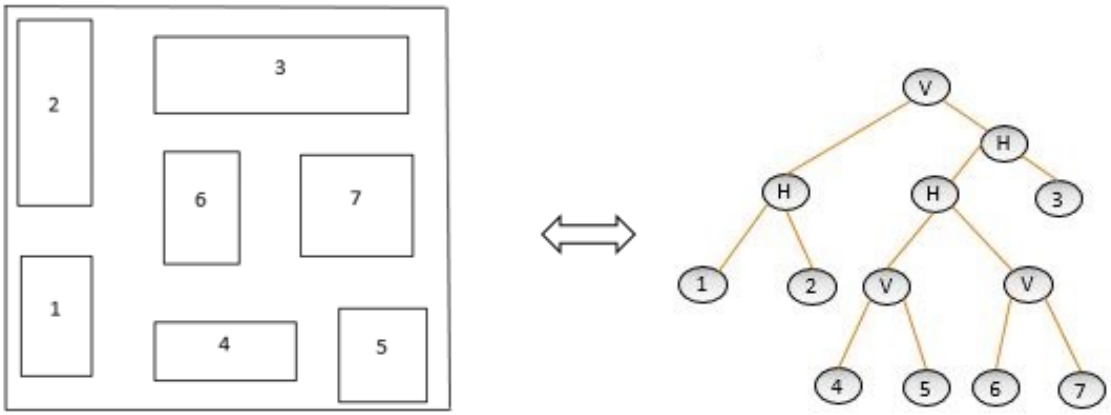


Figure 1.9: Sample Floorplan and its Graph Representation.

One of the biggest challenges is to find an optimal solution to the floorplanning problem within a reasonable amount of time. First, quick running heuristic algorithms are applied to find acceptable, if not optimal, solutions. Then, if time or computer resources permit, they can be further improved [13] [17].

Problem Classification: Most IC automated physical design problems can be classified into two types: decision problems and optimization problems. Decision problems only require yes or no answers, but optimization problems require a search for the best possible solutions.

One way to compare the difficulty of problems related to automated IC design is

through complexity analysis. The P and NP complexity classes are used to describe the difficulty of these types of optimization problems. Most of the optimization problems related to automated IC design are either NP hard or NP complete problems. However, in physical design automation (PDA), even if there is not enough time for an optimal solution, there is a need for some kind of solution [5].

To produce reasonable (if not optimal) solutions to PDA problems, designers are left with four choices of algorithms:

1. *Exponential Algorithms*: These are applied for exponential problems with smaller input sizes. Integer programming is an example [5].

2. *Special Case Algorithms*: When certain restrictions are applied to solve problems, these problems are solved using special case algorithms. For example, standard cells are easier to place if all the cells are the same height (site). Here, height is the restriction applied to solve the problem.

3. *Approximation Algorithms*: These are used when it is not necessary to find the optimal solution (i.e. just meet thresholds) or if near optimality is sufficient.

4. *Heuristic Algorithms*: Heuristic algorithms are classified as deterministic, stochastic, and structure based (constructive and iterative) [14].

Deterministic Algorithms: For a given input, if the algorithm gives the same result every time, then it is a deterministic algorithm.

Stochastic Algorithm: A stochastic algorithm is one based on random processes that give different results for the same input each time the algorithm is run. Simulated annealing is an example of stochastic algorithm.

Constructive Algorithm: A constructive algorithm starts with an incomplete solution and obtains a complete solution from it.

Iterative Algorithm: If an algorithm starts with an initial complete solution and works to improve that solution until it reaches a termination point, then it is an iterative algorithm.

Solution Quality: Most PDA algorithms are heuristic in nature. Sub-optimality [14] ε

of the solution is known based on optimality

$$\varepsilon = \frac{|cost(S_H) - cost(S_{opt})|}{cost(S_{opt})} \tag{1.1}$$

where $cost(S_H)$ is the cost of the heuristic solution S_H and $cost(S_{opt})$ is the cost of the optimal solution S_{opt}.

FLOORPLANNING

2.1 Introduction

Routinely, designers are challenged by time-to-market due to the constantly decreasing size of the transistor that causes ever-increasing chip densities. One solution to this problem is IP and design re-use. Most modern commercial VLSI designs have a large number of commercial IP blocks, which are modeled as blocks on a chip [18]. Floorplanning entails assigning locations and shapes (for soft macros) to all subsystem blocks in the design. We are addressing the problem of the location of these different, oddly shaped blocks.

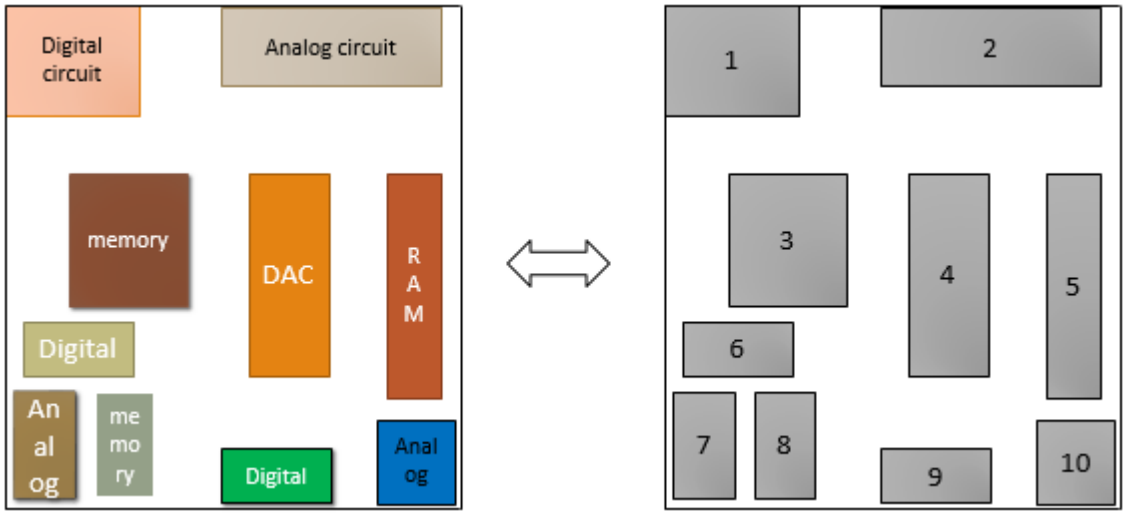


Figure 2.1: Sample ASIC prototype and its Floorplan Model.

2.2 Problem Statement

Let $B = \{b_1, b_2, ...b_m\}$ be the set of m rectangular modules, where w_i, h_i and a_i, 1<i<m are width, height and area of the modules on a chip, respectively [7] [13] [17]. The floorplan F is the assignment of the lower left co-ordinate (x_i,y_i) for each module b_i, 1<i<m such that no two modules overlap each other [5] [7] [13] [17]. The goal of the floorplanner is to place relatively all modules and to assign shapes as necessary such that the objective or cost function is optimized. In this thesis, the cost function is the area occupied by all the blocks on the die. Minimizing the cost function means minimizing the overall area occupied by the blocks on the die.

2.3 Floorplan Models

Floorplanning is classified into two categories: (1) slicing floorplans and (2) non-slicing floorplans [5] [13] [19].

Slicing Floorplans are obtained by slicing the layout horizontally and vertically.

Non-Slicing Floorplan are floorplans that cannot be sliced horizontally or vertically. A non-slicing floorplan cannot be represented by a slicing tree. A wheel is an example of a non-slicing floorplan (Figure 2.3).

