Layout and Structure Aware Synthesis of Integrated Circuits

Dissertation
zur Erlangung des Doktorgrades
der Ingenieurswissenschaften

vorgelegt beim Fachbereich
Elektrotechnik und Informationstechnik
der Universität Kaiserslautern

D 386

von
Thomas Kutzschebauch
geb. in Chemnitz

Kaiserslautern im Juni 2003

Vom Fachbereich Elektrotechnik und Informationstechnik der Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation.

Dekan:	Prof. Dr.-Ing. R. Urbansky
Berichterstatter:	Prof. Dr.-Ing. Wolfgang Kunz Universität Kaiserslautern
	Prof. Dr.-Ing. Jochen Jess Technische Universiteit Eindhoven, Niederlande
	Prof. Dr.-Ing. Jochen Beister Universität Kaiserslautern
Datum der Disputation:	27. Juni 2003

Layout- und Strukturorientierte Synthese Integrierter Schaltungen

Dissertation
zur Erlangung des Doktorgrades
der Ingenieurswissenschaften

Acknowledgments

This dissertation is the result of three exciting years as a research student at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. After finishing my research in logic synthesis and physical design, I spent a few months writing my thesis in the Electronic Design Automation Group under Prof. Wolfgang Kunz at the University of Kaiserslautern.

First and foremost I am greatly indebted to Leon Stok, head of the Design Automation Group at the IBM Watson Research Center, who was my supervisor during the time I spent at IBM. His constant professional and personal support and motivation was the driving force of my work. I have to thank him for the opportunity to work in such an excellent research environment while giving me enough space and time to try new ideas and finish this work. His ability to focus on the important things and drive me into the right direction during our numerous discussions has helped me greatly to achieve my goals. His constructive criticism and persistence was essential to my work and made me strive for more.

Even more so, I am obliged to Prof. Wolfgang Kunz for giving me the opportunity to conduct my thesis work at IBM. From the very beginning when I first knocked on his door and suggested this interesting venture of conducting my research in an industrial research center, he welcomed me with open arms and showed great flexibility and understanding. During the course of my thesis he was always helpful, sincere and supportive. His constant encouragement and happy nature helped me greatly to successfully finish my dissertation work.

I also am very thankful to Prof. Jochen Jess, Professor Emeritus of the Design Automation Group at the Technical University of Eindhoven, for volunteering to be the co-supervisor of my thesis. He helped me shape this work and provided valuable comments to improve its content and readability. In addition, he is a dear friend, and

secretly, I have always looked up to him as the wise man that I would like to be one day.

Furthermore, I would like to thank my colleagues at IBM for providing such an enjoyable time and helping me professionally and personally. Foremost, I thank Geert Janssen, a dear friend of mine, for the interesting discussions and his endurance to listen to my endless questions. Even though we are both overly critical at times, we share the passion of perfecting our work. I also thank Daniel Brand, Victor Kravets, Ulrich Finkler and Wilm Donath for many valuable remarks, and Louise Trevillyan for proofreading some of my papers and presentations. I owe a debt of gratitude to Juergen Koehl who introduced me to the Bonn Physical Design Tools and often provided me with good ideas and insights.

I am grateful to all my colleagues at the University of Kaiserslautern, Dominik Stoffel, Markus Wedler, Ingmar Neumann and Kolja Sulimma, for allowing me to spend some of the best times of my life.

In addition, I would like to thank my parents. Without their continuous support and care I could not have reached my goals in life.

Last but not least, I am most thankful to my girlfriend Stephanie Andersen for proofreading the final version of the manuscript and for always understanding and supporting me.

Thomas Kutzschebauch San Jose, June 2003

Habe nun, ach! Philosophie,
Juristerei und Medizin,
Und leider auch Theologie!
Durchaus studiert, mit heißem Bemühn.
Da steh ich nun, ich armer Tor!
Und bin so klug als wie zuvor.

Johann Wolfgang von Goethe, Faust

Contents

Summary

This dissertation presents new algorithms and provides an overall framework for the interaction of the classically separate steps of logic synthesis and physical layout in the design of VLSI circuits. Due to the continuous development of smaller sized fabrication processes and the subsequent domination of interconnect delays, the traditional separation of logical and physical design results in increasingly inaccurate cost functions and aggravates the design closure problem. Consequently, the interaction of physical and logical domains has become one of the greatest challenges in the design of VLSI circuits. To address this challenge, we propose different solutions for the control and datapath logic of a design, and show how to combine them to reach design closure.

The first part of this dissertation presents algorithms for the extraction of regular structures from arbitrary netlists and demonstrates their application to create compact regular layouts during physical design. Regular structures can be used to obtain high-density layouts with short wire length and improved timing while simplifying placement and routing tasks. However, generic logic synthesis destroys a substantial amount of structural regularity. We have addressed this problem and present a regularity-driven synthesis flow that identifies and preserves regular structures during logic synthesis.

In the second part of this thesis, we introduce algorithms that employ physical design information during early logic synthesis to optimize the structure of the netlist, which results in improved wiring congestion and timing. We create an initial placement of the technology-independent netlist, and use the physical location of the individual cells for layout aware technology decomposition, mapping and factorization. As such, we effectively combine the logical and physical stages of the chip design process.

Finally, we combine both methods and present an efficient design flow that combines logical, structural and physical design information. As a result, we improve timing and routing congestion to address the design closure problem.

Zusammenfassung

Diese Dissertation präsentiert neue Algorithmen für die layout- und strukturorientierte Logiksynthese, und stellt ein allgemeines Fundament für die Interaktion der klassischerweise getrennten Schritte der Logiksynthese und des physikalischen Designs beim Entwurf höchstintegrierter Schaltungen (VLSI) bereit. Aufgrund der kontinuierlichen Miniaturisierung der Entwurfsprozesse im Sub-Mikron Bereich wird die Verzögerungszeit in zunehmendem Maße durch die Chip-Verdrahtung anstatt durch die Gatterlaufzeiten bestimmt. Während sich bei der Miniaturisierung die Verzögerungszeit durch lokale Verdrahtung nur geringfügig erhöht, sind globale Verbindungen deutlich stärker betroffen, da ihre Verzögerungszeit hauptsächlich durch das *RC* der Leitung bestimmt wird. Dadurch stellen die bei der klassischen Logikoptimierung angewandten Kostenfunktionen, d.h. die Anzahl der Literale für die Minimierung der Fläche sowie die Anzahl der Stufen auf einem Pfad zur Berechnung der Verzögerungszeit der Schaltung eine in zunehmendem Maße ungenaue Approximation der eigentlichen Kostenfunktionen im späteren Layout dar.

Infolgedessen stellt die Integration von logischem und physikalischem Entwurf eine der größten Herausforderungen beim Entwurf moderner höchstintegrierter Schaltungen dar. Um sich diesen Herausforderungen zu stellen, muss eine enge Verknüpfung von Logik- und Layoutsynthese erfolgen. Zur Lösung dieses Problems schlagen wir eine getrennte Optimierung der Steuer- und Datenpfadlogik integrierter Schaltungen vor.

Struktur-Orientierte Synthese

Moderne höchstintegrierte Schaltungen sind durch einen hohen Anteil von Datenpfadlogik geprägt, um eine hohe Verarbeitungsgeschwindigkeit zu erreichen. Durch die wiederholte Anwendung identischer Bit-Operationen über die Breite eines

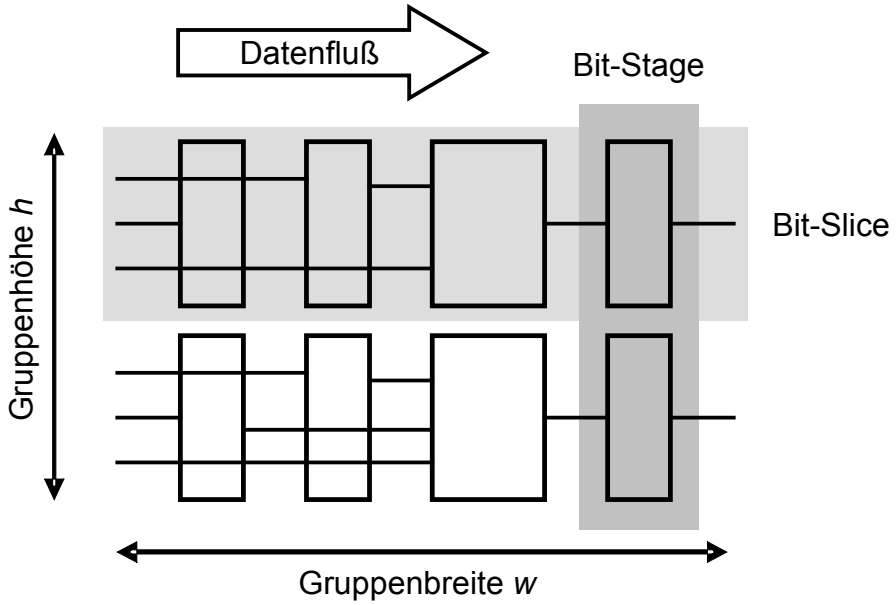


Abbildung 1: Beispiel einer regularen Gruppe

Datenpfades besitzen diese Schaltungen einen hohen Grad an Regularität. Ein Beispiel eines solchen Datenpfades ist in Abbildung 1 dargestellt. Diese Eigenschaft kann dazu genutzt werden, um eine hochkompakte Platzierung mit kürzesten Verbindungsleitungen zwischen den einzelnen Elementen eines Datenpfades zu erzeugen. Daraus ergibt sich eine erhöhte Geschwindigkeit der Schaltung, da die Verdrahtungslaufzeiten minimiert werden. Zusätzlich vereinfacht sich das Platzierungs- und Verdrahtungsproblem, da eine geringere Anzahl von Objekten platziert werden muss, und die Verdrahtung für alle Bits eines Datenpfades identisch ist. Aus diesem Grunde ist die Identifikation von regulären Strukturen von großer Bedeutung für den physikalischen Entwurf. Abbildung 2 zeigt ein Beispiel eines Layouts mit einem hohen Anteil an Regularität. Dabei handelt es sich hauptsächlich um Addierer-Blöcke und andere reguläre Datenpfade.

Im ersten Teil dieser Dissertation entwickeln wir einen schnellen Algorithmus für die Identifikation solcher regulärer Strukturen. Dabei modellieren wir zuerst funktionale Regularität, d.h. eine Struktur die aus mehreren funktionell identischen Teilen besteht. Danach zeigen wir, wie diese Eigenschaft zur Beschleunigung der Logikoptimierung genutzt werden kann. Durch die Wiederverwendung des Optimierungsergebnisses eines

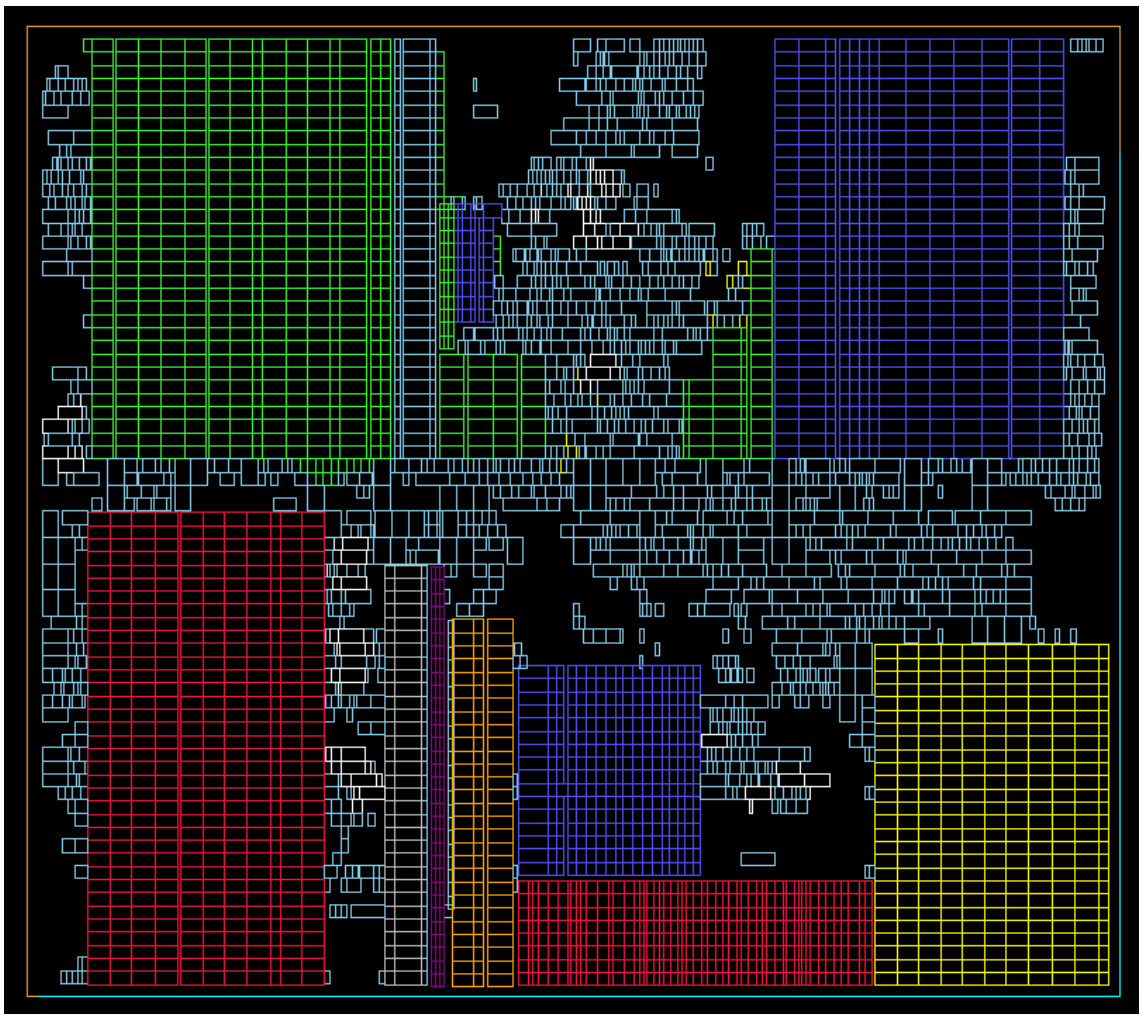


Abbildung 2: Beispiel einer regularen Platzierung

Teils einer Schaltung in anderen, funktional äquivalenten Teilen, kann eine signifikante Beschleunigung des Optimierungsprozesses erreicht werden.

Anschließend erweitern wir unseren Algorithmus zur Extrahierung funktionaler Regularitäten um die Identifikation struktureller Regularität. Schliesslich zeigen wir, wie die gefundene strukturelle Regularität zur Erzeugung hochkompakter Layouts genutzt werden kann. Aufgrund der Bedeutung der strukturellen Eigenschaften einer Schaltung für den physikalischen Entwurf präsentieren wir einen regulären Synthesealgorithmus, der die Regularität einer Schaltung während des gesamten Syntheseprozesses erhält. Die Erhaltung der Regularität ist notwendig, da die klassische

Logikoptimierung einen großen Teil der Datenlogik zerstört, insbesondere während der technologie-unabhängigen Optimierung und Technologieabbildung. Zusätzlich entwickeln wir eine Methode, mit der die Technologie-Optimierung beschleunigt werden kann. Ein Großteil der technologie-abhängigen Transformationen, zum Beispiel Flächen- und Timing-Optimierung wird mittels der so genannten trial-und-error Methode durchgeführt. Dabei werden zuerst eine Anzahl verschiedener Transformationen ausprobiert, bevor die bestmögliche Optimierung schließlich ausgewählt und implementiert wird. Dieser Prozess ist sehr aufwendig, wodurch es zu langen Laufzeiten bei der Optimierung kommt. Wir stellen ein auf dem Prinzip von regulären Schlüsseln basierendes Synthesemodel vor, das eine globale strukturbasierte Optimierung der Netzliste durchführt, und dabei bereits erfolgreiche Transformationen erneut auf einem ähnlichen Teil des Netzwerks anwendet, zum Beispiel innerhalb eines anderen Bits des gleichen Datenpfades.

Layout-Orientierte Synthese

Im zweiten Teil der Arbeit präsentieren wir layout-orientierte Synthesealgorithmen für die Optimierung der nichtregulären Steuerlogik. Dabei erzeugen wir zunächst eine Platzierung der technologie-unabhängigen Netzliste, in dem wir jedem Element eine physikalische Beschreibung zuweisen. Danach benutzen wir die Koordinaten der Objekte, um verbesserte Entscheidungen während der Logiksynthese zu treffen. Dabei werden zum Beispiel die Distanz und die Richtung von einfallenden Kanten bedacht, um Dekompositionen und Technologieabbildungen zu erhalten, die die Netzlänge und Verdrahtungsdichte minimieren. Basierend auf dieser Idee stellen wir layout-orientierte Algorithmen für die Technologie-Dekomposition und –Abbildung, sowie einen Algorithmus für die Faktorisierung Boolescher Ausdrücke unter Berücksichtigung voneinander abhängiger Divisoren vor. Für die Technologieabbildung benutzen wir einen Wellenfront-Algorithmus der auf einem lastunabhängigen Timing-Modell basiert. Um die erhöhte Anzahl von Objekten bei der Platzierung der technologie-unabhängigen Netzliste zu berücksichtigen, benutzen wir Algorithmen für die funktionelle Partitionierung der Logik und für das Clustering der Objekte während des physikalischen Entwurfs. Damit können wir eine deutliche Verbesserung der Laufzeiten der Logikoptimierung sowie der Platzierung erreichen. Im Anschluss zeigen wir, wie die vorgestellten layout-orientierten Algorithmen in einem Platzierungs-Synthese-Ablauf integriert werden.

Integrierter Synthese-Flow

Zum Abschluss zeigen wir anhand eines Beispiels, wie die vorgestellten struktur- und layout-orientierten Algorithmen für den Syntheseablauf von modernen hochkomplexen integrierten Schaltungen genutzt werden können. Dabei extrahieren wir zunächst die reguläre Datenpfadlogik und benutzen unsere reguläre Synthesemethode, um die Strukturen des Designs während des Optimierungsprozesses zu erhalten. Danach erzeugen wir zuerst eine Platzierung der regulären Gruppen und platzieren anschließend die technologie-unabhängige Netzliste. Anhand der gewonnenen Koordinaten der individuellen Objekte wenden wir layout-orientierte Synthesealgorithmen zur Optimierung der Steuerlogik an. Durch die Kombination dieser beiden Methoden können wir die Fläche, das Timing sowie die Verdrahtungsdichte entscheidend verbessern. Diese Methode eignet sich insbesondere für hochmoderne integrierte Schaltungen, die aus einem Gemisch von Datenpfad- und Steuerlogik bestehen.

1 Introduction

This dissertation explores the interaction of logical, structural and physical domains in the computer-automated design of VLSI circuits. Logic synthesis and physical design are one of the most important steps in the design of integrated circuits and greatly determine the overall quality of the circuit. In this chapter, we provide a brief introduction to the whole VLSI design process, show the motivation for combining layout and logical optimization, and outline the remainder of this dissertation.

1.1 VLSI Design Automation

Very Large Scale Integration (VLSI) is widely used in modern digital systems. The design of microprocessors, memories, and application-specific integrated circuits (ASICs) has become impossible without the use of CAD tools. Over the last thirty years, the size of integrated designs has steadily doubled every 18 months, widely known as *Moore's law*. Table 1.1 gives an overview of the complexity of designs exemplified on past and recent Intel® microprocessors. While at the time of the advent of the integrated chip, most circuits could still be designed by hand, the complexity of today's circuits challenges even the most advanced design tools. The design of such highly complex systems requires sophisticated CAD tools for all phases of the design process: synthesis, optimization, physical design, test and verification. While the easy availability of large-scale FPGAs has taken over pieces of the traditional ASIC market, the rising complexity and continuous demand for faster clock speeds and lower power consumption assures that the design of ASIC chips will remain a major driving force for many years to come.

A typical VLSI design process is shown in figure 1.1. It starts with a formal specification of the design in a hardware description language such as Verilog or

Processor	Year	Transistors	Clock Speed	Technology	MIPS
4040	1971	2,300	108 kHz	10 µm	0.06
8086	1978	29,000	10 MHz	3 µm	0.75
80286	1982	134,000	12.5 MHz	1.5 µm	2.66
80486	1989	1.2 million	50 MHz	1 µm	41
Pentium	1993	3.1 million	66 MHz	0.8 µm	112
Pentium II	1997	7.5 million	450 MHz	0.25 µm	800
Pentium 4	2002	55 million	2.53 GHz	0.13 µm	5000

Table 1.1: Illustration of design complexity

VHDL, the *high-level behavioral description*. At this abstraction level, the circuit design is specified using abstract data manipulation operators. *High-level synthesis* generates a structured view of the description by determining an assignment of the circuit functions to operators, called *resources*. This process generates the register-transfer level description (RTL) that determines the block-level structure of the circuit. A typical RTL description contains data storage elements (registers, memories, etc.), functional modules (adders, shifters, etc.), and data selector logic (multiplexors etc.). In the next design step, the *logic synthesis* step, a logic model in form of a gate-level representation of the circuit is created. In addition to the *logic function*, this circuit description exhibits a *structural representation* of the underlying network graph. The resulting network of technology-independent gates is minimized using two- and multi-level logic optimization algorithms, and finally mapped into a specific design library. After optimizing delay and area of the technology-mapped gate-level representation and verifying the electrical properties of the circuit, the design is suitable for input to the *physical design* step. Physical design involves the placement and routing of the netlist on the actual chip image, the creation of a suitable power and ground grid, and the generation of the clock tree.

Each of the above synthesis steps, that is, high-level, logic and physical design, has multiple optimization objectives and seeks to find a trade-off among optimizing the area, delay, power consumption, noise, testability and yield, among others. While minimizing area lowers the manufacturing cost of a chip thus yielding higher profit margins, it often decreases the speed of the circuit. On the other hand, delay minimization leads to faster chips, which is essential in high-performance applications. Similarly, faster circuits typically have increased power consumption, which is a crucial factor in the application of portable computers and other hand-held devices. This

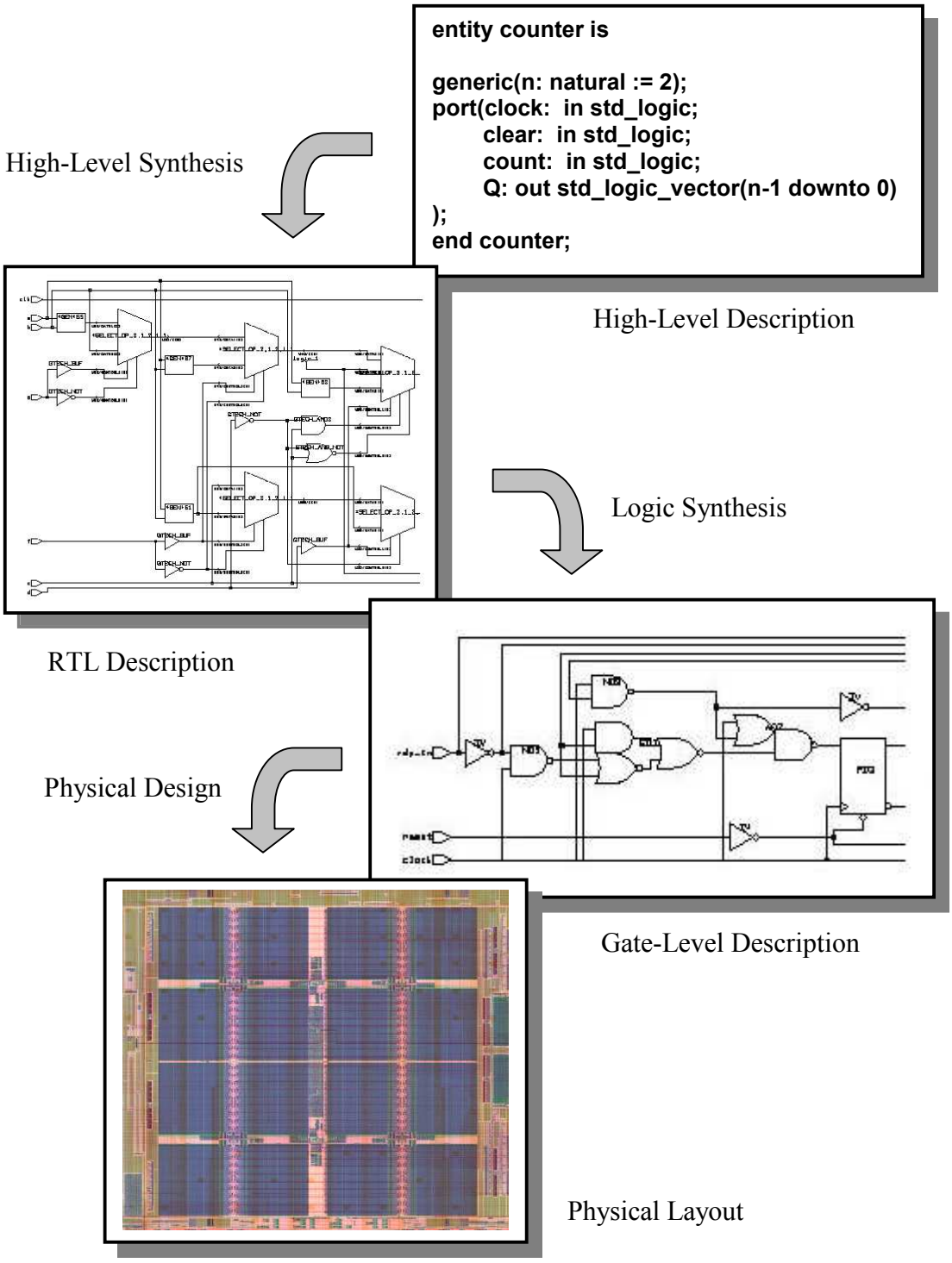


Figure 1.1: Typical synthesis design flow

behavior is shown in figure 1.2, depicting the tradeoff curve of Area A and delay t. Typically, the area-delay tradeoff curve is a convex function. The shaded area, which is

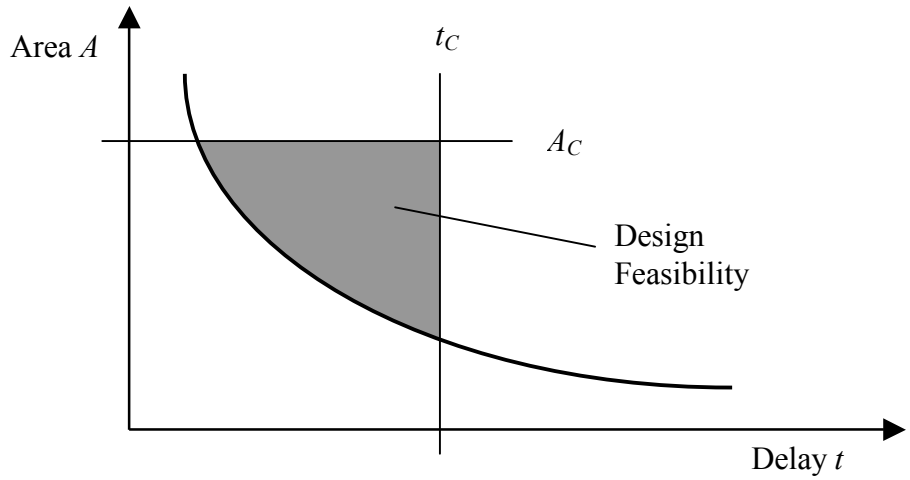


Figure 1.2: Area-delay tradeoff curve

bound by the area and delay constraints A_C and t_C, respectively, reflects designs that are feasible. On the other hand, extreme optimization of delay results in a steep area penalty and vice versa.

In general, all of these design objectives interact in complex ways and can only be approximated with second order mathematical models. In addition, the non-incremental nature of most synthesis transforms, that is, the order of the individual optimization steps affects the final result. Furthermore, due to the separation of the individual synthesis steps, little information about the physical properties of the design, for example, is known at earlier optimization steps. In conclusion, optimization across the whole set of design objectives is a very difficult task due to the tremendously large space of potential solutions.

1.2 Motivation and Thesis Overview

The increasing complexity of microelectronic designs and the continuous development of smaller sized fabrication processes creates new challenges to modern design automation tools. One of the most important problems in the design of VLSI circuits is the interaction of logical and physical domains. Due to the development of ever finer-featured integrated circuits in the sub-micron area and steadily increasing complexity, interconnect delay plays a dominant role in the synthesis and optimization process. Figure 1.3 shows a comparison of the average gate and interconnect delays for different

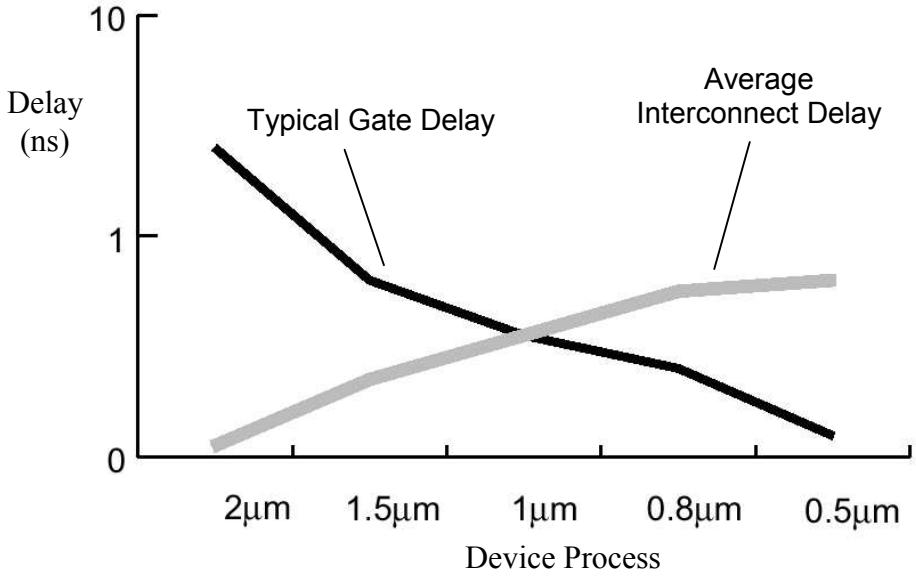


Figure 1.3: Comparison of average interconnect and gate delays for different design processes

device processes. At about 1 µm, the average gate and wire delays are about equal, but in the sub-micron area, interconnect delays are clearly dominating. As previously shown in Table 1.1, modern high-performance VLSI circuits are manufactured using design processes of 0.2 µm and below, known as *deep submicron* (DSM). While local wiring levels are relatively unaffected by traditional scaling, RC delay is dominated by global interconnect. As a result, the traditional separation of logic and physical design proves to be disadvantageous as the cost functions employed during logic synthesis become increasingly inaccurate during later design stages.

To solve these challenges, a tight coupling of logical and physical design must be developed. In the traditionally separate steps of the design flow, cost functions employed during logic synthesis do not reflect the actual physical properties of the design, particularly geometrical and structural information. While the problem of integrating synthesis and placement has been addressed extensively in the recent past, all existing solutions employ only trivial logic synthesis transforms late in the design flow, during physical design. These mainly local synthesis transformations attempt to reverse sub-optimal decisions made much earlier during logic-independent minimization and technology mapping, based on inaccurate cost functions, namely literal count and gate-based path delay. Instead of finding a good global solution incorporating logical and physical cost functions from the very beginning, the designers

try to fix the critical problems that appear locally at the very end of the design process. In addition, the increasing complexity of designs leads to infeasible turn-around times, often forcing the designer to partition a circuit which results in a more complex and mostly manual design flow and yields sub-optimal results compared to a flat approach.

To solve these problems, we have developed a design flow that employs physical and structural design information early during logic independent optimization. Specifically, we propose different solutions for data and control logic and show how to combine both methods to efficiently reach design closure.

Most modern designs are characterized by a large amount of datapath logic using bit-wise parallelism to achieve high performance. Such circuits contain a high degree of regularity due to the application of identical bit-operations across the width of the datapath. This important property can be used to obtain high-density layouts with short wiring length and improved timing during placement and routing. In addition, functional regularity can be used to speed up the optimization process by reusing identical parts. However, conventional logic synthesis is generally unable to use any notion of regularity. It is widely acknowledged that generic logic synthesis destroys a substantial amount of structural regularity, particularly during logic minimization and technology mapping. We have addressed this problem and present a regularity driven design flow that identifies and preserves regular structures during logic synthesis, and applies the extracted regularity during placement.

For the optimization of non-regular logic, that is, mainly control logic, we present layout-aware logic synthesis algorithms that make use of physical information already very early in the design process, during logic minimization and technology mapping, a major difference from existing placement driven synthesis solutions. Specifically, we create an initial placement of the non-optimized technology-independent gate-level netlist, and apply the gathered layout information during logic minimization and technology mapping. We utilize the geometric location of the objects in the netlist, for example the direction and distance of signal origins, and the estimated wiring length for improved functional decomposition of the network and to find better matches during technology mapping. Contrary to existing physical synthesis solutions, we do not try to fix problems at the end of the design process, but instead change the structure of the netlist during logic synthesis to reflect the physical properties of the design. In addition, to handle the increasing complexity that arises in the physical synthesis process, we employ clustering and partitioning algorithms which group the netlist based on its connectivity and functionality, for improved turn-around-time.

This thesis develops practical algorithms and provides a framework for the integration of logical, physical, and structural optimization in the design process.

The remainder of this dissertation is organized as follows:

Chapter 2 reviews some fundamentals of Boolean algebra and switching theory and introduces the essential notation being basic to the subject of this thesis.

Chapter 3 gives a brief overview of modern logic synthesis and physical design algorithms with an emphasis on the complex interactions between logic optimization and physical design, and exposes the problems of timing and design closure. Since this area is evolving at a rapid pace, we are able to give a much more up-to-date overview of the latest developments than most textbooks can do.

Chapter 4 introduces the notion of regularity and develops algorithms for the extraction of functional and structural regularity from arbitrary netlists. We show that regular structures can be used to create compact high-density layouts with short wire length and improved timing while simplifying placement and routing tasks. In addition, we reuse functionally identical parts to speed up logic minimization.

Chapter 5 shows that traditional logic synthesis destroys a significant amount of regularity through the application of mainly local transformations. Existing solutions either destroy regularity or preserve it by manually protecting and mapping datapath logic, resulting in unused optimization potential. We present a regularity-driven synthesis methodology that identifies, preserves and optimizes regular structures throughout the logic optimization process.

Chapter 6 develops algorithms for the integration of layout information into early logic synthesis. We create an initial placement of the technology-independent netlist using a quadratic placement algorithm and use the physical location of the individual cells to improve logic synthesis. To handle the increased number of objects at the technology-independent level, clustering and logic partitioning based on reconvergent region analysis is performed. We propose layout aware kernel extraction considering candidate cube divisor dependencies, decomposition and technology mapping algorithms to create a netlist with improved routability and timing.

Chapter 7 integrates the proposed layout and structure aware synthesis methodologies into a design flow. We show the extraction, preservation and placement of the regular (data) logic, and the layout aware optimization of the random (control) logic on a specific design example, a microprocessor core.

Chapter 8 concludes this thesis with a summary of the main results and describes future directions of this work.

2 Fundamentals

This chapter provides the notation, definitions and basic concepts needed throughout this dissertation. A short review of basic set and function theory is followed by an introduction to Boolean algebra and switching functions. Only topics necessary for the understanding of the following chapters are covered in this summary. For a more detailed introduction, we refer to the standard literature, e.g. [83], [49], [64], [16], and [88].

2.1 Sets, Relations, and Functions

We summarize some of the concepts and definitions of sets, relations, and functions, which are key to the understanding of Boolean algebra and function theory. For a more detailed introduction to the theory of sets, we refer the reader to the many excellent books providing comprehensive coverage of this subject, among those are [35] and [39].

2.1.1 Sets

Definition 2.1: A *set* is a collection of objects called *elements*, or *members*.

Sets can be ordered, denoted by parentheses, i.e. ( ), or unordered, denoted by braces, i.e. { }. We may also refer to an *ordered set* as a *vector*. By definition, each element of a set occurs exactly once. The *cardinality* of a set A, denoted by $|A|$, is the number of elements of the set. The empty set $\emptyset$ is the set with no elements, it is a subset of all sets. If a is an element of set A, we write $a \in A$. Similarly, *subset membership* or *inclusion* is denoted $a \subseteq A$. We briefly review the most common operations on sets from the same universe:

- *Intersection* $\cap$: The intersection of A and B, denoted $A \cap B$, is the set containing the elements of both sets A and B.
- *Complementation* $\bar{A}$: The complement of A in the universe U is the set of all elements U that are not elements of A.
- *Union* $\cup$: The union of A and B, denoted $A \cup B$ is the set containing the elements that are in either A or B.

We can also derive other operations from these, for example the operations of intersection and complementation allow us to define the *set difference*:

$$B - A = B \cap \bar{A}.$$

Given two sets A and B, we can define the *Cartesian product* $A \times B$ as follows:

$$A \times B = \{(a, b): a \in A \text{ and } b \in B\}.$$

Hence, the Cartesian product of A and B is the set of all *ordered pairs* of elements (a, b) such that the first element a is from set A, and the second element b is from set B. The definition of the Cartesian product is easily extended to n sets. It is the set of all ordered n-tuples taken from the n sets, denoted A^n. Furthermore, the *power set* of a set A, written 2^A, is defined as the set of all subsets of A:

$$2^A = \{B \subseteq A\}.$$

A simple and useful technique for the illustration of sets and set operations is the *Venn Diagram*. For further details we refer to the standard literature.

2.1.2 Relations

After introducing the concept of sets, we can define a *binary relation* as follows:

Definition 2.2: Given two sets A and B, a binary relation $\Re$ between two sets A and B, is a subset of the Cartesian product $A \times B$.

Higher-order relations are subsets of n-tuples from the Cartesian product of n sets. Since only binary relations are of importance in switching theory, we are only concerned with binary relations and simply refer to them as *relations*. Relations may exist among elements within a single set, as well as elements from distinct sets. We write $a \Re b$ if elements a and b are in relation $\Re$.

Below, we consider relations between elements of a single set A, thus $\Re \subseteq A \times A$. The following properties of a relation $\Re$ between the elements of set A can be defined:

- *Reflexivity*: Relation $\Re$ is *reflexive* if and only if, for every element $a \in A$, follows $a \Re a$.
- *Symmetry*: Relation $\Re$ is *symmetric* if and only if, for every pair $(a, b) \in \Re$, the pair (b, a) is also in $\Re$.
- *Antisymmetry*: Relation $\Re$ is *antisymmetric* if and only if the presence of both (a, b) and (b, a) in $\Re$ implies that $a = b$.
- *Transivity*: Relation $\Re$ is *transitive* if and only if the presence of (a, b) and (b, c) in $\Re$ implies the presence of (a, c).

