PLACER TOOL FOR RECONFIGURABLE LOGIC BLOCKS ON eFPGA

BY

NILAY CHANDRAKANT PARMAR

(07MCE013)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

MAY 2009

PLACER TOOL FOR RECONFIGURABLE LOGIC BLOCKS ON eFPGA

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Nilay Chandrakant Parmar

(07MCE013)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2009

Certificate

This is to certify that the Major Project entitled "Placer Tool For Reconfigurable Logic Blocks on eFPGA" submitted by Nilay Chandrakant Parmar (07MCE013), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. S.N. Pradhan
P.G. Coordinator, Guide,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

To Whom it may Concern

Certified that the above statement made by the student is correct to the best of our knowledge and belief.

Mrs. Jyoti Malhotra
Specialist

Mr. Himanshu Srivastava
Senior Software Engineer

Approved as to style and content by:

I certify that I have read this dissertation and that in my opinion it is fully adequate, in cope and quality, as a dissertation for the degree of Master Of Technology in Computer Science and Engineering.

Mrs. Namerita Khanna
Section Manager
NVM-BCD e-Configurable Logic

Abstract

The Training Semester from 15th September 2008 has given me an opportunity to work with one of the best semiconductor companies in the world, STMicroelectronics.

PiCoGA is STMicroelectronics Specific FPGA Chip which is an acronym for Pipelined Configurable Gate Array. The PiCoGA is designed to implement a peculiar pipeline where each stage corresponds to a piece of computation, so that high throughput circuits can be mapped. In this way a sequence of PiCoGA instructions can be processed filling the pipeline in order to exploit parallelism.

Along with this the configurable unit also preserved its state across instruction executions. A new PiCoGA instruction may directly use the results of previous ones, thus reducing the pressure on the register file. Moreover a tight integration in the processor core gives the opportunity to use the PiCoGA in many different computational cores. With the arrival of PiCoGA, the problem of multi-computing was solved to achieve a much faster computation.

The main goal of this Dissertation is to Design a CAD(Computer Aided Design) tool which perform the placement of Reconfigurable Logic Blocks on eFPGA, address the challenges occurring because of the different constraints due to the architecture of chip.

Acknowledgements

"Outstanding achievement is not possible in vacuums. It needs lot of help and assistance besides a healthy environment, luckily I have."

First and foremost I would like to specially thank **Dr. S.N. Pradhan**, M.Tech section head, Nirma University, Ahmedabad for providing me with an opportunity to take up this training and for their constant support and encouragement.

I would like to give my special thank to **Prof. D.J Patel**, Head, Computer Science & Engineering Department, Nirma University, Ahmedabad for his encouragement and motivation throughout the Major Project. I am also thankful to **Dr. Ketan Kotecha**, Director, Institute of Technology, Nirma University, Ahmedabad for his kind support in all respect during my study.

I am immensely grateful to **Mrs. Namerita Khannar** (Section Manager ECL) for providing me the opportunity to work on this project. Without her this project work could not have seen the daylight.

I would like to express my hearty thanks and indebtedness to my guide **Mrs. Jyoti Malhotra** & **Mr. Himanshu Srivastava** for their enormous help and encouragement throughout the course of this thesis, who happens to be my role model, has always given me a real example of how a researcher should be, proving 'Vidya Dadati Vinayam'. They give me an opportunity to do my thesis work and provide all resources required for my project work.

- Nilay Chandrakant Parmar

(07MCE013)

Contents

List of Figures

List of Tables

Abbreviations

ASIC	Application Specific Integrated Circuit
BB	Bounding Box
CAD	Computer Aided Design
CPS	Coarse Placement Solution
eFPGA	embedded Field Programmable Gate Array
FF	Flip Flop
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IC	Integrated Circuit
I/O	Input Output
KL	Kernighan Lin
LUT	Look Up Table
MCNC	Microelectronics Center of North Carolina
MPGA	Mask Programmable Gate Array
NRE	Non Recurring Engineering
OPF	Output Connected to F Pin
PiCoGA	Pipelined Configurable Gate Array
Pip	Placement Input Parameter
Reg	Register
RLC	Resistance Inductance Capacitance
SA	Simulated Annealing
SOC	System On Chip
SRAM	Static Random Access Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VPR	Versatile Place & Route

Chapter 1

Introduction

In this chapter, motivation behind the work, scope of work and thesis outline been presented.

1.1 General

All FPGAs consist of a large number of programmable logic blocks, which can each implement a small amount of digital logic, and programmable routing which allows the logic block inputs and outputs to be connected to form larger circuits. In this thesis three different issues investigated in FPGA architecture: two concern FPGA routing design, and one concerns FPGA logic block design.

Along with this the configurable unit also preserved its state across instruction executions. A new PiCoGA instruction may directly use the results of previous ones, thus reducing the pressure on the register file. Moreover a tight integration in the processor core gives the opportunity to use the PiCoGA in many different computational cores. With the arrival of PiCoGA, the problem of multi-computing was solved to achieve a much faster computation.

For PiCoGA architectures, one needs Computer Aided Design (CAD) tools capable of automatically implementing circuits. This thesis therefore has two major foci: the study of PiCoGA architectural issues, and the development of a CAD infrastruc-

ture for the PiCoGA chip.

The main goal of this Dissertation is to Design a CAD(Computer Aided Design) tool which perform the placement of Reconfigurable Logic Blocks on eFPGA, address the challenges occurring because of the different constraints due to the architecture of chip.

1.2 Motivation

In the thirteen years since their introduction, Field-Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits, growing into a $2 billion per year industry. The key to FPGAs' popularity is their programmability – an FPGA can implement any circuit simply by being appropriately programmed. Other circuit implementation options, such as Standard Cells or Mask-Programmed Gate Arrays (MPGAs), require that a different VLSI chip be fabricated for each design. Use of a standard FPGA, rather than a custom MPGA, has two key benefits: lower non-recurring engineering (NRE) costs, and faster time-to market.

To implement a circuit in an MPGA, one sends the completed design to a silicon foundry which manufactures a chip to implement exactly (and only) that design. The non-recurring engineering (NRE) fees to have the first chip manufactured are typically $100 000 to $250 000; these fees cover the cost of making lithography masks for the circuit and of running a new design through the fabrication plant. A design is implemented in an FPGA, however, simply by programming a standard FPGA to have the desired functionality, so there are no NRE costs. This makes FPGAs the lowest cost implementation medium for small and medium volume designs.

Time-to-market is the other key advantage of FPGAs. MPGA designs typically take 6 - 8 weeks to fabricate. If bugs are found in the finished chip it must be thrown away, and one must wait another 6 - 8 weeks to fabricate a corrected design. FPGAs, on the other hand, can be programmed in seconds, and any bugs found once the chip

is tested in system can be corrected in minutes by reprogramming the FPGA. With today's short product cycles, the time-to-market advantage this provides is often compelling.

FPGA programmability carries a price, however. In MPGAs and Standard Cells circuitry is interconnected with metal wires. FPGAs, in contrast, must connect circuitry via programmable switches. These switches have higher resistance than metal wires and add significant capacitance to connections, reducing circuit speed. As well, the switches take up more area than metal wires would, so an FPGA must be considerably larger than an MPGA to implement the same circuit. Typically a circuit implemented in an FPGA is about 10 times larger and 3 times slower than the same circuit implemented via an MPGA in an equivalent process. The larger size of FPGA circuitry makes FPGA implementations more expensive than MPGAs for high volume designs, and the limited speed of FPGAs precludes their use in very high-speed designs. Consequently, there is a compelling motivation to research new FPGA architectures which reduce these speed and density penalties as much as possible. As well, the FPGA marketplace is highly competitive, so each FPGA manufacturer is constantly searching for better FPGA architectures in order to gain a speed and density advantage over its competitors.

In order to investigate the quality of different FPGA architectures, one needs Computer Aided Design (CAD) tools capable of automatically implementing circuits in each FPGA architecture of interest. This thesis therefore has two major foci: the study of several FPGA architectural issues, and the development of a highly "flexible" CAD infrastructure that enables these investigations of different FPGA architectures.

1.3 STMicroelectronics

1.3.1 Introduction

ST Microelectronics is a global independent semiconductor company and is a leader in developing and delivering semiconductor solutions across the spectrum of microelectronics applications. An unrivalled combination of silicon and system expertise, manufacturing strength, Intellectual Property (IP) portfolio and strategic partners positions the Company at the forefront of System-on-Chip (SoC) technology and its products play a key role in enabling today's convergence trends. ST is one of the world's largest semiconductor companies.

ST is one of the world's largest semiconductor companies. In 2004, ST's net revenues were US$8,760 million and net earnings were US$601 million. According to the most recent data from independent sources, ST is the world's leading supplier of application-specific analog ICs overall with number one rankings in various segments within this field, including wireless ASICs, computer peripheral ASICs and automotive ASSPs. ST is also the leader in MPEG-2 decoder ICs, and ASICs/ASSPs overall, including a number one position in digital consumer ASSPs. Additionally, ST is ranked at number two for discrete products, and in the memory market, ST is ranked third in NOR Flash ICs. In application segments overall: ST is number one for ICs in set-top boxes; at number two in smart cards, at number three in automotive; and at number four in wireless.

The Company's products are manufactured and designed using a broad range of fabrication processes and proprietary design methods. To complement this depth and diversity of process and design technology, the Company also possesses a broad intellectual property portfolio that it has used to enter into cross-licensing agreements with many other leading semiconductor manufacturers.

The Company currently offers over 3,000 main types of products to more than 1,500 customers, Alcatel, Bosch, DaimlerChrysler, Ford, Hewlett-Packard, IBM, Motorola, Nokia, Nortel Networks, Philips, Seagate Technology, Siemens, Sony, Thomson

and Western Digital. Approximately two-thirds of ST's revenue is derived from differentiated products, a combination of dedicated, semi-custom and programmable products designed to suit a specific customer or a specific application and therefore having high system content. This result reflects ST's exceptionally early recognition of the importance of system-on-chip technology, which is key for addressing the fast growing market f or convergence products, and the success of the strategies it developed to ensure its leading position in this key emerging field.

1.3.2 Why ST?

Since its formation, the Company has significantly broadened and upgraded its range of products and technologies and has strengthened its manufacturing and distribution capabilities in Europe, North America, and the Asia Pacific region. This capacity expansion is an ongoing process with the upgrading of existing facilities and the creation of new 8-inch, sub-micron fabs around the world. ST currently has five 8-inch fabs in operation in: Rousset (France); Agrate Brianza, R2 (Italy); Crolles (France); Phoenix (Arizona); Catania (Italy); and Singapore. Furthermore, a new 12-inch manufacturing facility is currently under construction in Catania; and the company is now ramping up production from a 12-inch pilot line called Crolles2, in partnership with Philips and Freescale Semiconductor. The Crolles2 operation is also host to the joint development program between the three companies to develop leading-edge CMOS process technology down to the 32nm node, in conjunction with TSMC for process alignment.

The group totals close to 50,000 employees, 16 advanced research and development units, 39 design and application centres, 16 main manufacturing sites and 78 sales offices in 36 countries.

Corporate Headquarters, as well as the headquarters for Europe and for Emerging Markets, are in Geneva. The Company's U.S. Headquarters are in Carrollton (Dallas, Texas); those for Asia/Pacific are based in Singapore; and Japanese operations are

headquartered in Tokyo.

1.3.3 Area Of Products

- Analog and mixed signal IC's
- Memories
- Microcontrollers
- Power management IC's
- Transistors
- ASSP for home video
- ASSP for mobile systems
- Amplifiers & Linear
- Diodes
- EMI Filtering & Conditioning
- Logic, Signal Switch
- Protection Devices
- Sensors
- Smartcard ICs
- And many more

1.4 Scope of Work

The primary goal in placement process is to place the heavily connected blocks together for

- Decreased wire length.
- Increased Circuit speed.
- Optimize area usage.

The main objective of this thesis is to modify the existing annealing algorithms to utilize the PiCoGA architecture and to claim the performance improvement. But placement of logic blocks on eFPGA has many constraints due to the architecture of chip. Need to implement an algorithm in such a way that one can have a minimum wires and minimum size of a chip. There should be no long wires in a chip. Next challenging thing is to design an algorithm in such a way that it is independent on architecture on a chip. So, algorithm should not depend on the number of rows and number of columns of a chip.

The scope of this thesis work encompasses the architectural study and programming methodology of the PiCoGA architecture. Based on this knowledge two placement algorithms are selected: Simulated Annealing and Versatile Placement and Route (VPR). This VPR is an extended version of Simulated Annealing. The basic pseudo code for both these algorithm is same but there is some changes in VPR line Cost computation, Updation on Temperature, Computation of a swapping region, etc. After completion of a code of placer tool, need to do manual testing. In the whole process of designing a placer tool, very important thing is to understand the SDK used to design the placer tool. How best one can apply this SDK features in designing the placer tool. In addition the user been provided the facility to enter into the algorithm from any point.

1.5 Outline of Thesis

- Chapter 2 presents the details of the Chip architecture like detail of transistor used to design RLC, Routing architecture, different type of connection boxes which are used to connect the IO lines which global horizontal and vertical, Different switch boxes to connect horizontal and vertical lines.

- Chapter 3 explains various placement algorithms. If also describes the comparison between both the algorithms. Which algorithm is better and why choose that algorithm to design this tool.

- Chapter 4 describes the programming methods for Tool. It also describes the whole process of designing the placer tool. How the algorithm is partitioned to ease of design? How the core part of this placer tool which is implementation of SA algorithm is independent of the architecture of chip? How this tool can allow the user to enter from any point?

Chapter 2

PiCoGA Architecture

2.1 FPGA Architecture Issue

All FPGAs consist of a large number of programmable logic blocks, which can each implement a small amount of digital logic, and programmable routing which allows the logic block inputs and outputs to be connected to form larger circuits. There are three different issues in FPGA architecture: two concern FPGA routing design, and one concerns FPGA logic block design.

The first issue investigated is global routing architecture. The global routing architecture of an FPGA specifies the relative width of the various channels within the chip. Figure. 2.1 depicts an example global routing architecture in which the channels near the center of the FPGA are wider than those near the edges [1].

