Multi-FPGA Systems

by

Scott Hauck

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1995

Approved by ____

Co-Chairperson of Supervisory Committee

____

Co-Chairperson of Supervisory Committee

Program Authorized
to Offer Degree ____

Date ____

University of Washington

Abstract

Multi-FPGA Systems

by Scott Hauck

Chairperson of the Supervisory Committee: Associate Professor Gaetano Borriello
Department of Computer Science and Engineering

Associate Professor Carl Ebeling
Department of Computer Science and Engineering

Multi-FPGA systems are a growing area of research. They offer the potential to deliver high performance solutions to general computing tasks, especially for the prototyping of digital logic. However, to realize this potential requires a flexible, powerful hardware substrate and a complete, high quality and high performance automatic mapping system.

The primary goal of this thesis is to offer a disciplined look at the issues and requirements of multi-FPGA systems. This includes an in-depth study of some of the hardware and software issues of multi-FPGA systems, especially logic partitioning and mesh routing topologies, as well as investigations into problems that have largely been ignored, including pin assignment and architectural support for logic emulator interfaces. We also present Springbok, a novel rapid-prototyping system for board-level designs.

Table of Contents

List of Figures

List of Tables

List of Equations

Acknowledgments

A thesis is as much a product of the environment it was created in, and the support it received, as it is the work of a graduate student. This thesis is no exception. Although there have been many hands helping me in my work (and I apologize to those I have not recognized here), some people stand out in my mind.

In developing the pin assignment software, numerous people generously gave me detailed descriptions of their systems, and shared example mappings, making this work possible. These include Patrice Bertin, Duncan Buell, David Lewis, Daniel Lopresti, Brian Schott, Mark Shand, Russ Tessier, Herve Touati, Daniel Wong, and Kazuhisa Yamada.

When my partitioning work was in its infancy, Andrew B. Kahng gave me many helpful pointers into the literature, which helped crystallize my thoughts. He and Lars Hagen were kind enough to give me access to their spectral partitioning codes, helping to extend my work. Finally, D. F. Wong, and Honghua Yang also aided in this effort.

The routing topologies work was significantly improved by pointers from Michael Butts at Quickturn. He has been a consistent source of advice and good ideas, helping in many other places in this thesis. There have been several others in the multi-FPGA system community that have encouraged my work, and offered valuable criticism. These include members of Anant Agarwal's Virtual Wires group at MIT (especially Jonathan Babb and Russ Tessier), as well as Pak Chan and Martine Schlag at UC - Santa Cruz.

At the University of Washington, the graduate students and staff have been a great resource in my work. Donald Chinn provided some valuable insights into the lower bounds of the routing topologies work. The CAD group at U.W., including Pai Chou, Darren Cronquist, Soha Hassoun, Henrik Hulgaard, Neil McKenzie, and Ross Ortega, has also helped in many ways, not the least of which being a willing set of guinea pigs to innumerable practice talks (or was it just to get the free food?). Larry McMurchie ranks high in this list as well. Also, his providing his routing software to me, as well as helping me customize it to my problem, greatly helped the mesh routing topologies work.

Undoubtedly the biggest help from my fellow students came from Elizabeth Walkup. Countless times I ran to her with my graph theory problems, wild ideas, and minor complaints, and while we may not have solved them all, I always walked away better than I arrived. Her hand can be found in numerous places throughout this thesis.

Financial support is obviously critical to any prolonged research endeavor. This work was funded by both the Defense Advanced Research Projects Agency under Contract N00014-J-91-4041, as well as an AT&T Fellowship.

Finally, I am deeply indebted to my faculty advisors, both formal and informal, for their help throughout this process. Larry Snyder helped me in the beginning, and got me started on research. Ted Kehl has always been a support, even when he has been cutting me down (not that I didn't usually deserve it). Steve Burns, Carl Ebeling, and Gaetano Borriello have gotten me where I am now, both in pushing me in new directions and to new opportunities, as well as stepping back and letting me chart my own course. Without these people this thesis would never have been written. This is particularly true of Carl, who's offhand comments started this whole endeavor, and Gaetano, whose breakthrough comments at critical times radically improved much of this work.

I thank all the people who helped me in this work, and apologize to those I may have forgotten here.

Dedication

To my parents for giving me the tools, to Gaetano, Carl, and Steve for giving me the freedom, and to my wife Susan for giving me the support to complete this thesis.

Chapter 1. General Introduction

In the mid 1980s a new technology for implementing digital logic was introduced, the field-programmable gate array (FPGA). These devices could either be viewed as small, slow gate arrays (MPGAs) or large, expensive programmable logic devices (PLDs). FPGAs were capable of implementing significantly more logic than PLDs, especially because they could implement multi-level logic, while most PLDs were optimized for two-level logic. While they did not have the capacity of MPGAs, they also did not have to be custom fabricated, greatly lowering the costs for low-volume parts, and avoiding long fabrication delays. While many of the FPGAs were configured by static RAM cells in the array (SRAM), this was generally viewed as a liability by potential customers who worried over the chip's volatility. Antifuse-based FPGAs also were developed, and for many applications were much more attractive, both because they tended to be smaller and faster due to less programming overhead, and also because there was no volatility to the configuration.

In the late 1980's and early 1990's there was a growing realization that the volatility of SRAM-based FPGAs was not a liability, but was in fact the key to many new types of applications. Since the programming of such an FPGA could be changed by a completely electrical process, much as a standard processor can be configured to run many programs, SRAM-based FPGAs have become the workhorse of many new reprogrammable applications. Some uses of reprogrammability are simple extensions of the standard logic implementation tasks for which the FPGAs were originally designed. An FPGA plus several different configurations stored in ROM could be used for multi-mode hardware, with the functionality on the chip changed in reaction to the current demands. Also, boards constructed purely from FPGAs, microcontrollers, and other reprogrammable parts could be truly generic hardware, allowing a single board to be reprogrammed to serve many different applications.