A relation that is transitive, reflexive and symmetric is an *equivalence relation*. Relations that are transitive, reflexive and antisymmetric are called *partial orders*. Note that a binary relation cannot be both symmetric and antisymmetric unless it consists only of a set of pairs $\{(a, a)\}$. A set A in combination with a partial order is called a *partially ordered set*, or *poset*. Posets are often visualized using *Hasse diagrams*, as described in the standard textbooks.

A relation that is reflexive and symmetric, but not transitive is a *compatibility relation*. Compatibility relations play an important role in the minimization of incompletely specified finite state machines.

An equivalence relation partitions a set A into disjoint subsets A_i: $\bigcup A_i = A$. The subsets A_i induced by an equivalence relation are called *equivalence classes*. Further, given a subset A_i of A, an element $a \in A$ is an *upper bound* of A_i if and only if, for every $b \in A_i$ $b \Re a$. Similarly, it is called a *lower bound* if and only if, for every $b \in A_i$, $a \Re b$.

Having introduced posets and upper and lower bounds, we can define a lattice:

Definition 2.3: A *lattice* is a poset where every pair of elements has a unique greatest lower bound and a unique lowest upper bound.

The greatest lower bound of elements a and b of a lattice is also called the *meet* of a and b, denoted $a \cdot b$. Similarly, the lowest upper bound is called the *join* of a and b, denoted $a + b$. $(\cdot)$ and $(+)$ are called the *meet* and *joint operator*, respectively.

Consequently, all finite lattices have a greatest element, denoted **1**, and a least element **0**, where $\mathbf{1} \in A$ and $\mathbf{0} \in A$. Below, we define the properties of *complementation* and *distributivity*.

A lattice is *complemented* if and only if, for each element $a \in A$ there exists a unique element $\bar{a}$ such that $a \cdot \bar{a} = 0$ and $a + \bar{a} = 1$.

A lattice is *distributive* if the distributive properties are fulfilled, i.e.

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c), \text{ and}$$

$$a + (b \cdot c) = (a + b) \cdot (a + c).$$

We will later introduce the Boolean algebra as a complemented distributive lattice.

2.1.3 Functions

We define a *function* as a special binary relation:

Definition 2.4: A *function f* is a mapping from set A to set B, denoted $f : A \mapsto B$, such that every element $a \in A$ is associated with exactly one element $b \in B$.

The *range* of a function f is the subset of B that f can assume. The sets A and B of a function are called *domain* and *co-domain*, respectively. Furthermore, each element from the domain is related to an element of the range of the function in exactly one pair, contrary to an unrestricted relation.

2.2 Boolean Algebra

Boolean algebra is the fundamental basis underlying the analysis of digital circuits. Shannon [86] introduced the concept of describing the behavior of switching circuits by a two-valued Boolean algebra in the 1930s. In this section, we give a brief introduction to Boolean algebra. For more detailed coverage, we refer the reader to the extensive literature on this subject, e.g. [29], [83], and [16].

We introduce a *Boolean algebra* as a special lattice. The meet and joint operator are also referred to as *AND* (*conjunction*) and *OR* (*disjunction*), respectively.

Definition 2.5: A lattice is called a *Boolean algebra* if it fulfills the conditions of complementation and distributivity.

A Boolean algebra has the following properties:

- *Idempotency*: $a \cdot a = a$ and $a + a = a$
- *Commutativity*: $a \cdot b = b \cdot a$ and $a + b = b + a$
- *Associativity*: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ and $a + (b + c) = (a + b) + c$

- *Absorption*: $a + a \cdot b = a \cdot (a + b) = a$
- *Distributivity*: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $a + (b \cdot c) = (a + b) \cdot (a + c)$
- *Complement*: $a \cdot \bar{a} = 0$ and $a + \bar{a} = 1$

It is also possible to define a *Boolean ring*. This is the original formulation in the work of Boole [9], published in the 19th century. The Boolean ring is formed by the antivalence (EXOR) and the conjunction (AND) operation and has the following properties:

- Identity: $0 \oplus a = a$ and $a \cdot 1 = a$
- Null element: $a \cdot 0 = 0$
- Self Inversion: $a \oplus a = 0$
- Complement: $a \oplus \bar{a} = 1$ and $a \cdot \bar{a} = 0$
- Commutativity: $a \oplus b = b \oplus a$ and $a \cdot b = b \cdot a$
- Associativity: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

Please note that the various definitions of Boolean algebra are equivalent and can be transformed into one another.

In chapter 4 of this thesis, we formulate a framework for the extraction of functionally regular parts of a circuit based on the operations of the Boolean ring, AND and XOR.

2.3 Graph Theory

In the following section, we formally define a graph and introduce some properties and operations on graphs. For a more thorough description of graph theory, we recommend [40] and [8], for example.

Definition 2.6: A *graph* $G = (V, E)$ is a pair where V is a set of *vertices* and $E \subset V \times V$ is a set of *edges*.

The elements of the set V are called *vertices* whereas those of set E are called *edges* of the graph. In a *directed graph*, the edges are ordered pairs of vertices $(v_i, v_j) \in V \times V$. v_i is called the *immediate predecessor of* v_j, and v_j is the *immediate successor of* v_i. In an undirected graph the edges are unordered pairs $\{v_i, v_j\}$.

Two vertices that are joined by an edge are said to be *adjacent*, as are two edges that meet at a vertex. If two vertices are not joined by an edge, we say they are *nonadjacent*.

The *degree* of a vertex is the number of edges incident to it. In a directed graph, the *outdegree*, or *fanout*, of a node refers to the number of its immediate successors, whereas the *indegree*, or *fanin*, is defined by the number of its immediate predecessors. A node with indegree 0 is called a *source*, and a node with outdegree 0 is referred to as a *sink*.

Definition 2.7: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a one-to-one and onto function $f : V_1 \mapsto V_2$ such that $(v_1, v_2) \in E_1$ if, and only if, $(f(v_1), f(v_2)) \in E_2$. That is, f preserves adjacency and nonadjacency. The function f is called an *isomorphism*.

A *subgraph* G' (V', E') of graph G (V, E) is a graph whose vertex and edge sets are contained in the respective vertex and edge sets of G, i.e. if and only if $V' \subseteq V$ and $E' \subseteq E$. A *clique* of a graph is a complete subgraph. It is *maximal* if it is not contained in any other clique.

A *walk* is an alternating sequence of vertices and edges. A *trail* is a walk with distinct edges, and a *path* is a trail with distinct vertices. A *cycle* is a path such that the two end-point vertices coincide.

A *directed acyclic graph (DAG)* is a directed graph without cycles. One of the applications of DAGs is for example the representation of combinatorial logic. A DAG is called *rooted* if it has one distinguished node, called *root*, which does not have any predecessors.

A directed rooted tree is a DAG such that all other nodes have exactly one immediate predecessor. The immediate predecessor of a node v in a directed rooted tree is called a *parent* of v, and the immediate successors are called *children* of v. Nodes without a successor are called *leaves* of the tree.

A DAG represents a partially ordered set. A *topological order* is an ordering of the vertices of a DAG such that every edge (v_i, v_j) satisfies $i < j$. An ordering satisfying $j < i$ is called *decreasing topological order*.

Directed and undirected graphs can be *labeled*. Labels can be associated with vertices and/or with edges, i.e. the graph can be *vertex labeled* and/or *edge labeled*.

Definition 2.8: A labeled *graph* $G = (V, E, L)$ consists of a set V of *vertices*, a set $E \subset V \times V$ of *edges* and a set L of *labels*.

The label set of G is denoted $L(G)$ which is comprised of the label set on the edges $L(E)$ and/or the label set on the vertices $L(V)$ of graph G. For a vertex v and edge e, we denote $l(v)$ the label of v, and $l(e)$ the label of an edge e. Based on the definition of isomorphism on graphs, we can extend this relation to labeled graphs.

Definition 2.9: Two labeled graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$ are *strongly isomorphic* if there exists a one-to-one and onto function $f : V_1 \mapsto V_2$ such that $(v_1, v_2) \in E_1$ if, and only if, $(f(v_1), f(v_2)) \in E_2$, and $l(v_1) = l(f(v_1))$, $l(v_2) = l(f(v_2))$, and $l(v_1, v_2) = l(f(v_1), f(v_2))$.

Therefore, *strong isomorphism* preserves adjacencies and non-adjacencies of the graphs, and establishes a one-to-one mapping of the labels of corresponding edges and vertices of the graphs.

2.4 Boolean Functions and Switching Theory

This section reviews the basics of Boolean functions, their representations and implementation as switching circuit. Fundamentally, logic synthesis is the manipulation of sets of Boolean functions such that it is suitable for the implementation as a circuit. For a more comprehensive look at this subject, we refer to the standard literature, and particularly recommend [42], [68], and [69].

2.4.1 Boolean Functions

A *completely specified* Boolean function is a mapping f between an n-dimensional domain B^n and range B, that is $f : B^n \mapsto B$. Furthermore, we speak of an m-ary Boolean function if each point in the space is mapped to m elements of B, i.e. a mapping f, such that $f : B^n \mapsto B^m$. If we consider the two-valued set $B = \{0, 1\}$, also known as *switching algebra*, we can define *switching functions* $f : \{0, 1\}^n \mapsto \{0, 1\}$.

A variable of a Boolean function in complemented or uncomplemented form is called a *literal*. As an example, a and $\bar{a}$ are expressions of the same variable, however, they are two different literals.

An *incompletely specified* Boolean function is defined over a subset of B^n, denoted $f : B^n \mapsto \{0, 1, *\}$. The symbol $*$ represents a *don't care* condition and specifies the points where the function is undefined. For each output, the subsets of the domain for which the function takes the values 0, 1 and $*$ are called the *OFF set*, *ON set* and *don't care* or *DC set*, respectively.

A *Boolean relation* is a generalization of a Boolean function, where a point in the domain is associated with more than one point in the co-domain. This plays an important role in multi-level logic optimization.

We conclude that Boolean functions form the basis for describing the behavior of switching circuits. All methods for optimization, verification or analysis of a switching circuit rely on manipulating Boolean functions.

2.4.2 Operations on Boolean Functions

Let $f(x_1, x_2, ..., x_n)$ be a Boolean function of n variables. The set $\{x_1, x_2, ..., x_n\}$ is called the *support* of function f.

Definition 2.10: The *cofactors* of a function f are defined as

$$f_{\bar{x}_1} = f_{|x_1=0} = f(0, x_2, ..., x_n), \text{ and}$$
$$f_{x_1} = f_{|x_1=1} = f(1, x_2, ..., x_n).$$

They denote the function f restricted to the subdomain in which x_1 takes the value 0, or 1, respectively. The functions $f_{|x_1=0}(x_2, ..., x_n)$ and $f_{|x_1=1}(x_2, ..., x_n)$ are called the nega*tive* and *positive cofactors*.

Using the cofactor notation, the *Shannon expansion* of a function f is defined as:

$$f = \bar{x}_1 f_{\bar{x}_1} + x_1 f_{x_1}.$$

This expansion was originally introduced by Boole in [9] and as such is also referred to as *Boole's expansion*.

Using recursive expansion, any function f can be expressed as a *sum of products* (SOP) of n literals, also called the *minterms* of the function. Alternatively, it can be represented as a *product of sums*, commonly known as *maxterms*. A product term is also called a *cube*.

Definition 2.11: A *cube* is a set of literals such that $a \in C$ implies $\bar{a} \notin C$. The notation $|C|$ is used for the number of literals in cube C.

x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

Table 2.1: Example of a truth table

As a result of complete expansion, a Boolean function can be described by the set of its minterms. As such, operations and relations on Boolean functions can be viewed as operations and relations on their minterm sets. A Boolean function is completely described by a disjunction of all its ON-set minterms, we speak of the *disjunctive normal form (DNF)*, or by a conjunction of all of its OFF-maxterms, called the *conjunctive normal form (CNF)*. These normal forms are unique representations of a Boolean function. Therefore, two Boolean functions are equivalent, if and only if their normal forms contain the same minterms or maxterms, respectively. The unique representation of a Boolean function is also called *canonical*.

2.4.3 Representations of Boolean Functions

In general, the algorithmic solution of a given problem and the representation of the involved Boolean functions are closely related. As such, a suitable representation of Boolean functions is essential for the development of efficient algorithms for the optimization of switching circuits.

A Boolean function can be represented in several ways, which can be classified into *tabular representations*, *Boolean expressions* and *binary decision diagrams* (BDDs).

2.4.3.1 Tabular Expressions

A Boolean function can be represented using two-dimensional tables that are partitioned into two parts, corresponding to the inputs and outputs of the function. The simplest tabular form is a *truth table*. A truth table gives a complete listing of all points in the Boolean input space and the corresponding values of the outputs. An example of a truth

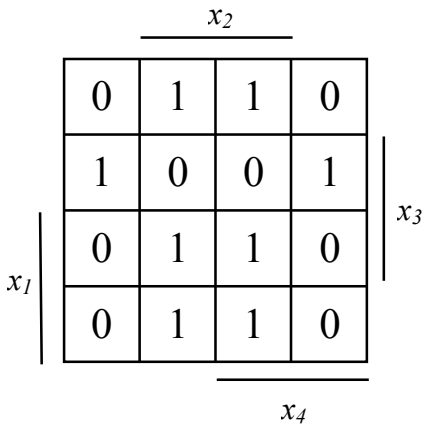


Figure 2.1: Karnaugh map of a Boolean function with four input variables

table is shown in Table 2.1. The input part is the set of all row vectors in B^n, while the output part is the set of corresponding vectors $\{0, 1, *\}^m$.

Since the size of a truth table is exponential in the number of inputs, this representation is only useful for small functions.

Another way to represent a Boolean function is by means of a *Karnaugh map*. An example is shown in figure 2.1. For further reading, we refer to the standard literature.

2.4.3.2 Boolean Expressions

Boolean functions can be represented by expressions of literals connected by the + (disjunction) and • (conjunction) operators, called *Boolean expressions*. We generally distinguish between *two-level* and *multi-level* expressions. Standard two-level forms are sum of products and product of sums. For example, the sum of products form of function *y* in Table 2.1 is:

$$y = x_1 \cdot x_2 + x_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot \overline{x}_2 \cdot x_3$$

Multi-Level forms contain an arbitrary nesting of Boolean operators by means of parentheses. A particular multi-level form is the *factored form*. A factored form is one and only one of the following: a literal, a sum of factored forms, or a product of factored forms. Therefore, factored forms contain sum of products and product of sums.

2.4.3.3 Binary Decision Diagrams

A *binary decision diagram (BDD)* is a representation of a Boolean function as a graph. BDDs were first proposed by Lee [63], and later by Akers [1]. A BDD is a directed acyclic graph (DAG) where each node has exactly two children, representing the decisions of the evaluation of the associated variable. Figure 2.2 shows an example of a BDD, the representation of the function $y = a \cdot (b + c)$. The sinks of the graph denote the two possible binary values of the function, TRUE (1) and FALSE (0).

A special type of BDD is the *ordered binary decision diagram (OBDD)*, first introduced by Bryant [17]. The use of OBBDs is motivated for the following reasons. First of all, OBDDs can be transformed into canonical forms that uniquely characterize the function. This is of particular importance in formal verification, for example to check the equivalence of two Boolean functions. Second, operations on OBDDs are only of polynomial complexity of their size, i.e. the cardinality of its vertex set. This is particularly true for the satisfiability problem but does not hold true for all functions. As an example, functions representing arithmetic multiplication have OBBDs of exponential size, regardless of the variable order.

A special type of OBBD is the *reduced ordered binary decision diagram (ROBDD).* An ROBDD is a canonical form of a Boolean function.

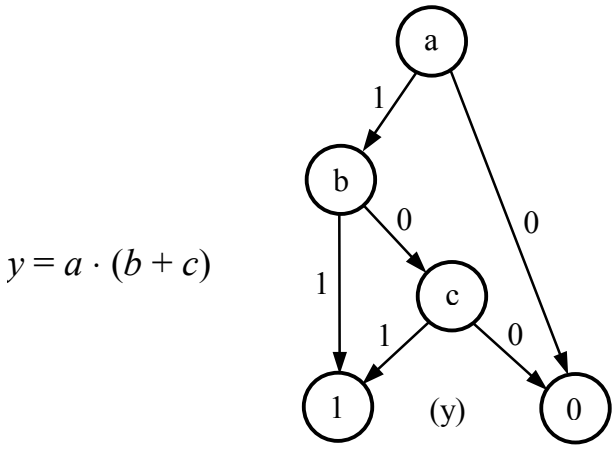


Figure 2.2: Representation of a Boolean equation as BDD

2.4.4 Algebraic and Boolean Division

In this section, we introduce the notion of algebraic division of representations of a Boolean function. Division is an essential operation in the minimization of a logic network. It represents the basis of factoring and decomposition algorithms, described in chapter 6.

An *algebraic expression* $F = \{C_i\}$ is an expression in which no cube contains another cube, that is $C_i \subseteq C_j$ for $i \neq j$. For example, expression $a + bc$ is algebraic, but $a + ab$ is not because $\{a\} \subseteq \{a, b\}$.

2.4.4.1 Division

Since a Boolean algebra has no additive or multiplicative inverses, there can be no division operation. However, it is possible to define operations which, when given functions f and p, find functions q and r such that $f = p \cdot q + r$ holds true. Every such operation is equivalent to the division operation and is therefore called *division* of f by p generating quotient q and remainder r.

Clearly, such a division operation is not unique. For a given division operation, the resulting q and r depend on the particular representation of f and p. It was shown in [11] that for any Boolean function f and g satisfying $f \subset g$, f can be written as $f = g \cdot h$. In this case, we call g a *Boolean factor* of f. If $f \cap g \neq \emptyset$ holds true, f can be written as $f = g \cdot h + r$. Here we call g a *Boolean divisor* of f. For any logic function f there are many more Boolean divisors and factors than algebraic ones. This makes finding a good Boolean decomposition of a function f a difficult problem.

We now introduce two classes of division operations that work on sum of products (SOP) forms. Division of SOP forms is based on the concept of the *product*. We define a product based on considering a cube as a set of literals, and an SOP form as a set of cubes.

Definition 2.12: The *product* of two cubes C and D is a cube defined by

$$CD = \begin{cases} \emptyset & \text{if } \exists x \,(\, x \in C \cup D \text{ and } \overline{x} \in C \cup D \,) \\ C \cup D & \text{otherwise.} \end{cases}$$

Definition 2.13: The *product* of two SOP expressions F and G is an SOP expression defined by

$$FG = F \times G = \{CD \mid C \in F \text{ and } D \in G \text{ and } CD \neq \emptyset\}.$$

Notice that $CD = \emptyset$ if and only if $C \cup D$ contains both a literal and its complement.

Definition 2.14: *FG* is an *algebraic product* if *F* and *G* have disjoint support, otherwise *FG* is a *Boolean product*.

Now we can define the division operation on SOP forms as follows:

Definition 2.15: An operation *OP* is called *division* if, given two SOP expressions *F* and *P*, it generates SOP expressions *Q* and *R* ($\langle Q, R\rangle = OP(F, P)$) such that $F = PQ + R$.

If *PQ* is an algebraic product, *OP* is called an *algebraic division*, otherwise *PQ* is a Boolean product and *OP* is therefore called a *Boolean division*.

We now briefly discuss the concept of *weak division* as a specific example of algebraic division.

Definition 2.16: Given two algebraic expressions *F* and *P*, a division is *called weak division* if:

- it generates *Q* and *R* such that *PQ* is an algebraic product;
- *R* has a minimum number of cubes;
- $PQ + R$ and *F* have the same set of cubes.

The motivation lies in the fact that, given the expressions *F* and *P*, *Q* and *R* generated by weak division are unique.

2.4.4.2 Kernels and Co-Kernels

The notion of a kernel was introduced in [12] to provide a framework for finding common subexpressions of two or more expressions. Identification of common subexpressions is an essential step in the decomposition and factoring of Boolean functions.

An expression is called *cube-free* if it has more than one cube and no other cube divides the expression evenly, that is $\neg\exists C$ such that $F = QC$ (no remainder), and *C* is a cube. For example, $ab + d$ is cube-free, but $bc + bd$ and *abc* are not.

The *primary divisors* of an algebraic expression *F* are the set of expressions

$$D(F) = \{F \,/\, c \mid c \text{ is a cube}\}.$$

We can now formally define the notion of a *kernel*.

Kernel	Co-Kernel
$a + b$	cd
$d + g$	ac
$ad + bd + e + ag$	c
$acd + bcd + ce + acg$	1

$$F = acd + bcd + ce + acg$$

Table 2.2: Kernels and Co-Kernels of an expression

Definition 2.17: The *kernels* of an expression F are the set of expressions

$$K(F) = \{g \mid g \in D(F) \text{ and } g \text{ is cube-free}\}.$$

Thus, the kernels of an expression F are the cube-free primary divisors of F.

A cube c used to obtain the kernel $K = F / c$ is called a *co-kernel* or *base* of K, and $C(F)$ is used to denote the set of co-kernels of F. As an example, table 2.2 shows the set of kernels and co-kernels for the function $F = acd + bcd + ce + acg$.

For further reading on algebraic and Boolean division, and algorithms for the computation of kernels, we refer the interested reader to [11], [12], and [13].

2.5 Boolean Networks

A *Boolean network*, also called a *logic network*, is a directed acyclic graph. It is the basic data structure used by multi-level logic synthesis algorithms. Each node of the graph corresponds to a gate and an edge connecting two nodes corresponds to a connection between two gates.

Definition 2.18: A Boolean network B is an interconnection of n Boolean functions defined by a five-tuple (f, y, I, O, d^*), consisting of

- $f = (f_1, f_2, ..., f_n)$ – the completely specified logic functions representing the *gates* of the network,
- $y = (y_1, y_2, ..., y_n)$ – the logic variables that are in one-to-one correspondence with f representing the signals of the network,
- $I = (I_1, I_2, ..., I_p)$ – the p externally controllable signals representing the primary inputs (PIs) of the network,

- $O = (O_1, O_2, ..., O_s)$ – the s externally observable signals representing the primary outputs (POs) of the network, and
- $d^* = (d_1^*, d_2^*, ..., d_s^*)$ – the completely specified functions that specify the set of don't care minterms on the outputs.

A Boolean network describes a *combinational circuit*.

Definition 2.19: A *circuit C* consists of a set G of gates implementing logic functions f, a set N of *nets*, and *pins* $P \subset G \times N$.

Thus, a *net* $n \in N$ is an interconnection of two or more pins of a circuit. The gates, which are elements of a library, are dependent on a particular technology library. A typical library consists of AND, OR, NOR, NAND, XOR, and XNOR gates. It is also possible to completely describe a Boolean network only using NAND or NOR gates, however, it often contains XOR gates as the implementation of the EXOR operator with NAND or NOR gates is exponential in the number of inputs.

A *subnetwork B'* of a Boolean network B is a subgraph of B. Furthermore, two Boolean networks B_1 and B_2 are said to have the same structure, written as $B_1 \equiv B_2$ if and only if the underlying graph representations are isomorphic.

3 Logical and Physical Design: An Overview

In this chapter, we give a general overview of logical and physical design algorithms with particular emphasis on their complex interactions and attempt to familiarize the reader with algorithms basic to the understanding of later chapters of this dissertation. We start by outlining the motivation for combining logical and physical design, followed by a brief introduction to traditional logic synthesis algorithms, and an overview of placement techniques. Thereafter, we analyze the current state-of-the-art showing various attempts to combine logical and physical design, commonly known as *physical synthesis*. Finally, we conclude the motivation of this work, i.e. the need for synthesis algorithms that consider structural and physical design properties.

Since this subject is evolving at a rapid pace, most of the literature is quickly out of date. We refer, however, to two excellent books. The first [87] gives a detailed overview of algorithms for physical design, the latter [41] is a general survey of latest trends and developments in logic synthesis and verification.

3.1 Introduction

Physical design is the process of producing a fabrication-ready mask description of the complete layout from a gate-level description while trying to satisfy a number of constraints, e.g. timing, area, blockages (regions where no cells can be placed), power consumption, yield, and technology constraints such as electromigration and output load, among others. As an example of a modern chip design, figure 3.1 shows the physical layout of a 64-bit microprocessor with 47 million transistors and a clock cycle of 1GHz. The layout, typical for complex designs, consists of a mix of standard cells, macro blocks and RAMs. Figure 3.2 shows a magnification of the actual wiring layers in the final chip in deep submicron technology.

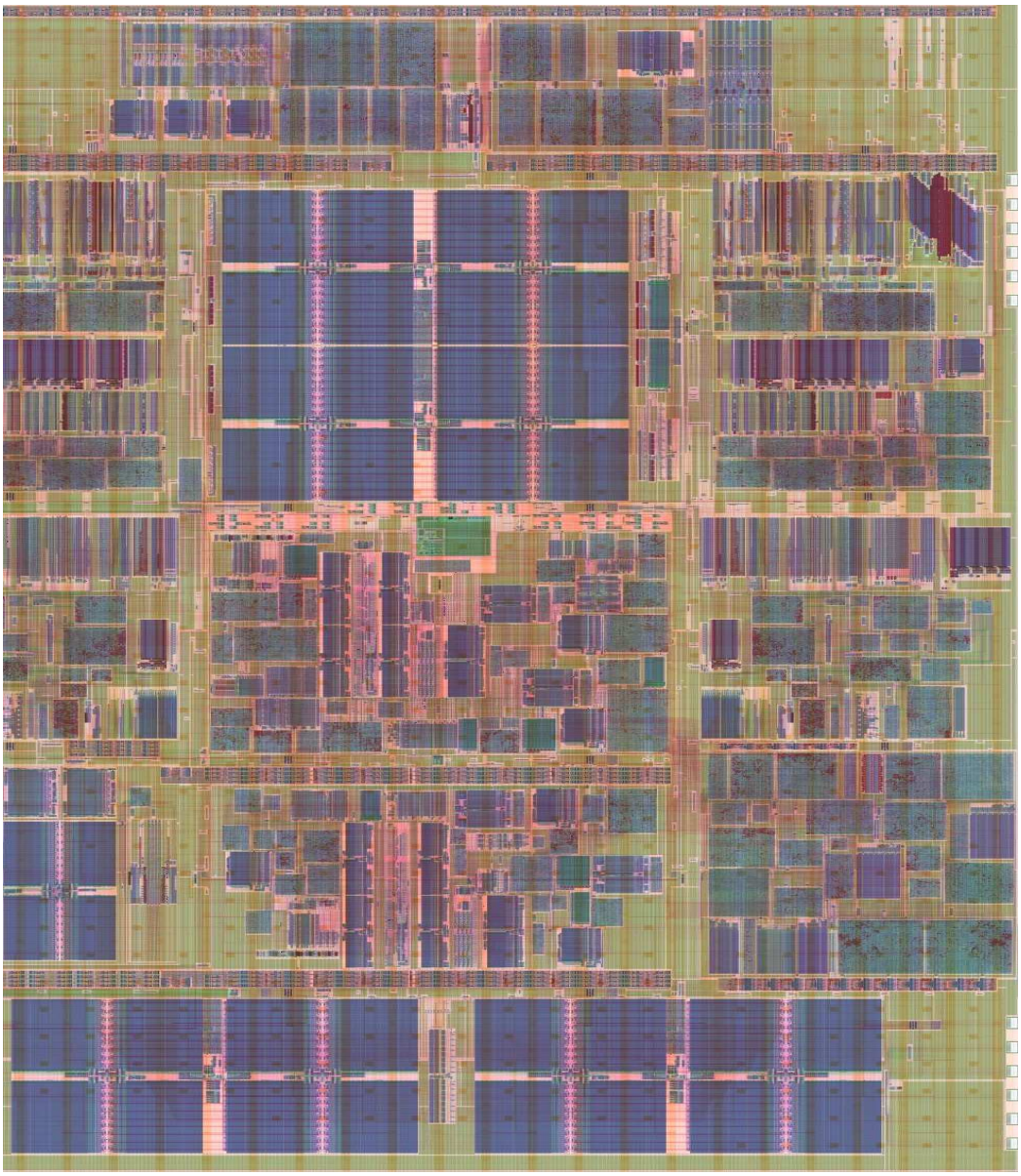


Figure 3.1: Layout of a complex microprocessor

As shown in chapter 1, interconnect delay has a dominating effect on circuit timing at feature sizes below 1µm. While local wiring levels are relatively unaffected by traditional scaling, RC delay, is dominated by global interconnect. Consequently, at about 0.2 µm and beyond, also known as *deep submicron (DSM)*, the traditional separation of logic synthesis and physical design is unable to produce satisfying results. As such, it is no longer possible to ignore the impact of *interconnect delay* on timing because it dominates gate delay. In addition, the increasing complexity of designs

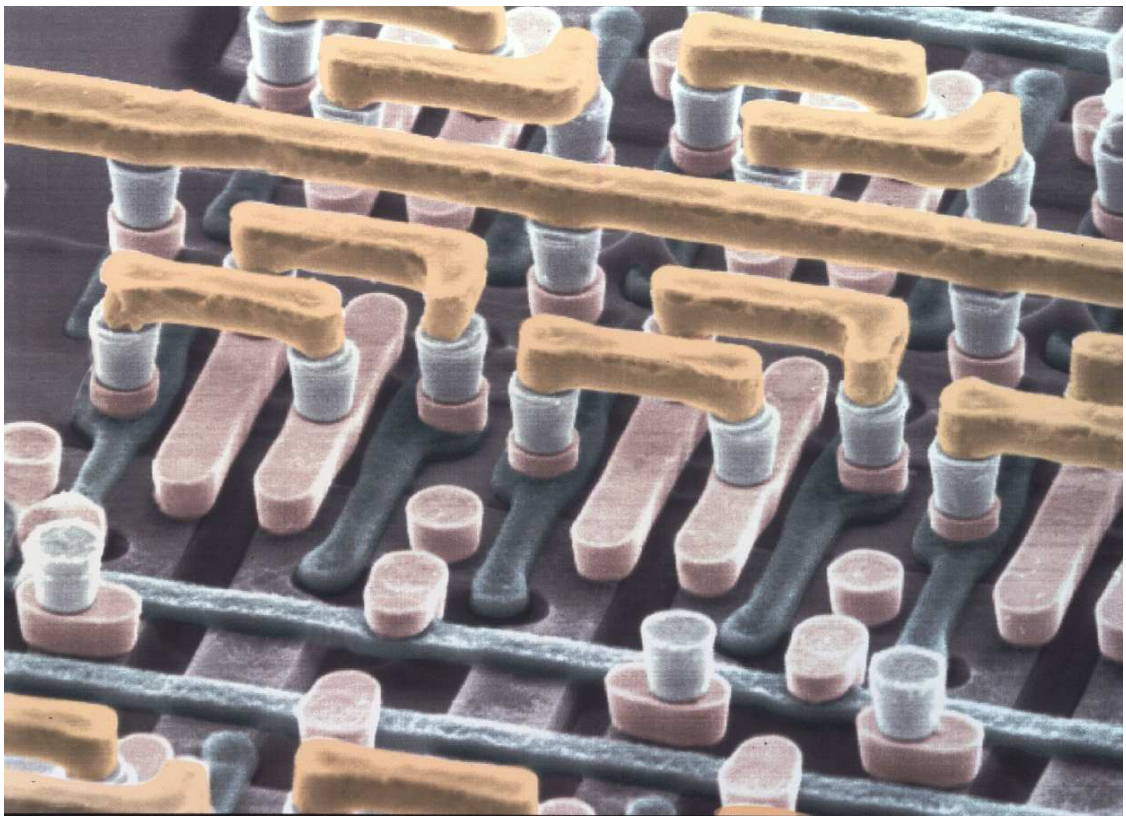


Figure 3.2: Wiring layers of an integrated chip

makes the process of routing more challenging, and thus, the minimization of *wiring congestion* becomes an important issue. Furthermore, finer-featured design processes, higher clock frequencies and lower voltages produce *signal integrity* problems.

In general, this problem is known as ***design closure***. It is the problem of satisfying all given design constraints, i.e. timing, area, congestion, power, and signal integrity, to create a fabrication-ready layout. Of particular importance in reaching design sign-off is *timing closure*.

Timing closure is the problem of satisfying the given timing constraints of a design. Until recently, delay estimation could be accurately done with static timing analysis based on only the gate delays. By ignoring net capacitance, delay could be optimized during logic synthesis, before handing off the design to physical implementation. As net capacitance is becoming a dominating factor, an accurate timing estimation without knowledge of the geometric location of the individual objects of the netlist is no longer possible. One attempt to account for interconnect delays is the use of

a statistical *wire load model*. While such a wire load model is a good prediction for the *average* wire load, the deviation is so large that the critical delays, i.e. the longest paths in a design often differ significantly. In addition, coupling capacitances between interconnects of the same layer become more dominant with every new process technology. As a result, net capacitance cannot accurately be determined without knowledge of its route and that of its neighboring nets.

Another problem is the steadily increasing complexity of designs, containing tens of millions of gates. To handle the sheer number of objects stressing memory and computational resources, designs are often divided into several blocks, creating a hierarchical design flow. Hierarchical flows create new problems, for example handling timing constraints between the chip and the block level, hierarchical design verification, and congestion estimation between blocks, among others.

We conclude that logical and physical aspects are closely related in many ways and can no longer be dealt with separately. As a result, the simultaneous optimization of logical and physical aspects with the goal of reaching design closure has become one of the biggest challenges in the design of VLSI circuits. In addition, as designs keep getting bigger and bigger, handling the complexity of designs is an important problem.

3.2 Logic Synthesis

In this section, we give a short overview of logic optimization techniques. Logic Synthesis generally consists of three distinguished steps, technology-independent minimization, technology mapping and technology-dependent optimization. We first outline the basics of two-level and multi-level logic minimization followed by a brief look at technology-based transformations.

3.2.1 Logic Minimization

Two-Level Logic Synthesis: The first attempt to minimize a logic network resulted in *two-level logic minimization* algorithms. Two levels of logic are the minimum required to implement an arbitrary Boolean function. Here, the first level consists of AND, and the second level of OR gates. In other words, two-level minimization attempts to find a minimal sum-of-products form of a Boolean network, thus seeking a circuit realization with the smallest area.

Quine [78, 79] proposed the first solution to this problem in the 1950s. He realized that a minimal SOP form must always consist of a sum of *prime implicants*, known as *Quine's Prime Implicant Theorem*. An *implicant* of a function f is a product term p that is included in the function f ($p \leq f$). A *prime implicant* of f is an implicant of f that is not included in any other implicant of f. Later, this method was improved by McCluskey [70] and has since become known as the Quine-McCluskey method. The drawback of this procedure is its exponential complexity in both space and time, which limits its application to relatively small functions. The advantage of two-level forms is that they can be directly implemented as a circuit using programmable logic arrays (PLAs). The general use of two-level synthesis however, is limited by the fact that the synthesis of large functions in two levels is computationally infeasible, and by technology-imposed limits on the maximum fan-in and fan-out of logic gates. Furthermore, multi-level realizations are often smaller and faster than their appropriate two level counterparts. Despite these shortcomings, two-level synthesis is sometimes used as a step in multi-level minimization.

Multi-Level Logic Synthesis: Multi-level circuits are of great practical significance because they represent the majority of designed circuits. This is particularly true for circuits that are implemented with standard cells or gate arrays. Multi-level circuits tend to be smaller, faster and often consume less power than two-level circuits. For some functions, for example certain arithmetic ones, two-level representations require an exponential number of product terms, whereas an appropriate multi-level representation only requires a linear or quadratic amount.

A fundamental concept in multi-level synthesis is that of *functional decomposition*. The decomposition of a Boolean function f is its expression in terms of a set of other, ideally simpler functions $g_1, \ldots, g_k$. Decomposition of a Boolean network refers to finding subexpressions common to one or more functions such that they can be implemented once and shared across the entire design. A function f has a non-trivial decomposition if it satisfies

$$f(x_1, \ldots, x_n) = h(g(x_1, \ldots, x_s), x_{s+1}, \ldots, x_n).$$

The concept of functional decomposition was introduced by Ashenhurst [4], and later improved on by Curtis [25], and Roth and Karp [82] to handle more complex decomposition forms. If carried to completion, these algorithms find all possible decompositions and will yield a circuit of strictly minimum cost. However, their computational complexity makes them infeasible for anything but small circuits.

The computational limitations of functional decomposition motivated the development of algorithms based on *algebraic factorization*. Factorization refers to representing a logic function in minimum (in terms of number of literals) factored form. Decomposition is similar to factorization except that each subexpression is formed as a new intermediate variable and substituted into the functions being decomposed. In [14], Brayton and McMullen proposed a fundamental theorem about finding algebraic common divisors using kernels:

Theorem 3.1: Two expressions F and G have non-trivial *common divisors* if and only if there exist kernels of F and G, $K_F \in K(F)$ and $K_G \in K(G)$ such that $| K_F \cap K_G | \geq 2$ (K_G and K_F have at least 2 terms in common).

Therefore, it is possible to find non-trivial common divisors, i.e. divisors having more than one cube, by computing the intersection among kernels of each expression. Rudell [84] proposed a formalization, called *rectangle covering*, to find intersections of kernels, common cubes, to do factorization and to find kernels of an expression.

Using kernels is an effective technique to reduce the search space for finding common divisors. However, in the worst case, the number of kernels can grow exponentially with the number of cubes in the expression. A very effective heuristic for the decomposition and factorization of Boolean expressions was proposed by Rajski and Vasudevamurthy [80, 81]. Their method only uses double-cube divisors and two-literal single cube divisors considered concurrently with their complements. Note that the complement of a two-literal single cube divisor is a double cube divisor. The set of double-cube divisors of an expression F, is defined as

$$D(F) = \{\{c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j)\} \mid c_i, c_j \in F, i \neq j\}.$$

It can be seen that the computational complexity of finding all double cube divisors is $O(n^2)$ where n is the number of cubes $c_1, c_2, \ldots c_n$ in F. It is obvious that the set of all double cube divisors is smaller than the set of all kernels. However, through repeated extraction, algebraic divisors of arbitrary size can be obtained. This decomposition technique using single and double cube divisors is the subject of our work based on layout driven decomposition and is described in more detail in chapter 6.

Other approaches to logic minimization of a network are based on *local transformations*. Early local optimization methods perform rule-based transformations, which are a set of ad hoc rules that are applied iteratively to patterns found in the network [26]. Another set of algorithms is based on the *optimization with don't cares*

[27]. Closely related are also other optimization methods such as redundancy removal [20], transduction [71] and global flow analysis [5, 6].

