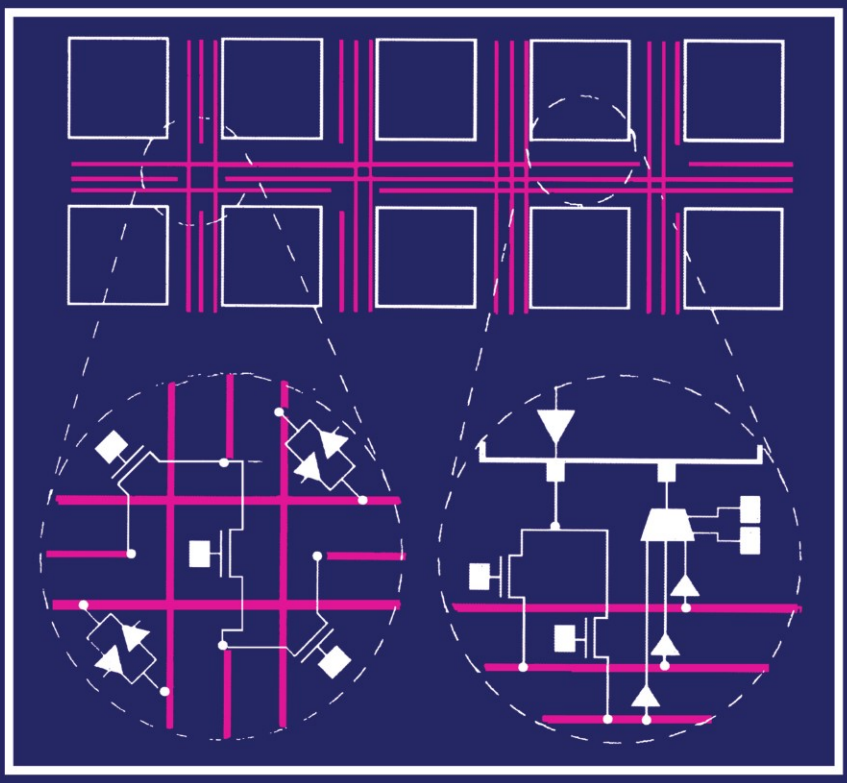
Architecture and CAD for Deep-Submicron FPGAs



Vaughn Betz
Jonathan Rose
Alexander Marquardt

ARCHITECTURE AND CAD FOR DEEP-SUBMICRON FPGAs

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

ARCHITECTURE AND CAD FOR DEEP-SUBMICRON FPGAs

Vaughn Betz
Jonathan Rose
Alexander Marquardt
University of Toronto

Springer Science+Business Media, LLC

Library of Congress Cataloging-in-Publication Data

Betz, Vaughn.
Architecture and CAD for deep-submicron FPGAs / Vaughn Betz, Jonathan Rose, Alexander Marquardt.
p. cm. -- (Kluwer international series in engineering and computer science ; SECS 497)
Includes bibliographical references and index.
ISBN 978-1-4613-7342-1 ISBN 978-1-4615-5145-4 (eBook)
DOI 10.1007/978-1-4615-5145-4
1. Field programmable gate arrays--Design and construction--Data processing. 2. Programmable array logic. 3. Computer-aided design.
I. Rose, Jonathan. II. Marquardt, Alexander. III. Title.
IV. Series.
TK7895.G36B48 1999
621.39'5--dc21 00-11904
CIP

Third Printing 2002.

Originally published by Kluwer Academic Publishers in 1999
Softcover reprint of the hardcover 1st edition 1999

Printed on acid-free paper.

This printing is a digital duplication of the original edition.

To Corinne, William, and Isabel,

Betty and Frank Rose,

Marie and Ron

Table of Contents

CHAPTER 1 *Introduction*

Since their introduction in 1984, Field-Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits and have grown into a $2 billion per year industry. As process geometries have shrunk into the deep-submicron region, the logic capacity of FPGAs has greatly increased, making FPGAs a viable implementation alternative for larger and larger designs. To make the best use of these new deep-submicron processes, one must re-architect one's FPGAs and Computer-Aided Design (CAD) tools. This book addresses several key issues in the design of high-performance FPGA architectures and CAD tools, with particular emphasis on issues that are important for FPGAs implemented in deep-submicron processes.

Three factors combine to determine the performance of an FPGA: the quality of the CAD tools used to map circuits into the FPGA, the quality of the FPGA architecture, and the electrical (i.e. transistor-level) design of the FPGA. In this book, we examine all three of these issues in concert, and we believe this is the first systematic study of FPGA architecture and CAD to do so.

In order to investigate the quality of different FPGA architectures, one needs CAD tools capable of automatically implementing circuits in each FPGA architecture of interest. Once a circuit has been implemented in an FPGA architecture, one next needs accurate area and delay models to evaluate the quality (speed achieved, area required) of the circuit implementation in the FPGA architecture under test. This book therefore has three major foci: the development of a high-quality and highly *flexible* CAD infrastructure, the creation of accurate area and delay models for FPGAs, and the study of several important FPGA architectural issues.

Throughout this book we compare FPGA architectures by estimating the area each requires and the speed each achieves in a deep-submicron (0.35 μm) process. Our area model is a detailed estimate of the layout area required by an FPGA, and our delay model includes important deep-submicron effects, such as the significant resistance and capacitance of metal lines, that have often been neglected in earlier research. We believe that many FPGA architectural issues can be adequately studied only with detailed models of FPGA circuitry implemented in the target fabrication process, so this book discusses such issues extensively.

Industrial FPGA research and development tends to focus on point solutions — industrial architects make reasonable design choices and educated guesses to get a product to market in a timely fashion. The documentation of industrial devices do not indicate which design choices were carefully researched and which were educated guesses, and they usually do not investigate (or at least do not publish) the effect radically different design choices would have had. In this book we deliver precisely this kind of knowledge — all the information leading to good choices for a broad set of FPGA architectures, and complete discussions of the trade-offs involved. We determine the best choice for various architecture parameters, the sensitivity of FPGA performance to each parameter, and useful "rules of thumb" for designing good FPGA architectures.

1.1 Overview of FPGAs

The key to FPGAs' popularity is their ability to implement *any* circuit simply by being appropriately programmed. Other circuit implementation options, such as Standard Cells or Mask-Programmed Gate Arrays (MPGAs), require that a different VLSI chip be newly fabricated for each design. The use of a standard FPGA, rather than these custom technologies, has two key benefits: lower non-recurring engineering (NRE) costs, and faster time-to-market.

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

Time-to-market is the other key advantage of FPGAs. Full fabrication typically takes 6 - 8 weeks. If problems are found in the finished chip it must be thrown away, and one must wait another 6 - 8 weeks to fabricate a corrected design. FPGAs, on the other hand, can be programmed in seconds, and any bugs found once the chip is tested in system can be corrected in minutes by reprogramming the FPGA. With today's short product cycles, the time-to-market advantage this provides is often compelling.

FPGA programmability carries a price, however. In MPGAs and Standard Cells circuitry is interconnected with metal wires. FPGAs, in contrast, must connect circuitry via programmable switches. These switches have higher resistance than metal wires and add significant capacitance to connections, reducing circuit speed. The switches also take up more area than metal wires would, so an FPGA must be considerably larger than an MPGA to implement the same circuit. A circuit implemented in an FPGA is typically 10 times larger and roughly 3 times slower than the same circuit implemented via an MPGA in an equivalent process [1]. The larger size of FPGA circuitry makes FPGA implementations more expensive than MPGAs for high volume designs, and the limited speed of FPGAs precludes their use in very high-speed designs. These differences mandate research into new FPGA architectures in order to reduce these speed and density penalties. In addition, because the FPGA marketplace is highly competitive each FPGA manufacturer is constantly searching for better FPGA architectures in order to gain a speed and density advantage.

1.2 FPGA Architectural Issues

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

The first issue we investigate is *global routing* architecture [2, 3]. The global routing architecture of an FPGA specifies the relative width of the various wiring channels within the chip. Figure 1.1 depicts an example global routing architecture in which the channels near the center of the FPGA are wider than those near the edges. In MPGA and Standard Cell implementations, a custom chip is created for each design, so routing channels can easily be made wider in areas of a chip where the demand for routing is greater. In FPGAs, however, all routing resources are prefabricated, so the width of all the routing channels is set by the FPGA manufacturer. Our goal, then, is to find the distribution of routing resources, or tracks, to the various channels that permits their efficient utilization by the largest class of circuits. If there are too few

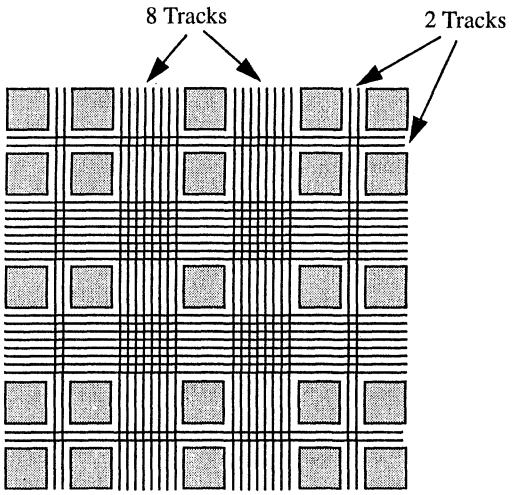


FIGURE 1.1 Example global routing architecture.

tracks in some area of the chip then many circuits will be unroutable, while if there are too many tracks, they may be wasted. There is no agreement amongst commercial FPGAs on the best global routing architecture [4, 5, 6, 7], so this question has clear relevance.

The second issue we explore is the use of *cluster-based* logic blocks in FPGAs [8, 9, 10]. These logic blocks are groups, or clusters, of look-up tables (LUTs) and flip flops along with local routing to interconnect the LUTs within a cluster; Figure 1.2 depicts an example logic cluster. In an FPGA using cluster-based logic blocks, many connections can be made using the local interconnect within a cluster. Since this local interconnect can be made faster than the general-purpose interconnect between logic

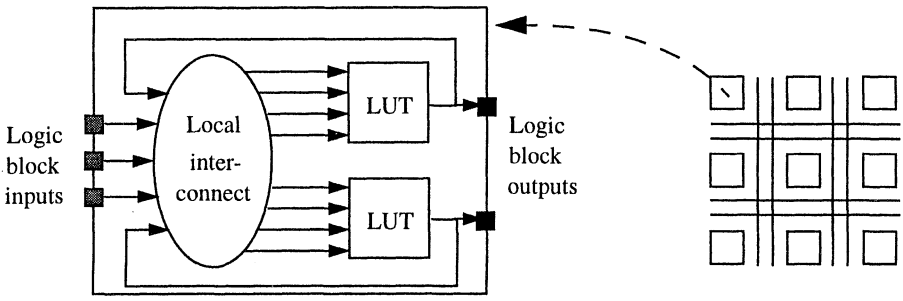


FIGURE 1.2 Example logic cluster containing two LUTs.

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

We will investigate how the number of LUTs per cluster affects FPGA speed, area, compile time and other important FPGA architecture parameters, such as the proper number of inputs to a logic block and the required flexibility of the general-purpose interconnect. The trade-offs among these architectural issues and metrics are quite complex. For example, grouping related LUTs together into a single logic block reduces the number of connections to be routed between logic blocks, saving routing area. Since the general-purpose interconnect consumes most of the die area in SRAM-based FPGAs [1], this is a significant area savings. On the other hand, in the logic clusters we study, the area required by the local routing grows quadratically with the number of LUTs in a cluster. For sufficiently large clusters, the area used by this local interconnect will exceed the area saved in the general interconnect. Similarly, since the local connections within clusters can be made very fast, we expect clusters containing a larger number of LUTs to attain higher circuit speeds. At some point, however, the speed gained by increasing the cluster size will diminish or be lost because increasing the cluster size slows the local cluster routing.

While recent FPGAs from Xilinx [11], Altera [7], Lucent Technologies [5], Actel [12] and Vantis [13] have all grouped several LUTs together into larger logic blocks, there has been little published work investigating the impact of the number of LUTs per logic block on FPGA speed or area.

The final FPGA issue we examine is that of *detailed routing* architecture [14]. The detailed routing architecture of an FPGA defines how logic block inputs and outputs can be interconnected. The style of detailed routing architecture we will investigate is the island-style architecture [1] used by Xilinx, Lucent, and Vantis FPGAs [4, 5, 13]. A simplified detailed-routing architecture is depicted in Figure 1.3. In many ways, detailed routing architecture is *the* key element of an FPGA because:

1. Most of an FPGA's area is devoted to routing;
2. Most of a circuit's delay is due to routing delays rather than logic block delays [15, 1]; and

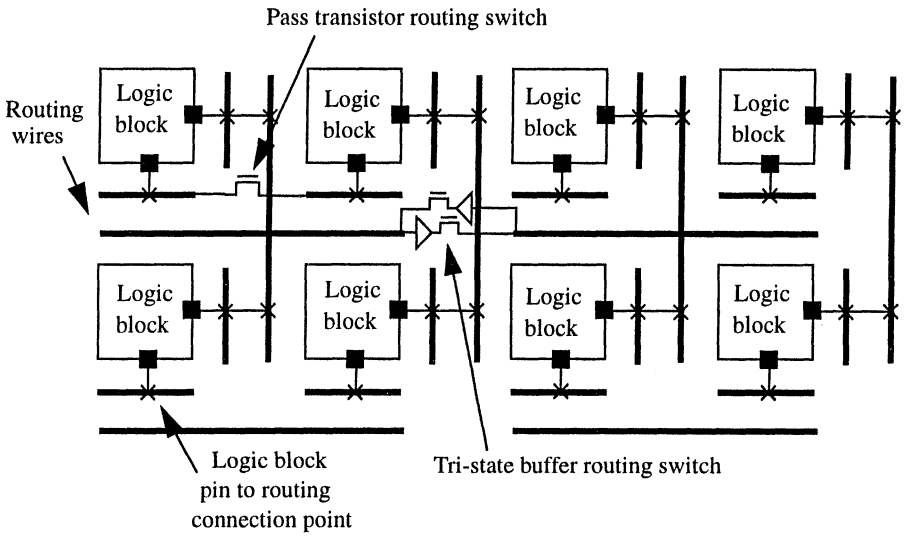


FIGURE 1.3 Example detailed routing architecture.

3. Interconnect delay does not scale as well as logic delay with process shrinks, so the fraction of delay due to routing in FPGAs is increasing with each process generation [16].

The detailed routing architecture defines such features as:

- Which routing wires in the channel adjacent to a logic block input or output can connect to that logic block pin,
- How many logic blocks each routing wire spans before terminating,
- Where routing switches are located and which routing wires they can connect together,
- Whether each routing switch is a pass transistor or a tri-state buffer,
- The sizes of the transistors used to build the various programmable switches, and
- The metal width and spacing of the various routing wires.

The design of a good detailed routing architecture is a very challenging problem, as the best value for each of the parameters above depends on complex trade-offs. For example, if we have too many programmable switches we will clearly waste area, and the increased delay caused by connections passing through many switches will degrade circuit speed. If, on the other hand, we have too few programmable switches, or if they are arranged in a poor topology, circuits will be either wasteful of wires or

completely unroutable. Achieving an appropriate balance between pass transistor based and tri-state buffer based switches is also crucial — pass transistors require less area and are faster for connections that go through only a few switches, but buffers are faster for connections that go through many switches. Similarly, if we have too few long routing wires, long connections will have to pass through many programmable switches and will be slow. On the other hand, too many long wires will result in short connections being made via long wires, wasting area and degrading speed.

It is difficult to look at one detailed routing architecture parameter in isolation because the architectural parameters interact in complex ways to determine an FPGA's speed and area-efficiency. In the research presented in this book, we have tried to take a holistic approach that looks at all the parameters determining a detailed routing together, and assesses both the speed and area of each FPGA we investigate. We believe that this is the first study to examine such a broad spectrum of detailed routing architectures.

1.3 Approach and CAD Tools

Our approach to exploring FPGA architecture is an experimental one. Benchmark circuits are technology mapped, placed and routed into each FPGA architecture of interest and the area and/or delay of each circuit in that architecture is measured. These measurements allow us to determine the best architectural parameters.

In order to evaluate different architectures, the CAD tools used in the empirical approach must be flexible enough to target all the architectures of interest. This precludes the use of commercial FPGA CAD tools, which target only the FPGAs of a specific manufacturer. Accordingly, we have created new CAD tools to pack multiple LUTs into a logic cluster, to place logic blocks within an FPGA, and to route the connections between logic blocks [17, 10]. In addition to making these CAD tools flexible, we also took considerable care to ensure they produced high quality results, as poor CAD tools can lead to inaccurate architecture conclusions. The CAD tools understand the special features of each architecture, and attempt to fully optimize for each architecture they target. They incorporate several new features and enhancements to existing algorithms, so the CAD algorithms are of interest in their own right. As well, these CAD tools are in use in numerous FPGA and CAD studies at the University of Toronto and around the world, and obtain the highest quality results on the set of standard benchmarks used to compare academic place and route tools [17].

Ideally, one would lay out each FPGA architecture of interest to obtain exact area measurements and highly accurate delay values. In this book we study hundreds of

different FPGA architectures, so this is clearly impossible. Accordingly, we have developed more abstract area models and delay estimates that do not require layout of the FPGAs, but which are still sufficiently accurate to allow fair comparison of different architectures.

1.4 Book Organization

Most of the material in this book is adapted from the Ph.D. thesis of Vaughn Betz [18], with important additional information (the T-VPack algorithm described in Chapter 3 and most of the experimental results and analysis concerning logic clusters presented in Chapter 6) adapted from the M.A.Sc thesis of Alexander Marquardt [19].

The next chapter provides background information and details some of the previous work in the relevant areas of both CAD and FPGA architecture. Chapter 3 describes the CAD algorithms we use to pack several LUTs together into a logic cluster, and the algorithms we use to perform placement of logic blocks in an FPGA. In Chapter 4 we detail the algorithms used in our FPGA router.

Chapters 5 through 7 then employ these CAD tools to investigate the three FPGA architectural issues outlined earlier. Each of these three chapters begins by describing the CAD flow and area and/or delay models used in the experimental study, and then presents the architectural results obtained. Chapter 5 evaluates different global routing architectures for FPGAs, and determines the most area-efficient architectures. In Chapter 6 we investigate cluster-based logic blocks and determine the logic cluster types that maximize area-efficiency and speed. In Chapter 7 we ascertain which detailed routing architectures lead to FPGAs with the best combination of speed and area-efficiency. The final chapter summarizes the architectural conclusions and provides suggestions for future work.

The three appendices provide additional useful information. Appendix A outlines the graphic visualization features incorporated in our placement and routing CAD tools, and highlights their usefulness in algorithm and architecture research. Appendix B describes the transistor-level design of an FPGA and our characterization of a deep-submicron IC process. Finally, in Appendix C we investigate how routing transistors and metal routing wires should be sized in order to create an FPGA with the best combination of area-efficiency and speed. The detailed information concerning the electrical design of an FPGA provided by Appendices B and C is essential to create accurate FPGA area and delay models, and hence to properly compare FPGA architectures.

1.5 Acknowledgments

The authors are indebted to Carl Harris of Kluwer Academic Publishers for his help and advice as we prepared this book. We would like to thank the other members of our research group at the University of Toronto — Steve Wilton, Mike Hutton, Mohammed Khalid, Yaska Sankar, Jordan Swartz, Rob McCready, and Paul Leventis — for the many valuable suggestions and insightful discussions they have provided over the years. Special thanks are due to Jordan Swartz for enhancing the VPR FPGA architecture generator to allow targeting of an architecture very similar to that of the Xilinx 4000X series FPGAs. Many other students and faculty in the University of Toronto Electrical Engineering department lent us their expertise on vexing problems; we would particularly like to thank Dave Lewis, Mark Stoodley, Guy Lemieux, Jason Anderson, and Jason Podaima. We also appreciate several interesting discussions on FPGA CAD with Russ Tessier of MIT.

The authors are grateful for many enlightening discussions with people in the FPGA industry. Particular thanks are due to Steve Trimberger, Steve Young, and Bill Carter of Xilinx, Frank Heile and David Mendel of Altera, Sinan Kaptanoglu of Actel, and Tim Lacey of Cypress. We appreciate the helpful suggestions of Jason Cong, Steve Trimberger, Sinan Kaptanoglu, Corinna Lee, Stephen Brown, Dave Lewis, and Derek Corneil concerning the Ph.D. thesis on which the majority of this book is based.

We gratefully acknowledge the financial support of our work provided by the Natural Sciences and Engineering Research Council of Canada (via an NSERC 1967 Science and Engineering Scholarship), the Information Technology Research Centre of Ontario, the Walter C. Sumner Foundation, a V. L. Henderson Research Fellowship, and Xilinx. Finally, we appreciate the 0.35 μm process data provided to us by the Canadian Microelectronics Corporation and the Taiwan Semiconductor Manufacturing Company (TSMC).

CHAPTER 2

Background and Previous Work

The first half of this chapter provides background information about FPGA architectures, and briefly describes the prior work relevant to this book. The second half of the chapter describes the CAD flow used to automatically map circuits into FPGAs and determine their speed, and summarizes some of the prior work in the relevant areas of CAD.

2.1 FPGA Architecture

All FPGAs are composed of three fundamental components: logic blocks, I/O blocks and programmable routing, as Figure 2.1 shows [20]. A circuit is implemented in an FPGA by programming each of the logic blocks to implement a small portion of the logic required by the circuit, and each of the I/O blocks to act as either an input pad or an output pad, as required by the circuit. The programmable routing is configured to make all the necessary connections between logic blocks and from logic blocks to I/O blocks. The following section briefly describes the basic technologies used to make FPGAs programmable. Sections 2.1.2 and 2.1.3 then describe some of the different architectures and prior research for FPGA logic blocks and routing, respectively.

2.1.1 FPGA Programming Technologies

There are three different approaches to making FPGAs programmable [1, 20]. The most popular technology today uses SRAM cells to control pass transistors, multi-

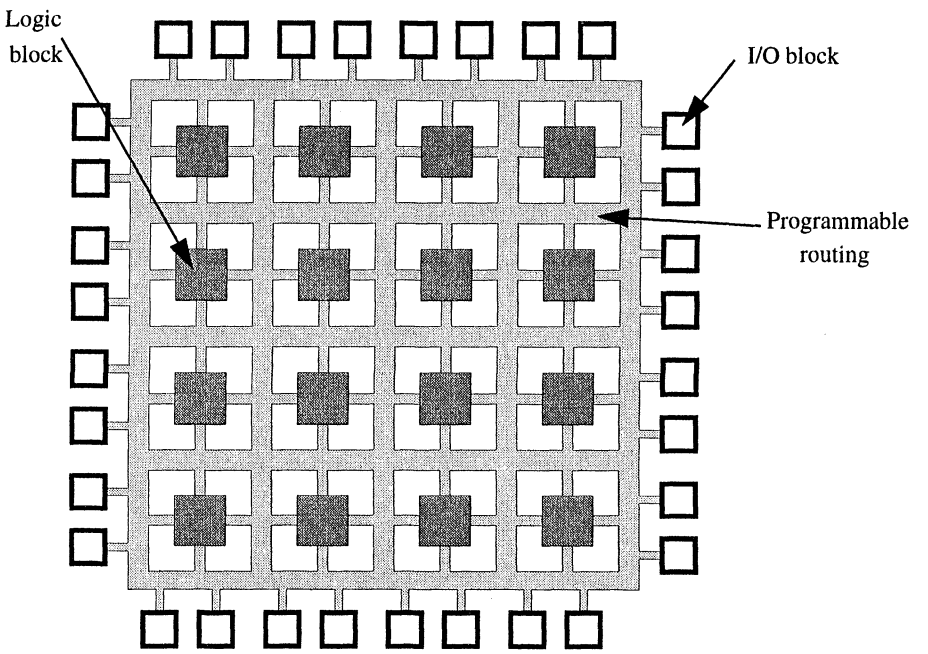


FIGURE 2.1 A generic FPGA (from [20]).

plexers and tri-state buffers in order to configure the programmable routing and logic blocks as required. Figure 2.2 shows these SRAM-based switches. Notice that pass gates are implemented with nMOS pass transistors, rather than complementary transmission gates, as this results in better speed due to the higher carrier mobility in nMOS transistors [21, 22]. Most Xilinx FPGAs [4], the larger Altera devices [7], the latest Actel FPGAs [12], all Lucent Technologies FPGAs [5] and the Vantis VF1 FPGAs [13] are SRAM-based. Alternative programming technologies are antifuses [23] which are used in many Actel FPGAs [6], and floating gate devices (e.g.

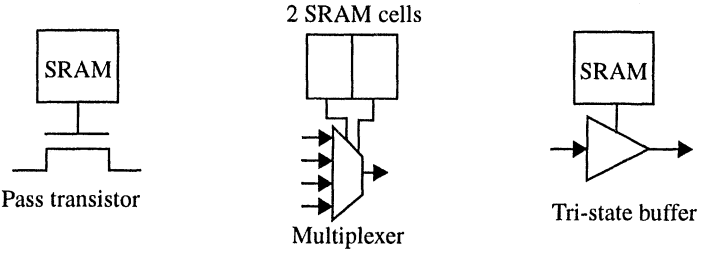


FIGURE 2.2 Three types of programmable switch used in SRAM-based FPGAs.

EPROM, EEPROM and Flash) which are used in many Complex Programmable Logic Devices (CPLDs) [24].

2.1.2 FPGA Logic Block Architecture

The logic block used in an FPGA strongly influences the FPGA speed and area-efficiency. While many different logic blocks have been used in FPGAs [1], most current commercial FPGAs use logic blocks based on look-up tables (LUTs). Accordingly, in this book we will investigate only LUT-based logic blocks. Figure 2.3 shows how a 2-input LUT can be implemented in an SRAM-based FPGA — a k-input LUT requires 2^k SRAM cells and a 2^k-input multiplexer. A k-input LUT can implement any function of k-inputs; one simply programs the 2^k SRAM cells to be the truth table of the desired function.

Previous research has shown that LUTs with 4-inputs lead to FPGAs with the highest area-efficiency [25], and most commercial FPGAs are based on 4-LUTs.

Most modern FPGAs are composed not of a single LUT, but of groups of LUTs and registers with some local interconnect between them, and there has been some research into logic blocks containing groups of LUTs. Jianshe He and Rose studied FPGAs that contained look-up tables of two different sizes [26]. Over a set of benchmark circuits, they found that an FPGA containing two-thirds 4-input LUTs and one-third 2-input LUTs reduced the number of SRAM bits within the LUTs by 22% and the number of logic block pins by 10% vs. an FPGA that contained only 4-LUTs. They never actually placed and routed an FPGA with a logic block composed of two 4-LUTs and one 2-LUT, but these results show that such a logic block should have good area-efficiency. In [27], Chung and Rose investigated logic blocks composed of several LUTs joined by inflexible "hardwired" connections. They found that some of these "hard-wired logic blocks" outperformed single 4-LUT logic blocks. In particular, a logic block composed of four 4-LUTs connected in a specific tree topology led

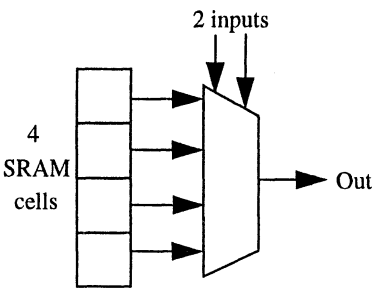


FIGURE 2.3 A 2-input LUT implemented in an SRAM-based FPGA.

to an FPGA 7% more dense and 28% faster than an FPGA using a single 4-LUT as its logic block. The area model used in this study modelled routing area by estimating the number of routing switches needed after global routing. Since this model did not account for the fact that the routing transistors would most likely have to be sized up for the larger logic blocks, these area numbers are probably somewhat inaccurate (overly optimistic) for logic blocks with more than one LUT. Cong and Hwang [28] examined how well four different logic blocks composed of groups of LUTs connected by hardwired connections, and in one case a multiplexer, were able to implement "wide" functions of many variables.

In [29], Aggarwal and Lewis investigated hierarchical FPGA architectures. Since the logic clusters we will investigate in Chapter 6 can be thought of as a logic block with a two-level hierarchy, this is somewhat related to our work.

2.1.3 FPGA Routing Architecture

Commercial FPGAs can be classified into three groups, based on their routing architecture. The FPGAs of Xilinx, Lucent and Vantis are *island-style* FPGAs, while Actel's FPGAs are *row-based*, and Altera's FPGAs are *hierarchical* [1]. In this book, we will almost exclusively investigate the island-style routing architecture,[1] so we describe this style of routing architecture below.

Figure 2.4 depicts an island-style FPGA. Logic blocks are surrounded by routing channels of pre-fabricated wiring segments on all four sides. A logic block input or output, which we call a pin, can connect to some or all of the wiring segments in the channel adjacent to it via a *connection block* [30] of programmable switches. At every intersection of a horizontal channel and a vertical channel, there is a *switch block* [30]. This is simply a set of programmable switches that allow some of the wire segments incident to the switch block to be connected to others; note that for clarity only a few of the programmable switches contained by switch blocks are shown in Figure 2.4. By turning on the appropriate switches, short wire segments can be connected together to form longer connections. In the FPGA of Figure 2.4, notice that some wire segments continue unbroken through a switch block. These longer wires span multiple logic blocks, and are a crucial feature in commercial FPGAs.

In [30], Rose and Brown developed a useful notation for describing some of the parameters of an FPGA's routing architecture, and we will use their notation in this book. The number of tracks, or wires, contained in a channel is denoted by W. The number of wires in each channel to which a logic block pin can connect is called the

1. Some of the results in Chapter 5 are also applicable to row-based FPGAs.

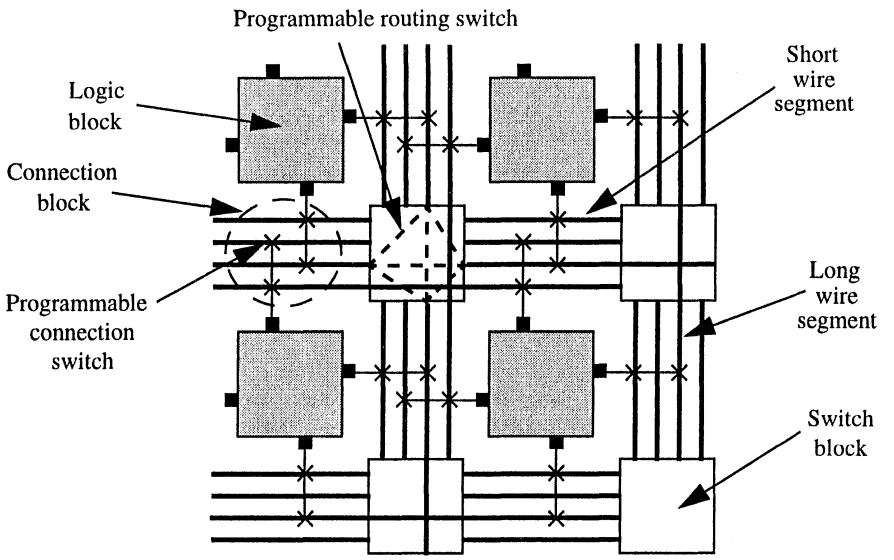


FIGURE 2.4 An island-style FPGA.

connection block flexibility, or F_c. The number of wires to which each incoming wire can connect in a switch block is called the switch block flexibility, or F_s. In the FPGA of Figure 2.4, for example, W is 4 for all channels, F_c is 2 and F_s is 3.

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

Most prior work has investigated FPGAs in which all wires span only one logic block before terminating at a switch block, and have compared architectures only on the basis of area-efficiency. The area-efficiency metric in these studies has usually been the number of programmable switches contained in the routing. In [30], Rose and Brown investigated FPGAs of this type and found that the most area-efficient architectures had $F_s = 3$ or 4 and $F_c = 0.7W$ to $0.9W$. Tseng et al showed that FPGAs with $F_s = 2$ can get reasonable area-efficiency despite the very low flexibility of their switch block if the CAD tools properly optimize for the architecture [31]. Even then, however, they found that $F_s = 3$ was still 25% more area-efficient than $F_s = 2$. Other

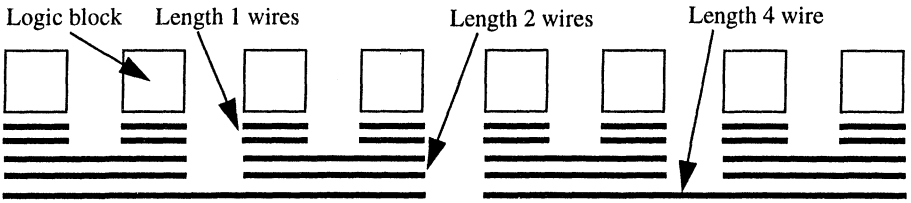


FIGURE 2.5 Example channel segmentation distribution.

researchers have focused on finding better $F_s = 3$ switch block topologies in order to enhance routability and hence area-efficiency [32, 33].

A few studies have looked at routing architectures that include different lengths of wires. Figure 2.5 shows one channel in such an FPGA. The *length* of a wiring segment is the number of logic blocks it spans; Figure 2.5 shows segments of length 1, 2 and 4. The *segmentation distribution* defines what fraction of the tracks in each channel are of each length. In Figure 2.5 for example, 40% of the tracks are of length 1, 40% are of length 2, and 20% are of length 4. When a wire spans the entire width or height of an FPGA, we will use Xilinx's terminology and call it a *long line.*

Greene et al briefly describe a procedure to choose an appropriate distribution of segment lengths for use in a antifuse-programmed row-based FPGA [34]. They looked at the distribution of connection lengths extracted from a set of global routings of circuits, and hand-crafted a segmentation distribution to match these lengths. Roy and Mehendale followed a similar procedure, again to find segmentation distributions for antifuse-programmed row-based FPGAs. They created a tool to automatically generate a segmentation length distribution that closely matched a specified length distribution [35]. They found that a Poisson distribution of segment lengths led to fewer series switches being used to make connections, on average, than a uniform distribution that made all segments span four logic blocks. Consequently, the Poisson distribution should lead to higher speed, but it required more routing area. Notice that both these studies assume that a distribution of connection lengths can be guessed or determined from a global routing. While this is possible in a row-based FPGA, island-style FPGAs usually perform global and detailed routing in one unified step, so these methods of generating segmentation distributions will not work well with island-style FPGAs.

Some prior work has also investigated segmentation for island-style FPGAs. In these studies, all the programmable switches within a switch block were assumed to be pass transistors; the possibility of tri-state buffer routing switches has not been investigated.

Brown et al investigated different segment length distributions in order to optimize the speed and area of an island-style FPGA [36, 37, 38]. They placed circuits, then routed them with a two-step (global then detailed) routing procedure, and evaluated the resulting speed and area for each FPGA architecture of interest. They found that segments of length greater than 3 or 4 were unnecessary, and that segments of length 1 were also not very important. The best segmentation distributions, in terms of speed/area trade-off, had 40 - 70% length 3 wires, 10 - 50% length 2 wires, and 10 - 20% length 1 wires. Their conclusions may have been affected by several factors, however. First, in the two-step routing scheme they use, the global router is unaware of the segmentation distribution, and hence is not attempting to optimize for it. This may make it difficult to use long wires effectively. Second, their area metric was number of tracks per channel required to route, W, while FPGA area is usually determined by transistor area. Third, their speed metric was average net delay, rather than the circuit critical path. Finally, the tools did not use timing analysis to determine which connections needed high-speed routing, and which could use area-efficient routes.

Chow et al [39] performed a similar study of segmentation distributions in an island-style FPGA, using similar tools to Brown et al [36]. They did not evaluate circuit speed, but assumed that longer wires would lead to faster circuits. Hence, they looked for segmentation distributions that incorporated length 2 and 3 wires but required only slightly higher W values to route circuits than a segmentation distribution where all wires were length 1. They also investigated a new architectural parameter, *internal population*, that is relevant to wires which span more than one logic block. All wires are assumed to be able to connect to other wires and to logic blocks at their endpoints. A wire segment is *internally switch-block populated* if it can make connections from its interior to other wire segments (via switch blocks); otherwise it is switch-block depopulated. Similarly, a wire that can connect to from its middle to logic blocks (via connection blocks) is *internally connection-block populated.* Figure 2.6 illustrates the four different internal population possibilities for segments. Notice that a length 3 wire that is connection-block depopulated can connect to only 2 of the 3 logic blocks it passes. A length 3 wire that is switch-block depopulated can connect to wires in only two of the four vertical channels that pass by it. Internally depopulated wire segments are faster than populated wire segments because there are fewer switches, and hence less parasitic capacitance, connected to them. Chow et al found that the use of switch-block depopulated wires resulted in a significant increase in the required value of W to route circuits, while connection-block depopulation caused only a small (~0 - 10%) increase in W. They also found that many architectures with 0 - 60% length 2, 10 - 40% length 3, and the remainder length 1 wires required only about 10% more tracks per channel than an all length 1 architecture.

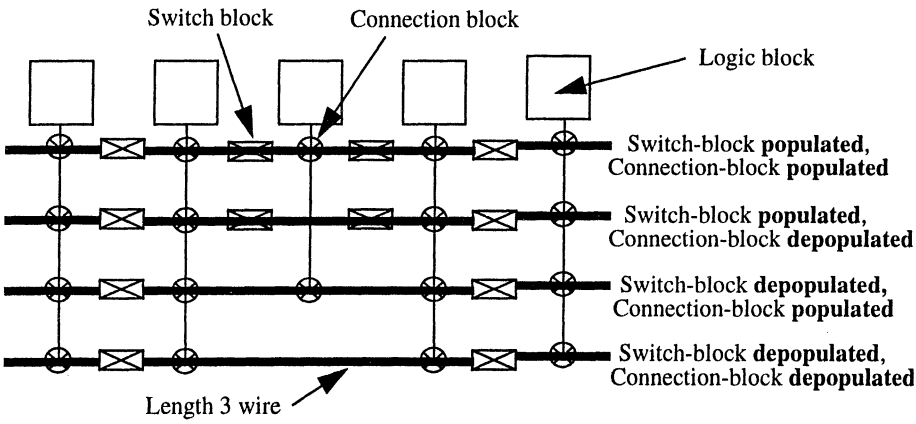


FIGURE 2.6 Internal population and depopulation of wiring segments.

In [40], Sun et al proposed a new type of FPGA routing structure that incorporated longer wires. This new structure is related to but not identical to the routing of an island-style FPGA. Its area-efficiency, as measured by the number of programmable switches in the routing, is 50% worse than that of an island-style FPGA that uses all length 1 wires. The number of active routing switches (switches that are actually turned on to connect wires) is 35% lower in this new structure, however. The authors of [40] therefore conclude that this new FPGA is faster than an island-style FPGA with all length 1 wires. This number of active routing switches delay metric is not very closely related to critical path delay, however, so there is a large margin of error in their delay results.

2.2 *CAD for FPGAs*

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

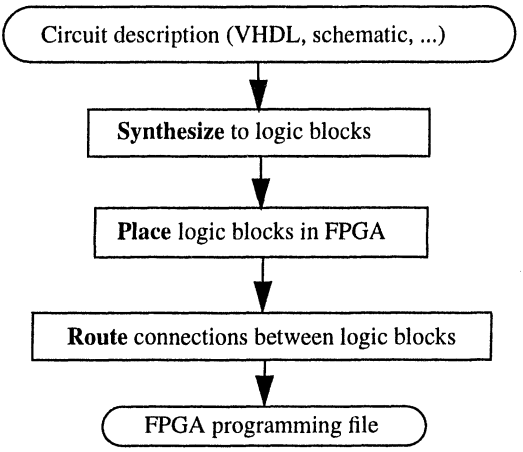


FIGURE 2.7 FPGA CAD flow.

Figure 2.7. In the following three sections we will describe the synthesis, placement and routing problems and briefly outline some of the prior work in each area. Once a circuit is placed and routed, delay models and timing analysis are used to assess its speed; we describe these problems in Sections 2.2.4 and 2.2.5.

2.2.1 Synthesis and Logic Block Packing

The first stage of synthesis converts the circuit description, which is usually in a hardware description language or schematic form, into a netlist of basic gates. Then, the logic synthesis process converts this netlist of basic gates into a netlist of FPGA logic blocks such that the number of logic blocks needed is minimized and/or circuit speed is maximized. Logic synthesis is sufficiently complex that it is usually broken into two or more subproblems [41]; in this book we will use a three-stage synthesis flow as shown in Figure 2.8.

Technology-independent logic optimization removes redundant logic and simplifies logic wherever possible [41, 42]. The optimized netlist of basic gates is then mapped to look-up tables [43, 44, 45, 46, 47]. Both of these problems have been extensively studied and good algorithms and tools capable of targeting the FPGAs we are interested in studying are publicly available, so this book does not study these phases of the synthesis process.

The third synthesis step in Figure 2.8 is necessary whenever an FPGA logic block contains more than a single LUT. Logic block packing groups several LUTs and reg-

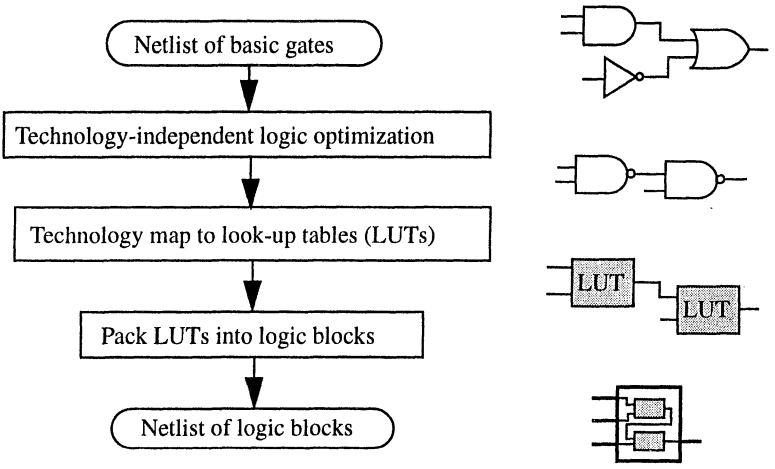


FIGURE 2.8 Details of synthesis procedure.

isters into one logic block, respecting limitations such as the number of LUTs a logic block may contain, and the number of distinct input signals and clocks a logic block may contain. The optimization goals in this phase are to pack connected LUTs together to minimize the number of signals to be routed between logic blocks, and to attempt to fill each logic block to its capacity to minimize the number of logic blocks used.

This problem is a form of clustering. Clustering and partitioning are essentially the same problem; divide a netlist into several pieces, such that certain constraints, such as maximum partition size, are respected, and some goal, such as minimizing the number of connections that cross partitions, is optimized [48]. When a circuit is to be divided into only a few pieces, the problem is called partitioning. When a circuit is to be divided into many small pieces in one step (as opposed to recursively partitioning into a few partitions in each step), the problem is usually called clustering.

Clustering has been extensively studied [48, 49, 50, 51, 52], with spectral [53, 54, 55, 56] and labelling [57, 58, 59, 60] methods being popular. Many of these methods, however, cannot easily incorporate simultaneous constraints on the maximum number of inputs, the number of clocks, and the number of LUTs and registers in a logic block, and these constraints are key in logic block packing. Spectral methods use analytic techniques to produce "natural" groupings of circuit elements, and then enforce constraints on cluster size, etc. in a post-processing step. With many cluster legality constraints, then, it becomes likely that this post-processing step will perturb

the "natural" clustering solution enough to significantly impact the result quality. It is difficult for labelling methods to simultaneously satisfy both a constraint on the number of inputs to a cluster (pin constraint) and the maximum number of items (e.g. LUTs) in a cluster (size constraint). Yang and Wong [60] present a labelling algorithm which can simultaneously satisfy both size and pin constraints, but it tends to produce large amounts of logic replication and to only partially fill clusters. Hence it requires a large number of clusters to cover a circuit.

In [61], Cong et al describe a logic block packing tool capable of targeting several different types of logic blocks. It uses a closeness metric to determine the desirability of putting two LUTs into the same logic block, and it respects any constraints restricting which LUTs can be packed together. In [61], the packing step is performed via maximum weighted matching on a graph representing the circuit. This is an O(nm) algorithm, where n is the number of LUTs, and m is the number of edges in the graph [62, 63]. For the logic clusters we study, the number of edges in this "closeness" graph, m, is $O(n^2)$, which would lead to an $O(n^3)$ algorithm.[1]

In [64], Shin and Kim describe a greedy algorithm to transform a netlist of circuit blocks into a netlist of clusters, where each cluster contains approximately the same number of circuit blocks, and the circuit blocks in each cluster tend to be highly connected. Initially each circuit block is a cluster. The "closeness" of two clusters is a function of how many nets the two clusters share and the size of the cluster that would result from merging them. More formally:

$$closeness(C, D) = \frac{NumCommonNet(C, D)}{min(NumPin(C), NumPin(D))} - \alpha \cdot \frac{ClusterSize(C, D)}{AverageClusterSize} \quad \textbf{(2.1)}$$

where α is a weighting factor controlling how balanced the cluster sizes should be. The two clusters with the largest closeness value are merged, and the closeness values of all the other clusters are updated to reflect the change. The merging process terminates when the number of clusters in the netlist falls below some user-specified value. While this algorithm was not developed for FPGAs, and does not understand the hard constraints on cluster size and number of inputs intrinsic to logic clusters, the general approach of greedily optimizing a closeness function is highly adaptable, and hence of interest.

1. If the edge weights are all integers within a limited magnitude range, algorithms with complexity somewhat worse than $O(n \cdot^5 m)$, but less than O(nm) exist [63]. With m of $O(n^2)$ this would still lead to an algorithm with complexity worse than $O(n^{2.5})$.

2.2.2 Placement

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

The three major classes of placers in use today are min-cut (partitioning-based) [65, 66, 67], analytic [68, 69, 70, 71, 72, 73, 74] which are often followed by local iterative improvement [75], and simulated annealing based placers [76, 77, 78, 79, 80, 81]. Recall that to fairly investigate architectures we must ensure that our CAD tools are attempting to use all of an FPGA's features — this means the optimization goals of our placer may change from architecture to architecture. It is much easier to add new optimization goals or constraints to a simulated annealing based placer than to a min-cut or an analytic placer, so we focus on this algorithm below.

Simulated annealing mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects [76]. Pseudo-code for a generic simulated annealing-based placer is shown in Figure 2.9. A cost function is used to evaluate the quality of any placement of logic blocks — for example, a common cost function in wirelength-driven placement is the sum over all nets of the half-perimeter

```
S = RandomPlacement ();
T = InitialTemperature ();
R_limit = InitialR_limit ();

while (ExitCriterion () == False) {        /* "Outer loop" */
    while (InnerLoopCriterion () == False) {    /* "Inner loop" */
        S_new = GenerateViaMove (S, R_limit);
        ΔC = Cost (S_new) - Cost (S);
        r = random (0,1);
        if (r < e^(-ΔC/T)) {
            S = S_new;
        }
    }          /* End "inner loop" */
    T = UpdateTemp ();
    R_limit = UpdateR_limit ();
}   /* End "outer loop" */
```

FIGURE 2.9 Pseudo-code of a generic simulated annealing-based placer.

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

The rate at which temperature is decreased, the exit criterion for terminating the anneal, the number of moves attempted at each temperature (InnerLoopCriterion), and the method by which potential moves are generated are defined by the annealing schedule. A good annealing schedule is crucial to obtain good results in a reasonable amount of CPU time. Many proposed annealing schedules are "fixed" schedules that have no ability to adapt to different problems. Such schedules can work well within the narrow application range for which they were developed, but their lack of adaptability means they are not very general [82]. Accordingly, we discuss below only "adaptive" annealing schedules that determine an annealing schedule based on statistics computed during the anneal itself.

Huang, Romeo, and Sangiovanni-Vincentelli [82], developed an annealing schedule and applied it to a standard-cell placement problem. Huang's schedule performs a set of random moves on the initial placement, and sets the initial temperature (InitialT) to 20σ, where σ is the standard deviation of the cost over these moves. New temperatures (UpdateT) are computed via:

$$T_{new} = T_{old} \cdot e^{\frac{-\lambda T_{old}}{\sigma}} \tag{2.2}$$

where λ is typically set to 0.7 and σ is the standard deviation of the moves accepted at T_{old}. The InnerLoopCriterion of Figure 2.9 is fairly complex for this schedule; it involves monitoring the fraction of new states generated that have their costs within a certain range of the average cost at that temperature, and there are several special cases and fall-back cases defined. Finally, the anneal terminates (ExitCriterion), when the difference between the maximum and minimum costs accepted at that tem-

perature equals the maximum cost change caused by any one move at that temperature.

The schedule of Lam and Delosme [83], employs feedback control to set the annealing schedule. It monitors the standard deviation of the cost, the average cost, and the fraction of proposed moves that were accepted, α, over the past τ moves. Typically τ is 100. These values are inputs to a sophisticated feedback system that determines a new temperature. In this schedule, a new temperature is computed every move — that is, the "inner loop" in Figure 2.9 executes only one iteration each time control reaches it. The anneal terminates when there has been no change in the average cost for the last $k \cdot \tau$ moves, where k is typically 5. This annealing schedule also employs a range limiter to control the move generation process. The R_{limit} parameter in Figure 2.9 controls how close together blocks must be to be considered for swapping. Initially, R_{limit} is fairly large, and swaps of blocks far apart on a chip are likely. Throughout the anneal, R_{limit} is adjusted to try to keep the fraction of moves accepted at any temperature close to 0.44. If the fraction of moves accepted, α, is less than 0.44, R_{limit} is reduced, while if α is greater than 0.44, R_{limit} is increased.

One disadvantage of the Lam schedule is its complexity. Fortunately, Swartz and Sechen have developed an annealing schedule incorporating some of the key ideas of the Lam schedule, and which achieves equivalent quality, but is much less complex [84]. In this schedule, the number of moves attempted in the "inner loop" is $10 \cdot N_{blocks}^{1.33}$. The range limiter, R_{limit} is updated according to a fixed (hard-coded) schedule; the exact form of R_{limit} is not specified in [84], but it likely initially spans the entire chip, and gradually shrinks to a small region. The "outer loop" is executed 150 times, and then the anneal terminates. The temperature is controlled by the fraction of moves accepted:

$$T_{new} = \left[1 - \frac{\alpha - 0.44}{40}\right] \cdot T_{old}, \tag{2.3}$$

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

Since the amount of routing in FPGAs is limited and set by the manufacturer when the FPGA is fabricated, some FPGA placement tools attempt to optimize not just the wirelength of a placement, but also its routability. In [85], Ebeling et al describe a simulated annealing based placer that targets the Triptych FPGA developed at the University of Washington. Its cost function incorporates not only a bounding-box wirelength term, but also a "porosity" term that monitors the fraction of logic blocks

in a local area that are being used. Since the Triptych FPGA is usually unroutable in regions where the logic blocks are completely used, maintaining a porosity of 50% or so across the FPGA is essential. In [86, 87], Nag and Rutenbar describe a simulated annealing based tool that performs placement and routing simultaneously in one combined step. After any swap of blocks, all the affected nets are re-routed via a maze router. To keep the CPU time reasonable, this maze router is constrained to look at only a small number of potential routes when the temperature is high; at lower temperatures, the router is allowed to spend more time looking for routes. If no suitable route is found among the allowed candidates, the net is marked as currently unroutable, and the placement cost is increased. The result quality of this tool is high, but the CPU time required is very large — a circuit containing only 461 Xilinx 4000 logic blocks required 11 hours of CPU time to place and route, and the complexity of this algorithm appears to be approximately $O(n^3)$, where n is the number of logic blocks in a circuit. Alexander et al have created a partitioning-based algorithm, FPR [88], that performs placement and global routing simultaneously, again in an attempt to maximize circuit routability.

Some work has also been done in routability-driven standard cell placement that is applicable to FPGAs. Cheng [89] describes a simulated annealing placer that divides a chip into many subregions and estimates the demand for wiring in each region. This demand is compared to the supply of routing in each region, and when the expected demand outstrips the routing supply in some regions the placement is penalized by having its cost increased. Dividing the chip into subregions and estimating wiring demand makes localized congestion visible, and merely estimating the wiring demand in a region is faster than actually routing each placement proposed during the anneal.

2.2.3 Routing

Once locations for all the logic blocks in a circuit have been chosen, a router determines which programmable switches should be turned on to connect all the logic block input and output pins required by the circuit. In FPGA routing, one usually represents the routing architecture of the FPGA as a directed graph [86, 85]. Each wire and each logic block pin becomes a node in this *routing-resource graph* and potential connections become edges. Some prior research has represented FPGAs as undirected graphs [90], but a directed graph representation is needed if directional switches, like tri-state buffers and multiplexers, are to be modelled correctly.

Routing a connection corresponds to finding a path in this routing-resource graph between the nodes representing the logic block pins to be connected. To avoid using up too many of the limited number of wires in an FPGA, one wants this path to be as short as possible. As well, it is important that the routing for one net not use up rout-

ing resources another net needs, so most FPGA routers have some kind of congestion avoidance scheme to resolve contention for routing resources. An additional optimization goal is to make nets on or near the critical path fast by routing them using short paths and fast routing resources. Routers that attempt to optimize timing in this way are called timing-driven, whereas delay-oblivious routers are purely routability-driven. Since most of the delay in FPGAs is due to the programmable routing, timing-driven routing is crucial to obtain good circuit speeds.

FPGA routers can be divided into two groups. Combined global-detailed routers [85, 90, 91, 92, 93, 94, 95, 96] determine a complete routing path in one step, while two-step routing algorithms first perform global routing [97, 98] to determine which logic block pins and channel segments each net will use, and then perform detailed routing [99, 100, 101, 34] to determine the wire(s) each net will use within each of the specified channel segments. A channel segment is the length of routing channel that spans one logic block — a channel that spans M logic blocks contains M channel segments. The task of an FPGA detailed router is often difficult or impossible because FPGA routing has limited flexibility and the detailed router is highly constrained by the decisions the global router made about which channel segments each net must use. Combined global-detailed routers have the potential to more fully optimize the routing, since they are free of such constraints.

Of the routers listed above, only those of [85, 95, 94] use timing analysis (see Section 2.2.5) to determine which nets are on, or "almost" on, the critical path so they can be given priority for fast routing paths. Since much of our work is concerned with timing-driven routing, we will focus on these three algorithms. At their core, all these routers use variants of maze routers [102] to connect the terminals of each net. A maze router essentially consists of running Dijkstra's algorithm [103] to find the shortest (lowest total cost) path between a net source node and a net sink node in a routing-resource graph. All of these algorithms perform multiple routing iterations in which some or all of the nets are ripped-up and rerouted by different paths to resolve competition for routing resources or improve circuit speed. Both [94] and [95] use timing analysis only to help identify good nets to rip-up and re-route — nets which are likely to lead to a circuit speed-up if they can be rerouted using a faster path. Ripping-up and re-routing these nets only changes the net ordering, however; they are all routed by the same maze routing algorithm, regardless of how timing-critical they are. The PathFinder negotiated congestion-delay algorithm [85] uses a more sophisticated technique in which the congestion-delay trade-off of each connection is controlled by how timing critical it is. In other words, a timing-critical connection will be routed by a minimum delay path even if it is congested, while a non-timing-critical net will take a longer, uncongested path. This algorithm produces excellent results and incorporates several important ideas, so we describe it in detail below.

Pathfinder repeatedly rips-up and re-routes every net in the circuit until all congestion is resolved — this idea is due to Nair [104]. Ripping-up and re-routing every net in the circuit once is called a *routing iteration.* During the first routing iteration, every connection is routed for minimum delay, *even if this leads to congestion,* or overuse, of some routing resources. A circuit routing in which some routing resource is over-used, such as a wire being used by two different nets, is not a legal routing. Consequently, when overuse exists at the end of a routing iteration, another routing iteration (or more) must be performed to resolve this congestion. After each routing iteration the cost of overusing a routing resource is increased, so that the probability of resolving all congestion increases. At the end of each routing iteration we have a complete, but potentially somewhat illegal, routing. We can therefore determine the net delays from this routing, and perform a full timing analysis to compute the slack (see Section 2.2.5) of each source-sink connection. These slack values are used in the next routing iteration to control how much attention each connection pays to delay, and how much is paid to congestion-avoidance. Pseudo-code for the algorithm is given in Figure 2.10.

The criticality of the connection from the source of net i to one of its sinks, j, is:

$$Crit(i, j) = 1 - \frac{slack(i, j)}{D_{max}} \tag{2.4}$$

where D_{max} is the delay of the circuit critical path, and slack(i,j) is the amount of delay that could be added to this connection before it affected the circuit's critical path. Crit(i,j) is therefore between 0 and 1.

The cost of using a routing resource node, n, as part of connection (i,j) is

$$Cost(n) = Crit(i, j) \cdot delay(n) + [1 - Crit(i, j)] \cdot [b(n) + h(n)] \cdot p(n). \tag{2.5}$$

The first term in (2.5) is the delay sensitive term — the criticality of the connection times the intrinsic delay of the node. The second term is the congestion sensitive term. b(n) is the base cost of node n, and is set equal to delay(n) in [85]. h(n) is the historical congestion of node n; it is increased after every routing iteration in which node n is overused and gives the router "congestion memory." p(n) is the present congestion cost of node n; it is 1 if using this node to route the current connection will not cause any overuse, and increases with the amount of overuse of the node. p(n) is also a function of the number of routing iterations that have been performed. In early iterations, p(n) grows slowly with the current overuse of node n; in later iterations, p(n) goes up very rapidly with overuse of node n.

```
Let: RT(i) be the set of nodes, n, in the current routing of net(i).

Crit(i,j) = 1 for all nets i and sinks j;
while (overused resources exist) {    /* Illegal routing? */

    for (each net, i) {
        rip-up routing tree RT(i) and update affected p(n) values;
        RT(i) = NetSource(i);

        for (each sink, j, of net(i) in decreasing crit(i,j) order) {
            PriorityQueue = RT(i) at PathCost(n) = crit(i,j)·delay(n) for
                            each node n in RT(i);
            while (sink(i,j) not found) {
                Remove lowest cost node, m, from PriorityQueue;
                for (all fanout nodes n of node m) {
                    Add n to PriorityQueue at PathCost(n) =
                        Cost(n) + PathCost(m);
                }
            }

            for (all nodes, n, in path from RT(i) to sink(i,j)) {  /* Backtrace */
                Update p(n);
                Add n to RT(i);
            }
        }
    }

    Update h(n) for all n;
    Perform timing analysis and update Crit(i,j) for all nets i and sinks j;
} /* End of one routing iteration */
```

FIGURE 2.10 Pseudo-code of the Pathfinder routing algorithm.

The excellent performance of Pathfinder is due to two innovations: allowing overuse of routing resources, and using the cost function of (2.5) to allow congestion to gradually be resolved, and timing to be directly optimized. By slowly increasing the cost of congestion, via p(n) and h(n), as more routing iterations are performed, connections that are on or near the critical path tend to take the fastest paths and stay there, while less timing-critical connections are gradually forced off any overused nodes onto slower paths.

Notice that the router of Figure 2.10 uses a breadth-first search through the routing resource graph to connect net terminals. The creators of Pathfinder [85] also describe

an enhancement to the basic algorithm that uses an A*, or directed, search [105] to speed execution. In Figure 2.10 a node is added to the priority queue with a PathCost equal to the sum of all the node costs along the path up to and including it. To convert this to an A* search, one simply adds this term to a lower bound on the sum of the node costs needed to reach the target sink from the current node; the result is then used as the sort value when the node is added to the priority queue.

Finally, Ebeling et al [85] also describe a purely routability-driven variant of the Pathfinder router, which they call the Pathfinder negotiated congestion algorithm. This algorithm simply sets the criticality of every net, Crit(i,j), to 0 so that the cost of a node is given solely by the congestion-sensitive term in (2.5). As well, this router connects the current routing tree to the first net sink found, rather than a pre-determined target sink, during maze expansion.

The router used in the simultaneous placement and routing tool developed by Nag and Rutenbar [86] has one unique feature of interest. This router is again maze-router based, and it is targeted at Xilinx XC4000 series FPGAs. When routing a multi-terminal net, a maze router will typically route to the closest sink, and then use this partial routing as the source (start point) when it attempts to route to the next closest sink, and so on. This can cause problems when routing high-fanout nets on FPGAs that contain some long wire segments. High-fanout nets typically span most of the FPGA, but the distance from one sink to the next closest sink is usually only a few logic blocks. Consequently, a traditional maze router, that looks only at how to connect the partial routing to the next closest sink, will tend to use short wires to build the routing trees for high-fanout nets, even though using longer wires would be more efficient and result in faster nets. Nag solved this problem by dramatically reducing the cost of using a long line when the net bounding box spanned most of the FPGA. The router then saw the cost of using a long wire to be less than that of even one short wire, so it would use long lines to connect even to nearby sinks and construct a routing tree of long wires for high-fanout nets. This idea of varying the cost of resources depending on the type of connection being routed, or *dynamic weighting*, is a powerful one; in some sense Pathfinder's varying the weighting of delay and congestion according to the criticality of the connection being routed is another example of this idea.

One shortcoming of current non-commercial timing-driven FPGA routers is that they are designed to optimize only the linear delay model, in which every routing resource has a fixed delay.[1] Most FPGAs contain at least some pass transistor switches in their routing, so the delay of a routing resource actually depends on the topology of the net

1. SEGA can use a more advanced delay model, but [36] showed that SEGA achieved better speed with a cost function that emphasized recombining the two-point nets passed to it by the global router into multi-terminal nets than it did with this delay-based cost function.

using it. As well, since the FPGAs we study in Chapter 7 include buffers in the routing, the router must understand when to use a buffered switch and when to use a pass transistor. Unfortunately, no non-commercial FPGA router is "buffer-aware." The Xilinx commercial router, described by Trimberger in [106], is buffer-aware and uses the Penfield-Rubinstein [107] delay model during routing. It is likely that the routers of other FPGA companies whose products contain a mix of pass transistors and buffers are also buffer-aware, but to our knowledge the algorithms used by these other companies have never been made public.

Considerable work has been done in the standard cell and MPGA routing areas on routing under more accurate delay models [108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121] and buffer insertion [122, 123, 124, 125, 126]. However, much of this work is not easily applicable to the FPGA routing problem because:

1. MPGA and standard cell routers can choose exactly where they want wires, how long these wires should be, and where buffers should be placed. Since all FPGA routing resources are prefabricated, however, FPGA routers are constrained to choose from a set of prefabricated wires and switches. The flexibility of FPGA routing is low enough that if the router decides to connect two wires together it usually has no further choice about whether to insert a buffer or not at the join. There is usually only one switch that can connect these two wires, and whether it is a buffer or a pass transistor was determined when the FPGA was fabricated. In graph theoretic terms, FPGA routing consists of finding Steiner trees embedded in a graph, while MPGA routing consists of finding Steiner trees embedded in the Manhattan plane [90].
2. The complexity of many MPGA and standard cell routing algorithms is quite high — $O(k^3)$ to $O(k^4)$ is common, where k is the number of terminals on a net. Since we will be routing circuits with thousands of nets, and a few of these nets have hundreds of terminals, we must use algorithms with reasonably low (ideally linear) complexity [127].

Nonetheless, some ideas from the MPGA world are relevant to our work. For example, in the absence of congestion, PathFinder attempts to greedily optimize a combination of wirelength and (linear delay model) delay to a net's sinks, which is similar to the approach of Alpert et al in [110]. In a similar vein, [109] and [111] attempt to greedily optimize a combination of the Elmore delay and wirelength; it should be possible to adapt these algorithms to FPGAs. The time complexity of these algorithms is quite high, however. The algorithm of [111] is $O(k^4)$, and while the authors do not give the complexity of their algorithm in [109], it appears to be at least $O(k^3)$. Since adapting these algorithms to FPGAs involves routing within a large routing-resource graph, their complexity may increase even further and make them impractical.

2.2.4 Delay Modelling

One must compute the delay of a route from a net source to any of its sinks in order to:

1. Determine the speed of a circuit after it has been routed, and
2. Determine the delay of different net topologies during routing.

Ideally, one would use a circuit simulator such as SPICE to obtain highly accurate delay estimates, but the CPU time required to run SPICE on the thousands of nets in a typical circuit is prohibitive. Instead, previous researchers have modelled pass transistors as linear resistors and wires as an RC pi-network, so that a net's routing may be modelled as an RC-tree [22]. In [22], the Penfield-Rubinstein delay model [107] was then used to determine an upper and lower bound on the delay of the RC-tree to each of the net sinks. An alternative to the Penfield-Rubinstein model is the Elmore delay [128], which is the most widely used delay estimate in routing research [108]. The Elmore delay was originally defined only for RC-trees, but it has been combined with a common model of buffer delay to allow its use with circuits that contain buffers, as well as resistors and capacitors [112]. Each buffer is modelled by a constant delay and a resistor. The constant delay accounts for the intrinsic delay of the buffer, while the resistance accounts for the load-dependence of the buffer delay. Figure 2.11 shows the RC-model for each of the three elements of FPGA routing. Note that pass transistors and buffers attached to a wire add parasitic capacitance regardless of whether they are on or off.[1]

The Elmore delay of a source-sink path is then [112]:

$$\sum_{i \in \text{ Source-sink path}} R_i \cdot C(subtree_i) + T_{d,i} \qquad \textbf{(2.6)}$$

where $T_{d,i}$ is the intrinsic delay of a buffer if element i is a buffer, and 0 otherwise. R_i is the equivalent resistance of element i (R_{wire}, R_{buf}, or R_{pass}). In (2.6), $C(subtree_i)$ is the total capacitance of the dc-connected subtree rooted at element i — that is the total downstream capacitance which is not isolated by buffers.

1. Notice that we model the capacitance of both "on" and "off" pass transistors as being purely due to diffusion capacitance. In fact the capacitance of an "on" pass transistor is larger than that of an "off" pass transistor, since the channel created when a transistor is "on" has capacitance to the gate and to the substrate. Since relatively few pass transistors are "on" at any time and most of the capacitance in an FPGA is metal capacitance, the error in total capacitance caused by this approximation is small (~1% to 2%).