A binary tree can be used to represent a slicing floorplan. A slicing tree is a type of binary tree with cut types H and V as nodes (operators) and blocks/modules as leaves (operands). An H cut divides the floorplan horizontally, and a V cut divides the floorplan vertically. By performing horizontal cuts, top and bottom children are formed. The children of horizontal cuts are either leaf cells/modules or sub-floorplans. By performing vertical cuts, left and right children are formed. The children of a vertical cut are either leaf cells/modules or sub-floorplans. An example of slicing and non-slicing floorplans are

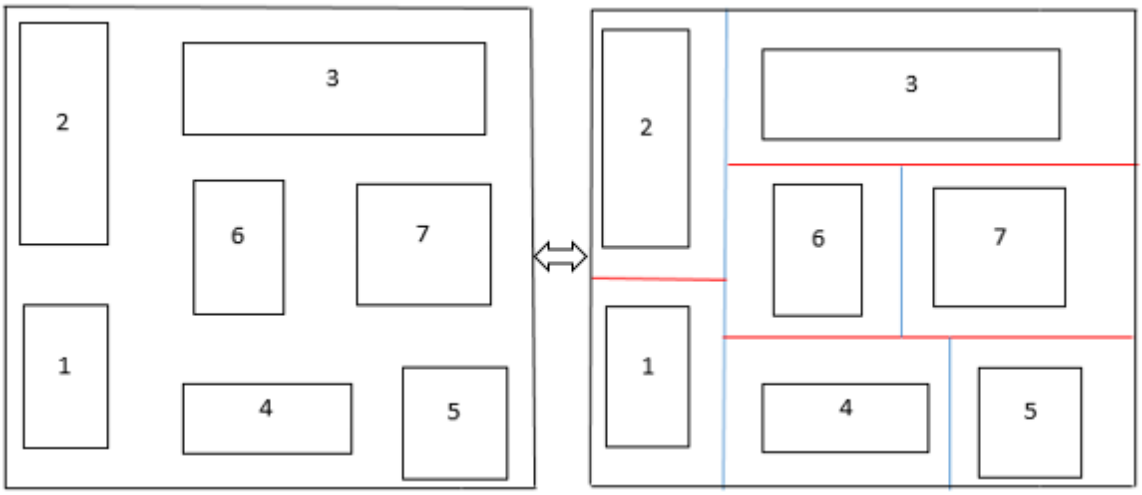


Figure 2.2: Sample Floorplan and its Corresponding Slicing Floorplan.

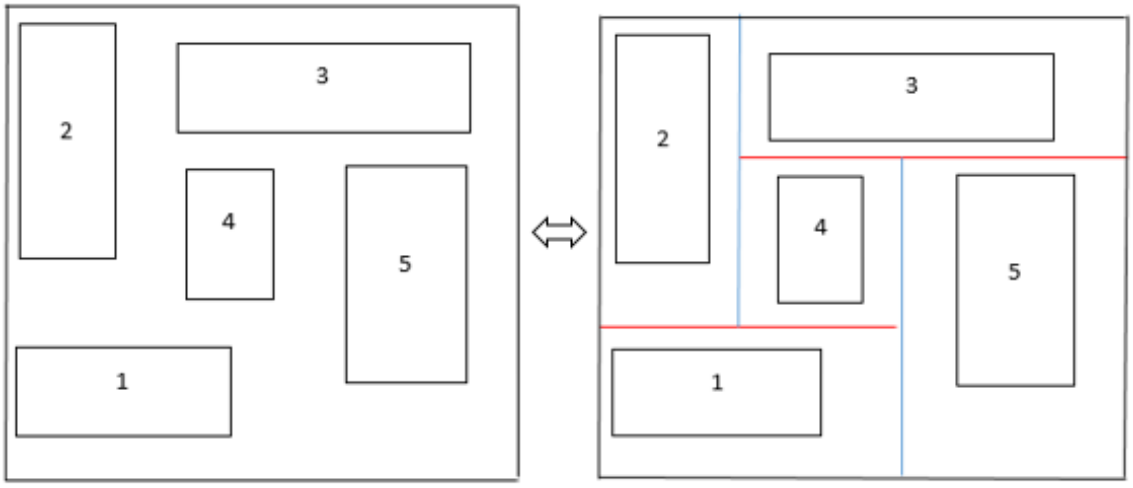


Figure 2.3: Sample Floorplan and its Corresponding Non-Slicing Floorplan.

shown in Figure 2.2 , Figure 2.3:

A slicing floorplan can be represented as more than one slicing tree. This actually leads to large ambiguous solution spaces and hence becomes more complex during optimization. Therefore, skewed slicing trees are introduced. Skewed slicing trees are defined as trees where no node is the same operator as its right child. For example, if a node represents an operator V, then its right child must be either an operator H or an operand. It cannot be a V operator. A non-skewed slicing tree can have nodes where the operator and its right child are the same [17]. For example, if a node represents an operator H and its right child is an operator H then it is a non-skewed slicing tree. The advantage of using a skewed representation is that skewed slicing trees represent a unique layout. On the other hand, for a non-skewed slicing tree there are more than one possible layouts.

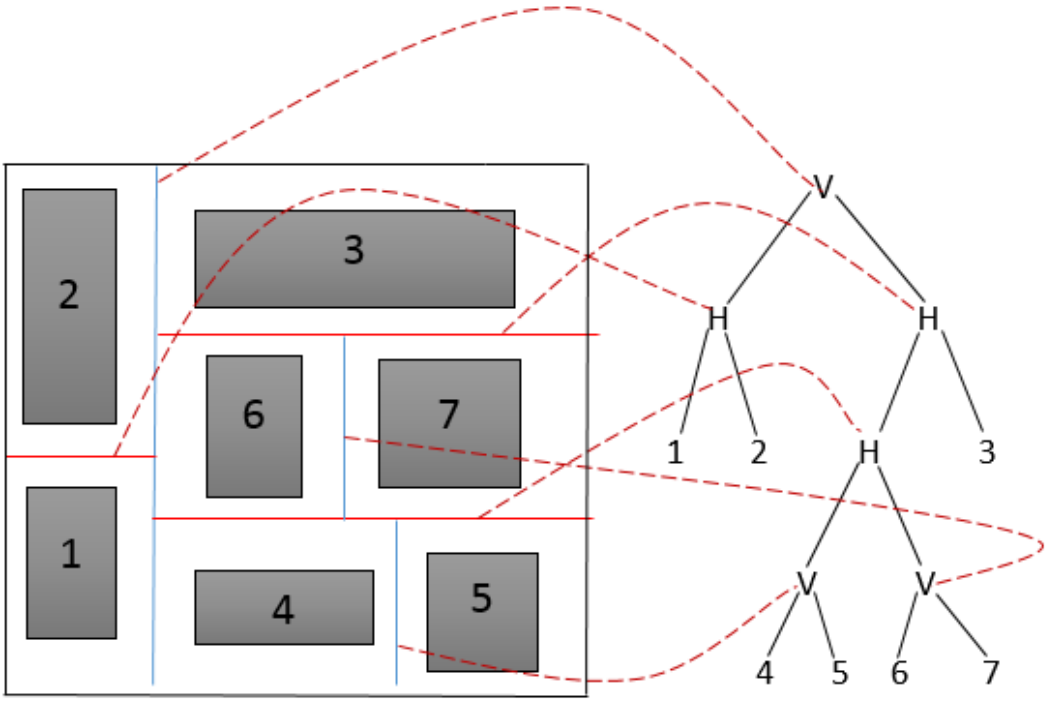


Figure 2.4: Sample Floorplan and its Corresponding Skewed Slicing Tree.