3.2.2 Technology Mapping

Technology Mapping transforms a technology-independent logic network into gates implemented in a specific technology library. It has a major impact on the global structure of the technology mapped logic, its delay and area characteristics. Conventional technology mapping consists of three phases: *decomposition*, *pattern matching* and *covering*. During decomposition, the technology-independent circuit is partitioned into primitive cells (e.g. two-input NANDs). This step produces a relatively simple structure of the circuit and improves the subsequent process of finding patterns in the network. Thereafter, a pattern match performs analysis on the circuit and library and determines a set of structural or functional matches for all nodes in the circuit. In the final phase, the best possible matches based on certain cost functions, usually area or delay, are selected such that every node in the circuit is covered at least once. This final set of matches consists of instantiations of cells of the target library. Different objective functions, namely delay, area, power and reliability motivate the use of different algorithms.

Many of the early technology mapping algorithms were based on *tree covering*, despite the fact that all practical circuits are represented by DAGs. Therefore, a non-trivial decomposition of the DAG into a forest of trees is necessary. Tree mapping algorithms gained popularity due to their optimality for various cost functions. A tree-based optimal area technology mapping algorithm is described in [46] by Keutzer, and was later extended by Rudell [84] to include a delay objective. Optimal delay mapping under the linear delay model was proposed in [95].

Early DAG mapping algorithms were based on heuristic transformations, e.g. [37, 10]. The introduction of load-independent delay models, e.g. the gain-based delay model, allows for optimal delay mapping algorithms on DAGs [92, 52]. Watanabe [98] introduced a technology mapping algorithm incorporating many different decomposition choices directly within the matching phase.

Exact technology mapping is inherently a difficult problem due to the many different cost functions that need to be satisfied. Current mapping algorithms cannot provide optimal solutions for minimum delay and area in the presence of complex design constraints, and in the absence of accurate realistic delay models. Area during

technology mapping is commonly calculated as the sum of the area of all technology cells, ignoring interconnect area. The same holds true for delay estimation, which is calculated by a static timing analysis tool using statistical wire-load models. As mentioned in the previous section, these models are good at predicting the average delay in a design, but are often far off at estimating path delays. In chapter 6, we introduce a layout driven technology mapping algorithm under the load-independent delay model for improved congestion and timing.

For a more detailed overview of technology mapping algorithms, we refer the interested reader to [41].

3.2.3 Technology-Dependent Optimization

Due to the inherent complexity of the mapping problem, heuristics are often used to achieve a trade-off between the various objective functions. These factors render the resulting technology mapped circuit sub-optimal and leave room for further improvement. Technology-dependent transformations can improve the circuit characteristics, such as delay, area and power, before handing the design off to physical layout. In this section, we give a quick overview of the most commonly used technology dependent optimization techniques. Note that most of these techniques also exist in a layout context, implemented during physical design, as described in section 3.4.

Fanout Optimization and *Buffering* seek to optimally distribute a signal from the driver gate, called the source, to the fanout gates, called the sinks, by adding buffers (or inverters), without violating the load limit of the source or those of the inserted buffers. As an example, buffering can reduce the arrival time at the output of a critical gate g along the critical path by reducing the total capacitive load driven by g. The additional delay due to inserted buffers will be added along non-critical paths. Various fanout optimization algorithms have been proposed in the recent past, e.g. by Alpert [2], Berman [7], and Kung [53], among others.

Gate Sizing is the problem of selecting the optimum size (i.e. drive strength) of each gate in the network such that some objective function, typically area and delay, is minimized or reduced without violating any constraints [22]. Due to the fact that technology mapping is unable to provide a mapping that optimizes both area and delay of a circuit, the size of the technology gates can be resized to improve delay and/or area. By adjusting the drive strength of a gate along the critical path, the delay can be

optimized, while area can be recovered by decreasing the size of gates along non-critical paths.

Gate Cloning is similar to buffer insertion in that it attempts to speedup a design by redistributing the fanout load [89]. The general idea is to copy or clone a gate *n* times and partition the fanout among all instances of the gate. Each clone drives less capacitive load than the initial gate, potentially reducing the delay along the path. However, gate replication results in an area penalty, and the speed-up of achieved by cloning the original gate has to be compared against the slow-down of its fanin gates that now need to drive a larger load.

Gate Decomposition partitions delay-critical simple (i.e. AND, OR etc.) multi-input gates into a tree of two or more gates to reduce delay. *Gate Collapsing* is the dual of gate decomposition. Recall that gate decomposition is also part of technology mapping, as shown in the previous section. A layout-driven technology decomposition to decompose a gate based on delay and wire length will be introduced in chapter 6 of this dissertation.

Resynthesis and *Remapping* applies logic minimization and attempts to find a better mapping using layout information [65]. Logic minimization and restructuring is applied to a region, typically a small set of gates, before it is remapped based on certain cost functions such as area and delay.

Pin Permutation is a simple technique exploiting asymmetric delays through swapping functionally symmetric pins [34]. Many technology gates show different timing delays at functionally equivalent, i.e. symmetric input. Therefore, it is possible to improve the delay by interchanging the signals at the symmetric input pins such that the late-arriving signal is connected to an input pin with smaller delay. This idea can also be applied to entire symmetric regions of the circuit, at the additional cost of identifying symmetries in the entire circuit.

3.3 Placement Techniques

In this section, we briefly review placement techniques, essential for the understanding of our layout-aware synthesis flow based on quadratic placement, as described in chapter 6.

Placement is the problem of generating a legal layout from a logic network satisfying a number of objective functions, i.e. timing, routability, cross-talk etc. A *legal*

placement is a non-overlapping placement of all cells within the given boundaries of the chip layout. In general, placement techniques can be classified into analytical (constructive) and move-based (iterative improvement) placement. Analytical placement uses linear or convex programming to minimize a closed-form, differentiable function. Per contra, algorithms based on iterative improvement evaluate different cell moves before deciding which moves to actually implement. Due to their heuristic nature, the initial seed placement often has a large impact on the final result and is susceptible to changes in the netlist (for example by logic optimization) during placement. Typically, the placement process consists of two phases: *Global placement* solves the placement problem down to a certain granularity while *detailed placement* refines the existing solution and makes the placement legal, i.e. non-overlapping.

3.3.1 Quadratic Placement

Quadratic placement is an analytical placement method based on minimizing the sum over the squared wirelengths. Every cell in the layout assumes a coordinate (x, y) in the placement plane. The cost of a wire is calculated using the squared L_2 distances, i.e. $(x_i - x_j)^2 + (y_i - y_j)^2$. The motivation of this formulation is that unlike trying to minimize the Manhattan distances which is NP complete, the problem translates into solving a system of sparse linear equations $A \cdot x + b = 0$, a relatively easy to solve problem. A is the connectivity matrix, x the vector of the movable cell coordinates and b the vector of the fixed cell coordinates. Without fixed cells (e.g. certain macro cells, blockages) and the legality constraints, the solution is trivial. Depending on the I/O pads along the outside of the circuit area, most of the cells usually end up on top of each other near the center of the layout area, because minimization of the quadratic wirelength does not result in an even distribution of netlength among the wires. To solve this problem, quadratic placement iterates between wirelength minimization and partitioning of the circuit. Recursive bi-partitioning, proposed by Johannes et al. in GORDIAN [48] divides the set of cells into two regions of the chip, alternating horizontal and vertical partitioning cuts. Vygen introduced the idea of quadrisection in quadratic placement [96, 97]; instead of alternating horizontal and vertical cuts, regions are simultaneously split into four parts, as shown in figure 3.3. The process of partitioning and wirelength minimization is iterated until a legal placed can be derived.

To consider timing, weights can be used to emphasize the criticality of a net. However, minimizing the squares of the wirelengths and not the actual linear wirelength overly minimizes long nets and therefore does not guarantee optimal timing. On the

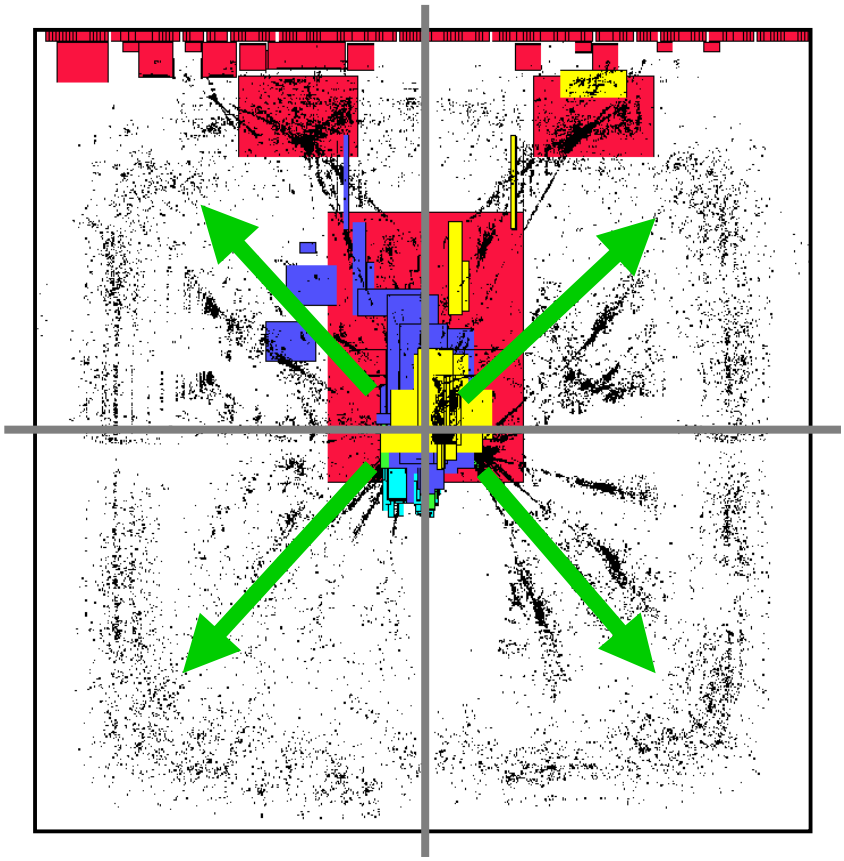


Figure 3.3: Quadri-section in quadratic placement

other hand, quadratic placement is very fast and can handle large designs. In addition, it is a relatively stable placement algorithm, which makes it suitable for integration with logic synthesis as changes in the netlist do not contradict previous placement information.

3.3.2 Force-Directed Placement

Force-directed placement models attractive and repulsive forces between the objects of the netlist, based on their interconnections, cell overlaps and timing criticality to achieve a balance between minimizing wirelength and yielding a legal placement [32]. The problem formulation is usually solved using the conjugate gradient method or other convex programming techniques. Force-directed placement has the advantage that due to the additional modeling of repulsive forces, the movement of larger blocks (e.g. macro-cells) is better captured. Thus it is better suited for the mixed placement of macro

and standard cells. However, this analytical placement method is considerably slower than quadratic placement, and additional cost functions are difficult to capture in an analytical objective function.

3.3.3 Simulated Annealing

Simulated Annealing is based on the idea of moving an object while evaluating the cost of the move [47]. A move is accepted if it improves the cost, e.g. wirelength, while a move with degrading cost is accepted with a probability of

$$P = e^{-\frac{|\Delta c|}{T}}.$$

$\triangle c$ is the difference in the cost resulting from the move while T is a global parameter called temperature. In the beginning, the temperature T is large, allowing larger moves with a higher probability of degrading cost. As the process is "cooled down", i.e. T is lowered, fewer cost degrading moves are accepted until a locally optimal solution is reached. However, depending on the cooling process, it is possible to find a global optimum. Simulated annealing is an extremely slow process, however, due its ability to potentially find the optimal solution, it is often used for detailed placement.

3.3.4 Min-Cut Placement

Min-Cut based placement provides another move-based heuristic solution. The netlist is partitioned into two sets of similar area while minimizing the number of nets across the cut-line [15]. Afterwards, cells are moved or swapped between partitions to reduce given cost functions. When a local minimum is reached, recursive partitioning of each set continues until a given set size is reached.

This placement technique produces good quality results and is, just like quadratic placement, widely used. Due to its move-based nature, min-cut is more suitable for timing-driven placement than a closed-form analytical solution. However, it is also more susceptible to local changes in the netlist.

3.4 Physical Synthesis

Previously, we gave an overview of the most common logic synthesis and placement algorithms. Now we take a closer look at their interaction and show existing solutions to

integrate synthesis and layout. *Physical synthesis* consists of changing the netlist to optimize the physical aspects of a design, i.e. timing, area, power, routing congestion etc. Recall that traditional physical design does not change the netlist but merely places the objects of the netlist on the chip image, and routes the interconnections.

3.4.1 Overview

To implement synthesis and placement, reliable timing information modeling interconnect delay is required. To accommodate for continuous changes in the netlist due to local synthesis transformations, the static timing analysis tool needs to be incremental. The accuracy of the delay model has a major impact on the success of the optimization process. The simplest delay model is based on total net capacitance. However, with shrinking feature sizes, the metal resistance per unit of length increases and cannot be ignored any longer. Even with the use of copper design processes, that tendency remains in the deep submicron area. As such, the delay on a net is estimated as a distributed RC network. The *Elmore delay model* [33], later improved by Wyatt [99], is a relatively simple and commonly used model for delay estimation on a RC tree. It is based on an exponential function with a single time constant being the sum of all RC products of the tree. This delay estimation works well enough for a first order approximation, but more accurate custom delay models with different delays at each fanout point of a net are needed as metal resistance increases.

The synthesis techniques applied in physical synthesis are mostly identical to the technology-based transformations reviewed in section 3.2.3, i.e. *gate sizing, buffering, gate cloning, local resynthesis and remapping, pin swapping and rewiring*, among others, now with the addition of having more accurate layout and delay information.

3.4.2 Physical Synthesis Design Flows

We present two of the most commonly used design flows to integrate physical synthesis under the constraints imposed by deep sub-micron layout. The first flow is the standard iterative approach, alternating synthesis and physical design, while the later one attempts a tighter integration of layout and synthesis.

3.4.2.1 Iterative Design Flow

This flow iterates between gate-level synthesis and place-and-route using custom wire load models. After each place-and-route step, if the constraints are not met, the netlist is

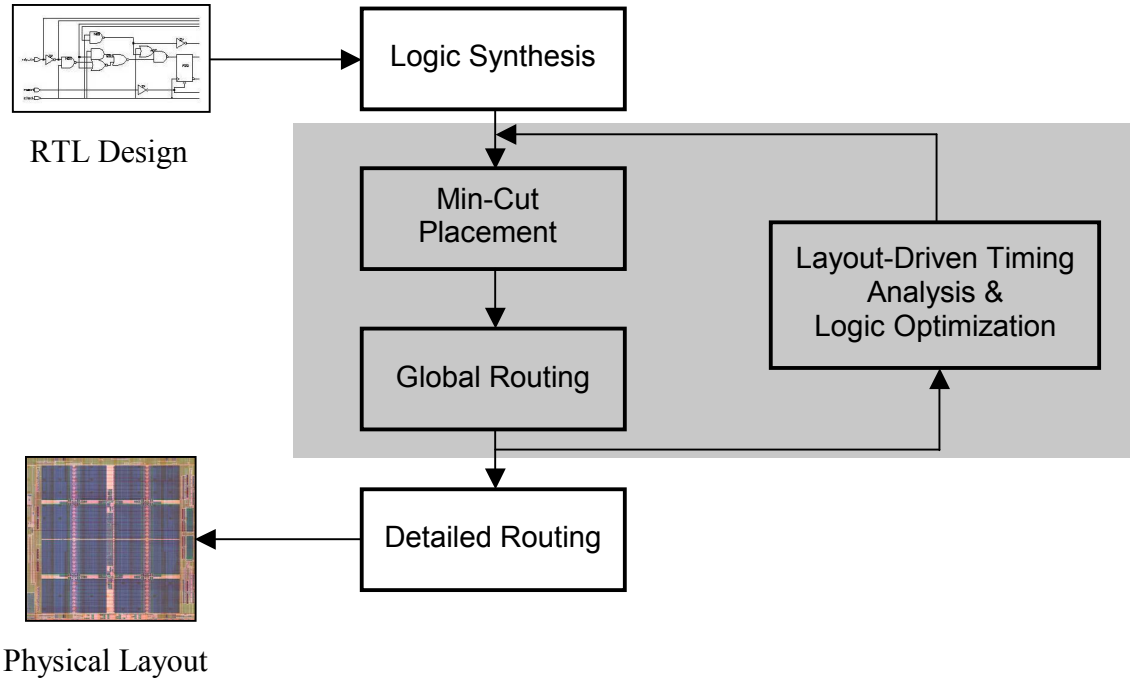


Figure 3.4: Iterative physical synthesis flow

resynthesized using the actual wire loads. The problem with this design flow is twofold. Usually, a logic change in the netlist results in a completely different placement solution (particularly if a heuristic placement algorithm is used). Therefore, this iterative process has no guarantee for convergence. Secondly, an iterative flow can be extremely time-consuming due to the many alternating heuristic synthesis and placement steps, until a satisfying solution has been found. Figure 3.4 shows a typical iterative flow integrating placement, global routing and synthesis.

For a tighter coupling of placement and synthesis, it is also possible to implement the synthesis transformations between the individual partitioning cuts of a min-cut placement as shown in [31].

3.4.2.2 Constant Delay Synthesis

This synthesis flow uses a load-independent delay model, e.g. the gain-based delay model [94]. For a better understanding of this and later parts of this thesis, we start by explaining the basics of load-independent delay modeling using the gain-based delay model as an example.

In the *gain-based delay model*, the delay d through a logic stage is expressed as a linear function of the *gain* g,

$$d = l \cdot g + p.$$

The gain is the ratio of the load to the input capacitance of the gate, C_L/C_{in}. l is called the logical effort, and p is the intrinsic delay of the gate. The logical effort of a gate is defined as the ratio of its input capacitance to that of an inverter that delivers equal output current. In other words, the logical effort gives the cost of driving a given capacitance by a certain type of logical gate (e.g. NAND vs. Inverter). As a result, the delay is formulated as a linear function of the gain. This is a very simple yet attractive delay model because the delay becomes independent of the driven load.

In the *constant delay synthesis* flow, a fixed delay (i.e. a fixed gain) is assigned to every logical stage such that timing constraints are met. The second step consists of placing and routing the netlist while preserving these assigned delays. Throughout the design process, gates are repeatedly sized to enforce the assigned delay. Area-sensitivities introduced in [93] allow for an accurate prediction of the effect of resizing a gate on the total circuit area. However, the simplicity of this delay model makes it difficult to capture the propagation of the actual waveform, e.g. to consider slew dependencies [54]. In addition, constant delay synthesis assumes continuous sizing, which means that mapping it onto a real discrete library may result in a sub-optimal solution due to the introduced discretization error [51]. On the other hand, the relative simplicity of this model allows for fast synthesis of complex designs, and provides an elegant closed-form solution to the synthesis and place-and-route integration.

3.5 Conclusion and Motivation

We have given an overview of logic synthesis and placement algorithms, together with commonly used models to combine these two steps during physical synthesis. Based on the previous observations, we can identify the following two problem areas:

Structure-Aware Synthesis. Existing logic minimization algorithms, whether two- or multi-level, have no conception of the structure of the netlist. Based on their locality, we can distinguish between algorithms having either global or local impact. Global optimization methods, for example kernel factoring, are known to perform poorly on structured data logic. As an example, kernel-factoring works well on random logic, i.e. control logic, but will completely destroy most adder structures resulting in poor

performance. Local transformations, on the other hand, are applied in arbitrary order in the network and thus can destroy regular structures in the network. As a result, synthesis should, in addition to area and timing objectives, also consider the structural properties of the network to improve the optimization of data logic.

Layout-Aware Synthesis. We have studied the various attempts to integrate synthesis and placement but in the end, all existing physical synthesis flows have one thing in common: they work on an already minimized and technology-mapped netlist. However, many decisions have already been made *before* physical synthesis, e.g. the decompositions have been chosen, the technology matches have been selected. Synthesis algorithms applied during physical synthesis try to locally fix problem areas instead of avoiding them right from the start. In addition, many of the sub-optimal decisions made much earlier in the design flow can only be reversed at substantial cost. In conclusion, a synthesis approach is necessary which introduces physical awareness already during logic restructuring and technology mapping, the steps that have the largest impact on the overall structure of the network.

In the remaining chapters of this dissertation, we present several new algorithms to address these problems, resulting in a layout and structure aware synthesis flow.

4 Regularity Extraction

This chapter formally introduces the notion of regularity and presents a fast functional regularity extraction algorithm based on structural equivalence. Then, we show that by reusing functionally equivalent structures of the design, it is possible to significantly speedup the process of logic optimization, particularly useful for complex circuits containing a large amount of datapaths.

In the second part of this chapter, we extend this concept to structural regularity and show that the placement of data logic in regular blocks yields higher packing densities, shorter wiring length and improved delay in physical layout.

4.1 Introduction

Most modern VLSI designs are characterized by a large amount of datapath circuitry that exert bit-wise parallelism to achieve high performance. Such circuits contain a high degree of *regularity* due to the application of identical bit-operations across the width of the data representation. An example of a datapath representing a *regular group R* with at least two (*bit-)slices* and (*bit-)stages*, which determine the *group height h* and *group width w*, respectively, is shown in figure 4.1.

In general, we can distinguish two types of regularity, *structural* and *functional*. Given a Boolean network B of a circuit C, we can define functional regularity as follows.

Definition 4.1: A *functionally regular group* $R_f = (B_0, \ldots, B_i, \ldots, B_n)$ is a set of two or more disjoint Boolean subnetworks B_i $(0 \leq i \leq n)$ of B, all of which are pairwise Boolean equivalent.

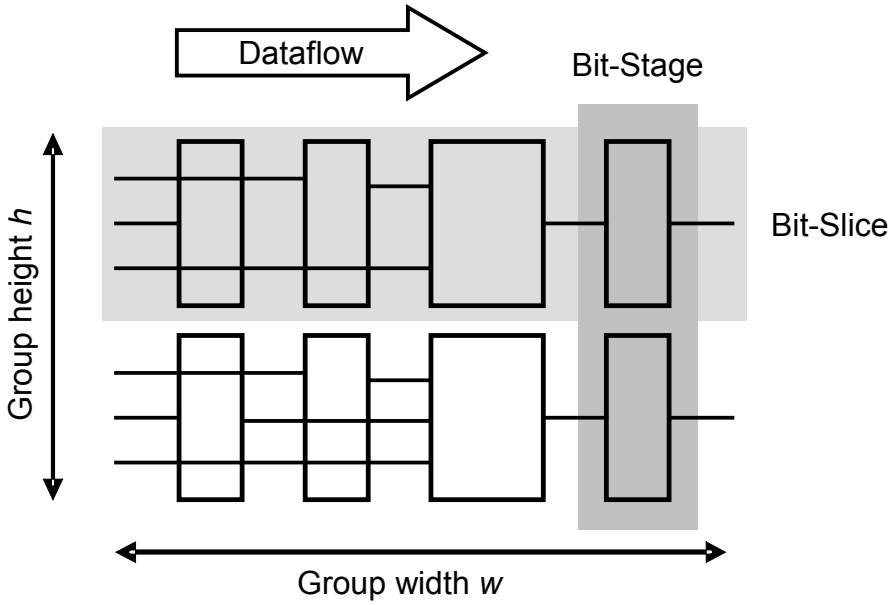


Figure 4.1: Illustration of a regular group

Please note that each subnetwork represents a *bit-slice*. *Functional regularity* requires all bit-slices of a regular group to be functionally equivalent, i.e. the canonical representation of the Boolean function of each bit-slice is identical. A special form of functional regularity is if all subnetworks are topologically equivalent, i.e. the underlying graph representations are isomorphic.

On the other hand, *structural regularity* only implies the existence of identical gates across all bit-slices of a bit-stage, for all bit-stages. Thus, it does not require complete topological equivalence of each bit-slice. For example, the regular group shown in figure 4.1 is structurally but not functionally regular, because the second bit-slice contains an extra connection between the first and third bit-stage. It follows that functional regularity does not imply structural regularity and vice versa.

4.2 Motivation

Usually, two different methods are employed in the design of datapath circuits: datapath synthesis based on generic logic synthesis, or by means of a specialized *data path compiler (DPC)*. In the generic approach using logic synthesis, no notion of regularity is used. All transformations such as factoring, redundancy removal, collapsing and

constant propagation are applied to the whole design. Therefore, its regularity properties are widely unused. In contrast, datapath compilers explicitly create regularity at the logical level and preserve it throughout the entire design flow. A major drawback of this technique, however, is a decrease in generality and flexibility caused by the implementation of a separate, mostly technology-dependent design flow, adding additional cost and integration overhead. This results in a need for methods that extract the existing regularity in a design and make beneficial use of it in an existing design flow.

One could argue that the easiest way to extract the regularity is directly during high-level synthesis. However, in a typical design flow, many different tools interact and it is nearly impossible to guarantee a coherent flow with tools of only one vendor. In addition, the gate-level netlist changes significantly before physical design. Therefore, it is generally desirable to be able to extract regularity from a flat netlist without the use of a specific HDL compiler, user hierarchy or name hints.

Several techniques for the extraction of regularity have been proposed in the literature. Many of these techniques focus on extracting functional regularity by matching the target circuit with templates [21, 23, 55]. A major problem with these methods is the creation of a suitable set of templates that needs to be provided by the user. Kurdahi et al. [55] and Chowdhary et al. [21] proposed approaches that automatically generate a set of templates under a set of simplifying assumptions. However, these techniques are limited in their practicality because they are computationally expensive and generate mostly simple templates. In addition, the attained results greatly depend on the particular design. A different approach by Odawara [74] identifies one-dimensional structural regularity in terms of bit slices of a datapath. Nijssen et al. [72, 73] and Arikati [3] extend this idea into two-dimensional regularity extraction, identifying bit slices as well as stages of a datapath. The result of this structural regularity extraction is then used in the placement process to obtain highly regular layouts.

A significant part of the basic regularity modeling is based on the ground-breaking work of Nijssen et. al. [72, 73] who first introduced two-dimensional structural regularity extraction and showed its application in placement. We extend this work by the introduction of functional regularity extraction and its application for logic optimization and regularity preservation. For further reading on structural regularity extraction and application in placement, we refer to his work.

4.3 Functional Regularity

While regular structures have been used to obtain high-density layouts in placement [67, 72, 70], this idea has not been applied to logic optimization. To make use of this important property, it is necessary to extract functionally equivalent parts of a design. However, conventional algorithms for functional regularity extraction based on template matching are in general too complicated and computationally expensive. Therefore, a new technique to extract functional regularity based on the structural properties of a circuit is proposed.

In a standard design flow, the functional regularity expressed in an HDL description is typically inherited by the netlist representation as structural regularity [72], characterized by identical interconnect structure and gate functions. The proposed approach uses this idea to extract functional regularity from a netlist before technology-independent optimization, i.e. before most of the inherited regularity might be destroyed. In addition, we apply simple network transformations to identify more regularity and allow regular structures to grow into every direction. This results in an *n*-dimensional regularity extraction process yielding regular structures of any possible shape including rectangular and triangular shapes.

The proposed algorithm is typically much faster than generic functional regularity extraction based on pattern matching or functional verification. The extracted regularity information is used to partition the design into fully regular groups. The final task of logic optimization yields a significant improvement in run time since only one entity of each regular structure needs to be optimized.

4.3.1 Regularity Model

In this section, we introduce our signature-based regularity model. The goal of regularity extraction for logic optimization is to partition a design into regular groups of functionally equivalent structures.

To achieve high efficiency, our regularity model is a heuristic approach that finds functional regularities based on the structural properties of a netlist.

Given a circuit C, consisting of a set G of gates, a set N of nets and pins $P \subset G \times N$, we assign a *gate type* γ to each gate g. It is characterized by the logic function f of gate g. Similarly, each pin p of a gate g is assigned a *terminal type* τ which is described by the pin function of p, i.e. input or output for symmetric combinatorial gates, or special functions for complex and dynamic gates. The basic relation of gate and terminal type is shown in figure 4.2. The primary inputs and outputs of the circuit, denoted by a solid circle, are treated as special pins assigned to a dummy gate with one output and no inputs (PIs), or no output and one input (POs). Based on the previous definitions, we can model a circuit C as a labeled directed graph D.

Definition 4.2: Let Γ and T be the set of gate and terminal types, respectively. A *labeled directed graph* $D(V, E, \Gamma, T)$ of a circuit C consists of a set V of vertices, a set $E \subseteq V \times V$ of edges, and a set of vertex and edge labels, $H: V \mapsto \Gamma$, and $I: E \mapsto T \times T$.

Note that every vertex of the labeled graph has a gate type γ, and every edge a pair of terminal types (τ_a, τ_b) assigned.

During regularity extraction, *regular groups* R, consisting of functionally identical subgraphs called *slices* S, are identified. Figure 4.3 a) and b) show an example of a regular group with two identical slices S_1 and S_2, represented as a labeled directed graph. Based on our regularity model, we can define functional regularity as follows.

Definition 4.3: Let $R_f = (S_1, ..., S_i, ..., S_n)$ be a set of n disjoint subgraphs of D. Then R_f is a *functionally regular group*, if, and only if, all *slices* S_i $(1 \leq i \leq n)$ in R_f are pairwise strongly isomorphic.

Recall that strong isomorphism, as previously introduced in definition 2.9, preserves adjacencies and non-adjacencies of the subgraphs, and establishes a one-to-one mapping of the labels of corresponding edges and vertices.

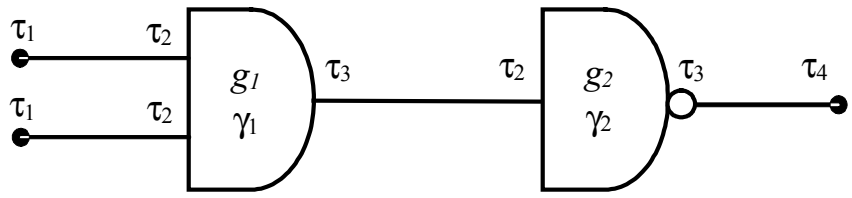
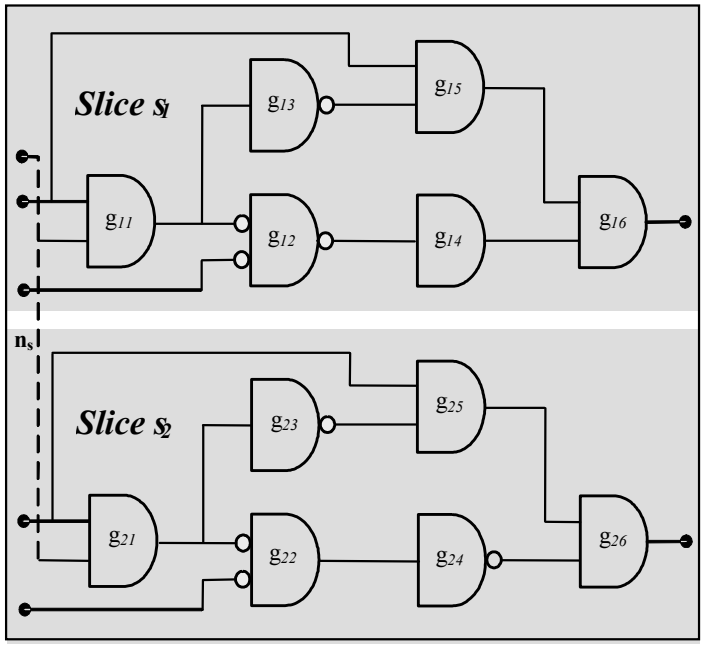


Figure 4.2: Relation of gate and terminal type

a)



b)

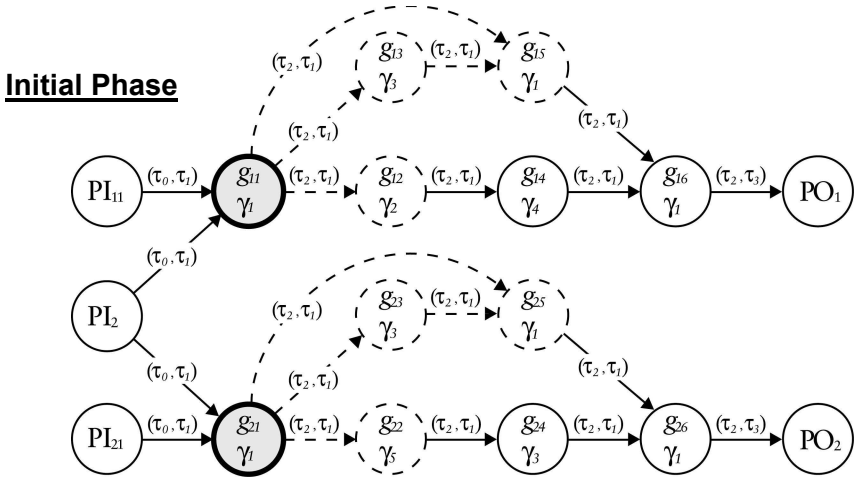


Figure 4.3: Part of a datapath, a) network representation, b) labeled directed graph representation (intial phase of the regularity extraction)

The extraction algorithm starts with an initial regular group, consisting of one vertex per slice, calculates *regularity signatures RS* and performs *regular expansions*, thus growing all slices of a group simultaneously.

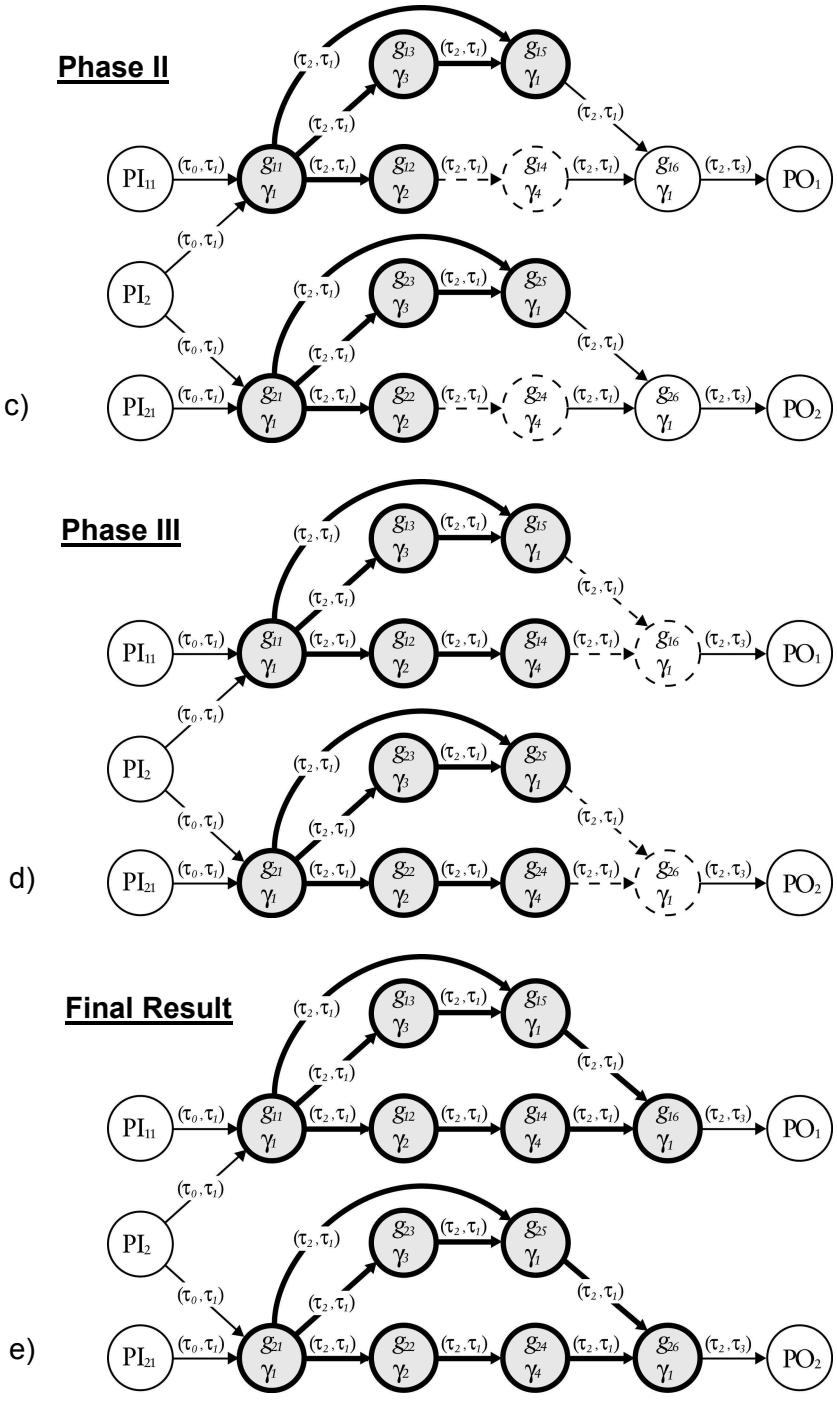


Figure 4.3: c), d), e) Phases of the regularity extraction algorithm

Definition 4.4: Let V_s be the set of vertices in slice S, and V_c be the set of immediate successors or predecessors of a vertex $v \in V_s$ such that $V_c \cap V_s = \emptyset$. The *regularity signature RS* of v with respect to V_s is the labeled directed graph consisting of V_c, the vertices of V_s being adjacent to V_c, and connecting edges $[(V_c \times V_s) \cup (V_s \times V_c)] \cap E$.

Regularity signatures are identical, if their respective graph representations are strongly isomorphic. Please note that the definition of a regularity signature considers all reconvergent connections between the to-be-included successor gates and gates already within the slice, to ensure functional equivalence.

Definition 4.5: Let $R_f = (S_1, ..., S_i, ..., S_n)$ be a functionally regular group with n slices, and let RS_i be the regularity signature of a vertex v_i in slice S_i. Then, if all regularity signatures RS_i $(1 \leq i \leq n)$ are identical, a *regular expansion* is defined as $S_i' = S_i \cup RS_i$.

Hence, a regular expansion defines the addition of adjacent logic to all slices such that all slices S_i remain isomorphic, which implies functional equivalence. Consecutive regular expansions will result in the extraction of functionally equivalent slices.

For a better understanding, consider the sample datapath in figure 4.3. Assume gates g_{11} and g_{21} as the starting point, hence we have initial slices $S_1 = \{g_{11}\}$ and $S_2 = \{g_{21}\}$, shown bold in fig. 4.3 b). (For the purpose of simplicity of representation of a slice, connecting edges between the vertices are assumed.) The calculation of regularity signatures for slices S_1 and S_2 yields identical signatures, shown dashed in fig. 4.3 b). Therefore, a regular expansion of both slices is possible, resulting in identical slices $S_1 = \{g_{11}, g_{12}, g_{13}, g_{15}\}$ and $S_2 = \{g_{21}, g_{22}, g_{23}, g_{25}\}$, shown bold in fig. 4.3 c). The complete regularity extraction algorithm will be outlined in the following section.

In such way, the regularity signature *RS* allows us to identify functionally regular structures. For the purpose of logic optimization, we are only concerned with completely regular structures. An extension of the basic definition of a regularity signature, allowing slightly less regular structures, is given in section 4.3.3.