In MPGA and standard cell implementations, a custom chip is created for each design, so routing channels can easily be made wider in areas of a chip where the demand for routing is greater. In FPGAs, however, all routing resources are prefabricated, so the width of all the routing channels is set by the FPGA manufacturer. Then to find the distribution of routing resources, or tracks, to the various channels that permits their efficient utilization by the largest class of circuits. If there are too few tracks in some area of the chip then many circuits will be unroutable, while

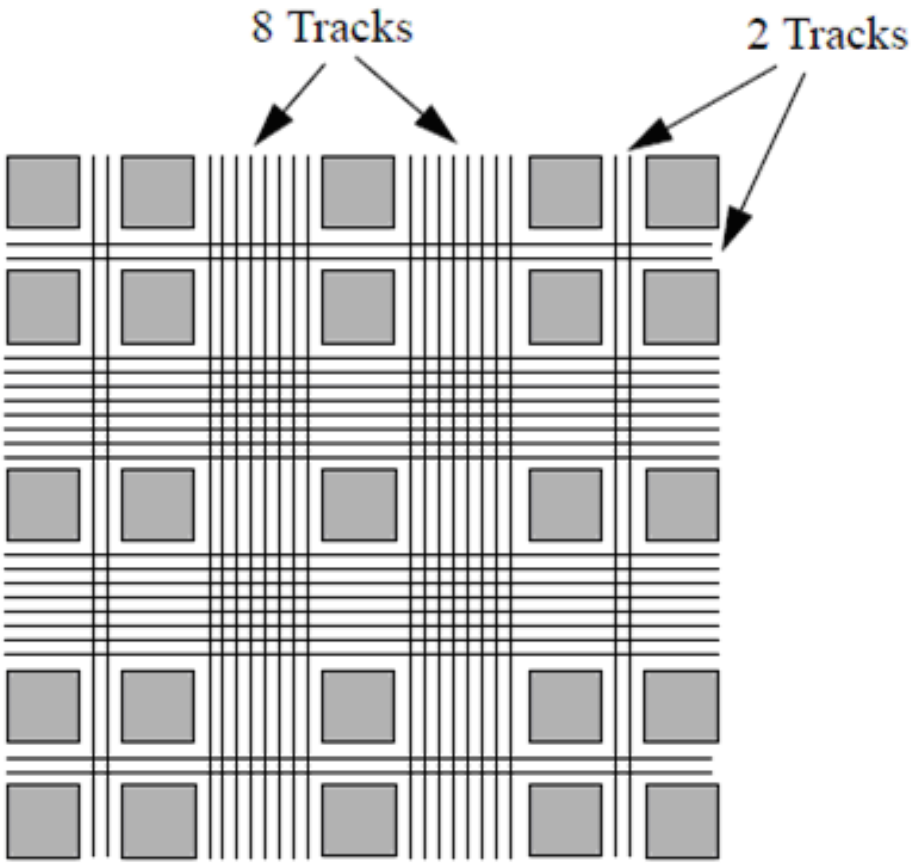

Figure 2.1: Example Global Routing Architecture

if there are too many tracks, they may be wasted. There is no agreement amongst commercial FPGAs on the best global routing architecture, so this question has clear commercial relevance.

The second issue is the use of cluster-based logic blocks in FPGA. These logic blocks are groups, or clusters, of look-up tables (LUTs) and flip flops along with local routing to interconnect the LUTs within a cluster; Figure 2.2 depicts an example logic cluster[1]. In an FPGA using cluster-based logic blocks, many connections will then be made via the local interconnect within a cluster. Since this local interconnect can be made faster than the general purpose interconnect between logic blocks, cluster-based logic blocks can improve FPGA speeds. As well, an FPGA in which every logic

block contains several LUTs will need fewer logic blocks to implement a circuit than an FPGA in which each logic block is a single LUT. These reduce the size of placement and routing problem considerably. Since placement and routing is usually the most time-consuming step in mapping a design to an FPGA, cluster-based logic blocks can significantly reduce design compile time. As FPGAs grow larger, it is important to keep this compile time from growing too large or one of the key advantages of FPGAs, rapid prototyping and quick design turns, will be lost.

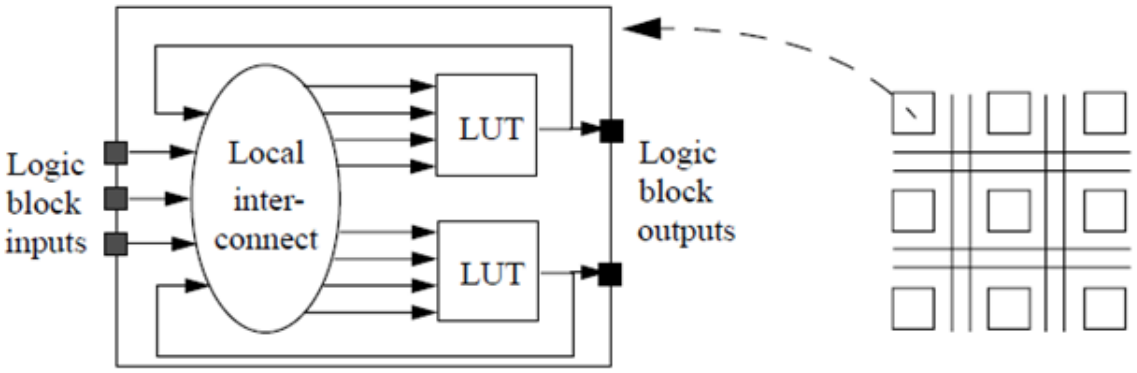

Figure 2.2: Example logic cluster containing two LUTs

The effect of cluster-based logic blocks on FPGA area is complex, and is the focus of our investigation of these logic blocks. On the one hand, grouping related LUTs together into a single logic block reduces the number of connections to be routed between logic blocks, saving routing area. Since the general-purpose interconnect consumes most of the die area in SRAM-based FPGAs, this is a significant area savings. On the other hand, in the logic clusters, the area required by the local routing grows quadratically with the number of LUTs in a cluster. For sufficiently large clusters, the area used by this local interconnect will exceed the area saved in the general interconnect. Then, investigate how the number of LUTs per cluster affects both FPGA area and other important FPGA architecture parameters, such as the proper number of inputs to a logic block and the required flexibility of the general-purpose interconnect. While recent FPGAs from Xilinx, Altera, Lucent Technologies,

Actel and Vantis have all grouped several LUTs together into logic clusters, there has been little published work investigating the impact of the logic cluster used on FPGA area-efficiency.

The final FPGA issue is that of detailed routing architecture. The detailed routing architecture of an FPGA defines how logic block inputs and outputs can be interconnected. The style of detailed routing architecture is the island-style architecture. A simplified detailed-routing architecture is depicted in Figure 2.3.

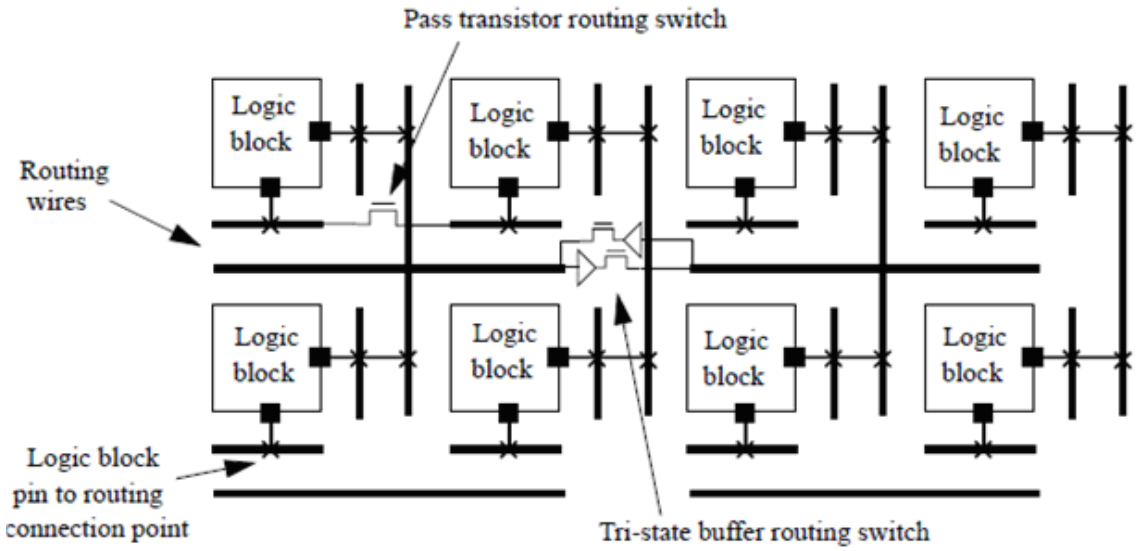

Figure 2.3: Example Detailed Routing Architecture

Detailed routing architecture is the key element of an FPGA because most of an FPGA's area is devoted to routing, and most of a circuit's delay is due to routing delays rather than logic block delays, so creating detailed routing architectures that are both fast and area-efficient is clearly crucial.

2.2 FPGA Architecture

All FPGAs are composed of three fundamental components: logic blocks, I/O blocks and programmable routing, as Figure 2.4 shows. A circuit is implemented in an

FPGA by programming each of the logic blocks to implement a small portion of the logic required by the circuit, and each of the I/O blocks to act as either an input pad or an output pad, as required by the circuit. The programmable routing is configured to make all the necessary connections between logic blocks and from logic blocks to I/O blocks. The following section briefly describes the basic technologies used to make FPGAs programmable.

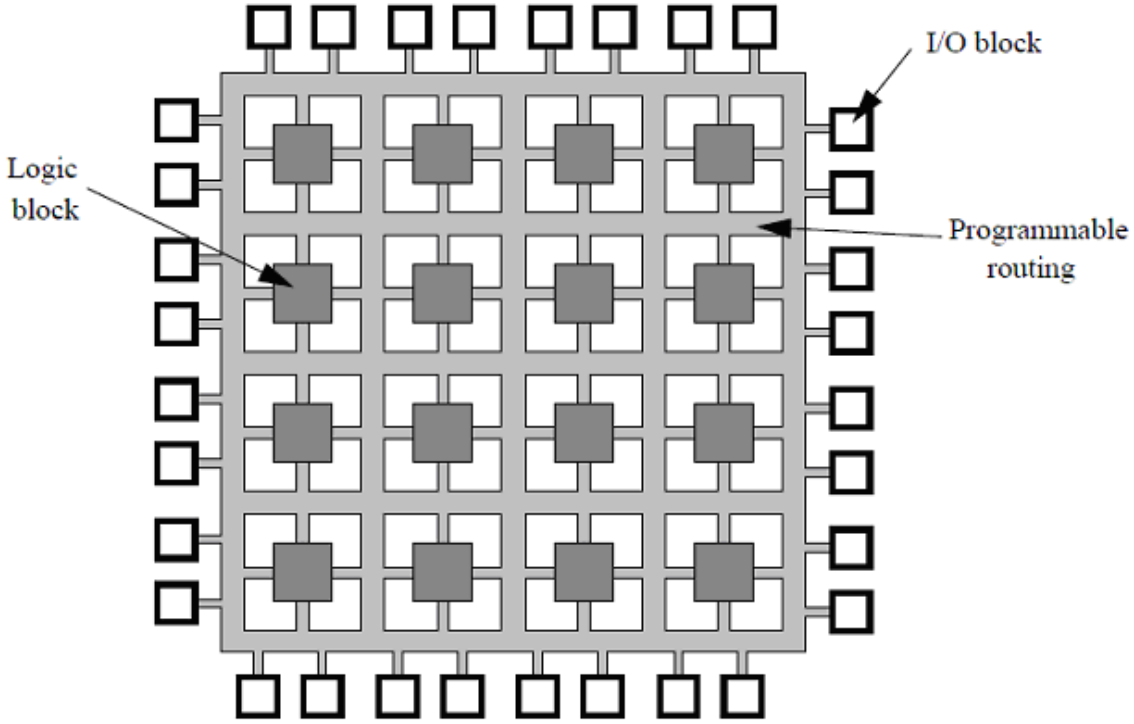

Figure 2.4: Generic FPGA

2.3 FPGA Logic Block Architecture

The logic block used in an FPGA strongly influences the FPGA speed and area-efficiency. While many different logic blocks have been used in FPGAs, most current commercial FPGAs are using logic blocks based on look-up tables (LUTs). Figure 2.5 shows how a 2-input LUT can be implemented in an SRAM-based FPGA. A k-

input LUT requires 2^k SRAM-cells and a 2^k-input multiplexer. A k-input LUT can implement any function of k-inputs; one simply programs the 2^k SRAM cells to be the truth table of the desired function.

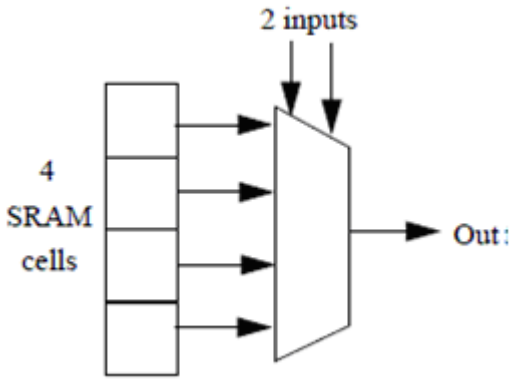

Figure 2.5: A 2-input LUT implemented in an SRAM-based FPGA

Most modern FPGAs are composed not of a single LUT, but of groups of LUTs and registers with some local interconnect between them, and there has been some research into logic blocks containing groups of LUTs. The area model used in this study modeled routing area by estimating the number of routing switches needed after global routing. Since this model did not account for the fact that the routing transistors would most likely have to be sized up for the larger logic blocks, these area numbers are probably somewhat inaccurate (overly optimistic) for logic blocks with more than one LUT.

2.4 FPGA Routing Architecture

Commercial FPGAs can be classified into three groups, based on their routing architecture. The FPGAs of Xilinx, Lucent and Vantis are island-style FPGAs, while Actel's FPGAs are rowbased, and Altera's FPGAs are hierarchical. In this thesis, almost exclusively investigated the island-style routing architecture, so description of this style of routing architecture below.

Figure 2.6 depicts an island-style FPGA. Logic blocks are surrounded by routing channels of pre-fabricated wiring segments on all four sides. A logic block input or output, which called a pin, can connect to some or all of the wiring segments in he channel adjacent to it via a connection block of programmable switches. At every intersection of a horizontal channel and a vertical channel, there is a switch block. This is simply a set of programmable switches that allow some of the wire segments incident to the switch block to be connected to others; note that for clarity only a few of the programmable switches contained by switch boxes are shown in Figure 2.6. By turning on the appropriate switches, short wire segments can be connected together to form longer connections. In the FPGA of Figure 2.6, notice that some wire segments continue unbroken through a switch block. These longer wires span multiple logic blocks, and are a crucial feature in commercial FPGAs.