An example of a non-skewed slicing tree is shown in Figure 2.5 :

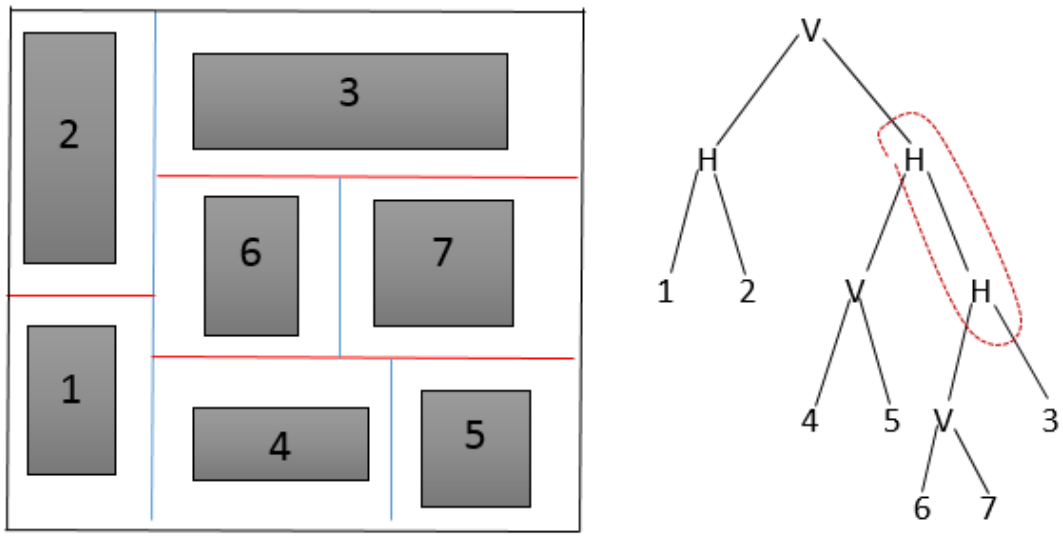


Figure 2.5: Sample Floorplan and its Corresponding Non-Skewed Slicing Tree.

2.4 Polish Expression for Floorplan

A polish expression of length (2n-1), $E = e_1e_2e_3.....e_{2n-1}$, where $e_i \in$ {1, 2, ... , n, H, V}, $1 \leq$ i $\leq$ 2n-1, if and only if:

- Every operand k, $1 \leq$ k $\leq$ n, appears exactly once in the expression; and
- The expression E has the balloting property, i.e., for every sub-expression $E = e_1e_2e_3.....e_{2n-1}$, the number of operands is greater than the number of operators [7] [13] [17] [18].

For example: E = 12H34H5HV as shown in Table 2.1

Table 2.1: Verification of Balloting Property

Polish Expression	**1**	**2**	**H**	**3**	**4**	**H**	**5**	**H**	**V**
Number of Operands:	1	2	2	3	4	4	5	5	5
Number of Operators:	0	0	1	1	1	2	2	3	4

H and V are called operators that represent horizontal and vertical cuts. 1,2,3,.... n are the n operands.

```
Algorithm PostorderTraversal(T: Binary Tree)
Begin
   If Root(T) ≠ nil Then
    Begin
       PostorderTraversal(LeftSubtree(T));
       PostorderTraversal(RightSubtree(T));
       Visit(root(T));
      End;
  End.
```

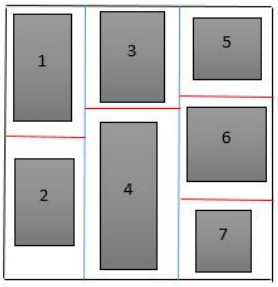
Figure 2.6: PostOrder Traversal of a Binary Tree [7]

A polish expression is the representation of the skewed slicing tree using the postorder traversal algorithm is shown in Figure 2.6:

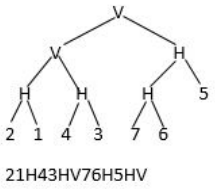
A slicing floorplan is represented as more than one slicing trees, as shown in Figure 2.7. But, there is one-to-one correpondence between normalized polish expression and slicing floorplan [7] [17].

In order to reduce the solution space, only normalized polish expressions are considered. A normalized polish expression is one with no consecutive H or V operators, that is, there should not be any HH or VV patterns throughout the expression. This normalized polish expression corresponds to a unique skewed slicing tree and to a unique floorplan with operands that follow the bottom-up and left-right approach [5] [7] [17].

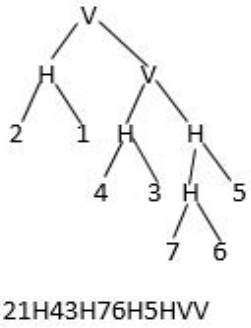
A slicing floorplan has many slicing trees, but a skewed slicing tree has only one slicing floorplan. In other words, a slicing floorplan has many polish expressions, but a normalized polish expression represents one unique floorplan. If a and b are two modules or sub-floorplans, then abH implies that a is below b when divided by a horizontal cut (Figure 2.8). Similarly, abV implies operand a is left of operand b when divided by vertical cut (Figure 2.9).



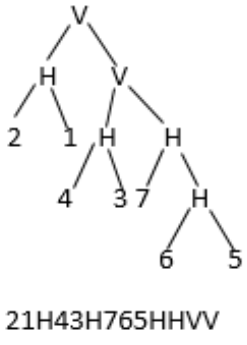
(a) Sample Floorplan



(b) Normalized Polish Expression



(c) Un-Normalized Polish Expression



(d) Un-Normalized Polish Expression

Figure 2.7: Sample Floorplan and its Different Representations.

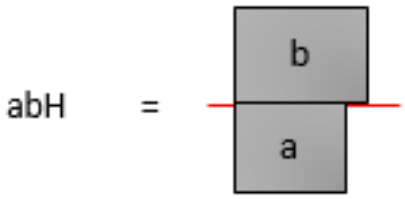


Figure 2.8: Horizontal Cut Representation.

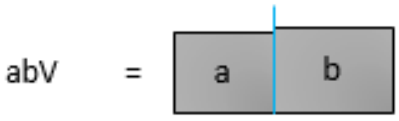


Figure 2.9: Vertical Cut Representation.

2.5 Perturbation

Perturbation is the process of modifying an existing solution to form a new or different solution. Below, we describe how slicing floorplans have traditionally been represented and how common methods (Op_1, Op_2 and Op_3) have been applied for perturbing a solution to get a new or different solution [7] [17] [18]. We have come up with ways to modify these so that genetic algorithms (GAs) can easily be applied to slicing floorplans.

Let$C$ = $c_1c_2c_3.....c_k$ be a chain of k operators, where k is the length of the chain. No two adjacent operators in the chain are the same (if and only if $c_i \neq c_{i+1}$, $1 \leq i \leq k$-1), and every c_i is in {V,H} [7].

Let E = $e_1e_2e_3.....e_{2n-1}$ be a normalized polish expression that is also expressed as $P_1C_1P_2C_2.....P_nC_n$, where C_i represents chain i of operators, and P_i represents operand i (note: some C_i chains may be empty) [7]. There are $P_1P_2P_3....P_n$ different operands within the set of operands, {1,2,....,n}. Two operands are called adjacent if they are consecutive elements in $P_1P_2P_3....P_n$. Adjacent operands have no operator, {V,H}, between them. An operand and an operator are said to be adjacent if they are consecutive elements in E = $e_1e_2e_3.....e_{2n-1}$ [18].

Three basic types of operations are used to perturb the floorplan and go from one normalized polish expression to another [17].

Op1: Swapping two adjacent operands:

This corresponds to swapping the two adjacent operands in the normalized polish expression. By performing this operation, the balloting property will not be affected, and the resulting polish expression is a valid normalized polish expression.