4.3.2 Extraction Algorithm

The proposed algorithm identifies suitable start points, e.g. control nets of datapaths, gates that have control logic in common etc., and expands search waves through the entire design. A regularity signature *RS* as described in the previous section is used to grow regular structures. Existing regular structures, i.e. the slices S_i of a group R are grown using *regular expansions*. An expansion is made only if the regularity signatures

of all slices within the regular group are identical. The procedure finishes when no further regular expansion is possible.

Our algorithm [56, 57] is applied to a netlist represented only by AND and XOR gates with negated inputs and/or outputs, and of course, dynamic logic. For example, an OR gate is represented as an AND gate with negated inputs and output. The Boolean algebra formed by the conjunction and antivalence operations is called a Boolean ring [9]. This representation is essential since a design consisting of few simple types of gates allows us to achieve a much higher correlation of functional and structural regularity with the help of simple transformations.

Similar to the idea used in placement-based regularity extraction [72], the algorithm starts by automatically identifying promising start points in the design. These so called *seed nets* and/or *gates* are usually high fan-out control nets such as control lines, address selectors, enable lines, gates with common control logic etc. Therefore, they are easily determined by searching for nets connected to input pins of gates with identical gate type. In addition, the algorithm identifies similar net and gate names by matching string patterns. Usually, the names of corresponding nets or gates in the bit-slices of a datapath created by an HDL tool are very similar. In many cases, they differ only by a part identifying the particular bit vector. This property is used to find additional seed nets.

The complete algorithm is outlined in figure 4.4. It uses two different priority queues P and Q, holding *seed nets sn* and *reference sets ss*, respectively. Queue P is ordered by the number of identical gate types of a seed net to ensure that the most promising candidates are examined first. During the extraction process, each gate is assigned a unique ID consisting of a group, slice and member number. A member number uniquely describes the position of a gate within a slice. At start, all gates are initialized with an empty ID representing an undefined slice/group membership. A *seed set* ss_i, defined as the set of gates with identical labels (i.e. gate types), is created from each seed net sn_i and put into priority queue Q. Each gate in a seed set represents an initial slice, while the sum of all gates form an initial regular group R. For example, gates g_{11} and g_{21} in figure 4.3 represent initial slices S_1 and S_2.

Priority queue Q is ordered by the number of slices in a *reference set rs*. A reference set is simply the set of gates with identical member numbers (i.e. it represents a bit-stage of a regular group), which is subject to the next regular expansion. Please note that a seed set is simply the initial reference set of a regular group.

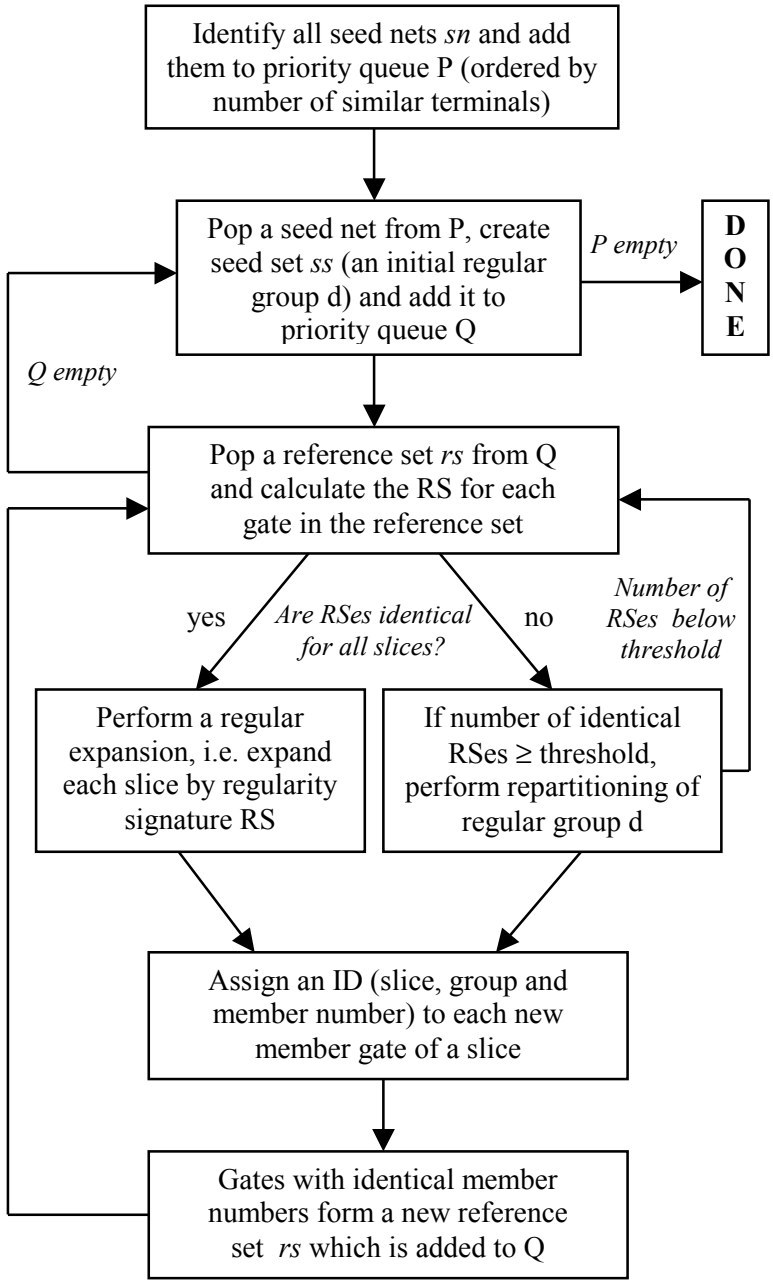


Figure 4.4: Regularity extraction algorithm

For each gate in the reference set *rs*, a regularity signature *RS* is calculated. If all slices contain an identical *RS*, each slice of the regular group is grown using a *regular expansion*. By extending the basic definition of a regularity signature to include all predecessor instead of successor vertices, a regular expansion can be performed

backwards. In that way, a slice is allowed to grow into every possible direction. Each gate of the expansion is assigned the slice and group number of its originating gate. Corresponding gates among the slices are associated with an identical member number. They form a new reference set and are added to priority queue Q.

In addition, we employ simple network transformations to identify structurally different, but functionally equivalent parts of the circuit. Among these transforms are Identity, and DeMorgan's Law, which are easily implemented by shifting the negations at the inputs and outputs of the primitive gates. During the calculation of regularity signatures, these matching techniques are performed in a search space of up to K levels deep. The search continues until identical regularity signatures could be obtained, or the search space of K levels has been exhausted. K is a user-defined constant.

If only a subset of all slices yield identical signatures, the regular group is repartitioned. This process is shown in figures 4.5 and 4.6 using a simplified two-dimensional block representation. In general, there is a tradeoff between the height of regular groups, i.e. the number of slices in a group and the maximum width of a group, i.e. the number of gates within a slice. As depicted in figure 4.5, vertical repartitioning increases the average height of a regular group. This results in a better run-time improvement of the logic optimization since an optimized slice is reused more often. In contrast, a horizontal repartitioning increases the average number of gates in a slice. This generally improves the quality of the optimization result by allowing it to operate on a larger structure.

To achieve better optimization results during logic minimization, the existing regular group is partitioned horizontally into two regular groups to obtain slices with a maximum number of gates, as shown in figure 4.6. All slices of group R_1 contain identical regularity signatures and can be expanded as shown by the hatched area.

Theoretically, also the slices of group R_2 may contain identical regularity signatures and could continue to grow. This repartitioning process is controlled by threshold values determining the minimum group width and height.

The algorithm continues until all reference sets have been visited and no seed nets are left, i.e. no further regular expansions are possible. To better illustrate this process, we outline the extraction process on the example shown in figure 4.3. Initially, all gates are unassigned to any slice or group. The algorithm first identifies the net that connects primary input port PI_2 via terminal type τ_1 to gates g_{11} and g_{21} with identical gate type γ_1 as a seed net, shown by a dashed line. Since no further seed nets exist, we have a seed

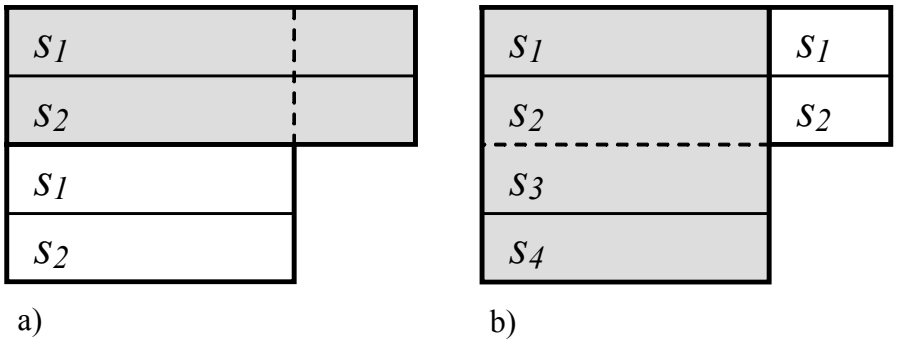


Figure 4.5: a) Horizontal, and b) Vertical repartitioning

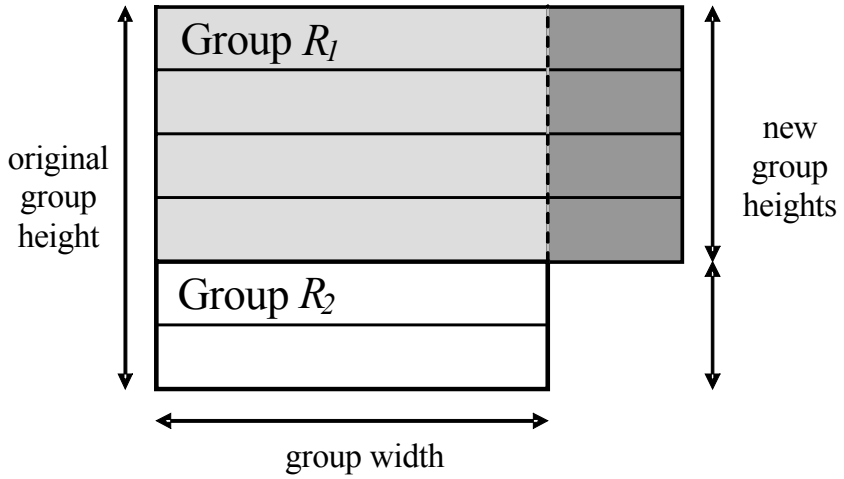


Figure 4.6: Group repartitioning

set ss = {g_{11}, g_{21}}. It represents initial slices S_1 = {g_{11}} and S_2 = {g_{21}} of group R_1, shown bold in fig. 4.3 b). (Again, in this representation of a slice, all connecting edges between the vertices are assumed.) The seed set is the initial reference set, subject to regular expansion. For each gate in the reference set rs_1 = {g_{11}, g_{21}}, the regularity signature is calculated, shown dashed. Since gate g_{12} is an OR but gate g_{22} is a NOR gate, the signatures are different, but by shifting the negation from gate g_{24}, we create equal signatures. Since our network consists of only AND and XOR gates, many such simple matches can easily be performed. The resulting regularity signatures, shown in bold, are now identical for both slices, hence a regular expansion can be performed. Subsequently, we obtain slices S_1 = {g_{11}, g_{12}, g_{13}, g_{15}} and S_2 = {g_{21}, g_{22}, g_{23}, g_{25}}, as

shown in fig. 4.3 c), and new reference sets $rs_2 = \{g_{12}, g_{22}\}$, $rs_3 = \{g_{13}, g_{23}\}$, and $rs_4 = \{g_{15}, g_{25}\}$. Further expansions by rs_2 and rs_4 yield slices $S_1 = \{g_{11}, g_{12}, g_{13}, g_{14}, g_{15}\}$ and $S_2 = \{g_{21}, g_{22}, g_{23}\ g_{24}, g_{25}\}$, shown in fig. 4.3 d), and reference sets $rs_5 = \{g_{14}, g_{24}\}$, and finally $S_1 = \{g_{11}, g_{12}, g_{13}, g_{14}, g_{15}, g_{16}\}$ and $S_2 = \{g_{21}, g_{22}, g_{23}\ g_{24}, g_{25}, g_{26}\}$. The final slices are depicted bold in fig. 4.3 e). Since no further expansion is possible, the algorithm terminates.

Thereafter, the design is hierarchically partitioned into the identified regular groups and their respective slices, and the remaining non-regular logic. We run logic optimization on one entity of all functionally identical structures, i.e. one slice of each regular group, and on the remaining non-regular logic. Then, the optimization result of one slice is copied to all other slices of the group. Finally, the hierarchy is flattened. Alternatively, the partitioning can be retained for further usage of the extracted regularity in the design flow, for example during placement.

This method significantly improves the overall run time since only one entity of all identical structures needs to be optimized.

4.3.3 Complexity Issues

In this section, we analyze the complexity of the proposed regularity extraction algorithm. In the worst case, all nets are seed nets, and thus all edges and all vertices in the network graph will be visited. Furthermore, a design could be completely regular, in which case again all nodes and all vertices will be visited. Therefore, the complexity of the algorithm is $O(m+n)$ where m is the number of vertices and n the number of edges in the network graph.

4.3.4 Extensions to the Algorithm

The proposed algorithm can be extended in several ways. A simple extension to the basic algorithm is to find identical slices in different groups as the final step of regularity extraction. In that way, more occurrences of a particular structure can be identified while overall regularity remains unaffected.

An approach to improve the amount of regularity extracted from a design is to use a slightly weaker definition of the regularity signature *RS*. The previous definition of *RS* considers *all* gates connected to a net *n*. It demands that each net, being part of a regular expansion, is connected to gates of the same slice only, therefore, not allowing nets that

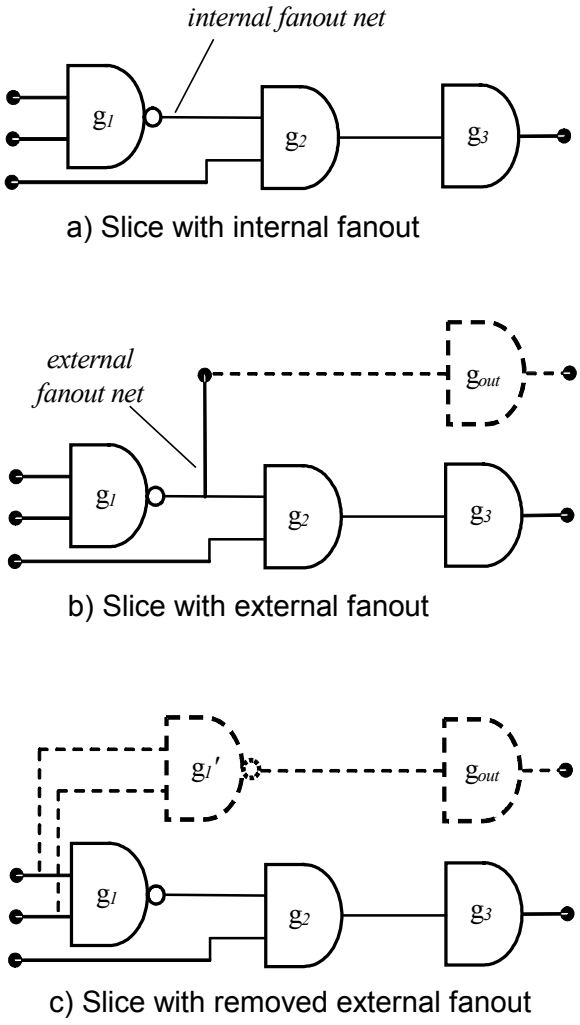


Figure 4.7: Illustration of fanout nets

are connected to the surrounding logic of a different slice, called *external fanout nets*. Figure 4.7 shows an example of a slice that contains an external fanout net that is connected to gate g_{out} outside the slice.

This definition of the regularity signature *RS* represents a logical assumption for optimization purposes since combinatorial logic optimization is unable to optimize over a boundary created by an external fanout net. However, a different definition of the regularity signature that includes only one successor vertex can be chosen.

This weaker criterion finds slightly more regularity and larger slice sizes allowing slightly irregular slices that differ only by its external fanout nets. For example, the slices in figure 4.7 a) and 4.7 b) are identical except for the external fanout.

However, such slices can be made regular by duplicating parts of the logic leading to an external fanout net, hence removing external fanout nets from a slice. Consider the slice in figure 4.7 c). After removing the external fanout by duplicating gate g_1, it is identical with slice 6a.

This method allows optimization to operate over previously existing boundaries. Interestingly, the procedure of duplicating gates is also used in the timing optimization of logic synthesis. The path delay in 4.7 a) is smaller than in b) due to the larger fanout of gate g_1 in b). Duplicating gate g_1 in the same manner as introduced for regularity extraction is used to correct the timing during technology dependent optimization. Therefore, this process can even be beneficial, if the particular slice is part of the critical path.

4.3.5 Experimental Results

The proposed regularity extraction algorithm was implemented in C++ within the framework of the logic synthesis tool BooleDozer™ [91].

Input to the program is a technology-independent netlist of the design. The program first extracts the regularity in a design, automatically partitions it accordingly, and performs logic minimization in BooleDozer using script mode *destruct*. Table 4.1 presents the functional regularity extracted from a number of designs dependent on the minimum number of gates in a slice, i.e. the minimum width of a group, w_{min}. The *group height threshold*, which determines the minimum height of a regular group, i.e. the number of slices in a group, was chosen as 2 for all test runs. Design *CRM* is part of an ATM controller, *ALU* a 32-bit arithmetic-logic unit, *AST* and *Condor* are small microprocessor kernels. Shown are the amount of functional regularity as the percentage of gates (relative to all gates in the circuit) within regular groups Reg_f, and maximum and average group height h_{max} and h_{avg}, and the average group width w_{avg}. In addition, an estimation of the reduction in synthesis effort is given. Assuming at least *O(n)* runtime complexity, we can pessimistically estimate the run time as *1/group_height* for each regular group, comparing the number of literals per slice, since the optimized result can be reused at least *group_height – 1* times.

Design	*Gates*	w_{min}	Reg_f	h_{max}	h_{avg}	w_{avg}	$effort_{opt}$	*CPU*
S1423	653	2	46.9%	12	5.24	4.0	-36.4%	0.8 s
S1423	653	10	16.1%	4	4.0	12.5	-11.7%	0.7 s
S38417	11890	2	85.8%	88	5.6	24.3	-64.2%	8.8 s
S38417	11890	10	72.6%	40	5.5	176.0	-53.4%	8.5 s
S38584	14489	2	77.9%	160	3.8	11.8	-53.3%	13.3 s
S38584	14489	10	57.5%	8	10.4	114.0	-38.6%	12.5 s
ALU	2229	2	68.4%	32	5.5	4.1	-48.1%	2.2 s
ALU	2229	10	30.9%	16	9.5	28.0	-20.7%	2.1 s
CRM	1825	2	67.8%	64	4.9	3.4	-46.0%	2.8 s
CRM	1825	10	24.9%	32	16.1	11.3	-16.3%	2.6 s
AST	5514	2	84.3%	16	2.9	25.2	-62.1%	4.2 s
AST	5514	10	64.0%	8	2.0	588	-45.8%	3.9 s
Condor	4522	2	58.2%	36	6.2	15.1	-32.4%	3.4 s
Condor	4522	10	32.2%	32	5.7	47.0	-22.2%	3.2 s

Table 4.1: Functional regularity extraction results

The results show that modern designs contain a large amount of functional regularity. Most designs from the ISCAS benchmarks however, are relatively out of date and do not represent the current state of technology. Only the larger benchmarks, S38417 and S38584 contain a considerable amount of datapaths.

Table 4.2 summarizes the results for the final logic optimization applied on the whole design. *lit std* and *lit reg* show the number of literals for the standard flow and with regularity extraction, respectively, in comparison with the non-optimized design, *lit orig*. The percentage of functional regularity (i.e. the percentage of gates that are within a regular group vs. all gates in the design) for both flows is given by Reg_f *std* and Reg_f *reg*. The optimization process performs restructuring through elimination, algebraic optimization, redundancy removal, transduction and kernel factoring, among others. *CPU impr* gives the decrease in runtime achieved by re-using functionally equivalent parts of the design. The overhead to extract the regularity and partition the design is included. All run times were measured on an IBM PowerPC/140™ workstation.

Design	$slice_{min}$	Reg_f *std*	Reg_f *reg*	*lit orig*	*lit std*	*lit reg*	CPU_{impr}
S1423	10	9.8%	12.2%	1262	1012	1022	-21.7%
S38417	100	55.4%	65.2%	23496	18900	19132	-60.7%
S38417	1000	27.5%	36.7%	23496	18900	19087	-33.0%
S38584	100	15.0%	18.5%	30136	24402	24818	-23.1%
ALU	10	27.6%	32.1%	4207	2101	2211	-22.3%
CRM	10	25.8%	27.2%	3918	2130	2386	-25.4%
AST	10	42.2%	62.1%	19124	18279	18313	-29.9%
Condor	10	21.1%	34.7%	16022	15553	15902	-21.1%
Average	**-**	**28.1%**	**36.1%**	**15208**	**12660**	**12859**	**-29.7%**

Table 4.2: Optimization results

In general, it is evident that logic optimization on a partitioned design yields a slightly higher literal count since the optimization process is restricted to smaller structures, resulting in local optima. However, the increase is generally negligible. Overall, the results are better if the design contains large regular structures. In addition, the result is dependent on the connectivity between the regular structures. For example, design *AST* contains a high amount of datapath logic, in contrast to most of the *ISCAS* benchmark circuits. The slices in a datapath circuit show less interconnectivity, hence they yield better results in the optimization of the partitioned design. Therefore, the average size of regular slices, the percentage of regular circuitry as well as design-specific factors, such as the connectivity between different slices in a design determine the quality of the obtained results.

For example, circuit *S38417* contains large regular groups covering over 55% of the whole design, resulting in a runtime improvement of more than 60% for the optimization process while producing only slightly more literals in the final design.

On average, the test designs yielded a run time decrease of 30% while the number of literals increased by only 1.5%. The run time of the regularity extraction is superior in comparison to other extraction algorithms that are mostly based on template matching. Only little overhead is added to the actual task of logic optimization.

4.3.6 Conclusion

We have presented a fast algorithm to extract functional regularity and perform efficient logic optimization by reuse of identical structures. In contrast to regularity extraction algorithms using template matching, it is based on structural properties and finds regular structures of arbitrary shape by expanding an existing regular structure into every possible direction. Therefore, it is independent on the particular design and specific template libraries. For designs with large regular structures, a superior speedup of the optimization process with minimal trade-off in the final result is obtained. In addition, the existent regularity is preserved and can be beneficially used in the following physical design steps.

4.4 Structural Regularity

The motivation for the extraction of structural regularity lies in the fact that identification and placement of regular structures creates a compact two-dimensional layout of the logic, resulting in shorter wire length and improved delay. This process is typically performed after technology mapping and technology optimization, providing input to the placement program. The problem here lies in the preservation of regularity throughout logic synthesis, because most of the regular structures are destroyed in a standard design flow. This, however, will be addressed in detail in chapter 5. Here, we extend our regularity extraction algorithm to identify structural regularity and show its benefits in physical design.

4.4.1 Regularity Model

In this section, we extend our signature-based functional regularity model to the extraction of structural regularity.

Recall that structural regularity only requires an identical partial order of the gates in each bit-slice of a regular group. Given a circuit C, consisting of a set G of gates, a set N of nets and pins $P \subset G \times N$, we assign a *gate type* γ to each gate g. Earlier, the gate type was characterized by the logic function f of gate g. Now, since we are concerned with the creation of compact regular layouts, the gate type has to be described by the physical footprint, i.e. the height and width of the physical cell. Based on these assumptions, we can model a circuit C as follows.

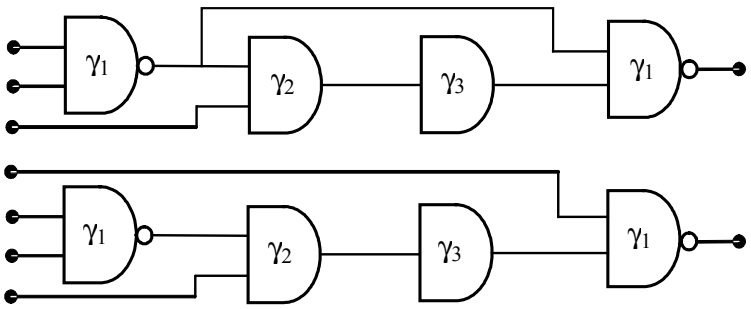


Figure 4.8: Example of a regular structure

Definition 4.6: Let Γ be the set of gate types. A *labeled directed graph D(V, E, Γ)* of a circuit C consists of a set V of vertices, a set $E \subseteq V \times V$ of edges, and a set of vertex labels H: $V \mapsto \Gamma$.

Consequently, we define structural regularity with respect to our regularity model as follows.

Definition 4.7: Let $R_s = (S_1, ..., S_i, ..., S_n)$ be a set of n disjoint subgraphs of D, and let O_i $(1 \leq i \leq n)$ be the topologically ordered multi-set of gate types in S_i. Then R_s is a *structurally regular group*, if all O_i $(1 \leq i \leq n)$ are pairwise identical.

An example of a structurally regular group is shown in figure 4.8. Both slices have the topologically order multi-set (γ_1, γ_2, γ_3, γ_1), but are not functionally equivalent.

Now we define a regularity signature RS, which is essential for the expansion process.

Definition 4.8: Let V_s be the set of vertices in slice S, and v_c be an immediate successor or predecessor $v_c \notin V_s$ of a vertex $v_s \in V_s$. The *structural regularity signature RS* of v_s is the labeled directed graph consisting of v_s, v_c, and the connecting edge (v_s, v_c).

A regular expansion is performed by adding identical regularity signatures to the existing bit-slices. Regularity signatures are identical if their labeled directed graphs are strongly isomorphic. Note that regular expansions have to be of the same topological order to preserve the topological order within each slice.

Definition 4.9: Let $R_f = (S_1, ..., S_i, ..., S_n)$ be a structurally regular group with n slices, and let RS_i be the structural regularity signature of a vertex v_i in slice S_i. Then, if all regularity signatures RS_i $(1 \leq i \leq n)$ are identical, a *regular expansion* is defined as $S_i' = S_i \cup RS_i$.

The regular expansion process is identical to our definition in 4.3, only the definition of the regularity signature differs. Therefore, a regular structure is expanded by exactly one gate at a time. Similarly to the algorithm for functional regularity extraction, expansions are performed forward, i.e. including successor gates, and backward, including predecessor gates. To preserve the topological order of the gate types within the slices, the gates of the regularity signatures are processed in topological order.

4.4.2 Extraction Algorithm

The extraction algorithm works in the same way as the previously introduced algorithm for extracting functional regularity, shown in figure 4.4. In the beginning, seed nets with a high fanout are identified, and initial slices are grown by regular expressions backward and forward through the circuit until no further regular expansion is possible. Each regular expansion has to be of the same topological order with respect to the slices of the regular group.

The regularity signature for structural regularity, however, differs from that for functional regularity in that it expands each slice by a single gate at a time. Since this is a generally weaker constraint, i.e. it does not require functional equivalence, designs typically contain more structural than functional regularity.

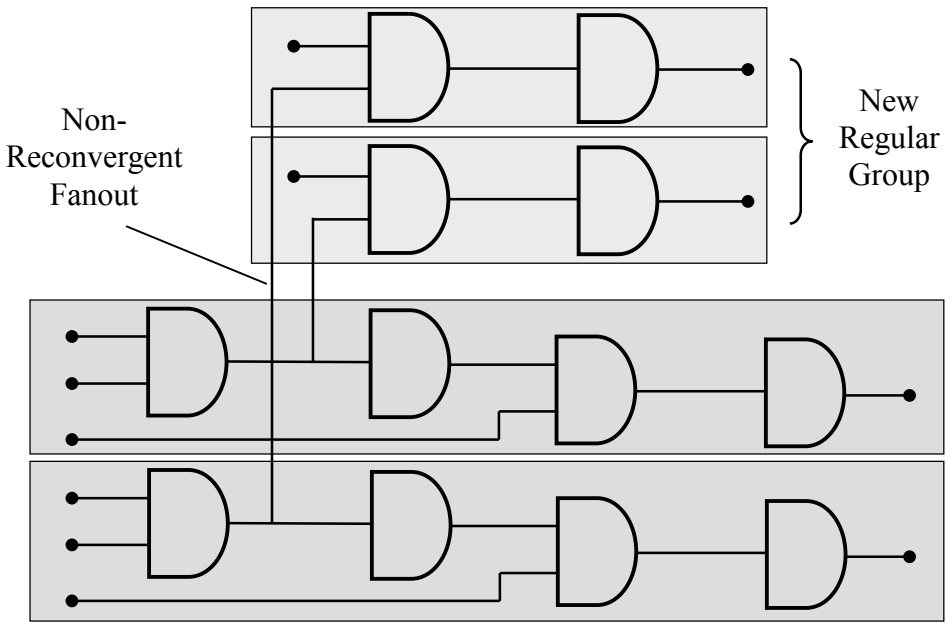


Figure 4.9: Partitioning of non-reconvergent fanouts

Design	*Gates*	w_{min}	Reg_s	h_{max}	h_{avg}	w_{avg}	*CPU*
S1423	653	2	49.2%	12	6.2	4.5	0.6 s
S1423	653	10	19.0%	4	4.0	14.2	0.5 s
S38417	11890	2	92.2%	88	6.2	28.1	7.2 s
S38417	11890	10	78.4%	40	5.8	144	6.5 s
S38584	14489	2	82.8%	160	4.2	12.0	9.4 s
S38584	14489	10	62.3%	8	11.0	122	9.5 s
ALU	2229	2	69.5%	32	6.4	4.2	1.8 s
ALU	2229	10	32.3%	16	10.1	31.7	2.0 s
CRM	1825	2	71.2%	64	5.1	4.2	2.4 s
CRM	1825	10	32.8%	32	18.2	14.2	2.2 s
AST	5514	2	84.0%	16	3.2	28.4	3.7 s
AST	5514	10	62.4%	8	2.5	660	3.5 s
Condor	4522	2	64.4%	36	9.5	18.2	2.9 s
Condor	4522	10	46.3%	36	7.8	52.0	2.5 s

Table 4.3: Structural regularity extraction results

During the regularity expansion, the slices of a regular group can contain non-reconvergent fan-outs. In this case, we have to decide whether the part of the slice growing along the extra fanout(s) should be part of the current regular group, or whether it might represent a new regular group that simply fans out from within the initial group. In general, if a fanout within all slices of a group is non-reconvergent (not even reconverging outside the regular group), and the fanout-tree is large enough to create a second regular group, we will partition the original structure accordingly. This process is shown in figure 4.9.

In practice, many structures are not completely regular, for example the last or first bit of a data path is often different. To deal with irregular structures, we specify a user-supplied parameter that allows *n* irregularities in the slices of a bit-stage. If such an irregularity is encountered, the algorithm first attempts to find a similar gate, e.g. one with similar footprint or logic function of the gates in the other slices of the same bit-stage. If that fails, the gap in the irregular slice will be filled with a dummy placeholder to allow regular two-dimensional placement.

4.4.3 Experimental Results

Table 4.3 shows the amount of structural regularity as the percentage of gates within regular groups Reg_s, maximum and average group height h_{max} and h_{avg}, and the average group width w_{avg}. *CPU* shows the runtime on an IBM Power PC/140.

The results demonstrate that modern designs contain a large amount of structural regularity due to the repetition of bit-operations in data logic. However, only regular structures of a certain minimum size can be deemed useful regularity. For example, design *S1423* contains almost 50% regularity if only a group width of two is required, but rapidly degrades with larger widths. As a result, it contains a lot of small regular

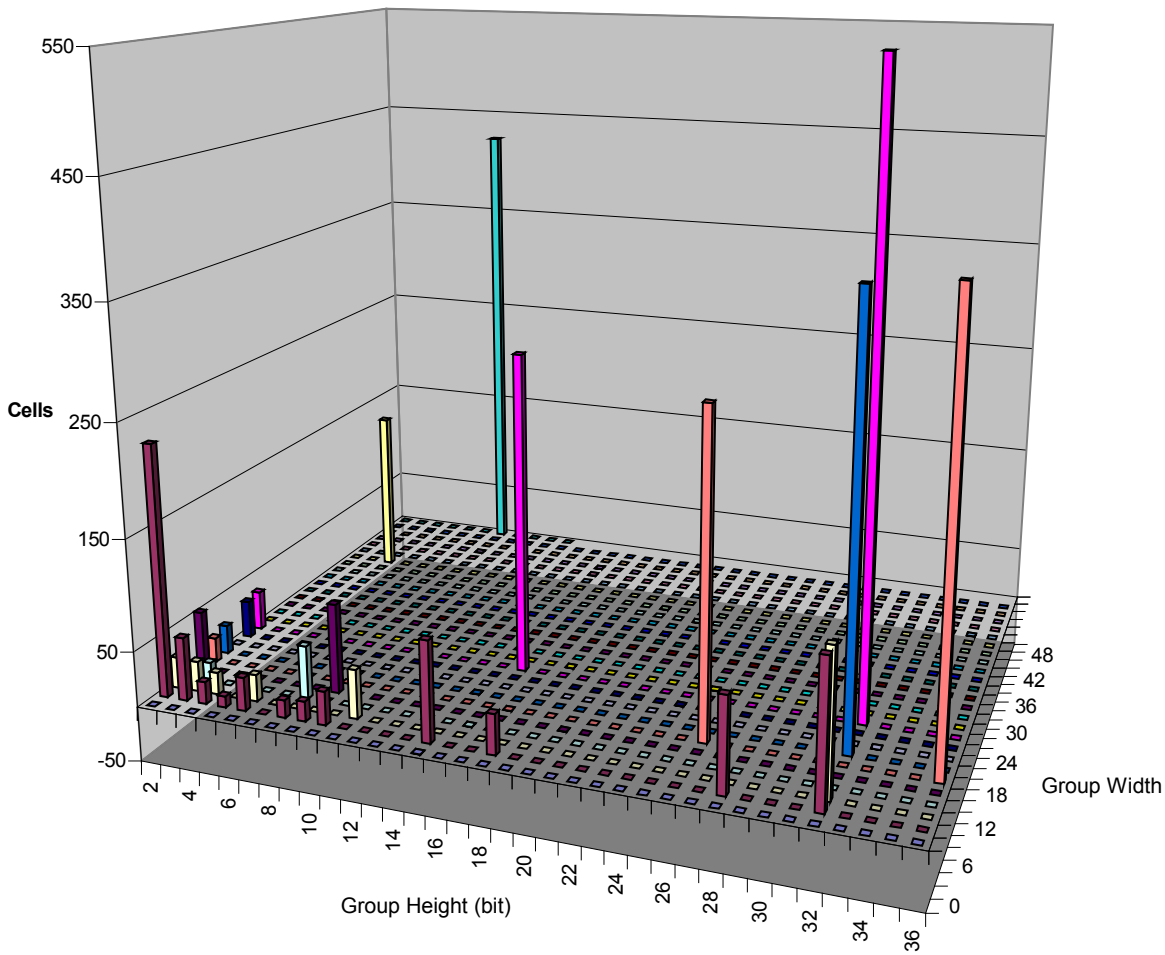


Figure 4.10: Distribution of regular groups for design *Condor*

groups that mostly represent random regular structures instead of larger useful datapaths. Clearly, the percentage of regularity alone is not a good measure for the quality of regularity. To assess the quality of the extracted regularity, a distribution of all regular groups is useful. Figure 4.10 shows the distribution of regular groups of design *Condor*. While a certain amount of regularity is contained within small groups (2x2 etc.), a significant part consists of large groups with 16, 32 or more slices.

4.4.4 Application of Regularity in Placement

After the extraction of structural regularity, the next step is its application in the placement process. Two-dimensional placement of regular structures in rows and columns generally yields a high packing density and short wire length and improves the area and delay of the final layout.

To exemplify this statement, consider the layout of a small microprocessor containing a significant amount of data logic. Table 4.4 gives an overview of the specification of the design. Reg_{func} and Reg_{struc} give the percentage of functional and structural regularity, with at least 4 slices and 4 stages.

Figure 4.11 shows the generic timing-driven placement of the circuit without using any regularity information. In contrast, consider the layout shown in figure 4.12. Here, the extracted regularity was used in the creation of the layout. Each regular group is placed as a two-dimensional array of rows (bit-slices) and columns (bit-stages). Clearly, the placement in fig. 4.12 is much more compact. For better illustration, the same chip image was used for both experiments. The empty space in fig. 4.12 clearly indicates the higher packing density of the utilized chip area. The generic min-cut based placement results in a close grouping of the data logic, but it is unable to identify its underlying structure. Even more important is the wire-length minimization. Consider the magnification of one of the regular groups in figure 4.13, showing its interconnections. One can clearly identify the control nets running across the width of the group, and the horizontal connections along the direction of the dataflow, providing the shortest

Design	*Gates*	Reg_{func}	Reg_{struc}	*#Groups*	h_{max}	*Cycle*
Condor	4522	58.2%	64.4%	28	36	4.15 ns

Table 4.4: Specifications of design *Condor*

possible route. As a result, the placement in regular groups yields a 0.8ns improvement of the critical delay, a tremendous gain that comes at a relatively low cost.

The actual regular placement process is not the subject of this dissertation, but a general overview of the procedure is given in chapter 7.

4.5 Conclusion

In this chapter, we have formally introduced the notion of functional and structural regularity. We have presented a fast functional regularity extraction algorithm of complexity O(m+n) where m is the number of vertices and n the number of edges in the network graph. We then showed that the usage of functional regularity is able to significantly speed up logic optimization by reuse of functionally equivalent slices of a regular group.

Thereafter, we extended our regularity extraction algorithm to identify structural regularity. We showed that regular structures produce high-density layouts with short wire length and improved timing during placement and routing. The main work on structural regularity extraction and its placement application was performed by Nijssen et. al., therefore we refer to his work [72, 73] for further reading.

Having shown the importance of considering regular structures, a number of questions remain. Since structural regularity is extracted on a technology-mapped and optimized network, i.e. before physical design, the question appears, how much of the initial regularity inherited from a high-level description actually remains after optimization. While the reuse of optimized bit-slices does also preserve regularity to some point, it does not allow optimization across the boundaries of the created hierarchy. In addition, hierarchical synthesis is not very well suited for technology-dependent optimization, where for example timing information must be propagated across the entire design. Therefore, one must tackle the problem of preserving and optimizing the amount of regularity in a design. All of these, and several other problems will be addressed in the following chapter.