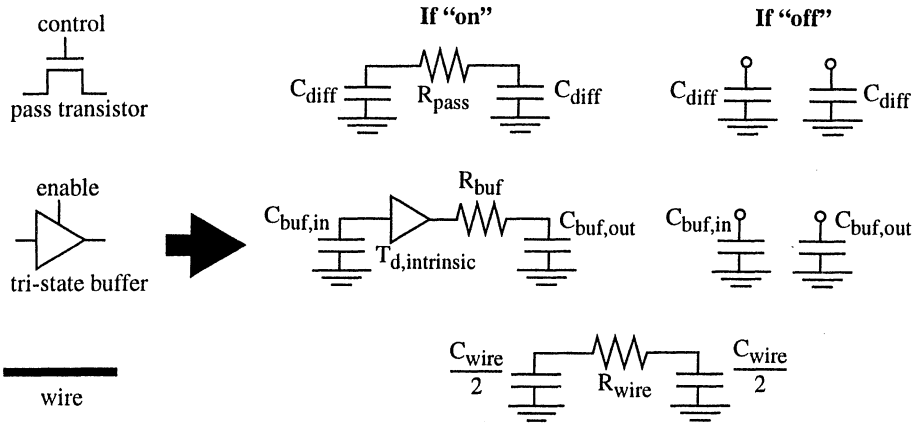


FIGURE 2.11 Equivalent circuits for FPGA routing elements.

The Elmore delay of an RC-tree can be computed in linear time, and we will show in Section 7.2.3 that its accuracy is quite good. As well, [129, 130] showed that even when the Elmore delay is somewhat inaccurate it still tends to have high *fidelity*. That is, it still tends to rank the speed of different routing topologies correctly. This is a useful feature, as it means that the relative comparisons of topologies made by a router using it will correctly rank the alternatives, and that comparisons of FPGA architecture speed made via Elmore delay will correctly rank the different architectures.

2.2.5 Timing Analysis

Timing analysis [131] is used for two basic purposes:

1. To determine the speed of circuits which have been completely placed and routed, and
2. To estimate the *slack* of each source-sink connection during routing (or other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

Normally one performs timing analysis on a directed graph representing the circuit structure [131]. Nodes represent the input pins and output pins of basic circuit elements, such as registers and LUTs, and edges are added between the inputs of combinational logic blocks (e.g. LUTs) and their outputs and between pins which the circuit netlist specifies are connected. Each edge is annotated with the delay required to pass

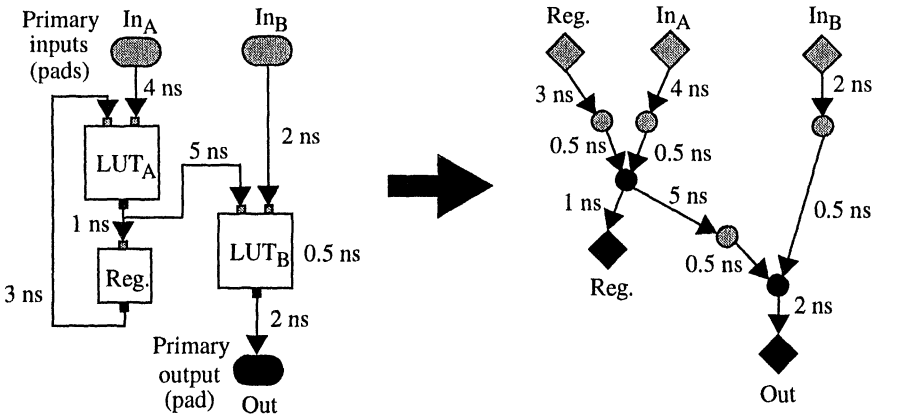


FIGURE 2.12 The timing graph for a simple circuit.

through the circuit element or routing. Register input pins are not joined to register output pins — register outputs have no edges incident to them, and register inputs have no edges leaving them. Similarly, primary inputs (input pads) have no incident edges, and primary outputs (output pads) have no exiting edges. Figure 2.12 shows a simple circuit implemented via 2-input LUTs and registers, and the corresponding timing graph.

One can determine the minimum required clock period with O(n) computation for a timing graph with n nodes via a breadth-first traversal of the timing graph. The traversal begins at nodes with no incident edges (primary inputs and register outputs) and labels each with a signal *arrival time*, $T_{arrival}$, of 0. Each node which has incident edges only from labelled nodes is then labelled with its arrival time according to

$$T_{arrival}(i) = Max_{\forall j \in fanin(i)}\{T_{arrival}(j) + delay(j, i)\} \quad \textbf{(2.7)}$$

where node i is the node being labelled, and delay (j,i) is the delay value marked on the edge joining node j to node i. This procedure continues until every node in the graph has been labelled. The node with the largest arrival time, which will always be a primary output or a register input, then defines the maximum delay, D_{max}, (= minimum clock period) through the circuit. In Figure 2.12, for example, the arrival time at node *Out* is 12 ns, and this is the largest arrival time, and hence the maximum delay, in the circuit.

The above procedure is sufficient to determine the speed of a circuit, but during timing-driven placement or routing we also want a metric indicating how important each source-sink connection is in terms of its effect on the circuit's delay. The *slack* of a

connection is defined as the amount of delay which could be added to this connection without increasing the minimum cycle time of the circuit (if the delays of all other connections remained constant). To compute the slack, we perform a second breadth-first traversal of the timing graph. This time each node with no outward edges has its *required time*, $T_{required}$, set to D_{max} and a backward breadth-first traversal of the graph is performed to label the remaining nodes. The required time of any node with fanout (outward edges) is

$$T_{required}(i) = Min_{\forall j \in fanout(i)}\{T_{required}(j) - delay(i, j)\} \quad \textbf{(2.8)}$$

The slack of the connection from node i to node j is then

$$slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j) \quad \textbf{(2.9)}$$

Connections with a slack of zero are on the circuit critical path — any increase in the delay of such a connection will lead to a corresponding increase in the circuit's delay. Connections with large slacks, on the other hand, could be routed via significantly slower paths without affecting a circuit's delay. In Figure 2.12, for example, the slack of the connection between In_A and LUT_A is 0, while the slack between In_B and LUT_B is 7.5 ns.

Slack allocation [132, 133, 95] is used to determine a set of upper bounds on connection delays from a set of slacks, such that the circuit speed is guaranteed not to decrease as long as each connection is routed with delay less than its upper bound. Some timing-driven placement and routing tools require such upper bounds to guide them, while other tools work with the net slack values directly.

2.3 Summary

In this chapter we first reviewed the basics of FPGA architecture. We then detailed the terminology used to describe FPGA logic blocks and FPGA routing architecture, and presented some of the important prior research into both areas. In the second half of the chapter, we described the CAD flows used with FPGAs. We focused particularly on algorithms for logic block packing, placement and routing, and described some of the strengths and weaknesses of prior approaches. Finally, we described the delay modelling and timing analysis procedures used both to determine how fast a circuit will operate, and to help guide timing-driven CAD tools.

The next chapter describes the CAD tools we developed to perform logic block packing and placement within an FPGA. Chapter 4 describes the new FPGA router we have developed. Our placement tool is simulated annealing based, and our router makes use of many of the ideas of the Pathfinder algorithm. Since both simulated annealing and Pathfinder were described in this chapter, Chapters 3 and 4 will focus on the new ideas and enhancements in these tools.

Chapters 5, 6 and 7 then apply these CAD tools to investigate three different aspects of FPGA architecture: global routing architecture, logic block clusters, and detailed routing architecture.

CHAPTER 3 CAD Tools: Packing and Placement

This chapter describes the logic block packing tool we developed to target cluster-based logic blocks and the new and novel parts of our FPGA placement tool.

A key goal for these tools, and of the router discussed in Chapter 4, is to be able to target a wide variety of FPGA architectures in order to enable architectural exploration. Furthermore, these tools should be responsive to the variation of the FPGA architectural parameters — they should directly optimize for the different features of the FPGAs we investigate. Finally, we require the result quality of the tools to be high, since low-quality tools can lead to inaccurate architectural conclusions.

3.1 Logic Block Packing

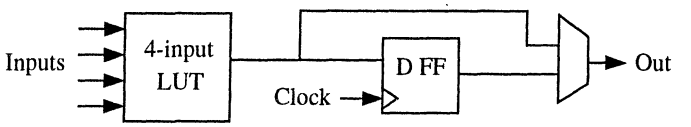
Recall that the logic block packing problem (introduced in Section 2.2.1) consists of packing the LUTs and registers generated by a LUT-based technology mapping program into larger logic blocks. The next section describes the exact structure of the "cluster-based" logic blocks we wish to target. Section 3.1.2 details the first algorithm we developed to pack LUTs and registers into these logic blocks, while Section 3.1.3 describes enhancements to this basic algorithm to make it timing-driven and improve its performance. While Altera has an in-house tool targeting cluster-based logic blocks and Xilinx has an in-house tool targeting the "cluster-like" logic blocks of the 5200 FPGA, we know of no published work describing algorithms that target cluster-based logic blocks.

3.1.1 Cluster-Based Logic Blocks

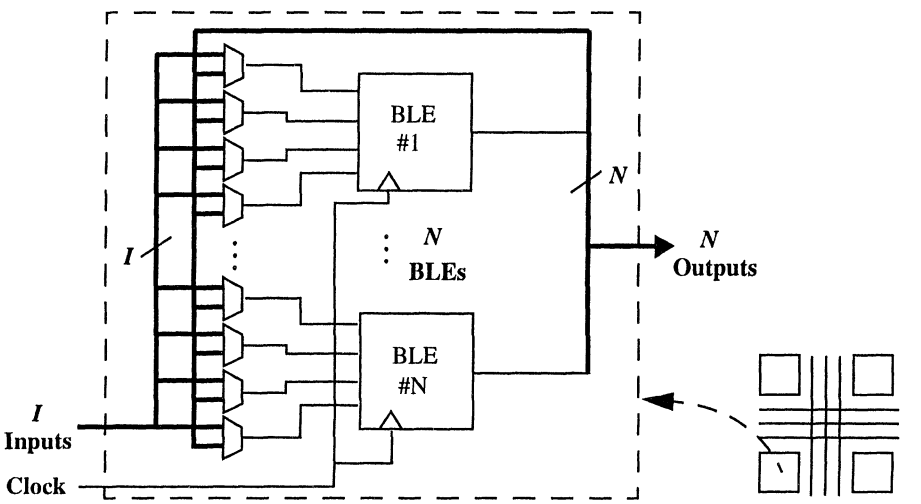
The structure of a cluster-based logic block, which we also call a *logic cluster*, is shown in Figure 3.1 [8, 9]. The cluster-based logic blocks we investigate have a two-level hierarchy: the overall block is a collection of *basic logic elements* (BLEs). As shown in Figure 3.1a, our basic logic element consists of a LUT and a register, and the BLE output can be either the registered or unregistered version of the LUT output. This is how many commercial FPGAs combine a LUT and a register to create a structure capable of implementing either combinational or sequential logic. The complete logic block contains N BLEs and local routing to interconnect them, as shown in Figure 3.1b.

A logic cluster is described with the following four parameters:

1. the size of (number of inputs to) a LUT (K),
2. the number of BLEs in a cluster (N),



(a) Basic logic element (BLE)



(b) Logic cluster

FIGURE 3.1 Structure of (a) basic logic element (BLE) and (b) logic cluster.

3. the number of inputs to the cluster for use as inputs by the LUTs (I), and
4. the number of clock inputs to a cluster (for use by the registers), M_{clk}.

In this book, we focus on logic clusters in which the LUT size, K, is 4 and the number of clock pins on a cluster, M_{clk}, is 1 — this is the case shown in Figure 3.1. Nonetheless, out logic block packing tools are capable of targeting logic clusters with different values of K and M_{clk} [134].

As Figure 3.1 shows, not all $K{\cdot}N$ look-up table inputs are accessible from outside the logic cluster. Instead, only I external inputs are provided to the logic cluster — multiplexers allow arbitrary connections of these cluster inputs to the BLE inputs. The same multiplexers also connect to each of the N BLE outputs, allowing the output of any BLE within the cluster to be connected to any of the BLE inputs. All N outputs of the logic cluster can be connected to the FPGA routing for use by other logic clusters.

Notice that each of the BLE inputs can be connected to any of the cluster inputs or any of the BLE outputs. Logic clusters are therefore *fully-connected.* This is a useful feature, as it simplifies our CAD tools considerably. For example, determining if all the appropriate signals can be connected to the BLE inputs reduces to counting the number of different input signals needed by the BLEs in a cluster that are not generated within the cluster, and comparing it to the number of cluster inputs, I. The logic cluster used in the Altera 8K and 10K FPGAs is fully connected [7], and the logic block used in the Xilinx XC5200 FPGA is nearly fully connected [11].

3.1.2 Basic Logic Block Packing Algorithm: VPack

In this section we describe our basic logic block packing tool, VPack. VPack takes as input a netlist of LUTs and registers, and outputs a netlist of logic clusters. The parameters describing the exact logic cluster to be targeted (N, I, K, and M_{clk}) are all specified on the command line; VPack can target a logic cluster corresponding to any combination of these parameters. VPack groups the LUTs and registers into logic clusters in two stages: first, it packs LUTs and registers together into BLEs, and then it packs multiple BLEs into logic clusters. The entire algorithm is outlined via pseudo-code in Figure 3.2, and is described in detail below.

The first stage of the algorithm uses a simple and optimal pattern matching algorithm to pack a register and a LUT together into one BLE whenever possible. Figure 3.3 shows that when the output of a LUT fans out only to a single register, both the LUT and register can be packed into a single BLE. Any other interconnection pattern requires that a LUT and register be assigned to two separate BLEs.

```
Let: UnclusteredBLEs be the set of BLEs not contained in any cluster
     C be the set of BLEs contained in the current cluster
     LogicClusters be the set of clusters (where each cluster is a set of BLEs)

UnclusteredBLEs = PatternMatchToBLEs (LUTs, Registers);
LogicClusters = NULL;

while (UnclusteredBLEs != NULL) {   /* More BLEs to cluster */
    C = GetBLEwithMostUsedInputs (UnclusteredBLEs);
    while (|C| < N) {   /* Cluster is not full */
        BestBLE = MaxAttractionLegalBLE (C, UnclusteredBLEs);
        if (BestBLE == NULL)   /* No BLE can be added to cluster */
            break;
        UnclusteredBLEs = UnclusteredBLEs - BestBLE;
        C = C ∪ BestBLE;
    }

    if (|C| < N) {   /* Cluster not full — try hill-climbing */
        while (|C| < N) {
            BestBLE = MinClusterInputIncreaseBLE (C, UnclusteredBLEs);
            C = C ∪ BestBLE
            UnclusteredBLEs = UnclusteredBLEs - BestBLE;
        }
        if (ClusterIsIllegal (C)
            RestoreToLastLegalState (C, UnclusteredBLEs);
    }

    LogicClusters = LogicClusters ∪ C;
}
```

FIGURE 3.2 Pseudo-code for VPack.

The second, and more complex, stage of VPack packs these BLEs into logic clusters. The optimization goals are twofold. First, we want to fill each logic cluster to its capacity, N, in order to minimize the number of logic blocks needed. Second, we want to minimize the number of inputs to each cluster in order to reduce the number of connections to be routed between clusters, and hence enhance routability. To be packed into a single logic cluster, a group of BLEs must satisfy three conditions:

1. The number of BLEs must be less than the cluster size, N;
2. The number of distinct signals generated outside the cluster and used as inputs to the cluster BLEs must be less than or equal to I; and
3. The number of different clocks used by the BLEs must be less than M_{clk}.

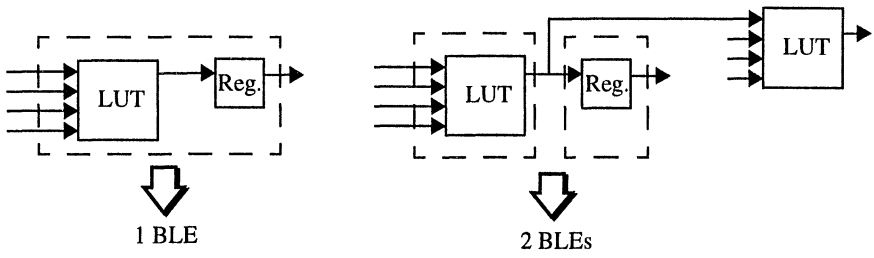


FIGURE 3.3 Packing LUTs and registers into BLEs.

VPack constructs each cluster sequentially. The algorithm has two phases: an initial phase that greedily packs BLEs into a cluster, and a hill-climbing phase that is invoked only if the greedy phase is unable to completely fill a cluster. The greedy phase begins by choosing a seed BLE for the current cluster. We have found that the best way to choose this seed is to select the unclustered BLE with the most used inputs, as such BLEs use the most cluster inputs, which are a scarce resource. Next, VPack selects the BLE with the highest *attraction* to the current cluster which could legally be added to the current cluster, and adds it to the cluster. The attraction between a BLE, B, and the current cluster is the number of inputs and outputs they have in common:

$$Attraction(B) = |Nets(B) \cap Nets(C)| \tag{3.1}$$

where C denotes the current cluster. This attraction function tends to group related BLEs into clusters, and minimize the number of distinct inputs to cluster. A BLE with a high attraction to the current cluster will not be added to the cluster if it would result in an illegal cluster — a cluster that needs more than I inputs, or more than M_{clk} different clocks.

This procedure of greedily selecting a BLE to add to the current cluster continues until either the cluster is full or adding any unclustered BLE would make the current cluster illegal. If the cluster is full, we select a new seed BLE and begin packing BLEs into a new cluster. If however, the cluster occupancy is less than N and we cannot add any BLEs because of a lack of cluster inputs, a second, hill-climbing, phase of VPack is invoked.

Since we know that any clusters that reach this second phase will be difficult to pack to capacity, VPack now selects BLEs to add to the cluster in order to minimize the increase in the number of cluster inputs required. The number of additional cluster inputs required when a BLE, B, is added to a logic cluster is simply

$$\Delta_{\text{cluster inputs}}(B) = |Fanin(B)| - |Nets(B) \cap Nets(C)| \quad \textbf{(3.2)}$$

In this hill-climbing phase, VPack also allows BLEs to be added to a cluster even if it results in a cluster that is infeasible because it needs more than I inputs (but it does not allow the number of clocks to exceed M_{clk}). Note that adding a BLE to a cluster in which all of its inputs are already present, and in which the output of the BLE is used by some other BLE already in the cluster *decreases* the number of distinct inputs to the cluster by one. Figure 3.4 shows this situation. This is the key to the hill-climbing phase; while adding one BLE to a cluster makes it infeasible, it may become feasible again when additional BLEs are added. The hill-climbing phase terminates when the cluster is full; if the cluster is still infeasible VPack backs up to the last point at which the cluster was feasible. VPack then selects a seed BLE for the next cluster and invokes the first phase again, as before.

The hill-climbing phase leads to only a small improvement in the number of BLEs packed into each cluster, which we call logic utilization, vs. using the greedy phase alone. Many circuits show no improvement, while others show only 1 - 2% gains in logic utilization. The overall average logic utilization improvement is under 1%. A likely reason for this small improvement is that the greedy first phase of VPack works very well for clustering problems of this type. It is possible, however, that the small improvement is because of the limited type of hill-climbing we are allowing. Logic replication moves could be added to the hill-climbing phase in a very natural manner, and this would likely improve the utility of hill-climbing.

This packing algorithm is very efficient. The largest benchmark circuit used in this book contains 8381 four-input LUTs and 33 registers — packing these LUTs and registers into logic clusters requires 3 seconds of CPU time on a 300 MHz UltraSparc workstation. The complexity of the algorithm is $O(k_{max} \cdot K \cdot n)$, where k_{max} is the maximum number of terminals on a net, K is the number of inputs to each LUT, and n is the number of LUTs plus the number of registers in the circuit.

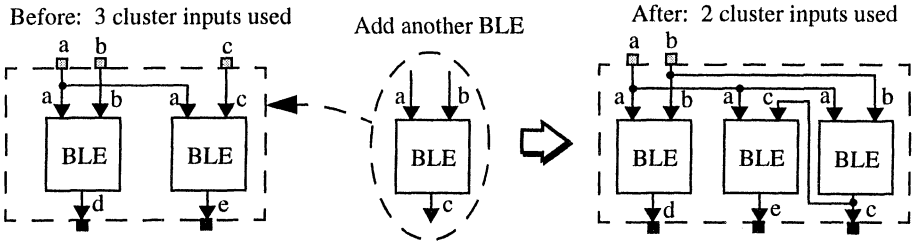


FIGURE 3.4 Adding a BLE to a cluster can decrease the number of used cluster inputs.

3.1.3 Timing-Driven Logic Block Packing: T-VPack

Our timing-driven logic block packing algorithm, T-VPack [10, 19], attempts not only to pack each logic block to capacity, but also to minimize the number of inter-cluster (between cluster) connections on the critical path. The local routing within clusters is faster than the general-purpose routing between logic clusters, so reducing the number of inter-cluster connections on the critical path reduces circuit delay. The basic operation of the algorithm is the same as that of the VPack algorithm described in the previous section: T-Vpack first chooses an unclustered BLE as the seed of a new cluster, and then sequentially adds unclustered BLEs with the greatest attraction to the current cluster until the cluster is full. The T-VPack algorithm differs from VPack in how it chooses the seed BLE for a new cluster and in the attraction function it uses.

Cluster Seed and Attraction Function

In order to minimize the number of inter-cluster connections on the critical path, T-VPack first needs to determine which connections are on the critical path. Accordingly, T-VPack performs timing analysis (described in Section 2.2.5) to determine the slack of each connection between BLEs. The timing analyzer within T-VPack models three types of delay: the delay through a BLE, or *LogicDelay*, the connection delay between blocks within the same cluster or *IntraClusterConnectionDelay*, and the connection delay between blocks that are in different clusters, or *InterClusterConnectionDelay*. The delay of a connection between two BLEs in different logic clusters is not known until after a circuit has been placed and routed, so T-VPack approximates the delay between clusters as a constant InterClusterConnectionDelay. Note that this leads to some inaccuracy in T-VPack's estimate of where the critical path lies, so that sometimes T-VPack will be attempting to shorten a path which will not be part of the post-place-and-route critical path. The performance of T-VPack is not very sensitive to the exact values chosen for these three delay parameters. Throughout this work we set LogicDelay to 0.1, IntraClusterConnectionDelay to 0.1 and InterClusterConnectionDelay to 1.0.

We define the criticality of a connection, i, to be

$$ConnectionCriticality(i) = 1 - \frac{slack(i)}{MaxSlack} \quad (3.3)$$

where *MaxSlack* is the largest slack amongst all connections in the circuit.

The first (seed) BLE packed into a new cluster is the unclustered BLE with the highest *Criticality*. We explain exactly how the Criticality of a BLE is computed below; basically, BLEs attached to connections with high ConnectionCriticalities have high

criticalities. Once a seed is chosen, the attraction function used to determine the next unclustered BLE, B, to be added to the current cluster, C, is:

$$Attraction(B) = \lambda \cdot Criticality(B) + (1 - \lambda) \cdot \frac{|Nets(B) \cap Nets(C)|}{MaxNets} \quad (3.4)$$

Notice that the second term in (3.4) is essentially the attraction function from the original VPack algorithm. The MaxNets factor in the denominator of this term is the maximum number of nets that could connect to any BLE ($I + N + M_{clk}$) and simply normalizes the magnitude of the second term. The first term in (3.4) promotes the grouping of BLEs with high criticalities (defined below) into the current cluster in order to minimize delay. λ is a parameter that controls the trade-off between net sharing and delay minimization. If λ is 0, we have an algorithm that focuses solely on minimizing the number of used inputs to a cluster, and is equivalent to the basic VPack algorithm of Section 3.1.2. If λ is 1, the algorithm focuses solely on minimizing the delay of a circuit, with no regard for how many nets are shared by the BLEs within a cluster. We have experimentally determined that any λ value in the range from 0.4 to 0.8 produces packings of essentially the same quality in terms of both post-place-and-route delay and routing area. Throughout this work we set λ to 0.75.

The *Criticality* of a BLE is computed as follows. Let us first define the base criticality of an unclustered BLE, or *BaseBLECrit(B)*. BaseBLECrit is defined slightly differently depending on whether we are choosing a seed BLE for a new cluster or computing the attraction of a BLE to the current cluster:

1. When we are choosing a seed BLE, BaseBLECrit(B) is the maximum ConnectionCriticality value amongst all of BLE B's connections; and
2. When we are computing the attraction of a BLE to the current cluster, BaseBLECrit(B) is the maximum ConnectionCriticality value amongst all the connections joining BLE B to BLEs within the cluster currently being packed, C. If a BLE does not have any connections to C then its base criticality score is zero.

In Figure 3.5 we illustrate how the BaseBLECrit values are assigned when we are computing the attraction of a BLE to the current cluster. Each connection between unclustered BLEs and BLEs within the cluster C is labelled with its ConnectionCriticality value. Notice how the base criticality of each BLE is set to the highest criticality amongst the connections between it and the cluster being packed.

During packing, multiple BLEs often have the same base criticality value. In this case, we use a tie-breaker mechanism to select which BLEs are the most beneficial to pack. This mechanism is designed to choose (from the BLEs tied with the highest base criticality value) the BLE whose packing would reduce the length of the largest number of critical paths. This is best illustrated by an example.

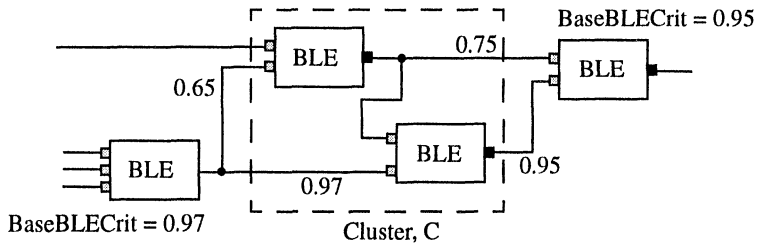


FIGURE 3.5 Determining BaseBLECrit from connection criticalities.

In Figure 3.6 we have darkened connections and BLEs on the critical paths. Notice that when selecting which BLEs to place into a cluster, it is more beneficial to absorb certain critical BLEs over other critical BLEs. In this case, absorbing BLEs X, Y, and Z would be much more beneficial than absorbing BLEs Q, T, and V. We can see that absorbing X, Y, and Z affects three partially-overlapping critical paths, and will reduce the criticality of seven other BLEs (Q, R, S, T, U, V and W). On the other hand, absorbing Q, T, and V affects only one critical path, and will not reduce the criticality of any other BLEs, since all the other critical BLEs would still lie on a critical path after the packing of Q, T, and V into a single cluster. Clearly it is best to cluster BLEs that reduce the criticalities of the most other BLEs.

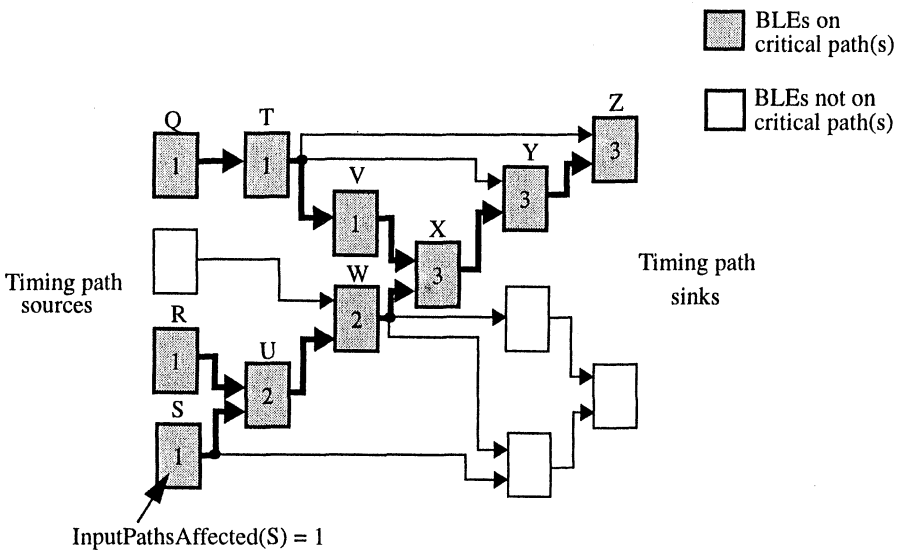


FIGURE 3.6 Example of first criticality tie-breaker.

We define three variables that keep track of the number of critical paths that each BLE in the circuit affects. First we define *InputPathsAffected* as the number of critical paths between timing path sources (primary inputs or register outputs) in the circuit and the BLE currently being labelled. Next we define *OutputPathsAffected* as the number of critical paths between the BLE currently being labelled and the timing path sinks (primary outputs or register inputs). Finally, we define *TotalPathsAffected* as the sum of the previous two variables. The calculation of these variables is explained below.

In Figure 3.6 each BLE is labelled with its InputPathsAffected value. We assign any timing path source nodes that are on the critical paths an InputPathsAffected value of one, and all other timing path source nodes are assigned an InputPathsAffected value of zero. Then we perform a breadth-first traversal of the circuit starting at the sources, and define the InputPathsAffected value as

$$InputPathsAffected(B) = \sum_{\forall D \in critical\ inputs(B)} InputPathsAffected(D) \quad (3.5)$$

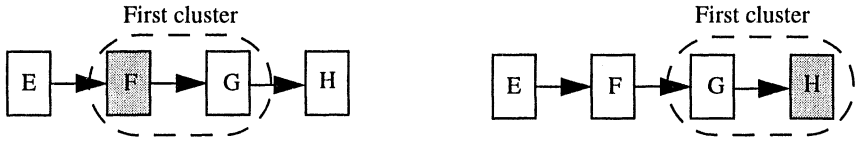
where *critical inputs(B)* refers to the BLE(s) driving the connections on *B*'s input(s) that are on the critical path. The OutputPathsAffected variable is calculated in the same manner, but it starts at timing path sink nodes and works back toward the timing path sources.

$$OutputPathsAffected(B) = \sum_{\forall D \in critical\ outputs(B)} OutputPathsAffected(D) \quad (3.6)$$

TotalPathsAffected is then simply

$$TotalPathsAffected(B) = InputPathsAffected(B) + OutputPathsAffected(B) \quad (3.7)$$

When two BLEs have the same BaseBLECrit, we break the tie by choosing to insert the BLE with the higher TotalPathsAffected value in the cluster. While this breaks most ties, it does not resolve all of them. Consider the simple circuit shown in Figure 3.7, and the selection of a seed BLE for the first cluster. There is only one path through the circuit, so all 4 BLEs are on the critical path, and have a BaseBLECrit value of 1. Similarly, all 4 BLEs have a TotalPathsAffected value of 2, so we have a four-way tie for the best BLE to use as a seed. Figure 3.7(a) shows a potential outcome if we randomly choose one of the four tied BLEs as the cluster seed when the cluster size is 2. If we choose BLE F as the cluster seed, and then BLE G is chosen as the second BLE in this cluster, we have "marooned" BLEs E and H — it is not possible to pack either E or H with its fan-in or fan-out. If instead, we choose BLE H as the seed of the first cluster (as Figure 3.7(b) shows), the first cluster would contain G



(a) BLE F chosen as seed of first cluster (b) BLE H chosen as seed of first cluster

FIGURE 3.7 Example of second criticality tie-breaker.

and H, and E and F could still be packed together into one cluster. Clearly, the clustering shown in Figure 3.7(b) is preferable to that of Figure 3.7(a).

We use a second tie-breaker mechanism to break ties not resolved by the first tie-breaker in order to reduce the likelihood of "marooning" BLEs. We always choose to pack the tied BLE that is the farthest from the timing path sources (i.e. is the closest to the timing sinks).[1] In Figure 3.7, for example, this second tie-breaker causes T-VPack to always choose BLE H as the seed of the first cluster, so the superior clustering solution of Figure 3.7(b) is achieved.

The criticality of a BLE is its BaseBLECrit slightly adjusted by these two tie-breakers:

$$Criticality(B) = BaseBLECrit(B) + \varepsilon \cdot TotalPathsAffected(B) + \varepsilon^2 \cdot D_{source}(B) \quad \textbf{(3.8)}$$

where ε is a small number (e.g. 0.01) to ensure that the second two terms function only as tie-breakers, and $D_{source}(B)$ is a BLE's distance, or level, from the timing path sources.

We have found that incorporating the two tie breakers described above into the criticality portion of the BLE attraction function leads to a 3% increase in the average post-place-and-route circuit speed vs. not including the tie breakers.

Computational Complexity vs. Frequency of Timing Analysis

The computational complexity of T-VPack depends on how often timing analysis is performed on the circuit as BLEs are packed into logic clusters. If timing analysis is performed after each BLE is packed, one has the most up-to-date view of the critical-

1. Note that choosing to always pack the tied BLE closest to the timing path sources would work just as well. The key is simply to ensure that one consistently chooses BLEs from one end of a chain of tied BLEs, rather than from the middle or from a mix of the two ends.

ity of each BLE, but the algorithm is $O(n^2)$, where n is the number of BLEs in the circuit. In this case T-VPack requires about fifteen minutes on a 300 MHz UltraSparc workstation to pack the largest (8383 BLEs) MCNC benchmark circuit [136].

The CPU time can be reduced if timing analysis is performed less often. If timing analysis is performed only after every P BLEs are packed, the algorithm complexity is $O(n^2/P)$. If one performs timing analysis only once before any BLEs have been packed, the algorithm has the same complexity as VPack: $O(k_{max} \cdot K \cdot n)$ where k_{max} is the maximum number of terminals of any net and K is the number of inputs to each BLE. In this case, T-VPack requires only a few seconds to pack an 8383 BLE circuit. Interestingly, performing only a single timing analysis and never updating connection criticalities as BLEs are packed does not noticeably degrade the result quality of T-VPack. Apparently the inaccuracy in the post-place-and-route critical path predicted by the timing analyzer in T-VPack (which assumes the same delay for each inter-cluster connection) is large enough that there is no benefit to updating connection criticalities as more information about the circuit packing becomes available. Performing no timing analysis at all (and simply assuming every connection in the circuit is equally critical) does reduce the effectiveness of T-VPack, and reduces circuit speed by 5% to 10%. Accordingly, throughout this book we present the T-VPack results obtained when timing analysis is performed only once, before any BLEs have been packed.

3.1.4 Result Quality of T-VPack vs. VPack

Table 3.1 summarizes the performance of the basic VPack algorithm and the enhanced, T-VPack, algorithm on the 10 largest MCNC benchmark circuits. The logic cluster targeted in this experiment contains 8 BLEs. The inputs required for 98% logic utilization columns in Table 3.1 list the minimum number of input pins (I) a logic cluster must have in order for the packing algorithm to fill clusters to 98% capacity, on average. Notice that T-VPack requires slightly (3%, on average) fewer cluster inputs to achieve 98% utilization than the basic VPack algorithm. The criticality term in the T-VPack attraction function, (3.4), makes T-VPack favour clustering a BLE with its fan-in or fan-out vs. clustering it with BLEs with which it shares inputs. For large clusters (5 or more BLEs) this preference to cluster a BLE with its fan-in and fan-out causes T-VPack to achieve slightly better logic utilization than VPack. Other researchers have also found that grouping circuit blocks with their fan-in or fan-out tends to be an effective clustering technique [52].

The remaining columns in Table 3.1 compare the post-place-and-route performance of circuits packed with the two different algorithms. To generate the results listed in these columns, the number of inputs per cluster was set to 18, which allows good logic utilization for all 10 circuits. For details of the FPGA routing architecture used, see Section 6.2.5.

TABLE 3.1 Comparison of VPack and T-VPack result quality.

Circuit	Cluster Inputs (I) Required for 98% Logic Utilization		Minimum Channel Width for Successful Routing (W_{min})		Post-Place-and-Route Critical Path (ns) $W = \infty$[a]		Post-Place-and-Route Critical Path (ns) $W = W_{min} + 20\%$	
	VPack	T-VPack	VPack	T-VPack	VPack	T-VPack	VPack	T-VPack
apex2	19	19	58	55	38.6	32.7	37.9	34.5
clma	17	17	75	64	83.0	64.7	84.3	77.1
elliptic	16	17	57	49	50.1	40.1	60.5	49.1
ex1010	20	20	61	58	45.7	42.8	53.0	56.3
frisc	16	16	57	58	75.1	56.4	80.2	65.3
pdc	20	18	82	76	56.7	53.3	57.9	81.8
s298	18	15	48	28	49.9	49.7	58.0	63.3
s38417	14	14	47	42	51.2	42.3	59.7	45.0
s38584.1	13	12	43	44	40.3	29.3	40.2	30.3
spla	19	18	76	59	49.4	41.0	48.3	47.0
Arith. Av.	17.2	16.6	60.4	53.3	54.0	45.2	58.0	55.0
Geom. Av.	17.0	16.4	59.1	51.6	52.5	44.1	56.3	52.5

a. Note that the congestion-oblivious (infinite routing) delay is *not* a lower bound on the achievable delay, so occasionally the delay with limited routing is lower than the infinite-routing delay. In fact there is currently no known algorithm to route a net with guaranteed minimum Elmore delay, short of exhaustively searching all possibilities [129].

Two columns in Table 3.1 list the minimum number of tracks per channel (W_{min}) required to successfully route the packed circuit produced by each algorithm. Somewhat surprisingly, using T-VPack instead of VPack results in circuits that require 12% fewer tracks for successful routing. To understand the reason for this result, one must compare the structure of the packed circuits produced by T-VPack and by VPack. The criticality term in the T-VPack attraction function, (3.4), makes T-VPack prefer to cluster a BLE with BLEs in its fan-in or fan-out, rather than with other BLEs that share inputs with it. As a result, T-VPack produces circuit packings in which many low-fanout nets have been completely absorbed into logic clusters. Overall, the output of T-VPack has fewer nets to route between clusters than the output of VPack, but the average fanout of each inter-cluster net is higher with T-VPack than with VPack.

The net result is that the output of T-VPack is somewhat easier to route than the output of VPack.[1]

The post-place-and-route critical path columns in Table 3.1 compare the relative speeds of circuits implemented with the two different packing algorithms. One set of post-place-and-route speed results assumes that the circuits have essentially an infinite amount of general purpose routing available. In this case, the router is able to focus entirely on speed optimization, rather than congestion avoidance, so we obtain a good estimate of the speed difference between the two packings when the circuits are mapped into routing-rich FPGAs. The W = W_{min} + 20% speed results, on the other hand, show the speed difference between the two packings when an FPGA has a more limited amount of interconnect — only 20% more than the minimum required by each circuit packing. As one would expect, T-VPack increases circuit speed vs. VPack, by 19% for the unlimited interconnect case, and 7% for the limited interconnect case, on average. Note also that in the limited interconnect case, the T-VPack circuits are faster than those of VPack despite the fact that the router is being given significantly (on average 12%) fewer tracks to route them than it is for the VPack-generated packings.

Overall it is clear that T-VPack outperforms the basic VPack algorithm in terms of both circuit speed and routing area required.

3.2 Placement: VPR

The Versatile Place and Route (VPR) CAD tool performs placement and either global routing or combined global-detailed routing for FPGAs [17, 134]. The name Versatile Place and Route reflects the primary goal in this tool's design: be able to target a wide variety of FPGA architectures. In the following sections we describe the capabilities of and algorithms used in the VPR placement tool. In Chapter 4, the routing portion of VPR is described.

1. This result shows the importance of using a full CAD flow, including placement and routing, to evaluate many FPGA issues. It would have been difficult or impossible to guess that the output of T-VPack would be easier to route than the output of VPack without actually placing and routing the outputs of both packing algorithms. In fact, since the circuit packings produced by T-VPack have more point-to-point connections to route between clusters (despite having fewer nets) one would likely guess that VPack's circuits would be easier to route.

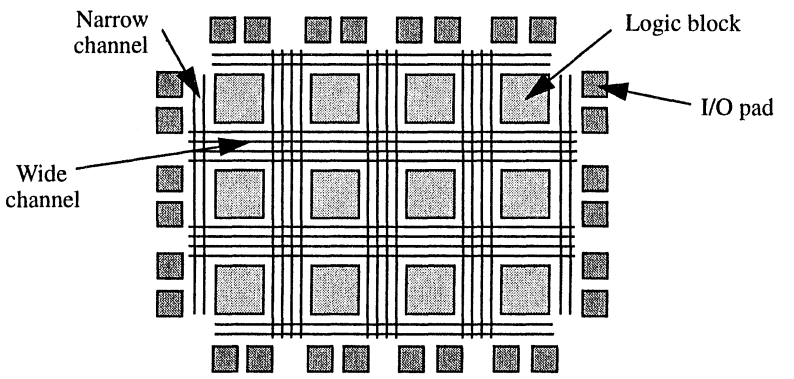


FIGURE 3.8 FPGA model assumed by VPR placer.

3.2.1 Overview of the VPR Placement Tool

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

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

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

In Figure 3.8, for example, two I/O pads fit into each row or column of the FPGA, the logic block array is 4 x 3 logic blocks, and the channels between the logic block array and the I/O pads are only half as wide as those within the logic block array. Notice that perimeter I/O is assumed (i.e. wire-bonded pads, rather than flip-chip), so I/O blocks can only be placed on the edges of the FPGA.

Users can have VPR determine good locations for the I/O pads during placement, or they can lock the I/O pads in a configuration specified by an input file, or have VPR generate a random locked I/O configuration. This last capability allows the tool to measure the effect of random pin assignment on different architectures — an important issue in FPGAs [135]. Locked I/Os occur frequently in real FPGA designs, since the circuit board on which an FPGA is mounted is often fabricated before the FPGA circuit design is competed. In such cases, the placement tool must keep the circuit I/Os in the locations required by the circuit board design.

VPR uses the simulated annealing algorithm, which was described in detail in Section 2.2.2. Since the basic application of simulated annealing to placement is well known, the following sections focus on the aspects of our implementation that are improvements to the prior art.

3.2.2 New Adaptive Annealing Schedule

As described in Section 2.2.2, a good annealing schedule is essential to obtain high-quality solutions in a reasonable computation time with simulated annealing. Recall that an annealing schedule specifies the number of moves to attempt per temperature, how temperature varies throughout the anneal, and when the anneal should terminate. Our placement tool targets many different FPGA architectures, can use several different cost functions, and is used with a wide variety of circuits that span a large range of sizes. Consequently, we need an annealing schedule that automatically adapts to the current placement problem; fixed annealing schedules will not work well. We have developed a new schedule that incorporates some of the best features from the Huang et al [82], Lam and Delosme [83] and Swartz and Sechen [84] schedules, while using a new temperature update scheme and exit criterion.

Three parts of this annealing schedule are taken from prior work. First, we compute the initial temperature in the same way as Huang et al [82]. Let N_{blocks} be the total number of logic blocks plus the number of I/O pads in a circuit. We first create a random placement of the circuit. Next, we perform N_{blocks} moves (pairwise swaps) of logic blocks or I/O pads, and compute the standard deviation of the cost of these N_{blocks} different configurations. The initial temperature is set to 20 times this standard deviation, ensuring that virtually every move is accepted at the start of the anneal.

The second feature of our annealing schedule taken from prior work is the number of new placement configurations evaluated at each temperature. As in [84], we set the number of moves per temperature to

$$MovesPerTemperature = InnerNum \cdot (N_{blocks})^{4/3} \quad \textbf{(3.9)}$$

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

It was shown in [83, 84] that it is desirable to keep the fraction of moves accepted, α, near 0.44 for as long as possible. We accomplish this in the same way as [83], by using the value of α to control a range limiter — only interchanges of blocks that are less than or equal to R_{limit} units apart in the x and y directions are attempted. A small value of R_{limit} increases α by ensuring that only blocks which are close together are considered for swapping. These "local swaps" tend to result in relatively small changes in the placement cost, increasing their probability of acceptance. Initially, R_{limit} is set to the span of the entire chip. Whenever the temperature is reduced, the value of R_{limit} is updated according to the value of α measured at the old temperature:

$$R_{limit}^{new} = R_{limit}^{old} \cdot (1 - 0.44 + \alpha) \tag{3.10}$$

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

The key difference between our new annealing schedule and previous schedules lies in our method of updating the temperature as the anneal progresses. When the temperature is so high that almost any move is accepted, we are essentially moving randomly from one placement to another and little improvement in cost is obtained. Conversely, if very few moves are being accepted (because the temperature is very low and the current placement is of fairly high quality), there is also little improvement in cost. With this motivation in mind, we created a temperature update scheme that increases the amount of time spent at the most productive temperatures — those where a significant fraction of, but not all, moves are being accepted. We use the fraction of moves being accepted, α, to directly control how quickly the temperature drops. A new temperature is computed as $T_{new} = \gamma \cdot T_{old}$, where the value of γ depends on the fraction of attempted moves that were accepted (α) at T_{old}, as shown in Table 3.2. The exact values of α and γ listed in Table 3.2 were found via experimentation. While the values in Table 3.2 led to the best performance, the performance of the annealer is not extremely sensitive to the exact value of γ as a function of α. So long as the function $\gamma(\alpha)$ has the right form — γ is near 1 for α around 0.44, and γ is significantly smaller for α near 1 or 0 — the annealer performs reasonably well.

We terminate the anneal when:

$$T < \varepsilon \cdot \frac{Cost}{N_{nets}}, \tag{3.11}$$

TABLE 3.2 Temperature update schedule.

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

where N_{nets} is the number of nets in the circuit, and we use an ε value of 0.005. The movement of a logic block will always affect at least one net. When the temperature is less than a small fraction of the average cost of a net, it is unlikely that any move that results in a cost increase will be accepted, so we terminate the anneal. Again, the performance of the annealer is not terribly sensitive to the ε factor in (3.11). Any value between 0.05 and 0.005 is reasonable, with smaller values giving slightly higher quality placements at the cost of slightly increased CPU time.

The annealing schedule described above has produced excellent results with a wide variety of cost functions, FPGA architectures, circuits, and moves per temperature (i.e. desired quality) values. For comparison purposes, we also implemented an annealing schedule based on that of Huang et al [82][1]. This annealing schedule was not sufficiently robust for our purposes, particularly with large circuits. For some circuits its temperature update scheme became too conservative, and the temperature decreased extremely slowly. For some other circuits its exit criterion did not function correctly, and a large amount of CPU time was wasted at very low temperatures with no significant quality improvement. Table 3.3 compares the two annealing schedules on a set of eight MCNC benchmark circuits [136]. On the six smaller benchmarks the two schedules were fairly comparable — our schedule was 1.35x faster, but resulted in placements with 0.9% higher cost. The Huang schedule broke down on the two larger circuits, however; it took 3.8x more CPU time on alu4, and did not complete after 40x as much CPU as our schedule required on bigkey. We want to be able to control the quality-time trade-off of our tools, so an annealing schedule that is unpredictable, i.e. spends huge amounts of time on some circuits, is not suitable for our purposes.

1. This annealing schedule used the initial temperature, exit criterion and temperature update scheme of [82]. It used the number of moves per temperature value from [84] (equation (3.10)), however, as this provides a parameter for controlling the CPU time / quality trade-off.

TABLE 3.3 Comparison of new annealing schedule to schedule of Huang et al.

Circuit	N_{blocks}	Schedule of Huang et al vs. new schedule	
		CPU ratio	Final cost difference
s1423	338	1.03x	+0.6%
e64	466	1.16x	-0.8%
table3	560	1.30x	-1.6%
sbc	561	1.24x	-0.4%
x3	648	1.40x	-2.4%
C5315	1200	1.99x	-0.7%
bigkey	1914	terminated after 40x	-5.3%
alu4	2319	3.82x	-1.5%
7 Circuit Average	870.3	1.71x	-1.0%
8 Circuit Average	1000.8	6.5x	-1.5%

We have not implemented the schedules of Lam and Delosme [83] or Swartz and Sechen [84], so we cannot compare our performance to theirs. Note however, that our schedule is much simpler than that of Lam. It is also completely adaptive (all parameters adapt to the problem at hand), while the Swartz schedule uses a hard-coded range-limiter variation and number of temperatures to visit.

3.2.3 New Cost Function: Linear Congestion

One of the FPGA architectural issues we investigate is global routing architecture. Recall that many global routing architectures have wider channels in some regions of the FPGA than in others; see Figure 1.1 for an example. To fully optimize for such architectures, those portions of a circuit that require more routing should be placed in regions of the FPGA that have wider routing channels. The key to obtaining such global-routing-architecture-aware placements is ensuring that the cost function used properly models the relative difficulty of routing connections in areas with different channel widths. Accordingly, we developed what we call a *linear congestion* cost function. Of all the alternatives we have explored, this cost function provides the best results in a reasonable computation time. Its functional form is

$$Cost_{linear\ congestion} = \sum_{i=1}^{N_{nets}} q(i)\left[\frac{bb_x(i)}{C_{av,x}(i)^{\beta}} + \frac{bb_y(i)}{C_{av,y}(i)^{\beta}}\right], \quad (3.12)$$

where the summation is over the N_{nets} in the circuit. For each net, i, $bb_x(i)$ and $bb_y(i)$ denote the horizontal and vertical spans of its bounding box, respectively. The q(i) factor compensates for the fact that the bounding box wire length model underestimates the wiring necessary to connect nets with more than three terminals [89]. Its value depends on the number of terminals of net i. We obtained the appropriate values of q(i) from [89]; q(i) is 1 for nets with 3 or fewer terminals, and slowly increases to 2.79 for nets with 50 terminals. $C_{av,x}(i)$ and $C_{av,y}(i)$ are the average channel capacities (in tracks) in the x and y directions, respectively, over the bounding box of net i.

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

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

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

For comparison, we also implemented the cost function of [89], which we call a *nonlinear congestion* cost function. We described this cost function in Section 2.2.2. Recall that it divides the FPGA into an M x M array of regions and attempts to model

the routing resource demand and supply in each of these regions, penalizing placements that appear to result in routing demand exceeding supply in some regions.

Table 3.4 shows how the linear and nonlinear congestion cost functions compare to a bounding box cost function on one FPGA architecture. In this architecture, the 50% of the FPGA channels nearest the center are twice as wide as the other channels. 36 MCNC [136] benchmarks were placed and globally routed in the smallest FPGA that could accommodate each. Table 3.4 shows how the number of tracks in the FPGA required to complete global routing and the placement CPU time is affected by the cost function used. Note that 10 of the 36 circuits were I/O bound in the FPGA assumed, so separate columns are provided for the average over these 10 circuits, the average over the 26 logic-bound circuits, and the overall average. The routability improvements with the more advanced cost functions are larger for the I/O bound circuits, since there are "spare" logic blocks in the FPGAs in these cases, giving the placement tool the option of spreading out the logic to improve routability.

TABLE 3.4 Cost function performance comparison on an example FPGA.

Cost function	Average Tracks Required for Global Routing			CPU Time
	All 36 circuits	26 logic-bound	10 I/O bound	
Bounding box	1x	1x	1x	1x
Linear congestion	-7%	-4%	-16%	1x
Nonlinear cong. (16 regions)	-8%	-5%	-16%	5x
Nonlinear cong. (256 regions)	-10%	-6%	-22%	80x

Notice that the non-linear congestion cost function, when computed on a 4 x 4 grid (16 regions), produced placements which required, on average, only 1% fewer tracks to route than those produced by the linear congestion cost function. However, keeping track of the routing resource demand in the various chip regions is computationally expensive, and placement with this cost function requires five times greater CPU time than the linear congestion function. Dividing the FPGA into smaller subregions to make localized congestion more visible improved the performance of this cost function slightly; a non-linear congestion cost function computed on a 16 x 16 grid (256 regions) performs 3% better than our linear congestion cost function on this architecture. However, it consumes 80 times the CPU time. We consider the routability improvements gained by the nonlinear congestion cost function too small to warrant the huge increase in CPU time, and so prefer the linear congestion cost function developed in this work.

3.2.4 Incremental Net Bounding Box Updates

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

Consider the computation of ΔC caused by the swap of two blocks. The only terms in the summation of (3.12) that change are those corresponding to the nets attached to the two swapped blocks. The bounding boxes of all the nets attached to either of these two blocks must be recomputed, and then (3.12) can be used to determine ΔC. The recomputation of the net bounding boxes is the key step here; unless care is taken, it can dramatically slow the placer.

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

We have developed an alternative to this brute-force computation, which we call *incremental bounding box evaluation.* For each net, we store the coordinate of each of the four sides of its bounding box (x_{min}, x_{max}, y_{min}, y_{max}), and the number of net terminals that lie on each of these sides (N_{xmin}, N_{xmax}, N_{ymin}, N_{ymax}). Figure 3.9 shows an example of this data storage.

Now say that some terminal of this net is moved via a swap from (x_{old}, y_{old}) to (x_{new}, y_{new}). Since we have stored the extra information shown in Figure 3.9, we can usually determine the new net bounding box by looking only at the terminal which moved, rather than all k terminals. Figure 3.10 lists the pseudo-code used to update the x_{min} and N_{xmin} values for a net i; the code for the other four sides is similar.

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

We have experimentally determined that our incremental bounding box update method is faster than the brute-force method for all nets with more than 4 terminals.

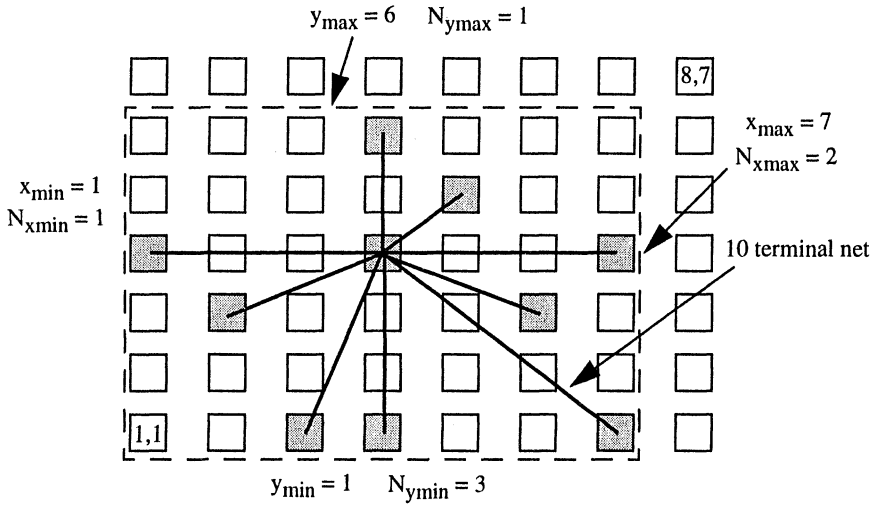


FIGURE 3.9 Data stored to enable incremental bounding box updates.

```
if (x_new != x_old) {    /* Terminal has moved horizontally */
    if (x_new < x_min(i)) {    /* Terminal moved left past old x_min edge */
        x_min(i) = x_new;
        N_xmin(i) = 1;
    }

    else if (x_new == x_min(i)) {    /* Terminal moved left to lie on the old x_min edge */
        N_xmin++;
    }

    else if (x_old == x_min(i)) {    /* Terminal was on x_min edge; moved right */
        if (N_xmin > 1) {            /* Still terminals on x_min edge? */
            N_xmin--;
        }
        else {
            BruteForceBoundingBoxRecompute (i);
        }
    }
}
```

FIGURE 3.10 Pseudo-code to update the bounding box of net i incrementally.

Using this more sophisticated technique for nets with more than 4 terminals yields, on average, a more than five times speedup in the placer. Table 3.5 compares the CPU time on a 300 MHz UltraSparc needed to place the ten largest MCNC benchmark circuits with and without incremental bounding box recalculation. In this experiment each logic block is a BLE (4-LUT / FF pair). The speedup due to incremental bounding box updates ranges from 2.52 times to 9.41 times, with an average speedup of 5.39 times. The variation in speedup is due to the different fanout distributions of these circuits — circuits with a higher average fanout benefit more. Since placement is so time-consuming, and there is considerable need for fast CAD tools as FPGA sizes increase [127], this speedup is very important. Another useful feature of the incremental bounding box code is that it makes the CPU time required to place a circuit highly predictable from the circuit size; this allows a CAD tool to give a user an accurate estimate of the time required to place a circuit.

TABLE 3.5 Placement CPU time vs. bounding box recalculation method.

Circuit	# Logic Blocks	CPU (s) without incremental bounding box	CPU (s) with incremental bounding box	Speedup
apex2	1878	116	46	**2.52x**
s298	1931	386	41	**9.41x**
frisc	3556	599	127	**4.72x**
elliptic	3604	864	125	**6.91x**
spla	3690	434	125	**3.47x**
pdc	4575	664	172	**3.86x**
ex1010	4598	706	183	**3.86x**
s38417	6406	1405	315	**4.46x**
s38584.1	6447	2815	340	**8.28x**
clma	8383	3207	499	**6.43x**
Arithmetic Average:	4507	1120	197	**5.39x**
Geometric Average:	4071	755	151	**4.99x**

The InnerNum value for these results was set to 1; to obtain the highest quality (5 - 10% better) results, InnerNum is set to ten, and therefore ten times more CPU is required than Table 3.5 lists. Hence one can see why speeding up placement is important!

3.3 Summary

In this chapter we have described our logic block packing tools, VPack and T-VPack, and the placement portion of the VPR placement and routing tool. VPack and T-VPack are the first publicly-described logic block packing tools targeting cluster-based logic blocks. VPR incorporates three enhancements over prior simulated-annealing placement algorithms: a new annealing schedule, a routability-enhancing linear congestion cost function, and an incremental net bounding box update method that reduces placement CPU time by a factor of over 5. The next chapter continues the description of the CAD tools developed for this book by describing the VPR router. At the end of the next chapter, the performance of the VPR placement and routing tool will be compared to that of prior CAD tools, allowing us to evaluate the result quality of the placement tool described in this chapter.

CHAPTER 4

Routing Tools and Routing Architecture Generation

In this chapter we describe how the routing portion of VPR works. We begin by describing the spectrum of FPGA architectures that the router has targeted, and the understandable architecture parameters used to describe an FPGA to VPR. We then explain how a routing architecture is represented internally, and how the succinct description provided by a user is *automatically* turned into this highly detailed architecture representation. Next, we describe the two routers built into VPR; one is purely routability-driven, while the other is both timing- and routability-driven. The timing-driven router requires a fast and accurate net delay extractor and a path-based timing analyzer, both of which are also discussed. Finally, we compare the performance of VPR to that of several other published CAD tools, and show that it outperforms all the tools to which we have been able to compare.

4.1 Position within the CAD flow

Figure 4.1 shows where the VPR router fits into the CAD flow. Its input is a netlist of logic blocks and an architecture file which describes the target FPGA. A placement is either read in or the VPR placer is used to place the circuit, and then either the routability-driven or the timing-driven router built into VPR is invoked. Each of these routers can perform either a combined global-detailed routing, or global routing only. Once routing is complete, various statistics, such as routed wirelength, routing resource utilization and the delay of the critical path are output.

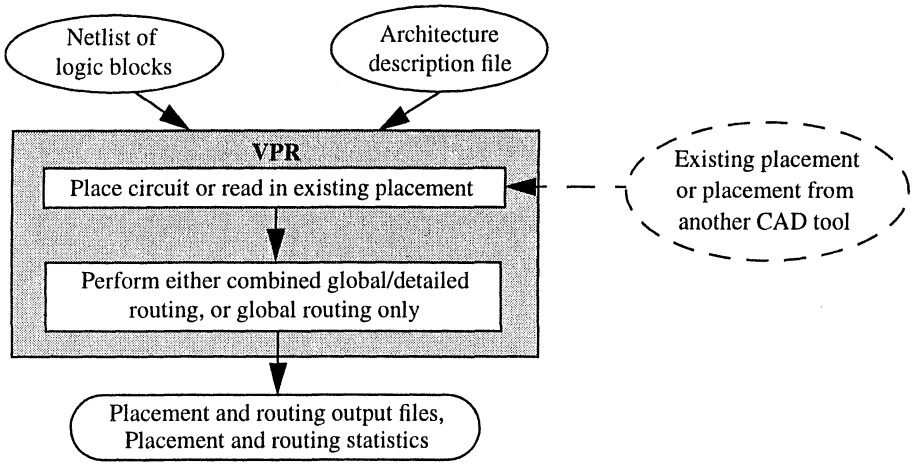


FIGURE 4.1 VPR router operation and position within the CAD flow.

4.2 Architecture Parameterization and Generation

Since our primary goal in this research is to investigate many different FPGA architectures, we would like our tools to:

1. Be as architecture-independent as possible, and
2. Be *easy* to use with different architectures — that is, we would like to be able to describe different architectures to VPR quickly.

To achieve the first goal, we have made the router, graphics routines, delay extractor, and various other routines all operate on a directed graph that describes the FPGA. This *routing-resource graph* representation is very general, and can describe a wide variety of FPGA architectures. Unfortunately, describing a new architecture by creating a routing-resource graph by hand is not feasible — the routing-resource graph to describe a typical FPGA containing 8000 4-LUTs is almost 30 MB in size. One possibility is to design a basic tile (a single logic block and its associated routing) manually, and create a program to automatically replicate and stitch together this tile into a routing-resource graph describing the entire FPGA. Even creating a basic tile manually is too time-consuming for our purposes, however. A typical tile contains several hundred programmable switches and wires, so it can take hours or days to describe even one tile. Furthermore, such a hand-crafted tile is designed for one value of routing channel width, W. In many of our experiments we wish to vary W in order to see

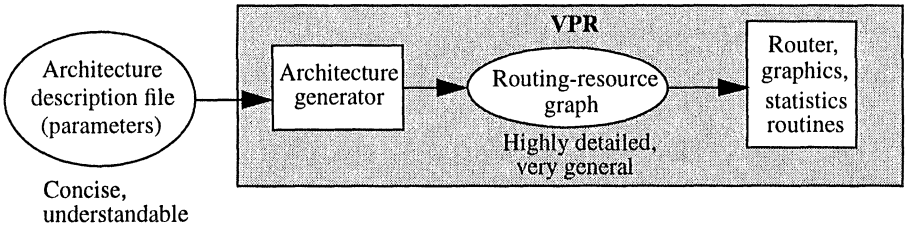


FIGURE 4.2 A concise architecture description is converted to a detailed graph description.

how routable a given FPGA architecture is, or how well our CAD tools optimize for routability. With a tile-based approach, we would have to hand-craft one tile for each different value of W, for each architecture. Since we wish to investigate hundreds of different FPGA architectures, and tens of W values for each of these architectures, we would have to create thousands or tens of thousands of these basic tiles.

In order to satisfy the ease of use goal, therefore, we have designed VPR so that it takes a concise, human-comprehensible architecture definition file, and uses an internal graph generator to create the highly detailed routing-resource graph representation which the router and other CAD routines use. Figure 4.2 illustrates the basic idea. If one wants to look at a completely new class of FPGAs that does not fit into the current format of the architecture description file, then, only the routing graph generator has to be modified; the router, graphics, timing analyzer, and statistics routines will all function correctly.

4.2.1 Architecture Parameterization

We want our architecture description file to be easy to create, so we tried to parameterize architectures in ways that are intuitive to FPGA researchers. By parameterizing our architectures we also make it easier to describe our results to other researchers, and to understand why one architecture is better than another. (Simply showing that one 30 MB routing-resource graph was superior to most others would not allow us to describe our results to others very easily!)

To perform routing we need more information about the architecture than is required for placement only. The architecture description file specifies:

- The number of logic block input and output pins,
- The side(s) of the logic block from which each input and output is accessible,

- The logical equivalence between the various input and output pins (e.g. all LUT inputs are functionally equivalent),
- The number of I/O pads that fit into one row or column of the FPGA,
- The relative widths of the horizontal and vertical channels, and
- The relative widths (number of tracks) of the channels in different regions of the FPGA.

If only global routing is desired, these are all the parameters needed. If combined global-detailed routing is desired, however, the architecture file must also specify:

- The switch block topology used to connect the routing tracks (i.e. which tracks connect to which at a switch block),
- The number of tracks to which each logic block input pin connects, $F_{c,input}$,
- The F_c value for logic block outputs, $F_{c,output}$,
- The F_c value for I/O pads, $F_{c,pad}$, and
- One or more *wire segment types.* For each segment type, one specifies:
    - The fraction of tracks in a channel that are of this segment type,
    - The segment length (number of logic blocks spanned by a wire segment),
    - The type of switch (pass-transistor or tri-state buffer, drive strength, etc.) used to connect a wire segment of this type to other routing segments,
    - The switch-block internal population of this segment type (discussed below), and
    - The connection-block internal population of this segment type (discussed below).

Note that the segmentation distribution (the fraction of routing tracks of each length), is specified as part of the wire type definitions.

We define internal population (described in Section 2.1.3) in a more general way than prior researchers. Recall that internal population describes whether or not logic blocks and routing wires can connect to the interior of a wire segment, or if connections to a wire can be made only at its ends. In [21], a wire segment is either completely internally populated or completely depopulated. We allow *partial depopulation* of the interior of a wire segment. For example, a length five segment spans five logic blocks. If we specify a connection-block population of 100%, this wire segment can connect to all five logic blocks it passes, so it is fully internally populated. If the connection-block population is 40%, it can only connect to the two logic blocks at its ends, so it is internally depopulated. If we specify a connection-block population of 60%, however, the wire can connect to the two logic blocks at its

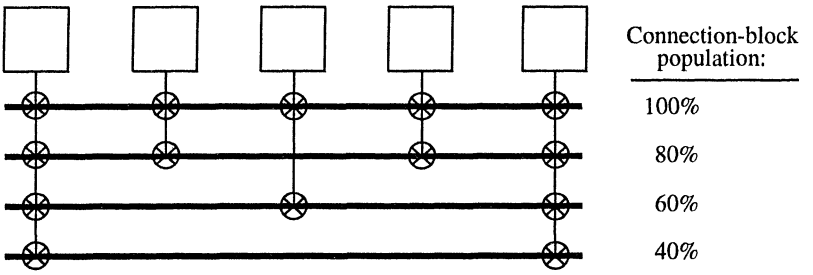


FIGURE 4.3 Possible connection-block population values for length 5 wire segments.

ends and one logic block in its interior, so it is partially internally depopulated. Figure 4.3 illustrates the four possible values of connection-block population for a length five wire. Switch-block population is specified in a similar, percentage, form.

Notice that we specify the distribution of wire types as fractions of the channel width, W, rather than as an absolute number of tracks of each type. For example, one might say there are 20% length = 2 wires and 80% length = 5 wires. This allows a user to attempt routings with different W values, to determine the routability of an architecture, without changing the architecture file. Similarly, the various F_c values can be specified either as absolute numbers (e.g. 5 tracks), or as a fraction of the tracks in a channel (e.g. 0.2·W).

The number of tracks per channel, W, and the size of the logic block array size can be specified on the command line. If one or more of these parameters is not specified, VPR will determine the minimum value(s) needed to fit the circuit in the specified FPGA architecture.

Finally, to allow extraction of the delay of routed nets and path-based timing-analysis, one must specify various timing parameters in the architecture description file. These include:

- The input and output capacitance, equivalent resistance, and intrinsic delay of each type of switch used in the routing; as many switch types as desired can be defined.
- The capacitance and resistance of each type of wire segment,
- The delays of all the combinational and sequential elements within each logic block, and
- The delays of the I/O pads.