Op2: Inverting the chain:

Normalized polish expression is shown in Figure 2.10; H, H, and HV are the chains. If we are to invert the chain HV, it becomes VH [7] [17]. By performing this operation, the balloting property will not be affected, and the resulting polish expression is a valid

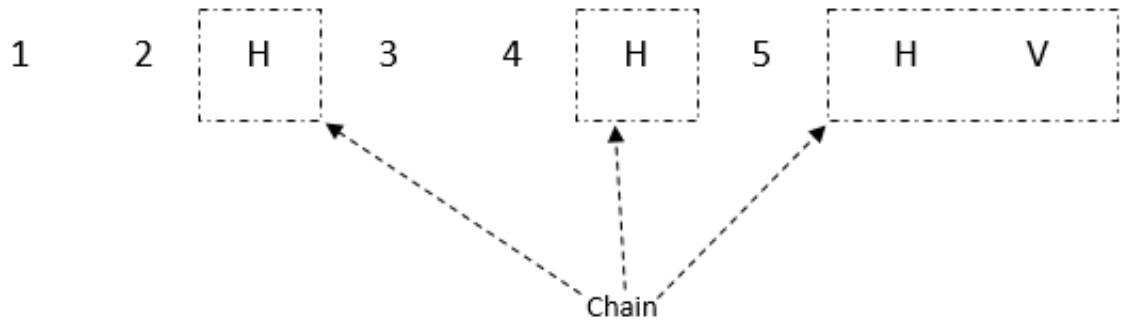


Figure 2.10: Identifying Chains in the Polish Expression.

normalized polish expression.

Op3: Swapping the adjacent operand and the operator:

Performing Op1 and Op2 produces normalized polish expressions without violating the balloting property. However, by performing Op3, there is a possibility of violating the balloting property (which could result in a non-unique floorplan by forming a non-skewed slicing tree) [7] [17]. The result of Op3 is only valid (kept) as long as the balloting property is not violated.

A normalized polish expression, which is generated by performing these three operations, produces a $neighboring$ solution for a given normalized polish expression. In Figure 2.11, we show an example of each of the operations. First, we see operands 3 and 4 swapped using Op_1. The floorplan result shows how module 3 and 4 are swapped in the layout. Next, we see operators in the chain C_5 = VH swapped by Op_2 to give C_5 = HV. Finally, Op_3 is applied to swap the adjacent operator H and operand 4 to give 4H, which produces another polish expression that satisfies the balloting property. We only perform Op_3 if the resulting expression satisfies the balloting property.

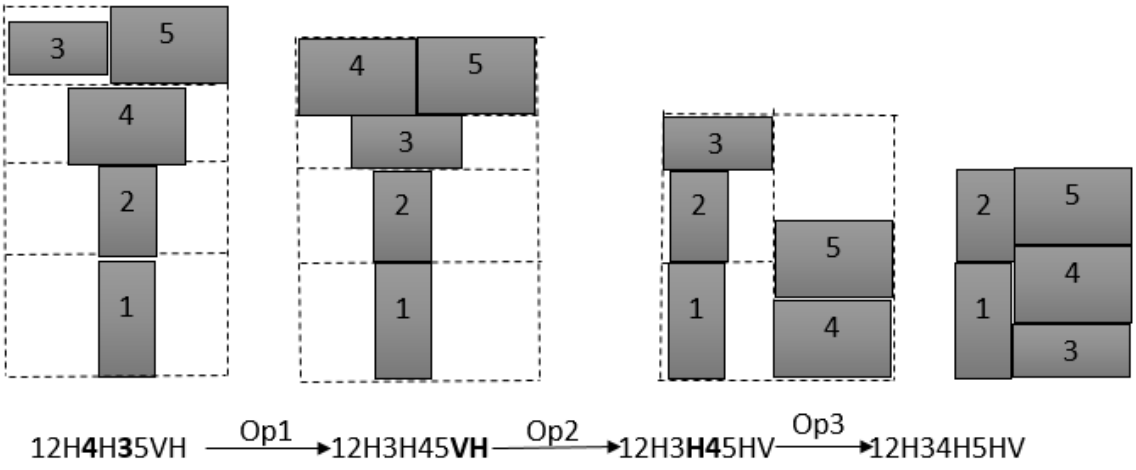


Figure 2.11: Basic Perturbation Moves.

GENETIC ALGORITHM

3.1 Introduction

GAs, also called simulated evolution algorithms [20] - [24] are computational procedures that mimic genetic processes. $Chromosomes$ contain information about the behavioral features of living organisms. All genetic algorithms start with an $initialpopulation$, which is a set of randomly generated solutions to a given problem. From this initial random population, a new population is generated. In each iteration of the GA, a new generation is obtained based on the idea of obtaining better solutions from previous generations.

During reproduction, two parents are selected, and an offspring chromosome is generated by inheriting portions of chromosomes from both of the parents.

3.2 Genetic Terminology

Basic genetic terminology starts with an initial population, which is a set of generated individuals (usually suboptimal solutions). These initial populations are perturbed to form more healthy populations (better solutions). Each individual (solution) in that set is called a chromosome; each element in that chromosome is called a $gene$ [20].

A fitness value is calculated for each individual in the population. The calculation is highly dependent on the actual application modeled by the GA (in our case, floor-

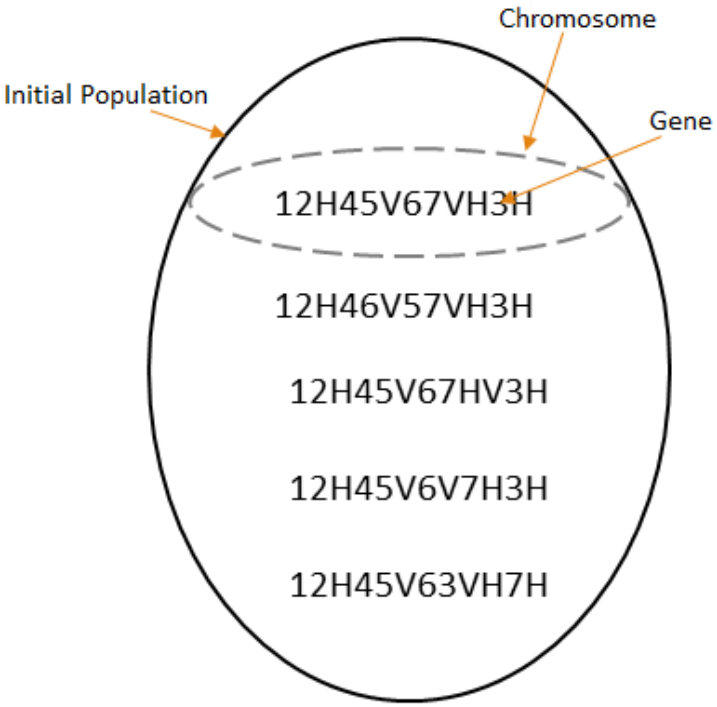


Figure 3.1: Genetic Algorithm Terminology.

plannning). The fitness is what we call the cost function. Parents are the selected individuals from the population.

Some common perturbation functions used to create children from parents are described below.

Crossover: Crossover is the main operator used in reproduction. The child or offspring is generated by inheriting portions of the features of their parents. Crossover is the primary method of optimization in GAs. There are three types of crossover operations [20] [21]:

Order Crossover: In this operation, we randomly select the cut point (the point where the solution is sliced), and copy the left segment of elements from parent 1 into the left segment of the offspring. We fill the rest of the elements in the offspring by scanning through parent 2 and picking out the missing elements from parent 1 in such a way that all the elements are included and none are repeated (Figure 3.2).