Some of the most exciting new uses of FPGAs move beyond the implementation of digital logic, and instead harness large numbers of FPGAs as a general-purpose computation medium. The circuit mapped onto the FPGAs need not be standard hardware equations, but can even be operations from algorithms and general computations. While these FPGA-based custom-computing machines may not challenge the performance of microprocessors for many applications, for computations of the right form an FPGA-based machine can offer extremely high performance, surpassing any other programmable solution. While a custom hardware implementation will be able to beat the power of any generic programmable system, and thus there must always be a faster solution than a multi-FPGA system, the fact is that few applications will ever merit the expense of creating application-specific solutions. An FPGA-based computing machine, which can be reprogrammed like a standard workstation, offers the highest realizable performance for many different applications. In a sense it is a hardware Supercomputer, surpassing traditional machine architectures for certain applications. This potential has been realized by many different research machines. The Splash system [Gokhale90] has provided performance on genetic string matching that is almost 200 times greater than all other Supercomputer implementations. The DECPeRLe-1 system [Vuillemin95] has demonstrated world-record performance for many other applications, including RSA cryptography.

One of the applications of multi-FPGA systems with the greatest potential is logic emulation. The designers of a custom chip need to verify that the circuit they have designed actually behaves as desired. Software simulation and prototyping have been the traditional solution to this problem. However, as chip designs become more complex, software simulation is only able to test an ever decreasing portion of the chip's functionality, and it is quite expensive in time and money to debug by repeated prototype fabrications. The solution is logic emulation, the mapping of the circuit under test onto a multi-FPGA system. Since the logic is implemented in the FPGAs in the system, the emulation can run at near real-time, yielding test cycles several orders of magnitude faster than software simulation, yet with none of the time delays and inflexibility of prototype fabrications. These benefits have led many of the advanced microprocessor manufacturers to include logic emulation in their validation methodologies.

While multi-FPGA systems have great potential to provide high-performance solutions to many applications, there are several problems that hold back current systems from achieving their full promise. Their hardware structures tend to have much too few I/O connections for their logic capacity to be adequately used. This leads to very low logic utilization, with only 10% to 20% of the available capacity usable. While some of these problems may be unavoidable, since we are using FPGAs targeted to single-chip applications to build multi-chip systems, the routing topologies of multi-FPGA systems need to be improved to compensate for this. Also, while there are logic emulation systems that meet the needs of single-chip ASIC prototyping, the domain of system-level or board-level prototyping has largely been ignored. These emulation opportunities are quite different from single chip systems, since multi-chip designs have a heavy reliance on premade components, and have resource demands that vary much more widely than that of a single ASIC. Not only does board-level prototyping demand a flexible and extensible emulation hardware system, one that allows the inclusion of arbitrary premade components, but it also brings into sharper focus the need to support external interfaces. Even in single-chip systems, one of the major benefits of an emulation is that it may be capable of being put into its target environment, thus encountering the true input-output demands of the final implementation. However, an emulation of a circuit will almost always run slower than the final implementation, meaning that it will no longer meet the timing requirements of the external signaling protocols. Current emulation systems have no answer to this problem, forcing the user to develop ad hoc solutions. For board-level prototyping, where a system will be required to meet several signaling protocols at once, this problem becomes an even greater concern. Thus, a general-purpose solution to the logic emulator interface problem is critical to the success of board-level prototyping, and can improve emulations of individual ASICs as well.

The problems with current multi-FPGA systems are not solely with the hardware. Automatic mapping software is an important part of any multi-FPGA system that hopes to achieve widespread utility, and is an absolute requirement for any successful logic emulation system. While there are niches for multi-FPGA systems with hand-mapped solutions, very few people are capable and willing to expend the effort to hand-map to a multi-FPGA system, and without software these systems will only realize a small portion of their promise. Unfortunately, today's mapping software is inadequate for most potential applications. They deliver quite poor mappings, and can take huge amounts of time to complete. Creating a mapping with current tools can take almost an entire day to finish. Obviously, this is a tremendous disadvantage, since any time savings that could be achieved by running an algorithm on a multi-FPGA system is swallowed up by the time to create the mapping in the first place. Thus, high-quality, high-performance mapping tools are a necessity for realizing the full potential of multi-FPGA systems.

Part of the reason why these deficiencies exist in current multi-FPGA systems is that there has been relatively little work done on optimizing the hardware and software structures of these machines. While there are a huge number of multi-FPGA systems proposed and constructed, there are almost no studies investigating what the best architectures really are. The software systems suffer as well, with many individual optimizations proposed, but no real understanding of how to bring these optimizations together to achieve the best possible results. Thus, today's systems are largely grab-bags of appealing features with little knowledge of what the systems should really look like.

In 1992 we were in the same position as many other potential multi-FPGA system builders. We saw the need for a new system, which in our case was an emulation system optimized for the demands of board-level prototyping. While we knew in general what the system should look like, there was little guidance on exactly what the hardware and software architecture should be. At this point, instead of blindly going forward with our best guesses on how the system should be, we set out to look at the underlying issues and determine what really were the right architectures. By performing a disciplined investigation of multi-FPGA hardware and software systems, we not only would learn how to build our prototyping system correctly, but would also give a better understanding of some of the issues to the multi-FPGA community at large. It is these investigations that form the heart of this thesis. While we have not yet built our prototyping system (the high-level design of which is contained in Chapter 6), our work has yielded insight into many of the problems facing today's multi-FPGA systems.