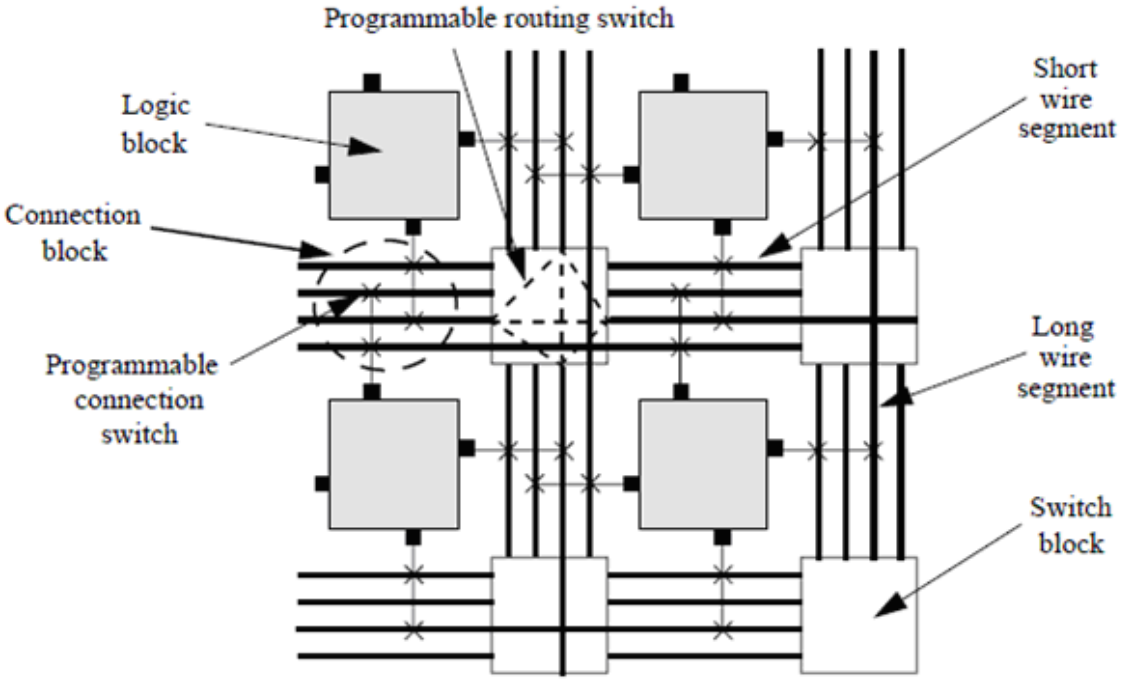

Figure 2.6: An island-style FPGA

The number of tracks, or wires, contained in a channel is denoted by W. The number of wires in each channel to which a logic block pin can connect is called the connection block flexibility, or Fc. The number of wires to which each incoming wire can connect in a switch box is called the switch block flexibility, or Fs[1]. In the FPGA of Figure 2.6, for example, W is 4 for all channels, Fc is 2 and Fs is 3.

Considering the importance of an FPGA's routing architecture to both its area-efficiency and speed, relatively little prior research has been conducted. The question of how many wires each channel in an FPGA should contain relative to the other channels, which is called as a global routing architecture of an FPGA has not been studied before. There is some prior work concerning the detailed routing architecture of FPGAs, however.

Most prior work has investigated FPGAs in which all wires span only one logic block before terminating at a switch block, and have compared architectures only on the basis of area efficiency. The area-efficiency metric in these studies has usually been the number of programmable switches contained in the routing.

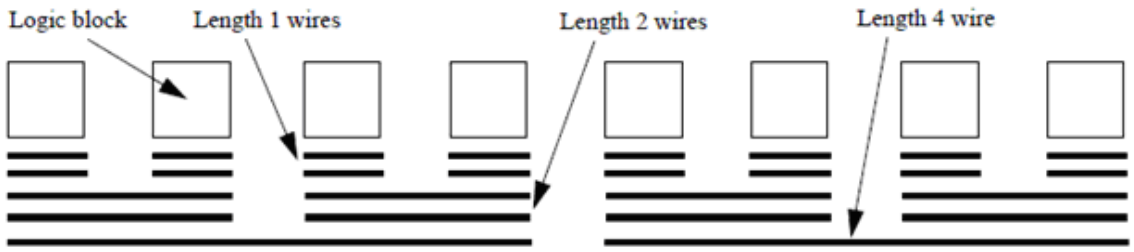

Figure 2.7: Example channel segmentation distribution

A few studies have looked at routing architectures that include different lengths of wires. Figure 2.7 shows one channel in such an FPGA. The length of a wiring segment is the number of logic blocks it spans; Figure 2.7 shows segments of length 1, 2 and 4. The segmentation distribution defines what fraction of the tracks in each channel are of each length. In Figure 2.7 for example, 40% of the tracks are of length 1, 40% are of length 2, and 20% are of length 4. When a wire spans the entire width

or height of an FPGA, use Xilinx's terminology and call it a long line.

2.5 PiCoGA Structure

The PiCoGA is an array of rows, each representing a possible stage of a customized pipeline. The width of the datapath obtained should fit the processor one, so each row is able to process 32-bit operands. As shown in Figure 2.8, each row is connected to the other ones with configurable interconnect channels and to the processor register file with six 32-bit busses[2].

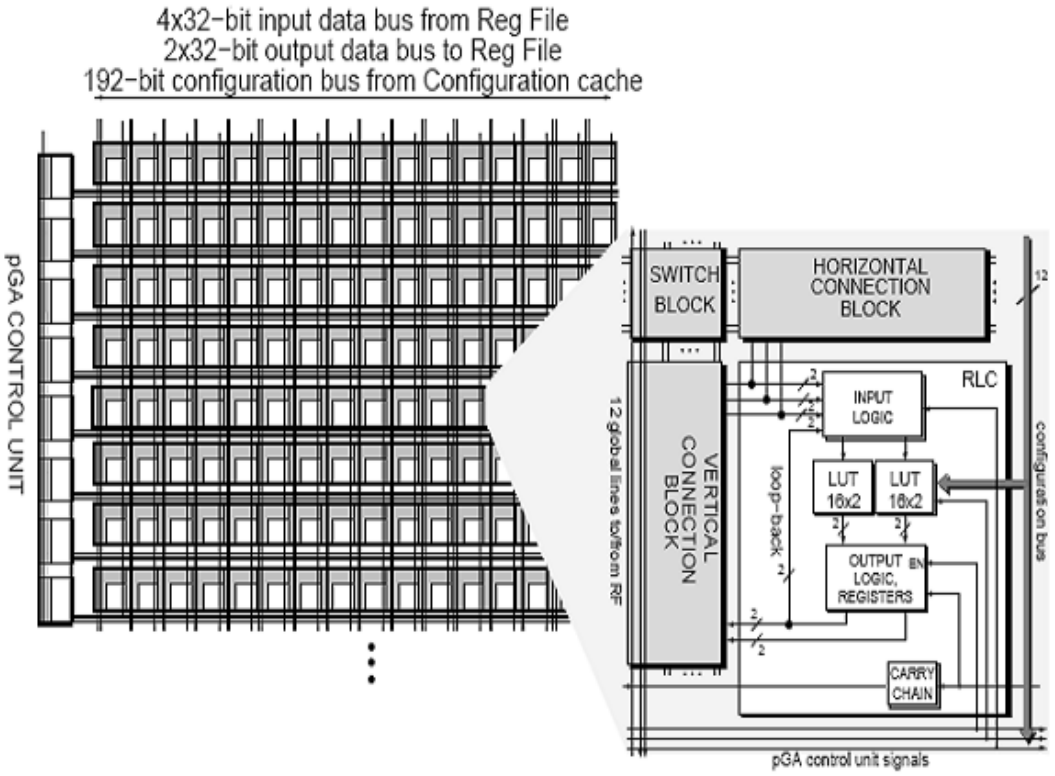

Figure 2.8: PiCoGA Structure

In a single cycle, four words can be received from the register file and up to two words can be produced for write-back operations. The busses span the whole array, so that any row can access them, improving routability. Pipeline activity is controlled by

a dedicated configurable control unit, which generates three signals for each row of the array. The first one enables the execution of the pipeline stage, allowing the registers in the row to sample new data. In every cycle, only rows having input data ready are activated. In this way, a state stored in flip-flops inside the array can be correctly held and at the same time unnecessary power dissipation is avoided. The second signal controls initialization steps of a state held inside the array, while the third enables a burst write of LUTs with data available in the processor Register File. Each row is composed of 16 Reconfigurable Logic Cells (RLC) and a configurable horizontal interconnect channel. Vertical channels have 12 pairs of wires while horizontal ones have only 8 pairs of wires. Switch blocks adjacent to each RLC connect vertical and horizontal wires. Since most of the remaining portion of control logic not mapped in the processor standard dataflow is implemented in the configurable control unit, the array core can be datapath oriented. Therefore, the PiCoGA has a 2-bit granularity for both interconnections and LUTs, except for input connection blocks which have 1-bit granularity. This is a good compromise if considered that bit-level operators such as bit permutation, which are frequent in cryptography algorithms, are not well supported by other functional units.

Chapter 3

Placement Algorithms

Placement algorithms determine which logic block within an FPGA should implement each of the logic blocks required by the circuit. The optimization goals are to place connected logic blocks close together to minimize the required wiring (wirelength-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement).

The three major classes of placers in use today are min-cut (partitioning-based), analytic which are often followed by local iterative improvement, and simulated annealing based placers. This means the optimization goals of our placer may change from architecture to architecture. It is much easier to add new optimization goals or constraints to a simulated annealing based placer than to a min-cut or an analytic placer, so the focus on this algorithm below.

3.1 Force Directed Placement

Force-directed placement algorithms are rich in variety and differ greatly in implementation details[3]. The common denominator in these algorithms is the method of calculating the location where a module should be placed in order to achieve its ideal placement. This method is as follows. Consider any given initial placement.

Assume the modules that are connected by nets exert an attractive force on each other (Figure 3.1).

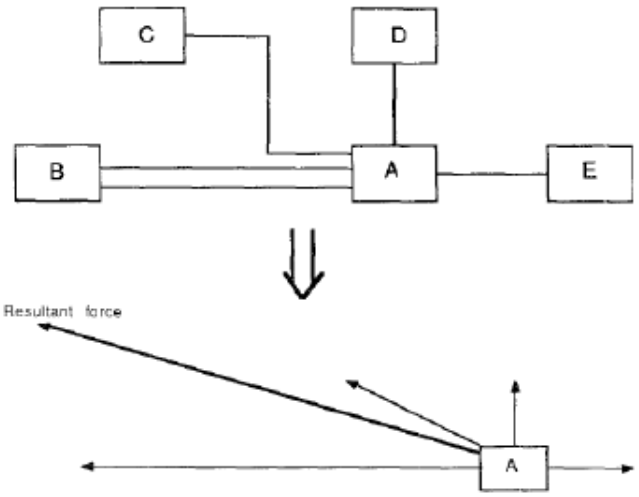

Figure 3.1: Example Global Routing Architecture

The magnitude of the force between any two modules is directly proportional to the distance between the modules. as in Hookes law for the force exerted by stretched springs, the constant of proportionality being the sum of weights of all nets directly connecting them. If the modules in such a system were allowed to move freely, they would move in the direction of the force until the system achieved equilibrium in a minimum energy state, that is, with the springs in minimum tension (which is equivalent to minimum wire length), and a zero resultant force on each module. Hence the force-directed placement methods are based on moving the modules in the direction of the total force exerted on them until this force is zero.

3.1.1 Force Directed Placement Techniques

The early implementations of the force-directed placement algorithm were in the 1960s. There are many variations in existence today. Some are constructive; some are based on iterative improvement [3].

Here, one problem is to decide the order in which to select the modules for moving to the target location. In most implementations, the module or seed module with the strongest force vector is selected. In other implementations, the modules are selected randomly. In still others, the modules are selected on the basis of some estimate of their connectivity. Another problem is where to move the selected module if the slot nearest to the zero force target location is already occupied, as it most probably will be.

One solution is to move it to the nearest available free location. But the nearest free location may be very far in some cases. This is an approximate method and, at best, will need more iterations to achieve a good solution.

The second solution is to compute the target location of a module selected as described above, then evaluate the change in wire length or cost when the module is interchanged with the module at the target location. If there is a reduction in the wire length, the interchange is accepted; otherwise it is rejected. It is necessary to evaluate the wire length because it is possible that in an attempt to interchange the selected module with the module previously at the target point, we are moving that other module far away from its own target point; hence the move can result in a loss instead of a gain.

The third solution is to perform a ripple move; that is, select the module previously occupying the target point for the next move. This process is continued until the target point of a module lies at an empty slot. Then a new seed is selected.

The fourth solution is to compute the target point of each module, then look for pairs of modules such that the target point of one module is very close to the current location of the other. If such modules are interchanged, both of them will achieve their target locations with mutual benefit.

The fifth solution uses repeated trial interchanges. If an interchange reduces the cost, it is accepted; otherwise it is rejected. The cost function in this case is the sum of the forces acting on the modules.

3.2 Placement by Partition

Placement by partitioning is an important class of placement algorithms based on repeated division of the given circuit into densely connected sub-circuits such that the number of nets cut by the partition is minimized [4]. Also, with each partitioning of the circuit, the available chip area is partitioned alternately in the horizontal and vertical direction. Each sub-circuit is assigned to one partition of the chip area. If this process is carried on until each sub-circuit consists of only one module, then each module will have been mapped to a unique position on the chip. Most placement by partitioning algorithms, or Min-cut algorithms, use some modified form of the Kernighan-Lin and Fiduccia-Mattheyses heuristics for partitioning.

The Kernighan-Lin partitioning algorithm is as follows. Start with a random initial partition that divides the set of modules into two disjoint sets A and B. Evaluate the net cut (the number of nets connecting modules in A to modules in B and are therefore cut by the partition). For all pairs (a, b), a cA, b e B, find the reduction g in the net cut obtained by interchanging a and b (moving a to set B and b to A). g is called the gain of the interchange. If g $>$ 0, then the interchange is beneficial. Select the module pair (a_l, b_l) with the highest gain gl. Remove al and bl from A and B, and find the new maximum gain g_2 for a pairwise interchange (a_2, b_2). Continue this process until A and B are empty.

3.2.1 Breuer's Algorithms

Breuers algorithms are among the early applications of partitioning for placement[5]. They minimize the number of nets that are cut when the circuit is repeatedly partitioned along a given set of cut lines. Consider a set of modules connected by a set of nets. Let c be a line crossing the surface of the chip. If one or more elements connected to a net s are on one side of c and one or more elements are on the other side, then, while routing the net, at least one connection must cross line c. The cut line c is said to cut the net s.