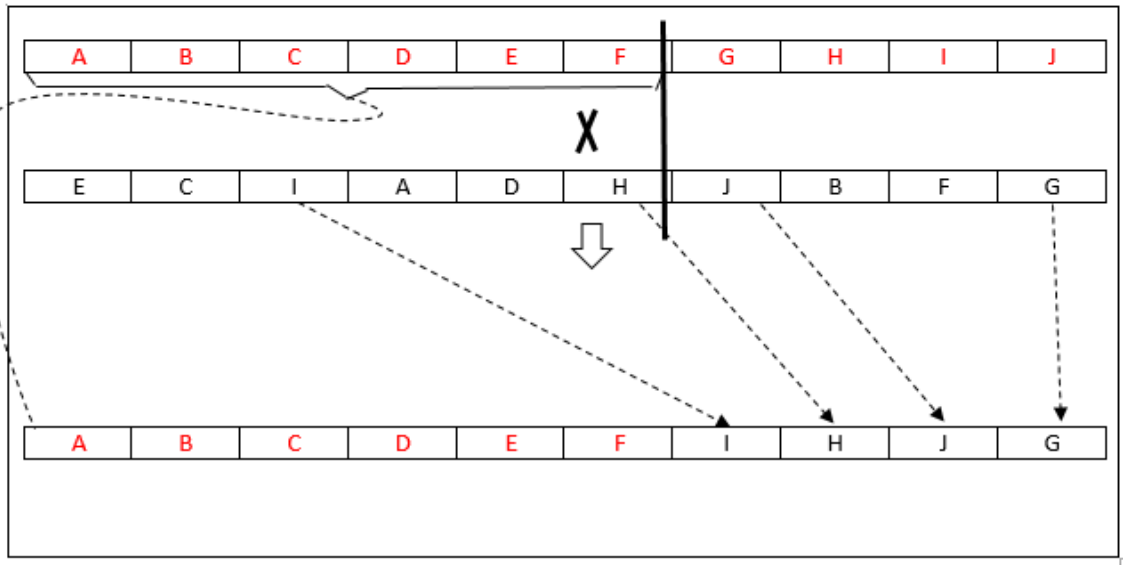


Figure 3.2: Order Crossover Example.

PMX Crossover: The PMX (partially mapped crossover) also chooses the same random cut point for both of the parents. In this technique, the cells beyond the marginal point from parent 2 are taken and inherited by the offspring beyond the same marginal point. Next, the corresponding cells are located from both parents and replaced with the elements

from the first half of the first parent. After replacing all elements from the first parent, the first portion of parent 1 is passed down to the offspring (Figure 3.3).

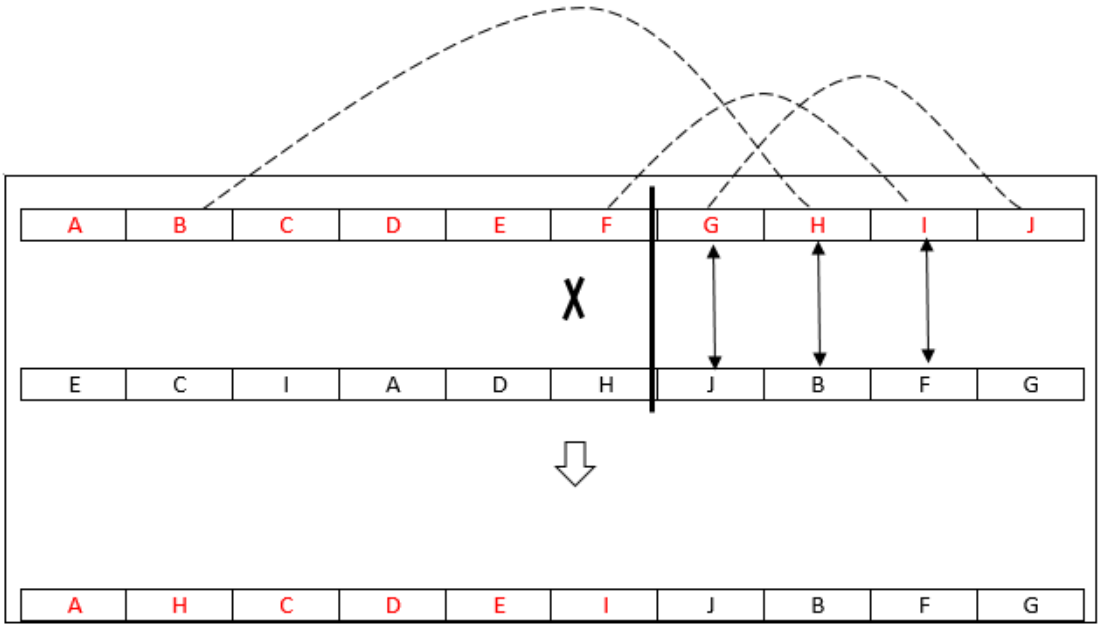


Figure 3.3: PMX Crossover Example.

Cycle crossover: The cycle crossover starts with the first cell location of parent 1, which is passed down to the first cell location of the offspring. The first location of parent 2, then, cannot be inherited into the same location of the offspring. This first cell location of parent 2 is at the x location in parent 1. The x cell location in parent 1 is passed down to the x cell location of the offspring. By doing this, the x location of parent 2 cannot be inherited. The x cell location of parent 2 is found in parent 1 and is passed down to the offspring. The process continues until a cell is found that has already been passed down to the offspring. At that point, another cell is chosen from parent 2 to be passed down to the offspring, and the process continues through another cycle. This process of performing cycle chain operation is called cycle crossover (Figure 3.4).

Mutation: Mutation is the process of changing a chromosome with the smallest prob-

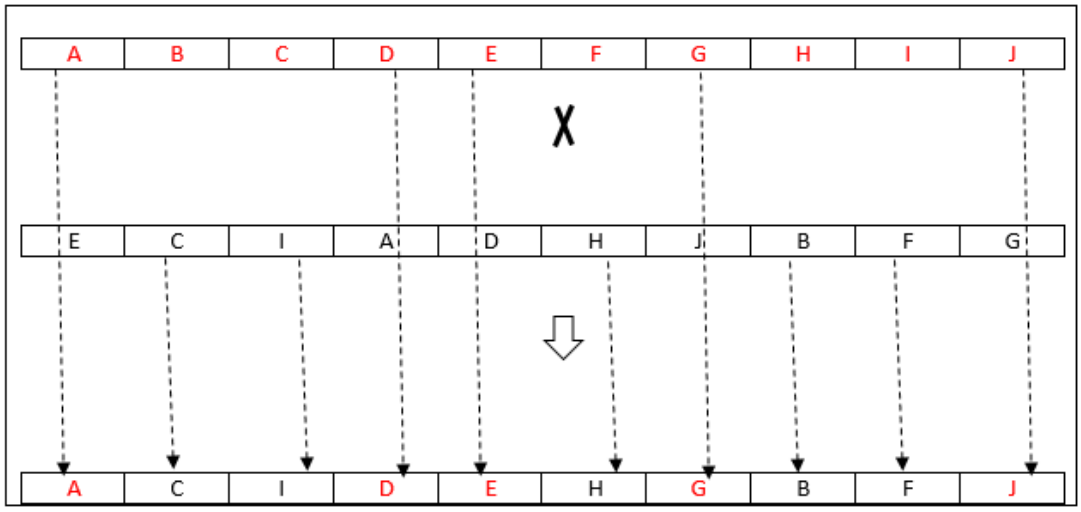


Figure 3.4: Cyle Crossover Example.

ability. Mutation brings out new features within the individuals.

Inversion: Whereas in mutation there are incremental changes within the chromosome, inversion changes the chromosome with a much higher probability.