Our investigations have focused on several different areas. In the hardware arena, we examined the routing topologies of multi-FPGA systems. Although the FPGAs in a multi-FPGA system are reprogrammable, the connections between the FPGAs are fixed by the circuit board implementing the system. With the main bottleneck in current multi-FPGA systems being the limited bandwidth for inter-FPGA routing, the routing topology of a multi-FPGA system has a significant impact on the capacity, as well as performance, of a multi-FPGA system. In our investigations we examined how best to construct a multi-FPGA system, yielding topologies with significantly lower I/O requirements, faster performance, and higher bandwidth.

Since our goal was to construct a prototyping system for board-level designs, we also had to consider the issue of how to support the external interfaces of a logic emulation system. By examining how to solve this problem in general, we have shown that a generic interface transducer, a simple board with FPGAs and memories for filtering the incoming and outgoing datastreams, is capable of supporting many different types of protocols. It has a flexible architecture that enables it to be prefabricated, yet it still allows customization to a specific application's requirements. Also, the required customization is easy to develop, and a small amount of programming in a high-level hardware description language is usually sufficient.

We have also spent time investigating the software required to map to a multi-FPGA system. Perhaps the most critical of these algorithms is partitioning, the process of splitting an input mapping into pieces small enough to fit into the individual FPGAs in a multi-FPGA system. While much work has been done on this problem previously, there was little insight into the best possible approach. Many researchers have proposed software solutions, but there was little comparative assessment of different approaches. Thus, one major portion of our work was to sift through the different approaches, and develop an algorithm that brings together the best possible set of optimizations to achieve the best overall results. This produced a survey of many of the different approaches, with each optimization implemented in a common framework, and compared quantitatively on a set of benchmark circuits. We now have an algorithm that produces results significantly better and faster than any other published approach.

We also recognized that flexible routing topologies are becoming more common in multi-FPGA systems, with architectures capable of significant customization in their interconnection patterns being proposed. To deal with these systems, it became apparent that the mapping tools not only needed to be fast and high-quality, but they also must be able to adapt to an arbitrary topology. This is not only important for flexible multi-FPGA hardware, but it also results in software that can easily be adapted to different machines. In pursuit of this goal, we have developed partitioning software capable of mapping to an arbitrary multi-

FPGA system topology, as well as routing software that can do the same. This routing software has the added advantage that it improves the quality of the mapping over current approaches, yielding both better-quality mappings and faster run-times. This latter benefit is due to the fact that poor routing complicates the following mapping steps, and thus by improving the routing quality we decrease the total mapping time.

The result of all of this work is new insights into how best to build and use multi-FPGA systems, as well as solutions to some of the outstanding problems. These results are not just important for building prototyping systems like the one we envisioned, but to multi-FPGA systems in general. The routing topologies are generally useful, offering improved topologies for both emulation and custom-computing. The mapping tools provide fast, high-quality mapping solutions for arbitrary system topologies. Finally, the interface transducers deliver a generic solution for interfacing both current single-chip emulators, as well as future board-level prototyping systems.

This thesis details the results of these investigations. It begins with a discussion of the underlying technologies that make multi-FPGA systems practical, including a survey of FPGAs and other logic implementation technologies in Chapter 2, as well as the application domains of multi-FPGA systems in Chapter 3. Chapter 4 delves deeper into one of the most promising application areas for multi-FPGA systems, logic emulation. Chapter 5 then gives an overview of current multi-FPGA system hardware. With the background provided by these chapters, we then present some new investigations and approaches to multi-FPGA system hardware. In Chapter 6 we present a new multi-FPGA system optimized for the prototyping of board-level designs. This system combines the inclusion of arbitrary devices with a flexible and extensible architecture to provide an extremely adaptable prototyping solution. Then, Chapter 7 examines routing topologies for multi-FPGA systems. This yields constructs with lower I/O demands, smaller delays, and higher bandwidth than standard topologies. Finally, Chapter 8 investigates how to support external interfaces for logic emulators, one of the significant problems with current systems.

The discussion of multi-FPGA system software begins with a general introduction in Chapter 9. Chapter 10 considers logic partitioning for multi-FPGA systems, presenting a comprehensive investigation into bipartitioning techniques. This results in a combined partitioning algorithm that achieves results better than the current state of the art, in significantly less time. Chapter 11 presents an iterative partitioning algorithm that automatically adapts to an arbitrary topology. This allows bipartitioning to be applied to any multi-FPGA system, as well as multi-sectioning and multi-way partitioning algorithms where appropriate. Finally, Chapter 12 investigates pin assignment algorithms for multi-FPGA systems. This speeds up the placement and routing time for multi-FPGA systems and produces higher quality results. We conclude this thesis with some overall results and potential future work in Chapter 13.

Chapter 2. Circuit Implementation Alternatives and Technologies

Overview of chip design styles

Circuit designers have a range of methods for implementing digital logic, with widely differing capabilities and costs. In the following sections we will discuss many of these different alternatives. This will serve both to explain what FPGAs are, and also to show how FPGAs enable new opportunities that were not possible with other technologies. Note that throughout this chapter, the pictures are meant merely to illustrate the concepts discussed. Many (less important) details are omitted, and the quantities of resources are often reduced to aid visibility.

The different methods of implementing digital logic can have significantly different capacity, performance, and costs (both in time and money), as well as differing levels of automation of the design process. Full custom designs offer the highest performance and capacity possible. However, they also take the longest time to be fabricated, have little or no automation in the design process, and cost the most money, in both per-chip and NRE (non-recurring engineering) costs. NRE's are initial costs to set up the production process, and thus are paid once for the first chip produced. Other design styles sacrifice some of the performance and capacity of full-custom design in order to lower costs, speed up the design process, and reduce the fabrication time. Many such technologies will be covered in the following sections.

Full custom

In full custom design, the designer specifies all the details of how an integrated circuit will be fabricated. This gives the designer the maximum flexibility, yielding the highest possible performance and gate capacity. Thus, for some applications, this is the only possible method of implementing the desired functionality, since other technologies may not accommodate the application's capacity or performance demands. However, this power and flexibility does come at a cost. Since the circuit will be custom fabricated, sharing no features with other designs, the fabrication will be costly, and will take a significant amount of time. Other technologies will share some features from design to design, in general yielding lower costs due to economies of scale, and faster turnaround due to prefabrication of shared features.