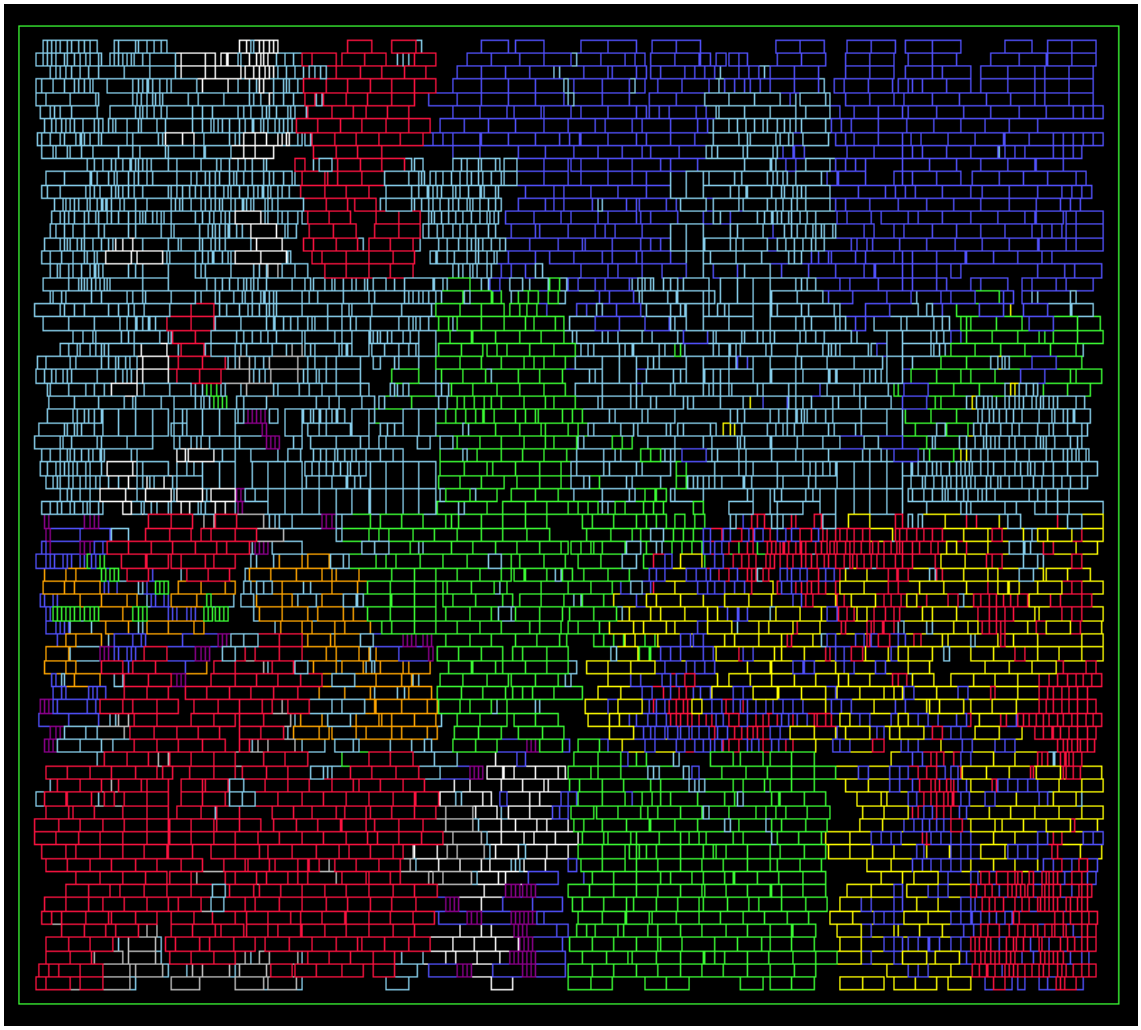


Figure 4.11: Generic placement of design *Condor*

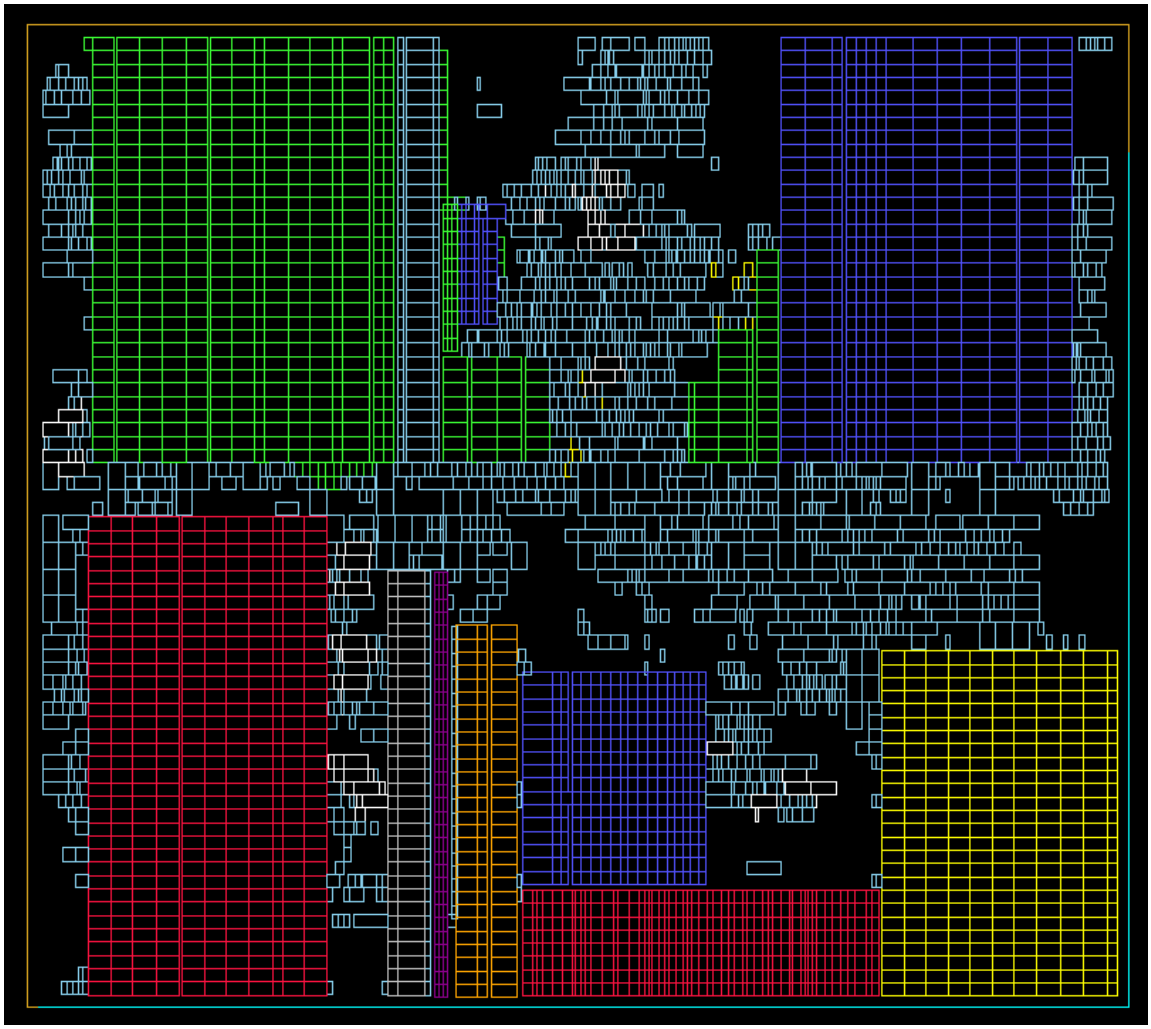


Figure 4.12: Regular placement of design *Condor*

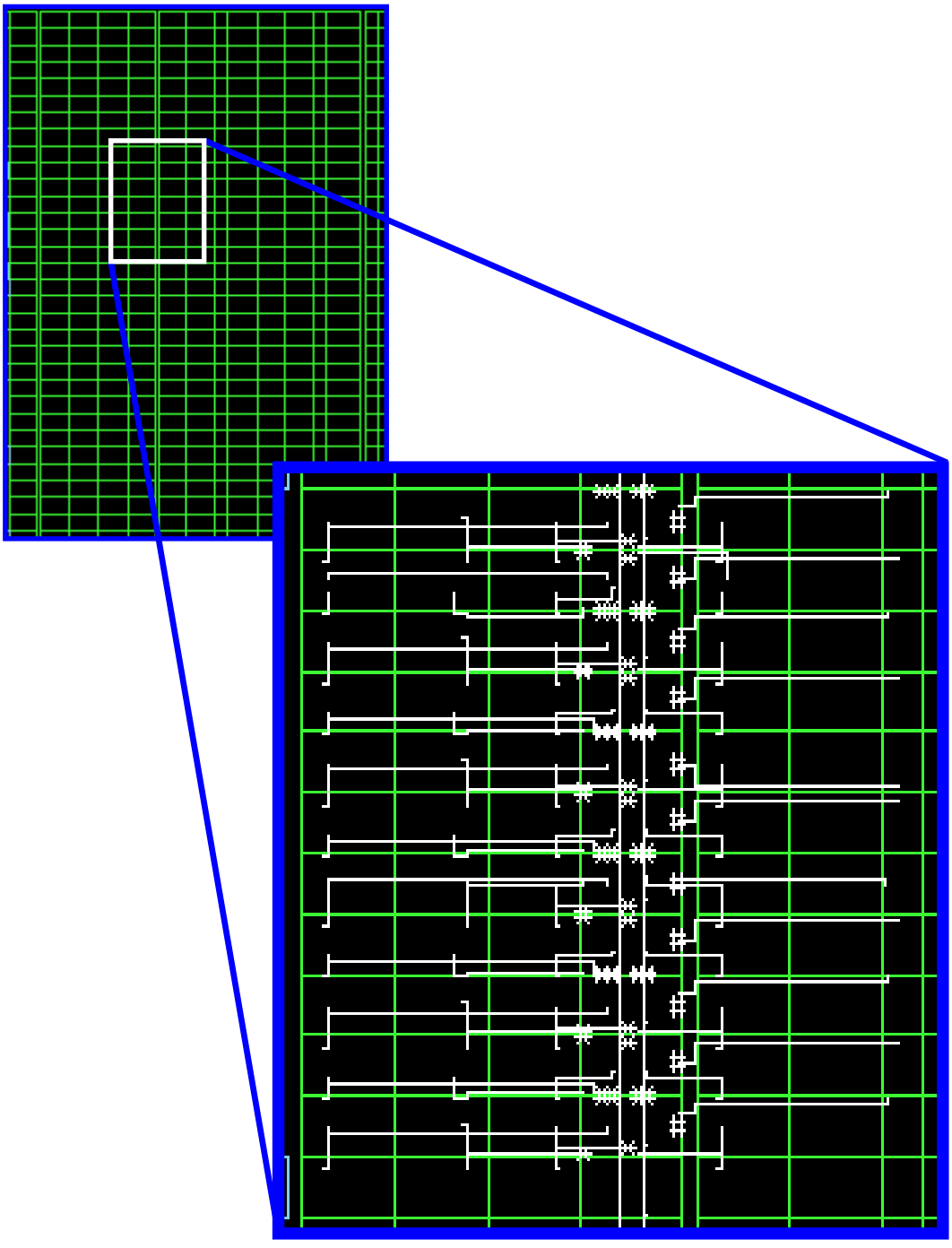


Figure 4.13: Interconnections within a regular group

5 Regularity Driven Logic Synthesis

In this chapter, we present a new and innovative logic synthesis approach using regularity information of a design to selectively apply transformations and globally guide the synthesis process.

Since traditional logic synthesis applies transformations without consideration of global design characteristics such as regularity and dataflow, it destroys a substantial amount of regular structures. In addition, due to the non-incremental nature of most logic transformations, synthesis relies vastly on the computationally expensive concept of trial and error application of transformations, a time-consuming process in the synthesis of large designs.

Our proposed approach addresses both shortcomings of traditional logic synthesis and describes a mechanism to speed up logic synthesis and preserve regularity. It selectively applies transformations to places with similar characteristics and to the same stage of a regular structure, introducing a notion of dataflow-aware synthesis.

Preservation of regular structures has tremendous advantages for the following physical design stages. It yields high-density layouts, shorter wiring length and improved delay. In addition, the layout becomes more predictable at an earlier design stage.

5.1 Motivation

It is widely acknowledged that generic logic synthesis destroys a substantial amount of structural regularity, particularly during logic minimization and technology mapping [18, 21]. Maintaining regularity has significant advantages to the subsequent physical design stages. In addition to improved delay and area, it simplifies the overall placement

task. In general, the layout of a regular design becomes more predictable at an early design stage. However, currently known approaches only focus on the extraction of regularity. The problem of maintaining regularity throughout the design flow has rarely been addressed [50, 56], and to date, no solutions have been presented. A commonly used concept to avoid destruction of regularity is to skip logic minimization and map the technology by manual assignment of library cells [28]. This process does not only demand a substantial amount of manual work, it is also unable to benefit from potential improvements by logic minimization algorithms and is therefore generally undesirable.

Future synthesis tools are expected to handle millions of gates in a reasonable amount of time. However, it is questionable whether traditional logic synthesis is able to satisfy these increasing demands. Due to the amount of available topological and functional transformations and their non-incremental nature, it is generally difficult and computationally expensive to select the best suitable subset of transformations and their respective sequence. Therefore, current synthesis methodologies mostly rely on the concept of applying trial and error sequences of transformations. This process, however, produces a tradeoff between efficiency and quality and generally does not yield optimal results. In addition, the achieved results vary depending on the actual design, hence the overall optimization is left to the experienced designer.

In the logic synthesis of large designs, it is very important to efficiently find places where to apply transformations to the design. When a transform succeeds at a certain place, it would be desirable to be able to quickly apply it at other places in the design with the same characteristics. Due to similarity of individual bit-slices in a datapath structure, the regularity properties of a design provide a framework to guide logic synthesis transforms, as will be shown in this chapter.

We present solutions to these shortcomings of conventional logic synthesis and describe a method that is able to speed up logic synthesis while maintaining regular structures. Since conventional logic synthesis is generally unable to understand and use any notion of regularity, transformations are applied without any notion of the dataflow or other global design information. In contrast, the presented methodology extracts regularity information of the design and selectively applies suitable transformations to places with similar regularity signatures. Through reapplication of successful transformations, a faster and more complex identification of a suitable subset and sequence of logic transformations is possible. In addition, more regularity is preserved, since identical transformations are applied to the same stage within a regular structure.

In contrast to generic logic synthesis, the resulting process is aware of global design characteristics, such as regularity and dataflow.

5.2 Concept of Drivers and Transforms

The proposed synthesis algorithm utilizes the concept of *drivers* and *transforms* within the framework of an electronic design automation tool [91].

A *driver* is defined as the part of an algorithm that decides where and how to apply an action in a design, i.e. it provides the guidance to the optimization. In contrast, we define a *transform* as the part of an algorithm that applies an action. A driver iterates through the design, determines a beneficial sequence of transformations and selectively applies them to a set of logic gates. This concept establishes the basis for controlling the synthesis flow and applying actions dependent upon global design characteristics such as regularity, symmetry and dataflow.

We use two primary groups of drivers, *generic drivers* and *timing drivers*. *Generic drivers* iterate through the design and apply a set of transformations to all gates or nets in the network. In some cases, it is desirable to process gates or nets in a specific order. To address this need, *levelized drivers* are used to process gates or nets in increasing or decreasing topological order. Generic drivers are mainly used during technology-independent optimization and technology mapping.

The second group of drivers, *timing drivers*, uses timing information of the design to improve cost functions like critical delay, area or power consumption. In general, complicated delay rules reduce the ability of an algorithm to estimate the effect of changes to the network on the delay. Thus, *timing drivers* apply a number of transformations, collect cost and benefit data, and undo the transformations. Drivers may try different transformations and places in the network before determining the most effective transformation and its place of application. In general, this *trial and error* sequence of applying transforms is computationally very expensive.

We define two different types of timing drivers, *critical* and *non-critical drivers*. The *critical driver* applies a set of transformations to the critical path in the network to reduce the critical delay of the design. In contrast, the goal of the *non-critical driver* is to optimize cost functions along non-critical paths, for example area and power reduction. Timing drivers are applied during technology dependent optimization such as delay and area reduction.

5.3 Regularity Model

The proposed regularity driven synthesis methodology utilizes *global* and *local regularity metrics* to drive the synthesis process, as illustrated in the following sections.

5.3.1 Global Regularity Metric

Global regularity refers to structural regularity that improves cost functions, such as area and delay during the physical design process. As we determined through extensive experiments, it is essential to identify and maintain structural regularity at an early design stage, as particularly technology independent optimization and technology mapping destroy a substantial amount of regularity. In fact, the area and delay optimization criteria used during these optimization steps are often inaccurate since no physical and structural design information is utilized. In many cases, the actual placement area and delay of a highly regular design can be significantly smaller. Therefore, the use of structural regularity as additional optimization criteria during logic synthesis improves and yields a more accurate prediction of the final physical layout.

To measure the amount of structural regularity that effectively improves area and delay cost functions in the final physical layout, we introduce the *physical regularity index* as follows:

$$RI_p = \frac{1}{n_{grp} + n_{nr}} \cdot \left(n_{nr} + \sum_{i=1}^{n} \left(S_{grp_i} \cdot \frac{2 \cdot \sqrt{S_{grp_i}}}{h_i + w_i} \right) \right) - 1 \qquad (5.1)$$

In the above equation, n_{grp} is the number of regular groups while n_{nr} is the number of gates not within a regular group, i.e. the number of objects with group size 1. Hence, the first product term of the equation measures the total number of objects in a design. S_{grp} is the size of a regular group, i.e. the product of its height h and width w. The second product term of the equation represents the size of all non-regular objects, i.e. all gates not within a regular group plus the size of all regular groups, multiplied by a factor that considers the shape of each regular group. Thus, the regularity index is simply the average weighted size of an object in the design. To normalize the index, we subtract 1 such that a design with no regularity (average weighted size is 1) has a regularity index of 0 (i.e. no regularity). The shape of a group, i.e. its height versus its width is very important during placement. It is obvious that a group which is either substantially wider than higher, or vice versa, is more difficult to place in the layout.

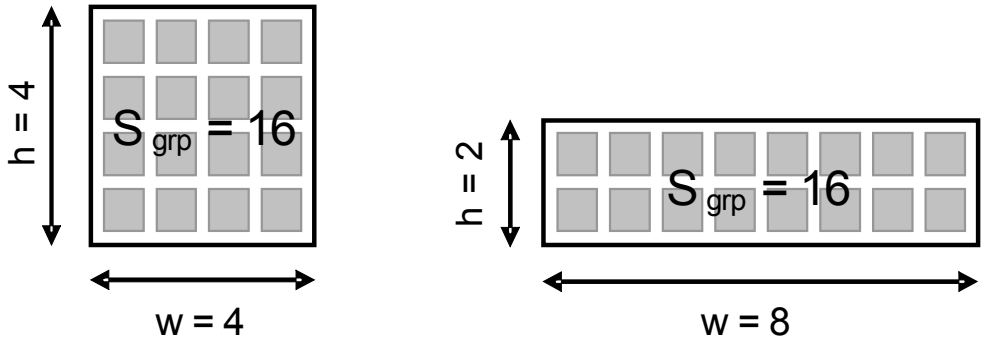


Figure 5.1: Shape of regular groups

Most placement algorithms employ vertical and horizontal cuts or partitioning to move objects in the placement area. The larger the height or width of an object, the more likely it will become frozen at an early placement stage, often stuck in a less than optimal position. In addition, the larger perimeter of such a group complicates routing. In figure 5.1, while both groups have the same size, the sum of height and width of the right group is clearly larger than that of the left group, and in conclusion, it is more difficult to place. A square has the minimum sum of height and width $2 \cdot \sqrt{S_{grp}}$ of any rectangle. For simplicity, we assume the size of each gate to be identical. In general, the physical size of a logic gate g can either be defined by the area footprint of its library cell, or alternatively, by the number of literals for an unmapped gate. The higher the regularity index, the higher is the estimated benefit during physical design phases. A value of zero indicates that no usable regularity exists.

5.3.2 Regularity Signatures

We utilize the concept of regularity signatures that serves two primary purposes. First of all, regularity signatures are used to speed up the synthesis process by applying transformations to objects with similar characteristics, and secondly, to observe and control local changes made by transformations during synthesis to maintain global regularity.

We use the same underlying graph model introduced in our regularity extraction algorithm. Given a circuit C, consisting of a set G of gates, a set N of nets and a set of pins $P \subset G \times N$, we assign a *gate type* γ to each gate g and a *terminal type* τ to each pin p. In addition, we define a *net type* ν for each net n. A net type is characterized by timing, load and capacitance of n, among others.

Definition 5.1: Let Γ, T and M be the set of gate, terminal and net types, respectively. A *labeled directed graph D(V, E, Γ, T, M)* of a circuit C consists of a set V of vertices, a disjoint set $E \subseteq V \times V$ of edges, and mappings H: $V \mapsto \Gamma$, I: $E \mapsto T$ x T, and K: $E \mapsto N$.

We define the *regularity signature* of net n denoted *RS(n)* as the labeled directed graph consisting of the source vertex v, its *immediate predecessors* and connecting edges.

Definition 5.2: The *regularity signature* of a net *n,* denoted *RS(n)* is the labeled directed graph consisting of its source vertex v within a structurally regular group R, the set of its immediate predecessors V_p within R and connecting edges $[(v \times V_p) \cup (V_p \times v)] \cap E$.

Hence, a regularity signature represents the preceding logic levels that compute the value of net n within a structurally regular group R (as previously defined in section 4.4). It is used to determine "similar" places in the design, for example during timing optimization, and to preserve the structural regularity during logic optimization, as detailed in the next section. Regularity signatures are equivalent if their labeled directed graphs are strongly isomorphic.

In addition, we need a measure to control and compare the changes made by a transformation to the network. Therefore, we introduce a more complex signature that includes all logic levels affected by a particular optimization algorithm, called an *adaptive regularity signature*. It is dependent on the particular transform T and is defined as follows.

Definition 5.3: Let D be the labeled directed graph of the original circuit, and D' be the respective graph after applying transformation T. Then, an *adaptive regularity signature* is defined as $RS_a = \mathrm{D} \ominus \mathrm{D}' = (D - D') \cup (D' - D)$.

I.e. an adaptive regularity signature is defined as the symmetric difference between D and D'. Hence, it is an exact measure of the effect of transform T on a particular design. By including additional information to terminal, gate and net type, we can differentiate the following types of regularity signatures: *logic*, *timing, area* and *power signatures*. Signature types are combined to satisfy the specific requirements and goals of the optimization process. Additional signature types can be defined, depending on the particular requirements.

5.3.2.1 Logic Signatures

The *logic signature* of a net *n* represents the logic function *f* of a regularity signature. The labeled directed graph representing a logic signature contains the logic function within gate type γ, and the pin function of a pin within terminal type τ.

In addition, a logic signature *LS* contains information about the bit-stage of each gate to account for the order of the dataflow. Logic signatures are applied during technology dependent and independent transformations, they are the basic building block of every regularity signature.

5.3.2.2 Timing Signatures

A *timing signature* contains information about the arrival and required arrival times at the pins *p* of gates *g*. The required arrival time *RAT* is associated with a pin, and is therefore modeled in the terminal type τ. To obtain identical signatures for identical structures, we model a unique wire delay per net, hence the arrival time *AT* is identical for all input pins attached to a net. Therefore, the arrival time is modeled in the net type.

In addition, timing signatures may contain information about the slew and load at the source and sink, respectively. Time signatures are equivalent, if all its timing characteristics vary within a given boundary. Timing signatures are usually used during technology dependent optimization, e.g. to identify similar places in critical paths.

5.3.2.3 Area Signatures

Similar to timing signatures, an *area signature* is calculated by considering the area ***A*** of all gates *g* within the regularity signature.

Area signatures are considered equivalent, if the area of its gates *g*, modeled in gate type γ, varies within defined boundaries. They are applied during technology dependent optimization.

5.3.2.4 Power Signatures

A power signature describes the power properties of a net *n* and is characterized by

- the power consumption *P* of gate *g*,
- the net capacitance *c*,
- the switching factor *s*.

The power consumption *P* is modeled in gate type γ, the net capacitance *c* and switching factor *s* within net type τ.

Similar to time and area signatures, power signatures are defined as equivalent, if all of its characteristics vary within a defined limit.

5.4 Regularity Synthesis

This section outlines how the previously introduced regularity model is applied within a regularity driven logic synthesis flow.

Within the framework of drivers and transformations as described in section 5.2, we introduce a *generic regularity driver*, and a *timing regularity driver* to guide the synthesis process.

5.4.1 Generic Regularity Driver

The generic regularity driver is applied during the first step of logic synthesis, logic minimization and technology mapping. Our main objective at this design stage is to maintain global regularity. To achieve this goal, we employ the following algorithm, as shown in figure 5.2.

At first, we identify structural regularity using our efficient regularity extraction algorithm. Gates within regular groups are processed stage by stage, in the order of the dataflow, applying a transform to all gates within a stage. Thereafter, adaptive regularity signatures, based only on logic signatures since we employ logic independent optimization, are calculated for all gates within the stage. The signatures also contain information about the bit-stage of each gate. To maintain existing regularity, signatures only include gates that are part of a regular structure, since changes to the network outside a regular group are allowed. If signatures of gates within the same stage are not

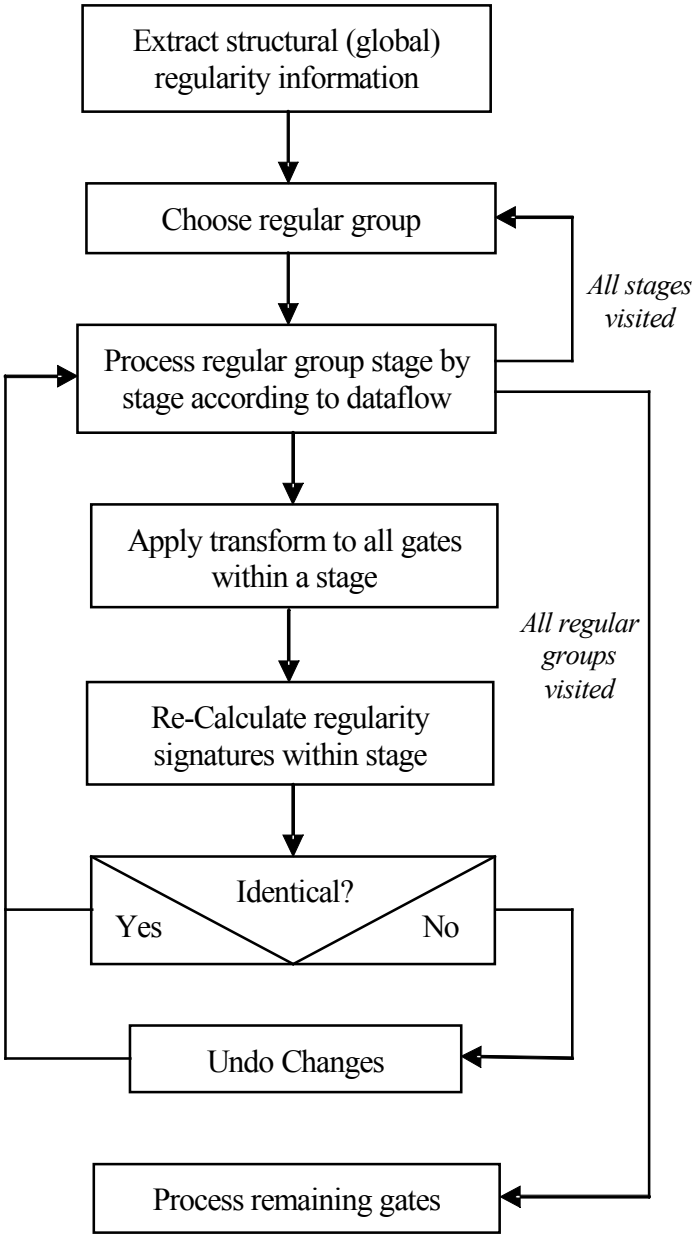


Figure 5.2: Generic regularity driven synthesis algorithm

identical, the transform will be undone. This process continues until all regular groups have been processed. Thereafter, all remaining gates are processed.

The synthesis process identifies regularity at an early stage, applies transforms according to the dataflow simultaneously to gates within one stage and ensures that

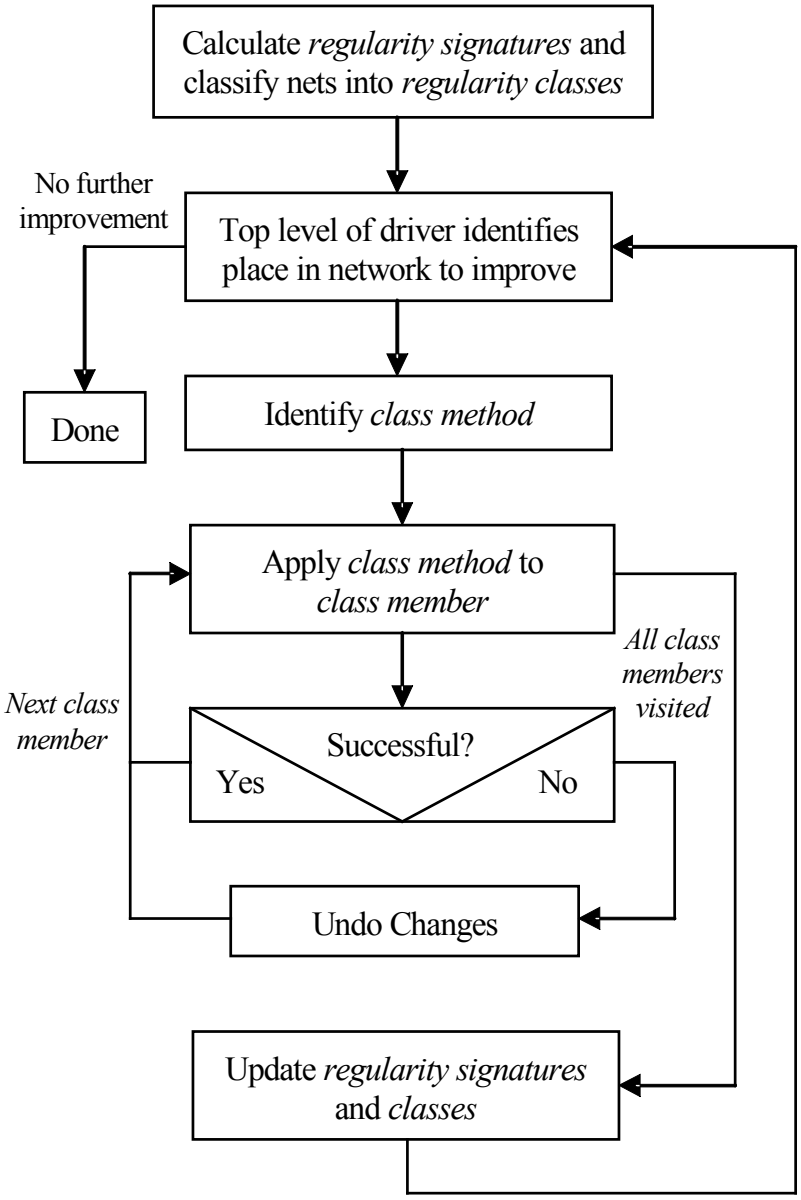


Figure 5.3: Timing regularity driven synthesis algorithm

regularity signatures remain identical. Therefore, it effectively preserves more regular structures than a traditional synthesis process, which applies transforms randomly and in arbitrary order.

5.4.2 Timing Regularity Driver

Technology dependent optimization such as delay and area reduction is, compared to logic independent optimization, a computationally significantly more expensive process. Due to complex delay rules, transformations are repeatedly applied and undone utilizing a trial and error concept, until the most beneficial place and transformation has been identified. In general, this procedure yields unsatisfactorily high turn-around-times, particularly for very large designs.

The proposed regularity driven synthesis methodology uses local regularity signatures to identify objects with similar characteristics in the design, and selectively applies transforms. The overall procedure is detailed in figure 5.3.

In the first step, regularity signatures, i.e. timing and logic signatures, and depending on the optimization goal, power and area signatures, are calculated for all nets *n* in the design. Using the signatures, the set of nets is partitioned into equivalence classes, referred to as *signature classes*. The top level of the optimization process determines the particular part of the design to work on, e.g. the critical path to optimize delay, or area reduction on a non-critical path. A sequence of transformations is applied to a place in the network, until a set of transformations which achieves a favorable or

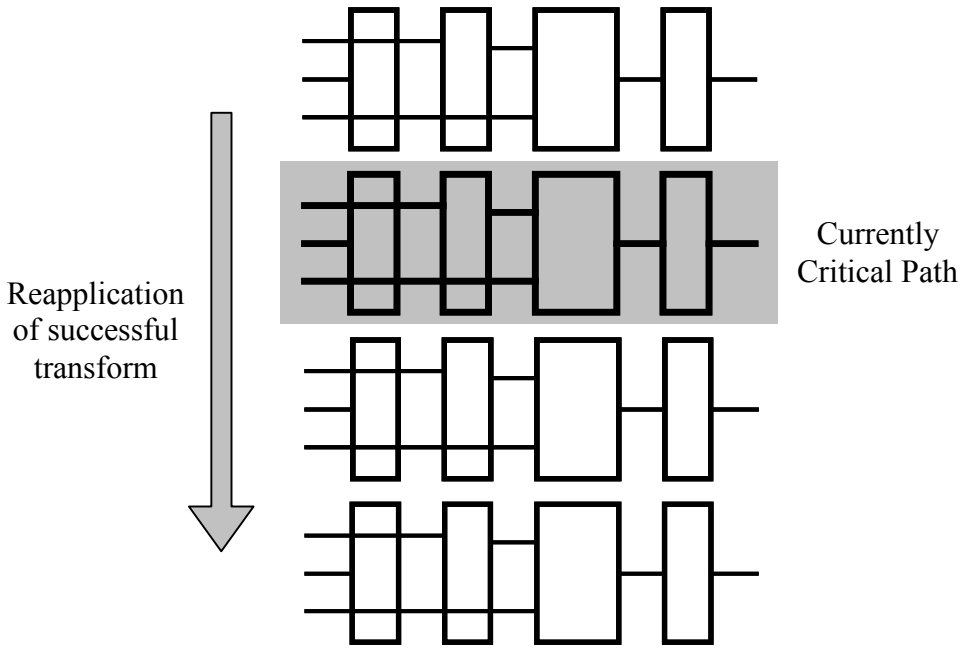


Figure 5.4: Reapplication of transforms within a regular group

Design	*Gates*	RI_p *orig*	*Reg orig*	RI_p *std*	*Reg std*	RI_p *rd*	*Reg rd*	*Lit orig*	*Lit std*	*Lit reg*
S1423	653	0.41	32.0 %	0.34	30.7 %	0.52	35.4 %	1262	1242	1204
S38417	11890	3.08	84.0 %	1.11	63.1 %	1.99	82.8 %	23496	21215	21320
S38584	14489	0.74	53.2 %	0.69	47.3 %	0.83	49.7 %	30136	25155	25480
Condor	4522	1.39	63.5 %	0.83	52.3 %	1.73	72.4 %	15905	15817	15847
Solar	21728	1.02	55.6 %	0.98	51.2 %	1.12	54.9 %	54223	56143	57098
Max2	10047	0.97	56.4 %	0.62	46.9 %	0.84	54.1 %	27619	28432	28501
Pipe	84292	1.31	69.1 %	1.02	56.2 %	1.35	70.1 %	309315	308266	309373
Micro	2575	1.27	62.1 %	0.88	54.7 %	2.70	75.4 %	8851	8515	8235
Stealth	5876	1.20	60.3 %	1.10	58.7 %	1.23	60.8 %	17753	17054	17040
Redox	77215	0.67	70.8 %	0.61	68.2 %	0.66	70.2 %	195412	192713	193192

Table 5.1: Technology independent optimization results

the best solution, called a *class method*, has been identified. This *class method* is applied to each *class member*, and, if unsuccessful, any changes in the network are undone. Thereafter, the top level continues optimization on the next design part, for example the newly identified critical path. This procedure continues until a particular optimization goal has been achieved.

For example, consider the regular group shown in figure 5.4. Regularity driven optimization of the critical delay in the network would first identify and optimize the most critical path. Assuming the most critical path is part of a datapath, all other paths contained in the same datapath are members of the same class, and will be optimized using the same set of transformations.

The application of transformations to similar parts of the design speeds up the overall optimization process. In addition, since transformations are applied to all class members, regularity is widely preserved.

5.5 Experimental Results

The presented regularity driven synthesis methodology was implemented in C++ within the framework of the logic synthesis tool BooleDozer™ [91].

Table 5.1 presents results of the technology independent optimization that performs logic restructuring on several benchmark and actual production designs with a high amount of datapath circuitry. *Condor*, *Micro* and *Stealth* are microprocessor kernels, *Solar* and *Max2* are encryption circuits, *Redox* and *Pipe* are communication controllers. Shown are the previously introduced physical regularity index RI_p (defined in eq. 5.1), the amount of gates within regular groups (*Reg*), and the literal count (*Lit*) of the original non-optimized (*orig*), conventionally optimized (*std*) and the regularity driven optimized (*reg*) designs. The latter uses the generic regularity driver from section 5.4.1.

It is evident that the amount of physical regularity measured by RI_p is significantly improved, ranging from 8.2% for the *Redox* chip to over 200% for the *Micro* design. On average, an improvement of more than 57% has been shown. At the same time, the literal count remains almost identical, increased only slightly by 0.4%.

Furthermore, run times are increased by roughly 8% on average, as regularity extraction and signature calculation is performed in addition to logic optimization.