4.2.2 The Routing-Resource Graph

While the architecture parameters listed above are easy for FPGA architects to understand and specify, they are not appropriate for use as an internal architecture representation for a router. Internally, VPR uses a routing-resource graph [85] to describe the FPGA; this is more general than any parameterization, since it can specify arbitrary connectivity. It also makes it much faster to determine connectivity information, such as the wires to which a given wire segment can connect, since this information is explicitly contained in the graph.

Each wire and each logic block pin becomes a node in this routing-resource graph and each switch becomes an directed edge (for unidirectional switches, such as buffers) or a pair of directed edges (for bidirectional switches, such as pass transistors) between the two appropriate nodes. Figure 4.4 shows the routing-resource graph corresponding to a portion of an FPGA whose logic block contains a single 2-input, 1-output LUT.

Often FPGAs have logically equivalent pins; for example, all the input pins to a LUT are logically equivalent. This means that a router can complete a given connection using any one of the input pins of a LUT; changing the values stored in the LUT can compensate for any re-ordering of which connection connects to which input pin performed by the router. We model this logical equivalence in the routing-resource graph by adding *source* nodes at which all nets begin, and *sink* nodes at which all net terminals end. There is one source node for each set of logically-equivalent output pins, and there is an edge from the source to each of these output pins. Similarly, there is one sink node for each set of logically-equivalent input pins, and an edge from each of these input pins to the sink node.

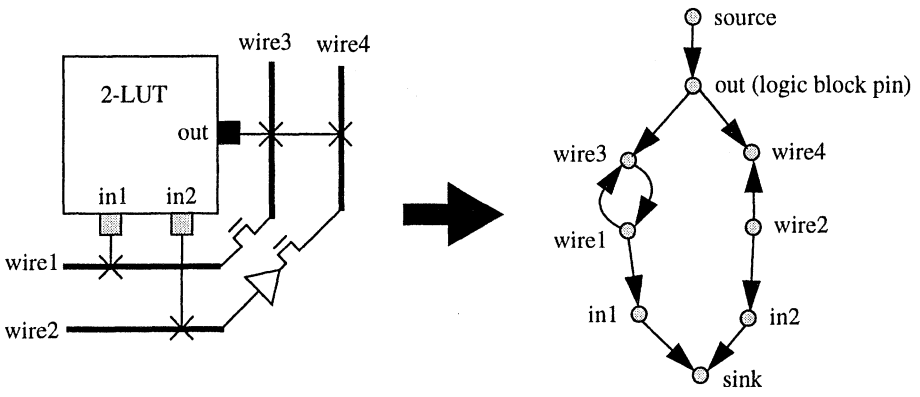


FIGURE 4.4 Modelling FPGA routing architecture as a directed graph.

To reduce the number of nodes in the routing-resource graph, and hence save memory, we assign a capacity to each node. A node's capacity is the maximum number of different nets which can use this node in a legal routing. Wire segments and logic block pins have capacity one, since only one net may use each. Sinks and sources can have larger capacities. For example, in a 4-input LUT, there is one group of four logically-equivalent inputs, so we have one sink of capacity four. If we could not assign a capacity of four to the sink, we would be forced to create four logically-equivalent sinks and connect them to the four input pins via a complete bipartite graph ($K_{4,4}$), wasting considerable memory.

By constructing an appropriate routing-resource graph we can perform global routing, instead of combined global-detailed routing, with no changes to our router code. Figure 4.5 shows a routing-resource graph for global routing. Again, each logic block pin becomes a node, and source and sink nodes are added for each logically-equiva-

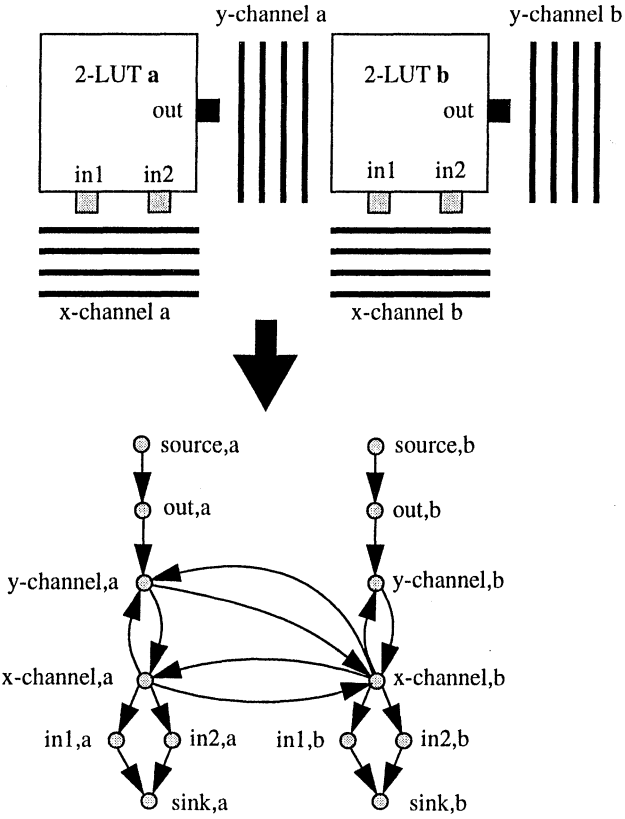


FIGURE 4.5 Modelling the FPGA global routing problem via a directed graph.

lent group of pins. Now, however, instead of each wire becoming a node, each channel segment (the length of channel that spans one logic block) becomes a routing-resource node. The capacity of each of these nodes is equal to the number of wires, or tracks, in the corresponding channel segment; in Figure 4.5, for example, the capacity of each channel segment node is 4. Edges are added to connect logic block input and output pins to the adjacent channel segments, and to connect adjacent channel segments together.

To perform timing-driven routing, timing analysis, and to graphically display the architecture (see Appendix A), we need more information than just the raw connectivity embodied in the nodes and edges of the routing-resource graph. Accordingly, we annotate each node in the graph with its type (wire, input pin, etc.), location in the FPGA array, capacitance and metal resistance. Each edge in the graph is marked with the index of its "switch type," allowing retrieval of information about the switch intrinsic delay, equivalent resistance, input and output capacitance and whether the switch is a pass transistor or tri-state buffer.

4.2.3 Automatic Architecture Generation from Parameters

As the previous two sections have described, there are compelling reasons to allow designers to specify architectures in an understandable, parameterized format, and for the routing tools to work with a more detailed, graph-based, description. As Figure 4.2 shows, then, VPR must be capable of automatically generating a routing-resource graph from a set of specified architecture parameters. Generating an architecture automatically is a difficult problem for two reasons:

1. We want to create a *good* architecture with the specified parameters. That is, the unspecified properties of the architecture should be set to "reasonable" values.
2. Simultaneously satisfying all the parameters defining the architecture is difficult. In some cases, the specified parameters conflict and overspecify the FPGA, making it impossible to simultaneously satisfy all the specified constraints.

Consider the first problem. If we require a user to specify every conceivable parameter, and every interaction between these parameters, describing an architecture will be very time-consuming. Instead, we want to allow users to specify the important parameters, and have the VPR tool automatically adjust other parameters of the architecture so that a good FPGA results. For example, we require that a user specify the number of tracks to which input and output pins can connect, $F_{c,input}$ and $F_{c,output}$, rather than requiring a user to specify the complete connection block switch pattern. This certainly simplifies the task of describing an FPGA, but it means that VPR must generate a good connection block switch pattern automatically.

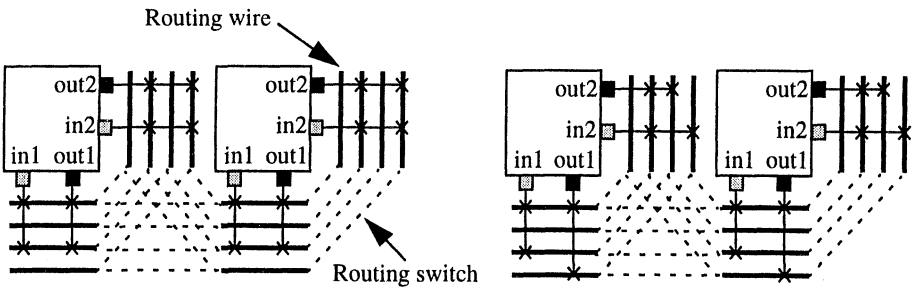
Let us consider this connection block problem in more detail. We decided that the switch pattern chosen should:

- Ensure that each of the W tracks in a channel can be connected to roughly the same number of input pins, and roughly the same number of output pins,
- Ensure that each pin can connect to a mix of different wire types (e.g. different length wires),
- Ensure that pins that appear on multiple sides of the logic block connect to different tracks on each side, to allow more routing options,
- Ensure that logically-equivalent pins connect to different tracks, again to allow more routing options, and
- Ensure that pathological switch topologies in which it is impossible to route from certain output pins to certain input pins (see Figure 4.6) do not occur.

Clearly this is a complex problem. In essence, the proper connection block pattern is a function of $F_{c,input}$, $F_{c,output}$, W, the segmentation distribution, the logical equivalence between pins, and the side(s) of a logic block from which each pin is accessible. The last condition is also a function of the switch block topology. VPR attempts to build a connection block that satisfies the five criteria above, but it will not necessarily perfectly satisfy them all for all architectures.

The second difficulty in generating an architecture automatically is simultaneously meeting all the user-defined specifications. We will illustrate this difficulty with an example that shows it often takes considerable thought to simultaneously satisfy the specifications. Consider an architecture in which:

- Each channel is three tracks wide.
- Each wire is of length 3.



(a) Nets starting at out2 can only reach in2, nets starting at out1 can only reach in1

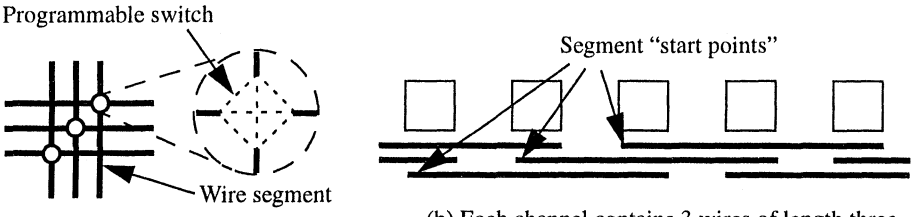
(b) Nets starting at either output can reach either input; vastly improved routability

FIGURE 4.6 Example connection block patterns: (a) pathologically bad; (b) good.

- Each wire has an internal switch block population of 50%. That is, routing switches can connect only to the ends of a wire segment (2 of the 4 possible switch block locations).
- The switch block topology is *disjoint* [101]. In this switch block, wires in track 1 always connect only to other wires in track 1, and so on. This is the switch block topology used in the original Xilinx 4000 FPGAs [137].

Figure 4.7 shows the disjoint switch block topology, and a channel containing 3 wires of length 3. Notice that the "start points" of the wire segments are staggered [37]. This enhances routability, since each logic block in the FPGA can then reach a logic block two units away in either direction using only one wire segment. It also arises naturally in a tile-based layout, so staggering the start points of the segments in this way makes it easier to lay out the FPGA. A tile-based FPGA layout is one in which only a single logic block and its associated routing (one vertical channel segment and one horizontal channel segment) have to be laid out — the entire FPGA is created by replication of this basic tile.

The most straightforward way to create an FPGA with this architecture is to create one horizontal channel and one vertical channel, and replicate them across the array. Switches are then inserted between horizontal and vertical wire segments which the switch block and internal population parameters indicate should be connected. Figure 4.8 shows the results of such a technique, where only a few of the routing switches have been shown for clarity. Notice that this FPGA *does not* meet the specifications. By inserting routing switches at the ends of the horizontal segments, we are allowing connections into the middle of vertical segments. However, our specifications said that segments should have routing switches only at their ends. If we do not insert switches at the ends of the horizontal segments, however, we cannot connect to the ends of the horizontal segments, so the specifications are again violated. We call this problem a conflict between the *horizontal constraints* and the *vertical constraints.*



(a) Disjoint switch block

(b) Each channel contains 3 wires of length three.

FIGURE 4.7 Architecture specification: (a) disjoint switch block; (b) segmentation distribution.

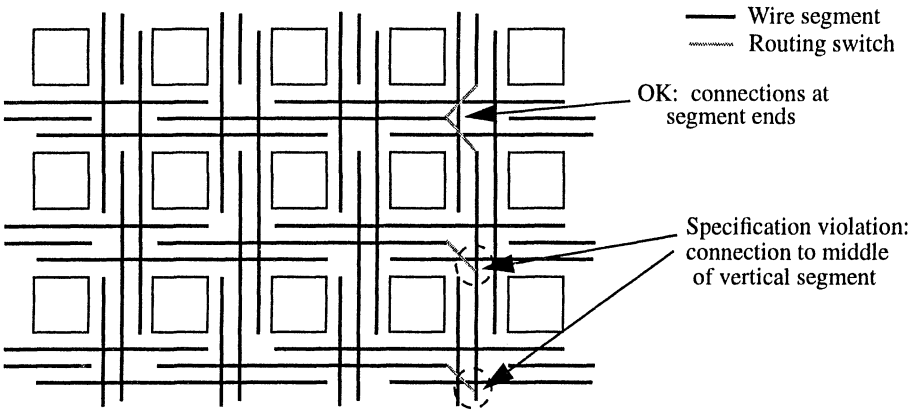


FIGURE 4.8 Replicating one channel causes the horizontal and vertical constraints to conflict.

The solution to this problem is shown in Figure 4.9. Instead of simply replicating a single channel, the "start points" of the segments in each channel have to be adjusted. As Figure 4.9 shows, this allows the horizontal and vertical constraints to be simultaneously satisfied. The specification for the FPGA has been completely realized — every segment connects to others only at its ends, and the switch block topology is disjoint. Figure 4.10 shows how one can implement this architecture using a single

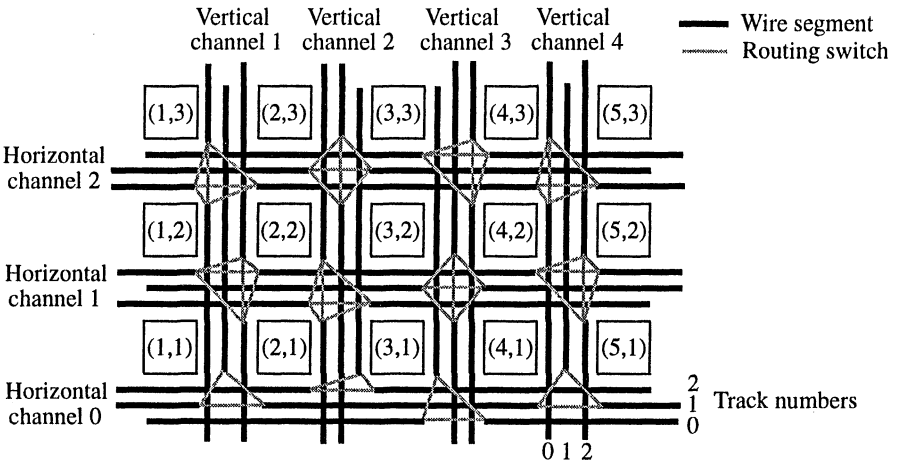


FIGURE 4.9 Adjusting the segment start points allows both the horizontal and vertical constraints to be satisfied. The FPGA coordinate system is also shown.

layout tile. This is an additional bonus of this "segment start point adjustment" technique -- we not only meet our specifications fully, but create an easily laid-out FPGA.

In order to describe the adjustment of the segment start points more clearly, let us define an FPGA coordinate system. Let the logic block in the lower left corner of the logic block array have coordinates (1,1). The logic block to its right has coordinates (2,1), and the logic block above it has coordinates (1,2), as Figure 4.9 shows. A horizontal channel has the same y-coordinate as the logic block below it, and a vertical channel has the same x-coordinate as the logic block to its left. We also number the tracks within each channel from 0 to 2, with track 0 being the bottommost track in a horizontal channel, or the leftmost track in a vertical channel.

The proper adjustment shifts the start point of each segment back by 1 logic block, relative to its start point in channel j, when constructing channel j+1. For example, in Figure 4.9, the left ends of the wire segments in track 0, horizontal channel 0 line up with the logic blocks that satisfy

$$(i+2)\ modulo\ 3 = 0, \tag{4.1}$$

where i is the horizontal (x) coordinate of a logic block. In channel 1, track 0, however, the left ends of the wire segments line up with logic blocks that satisfy:

$$(i+3)\ modulo\ 3 = 0 \tag{4.2}$$

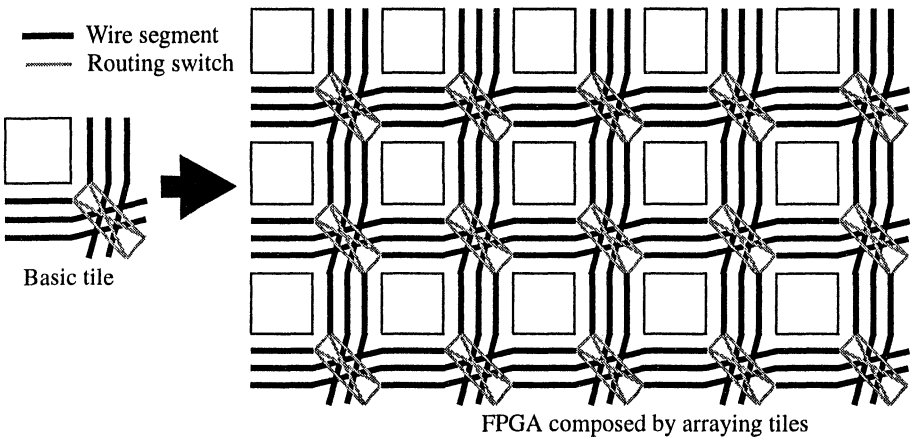


FIGURE 4.10 Tiled layout to implement FPGA of Figure 4.9.

A similar shifting back of start points must be performed in the vertical channels — the start point of segments in channel i+1 is moved back one logic block relative to their start point in channel i.

The shifting of segment start points above allows the horizontal and vertical constraints on an FPGA to be met if either of the following two conditions is met:

- The disjoint switch block topology is used. The segmentation distribution and segment internal populations can have any values. Or,
- All segments are fully switch-block populated. The segmentation distribution and switch block topology can have any values.

If either of these conditions is satisfied, the shifting of segment start points also makes a tile-based layout possible if one additional constraint is satisfied: the number of tracks of length L is divisible by L, for all segment lengths L.

We have not yet found a method to simultaneously satisfy the horizontal and vertical constraints when a switch block topology other than disjoint is used with internally-depopulated segments. It is an open question as to whether there is any method of satisfying both sets of constraints in this most general case. In cases where we cannot make the horizontal and vertical constraints agree, there are locations in the FPGA where a vertical wire wishes to connect to a horizontal wire, but the horizontal wire does not want a switch there, or vice versa. We resolve this conflict by inserting the switch, preferring to err on the side of too many switches in the routing, rather than too few.

The architecture generator currently contained in VPR is limited to generating architectures for detailed routing only when all the channels in the FPGA have the same width and segmentation distribution (VPR can generate routing graphs for global routing in which different channels have different capacities). Even with this restriction, we can explore a very large FPGA design space, and the architecture generation problem presents interesting challenges. A project is underway at the University of Toronto to extend the VPR architecture generator so that a user can specify different channel width and segmentation distributions for the horizontal and vertical directions [138]. This will allow architectural experimentation with Altera-like FPGAs.

We believe that the automatic generation of FPGA architectures to match a set of specifications is a fertile research area. Future research can investigate ways to ensure that the architectures created match all the specifications, and choose unspecified parameters intelligently enough that the best routability always results. Such research can explore not only better ways of generating architectures from the parameters we have listed in this section, but also whether a different architectural parameterization can be found such that all the specifications can always be satisfied.

4.3 Routability-Driven Router

Recall that VPR incorporates two different routers: one that is purely routability-driven, and one that is both routability and timing-driven. Both these routers can perform either combined global-detailed routing or global routing alone simply by changing the routing-resource graph passed to them. In this section we describe a purely routability-driven router, while the next section describes the timing-driven router. Once a routing is complete, VPR's graphics can be used to examine it — see Appendix A for sample pictures.

4.3.1 Cost Functions and Routing Schedules

Our routability-driven router is based on the Pathfinder negotiated congestion algorithm [85]; this purely routability-driven variant of the Pathfinder algorithm was described in Section 2.2.3. In the discussion below we will focus on new enhancements in our router and on important portions of the algorithm implementation that were not described in [85].

We define the cost of a node somewhat differently than [85] (see Equation (2.5)); the cost of using routing resource n when it is reached by connecting it to routing resource m is:

$$Cost(n) = b(n) \cdot h(n) \cdot p(n) + BendCost(n, m), \quad (4.3)$$

where the b(n), h(n) and p(n) are the base cost, historical congestion, and present congestion terms defined in Section 2.2.3. The BendCost(n,m) term is an enhancement we have made to improve the results of global routing. It penalizes bends when global routing is being performed, since global routes with many bends make it difficult or impossible for a subsequent detailed routing phase to utilize long wire segments [36, 31]. Hence reducing the number of bends in a global routing tends to lead to detailed routes that are both faster and require fewer tracks. If global routing only is being performed, BendCost(n,m) is 1 if making the connection from node m to node n implies a bend — i.e. node m is a horizontal channel segment and node n is a vertical channel segment or vice versa. Including this BendCost term in the total cost of using a node produces routes with very few unnecessary bends and does not significantly increase the global routing track count. If combined global-detailed routing is being performed there is no need to penalize bends, so BendCost(n,m) is always zero in this case.

Notice that the functional form of (4.3) is different than that of (2.5); we multiply b(n) and h(n) together rather than adding them. When adding terms in cost functions, it is

important to ensure they are properly normalized to roughly the same range of magnitude so that both terms have an effect. We avoid having to normalize h(n) to b(n) by converting the addition to a multiplication.

In [85], the base cost of a node, b(n) was set to its intrinsic delay. We have found that this is not the best choice; on average, about 10% fewer tracks per channel are required when the base costs of Table 4.1 are used instead. Note that the performance of the router is not extremely sensitive to the exact base costs chosen; the congestion avoidance terms in (4.3) ensure that the primary goal of the router is congestion avoidance, regardless of the b(n) values.

TABLE 4.1 Base costs of different types of routing resource.

Routing Resource, n	Base Cost, b(n)
Wire segment	1
Logic block output pin	1
Logic block input pin	0.95
Source	1
Sink	0

Four of the five b(n) values in Table 4.1 have virtually the same value — this encourages the router to use as few of these resources as possible to route each connection. The b(n) value for an logic block input pin and for a sink are set to less than 1 to save CPU time. Since the maze expansion used to route a connection terminates when it reaches a sink corresponding to one of the net terminals, some CPU savings can be obtained by costing resources so that sinks tend to be reached earlier in the maze expansion. In other words, we would like to cost logic block input pins and sinks so that the maze expansion checks if the logic block next to a wire segment contains one of the net sinks for which we are searching before it expands more wire segments. To achieve this behaviour, a logic block input pin has a base cost of slightly less than 1, and a sink has a base cost of zero. Since congestion can not occur at sinks, using a base cost of zero for them will not cause route failures. Using a lower cost for logic block inputs and sinks in this way speeds the routability-driven router up by 1.5 to 2 times, depending on the FPGA architecture and circuit being routed.

We experimented with several different possibilities for the base costs of wire segments of different lengths: intrinsic delay, 1, length, the square root of length, and length+1. For both the routability-driven router described in this section, and the timing-driven router of the next section, setting b(n) to 1 regardless of a wire segment's length yields the best results. Using the intrinsic delay of each routing resource as its

b(n) value increased the number of tracks required per channel by 10% on average. The other choices of b(n) led to performance between these extremes. Setting the cost of a wire segment to 1 regardless of its length also led to the highest speed circuits for the routability-driven router, since it encourages connections to use the smallest number of routing resources possible.

Ebeling et al [85] did not describe the exact functional form of the h(n) and p(n) congestion avoidance terms in (4.3), so we list the equations we use below. The present congestion penalty is updated whenever any net is ripped-up and re-routed according to

$$p(n) = 1 + max(0, [occupancy(n) + 1 - capacity(n)] \cdot p_{fac}), \quad \textbf{(4.4)}$$

where occupancy(n) is the number of nets currently using routing resource n, and capacity(n) is the maximum number of nets that can legally use node n. The historical congestion penalty is updated only after an entire routing iteration. Its value during routing iteration i is:

$$h(n)^i = \begin{cases} 1, & i = 1 \\ h(n)^{i-1} + max(0, [occupancy(n) - capacity(n)] \cdot h_{fac}), & i > 1 \end{cases} \quad \textbf{(4.5)}$$

The values of h_{fac} and p_{fac} in each routing iteration define what we call the *routing schedule* [2]. We have found that it is sufficient to keep h_{fac} constant for all routing iterations; the fact that h(n) is incremented after every iteration in which node n is overused provides sufficient increase in the historical congestion penalty. Any value of h_{fac} between 0.2 and 1 works equally well. To achieve the absolute highest quality results, p_{fac} should initially be small, allowing congestion with little penalty, and gradually increase from iteration to iteration. For these highest quality results, p_{fac} should be 0.5 or less in the first iteration, and 1.5 to 2 times its previous value in each subsequent iteration. When the cost of congestion is increased this gradually, however, it takes several routing iterations (typically 5 - 10) to route even fairly "easy" routing problems, where there is more routing available than is needed by the circuit. For such "easy" problems, the router can be sped up by a factor of two to three times by making congestion very expensive immediately — in this case, we make p_{fac} 10000 in the first routing iteration, ensuring the router will always avoid congestion if it can, even in the first routing iteration. The amount of quality sacrificed by making congestion expensive immediately is quite small — typically the minimum achievable track count increases by only 2% to 4%. Throughout this book we will use the slowly increasing congestion cost schedule to achieve the absolute highest quality, however.

If a circuit has not routed in 30 routing iterations we normally classify it as unroutable. Allowing the router to try 45 routing iterations before giving up reduces the minimum track count by only 1% to 4%, on average.

4.3.2 Speed Enhancements

In addition to tuning the routing schedule and optimizing the routing-resource base costs, we made two other speed enhancements to the Pathfinder negotiated-congestion algorithm. First, we do not allow net routings to go more than three channels outside of their net bounding box. Since our routability-driven router uses a breadth-first maze router to make connections, this enhancement significantly reduces CPU time. Restricting each route to remain within 3 channels of its bounding box had no noticeable effect on the quality of the routing.

The second speed enhancement aids the routing of high-fanout nets, and results in an order-of-magnitude router speedup on large circuits. Recall that to route a k-terminal net, the maze router contained in VPR is invoked k-1 times. In the first invocation, the maze routing wavefront expands out from the net source until it reaches any one of the k-1 net sinks. The path from source to sink is now the first part of this net's routing. In a traditional maze router, and in the Pathfinder algorithm, the maze expansion (the PriorityQueue in Figure 2.10) is emptied, and a new wavefront expansion is started from the entire net routing found thus far. After k-1 invocations of the maze router all k terminals of the net will be connected.

This approach requires considerable CPU time for high-fanout nets. High-fanout nets usually span most or all of the FPGA. Therefore, in the latter invocations of the maze router the partial routing used as the net source will be very large, and it will take a long time to expand the maze router wavefront out to the next sink. We have developed a more efficient method. When a net sink is reached, we add all the routing resource segments required to connect the sink and the current partial routing to the wavefront (i.e. the PriorityQueue) with a cost of 0. We do not empty the current maze routing wavefront, but instead continue expanding normally. Since the new path added to the partial routing has a cost of zero, the maze router will expand around it first. Since this new path is typically fairly small, it will take relatively little time to add this new wavefront, and the next sink will be reached much more quickly than if the entire wavefront expansion had been restarted. Figure 4.11 illustrates the difference graphically. The shortest path computed to most of the nodes in the wavefront is still valid after a net sink is reached; we are taking advantage of this by restarting only a local portion of the wavefront instead of restarting the entire wavefront.

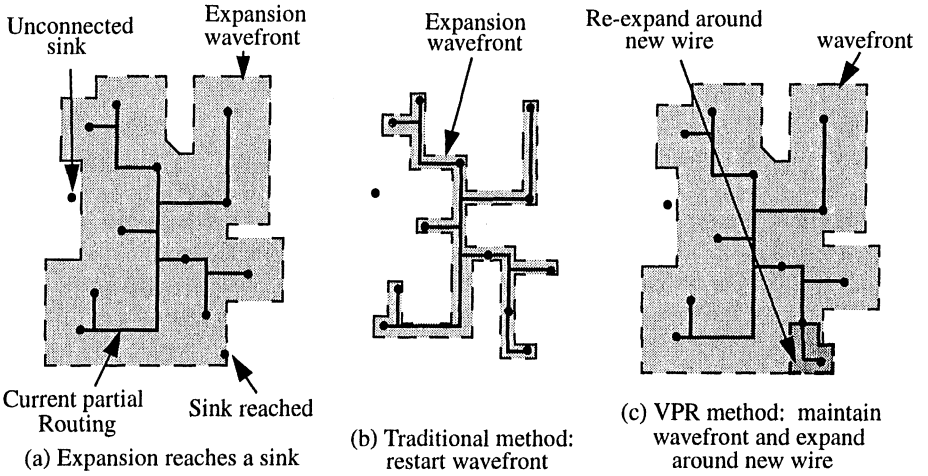


FIGURE 4.11 Difference between (b) complete and (c) local wavefront restart.

4.4 Timing-Driven Router

The timing-driven router developed for this work uses many of the ideas and algorithms of the Pathfinder negotiated congestion-delay router [85], which is described in Section 2.2.3. Our router contains two major improvements relative to the Pathfinder router, however: we optimize delay under the Elmore, rather than linear, delay model, and we dynamically vary the base cost of routing resources.

4.4.1 Superiority of Elmore Delay to the Linear Delay Model

Recall (as discussed in Section 2.2.3) that current academic FPGA combined global-detailed routers are either purely routability-driven or optimize only the linear delay model, in which every routing resource has a constant delay. The linear delay model is highly inaccurate for any FPGA which contains pass transistors in its routing, as the delay of a pass transistor strongly depends on the topology of the net routing into which it is inserted. For example, consider the two routing topologies in Figure 4.12, and the equivalent RC-circuit for each, as given in Section 2.2.4. Both the linear delay and Elmore delay models accurately predict the delay of a routing formed by a series of M wire segments connected by buffers:

$$T_{d,\,buffered} = M \cdot [T_{buf,\,intrinsic} + R_{buf}C_{total}] = M \cdot 250 \text{ (ps)} \qquad \textbf{(4.6)}$$

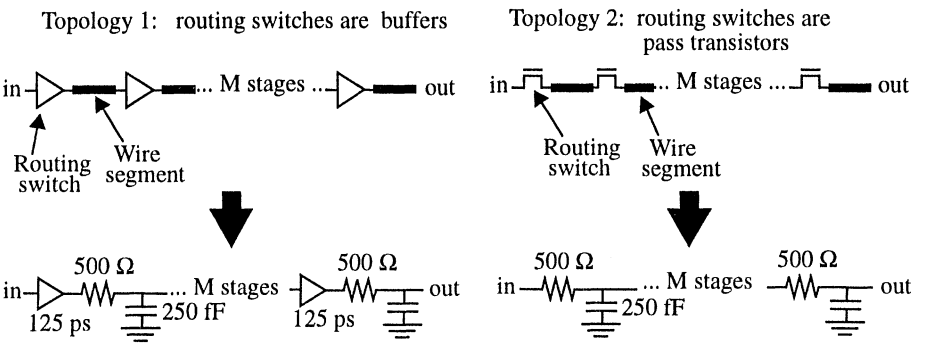


FIGURE 4.12 Two routing topologies and their equivalent circuits.

where the values of R_{buf}, $T_{buf,intrinsic}$, and C_{total} are taken from Figure 4.12. Note that C_{total} includes both the metal capacitance of a routing wire and the parasitic capacitance of the routing transistors attached to it.

The linear and Elmore delay models predict vastly different delays for a chain of pass transistors, however:

$$T_{d,\,pass,\,linear} = M \cdot R_{pass} \cdot C_{total} = M \cdot 125 \quad \text{(ps)} \tag{4.7}$$

$$T_{d,\,pass,\,Elmore} = \sum_{i=1}^{M} i \cdot R_{pass} C_{total} = M\frac{(M+1)}{2} R_{pass} C_{total} = M\frac{(M+1)}{2} \cdot 125 \text{ (ps)} \tag{4.8}$$

Notice that the Elmore delay accurately models the fact that the delay of a chain of pass transistors grows quadratically with the length of the chain, while the linear delay model does not. Setting equations (4.8) and (4.6) to be equal, we find that the chain of buffers is faster than the chain of pass transistors (for the R and C values we have assumed) whenever the number of routing switches in series, M, is more than 3. The linear delay model predicts that a chain of pass transistors is *always* faster than a chain of buffers, regardless of the length of the chain, M. It is obvious then, that using the linear delay model in an FPGA router will cause the router to make very poor decisions about when to use buffered routing switches and when to use pass transistor routing switches.

The second problem with the linear delay model is that it does not properly model the effects of fanout on net routings using pass transistors. Consider the two net topologies shown in Figure 4.13. Under the linear delay model, both topologies have the same delay to sink 1:

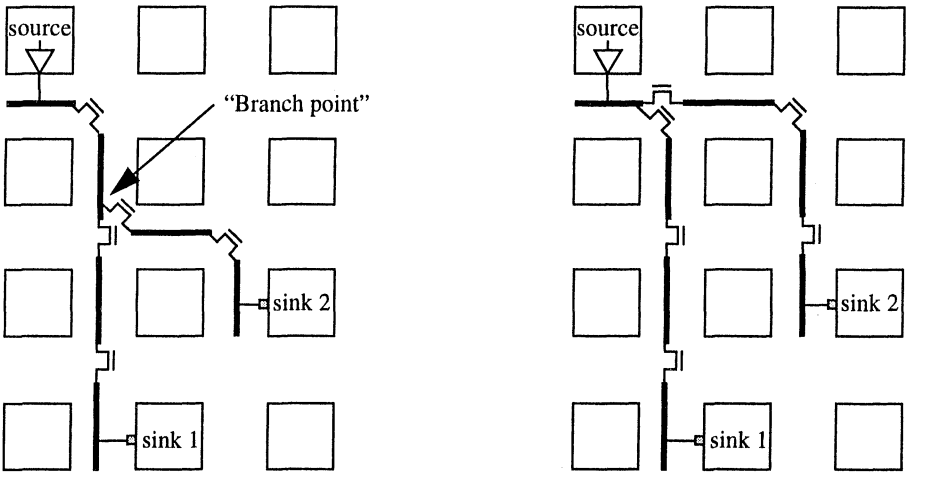


FIGURE 4.13 Two net topologies routed using pass transistor routing switches.

$$T_{linear} = T_{obuf} + R_{obuf}C_{total} + 3R_{pass}C_{total} \tag{4.9}$$

where we assume each wire has a total capacitance (metal plus parasitic switch capacitances) of C_{total}; T_{obuf} and R_{obuf} denote the intrinsic delay and equivalent resistance of the logic block output pin buffer; and R_{pass} is the equivalent resistance of each pass transistor. Since both topologies have the same delay under the linear delay model, a router using this delay model would always pick the left-hand topology as superior, since it uses fewer routing resources.

Under the Elmore delay, however, the two topologies have different delays:

$$T_{Elmore,\,left} = T_{obuf} + 6R_{obuf}C_{total} + 8R_{pass}C_{total} \tag{4.10}$$

$$T_{Elmore,\,right} = T_{obuf} + 7R_{obuf}C_{total} + 6R_{pass}C_{total}. \tag{4.11}$$

If the resistance of the logic block output buffer is less than twice the resistance of a routing pass transistor (the typical case), the topology on the right has lower delay to sink 1. If sink 1 is time-critical therefore, the topology on the right is superior even though it uses more wirelength.

A related problem with the linear delay model is that it does not model the beneficial effect of isolating large capacitive loads with buffers. For example, if a buffered rout-

ing switch was used instead of a pass transistor at the "branch point" in the left routing topology of Figure 4.13, the delay to sink 1 drops because the capacitance in the branch of the routing leading to sink 2 is now isolated. This capacitance therefore does not load the output pin buffer or the pass transistor leading back to the net source. The Elmore delay correctly models this effect, but the linear delay model is completely oblivious to it.

It is clear then, that if an FPGA contains pass transistors, the linear delay model does not correctly rank routing alternatives. As mentioned in Section 2.2.4, however, the Elmore delay is much more accurate, and it has high "fidelity" [129, 130], meaning that it has a high probability of correctly ranking the routing alternatives.

4.4.2 Directly Optimizing the Elmore Delay

For all the reasons listed above, then, we would like our router to directly optimize the Elmore delay. Figure 4.14 shows the pseudo-code for this router. Like the Pathfinder algorithm described in Section 2.2.3, we perform routing iterations until there are no congested resources, where a routing iteration consists of ripping-up and re-routing every net in the FPGA. The manner in which we initially assume each connection is critical and route it for minimum delay, and gradually increase the cost of congestion, is also the same as in Pathfinder. The use of the Elmore delay, rather than the linear delay, leads to a significantly different algorithm to route each net, however. We focus on these new aspects of the router below.

Consider the routing of a single net, i, as described in the pseudo-code of Figure 4.14. Refer to Section 2.2.5 for a review of the nomenclature relevant to timing analysis and connection slacks. Initially the routing tree for net i, RT(i), is just the routing-resource node corresponding to the net source. A wave expansion algorithm is invoked k-1 times to connect RT(i) to each of the net's k-1 sinks, with the most critical sink being connected first, and the least-critical sink being connected last. Consider the connection to sink j of net i. The cost to include a node, n, in a net's routing is:

$$Cost(n) = Crit(i, j) \cdot delay_{Elmore}(n, \text{topology}) + [1 - Crit(i, j)] \cdot b(n) \cdot h(n) \cdot p(n) \quad \textbf{(4.12)}$$

We define the criticality of a connection, crit(i,j) as:

$$Crit(i, j) = max\left(\left[MaxCrit - \frac{slack(i, j)}{D_{max}}\right]^{\eta}, 0\right) \quad \textbf{(4.13)}$$

Let: **RT(i)** be the set of nodes, n, in the current routing of net(i); RT(i) also stores the $T_{Elmore}(n)$ and $R_{upstream}(n)$ for each of these nodes.
PriorityQueue be a set of {TotalCost(n), PathCost(n), $R_{upstream}(n)$} tuples for each node, n, in the current wave expansion, sorted on TotalCost.
StoredPathCost and **StoredTotalCost** be arrays with one entry per routing resource node, with all entries initialized to a huge number.

```
Crit(i,j) = MaxCrit for all nets i and sinks j; /* First iteration routes all for high speed */
while (overused resources exist) {    /* Illegal routing? */
   for (each net, i) {
      DynamicallyUpdateBaseCosts(i);  /* Set base costs based on properties of net i */
      rip-up routing tree RT(i) and update affected p(n) values;
      RT(i) = NetSource(i);

      for (each sink, j, of net(i) in decreasing crit(i,j) order) {
         PriorityQueue = RT(i) at {TotalCost(n) = PathCost(n) + α·ExpectedCost(n),
               PathCost(n) = crit(i,j)·delay_Elmore(n), R_upstream(n)} for each
               node n in RT(i);
         while (sink(i,j) not found) {   /* Wave expansion */
            Remove lowest TotalCost node, m, from PriorityQueue;
            if (TotalCost(m) < StoredTotalCost(m) && PathCost(m) <
                  StoredPathCost(m) {   /* Best path found to this node? */
               StoredPathCost(m) = PathCost(m);
               StoredTotalCost(m) = TotalCost(m);
               for (all fanout nodes n of node m) {  /* Expand node n */
                  Compute R_upstream(m) from R_upstream(n) via (4.14)
                  Add n to PriorityQueue at {TotalCost(m) = PathCost(m) +
                     α·ExpectedCost(m), PathCost(m) = Cost(m) + PathCost(n),
                     R_upstream(m)};
               }
            }
         }  /* End wave expansion */
         for (all nodes, n, in path from RT(i) to sink(i,j)) {  /* Backtrace */
            Update p(n);
            Add n to RT(i) and store R_upstream(n);
         }
         Update Elmore delay of RT(i);
         For (all nodes n expanded during wave expansion) {
            StoredPathCost(n) = HugeNumber;
            StoredTotalCost(n) = HugeNumber;
         }
      }
   }
   Update h(n) for all n;
   Perform timing analysis and update Crit(i,j) for all nets i and sinks j;
} /* End of one routing iteration */
```

FIGURE 4.14 Pseudo-code of the timing-driven routing algorithm.

where D_{max} is the delay of the circuit critical path, and slack(i,j) is the slack of the connection between the source and sink j of net i. η and MaxCrit are parameters that control how a connection's slack impacts the congestion-delay trade-off in cost function (4.12).

Note that the criticality equation used by Pathfinder is a special case of (4.13) with η and MaxCrit both equal to 1. When η is larger, nets with positive slack pay less attention to delay and more to congestion. $MaxCrit^{\eta}$ is the maximum criticality any connection can have. Setting MaxCrit to 1 allows connections with 0 slack (i.e. connections on the critical path) to completely ignore congestion, while a MaxCrit of 0 forces all connections to ignore delay and makes the router completely routability-driven. We have experimentally determined that setting η to 1 is the best value, but setting MaxCrit to 0.99 results in better routability than setting it to 1, with no impact on circuit delay. Setting MaxCrit to 1 is dangerous, since nets on the critical path then pay no attention to congestion. Consequently, if two nets on the critical path are sharing the same routing-resource node the congestion may never be resolved if MaxCrit is 1.

Notice that the $delay_{Elmore}$ factor in (4.12) is a function of both the node, n, being expanded and the topology used to reach it. In order to calculate the Elmore delay increase resulting from adding a node to a partial routing, we need to know the *upstream resistance*, $R_{upstream}$, through which this node's capacitance must be discharged. The upstream resistance of a node, *n*, that was reached by connecting it to node *m* via a routing switch, *switch*, is

$$R_{upstream}(n) = \begin{cases} R(switch) + R_{metal}(n), & \text{if switch is a tri-state buffer} \\ R_{upstream}(m) + R(switch) + R_{metal}(n), & \text{if a pass transistor} \end{cases} \quad \textbf{(4.14)}$$

Notice that the upstream resistance accumulates through a chain of pass transistor switches, but not through a chain of buffered routing switches. $R_{metal}(n)$ is the metal resistance of this routing-resource node, and is 0 for all nodes that are not wiring segments. The Elmore delay of node n in this topology is then:

$$T_{Elmore}(n, \text{topology}) = T_{d,\,intrinsic}(switch) + \left[R_{upstream}(n) - \frac{R_{metal}(n)}{2}\right] C_{total}(n) \quad \textbf{(4.15)}$$

where $C_{total}(n)$ gives the total capacitance (metal plus parasitic switch capacitance) at node n. We subtract one-half of $R_{metal}(n)$ from $R_{upstream}(n)$ in (4.15) to account for the fact that the metal resistance is distributed. Note that in order to apply (4.14) and (4.15) we need to know the $R_{upstream}$ value for the node via which node n was

reached; this implies we must store the $R_{upstream}$ value associated with every node currently in the PriorityQueue.

We define PathCost(n) to be the total cost of the path from the current partial routing tree to node n:

$$PathCost(n) = \sum_{l \in \text{ path from RT(i) to n}} Cost(l) \quad \textbf{(4.16)}$$

Using the PathCost of each node as the value on which we sort in the PriorityQueue results in a breadth-first search through the routing graph to route each connection. Note that the Crit(i,j) values change for each sink, j, and also that the Elmore delay values of the partial routing tree RT(i) must be updated after each sink is reached. Consequently, after a sink is reached the PathCost values stored in the PriorityQueue become invalid for every node. Hence, we cannot use our optimized breadth-first search from Section 4.3.2; the PriorityQueue must be emptied after each sink is reached, and the entire search begun from scratch. Since this would result in a very slow algorithm, and since Swartz et al have recently shown that a directed-search routing algorithm is faster than even our optimized breadth-first search [127], we employ a directed-search technique [105] instead.

To perform a directed-search, we must be able to estimate the total remaining cost, ExpectedCost(n), from the current node to the target sink node. For source nodes and logic block output pin nodes we set ExpectedCost(n) to zero. Since there is only one source node and at most a few output pin nodes explored during the maze expansion, this has little effect on CPU time. As well, it ensures that the router considers the option of connecting a logic block output pin to multiple tracks to optimize the routing of high-fanout nets.

To compute the ExpectedCost to route from a wire segment, n, to the target sink, j, we assume that:

- The connection to sink j will be completed using wires of the same type (length, connecting switch type, etc.) as node n, and
- There is no congestion along the shortest path to the target. That is, p(m) and h(m) are both 1 for all the nodes, m, that will be used to route the remainder of the connection.

We know the coordinates of both sink j and the current node, n, within the FPGA, so we can compute how many wires of the same length as node n will be needed to reach sink j. With this information and knowing the type of switch used to connect one wire of this type to another, we can compute the expected Elmore delay to the target. This is all the information we need to use (4.12) to compute the ExpectedCost to the target.

We then perform a directed search by using the TotalCost of a node n as the sort value in the PriorityQueue:

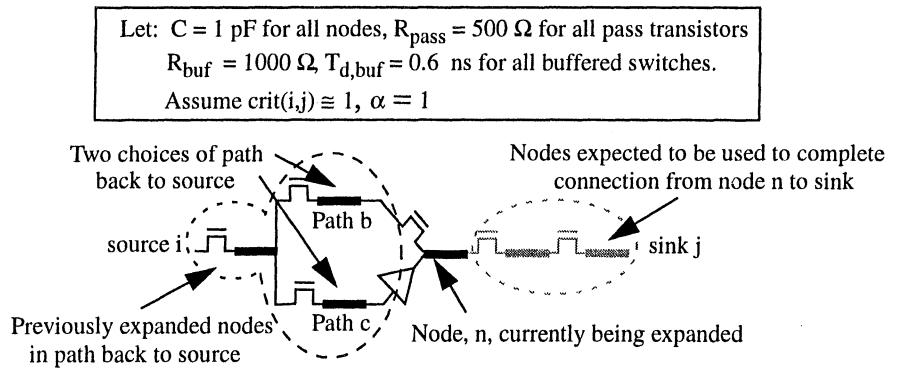
$$TotalCost(n) = PathCost(n) + \alpha \cdot ExpectedCost(n, j) . \quad (4.17)$$

The α parameter in (4.17) controls the aggressiveness of the directed search. An α of 0 leads to a breadth-first search, while an α of 1 leads to an A* search if ExpectedCost(n,j) is a lower bound. We have found that an α value of 1.2 leads to the best CPU time without compromising quality; any value of α between 1 and 1.4 performs fairly well, however.

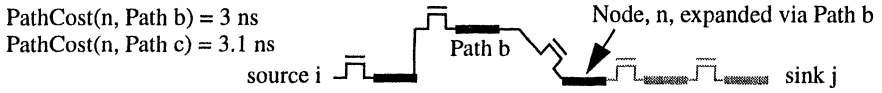
By choosing nodes, n, that minimize the TotalCost(n) given by (4.17), we do *more* than just change the graph search method from breadth-first to directed; we also allow the router to perform more intelligent *buffer insertion.* To decide whether or not using a buffered switch is a net benefit at some point in the wave expansion, we need to know not only the discharge resistance back toward the net source, $R_{upstream}$, but also the total expected downstream capacitance needed to go from this point to the target sink. Equation (4.17) makes an approximation of this downstream capacitance visible through the ExpectedCost from node n to the target sink, and therefore leads to more intelligent decisions as to what type of switch to use. Figure 4.15 illustrates a decision regarding whether to use a buffered switch or a pass transistor at some point in a wave expansion. If we choose the next node to expand in order to minimize the PathCost, given by (4.16), back to the net source, we create the net routing shown in Figure 4.15(b). If we instead choose the next node to expand in order to minimize the TotalCost, (4.17), to the target sink, we create the net routing shown in Figure 4.15(c). The routing tree of Figure 4.15(c) has lower delay to the sink and is therefore the correct choice.

Note that in most FPGA architectures it is not possible to separate the issue of buffer insertion from that of routing tree topology construction. Usually, once a routing topology is chosen for a net (to minimize wirelength, avoid congestion, use wires of the appropriate length, etc.) there is little or no choice remaining about which switches to use to connect the nodes in this routing tree. Consequently, we believe algorithms which simultaneously balance congestion avoidance, topology and wire length selection and buffering issues in one step are crucial in FPGA routing, and this is the approach we have taken in our timing-driven router.

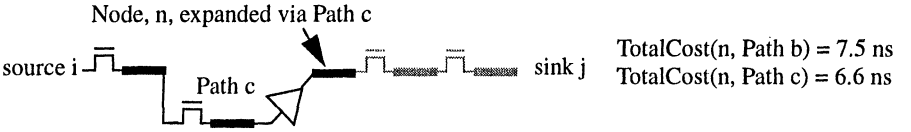
ExpectedCost(n) is not a lower bound on the cost needed to reach a sink from node n in some FPGA architectures, and this introduces one further requirement into the maze expansion. A node, n, can be included along the same path twice and result in a lower TotalCost than would result if it were included in the routing path once. Figure 4.16 shows an example of this phenomenon. To avoid the creation of electri-



(a) Two alternative paths reach node n during wave expansion.



(b) If PathCost is used to rank (sort) alternatives, Path b is chosen. Sink delay = 7.5 ns.



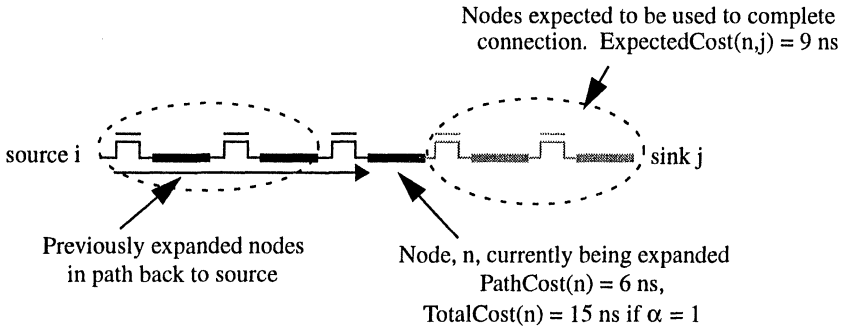
(c) If TotalCost is used to rank (sort) alternatives, Path c is chosen. Sink delay = 6.6 ns.

FIGURE 4.15 Wave expansion showing why ranking alternatives by TotalCost is necessary.

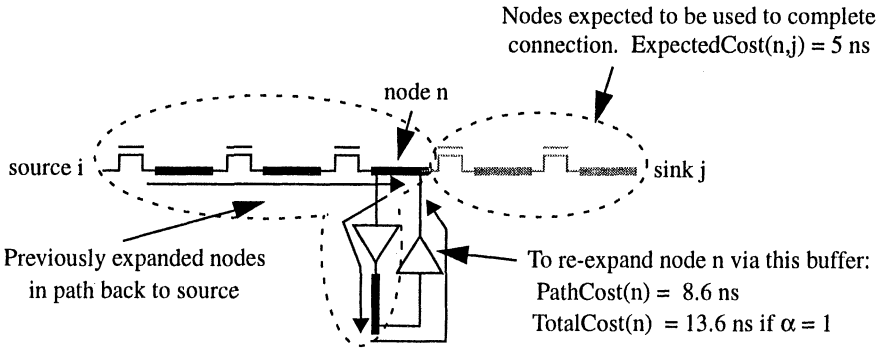
cally-illegal loops in a net's routing, we record both the TotalCost and PathCost at a node when it is expanded (these values are called StoredTotalCost(n) and StoredPathCost(n) in Figure 4.14), and we only allow a node to be re-expanded (re-inserted in the PriorityQueue) only if both the new PathCost and the new TotalCost are lower than the old values. In graph theoretic terms, the TotalCost defined by (4.17) is not a monotone path function, and maze routing algorithms are designed to operate only on monotone path functions [105]. Consequently, we need to augment the basic maze router algorithm to check for loops when using (4.17) as the cost of a path ending at a node, n.

Notice that in order to do a wave expansion that optimizes the Elmore delay, we must store three values for every entry, n, in the PriorityQueue: {TotalCost(n), PathCost(n), $R_{upstream}$(n)}. The queue is sorted based on the value of TotalCost(n), and

Let R = 1000 Ω for all switches, C = 1 pF for all nodes,
and $T_{d,buf}$ = 0.3 ns for all buffered switches. Assume crit(i,j) ≅ 1, α = 1



(a) Expanding node n the first time leads to a TotalCost of 16.8.



(b) Expanding node n a second time leads to a lower TotalCost(n), but creates an electrically-illegal loop.

FIGURE 4.16 Adding a loop to a potential route can reduce the TotalCost of a node.

PathCost(n) and $R_{upstream}(n)$ are used only to expand a node n when it is removed from the PriorityQueue.

Once the target net sink, j, is reached, the PriorityQueue is emptied and a backtrace is performed to add the new connection to the routing tree, RT(i). This new connection can add extra load capacitance to the rest of the routing tree, and hence affect the Elmore delay of other parts of the tree. We compute the load capacitance not isolated by buffers introduced by the new connection, and propagate it up the routing tree,

RT(i), until we encounter a buffer isolating us from the rest of the tree. The Elmore delay of the entire subtree rooted at this isolating buffer is then updated.

If there is another sink, j′, to connect, every node r in RT(i) is then placed back on the PriorityQueue with a PathCost of

$$PathCost(r) = crit(i, j') \cdot delay_{Elmore}(r, \text{topology}), \tag{4.18}$$

a TotalCost(r) given by (4.17), and the $R_{upstream}(r)$ value stored for each node in the routing tree. The directed search procedure is then started again to connect this routing tree to sink j′.

4.4.3 Net Routing Algorithm Complexity

The net routing algorithm above has the same complexity to route a k-terminal net as a conventional maze router, since we can evaluate both the cost of a node and the ExpectedCost from a node to the target sink in constant time. Routing a k-terminal net is an $O(k^2 \log k)$ operation if the FPGA is uncongested and the directed search is able to proceed straight to each target sink without backtracking. If the FPGA is highly congested, the directed search may have to search the entire routing graph before finding each sink in the worst case. In this case, the complexity of both the net routing algorithm above and a conventional maze router is $O(k \cdot R \log R)$, where R is the number of nodes in the routing-resource graph. This worst case complexity almost never occurs, however, even for "high-stress" routing in which there is considerable congestion. The typical complexity of the router is $O(k^2 \log k)$.

4.4.4 Dynamic Base Costs

One problem with signal routing algorithms based on any form of maze router is that they route a k-terminal net one connection, or sink, at a time. The best choice of wires to use in connecting to a sink depends on the other sinks to be connected, but a maze router does not see such overall issues. In [86], Nag and Rutenbar changed the cost of long wire segments as a function of the span of the net being routed, in order to bias the router to use the proper type of routing resources for each net. In a similar manner, we dynamically change the base cost of wire segments that connect via pass transistors, as a function of the net fanout.

Wire segments connected by pass transistors are not as efficient at routing high-fanout nets as wire segments connected by buffered routing switches. A routing tree connected entirely with pass transistors has no buffers isolating the capacitance of portions of the tree from other parts, so the switches tend to become heavily loaded,

resulting in a large delay, as the tree becomes large. By updating the Elmore delay of the routing tree, RT(i), after each sink is reached, we make the router aware of some of these issues. In subsequent connections it will notice that portions of the routing tree are becoming slow, and will tend to avoid starting time-critical connections from these portions of the routing tree. Consequently, either the net's routing will become more "star-like," with many connections sharing relatively little wiring, or many of the subsequent connections will be made using wires that can be connected with buffers. Routing a high-fanout net as a star wastes considerable wiring, however. Similarly, routing the first several sinks of a net on wires that connect with pass transistors before beginning to use more wires connected with buffers also wastes wiring. A second problem is that if the router does not use a star-like topology because many of the net sinks are not delay-critical, it may slow down timing-critical connections made earlier for this net by adding capacitive loading to the routing tree in undesirable places. It would be better if the routing of a high-fanout net preferred to use buffered switches wherever possible, in order to maximize the possibility of re-using some of this wiring in later connections, and to minimize the possibility of later connections affecting the timing of the early connections.

The default base costs used in our timing-driven router are the same as those given in Table 4.1, except they are all normalized to the average delay of a routing resource so that both terms in (4.12) have roughly the same magnitude. We ran experiments to evaluate the effect of different base costs of wire segments as follows: we set the base cost of each wire to be proportional to its intrinsic delay, its length, the square root of its length, its length+1, and 1. Just as for the routability-driven router, we found that a using the same base cost for all wire segments (in this case, the average delay of a routing resource), regardless of length, was best.

Before we begin the routing of a net, however, the DynamicallyUpdateBaseCosts routine of Figure 4.14 is called to alter these default base costs as a function of the net fanout. The base costs of all wire segments that use pass transistors to connect to other wires of the same type are increased as a function of the net fanout:

$$b_{\text{pass transistor wires}}(n) = AverageRoutingResourceDelay \cdot \sqrt{k-1} \qquad \textbf{(4.19)}$$

where k is the number of terminals on the net. Wire segments that connect to other wires via buffers have their base cost left as the average delay of a routing resource. By increasing the base cost of certain nodes this way, we ensure that nets with high fanout prefer to use buffered routing resources, allowing the creation of buffered "spines" early in the net construction. Note that connections that are very delay critical will still use pass transistor based routing if it is the fastest, since such connections pay little attention to base cost. Similarly, if the buffered routing resources become congested, the congestion avoidance terms in (4.12) will eventually force the use of

unbuffered routing resources even for high-fanout nets. We tried using (k-1) instead of the square root of k-1 in Equation (4.19), but this made pass transistor based routing too expensive, and led to slightly inferior results.

The routing-resource graph is very large, so scanning through the entire graph changing b(n) for each node n before routing each net would be prohibitively slow. Instead, at each node, n, we store a pointer to a base cost to be used for a routing resource of n's type. Consequently, to change the base costs dynamically the number of values we must update is equal to the number of different routing-resource types in the FPGA. Since FPGAs tend to contain only a few different lengths of wire, and they are interconnected using only a few different types of switches, updating the base costs this way does not consume a significant amount of CPU time.

Table 4.2 shows the performance of our timing-driven router with and without this dynamic costing enhancement for 15 large MCNC circuits. The FPGA architecture in this experiment contains five different wire types, in equal proportions: lengths 1, 2, 4, 8 and long lines. The length 1 and 2 lines use pass transistor switches to connect to each other, while the other wires use buffered routing switches. Each logic block is a logic cluster containing 4 BLEs and with 10 inputs. Finally, each circuit is routed in an FPGA with 20% more tracks per channel than the absolute minimum required for successful routing. On 9 of the 15 circuits, dynamic costing improved the circuit speed; on 3 circuits dynamic costing slowed the circuits slightly, and on the remaining 3 there was no change. On average, the circuits sped up by 4%. This speedup is more significant than one might first assume. Without dynamic costing, the circuit delay is 5% higher than the minimum the router can achieve while ignoring congestion and routing every connection for minimum delay. With dynamic costing, the circuit delay is only 1% higher than this congestion-oblivious delay. Note also that the CPU time to route the circuits is 33% higher without dynamic costing than with it. By minimizing the construction of "stars" for high-fanout nets, dynamic costing results in shorter wave expansions and saves CPU time.

4.4.5 Routing Schedule

We found that the best routing schedule for the timing-driven router is very similar to the best schedule for the routability-driven router, which was described in Section 4.3.1. We set h_{fac} to 1 for all routing iterations, p_{fac} to 0.5 for the first two routing iterations, and to twice its previous value for all subsequent iterations. Making the cost of congestion very high even in the first few iterations sacrifices considerable result quality, often increasing the routed critical path delay by 33%. As well, making congestion expensive immediately does not consistently save CPU time for the timing-driven router. While making congestion very expensive early reduces the number of router iterations, it tends to make the directed search detour around congestion

TABLE 4.2 Effectiveness of dynamic base costs for routing resources.

Circuit	# 4-LUTs	With Dynamic Costing		Without Dynamic Costing		Delay (ns) Ignoring Congestion[a]
		Delay (ns)	CPU (s)	Delay (ns)	CPU (s)	
alu4	1522	35.2	23	35.7	30	34.1
apex2	1878	40.7	38	42.3	38	40.7
apex4	1262	35.0	22	34.6	28	35.0
bigkey	1707	22.2	19	22.2	26	22.2
des	1591	35.6	25	35.6	37	34.3
diffeq	1497	39.9	14	40.5	20	39.8
dsip	1370	23.0	18	23.8	33	22.7
elliptic	3604	65.8	56	66.9	81	65.3
ex5p	1064	35.1	19	36.2	27	35.1
frisc	3556	76.2	74	74.8	100	73.9
misex3	1397	31.2	22	31.2	32	31.5
s298	1931	59.7	26	62.0	37	59.3
seq	1750	36.9	31	35.5	43	36.0
spla	3690	53.7	138	72.0	171	54.9
tseng	1047	38.8	8	39.7	10	38.2
Arith. Av:	1924	41.9	36	43.5	48	41.5
Geom. Av:	1764	39.6	27	40.7	37	39.2

a. Recall that the there is no known polynomial-time algorithm to route a net with guaranteed minimum Elmore delay [129], so the congestion-oblivious routing delay is *not* a lower bound on the achievable delay.

more in the early iterations. The net effect is sometimes a reduction in CPU time, and sometimes an increase.

For most FPGA architectures 30 router iterations are sufficient to route even difficult problems. For FPGAs composed of very large logic blocks with many pins, however, we found that the timing-driven router sometimes needed 60 iterations to route difficult problems.

4.5 Delay Extraction and Timing Analysis

VPR contains a delay extractor that computes the Elmore delay of any routed net. To make delay extraction faster, after the routing-resource graph is built we traverse the graph once and lump all the parasitic switch capacitance, plus the metal capacitance, into a total capacitance value, C_{total}, at each node. Every node in the routing-resource graph can have a different C_{total}, and a different distributed metal resistance, R_{metal}. Similarly, every switch in the FPGA can have a different R_{switch} and intrinsic delay. We compute the Elmore delay to every routing-resource node in a net in linear time via two depth-first traversals of the tree describing the net's routing. We can also incrementally update the Elmore delay of a tree as new connections are added to the tree.

VPR also includes a full path-based timing analyzer (timing analysis was described in Section 2.2.5) that can determine both the critical path of a circuit and the slack of every connection to be routed. This timing analyzer will work with *any* logic block. The architecture description file lists the delay of every combinational and sequential element in a logic block, and the circuit netlist can specify a different interconnection pattern between logic block inputs, outputs and these internal combinational and sequential elements for each logic block in the circuit. After the netlist is read, a graph describing the circuit timing relationships is created. We perform a breadth-first traversal of this graph in order to levelize it. Graph nodes with no inputs, such as input pads, are inserted in the level 0 list, while nodes with fan-in only from level 0 nodes go into the level 1 list and so on. Levelizing the timing graph in this way speeds the subsequent breadth-first traversals needed to determine the net slacks after each iteration of the router. After routing, the net delays computed by the delay extractor are back-annotated into the timing graph, and two breadth-first traversals of the graph are used to compute the critical path and the slack of each connection. This timing analyzer can also be operated as a stand-alone tool by invoking VPR with the appropriate command line options [134].

Since optimizing the timing of a routing usually requires multiple iterations of routing, extracting the net delays and performing timing analysis, we took considerable care to ensure delay extraction and timing analysis were both fast. Table 4.3 summarizes the time required to perform various operations on the largest MCNC benchmark, clma, which contains 8381 4-input LUTs and 33 registers. Finally, as Appendix A shows, VPR's built-in graphics can display the nets on the critical path to make the speed bottlenecks in an FPGA apparent.

TABLE 4.3 CPU times to perform operations on circuit clma (8381 4-input LUTs).

Operation	CPU (s), 300 MHz UltraSparc
Extract net delays (30 455 two-point connections)	0.7 s
Build and levelize timing graph (78 000 nodes)	0.25 s
Back-annotate net delays and perform timing analysis	0.08 s

4.6 Router and Placement Algorithm Validation

4.6.1 Routability-Driven Router and Placement Algorithm

In this section we evaluate the quality of our routability-driven routing algorithm and our placement algorithm by comparing their results to those of other published CAD tools on a set of standard benchmark circuits [17]. The various FPGA parameters used in this section were always chosen to allow a direct comparison with previously published results. All the results in this section were obtained with a logic block consisting of a single BLE; that is, a 4-input LUT plus a flip flop, as shown in Figure 3.1. The clock net was not routed in sequential circuits, as it is usually routed via a dedicated routing network in commercial FPGAs. Each LUT input appears on one side of the logic block, while the logic block output is accessible from both the bottom and right sides, as shown in Figure 4.17. Each logic block input or output can connect to any track in the adjacent channel(s) (i.e. $F_c = W$). Each wire segment can connect to three other wiring segments at channel intersections (i.e $F_s = 3$) and the switch block topology used is the disjoint topology shown in Figure 4.7(a). There are no standard benchmark results available for routing FPGAs that contain long wire segments, so all routing wires are of length 1 in the target architecture.

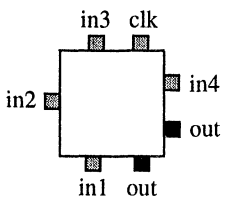


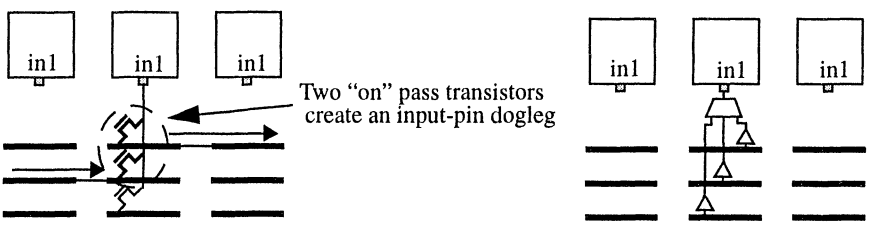
FIGURE 4.17 Logic block pin locations.