The designer of a full custom implementation specifies exactly where the different features of the circuit will be by specifying where the various materials will be placed on the chip. That is, the designer decides where diffusion, polysilicon, and metal will be applied to create the desired functionality. There are some design rules that restrict the allowed circuit designs. However, these rules are local constraints on circuit geometries required to avoid structures that cannot be fabricated. For example, wires must be a certain width, and must be spaced apart by a certain distance, in order to avoid breaks or short circuits in signal lines. However, the designer still has a considerable amount of flexibility in deciding how to design the circuit.

Because of the high degree of flexibility in full custom circuits, the process of designing such a circuit is quite complex and time-consuming. Since the designer must decide where each circuit feature will be placed, there are a huge number of decisions to be made. Thus, it takes a large amount of time and effort to develop a full custom design, time beyond what is required simply to fabricate the circuit. Since each fabrication of the circuit is quite costly, the designer will also expend a significant amount of time to make sure the design is correct before fabrication. While software can be used to simplify some of this process, most designers of full custom circuits are using this implementation technology specifically to achieve the highest quality implementation possible, and thus avoid inefficiency due to software designed circuits. Thus, software support for full-custom design is in general restricted to validation of the circuit, and actual design of only non-critical subcircuits, places where some inefficiency can be tolerated.

The process of fabricating the design is done in multiple steps, building up the layers necessary to implement the circuit (Figure 1). Early stages make wells inside which the transistors will be placed. Next, diffusion and polysilicon layers will be added to build the actual transistors. Finally, metal layers will be created to route signals between the different transistors. Note that many processes have multiple metal layers, so steps **h**-**j** will be repeated for each metal layer.

Each of these layers is created by a photolithographic process (Figure 2) that patterns the layer of material. Photoresist, an acid resistant substance that changes properties when exposed to a light source, is applied on top of the layer material to be patterned. The photoresist is then exposed to a light source shone through a mask, so that the mask's image selectively alters the photoresist. Unaltered portions of the photoresist are removed, and the underlying layer is etched by an acid. Since the areas still coated with photoresist will not be affected by the acid, the layer is selectively etched in the pattern described by the mask.

Because all layers of the design of a full custom circuit are specified by the designer, there is no sharing of masks or other layer processing between different designs. Thus, full custom designs are in general the most expensive and time-consuming, since there are little economies of scale. Many other design styles will standardize some or all processing steps, speeding up the processing, and lowering costs.

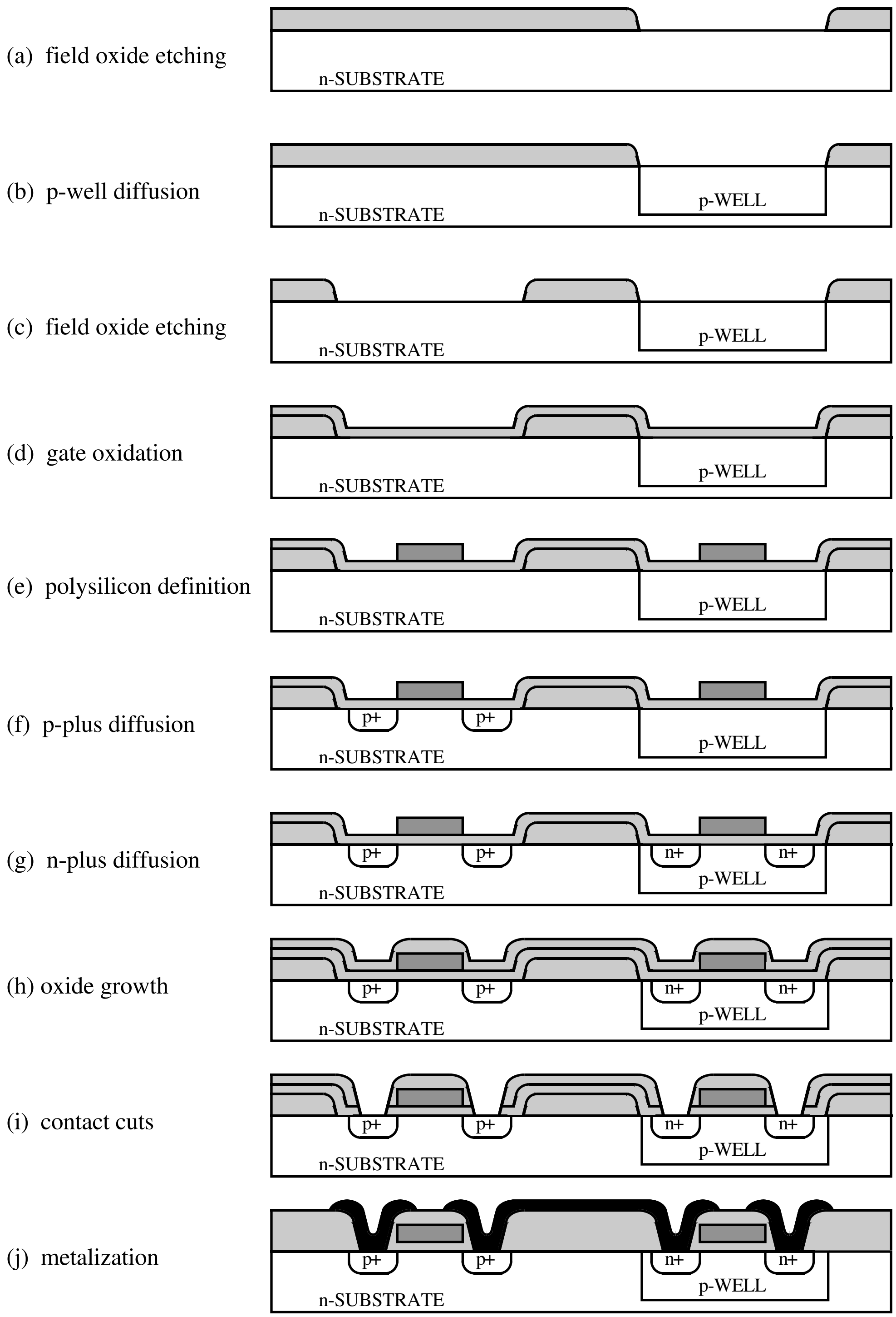


Figure 1. Steps in the fabrication of a full custom design (p-well CMOS) [Weste85].