Interestingly, the percentage of regular gates during optimization only decreases by a small amount, while the actual physical regularity measure often decreases substantially. This is explained by the fact that large regular groups are split into several

Design	*Gates*	*Delay std*	*Delay rd*	RI_p *std*	RI_p *rd*	*Area std*	*Area rd*	*CPU*
Condor	4669	4.15 ns	4.15 ns	1.28	1.32	16022	16010	-17 %
Solar	21838	1.32 ns	1.32 ns	1.03	1.05	55102	54811	-15 %
Max2	10453	2.20 ns	2.21 ns	0.80	0.82	27499	27380	-23 %
Pipe	100223	4.57 ns	4.57 ns	1.14	1.18	322499	322831	-29 %
Micro	3015	2.11 ns	2.08 ns	1.06	1.12	8419	8522	-19 %
Stealth	5992	3.52 ns	3.52 ns	1.02	1.02	18653	18701	-11 %
Redox	90982	6.33 ns	6.30 ns	0.62	0.65	210549	211431	-32 %

Table 5.2: Delay optimization results

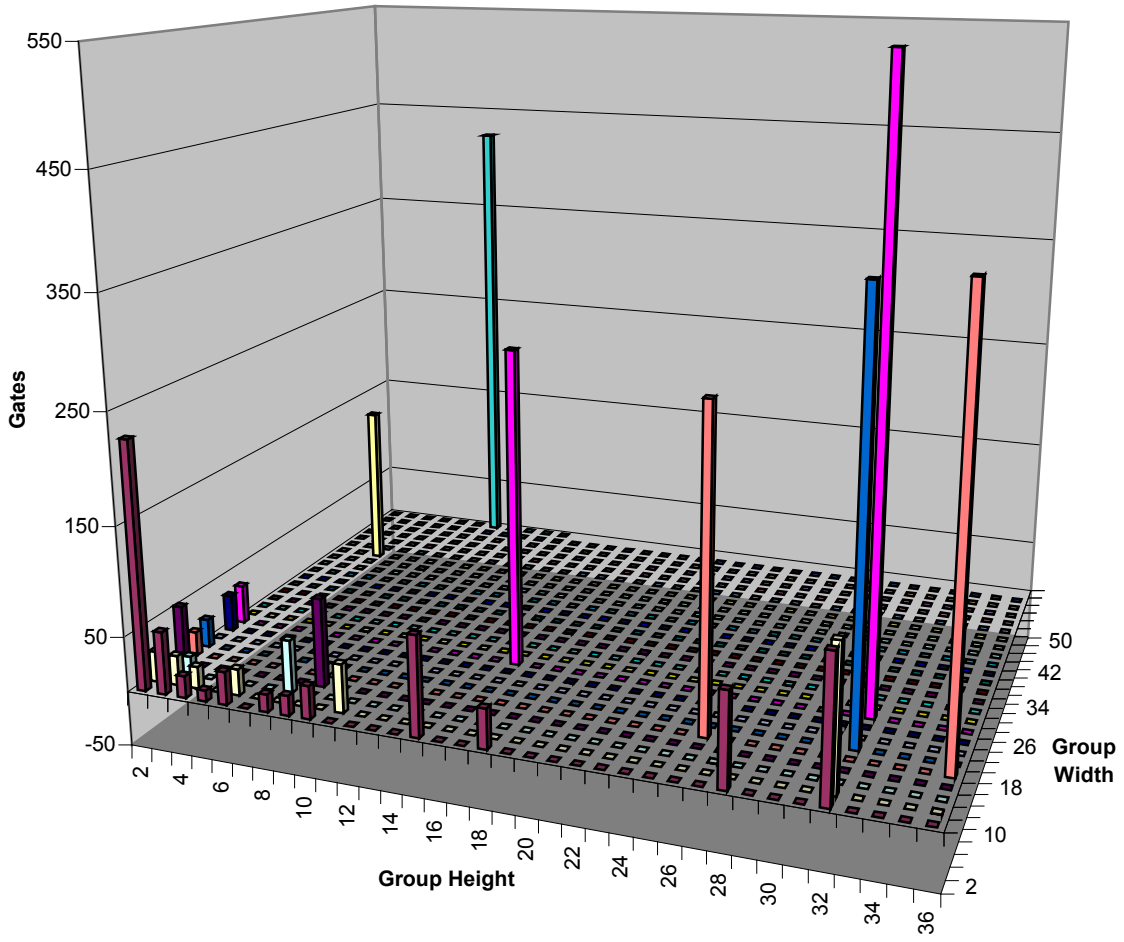


Figure 5.5: Regularity distribution for design *Condor* (regular synthesis)

small ones during logic optimization. Furthermore, the physical regularity index after logic optimization is in some cases higher than the original one. This is due to the fact that logic outside regular groups is minimized to a higher extent than regular logic because non-regular transformations to regular groups are being rejected.

Figure 5.5 and 5.6 show the distribution of regular groups by group height and width for design *Condor* after regularity driven and conventional logic optimization, respectively. It is clearly visible that the regularity optimized design contains several large regular groups, while the conventionally optimized design only features smaller and less useful regular groups.

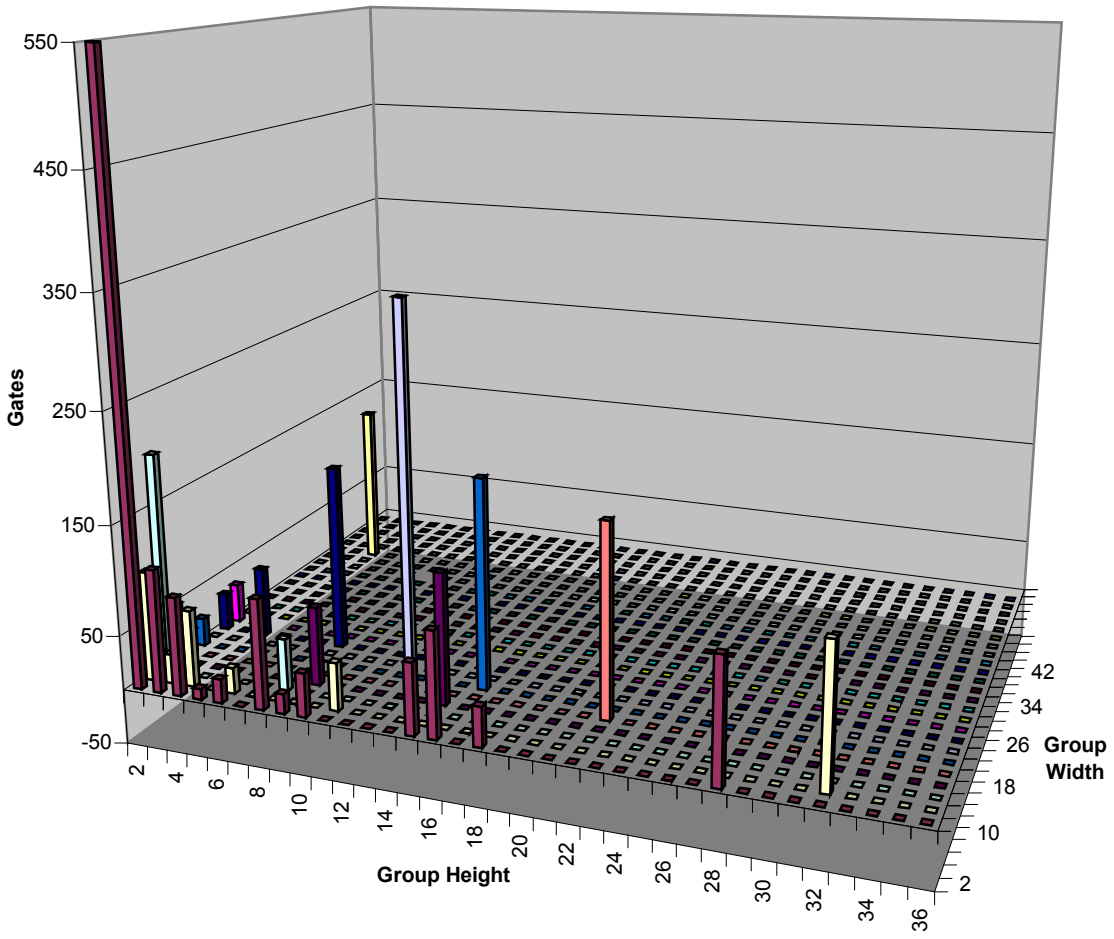
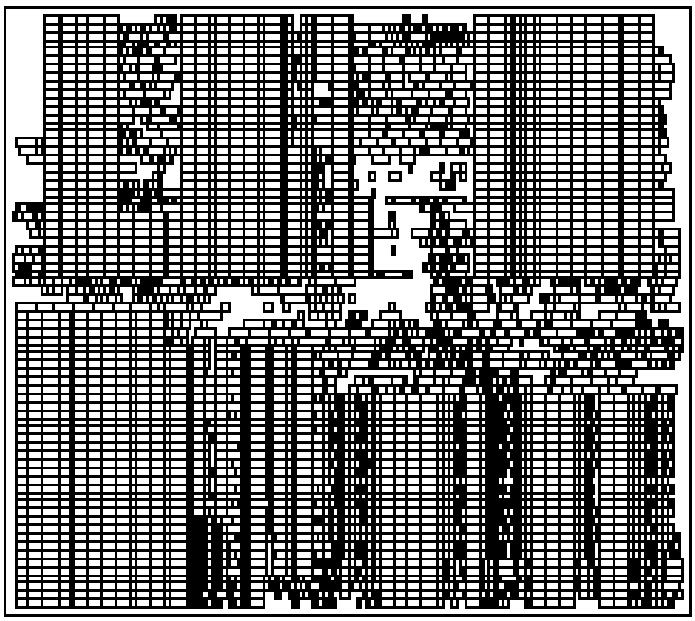


Figure 5.6: Regularity distribution for design *Condor* (conventional synthesis)

Table 5.2 summarizes the results for the delay optimization of the technology mapped designs. Shown are the critical delay, physical regularity index and area of the conventionally and regularity driven optimization. The main goal of regularity driven optimization at this point is the improvement of run-time by reapplication of successful transformations to points with similar characteristics. On average, a run-time improvement of approximately 21 % has been shown, at identical or better delay and area. Figure 5.7 shows the final placement of design *Condor* utilizing the available regularity during datapath placement. All non-regular gates have been placed with a conventional min-cut placement algorithm. The placement of the regularity optimized design shows a significant amount of structural regularity that has been placed as two-dimensional structures. As shown in chapter 4, a smaller, more compact layout,

a)



b)

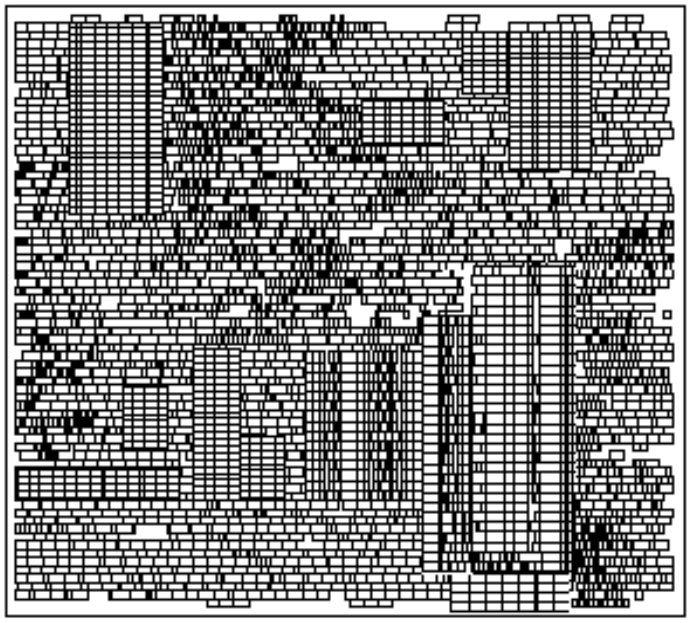


Figure 5.7: Layout of design *Condor*, a) regularity driven, b) conventional optimization

achieved due to the preservation and exploitation of the regularity, also results in faster circuits. In contrast, the conventionally optimized design cannot benefit from the same

amount of regularity, as most of it has been destroyed during aggressive logic restructuring algorithms. The regularity optimized design, shown in figure 5.6a, shows an improvement of more than 8% in critical delay over the conventionally optimized design, resulting from shorter wiring length. In addition, the packing density of the regularity driven optimized design is significantly higher, as shown in the final layout.

5.6 Conclusion

The proposed logic synthesis methodology combines regularity information of the design with a driver-transform concept utilizing global design information to drive local transformations in the synthesis process. It has been shown that regularity driven synthesis is able to speed up the synthesis process and improve synthesis quality while maintaining design regularity. Due to the re-application of previously identified transforms to identical places in the network, more attention can be given to the careful selection of a good set of transforms to achieve higher optimization results. In addition, reuse of transforms yields a faster optimization process. Therefore, it is particularly suitable for the challenges imposed by the synthesis of very large designs. In addition, the preservation of regularity improves the final layout significantly.

6 Layout Aware Synthesis

In this chapter, we present novel algorithms to effectively combine physical layout and early logic synthesis to improve overall design quality.

With the increasing complexity of designs, the traditional separation of logic and physical design leads to sub-optimal results as the cost functions employed during logic synthesis do not accurately represent physical design information. While this problem has been addressed extensively, the existing solutions apply only simple synthesis transforms during physical layout and are generally unable to reverse decisions made during logic minimization and technology mapping, that have a major negative impact on circuit structure.

In our novel approach, we propose congestion aware algorithms for layout driven cube extraction and technology mapping to effectively decrease wire length and improve congestion. These logic synthesis steps have a large impact on congestion because the significantly determine the structure of the netlist. In addition, to improve design turn-around-time and handle large designs, we present an approach where synthesis partitioning and placement clustering coexist, reflecting the different characteristics of logical and physical domain.

6.1 Introduction

One of the most important problems in the design of VLSI circuits is the interaction of logical and physical domains. While this problem has been addressed extensively [31, 90], and various physical synthesis flows exist (as shown in chapter 3), the existing solutions employ only trivial logic synthesis transforms late in the design process, *after* the circuit has already been optimized and technology mapped based on inaccurate cost

functions, namely literal count and gate-based path delay, not reflecting the physical properties of the design. Existing layout driven logic synthesis algorithms [75, 77] only utilize the layout information for improved timing information using estimated wire delay and minimizing area using cell and estimated wiring area. However, reducing overall congestion for better routability in conjunction with timing and area optimization has not been addressed. In [76], Pedram et al. propose a decomposition and factoring algorithm based on a relatively simple one-dimensional input location assignment. Instead, we present an exact and a greedy algorithm to perform decomposition based on a two-dimensional input location model.

In addition, the increasing size of designs leads to infeasible turn-around times. This particularly applies to the field of logic synthesis. Many logic minimization algorithms, e.g. kernel extraction, among others, are *NP complete*. Effective approaches for multi-level clustering have been proposed in [19, 44, 45], and for logic partitioning in [30]. To incorporate this into a layout driven synthesis flow, *simultaneous* logic partitioning (for logic optimization) and clustering (for placement) is a necessity.

To solve these problems, we have developed algorithms employing physical design information early during logic independent optimization. Specifically, we create an initial placement of the technology-independent netlist and apply the thereby gathered layout information during logic minimization and technology mapping. We utilize the geometric location of the objects in the netlist, for example the location of a signal's origin, and the estimated wire length to perform congestion aware decomposition and technology mapping. By decomposing gates depending on the location of their sources and grouping gates that are located close to each other during technology mapping, we are able to reduce total wire length and overall congestion by distributing the wiring instead of concentrating it in a small area. Furthermore, to handle the increasing complexity of designs, we use a logic partitioning algorithm that identifies and groups reconvergent signal regions based on corollas [30]. In addition, we employ clustering for placement to reduce the additional complexity introduced by the typically greater number of objects in a technology-independent netlist. Furthermore, clustering generally improves overall congestion as high connectivity is grouped in clusters resulting in a more even distribution over the entire layout area.

The main contributions of this chapter are congestion aware decomposition [61] and technology mapping algorithms [62] using geometric information coupled with effective simultaneous logic partitioning and placement clustering to form an integrated synthesis and placement flow.

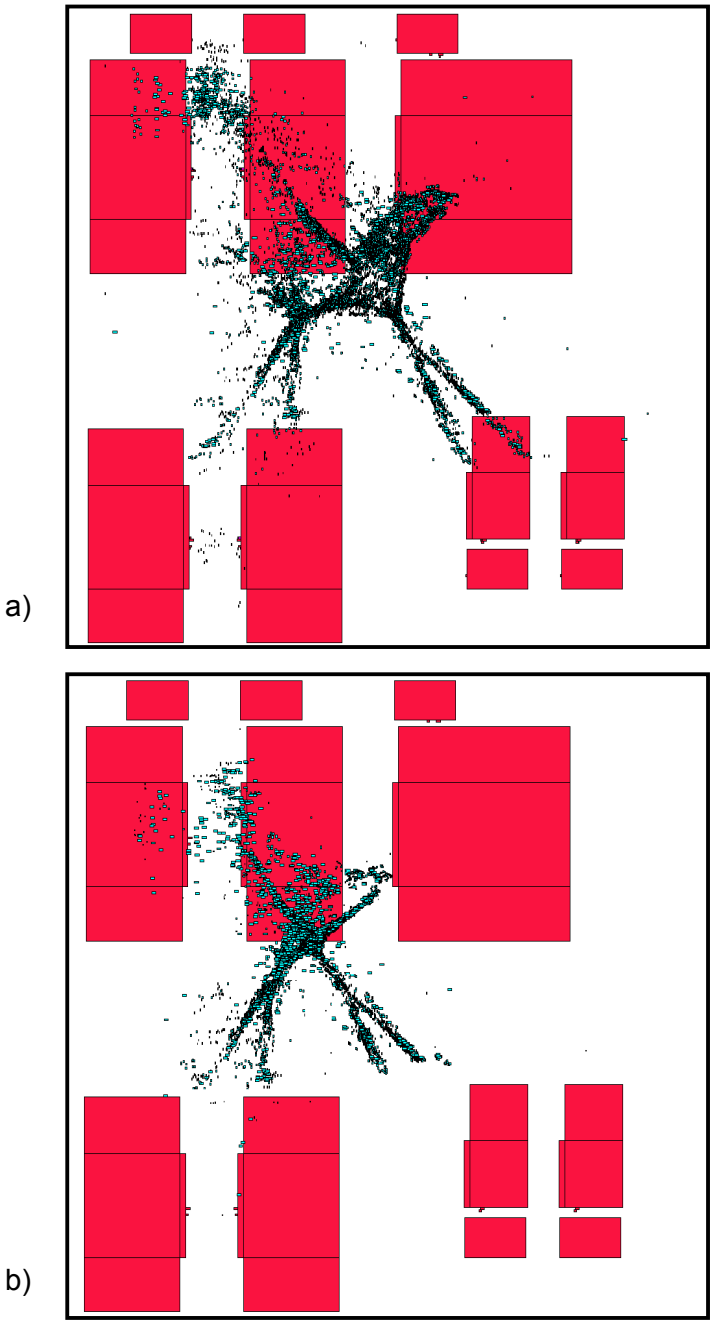


Figure 6.1: Comparison of a) original circuit and b) technology-independent placement

6.2 Technology-Independent Placement

To effectively combine logic synthesis and placement, we create an initial placement of the technology-independent netlist and use the placement coordinates of the objects to improve synthesis transformations. For this purpose, we use a quadratic placement algorithm with quadri-section based on the conjugate gradient method to minimize quadratic net length [96]. Contrary to heuristic move-based approaches, e.g. the min-cut algorithm, this type of placement algorithm is relatively stable with respect to local changes in the net list. This is a necessity because the structure of netlist might change considerably during logic minimization. To accurately represent technology-independent cells, we create a physical description based on the actual technology target library. The placement algorithm uses a density function that allows the circuit to be spread across the layout image accordingly, which can be tuned to achieve an accurate placement representation of the technology-independent circuit that provides a good estimation of the actual location of the objects in the final technology-dependent implementation.

In general, using a quadratic placement algorithm, the objects would end up all on top of each other in the center of the layout area. Therefore, the actual layout is mostly determined by the position of the primary inputs and outputs along the chip image, which remain fixed at all times, and by the individual cell areas. As an example, consider the placement of a technology-mapped netlist, shown in figure 6.1a, in comparison with the placement of the technology-independent cells, shown in figure 6.1b. Clearly, the two layouts look very “similar”. Of course, this is a subjective metric and is only used for illustration purposes. To actually compare two placements, we calculate the difference of the locations of the individual objects in the two layouts. Since the name of a net usually does not change during technology mapping, i.e. nets might be removed or created, but most nets remain, we can find many reference points in the design by comparing the locations of the source gate of the same net in both layouts. We then seek to minimize the sum of the differences of all reference locations, and also its standard deviation. It turns out that indeed, most differences in the layout are only of local nature. Additionally, with shrinking feature sizes, the delay is dominated by global interconnect, i.e. the relative RC delay of long wires increases while the impact of local wiring has less significance on overall delay, as a study shown in figure 6.2 proves. Generally, best results are achieved by approximating the

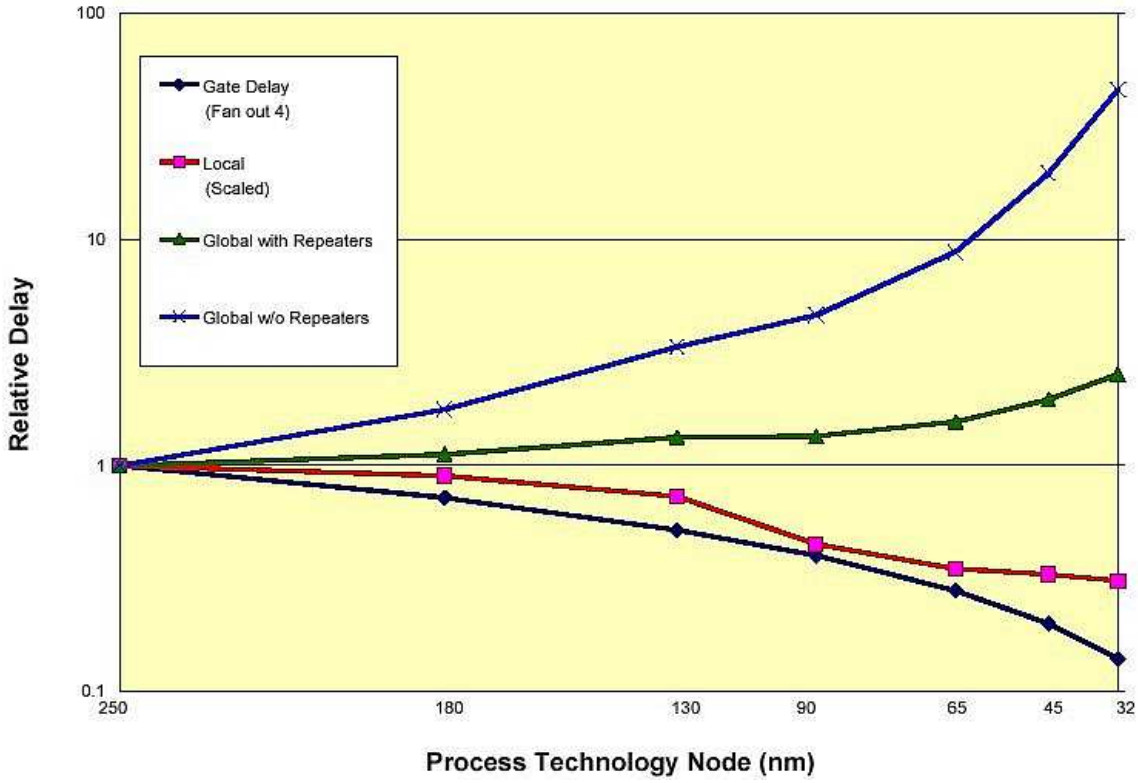


Figure 6.2: Impact of local vs. global interconnect delay with shrinking process technology

technology-independent cell area with a comparable gate from the technology library, e.g. a technology-independent two-input AND gate is assigned the cell area of an AND2 gate from the actual target library. Gates with a higher fanin than the implementations available in the library are scaled accordingly, e.g. consider an AND gate with 64 inputs. In a typical standard cell library, most gates have the same row height, thus for most gates, only the width of the gate needs to be adjusted.

6.3 Layout Aware Technology Mapping

Technology mapping, including the preceding step of technology decomposition is one of the most influential steps on overall congestion and wire length, as we break up technology-independent gates and group them into their technology implementation. In this section, we will outline our congestion aware technology decomposition and mapping algorithms.

6.3.1 Technology Decomposition

Our decomposition algorithm first collapses the gates in the design into gates with a high fanin. Then, the resulting high-fanin gates are decomposed into two-input gates. It works on a network consisting of only AND and XOR primitive cells with input and output negations. In the first step, we reduce the number of levels in the design by moving a fanout stem from the output of a gate to its inputs. Thereafter, we combine adjacent gates of similar functionality. In the final step, the circuit is transformed into a fanout-free decomposition consisting of two-input gates.

6.3.1.1 Motivation

Traditionally, the decomposition algorithm [92] uses the arrival time at the input pins to create a delay optimal decomposition. In our layout aware approach, we include wiring delay based on the estimated net length and geometric location of the input signals.

Let us first demonstrate the importance of geometric location for decomposition on a simple example. Consider the case of decomposing a gate with four input pins and

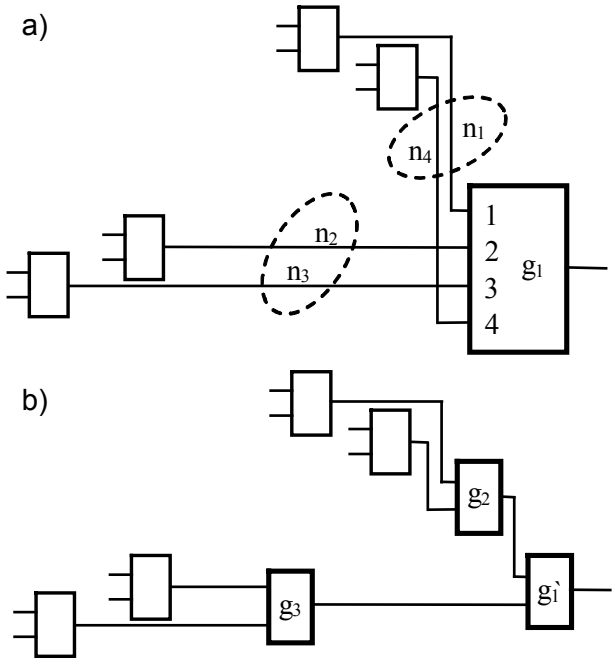


Figure 6.3: Layout aware decomposition example

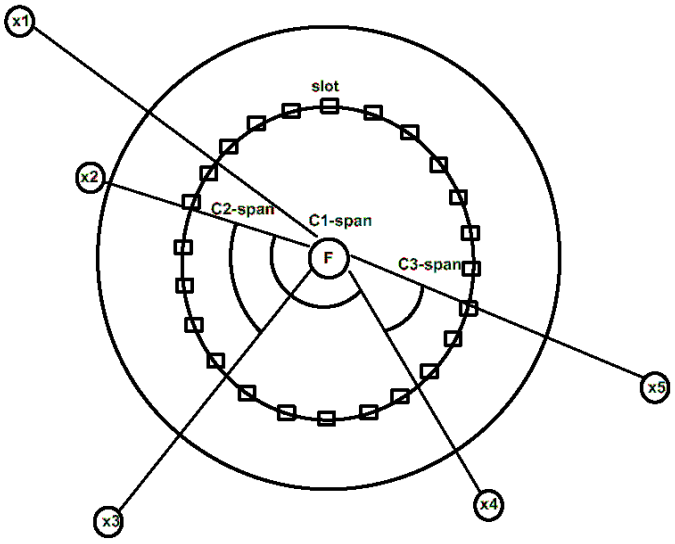


Figure 6.4: One-dimensional decomposition model

respective arrival times $t_{AT,1}$ … $t_{AT,4}$, as shown in figure 6.3a. Assume the order of arrival times to be $t_{AT,1} \leq t_{AT,2} \leq t_{AT,3} \leq t_{AT,4}$, however, all arrival times are within close range, i.e. the slowest signal is only marginally slower than the fastest signal, or even identical. A timing based decomposition algorithm would pick the first two fastest pins, i.e. pin p_1 and p_2 of gate g_1 and group them into a two-input gate. Next, pins p_3 and p_4 (now the fastest pins) will be decomposed. It is obvious that due to the geometric location of the feeding signals, this will not only cause longer net length, but also higher wiring congestion. Consequently, the better choice is to decompose pins p_1 and p_4 and their respective nets n_1 and n_4 into gate g_2, and nets n_2 and n_3 feeding pins p_2 and p_3 into gate g_3, as shown in figure 6.3b, effectively resulting in shorter wiring length and better congestion. In conclusion, we comprehend that it is important to group input pins whose signal origins are close to each other. Accurate decomposition is of particular importance as any decision made at this point is in most cases only reversible at significant cost in later design stages.

In [76], Pedram et al. propose a layout based decomposition and factoring algorithm that uses a relatively simple assignment of the input signal locations, shown in figure 6.4. All incoming signal edges are placed along a circle around the target node. Slots are assigned to an imaginary inner cycle and the best combination of nearby signals along the circle is selected. However, this approach only considers one

dimension, the entrance angle of an input edge. In fact, signals could be nearby, yet placed on exactly opposite sides of the circle. Furthermore, two signals could enter the circle at approximately the same angle, while their sources are actually located in completely different regions of the circuit. Clearly, a two-dimensional approach is needed.

6.3.1.2 Problem Formulation

Based on the previous observations, we can formulate the problem of decomposing a subject graph into a network of two-input gates as follows: Consider a gate g subject to decomposition. We define the cost of grouping two candidate pins of g as the Manhattan distance between their respective signal origins. Next, we seek to minimize the total cost of grouping, subject to a timing constraint. In order to combine layout information with timing, we only consider candidate pins within a certain timing range. Therefore, the problem can be formulated as follows:

$$\begin{aligned} & Min(\sum_{i} L_1(p_{1,i}, p_{2,i})) \\ s.t. \quad & Max(t_{AT,1,i}, t_{AT,2,i}) < T_{\max,i} \end{aligned} \tag{6.1}$$

In the above equation, $L_1(p_{1,i}, p_{2,i})$ is the Manhattan distance between the signal origins of candidate pins p_1 and p_2 in iteration i of the decomposition process. Every iteration i groups two input pins of g by creating a new gate g_i'. Only candidate pins with an arrival time t_{AT} below an upper bound T_{max} are considered. To find the optimal decomposition minimizing our cost function, we have to consider at most $n \cdot (n-1)/2$ possible decompositions during the first iteration, where n is the number of input pins of the original target gate to be decomposed. Each iteration of the decomposition process removes two candidate pins, but also introduces a new candidate pin with a different layout location (the output pin of the newly created gate g_i'), thus changing the input space of the problem. In the next iteration, we now have at most $(n-1) \cdot (n-2)/2$ possible decompositions. Therefore, finding the complete layout aware decomposition of an n-input gate is of complexity $O(n! \cdot (n-1)!/2^{n-1})$.

6.3.1.3 An Exact Solution

To reduce the complexity of the algorithm, we employ a branch and bound algorithm. We traverse the search space depth first choosing the path with minimum cost. Our

upper bound is the best solution found thus far. Since our objective function subject to minimization is monotonous, branches yielding a cost higher than the upper bound can be pruned. In addition, the search space is generally smaller since only candidate pins within a certain arrival time range are considered. However, the algorithm is still prohibitively expensive for large n and a less expensive approach is needed.

6.3.1.4 A Greedy Approach

Our greedy approach chooses the best possible solution in each iteration of the decomposition process. Thus, we select the pair of candidate pins with the minimum distance between their respective input signal origins. Obviously, the greedy approach might yield sub-optimal results because the best solution within an iteration might force sub-optimal input pin pairing during later iterations. However, particularly for large n (typically n>100), the input space is large enough to yield good results at a much lower complexity compared to the exact algorithm. The pseudo code of our greedy algorithm is shown in figure 6.5. We first order the input pins by increasing arrival time t_{AT}, considering gate delays and estimated wiring delay and calculate the Manhattan distances between all pairs of pins. The wiring delay is estimated assuming one-bend connections. Note that calculating distances for all pairs of candidate pins is of quadratic

```
Order candidate pins by arrival time;
for (i=1 to n)
  for(j=i+1 to n) {
    Calculate L1 distance(pin i, pin j);
  }
}

Greedy_Decomposition(gate g) {
  For_all_pairs_of_pins(t_AT < t_AT,min + T)
    (p1,p2) = Pins with min L1 distance;
  Decompose(g, p1, p2);
  If (pins of g == 2) return;
  Remove p2 from candidate pins;
  Update arrival time of p1;
  Update L1 distances(p1);
  Greedy_Decomposition(g);
}
```

Figure 6.5: Greedy decomposition algorithm

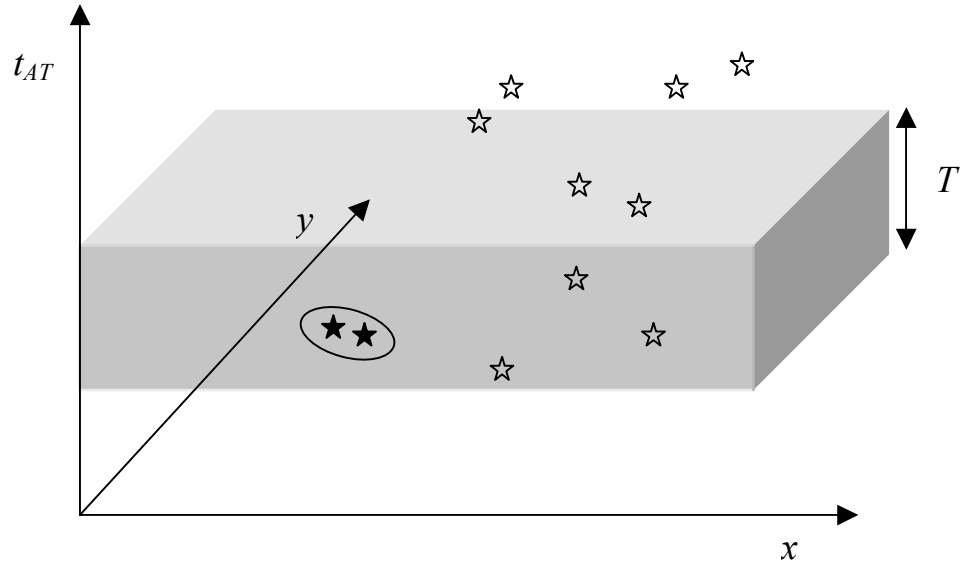


Figure 6.6: Timing window based selection of candidate pins

time complexity, however, the candidate set is a function of the maximum number of inputs of the decomposed gate, and not related to the size of the design. We then choose candidate pins using a timing window of a defined width T, hence we only consider pins with an arrival time faster than the minimum arrival time of all pins plus the width of our timing window T. Typically, we choose the value of T as the delay of one two-input gate, i.e. one level of the decomposition tree. In the next step, we select the pair of pins (p_1, p_2) with the minimum cost of grouping, i.e. the minimum distance between the locations of their signal origins. After the actual decomposition, pin p_2 is removed from the set of candidate pins and pin p_1 is substituted with the newly created pin. Arrival times of p_1 and L_1 distances to all other pins are updated. The process stops when the original target gate is completely decomposed into a tree of two-input gates.

In other words, we choose the pair of pins based on a combination of fast arrival time and closeness of its sources in the physical layout. The selection process in the three-dimensional space (x, y, t), where (x, y) are Cartesian coordinates and t is the time, is illustrated in figure 6.6. The timing window of size T is shown as shaded region while candidate pins are shown as stars. In the example, the two pins in the lower left corner would be selected even though they are not the fastest pins because within the timing window, their sources are located closest to each other in the layout.

Each newly created gate in the decomposition process has to be assigned to a placement coordinate. We calculate the new location of the gate as the center of gravity

of all input and output connections of the new gate to existing objects with a given location. Therefore, we maintain layout locations throughout the entire decomposition process for all objects in the design.

In general, by ignoring the arrival time at the input pins, the algorithm performs purely layout based decomposition, e.g. grouping pins based only on geometric location. In that case, the set of candidate pins includes all input pins of the gate to be decomposed.

6.3.2 Technology Mapping

Following decomposition, we technology map the netlist using layout information in a similar manner. Our technology mapping algorithm is based on the concept of wavefront technology mapping as described in [92].

The wavefront algorithm traverses the network in a levelized order from the primary inputs to the outputs using a wavefront of specified width w. Within the width of the wavefront, bound by its *head* and *tail*, all possible technology matches are created and implemented. This results in the creating of multi-source nets. The final technology implementation is chosen by identifying the fastest output pin that is driving the multi-source net. Due to the load-independent delay modeling in our gain-based delay model, a delay optimal technology implementation is created. We refer to [92] for a complete description of wavefront technology mapping under a gain-based delay model.

Let us illustrate on an example how layout information benefits technology mapping. Consider the example circuit in figure 6.7a. We assume a wavefront width of 3 levels, as shown by the dashed lines. Within the width of the wavefront, all technology matches will be created. In our example, we create the following matches, among others. Technology-independent gates g_1, g_2, g_3, g_5 and g_6 can be implemented as an AND-OR gate and g_4 as an AND gate. Alternatively, gates g_1, g_2, g_3 could be implemented as and AND-OR, g_4 and g_5 as a 3-way AND, and g_6 as an OR gate. Only considering timing or area constraints, the first implementation, shown in figure 6.7b might be chosen. However, in our example, gates (g_1, g_2, g_3), (g_4, g_5) and (g_6) are placed in different parts of the circuit, i.e. their locations are distant from one another. Therefore, grouping gates (g_1, g_2, g_3, g_5, g_6) will result in increased wiring and local congestion because their input sources are located in different regions of the circuit, which is exactly the reason why these gates are placed far away from each other. If instead, we group gates (g_1, g_2, g_3), (g_4, g_5) and (g_6), as shown in figure 6.7c, we not

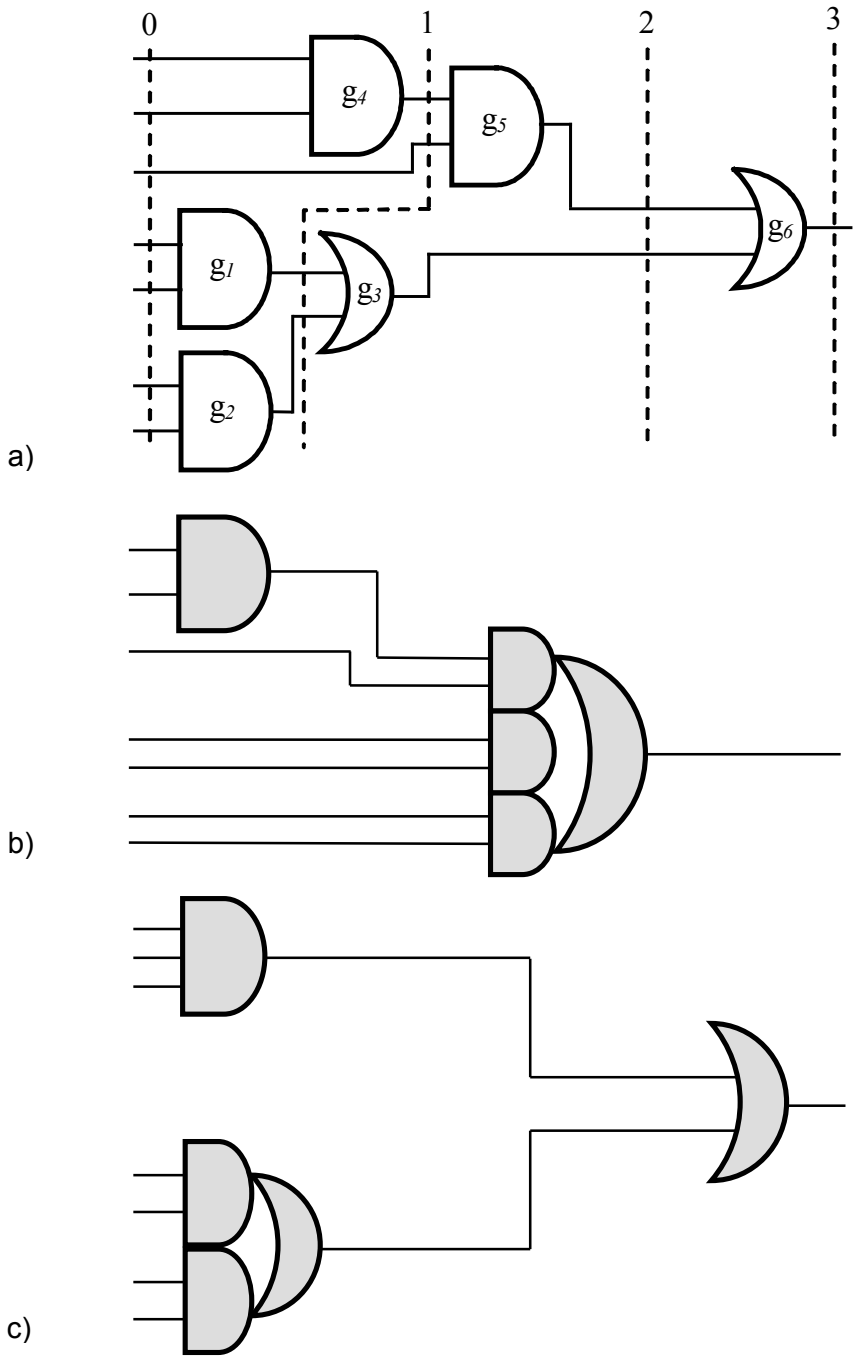


Figure 6.7: Possible technology matches based on the layout of the circuit
a) original circuit, b) generic mapping, c) layout aware mapping

only decrease total wire length because fewer wires are used along the long connections, we effectively improve overall congestion. In figure 6.7b, congestion is created because

routing is concentrated by the large AND-OR gate having many long input wires. Contrary, we have fewer long wires and a better distribution of wiring in figure 6.7c. Clearly, it is generally undesirable to group gates into a technology cell that are located in different regions of the chip. This would create a lot of extra wiring because all fanins and fanouts of the match need to be routed to a non-optimal position near the center of the bounding box of the fanin and fanout gates.