Cut Oriented Min-Cut Placement Algorithm. Start with the entire chip and a given set of cut lines. Let the first cut line partition the chip into two blocks. Also partition the circuit into two subcircuits such that the net-cut is minimized. Now partition all the blocks intersected by the second cut line, and partition the circuit correspondingly. Repeat this procedure for all cut lines. This process is shown in Figure 3.2a.

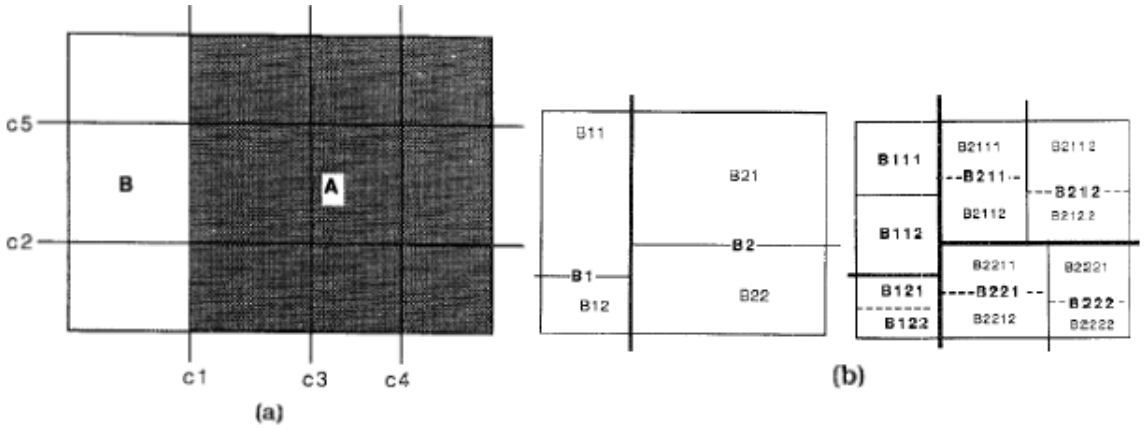

Figure 3.2: (a) Cut-oriented, (b) Block-oriented

Block-Oriented Min-Cut Placement Algorithm. In this algorithm, we select a cut line to partition the chip into two regions. Then we select a separate cut line for each region and partition the regions further. This process is repeated until each block consists of one slot only. Here, different regions cam have different cut lines, as shown in Figure 3.2b.

The cut lines for partitioning the chip may be selected in any sequence. Breuer has given three sequences, which are most suitable for three different types of layout. These are as follows:

a. Quadrature Placement Procedure

b. Bisection Placement Procedure

c. Slice Bisection Procedure

3.3 Clustering Approach

Clustering is another class of placement technique[6]. We have given a netlist hypergraph H(V, E) consist of a set of modules (vertices) V = v1, v2, , vn and a set of nets (hyperedges) E = e1, e2, , en. So, we can define a cluster Ci is a non-empty subset of V. K-way clustering (Pk) is a set of k clusters such that every vi from set V belongs to exactly one cluster in Pk. Given H(V,E), a value 2¡=k¡=n, and cluster size bounds Lower (L) & Upper (U), construct Pk = C1, C2, , Ck with L¡=—Ci—¡= U, 1¡=i¡=k, that optimizes a given objective function f(Pk). Pk is referred as clustering when k size is large and as partitioning when k is small, k ¡= 10.

Clustering contracts a large problem instance into smaller instance this saves runtime or allows better search of the solution space. When it is known that cells should be near each other in the placement, clustering them together prevents from making a mistake. Clustering can incorporate knowledge of problem structure that the placer would otherwise have to ignore.

So, Clustering can speed up placement process and lead to better placement solution quality. Output generated by clustering can applied to any placement techniques to improve the performance of the technique. This technique gives poor results but it is the fastest method for placement.

3.4 Simulated Annealing

Simulated annealing mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects. Pseudo-code for a generic simulated annealing-based placer is shown in Algorithm 3.1 [1] [7]. A cost function is used to evaluate the quality of any placement of logic blocks – for example, a common cost function in wirelength-driven placement is the sum over all nets of the half perimeter of their bounding boxes. An initial placement is created by assigning logic blocks randomly to the available locations in the FPGA. A large number of moves, or local

improvements, are then made to gradually improve the placement. A logic block is selected at random, and a new location for it is also selected at random. The change in cost function that would result from moving the selected logic block to the proposed new location is computed. If the cost would decrease, the move is always accepted and the block is moved. If the cost would increase, there is still a chance of the move being accepted, even though it makes the placement worse. This probability of acceptance is given by $e^{-DC/T}$, where DC is the (positive) change in cost function the move causes, and T is a parameter called temperature that controls the likelihood of accepting moves that make the placement worse. Initially, T is very high so almost all moves are accepted; it is gradually decreased as the placement is refined so that eventually the probability of accepting a move that makes the placement worse is very low. This ability to accept hill-climbing moves that make a placement worse allows simulated annealing to escape local minima in the cost function.

Algorithm 3.1 Pseudo-code of a generic simulated annealing-based placer

```
    S = RandomPlacement ();
    T = InitialTemperature ();
    Rlimit = InitialRlimit ();
1   while (ExitCriterion () == False) { /* "Outer loop" */
2       while (InnerLoopCriterion () == False) { /* "Inner loop" */
3           Snew = GenerateViaMove (S, Rlimit);
4           DC = Cost (Snew) - Cost (S);
5           r = random (0,1);
6           if (r < e^(-DC/T)) {
7               S = Snew;
8           }
9       } /* End "inner loop" */
10      T = UpdateTemp ();
11      Rlimit = UpdateRlimit ();
12  } /* End "outer loop" */
```

The rate at which temperature is decreased, the exit criterion for terminating the anneal, the number of moves attempted at each temperature (InnerLoopCriterion), and the method by which potential moves are generated are defined by the annealing schedule. A good annealing schedule is crucial to obtain good results in a reasonable amount of CPU time. Many proposed annealing schedules are "fixed" schedules that have no ability to adapt to different problems. Such schedules can work well within the narrow application range for which they were developed, but their lack of adaptability means they are not very general. Accordingly, we discuss below only "adaptive" annealing schedules that determine an annealing schedule based on statistics computed during the anneal itself.

Annealing schedule performs a set of random moves on the initial placement, and sets the initial temperature (InitialT) to 20 s, where s is the standard deviation of the cost over these moves. New temperatures (UpdateT) are computed via:

$$T_{new} = T_{old}(e^{-\lambda T_{old}/\sigma}) \tag{3.1}$$

where λ is typically set to 0.7 and s is the standard deviation of the moves accepted at Told. The InnerLoopCriterion of Algorithm 3.1 is fairly complex for this schedule; it involves monitoring the fraction of new states generated that have their costs within a certain range of the average cost at that temperature, and there are several special cases and fall-back cases defined. Finally, the anneal terminates (ExitCriterion), when the difference between the maximum and minimum costs accepted at that temperature equals the maximum cost change caused by any one move at that temperature.

When we employs feedback control to set the annealing schedule. It monitors the standard deviation of the cost, the average cost, and the fraction of proposed moves that were accepted, α, over the past τ moves. Typically τ is 100. These values are inputs to a sophisticated feedback system that determines a new temperature. In this schedule, a new temperature is computed every move – that is, the "inner loop" in

Algorithm 3.1 executes only one iteration each time control reaches it. The anneal terminates when there has been no change in the average cost for the last $k * \tau$ moves, where k is typically 5. This annealing schedule also employs a range limiter to control the move generation process. The Rlimit parameter in Algorithm 3.1 controls how close together blocks must be to be considered for swapping. Initially, Rlimit is fairly large, and swaps of blocks far apart on a chip are likely. Throughout the anneal, Rlimit is adjusted to try to keep the fraction of moves accepted at any temperature close to 0.44. If the fraction of moves accepted, α, is less than 0.44, Rlimit is reduced, while if α is greater than 0.44, Rlimit is increased.

One disadvantage of this is its complexity. To overcome this problem we can make changes in inner loop exit criteria. In this schedule, the number of moves attempted in the "inner loop" is $10 * Nblocks^{1.33}$. The range limiter, Rlimit is updated according to a fixed (hard-coded) schedule; the exact form of this is not specified, but it likely initially spans the entire chip, and gradually shrinks to a small region. The "outer loop" is executed 150 times, and then the anneal terminates. The temperature is controlled by the fraction of moves accepted:

$$T_{new} = [1 - \frac{\alpha - 0.44}{40}]T_{old} \tag{3.2}$$

where 0.44 is desired acceptance rate, and 40 is a damping coefficient to prevent wild oscillations in temperature. While this schedule is much simpler than previous, it has sacrificed some of the adaptability of the previous schedule, since the range limiter variation and the number of outer loop iterations are now hard-coded.

Since the amount of routing in FPGAs is limited and set by the manufacturer when the FPGA is fabricated, some FPGA placers attempt to optimize not just the wirelength of a placement, but also its routability. Simulated annealing can be used in a tool that performs placement and routing simultaneously in one combined step. After any swap of blocks, all the affected nets are re-routed via a maze router. To keep the CPU time reasonable, this maze router is constrained to look at only a small

number of potential routes when the temperature is high; at lower temperatures, the router is allowed to spend more time looking for routes. If no suitable route is found among the allowed candidates, the net is marked as currently unroutable, and the placement cost is increased. The result quality of this tool is high, but the CPU time required is very large – a circuit containing only 461 Xilinx 4000 logic blocks required 11 hours of CPU time to place and route, and the complexity of this algorithm appears to be approximately O(n3).

3.5 Placement: VPR (Versatile Place and Route)

The Versatile Place and Route (VPR) CAD tool performs placement and either global routing or combined global-detailed routing for FPGAs. The name Versatile Place and Route reflects the primary goal in this tool's design: be able to target a wide variety of FPGA architectures. In the following sections we describe the capabilities of and algorithms used in the VPR placement tool.

3.5.1 Overview of VPR Placement Tool

In VPR, an FPGA is modelled as a set of legal slots, or discrete locations, at which logic blocks or I/O pads can be placed. In keeping with the versatility design goal, an FPGA architecture description file specifies:

- The number of logic block input and output pins
- The number of I/O pads that fit into one row or one column of the FPGA
- The relative widths of the various routing channels across the FPGA

The dimensions of the logic block array must also be set; if these are not specified on the command line, the smallest logic block array that will fit the circuit is used.

In Figure 3.3, for example, two I/O pads fit into each row or column of the FPGA, the logic block array is 4 x 3 logic blocks, and the channels between the logic block

array and the I/O pads are only half as wide as those within the logic block array. Notice that perimeter I/O is assumed (i.e. wire-bonded pads, rather than flip-chip), so I/O blocks can only be placed on the edges of the FPGA.

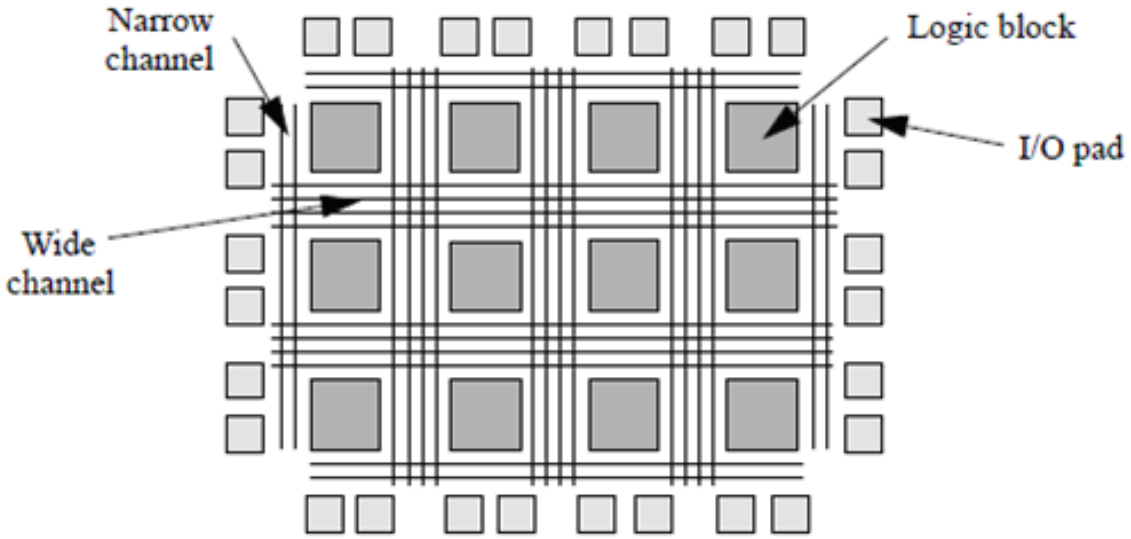

Figure 3.3: FPGA model assumed by VPR placer

Users can have VPR determine good locations for the I/O pads during placement, or they can lock the I/O pads in a configuration specified by an input file, or have VPR generate a random locked I/O configuration[1]. This last capability allows the tool to measure the effect of random pin assignment on different architectures – an important issue in FPGAs. Locked I/Os occur frequently in real FPGA designs, since the circuit board on which an FPGA is mounted is often fabricated before the FPGA circuit design is competed. In such cases, the placement tool must keep the circuit I/Os in the locations required by the circuit board design.

VPR uses the simulated annealing algorithm, which was described in detail in Section 3.4. Since the basic application of simulated annealing to placement is well known, the following sections focus on the aspects of our implementation that are improvements to the prior art.

3.5.2 New Adaptive Annealing Schedule

As described in Section 3.4, a good annealing schedule is essential to obtain high-quality solutions in a reasonable computation time with simulated annealing. Recall that an annealing schedule specifies the number of moves to attempt per temperature, how temperature varies throughout the anneal, and when the anneal should terminate. Our placement tool targets many different FPGA architectures, can use several different cost functions, and is used with a wide variety of circuits that span a large range of sizes. Consequently, we need an annealing schedule that automatically adapts to the current placement problem; fixed annealing schedules will not work well.

Three parts of this annealing schedule are taken from prior work. First, we compute the initial temperature in the same as equation 3.1. Let Nblocks be the total number of logic blocks plus the number of I/O pads in a circuit. We first create a random placement of the circuit. Next, we perform Nblocks moves (pairwise swaps) of logic blocks or I/O pads, and compute the standard deviation of the cost of these Nblocks different configurations. The initial temperature is set to 20 times this standard deviation, ensuring that virtually every move is accepted at the start of the anneal.