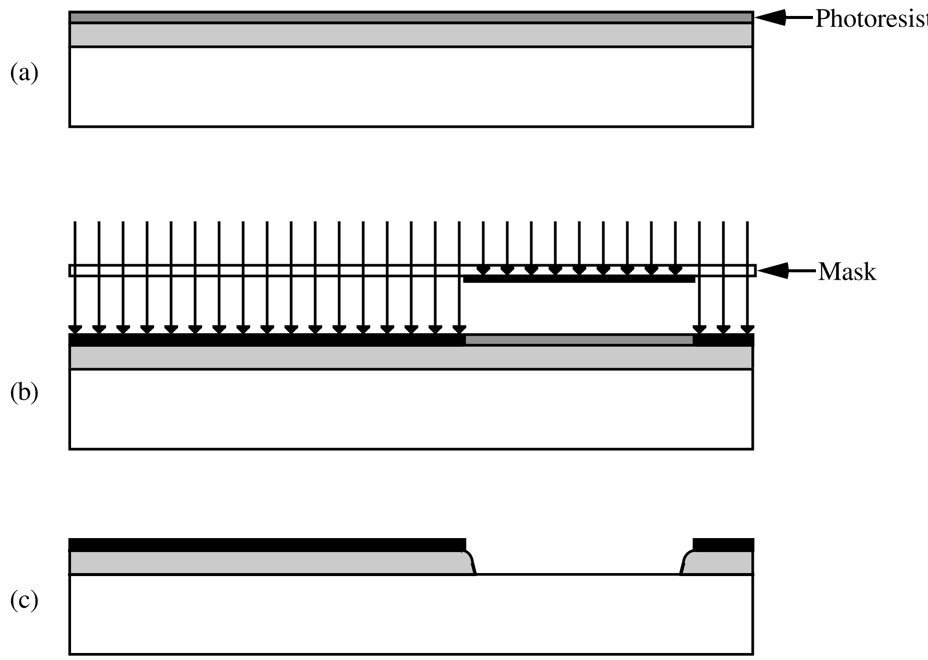


Figure 2. Steps in patterning a layer [Weste85]. (a) Photoresist covers the layer to be patterned. (b) A light source is shown through a mask, selectively polymerizing the photoresist. (c) The unpolymerized photoresist is removed, and the wafer is exposed to an acid, which etches only the areas not covered by photoresist.

Standard cells

Creating a complete full custom design takes a lot of time and effort, since all the details of the chip creation must be specified. However, in many cases the complete flexibility of the full custom design process are not needed for a given circuit. In these cases, it is advantageous to restrict some of the design flexibility to speed up the design process. This is exactly the approach taken in standard cell design. A standard cell design restricts the allowed circuit geometries and dictates a specific layout topology. By doing so, software tools can be designed to efficiently automate the physical design process, greatly speeding up the design process. However, since the layout is more restrictive, this approach delivers lower capacity and performance than a full custom design.

In a standard cell design, all gates are predefined in a library. This library consists of a set of standard cells that implement all the different gate structures necessary to map a circuit, as well as possibly some larger, more complex gate structures included for better mapping efficiency. All of these cells have the same height, the same placement of power and ground lines (which run horizontally across the cell), and have input and output terminals at the top and/or bottom. In this way cells can be abutted horizontally, creating a row of cells with uniform height, and with power and ground lines connecting together. Cells can be of arbitrary widths, so that cells with widely differing functionality (from a simple inverter to a multiple gate cell) can all be created efficiently. A standard cell chip is laid out in interleaved routing and logic rows (Figure 3). The logic rows are all of uniform height, while the routing rows are as high as necessary to accommodate the required routing. While horizontal routing is handled in these routing rows (or "channels"), vertical routing is either accomplished in higher metal layers, or by the inclusion of route-through cells which connect signals between adjacent routing channels.

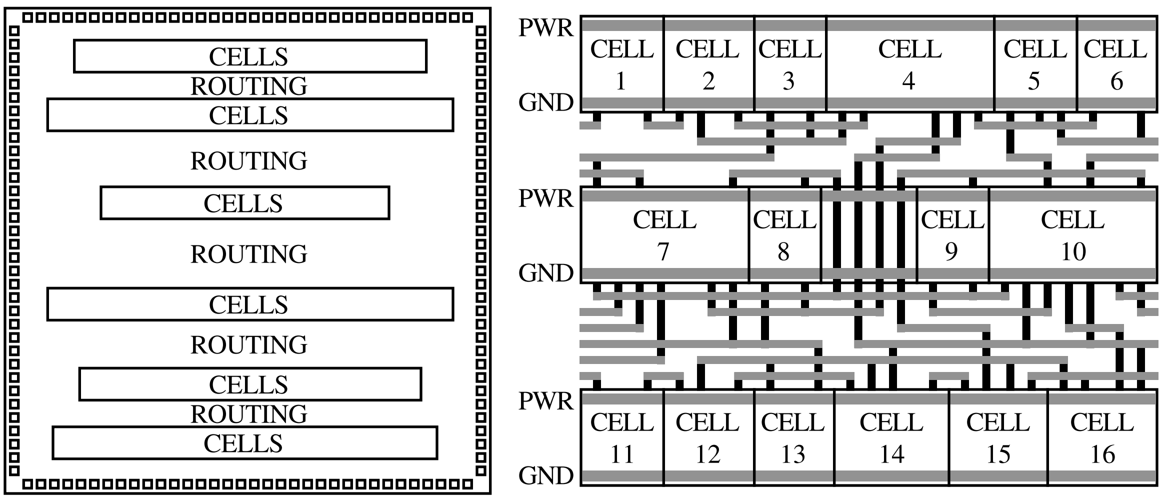


Figure 3. A standard cell chip layout (left), and a detail of a portion of the array (right), including a route-through between **CELL 8** and **CELL 9**.