Similar to our decomposition function, we define a cost function that weighs timing (and/or area) and layout information. To effectively group close objects, we define a cost of grouping by calculating the additional wiring created by moving each technology-independent gate from its existing location to the new location, i.e. the placement location of the actual implemented technology gate. The new location of the technology gate is determined by its center of gravity with respect to all input and output connections.

We define the cost of grouping n technology independent gates into a library gate m as follows:

$$C_{grp}(m) = \sum_{i=1}^{n} \left(\sum_{j \in P(i)} \left(L_1(s_{i,j}, g_i) - L_1(s_{i,j}, m) \right) - w_i \right) \tag{6.2}$$

In the above equation, $L_1(s_{i,j}, g_i)$ is the Manhattan distance between $s_{i,j}$, the external source or sinks of pin p_j of the technology independent gate g_i and $L_1(s_{i,j}, m)$ is the distance between the same source and sinks and the mapped gate m, while w_i is the wiring saved by eliminating the internal wires. Note that calculating the additional wiring in certain cases creates a small error when using the location of objects whose final technology implementation, and hence final location has not been determined yet. This error is generally small as it is offset by the fact that we only group close objects, thus the location of the final technology implementation is close to the location of the technology-independent objects it implements. However, it can be improved by using a two-pass approach. Instead of keeping only one match, two or more possible matches are stored. During the second pass, the best combination of matches can be chosen using the known location of the technology implementation.

In conclusion, instead of choosing the technology implementation with the minimal arrival time $t_{AT,m}$ (and/or minimal area), we choose the technology dependent implementation as the minimum of the cost function $\boldsymbol{C}_{grp}(m)$, subject to a timing (or area) constraint. Similar to our cost function for decomposition, we consider estimated wiring delay and choose a timing window to consider only technology matches within a

certain range of arrival times. However, by eliminating the additional timing constraint, the algorithm becomes completely layout based and will choose matches solely based on the geometric information.

By incorporating layout information into the cost function for technology mapping, we are able to reduce total wire length resulting in less global congestion. Even more important, by grouping gates that are located close to each other in the layout, we distribute the wiring instead of concentrating it in single points, which yields less local congestion.

6.3.3 Partitioning and Clustering

To handle the increasing complexity of designs, we apply partitioning and clustering to reduce the overall problem size. This allows us to handle large designs and achieves a faster overall design turn around time. In addition, our layout aware decomposition and mapping algorithm relies on an initial placement of the technology-independent netlist. In most cases, the number of objects in the technology-independent representation will be significantly higher than the final technology implementation, thus, clustering of the netlist to reduce the number of objects is a necessity.

An important aspect of clustering and partitioning in the design flow is the decidedly different nature of logical and physical domain. The objective functions of partitioning a netlist for logic optimization and physical design are substantially different. The main goal during placement is to minimize the total net length of a weighted graph, thus clustering algorithms focus on grouping *highly connected objects*, based on local [85] as well as global connectivity [24]. A clustering approach, however, is generally not suited for logic synthesis. Instead of clustering objects, which decreases the overall problem size by reducing the number of objects, the obvious solution is to partition the netlist into sections of disjoint functionality [30]. In most cases, complete disjointness does not exist, however, minimizing the sharing of common Boolean expressions between the partitions is an excellent objective function to allow the most freedom of choice for restructuring and technology mapping algorithms within each partition.

In order to integrate synthesis and placement, we need to combine the different clustering and partitioning approaches reflecting the distinct nature of logical and physical domain. Therefore, we propose an approach with simultaneously existing partitions for synthesis and clusters for placement. Effectively, we have two different

representations for each domain that simultaneously exist in the overall flow. Hence, each object is part of two different representations of the netlist, i.e. a gate is part of a placement cluster as well as part of a synthesis partition.

For placement, we group the netlist into fine-grained clusters based on their connectivity. Our cluster size is fairly small, which is necessary to still provide enough detail to accurately determine the geometric locations of the individual objects, which provides the input to our layout driven decomposition and technology mapping algorithms. Our synthesis partitioning is based on the analysis of reconvergent signal regions which has been applied to test generation and fault simulation [36, 66], and later to circuit partitioning for resynthesis of large networks [30]. For easier understanding, we revisit the following properties of a network graph $G(V, E)$.

Definition 6.1: Given nodes r and $s \in V$, if there exists more than one disjoint path from s to r, r is a *reconvergent node* for s, and s is a *reconvergent fanout stem*.

Furthermore, a *final reconvergent node* r for s is a node that has no successive reconvergent node for s.

Definition 6.2: Let s be a reconvergent fanout stem. Then, a *reconvergent region* of s consists of all the nodes and output edges that are located on all the paths from s to any of its final reconvergent nodes r.

We use a modified version of the algorithm in [66] to find all reconvergent regions of a node s by searching for all reconvergent nodes of a fanout stem. Unfortunately, this algorithm is of complexity $O(n^2)$, however, we noticed that the size of a reconvergent region is limited, and usually independent of the number of total nodes, i.e. the size of the design. Therefore, we can often limit the search space from a fanout node to some upper bound.

To create the final partitions, reconvergent regions are grouped based on region overlap. Contrary to the greedy approach used in [30], we group reconvergent regions in the order of decreasing overlap and favor regions that overlap within at least one of the final reconvergent nodes of either region and at least one other input node, also known as converging regions. Consequently, we group regions with the maximal number of common Boolean expressions to improve synthesis quality. The overlap of two regions is easily computed during the backward phase of our modified algorithm of [66] that identifies the reconvergent regions

In many cases, almost all regions are overlapping, thus grouping all regions would result in covering almost the entire circuit in only one or two partitions. Therefore, we

use an upper bound to control the maximum size of a partition. Hence some regions can initially not be included in any of the partitions because they overlap more than one existing partition and merging the partitions would result in too large of a partition. Their disjoint part is added to the partition with the most overlap. After grouping all regions into partitions, all remaining nodes which are not part of a reconvergent region are absorbed into their neighboring partitions based on their connectivity to input lines of these partitions.

After partitioning the design, we perform logic minimization and technology mapping on each of the individual partitions. For optimal speedup, the partitions would have to be synthesized in parallel. However, to take full advantage of parallelism, dynamic timing assertion and slack apportionment at the partition boundaries is necessary. This is generally a non-trivial task that could be addressed in future work.

6.3.4 Synthesis and Placement Integration

In the following part, we explain how the concepts outlined in the previous sections are combined to form an integrated synthesis and placement flow. The overall flow is depicted in figure 6.8. We start with the technology independent, non-optimized netlist of the design. Initially, we partition the design using our synthesis partitioning algorithm based on reconvergent signal regions, as explained in the previous section. In the next step, we run logic minimization algorithms, e.g. kernel factoring, common subexpression extraction, on the synthesis partitions. Subsequently, we cluster the netlist using our connectivity based clustering algorithm into relatively small clusters. In our model, both cluster and partition based representations coexist to serve the individual needs of logic synthesis and placement. In other words, pure synthesis operations only utilize the synthesis partitions while placement only uses the cluster representation. The layout-driven part of logic synthesis, i.e. our previously proposed decomposition and technology mapping algorithms use both representations.

After clustering, we create an initial placement using a quadratic placement algorithm with quadrisection to place the clustered technology-independent netlist. Each member of a cluster is assigned the layout location of the parent cluster. Succeeding, we perform congestion-aware layout driven decomposition and technology mapping, as described in the previous sections, on the synthesis partitions. Note that we retain placement information by assigning a layout location to each newly created gate. However, retaining cluster information at this point is an infeasible task. Instead, we

create updated clusters of bigger size, now using the actual layout locations of the objects in the netlist. We now have a technology-mapped pre-placed design and continue global placement using our quadratic placement algorithm. Finally, we uncluster the netlist and perform detailed placement and legalization.

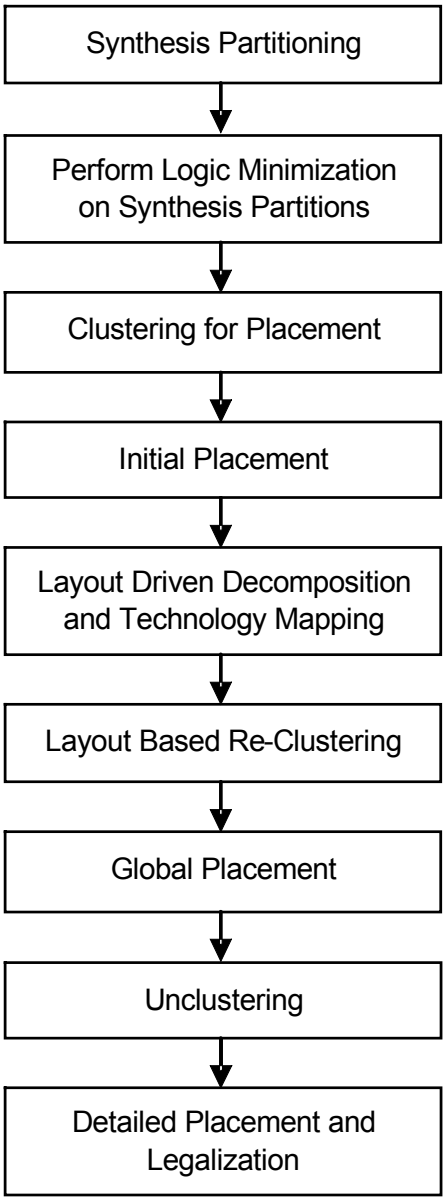


Figure 6.8: Integrated synthesis and placement flow

6.3.5 Experiments and Results

The proposed integrated synthesis and placement flow has been implemented in the logic synthesis tool BooleDozer™ [91]. We employ the proposed layout aware technology decomposition and mapping algorithms in conjunction with partitioning for synthesis and clustering for placement. During decomposition, we use a combination of the exact branch and bound based algorithm for small number of inputs and our greedy algorithm for larger ones. Table 6.1 presents results of the layout driven optimization process for a number of IBM ASIC designs. *Lit flat* and *Lit part* show the amount of literals after logic minimization on the flat and synthesis partitioned designs, respectively. On average, optimization of the partitioned design results in less than 2% more literals compared to the flat design, achieved due to the high degree of functional disjointness of the individual partitions. *Delay layout* represents the decrease (or increase) of the delay of the most critical path during layout driven optimization in comparison to the standard timing based decomposition and technology mapping. *Avg Con* and *Max Con* show the average and maximum horizontal and vertical congestion, for layout driven and standard timing based optimization. Due to the decreased amount of wire length and by avoiding local areas of high congestion, achieved by grouping only close objects, we can generally improve congestion compared to the timing based only optimization. In addition, by balancing timing and layout information, we can improve congestion while achieving timing closure. Figure 6.9 shows the congestion maps for circuit *Design1* after synthesis and placement in the standard flow in comparison with our layout driven approach. Congestion across the horizontal and vertical cuts is shown in different colors ranging from blue (light) for the lowest to red (dark) for the highest congestion. Of particular importance is the maximum congestion because it determines the routability of a design, which is consistently improved by avoiding local areas of high congestion.

Furthermore, we show results of the overall speedup achieved by our partitioning and clustering approach. *CPU layout* denotes the relative runtime of the integrated synthesis and placement flow in comparison to a completely flat approach. In general, further run time improvement is possible but will usually result in a degradation of the quality of results.

Circuit	*Gates*	*Lit flat*	*Lit part*	*Delay layout*	*Avg Con std*	*Avg Con layout*	*Max Con std*	*Max Con layout*	*CPU layout*
Design1	138332	375302	382698	-0.01 ns	24.3%	22.2%	122.1%	103.2%	-82%
Design2	92582	244417	249060	-0.04 ns	37.1%	32.2%	82.7%	78.1%	-54%
Design3	43145	127749	132521	0.00 ns	51.2%	48.2%	153.1%	127.3%	-27%
Design4	101452	334822	342081	0.00 ns	57.4%	52.2%	127.0%	121.8%	-69%
Design5	97165	331383	336697	-0.02 ns	48.2%	43.7%	121.1%	112.3%	-71%
Design6	84292	320308	326646	0.00 ns	39.2%	37.9%	101.7%	98.1%	-68%
Design7	77215	221437	224651	0.00 ns	47.1%	44.4%	158.5%	132.3%	-65%

Table 6.1: Results of layout driven technology mapping

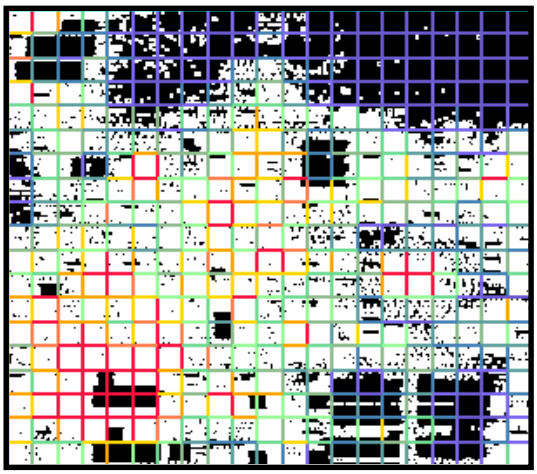
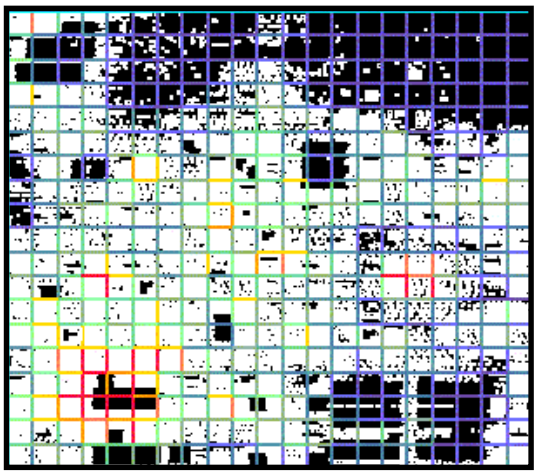
a)  b) 

Figure 6.9: Congestion maps of *Design1* for a) standard flow and b) layout driven optimization

6.4 Concurrent Factoring and Extraction

In this section, we extend the layout driven concept to logic minimization and present a novel algorithm that applies physical layout information during common subexpression extraction to improve wiring congestion and delay, resulting in improved design closure.

In our novel approach, we propose a layout driven algorithm for the concurrent extraction of common subexpressions based on the well known and highly effective *fast extract* algorithm by Rajski et al. [80, 81]. This is one of the most important steps affecting the overall circuit structure, and consequently congestion and wire length during logic synthesis. In addition, we consider dependency relations between candidate cube divisors to efficiently select the divisor with the best layout cost.

6.4.1 Introduction

Decomposition of a Boolean expression refers to extracting subexpressions common to one or more functions such that they can be shared across the network, effectively reducing the size of the synthesized design. It plays an important role in the design process because it has a major impact on the overall structure of the circuit. During the decomposition process, Boolean subexpressions common to one or more functions are identified and extracted such that they can be shared across the network. The most common decomposition algorithms of practical importance are the method by Brayton [14], which limits potential divisors to cube-free multiple-cube divisors (*kernels*), generally known as *kernel extraction*, and the concurrent decomposition method known as the *fast extract* algorithm by Rajski [80, 81].

Traditional common extraction algorithms only optimize cell area by minimizing the literal count. Due to the sharing of expressions, this process tends to create gates with high fanout and fanin, which in turn results in increased wiring congestion. Existing layout driven logic synthesis algorithms [76, 77] only exploit the layout information for improved timing information utilizing estimated wire delay and to minimize area employing cell and estimated wiring area. However, the reduction of overall wiring congestion for better routability, a very important objective function besides timing closure, in conjunction with timing and area optimization, has not been actively addressed.

6.4.2 Fast Extract Algorithm

Our layout driven decomposition algorithm is based on the method by Rajski et al. In the following part, we give a brief overview of the underlying procedure.

The *fast extract* approach limits itself to identifying and extracting *double-cube divisors*, that is, cube-free multiple-cube divisors having exactly two cubes, and *single-cube divisors* having exactly two literals, considered concurrently with their complements. By limiting objects to size 2, the complexity of candidate subexpression generation (the cube divisors) is polynomial, yet algebraic divisors of arbitrary size can be extracted by way of multiple subsequent extractions.

A simplified version of the *fast extract* algorithm is shown in figure 6.10. Double-cube divisors are generated only once, and updated during the extraction process. Their extraction is done separately for each node in the network. However, double-cube divisors can be shared between different nodes, that is, when the divisor has been identified for each node individually. In contrast, single-cube divisors are extracted from cubes belonging to different nodes in the network. In each iteration of the algorithm, the double- or single-cube divisor with the highest weight, that is, the largest literal savings is chosen. Please note that the algorithm is entirely greedy, the cube

```
begin
    Generate double-cube divisors with
     associated weights;
  while (weight >= 0) {
    Choose double-cube divisor dcd with
      maximum weight w_dc;
    Choose single-cube divisor scd with
      maximum weight w_sc;
    Select the divisor dcd or scd with
      maximum weight;
    Substitute extracted expression;
    Update weights of conflicting
       double-cube divisors;
   }
end
```

Figure 6.10: *Fast extract* algorithm

divisor with the maximum literal saving is chosen and fully extracted, even though it might affect the weight of other double-cube divisors. In fact, some kernels may entirely cease to exist as divisors of the remaining network.

6.4.3 Layout Driven Extraction

The extraction of common subexpressions has a significant impact on the structure of the netlist because it decomposes large two-level structures in sum-of-products form (SOP) into smaller (in terms of literals) multi-level representations. We first transform the netlist into a structure consisting of only NAND gates with a high fan-in. Hence, each node in the network can be represented as a sum-of-products form, which provides the input for the subsequent cube-divisor generation.

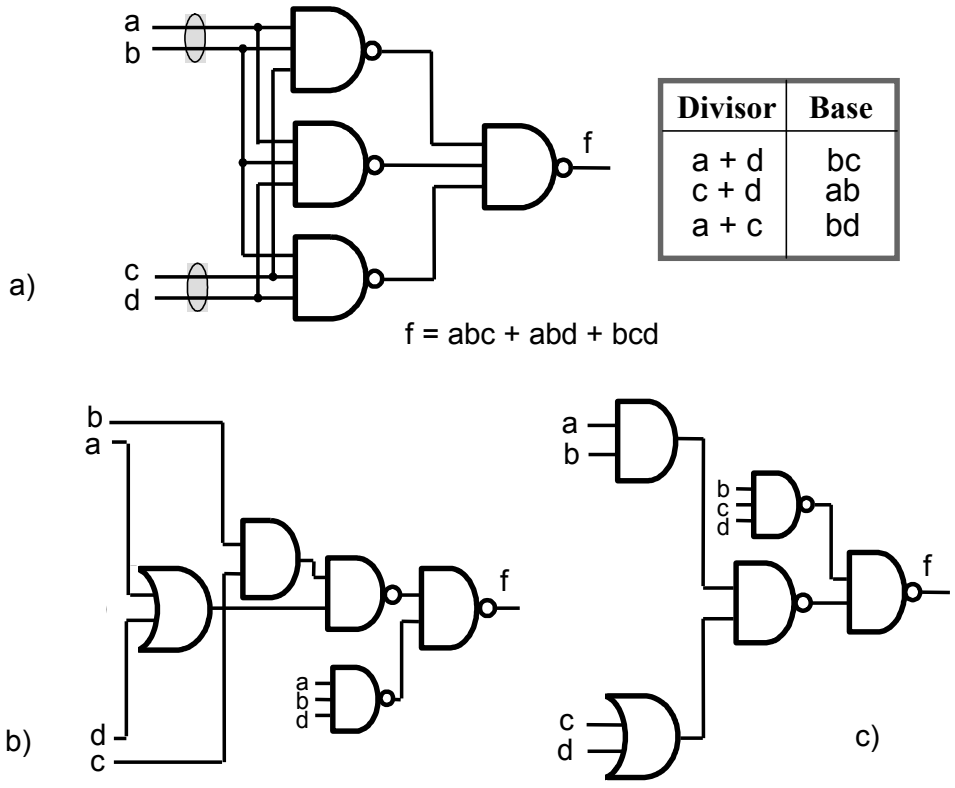


Figure 6.11: a) Original circuit, b) generic decomposition, c) layout aware decomposition

6.4.3.1 Motivation

Traditionally, the algorithm minimizes cell area by selecting the cube divisor with the maximum literal saving. In our layout aware approach, we present a decomposition procedure with the objective of optimizing wire length and wiring congestion. Let us first demonstrate the importance of geometric location during the extraction process on a simple example. Given the network implementing the Boolean function $f = abc + abd + bcd$, as depicted in figure 6.11a. The following double-cube divisors can be extracted: $a+d$, $a+c$, and $c+d$. Each of the divisors provides the same literal savings when extracted, however, the selection of the right kernel has a major impact on the circuit structure. As shown in the figure, assume that the sources of signals (a,b) and (c,d) are located close to each other, respectively. Extracting the kernel $a+d$ (or $a+c$), as shown in figure 6.11b, creates a local area of high congestion and increases wire lengths because the grouped signals (a,d) and (b,c) originate from opposite parts of the layout area. In contrast, if we choose the kernel $c+d$ as shown in figure 6.11c, wire length is decreased and congestion is distributed more evenly because the source signals of the extracted kernel $c+d$ and respective base (or co-kernel) ab are located close to each other, allowing the extracted gate to move towards its sources.

Furthermore, if a common expression can be extracted from different nodes in the network, it might be beneficial not to share the expression under certain conditions, that is, if the nodes are particularly far apart and/or the path under consideration is timing critical. Consider the example shown in figure 6.12. Assume that the common expression ab can be shared by nodes n_1, n_2 and n_3 as depicted in figure 6.12a. Depending on the location of the nodes, the delay on the path to node n_3 can be

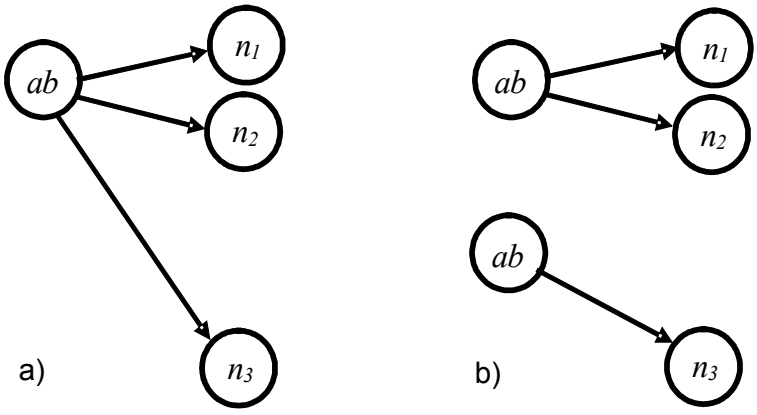


Figure 6.12: Sharing of an expression between different nodes,
a) generic, b) with layout consideration

improved by not sharing expression *ab* with nodes n_1 and n_2. Despite the fact that this procedure slightly increases *global* wire length, it improves the delay on the path along n_3 by reducing the maximum length of a wire and the fanout of the node subject to extraction. This can be achieved by considering the delay at each of the nodes that share a common expression, taking into account the wiring delay from the shared expression to the subject node.

Accurate decomposition is of particular importance as any decision made at this point is in most cases only reversible at significant cost in later design stages.

6.4.3.2 Problem Formulation:

Based on the previous observations, we can formulate the problem of extracting common subexpressions from a subject network under consideration of layout information as follows. We define the layout cost of extracting a kernel as the sum of the Manhattan distances between the signal origins of the literals in the divisor and the remaining base, also referred to as co-kernel, respectively. Considering the example in figure 6.10, the layout cost of extracting the double-cube divisor *c*+*d* is the sum of the L_1 distances between the sources of signals *c* and *d*, and the co-kernel, *a* and *b*, respectively. Consequently, we seek to minimize the total cost of grouping while concurrently minimizing literal count. As shown in [80], the weight of a double-cube divisor W_{dc}, that is, the literal saving if extracted, is defined as follows:

$$W_{dc} = d(r-1) - r + \sum_{i=1}^{r} |b_i| + C \tag{6.3}$$

In the above equation, *d* denotes the size of a double cube divisor, *r* is the number of times the divisor is being shared, i.e. the number of different bases, while b_i is the number of literals in base *i,* and *C* is the number of literals that can be saved by using the complement of the double-cube divisor. Similarly, the weight of a two literal single cube divisor W_{sc} is calculated as:

$$W_{sc} = C(i,j) - 2 \tag{6.4}$$

Here, *C*(*i*,*j*) denotes the coincidence of a single cube divisor with literals *i* and *j*, thus, the number of cubes which contain both *i* and *j*, minus 2 to implement the extracted divisor itself.

To choose the best possible divisor considering both layout information and literal saving, we combine the two cost functions. Consequently, we select the divisor with the minimum layout cost within a certain weight, i.e. literal saving range:

$$\begin{aligned} & Min(\sum_{i}(L_1(d_i)+L_1(b_i))) \\ s.t.\quad & Min(W_{dc,i},W_{sc,i}) > (W_{\max,i}-w) \end{aligned} \tag{6.5}$$

$L_1(d_i)$ *and* $L_1(b_i)$ denote the Manhattan distances between the signal origins of the divisor d_i and the base b_i in iteration i of the decomposition process. Only single- and double cube divisors with a weight larger than a certain weight are considered, that is, the maximum weight of any divisor minus a given window of size w.

6.4.3.3 Cube Divisor Selection

Due to the fact that extracting one divisor can affect the weight and layout cost of other divisors, an optimal solution of the formulated problem becomes infeasible. Therefore, similar to the original decomposition algorithm, a greedy approach is chosen. Instead of minimizing the total layout cost, in every iteration, the divisor with the smallest cost is selected from a set of candidate divisors having weights larger than a given lower bound. The number of candidate divisors is further reduced by the fact that only *dependent* candidate cube divisors affect each other's extraction.

Definition 6.3: Two different candidate cube divisors are considered to be *dependent* on each other if they have at least one cube of some function f in common.

In the example of figure 6.10, all three divisors $a+d$, $c+d$ and $a+c$ are dependent on each other because they each share a common cube. As a result, extracting a candidate divisor in one iteration of the algorithm prevents the extraction of its dependent divisors subsequently. It follows that candidate divisors that share no common cube are *independent* of each other. All dependent divisors and their relations can be represented as a *dependency graph* $G(D, R)$ with cube divisors $d \in D$ as vertices and their respective dependency relations $r \in R$ as edges.

Figure 6.13 shows the dependency graph for the three cube divisors of our previous example, denoted by i). Because all divisors are dependent on each other, the graph is a clique.

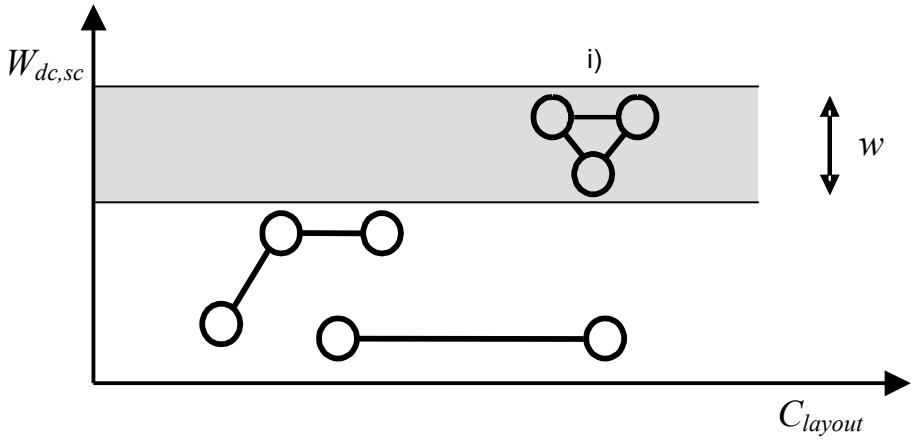


Figure 6.13: Candidate cube divisor selection

The extraction of a divisor does not prevent the subsequent extraction of its independent divisors. This observation is useful for the selection of candidate divisors with minimum layout cost because only divisors that belong to the same dependency graph need to be considered. In other words, a better choice for a divisor with an undesired high layout cost can only be found within the set of divisors of its dependency graph. Figure 6.13 illustrates the selection process. C_{layout} denotes the layout cost of extracting a cube divisor, and W_{sc} and W_{dc} are the weights of single- and double-cube divisors, respectively. Dependency relations between the candidate divisors are shown as solid lines. In every iteration, the candidate with the maximum weight (literal savings) is chosen. Next, all cube divisors within its dependency graph that fall within the width of window w, shown as shaded region, are concurrently considered for extraction, and the divisor with minimum layout cost is selected. After each extraction, affected cube divisors are updated. Generally, we choose a window size of one literal, which is our lower bound for w. However, at least one other divisor in the dependency graph, other than the subject divisor, should be considered. Therefore, w can be increased up to a fixed upper bound, such that at least one alternative candidate divisor from the dependency graph will be considered.

In addition, when a cube divisor is shared between different functions, we only share it between nodes whose timing is estimated to be non-critical, as previously illustrated by the example in figure 6.12.

Each newly created gate in the decomposition process has to be assigned a placement location. An exact solution can be determined by solving a quadratic placement problem formulated within the boundary box of the affected objects.

However, for efficiency, placing only the extracted expression at the center of gravity of its input and output connections can be chosen.

In conclusion, we extract common expressions based on their layout cost and associated weight, the estimated literal saving. Consequently, overall wiring length and local congestion is reduced.

6.4.4 Layout Driven Synthesis Flow

Subsequently, we illustrate how the concepts outlined in the previous sections are combined in a layout driven design flow. The overall flow is shown in figure 6.14. We start with a technology independent, non-optimized net list of the design and run initial logic optimization transformations. Thereafter, we create an initial placement using a

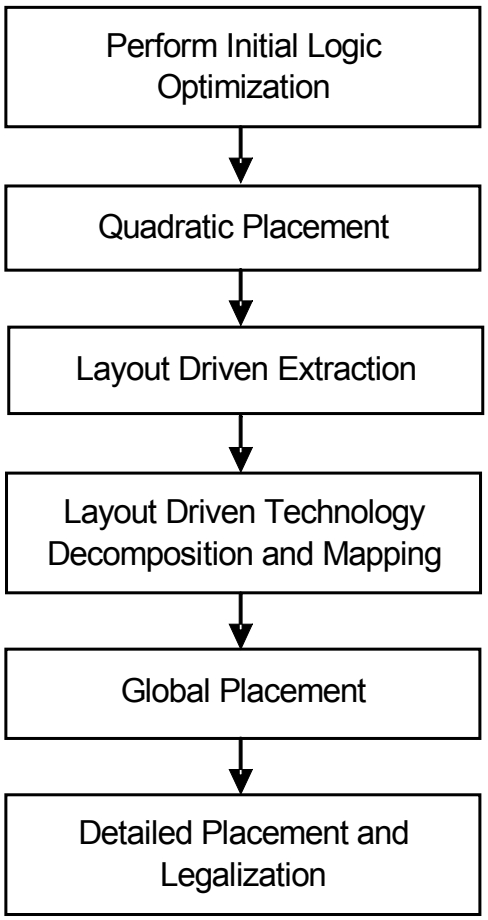


Figure 6.14: Complete layout driven synthesis flow

quadratic placer. After each cell has been assigned a placement location, layout aware extraction of common subexpressions with congestion consideration as described before is performed. Thereafter, we apply our layout aware technology decomposition and mapping algorithms as described earlier. However, for the purpose of the experimental results described in the following section, only the layout driven decomposition described in this paper was used. Placement information is retained by means of assigning a layout location to each newly created cell. We continue global placement on the technology-mapped net list using our quadratic placement algorithm. Finally, we perform detailed placement and legalization.

6.4.5 Experiments and Results

We employ the illustrated layout aware decomposition and extract cube divisors based on literal and layout cost. In addition, we share common expressions between different nodes dependent on their delay. Table 6.2 presents results of the layout driven optimization process for a number of industrial ASIC designs. *Lit fx* and *Lit lds* show the amount of literals after performing logic minimization using the standard implementation of the *fast extract* and our layout aware extraction algorithm, respectively. *Area fx* and *Area lds* represent the active cell area after all optimization and final placement. On average, optimization using the layout aware approach produces about 7% more literals compared to the generic extraction due to the fact that cubes are extracted in a different order and sharing between different nodes is reduced.

Circuit	*Lit fx*	*Lit lds*	*Area fx*	*Area lds*	*Delay lds*	*Avg Con fx*	*Avg Con lds*	*Max Con fx*	*Max Con lds*
Design1	142521	154922	373214	375152	-0.15 ns	42.2%	38.5%	94.7%	82.0%
Design2	115432	122517	317661	324424	-0.21 ns	48.5%	44.2%	144.1%	121.4%
Design3	232484	247190	689458	709799	-0.26 ns	62.4%	59.2%	108.3%	91.4%
Design4	303207	321802	792388	799012	-0.04 ns	52.2%	50.7%	151.0%	102.8%
Design5	196108	206421	523652	525010	-0.27 ns	72.2%	68.5%	114.7%	91.7%
Design6	112358	119708	342481	339032	-0.08 ns	51.9%	48.3%	88.5%	73.9%

Table 6.2: Results of layout driven common cube extraction

However, the cell area after technology mapping and placement is actually smaller. This is due to the fact that due to the decreased output load, gates of lower power levels are chosen and less buffers need to be inserted to meet the timing requirement. *Delay lds* represents the reduction of the delay of the longest path during layout driven optimization due to the fact that wire length has been reduced. It represents the actual delay after physical design, using *RC* extraction to calculate wire delays. In addition, *Avg Cong* and *Max Cong* show the average and maximum horizontal and vertical wiring congestion of the layout driven and generic extraction runs. Due to the reduced amount of wire length and by preventing areas of high local congestion, which is achieved by extracting signals of close proximity, we can significantly improve congestion compared to the standard flow. Of particular importance is the maximum congestion because it determines the routability of a chip, which is consistently reduced. By creating a layout aware structure of the network, wiring is decreased and better distributed, and we are able to significantly reduce delay and congestion at identical or better area.

6.5 Conclusion

We have presented layout driven logic synthesis algorithms for technology decomposition and mapping, and proposed a layout driven concurrent extraction algorithm that successfully integrates the use of layout information during logic restructuring. By extracting common expressions depending on the proximity of the input and output signals, and choosing technology matches based on the location of the technology-independent objects, we are able to change the structure of the netlist such that it reflects the layout properties. This results in improved timing and reduced routing congestion. In addition, we outlined the concept of co-existing synthesis partitions and placement cluster to reflect the different requirements of optimization in the logical and physical domains.

We conclude that there often is a choice of selecting technology matches or extracting cube divisors at identical or similar implementation cost, which gives us the freedom to make a selection considering additional optimization functions, e.g. wire cost, without any penalty in area or delay.

In the next chapter, we illustrate how the introduced layout and structure aware synthesis algorithms are applied in the design of integrated circuits.

7 Applications

This chapter shows how the proposed layout and structure aware synthesis algorithms are integrated into a typical ASIC design flow. We use a small design example, the core of an industrial microprocessor to demonstrate the different synthesis approaches for data and control logic, and give an overview of the placement of regular logic.

7.1 Overview

The fundamental concept in the design of high-performance integrated circuits that contain a large amount of datapaths is to separate data from control logic and treat them differently in the design process. As outlined in chapter 3, most standard logic optimization algorithms are tailored to random logic and perform poorly on regular structures like adders. The very same problem appears in physical design. Placement algorithms perform well on optimizing a single objective function, e.g. total wire length. Unfortunately, this does not automatically guarantee minimal delay or congestion. Various attempts of timing and congestion driven placement algorithms exist, however, they can still provide only a sub-optimal trade-off of the many different optimization criteria in physical layout as there is no feasible closed-form description of the various objective functions and their complex interactions. This is where sometimes a simple idea outperforms the most complicated models. Instead of viewing a circuit as a large collection of random objects, we actually consider its global structure.

7.2 Layout and Structure Aware Design Flow

Figure 7.1 gives an overview of the suggested design flow, integrating layout and structure aware algorithms. At first, we extract the structural regularity from the gate-level description of the design. Assuming the circuit contains a significant amount of regular structures, we optimize and preserve them using our regularity driven synthesis method.