The second feature of our annealing schedule taken from prior work is the number of new placement configurations evaluated at each temperature. We set the number of moves per temperature to

$$MovesPerTemperature = InnerNum * (N_{blocks})^{4/3} \tag{3.3}$$

where the default value of InnerNum is 10. This default number can be overridden on the command line, however, to allow different CPU time / placement quality trade-offs. Reducing the number of moves per temperature by a factor of 10, for example, speeds up the placer by a factor of 10 and reduces the final placement quality by less than 10%.

It is desirable to keep the fraction of moves accepted, α, near 0.44 for as long as possible. We accomplish this, by using the value of α to control a range limiter – only interchanges of blocks that are less than or equal to R_{limit} units apart in the x and y directions are attempted. A small value of R_{limit} increases α by ensuring that only blocks which are close together are considered for swapping. These "local swaps" tend to result in relatively small changes in the placement cost, increasing their probability of acceptance. Initially, R_{limit} is set to the span of the entire chip. Whenever the temperature is reduced, the value of R_{limit} is updated according to the value of α measured at the old temperature:

$$R_{limit}^{new} = R_{limit}^{old} * (1 - 0.44 - \alpha) \tag{3.4}$$

and then clamped to the range $1 \leq R_{limit} \leq$ maximum FPGA dimension. This results in Rlimit being the size of the entire chip for the first part of the anneal, shrinking gradually during the middle stages of the anneal, and finally being 1 logic block at low temperatures.

The key difference between our new annealing schedule and previous schedules lies in our method of updating the temperature as the anneal progresses. When the temperature is so high that almost any move is accepted, we are essentially moving randomly from one placement to another and little improvement in cost is obtained. Conversely, if very few moves are being accepted (because the temperature is very low and the current placement is of fairly high quality), there is also little improvement in cost. With this motivation in mind, we created a temperature update scheme that increases the amount of time spent at the most productive temperatures – those where a significant fraction of, but not all, moves are being accepted. We use the fraction of moves being accepted, α, to directly control how quickly the temperature drops. A new temperature is computed as $T_{new} = \gamma T_{old}$, where the value of γ depends on the fraction of attempted moves that were accepted (α) at T_{old}, as shown in Table I. The exact values of α and γ listed in Table I were found via experimentation. While the

α	γ
α >0.96	0.5
0.8 <$\alpha \leq$ 0.96	0.9
0.15 <$\alpha \leq$ 0.8	0.95
$\alpha \leq$ 0.15	0.8

Table I: Temperature Update Schedule

values in Table I led to the best performance, the performance of the annealer is not extremely sensitive to the exact value of γ as a function of α. So long as the function $\gamma(\alpha)$ has the right form – γ is near 1 for a around 0.44, and γ is significantly smaller for α near 1 or 0 – the annealer performs reasonably well.

We terminate the anneal when:

$$T < \epsilon \frac{Cost}{N_{nets}} \tag{3.5}$$

where N_{nets} is the number of nets in the circuit, and we use an ϵ value of 0.005. The movement of a logic block will always affect at least one net. When the temperature is less than a small fraction of the average cost of a net, it is unlikely that any move that results in a cost increase will be accepted, so we terminate the anneal. Again, the performance of the annealer is not terribly sensitive to the ϵ factor in 3.5. Any value between 0.05 and 0.005 is reasonable, with smaller values giving slightly higher quality placements at the cost of slightly increased CPU time.

The annealing schedule described above has produced excellent results with a wide variety of cost functions, FPGA architectures, circuits, and moves per temperature (i.e. desired quality) values. This annealing schedule was not sufficiently robust for our purposes, particularly with large circuits. For some circuits its temperature update scheme became too conservative, and the temperature decreased extremely slowly. For some other circuits its exit criterion did not function correctly, and a large amount of CPU time was wasted at very low temperatures with no significant quality improvement.

3.5.3 New Cost Function

One of the FPGA architectural issues we investigate is global routing architecture. Recall that many global routing architectures have wider channels in some regions of the FPGA than in others. To fully optimize for such architectures, those portions of a circuit that require more routing should be placed in regions of the FPGA that have wider routing channels. The key to obtaining such global-routing-architecture-aware placements is ensuring that the cost function used properly models the relative difficulty of routing connections in areas with different channel widths. Accordingly, we developed what we call a linear congestion cost function. Of all the alternatives we have explored, this cost function provides the best results in a reasonable computation time. Its functional form is

$$Cost_{linearcongesion} = \sum_{i=1}^{N_{nets}} q(i)[\frac{bb_x(i)}{C_{av,x}(i)^\beta} + \frac{bb_y(i)}{C_{av,y}(i)^\beta}] \tag{3.6}$$

where the summation is over the N_{nets} in the circuit. For each net, i, bb_x(i) and bb_y(i) denote the horizontal and vertical spans of its bounding box, respectively. The q(i) factor compensates for the fact that the bounding box wire length model underestimates the wiring necessary to connect nets with more than three terminals. Its value depends on the number of terminals of net i. We obtained the appropriate values of q(i); q(i) is 1 for nets with 3 or fewer terminals, and slowly increases to 2.79 for nets with 50 terminals. $C_{av,x}$(i) and $C_{av,y}$(i) are the average channel capacities (in tracks) in the x and y directions, respectively, over the bounding box of net i.

This cost function penalizes placements which require more routing in areas of the FPGA that have narrower channels. The exponent, β, in the cost function allows the relative cost of using narrow and wide channels to be adjusted. When β is zero the linear congestion cost function reverts to the standard bounding box cost function. The larger the value of β, the more wiring in narrow channels is penalized relative to wiring in wider channels; we have experimentally found that setting β to one results in the highest quality placements.

C_{av} depends only on the channel capacities, which do not change during a placement, and on the maximum and minimum coordinates of a bounding box. We therefore precompute all possible $C_{av,x}$ and $C_{av,y}$ values and store them in a two-dimensional array indexed by the bounding box minimum and maximum coordinates. Consequently, recomputing this cost function is essentially as fast as recomputing the traditional bounding box cost function.

In an FPGA where all channels have the same capacity, C_{av} is also a constant and hence the linear congestion cost function reduces to a bounding box cost function. In FPGAs where some channels are wider than others, however, this cost function results in higher quality placements than a bounding box cost function. The exact amount of routability improvement depends on the precise global routing architecture used; as one would expect, those in which there is a large difference between the widths of channels in different regions show the largest improvement. For the architectures studied in this thesis, placements produced with the linear congestion cost function typically require 5 to 10% fewer tracks to route than placements produced with a bounding box cost function.

3.5.4 Incremental Net Bounding Box Update

Even with a good annealing schedule, millions of potential block swaps will be evaluated in a typical placement run. The most computationally expensive part of evaluating a swap is computing the change in cost, ΔC, the swap would produce; it is crucial that this computation be made as fast as possible.

Consider the computation of ΔC caused by the swap of two blocks. The only terms in the summation of (3.6) that change are those corresponding to the nets attached to the two swapped blocks. The bounding boxes of all the nets attached to either of these two blocks must be recomputed, and then (3.6) can be used to determine ΔC. The recomputation of the net bounding boxes is the key step here; unless care is taken, it can dramatically slow the placer.

The straightforward way to re-evaluate a net's bounding box is to examine the location of each of its terminals. Unfortunately, this is an O(k) operation for a k-terminal net. Large circuits typically have many high-fanout nets, a few of which have hundreds of terminals. As well, since high-fanout nets have terminals on so many blocks, swapping any two blocks has a high probability of disturbing some high-fanout nets.

We have developed an alternative to this brute-force computation, which we call incremental bounding box evaluation. For each net, we store the coordinate of each of the four sides of its bounding box (x_{min}, x_{max}, y_{min}, y_{max}), and the number of net terminals that lie on each of these sides (N_{xmin}, N_{xmax}, N_{ymin}, N_{ymax}). Figure 3.4 shows an example of this data storage.

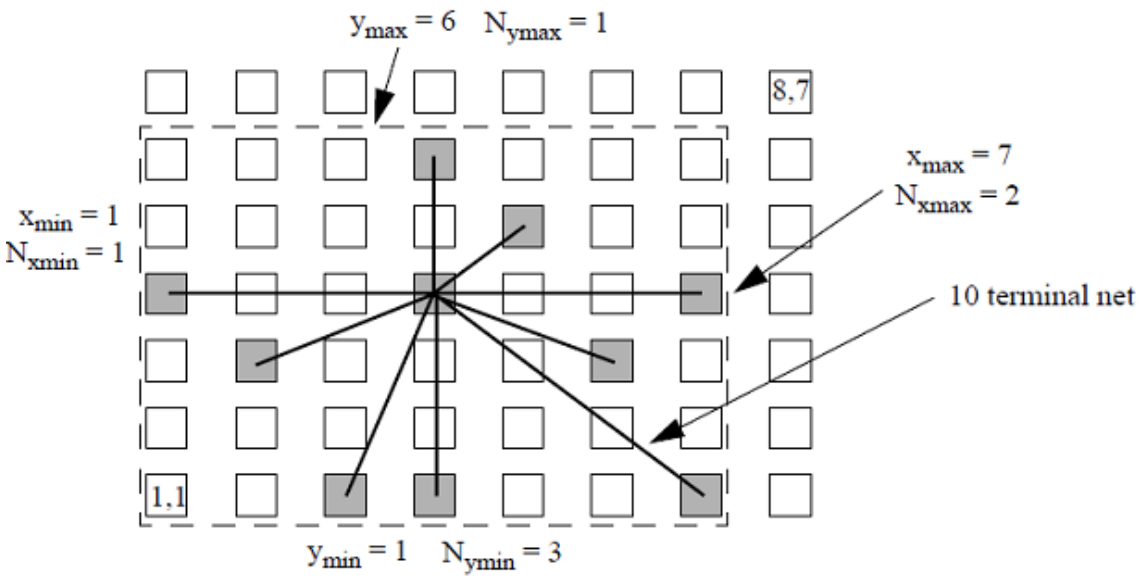

Figure 3.4: Data stored to enable incremental bounding box updates

Now say that some terminal of this net is moved via a swap from (x_{old}, y_{old}) to (x_{new}, y_{new}). Since we have stored the extra information shown in Figure 3.4, we can usually determine the new net bounding box by looking only at the terminal which moved, rather than all k terminals. Algorithm 3.2 lists the pseudo-code used

to update the x_{min} and N_{xmin} values for a net i; the code for the other four sides are similar.

Algorithm 3.2 Pseudo-code to update the bounding box of net i incrementally

```
1   if (x_new != x_old) { /* Terminal has moved horizontally */
2       if (x_new ¡ x_min(i)) { /* Terminal moved left past old x_min edge */
3           x_min(i) = x_new;
4           N_xmin(i) = 1;
5       }
6       else if (xnew == xmin(i)) { /* Terminal moved left to lie on the old x_min edge
        */
7           N_xmin ++;
8       }
9       else if (x_old == x_min(i)) { /* Terminal was on x_min edge; moved right */
10          if (N_xmin ¿ 1) { /* Still terminals on xmin edge? */
11              N_xmin –;
12          }
13          else {
14              BruteForceBoundingBoxRecompute (i);
15          }
16      }
17  }
```

Notice that there is only one case for which the net bounding box must be recomputed by the brute-force procedure: when the terminal moved is the only net terminal on a side of the bounding box, and it is moved inward, toward the bounding box center. In this case the recomputation is O(k), while in all other cases it is O(1). The probability of an arbitrary net terminal being on some side of the bounding box, and being the only terminal on that side of the bounding box is proportional to 1/k, however. Hence the average net bounding box recomputation is O(1 + (1/k).k)= O(1).

We have experimentally determined that our incremental bounding box update

method is faster than the brute-force method for all nets with more than 4 terminals. Using this more sophisticated technique for nets with more than 4 terminals yields, on average, a more than five times speedup in the placer. Table II compares the CPU time on a 300 MHz UltraSparc needed to place the ten largest MCNC benchmark circuits with and without incremental bounding box recalculation. In this experiment each logic block is a BLE (4-LUT / FF pair). The speedup due to incremental bounding box updates ranges from 2.52 times to 9.41 times. The variation in speedup is due to the different fanout distributions of these circuits – circuits with a higher average fanout benefit more. Since placement is so timeconsuming, and there is considerable need for fast CAD tools as FPGA sizes increase, this speedup is very important. Another useful feature of the incremental bounding box code is that it makes the CPU time required to place a circuit highly predictable from the circuit size; this allows a CAD tool to give a user an accurate estimate of the time required to place a circuit.

Circuit	***# Logic Blocks***	***Without incremental bounding box***	***With incremental bounding box***	***SpeedUp***
apex2	1878	116	46	2.52X
frisc	3556	599	127	4.72X
elliptic	3604	864	125	6.91X
s298	1931	386	41	9.41X
pdc	4575	664	172	3.86X

Table II: Placement CPU time with and without incremental bounding box recalculation

The InnerNum value for these results was set to 1; to obtain the highest quality (5 - 10% better) results, InnerNum is set to ten, and therefore ten times more CPU is required than Table II lists.

3.6 Conclusion

This paper discussed six classes of VLSI module placement algorithms. Simulated annealing is currently the most popular among researchers and is the best algorithm available in terms of the placement quality, but it takes an excessive amount of computation time. It is derived by analogy from the process of annealing, or the attainment of ordered placement of atoms in a metal during slow cooling from a high temperature.

Clustering algorithms would rank last in terms of placement quality but would probably be the best in terms of cost/performance ratio, since they are much faster than any other algorithms. These algorithms are based on a simple principle the groups of cells that are densely connected to each other grouped with each other and form a cluster/a bigger block.

Clustering followed by Simulated Annealing together these two techniques can reduce the total time and improves the quality of placement also. First we apply Clustering and then Simulated Annealing. For bigger configurations this approach gives best results.

Algorithm	***Result Quality***	***Speed***
Simulated Annealing	Near Optimal	Very Slow
Genetic Algorithm	Near Optimal	Very Slow
Force Directed	Medium...Good	Slow...Medium
Numerical Optimization	Medium...Good	Slow...Medium
Min-Cut	Good	Medium
Clustering and Other Constructive Placement	Poor	Fast

Table III: Comparison of Placement Algorithms

Chapter 4

Programming The Tool

Implementing a circuit in a modern FPGA requires that hundreds of thousands or even millions of programmable switches and configuration bits be set to the proper state, on or off. Clearly if a circuit designer has to specify the state of each programmable switch in an FPGA very few designs will ever be completed! Instead, users of FPGAs describe a circuit at a higher level of abstraction, typically using a hardware description language (such as VHDL) or schematic entry. Computer-Aided Design (CAD) programs then convert this high-level description into a programming file specifying the state of every programmable switch in an FPGA. To keep the complexity of this procedure tractable, the problem of determining how to map a circuit into an FPGA is normally broken into a series of sequential sub problems, as shown in Figure 4.1.