3.3 Genetic Algorithm

GAs work by evolving generations. Each individual is assigned with a fitness (cost function) value. New generations are formed by using crossovers, mutations, and inversions. Selection is based on keeping individuals that have the highest fitness levels. Over time, the fitness of the population should keep improving. An example of a GA is shown in Figure 3.5

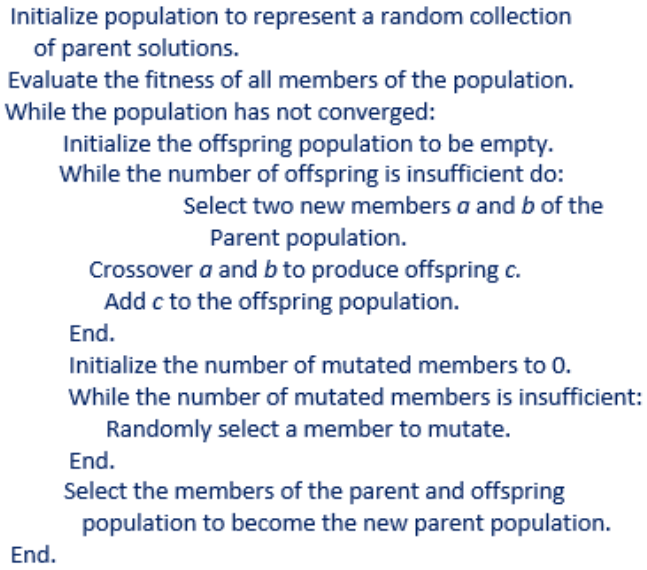


Figure 3.5: Genetic Algorithm [20].

RESULTS

4.1 Application

In this thesis, perturbation functions for polish expressions were modified and applied to make use of GAs for floorplanning ICs. We have compared our results to those produced by a common commercially available EDA tool.

One of the major challenges in improving EDA tools is understanding the tool itself. Every tool has its own format, but the generic design flow is the same among different tools from different vendors. We worked on developing a floorplan stage for the physical design flow of ASICs. The standard procedure in physical design flow has already been discussed (Figure 1.4).

Our research starts with loading a low-level (gate-level) netlist into a commercial SOA tool; this step is called design import [25] - [27]. Along with the netlist, cell libraries such as timing and physical libraries are loaded. These technology libraries are designed using the format called liberty format [28]. It is the most widely used semiconductor library format, used by over 60 EDA vendors [29]. Commercial tools use technology libraries in the format of $.lib$ [28] and $.lef$ [30]. The library format file $.lib$ contains the timing information. The layout extraction format file $.lef$ contains the physical layout information.

After the design import, floorplanning is the next step in the physical design flow. The floorplan step includes the placing of macros on a die. Hard macros are rigid and

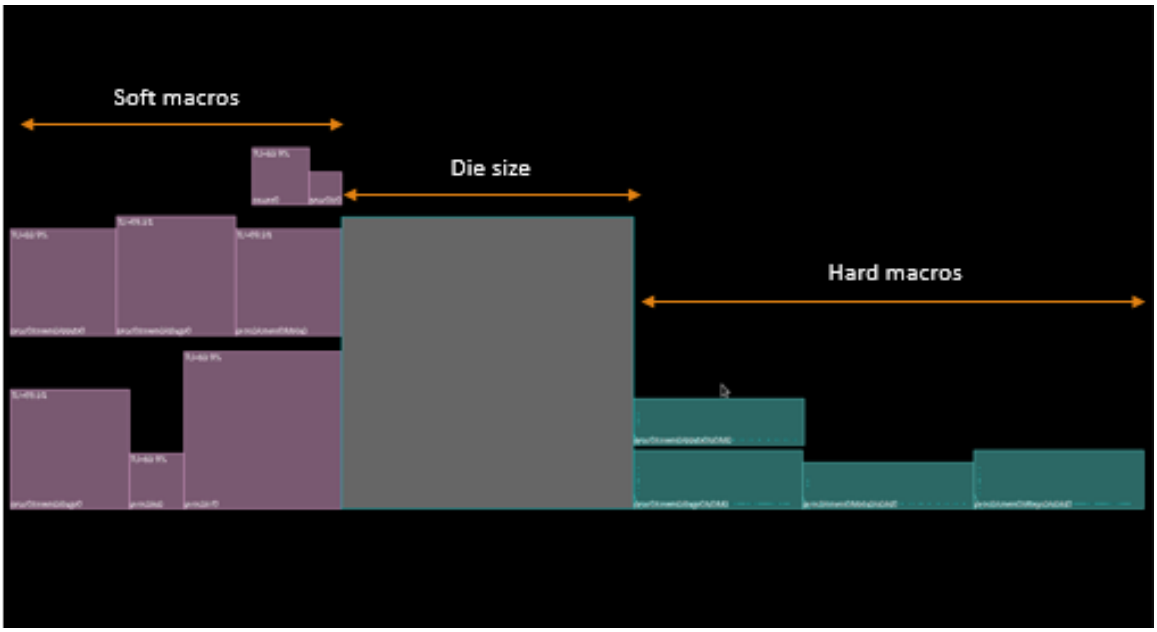


Figure 4.1: Netlist [31] Loaded into Commercial SOA Tool.

inflexible; soft macros are flexible. Most EDA tools have an option to place macros automatically. This is called automated floorplanning [26]. Once the macros are placed during the floorplanning stage, their relative locations for placement are fixed.

For our work, we took the floorplanning file from the SOA automated tool after the design import (Figure 4.2), and we floorplanned the circuit using our GA based approach. Then, we read our floorplanned file into the SOA CAD tool and completed the design to layout by using the SOA tool [25] [26].

For our floorplanner, we used the exact same information that the SOA EDA tool uses for its input file except for the height and width of the hard macros. The height and width of the hard macros are only available in the $.lef$ file. For the genetic algorithm, this information was provided in a table instead of reading in and parsing the whole $.lef$ file.

After parsing the file, each block was assigned a name and its properties, such as height, width, rotation, area, lower left coordinates and upper right coordinates (if available). This information was stored using C++ data structures.

A slicing tree was generated by performing horizontal and vertical cuts. This tree was

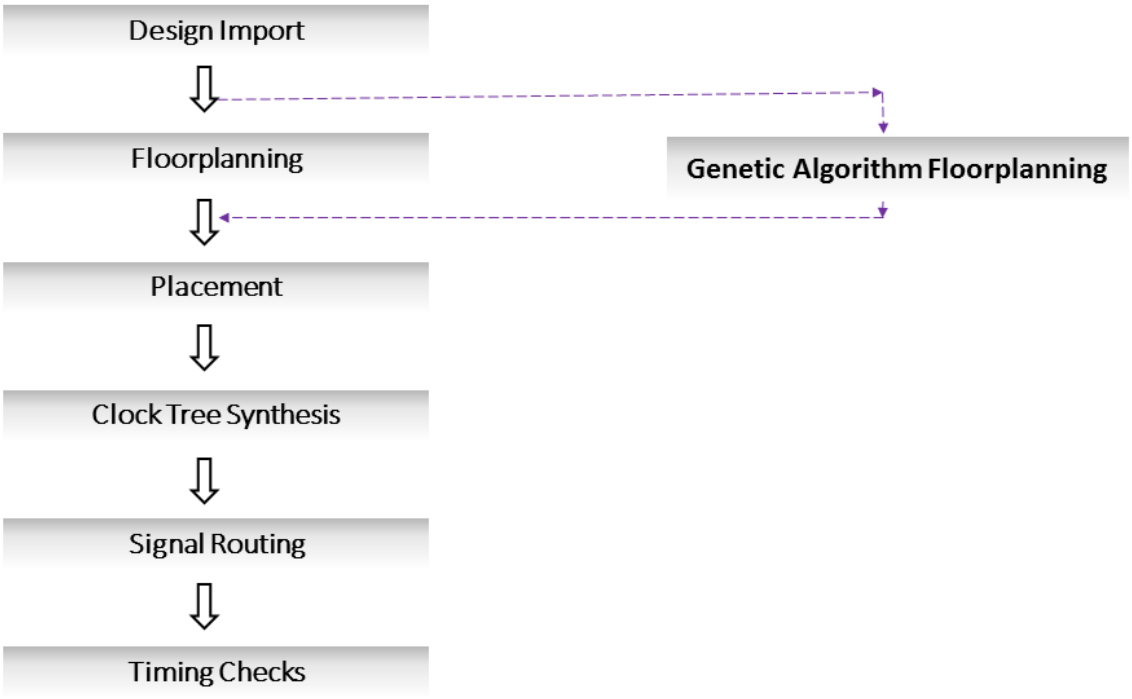


Figure 4.2: GA Floorplanner used in the Commercial SOA Tool Flow.

converted into a normalized polish expression.