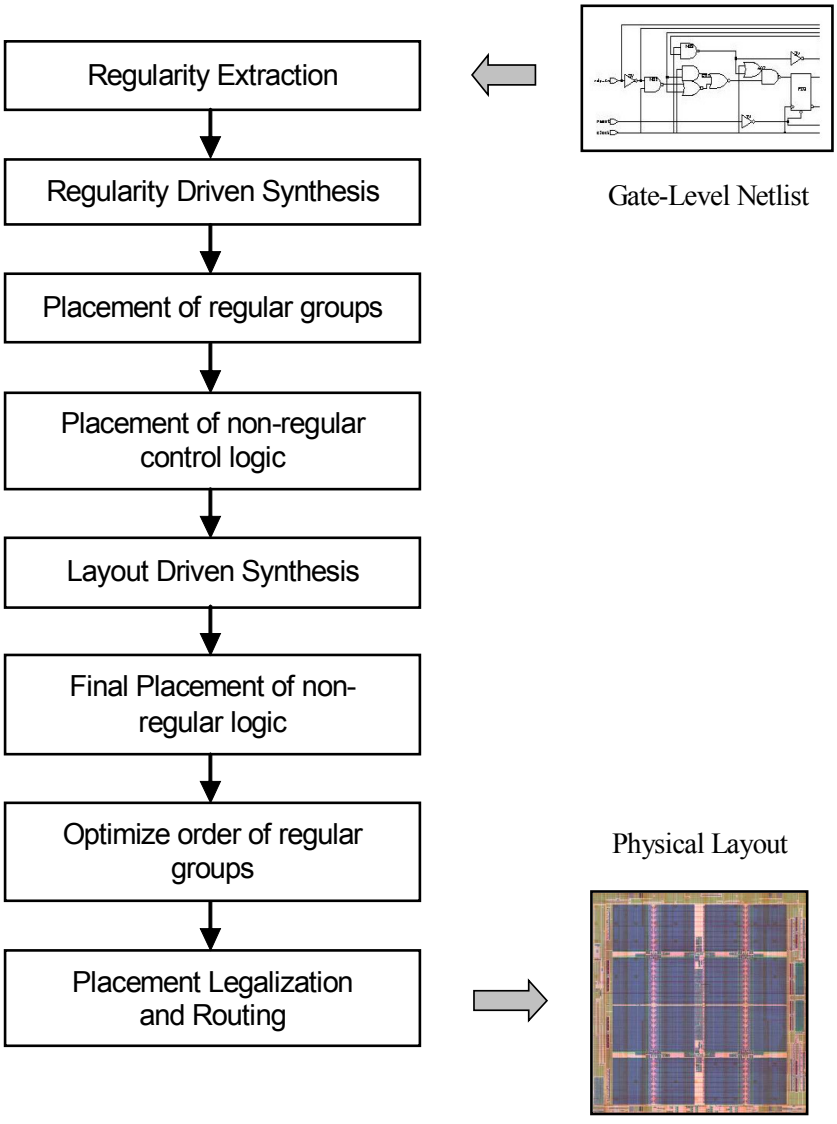


Figure 7.1: Layout and Structure Aware Design Flow

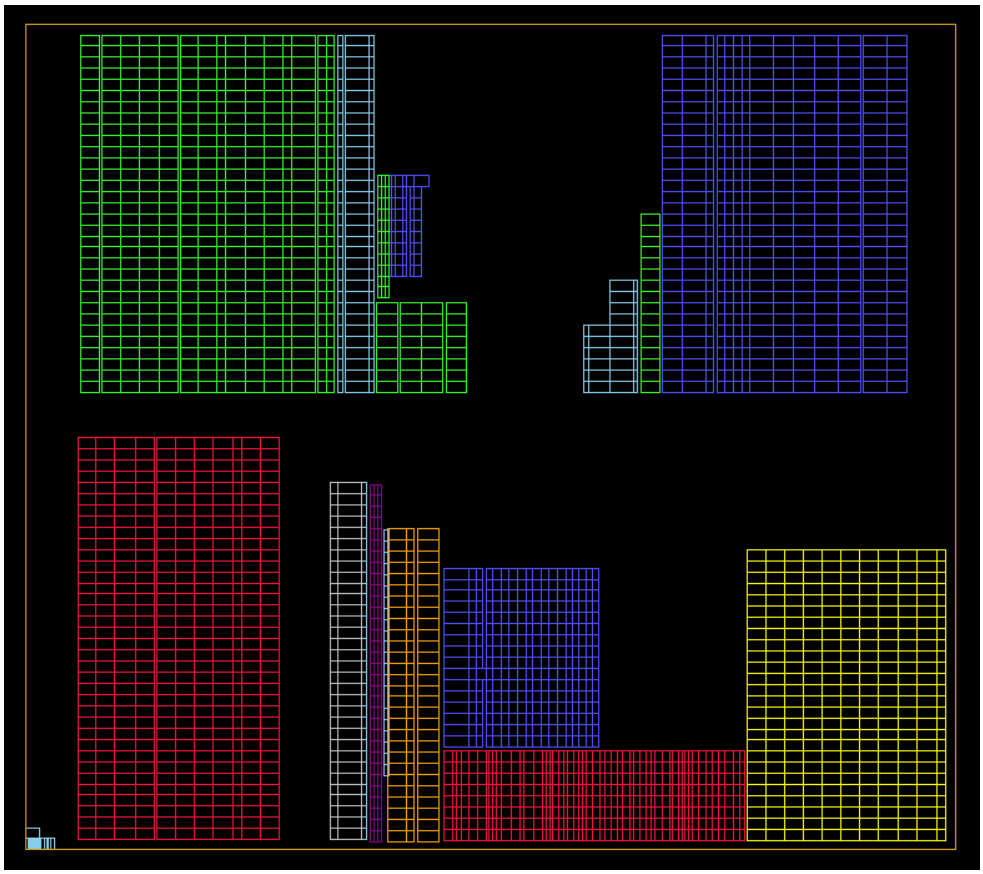


Figure 7.2: Placement of regular groups

In the next step, we create an initial placement of the design. We start by placing the regular structures as two-dimensional blocks on the chip image [43, 73], as illustrated by figure 7.2. The design contains a number of large adders and other regular structures that could all be found using our regularity extraction algorithm.

At this point, the order of the stages in a regular group is determined only by the topological order of the objects within each bit-slice. The order of the slices is random, in most cases, either an increasing or decreasing order of the bits will be chosen. Due to the large size of regular blocks, their automated placement is generally difficult. Most placement algorithms are unable to handle objects of significantly different size. Just like large macros, a regular group can become stuck in a suboptimal location if its size is close to the size of cut or partitioning area. In addition, the location of an object is usually described only as a point (at the center or origin of the object), which represents a rather inaccurate estimation of the location of a large object. However, good results in

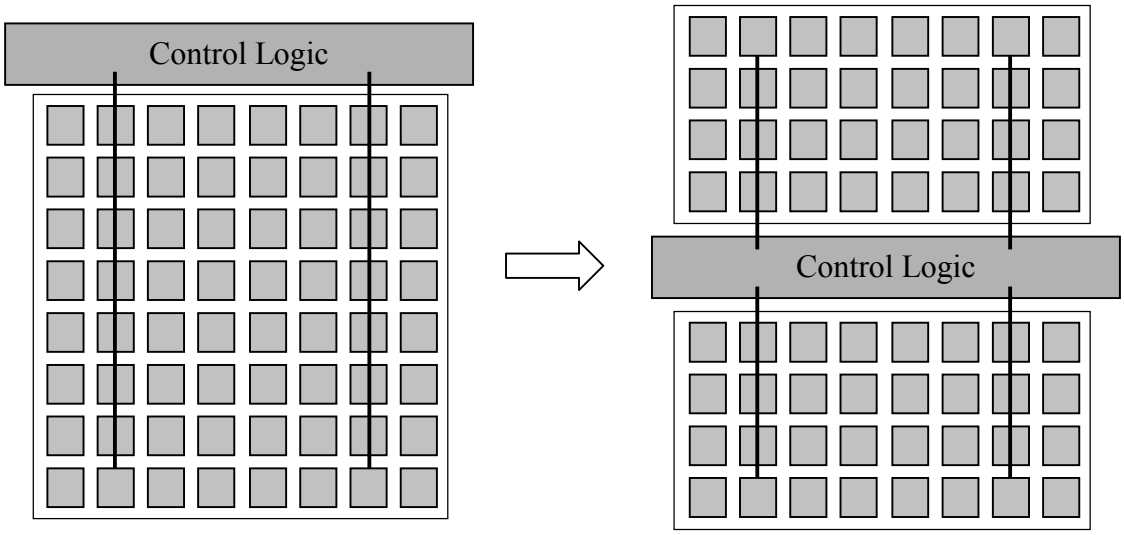


Figure 7.3: Placement of the control logic of a regular group

the placement of mixed standard and macro cell placement have been reported using a force-directed placement algorithm [32]. Another idea addressing the placement of regular groups is discussed in the next chapter.

After the placement of the regular groups, we place the surrounding random control logic using a standard placement algorithm. As previously described, we assign a physical representation to each technology-independent cell using a pseudo-library. A number of special design techniques for the integration of regular groups and the

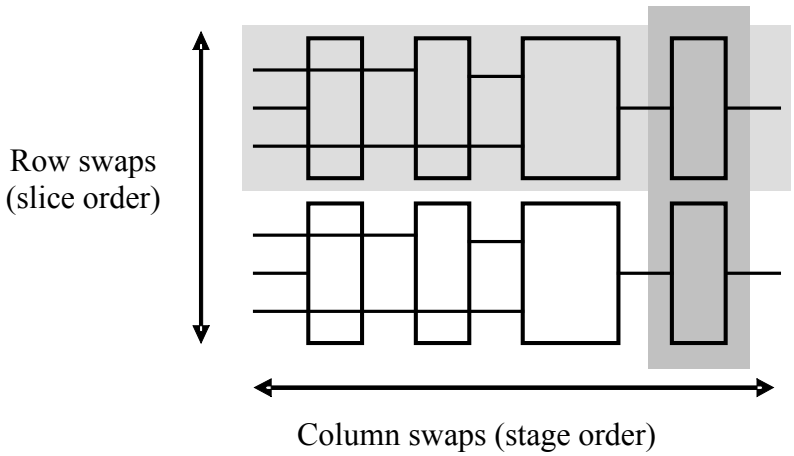


Figure 7.4: Row and column optimization

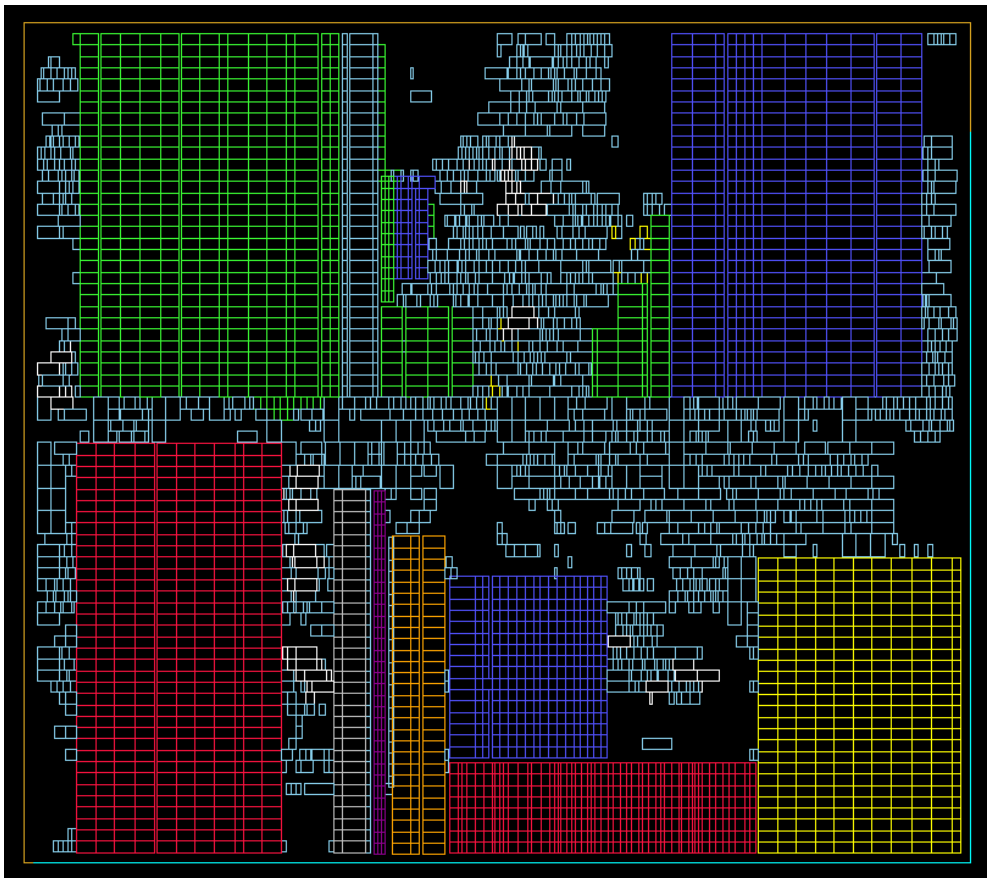


Figure 7.5: Final Optimized Layout

adjacent control logic can be applied. For example, it is possible to extract the control logic that belongs to a datapath directly during regularity extraction, as the control nets are already identified as seed nets to the extraction algorithm. Thereafter, the control logic can be placed right on top of the regular group. Another approach horizontally splits a large regular group and places the control logic centrally between the two resulting groups, as shown in figure 7.3. The advantage of this method is the creation of shorter control nets.

After creation of the technology-independent placement, we can apply layout aware synthesis algorithms, i.e. common cube divisor extraction and technology mapping. After updating the placement of the random logic reflecting their technology implementation, we optimize the placement of the regular groups. Initially, the location and order of the bit-slices and stages of each regular group was fixed and independent of the placement. We employ row and column swaps within a regular group,

comparable to operations on a matrix, to optimize the order of the slices and stages depending on the surrounding logic as illustrated in figure 7.4. For example, a sliding window based algorithm can be utilized to determine the order of the stages in a group. Within a given window size, the order of stages with the minimum wire length is determined. The window slides across the regular group and minimizes the wire length at each position of the window until all stages have been visited.

The final layout, shown in figure 7.5, yields a delay reduction of nearly 20% and is significantly more compact compared to the standard design flow. After completion of the placement of regular and random logic, the circuit is finally routed. Routing of regular groups is a relatively trivial task as each slice has similar connectivity, which also helps minimize congestion.

7.3 Conclusion

The extraction and preservation of regular logic results in the creation of highly dense layouts with improved delay and routability of the data logic, while the application of layout aware synthesis algorithms improves the delay and congestion of the control logic. In addition, while the placement of regular groups requires either a dedicated placement algorithm or manual placement of at least the larger groups, the overall complexity of the placement problem is significantly reduced. As a result, large designs can be placed virtually flat. Furthermore, the direct connectivity within each bit-slice simplifies the routing task.

In conclusion, we have presented an efficient design method that combines logical, structural and physical design information. It is particularly effective in the design of modern high-performance designs because they often contain of a mix of datapath and control logic.

8 Conclusion and Future Work

In this dissertation, we have presented novel algorithms and provided an overall framework for the interaction of the classically separate steps of logic synthesis and physical layout in the design of VLSI circuits. Due to the domination of interconnect delays, the traditional separation of logical and physical design results in increasingly inaccurate cost functions and aggravate the design closure problem. Consequently, the interaction of physical and logical domains has become one of the greatest challenges in the design of VLSI circuits. To address this challenge, we have presented layout and structure aware synthesis algorithms and showed how to integrate them into a design flow for high-performance integrated circuits containing a mix of data and control logic. The major contributions of this dissertation are as follows:

First, we presented a novel algorithm for the extraction of functional regularity from arbitrary netlists. By reusing functionally equivalent bit-slices of a regular group, we showed a significant speed-up of the optimization process. Later, we extended our algorithm to identify structural regularity and showed its application to create compact regular layouts during physical design. Regular structures can be used to obtain high-density layouts with short wire length and improved timing while simplifying placement and routing tasks.

After illustrating the importance of regular structures in a design, we addressed the problem of regularity preservation. Our regularity-driven synthesis flow identifies and preserves regular structures throughout the synthesis process and speeds-up technology-dependent optimization by reapplication of successful trial-and-error transformations.

Furthermore, we introduced novel congestion aware layout driven logic synthesis algorithms employing physical design information during early logic synthesis to optimize the structure of the netlist, resulting in improved routing congestion and timing. After creating an initial placement of the technology-independent netlist, we use

the physical location of the individual cells for layout aware technology decomposition, mapping and factorization algorithms, effectively combining the logical and physical design stages.

Finally, we combined structural and layout aware optimization and presented an efficient design flow that combines logical, structural and physical design information. As a result, we consistently improved timing and routing congestion on a number of industrial circuits.

8.1 Future Work

Our future work will address a number of important problems. First of all, the extraction of special data logic structures that are not regular by common means, for example multipliers represented by a Wallace tree, will be addressed. While such structures cannot be placed in the typical two-dimensional way, a dedicated extraction algorithm could identify and place its underlying structure in such way that wiring and timing is optimized. For example algorithms developed for the structural verification of multipliers are a good starting points.

In addition, the problem of placing large regular groups could be addressed. A first approach focuses on modeling the cells of a regular group as strongly connected but individual objects, instead of a single block. Connections between the objects of the group are assigned relatively high weights such that the structure of the group is preserved in the placement process, as shown in figure 8.1. Connections outside the group are treated normally, which would allow the group to move more freely. This idea can also be extended to macro cells and other large objects. Likewise, instead of feeding the placement algorithm a fixed two-dimensional structure, some flexibility in the placement of a regular structure might be beneficial. Instead of hard-placing two-dimensional structures, the concept of “soft groups” can be applied. Objects within regular groups would be allowed to move slightly during standard cell placement, i.e. the regularity is used to seed the actual placement.

In the area of regularity driven synthesis, future work will focus on the addition of symmetry information to the synthesis process. The identification of symmetric decomposition functions in a design appears to be a promising addition to structural regularity extraction. After finding symmetrical functions in a datapath, pins swaps can be performed with the goal of improving the critical delay. This is particularly useful

after placement and buffering because the estimated delays will be more accurate. In addition, this idea is very promising for relatively regular adder and multiplier structures that contain a large number of easily identified symmetries.

Furthermore, the application of multi-level clustering during placement and a more seamless integration of layout driven synthesis and placement algorithms is desirable. In addition, the non-trivial task of parallelizing logic minimization of the synthesis partitions yields the potential for further speedup. This, however, requires the dynamic generation and update of timing constraints at the partition boundaries.

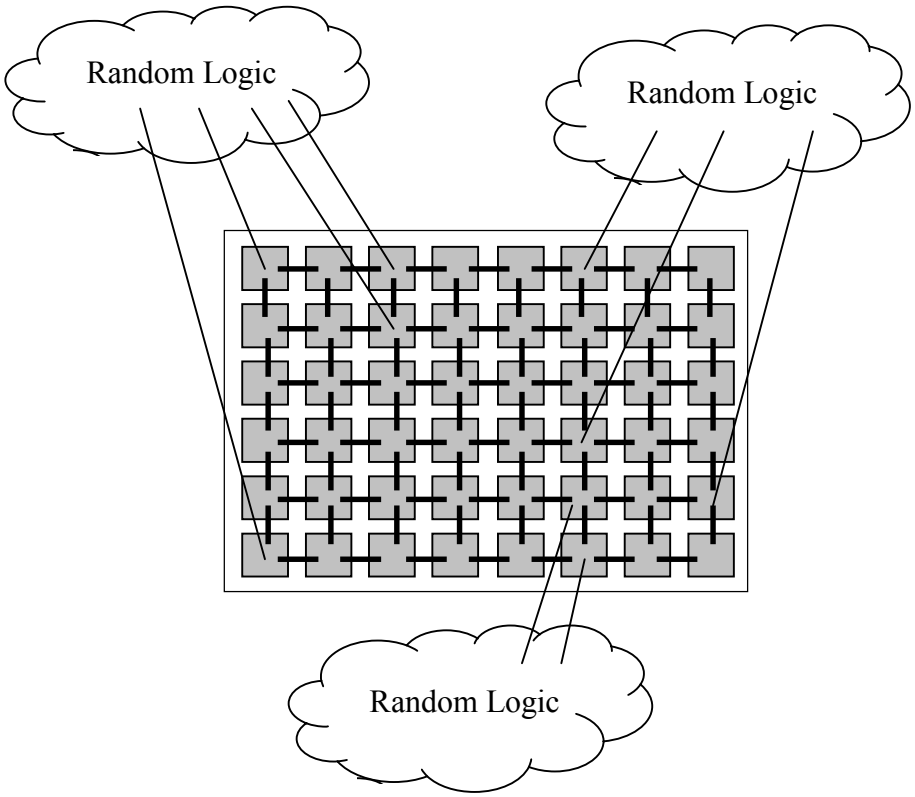


Figure 8.1: Placement of a regular group using net weights

Bibliography

[1] S. B. Akers. “Binary Decision Diagrams”, In *IEEE Transactions on Computers*, vol. C-27, pp. 509-516, June 1978.

[2] C. J. Alpert, A. Devgan and S. T. Quay. “Buffer Insertion for Noise and Delay Optimization”, In *Proc. of Design Automation Conference*, pp. 362-367, 1998.

[3] S. R. Arikati and R. Varadarajan. “A signature based approach to regularity extraction”, In *Proc. of the Int. Conference on CAD*, pages 542-545, Nov. 1997.

[4] R. L. Ashenhurst. “The Decomposition of Switching Functions”, In *Proceedings of the International Symposium on Theory of Switching*, Annals of Computing Laboratory of Harvard Univ., pp. 74-116, vol. 29, 1959.

[5] C. L. Berman and L. Trevillyan. “A Global Approach to Circuit Size Reduction”. In MIT Press, *Advanced Research in* VLSI, 5th MIT Conference, pp. 69-99, 1989.

[6] C. L. Berman and L. Trevillyan. “Global Flow Optimization in Automatic Logic Design”, In *IEEE Transactions on Computer-Aided Design of Integrated Circuits,* 10(5), pp. 557-562, May 1991.

[7] C. L. Berman, J. L. Carter, and K. F. Day. “The Fanout Problem: From Theory to Practice”, In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of 1989 Decennial Caltech Conference*, pp. 69-99, MIT Press, March 1989.

[8] B. Bollobas. *Graph Theory*. Springer-Verlag, New York, 1979.

[9] G. Boole. *An Investigation of the Laws of Thought*. Walton, London, 1854.

[10] D. Brand, W. Joyner, L. Trevillyan, T. Nix and S. Gunderson. “Technology Adapation in Logic Synthesis”, In *Proceedings 23rd Design Automation Conference*, pp. 94-100 June 1986.

[11] R. K. Brayton. *Algorithms for Multilevel Logic Synthesis and Optimization*. NATO ASI on Logic Synthesis and Silicon Compilation, Kluwer, Dordrecht, 1987.

[12] R. K. Brayton. "Factoring Logic Functions", In *IBM Journal of Research and Development*, 31, March 1987.

[13] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. *Multilevel Logic Synthesis*. Notes for Lectures at Oxford/Berkeley Summer Engineering Programm, July 1989.

[14] R. K. Brayton and C. T. McMullen. "The Decomposition and Factorization of Boolean Expressions", In *Proceedings of the IEEE Symposium on Circuits and Systems*, pp. 49-54, May 1982.

[15] M. A. Breuer. "A Class of Min-Cut Placement Algorithms", In *Proceedings of 14th Design Automation Conference*, pp. 284-290, 1977.

[16] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer, Boston, 1990.

[17] R. Bryant. "Graph-based Algorithms for Boolean Function Manipulation", In *IEEE Transactions on Computers*, vol. 35, pp. 677-691, August 1986.

[18] T. Chan, A. Chowdhary et al. "Challenges of CAD Development for Datapath Design", In *Intel Technical Journal*, 01/1999.

[19] T. F. Chan, J. Cong et al. "Multilevel Optimization for Large-Scale Circuit Placement", In *Proceedings of the International Conference on Computer Aided Design*, pp. 171-176, November 2000.

[20] K.-T. Chen and L. A. Entrena. "Multi-Level Logic Optimization by Redundancy Addition and Removal", In *Proceedings European Conference on Design Automation*, 1993.

[21] A. Chowdhary et al. "A general approach for regularity extraction in datapath circuits", In *Proc. of the International Conference on CAD*, pages 332-338, 1998.

[22] J. Cong and S. K. Lim. "Edge Separability Based Circuit Clustering with Application to Circuit Partitioning", In *Proceedings of the Conference on Asia and South Pacific Design Automation*, pp.429-434, 2000.

[23] M. R. Corazo et al. "Performance optimization using template mapping for datapath-intensive high-level synthesis", In *IEEE Transactions on CAD*, pages 877-887, August 1996.

[24] O. Coudert, R. Haddad and S. Manne. “New Algorithms for Gate Sizing: A Comparative Study”, In *Proceedings of the Design Automation Conference*, pp. 734-739, 1996.

[25] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, 1962.

[26] J. Darringer, W. Joyner, L. Bergman and L. Trevillyan. “LSS: Logic Synthesis through Local Transformations”, In *IBM Journal of Research and Development,* 25(4), pp. 272-280, July 1981.

[27] M. Damiano and G. De Micheli. “Observability Don’t Care Sets and Boolean Relations”, In *Proceedings International Conference on Computer-Aided* Design, pp. 502-505, November 1990.

[28] R. F. Damiano, A. Drumm et al. *Method for mapping in logic synthesis by logic classification*. U.S. Patent No. 5,537,330.

[29] M. Davio, J.-P. Deschamps, and A. Thayse. *Discrete and Switching Functions*. McGraw-Hill, New York, 1978.

[30] S. Dey, F. Brglez and G. Kedem. “Corolla Based Circuit Partitioning and Resynthesis”, In *Proc. Design Automation Conference,* pp. 607-612, June 1990.

[31] W. Donath, P. Kudva, L. Stok, P. Villarubia, L. Reddy and S. Sullivan. “Transformational Placement and Synthesis”, In *Proceedings of Design Automation and Test in Europe*, pp. 194-201, March 2000.

[32] H. Eisenmann, F. M. Johannes. “Generic Global Placement and Floorplanning”, In *Proceedings of 35th Design Automation Conference*, pp. 269-274, June 1998.

[33] W.C. Elmore. “The Transient response of Damped Linear Networks with Particular Regard to Wireband Amplifiers”, *Journal of Applied Physics*, Vol. 19, pp 55-63, January 1948.

[34] J. P. Fishburn. “LATTIS: An Iterative Speedup Heuristic for Mapped Logic”. In *Proceedings Design Automation Conference*, pp. 488-491, 1992.

[35] A. A. Fraenkel. *Abstract Set Theory*. North-Holland Publishing, Amsterdam, 1976.

[36] H. Fujiwara and T. Shimono. “On the Acceleration of Test Generation Algorithms”, In *IEEE Transactions on Computers,* (32), pp. 1137-1144, 1983.

[37] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. "Socrates: A System for automatically Synthesizing and Optimizing Combinational Logic", In *Proceedings 23rd Design Automation Conference*, pp. 79-85, 1986.

[38] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publisher, Dordrecht, 1996.

[39] P.R. Halmos. *Lectures on Boolean Algebras*. Nostrand Reinhold, London, 1963 .

[40] F. Harary. *Graph Theory*. Addison-Wesley, 1971.

[41] S. Hassoun and T. Sasao, editors. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Dordrecht, 2002.

[42] F. J. Hill and G. R. Peterson. *Computer Aided Logical Design*. fourth ed. John Wiley, New York, 1992.

[43] S.-T. Hui and D.M. Wong. *Method for regular placement of data path components in VLSI circuits.* U.S. Patent No. 5,359,538.

[44] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar. "Multilevel Hypergraph Partitioning: Application in VLSI Domain", In *Proceedings of the Design Automation Conference*, pp. 526-529, 1997.

[45] G. Karypis and V. Kumar. "Multilevel k-way Hypergraph Partitioning", In *VLSI Design*, Vol. 11, 3(2000), pp. 285-300.

[46] K. Keutzer. "DAGON: Technology Binding and Local Optimization by DAG Matching", In *Proceedings of the 24th Design Automation Conference*, pp. 341-347, June 1987.

[47] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. "Optimization by Simulated Annealing", In *Science*, 220(4598):671-680, 1983.

[48] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization", In *IEEE Transactions on CAD*, 10, pp. 356-365, March 1991.

[49] Z. Kohavi. *Switching and Finite Automata Theory*. Second Ed. McGraw-Hill, New York, 1978.

[50] V. N. Kravets and K. A. Sakallah. "M32: A Constructive Multilevel Logic Synthesis System", In *Proceedings of the Design Automation Conference*, pp. 336-341, June 1998.

[51] P. Kudva, D. Kung et al. “Gate-Size Selection for Standard Cell Libraries”, In *Proceedings of the International Conference on CAD*, pp. 545-550, 1998.

[52] Y. Kukimoto, R. K. Brayton and P. Sawkar. “Delay-Optimal Technology Mapping by DAG Covering”, In *Proceedings of the Design Automation Conference*, pp. 348-351, 1998.

[53] D. Kung. “A Fast Fanout Optimization Algorithm for Near-Continuous Buffer Libraries”, In *Proceedings of Design Automation Conference*, pp. 352-355, 1998.

[54] D. Kung and R. Puri. “Optimal P/N Width Ratio Selection for Standard Cell Libraries”, In *Proceedings of the International Conference on Computer Aided Design*, pp. 178-184, 1998.

[55] F. J. Kurdahi and D. S. Rao. “On clustering for maximal regularity extraction”, In *IEEE Transactions on CAD*, pages 1198-1208, August 1993.

[56] T. Kutzschebauch. “Efficient Logic Optimization Using Regularity Extraction”. In *Proceedings of the Internat. Conference on Computer Design*, pp. 487-493, September 2000.

[57] T. Kutzschebauch. “Logic Optimization Using Regularity Extraction”. In *International Workshop on Logic Synthesis*, pp. 264-270, June 1999.

[58] T. Kutzschebauch and L. Stok. “Regularity Driven Logic Synthesis”, In *Proc. of the International Conference on Computer Aided Design*, pp. 439-446, Nov. 2000.

[59] T. Kutzschebauch and L. Stok. *A Method for Preserving Regularity During Logic Synthesis*. U.S. Patent, May 2000.

[60] T. Kutzschebauch and L. Stok. “Regularity Driven Logic Synthesis”, In *International Workshop on Logic Synthesis*, pp. 303-310, June 2000.

[61] T. Kutzschebauch and L. Stok. “Congestion Aware Layout Driven Synthesis”, In *Proc. of the International Conference on Computer Aided Design*, pp. 216-223, November 2001.

[62] T. Kutzschebauch and L. Stok. “Layout Driven Decomposition with Congestion Consideration”, In *Proc. of Design Automation and Test in Europe*, pp. 672-676, March 2002.

[63] C. Lee. “Representations of Switching Circuits by Binary-Decision Programs”, *Bell Systems Technical Journal*, vol. 38, pp. 985-999, July 1959.

[64] S. C. Lee. *Modern Switching Theory and Digital Design*. Prentice-Hall, Englewood Cliffs, 1978.

[65] J. Lou, A. Salek, M. Pedram. "An Exact Solution to Simultaneous Technology Mapping and Linear Placement Problem". In *Proceedings International Conference Computer-Aided Design*, pp. 671-675, Novemeber 1997.

[66] F. Maamari and J. Rajski. "A Reconvergent Fanout Analysis for Efficient Exact Fault Simulation of Combinatorial Circuits", In *Proceedings 18th Int. Symposium Fault Tolerant Computing*, June 1988.

[67] Marshburn et al. "Datapath: A CMOS Datapath Silicon Assembler", In *Proceedings of the Design Automation Conference*, pages 722-12, 1986.

[68] E. J. McCluskey. *Introduction to the Theory of Switching Circuits*. McGraw-Hill, New York, 1965.

[69] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall, Englewood Cliffs, 1986.

[70] E. J. McCluskey. "Minimization of Boolean Functions". In *Bell Syst. Technical Journal,* 35 (1956), pp. 1417-1444.

[71] S. Muroga, Y. Kambayashi, H.C. Lai, and J. N. Culliney. "The Transduction Method – Design of Logic Networks based on permissible Functions", In *IEEE Transactions on Computers*, C-38(10), pp. 1404-1424, October 1989.

[72] R. X. T. Nijssen and J. A. G. Jess. "Two-dimensional Datapath Regularity Extraction", In *Proceedings of the 5th ACM/IEEE Physical Design Workshop*, Reston, Virginia, 1996.

[73] R. X. T. Nijssen and C. A. J. van Eijk. "Regular Layout Generation of Logically Optimized Datapaths", *In Proceedings of the International Symposium on Physical Design*, pages 42-47, 1997.

[74] G. Odawara et al. "Partitioning and Placement Technique for CMOS Gate Arrays", In *IEEE Transactions on CAD*, pages 355-363, May 1987.

[75] J. Lou, W. Chen and M. Pedram. "Concurrent Logic Restructuring and Placement for Timing Closure", In *Proc. of the International Conference on Computer-Aided Design,* pp. 31-36, 1999.

[76] M. Pedram and N. Bhat. "Layout Driven Logic Restructuring/Decomposition", In *Proc. of the Internat. Conference on Computer Aided Design,* pp.134-137, 1991.

[77] M. Pedram and N. Bhat. “Layout Driven Technology Mapping“, In *Proceedings of the 28th Design Automation Conference*, pp. 99-105, 1991.

[78] W. Quine. “The Problem of Simplifying Truth Functions”, In *American Math. Monthly*, 59 (1952), pp. 521-531 .

[79] W. Quine. “A Way to Simplify Truth Functions”, In *American Math. Monthly*, 62 (Nov. 1955), pp. 627-631.

[80] J. Rajski and J. Vasudevamurthy. “A Method for Concurrent Decomposition and Factorization of Boolean Expressions”, In *Proc. of the Int. Conference on Computer-Aided Design*, pp. 510-513, 1990.

[81] J. Rajski and J. Vasudevamurthy. “The Testability-Preserving Concurrent Decomposition and Factorization of Boolean Expressions“, In *IEEE Trans. on CAD,* Vol. 11, No. 6, June 1992, pp. 778-793.

[82] J. P. Roth and R. Karp. “Minimization over Boolean Graphs”, *IBM Journal of Research and Development, 6(2),* pp. 227-238, April 1962.

[83] S. Rudeanu. *Boolean Functions and Equations*. North-Holland Publishing, Amsterdam, 1974.

[84] R. Rudell. *Logic Synthesis for VLSI Designs*. PhD thesis, UC Berkeley, 1989.

[85] D. M. Schuler and E. G. Ulrich. “Clustering and Linear Placement”, In *Proceedings of the Design Automation Conference*, 1972.

[86] C. Shannon. “A Symbolic Analysis of Relay and Switching Circuits”, In *Transactions on AIEE*, vol. 57, pp. 713-723, 1938.

[87] Naweed Sherwani. *Algorithms for VLSI Physical Design Automation*. Third Ed., Kluwer Academic Publishers, Dordrecht, 1999.

[88] R. Sikorski. *Boolean Algebras*. Second Ed., Springer-Verlag, Berlin 1964.

[89] A. Srivastava, R. Kastner and M. Sarrafzadeh. “Timing Driven Gate Duplication: Complexity Issues and Algorithms”, In *Proceedings of the International Conference Computer Aided Design*, pp. 447-450, November 2000.

[90] G. Stenz et al. “Timing Driven Placement in Interaction with Netlist” Transformations. In *Proc. of the Internat. Symposium on Physical Design,* 1997.

[91] L. Stok, D. Brand et al. ”Booledozer: Logic Synthesis for ASICS” In *IBM Journal of Research and Development,* 40(4):515-547, July 1996.

[92] L. Stok, M. A. Iyer, and A. Sullivan. “Wavefront Technology Mapping”, In *Proceedings of Design Automation and Test in Europe*, March 1999.

[93] K. Sulimma, I. Neumann, W. Kunz, and L. Ginneken. "Improving Placement under the Constant Delay Model", In *Proceedings of Design Automation and Test in Europe*, pp. 677-682, 2002.

[94] I. Sutherland, and R. Sproull. “The theory of logical effort: Designing for speed on the back of an envelope.”, In *Advanced Research in VLSI*, University of California at Santa Cruz, 1991.

[95] H. Touati. *Performance Oriented Technology Mapping*. PhD thesis, University of California, Berkeley, 1990.

[96] J. Vygen. *Plazierung im VLSI Design und ein zweidimensionales Zerlegungs-problem. PhD thesis*. University of Bonn, 1996.

[97] J. Vygen. "Algorithms for Large-Scale Flat Placement", In *Proceedings of the 34th Design Automation Conference*, pp. 746-751, 1997.

[98] Y. Watanabe, E. Lehman, et al. “Logic Decomposition during Technology Mapping”, In *IEEE Transactions on CAD*, vol. 16, no. 8, pp. 813-834, August 1997.

[99] J. L. Wyatt. *Circuit Analysis, Simulation and Design*. Elsevier Science Publishers, North-Holland, 1987.

Curriculum Vitae

Thomas Kutzschebauch, born October 8, 1971 in Chemnitz, Germany

1978 – 1988	High School for Linguistically Talented Students in Chemnitz
1988 – 1991	Gymnasium Chemnitz (senior high school)
06/91	Abitur (high school graduation)
1991 – 1992	Military Service in the German Airforce
10/92 – 09/94	Undergraduate Studies in Electrical Engineering at the Technical University of Chemnitz, Germany
10/94 – 08/98	Graduate Studies in Electrical and Computer Engineering at the University of Stuttgart, Germany
09/95 – 08/96	Fulbright Scholarship, Graduate Studies in Computer Engineering at the University of Wisconsin-Madison, WI, USA
06/97 – 09/97	Summer Internship with IBM, Portsmouth, UK. Development of Distributed Systems Solutions.
02/98 – 07/98	Master Thesis at the IBM R&D Lab in Boeblingen, Germany. Static Timing Analysis and Development of False Path Algorithms in the Physical Design Group.
07/98	Graduation from the University of Stuttgart with the Academic Degree of Diplom-Ingenieur in Electrical Engineering
10/98 – 01/02	Employment as Research Student at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, while being Ph.D. Student with Prof. Wolfgang Kunz, Design Automation Group, Dept. of Computer Science at the University of Frankfurt
02/02 – 06/02	Ph.D. Student with Prof. Wolfgang Kunz, Design Automation Group, Dept. of Electrical Engineering at the University of Kaiserslautern
since 07/02	Employment with Magma Design Automation, Inc., Cupertino, CA, USA. Development of Physical Synthesis Algorithms, Member of Consulting Staff