With the layout of the gates handled by a predefined library, the process of creating a standard cell layout involves translating the circuit to be implemented into a collection of standard cell functions (technology mapping), placing these cells to achieve the required performance with the minimum routing channel widths (placement), and connecting the terminals of the standard cells together in the proper pattern (routing). Because of the regularity of the standard cell layout, each of these problems can be handled automatically via software. Thus, a logic designer need only specify the desired circuit functionality, and the software will automatically create a layout. This layout will then be fabricated at a silicon foundry exactly the same way as a full custom design.

Since standard cells and full custom designs require exactly the same fabrication processes, the fabrication time and costs are similar. The big differences between the two styles are the time it takes to ready a design for fabrication, and the resulting size and performance of the design. In a full custom design, the designer can optimize all aspects of the layout, and achieve the highest performance and capacity circuit. However, it takes a significant amount of time to create such a design. A standard cell design can be created much more quickly, but the capacity and performance will suffer. As a result, full custom design is used primarily where the capacity and performance penalties of standard cells cannot be tolerated, while other circuits will use standard cells.

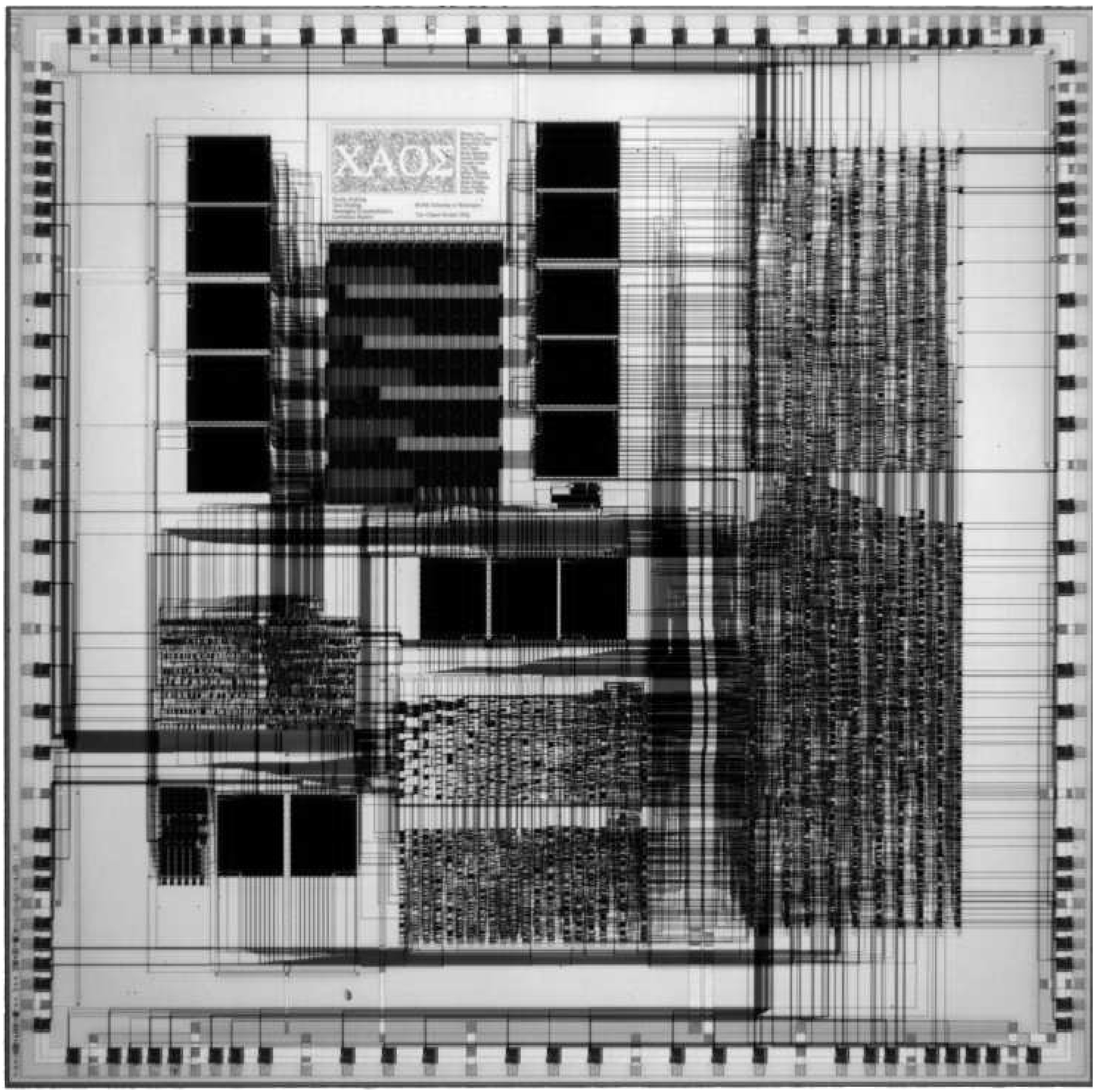


Figure 4. The Chaos router chip [Bolding93], which includes full-custom (the dark black regions) and standard cell (the regions of scattered black lines) elements.