Experimental Results with Input-Pin Doglegs

Most previous FPGA routing results have assumed that "input-pin doglegs" [101] are possible. One way to build the connection block between an input pin and the tracks to which it connects uses F_c independent pass transistors controlled by F_c SRAM bits. In this case it is possible to turn on two of these switches in order to electrically connect two tracks via the input pin, as shown in Figure 4.18(a). This type of connection is called an input-pin dogleg. Commercial FPGAs, however, use a multiplexer to implement the connection block from an input pin to a channel, as shown in Figure 4.18(b). Hence only one track may be connected to an input pin. Using a multiplexer rather than independent pass transistors saves considerable area in the FPGA layout. As well, normally there is a buffer between a track and the connection block multiplexers to which it connects in order to improve speed; this buffer also means that input-pin doglegs can not be used. Therefore, while we allow input-pin doglegs in this section in order to make a fair comparison with past results, it would be best if in the future FPGA routers were tested without input-pin doglegs.

Table 4.4 lists the minimum number of tracks per channel required for a successful routing by various CAD tools on a set of 9 MCNC benchmark circuits. All the results in Table 4.4 are obtained by routing a placement produced by Altor [67], a min-cut based placement tool. Three of the columns consist of two-step (global then detailed) routing, while the other routers perform combined global and detailed routing. VPR's routability-driven router requires 10% fewer tracks than the second best router, and the third best router consists of VPR's global route phase plus SEGA [100] for detailed routing.

Table 4.5 lists the number of tracks required to implement these benchmarks by tools which change both the placement and the routing of the circuits. The size column lists the number of logic blocks in each circuit. VPR uses 13% fewer tracks when it performs combined global and detailed routing than it does when SEGA is used to



(a) Connection block to an input pin implemented via F_c pass transistors allows input-pin doglegs.

(b) More realistic connection block to an input pin.

FIGURE 4.18 Two methods of building an input pin connection block.

TABLE 4.4 Tracks required to route placements generated by Altor (input-pin doglegs allowed).

Global Router	LocusRoute [97]		GBP [91]	OGC [92]	IKMB [93]	VPR	TRACER [94]	VPR [17]
Detail Router	CGE [99]	SEGA [100]				SEGA [101]		
9symml	9	9	9	9	8	7	6	**6**
alu2	12	10	11	9	9	8	9	**8**
alu4	15	13	14	12	11	10	11	**9**
apex7	13	13	11	10	10	10	8	**8**
example2	18	17	13	12	11	10	10	**9**
k2	19	16	17	16	15	14	14	**12**
term1	10	9	10	9	8	8	7	**7**
too_large	13	11	12	11	10	10	9	**8**
vda	14	14	13	11	12	12	11	**10**
Total	**123**	**112**	**110**	**99**	**94**	**89**	**85**	**77**

perform detailed routing on a a VPR-generated global route. The FPR tool [88] first performs placement and global routing simultaneously with a partitioning-based algorithm, and then performs a separate detailed routing step. It requires 87% more total tracks than VPR, so it appears that performing global and detailed routing in one step is more important than performing placement and global routing together. Finally, allowing VPR to place the circuits instead of forcing it to use the Altor placements reduces the number of tracks VPR requires to route them by 40%, indicating that VPR's simulated annealing based placer is considerably better than the Altor min-cut placer.

Experimental Results Without Input Pin Doglegs

Table 4.6 compares the performance of VPR with that of the SPLACE/SROUTE [33] tool set, which does not allow input pin doglegs. SROUTE is an iterated maze router — when some nets fail to route, it rips up every net in the circuit and tries routing them again, with nets that would not route in the last iteration routed first in this new iteration. When both tools are allowed only to route an Altor-generated placement VPR requires 13% fewer tracks than SROUTE. Since VPR's routability-driven router

TABLE 4.5 Tracks required to place and route circuits (input-pin doglegs allowed).

Placement / Global Routing / Detailed Routing	Number of Logic Blocks in Circuit	FPR [88]	VPR	VPR
			SEGA	
9symml	70	9	6	**5**
alu2	143	10	7	**6**
alu4	242	13	8	**7**
apex7	77	9	5	**4**
example2	120	13	5	**5**
k2	358	17	10	**9**
term1	54	8	5	**5**
too_large	148	11	7	**6**
vda	208	13	9	**8**
Total	—	**103**	**62**	**55**

is also based on a maze router, this difference is due to the more sophisticated congestion avoidance scheme used in VPR. When the tools are allowed to both place and route the circuits, VPR requires 29% fewer tracks than the SPLACE/SROUTE combination. Both VPR and SPLACE are based on simulated annealing. We believe the difference in result quality is due mainly to the higher quality of VPR's annealing schedule.

The benchmarks used in the previous three tables range in size from 54 to 358 logic blocks, and accordingly are too small to be very representative of today's FPGAs. Table 4.7 presents results for the 20 largest MCNC benchmark circuits, which range in size from 1047 to 8383 logic blocks. We use FlowMap [46] to technology map each circuit to 4-LUTs and flip flops, and VPack to combine flip flops and LUTs into BLEs. The number of I/O pads that fit per row or column is set to 2, in line with current commercial FPGAs. Each circuit is placed and routed in the smallest square FPGA which can contain it. Input-pin doglegs are not allowed. Note that three of the benchmarks, bigkey, des, and dsip, are pad-limited in the FPGA architecture assumed.

Table 4.7 compares the number of tracks required to place and completely route circuits with VPR with the number required to place and globally route the circuits with

TABLE 4.6 Tracks required to place and route circuits (no input-pin doglegs allowed).

Placement	Altor		SPLACE [33]	VPR
Global + Detailed Route	SROUTE [33]	VPR	SROUTE	
9symml	7	**6**	7	**5**
alu2	9	**8**	8	**6**
alu4	12	**10**	9	**7**
apex7	9	**9**	6	**4**
example2	11	**10**	7	**5**
k2	15	**14**	11	**9**
term1	8	**7**	5	**4**
too_large	11	**9**	8	**7**
vda	12	**10**	10	**8**
Total	**94**	**83**	**71**	**55**

VPR and then perform detailed routing with SEGA. Using SEGA to perform detailed routing on a global route generated by VPR increases the total number of tracks required to route the circuits by 70% vs. having VPR perform the routing completely. This large track count increase is likely due to the reduced optimization possibilities when global and detailed routing are performed separately, particularly when input-pin doglegs are not allowed.

To encourage other FPGA researchers to publish routing results using these larger benchmarks, we have issued an "FPGA challenge" to the research community. We will pay researchers $1 (Canadian!) for each track by which they reduce the total number of tracks required from that of the previously best results. The technology-mapped netlists, the placements generated by VPR and the currently best routing track total are available at *http://www.eecg.toronto.edu/~jayar/software/software.html*. In the year since this challenge was issued (in a presentation at the FPL '97 workshop in London, UK), no CAD tool has beaten our track count totals.

4.6.2 Timing-Driven Router

Unfortunately, there are no standard benchmark results available to compare timing-driven FPGA routers on the basis of timing. Lee and Wu [94] provided critical path

TABLE 4.7 Channel widths required to place and route the 20 largest MCNC benchmark circuits.

Circuit	# LBs	SEGA	VPR	Circuit	# LBs	SEGA	VPR	Circuit	# LBs	SEGA	VPR
alu4	1522	16	**10**	dsip	1370	9	**7**	s298	1931	18	**7**
apex2	1878	20	**11**	elliptic	3604	16	**10**	s38417	6406	10	**8**
apex4	1262	19	**12**	ex1010	4598	22	**10**	s38584.1	6447	12	**9**
bigkey	1707	9	**7**	ex5p	1064	16	**13**	seq	1750	18	**11**
clma	8383	25	**12**	frisc	3556	18	**11**	spla	3690	26	**13**
des	1591	11	**7**	misex3	1397	17	**10**	tseng	1047	9	**6**
diffeq	1497	10	**7**	pdc	4575	33	**16**	Total	—	334	**197**

delay values for several MCNC benchmark circuits routed with Tracer and with SEGA, but they did not provide information on the logic block delays, switch resistances etc. used to compute these delays. As well, they used the Penfield-Rubinstein rather than the Elmore delay, so even if we determined these values we could not directly compare our timing to theirs. Finally, the FPGA architecture they targeted included only unit length wires connected by pass transistors, so it is not a good model of current commercial FPGAs.

We instead validate our timing-driven router by comparing the minimum track count and the critical path delay it can achieve to that of our routability-driven router. The routing architecture we use for this comparison is very similar to that of the recent Xilinx 4000X series FPGAs [139, 140].[1] This routing architecture contains 25% length 1 wires, 12.5% length 2 wires, 37.5% length 4 wires, and 25% "one-quarter longs", whose length is one-fourth of the chip. The length 1 and 2 wires connect via pass transistors, while the longer wires connect via tri-state buffers. As well, pass transistor switches also allow the length 4 wires to connect to the length 1 and 2 wires, and the one-quarter longs to connect to length 1 wires. The logic block used in this experiment is a logic cluster of four BLEs, with 10 inputs per logic block — this is more typical of the complexity of modern FPGA logic blocks than a single BLE logic block. We extracted the timing parameters for this FPGA architecture from SPICE simulations of a 0.35 μm TSMC process (see Section 6.2.3).

1. Many thanks to Jordan Swartz, who enhanced the VPR routing graph generator to create this Xilinx 4000X-like routing architecture [140].

Table 4.8 summarizes the results on the 20 largest MCNC circuits. The W_{min} columns give the minimum number of tracks per channel required by each router to successfully route each circuit. We define a low-stress routing [127] to be a routing in which there are 20% more tracks per channel than the minimum number of tracks per channel, W_{min}, required by the timing-driven router. We present the critical path delay achieved by each router under low-stress routing conditions because this is the typical routing case in FPGAs — usually there are some spare tracks. The infinite-routing column gives the delay achievable by the timing-driven router when it ignores congestion completely — that is, when it assumes it has an infinite supply of routing. Finally, the CPU time column indicates the time required to complete each low-stress routing on a 300 MHz UltraSparc workstation.

In Section 4.6.1 we showed that other CAD tools to which we can compare required from 10% to 87% more tracks to route circuits than VPR's routability-driven router. Table 4.8 shows that our timing-driven router requires only 6% more tracks than our routability-driven router, on average, so it is clear that VPR's timing-driven router performs very well in terms of circuit routability.

The circuit timing columns show that a timing-driven router is crucial to good circuit speed. The circuits produced by the timing-driven router are from *1.5 to 5 times* faster than those produced by the routability-driven router, and the average circuit speedup due to timing-driven routing is over *2.6 times*. With only 20% extra tracks, the timing-driven router comes within 12% of the delay value it can achieve when ignoring congestion — that is, the delay value it could achieve with infinite tracks. Although not shown in Table 4.8, the average delay achieved by the timing-driven router for high-stress routing, where $W = W_{min}$, is only 31% higher than the delay value it can achieve with infinite tracks. This graceful degradation in circuit delay as the routing problem becomes increasingly difficult is an attractive property.

Finally, notice that the CPU time required to complete a low-stress routing is from 3.3 to 16.3 times less (ten times less on average) for the timing-driven router than for the routability-driven router. This shows the superiority of the directed-search algorithm employed in the timing-driven router over even the optimized breadth-first search used in the routability-driven router. A breadth-first search slows down considerably on FPGAs like this which have large logic blocks, and hence large W values. The CPU time of a depth-first search, on the other hand, is largely insensitive to W. The timing-driven router is very fast; it required only 3.2 minutes to route clma, which contains over 8300 4-LUTs.

TABLE 4.8 Comparison of timing-driven router to routability-driven router.

Circuit	Routability-Driven Router			Timing-Driven Router			
	W_{min}	Low-Stress Delay (ns)	Low-Stress CPU (s)	W_{min}	Low-Stress Delay (ns)	Low-Stress CPU (s)	∞-Routing Delay (ns)
alu4	32	91.6	186	35	41.5	56	35.7
apex2	35	91.6	257	38	43.6	40	42.2
apex4	32	80.1	163	35	40.7	28	34.9
bigkey	24	88.9	123	25	23.8	26	23.4
clma	43	233.8	2801	47	103.3	192	97.6
des	23	114.9	247	25	37.8	25	36.6
diffeq	24	68.7	67	25	44.5	11	38.7
dsip	22	174.9	115	22	35.4	26	25.0
elliptic	36	158.1	583	39	75.5	62	63.2
ex1010	35	173.6	785	37	62.6	93	54.6
ex5p	33	79.4	115	35	36.6	21	35.3
frisc	34	245.8	575	35	77.9	82	73.5
misex3	31	78.1	137	33	35.9	40	32.8
pdc	51	192.9	2624	55	75.4	161	71.2
s298	29	182.1	205	31	75.2	37	58.9
s38417	30	189.6	929	31	57.6	93	50.5
s38584.1	26	190.5	624	27	41.1	62	35.1
seq	34	81.7	250	36	41.0	41	36.7
spla	46	208.0	1400	49	63.8	100	57.2
tseng	22	71.5	34	23	40.1	8	38.2
Arith. Av.	32.1	139.8	611	34.2	52.7	60	47.1
Geom. Av.	31.2	127.3	312	33.1	49.3	45	44.0

4.7 Summary

In this chapter we described the routing portion of the Versatile Place and Route CAD tool. To achieve VPR's goal of flexibility, the entire routing portion of the tool works with a routing-resource graph representation of an FPGA; by changing the routing-resource graph, radically different FPGAs can be investigated. VPR includes an FPGA architecture generator that can generate a routing-resource graph from a succinct architectural description, making it easy to describe and investigate many architectures quickly. This automatic generation of good FPGA architectures from a list of parameter specifications is a new and interesting problem in FPGA CAD.

This chapter described the algorithms used in both the routability-driven and the timing-driven routers included in VPR. The VPR timing-driven router is the first FPGA router that directly optimizes the Elmore delay, and the first published router that can properly optimize for FPGAs that contain both pass transistors and tri-state buffers in their routing. VPR also includes high-quality net delay extraction and path-based timing analysis modules.

The result quality of the VPR routers and of the VPR placement algorithm is very high. The routability-driven router outperforms all routers to which we can compare in terms of routing circuits with the minimum amount of FPGA routing. As well, the combination of VPR's placement and routability-driven routing algorithms outperforms all combinations of place and route tools to which we can compare. The timing-driven router also produces excellent results. It requires only slightly (6%) more tracks per channel than the routability-driven router, and produces circuits that are 2.6 times faster, on average.

In the next three chapters, we use the VPR, VPack and T-VPack CAD tools described in this chapter and in Chapter 3 to investigate three different issues in FPGA architecture. The next chapter describes our investigation into the first of these three issues: FPGA global routing architecture.

CHAPTER 5

Global Routing Architecture

In this chapter we investigate which *global routing architectures* lead to the best FPGA area-efficiency [2, 3]. We use the term global routing architecture to refer to the distribution of routing tracks across an FPGA; that is, the relative number of tracks contained in each channel of the FPGA. In the next section we describe some of the different types of global routing architectures, and explain why this is an important problem in FPGA design. Section 5.2 describes the experimental flow we use to evaluate different global routing architectures — this flow is based on the CAD tools described in Chapters 3 and 4. In Section 5.3 we investigate *directionally-biased* global routing architectures, in which the channels in the vertical direction have a different width than those in the horizontal direction. Section 5.4 examines *non-uniform* global routing architectures, which have wider channels in some regions of the FPGA than in others.

5.1 Motivation

Recall that in FPGAs all routing resources are prefabricated. This means that the number of routing tracks in each channel is set by the FPGA manufacturer; this is quite different from the situation with MPGAs and standard cells, where channels can be made narrower or wider according to the needs of a circuit. The goal of an FPGA architect is to distribute these routing resources in a manner that permits their efficient utilization by the largest class of circuits. If there are too few tracks in some area of the chip then many circuits will be unroutable, while if there are too many tracks they

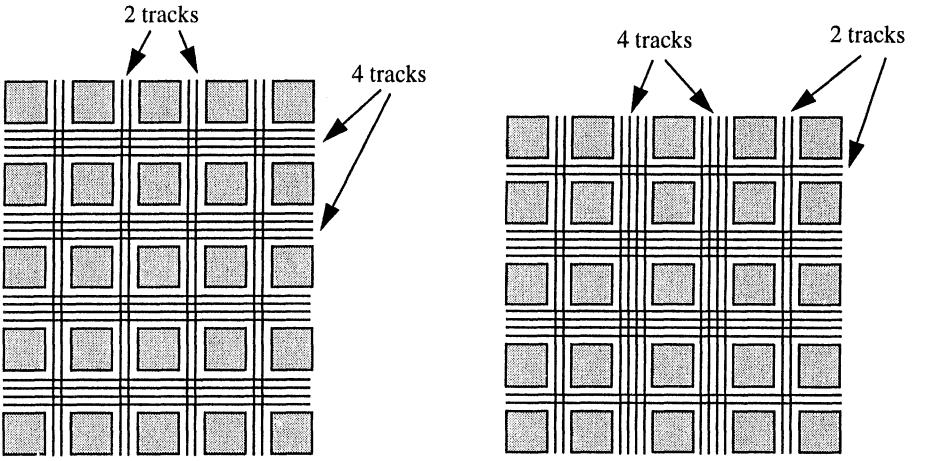


FIGURE 5.1 Global routing architectures: (a) directionally-biased; (b) non-uniform.

will often be wasted. The relatively low density of FPGAs makes an area-efficient global routing architecture essential.

This work addresses two fundamental questions concerning FPGA global routing architectures. First, should the number of tracks in the horizontal channels be different from the number in the vertical channels? Figure 5.1(a) depicts an FPGA with such a *directional bias*. In essence, we are investigating if there is an intrinsic property of circuits that makes a directional bias more area-efficient. If so, what amount of bias is best? Commercial FPGAs with both unbiased routing [4, 5] and biased routing [6, 7] exist, so this question has clear commercial relevance.

Second, should all routing channels in the same direction in an FPGA contain the same number of tracks or is a *non-uniform* routing architecture, in which some channels are wider than others, preferable? An example FPGA with a non-uniform routing architecture is shown in Figure 5.1(b).[1] Intuitively, such an architecture may facilitate routing in congested regions. However, if the wider channels cannot be used efficiently they will waste area. Many in the FPGA community believe that most routing congestion occurs near the center of an FPGA, and therefore expect that wider channels in this region would be beneficial. The Lucent Technologies ORCA 2C

1. Note that any given channel will always have the same number of tracks along its entire length. We did not consider varying the channel capacity along its length as this makes it very difficult, and likely impractical, to lay out the FPGA.

FPGAs employ a non-uniform routing architecture of this type, in which the center channel is wider than the others [141].

On the other hand, board-level constraints often force designers to fix the position of an FPGA's I/Os, and some believe that this increases congestion near the chip edges, and therefore the channel between the pads and the logic block array should be widened. The Xilinx 4000 and 5000 series FPGAs have a wide channel between the pads and logic, at least partially to improve routability when the I/O locations are fixed [11, 135]. In Section 5.4, we determine the best distribution of tracks across an FPGA both when the I/O assignment to pads is unconstrained and when it is fixed in a poor configuration.

5.2 Experimental Methodology

We evaluate FPGA architectures experimentally, using a CAD flow similar to that used in commercial FPGAs to automatically implement circuits. We compare the area-efficiency of different global routing architectures by technology-mapping, placing and globally routing 26 of the larger MCNC benchmark circuits [136] into each architecture. In this section we describe the CAD flow, the area-efficiency metric used to compare architectures, and several important architectural details.

5.2.1 CAD Flow

Figure 5.2 summarizes the CAD flow. First, SIS [142] is used to perform technology-independent logic optimization on a circuit. Next this circuit is technology-mapped by FlowMap [46] into four-input look-up tables (4-LUTs) and registers; the Flowpack post-processing algorithm [46] is then run to further optimize the mapping and reduce the number of LUTs required. The logic block used in these experiments is a single BLE (a 4-LUT plus a register); the structure of a BLE is shown in Figure 3.1(a). VPack packs 4-LUTs and registers together into these logic blocks.

The netlist of logic blocks and a description of the FPGA global routing architecture are then read into our placement and routing tool, VPR. VPR first places the circuit, and then repeatedly globally routes (or attempts to route) the circuit with different numbers of tracks in each channel, or *channel capacities*. The router used in this phase is the routability-driven router described in Section 4.3. VPR performs a binary search on the channel capacities, increasing them after a failed routing and reducing them after a successful one, until it finds the minimum number of tracks required for the circuit to globally route successfully on a given global routing architecture. Note

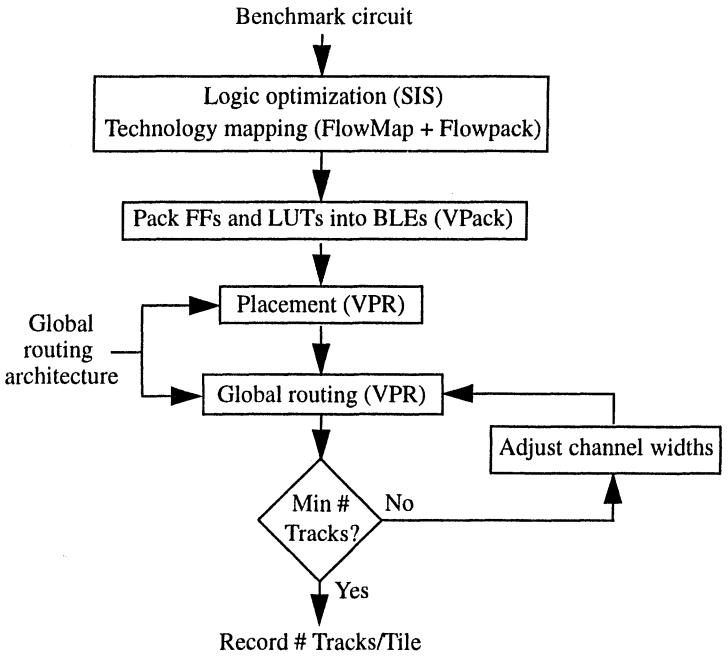


FIGURE 5.2 Architecture evaluation flow.

that while the absolute number of tracks per channel is adjusted upwards or downwards after each attempted routing, the *relative* numbers of tracks in the various channels across the FPGA are always kept at the values specified by the FPGA architecture. For example, VPR's first attempt at routing a circuit in an architecture with a two-to-one directional bias might assume horizontal channel capacities of twelve tracks and vertical channel capacities of six tracks. If this routing was successful, VPR would next attempt to route the circuit in an FPGA with horizontal channel capacities of six tracks and vertical channel capacities of three tracks, and so on until the minimum number of tracks required for routing is determined.

To compare architectures fairly, it is important that the CAD tools properly optimize for each architecture tested. Both the placement and routing phases of VPR take advantage of the biased and non-uniform nature of the different global routing architectures. Section 3.2.3 described the linear congestion cost function used by VPR during placement — this cost function leads to placements in which more congested parts of the design are placed in regions or directions with more routing. As described in Section 4.3.1, in the VPR routability-driven router the cost of a routing resource is a function of the congestion, or amount by which demand exceeds capac-

ity, at that resource. Consequently, our router will automatically act to relieve congestion in narrow channels by re-routing signals through wider channels whenever necessary.

The benchmark circuits used in this study consist of 14 combinational and 12 sequential MCNC benchmark circuits, which vary in size from 222 to 1878 BLEs.

5.2.2 Area-Efficiency Metric

Our goal is to measure the area-efficiency of different global routing architectures without reference to the detailed routing architecture (e.g. segmentation distribution and switch block topology). At this level, it is the amount of "global wiring" that changes as we vary the architecture. A simple track count will not accurately represent the wiring area of rectangular FPGAs, as the tracks in one direction are longer than those in the other. Accordingly, we define a *track segment* to be a prefabricated wire that spans one logic block; a channel of width W tracks that spans L logic blocks contains W·L track segments. The total number of track segments an FPGA must contain to globally route a circuit is a representative metric of the "global wiring" area. In order to average the results from circuits of differing sizes we use the average number of track segments per tile (i.e. per logic block) as our area measure. For example, in a square M x M uniform FPGA with W tracks in each channel, the total number of track segments is $2W \cdot M^2$, and the number of tracks per tile is 2W. Note that the routing area is given by the total number of track segments in the entire FPGA, and not the number of track segments which are actually used by a circuit.

5.2.3 Significant FPGA Architectural Details

Several architectural parameters other than the global routing architecture must be specified in order to define an FPGA. We set these parameters to be as close to those of commercial FPGAs as possible.

First, the size of the logic block array used for a given circuit is set to the smallest value with the desired aspect ratio (number of columns / number of rows) with sufficient logic blocks to accommodate the circuit (e.g. 11 x 11 logic blocks). This situation, in which there is minimal "spare room" in the FPGA, presents the greatest challenge to routing completion and is normally the case manufacturers wish to optimize.

In this study the number of I/O pads that can fit into the height or width of a logic block is set to two. This number is commensurate with the relative sizes of I/O pads

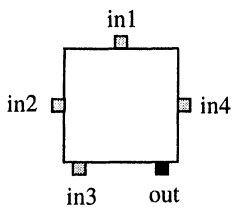
and 4-LUTs in current FPGAs [4, 5, 7] and ensures that none of the 26 benchmarks is pad-limited.

Finally, we do not route the clock net (all the MCNC benchmarks use only a single clock) in sequential circuits, since this net is normally distributed through a special clocking network in commercial FPGAs.

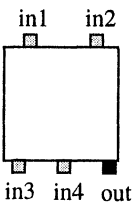
5.3 Experimental Results: Directionally-Biased Routing

The experimental framework and tools described above were employed to answer the questions posed in Section 5.1: first, is there an area-efficiency advantage to a directionally-biased architecture? Recall that in a directionally-biased FPGA the numbers of routing tracks per channel in the horizontal and vertical directions are different. In essence, we are investigating if there is an exploitable directional bias in the basic nature of circuits. We characterize directionally-biased FPGAs by the ratio of the width of a horizontal channel to the width of a vertical channel, denoted as R_h. For example, Figure 5.1(a) depicts an FPGA with a 2:1 directional bias, i.e. $R_h = 2$.

We need to define an additional architectural feature which markedly affects our results: the positioning of the pins on the logic block. The two main cases of interest are illustrated in Figure 5.3. In Figure 5.3(a), the logic block input and output pins are distributed evenly around the entire perimeter of each logic block. We call this the *full-perimeter* pin positioning, and it is similar to the pin positioning used in Xilinx and Lucent Technologies FPGAs [4, 5]. Figure 5.3(b) illustrates the *top/bottom* pin positioning, which restricts the logic block input pin locations to lie only on the top and bottom of the logic block; it is similar to the pin positioning used in Actel FPGAs [6]. In all the results we show in this and subsequent chapters, each logic block pin appears on (i.e. is accessible from) only one side of a logic block. We have found that for the F_c values found in today's commercial FPGAs this leads to the most area-efficient FPGAs [143].



(a) Full-perimeter pin positioning



(b) Top/bottom pin positioning

FIGURE 5.3 Logic block pin positions: (a) full-perimeter and (b) top/bottom.

We have also found that the ratio of the number of columns to the number of rows in an FPGA, which we call the logic block array aspect ratio, significantly affects area efficiency. Since most FPGAs have the same number of rows and columns, we first present the results for square (aspect ratio 1) FPGAs, before discussing the more general case of rectangular FPGAs in Section 5.3.2.

5.3.1 Results for Square Logic Block Arrays

Twenty-six large MCNC benchmarks were passed through the experimental flow of Figure 5.2 for values of R_h ranging from 1 to 4. As discussed in Section 5.2, the result for each circuit is the number of track segments per tile needed to successfully global route the circuit in an FPGA[1] with the specified value of R_h. Figure 5.4 is a plot of area-efficiency versus the degree of routing direction bias, R_h, for both types of pin positioning. The vertical axis is the average number of tracks per tile required to successfully route the 26 benchmarks. Table 5.1 summarizes how well each of the seven largest circuits maps into FPGAs with differing amounts of directional bias for both pin positionings; the results for individual circuits closely parallel the overall average shown in Figure 5.4.

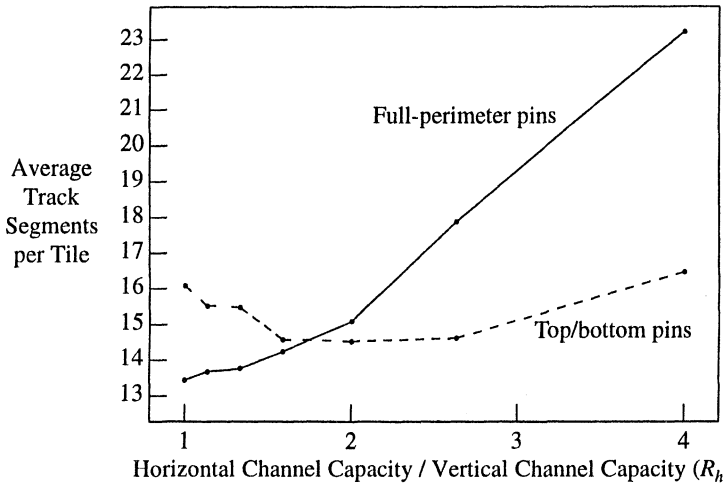


FIGURE 5.4 Area-efficiency vs. directional bias for square FPGAs.

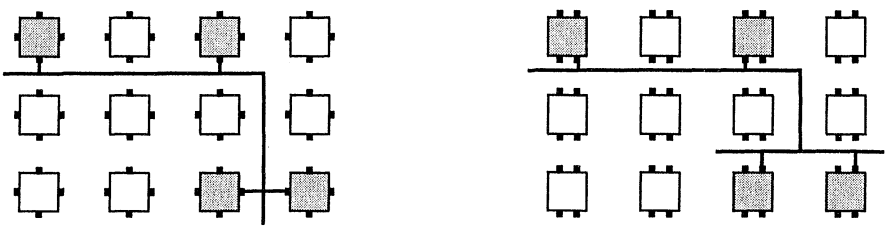
1. Track segments are counted whether or not they are actually used, so this is a true representation of the area that must be devoted to routing in the layout.

TABLE 5.1 Routing requirements of the largest benchmarks vs. directional bias.

Circuit	Number of Logic Blocks	Tracks per Tile Required for Global Routing							
		Full-Perimeter Pins Case				Top/Bottom Pins Case			
		$R_h = 1$	$R_h = 1.33$	$R_h = 2$	$R_h = 4$	$R_h = 1$	$R_h = 1.33$	$R_h = 2$	$R_h = 4$
alu4	1522	16	16	17	28	16	18	17	18
apex2	1878	18	18	20	28	20	19	18	20
apex4	1262	18	18	20	28	22	21	20	23
diffeq	1497	12	11	12	19	14	14	12	13
ex5p	1064	20	21	23	35	22	21	20	28
misex3	1397	16	16	20	31	18	16	17	20
seq	1750	16	18	20	33	20	18	18	23

Figure 5.4 shows that for the full-perimeter logic block pin positioning the best architecture has no directional bias. However, when the pins are restricted to the top and bottom of the logic block, the most efficient architecture has horizontal channels which are roughly twice as thick as the vertical channels. Another important conclusion is that the best full-perimeter architecture is about 8% more area-efficient than the best top/bottom pin architecture.

The full-perimeter architecture is more area-efficient because there is a greater chance that the block input pins are closer to their desired connections when they are in this configuration than when they are in the top/bottom configuration. For example, consider the two routings of a multi-terminal net shown in Figure 5.5. The top/bottom



(a) Full-perimeter pins (b) Top/bottom pins

FIGURE 5.5 Example multi-terminal net routing with (a) full-perimeter and (b) top/bottom pins.

pin configuration needs six track segments to route this net, while the full-perimeter configuration requires only five. By making use of the functional equivalence of LUT input pins during routing, the router can often connect to a logic block pin adjoining a track segment it needs to use for other connections, essentially making the connection to this logic block for free. Since the top/bottom pin configuration has input pins bordering on only the horizontal channels, such "free" connections into logic blocks are less frequent, reducing area-efficiency.

The full-perimeter pins configuration achieves the highest area-efficiency when there is no directional bias to the routing because this makes the difficulty of routing to each of a logic block's nearest neighbors roughly equal. Consequently, the placement software can use all the nearby logic block locations equally to cluster the fanout of a net around its driver. Essentially, this allows one to cluster tightly coupled portions of logic in the smallest possible area. The top/bottom pins configuration, on the other hand, prefers a 2:1 directional bias because every connection to a logic block pin must come from a horizontal channel. This extra pressure on the horizontal routing resources is significant, since the average distance routed between pins is only about 3 track segments.

5.3.2 Results for Rectangular Logic Block Arrays

We call any logic block array in which the number of rows does not equal the number of columns *rectangular*. The basic reason to use a rectangular logic block array is to change the aspect ratio of the resulting FPGA die. For example, if the layout of the basic FPGA tile is not square, an FPGA manufacturer may want to create a square die by using more columns than rows, or vice versa. On the other hand, FPGA manufacturers may want to increase the I/O-to-logic ratio by creating a rectangular FPGA die, as this increases the die perimeter and hence the number of pads. In this case, if the basic tile layout is square an FPGA manufacturer may use a rectangular logic block array to create a rectangular FPGA. We refer to the ratio of the number of columns in an FPGA to the number of rows as its logic block array aspect ratio. Figure 5.6 depicts an FPGA with a logic block array aspect ratio of two.

When the logic block array is rectangular, the channels in one direction are longer and have more blocks connected to them than the orthogonal channel, so the best amount of directional bias may change. Figure 5.7 is a plot of the required tracks per tile (averaged over the 26 circuits) versus R_h for various logic block array aspect ratios with the full-perimeter logic block pin positioning. There are two features of interest in Figure 5.7. First, notice that the minimum of the aspect ratio = 1 curve is the lowest of the three, indicating that a square logic block array is the most area-efficient. Secondly, the value of R_h at which the minimum area occurs increases as the aspect ratio increases. As the aspect ratio increases, the horizontal channels become longer

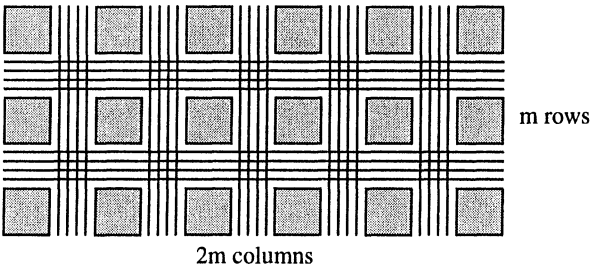


FIGURE 5.6 An FPGA with a logic block array aspect ratio of 2.

than the vertical channels and this results in greater demand for horizontal track segments. The best value of R_h increases from 1 for a square logic block array to 1.33 and 1.59 for aspect ratios of 2 and 3, respectively.

The solid curve in Figure 5.8 shows how area-efficiency varies with logic block array aspect ratio when we set R_h to the best value for each aspect ratio. The dotted curve in Figure 5.8 keeps R_h fixed at 1, which is the best value for a square logic block array. The routing resource requirements increase moderately with aspect ratio; an FPGA with a logic block array aspect ratio of 3 requires 18% more tracks per tile than an FPGA with a square logic block array when R_h is 1. When the most appropriate value of R_h is used for each aspect ratio, however, an FPGA with an aspect ratio of 3 requires only 4% more track segments than an FPGA with a square logic block array.

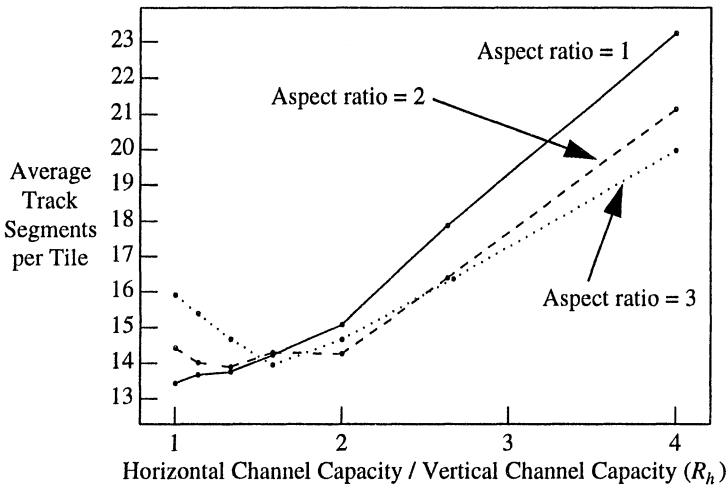


FIGURE 5.7 Area-efficiency of rectangular FPGAs with full-perimeter pins.

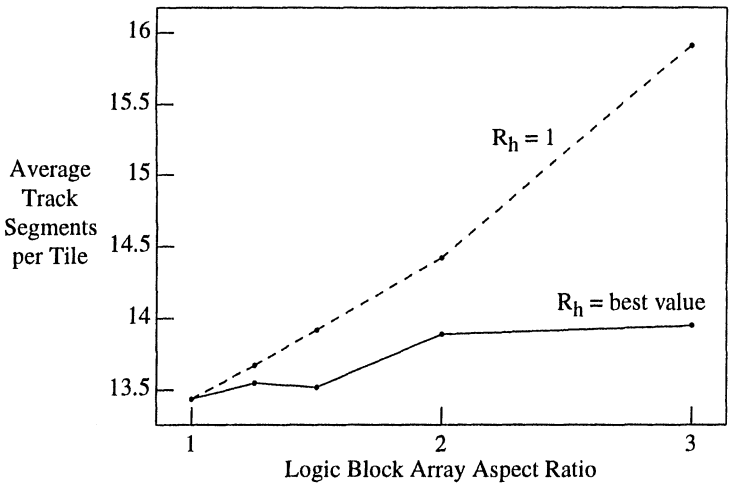


FIGURE 5.8 Area-efficiency vs. array aspect ratio for FPGAs with full-perimeter pins.

Thus we conclude that, as long as the horizontal and vertical channel widths are appropriately balanced, the logic block array aspect ratio can be increased with little impact on the core area.

The variation of routing area with logic block array aspect ratio is similar for FPGAs that use the top/bottom logic block pin positioning. In this case an FPGA with a logic block array aspect ratio of 3 requires only 5% more tracks per tile than an FPGA with a square logic block array. For FPGAs of this type, however, the increase in the best value of R_h as aspect ratio increases is less dramatic. The best square-array FPGA with top/bottom pins has horizontal channels which are twice as wide as vertical channels; the thicker horizontal channels are better able to cope with the increased pressure for horizontal tracks as aspect ratio increases.

5.4 Experimental Results: Non-Uniform Routing

The second key issue we explore concerns the area-efficiency obtained when the channels in different regions of an FPGA have different capacities. We only investigate FPGAs which use the full-perimeter pin positioning, as Section 5.3 showed that this pin positioning is best.

We define a non-uniform routing architecture to be one in which the number of tracks per channel changes from channel to channel across an FPGA. For example, Figure 5.1(b) illustrates a non-uniform FPGA in which the channels near the chip center are wider than those near the periphery. If congested regions of a circuit can be localized and placed in the portions of the FPGA with the widest channels, a non-uniform FPGA could have better area efficiency than a uniform FPGA. We will investigate three types of non-uniform FPGAs in which we vary the center/edge channel capacity ratio, the capacity of only the center channel, and the I/O channel capacity, respectively.

5.4.1 Center/Edge Capacity Ratio

Many in the FPGA community believe that most congestion occurs in the center of FPGAs, and hence expect having wider channels near the FPGA center and narrower channels near the edges to improve area-efficiency. To keep the layout problem tractable, we restrict ourselves to FPGAs which use channels of only two different widths. We can describe global routing architectures of this form with two parameters. Let R_w be the ratio of the widths of the channels near the center of the FPGA to the widths of the channels near the FPGA edges, i.e. W_{center} / W_{edge}. Let R_c be the ratio of the number of channels with width W_{center} to the total number of channels. For example, the FPGA of Figure 5.1(b) has $R_w = 2$ and $R_c = 0.5$.

Using the flow of Section 5.2, we again implemented 26 benchmark circuits in different architectures to determine their area-efficiency. We examined FPGAs with R_w

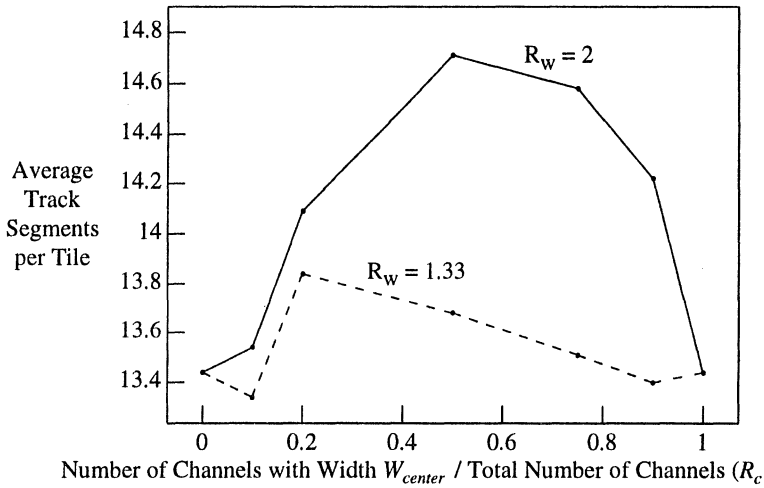


FIGURE 5.9 Area-efficiency vs. non-uniform routing architecture.

equal to 0.75, 1.18, 1.33, and 2, and with R_c values varying from 0 to 1. The relative density of FPGAs with $R_w = 1.33$ and $R_w = 2$ is summarized in Figure 5.9, which plots the average number of tracks per tile required by the 26 benchmarks in each architecture. Note that the points at which R_c equals 0 or 1 correspond to a *uniform* FPGA, in which all channels have the same capacity. The area required by each of the seven largest benchmarks in several representative architectures is listed in Table 5.2; again the results for individual circuits generally match the overall average, although there is some circuit-dependent behaviour. Note that the number of tracks per tile required for routing individual circuits in a non-uniform architecture is generally not an integer. Since some portions of the FPGA have wider channels than others, the number of tracks per tile is an average over the various tiles such an FPGA must contain.

TABLE 5.2 Track segments / tile required for global routing by the largest benchmark circuits vs. non-uniform architecture.

Circuit	Uniform ($R_w = 1$)	$R_w = 1.33$			$R_w = 2$		
		$R_c = 0.2$	$R_c = 0.5$	$R_c = 0.75$	$R_c = 0.2$	$R_c = 0.5$	$R_c = 0.75$
alu4	16	14.7	15.9	16.8	16.0	16.6	15.6
apex2	18	16.8	17.9	18.8	19.2	23.5	18.9
apex4	18	17.1	17.8	18.7	18.6	22.4	20.1
diffeq	12	12.8	13.8	11.4	14.4	13.6	15.6
ex5p	20	21.1	20.8	20.2	20.8	19.6	20.5
misex3	16	16.7	17.9	14.8	16.5	16.9	16.9
seq	16	16.7	18.0	16.9	18.6	19.9	17.2

The results generally show that the less uniform the channel widths, the worse the FPGA area-efficiency. The worst area-efficiency with $R_w = 2$ occurs when R_c is 0.5, meaning that half the FPGA channels are twice as wide as the other half. In fact, only two non-uniform FPGAs show even marginal area-efficiency improvements over the uniform case and both these FPGAs are very close to a uniform architecture. In one, the 10% of channels nearest the center are 33% wider than the other channels, while in the other the 90% of channels closest to the center are 33% wider than the channels nearest the edges. The reduction in tracks per tile over a uniform FPGA is less than 1% for both of these FPGAs; this marginal improvement is certainly not enough to justify the extra layout effort required in the physical design of such an FPGA.

These results are significant because many FPGA architects believe that there should be considerable benefit to these kinds of non-uniform architectures. The fundamental reason they do not show any benefit is that there is not much more congestion in the center of an FPGA than there is near its edges. In order to determine the "natural" routing demand distribution of circuits, we placed and routed the 26 benchmark circuits with all congestion avoidance features disabled, so that placement minimized wirelength and the router connected each net by the shortest path. Figure 5.10 plots the maximum and average number of tracks required by the horizontal channels as a function of the channel position within the FPGA, averaged over the 26 benchmark circuits. Demand for routing tracks is relatively constant over the middle 90% of the FPGA, and there is only a moderate decrease as one gets very close to the chip edges. Figure 5.11 plots the average and maximum tracks required by each horizontal channel versus the channel position for three representative benchmark circuits. There is some high-frequency variation from channel to channel, since the router is, in this case, not making any effort to route nets around congestion. Nevertheless, it is clear that these circuits closely mirror the behavior of the overall averages of Figure 5.10.

An additional reason for the poor area-efficiency of FPGAs with extra routing near their center is that typical circuits contain numerous local congestion "hotspots" (small regions where all the channels are full) and some of these hotspots occur quite close to the FPGA edge. Consequently, in order for an FPGA with thicker channels near its center to use fewer routing resources, the placement software must move all of these hotspots into the FPGA center. As discussed in Section 3.2.3, we spent con-

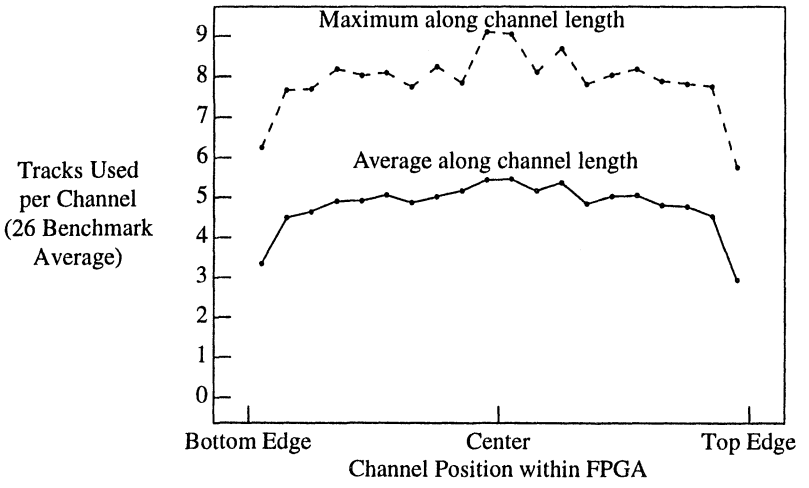
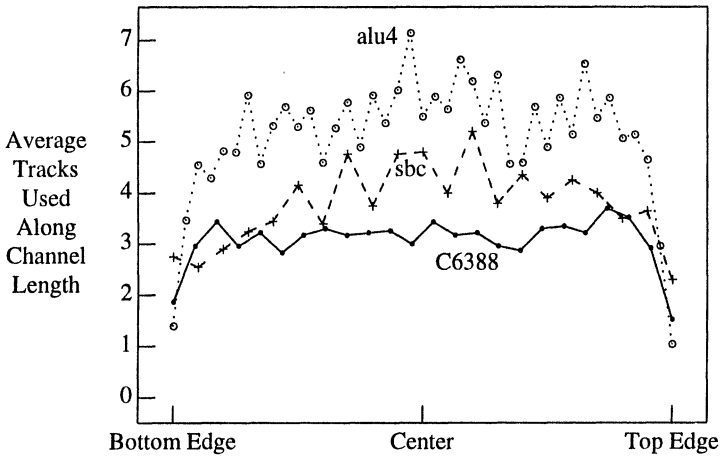
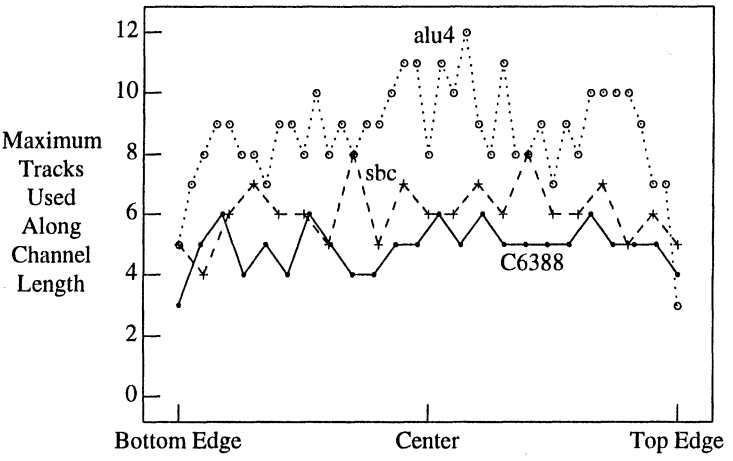


FIGURE 5.10 Average over benchmarks of track demand vs. position for horizontal channels.



(a) Average track demand vs. position



(b) Maximum track demand vs. position

FIGURE 5.11 (a) Average and (b) maximum track demand vs. channel position for three circuits.

siderable time investigating placement cost functions that modelled congestion well. Of the cost functions we examined, the non-linear congestion cost function (see Section 2.2.2) performs the most detailed congestion modelling, at the cost of a very high CPU time requirement. When compared to the linear congestion cost function that

we used to generate the results of this chapter, the non-linear congestion cost function improved the performance of a uniform FPGA slightly more than it improved the performance of the non-uniform FPGAs tested. We believe it is therefore more effective for CAD tools to attempt to spread out congestion as much as possible, rather than to try to localize it to a designated portion of a chip.

Engineers with MPGA or standard cell design experience may find it surprising that having wider channels near the center of an FPGA doesn't improve area-efficiency. Since standard cell and sea-of-gates Mask-Programmed Gate Array (MPGA) designs often have some channels wider than others, one might expect that FPGA-based designs would also benefit from a mix of wide and narrow channels. There is a key difference between FPGAs and standard cells or MPGAs, however — in FPGAs the width of the various channels is determined by the FPGA architecture, not the circuit being implemented. If one builds an FPGA with some channels wider than others, *all* circuits are forced to route most of their connections through the predefined wide channels, whether it is efficient for a particular circuit to do so or not. In a standard cell or sea-of-gates MPGA design, on the other hand, a channel is made extra wide or extra narrow only if the circuit being implemented naturally demands it.

5.4.2 Single Center Channel

In an effort to improve routability, Lucent Technologies, has introduced an FPGA in which the center channel in each direction is extra-wide [141]. We define R_m to be the ratio of the width of these center channels to the width of the other channels. Figure 5.12 depicts an FPGA with this extra routing in its center.

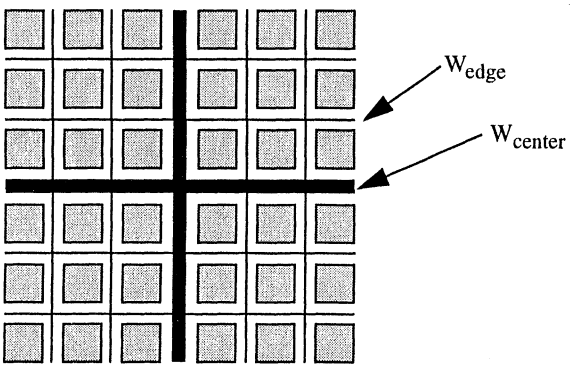


FIGURE 5.12 An FPGA with an extra-wide center channel.

The solid and dashed lines in Figure 5.13 show how area-efficiency varies with R_m for this type of FPGA when a linear congestion cost function and when a bounding box cost function are used during placement, respectively. The data show that the most area-efficient FPGA is one without an extra wide channels in the middle — i.e. $R_m = 1$. There is a sharp dip in the number of tracks/tile required at $R_m = 2$ when the linear congestion cost function is used, indicating an FPGA with area-efficiency almost as good as one with $R_m = 1$. This dip occurs at the first point at which the linear congestion cost function considers the cost of routing through narrow channels to connect two adjacent blocks to have the same cost as connecting two blocks separated by one intervening block through the extra wide channel. Consequently, the placer is able to make better use of the extra-wide channel at this point. The bounding box cost function leads to worse area-efficiency than the linear congestion cost function and the dip in required area near $R_m = 2$ does not occur. As with the non-uniform FPGAs of Figure 5.9 then, the best results are obtained by spreading extra routing resources over the entire FPGA rather than by adding them to only one region.

Aside from trying to create a more area-efficient FPGA, there is an alternate reason for making the center channel of an FPGA wider. FPGA manufacturers tend to create several FPGAs with the same basic architecture but with different numbers of logic blocks in order to appeal to customers with different capacity needs. As the number of logic blocks in a circuit increases, the demand for routing also increases [144], so at some point the channels should be widened. However, widening the channels for FPGAs with more logic blocks requires redoing the layout of the basic tile, which involves considerable time and expense. Since the center of an FPGA typically con-

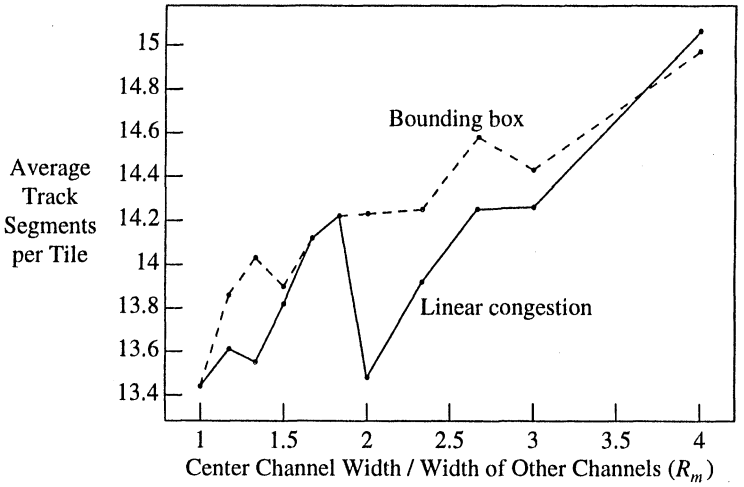


FIGURE 5.13 Effectiveness of an extra-wide center channel.

tains the programming logic, this area already requires its own custom layout, so extra routing tracks can be added to it with relatively little effort.

We found, however, that in most cases adding tracks to the center channel did not result in any significant reduction in routing pressure on the other channels. The only exceptions were FPGAs with R_m near two. When circuits were mapped into such FPGAs by a placer using the linear congestion cost function the routing pressure on the other channels was reduced. It is relatively difficult to improve the routability of an FPGA simply by adding tracks to the center channel because the placement tool must now move all portions of the circuit that require more routing than the "normal" channels can provide close to the center channel so that they can make use of the extra routing available there.

5.4.3 I/O Channel

Xilinx has added extra routing resources to the "I/O-channel" that runs between the I/O pads and the logic blocks, at least in part to ensure that fixed or "locked" I/O pad placement does not impact routability and speed [11]. We define R_{IO} to be the ratio of the width of this outermost channel to the width of the other channels. For example, Figure 5.14 depicts an FPGA with R_{IO} equal to 0.5. In the experiments of this section, all the channels running between logic blocks have the same width, W_{Logic}, so R_{IO} completely describes the global routing architecture.

Figure 5.15 is a plot of the average track segments per tile required for the 26 benchmark circuits versus R_{IO}. The solid line in Figure 5.15 shows the trend when the I/O locations are chosen by the placement tool, while the dashed line is found when the

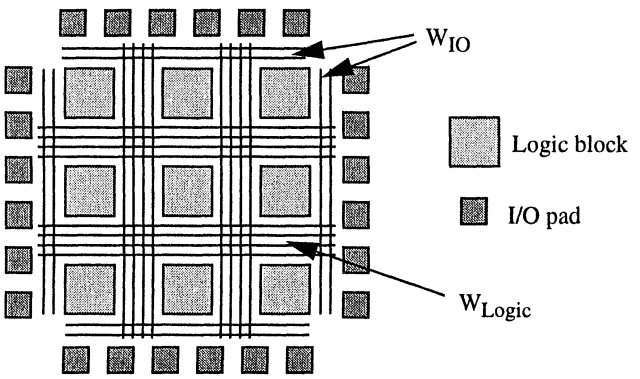


FIGURE 5.14 An FPGA with $R_{IO} = 0.5$.

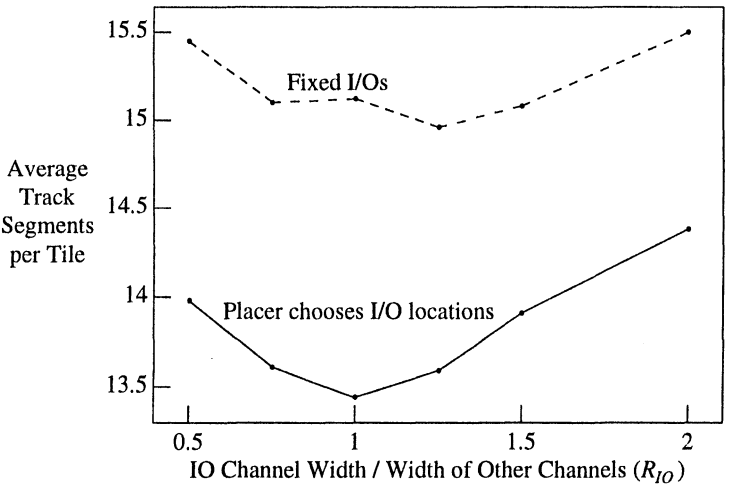


FIGURE 5.15 Routability vs. I/O channel width.

I/O pads are "fixed" in a random location, to model the effect of poor (from the FPGA's point of view) pin constraints.

There are several features of interest in Figure 5.15. First notice that fixing the I/O locations increases the number of routing tracks required by 12% on average. Architects must take this into account when designing FPGAs. Secondly, the curve where the I/O locations are chosen by the placement tool has its minimum value when R_{IO} = 1, again showing that it is best to spread routing resources evenly across the chip. Fixing the I/O pins shifts the minimum in the tracks per tile curve slightly so that it now occurs when R_{IO} = 1.25. While fixing the I/O pins leads to a significant increase in the number of routing tracks required, this increase is, for the most part, spread over the FPGA and not confined to the channels connecting to the I/O pads. Consequently, one should not make very wide channels adjoining the pads in order to improve routability with pin constraints, although a small increase in the I/O channel capacity is a net benefit.

In interpreting the best values of R_{IO} we found, one should remember that in the FPGAs we consider the number of pins connecting to an I/O channel is roughly the same as the number of pins connecting to a logic channel. In architectures with different ratios of I/O pad to 4-LUT area, the best R_{IO} values will change as the ratio of the number of pins bordering a logic channel to the number bordering an I/O channel changes.

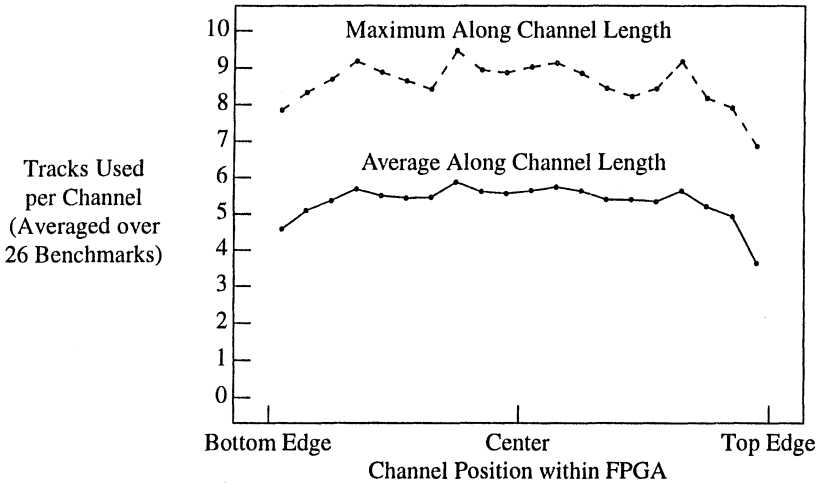
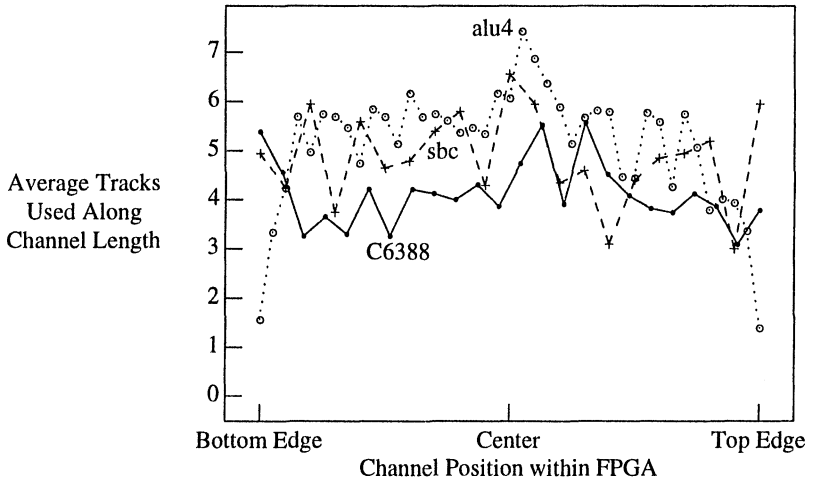
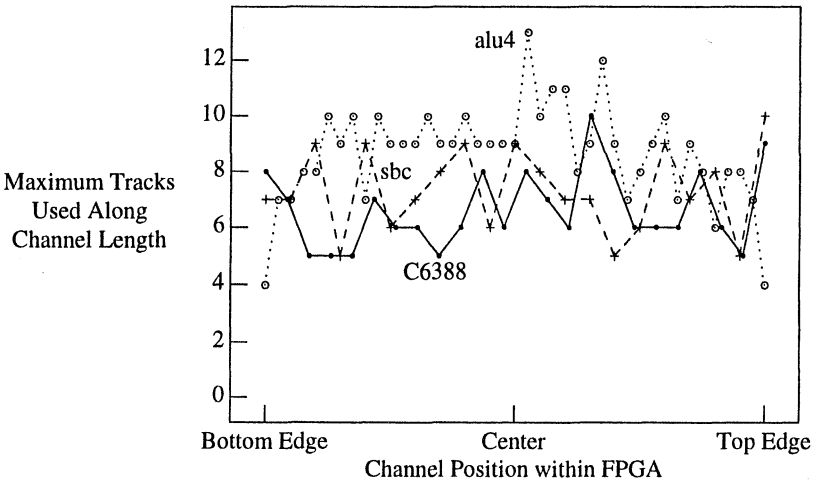


FIGURE 5.16 Average over benchmarks of horizontal track demand when I/Os are fixed in a random configuration.

In order to determine how the "natural" demand for tracks is altered when the I/O locations of a circuit are fixed in a poor configuration, we repeated the congestion-oblivious placement and routing experiments described in Section 5.4.1 with the I/O locations fixed in a random configuration. Figure 5.16 plots the maximum and average number of tracks required by the horizontal channels as a function of the channel position within the FPGA, averaged over the 26 benchmark circuits. Comparing with the corresponding curve obtained with movable I/Os (Figure 5.10), one sees that the curves have shifted up by approximately half a track, and that the drop off in track demand near the chip edges is less pronounced. Figure 5.17 shows how the "natural" track demand of three typical circuits vary with channel position. By comparing with Figure 5.11, one sees that the curve for alu4 has changed little, while the sbc and C6388 curves have each shifted up by approximately a track and show significantly more demand for routing tracks near the chip edges than they did when the I/Os were movable. This is due to the different I/O to logic ratios of these three circuits. Alu4 has very few I/Os; it uses only 7% of the I/O pads available in the FPGA to which it is mapped. C6388 and sbc, on the other hand, have considerably more I/O, and use 35% and 61% of the I/O locations available to them, respectively. As one would expect, then, fixing I/O locations has little effect on circuits with few I/Os. On the other hand, circuits with larger I/O requirements show an increase in routing track demand across the entire FPGA, with the greatest increase near the chip edges.



(a) Average track demand vs. position



(b) Maximum track demand vs. position

FIGURE 5.17 (a) Average and (b) maximum track demand vs. channel position for three circuits when I/Os are fixed in a random configuration.

5.5 Summary

The most interesting (and unexpected) result in this chapter is that the most area-efficient global routing structure is one with completely uniform channel capacities across the entire chip and in both horizontal and vertical directions. The basic reason is that most circuits "naturally" tend to have routing demands which are evenly spread across an FPGA. The only (slight) exception we found to this "uniform is better" rule occurred when the I/O locations of circuits were fixed by board-level constraints. In this case making the I/O channel 25% wider than the other channels was a net benefit.

Of almost equal note, the area-efficiency is decreased only slightly by some non-uniform or directionally-biased architectures, provided the pin placement on the logic blocks is well-matched to the channel capacity distribution. Hence if such architectures are desirable for other reasons the impact on core area doesn't preclude their use.

More specifically, of the FPGA architectures studied, a full-perimeter pin position FPGA with no directional routing bias and uniform channel widths is most area-efficient. Employing a logic block with the top/bottom pin position requires approximately 8% more routing resources than full-perimeter FPGAs, and the most area-efficient top/bottom FPGA has twice as many horizontal routing tracks as vertical ones. We also found that one can construct FPGAs with rectangular logic block arrays which are only slightly less dense than square arrays provided one adjusts the degree of directional bias in the routing resources to best match the array aspect ratio.

Our experimental results in this chapter were gathered using VPR's linear congestion cost function during placement because we felt the non-linear congestion cost function was too slow to be commercially viable. However, it is interesting to note that while the non-linear congestion cost function slightly improved the routability of circuits for all FPGA architectures, it improved routability the most for uniform routing architectures. Apparently it is easier for advanced CAD tools to spread out congested regions than it is to localize them to designated portions of a chip that have extra routing resources. Consequently, we expect that future advances in CAD tools will tend to slightly increase the advantages of uniform routing architectures over their non-uniform counterparts.

CHAPTER 6

Cluster-Based Logic Blocks

In this chapter we investigate the speed and area-efficiency of FPGAs which use *logic clusters* as their logic block. A logic cluster is composed of several look-up tables and registers interconnected by local routing, as described in Section 3.1.1. In the next section we motivate our research by describing some of the advantages of cluster-based logic blocks, and by showing that these logic blocks are commercially relevant. Section 6.2 describes the experimental flow we use to evaluate different logic clusters. Sections 6.3 through 6.6 then explore several key architectural questions concerning these logic blocks: how many inputs (I) should the FPGA routing provide to each logic cluster; how should the logic block to general routing interface change as a function of logic cluster size (N); and how are circuit speed, FPGA area-efficiency, and design compile time affected by the size of the logic cluster used?

6.1 Motivation

As described in Section 2.1.2, most SRAM-based FPGAs use logic blocks based on look-up tables (LUTs). A look-up table with more inputs can implement more logic, and hence one needs fewer logic blocks to implement a circuit. This saves routing area, as there are fewer connections to route between logic blocks. However, look-up table complexity grows exponentially with the number of inputs, so it is impractical to use a LUT with a large number of inputs as a logic block.