The whole chip has been described in a particular hierarchy. There are some basic units in a chip like Input Registers, Output Register, RLCs and nets to connect these RLCs. The algorithm which is used can directly be applied on RLCs but this process is very time consuming. So, to ease of use and to save the time plus to reduce the complexity of coding, a hierarchy of these units has been created.

- Pipeline
- Cluster

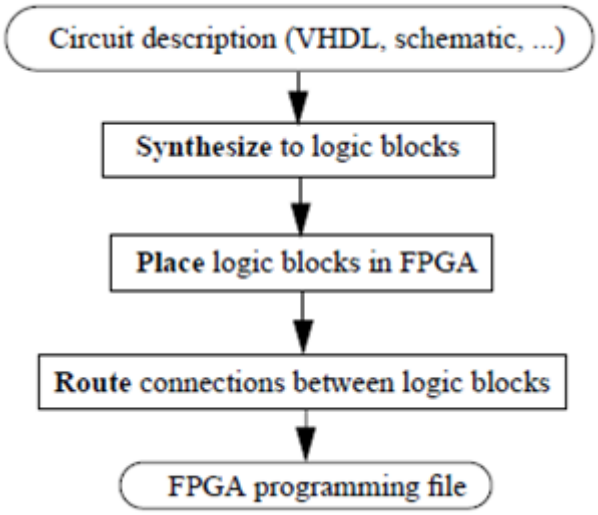

Figure 4.1: FPGA CAD Flow

- Macro
- RLC

Now start from RLC. It is a basic unit of a chip. For PiCoGA we have 24 rows and 16 columns. So, each row contains 16 RLCs. In each tile it has RLC in it because it has an Island routing architecture. According to the requirement and functioning of a chip some RLCs may be strongly connected and some RLCs may loosely connected.

Macro is a collection of RLCs. Based on the connectivity of RLCs, macro can be define. All the RLCs which are responsible for implementing the single functionality can reside in a single macro. A macro can contain two or more RLCs. At the stage of detailed placement, placement has been done at macro level.

Cluster is a collection of Macros. Based on the connectivity of Macros, cluster can be define. All the Macros which are responsible for implementing the single functionality can reside in a single cluster. A cluster can contain more than one Macros. One constraint is that cluster should not expanded more than one physical row. So, in this chip one can have minimum 24 clusters. At the stage of global placement, placement has been done at cluster level.

Pipeline is a collection of clusters. Based on the connectivity of clusters, pipeline can be define. All the clusters which are responsible for implementing the single functionality can reside in a single pipeline. A pipeline can contain more than one clusters. Pipeline can be expanded more than one physical row. So, in this chip one can have minimum 1 pipeline and maximum 24 pipelines.

Specific in this dissertation to design a placer tool reconfigurable logic blocks on eFPGA whole process is divided in subgroups. The whole process is shown in Figure 4.2.

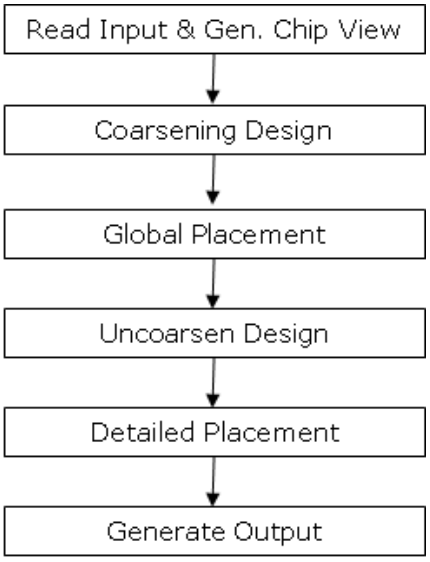

Figure 4.2: PiCoGA Placer Tool Flow

4.1 Read Input & Generate Chip View

This is the first sub module to design a tool. In this sub module, read the input given by user, check the validity of given data, assign these values to appropriate data classes and generate the chip view. User has been provided two types of facilities to give input to tool. A GUI as well as command line option provided to user to input the data. Through GUI user just need to select the particular input files. System

will automatically reads the contents from those input files and design a chip view. Next is command line input option in which user enters the file names at command line and then system reads those files and generates a chip view.

In detail when system have input from user it need to check the validity of all the data. For example, first it has to check the availability of input files. Then check the contents of all the files. Suppose it has detail of RLC, checks the coordinates of it, its connectivity of other RLCs, number of OPF nets, RLCs feeding output register in appropriate column.

Then it checks the availability of all the files which are given by the user. It checks that given path and check the existence of those files, names and path given by user. After checking this thing system will generate the list of units which can be used in placement of chip. It generates RLC list information. In which it contains all the RLC and its detail like source of RLC, sink of RLC, id, name, coordinates, number of OPF nets, parent block of that RLC, reference list index of RLC, Output register list, etc. Same as RLC, create Cluster, Macro and Pipeline list information. Pipeline list contains the objects of pipeline. It has some other attributes like constituent block list, physical row index, etc. Cluster list contains the objects of cluster. Cluster is a part if pipeline. We can say a pipeline can be described as a collection of one or more cluster. Next, it generates the list of input and output registers. These also has some additional parameters. All these registers are 32 bits. According to architecture, it needs total 12, 32 bit input registers to feed all input lines of RLCs. Same as it needs 4, 32 bit output registers to receive the output from RLCs.

After creating a chip view it generates the net information. In which it creates a RLC level, Macro level and pipeline level source and sink information. It designs the complete hierarchy of units. Input registers feeding the RLCs, these RLCs are part of macro and it is part of pipeline. So, on the basis of contents in input file the whole hierarchy is defined this way.

Refer Figure 4.3 at the end of this chapter for flow of this module.

4.2 Clustering

Here clustering algorithm is used to speed up the placement algorithm. Clustering algorithm creates a group of macros which are heavily connected to each other. This algorithm has a list of pipelines, their constituent macros and other related information. From this, create a list of clusters which covers all the existing macros of a single pipeline. More than one cluster of a single pipeline can be created.

Algorithm 4.1 Clustering Algorithm

```
    Priority Queue seedPQ; // based on the degree and connectivity factor of macro.
    Priority Queue gainPQ; // based on the gain.
1   For each pipeline
2       seedPQ = 0;
3       gainPQ = 0;
4       For each Macro
5           Make it seed node.
6           Insert this macro in seedPQ;
7       End
8       For each macro in seedPQ
9           gainPQ = 0;
10          If(current cluster block is full OR cluster block not exist)
11              Create new cluster block;
12          If(macro not in cluster)
13              Add macro to cluster block;
14          createGainPQ(macro);
15          For all macros of gainPQ
16              If(cluster is not full & & macro is not part of cluster)
17                  Add this macro block to cluster;
18          End
19      End
20  End
```

```
21 Function createGainPQ(macro)
22 Create a list of connected and unconnected macros list of this macro.
23 For all macros of this list
24      If(eligible)
25          Compute gain;
26          Add this macro to gainPQ;
27 End
```

To understand the algorithm 4.1 we have to understand some terminologies like; degree, separation, gain, connectivity factor. Degree is the number of connected nets to the macro block. Separation is the number of nodes which are connected to the nets which are inlined to the macro. Connectivity factor is $\frac{degree}{separation^2}$. Gain is depending on the number of pins connected to the net of macro.

$Gain = 2n\frac{2}{r} * (1 + \alpha x)$

Where,

n = number of macros of cluster.

r = number of connected pins to the net.

αx= number of connected pins to the net which are part of cluster.

As shown in algorithm 4.1, system has to add all the macros to an identical cluster. Initially it takes two priority queues for data storage purpose. Then for each pipeline take all constituent macro list. Now add this macros in first priority queue on the basis of degree and connectivity factor.

Now, for each macros in this priority queue add this macro to a cluster block. Either create a new cluster block or use an existing cluster to add this macro block. Now, find the connected and unconnected macro blocks from the same pipeline. And check their eligibility that they are able to add in the current cluster block or not. If the macro block is eligible then add this macro block in the second priority queue on the basis of gain of that macro block.

These eligibility criteria are on the basis of architecture, design, algorithm constraints. There are four eligibility criteria. First, the size of the macro block should not exceed the cluster size after adding this macro block. Second, this macro block should not already be the part of any other cluster. Third, OPF constraints should not violated by adding this macro. Fourth, current cluster is already connected to an output register of a half then added macro should not drive the output register from the same half. It must not drive any output register of the same half.

Now, second queue which contains the macros which can be added to the cluster. Then add these macro blocks one bye one to the current cluster block if they are not part of any other cluster. Subsequently create gain priority queue add keep adding macros to cluster till all the macros are part of some cluster.

At the end list of cluster blocks is been created which contains macro blocks of same pipeline.

Refer Figure 4.4 at the end of this chapter for flow of this module.

4.3 Global/Detailed Placement

This module is the heart of the system. It implements simulated annealing algorithm. The simulated annealing algorithm is implemented twice in whole placer tool. Global Placement do placement of logic cluster blocks which are generated during previous phase which is clustering. Second implementation is Detailed placement which does the placement of logic macro blocks.

To implement this module there is some complecations realted to architecture of a chip. We are implementing placer tool for a ST-PiCoGA chip which has a specific architecture. If the algorothm implemented only for this chip then it will be architecture specific. If in future one want to design the placer tool for some other chip which has some other kind of architecture then this implemetation won't be re used. Then this algorithm has to be redesign for another chip. Each chip has their own constraints. There are 3 kinds of constraints like Architecture, Design and

Algorithm constraints.

Ideally this placer algorithm should be designed in such a way that it should do placement of blocks on chip according to its constraints. But, in this way it will be architecture specific implementation which won't be reused for other chip of different architecture. To avoid this kind of dependencies placement algorithm been implemented in such a way that it can be reused for other chip also.

System already have a database to implement this algorithm. What it does is, it generates a local database which reads all the database blocks and their dependencies. According to constraints it creates a table which stores the available and most feasible place coordinates where a logic block can be placed. And placement algorithm only has to place that logic blocks on chip as per the available space coordinates. So, in this way it has pulled all the constrains dependent components out of the placement implementation. When we want to implement placement of some other chip then this implementation of algorithm will be reused.

Lets first start with Global placement. System implements simulated annealing algorithm on cluster list. This cluster list is generated in previous phase which is coarsening. Cluster is a part of pipeline block. There can be more than one cluster in a single pipeline. Now, initially these pipeline are logically placed in different rows of a chip. Coordinates to these blocks are still not assigned. What is to be done in global placement is it has a table that contains information of child blocks of a pipeline which are nothing but cluster blocks which are generated in previous phase and their feasible row ids where these child blocks can be placed.

In this global placement strategy system has a database which is list of cluster blocks of a pipeline. In placement it moves the cluster blocks to some other empty physical row or swap the blocks to some other cluster block which can belongs to some other pipeline. To move this block apply the same simulated annealing algorithm.

Now, look at the second approach of placement which is detailed placement. This algoritms works same as global placer but one difference is global placer is applied on the cluster blocks where detailed placer is applied on macro blocks. In our design

we already have a set of macros. Each marco is a set of number of RLCs. A macro can contains one or more RLCs. As per the architecture of PiCoGA it has a macro of maximum size 8 means a macro can contains upto 8 RLCs. In detailed placement system creates a table which stores the information of macros of a pipeline. This information contains the valid and feasible space coordinates where we can place this macro. After generating this local database, apply simulated annealing algorithm to place all these macro blocks.

This simulated annealing algorithm gives optimum result but is takes maximum time for execution. The advantage of using this approach is the execution of simulated annealing becomes faster. In global placer the placement at cluster lavel is done so this can be helpful at detailed placer. At detailed placer system just has the scope of available place is limited by the size of the pipeline. So, we don't have to look at the other physical rows of the chip to place macro block of a pipeline where that pipeline is not expanded.

So, using this approach one can create a placement algorithm which is architecture independent. This algorithm is reuseable and can be applied to other chips provided that we have to modify the constraints handler according to the constraints of that chip. But the main implementation which is heart of placer tool won't be affected by changing the chip architecture.

So, this way the whole placement process is done to place the reconfigurable logic blocks on FPGA to reduce the size of a chip. As well as it also reduces the wire length. And hence the main purpose is to reduce the NRE cost also satisfied. We can clearly see that clustering + SA approach gives better results than other algorithms.

Refer Figure 4.5 at the end of this chapter for flow of this module.

As far as the results are concern it is clearly shown that it reduces the bounding box size. This algorithm works in both the ways like it measures cost as well as bounding box dimensions. Below figures 4.6 and 4.7 shows the results computed on 17*17 logic block array which initially has bounding box cost is 75 units which reduced to 30.66 units after applying this algorithm.

4.4 Generate Output

This is a final and a very important sub module for the placement phase. In this module it generates the output as desired by the user. In this module it generates different placement output files, statistics files and log files which stores all the necessary information for next phase of this CAD tool which is routing.

This module generates the command line output as well as GUI based output options depending on the type of input provided by user. If the input to the system given by user is from command line the output will be generated on command line. Where if the input to the system given by user is from command line the output will be generated on command line. System generates ".rlc" file after detail placement which will be the input to the routing phase of CAD tool. Also generate placement statistics file and placement log file and ".cps" file which is coarse placement solution file.

Now if we look into detail mainly we have detail placement solution file which is ".rlc" file which is the most important input to the next phase which is routing. In this file, dump all the details of each RLCs like name, id, source register, destination register, operation code, operands, LUTs, details of input output pins of RLCs. It also contains some additional details like carry bit information, synchronization bit information, **coordinates detail**, etc. Here, a simple example is given below. In our placer tool system generates the same type of information at the end of detail placement.

For Example,

Macro1

RLC1

Details:

Coordinates:

RLC2

Details:

Coordinates:

Then the coarse placement solution file which contains input register information, output register information, pipeline and cluster level information. For IO register this file contains register name, register ID, coordinates, etc. Now the main thing of these file is information about pipeline and cluster level information. It contains cluster level information like, cluster id, name, physical row, number of macros in it, etc for each clusters of each pipeline. Here, a simple example is given below. In this placer tool, it generates the same type of information at the end of coarse placement.

For example,

IO register information

Register name

Register Id

Pipeline Id

Cluster id

Etc

Pipeline id

Then placement statistics file which contains the information about the statistics of placement. It dumps time for coarse placement, time for detail placement, total time taken for placement. Number of clusters, number of pipelines, number physical rows all these information dumps in this placement statistics file. We also have the placement log file which keeps track of all the functionalities of placement of logic blocks.