The decision of what style to use can occur not only at the chip level, with an entire design created either as a full custom or standard cell layout, but also at the logic level, with the two styles combined within a single chip. For example, a multiprocessor routing chip may require the highest performance possible in the datapath, while the control circuitry can be designed somewhat less optimally. Thus, the design will include a full-custom datapath, while the control logic will be generated with standard cells (Figure 4). This tradeoff works especially well in this case, since the datapath will have a large amount of regularity, and will thus be somewhat simpler to design by hand, while the potentially complex and irregular control circuitry can be automatically created. This gives much of the benefits of both full custom and standard cells in a single design.

Another possibility in a standard cell is to include megacells or cores. These are complex, predesigned circuit elements that can be inserted in a design. For example, there are microcontrollers, DSPs, multipliers and other mathematical functions, standard bus interfaces, and many other functions available as megacells or cores [Arnold94]. These elements can be included in a standard cell design, providing complex, compact, high performance functionality without the need to custom design these parts. They will be significantly larger than standard cells, so some provision for including these elements is required. However, current software can in general automatically include these elements in a standard cell design.

Mask-programmable gate arrays

While standard cells simplify the design process over full custom designs, they do nothing about the time and cost involved in the actual fabrication of the chips. Although standard cell designs restrict the topologies and logic structures available to the designer, a standard cell design is still fabricated from scratch. Mask-programmable gate arrays (MPGAs) take the logic standardization of standard cells a step further. They speed up the fabrication process and lower costs, while retaining the ability of standard cells to harness automatic mapping software.

In an MPGA [Hollis87] much of the fabrication of the chip is carried out ahead of time. An MPGA has a standardized logic structure, with the layout of all transistors in the system fixed and predefined. That is, a given type of MPGA has a specific logic structure dictated to its users. For example, the MPGA might consist of simply a sea of individual transistors covering the chip. Others may have sets of four, six, or more transistors partially interconnected in a pattern optimized for mapping certain types of gates. To customize the MPGA in order to implement a given functionality, the designers are able to interconnect these logic elements in any manner they desire. Thus, one design can connect four transistors to power, ground, and each other in the proper manner to implement a 2-input NAND gate, while another design uses these transistors as a portion of a complex latch, or whatever else is desired.

Because the transistor layouts are fixed for a given MPGA, the foundry can stockpile partially fabricated chips, with all but the metal layers already created. A designer then specifies the interconnection pattern for the routing layers, and the fabrication of the chips is completed accordingly. In this way, most of the processing steps are already completed ahead of time, with the masks shared across all the designs that use a given MPGA. This decreases both processing time, since much of the processing has already been completed, as well as amortizing the cost, since much of the processing is shared across numerous designs. The designers still have a great deal of design flexibility, since they can specify an arbitrary interconnection pattern for the logic elements in the system, as long as there is enough routing area to accommodate the routing demands.

There are some limitations to MPGAs. Since the layout of the logic in the MPGA is predetermined, it will not be optimal for a given design. Some designs may require more or less of a given resource than the MPGA is designed for, leading to inefficiencies. Also, since the locations are predetermined, the routing may be more complex than in either a standard cell or full custom design. This is especially true in MPGAs that predefine the routing channels on the chip, since these routing channels will inevitably be either too large or too small for a given design. Thus, either routing resources are wasted because the channels are too large, or logic resources are sacrificed since the routing necessary to use them is not available. Some MPGAs are designed as Sea-of-Gates. In a Sea-of-Gates design, the entire chip area has logic fabricated on it. To handle routing, some of these logic functions are ignored, and arbitrary routing is performed over the top of the logic functions. In this way, the restrictions on routing channel size are removed, yielding greater flexibility in the design. However, even with a Sea-of-Gates design, there are still numerous inefficiencies due to fixed logic placement and resource mixes, yielding lower performance and logic capacities than either standard cell or full custom designs.

Programmable logic devices

Mask-programmable gate arrays reduce some of the fabrication time by prefabricating the logic on the chips. However, customizing an MPGA still requires processing at a silicon foundry, with the associated time lag and mask costs. Programmable logic devices (PLDs) in general avoid these costs by prefabricating the entire chip ahead of time. In order to customize the design, the PLD has configuration points scattered throughout the system. These configuration points are controlled by static memory cells, antifuses, or other devices that can be programmed after the chip has been fabricated. Thus, to customize the chip for a given design, the user programs these elements, which causes the chip to implement the desired functionality. The amount and locations of these configuration points will be optimized for a given PLD. Large amounts of configurability yield a much more flexible part, while less configurability yield higher capacity and performance (since the configuration points and their programming elements consume chip area and add delay to the signals connected to them).

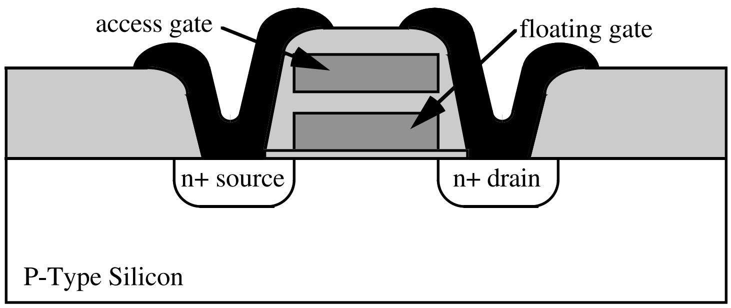


Figure 5. Floating gate structure for EPROM/EEPROM. The floating gate is completely isolated. An unprogrammed transistor, with no charge on the floating gate, operates the same as a normal n-transistor, with the access gate as the transistor's gate. To program the transistor, a high voltage on the access gate plus a lower voltage on the drain accelerates electrons from the source fast enough to travel across the gate oxide insulator to the floating gate. This negative charge then prevents the access gate from closing the source-drain connection during normal operation. To erase, EPROM uses UV light to accelerate electrons off the floating gate, while EEPROM removes electrons by a technique similar to programming, but with the opposite polarity on the access gate ([Altera93] pp. 467-471, [Wakerly94] pp. 399-400).