Instead of creating a larger logic block by increasing the number of inputs to a LUT, we can simply group several LUTs together and provide local routing to interconnect

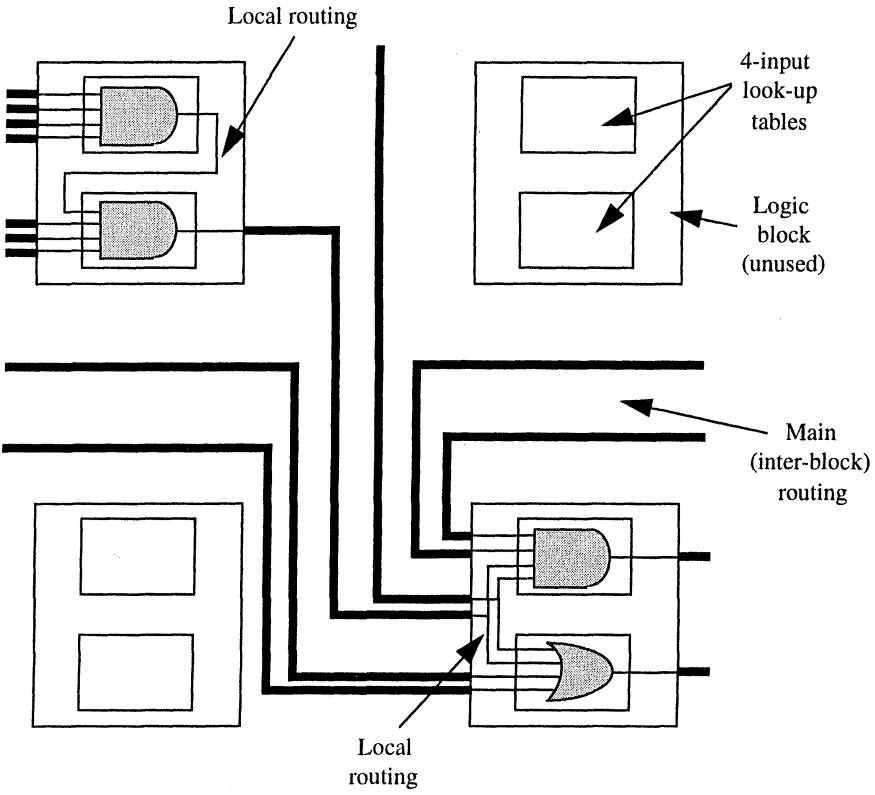


FIGURE 6.1 Implementation of a circuit in an FPGA with a cluster size of 2.

them. We call the resulting logic block a *logic cluster* [8, 9, 10]; the exact structure of a logic cluster was described in Section 3.1.1. Figure 6.1 shows a circuit implemented in an FPGA in which each logic cluster contains two four-input look-up tables. Notice that many connections are made via the local interconnect within a cluster.

One major advantage of an FPGA employing a logic block that contains several LUTs is that fewer logic blocks will be needed to implement a circuit than would be needed if each logic block was a single LUT. This reduces the size of the placement and routing problem considerably. Since placement and routing is usually the most time-consuming step in mapping a design to an FPGA, cluster-based logic blocks can significantly reduce design compile time. As FPGAs grow larger, it is important to keep this compile time from growing too large or one of the key advantages of FPGAs, rapid prototyping and design spins, will be lost [127].

Cluster-based logic blocks also have the potential to significantly improve FPGA speed. In FPGAs composed of logic clusters, many connections will be made via the local routing within a cluster. Since this local routing can be made faster than the general-purpose routing between logic blocks, cluster-based logic blocks can improve FPGA speed. It is not obvious which logic cluster size leads to the highest speed FPGA, however. On the one hand, larger logic clusters lead to a higher fraction of connections being made via local interconnect. On the other hand, as the size of a logic cluster increases, the local cluster interconnect becomes slower, potentially resulting in a speed decrease even though more connections are captured within the logic clusters.

The area impact of grouping multiple LUTs into a logic cluster is also complex. Grouping related LUTs together into a single logic block reduces the number of connections to be routed between logic blocks, which saves routing area. Since the general-purpose interconnect consumes most of the die area in SRAM-based FPGAs, this is a significant area savings. On the other hand, in the logic clusters we study the area required by the local routing within a cluster grows quadratically with cluster size. For sufficiently large clusters, then, the area used by this local interconnect will exceed the area saved in the general interconnect.

We explore four questions concerning the design of cluster-based logic blocks. First, how many distinct inputs should the FPGA routing provide to a cluster of LUTs? Reducing the number of inputs to a logic block saves routing area, but if the number of inputs is too low many circuits will be unable to use all the LUTs in a logic cluster, wasting area. Secondly, how should the flexibility of the logic block / routing interface (i.e. F_c [1]) change as the number of LUTs in a logic cluster changes? Third, how many LUTs should be included in a cluster to create FPGAs with the best combination of speed and area-efficiency? Finally, how is the time required to compile a circuit affected by the size of logic cluster used? Recent FPGAs from Xilinx, Altera, Lucent Technologies, Actel and Vantis have all grouped several LUTs together into a larger logic block, but there has been little published work investigating any of these questions.

Recall from Section 3.1.1 that we can describe a logic cluster with four parameters: the number of logic inputs (I), the number of BLEs (LUTs and registers) in a cluster (N), the number of clock inputs (M_{clk}), and the number of inputs to each LUT (K). In this chapter we fix the number of clocks per cluster at one for all our experiments, since the MCNC benchmark circuits we use to evaluate architectures all have only one clock. We set the number of inputs to each LUT, K, to 4, since previous research has shown LUTs of this size are the most area-efficient [25], and because this is the LUT size used in most commercial FPGAs. We will investigate logic clusters with

different values of I and N, however, as answering the questions posed above involves finding the most appropriate values of I and N.

6.2 Experimental Methodology

Our goal is to determine the cluster parameters that lead to the fastest and most area-efficient FPGA architectures. There are no detailed analytic models of FPGA architectures and circuitry, so we must evaluate architectures experimentally.

We implement a set of twenty benchmark circuits into each FPGA architecture of interest, and measure the circuit speed and area required for each architecture. We implement each circuit using an automatic CAD flow similar to that used by typical FPGA users: technology-mapping, placement and routing. The benchmark circuits used are the 20 largest MCNC circuits [136]; they range in size from 1064 to 8383 BLEs. These circuit sizes are typical of the designs being implemented in current commercial FPGAs.

6.2.1 CAD Flow

Figure 6.2 illustrates the CAD flow used in these experiments. First, the SIS [142] synthesis package is used to perform technology-independent logic optimization of each circuit. Next, each circuit is technology-mapped into 4-LUTs and flip flops by FlowMap and the Flowpack post-processing algorithm is used to optimize the mapping [46]. Our timing-driven T-VPack program (described in Section 3.1.3) then maps this netlist of 4-LUTs and flip flops into logic clusters with the specified values of N and I. At this point, then, the circuit is described as a set of interconnected logic blocks of the exact type that exist in the FPGA we're targeting. Finally, we use VPR to place and completely (combined global and detailed) route the circuit. In this chapter all routing was performed by the VPR timing-driven router described in Section 4.4.

As Figure 6.2 shows, the circuit is repeatedly routed with different channel capacities until VPR finds the minimum number of wire segments per channel required to successfully route the circuit, which we call W_{min}. FPGA manufacturers normally build enough routing into their FPGAs that "average" circuits have some spare routing available. We model this by performing a final "low-stress" routing of each circuit with the number of tracks per channel set to $1.2 \cdot W_{min}$. Our delay model then estimates the circuit critical path, and our area model estimates the total transistor area needed to lay out the FPGA. At the end of this CAD flow, then, we have enough

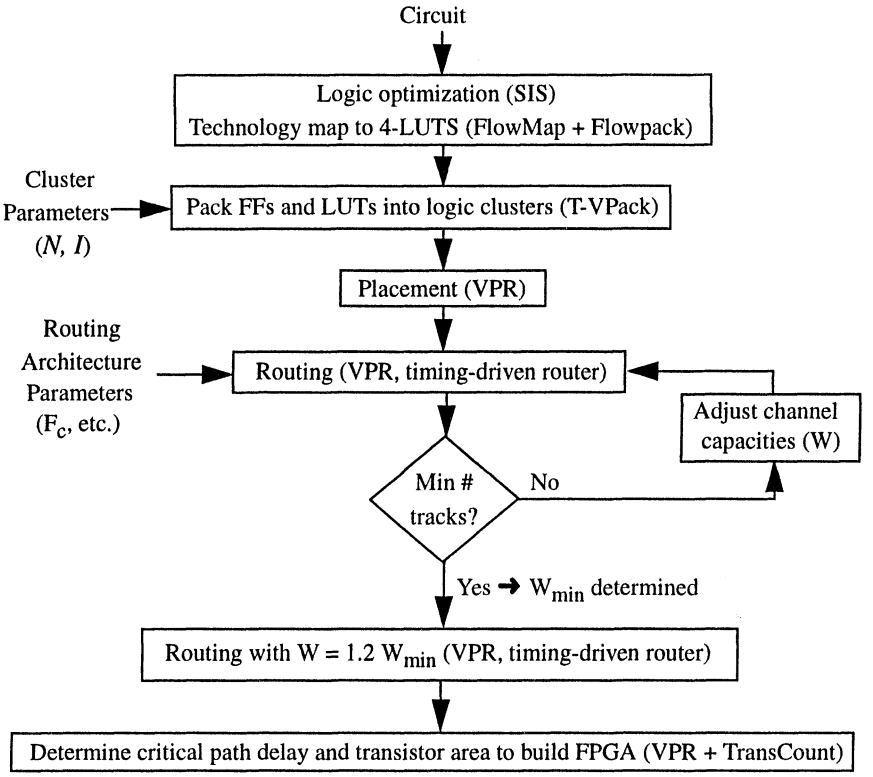


FIGURE 6.2 Architecture evaluation flow.

information to compare both the speed and the area-efficiency of one logic block architecture to another.

Notice that in the above CAD flow we are allowing the channel width to vary according to the needs of each circuit. By allowing the channel width to vary and searching for the minimum routable width, we can detect small improvements in FPGA architectures or CAD algorithms that might otherwise go unnoticed. If instead we mapped each circuit into an FPGA with a fixed channel width, we would only know whether each circuit successfully routed or not. It is more difficult to draw architectural conclusions from such a "binary" result.

6.2.2 Area Model

Our area model is based on counting the number of *minimum-width transistor areas* required to implement each FPGA architecture. A minimum-width transistor area is simply the layout area occupied by the smallest transistor that can be contacted in a process, plus the minimum spacing to another transistor above it and to its right, as shown in Figure 6.3. Since the area of typical commercial FPGAs is dominated by transistor area,[1] the most accurate way to assess the area of an FPGA architecture, short of actually laying out each FPGA architecture studied, is to estimate the total transistor area required by its layout. By counting the number of minimum-width transistor areas required to implement an FPGA, rather than the number of square microns which these transistors would occupy, we obtain a process-independent estimate of the FPGA area.

Some transistors in FPGAs require a drive strength greater than that of a minimum-width transistor. These transistors must either be made wider than minimum-width or

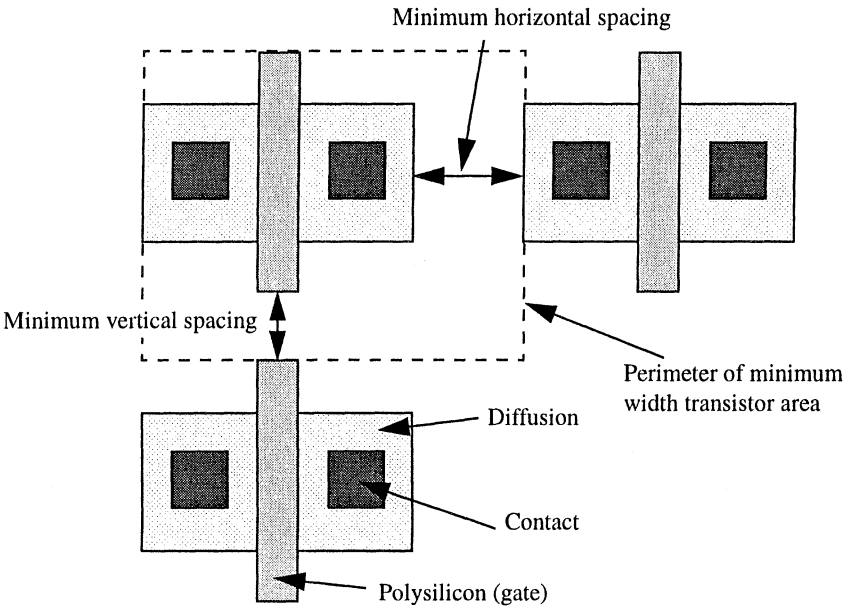


FIGURE 6.3 Definition of a minimum-width transistor area.

1. We have discussed this issue with FPGA architects at both Xilinx and Altera, and they have confirmed that transistor area determines the die size of their current FPGAs.

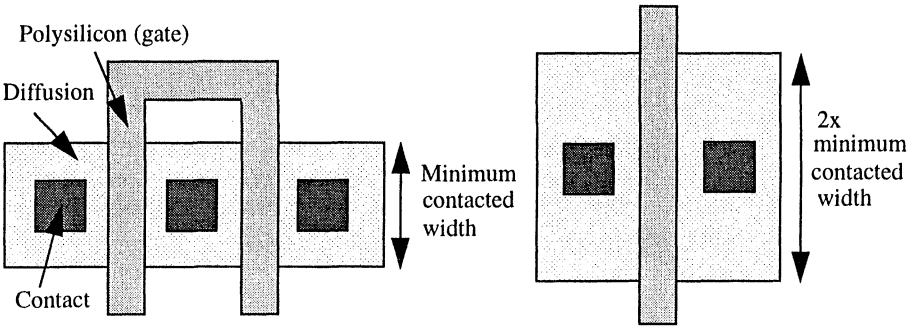


FIGURE 6.4 Methods to create a structure with 2x the drive strength of a minimum-width transistor.

their drive strength must be increased via parallel diffusion regions, as Figure 6.4 shows. The transistor active area increases in either case, but notice that if the parallel diffusions technique of Figure 6.4(a) is used, the transistor active area increases by less than a factor of two. As well, the spacing to the next transistor does not increase when a transistor's drive strength is increased, so the active area plus spacing area of a transistor with twice the minimum drive strength is less than two minimum-width transistor areas. We examined the layout rules from a TSMC 0.35 μm process and from an LSI Logic 0.4 μm process, and determined how much extra area was required to give transistors greater drive strength, either by making them wider or by paralleling diffusions. In both the LSI and TSMC processes, the number of minimum-width transistor areas required by a transistor, *trans*, (averaged over the different layout options) is:

$$\text{Minimum width transistor areas(trans)} = 0.5 + \frac{DriveStrength(trans)}{2 \cdot DriveStrength(MinimumWidth)}. \quad \textbf{(6.1)}$$

A transistor with the minimum drive strength therefore takes 1 minimum-width transistor area, while a transistor with double the minimum drive strength requires 1.5 minimum-width transistor areas.

In order to apply (6.1) to determine an FPGA's area, we must determine the number and size of the transistors required to build every structure in the FPGA. Appendix B.1 provides schematics showing how we build the key structures in an FPGA, and Section 6.2.5 discusses transistor sizing issues. In general, we tried to build an FPGA with as few transistors as possible without unduly compromising speed. We created a program, *TransCount*, that determines the area of a cluster-based logic block (includ-

ing the local cluster routing) with any values of N, I, K, and M_{clk}. This program is fairly sophisticated, and models such effects as buffer resizing as a function of the fanout of the connections within a logic block, and builds multi-stage buffers when high drive strengths are required. Of course the area of an FPGA includes not only the logic block area, but also routing area. VPR determines the routing area of each FPGA of interest, and by adding this area to the logic block area we obtain the total FPGA area. Recall that to evaluate the area of the FPGA needed to route a given circuit we build an FPGA with a channel width, W, of $1.2W_{min}$.

6.2.3 Delay Model

Our delay values are all based on the delays in TSMC's 0.35 μm, 3.3 V CMOS process. Some of the delays we use are listed in this section and in Appendix B.2, while some delays cannot be listed because the process information is proprietary and was obtained under a non-disclosure agreement.

To determine the critical path of a circuit, we must:

1. Determine the delay of every connection internal to a logic block,
2. Determine the delay of every connection between logic blocks, and
3. Perform a path-based timing analysis of the circuit using these delay values.

We found the delay of the connections within logic blocks by performing SPICE simulations of every structure in a logic block. Figure 6.5 shows the major structures and speed paths in a logic cluster, while Appendix B.1 contains transistor-level schematics that show how we build the multiplexers, buffers, look-up tables, and latches contained in a logic cluster. Since loading effects and input signal swing times can considerably change the delay of a circuit, we always simulated speed paths with their loads in place, and with the input to the path driven by the circuit which would drive it in a real FPGA. Notice that as the number of BLEs in a cluster increases, the number of inputs to the multiplexers forming the local cluster routing increases. As well, the fanouts (within the cluster) of the I cluster input signals and N cluster output signals increase, so the buffers driving these signals must be made larger (which results in a longer, and slower, inverter chain). For both these reasons, the delays of some of the paths within a logic cluster increase as the cluster size increases, as Table 6.1 shows.

After a routing is complete, we can perform step 2 above — determine the delay of every routed connection. The circuits we map contain thousands of nets, so SPICE simulation would be prohibitively time-consuming. Instead, we follow the procedure described in Section 2.2.4; we model pass transistors and buffers by equivalent circuits composed of resistors, capacitors, and idealized, constant delay elements. The values of the various equivalent resistances, capacitances and buffer intrinsic delays

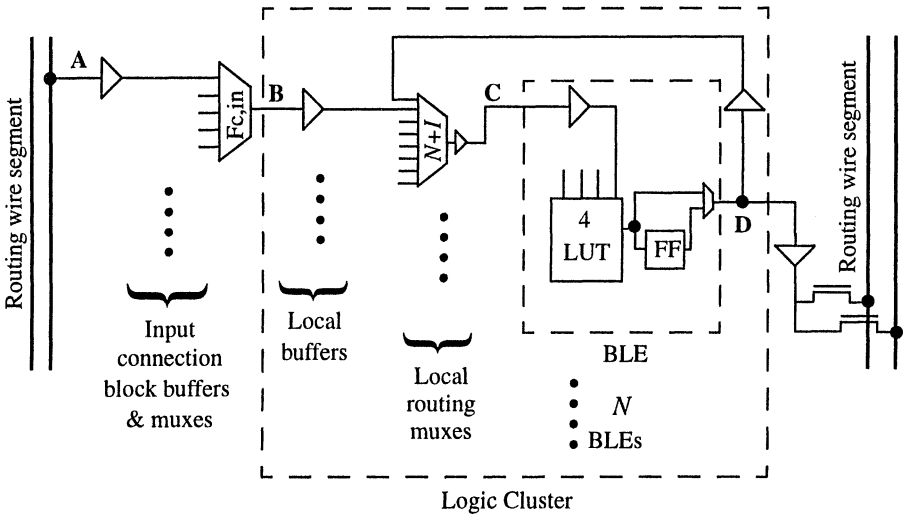


FIGURE 6.5 Structure and speed paths of a logic cluster.

TABLE 6.1 Important logic cluster delays in TSMC's 0.35 μm CMOS process.

Cluster Size (N)	A to B (ps)	B to C and D to C (ps)	C to D (ps)	B to D (ps)
1 (No local routing muxes)	760	55 (and no **D** to **C** path)	465	520
2	760	540	465	1005
4	760	675	465	1140
8	760	815	465	1280
16	760	970	465	1435
20	760	1000	465	1465

were again determined via SPICE simulations of the TSMC 0.35 μm process. For details of the procedure and circuit assumptions, see Appendix B. After a routing is complete, VPR uses these simplified models of pass transistors and buffers, as well as metal capacitance and resistance data, (all of which is specified in the architecture description file) to build an equivalent RC-tree for each net. It then computes the Elmore delay from the source to each of the sinks, as described in Section 4.5. We have found this linearized circuit model to be quite accurate: in Section 7.2.3 we

show that for a wide variety of different routing structures the delays computed by VPR are almost always within 9% of the delays computed by SPICE.

Finally, VPR performs a path-based timing-analysis using these delay values to determine the circuit critical path. Section 2.2.5 described the algorithms used in path-based timing analysis, and Section 4.5 described the VPR timing analyzer implementation.

6.2.4 Architecture Evaluation Metric: Area-Delay Product

One metric that we will use to evaluate the quality of different FPGA architectures is the area-delay product. This is a reasonable architecture metric for two reasons:

1. Intuitively, we want to find the point at which we are sacrificing the least amount of area for the most improvement in speed. Given that we can always trade area for speed (see below), and speed for area, it makes sense to combine these two factors into one curve to see where the best trade-off occurs.
2. The computational throughput of an FPGA (on a parallel algorithm) is simply the number of functional units multiplied by the clock speed. Another way of looking at this is, *throughput = (1/area per functional unit) · (1/delay)*. Therefore by minimizing the area-delay product, we maximize throughput.

There are two main factors which can affect the area-delay product of an FPGA: transistor sizing and the FPGA architecture. In general, the speed of an FPGA can be increased (to a point) by sizing up the buffers and transistors within the FPGA, but this increases area. Alternatively, the FPGA can be made smaller by sizing down the buffers and transistors, but this degrades the FPGA performance.

Throughout this chapter, we will size the transistors in each FPGA architecture to minimize the FPGA's area-delay product. Only by resizing transistors appropriately for each architecture in this way can we fairly compute the speed and area-efficiency of FPGAs with different logic block architectures.

6.2.5 FPGA Architectural Assumptions

To evaluate the speed and area of an FPGA we must choose not only the logic block architecture, but also a routing architecture and transistor sizes. The following sections detail all of our architectural choices.

Basic Architecture

We investigate island-style FPGAs in which each logic block is surrounded by routing channels on all four sides. The logic block input and output pins are evenly distributed around the logic block perimeter. Each circuit is mapped to the smallest square FPGA with enough logic blocks and pads to accommodate it.

Routing Architecture

We define the number of logic blocks which a routing segment spans as the logical length of that segment. In Chapter 7 we show that an architecture in which routing segments have a logical length of four, with 50% of the segments connected by tri-state buffers and 50% connected by pass-transistors, provides good area-efficiency and speed for FPGAs containing logic clusters of size four. This routing architecture is shown in Figure 6.6. We implicitly assume that this routing architecture is good for architectures containing logic clusters of all sizes, and we use this routing architecture in all of our experiments. Ideally, one would find the best routing architecture for each FPGA employing a different cluster size, but this would require a huge amount of effort. By basing all of our experiments on this routing architecture, we may slightly favor architectures with size four clusters over other architectures.

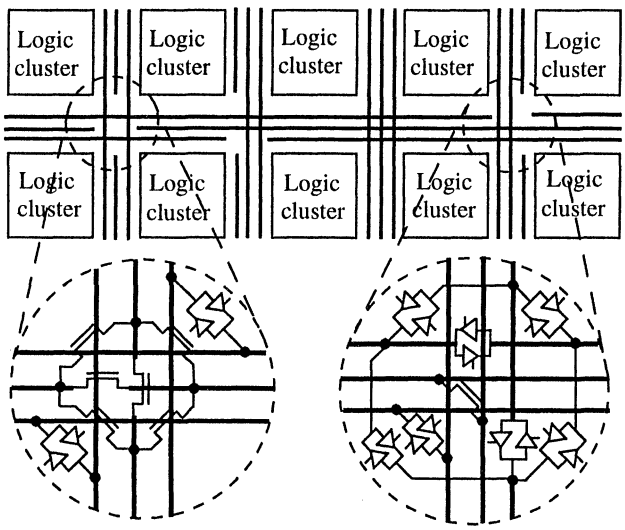


FIGURE 6.6 FPGA with length 4 segments, 50% buffered and 50% pass transistor switches.

Throughout this chapter we assume that metal all routing wires are laid out in metal 3, using the minimum width and spacing. See Appendix C.4 for a discussion of metal width and spacing issues in FPGAs.

Effect of Cluster Size on the Physical Length of FPGA Routing Segments

As we increase the cluster size, both the logic area per cluster and routing area per cluster grow. Figure 6.7 demonstrates how a tile (a logic block plus its associated routing) grows as cluster size is increased. This increased tile size results in routing segments with the same logical length having different physical lengths for logic clusters of different sizes.

We define the measured length of a routing segment as its physical length. The resistance and capacitance of a routing segment grow linearly with the segment's physical length. We have experimentally determined the average rate at which the FPGA tiles grow with cluster size, and have used this information to appropriately scale the routing segment resistance and capacitance values for the various cluster sizes. The increase in the resistance and capacitance of routing segments as the size, or granularity, of the FPGA logic block increases is an important effect that has often been neglected in prior FPGA architecture research.

Sizing Routing Transistors to Compensate for Different Physical Segment Lengths

To compensate for differences in the capacitance and resistance of routing segments in FPGAs using different sizes of logic clusters, we scale the routing pass transistors and buffers. All of our pass transistor and buffer scaling is in relation to a base archi-

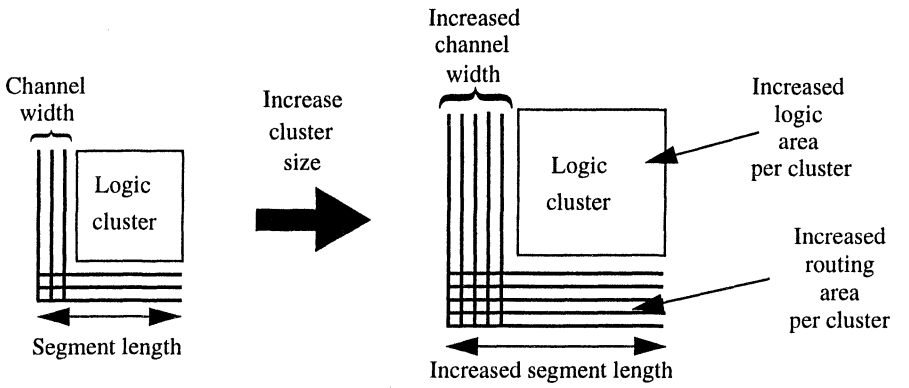


FIGURE 6.7 Effect of cluster size on physical length of routing segments.

tecture that has been area-delay optimized for clusters of size four. The method used to size routing transistors for this base architecture is described in Appendix C. From this base architecture, we linearly scale routing buffers and pass transistors depending on the relation between the new segment lengths and the base segment length. For example, in an FPGA with size 16 clusters, the physical segment length is approximately 2x longer than in an architecture with size 4 clusters. To maintain roughly the same speed per routing segment, we increase the size of the routing switches connecting to each wire by a factor of 2. In Section 6.5 we verify that this linear scaling of buffers and pass-transistors with physical segment length provides good results.

In our architecture models, we account for variations in delay caused by resizing buffers and pass-transistors. Also, changes in area due to the use of different sizes of routing pass-transistors and buffers are automatically calculated by VPR.

6.3 Cluster Inputs Required vs. Cluster Size

As discussed in Section 6.1, the first question we wish to answer is how many distinct inputs, I, should be provided to a cluster of size N. Since the number of transistors required to implement each of the multiplexers in the cluster local routing (see Figure 3.1(b)) grows linearly with I (for large I), we would like to make I as small as possible. On the other hand, if I is made too small, many of the BLEs in a logic cluster may become essentially unusable, reducing logic utilization and wasting area. We find the minimum value of I that allows good cluster utilization by running benchmark circuits through the first two steps shown in Figure 6.2, technology-mapping and cluster packing, and measuring the resulting logic utilization for different values of I. We define logic utilization to be the average number of BLEs per cluster that a circuit is able to use divided by the total number of BLEs per cluster, N.

Figure 6.8 shows how the average logic utilization of our 20 benchmarks varies with I for three different logic cluster sizes. The horizontal axis is the number of distinct inputs to the cluster relative to the total number of BLE inputs in a cluster, i.e. $I/(4N)$. For very low values of I, the logic utilization is very low, as one would expect. It is interesting, however, that when I is only 50 to 60% of the total number of BLE inputs, the logic utilization is essentially 100%. Clearly it is possible to pack BLEs together so that they have many common inputs and can reuse locally generated outputs. The relative amount of input sharing and output reuse increases slightly with logic cluster size, causing the curves in Figure 6.8 to shift to the left as cluster size increases.

The solid line in Figure 6.9 shows the value of I required to achieve 98% logic utilization as the cluster size, N, is varied, while the dashed line shows how the average

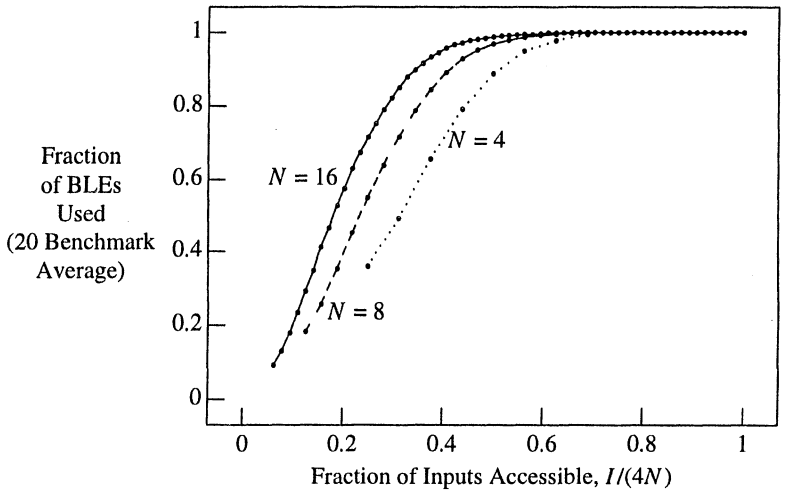


FIGURE 6.8 Logic utilization vs. number of logic cluster inputs.

number of logic cluster inputs that are actually used varies with cluster size. Although there are $4N$ BLE inputs in a logic cluster of size N, the number of inputs required to achieve 98% logic utilization is approximately $2N + 2$. Furthermore, the average number of logic cluster inputs that are actually used grows even more slowly. On average, a cluster of size 1 uses 3.57 of its inputs, while an cluster of size 20 uses only 25.2 of its inputs. In other words, while the logic per cluster has increased by a

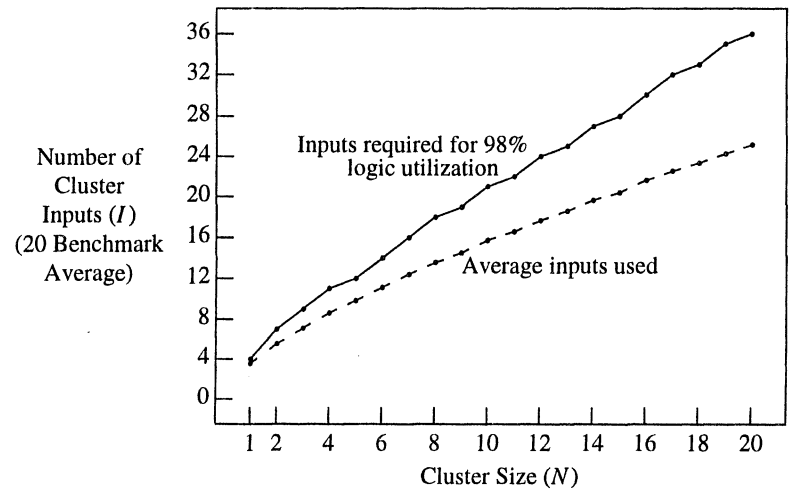


FIGURE 6.9 Variation in inputs required and inputs used with cluster size.

factor of 20, the average number of connections that must be routed to each cluster has increased by a factor of only 7.

Our results indicate that commercial FPGAs can be more aggressive in reducing the value of I. For example, the Altera Flex 8K FPGAs use logic clusters with $N = 8$ and $I = 24$, while our results indicate that $I = 18$ suffices for a cluster of this size. Similarly, the Xilinx 5200 FPGA uses a logic block very similar to a logic cluster with $N = 4$, and makes all 16 LUT inputs accessible, while our results suggest 10 to 11 inputs are sufficient. Reducing I in this manner simplifies the local routing multiplexers within a logic cluster and reduces the number of logic block pins that must be connected to the FPGA routing, resulting in considerable area savings.

6.4 *Flexibility of Logic Block to Routing Interconnect vs. Cluster Size*

Before we can apply the experimental flow of Section 6.2 to see how area-efficiency varies with cluster size, we must choose F_c, the number of routing tracks to which each logic block pin can connect. On the one hand, using a smaller value of F_c reduces the number of programmable switches in the FPGA routing, which improves area-efficiency. On the other hand, smaller values of F_c make an FPGA less routable so that larger channel capacities, W, will be required to successfully route circuits. This reduces area-efficiency by increasing the routing area. The goal is to choose a value of F_c that balances these two competing objectives and achieves good area-efficiency.

For a cluster of size 1, our experiments and those of Rose and Brown [30] have shown that a good value of F_c is W; i.e. each logic block pin can be connected to any routing track in an adjacent channel. For larger clusters, however, setting F_c to W provides far more routing flexibility than is required, wasting area.

Recall from Section 3.1.1 that the logic clusters we investigate are *fully-connected.* In other words, a BLE input can be connected to any cluster input or to the output of any of the BLEs within the cluster via the local routing. In a fully-connected logic cluster all the cluster inputs and all the cluster outputs are *logically-equivalent.* That is, all of the inputs are functionally identical, and all of the outputs are functionally identical. This means that a net which is an input to a cluster can be connected to *any* of the I cluster inputs, and a net which is driven by a cluster output can be connected to *any* of the N cluster outputs. Changing the connections made by the multiplexer-based local routing can compensate for any pin swapping performed on the cluster input pins by the router. Changing which BLE within the cluster generates each of the functions

required by the netlist can compensate for any pin swapping performed by the router on the cluster output pins. Therefore the router has a great deal of flexibility in how it routes inter-cluster nets.

The logical equivalence of cluster inputs and of cluster outputs means that keeping F_c fixed at W, regardless of the cluster size, *N*, results in an excessive number of ways to connect to large logic clusters. For example, a cluster of size one has 4 inputs and one output. If F_c = W, then, there are 4W ways to connect to a cluster input and W ways to connect to the cluster output. A cluster of size 20, on the other hand, has 36 inputs and 20 outputs, so there are 36W ways to connect to a cluster input and 20W ways to connect to a cluster output if F_c = W. This excessive routing flexibility for a cluster of size 20 wastes a large amount of routing area, since we have added many more programmable switches to the routing than is necessary.

We have experimentally found that a more appropriate level of routing flexibility results when the F_c value for logic block output pins, $F_{c,output}$ is set to W/N, and all the experiments in the next section use this value. This choice of $F_{c,output}$ means that each of the W routing tracks can be driven by one output pin on each logic block, ensuring that all the routing tracks in a channel can be readily used to interconnect blocks. Since connections to logic block input pins require less area than connections to logic block output pins (as described in Appendix B.1), it is best to decrease $F_{c,input}$ more slowly than $F_{c,output}$ as the cluster size increases. We reduce $F_{c,input}$ from W to 0.2W as the cluster size increases from 1 BLE to 20 BLEs.

6.5 *Speed and Area-Efficiency vs. Cluster Size*

We are now in a position to examine which cluster size leads to FPGAs with the best speed and area-efficiency. Throughout this section, the number of inputs, *I*, to a cluster of size *N* is chosen to be the minimum value that allows T-VPack to achieve 98% logic utilization. This value of *I* allows our logic clusters to be essentially fully utilized, while minimizing the complexity of the cluster input multiplexers and the number of logic block pins to be connected to the main FPGA routing. We ran 20 benchmark circuits through the experimental flow described in Section 6.2, and determined the area-efficiency and speed of each in FPGA architectures employing different logic cluster sizes. We present only the average results across all 20 circuits in the benchmark suite; the results for individual circuits track the average quite closely.

In Figures 6.10 and 6.11, we show the geometric average over the benchmark circuits of the total FPGA area required and the critical path delay, respectively. Note that we are showing three different routing transistor sizings in each of these graphs to ensure

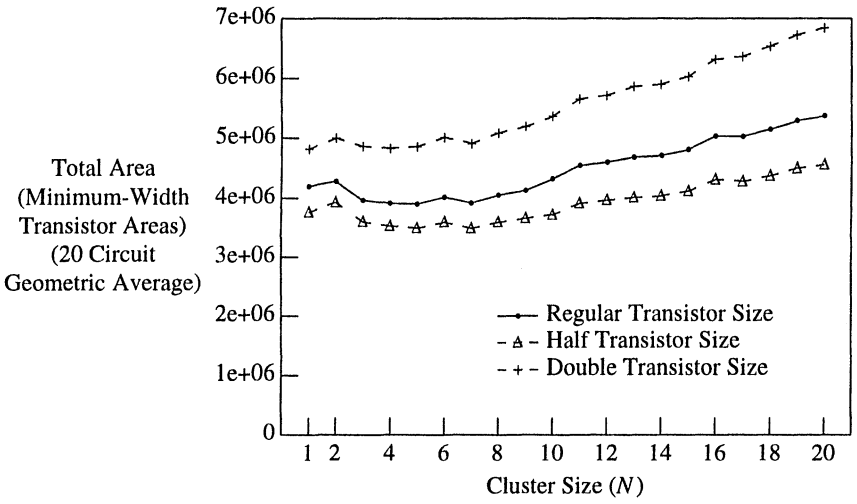


FIGURE 6.10 Total area vs. cluster size.

that we do not unfairly penalize any architecture with an inappropriate transistor sizing. The solid curves show the area and delay when we use the "normal" transistor sizing described in Section 6.2.5, while the dashed curves show the results when we use transistors that are one-half or double the size of those in the "normal" case.

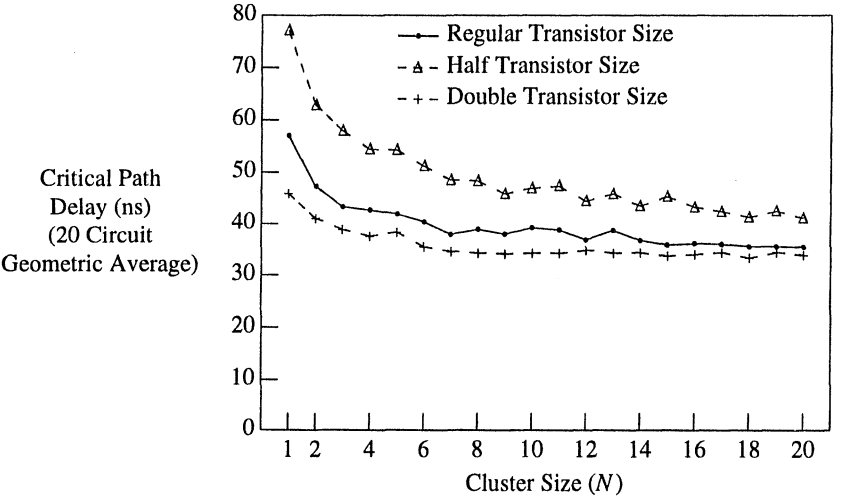


FIGURE 6.11 Critical path delay vs. cluster size.

Notice that we can indeed trade speed for area by resizing routing transistors — the half transistor size results have the smallest area, but the highest delay, while the double transistor size results have the smallest delay, but the highest area.

In Figure 6.10, all three curves show the same basic trend. Area initially increases as one moves from a cluster of size 1 to a cluster of size 2. This area increase occurs because a logic cluster of size 1 contains no local routing (it is a single BLE), while a cluster of size 2 does. The addition of this local routing to the FPGA requires a considerable amount of area, and at a cluster of size 2 it has not yet reduced the number of connections to route between clusters enough to compensate. Further increases in cluster size, to $N = 3$ and 4, improve area-efficiency because the local routing is able to more significantly reduce the amount of routing required between logic blocks. As the cluster size rises past $N = 8$ to 10, area-efficiency gradually degrades. The complexity of the local routing in a logic cluster grows quadratically with cluster size, and for sufficiently large clusters this swamps the area improvements gained by reducing the routing required between logic blocks.

Figure 6.11 shows that circuit speed increases significantly as we increase the cluster size. As one increases the cluster size from size 1 to 7, the circuit speed rapidly increases — with the "normal" transistor sizing, a size 7 logic cluster leads to circuits which are 51% faster than those implemented with a size 1 cluster. Increases in cluster size past $N = 7$ produce smaller incremental speed gains. For example, with the "normal" transistor sizing, a cluster of size 20 is 7% faster than a cluster of size 7.

In Figure 6.12, we show how the geometric average of the area-delay product achieved by the benchmark circuits varies with cluster size, again for three different transistor sizings. Notice that the "normal" transistor sizing provides the best area-delay product for all the architectures except a cluster size of 1, indicating that linearly scaling routing transistor size with the length of a layout tile is a good method to size transistors. For a cluster of size 1, however, the normal transistor sizing is smaller than optimal, and the double transistor size FPGA has a 7.5% lower area-delay product than the normal transistor size FPGA. There is a broad minimum in the area-delay product for cluster sizes from 4 to 10. A cluster of size 7 has the lowest area-delay product, but any cluster size between 4 and 10 is within 12% of the minimum, and hence would be a reasonable choice. Notice that moderate-size logic clusters significantly improve the area-delay product of an FPGA vs. using a single BLE logic block. Comparing a size 7 logic cluster (with the normal transistor sizing) to a size 1 logic cluster (with double-sized transistors — the best for this cluster size), one sees that the size 7 logic cluster has an area-delay product that is 33% lower than that of a size 1 cluster. An FPGA using a size 7 logic cluster is simultaneously 21% faster (a 17% delay reduction), and requires 19% less area than an FPGA using a size 1 logic cluster.

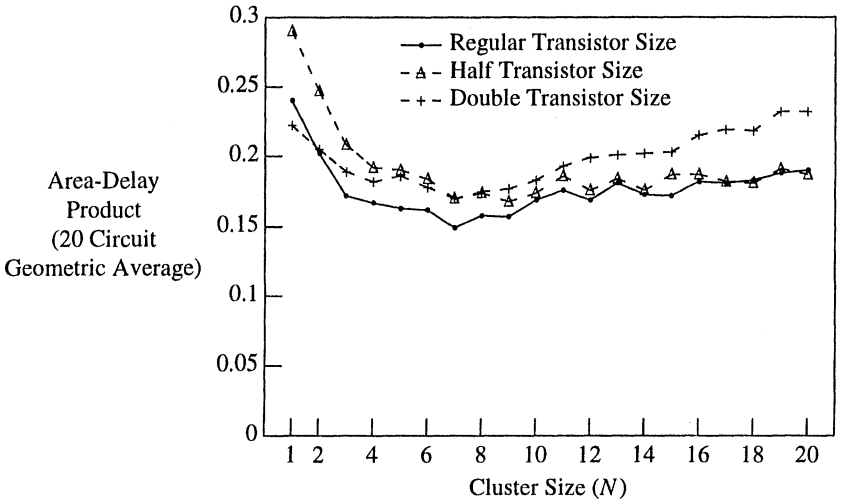


FIGURE 6.12 Area-delay product vs. cluster size.

6.5.1 Discussion of Delay vs. Cluster Size Results

In Figure 6.13 we show the relationship between the number of intra-cluster (fast) and inter-cluster (slower) connections on the critical path as a function of cluster size. As cluster size is increased, the number of intra-cluster connections on the critical path increases, and the number of inter-cluster connections decreases. This provides a circuit speedup since intra-cluster connections are faster than inter-cluster connections.[1]

Interestingly, the number of inter-cluster nets on the critical path does not decrease as much with cluster size as the inter-cluster delay decreases with cluster size (see Figure 6.14). From size 2 to size 20 we have a reduction in the number of inter-cluster nets on the critical path of 13% (Figure 6.13); compare this to the inter-cluster component of the critical path delay, which has been reduced by 39% over the same range (Figure 6.14). This means the circuit speedup visible in Figure 6.14 for larger cluster sizes is not only caused by a reduction in the number of inter-cluster connections on the critical path, but also by *inter-cluster connections on the critical path becoming faster.*

1. As cluster size is increased, intra-cluster multiplexer, buffer and wiring delays increase. If we were to increase the size of cluster to very large values, this effect would eventually result in intra-cluster delays becoming large enough that any gains obtained by making connections local to the cluster would be lost.

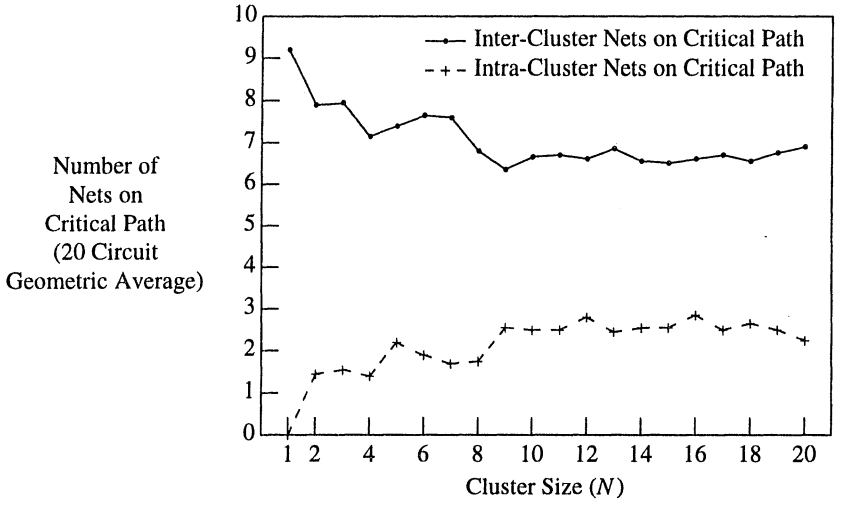


FIGURE 6.13 Inter- and intra-cluster nets on the critical path.

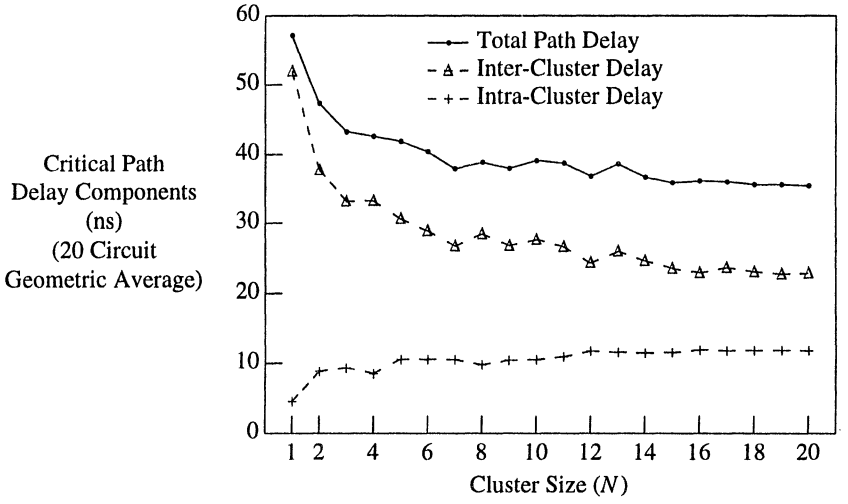


FIGURE 6.14 Breakdown of critical path delay into inter- and intra-cluster components.

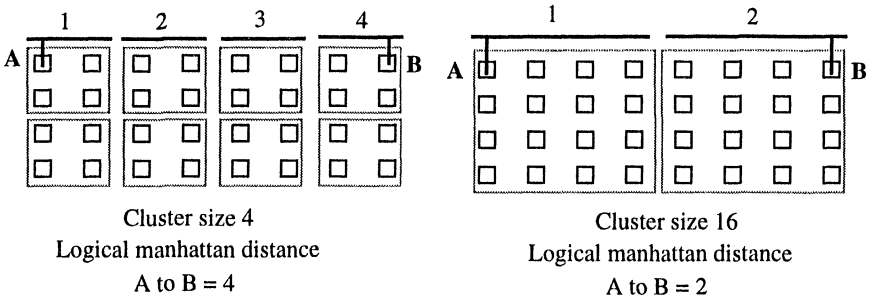


FIGURE 6.15 Decrease in logical manhattan distance as cluster size increases.

The improvement in inter-cluster delay with increased cluster size is caused primarily by a reduction in the "logical" manhattan distance spanned by connections in the FPGA, as shown in Figure 6.15. By sizing the routing pass transistors and buffers[1] to compensate for the increased physical length of routing wire segments associated with larger clusters, the delay of each routing segment has remained roughly constant. Since the total number of routing segments on the critical path has decreased due to the reduction of the "logical" manhattan distance, the result is a greater improvement in circuit delay than the reduction in the number of inter-cluster nets on the critical path would indicate.

6.6 *Effect of Cluster Size on Compile Time*

As the logic capacity of FPGAs increases and FPGAs capture larger and larger designs, compile times are increasing and threatening a key FPGA advantage: fast design spins. Consequently, FPGA architectures that lead to reduced circuit compile times are clearly desirable.

Figure 6.16 shows how the average CPU time (on a 300 MHz UltraSparc workstation) required to implement circuits varies with cluster size. The solid line in Figure 6.16 shows the total (packing, placement plus routing) compile time, while the two dashed lines show the placement and routing components of this compile time. The logic block packing time required is not significant — on average, T-VPack requires only about 1 second to pack a benchmark circuit.

1. Changes in delay and area due to different size routing buffers and pass transistors are accounted for in VPR's timing and area models.

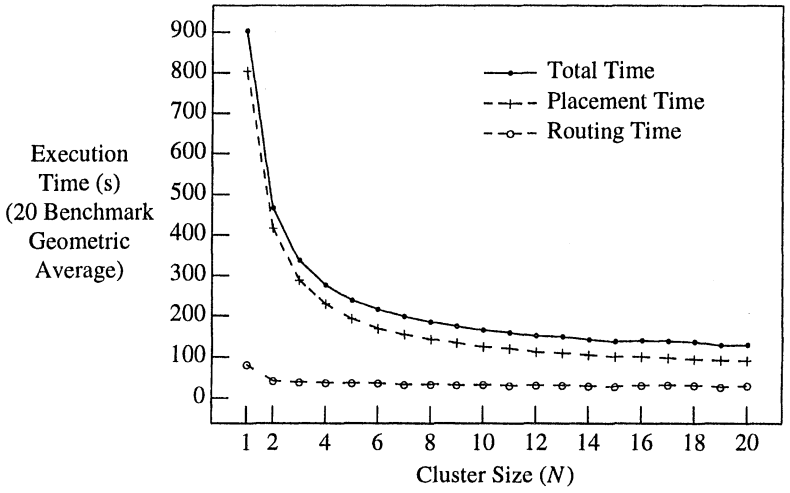


FIGURE 6.16 Variation of circuit compile time with logic cluster size.

Clearly, as larger logic clusters are employed in an FPGA the time to compile circuits is dramatically reduced. Building an FPGA with a size 20 logic cluster reduces the total CPU time required for placement and routing by *7x* vs. a size 1 logic cluster. As the number of BLEs per logic block increases, a circuit can be implemented with fewer logic blocks. Since the size of a placement problem is proportional to the number of logic blocks required to implement a circuit, this dramatically reduces placement time. In Figure 6.16, for example, one can see that the placement time is reduced by a factor of 8.8x as the cluster size increases from 1 to 20. Larger logic clusters also reduce the routing time. Since more connections are made via the local cluster routing in larger clusters, there are fewer inter-cluster connections to route. For example, using a size 20 logic cluster reduces routing time by 2.7x vs. using a size 1 cluster.

6.7 Summary

There are four main conclusions to be drawn from this work. First, the number of distinct inputs required by a logic cluster grows fairly slowly with cluster size, N. A cluster of size N requires approximately $2N + 2$ distinct inputs (for $N \leq 20$). Second, because all the input and output pins of a cluster are logically equivalent, one can significantly reduce the number of routing tracks to which each logic cluster pin can connect, F_c, as one increases the cluster size. We have found that a good value for

$F_{c,output}$ is W/*N*; it is best if the $F_{c,input}$ value is somewhat higher. Third, logic clusters containing between 4 and 10 BLEs all achieve good performance (as measured by their area-delay product) so any cluster in this range is a reasonable choice. Logic clusters in this size range achieve significant area and speed improvements relative to a single BLE logic block. For example, an FPGA employing a size 7 logic cluster requires 19% less area, achieves 21% higher speed, and has an area-delay product 33% lower than an FPGA using a single BLE logic block. Finally, large logic clusters significantly reduce design compile time. Size 7 and 20 logic clusters reduce placement and routing time by 4.6x and 7x compared to a single BLE logic block, respectively.

CHAPTER 7

Detailed Routing Architecture

In this chapter we explore a series of *detailed routing architectures* to find which ones lead to the best FPGA area and speed [14]. The detailed routing architecture of an FPGA specifies the length of every wire in the FPGA, the type of switch used to make every connection, the switch block topology, the metal width and spacing of each routing wire, and several other related parameters. In the next section we more precisely define all the parameters determining an FPGA's detailed routing architecture, and explain why detailed routing architecture issues are so crucial in FPGA design. Section 7.2 then describes the experimental flow we use to evaluate different routing architectures.

In Sections 7.3 through 7.7 we employ this flow to explore several key detailed routing architecture issues. We begin in Section 7.3 by investigating the best switch block topology and best F_c values for FPGA architectures that include wires that span more than one logic block. We also determine the best routing wire length when all wires in the FPGA have the same length in Section 7.3. In Section 7.4 we investigate FPGAs that contain two different lengths of routing wires, and a mix of pass-transistor and tri-state buffer routing switches. In Section 7.5 we evaluate the utility of internally depopulating wire segments, and determine the best amount of depopulation. Section 7.6 examines the speed gains attainable by increasing the spacing between some or all of the routing wires. Finally, Section 7.7 provides an overview of all the architectures examined in this chapter, comparing the speed and area achieved by the best architectures against a routing architecture similar to that of the Xilinx XC4000X series FPGAs.

7.1 Motivation

Most circuit delay in FPGAs is due to routing delays, rather than logic block delays, and most of an FPGA's area is devoted to routing [1]. In many ways, then, detailed routing architecture is *the key* architectural issue in FPGAs. When we refer to the *detailed routing architecture* of an FPGA we are referring to the values of a very wide range of parameters, including:

- The number of wires, F_c, to which each logic block input pin or output pin can connect,
- The switch block topology (which defines which wires can connect at a switch block),
- The segmentation distribution; that is, the different lengths of wire segments in the FPGA, and the fraction of tracks in a channel that are composed of wires of each length,
- The switch-block internal population and connection-block internal population of each wire segment,
- The type of switch (pass transistor or tri-state buffer) used to connect each routing wire to other wires,
- the sizes of the transistors used to build the various programmable switches, and
- the metal width and spacing of the various routing wires.

Recall that definitions of all the parameters listed above were provided in Section 2.1.3, and Section 4.2.1 provided a new and more general definition of internal switch population. The first four parameters above are topological parameters that define the connectivity of the FPGA routing architecture; the last two parameters are purely electrical, and the type of switch used to make each connection is both a topological and an electrical parameter.

Every parameter listed above is important, and the choice of each involves balancing complex trade-offs and interactions with other parameters. For example, if one chooses a segmentation distribution with too many short wires, long connections will have to be constructed using several short wires connected in series, resulting in poor speed. If the segmentation distribution includes too many long wires, however, some short connections will be forced to use long wires, degrading speed and wasting area. Similarly, an architecture with too many or too few tri-state buffer routing switches will clearly be suboptimal. Pass transistors require less area, and they are faster than buffers for short connections, but connections that pass through many switches are better served by tri-state buffers. As well, the best mix of routing switches is dependent on the segmentation distribution. A segmentation distribution with many long

wires will rarely have to connect many switches in series to make a connection. Consequently, an FPGA with many long wires can use a higher fraction of pass transistor-based switches in its routing than an architecture with a shorter segmentation distribution.

Previous research into detailed routing architecture (as described in Section 2.1.3) has generally focused on only one or two of the parameters listed above, and neglected the others. In this work we have tried to take a more holistic approach, investigating appropriate values for all of these parameters, and determining important parameter interactions. In addition, some important parameters have never been investigated before. For example, all prior studies have assumed that every routing switch is a pass transistor — the possibility of some routing switches being tri-state buffers has not been considered. Similarly, no prior work has considered varying the metal width and spacing of the routing wires. The internal population issue has been investigated only in a very limited way [21] — the only question addressed to date has been whether or not to completely depopulate length 3 wires. Similarly, while considerable work has been conducted on finding the best switch block topologies for FPGAs that contain only length 1 wires, there has been no research into the best switch blocks for use with FPGAs that contain longer wires.

In addition to examining a broader spectrum of FPGA architectures than prior researchers, we have also taken considerable care to create accurate area and delay models. Simplified and abstracted models can lead to questionable conclusions, particularly when we can compare such a vast array of widely different FPGAs. Consequently, we have used the "true" delay metric — critical path delay — and a highly detailed area model based on estimating the total transistor area required by a routing architecture.

Finally, in this study we spent considerable effort to ensure our CAD tools properly optimized for every routing architecture in the entire spectrum investigated. Hence we created a router that was both routability- and truly timing-driven (as described in Section 4.4) and understood issues such as segment lengths, buffer insertion, and metal capacitance and resistance. It is the flexibility of the VPR timing-driven router, as well as the flexibility of our area and delay models, that have enabled us to investigate such a broad spectrum of architectures.

7.2 *Experimental Methodology*

We explore different architectures by implementing the twenty largest MCNC benchmark circuits (which range in size from 1064 to 8383 BLEs) into each FPGA archi-

tecture of interest. We implement each circuit with an automatic CAD flow similar to that used by FPGA users: technology-independent logic optimization, technology-mapping, placement and routing. We then compare the circuit delay achieved and the area required by each architecture.

In the next section we describe the portions of the FPGA architecture, such as the logic block used, that are held constant throughout all the experiments of this chapter. The subsequent sections detail the CAD flow used to implement circuits in each FPGA, the area and delay models used to evaluate the result quality achieved by the various architectures, and the "experimental philosophy" that governed how we explored the huge detailed routing architecture design space.

7.2.1 FPGA Architectural Assumptions

In this chapter we are investigating different detailed routing architectures, so we hold the other architectural parameters, such as the logic block used, constant throughout the experiments. All the FPGAs we investigate are island-style FPGAs (described in Section 2.1.3), and each channel contains the same number of tracks and has the same segmentation distribution.

The logic block of all the FPGAs studied in this chapter is a logic cluster of four BLEs, with ten inputs and one clock. Using the notation of Section 3.1.1, then, the logic block is a cluster with $N = 4$, $I = 10$, $K = 4$ and $M_{clk} = 1$. This logic block is similar in size to the logic blocks employed in recent commercial island-style FPGAs [145, 13]. As well, in Section 6.5 we showed that this logic block has good speed and area-efficiency. The input and output pins are evenly distributed around the perimeter of the logic block, since the results of Section 5.3.1 showed this pin positioning is best.

The number of I/O pads that fit into the height or width of a logic block is set to four, in line with the relative sizes of pads and 4-LUTs of current FPGAs [4, 5, 7]. With this assumption three of the twenty benchmark circuits we use (bigkey, des and dsip) are pad-limited. We always map each circuit to the smallest square logic block array that has enough logic blocks and pads to accommodate it. Since commercial FPGAs normally distribute the circuit clock through a special, dedicated routing resource, we do not route the clock net in sequential circuits.

Finally, just as in Section 6.2.5, the size of the transistors used in the routing switches is a key architectural issue. We believe it is most appropriate to size these transistors to achieve the minimum area-delay product of the resulting routing resources. Appendix C details the transistor sizing experiments we conducted with the TSMC 0.35 μm process to find the best routing transistor sizes for use in an FPGA employ-

ing a size 4 logic cluster as its logic block. In Appendix C, we show that for a very wide range of routing wire segment lengths, tri-state buffers achieve the best area-delay product when their nMOS transistors have five times the drive strength of a minimum width nMOS transistor. To achieve equal rise and fall times, the pMOS transistors used in these buffers must be larger — 9.5 times the drive strength of a minimum width pMOS transistor. For routing switches composed of pass transistors, the best area-delay product is achieved when the transistors have ten times the drive strength of a minimum-width nMOS transistor. We use these routing transistor sizings throughout this chapter. Appendix C.4 also shows that to minimize the area-delay product of an FPGA one should use minimum width routing wires. Consequently all the results in this chapter assume minimum-width metal for routing wires.

7.2.2 CAD Flow

The CAD flow we use to evaluate routing architectures is almost identical to that of the previous chapter, as Figure 7.1 illustrates. SIS [142] and FlowMap [46] are used to optimize each circuit's logic and technology map each circuit into 4-LUTs and reg-

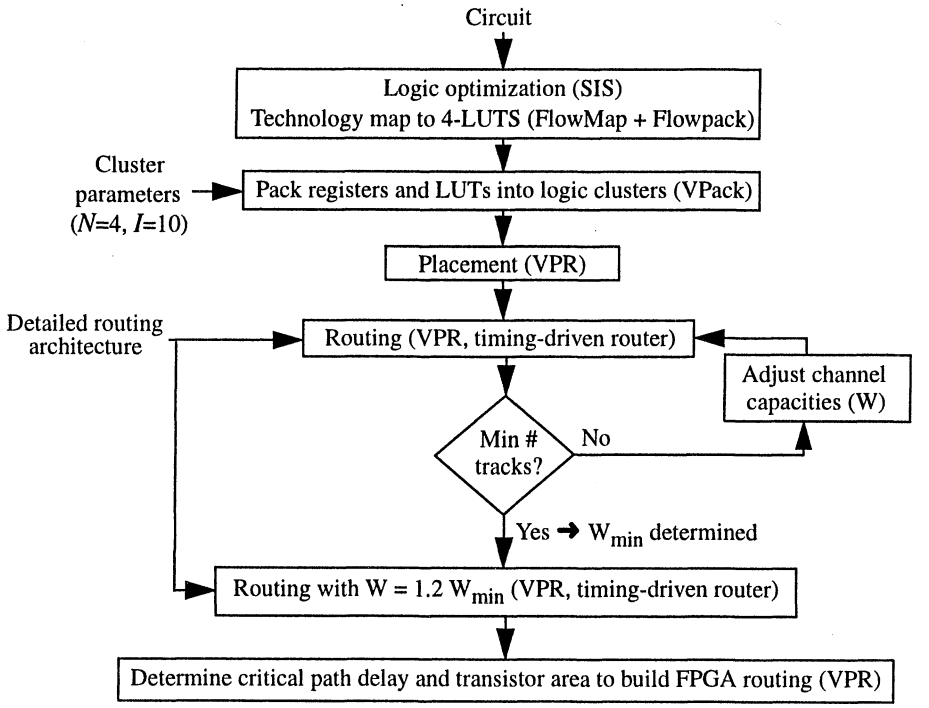


FIGURE 7.1 Architecture evaluation flow.

isters, respectively. VPack (described in Section 3.1.2) then groups these 4-LUTs and registers into logic clusters of the desired size (4 BLEs per cluster, using no more than 10 distinct inputs). While the timing-driven T-VPack tool is a superior to VPack, it was not completed at the time this study was done. Since throughout this chapter we always use the same logic block and vary only the routing architecture, however, it is not crucial that the logic block packing be the best possible. In other words, using T-VPack instead of VPack would make the circuits somewhat (~10%) faster in *every* routing architecture we evaluate, but should not affect the relative "goodness" rankings of the various routing architectures.

Once logic block packing is complete, VPR then places the circuit, and the VPR timing-driven router is repeatedly invoked with different channel capacities to determine the minimum number of tracks per channel, W_{min}, required to route the circuit. Since FPGAs are normally built with enough routing that "average" circuits have some spare routing available, we perform a final "low-stress" routing of each circuit with the number of tracks per channel set to $1.2 \cdot W_{min}$. We then apply our delay model to estimate the delay of the circuit critical path, and our area model to estimate the total transistor area needed to lay out all the routing in this FPGA. At this point, then, we have enough information to compare both the speed and the area-efficiency of one routing architecture to another.

7.2.3 Delay Model Accuracy

Our delay model was described in detail in Section 6.2.3. Recall that we model pass transistors and buffers in the FPGA routing with simplified RC equivalent circuits and compute the delay of a routed connection as the Elmore delay of this equivalent circuit.

Tables 7.1 and 7.2 compare the delays computed by VPR, using these linearized RC models of transistors and buffers and the Elmore delay, to those computed by SPICE for various routing structures. Overall, the accuracy of our delay model is very good; the VPR delays are within 9% of those of SPICE for all but two cases. Those two cases consist of very short connections: a signal passing through a logic block output buffer to a wire segment of length 1 or length 4 with the routing switches loading this wire being pass transistors. The signal does not pass through any switch block routing switches. While the relative error in these two cases is 23.8% and 14.1% vs. SPICE, the delays are very small, so the absolute error in delay is only 112 ps and 89 ps, respectively. Consequently, these errors in delay will have little impact on the critical path delay reported by VPR.

TABLE 7.1 Delays computed by VPR vs. SPICE delays for buffered routing resources.

Wire Segment Length (in Logic Blocks)	VPR Delay (ns)	SPICE Delay (ns)	Difference
1	0.448	0.472	-5.1%
4	0.777	0.850	-8.6%
8	1.24	1.34	-7.5%
16	2.27	2.42	-6.2%
64	11.50	11.02	-7.9%

TABLE 7.2 Delays computed VPR vs. SPICE delays for pass-transistor-switched routing resources.

Wire segment length (in logic blocks)	Number of Wire Segments in Series	VPR Delay (ns)	SPICE Delay (ns)	Difference
1	1	0.359	0.471	-23.8%
4	1	0.543	0.632	-14.1%
8	1	0.816	0.876	-6.8%
16	1	1.46	1.47	-0.7%
1	4	0.883	0.92	-4.0%
4	4	2.32	2.14	+8.4%
8	4	4.70	4.33	+8.5%
16	4	11.04	10.61	+4.1%
1	16	7.03	6.48	+8.5%
4	16	24.06	22.10	+8.9%
8	16	54.06	52.03	+4.0%
16	16	139.6	141.5	-1.3%

7.2.4 Area Model

In this chapter we once again use the transistor-based area model described in Section 6.2.2. This model is built into VPR; once VPR has generated the routing resource

graph for the desired architecture, it traverses this graph, "builds" every structure required by the FPGA's routing, and determines the total number of minimum-width transistor areas required by the FPGA routing. To allow averaging of results from circuits of different sizes, we use a normalized area metric: the number of minimum-width transistor areas per tile (i.e. per logic block). All the results in this chapter give only the routing area of the FPGA, since the logic block is held constant throughout all the experiments. The logic block used in this chapter occupies 1678 minimum-width transistor areas, and hence the addition of 1678 to any of the routing area results presented in this chapter yields the total area per tile.