Refer Figure 4.8 at the end of this chapter for flow of this module.

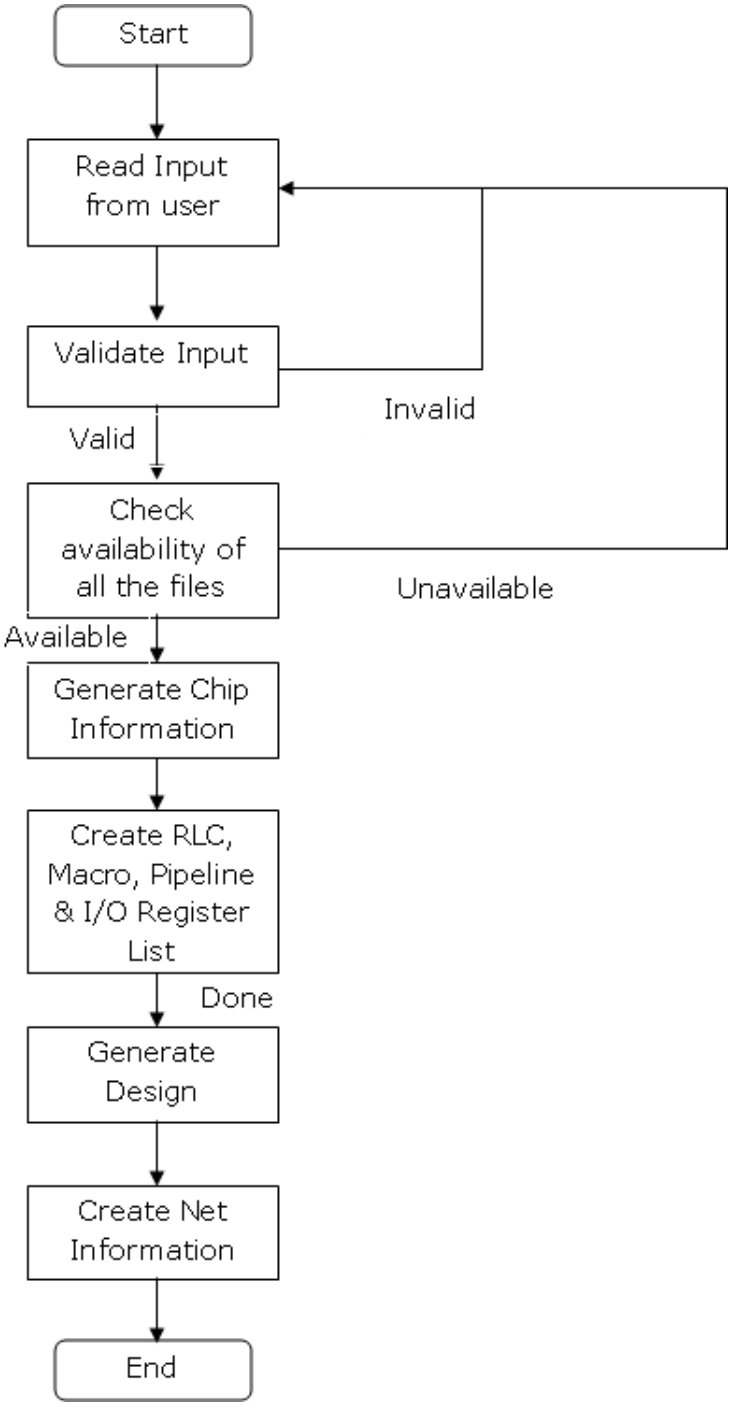

Figure 4.3: Read Input and Generate Chip View Flow Diagram

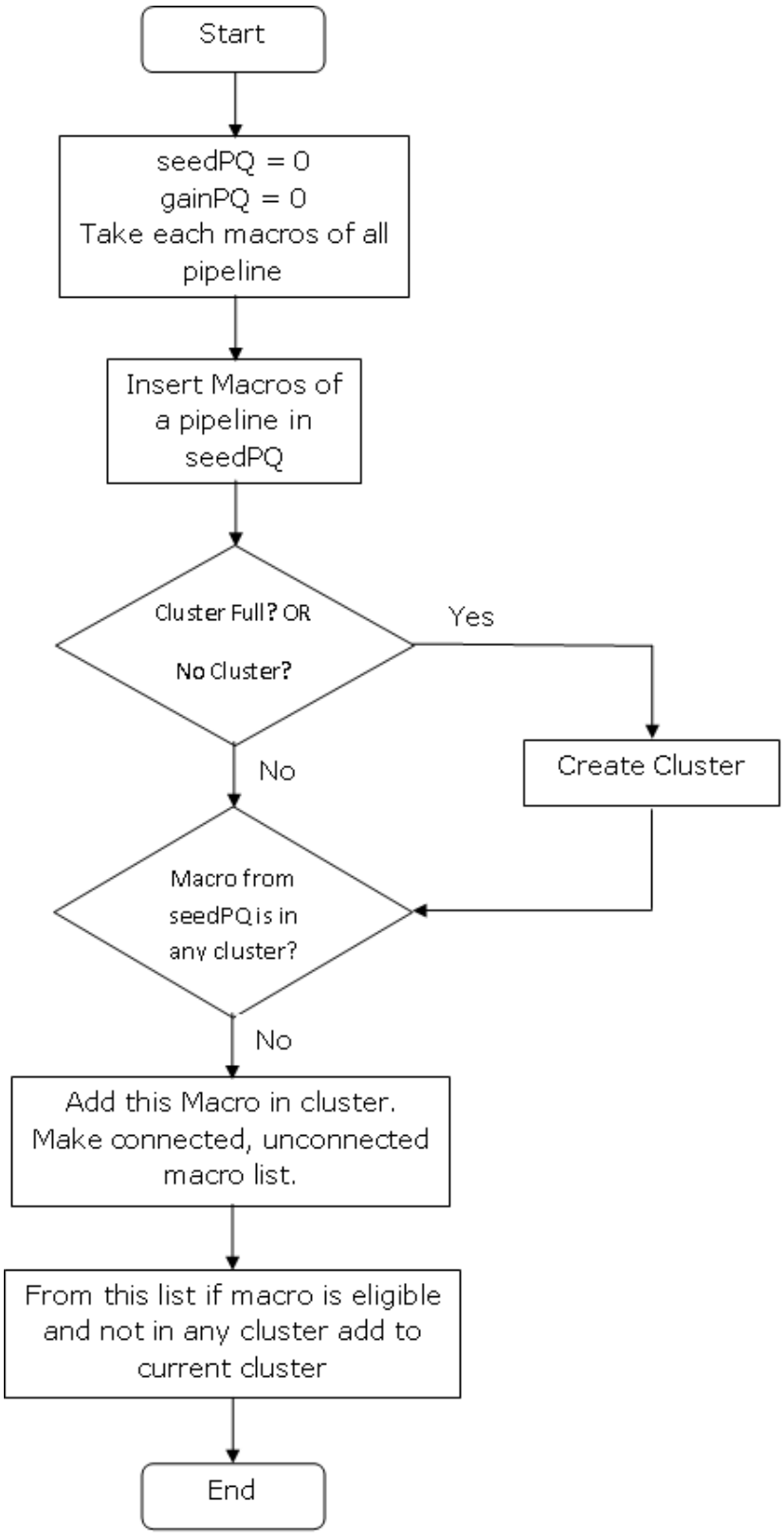

Figure 4.4: Clustering Flow Diagram

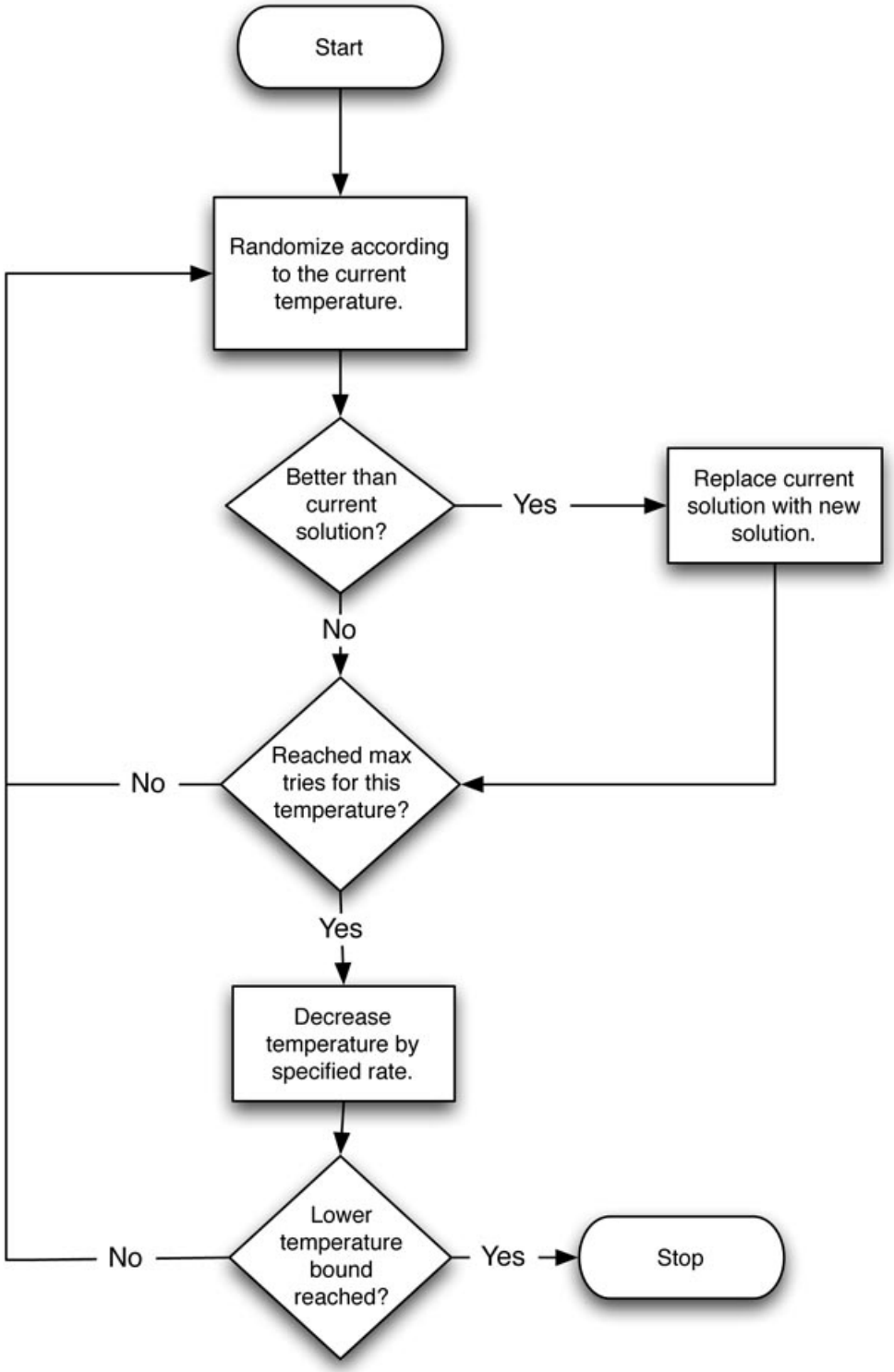

Figure 4.5: Global/Detailed Placement Flow

```
Input netlist file: e64-4lut.net
num_p_inputs: 65, num_p_outputs: 65, num_clbs: 274
num_blocks: 404, num_nets: 339, num_globals: 0
Array size: 17 x 17 logic blocks

There are 995 point to point connections in this circuit

Initial Placement Cost: 1.00608 bb_cost: 75.4585 td_cost: 7.95348e-006 delay_cost: 9.53166e-006
```

T	Cost	Av. BB Cost	Ac Rate	Std Dev	R limit	Exp	Tot. Moves	Alpha
0.15179	0.979092	74.6665	0.9962	0.0112	17	1	29859	0.5
0.075894	0.996279	74.4924	0.9922	0.0116	17	1	59718	0.5
0.037947	1.00237	74.2837	0.9861	0.0108	17	1	89577	0.5
0.018973	0.984255	74.0162	0.9713	0.0107	17	1	119436	0.5
0.0094867	0.978198	73.6597	0.9423	0.0125	17	1	149295	0.9
0.008538	0.987354	73.7311	0.9362	0.0107	17	1	179154	0.9
0.0076842	0.971827	73.3056	0.9303	0.011	17	1	209013	0.9
0.0069158	0.991517	73.345	0.9182	0.0108	17	1	238872	0.9
0.0062242	0.989399	73.2014	0.9154	0.0112	17	1	268731	0.9
0.0056018	0.984362	72.7875	0.905	0.011	17	1	298590	0.9
0.0050416	0.999181	72.7051	0.8891	0.0114	17	1	328449	0.9
0.0045375	0.989816	72.4311	0.8771	0.0112	17	1	358308	0.9
0.0040837	0.992709	72.143	0.8635	0.0125	17	1	388167	0.9
0.0036753	0.995719	71.8924	0.849	0.0124	17	1	418026	0.9
0.0033078	0.98395	71.7114	0.8309	0.012	17	1	447885	0.9
0.002977	0.972265	71.2887	0.811	0.0121	17	1	477744	0.9
0.0026793	0.992633	70.8025	0.788	0.011	17	1	507603	0.95
0.0025454	0.980338	70.4612	0.7726	0.0125	17	1	537462	0.95
0.0024181	0.982159	70.1497	0.7642	0.0127	17	1	567321	0.95
0.0022972	0.989762	70.0788	0.7513	0.0118	17	1	597180	0.95
0.0021823	0.984595	69.9287	0.7447	0.0122	17	1	627039	0.95
0.0020732	0.979223	69.5171	0.7305	0.0127	17	1	656898	0.95
0.0019695	0.986014	69.2953	0.7082	0.0129	17	1	686757	0.95
0.0018711	0.977857	68.9857	0.6992	0.0122	17	1	716616	0.95
0.0017775	0.964947	68.941	0.6924	0.0119	17	1	746475	0.95
0.0016886	0.99495	68.3479	0.6606	0.0117	17	1	776334	0.95
0.0016042	0.970146	68.0914	0.6465	0.0127	17	1	806193	0.95
0.001524	0.980027	67.698	0.6268	0.0144	17	1	836052	0.95
0.0014478	0.981607	67.2095	0.6052	0.0132	17	1	865911	0.95
0.0013754	1.00304	66.124	0.5711	0.0134	17	1	895770	0.95
0.0013066	0.989977	65.6598	0.5503	0.0133	17	1	925629	0.95
0.0012413	0.985296	65.4248	0.5341	0.0128	17	1	955488	0.95
0.0011792	0.960278	64.8491	0.5098	0.0141	17	1	985347	0.95
0.0011203	0.972855	63.7183	0.4734	0.0151	17	1	1015206	0.95
0.0010643	0.961734	63.083	0.458	0.016	17	1	1045065	0.95
0.0010111	0.974344	61.8512	0.4191	0.016	17	1	1074924	0.95
0.0009605	0.959878	61.5638	0.4339	0.0152	16.65	1.155	1104783	0.95
0.00091247	0.954059	60.1117	0.4013	0.0192	16.54	1.199	1134642	0.95
0.00086685	0.971898	57.5255	0.3363	0.0191	15.9	1.479	1164501	0.95
0.00082351	0.954939	57.0592	0.3227	0.0142	14.25	2.201	1194360	0.95