4.2 Functions

Below are the functions we used for our GA approach to floorplanning. Wherever possible, we described the adaptions made in order to modify the expressions so that GA could be applied. In addition, we described our own perturbation functions used to improve the technique.

Initial Population: The initial population was the set of randomly generated polish expressions from the generated root polish expression.

Post Order Traversal: This function took all the block information and performed horizontal and vertical cuts to generate an initial polish expression.

Polish Area: A given polish expression was fed as input into this function, and the function gave the area of that polish expression. This formed the fitness indicator for the

floorplanner.

Op1: Swapped two adjacent operands, which were blocks.

Op2: Inverted some part of the chain within the polish expression from V to H and H to V.

Op3: Swapped the adjacent operand and operator without violating the balloting property.

Crossover: We adapted the PMX crossover operation (Figure 3.3) to perform the crossover for the genetic algorithm. Because both the parent polish expressions do not have the operators in the same place, it was difficult to adapt the GA crossover function. The strings of the operands and operators could not be directly swapped without violating the balloting property. Therefore, we separated the chains from the blocks of both of the parents and performed a PMX crossover on the blocks. The chains for the resulting offspring were applied by choosing randomly from the parent 1 or parent 2 chains.

An illustration of the above crossover function was: Parent1: 12H46V57VH3H; Parent2: 14H356V2VH7H; (Figure 4.3).

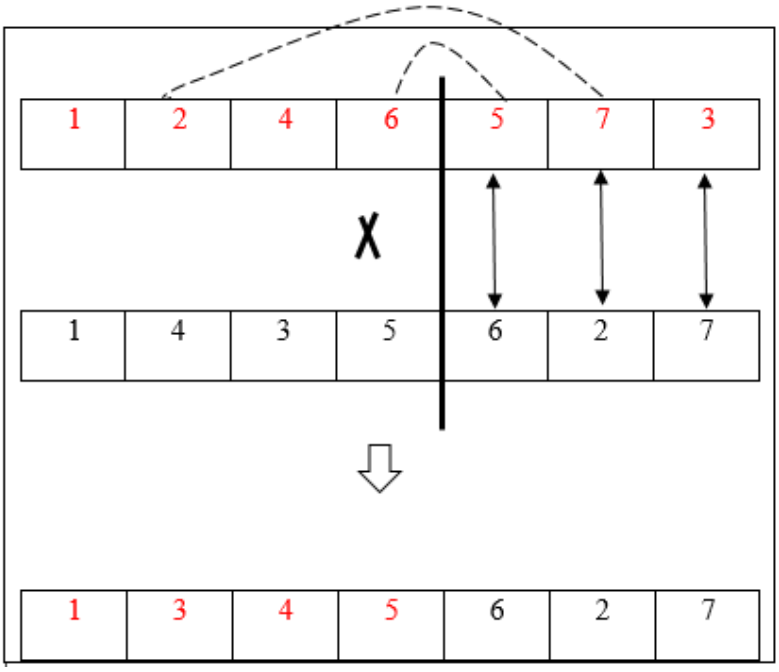


Figure 4.3: Performing PMX Crossover Operation over Polish Expressions.

The set of chains from parent1 was applied to the offspring. Therefore, the final offspring polish expression was 13H45V62VH7H.

Mutation: This referred to incremental changes in the polish expression. This was achieved by applying Op1, Op2, and Op3 operations.

Inversion: This was slightly different from the mutation operation. In mutation, we made only adjacent operand or operator changes, but by using inversion, we made use of non-adjacent operators or operands.

Rotation: This function performed the rotation of some random blocks to change the orientation.

Genetic Algorithm: The GA started with the initial population. Offspring was generated by randomly selecting two of the parents from the initial population and performing the crossover. Mutation, inversion, and rotation were performed on this offspring. Within this generation, some of the best fitness chromosomes were preserved for producing the next generation. For every new generation, the fittest chromosome was verified over the stopping criteria. If the stopping criteria was met by any of the fittest chromosomes, the algorithm was stopped. The stopping criteria for this thesis was based on the 5/4 bin-packing concept [31], with area being the cost function or fitness function.

If the area did not meet the margin within the allotted time, we used the best area of the polish expression found. Once the algorithm finished, the coordinates for blocks were assigned to the final polish expression.

4.3 Results

A total of six netlists were used for comparing the area of macros on the die using the commercial SOA tool to our GA based floorplanner. All of these netlists were synthesized gate level circuits. The $leon.v$ and $asic\_entity.v$ were taken from Cadence Rapid Adoption Kits [16]. The rest of the gate level netlists $FFT32.v$, $FFT64.v$, $FFT128.v$, and

$FFT256.v$ were generated from our own synthesizable HDL netlists.

Initially, all these netlists were loaded onto the commercial SOA tool (which was the design import step) along with timing libraries and physical libraries. Once the netlists were loaded, floorplanning was performed using the automatic floorplanner [25] - [27].

Then, the floorplan file from the commercial SOA was taken out and fed as a command line argument to our GA. Our GA was implemented using the C++ programming language in a 3.20GHz 3.6G memory Linux environment. After the application of our GA, its modified floorplan file was fed back into the commercial SOA tool for comparison purposes.

The table below shows the IC core areas for every netlist, the total area of the floorplans found by the commercial SOA tool and our GA tool, and the percent difference of those two values:

Table 4.1: Area Comparison of Different Netlists

Netlist (IC core area um2)	**Number of Macros**	**Commercial SOA Automated Floorplan Macro area um2**	**Genetic Algorithm Floorplan Macro area um2**	**Percentage Difference**
asic_entity.v (29992550)	29	29992550	29599846.5	1.30 %
FFT256.v (986236.79)	12	396224.136	502203.24	-10.764 %
FFT128.v (717895.51)	11	263266.978	203113.786	8.38 %
FFT64.v (646588.42)	8	442842.422	489456	-7.13 %
FFT32.v (481683.5)	5	432669.082	350629.171	17.032 %
leon.v (453649.64)	4	429557.48	231549.615	43.64 %

The floorplan macro area (columns 3 and 4 of Table 4.1) is the total area taken up by the macros on the IC core. This is not equal to the die area nor the core area. It is just the area covered by the macros in each core. Every netlist consists of macros and glue logic

Table 4.2: Time Comparison of Different Netlists

Netlist	**Commercial SOA Floorplan Time (Sec)**	**Genetic Algorithm Floorplan Time (Sec)**
asic_entity.v	108	50
FFT256.v	90	70
FFT128.v	41.7	35
FFT64.v	8	15
FFT32.v	2	5
leon.v	32	17

(unplaced standard cells). The size of the macro vary from macro to macro and netlist to netlist. The commercial tool was run using the highest available effort in placing the macros.

Percentage difference (column 5 of Table 4.1) is the difference between percentage of GA macro area with respective to core area ($\%GA\ macro\ w.r.t\ to\ core$) to percentage of commercial SOA macro area wth respect to core area ($\%SOA\ macro\ w.r.t\ to\ core$).

$$\%GA\ macro\ w.r.t\ to\ core\ = \frac{\text{IC core area} - \text{GA macro area}}{\text{IC core area}} \times 100 \tag{4.1}$$

$$\%SOA\ macro\ w.r.t\ to\ core\ = \frac{\text{IC core area} - \text{SOA macro area}}{\text{IC core area}} \times 100 \tag{4.2}$$

Relative to the overall area, a positive percent difference is the improvement shown using the GA floorplanner. A negative percent difference shows that the commercial SOA tool did a better job of placing the macros than the GA floorplanner.