Importance of a Detailed Area Model

Most prior researchers have evaluated routing area either by counting the number of tracks per channel required to successfully route, or by counting the number of programmable switches in the routing. Counting the number of tracks required to route a circuit is not a good area metric for architecture studies (such as this study) in which the number of switches per track segment can vary, since the area required by each routing track is then variable. Counting the number of programmable switches in the routing is a better area metric, but is still not sufficiently accurate for our purposes. Modern FPGAs use three different types of programmable switch, and the different switches require considerably different layout areas. The connection blocks from routing tracks to logic block input pins are implemented with multiplexers; the connection blocks from logic block output pins to routing tracks, and some of the routing switches, are implemented via pass transistors; and some routing switches are tri-state buffers. Table 7.3 lists the area required by each of these switch types, including any area required by SRAM bits to control each switch. The area per switch varies by a *factor of 6.8* from the most area-efficient switch to the least area-efficient. Clearly, simply counting the number of programmable switches in a routing architecture does not provide a good area estimate.

TABLE 7.3 Comparison of programmable switch areas.

Switch Description	Minimum-Width Transistor Areas
Multiplexer (32 inputs)	2.88 per switch (92 / 32 inputs)
Multiplexer (4 inputs)	4.5 per switch (18 / 4 inputs)
Pass transistor (10x minimum drive)	11.5
Tri-state buffer (5x minimum drive)	19.7

7.2.5 Experimental Philosophy

The detailed routing architecture design space is huge, so we cannot simply explore the entire N-dimensional space of all possible detailed routing architectures. Instead, we perform a series of experiments in which we vary one or two architectural parameters, while fixing the other parameters to reasonable values. While varying a single parameter, we are optimizing along a line in the N-dimensional architecture space. Once we have found the best value of one parameter, we set it to this value, and vary a different parameter. Essentially we are exploring the design space in a manner similar to the way matrix solvers like the conjugate-gradient method explore the solution space of a matrix.

We begin with relatively simple architectures, and search for the best values of the parameters that are unlikely to interact strongly with other architectural parameters. We can then fix these parameters at their best values, and vary other architectural parameters to create increasingly complex architectures.

7.3 Single Wire Length Architectures

In this section we evaluate architectures in which every routing wire segment has the same length, and in which all the routing switches in switch blocks are tri-state buffers. Our goals in this section are to:

1. Determine the most appropriate switch block for segmented architectures,
2. Gain insight into the wire segment length that is most important in an FPGA, and
3. Find the best F_c value(s) for use with segmented architectures and the logic block used in this chapter (a cluster of size 4).

We ran the twenty largest MCNC benchmarks through the flow of Figure 7.1 and determined the routing area required and the critical path delay achieved by each circuit in each architecture of interest. Figure 7.2 plots the average (over the twenty circuits) of the critical path delay for each architecture, while Figure 7.3 plots the average routing area required in each architecture. The horizontal axis in both Figures 7.2 and 7.3 is the length of the routing wire segments. Recall that each architecture in Figures 7.2 and 7.3 potentially contains a different number of tracks per channel — each circuit is routed in an FPGA with 20% more routing tracks than the minimum the circuit needs to route in an FPGA with the given architecture.

In both Figures 7.2 and 7.3, the solid lines use the disjoint switch block topology [137] described in Section 4.2.3, while the dashed lines use the Wilton switch block

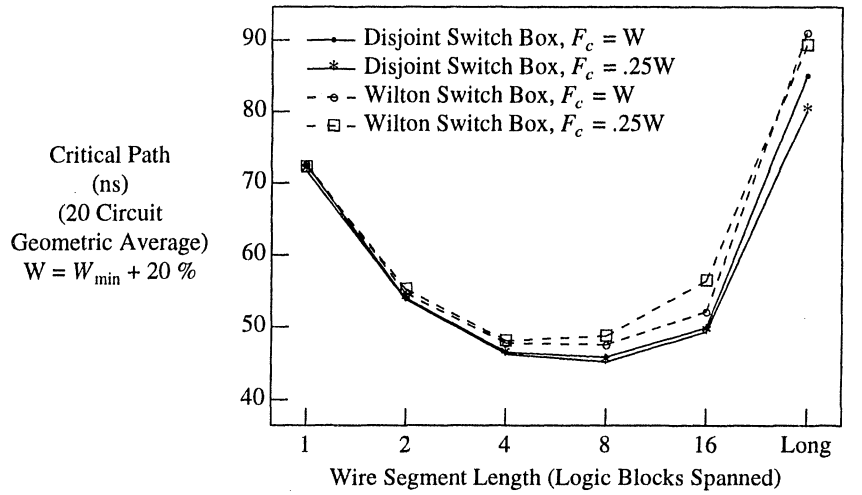


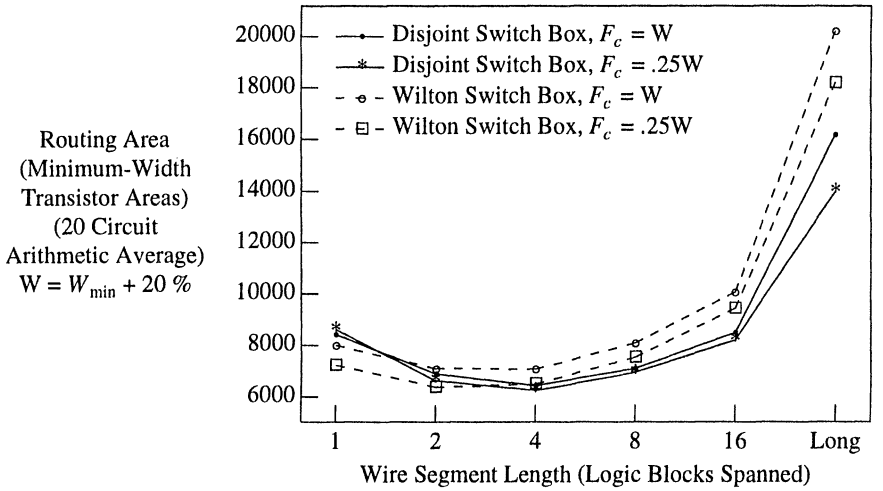
FIGURE 7.2 Circuit speed vs. switch block topology, F_c, and routing segment length.

topology [33]. Each curve plots the routing area or the critical path delay versus the length (in logic blocks) of the routing segments used in the FPGA. The "long" point on the horizontal axis indicates the performance of a routing architecture when all its wires are long lines that span the entire chip.

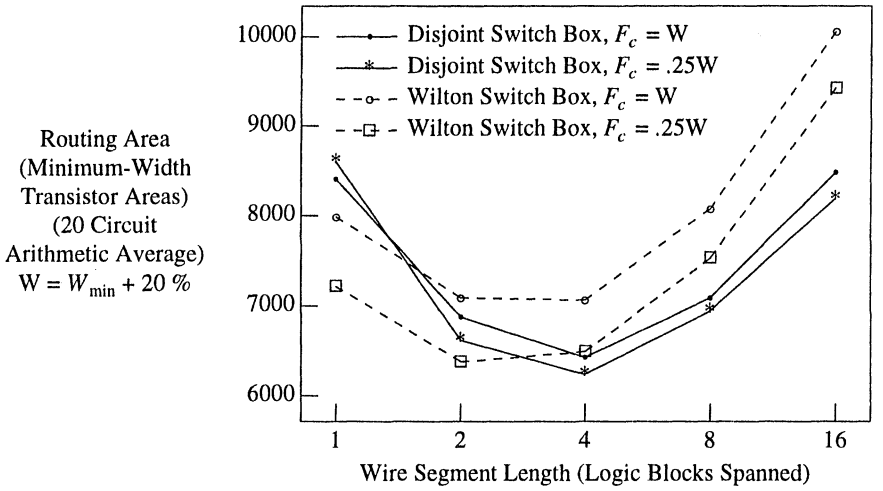
7.3.1 Switch Block Issues

The first feature of interest in Figures 7.2 and 7.3 is the performance of the two different switch blocks. Notice that the critical path delay achieved by both switch blocks is about the same for FPGAs composed of length 1 wires. However, as the length of the wire segments increases, the disjoint switch block leads to faster circuits than the Wilton switch block. Similarly, Figure 7.3 shows that the Wilton switch block leads to the best area-efficiency when all the routing wires span only one logic block. As the wire length increases to 2 logic blocks, however, the area required by the Wilton and disjoint switch blocks become similar and for wire lengths greater than 2 the disjoint switch block is more area-efficient than the Wilton switch block.

When a wire terminates at a switch block, both the disjoint and Wilton switch block topologies allow it to connect to three other wires, (i.e. F_s = 3). Hence both switch block topologies require the same number of switches in this case, as Figure 7.4(a) shows. Longer wires, however, do not always terminate at a switch block — they often pass through the switch block. Figure 7.4(b) shows this case. For the disjoint switch block, the six routing switches required by the two horizontal wire segments in



(a) Wire segment lengths from 1 logic block to the entire chip



(b) Close-up of wire segment lengths from 1 to 16 logic blocks

FIGURE 7.3 Routing area vs. routing architecture: (a) all wire lengths; (b) close-up.

Figure 7.4(a) now all connect the same two pieces of metal together. Consequently we only need to implement *one* of these switches — the other five are redundant. In the Wilton switch block case, however, only two of the six switches that would be inserted if the wires ended at this point are redundant — a total of *four* routing

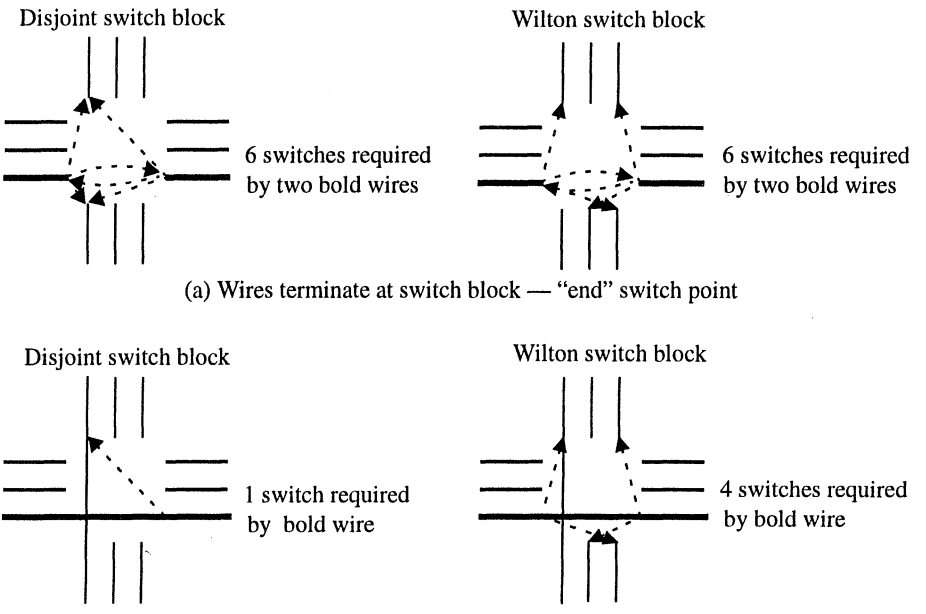


FIGURE 7.4 Switch block examples at (a) end and (b) interior of wires with length greater than 1.

switches are still required for this wire segment at this switch block. So at "internal" switch blocks, where a wire segment passes through the switch block without terminating, the disjoint switch block has an $F_{s,internal}$ of one, while the Wilton switch block has an $F_{s,internal}$ of up to four. This is a key point — switch blocks that use the same number of switches for FPGAs in which all wires are of length one can use *different* numbers of switches in FPGAs with longer wires. As well, it is generally not possible to describe switch block topologies with a single F_s number when some routing wires span more than one logic block.

The reason for the behaviour seen in Figures 7.2 and 7.3 is now clear. If all the wires in an FPGA have length one, the Wilton and disjoint switch blocks use the same number of switches per wire. The Wilton switch block results in better routability [33], however, so it requires fewer wires per channel, and hence less area. As the wire length increases, however, we have more and more wires passing through switch blocks without terminating, and hence more and more "internal" switch points. Since the Wilton switch block uses more switches at these points than the disjoint switch block, it loads the wires more heavily, degrading speed. As well, the reduction in the

number of wires per channel required to successfully route a circuit is not large enough to outweigh the area of these extra switches, so the Wilton topology's area-efficiency is worse than that of a disjoint topology for FPGAs with longer wires.

These switch block results are interesting because prior researchers have focused on determining the best switch block for architectures that contain only length 1 wires. Clearly the best switch block for such an architecture is *not* necessarily the best switch block for architectures containing longer wires, so research into good switch blocks for FPGAs with longer wires is a fertile area for research. Throughout the remainder of this chapter we will use the disjoint switch block. This switch block topology is almost certainly suboptimal, however. The fact that this switch block only allows wires in track *i* to connect to other track *i* wires reduces routability somewhat, and it means that wires of differing lengths can never be connected together. The routing of some nets would likely benefit from the ability to connect wires of differing lengths. For example, one intuitively expects that a good way to route a high-fanout net is to use long wire segments to create a "backbone" that spans the FPGA, and then use shorter wires to reach any sinks that are not immediately adjacent to the backbone. VPR can route FPGAs with arbitrary switch blocks, so a useful future research project would be to employ our tools in a search for better switch blocks for FPGAs that contain longer wires.

7.3.2 Best Single Wire Length

From Figure 7.2, one can see that the fastest FPGA which uses only one length of wire uses wires of length 4 or length 8, regardless of the switch block and F_c value used. Shorter wires lead to poor speed because long connections must pass through too many buffers. Very long wires degrade speed in two ways. First, short connections are forced to use long wires, which are slower than short wires due to their larger capacitance. Second, even connections that travel the entire length of a long wire become slow when the wire is too long because the metal resistance of the wire becomes large, and eventually reduces speed below that of a larger number of short wires connected by buffers.

Figure 7.5 quantifies the relative speeds of short and long wires, and provides insight into why FPGAs using length 4 or length 8 wires have the best performance. Figure 7.5 shows the delay *per logic block* spanned by a connection as a function of the length of wire used, when the routing switches between wires are tri-state buffers. Increasing the wire length from 1 to 2 logic blocks increases the speed of long connections by 61% by reducing the number of routing switches in series by a factor of 2. As wire length increases past 4 logic blocks we clearly have diminishing speed returns. Note that a wire of length 32 (for metal layer 2) or of length 64 (for metal layer 3) is *slower,* even for long connections that span the entire wire, than the series

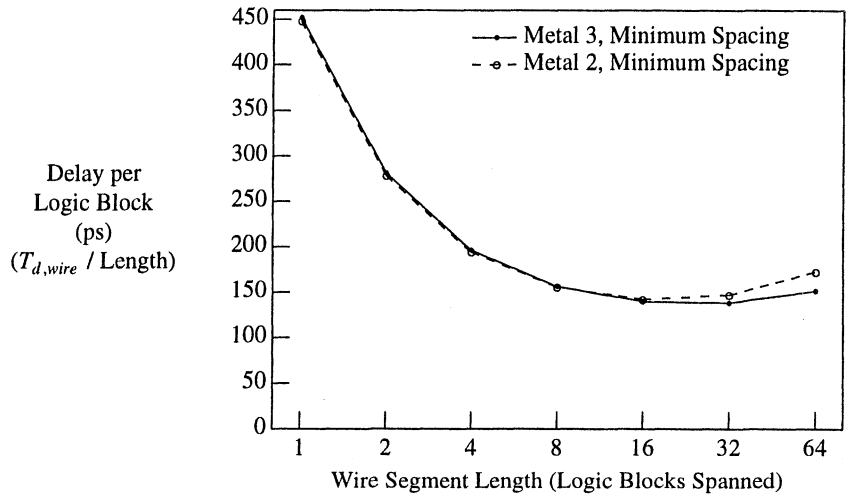


FIGURE 7.5 Delay per logic block spanned by a connection vs. wire segment length.

connection of several shorter wires. Since metal 3 has a lower resistance, wires can be made somewhat longer (64 logic blocks rather than 32) before the metal resistance slows them down more than extra routing switches would. Since very long wires are also less flexible than short wires in terms of the connections for which they can be used, it is clear that very long wires are not very useful for FPGAs in modern IC processes.

Figure 7.3 shows that wires of length 2 or length 4 lead to the most area-efficient FPGA architecture. Length 2 wires are the most area-efficient if the Wilton switch block topology is used, while length 4 wires are the most area-efficient if the disjoint switch block topology is used. As we increase the length of the routing wires two competing factors determine the resulting architecture's area-efficiency. First, longer wires are less "flexible"; they cannot be split in the middle, so short connections will waste part of a wire segment. This means the number of tracks per channel required to successfully route a circuit increases as the wire segment length increases. On the other hand, longer wires pass through more switch blocks before terminating, so the fraction of "internal" switch points in switch blocks increases. As Figure 7.4 shows, these internal switch points require fewer programmable switches, resulting in decreased area. When the disjoint switch block is employed, each internal switch point requires only one programmable switch; with a Wilton switch block, internal switch points require up to 4 programmable switches. Consequently, longer wire segments reduce the number of switches per wire more dramatically with a disjoint switch block than with a Wilton switch block.

Length 4 wires achieve the best combination of low delay and high area-efficiency. An architecture using all length 8 wires can achieve slightly (~2%) better speed, but requires 11% more area. While real FPGA architectures can of course use more than one wire length, this result is evocative. It leads one to expect that the best FPGA architectures will either include significant numbers of length 4 or length 8 wires, or will include some wires shorter and some wires longer than length 4.

7.3.3 Amount of Connectivity Between Logic Blocks and Channels

Figure 7.2 shows that F_c has little effect on circuit speed. Figure 7.3 shows that an F_c value of 0.25·W is superior in terms of area-efficiency to F_c = W for any segment length if the Wilton switch block topology is used. If the disjoint switch block topology is used, F_c = 0.25·W is superior to F_c = W for all wire segment lengths except length = 1.

Figure 7.6 shows how routing area varies with F_c when all routing wires span 4 logic blocks (we use length 4 wires since they were found to be the most area-efficient in the previous section). For the Wilton switch block topology, the best area-efficiency occurs at F_c = 0.25·W. When the disjoint switch block topology is used, however, the best value of F_c is 0.5·W. Since the disjoint switch block is less routable than the Wilton switch block, a higher value of connection block flexibility is needed for the best area-efficiency.

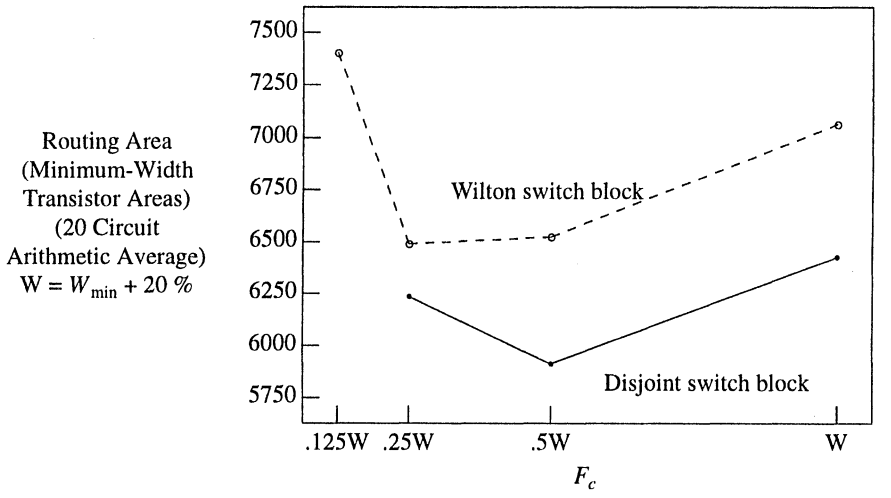


FIGURE 7.6 Routing area vs. F_c for a routing architecture with all length 4 lines.

With the disjoint switch block topology some additional routing area can be saved by using a lower value F_c value for logic block output pins ($F_{c,output}$) than for input pins ($F_{c,input}$). Recall that the connection block from a logic block output pin to routing tracks is constructed via $F_{c,output}$ pass transistors controlled by $F_{c,output}$ SRAM cells. The connection block from routing tracks to a logic block input pin, on the other hand, is constructed via a multiplexer with $F_{c,input}$ inputs. The area per switch is therefore higher for the logic block output pin switches than for the switches to logic block input pins, as Table 7.3 shows. Consequently, making $F_{c,output}$ lower than $F_{c,input}$ makes intuitive sense. We have found that, with the disjoint switch block, the best value of $F_{c,output}$ is 0.25·W, while the best value of $F_{c,input}$ is 0.5·W. This lower value of $F_{c,output}$ saves some routing area (2% - 5%, depending on the exact routing architecture) versus using an $F_{c,output}$ of 0.5·W. An $F_{c,output}$ value of 0.25·W ensures that each routing track can be driven by one output on each logic block (since our logic block has four logically-equivalent outputs).

7.4 Two Types of Wire Segment Architectures

In this section we examine somewhat more complex architectures: those that contain two *types* of wire segments. Two wires are of different types if their lengths are different, or if they use different types of routing switches to connect to other wires (e.g. pass transistors vs. tri-state buffers). All the experiments in this section use the disjoint switch block topology and an F_c value of 0.5·W, since the previous section showed these are good choices for a wide range of architectures.

7.4.1 Tri-State Buffer Routing Switches Only

We investigated a large number of architectures that contained two different lengths of routing wires, and in which all the routing switches in each switch block were tri-state buffers. We found that we could achieve only small improvements compared to the best single wire length architecture (length = 4). Table 7.4 compares the performance of the two best architectures investigated in this section to the best single wire length architecture. Both these architectures are fairly similar to an architecture in which all wires have length 4 — one has 25% length 2 wires and 75% length 8 wires, while the other has 75% length 4 wires and 25% length 8 wires. Three columns in Table 7.4 compare the speeds of these architectures to that of a length 4 architecture under three different assumptions of the metal layer and spacing used for routing wire segments. The average speedup vs. a length 4 architecture is only 5.3% for the first architecture, and 5.7% for the second. Both of these architectures are slightly *less*

dense than an architecture that contains only length 4 wires. Clearly length 4 wires provide an efficient way to make both short and long connections!

TABLE 7.4 Best buffered, two different wire length architectures vs. best single wire length architecture. Average over 20 benchmark circuits.

Architecture	Critical Path Delay (ns)			Routing Area (Minimum-Width Transistor Areas)
	Metal 2, Min. Spacing	Metal 3, Min. Spacing	Metal 3, Wide Spacing	
All length 4, buffered	45.26	45.62	39.81	5901
25% length 2, 75% length 8	43.45	43.74	37.08	6034
75% length 4, 25% length 8	43.18	43.44	37.06	5948

7.4.2 Length 4 Buffered Wires Plus Pass-Transistor-Switched Wires

Since a routing architecture composed solely of length 4 wires that use tri-state buffers as their routing switches performs so well, in this section we investigate architectures in which some routing tracks contain wires of this type. The other routing tracks contain wires that connect to each other with pass transistor routing switches. We will investigate different lengths of these "pass-transistor-switched" wires, and different proportions of the two types of wires.

Figure 7.7 shows the speed achieved by FPGA architectures in which 50% of the routing tracks use length 4 wires connected by buffered switches, and the other 50% consist of some other length of wires connected with pass transistors. The horizontal axis in Figure 7.7 is the length of the pass-transistor wires used. The three different curves in Figure 7.7 assume three different metal layer / spacing options for routing wires; clearly the metal width and spacing used does not significantly change the shape of the curve.

Figure 7.8 provides the area results for these architectures. The best area-efficiency occurs when the pass-transistor-switched wires are length 2, but length 4 and length 8 wires also have reasonable area-efficiency, and Figure 7.7 showed they lead to superior speed.

Figures 7.9 and 7.10 investigate the performance of length 4 buffered wires combined with either length 1, 2, 4, or 8 pass-transistor-switched wires in different proportions, (i.e. not just 50 / 50). The horizontal axis in these figures is the fraction of routing tracks composed of the pass-transistor-switched wires; the remainder of the routing

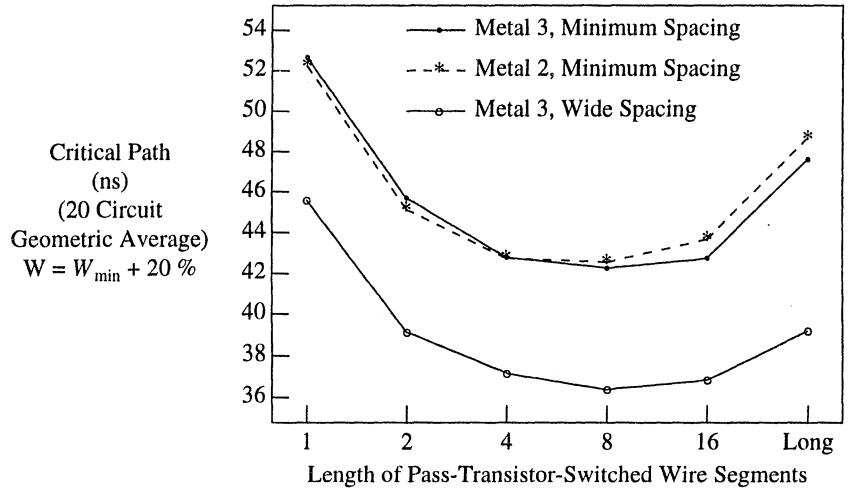


FIGURE 7.7 Speed of architectures with 50% length 4 buffered and 50% pass-transistor-switched wires.

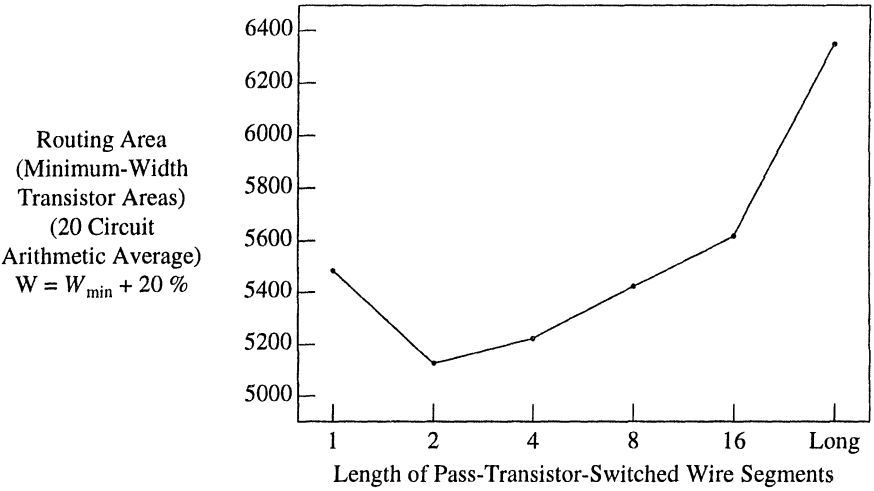


FIGURE 7.8 Area of architectures with 50% length 4 buffered and 50% pass-transistor-switched wires.

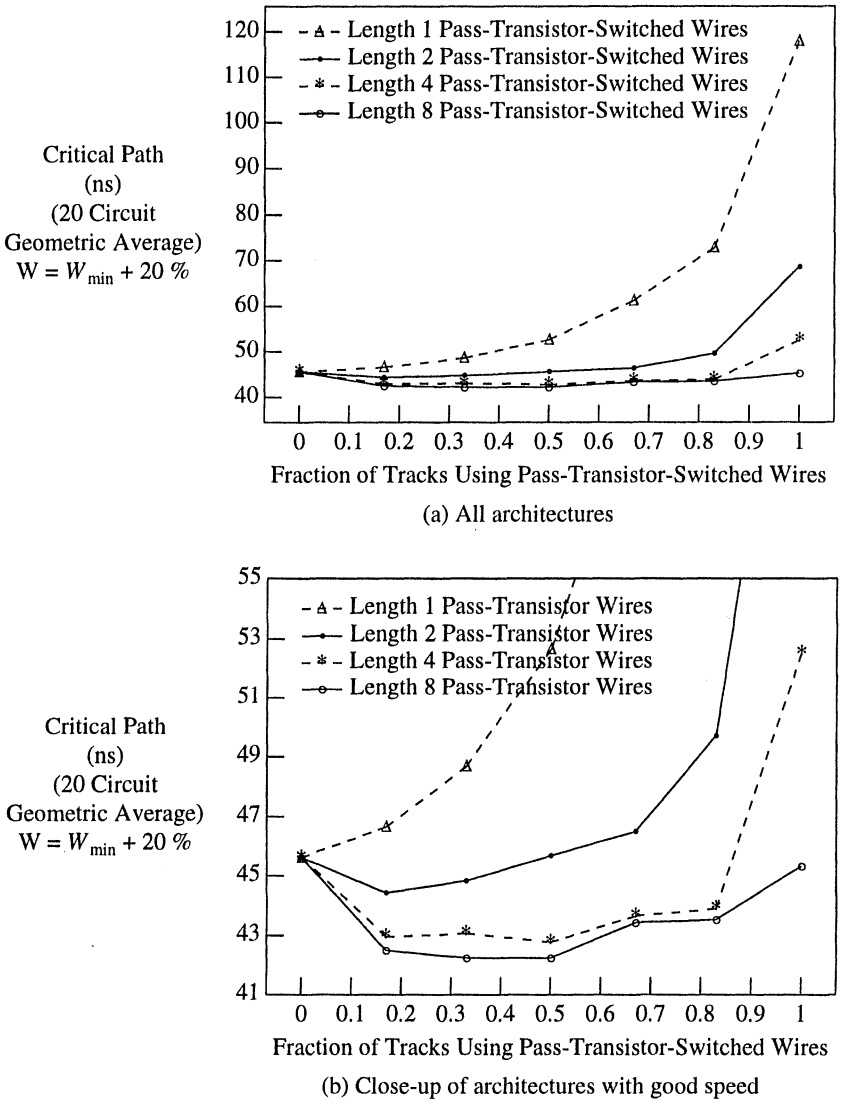


FIGURE 7.9 Speed of FPGAs with a mix of length 4 buffered wires and pass-transistor-switched wires.

tracks are composed of length 4 buffered wires. The "0" point on the horizontal axis corresponds to an architecture composed solely of length 4 buffered wires, while the "1" point corresponds to architectures composed solely of wires that connect to each other via pass transistors. Figure 7.9 shows the speed of the various architectures,

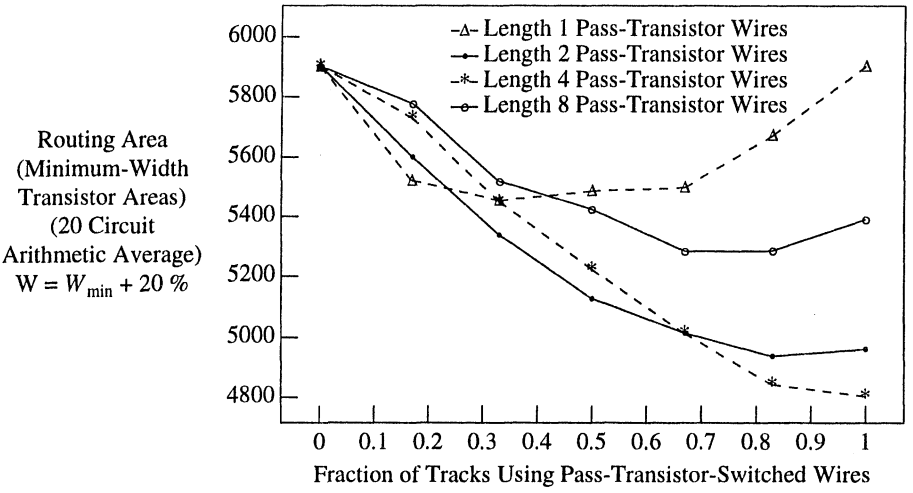


FIGURE 7.10 Area of FPGAs with a mix of length 4 buffered wires and pass-transistor-switched wires.

assuming routing wires are implemented in metal 3 with the minimum metal spacing, while Figure 7.10 shows their area. Clearly, adding length 1 wires to an architecture is not a good idea. If 33% of the routing tracks are length 1 pass-transistor-switched wires, then area-efficiency improves by 8% (vs. all tracks being length 4 buffered wires), but speed degrades by 7%. Larger fractions of length 1 wires degrade both area and speed — these wires are simply too short to be of much use. Many commercial architectures make heavy use of length 1 wires [4, 139], but our results suggest they could improve both their speed and area-efficiency by using longer wires instead. Note that the most widely studied architecture in academic research, an architecture composed entirely of length 1 wires connected by pass transistors, has extremely poor speed — it is 2.8 times slower than the fastest architecture in Figure 7.9.

Adding longer pass-transistor-switched wires to an architecture yields better results. Figure 7.9 shows that making between 17% and 83% of the routing tracks pass-transistor-switched wires of length 4 or 8 increases the FPGA speed. Pass transistors do not have the intrinsic delay of the multi-stage buffers used in buffered routing switches, and they have higher drive strength than a tri-state buffer for the same area, since they use only one transistor, rather than several. On the other hand, the delay through a series chain of pass transistors grows quadratically with the number of pass transistors. The net effect is that pass transistor switches are faster than buffers for connections that pass through a few series switches, but buffers are faster for connections that pass through many series switches. If long wires are used, fewer series routing switches are needed for long connections, making pass transistor switches

more competitive with buffers for longer connections. Consequently a mix of moderate length buffered and moderate length pass-transistor wires leads to better speed than using all buffered routing switches or all pass transistor routing switches.

Figure 7.10 shows that increasing the fraction of routing tracks using length 2, 4 or 8 pass-transistor wires improves the FPGA area-efficiency until this fraction reaches approximately 83%; after that area-efficiency degrades (or levels off, for length 4 wires). A pass transistor switch requires less area than a tri-state buffer, and since pass transistors are bidirectional, one pass transistor can replace two tri-state buffers in the routing. On the other hand, pass transistor switches are not well-suited to routing high-fanout nets, as described in Section 4.4.4. To maintain reasonable speed, a high-fanout net routed using pass-transistor switches tends to use a "star" topology. This requires more wiring, and hence more routing tracks, and hence more area. Making all routing switches pass transistors forces even high-fanout nets to be routed using pass transistors, degrading area-efficiency.

TABLE 7.5 Best architectures combining length 4 buffered and pass-transistor-switched wires.

Wire Distribution of Architecture	Delay (ns)	Speedup vs. Length 4 Buffered Architecture	Area (Minimum-Width Transistor Areas)	Area vs. Length 4 Buffered Architecture
100% Length 4 Buffered	45.62	—	5901	—
50% Length 4 Buffered, 50% Length 4 Pass Transistor	42.78	+6.6%	5223	-11.5%
17% Length 4 Buffered, 83% Length 4 Pass Transistor	43.90	+3.9%	4843	-17.9%
50% Length 4 Buffered, 50% Length 8 Pass Transistor	42.25	+8.0%	5424	-8.1%
17% Length 4 Buffered, 83% Length 8 Pass Transistor	43.54	+4.8%	5285	-10.4%

Considering both area and speed, the best architectures use 50% - 83% pass-transistor switched wires, with these wires having a length of 4 or 8. Table 7.5 summarizes the performance of several of the best architectures found in this section, and lists performance of an FPGA using all length 4 buffered wires for comparison. The area improvement due to a mix of pass transistors and buffers ranges from 8% to 18%, while the speed improvement ranges from 4% to 8%. The architectures with 83% pass-transistor switched routing tracks have better area-efficiency, at the cost of

slightly reduced speed, compared to FPGAs with only 50% pass-transistor switched routing tracks.

A major conclusion of this section is simply that the best routing architecture contains a mix of pass transistors and tri-state buffers. This fact is not widely known. Virtually all prior academic research has focused on FPGAs that contain only pass transistors, while a new FPGA start-up company has made the fact that their FPGAs contain no pass transistors (all routing switches are tri-state buffers) a marketing feature [146].

7.4.3 Length 8 Buffered Wires Plus Pass-Transistor-Switched Wires

In Section 7.3.2 we found that an FPGA composed entirely of length 8 wires interconnected by tri-state buffers performed almost as well as an FPGA that used only length 4 wires. Consequently, in this section we investigate routing architectures in which some routing tracks are composed of length 8 buffered wires, while the other routing tracks are composed of wires connected by pass transistors (and which may have a length different than 8). Our goals are to find the best length of pass-transistor switched wire to combine with length 8 buffered wires, and to find the best proportion of pass-transistor-switched routing tracks to length 8 buffered routing tracks.

The architectural conclusions of this section are very similar to those of the previous section:

1. The best combination of area and delay results when the pass-transistor switched wires are of length 4 or 8.
2. The best architectures contain from 50% to 83% pass-transistor-switched routing tracks, with the 50% pass-transistor architectures giving the best speed, and the 83% pass-transistor architectures yielding the best area-efficiency.

Table 7.6 summarizes the performance of the best architectures that combine length 8 buffered wires with some other length of pass-transistor switched wires. The performance of an architecture using all length 8 buffered wires is also listed for comparison. Comparing these results to those of Section 7.4.2 shows that using length 8 buffered wires instead of length 4 buffered wires has slightly increased the FPGA speed, at the cost of slightly decreased area-efficiency.

7.4.4 Length 4 Pass-Transistor-Switched Wires Plus Buffered Wires

Recall from Section 7.2.5 that we are exploring the highly complex detailed routing architecture space by fixing most of the architectural parameters and varying only one

TABLE 7.6 Best architectures combining length 8 buffered and pass-transistor-switched wires.

Wire Distribution of Architecture	Delay (ns)	Speedup vs. Length 8 Buffered Architecture	Area (Minimum-Width Transistor Areas)	Area vs. Length 8 Buffered Architecture
100% Length 8 Buffered	45.03	—	6339	—
50% Length 8 Buffered, 50% Length 4 Pass Transistor	41.29	+9.1%	5227	-17.5%
17% Length 8 Buffered, 83% Length 4 Pass Transistor	43.69	+3.1%	4869	-23.2%
50% Length 8 Buffered, 50% Length 8 Pass Transistor	41.56	+8.3%	5732	-9.6%
17% Length 8 Buffered, 83% Length 8 Pass Transistor	43.15	+4.4%	5447	-14.1%

or two in a set of experiments. The danger in such an exploration approach is that we may miss some important architectures by fixing some parameters at values that initial experiments led us to believe were reasonable, but which were suboptimal for the more complex architectures studied later.

To guard against such occurrences, we often investigated a certain portion of the architecture space from more than one perspective. For example, in this section we again investigate FPGAs in which some of the routing tracks consist of wires that connect via pass-transistor switches, while the other routing tracks connect via tri-state buffers. In this section, however, we assume that the pass-transistor-switched wires are of length 4, and search for the length of buffered wiring segment that best complements these wires and leads to the best FPGA.

Figure 7.11 shows how circuit delay varies with the length of the buffered wiring segments, while Figure 7.12 illustrates the variation of FPGA routing area with the length of the buffered routing segments. In both Figures 7.11 and 7.12, 50% of the routing tracks are composed of length 4 wires that connect via pass-transistor switches, while the other 50% of the routing tracks use some length of buffered wire segment. Figure 7.11 shows that the best delay occurs when the buffered wire segments have length 8, although length 4 and length 16 wire segments lead to FPGAs that are only slightly slower. Figure 7.12 indicates that the best area-efficiency occurs when the buffered wire segments have length 4 or length 8.

Considering both speed and area-efficiency, the best architecture studied in this section combines length 4 pass-transistor wires with length 8 buffered wires, and the second best architecture uses length 4 buffered wires instead of length 8 buffered wires. The fact that length 8 and length 4 buffered wires were again found to be the best

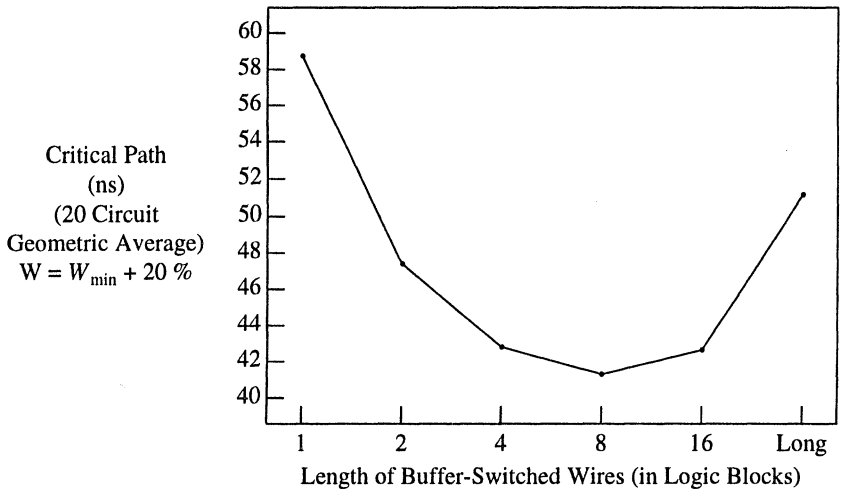


FIGURE 7.11 Speed of FPGAs using 50% length 4 pass-transistor wires and 50% buffered wires.

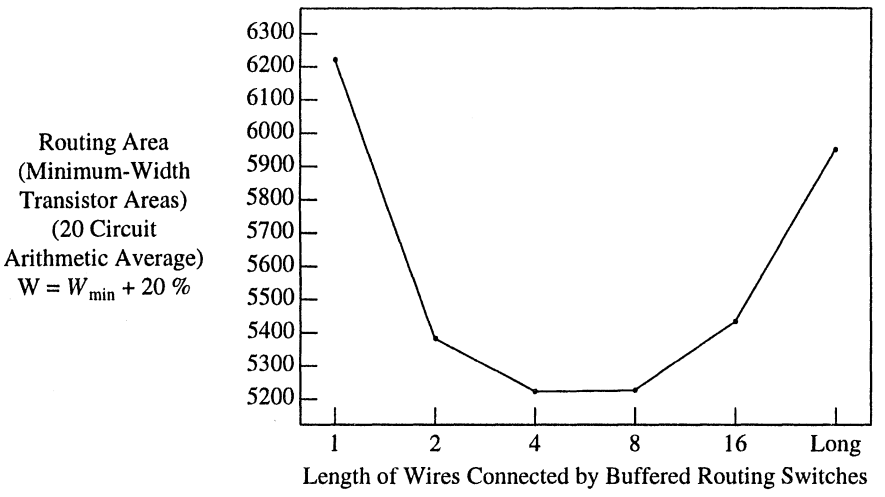


FIGURE 7.12 Area of FPGAs using 50% length 4 pass-transistor wires and 50% buffered wires.

increases our confidence that we fully explored the most promising two-wire-type architectures in Sections 7.4.2 and 7.4.3.

7.5 Internal Population

In this section we investigate the internal population question, which was described in Section 4.2.1. Recall that the connection-block internal population of a wire specifies how many of the internal points along the length of a wire that spans multiple logic blocks are accessible by switches to logic blocks. Similarly, switch-block internal population specifies how many of the internal points along a wire can connect to other wires via routing switches. In this section we determine what fraction of an FPGA's routing tracks should be composed of internally-depopulated wires, and exactly how depopulated these wires should be.

In all the experiments of this section we assume that routing wires are laid out in metal 3 with minimum spacing. $F_{c,output}$ is always 0.25·W, $F_{c,input}$ is always 0.5·W, and the switch block topology is disjoint.

7.5.1 All Length 4 Buffered Wires

In Section 7.3.2 we found that the best FPGA architecture that employs only one length of wire connected by buffers was an FPGA with length 4 wires. Consequently, in this section we investigate architectures that use all length 4 wires connected by tri-state buffers with varying amounts of connection-block and switch-block internal population.

We first performed a set of experiments in which all the wires in the architecture had the same amount of internal population. We found that if the wires were more than slightly depopulated, some circuits became completely unroutable (would not route at *any* channel width). More specifically, if the connection-block population is less than 75% (a wire connects to fewer than 3 out of the 4 logic blocks it spans) some circuits become unroutable. Similarly, if the switch-block population is less than 80% (a wire connects to other wires at fewer than 4 out of the 5 switch blocks it spans) some circuits will not route. Figure 7.13 shows why reducing the connection-block population to 50% makes some circuits unroutable. In Figure 7.13 the logic block has only one output, which is located on the bottom side of the logic block. Notice that when all the routing wires are length 4 and have a connection-block population of 50%, there is *no possible way* to connect many logic blocks. For example, logic block *A* in Figure 7.13 cannot connect to either of the shaded logic blocks. Similar, but more

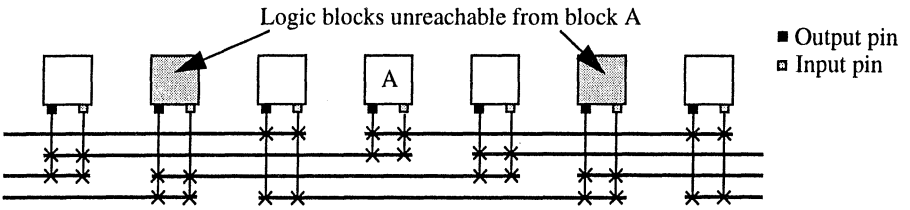


FIGURE 7.13 An FPGA with all length 4 wires and 50% connection-block population.

subtle effects render many circuits unroutable in FPGAs in which all the wires have less than 80% switch-block population. Clearly then one should not depopulate all the wiring tracks in an FPGA — some wires should remain fully populated to guarantee it is possible to connect every pair of logic blocks in the FPGA.[1]

Figures 7.14 and 7.15 show how circuit speed and routing area, respectively, are affected as the internal switch population of 50% of the routing wires in an FPGA is

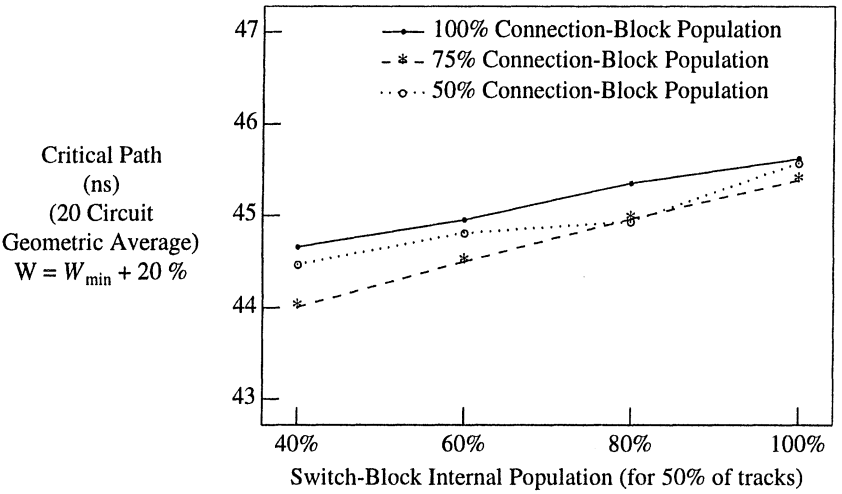


FIGURE 7.14 Speed of architectures with 50% of routing tracks depopulated in some manner.

1. A useful direction for future research would be to explore wire segments with differing amounts of internal connection-block population for logic block inputs and outputs. Switches to logic block output pins require the most area, so depopulating these switches yields the largest area gains. Keeping logic block input pin switches fully populated for a wire guarantees that a logic block can always reach any other logic block even if all wires are internally output connection-block depopulated.

varied. The other 50% of the routing wires are always fully populated. Each line in Figures 7.14 and 7.15 corresponds to a different amount of connection-block population for 50% of the routing wires, while the switch-block internal population for these wires varies along the horizontal axis. Figure 7.14 indicates that reducing the internal population of routing wires leads to a small (2% - 3%) speed gain. While reducing the internal population of a wire reduces its capacitive loading somewhat, most of the capacitance loading a wire is the metal capacitance of the wire itself, and this limits the achievable speedup.[1] As well, as the internal population of a wire segment is reduced, some nets may have to use more circuitous routes to connect their terminals because of the reduced flexibility of the routing fabric. This tends to decrease speed, and offsets some of the speed gain due to reduced capacitive loading.

Figure 7.15 details the effect of internal population on routing area. As the internal population of wires is reduced, the number of switches per wire drops, saving routing area. On the other hand, more tracks per channel are required to successfully route circuits because of the reduced flexibility of the routing, and this increases routing area. The best routing area is achieved when the internal population of the routing wires best balances these two factors. Figure 7.15 shows that reducing the connec-

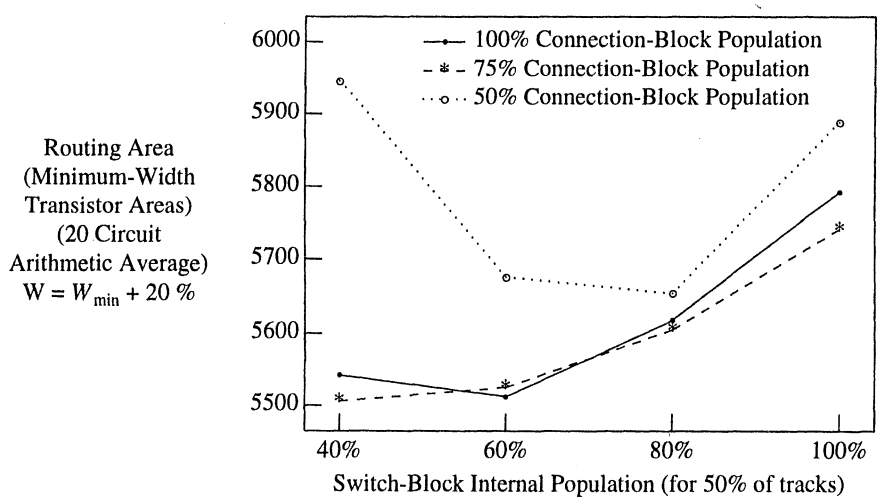


FIGURE 7.15 Area of architectures with 50% of routing tracks depopulated in some manner.

1. Larger speed gains are possible if one sizes up the routing transistors connecting to the internally-depopulated wire segments, as this reduces the portion of the delay due to metal capacitance. This requires additional area, but may be desirable when an FPGA architect is more concerned with speed than area.

tion-block population from 100% to 75% has only a marginal impact on area-efficiency, and reducing it further, to 50%, degrades area-efficiency. Reducing switch-block population is more effective — there is a clear area win as the switch-block internal population is reduced from 100% to 60%. Further reductions to a switch-block population of 40% provide little or no area gain. The three best architectures are those in which 50% of the routing tracks have:

- 100% connection-block population, and 60% switch-block population, or
- 75% connection-block population and 60% switch-block population, or
- 75% connection-block population and 40% switch-block population.

Figure 7.16 shows the average number of tracks per channel required to successfully route the benchmark circuits (W_{min}) for the various architectures. Notice that W_{min} increases as the internal population (of 50% of the wires) decreases. The three architectures listed above require roughly the same transistor area, but the 100% connection-block population, 60% switch-block population architecture requires the lowest number of routing tracks per channel and hence the least metal area. This probably makes it slightly preferable to the other two architectures.

Figure 7.17 shows why we would intuitively expect an FPGA that contains some length 4 wires with 60% switch-block population to have good routability. These wires can always be interconnected so that two sides of any logic block in the FPGA can be reached from any point. Since the input and output pins on our logic blocks

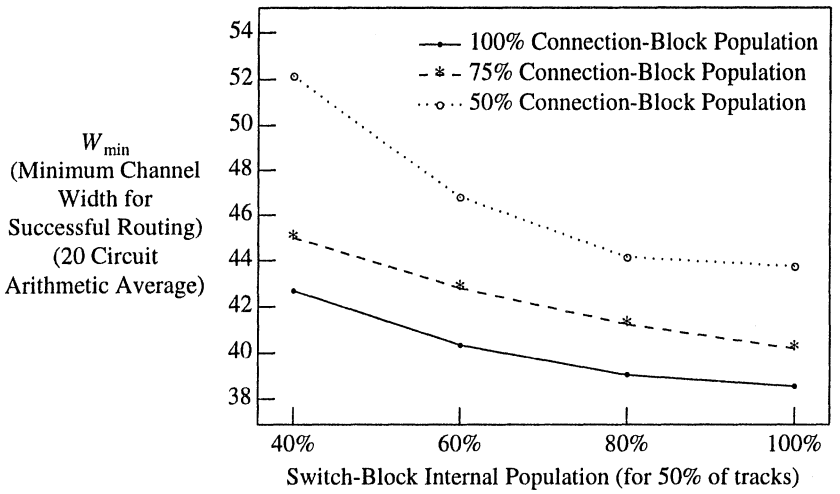


FIGURE 7.16 W_{min} for architectures with 50% of routing tracks depopulated in some manner.

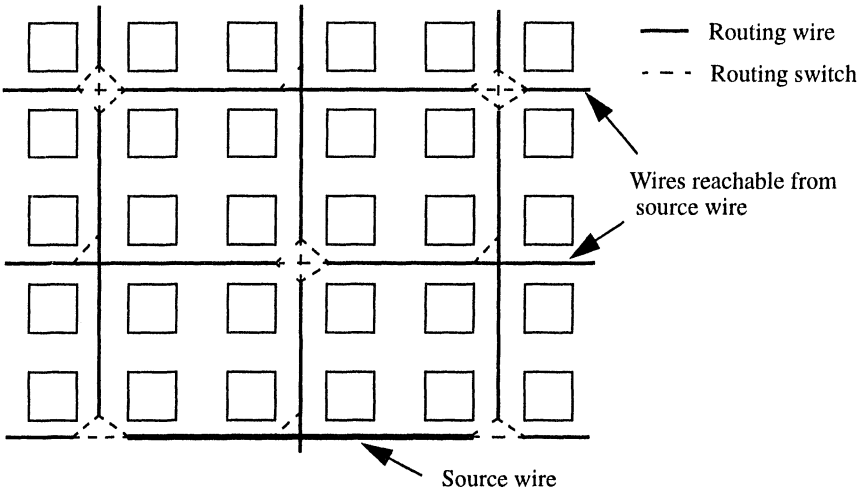


FIGURE 7.17 Length 4 wires with 60% switch-block population can reach 2 sides of any logic block.

are all logically-equivalent, a connection can use any logic block input or output pin. Consequently, as long as all the pins on the "reachable" sides of a target logic block are not already used, the connection can be made to one of these "reachable" sides. Some wiring tracks must have higher than 60% internal population, however, so that they can reach any side of a logic block and route connections when all the pins on one side of a logic block have already been used. Note that wires with lower switch-block population (e.g. 40%) cannot reach *any* side of some logic blocks from some points, significantly reducing routing flexibility.

In the experiments above, we arbitrarily decided that only 50% of the routing tracks would be internally depopulated in some way, while the other 50% of routing tracks were fully populated. We found that for such architectures, a connection-block population of 100% and a switch block population of 60% was a good choice for the depopulated wire segments. In Figures 7.18 and 7.19, we vary the proportions of these two types of wire segments: some wire segments have a switch-block population of 60% while the remaining wire segments have a switch block population of 100%. All wire segments have a connection-block population of 100%.

Figure 7.18 shows the how speed varies as the fraction of tracks that have only 60% internal switch-block population is varied. Again, the speed increase is small — only about 2%. Notice also that essentially all of this speed gain has already occurred when only 17% of the routing tracks have been internally-depopulated. This indicates

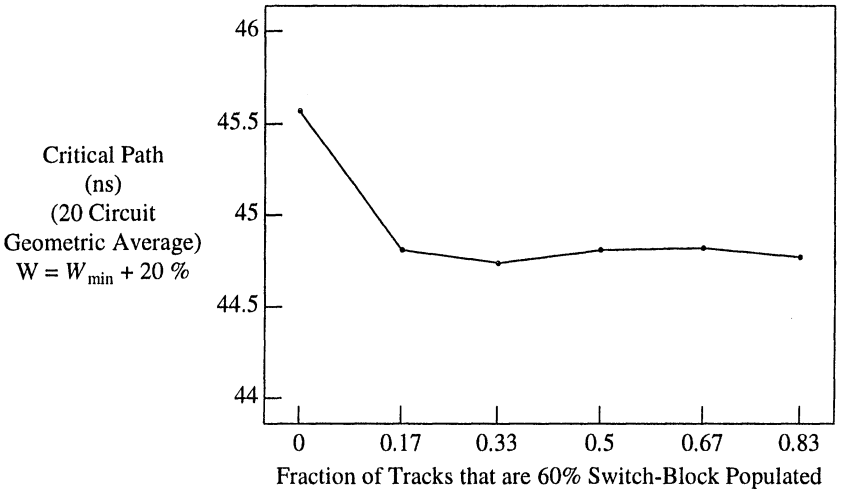


FIGURE 7.18 Speed versus the fraction of routing tracks with a switch-block population of 60%.

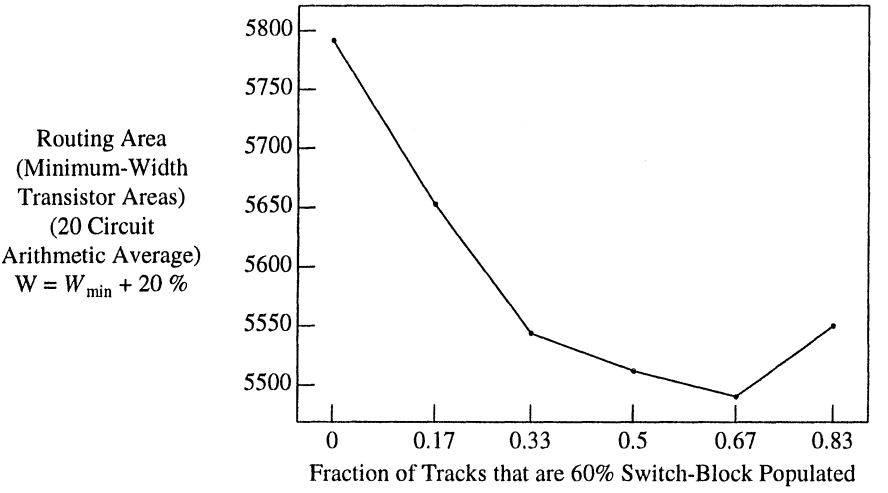


FIGURE 7.19 Area versus the fraction of routing tracks with a switch-block population of 60%.

that our router is able to correctly identify the time-critical nets and move them onto the fastest resource available.

Figure 7.19 shows how routing area varies with the fraction of tracks that have only 60% internal switch-block population. The best area occurs when 67% of the routing tracks have been depopulated in this way — any more and the increase in tracks required to successfully route a circuit outweighs the reduction in the average number of switches per wire. Using a switch-block population of 60% for two-thirds of the routing tracks yields a 5.2% reduction in routing area versus fully populating all routing tracks.

7.5.2 Two-Wire-Type Architectures

To determine if the routing architectures of Section 7.4 could also benefit from reductions in the internal population of some routing wires, we conducted experiments in which we reduced the internal population of some of the best architectures found in Section 7.4. Figure 7.20 shows how area-efficiency varies as the fraction of routing tracks using a switch-block internal population of 60% is varied, for a representative architecture. The routing architecture shown in Figure 7.20 contains 33% length 4 buffered wires and 67% length 4 pass-transistor-switched wires. The connection-block population of all wires is 100%. The three curves in Figure 7.20 correspond to

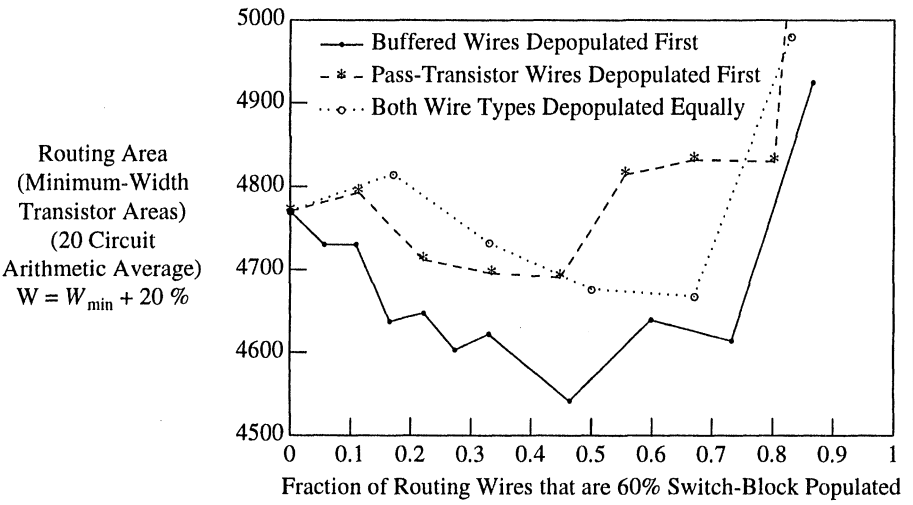


FIGURE 7.20 Area versus fraction of routing tracks with 60% switch-block population. Architecture contains 33% buffered length 4 wires and 67% pass-transistor-switched length 4 wires.

three different ways to depopulate the routing wires. The solid curve depopulates the buffered wires first, and depopulates the pass-transistor-switched tracks only once all the buffered wires are depopulated. The 0.33 point on this curve therefore corresponds to an architecture in which all the buffered routing tracks have a switch-block internal population of 60%, and all the pass-transistor switched tracks have a switch-block population of 100%. The dashed line, on the other hand, depopulates the pass-transistor-switch wires first. The 0.67 point on this curve, then, corresponds to an architecture in which all the pass-transistor-switched wires have a switch-block population of 60%, and all buffered wires have a switch-block population of 100%. Finally, the dotted line depopulates pass-transistor-switched and buffered wires in equal proportion. For example, the 0.67 point on this curve denotes an architecture in which 67% of the buffered wires and 67% of the pass-transistor-switched wires have a switch-block population of 60%; the other 33% of the routing wires have a switch-block population of 100%. Note that the "0" point on the horizontal axis corresponds to an architecture in which all routing wires are fully switch-block populated, for all three curves.

From Figure 7.20 one can clearly see that depopulating buffered routing tracks first leads to the best area-efficiency. This makes intuitive sense — since tri-state buffer routing switches consume more area than pass transistors, eliminating these buffered switches saves more area than eliminating pass transistors. The best area-efficiency occurs near the 0.5 point on the solid curve of Figure 7.20. In this architecture all the buffered wires (33% of routing tracks) have a switch block internal population of 60%. 20% of the pass-transistor-switched wires (13% of the routing tracks) also have a switch-block population of 60%, while the other pass-transistor-switched wires (54% of the routing tracks) have a switch-block population of 100%. This architecture has 4.8% better area-efficiency than an FPGA in which all routing tracks are fully switch-block populated. Depopulating routing wires for this architecture did not produce a significant speed gain, however.

We have experimented with other two-wire-type FPGA architectures, and found that they also tend to have the best area-efficiency when the buffered routing tracks have a switch block only in every second potential location. Depopulating some of the pass-transistor-switched wires as well can lead to a small further area improvement if less than 50% of the routing tracks consist of buffered wires. For example, an architecture containing 50% length 8 buffered wires and 50% length 4 pass-transistor-switched wires has 5.3% better area-efficiency when the buffered wires have a switch-block population of 56% (i.e. 5 out of 9 potential switch blocks) than when all wires are fully switch-block populated.

7.6 Wire Spacing for Speed

The previous sections considered optimizing the topology of an FPGA's routing architecture — i.e. the connectivity of the switches and wires. In this section we consider a purely electrical optimization: varying the spacing between routing wires. In modern IC processes, the coupling capacitance between adjacent wires in the same metal layer is a large fraction of the total wire capacitance. Increasing the spacing between wires reduces this coupling capacitance, and hence speeds up an FPGA's routing. For example, in TSMC's 0.35 μm process, increasing the spacing between metal 3 wires from the minimum value (0.5 μm) to five times the minimum value (2.5 μm) reduces the coupling capacitance between adjacent wires to almost nothing, and reduces the total metal capacitance by over 48%. Of course, this increase in metal spacing reduces metal routing density — the pitch of the wires is increased by a factor of 2.8. Often, therefore, it will not be possible to increase the metal spacing of every routing wire this much without requiring more metal area than is available in the FPGA layout (i.e. all the required metal would no longer fit above the area required by the transistors, so the chip layout would have to be expanded).

A natural question then, is whether it is necessary to reduce the capacitance of every routing wire in this way to speed up an FPGA, or if reducing the capacitance of only a subset of the routing wires is sufficient. Figure 7.21 shows how the speed of a length 4 buffered architecture (the best single-wire-length FPGA found in Section 7.3.2) varies as the fraction of routing tracks that are widely spaced is varied. In Figure 7.21 all

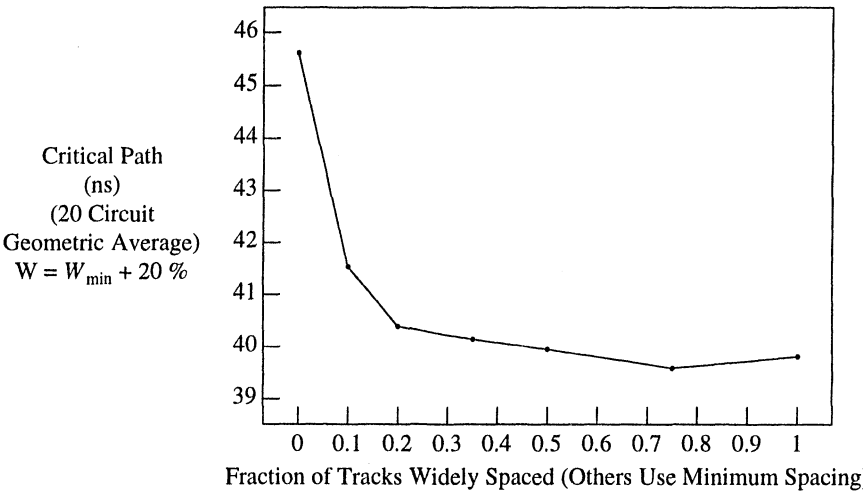


FIGURE 7.21 Speed of a length 4, buffered wire FPGA vs. routing wire spacing.

routing wires have length 4, and connect via tri-state buffers. The "0" point on the horizontal axis corresponds to an FPGA that uses the minimum metal spacing between all the routing tracks, while the "1" point corresponds to an FPGA that using a wide (2.5 μm) spacing between all the routing tracks. Observe that increasing the metal spacing significantly improves speed — an FPGA that widely spaces all the routing wires is 14.6% faster than an FPGA that uses the minimum spacing for all the wires. Even more interesting is the fact that we can obtain most of this speedup by increasing the spacing between only 10% to 20% of the routing wire segments. If we use a wide spacing between only 10% of the routing wires, we obtain a 9.8% speedup, and widely spacing 20% of the routing wires yields a 12.9% speedup. By increasing the spacing between only 10% to 20% of the routing wires we minimize the increase in metal area required, but still obtain almost all the achievable speedup.