Figure 4.6: Simulated Annealing Result 1

```
 0.00049306     0.94349     43.1974  0.3468   0.0163    4.283    6.564     1492950     0.95
 0.00046841    0.927158     41.5113  0.401    0.018     3.884    6.738     1522809     0.95
 0.00044499    0.906626     41.5409  0.391    0.0204    3.732    6.805     1552668     0.95
 0.00042274    0.913299     40.9239  0.3694   0.0158     3.55    6.885     1582527     0.95
  0.0004016    0.923981     40.4544  0.3604   0.0143    3.299    6.994     1612386     0.95
 0.00038152    0.929808     39.6898  0.3442   0.0151    3.037    7.109     1642245     0.95
 0.00036245    0.936117     38.9752  0.4277   0.0118    2.745    7.236     1672104     0.95
 0.00034433    0.922056     38.3171  0.4106   0.0146    2.712    7.251     1701963     0.95
 0.00032711     0.93464     38.1174  0.398    0.0146    2.632    7.286     1731822     0.95
 0.00031075    0.932475     37.3547  0.3772   0.0129    2.521    7.334     1761681     0.95
 0.00029522    0.950116     36.7286  0.356    0.0102    2.363    7.404     1791540     0.95
 0.00028046     0.94296     36.7226  0.3527   0.0143    2.165     7.49     1821399     0.95
 0.00026643    0.952598     35.7811  0.4711  0.00977    1.976    7.573     1851258     0.95
 0.00025311    0.950776     35.4791  0.3044   0.0127    2.037    7.546     1881117     0.95
 0.00024046    0.947577      34.939  0.4295    0.011    1.761    7.667     1910976     0.95
 0.00022843     0.94804     34.8305  0.4235    0.014    1.742    7.675     1940835     0.95
 0.00021701    0.962842     34.5062  0.397    0.0107    1.713    7.688     1970694     0.95
 0.00020616    0.955472     33.9105  0.3848   0.0109     1.64     7.72     2000553     0.95
 0.00019585    0.973865     33.4838  0.3592  0.00803    1.549     7.76     2030412     0.95
 0.00018606    0.961311     33.0206  0.3392  0.00642    1.424    7.814     2060271     0.95
 0.00017676    0.961435     33.2563  0.3351   0.0115    1.281    7.877     2090130     0.95
 0.00016792    0.955298     32.6346  0.3108  0.00984    1.146    7.936     2119989     0.95
 0.00015952    0.979691     32.5495  0.2903  0.00602        1        8     2149848     0.95
 0.00015155    0.976172     32.4026  0.2845  0.00511        1        8     2179707     0.95
 0.00014397    0.967172     32.3213  0.2654  0.00991        1        8     2209566     0.95
 0.00013677    0.964262     32.2095  0.2601  0.00835        1        8     2239425     0.95
 0.00012993     0.96473     32.0919  0.2546  0.00924        1        8     2269284     0.95
 0.00012344     0.97692     31.8031  0.2246  0.00557        1        8     2299143     0.95
 0.00011726    0.975367      31.477  0.21    0.00568        1        8     2329002     0.95
  0.0001114    0.979513     31.7295  0.2093  0.00645        1        8     2358861     0.95
 0.00010583     0.98555     31.5572  0.1981  0.00505        1        8     2388720     0.95
 0.00010054    0.987171     31.3026  0.185   0.00325        1        8     2418579     0.95
9.5512e-005    0.990467     31.2339  0.1861  0.00246        1        8     2448438     0.95
9.0737e-005    0.988172     31.2289  0.1725  0.00312        1        8     2478297     0.95
  8.62e-005    0.986761     31.2364  0.1628  0.00303        1        8     2508156     0.95
 8.189e-005    0.992262     31.2253  0.1485   0.0019        1        8     2538015      0.8
6.5512e-005    0.988605     31.0574  0.1211  0.00319        1        8     2567874      0.8
5.2409e-005    0.991098     30.9741  0.1084  0.00203        1        8     2597733      0.8
4.1928e-005    0.994011     30.9469 0.09635  0.00139        1        8     2627592      0.8
3.3542e-005    0.994473     30.8124 0.07217  0.00146        1        8     2657451      0.8
2.6834e-005    0.995634     30.7457 0.06256  0.00117        1        8     2687310      0.8
2.1467e-005    0.996335     30.6807 0.05369 0.000885        1        8     2717169      0.8
1.7174e-005    0.997323     30.7307 0.04659 0.000661        1        8     2747028      0.8
          0    0.994757     30.6975 0.005626 0.000877       1        8     2776887

bb_cost recomputed from scratch is 30.6597.

Completed placement consistency check successfully.

Placement. Cost: 0.994113  bb_cost: 30.66
```

Figure 4.7: Simulated Annealing Result 2

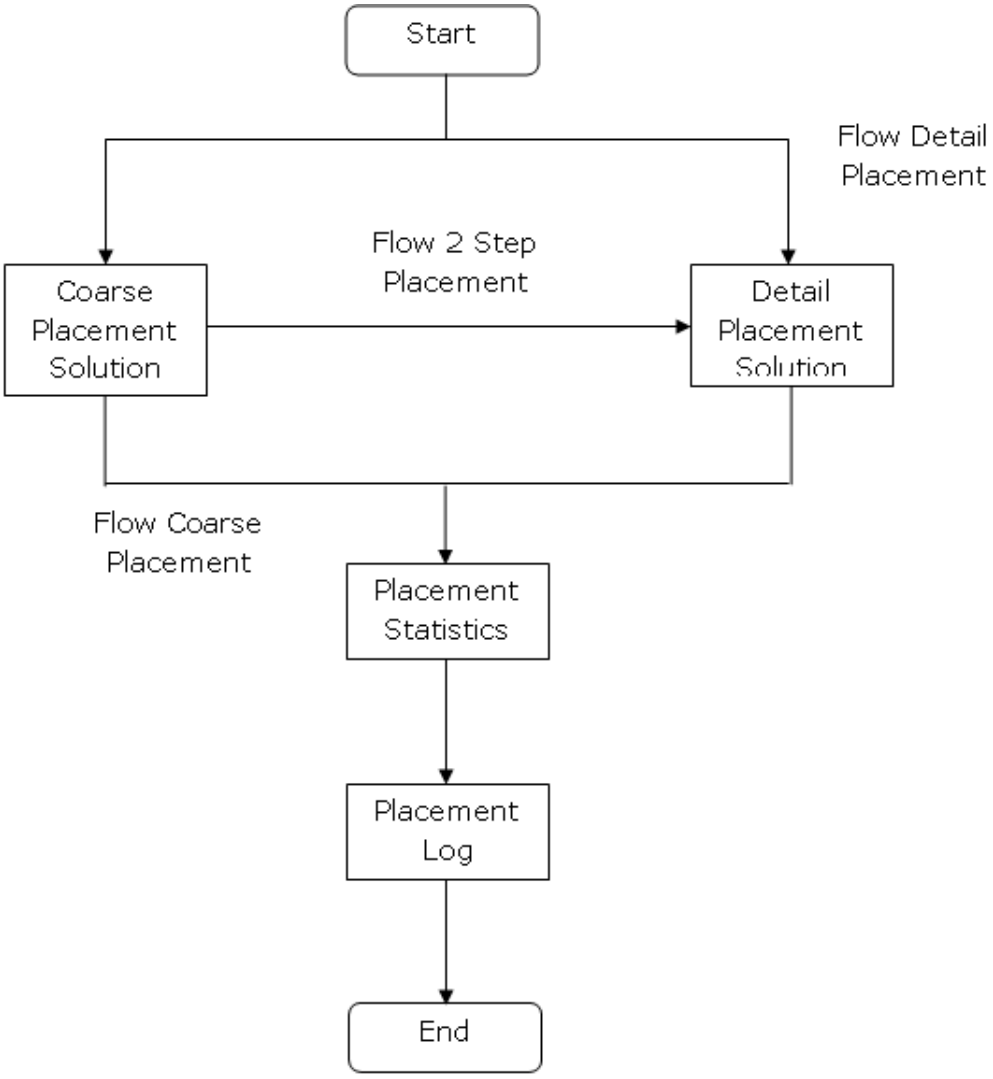

Figure 4.8: Generate Output Flow Diagram

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

In this dissertation placement tools presented here that take advantage of the PiCoGA architecture. Placement is particularly difficult for this architecture because of the limited routing resources available. It also shows the modifications required to adapt these placement tools for implementing asynchronous circuits on Montage. These algorithms have implications beyond that of this single architecture.

Our three stage hierarchical placement methodology combines clustering technique with SA. First stage, Clustering technique is used to condense the input network. Produced clusters have similar sizes, which greatly aids the following annealing stage. Second stage, Condensed network is placed using SA. Third Stage use a low temperature annealing algorithm on the flattened network to optimize the local ordering of the cells. Cells which belonged to different (nearby) clusters may be exchanged.

This dissertation has contributed to two related research areas: FPGA CAD algorithms and FPGA architecture. The new CAD algorithms and tools developed in this research were described in Chapters 3 and 4, and are briefly summarized in Table I. In Chapter 4, This is the logic block packing tool targeting cluster-based logic blocks for private use of company. Here a new simulated annealing based placement tool cre-

CAD Area	***Contributions***
Placement	Architecture Independent Placement Tool New, robust annealing schedule Fast incremental update scheme High Quality results

Table I: Summary of CAD Contributions

ated(the placement portion of our Versatile Place and Route (VPR) program) which incorporates three new enhancements over prior tools. First, a new annealing schedule implemented that adapts automatically to different placement problems, provides good result quality and is more robust. Second, a new, linear congestion placement cost function designed that enhances the routability of circuits mapped to FPGAs in which different channels have different widths. Finally, an incremental net bounding box update algorithm implemented that reduces the CPU time required for placement by more than a factor of five, on average, vs. using the traditional brute-force bounding box recomputation.

5.2 Future Scope

There are two different ways in which one can enhance our CAD tools: by improving the core algorithms to increase result quality, and by increasing the flexibility of the FPGA architecture generator to allow easy investigation of a wider class of FPGAs.

The VPR placement algorithm could also be enhanced by adding a cost function that considers the underlying detailed routing architecture when evaluating the cost of a placement. For example, in an FPGA that contains only length 4 wires, any connection between logic blocks in the same row or column is likely to be completed with one wire segment if it spans less than four logic blocks. If, however, the connection spans five logic blocks, or if the connection is between logic blocks that are not in the same row or column, it must use more than one wire segment. Consequently,

the cost function should consider not only wirelength, but also the number of wire segments required to route from a net source to a net sink. Using a cost function of this type results in a routing-aware placement algorithm, without the high CPU time required by simultaneous placement and routing approaches.

Automatically generating good FPGA architectures to match a set of parameters is a new problem in FPGA CAD, and there are many avenues for future research in this area.

Appendix A

Placement Related Algorithms

Start which Min-cut algorithm, first we see Kernighan-Lin bi-partition algorithm[8].

Algorithm A.1 Kernighan-Lin

Input: G = (V,E), |V|= 2n.

Output: Balanced bi-partition A and B with "small" cut cost.

1 Begin

2 Bipartition G into A and B such that $|V_A|=|V_B|$, $V_A \cap V_B = \emptyset$, and $V_A \cup V_B$ = V.

3 repeat

4 Compute D_v, $\forall$ v$\in$ V.

5 for i = 1 to n do

6 Find a pair of Unlocked vertices $v_{ai} \in V_A$ and $v_{bi} \in V_B$ whose exchange makes the largest decrease or smallest increase in cut cost;

7 Mark v_{ai} and v_{bi} as locked, store the gain $\bar{g}_i$, and compute the new C_v, for all unlocked v$\in$ V.

8 Find k, such that $G_k = \sum_{i=1}^{k} \bar{g}_i$ is maximized;

9 if G_k ¿ 0 then

10 Move $v_{a1}...v_{ak}$ from V_A to V_B and $v_{b1}...v_{bk}$ from V_B to V_A;

11 Unlock v, $\forall$ v$\in$ V.

12 until $G_k \leq 0$;

13 end

Now, let us see force directed placement algorithm [9].

Algorithm A.2

```
    set up initial node valocities to (0,0).
    set up initial node positions randomly // make sure that there should no overlaping.
1   Loop
2       total kinetic energy = 0 //running sum of total kinetic energy over all particles
3       for each node
4           net-force = (0, 0) // running sum of total force on this particular node
5           for each other node
6               net-force = net-force + Coulomb repulsion( this node, other node )
7           next node
8           for each spring connected to this node
9               net-force = net-force + Hooke attraction( this node, spring )
10          next spring
11          Find total kinetic energy from the kinetic energy of nodes.
12      next node
13  until total kinetic energy is less than some small number.
```

References

[1] V. Betz, "Architecture and CAD for speed and area optimization of FPGAs," tech. rep., Electrical and Computer Engineering,University of Toronto.

[2] C. Mucci, "Introduction to the DREAM architecture," tech. rep., STMicroelectronics,Bologna, 2006.

[3] Shahookar and Mazumder, "VLSI cell placement techniques," tech. rep., Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan.

[4] D. A. E. and Kernighan, "A procedure for placement of standard cell vlsi circuits," tech. rep., IEEE Trans, Computer-Aided Design CAD-4, Near Visat Petrol Pump, Chandkheda, Gandhinagar, May 1985.

[5] Breuer, "Min-cut placement and design automation and fault tolerant computing," tech. rep., 1977.

[6] J. Cong and M. Smith, "A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design," tech. rep., DAC, 1993.

[7] G. L. K, "Standard cell placement using simulated annealing," tech. rep., In Proceedings of the 24th Design Automation Conference, 1987.

[8] K. Hadi, "A fundamental bi-partition algorithm of kernigan-lin," tech. rep.

[9] Frunchterman, T. M. J., and Reingold, "Graph drawing by force-directed placement," tech. rep., 1991.