$asic\_entity.v$ is the largest netlist with 29 macros. Even though the improvement made by the GA floorplanner was just 1.3%, the run time was cut in half (Table 4.2). $FFT256.v$ is the next biggest netlist with 12 macros. For this netlist, the commercial SOA tool did a slightly better job placing the macros relative to the area, but took longer than the

GA to reach that result. One of our future goals is to improve this placement with better stopping criteria. For $FFT32.v$, the commercial SOA tool took just 2 seconds to place the macros, but the area of this placement was large. The GA tool took longer (5 seconds), but it improved the area by 17.032 %.

As we mentioned earlier, not every macro is the same size. $Leon.v$ has larger macros than other FFT macros. Figure 1.5 shows the placement of these macros using the commercial SOA tool, and Figure 1.6 shows the placement of the same macros using the GA tool. Interestingly, the GA placed the macros much more effectively in terms of area.

Table 4.3: PDA Comparison

Netlist	**Commercial SOA tool Power * Delay * Area (mW*nsec*um2)**	**Genetic Algorithm Power * Delay * Area (mW*nsec*um2)**
leon.v	2.899E9	2.756E9
asic_entity.v	-	-
FFT256.v	4.5788E10	4.5753E10
FFT128.v	2.034E10	2.026E10
FFT64.v	1.07E10	0.575E10
FFT32.v	1.243E9	1.4E9

Table 4.3 shows the PDA (power * delay * area) comparison of the commercial SOA tool to GA floorplanner insert. As from Table 4.1, we know that GA performed better floorplanning interms of area for $leon.v$, but the overall PDA does not seemed to be huge difference. If we would have considered interconnect delays as a part of cost function, we would have improved this to a much better result. The netlist $asic\_entity.v$ was used only for floorplan demonstration purposes. For $FFT64.v$, commercial SOA tool has better macro area than GA, but, the overall PDA was better with GA floorplanner. $FFT32.v$ has 17.032 % of improvement in macro area using GA floorplanner, but, the commercial SOA tool has better overall PDA. As we mentioned earlier we hope to fix the overall PDA for all the netlist except $asic\_entity.v$ using GA by considering interconnect delays.

FUTURE WORK

5.1 Modified Cost Function

The goal of the current GA floorplanner is to minimize area. We plan to include some measure of the circuit interconnect to drive our floorplanning cost function. For example, the area measurement can be defined as A, and the total interconnect measurement can be denoted as W. A modified cost function Ψ that includes both parameters could be:

$$\Psi = \mathrm{A} + \lambda_w \mathrm{W} \tag{5.1}$$

Area (A) and interconnections (W) are controlled by λ_w [18].

5.2 Stopping Criteria

Determining when to stop a probabilistic algorithm is a difficult problem. In future work, we would like to develop a more sophisticated stopping criteria and also we hope to provide user options such as a fixed time (or number of iterations) to execute the GA or a target percentage of the absolute minimum area with interconnect considerations.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.," in *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 5, pp. 33-35, Sept. 2006.

[2] K. S. Yeo, K. T. Ng, Z. H. Kong, and T. B. Dang, *Intellectual property for integrated circuits.* Baltimore, MD, United States: J Ross Publishing, 2010.

[3] J. S. Kilby, "Invention of the integrated circuit," in *IEEE Transactions on Electron Devices*, vol. 23, no. 7, pp. 648-654, Jul 1976.

[4] I. M. Ross, "The invention of the transistor," in *Proceedings of the IEEE*, vol. 86, no. 1, pp. 7-28, Jan 1998.

[5] N. A. Sherwani, *Algorithms for VLSI physical design automation.* 3^{rd} ed. Boston: Kluwer Academic Publishers, 2002.

[6] G. E. Moore, "Microprocessors and integrated electronic technology," in *Proceedings of the IEEE*, vol. 64, no. 6, pp. 837-841, June 1976.

[7] S. M. Sait and H. Youssef, *VLSI physical design automation: Theory and practice. Singapore.* Singapore: World Scientific Publishing Co Pte, 1999.

[8] E. S. Kuh and T. Ohtsuki, "Recent advances in VLSI layout," in *Proceedings of the IEEE*, vol. 78, no. 2, pp. 237-263, Feb 1990.

[9] Intel Corporation, "How Intel makes chips: Transistors to Transformations," Intel. [online]. Available: http://www.intel.com/content/www/us/en/history/museum-transistors-to-transformations-brochure.html. Accessed: Dec. 5, 2016.

[10] R. Saleh *et al.*, "System-on-Chip: Reuse and integration," in *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050-1069, June 2006.

[11] M. J. S. Smith, *Application-specific integrated circuits.* Reading, MA: Addison-Wesley Educational Publishers, 1997.

[12] H. Mair and Liming Xiu, "An ASIC design flow of deep submicron suceeds on first pass," *1998 5th International Conference on Solid-State and Integrated Circuit Technology. Proceedings (Cat. No.98EX105)*, Beijing, 1998, pp. 352-355.

[13] C. J. Alpert, S. S. Sapatnekar, and D. P. Mehta, Eds., *Handbook of algorithms for physical design automation.* Boca Raton: Auerbach Publishers, 2008.

[14] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: From graph partitioning to timing closure.* Dordrecht: Springer.

[15] K. Golshan, *Physical design essentials: An ASIC design implementation perspective.* New York: Springer-Verlag New York.

[16] Cadence, "Digital Implementation and Sign-off Flow," in *Rapid Adoption Kits.* [Online]. Available: http://support.cadence.com/. Accessed: 2016.

[17] L. -T. Wang, Y. -W. Chang, and K. -T. Cheng, Eds., *Electronic design automation: Synthesis, verification, and test.* Amsterdam: Morgan Kaufmann Publishers, 2009.

[18] D. F. Wong and C. L. Liu, "A New Algorithm for Floorplan Design," in *23rd ACM/IEEE Design Automatin Conference*, 1986, pp. 101-107.

[19] R. H. J. M. Otten, "Automatic Floorplan Design," in *19th Design Automation Conference*, Las Vegas, NV, USA, 1982, pp. 261-267.

[20] P. Mazumder and E. M. Rudnick, *Genetic Algorithms for VLSI design, layout and test automation.* Prentice Hall PTR, 1999.

[21] A. Ghosh and S. Tsutsui, *Advances in evolutionary computing: theory and applications.* Berlin: Springer, 2003, pp.683-712.

[22] S. Nakaya, T. Koide and S. Wakabayashi, "An adaptive genetic algorithm for VLSI floorplanning based on sequence-pair," *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No. 00CH36353)*, Geneva, 2000, pp. 65-68 vol.3.

[23] J. P. Cohoon and W. D. Paris, "Genetic Placement," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 956-964, November 1987.

[24] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. S. Richards, "Distributed genetic algorithms for the floorplan design problems," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 483-492, Apr 1991.

[25] Cadence. Encounter Digital Implementation System User Guide, Product Version 9.1.3, October 2010.

[26] Cadence. EDI System Menu Reference, Product Version 13.21, October 2013.

[27] Cadence. EDI System Text Command Reference, Product Version 13.21, October 2013.

[28] Synopsys. Liberty User Guides and Reference Manual Suite, Product Version 2015.12.

[29] "OpenSource liberty," 2016. [Online]. Available: https://www.opensourceliberty.org/opensourceliberty.html. Accessed: 2016.

[30] Cadence. LEF/DEF Language Reference, Product Version 5.7, November 2009.

[31] J. Békési, G. Galambos, and H. Kellerer, "A 5/4 linear time bin packing algorithm," *Journal of Computer and System Sciences*, vol. 60, no. 1, pp. 145-160, Feb. 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000099916677. Accessed: 2016.