Note that Figure 7.21 also provides an indirect validation of our router. If our router used the faster wires equally for all connections, Figure 7.21 would be a straight line from the slowest architecture (when all routing wires use the minimum spacing) to the fastest architecture (when all routing wires are widely spaced). Instead, however, we obtain most of the achievable speedup when only 10% to 20% of the wires are widely spaced. This implies that our router is able to correctly identify the most time-critical nets and route them via the fastest routing resources.

Figure 7.22 shows how the speed of one of the best two-wire-type architectures varies with the spacing between routing wires. In this architecture 50% of the routing wires

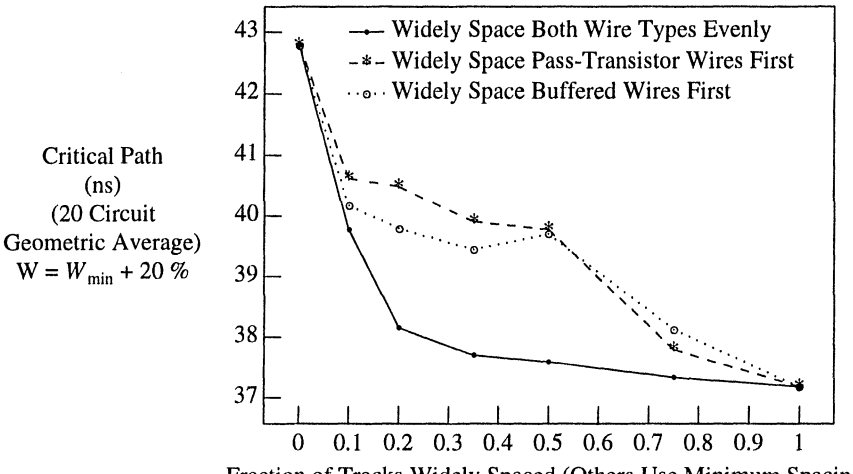


FIGURE 7.22 Speed of a length 4, 50% buffered, 50% pass-transistor FPGA vs. wire spacing.

are length 4 wires connected by tri-state buffers, while the other 50% are length 4 wires connected by pass transistors. Again, the horizontal axis denotes the fraction of tracks that are widely spaced (2.5 μm spacing), while the other tracks use the minimum metal spacing. The solid line shows the speed obtained when the widely spaced tracks are allocated evenly to buffered and pass-transistor-switched wires. The dashed line is obtained when we increase the spacing of the pass-transistor-switched tracks first, and increase spacing of the buffered tracks only after all the pass-transistor-switched tracks have wide spacing. The dotted line occurs when we do the opposite — we first increase the spacing of the buffered tracks, and increase the spacing of the pass-transistor-switched wires only after all the buffered wires are widely spaced.

Once again, increasing the spacing of all the metal tracks leads to a significant speedup; in this case the speedup is 15.1%. The solid line in Figure 7.22 shows that most of this speedup is again obtained when only 10% to 20% of the routing tracks are widely spaced. In this case, we obtain a speedup of 7.6% by spacing out only 10% of the routing tracks, and a speedup of 12.1% by spacing out 20% of the routing tracks. Comparing the solid line in Figure 7.22 to the other two lines shows that one must increase the spacing of both some pass-transistor-switched wires and some buffered wires. About half the total speedup is due to speeding up the pass-transistor-switched wires, and the other half is due to speeding up the buffered wires. This indicates that both these types of wire are useful for making speed-critical connections — the pass-transistor-switched wires are better for short connections, while the buffered wires are better for long connections.

While this section has examined speeding up an FPGA by spacing out some of its wires, one can achieve similar effects by increasing the size (and hence the drive strength) of the routing switches used to interconnect wires on certain routing tracks. From the results of this section, we can predict that increasing the size of the switches on only 20% of the routing tracks will produce almost as much speed gain as increasing the switch size on every routing track.

7.7 Overall Architecture Comparison

In the previous sections we have examined FPGA routing architectures of gradually increasing complexity and looked for important architectural trends by varying the key architectural parameters. In this section we provide an overview of all the results of this chapter by comparing some of the best architectures found in each of the prior sections against each other and against a routing architecture that is similar to that of the popular Xilinx XC4000X series FPGAs [139, 140]. As described in Section 4.6.2, this "4000X-like" architecture contains 25% length 1 wires, 12.5% length 2

wires, 37.5% length 4 wires, and 25% "one-quarter longs", whose length is one-fourth of the chip. The length 1 and 2 wires connect via pass transistors, while the longer wires connect via tri-state buffers. As well, pass transistor switches also allow the length 4 wires to connect to the length 1 and 2 wires, and the one-quarter longs to connect to length 1 wires. While this routing architecture is very similar to that of the Xilinx XC4000X, it simplifies a few features [140].

Table 7.7 compares the speed and density of some of the best architectures found in each of the preceding sections to those of this 4000X-like architecture. We also include the performance of an FPGA composed entirely of length 1 wires connected by pass transistors, since most prior FPGA research has focused on this architecture. All the architectures below have a connection-block internal population of 100%, and all routing wires are minimum width, minimum spacing metal 3 wires. All the architectures have an $F_{c,ouput}$ of 0.25·W and an $F_{c,input}$ of 0.5·W.

TABLE 7.7 Comparison of key architectures (20 circuit average).

Segmentation of Routing Tracks, and Switch Types Used	Switch-Block Internal Population	Delay (ns)	Speedup vs. 4000X-like FPGA	Routing Area (Min. Width Transistor Areas)	Routing Area vs. 4000X-like FPGA
Xilinx 4000X-like: 25% L1, 12.5% L2, 37.5% L4, 25% one-quarter longs; mix of buffers and pass transistor switches	100% for all wire lengths except L2. For L2, 66%.	48.83	—	4425	—
100% L1, pass-transistor switched	Not relevant	120.7	-60%	5891	+33.1%
100% L8, pass-transistor switched	100%	46.22	+5.6%	5131	+16.0%
100% L4, buffer switched	100%	45.57	+7.2%	5792	+30.9%
100% L4, buffer switched	60% for 67% of tracks; 100% for 33% of tracks	44.82	+8.9%	5490	+24.1%
67% L4, pass-transistor switched; 33% L4, buffer switched	100%	42.91	+13.8%	4771	+7.8%

TABLE 7.7 Comparison of key architectures (20 circuit average).

Segmentation of Routing Tracks, and Switch Types Used	Switch-Block Internal Population	Delay (ns)	Speedup vs. 4000X-like FPGA	Routing Area (Min. Width Transistor Areas)	Routing Area vs. 4000X-like FPGA
83% L4, pass-transistor switched; 17% L4, buffer switched	100%	44.31	+10.2%	4569	+3.3%
50% L4, pass-transistor switched; 50% L8, buffer switched	100%	41.04	+19.0%	5039	+13.9%
83% L4, pass-transistor switched; 17% L8, buffer switched	100%	43.84	+11.4%	4539	+2.6%
67% L4, pass-transistor switched; 33% L4, buffer switched	60% for buffer-switched wires; 100% for pass-transistor-switched wires	43.15	+13.2%	4622	+4.5%
50% L4, pass-transistor switched; 50% L8, buffer switched	56% for buffer-switched wires; 100% for pass-transistor-switched wires	41.23	+18.4%	4708	+6.4%
83% L4, pass-transistor switched; 17% L8, buffer switched	56% for buffer-switched wires; 100% for pass-transistor-switched wires	44.04	+10.9%	4426	0%

The architectures are listed (after the 4000X-like architecture) in order of increasing complexity. Notice the extremely poor performance of an FPGA using only length 1 wires connected via pass transistors — 147% slower (-60% speedup) and 33.1% percent larger than the 4000X-like architecture. The best architecture we found using only pass transistors and one length of wire used length 8 wires. This architecture performs much better than a length 1 architecture; it is 5.6% faster than the 4000X-like architecture, at a cost of 16% larger area than that of the 4000X-like architecture. Although the speed and area-efficiency of this length 8, all pass-transistor FPGA are reasonably competitive on average, we consider FPGA architectures that contain no

buffers dangerous. As circuit size increases, the longest connections in an FPGA grow longer, and will pass through more series switches. Since the delay of pass-transistor switches grows quadratically with the number of switches in series, it is difficult for an architecture that contains only pass transistors to maintain good speed as the size of the logic block array grows. As well, larger circuits can contain nets with a higher maximum fanout, and purely pass-transistor based routing is inefficient for routing high-fanout nets. For both these reasons, architectures that contain only pass transistors do not scale as well with increasing circuit size as architectures that contain some buffers.

The best single-wire-type architecture we found, in which all wires are length 4 and all switches are buffers, is 7.2% *faster* than the Xilinx 4000X-like FPGA. Its area is 30.9% larger, however. It is interesting that such a simple FPGA architecture is reasonably competitive with the complex routing architecture of the 4000X-like FPGA. Simpler routing architectures make it easier to develop CAD tools. As well, they likely make it easier to implement intellectual-property "cores." These cores are sometimes provided as "hard" (placed-and-routed) macros. If there is only one type of routing resource in the FPGA, it is easier to map several of these hard cores into one FPGA and ensure each gets the wires it needs. These factors may make a simple FPGA architecture, in which all the wires are essentially the same, attractive despite its suboptimal speed and area performance.

By reducing the internal switch-block population of some of the wires in this all length 4 architecture to 60% we can reduce the area overhead versus the 4000X-like architecture from 30.9% to 24.1%.

Table 7.7 also lists four of the best two-wire-type architectures. Each of these architectures combines some buffered wires with some pass-transistor-switched wires. Two of these architectures use only length 4 wires, while the other two use some length 4 and some length 8 wires. Notice that these architectures are all significantly (10.2% to 19%) faster than the 4000X-like architecture, but none of them is as area-efficient. The area penalty for using an architecture that is 19% faster than the 4000X-like architecture is 13.9%, while the area penalty for an architecture that is 11.4% faster than the 4000X-like architecture is only 2.6%. Both the speed and the area of these architectures are significantly better than the best single-wire-type architecture discussed above, showing that a mix of pass transistors and buffers is very useful in FPGA routing.

The last three lines in Table 7.7 show the benefits of reducing the switch-block population of these two-wire-type architectures so that the buffered wires have a switch block only once every two logic blocks. These architectures are from 2.5% to 7.5% more area-efficient than those that used a switch-block population of 100% for all

wires; the exact amount of area improvement depends on what fraction of the routing wires use buffered switches. One of these architectures is 18.4% faster than the 4000X-like architecture and only 6.4% larger, and another architecture is 10.9% faster and uses the same area as the 4000X-like architecture.

From the results of Table 7.7 one can see that we have found many architectures with speed superior to that of the 4000X-like architecture, but none with superior density. We believe the 4000X contains too many short wire segments, and this reduces its speed versus many of the architectures we have investigated. The 4000X uses a more advanced switch block than the disjoint switch block used by all the other architectures in Table 7.7, however. The 4000X switch block contains some switches that allow wires of different lengths to connect, and that allow wires in different tracks to connect. These features make the 4000X switch block more routable than the disjoint switch block, yet it contains only a few more switches than the disjoint switch block. Consequently, this switch block tends to result in FPGAs with superior density. As discussed in Section 7.3.1, we consider the investigation of better switch block topologies for use with FPGAs that contain some long wires to be a fertile area for future research. We expect that combining the segmentation distributions of some of the architectures listed in Table 7.7 with a better switch block would lead to significantly improved area-efficiency.

Finally, note that the speed enhancement of increasing the spacing between 20% or so of the routing wires (described in Section 7.6) can be applied to any of the architectures listed above to yield a further speed increase of approximately 13%.

7.8 Summary

In this chapter we have investigated a large number of different detailed routing architecture issues. First, we showed that a disjoint switch block topology is superior to a Wilton switch block topology for FPGAs that contain wires longer than one logic block. The switch block used in the Xilinx XC4000X FPGA is superior to either of these alternatives, however, and considerable further research is needed in this area.

Secondly, we showed that it is most important for FPGAs to contain wires of moderate length (4 to 8 logic blocks). While most commercial FPGAs contain some very short and some very long wires, we have found FPGAs that use significant numbers of these types of wires to be inferior to those that employ medium-length wires.

We also found that FPGAs that contain a mix of pass-transistor and tri-state buffer routing switches are superior to FPGAs that employ only one type of switch. The

fastest FPGAs tend to contain about 50% pass-transistor switches and 50% tri-state buffer switches, while the most area-efficient FPGAs contain about 80% pass-transistor switches and only 20% tri-state buffer switches.

We also found that reducing the switch-block internal population of routing wires that interconnect with tri-state buffers produces an area gain of 2.5% to 7.5% for typical architectures. The switch block population of these buffered wires should be set so that each wire contains a switch block at only every second potential location along its length. Reducing the connection-block internal population of long wire segments did not produce any significant area gain, however.

We showed that by increasing the metal spacing of only about 20% of the routing wires, one can increase the speed of an FPGA by over 12%. Increasing the spacing between wires reduces capacitance, and our router can route almost all the time-critical nets on only 20% of the routing tracks. Consequently, increasing the spacing of 20% of the routing wires yields almost as large a speedup as increasing the metal spacing on all the routing wires.

Finally, we showed that the best architectures examined in this chapter have significantly superior speed to a (slightly simplified) Xilinx XC4000X routing architecture; some architectures are 19% faster than the 4000X architecture. This 4000X-like routing architecture has better area-efficiency than all but one of the architectures we examined, however. The best architectures in this chapter have a better area-delay product than the 4000X-like architecture, indicating that they have gained more in speed than they have sacrificed in area. They are also less complex than the 4000X. This is a useful feature, as it simplifies both CAD tool design and the implementation of pre-placed and pre-routed intellectual property cores.

CHAPTER 8 *Conclusions and Future Work*

8.1 Summary and Contributions

This work has contributed to two related research areas: FPGA CAD algorithms and FPGA architecture. The new CAD algorithms and tools developed in this research were described in Chapters 3 and 4, and are briefly summarized in Table 8.1. In Chapter 3, we developed the first publicly-described logic block packing tools targeting cluster-based logic blocks.[1] We also created a new simulated annealing based placement tool (the placement portion of our Versatile Place and Route (VPR) program) which incorporates three new enhancements over prior tools. First, we implemented a new annealing schedule that adapts automatically to different placement problems, provides good result quality and is more robust than the schedule of Huang et al [82]. Second, we developed a new, "linear congestion" placement cost function that enhances the routability of circuits mapped to FPGAs in which different channels have different widths. Finally, we developed an incremental net bounding box update algorithm that reduces the CPU time required for placement by more than a factor of five, on average, vs. using the traditional brute-force bounding box recomputation.

Chapter 4 described new routing algorithms and a parameterized FPGA routing architecture generator that were implemented in the VPR placement and routing program. We described a new problem in FPGA CAD: the automatic generation of *good* rout-

1. Altera Corporation has an internal logic block packing tool that targets logic clusters, and Xilinx has a tool targeting logic blocks very similar to logic clusters for use with the XC5200 FPGA. The algorithms used by these tools have never been disclosed, however.

TABLE 8.1 Summary of CAD contributions.

CAD Area	Contributions
Logic Block Packing	• First academic tools to target parameterized cluster-based logic blocks
Placement	• Architecture independent placement tool • New, robust annealing schedule • Fast incremental update scheme • High quality results
Routing	• Architecture-independent router • Architecture generator for ease of specification • Fully timing-driven • Directly optimizes the Elmore delay • Very high quality
Modelling	• Transistor-based area model • Full delay extraction • Path-based timing analysis

ing architectures from a succinct list of understandable parameters. We showed that it is often difficult to simultaneously satisfy all the specified parameters, and highlighted some of the novel features of our architecture generator that allow all the architecture parameters to be satisfied under most circumstances, and result in an FPGA with good routability.

Chapter 4 also described the two routers we developed and implemented in VPR: one is purely routability-driven, while the other is both routability- and timing-driven. The routability-driven router is based on the Pathfinder negotiated congestion algorithm [85] but includes enhancements to the routing cost function and "routing schedule." As well, the VPR routability-driven router employs a new method we developed for the breadth-first routing of multi-terminal nets which reduces the CPU time to route high-fanout nets by an order of magnitude compared to the traditional breadth-first maze router method. In terms of circuit routability, this routability-driven router outperforms all other routers to which we can compare, and the combination of the VPR placement algorithm and the VPR routability-driven router significantly outperforms all combinations of placement and routing tools to which we can compare.

The VPR timing-driven router employs new algorithms we developed to directly optimize the Elmore delay, while simultaneously resolving routing congestion. Previous non-commercial FPGA routers have optimized either wirelength or the (inaccurate) linear delay model; by directly optimizing the more complex Elmore delay model our router can make much more intelligent routing decisions. As well, our timing-driven

router is the first non-commercial "buffer-aware" FPGA router,[1] and it performs buffer insertion automatically as it constructs net routing topologies. This timing-driven router also has excellent performance: it is capable of optimizing for routability almost as well as our routability-driven router, while producing circuits which are 2.6 times faster, on average. The run-time of this router is also quite low — it can route an 8300 LUT circuit in 3.2 minutes.

As well as the new algorithms incorporated in VPack, T-VPack and VPR, we consider the VPR / (T-)VPack CAD infrastructure itself to be a major contribution of this work. VPR not only produces high-quality placements and routings, it is also very flexible; i.e. it can target a wide variety of FPGA architectures. VPR includes fast net delay extraction and path-based timing analysis modules, allowing easy evaluation of the speeds of different FPGA architectures. It also includes a detailed, transistor-based area model, allowing easy comparison of the area of different FPGA architectures. Taken together, these capabilities enable more sophisticated and detailed comparison of FPGA architectures than were possible with previous tool sets. In addition to the three architectural studies described in this book, VPR has been used to evaluate the quality of synthetic benchmark circuits [147, 148, 149], to investigate laser-programmed gate array architectures [150], to generate good global routings for use with an FPGA detailed router [101], and as a CAD infrastructure for research into ultra-fast FPGA routing algorithms [151, 127, 152] and ultra-fast FPGA placement algorithms [153]. Ongoing research at the University of Toronto is enhancing VPR in order to research the routing architecture of Altera-like (hierarchical) FPGAs [138]. Several FPGA companies are currently using VPR to aid in the development and evaluation of new FPGA architectures, and researchers from over 80 universities and 30 companies have downloaded copies of VPR from our web site. VPR, VPack and T-VPack are all available for download from *http://www.eecg.toronto.edu/~jayar/software/software.html*, and may be freely used for academic FPGA research.

In chapters 5, 6 and 7 we used these VPR, VPack, and T-VPack CAD tools to investigate three different FPGA architecture issues; Table 8.2 briefly summarizes our most important architecture conclusions. In Chapter 5 we explored FPGA global routing architectures, which specify the relative widths of the various channels within an FPGA. We found that the best global routing architecture distributes the logic block pins evenly around the logic block perimeter, and has the same width for every channel within the FPGA. Row-based FPGAs, which have logic block pins only on the top and bottom of each logic block, are only 8% less area-efficient than this best architecture, however, as long as the horizontal channels are twice as wide as the vertical channels. We also found that by adjusting the relative widths of the horizontal

1. The routing tools of Xilinx Inc. are buffer-aware [107], and those of other companies whose FPGAs contain a mix of pass transistors and buffers are also likely buffer-aware.

and vertical channels, one can create FPGAs with rectangular logic block arrays which are nearly as area-efficient as FPGAs with square logic block arrays. We showed that making the routing channel between the I/O pads and the logic block array moderately (25%) wider than the other channels is a net benefit when the I/O positions are locked. We showed that making one or more routing channels near the center of the array extra-wide, as some commercial FPGAs do [141], is not very useful, however. Although these issues have been widely discussed and speculated upon, the work presented here is the first systematic study of global routing architecture.

TABLE 8.2 Summary of FPGA architecture conclusions.

Architectural Issue	Key results
Global routing architecture	• Uniform channels better than non-uniform channels • No directional bias is best
Cluster-based logic blocks	• Best cluster size: 4 - 10 look-up tables • Better speed, area and compile time than single-BLE logic block • Need only about half as many cluster inputs as there are LUT inputs • Should reduce logic to routing interconnect flexibility for large clusters
Detailed routing architecture	• Relatively simple architectures perform well • Length 1 wires not useful • Best single wire segment length = 4 • 50% to 80% of routing switches should be pass transistors; remainder tri-state buffers • Slight switch-block depopulation yields area gain • Decreasing the delay of only 20% of the routing wires yields the same circuit speed as decreasing the delay of 100% of the routing wires

In Chapter 6, we presented an extensive study of "cluster-based" logic blocks, which are logic blocks that contain several LUTs and registers interconnected by local routing. While Altera FPGAs use cluster-based logic blocks, this is the first study of how one should choose the key logic cluster parameters, and of the speed and area-efficiency achievable with these logic blocks. We investigated several key logic cluster issues. First, we found that one need not provide $4N$ distinct inputs to a cluster of N 4-LUTs — approximately $2N+2$ distinct inputs suffice to achieve full logic utilization, and result in a more area-efficient logic block. Second, we found that as the size of the logic cluster increased, one should reduce F_c to achieve the best area-efficiency; a good $F_{c,output}$ value is W / N, while the best F_c value for logic block inputs is somewhat higher. Third, we found that logic clusters containing between 4 and 10 LUTs have the best area-delay product. An FPGA employing a logic cluster of size 7, for

example, requires 19% less area, achieves 21% higher circuit speed, and has a 33% lower area-delay product than an FPGA with a single-BLE logic block. Finally, we found that the use of larger logic clusters significantly reduced the time required to place and route designs. For example, use of a logic cluster of size 10 reduced the time to place and route circuits by a factor of 5.5 vs. using a single-BLE logic block.

In Chapter 7, we investigated FPGA detailed routing architecture, which specifies the exact pattern of wires and switches forming the routing fabric, for island-style FPGAs. While some detailed routing architecture questions have been investigated before, this study is the first to use a truly timing-driven router, the first to evaluate an architecture's speed via the critical path delay, and the first to model routing area using a new minimum-width transistor equivalent area model. As well, many of the detailed routing architecture issues we investigated had not been previously studied. More specifically, our work is the first to consider routing that contains tri-state buffer routing switches, partially-internally depopulated routing wires, and routing wires with varying metal spacing.

One of the most important conclusions of this study is that most of the wires in an FPGA should be of moderate length (4 to 8 logic blocks); the very short and very long wires often found in commercial island-style FPGAs lead to inferior speed and area-efficiency. As well, we found that FPGAs should contain a mix of tri-state buffer and pass transistor routing switches; using pass transistor switches on approximately 50% to 80% of the routing tracks is best. Internally depopulating wire segments that use tri-state buffer routing switches so that they have a switch block in only every second location leads to a 2.5% to 7.5% area-efficiency gain. Neither reducing the switch-block population of wires that use pass transistor switches nor reducing the connection-block population of wires is as successful at reducing routing area. We also found that increasing the spacing between metal wires significantly improved the FPGA speed, and that one could obtain virtually the entire speedup by increasing the spacing of only 20% of the routing wires. Specifically, increasing the spacing of 20% of the routing wires typically improves FPGA speed by 12% to 13%, while increasing the spacing of all the wires increases the speed by about 15%. Finally, we found that properly designed routing architectures employing only two different lengths of wire and a simple switch block lead to 11 - 18% better speed than a routing architecture similar to that of the Xilinx XC4000X FPGA, with only a 0 - 6% area-efficiency penalty. These architectures are considerably simpler than that of the Xilinx XC4000X, and this makes them even more attractive, since simple architectures ease the task of writing CAD tools and implementing "hard" intellectual property cores.

8.2 Future Work

8.2.1 CAD Tool Enhancements

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

In this work we have created timing-driven logic block packing and timing-driven routing tools, but the placement phase of VPR is purely routability-driven. Accordingly, the VPR placement algorithm should be enhanced to make it timing-driven — work is already underway on this project at the University of Toronto [19]. Another interesting project would explore whether incorporating more advanced delay models into T-VPack (such as fanout-based delay estimates) would improve circuit speed by giving T-VPack a better estimate of where the post-place-and-route critical path will lie.

The VPR placement algorithm could also be enhanced by adding a cost function that considers the underlying detailed routing architecture when evaluating the cost of a placement. For example, in an FPGA that contains only length 4 wires, any connection between logic blocks in the same row or column is likely to be completed with one wire segment if it spans less than four logic blocks. If, however, the connection spans five logic blocks, or if the connection is between logic blocks that are not in the same row or column, it must use more than one wire segment. Consequently, the cost function should consider not only wirelength, but also the number of wire segments (and for timing-driven placement, the minimum delay) required to route from a net source to a net sink. Using a cost function of this type results in a "routing-aware" placement algorithm, without the high CPU time required by simultaneous placement and routing approaches.

The other major class of CAD enhancements concern creating a wider variety of FPGA architectures automatically. As discussed in Chapter 4, the architecture generator in VPR can build routing-resource graphs for detailed routing only when the segmentation distribution in the horizontal and vertical channels is the same. While this is generally true of island-style FPGAs, it is not true of the hierarchical style of FPGAs used by Altera. A research project is currently under way at the University of Toronto to remove this restriction, and hence allow the generation and evaluation of Altera-style FPGAs [138].

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

area. While we have found ways to simultaneously satisfy all the specified parameters in many cases, in some situations we cannot meet all the user's specifications. An important question, then, is whether it is impossible to satisfy all the constraints in these cases, or if there is some better architecture generation algorithm which enables all the constraints to be met. If it is impossible to satisfy all the specified parameters in some cases, is there some parameterization (different than the one we have chosen) of equivalent flexibility which guarantees that any set of parameters can always be simultaneously satisfied? Another problem in automatic architecture generation concerns the automatic generation of good connection block switch patterns. While the patterns currently generated by VPR work well for a wide variety of architectures, they are not necessarily the best possible patterns.

8.2.2 Future FPGA Architecture Research

In Chapter 6 we examined only fully-connected logic clusters, but it is not clear that fully-connected local routing is best. The complexity of fully-connected local routing grows quadratically with the cluster size, so the use of a less flexible local routing structure might make larger logic clusters more attractive. On a related note, is it necessary to allow all N BLE outputs to connect to the main FPGA routing? In large logic clusters many signals are used only within the logic cluster, so the average number of outputs used is less than N. By forcing some BLE outputs to be used only with a logic cluster, an FPGA would contain fewer logic block output pin buffers and connection blocks, saving area. On the other hand, logic utilization might suffer, so the overall effect on area is unclear.

Many interesting questions about detailed routing architecture remain. Further research is needed to determine good switch block topologies for use with FPGAs that contain wires longer than one logic block. This switch block research should pay careful attention to the interaction between the distribution of the wire lengths and the performance of different switch blocks, as our results have shown these two issues to be coupled. In Chapter 7, we found that reducing the connection-block internal population of wires did not provide a significant area-efficiency gain, and that depopulating the connection block on all the wires led to unroutable FPGAs. We always depopulated the switches from logic block output pins and the switches to logic block input pins equally, however. A future research project could look into depopulating the output pin switches and the input pin switches by different amounts. Since the switches from logic block output pins to routing tracks require more area than the switches from routing tracks to input pins, depopulating the output pin switches would yield the largest reduction in switch area per wire. By depopulating the input pin switches by a smaller amount, or not at all, one could maintain good routability, and hopefully achieve a net area gain.

The best architectures we found in Chapter 7 had critical path delays of approximately 40 ns. About 80% of this delay is still inter-cluster routing delay, so there is still considerable room for speed improvement by optimizing the routing architecture. However, almost 40% of the routing delay (about 30% of the total delay) is due to the delay to get out of a logic block via an output pin buffer, and back into another logic block via a track buffer and connection-block multiplexer. Even if we could somehow create routing wires and routing switches with zero delay, then, we would only double the FPGA speed if these fixed delays to enter and leave a logic block were left unchanged. We believe research is therefore needed into "fast paths" which bypass most of the connection block multiplexer, track buffer and perhaps the output pin buffer delays; such fast connections are employed in many commercial FPGAs [4, 7]. For example, one can build direct connections (as Xilinx does) between adjacent logic blocks. These direct connections are short enough that they should not require track buffers. Fast path connections can also connect into a special, fast input in the connection block multiplexers. If the multiplexer is a tree of pass transistors (see Appendix B.1), this fast input is created by making the tree unbalanced; the fastest input has the fewest pass transistors between it and the multiplexer output. Obviously the number of fast paths one can create is very limited (if one makes too many fast paths they all become slow paths again!). Research is needed to decide if such fast paths can be used effectively enough by CAD tools to significantly speed up an FPGA, and if so, which logic blocks should connect via fast paths.

Finally, since power dissipation is of increasing concern in all VLSI design, future research should investigate detailed routing architectures and logic block architectures that lead to the best combination of not only speed and area, but also power.

APPENDIX A *Graphic Visualization in VPR*

VPR contains built-in graphics that we have found very valuable in both CAD algorithm and FPGA architecture research [134]. The graphical visualization of a new FPGA routing architecture or a routed circuit allows much quicker understanding of how well or how poorly the various algorithms in VPR are performing than trying to comprehend megabytes of textual data. On-screen buttons allow the user to pan and zoom various views of the FPGA's routing architecture and placed and/or routed circuits. PostScript output of any display on screen is always available simply by clicking on a "PostScript" button; all the figures in this appendix were generated directly by VPR in this manner. While printed here in black and white, VPR's graphics are actually in colour — see *http://www.eecg.toronto.edu/~jayar/software/software.html* for colour versions of these pictures.

Figures A.1 to A.4 show various steps in the placement and routing of the MCNC benchmark circuit e64. This circuit contains 274 four-input LUTs, and is mapped to an FPGA where every logic block is a single BLE. The FPGA routing contains a mixture of length 1, 2 and 4 wires. Figure A.1 shows a "rat's nest" view of the connected blocks before placement. The user has selected a logic block by clicking on it with the mouse; its fan-out is highlighted in dark grey, while its fan-in is highlighted in black. Notice that the fan-in and fan-out of the selected block are widely scattered across the FPGA.

Figure A.2 shows the same circuit and the same selected block after placement. Now the fan-in and fan-out of this block have been clustered quite closely around it. Most connected blocks have been placed close together, resulting in a much less cluttered view than in Figure A.1.

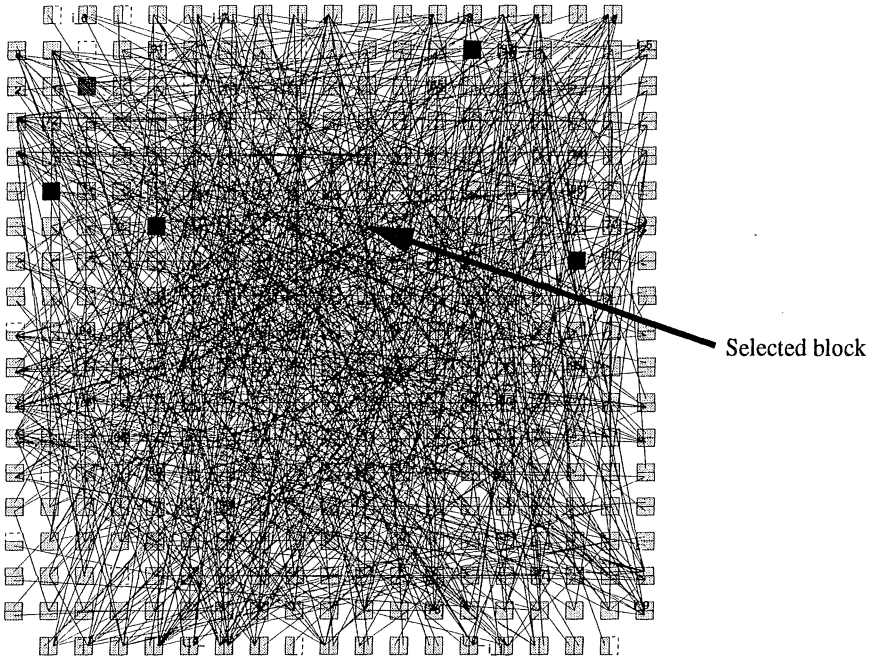


FIGURE A.1 Initial (random) placement of benchmark circuit e64.

Figure A.3 shows e64 after it has been successfully routed. By examining graphical views of routed circuits one can quickly determine how well VPR routes different classes of nets, and identify areas where the routing algorithm is not performing well.

Figure A.4 shows the circuit critical path. A user can highlight each net on the circuit critical path, one by one. Figure A.4(a) shows only the first net on the critical path — the critical connection begins at an input pad (the lighter grey block) and connects to the dark grey logic block. Figure A.4(b) shows the entire critical path, which in this case spans five nets and four logic blocks. The first four nets on the critical path are shown in dark grey. The last net on the critical path is shown in light grey; it begins at the lighter grey logic block and ends on the dark grey output pad. Being able to view the critical path of a circuit in this way has been very valuable in optimizing the CAD tools, since one can quickly determine if the critical path is routed via direct, fast connections. As well, being able to see the structure of the critical path — for example, is it mostly short, local connections or mostly long connections — gives one insight into what kind of nets are determining the circuit speed and how the routing architecture could be changed to speed their routing.

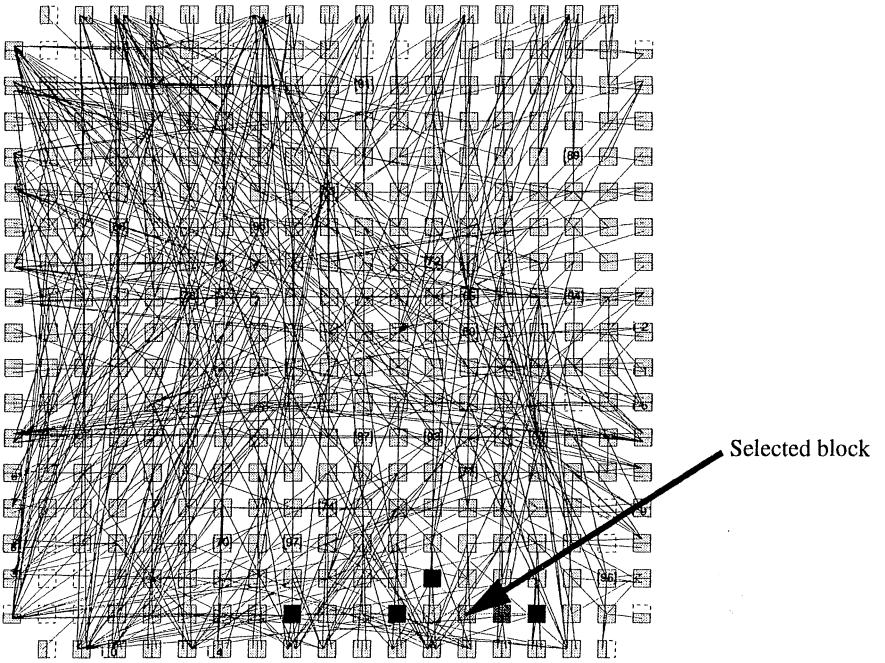


FIGURE A.2 Final placement of e64 found via simulated annealing.

When a routing fails because there are insufficient tracks, or because a switch pattern is simply unroutable, one can view all the routing resources that are still overused after the final iteration of the router. Showing where congestion is occurring in this way makes it much easier to determine why a circuit is not routing on a particular architecture — sometimes it is a simple matter of insufficient wiring, but other times there are intrinsic flaws in the architecture's switch pattern that make certain connection topologies difficult or impossible to route. The e64 circuit requires 10 tracks per channel to successfully route. Figure A.5 shows the congestion remaining after the last (thirtieth) router iteration when each channel contains only 8 tracks. Notice that both some wire segments and some logic block pins are overused.

VPR can also display the routing architecture of an FPGA. Every routing wire, logic block pin, and switch is displayed, and the data to be drawn is taken directly from the routing-resource graph. This allows a user to visualize an entire routing architecture, and quickly determine if he or she has set all the architecture parameters correctly in the architecture file. It also allows one to examine the quality of the switch patterns generated by the VPR architecture generator, and to quickly test and debug improvements to the architecture generator. Figure A.6 shows the entire routing fabric of the

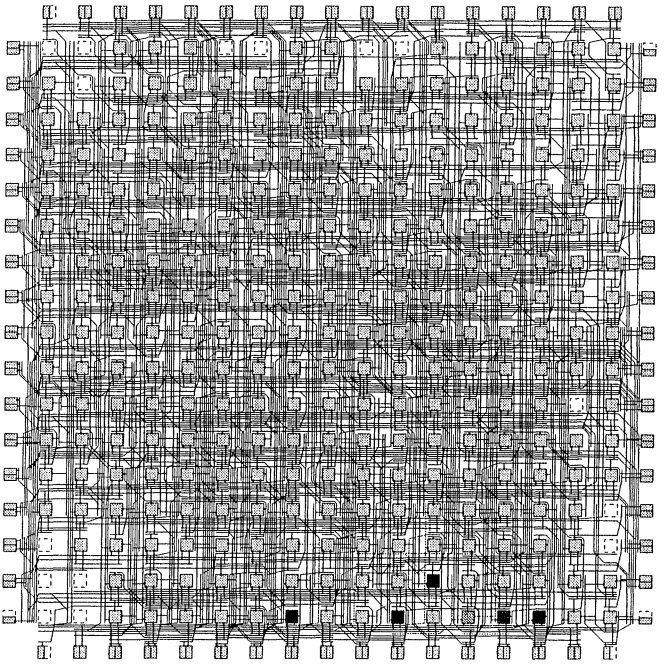
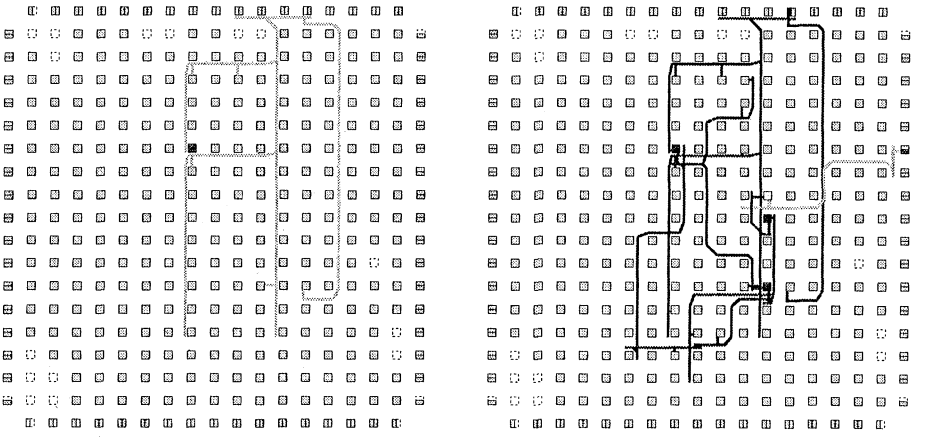


FIGURE A.3 Routing of benchmark circuit e64.



(a) (b)

FIGURE A.4 (a) First net and (b) all five nets on the critical path of e64.

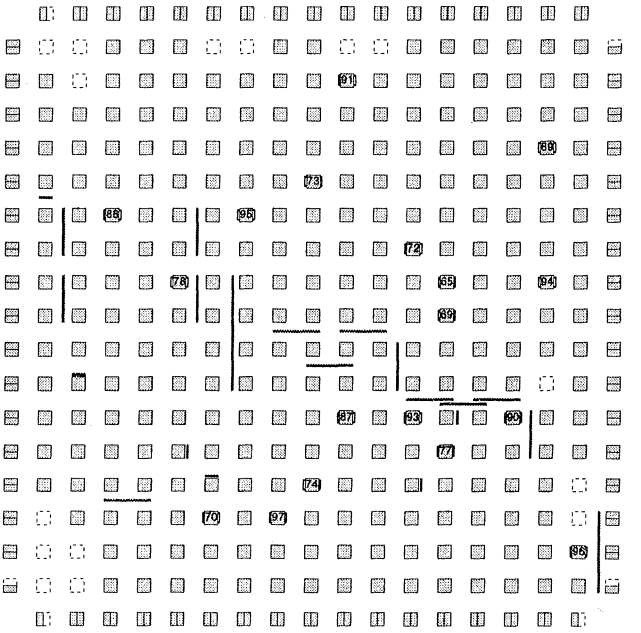


FIGURE A.5 Congestion remaining after an unsuccessful attempt at routing e64.

FPGA in which the e64 circuit was implemented in the previous figures. Figure A.7 is a close-up of a small part of the FPGA so the switch pattern is visible. The black lines are routing wires, while the small black squares are logic block input and output pins. Switches between logic block pins and routing wires are shown as x's. The routing switches in switch blocks are shown as grey lines between routing wires; a small triangle indicates the switch is a tri-state buffer, while a circle indicates it is a pass transistor.

Figure A.8 is a simpler view in which the routing switches are not displayed. The segmentation distribution of the FPGA is clearly visible: 20% of the routing tracks are length 1 wires, 40% are length 2 wires, and 40% are length 4 wires.

Finally, Figure A.9 shows a small portion of an FPGA with a more complex logic block. In Figure A.9 the logic block is a logic cluster of size 4, with 10 logic inputs, and one clock input. Notice that there are no switches connecting the clock input, which is in the upper right corner of each logic block, to routing wires, since in this architecture it was specified that the clock is routed on a special dedicated resource.

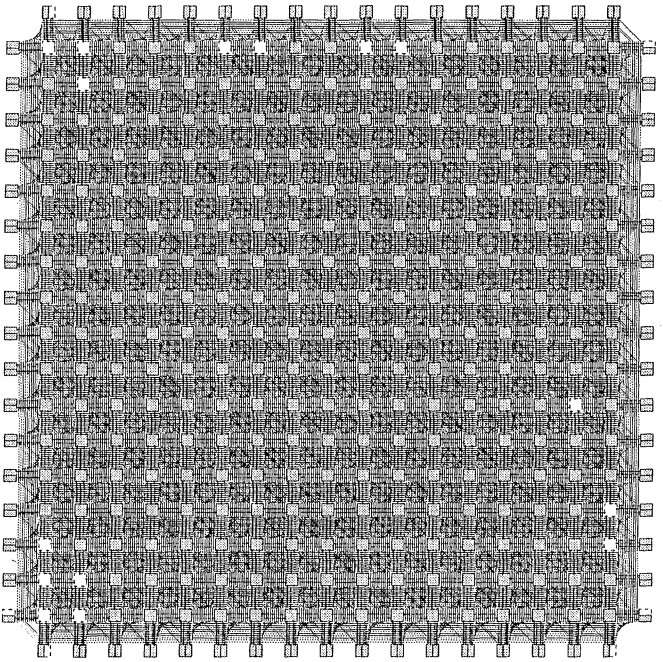


FIGURE A.6 Routing architecture of an FPGA with a single BLE logic block.

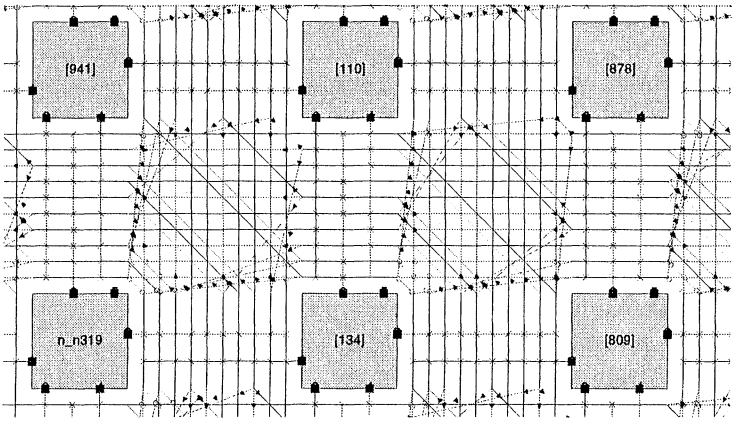


FIGURE A.7 Close-up of routing architecture of an FPGA with a single BLE logic block.

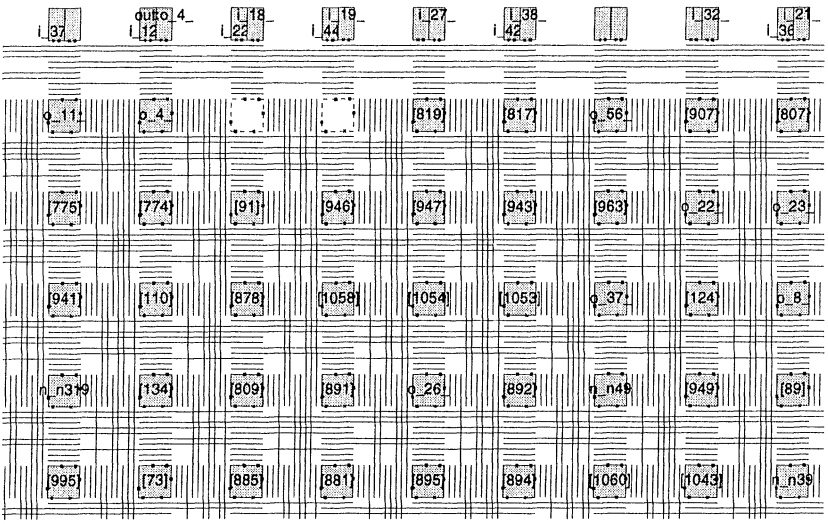


FIGURE A.8 Segmentation distribution of an example FPGA.

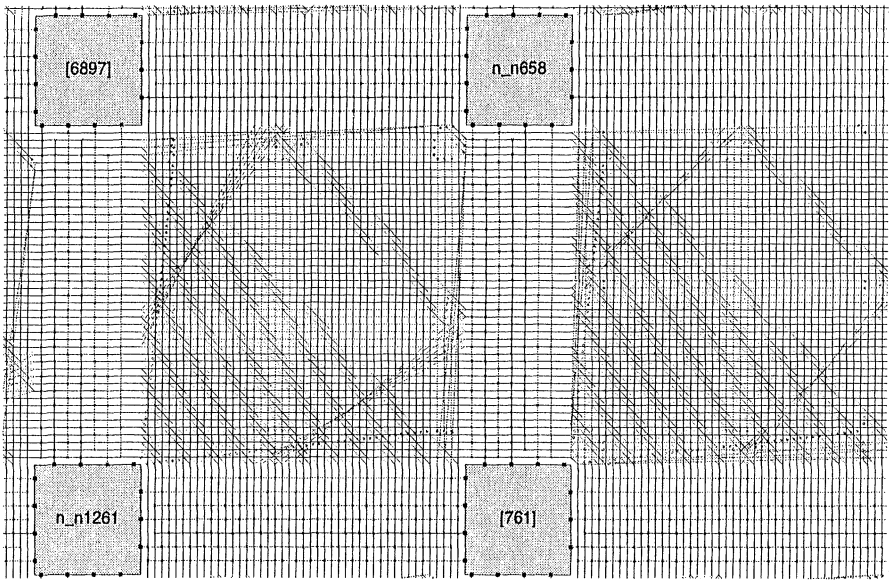


FIGURE A.9 An FPGA where each logic block is a size 4 logic cluster with 10 logic inputs and one clock input.

APPENDIX B

FPGA Circuitry and Process Modeling

This appendix provides more detail on how the area and delay of the FPGAs studied in Chapters 6 and 7 was determined. The next section provides transistor-level schematics showing how each structure in an FPGA is implemented. Most of these schematics are process-independent, but occasionally transistor sizing issues that are somewhat specific to a specific process arise. In such cases, we size the transistors assuming the FPGA would be implemented in TSMC's 0.35 μm, 3.3. V process [154]. Section B.2 describes how we determined the delay or equivalent RC circuit for each structure in an FPGA, and lists the delay through some of the key FPGA structures. This data is required to apply the FPGA delay model described in Section 6.2.3.

B.1 Transistor-Level Schematics and Assumptions

In this section we detail the transistor-level schematics for each FPGA structure used in this research. These schematics show how many transistors each structure contains, and how large the various transistors are, which is exactly the information required by the area model described in Section 6.2.2. As well, these transistor-level schematics are the structures simulated to extract the delay values used in Chapters 6 and 7. Chow et al [21] contains another discussion of the transistor-level implementation of an FPGA; most of the structures we use are similar.

B.1.1 FPGA Routing Structures

A key circuit element of an SRAM-based FPGA is the SRAM cell itself. We used the 6-transistor, double-ended programming cell illustrated in Figure B.1. Although a five-transistor SRAM cell is possible, a six-transistor cell is generally more stable — i.e. more resistant to state flipping due to crosstalk or charge sharing. While some of these transistors may have slightly greater than minimum width (or some may have more than the minimum channel length of 0.35 μm), we assume the total structure requires 6 minimum-width transistor areas. Since the layout of an SRAM-cell is usually the most heavily optimized layout in an FPGA (often to the point that special, relaxed design rules are used in its layout), the assumption that it requires only 6 minimum-width transistor areas is reasonable. We assume that the data and $\overline{\text{data}}$ values stored in the cell can be connected directly to other wires without any isolating inverters — the stability of a six-transistor SRAM cell is therefore essential.

Figure B.2(a) shows a pass transistor used in an FPGA's routing. Each routing pass transistor is controlled by a single SRAM cell. Note that usually this pass transistor will be considerably larger than minimum-width; Appendix C.1 studies the question of how best to size this transistor. If the width of this transistor is ten times the minimum, which is the size we use in Chapter 7, the number of minimum-width transistor areas required by a pass transistor routing switch is 6 (SRAM cell) + 5.5 (ten times minimum size pass transistor; see Equation (6.1)) for a total of 11.5.

Gate Boosting

Figure B.2(b) shows a potential problem with the use of nMOS pass transistors. In Figure B.2(b), an "on" routing pass transistor is passing a logic high value to a routing wire. Notice that the voltage of the routing wire is less than V_{dd} (3.3 V), since an nMOS pass transistor degrades a logic high value by a threshold voltage (including any increase in the nominal threshold voltage due to the body effect [155]). In

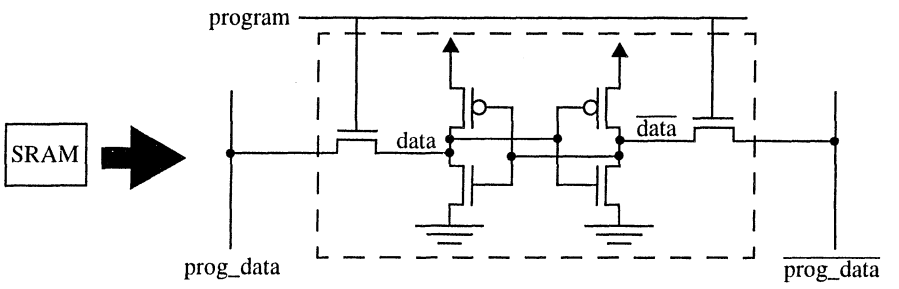
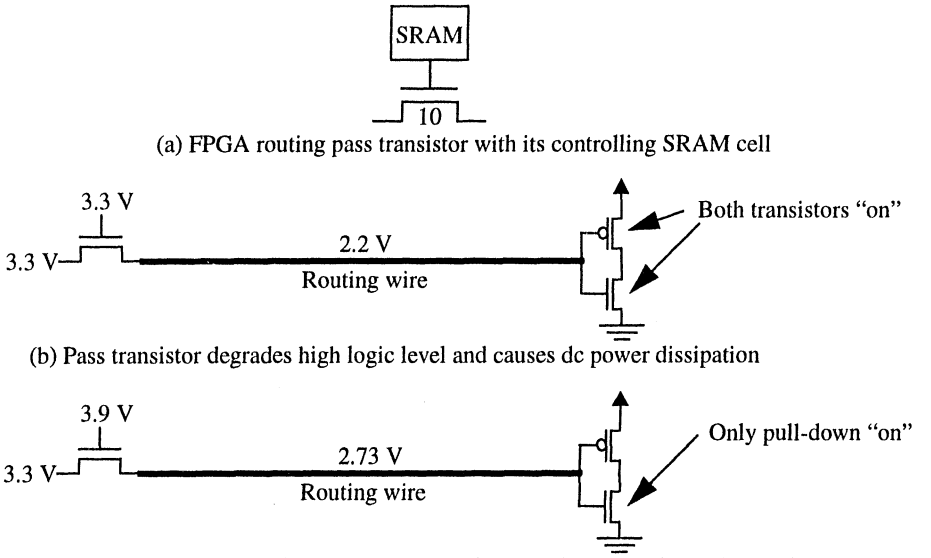


FIGURE B.1 Six-transistor SRAM cell schematic.



(a) FPGA routing pass transistor with its controlling SRAM cell

(b) Pass transistor degrades high logic level and causes dc power dissipation

(c) Boosting the pass transistor gate voltage solves the dc power dissipation problem

FIGURE B.2 (a) FPGA routing pass transistor; (b) static power problem; and (c) solution.

TSMC's 0.35 μm process, for example, the routing wire voltage will rise only to 2.2 V. The output voltage is degraded by 1.1 V, even though the nominal threshold voltage in this process is only 0.6 V, because the pass transistor suffers from the body effect. Downstream, a buffer is used to sense and re-power the signal on the routing wire. With an input voltage of only 2.2 V, both the pMOS and nMOS transistors in the first stage of this buffer will be at least partially "on", dissipating unacceptable amounts of DC power. For example, an FPGA large enough to accommodate the clma benchmark circuit (which requires 2121 size 4 logic clusters) would dissipate 21 W of static power in these partially "on" inverters alone.

One solution to this problem is to boost the gate voltage of pass transistors above the power supply; this technique has been used by Xilinx in their FPGAs. We assume that the gates of all the pass transistors in the FPGAs we construct are boosted to 3.9 V (one nominal threshold voltage above V_{dd}) when designing FPGA circuitry or extracting circuit delays. Figure B.2(c) shows that with a gate boosted to 3.9 V, the pass transistor will now pass a logic high level of 2.73 V, which is enough to turn off the pMOS pull-up transistor in any downstream buffer. For an FPGA large enough to accommodate the clma benchmark circuit, this gate boosting technique reduces the static power dissipated in these "downstream" buffers from 21 W to 0.041 W.

Buffers

Multi-stage buffers are heavily used in the design of FPGAs. To drive a large load, one usually builds a chain of inverters of gradually increasing size — a multi-stage buffer. We build buffer chains with a stage ratio of as close to 4 as possible; this yields high speed while minimizing area [155]. In these buffer chains the pull-up pMOS transistors are 1.9 times as wide as the nMOS transistors. This is roughly the square root of the ratio between the drive strength of an nMOS and the drive strength of a pMOS transistor of the same size in TSMC's 0.35 μm process. Sizing the pMOS transistors in this way minimizes the delay through the buffer chain [156]. Even more importantly, using a pMOS to nMOS width ratio of 1.9 results in equal rise and fall times for the tri-state buffer structures discussed below. Figure B.3 shows an example buffer chain used to create an output stage with 16 times the drive strength of a minimum-size buffer. Note that all the transistor width labels in Figure B.3 (and the other figures in this appendix) are "times minimum width" values, rather than widths in μm.

Recall that signals generated by nMOS pass transistors swing only from 0 V to 2.73 V (rather than 3.3 V). To achieve equal rise and fall times, the inverter that senses the state of a wire driven by an nMOS pass transistor uses a pMOS to nMOS width ratio of only 1.7, rather than the 1.9 we usually use. This reduces the switching threshold of the inverter to 1.35 V (about midway between 0 and 2.73 V), and yields equal rise and fall times on the nMOS pass transistor output.

The buffered switches in FPGA switch blocks and the switches between logic block output pins and routing tracks (in the "output pin" connection blocks) are tri-state buffers. There are several ways to construct tri-state buffers; Figure B.4 shows the form of tri-state buffer we assume. It is simply a combination of a (potentially multi-stage) buffer with an nMOS pass transistor. We assume that the gate of this pass transistor (like all the other pass transistors in the FPGA) is boosted to 3.9 V. The drive strength of the tri-state buffer in Figure B.4 is five times the minimum, which is the tri-state buffer size used in Chapter 7. It occupies 19.7 minimum-width transistor areas.

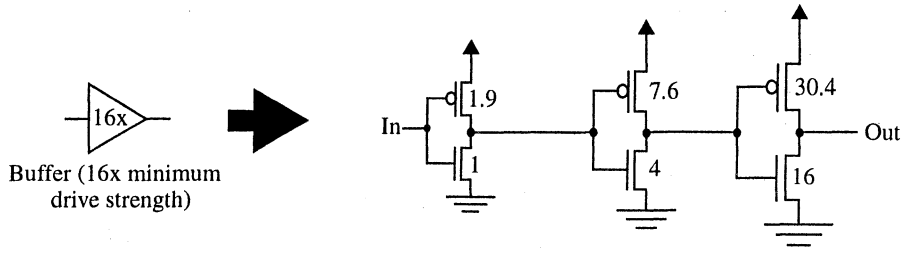


FIGURE B.3 Schematic of a multi-stage buffer.

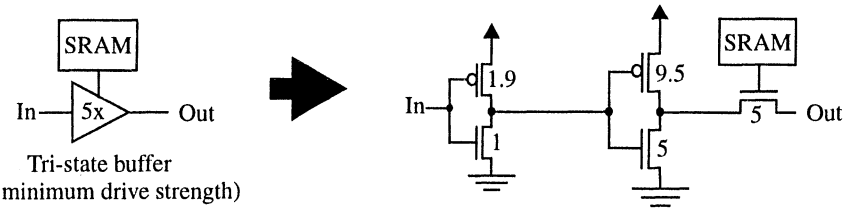


FIGURE B.4 Schematic of a tri-state buffer with 5 times minimum drive strength.

At a point where there are multiple buffered routing switches with the same input, there are two ways to implement the group of switches. We can either build each switch as a completely separate tri-state buffer, as Figure B.5(a) shows, or we can build only one buffer chain, with several pass transistors connecting to the output of this buffer, as shown in Figure B.5(b). We call the two implementations "no buffer sharing" and "fan-out buffer sharing," respectively. Implementing only one buffer chain (i.e., fan-out buffer sharing) saves a significant amount of area, but it can degrade speed by a small amount since this buffer will be more heavily loaded if several of the pass transistors connecting to its output are "on."

The area model built into VPR outputs two area estimates: one assuming no buffer sharing, and the other assuming fan-out buffer sharing. The no-buffer-sharing area assumes that each tri-state buffer switch in a switch block, or in an output pin connection block, is implemented as a separate buffer chain plus pass transistor, as Figure B.5(a) shows. This area estimate is pessimistic because significant area savings can be achieved by sharing one buffer chain. The fan-out-buffer-sharing area model assumes instead that only one buffer chain is implemented for each logic block output pin, or set of tri-state buffer routing switches with the same input and at the same switch block. This area model is depicted in Figure B.5(b). We call this area model slightly optimistic for two reasons. First, in a real FPGA it is likely that the size of the buffer chain would be increased somewhat as the number of pass transistors connected to its output increased, in order to minimize any speed degradation due to several of the pass transistors being "on" at once. Hence the fan-out-buffer-sharing area model slightly underpredicts routing area. Second, the delay model in VPR assumes each switch is independent — that is, does not share a buffer chain with another switch. Consequently, VPR's delay calculator does not account for the speed reduction caused by several of the pass transistors connected to one buffer chain being "on." In other words, the delay model built into VPR models the no-buffer-sharing case. The difference in delay caused by sharing buffers is not very significant, however, so we consider the fan-out-buffer-sharing area model the more realistic one.

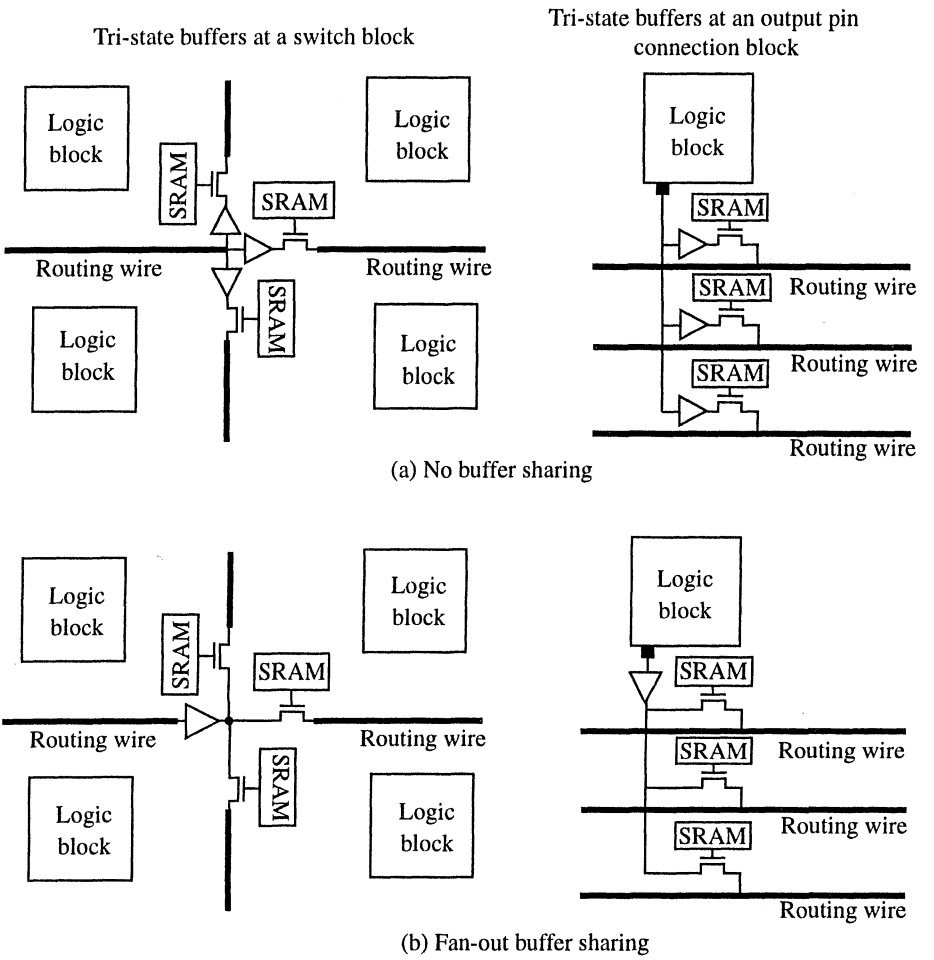


FIGURE B.5 Methods to build tri-state buffers at switch blocks and output pin connection blocks.

Accordingly, all the area results in Chapters 6 and 7 were determined with the fan-out-buffer-sharing area model.

Connection Block to Logic Block Input Pins

The connection block from a set of routing tracks to a logic block input pin is implemented as a multiplexer, as shown in Figure B.6. Notice that buffers isolate each track from the input capacitance of the connection blocks attaching to it. We assume

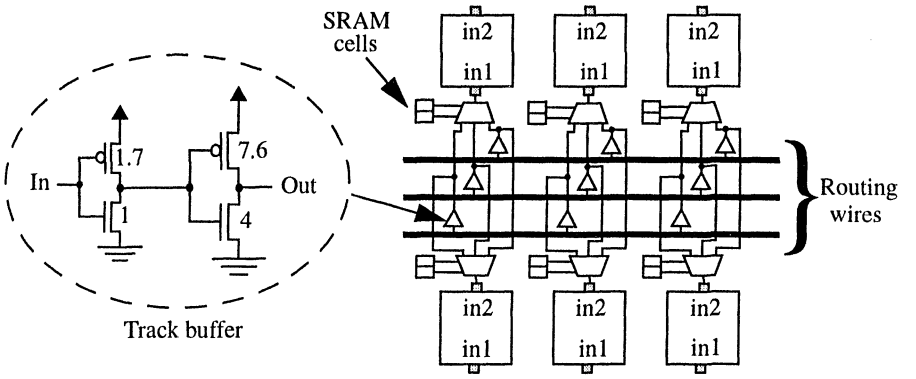


FIGURE B.6 Implementation of input pin connection blocks.

only one buffer is needed at each logic block location along a wire — a wire of length 4, for example, has four attached "track buffers." The multiplexers attaching to a wire from the logic block above a certain point on the wire and the logic block below that point on the wire connect to the output of one track buffer. Figure B.6 also shows that we assume the track buffer is a two-stage buffer with the second stage having a drive strength of four times the minimum. Notice that the first stage of the track buffer uses a pMOS to nMOS width ratio of 1.7 to achieve equal rising and falling delays on the routing wires.

Figure B.7 shows that multiplexers are implemented as a binary tree of pass transistors. All the transistors in this tree are minimum-width. For input pin connection blocks, SRAM cells control which multiplexer input is selected. The total number of minimum-width transistor areas required by such a multiplexer with N_{inputs} inputs is:

$$Area(N_{inputs}) = 6\lceil \log_2(N_{inputs}) \rceil + 2N_{inputs} - 2 . \quad \textbf{(B.1)}$$

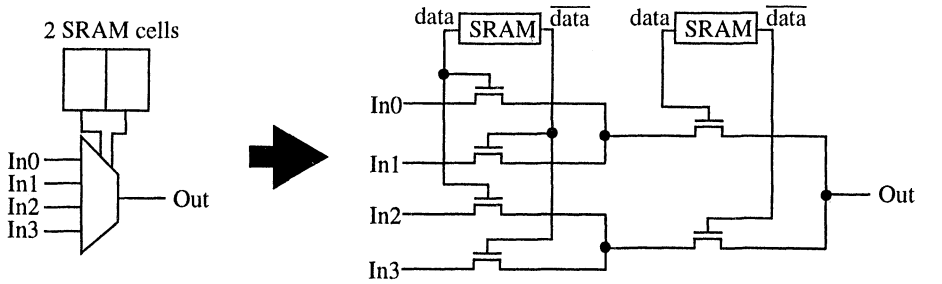


FIGURE B.7 Schematic of a 4-input SRAM-controlled multiplexer.

Since we are implementing multiplexers as pass transistor trees, we need buffers on the inputs and the output of the multiplexer. If these buffers are not already present in the circuit, they must be added and their area included in the total area.

B.1.2 Logic Block Structures

Figure B.8 shows the logic block used in Chapter 7: a logic cluster of size 4 with 10 logic inputs and one clock. We also assume there is one set/reset signal of programmable polarity shared by all four registers in the logic block. Notice that signals that have to travel a significant distance across the logic block or which have a large fan-out pass through two-stage buffers with a drive strength of two to four times the minimum to amplify them. The fan-out of some of these buffers is a function of the logic cluster size — our area model resizes these buffers appropriately for the fan-out present in each logic cluster. Note that to keep Figure B.8 understandable, we do not show all the repeated structures in a logic cluster of size 4. Specifically, we show only two of the ten input pin buffers, two of the sixteen 14:1 multiplexers, and one each of the four 4-LUTs, registers, and BLE output buffers contained in a size 4 logic cluster.

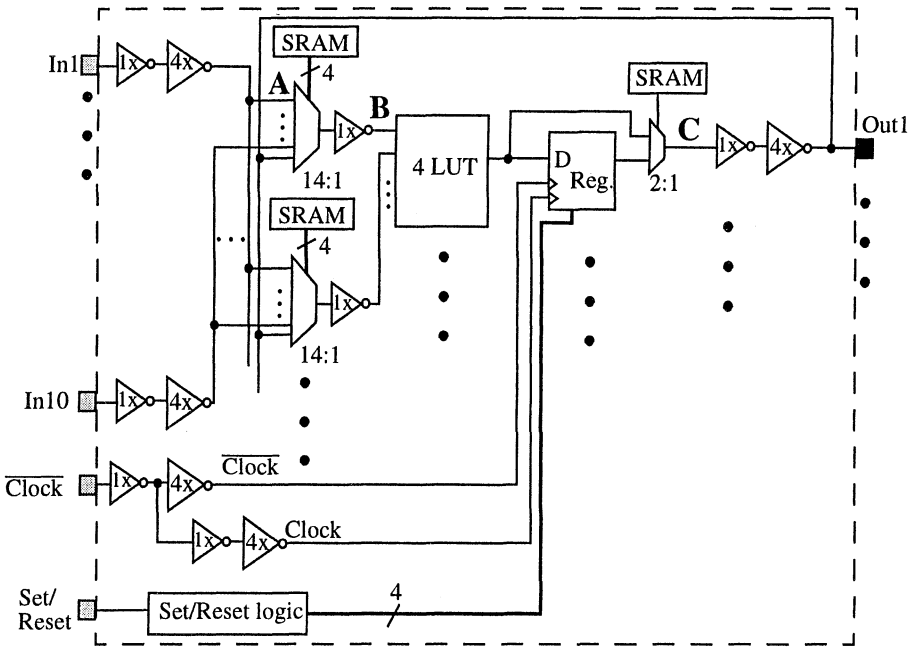


FIGURE B.8 Schematic of a logic cluster of size 4 with 10 logic inputs, 1 clock and a set/reset input.

As well, note that the four times minimum strength buffer driving the output pin is strong enough to drive the local routing, but not, in general, the routing tracks. Appendix C.3 discusses the size of output buffer we use to drive different types of routing wires. The **A**, **B**, and **C** points labelled in Figure B.8 will be referred to in Section B.2, where we summarize the delay of the various paths in this logic block.

Figure B.9 shows our implementation of a 4-input look-up table. It uses sixteen SRAM cells, a sixteen-input pass transistor multiplexer, and a set of buffers to amplify the select inputs and generate their complements. Note that only 6 of the 16

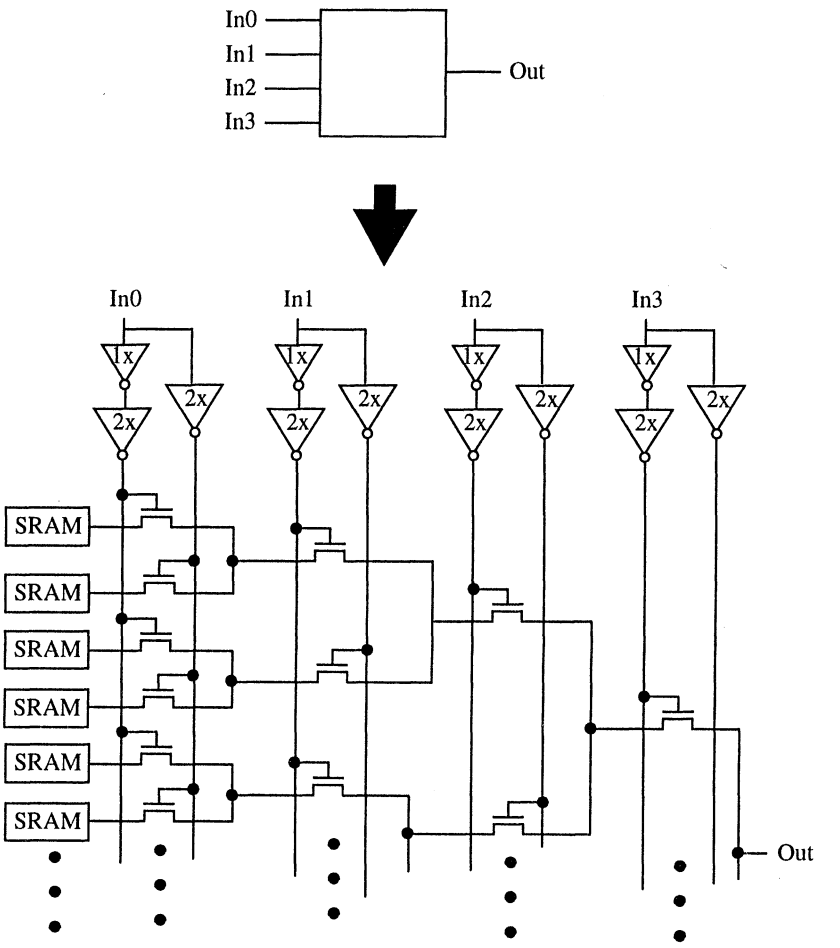


FIGURE B.9 Schematic of a 4-input look-up table.

SRAM cells are shown, and only a portion of the pass transistor multiplexer tree is shown. The area of this 4-LUT is 96 (SRAM cells) + 30 (multiplexer tree) + 41 (input buffers and complementers) = 167 minimum-width transistor areas.

Figure B.10 shows the master-slave register with asynchronous set and reset inputs that we use. Numbers given after a slash (/) denote the channel length (in μm) of transistors that use a longer channel than the minimum 0.35 μm. This register contains 16 transistors, but some are larger than minimum width and some have greater than the minimum channel length, so it occupies 19 minimum-width transistor areas.

Finally, Figure B.11 shows the set/reset logic circuit used to enable programming of both the polarity of the set/reset input to a logic block, and whether this signal sets or resets the registers in a logic block. This circuitry occupies 48.5 minimum-width transistor areas.

Summing the areas of all the structures in the size four logic cluster of Figure B.8 yields the total area: 2·9.25 (clock buffers) + 16·50 (local routing multiplexers) + 16·2.35 (buffers on outputs of local routing multiplexers) + 4·167 (four 4-LUTs) + 4·19 (four registers) + 4·8 (four 2:1 multiplexers) + 48.5 (set/reset logic) = 1678 minimum-width transistor areas for this logic block. The highly attentive reader will notice that this summation does not include the ten buffers on the logic block input pins, nor the four buffers on the logic block output pins. The area of these buffers is counted as part of the routing area by VPR.

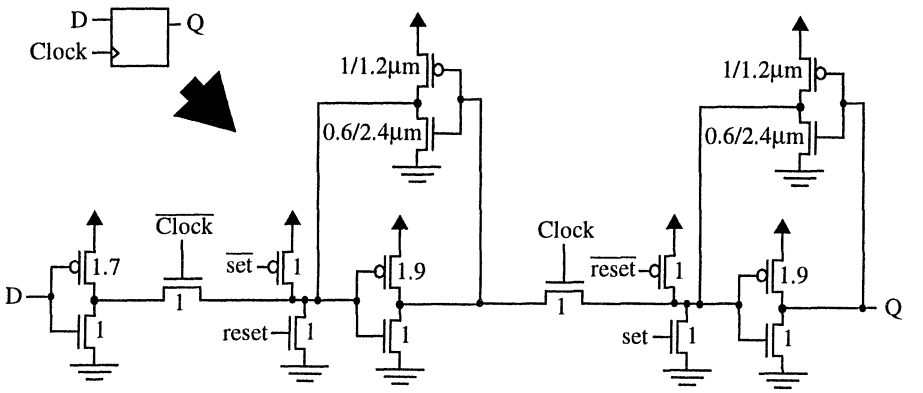


FIGURE B.10 Master-slave register with asynchronous set and reset.

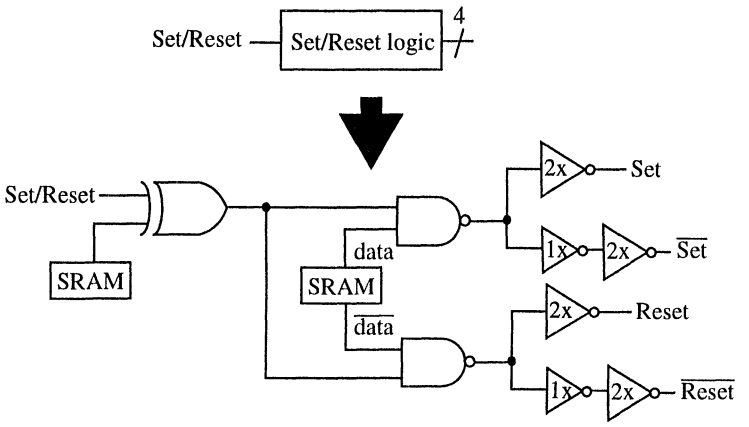


FIGURE B.11 Circuitry to generate and amplify the set and reset signals within a logic block.

B.2 Delay and RC-Equivalent Circuit Extraction

In this section, we describe how we determined the delay or equivalent RC circuit for each FPGA structure, and we list the delay through some of the key FPGA structures. This delay and RC data is required to apply the FPGA delay model described in Section 6.2.3.

We first performed SPICE simulations to determine the delay of all the paths in the logic block shown in Figure B.8, as well as the delay from a routing track to a logic block input pin. The delay of a path is the difference between the time at which the path input signal passes through the midpoint of its voltage swing and the time at which the output signal passes through the midpoint of its voltage swing. The results are listed in Table B.1. The letters in Table B.1 refer to the points labelled on Figure B.8.

We performed SPICE simulations to determine the intrinsic delay of the (often multi-stage) buffers we use in the routing. The buffer intrinsic delay is the delay, when the buffer is unloaded, between the buffer input passing through its 50% swing voltage and the buffer output passing through its 50% swing voltage. Because FPGA routing switches are heavily loaded, the slew rate of the voltage input signal to a routing buffer is fairly low. This slowly changing input affects the delay through the buffers — we found that the buffer delay is 100 ps higher for buffers in the FPGA routing

TABLE B.1 Cluster-based logic block of size 4 delay values in TSMC's 0.35 μm CMOS process.

Path	Passes through	Delay (ps)
Routing track → Point **A**	Track buffer, connection block multiplexer, input pin buffers	1040
Point **A** → Point **B**	14:1 multiplexer, buffer	395
Point **B** → Point **C**	4-LUT, 2:1 multiplexer	465
Point **C** → Point **A**	Local routing buffers	280
Point **B** → Register storage node	4-LUT, register set-up time	670
Register storage node → Point **C**	Register clock-to-Q delay, 2:1 multiplexer	332

than it is for buffers in the FPGA logic block because of the difference in input slew rates. We do not provide the buffer intrinsic delays here because it would violate our non-disclosure agreement with TSMC.

We found the equivalent resistance of nMOS pass transistors via the circuit shown in Figure B.12. The Elmore delay of this circuit is simply $R_{eq,n} \cdot C$. Since our logic block and buffer intrinsic delays all give the time required for 50% signal swing, we want the Elmore delay of the circuit in Figure B.12 to give us the 50% signal swing time as well. We accomplish this simply by setting $R_{eq,n} \cdot C$ equal to the 50% swing time measured from SPICE, and solving for $R_{eq,n}$. We found that the rise and fall times of signals generated by pass transistors were essentially equal, so long as we measure the time for the signal to change to 50% of its final value, and not to $V_{dd}/2$. By properly sizing the n and p transistors in the buffer used to sense the state of this routing wire (the dotted buffer in Figure B.12), we can adjust the "switch point volt-

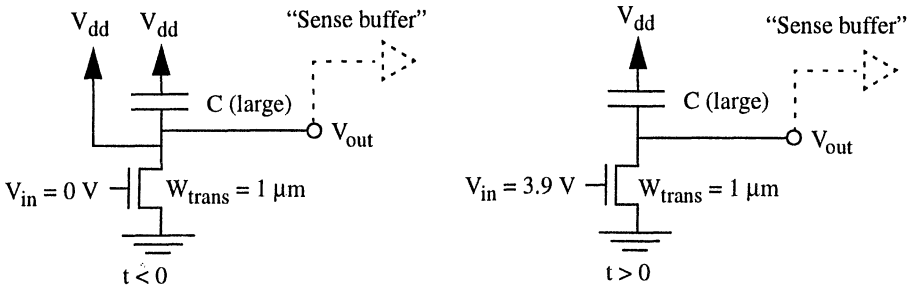


FIGURE B.12 Circuit used to determine equivalent resistance of nMOS transistors.

age" to lie at this "signal swing midpoint" voltage. Note that the drive strength of CMOS transistors increases linearly with transistor width, so the equivalent resistance of transistors with widths different than 1 μm is simply R_{eq} / W_{trans}. A similar procedure was used to find the equivalent output resistance of pMOS transistors and tri-state buffers. Again, our nondisclosure agreement precludes the listing of these equivalent resistances here.

To determine the metal resistance and capacitance of routing wires, we need to know how long these wires are. In other words, we need to know the size of a basic FPGA tile. For example, consider an FPGA employing a size 4 logic cluster as its logic block. We determined the basic tile size by examining a representative routing architecture (50% length 4 buffered wires, 50% length 4 pass-transistor-switched wires) as follows:

1. The most difficult to route benchmark circuit, pdc, requires 62 tracks per channel in this architecture; hence we set the channel width to 62.
2. The number of minimum-width transistor areas in the routing for this architecture and a channel width of 62 is 6857. Adding in the 1678 minimum-width transistor areas in the logic block yields a total of 8535.
3. We assume that a good layout team can achieve 60% of the maximum achievable transistor density. Therefore:

$$Area(\mu m^2) = MinWidthTransAreas \cdot \frac{6.2275\ \mu m^2 / \text{MinWidthTrans}}{0.6} = 88\,600\ \mu m^2 \quad \textbf{(B.2)}$$

Assuming a square logic block, then, we obtain a tile size of 300 μm on a side; we assume this is the tile size for each architecture examined in Chapter 7. To check that this value is reasonable, we also measured the area per logic block of a Xilinx XC4085XL FPGA (in a 0.35 μm process) and found that it is 115 000 μm^2, which leads to an (assumed square) tile size of 340 μm x 340 μm. The area of an Altera Flex10K100 tile is 259 000 μm^2 in a 0.5 μm^2 process; scaling to a 0.35 μm process and dividing by two (since each Altera logic block contains 8 4-LUTs, rather than 4), yields an area per four 4-LUT tile of 129 600 μm^2, or 360 μm x 360 μm in a square layout. Consequently, our estimate of a tile size of 300 μm on a side for a logic block containing four 4-LUTs looks quite reasonable.

In modelling metal resistance, we want $R_{metal,eq} \cdot C$ to give the 50% signal swing delay so that it is compatible with our other delay values. Since metal resistance is a true linear resistor, we know that the voltage on a capacitor being charged by resistive metal will follow an exponential curve in time. The time constant of this exponential is $R_{metal} \cdot C$, where R_{metal} is the "true" metal resistance given in the foundry data sheets. Consequently, we know that the 50% signal swing delay will occur at:

$$t_{50\%\,\text{swing}} = \ln(0.5) \cdot R_{metal} \cdot C = 0.693 \cdot R_{metal} \cdot C\,. \tag{B.3}$$

From (B.3) it is clear that $R_{metal,eq}$ is $0.693 \cdot R_{metal}$.

The metal capacitance data can be taken directly from the foundry data sheets. We extracted the transistor gate capacitance, and the pMOS and nMOS diffusion capacitances, from the foundry SPICE models. Since the diffusion capacitances are bias voltage dependent, we obtained the average diffusion capacitances over the entire bias range via:

$$C_{\text{diff,av}} = \frac{1}{3.3} \cdot \int_0^{3.3} C_{\text{diff}}(V)dV \tag{B.4}$$

and used this average diffusion capacitance value in our RC equivalent circuit model, as suggested in [156].

Finally, recall from Section B.1.1 that we are boosting the gates of pass transistors to 3.9 V in order to control the FPGA static power dissipation. It is interesting to note that this gate boosting does not change the routing delay very much — the transistor drive strength increases, but so does the signal swing, and the two effects largely cancel each other.

APPENDIX C

Sizing of Routing Transistors and Metal

Both the area and delay of an FPGA depend strongly on the size of the routing transistors and the metal width of the routing wires. In this appendix we show how we determined appropriate sizes for the pass transistors and tri-state buffers used in FPGA routing, and the best metal width for routing wires, for use with a size 4 logic cluster. All the delay values in this section were determined from the switch-level (resistors, capacitors and ideal delay elements) delay model we extracted from TSMC's 0.35 μm process; this delay modelling procedure was described in Appendix B.2. Throughout this section we also assume that the layout area of a basic FPGA tile (in this process) is 300 μm x 300 μm, as described in Appendix B.2. A size 4 logic cluster results in a basic tile of this size; for logic clusters of different sizes the tile size, and hence the best routing transistor sizes, will change.

C.1 Sizing Pass Transistor Routing Switches

In this section, we determine the best routing pass transistor width by evaluating the effect of different pass transistor widths on both the speed of the routing wire shown in Figure C.1, and the area of the routing switches. While Figure C.1 shows a wire of length 4, we will vary the length (L_{wire}) of this routing wire over a wide range in this section. We assume the wire is fully switch-block and connection-block populated, and that the switch block topology used leads to an F_s of 3 at the wire ends, and an F_s value of 1 at the "internal" switch block points. The disjoint switch block topology used in Chapters 6 and 7 matches these F_s values. We also assume there is a pass transistor allowing this wire to connect to a logic block output pin at each of the L_{wire}

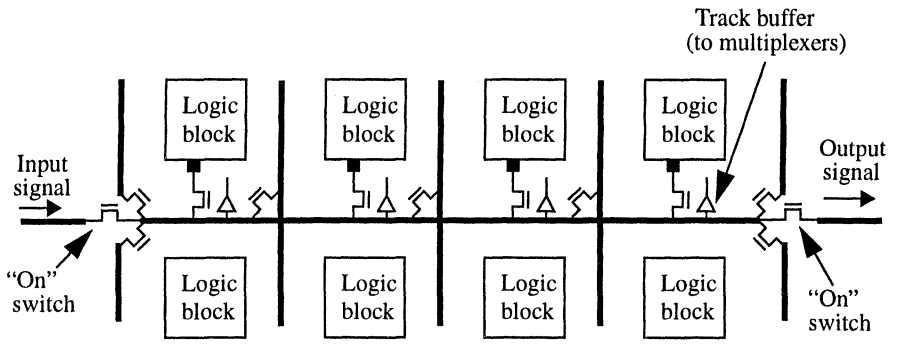


FIGURE C.1 Wiring topology assumed in pass transistor sizing experiments.

logic blocks spanned by the wire. These transistors connecting the wire to logic block output pins are the same size as the other routing transistors. A track buffer with a small first stage (as described in Appendix B.1.1) also loads the wire at each logic block it passes. Finally, in this section we assume that the routing wire is laid out in metal 3 with minimum width and minimum spacing. We are interested in determining the delay of the wire in Figure C.1 when one pass transistor at each end is "on" to create a multi-wire connection — all the other switches are off and only add capacitive loading.

The Elmore delay of a connection that passes through M pass transistor switches and routing wires is:

$$T_{Elmore}(M) = M \cdot \left[\frac{(M+1)}{2}(R_{pass} + R_{metal})C_{total} - \frac{R_{metal}C_{total}}{2}\right] \quad \textbf{(C.1)}$$

where C_{total} is the total (metal + transistor parasitics) capacitance loading the wire, R_{pass} is the equivalent resistance of a pass transistor, and R_{metal} is the end-to-end resistance of the wire. The dominant (quadratic in the number of routing switches in series) delay term is proportional to:

$$\text{Dominant delay constant} = (R_{pass} + R_{metal})C_{total}. \quad \textbf{(C.2)}$$

Figure C.2 plots this dominant delay constant as a function of the pass transistor width for several different wire lengths. The delay constant of each wire has been divided by the wire length, L_{wire}, to allow all the curves to be plotted on the same axes — this does not affect the shape of the curves in any way. The horizontal axis in Figure C.2 is the width, relative to the minimum contacted transistor width of 0.7 μm, of the routing pass transistors. As the width of the pass transistors increases, the wir-

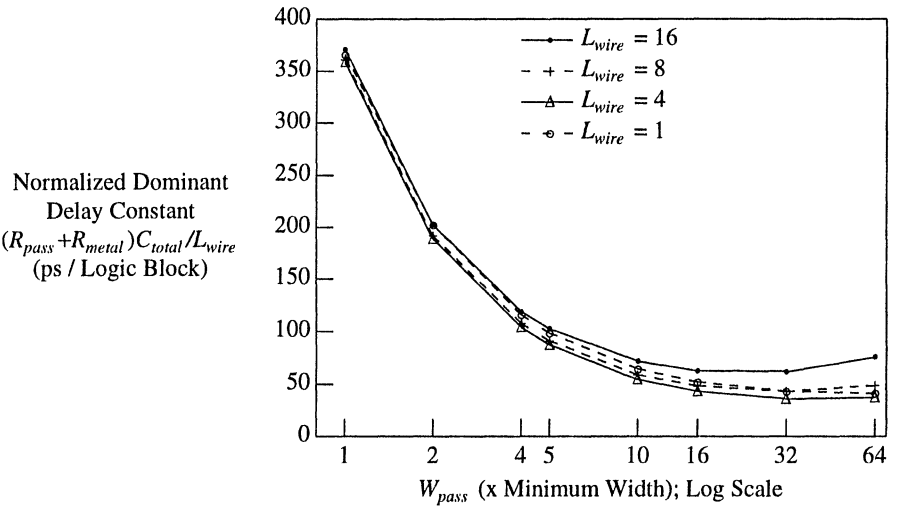


FIGURE C.2 Normalized dominant delay constant vs. routing pass transistor width.

ing delay clearly drops significantly. For very large pass transistors the delay stops improving, since the switch capacitances become larger than the metal capacitance and the wire becomes self-loaded. For the longer wires, delay actually degrades once the transistor width is more than about 30 times minimum, since the metal resistance begins to become more significant than the pass transistor resistance.

Figure C.3 shows the area required by each pass transistor routing switch, including the area of the controlling SRAM cell. Obviously the area of each routing switch increases as the pass transistor size is increased. The question, then, is how best to size the routing pass transistors to achieve the best area-delay trade-off. Intuitively, we want to choose the transistor width that is nearest the "knees" of both the delay constant and switch area curves. A more quantitative way of formulating this goal is to size routing transistors to minimize the area-delay product of the resulting FPGA. As described in Section 6.2.4 minimizing the area-delay product of an FPGA maximizes the FPGA's computational throughput (for parallel algorithms).

To find the FPGA with the minimum area-delay product achievable with a given logic block and routing architecture, we would have to simultaneously vary the size of every transistor in a basic tile of the FPGA, and determine the area and delay achieved by the FPGA under each transistor sizing. Clearly this is a dauntingly large and complex search space. We can, however, *approximately* minimize the area-delay product

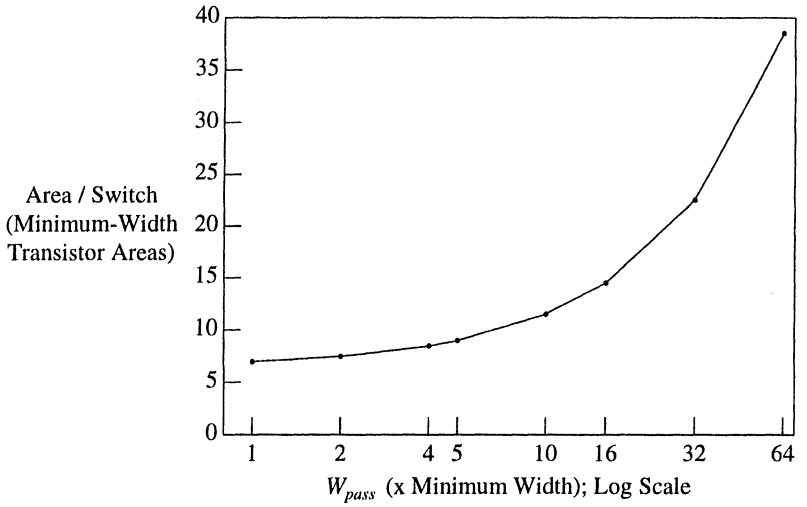


FIGURE C.3 Area of a pass transistor routing switch (including the controlling SRAM bit) vs. pass transistor width.

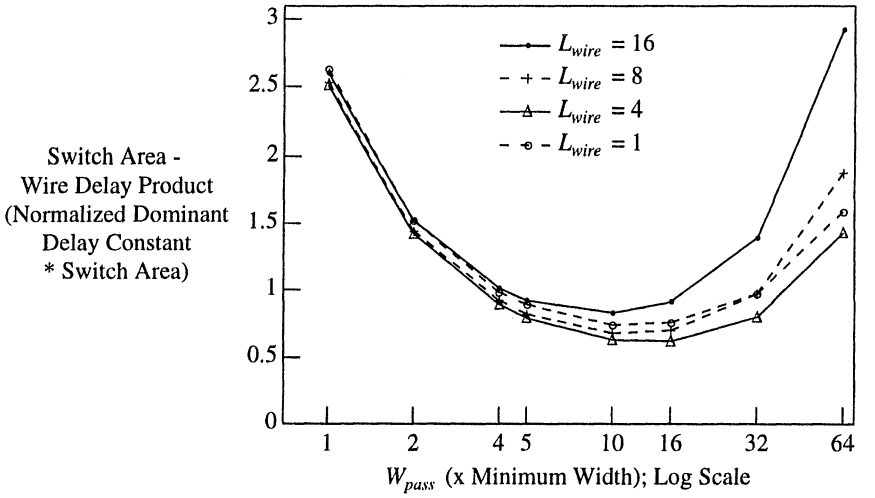


FIGURE C.4 Switch area - wire delay product vs. routing pass transistor width.

of an FPGA by minimizing the routing switch area - wire delay product for each routing resource, and this is the approach we take to routing transistor sizing.

Figure C.4 plots the area of a routing switch multiplied by the dominant wire delay constant as a function of pass transistor width. Again, curves for several different routing wire lengths are shown, and the switch area - wire delay product of each is normalized by dividing by the wire length so they can all be plotted on the same axes. For wire lengths of 1, 4, or 8 logic blocks, transistor widths of 10 and 16 times the minimum are essentially tied for the best area - delay product. For a wire of length 16, a pass transistor width of 10 is clearly preferable to a width of 16. Consequently we use ten times minimum width (i.e. 7 μm width) pass transistor routing switches throughout Chapter 7.

Although we assumed minimum spacing metal 3 wiring in this section, laying out routing wires in metal 2 or using a different metal spacing does not significantly change the point at which the best switch area - wire delay product occurs.

C.2 Sizing Tri-State Buffer Routing Switches

To determine the best size of tri-state buffers in the routing, we followed a procedure identical to that of the prior section. The wire simulated is the same as that in Figure C.1, except every pass transistor between two routing wires is replaced by two tri-state buffers — one in each direction. We assume that one buffer chain is shared by the three tri-state buffers at each of the two end points of a wire; i.e. we are assuming the fan-out-buffer-sharing case described in Appendix B.1. Once again the routing wire is fabricated in metal 3 using the minimum width and spacing.

Figure C.5 shows the wire delay divided by the wire length (i.e. the delay to pass one logic block) versus the size of the tri-state routing buffers. A buffer of minimum size uses minimum width nMOS transistors, while the pMOS pull-up is 1.9 times minimum width, as described in Appendix B.1. Also recall from Appendix B.1 that the larger buffers are multi-stage buffers, and the stage ratio is kept as close to 4 as possible. As buffer size increases from the minimum size, speeds improve for all wire lengths. Once the buffer is larger than four times the minimum, however, the speed of length 1 wires starts to degrade. This occurs because the increase in buffer intrinsic delay as the buffer grows is larger than the decrease in the time it takes the larger buffer to discharge the routing wire (and attached switch) capacitance. Longer wires continue to see some speed improvement until the buffer size reaches 16 times the minimum (for length 4 wires) or 32 times the minimum (for length 8 and 16 wires).

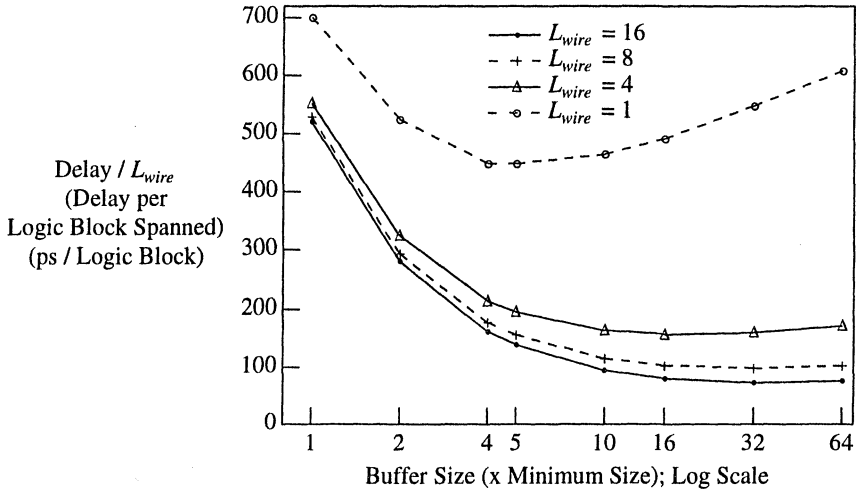


FIGURE C.5 Delay per logic block spanned for buffered routing wires vs. buffer size.

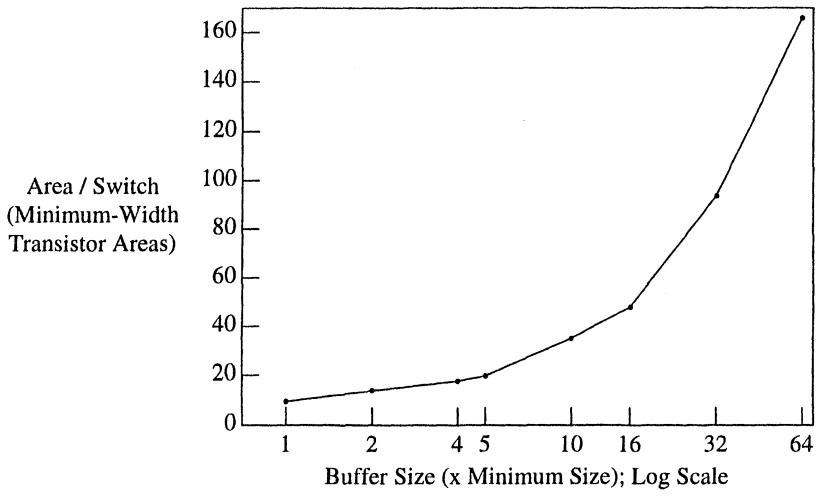


FIGURE C.6 Area of a tri-state buffer switch (including controlling SRAM bit) vs. drive strength.

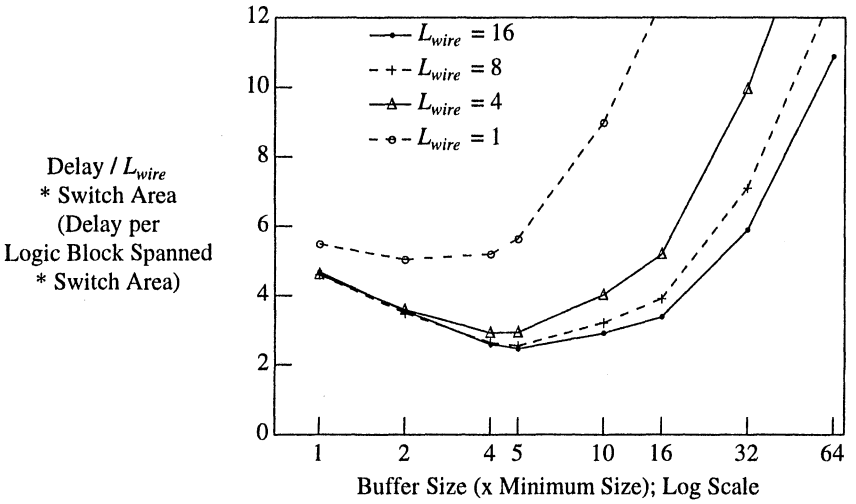


FIGURE C.7 Switch area - wire delay product versus routing tri-state buffer size.

Figure C.6 shows how the area of each tri-state buffer routing switch (including the controlling SRAM bit) varies with the size of the tri-state buffer. Once again, we want to find the buffer size closest to the "knees" of both the area and the delay curves, and once again we take this best trade-off point to be the minimum of the wire delay - routing switch area product. Figure C.7 shows the wire delay - routing switch area product curve for four different routing wire lengths. For wires of length 4, 8 and 16 the best switch area - wire delay product occurs with a buffer that is five times the minimum size. For a wire of length 1, the best area-delay product occurs with a buffer size of only twice the minimum. Since the area-delay product for a wire of length 1 is not much worse with a buffer size of five times the minimum, however, we make all tri-state routing buffers five times the minimum size throughout Chapter 7.

It is interesting to note that the best routing pass transistor width was ten times the minimum, while the best tri-state buffer size is only five times the minimum. There are two reasons for this behaviour. First, as the size of a tri-state buffer is increased, more stages are added to the buffer chain. Thus some of the speed gained by the increased buffer drive strength is counteracted by the increased intrinsic delay of the buffer. Second, since a tri-state buffer contains several transistors, it consumes more area at a given size than a pass transistor. Consequently, as a tri-state buffer is sized up, it more rapidly swamps the fixed area overhead due to its controlling SRAM bit, so its area growth is closer to linear in the buffer size than that of a pass transistor.

C.3 Tri-State Buffers in Output Pin Connection Blocks

The experiments in the prior two sections determined the proper transistor sizes for routing switches contained in FPGA switch blocks. We must also determine a proper size for the tri-state buffers connecting logic block output pins to routing tracks. It makes little sense to connect a routing wire to a logic block output pin with a large tri-state buffer if the routing switches the wire uses to connect to other routing wires are small. The large tri-state buffer from the output pin will make the connection from the logic block to the routing wire fast, but subsequent connections through switch blocks will be slow. Consequently, the extra area used to create a large tri-state buffer from the output pin to the routing wire is largely wasted.

On the other hand, it also makes little sense to connect a logic block output pin to a routing wire with a tri-state buffer that is smaller than the switch block routing switches on this wire. The connection from the output pin to the routing wire will be slow, degrading the speed of what was intended to be a fast routing resource. Consequently, in Chapters 6 and 7 we always make the tri-state buffer connecting a logic block output pin to a routing wire the same size as the routing switches the wire uses to connect to other routing wires. This means the tri-state buffer connecting output pins to pass-transistor-switched wires are ten times the minimum size, while those connecting to buffer-switched wires are only five times the minimum size.

C.4 Metal Width and Spacing

The results in Appendices C.1 and C.2 assumed that routing wires used the minimum metal width and spacing. Increasing the spacing between metal wires reduces the metal capacitance, while increasing the metal width reduces the metal resistance, at the cost of some increase in the metal capacitance. Of course, increasing either the metal width or the metal spacing increases the metal pitch; this may cause in an increase in the FPGA area (if the metal area becomes too large to fit over the transistor area). In this section we determine if FPGA routing wires benefit from greater than minimum metal width or spacing. We again assume that routing wires are laid out in metal 3.

Figure C.8 shows the switch area - dominant delay constant product for a wire of length 16 connected by pass transistor routing switches as a function of the pass transistor width. The three curves in Figure C.8 show the area-delay product achievable under three different metal widths and spacings. The solid curve occurs with minimum metal width and spacing; the dashed curve occurs with minimum width and

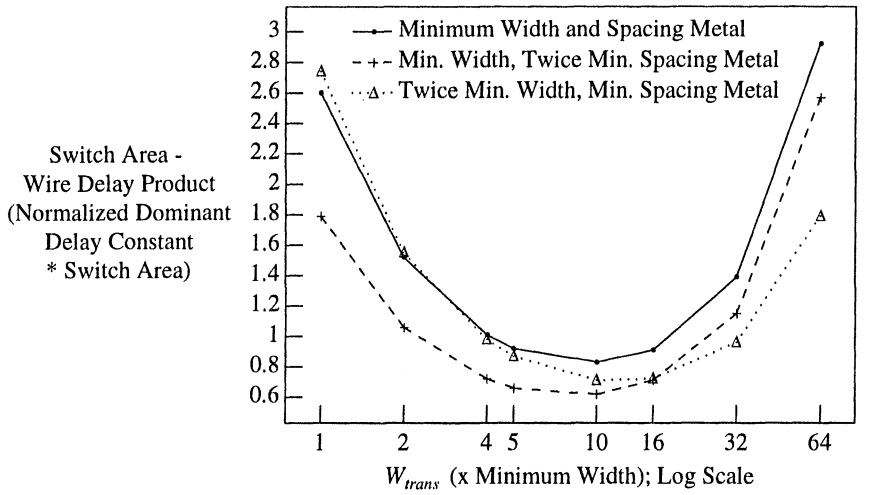


FIGURE C.8 Effect of metal width and spacing on the switch area - wire delay product for a length 16 pass-transistor-switched wire.

twice the minimum spacing (leading to a metal pitch of 1.5 times the minimum metal pitch); and the dotted curve occurs with twice the minimum width and minimum spacing (also leading to a metal pitch of 1.5 times the minimum metal pitch). Notice first that the minimum area-delay product occurs at a pass transistor width of ten times the minimum width for all three curves, showing that the transistor sizing result we obtained in Appendix C.1 is not very sensitive to the metal width and spacing assumed. Secondly, notice that the twice-minimum-spacing curve has an area-delay product at this minimum point that is 13% better than the area-delay product of the twice-minimum-width curve. The entire 13% area-delay product improvement is due to lower delay, since both curves are using the same size of routing switch (ten times minimum width) at this point. Clearly, spacing out metal wires yields greater speed improvements than widening wires in the routing transistor width regime of interest, and is hence a better use of metal area. Note also that we have intentionally presented results for the case that is most favourable to widening the routing wires — a long (length 16) wire. Widening the routing wires performs even more poorly, relative to increasing the wire spacing, for shorter wires.

Figure C.9 shows how the switch area - wire delay product varies with metal width and spacing for a length 16 wire that is driven by tri-state buffer routing switches. Once again, the solid line uses the minimum metal width and spacing, while the dashed line uses twice the minimum metal spacing, and the dotted line uses twice the minimum metal width. Notice that the minimum switch area - wire delay product

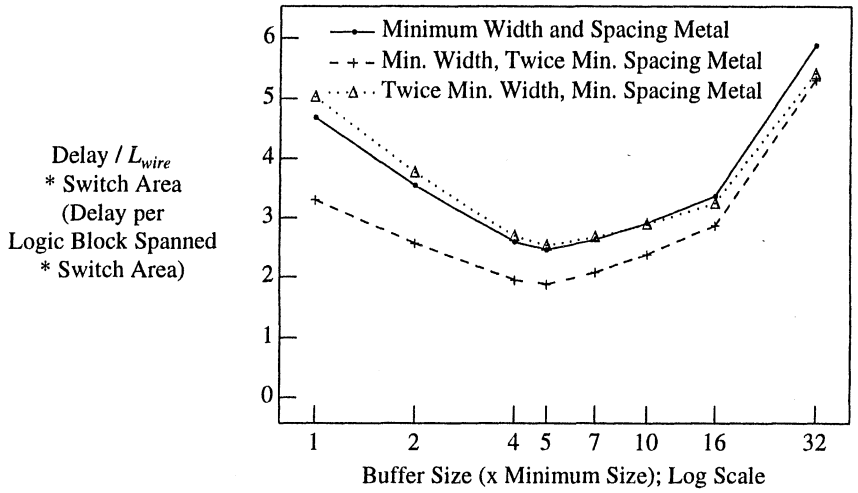


FIGURE C.9 Effect of metal width and spacing on the switch area - wire delay product for a a length 16 buffered routing wire.

occurs at a buffer size of five times the minimum for each of the three curves, indicating that our buffer sizing results of Appendix C.2 are also fairly insensitive to the metal width and spacing of the routing wires. The routing wire using twice the minimum spacing now has a 26% better area-delay product at the minimum area-delay point than the routing wire that uses twice the minimum width. Once again the entire 26% improvement in the area-delay product is due to delay improvement, since both the curves are using the same buffer size (five times minimum size) at this point. Buffered routing switches prevent the "build-up" of wiring resistance when several routing wires and switches are connected in series to form a longer connection. Since the sole benefit of widening a routing wire is a reduction in the wire metal resistance, the fact that widening a routing wire is an even less effective speed optimization with buffered routing wires than with pass-transistor-switched wires is understandable. In fact, widening the routing wire has slightly increased the area-delay product at the minimum point versus using a minimum width wire. Notice that we have again presented results for a long (length 16) routing wire — shorter wires favour increasing the wire spacing over increasing the wire width even more.

It is understandable that increasing the metal spacing improves the routing wire speed, since a wider metal spacing reduces the metal capacitance. On the other hand, it may seem surprising that increasing the metal width is so ineffective, and in fact degrades speed for buffered wires or short pass-transistor-switched wires. Recall that while increasing metal width decreases the metal resistance, it increases the metal capacitance. In standard cell designs this increase in metal capacitance is not an over-

riding issue — one can simply increase the size of the buffer driving the wire to compensate. In FPGA routing, however, we have not one buffer or pass transistor driving a wire, but many possible drivers hanging off the wire along its length. The area cost of sizing up all these buffers or pass transistors to compensate for the increased metal capacitance is significant, and cannot be ignored. Normally these buffers or pass transistors are small enough that their equivalent resistance is considerably higher than the metal resistance of even a fairly long, minimum width wire. Consequently, when one is unwilling to tolerate a large increase in the routing area, increasing the spacing between metal wires is a more effective method of speeding up an FPGA's routing than increasing the wire width (in a 0.35 μm process — future process shrinks may eventually make wider wires more attractive). Accordingly, we investigate the effect of increasing the spacing between routing wires on FPGA speed (in Section 7.6), but we use minimum width metal wires throughout this work.

References

[1] S. Brown, R. Francis, J. Rose and Z. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[2] V. Betz and J. Rose, "Directional Bias and Non-Uniformity in FPGA Global Routing Architectures," *ICCAD*, 1996, pp. 652 - 659.

[3] V. Betz and J. Rose, "Effect of the Prefabricated Routing Track Distribution on FPGA Area-Efficiency," *IEEE Trans. on VLSI*, Sept. 1998, pp. 445 - 456.

[4] Xilinx Inc., *The Programmable Logic Data Book*, 1994.

[5] Lucent Technologies, *FPGA Data Book*, 1998.

[6] Actel Inc., *FPGA Data Book and Design Guide*, 1994.

[7] Altera Inc., *Data Book*, 1998.

[8] V. Betz and J. Rose, "Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size," *CICC*, 1997, pp. 551 - 554.

[9] V. Betz and J. Rose, "How Much Logic Should Go in an FPGA Logic Block?," *IEEE Design and Test Magazine*, Spring 1998, pp. 10 - 15.

[10] A. Marquardt, V. Betz and J. Rose, "Using Cluster-Based Logic Blocks to Improve FPGA Speed and Density," *To appear in 1999 ACM Symp. on FPGAs.*

[11] D. Tavana, W. Lee, S. Young, and B. Fawcett, "Logic Block and Routing Considerations for a New SRAM-Based FPGA Architecture," *CICC*, 1995, ppp. 24.6.1 - 24.6.4.

[12] D. Bursky, "Programmable Arrays Mix FPGA and ASIC Blocks," *Electronic Design*, Oct. 14, 1996, pp. 69 - 74.

[13] Vantis Corporation, "VF1 Field Programmable Gate Arrays," *Preliminary Data Sheet*, 1998.

[14] V. Betz and J. Rose, "FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density," *To appear in 1999 ACM Int. Symp. on FPGAs.*

[15] S. Trimberger, "A Reprogrammable Gate Array and Applications," *Proceedings of the IEEE*, July 1993, pp. 1030 - 1041.

[16] J. Rose and D. Hill, "Architectural and Physical Design Challenges for One-Million Gate FPGAs and Beyond," *ACM Int. Symp. on FPGAs*, 1997, pp. 129 - 132.

[17] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Int. Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213 - 222.

[18] V. Betz, "Architecture and CAD for Speed and Area Optimization of FPGAs," *Ph.D. Thesis*, University of Toronto, 1998.

[19] A. Marquardt, *M.A.Sc. Thesis*, Univeristy of Toronto, *in Preparation.*

[20] J. Rose, A. El Gamal and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE*, July 1993, pp. 1013 - 1029.

[21] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez and I. Rahardja, "The Design of an SRAM-Based Field-Programmable Gate Array, Part II: Circuit Design and Layout," *To appear in IEEE Trans. on VLSI.*

[22] M. Khellah, S. Brown and Z. Vranesic, "Modelling Routing Delays in SRAM-based FPGAs," *Proc. Canadian Conf. on VLSI*, 1993, pp. 6B.13 - 6B.18.

[23] J. Greene, E. Hamdy and S. Beal, "Antifuse Field Programmable Gate Arrays," *Proceedings of the IEEE*, July 1993, pp. 1042 - 1056.

[24] S. Brown, "An Overview of Technology, Architecture and CAD Tools for Programmable Logic Devices," *CICC*, 1994, pp. 69 - 76.

[25] J. Rose, R. Francis, D. Lewis and P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *JSSC*, Oct. 1990, pp. 1217 - 1225.

[26] J. He and J. Rose, "Advantages of Heterogenous Logic Block Architectures for FPGAs," *CICC*, 1993, pp. 7.4.1 - 7.4.5.

[27] K. Chung, S. Singh, J. Rose and P. Chow, "Using Hierarchical Logic Blocks to Improve the Speed of Field-Programmable Gate Arrays," *Int. Workshop on Field Programmable Logic and Applications*, 1991.

[28] J. Cong and Y. Hwang, "Boolean Matching for Complex PLBs in LUT-based FPGAs with Application to Architecture Evaluation," *ACM Symp. on FPGA*s, 1998, pp. 27 - 34.

[29] A. Aggarwal and D. Lewis, "Routing Architectures for Hierarchical Field Programmable Gate Arrays," *ICCD*, 1994, pp. 475 - 478.

[30] J. Rose and S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *JSSC*, March 1991, pp. 277 - 282.

[31] B. Tseng, J. Rose and S. Brown, "Using Architectural and CAD Interactions to Improve FPGA Routing Architectures," *ACM Workshop on FPGAs*, 1992, pp. 3 - 8.

[32] Y. Chang, D. F. Wong, and C. K. Wong, "Universal Switch Modules for FPGA Design," *ACM Trans. on Design Automation of Electronic Systems*, Jan. 1996, pp. 80 - 101

[33] S. Wilton, "Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories," *Ph.D. Dissertation*, University of Toronto, 1997. (*Available for download from http://www.ee.ubc.ca/~stevew/publications.html*).

[34] J. Greene, V. Roychowdhury, S. Kaptanoglu and A. El Gamal, "Segmented Channel Routing," *DAC*, 1990, pp. 567 - 572.

[35] K. Roy and M. Mehendale, "Optimization of Channel Segmentation for Channelled Architecture FPGAs," *CICC*, 1992, pp. 4.4.1 - 4.4.4.

[36] S. Brown, M. Khellah and G. Lemieux, "Segmented Routing for Speed-Performance and Routability in Field-Programmable Gate Arrays," *Journal of VLSI Design*, Vol. 4, No. 4, 1996, pp. 275 - 291.

[37] M. Khellah, S. Brown and Z. Vranesic, "Minimizing Interconnection Delays in Array-Based FPGAs," *CICC*, 1994, pp. 181 - 184.

[38] S. Brown, M. Khellah and Z. Vranesic, "Minimizing FPGA Interconnect Delays," *IEEE Design and Test Magazine*, Winter 1996, pp. 16 - 23.

[39] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez and I. Rahardja, "The Design of an SRAM-Based Field-Programmable Gate Array, Part I: Architecture," *To appear in IEEE Trans. on VLSI.*

[40] Y. Sun, T. Wang, C. Wong and C. Liu, "Routing for Symmetric FPGA's and FPIC's," *IEEE Trans. on CAD*, Jan. 1997, pp. 20 - 31.

[41] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, "Synthesis Methods for Field-Programmable Gate Arrays," *Proceedings of the IEEE*, July 1993, pp. 1057 - 1083.

[42] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel Logic Syntesis," *Proceedings of the IEEE*, Feb. 1990, pp. 264 - 300.

[43] R. Francis, J. Rose and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," *DAC*, 1991, pp. 227 - 233.

[44] R. Francis, J. Rose and Z. Vranesic, "Technology Mapping Lookup Table-Based FPGAs for Performance" *ICCAD*, 1991, pp. 568 - 571.

[45] K. C. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar, "DAG-Map: Graph-Based FPGA Technology Mapping for Delay Optimization," *IEEE Design and Test Magzine*, Sept. 1992, pp. 7 - 20.

[46] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *IEEE Trans. on CAD*, Jan. 1994, pp. 1 - 12.

[47] J. Cong and Y. Ding, "On Area / Depth Trade-Off in LUT-Based FPGA Technology Mapping," *IEEE Trans. on VLSI*, June 1994, pp. 137 - 148.

[48] C. Alpert and A. Kahng, "Recent Directions in Netlist Partitioning: A Survey," *Integration, the VLSI Journal*, Vol. 19 (1-2), 1995, pp. 1 - 81.

[49] J. Cong, H. Li, S. Lim, T. Shibuya, and D. Xu, "Large Scale Circuit Partitioning with Loose / Stable Net Removal and Signal Flow Based Clustering," *ICCAD*, 1997, pp. 441 - 446.

[50] L. Hagen and A. Kahng, "A New Approach to Effective Circuit Clustering," *ICCAD*, 1992, pp. 422 - 427.

[51] J. Cong and M. Smith, "A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design," *DAC*, 1993, pp. 755 - 760.

[52] G. Saucier, D. Brasen, J. Hiol, "Partitioning with Cone Structures," *ICCAD* 1993, pp. 236 - 239.

[53] L. Hagen and A. Kahng, "Fast Spectral Methods for Ratio Cut Partitioning and Clustering," *ICCAD*, 1991, pp. 10 - 13.

[54] C. Alpert and A. Kahng, "Geometric Embeddings for Faster and Better Multi-Way Netlist Partitioning," *DAC*, 1993, pp. 743 - 748.

[55] C. Alpert and S. Z. Yao, "Spectral Partitioning: The More Eigenvectors, The Better," *DAC*, 1995, pp. 195 - 200.

[56] C. Alpert and A. Kahng, "A General Framework for Vertex Ordering, With Applications to Netlist Clustering," *ICCAD*, 1994, pp. 63 - 67.

[57] E. Lawler, K. Levitt and J. Turner, "Module Clustering to Minimize Delay in Digital Networks," *IEEE Trans. on Computers*, Jan. 1966, pp. 47 - 57.

[58] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli, "On Clustering for Minimum Delay / Area," *ICCAD*, 1991, pp. 6 - 9.

[59] R. Rajaraman and D. F. Wong, "Optimal Clustering for Delay Minimization," *DAC*, 1993, pp. 309 - 314.

[60] H. Yang and D. Wong, "Area/Pin-Constrained Circuit Clustering for Delay Minimization," *ACM Int. Workshop on FPGAs*, 1994, pp. 7.1.1 - 7.1.10.

[61] J. Cong, J. Peck, and Y. Ding, "RASP: A General Logic Synthesis System for SRAM-based FPGAs," *ACM Symp. on FPGAs*, 1996, pp. 137 - 143.

[62] D. West, *Introduction to Graph Theory,* Prentice Hall, 1996, p. 130.

[63] H. Gabow and R. Tarjan, "Faster scaling algorithms for general graph-matching problems," *Journal of the ACM*, Oct. 1991, pp. 815-853.

[64] H. Shin and C. Kim, "A Simple Yet Effective Technique for Partitioning," *IEEE Trans. on VLSI*, Sept. 1993, pp. 380 - 386.

[65] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Trans. on CAD*, Jan. 1985, pp. 92 - 98.

[66] D. Huang and A. Kahng, "Partitioning-Based Standard-Cell Global Placement with an Exact Objective," *ACM Symp. on Physical Design*, 1997, pp. 18 - 25.

[67] J. Rose, W. Snelgrove and Z. Vranesic, "ALTOR: An Automatic Standard Cell Layout Program," *Canadian Conf. on VLSI*, 1985, pp. 169 - 173.

[68] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, "Gordian: VLSI Placement by Quadratic Programming and Slicing Optimization," *IEEE Trans. on CAD*, March 1991, pp. 356 - 365.

[69] G. Sigl, K. Doll and F. Johannes, "Analytical Placement: A Linear or a Quadratic Objective Function?," *DAC*, 1991, pp. 427 - 432.

[70] C. Alpert, T. Chan, D. Huang, A. Kahng, I. Markov, P. Mulet and K. Yan, "Faster Minimization of Linear Wirelength for Global Placement," *ACM Symp. on Physical Design*, 1997, pp. 4 - 11.

[71] C. Alpert, T. Chan, D. Huang, I. Markov and K. Yan, "Quadratic Placement Revisited," *DAC*, 1997, pp. 752 - 757.

[72] A. Srinivasan, "An Algorithm for Performance-Driven Initial Placement of Small-Cell ICs," *DAC*, 1991, pp. 636 - 639.

[73] A. Srinivasan, K. Chaudhary and E. Kuh, "Ritual: A Performance Driven Placement Algorithm for Small Cell ICs," *ICCAD*, 1991, pp. 48 - 51.

[74] B. Riess and G. Ettelt, "Speed: Fast and Efficient Timing Driven Placement," *IEEE Int. Symp. on Circuits and Systems*, 1995, pp. 377 - 380.

[75] K. Doll, F. Johannes and G. Sigl, "Domino: Deterministic Placement Improvement with Hill-climbing Capabilities," *Proc. VLSI*, 1991, pp. 3b.1.1 - 3b.1.10.

[76] S. Kirkpatrick, C. Gelatt and M. Vecchi, "Optimization by Simulated Annealing," *Science*, May 13, 1983, pp. 671 - 680.

[77] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *JSSC*, April 1985, pp. 510 - 522.

[78] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package," *DAC*, 1986, pp. 432 - 439.

[79] C. Sechen and K. Lee, "An Improved Simulated Annealing Algorithm for Row-Based Placement," *ICCAD*, 1987, pp. 478 - 481.

[80] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Trans. on CAD*, March 1995, pp. 349 - 359.

[81] W. Swartz and C. Sechen, "Timing Driven Placement for Large Standard Cell Circuits," *DAC*, 1995, pp. 211 - 215.

[82] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *ICCAD*, 1986, pp. 381 - 384.

[83] J. Lam and J. Delosme, "Performance of a New Annealing Schedule," *DAC*, 1988, pp. 306 - 311.

[84] W. Swartz and C. Sechen, "New Algorithms for the Placement and Routing of Macro Cells," *ICCAD*, 1990, pp. 336 - 339.

[85] C. Ebeling, L. McMurchie, S. A. Hauck and S. Burns, "Placement and Routing Tools for the Triptych FPGA," *IEEE Trans. on VLSI*, Dec. 1995, pp. 473 - 482.

[86] S. Nag and R. Rutenbar, "Performance-Driven Simultaneous Place and Route for Island-Style FPGAs," *ICCAD*, 1995, pp. 332 - 338.

[87] S. Nag and R. Rutenbar, "Performance-Driven Simultaneous Place and Route for Row-Based FPGAs," *DAC*, 1994, pp. 301 - 307.

[88] M. Alexander, J. Cohoon, J. Ganley and G. Robins, "Performance-Oriented Placement and Routing for Field-Programmable Gate Arrays," *European Design Automation Conf.*, 1995, pp. 80 - 85.

[89] C. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling," *ICCAD*, 1994, pp. 690 - 695.

[90] M. Alexander, J. Cohoon, J. Ganley and G. Robins, "An Architecture-Independent Approach to FPGA Routing Based on Multi-Weighted Graphs," *European Design Automation Conf.*, 1994, pp. 259 - 264.

[91] Y.-L. Wu, M. Marek-Sadowska, "An Efficient Router for 2-D Field-Programmable Gate Arrays," *European Design Automation Conf.*, 1994, pp. 412 - 416.

[92] Y.-L. Wu, M. Marek-Sadowska, "Orthogonal Greedy Coupling -- A New Optimization Approach to 2-D FPGA Routing," *DAC*, 1995, pp. 568 - 573.

[93] M. J. Alexander, G. Robins, "New Performance-Driven FPGA Routing Algorithms," *DAC*, 1995, pp. 562 - 567.

[94] Y.-S. Lee, A. Wu, "A Performance and Routability Driven Router for FPGAs Considering Path Delays," *DAC*, 1995, pp. 557 - 561.

[95] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-Driven Layout and FPGA Routing," *DAC*, 1992, pp. 536 - 542.

[96] M. Placzewski, "Plane Parallel A* Maze Router and Its Application to FPGAs," *DAC*, 1992, pp. 691 - 697.

[97] J. S. Rose, "Parallel Global Routing for Standard Cells," *IEEE Trans. on CAD*, Oct. 1990, pp. 1085 - 1095.

[98] Y. Chang, S. Thakur, K. Zhu and D. Wong, "A New Global Routing Algorithm for FPGAs," *ICCAD*, 1994, pp. 356 - 361.

[99] S. Brown, J. Rose, Z. G. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Trans. on CAD*, May 1992, pp. 620 - 628.

[100] G. Lemieux, S. Brown, "A Detailed Router for Allocating Wire Segments in FPGAs," *ACM/SIGDA Physical Design Workshop*, 1993, pp. 215 - 226.

[101] G. Lemieux, S. Brown, D. Vranesic, "On Two-Step Routing for FPGAs," *ACM Symp. on Physical Design*, 1997, pp. 60 - 66.

[102] C. Y. Lee, "An Algorithm for Path Connections and its Applications," *IRE Trans. Electron. Comput.*, Vol. EC=10, 1961, pp. 346 - 365.

[103] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numer. Math.*, Vol. 1, 1959, pp. 269 - 271.

[104] R. Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Trans. on CAD*, March 1987, pp. 165-172.

[105] F. Rubin, "The Lee Path Connection Algorithm," *IEEE Trans. Computers*, Sept. 1974, pp. 907 - 914.

[106] S. Trimberger, *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[107] J. Rubinstein, P. Penfield and M. Horowitz, "Signal Delay in RC Tree Networks," *IEEE Trans. on CAD*, 1983, pp. 202 - 211.

[108] J. Cong, L. He, C. Koh and P. Madden, "Performance Optimization of VLSI Interconnect Layout," *Integration, The VLSI Journal*, Vol. 21, 1996, pp. 1 - 94.

[109] S. Prasitjutrakul and W. Kubitz, "A Timing-Driven Global Router for Custom Chip Design," *ICCAD*, 1990, pp. 48 - 51.

[110] C. Alpert, T. Hu, J. Huang and A. Kahng, "A Direct Combination of the Prim and Dijkstra Constructions for Improved Performance-Driven Global Routing," *IEEE Int. Symp. on Circuits and Systems*, 1993, pp. 1869 - 1872.

[111] K. Boese, A. Kahng and G. Robins, "High-Performance Routing Trees with Identified Critical Sinks," *DAC*, 1993, pp. 182 - 187.

[112] T. Okamoto and J. Cong, "Buffered Steiner Tree Construction with Wire Sizing for Interconnect Layout Optimization," *ICCAD*, 1996, pp. 44 - 49.

[113] J. Cong, A. Kahng, G. Robins, M. Sarrafzadeh and C. Wong, "Provably Good Performance-Driven Global Routing," *IEEE Trans. on CAD*, June 1992, pp. 739 - 752.

[114] M. Kang, W. Dai, T. Dillinger and D. LaPotin, "Delay Bounded Buffered Tree Construction for Timing Driven Floorplanning," *ICCAD*, 1997, pp. 707 - 712.

[115] J. Lillis, T. Lin and C. Ho, "New Performance Driven Routing Techniques with Explicit Area / Delay Tradeoff and Simultaneous Wire Sizing," *DAC*, 1996, pp. 395 - 400.

[116] J. Cong, K. Leung and D. Zhou, "Performance-Driven Interconnect Design Based on Distributed RC Delay Model," *DAC*, 1993, pp. 606 - 611.

[117] A. Lim, S. Cheng and C. Wu, "Performance Oriented Rectilinear Steiner Trees," *DAC*, 1993, pp. 171 - 176.

[118] X. Hong, T. Xue, E. Kuh, C. Cheng and J. Huang, "Performance-Driven Steiner Tree Algorithms for Global Routing," *DAC*, 1993, pp. 177 - 181.

[119] K. Boese, A. Kahng, B. McCoy and G. Robins, "Rectilinear Steiner Trees with Minimum Elmore Delay," *DAC*, 1994, pp. 381 - 386.

[120] A. Vittal and M. Marek-Sadowska, "Minimal Delay Interconnect Design Using Alphabetic Trees," *DAC*, 1994, pp. 392 - 396.

[121] J. Cong and C. Koh, "Interconnect Layout Optimization Under Higher-Order RLC Model," *ICCAD*, 1997, pp. 713 - 720.

[122] L. Kannan, P. Suaris and H. Fang, "A Methodology and Algorithms for Post-Placement Delay Optimization," *DAC*, 1994.

[123] J. Lillis, C. Cheng and T. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model," *ICCAD*, 1995, pp. 138 - 143.

[124] K. Sato and M. Kawarabayashi, "Post-Layout Optimization for Deep Submicron Design," *DAC*, 1996, 740 - 745.

[125] C. Alpert and A. Devgan, "Wire Segmenting for Improved Buffer Insertion," *DAC*, 1997, pp. 588 - 593.

[126] C. Chu and D. Wong, "A New Approach to Simultaneous Buffer Insertion and Wire Sizing," *ICCAD*, 1997, pp. 614 - 621.

[127] J. Swartz, V. Betz and J. Rose, "A Fast Routability-Driven Router for FPGAs," *ACM Symp. on FPGAs*, 1998, pp. 140 - 149.

[128] W. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers," *Journal of Applied Physics*, Jan. 1948, pp. 55 - 63.

[129] K. Boese, A. Kahng, B. McCoy and G. Robins, "Fidelity and Near-Optimality of Elmore-Based Routing Constructions," *ICCD*, 1993, pp. 81 - 84.

[130] J. Cong and L. He, "Optimal Wiresizing for Interconnects with Multiple Sources," *ACM Trans. on Design Automation of Electronic Systems*," Oct. 1996, pp. 478 - 511.

[131] R. Hitchcock, G. Smith and D. Cheng, "Timing Analysis of Computer-Hardware," *IBM Journal of Research and Development*, Jan. 1983, pp. 100 - 105.

[132] P. Hauge, R. Nair and E. Yoffa, "Circuit Placement for Predictable Performance," *ICCAD*, 1987, pp. 88 - 91.

[133] H. Youssef and E. Shragowitz, "Timing Constraints for Correct Performance," *ICCAD*, 1990, pp. 24 - 27.

[134] V. Betz, "VPack and VPR User's Manual," 1998. (*Available for download from http://www.eecg.toronto.edu/~jayar/software/software.html).*

[135] M. Khalid and J. Rose, "The Effect of Fixed I/O Pin Positioning on The Routability and Speed of FPGAs," *Canadian Workshop on Field-Programmable Devices,* 1995, pp. 94 - 102.

[136] S. Yang, "Logic Synthesis and Optimization Benchmarks, Version 3.0," *Tech. Report,* Microelectronics Center of North Carolina, 1991.

[137] H. Hseih, et al, "Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays," *CICC,* 1990, pp. 31.2.1 - 31.27.

[138] P. Leventis, *Undergraduate Thesis,* University of Toronto, *In Preparation.*

[139] Xilinx Inc., "XC4000E and XC4000X Series Field-Programmable Gate Arrays," *Data Sheet,* 1997.

[140] J. Swartz, "A High-Speed Timing-Aware Router for FPGAs," *M.A.Sc. Thesis,* University of Toronto, 1998.

[141] B. K. Britton et al., "Second Generation ORCA Architecture Utilizing 0.5 μm Process Enhances the Speed and Usable Gate Capacity of FPGAs," *IEEE Int. ASIC Conf.*, Sept. 1994, pp. 474 - 478.

[142] E. M. Sentovich et al, "SIS: A System for Sequential Circuit Analysis," *Tech. Report No. UCB/ERL M92/41*, University of California, Berkeley, 1992.

[143] V. Betz and J. Rose, "On Biased and Non-uniform Global Routing Architectures and CAD Tools for FPGAs," Technical Report, University of Toronto, 1996.

[144] M. Feuer, "Connectivity of Random Logic," *IEEE Trans. on Computers*, Jan. 1982, pp. 29 - 33.

[145] C. Matsumoto, "Million-Gate Architecture Will Vie with ASIC-like Approach from Altera, Lucent, and Gatefield," *Electronic Engineering Times*, Oct. 26, 1998, p. 1.

[146] P. Clarke, "Dynachip Claims Speed Breakthrough in its FPGAs," *Electronic Engineering Times*, June 9, 1997, p. 10.

[147] M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and Parameterized Random Generation of Combinational Benchmark Circuits," *to appear in IEEE Trans. on CAD.*

[148] M. Hutton, J. Rose, D. Corneil, "Generation of Synthetic Sequential Benchmark Circuits," *ACM Symp. on FPGAs*, 1997, pp. 149 - 155.

[149] M. Hutton, J.P. Grossman, J. Rose, D. Corneil, "Characterization and Parameterized Random Generation of Digital Circuits," *DAC*, 1996, pp. 94 - 99.

[150] J. Anderson and S. Brown, "An LPGA with Foldable PLA-Style Logic Blocks," *ACM Symp. on FPGAs*, 1998, pp. 244 - 252.

[151] J. Swartz, V. Betz and J. Rose, "A Routability-Driven and Timing-Aware Router for FPGAs," *Submitted to IEEE Trans. on CAD.*

[152] R. Tessier, "Negotiated A* Routing for FPGAs," *Canadian Workshop on Field-Programmable Devices*, 1998, pp 14 - 19.

[153] Y. Sankar and J. Rose, "Ultra-Fast Placement for FPGAs," *To appear in 1999 ACM Symp. on FPGAs.*

[154] Canadian Microelectronics Corporation, "0.35 μm Mixed-Mode Polycide HSPICE Models," *Confidential Process Documentation*, 1997.

[155] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, Second Edition*, Addison-Wesley, 1993.

[156] D. Lewis, *VLSI Arithmetic Structures, Course Notes*, University of Toronto, 1993.

Index


9 780792 384601