A simple example of this type of device is a memory. A memory is a device with N address signals, M outputs, and $M*2^N$ configuration points. The configuration points can take on the value 0 or 1. The address lines select one of 2^N sets of M configuration bits, and the outputs are set to the value of the selected M bits. Thus, the memory can be used to implement any M functions of N inputs, since each combination of these N inputs can produce a unique output value on each of the M outputs. There are a significant number of different configuration technologies ([Brown92a] pp. 2-3). Mask-programmable Read-Only Memories (ROM) configure their logic like MPGAs, with the configuration fixed during manufacture at a silicon foundry. Field-programmable versions also exist, with the programming accomplished without custom fabrication. Thus, users can easily customize field-programmable chips at their workbenches. Some Programmable Read-Only Memories (PROMs) contain fuses that can be blown by selectively applying a high voltage, permanently configuring the PROM. Erasable PROMs (EPROM) allow the configuration to be changed by exposure to a UV light source, while Electrically Erasable PROMs (EEPROM) allow the chip to be reprogrammed via a completely electrical process (Figure 5). Finally, Random-Access Memories (RAM) allow the configuration to be changed on the fly during circuit operation, allowing read-write access to the chip, while ROMs are read-only. Thus, configuration technology enables a large variety of methods for setting the memory, from custom fabrication to one-time-programmable fuses to the complete read-write flexibility of a RAM. Other PLDs (as well as FPGAs, discussed later) will offer most or all of these same programming technologies to configure their logic.

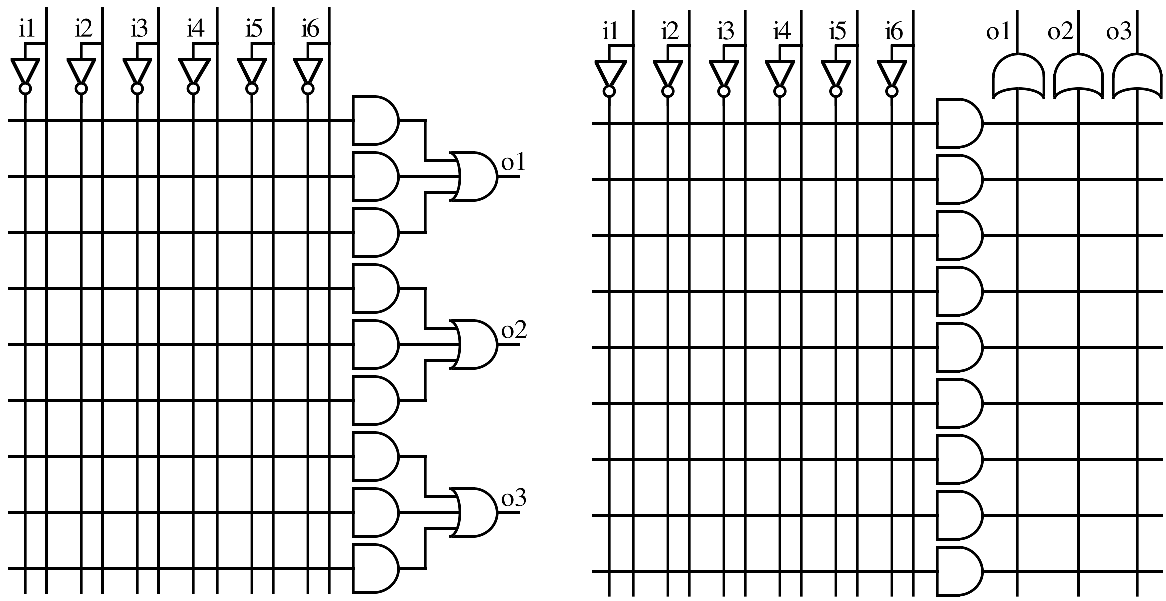


Figure 6. PAL (left) and PLA (right) structures [Biehl93].

While memories are able to implement logic functions, they are in general too inefficient for most applications. The problem is that most logic functions do not need the complete flexibility of the full memory structure, and thus using a memory to implement these functions is quite inefficient (the memory must double in size with each additional input). A better alternative is the Programmable Array Logic (PAL), a device optimized for implementing two-level sum-of-products logic structures (Figure 6 left). The device consists of a set of arbitrary product terms (the AND gates) leading to OR gates that produce the chip's outputs. As shown in Figure 7 , these product terms are logic structures that can be programmed to implement an AND of any or all of its inputs. That is, the output is the NOR of all inputs whose programmable connection is enabled, while the function is insensitive to inputs whose programmable connections are disabled. By inverting the signals coming into this function, the NOR function becomes an AND. Thus, the product term can become the AND of A and $\overline{B}$ by making connections **P2** and **P3**, and disabling all other connections, while connections **P2**, **P4**, and **P6** would make the AND of *A*, *B*, and *C*. Since these product terms feed into an OR gate, multiple product terms can be ORed together. By programming two product terms leading to the same OR gate with the functions just discussed, the output would be $(A\overline{B}) + (ABC) = A(\overline{B} + C)$. While the implementation just discussed is less efficient than a similar function in a small memory, the efficiency of the memory implementation degrades significantly as the number of inputs increase. However, a PAL is usually not able to implement all possible functions because there are usually not enough product terms leading to the OR gates. For example, to compute an XOR function of 3 inputs requires 4 product terms (ABC, $\overline{A}\overline{B}C$, $\overline{A}B\overline{C}$, $A\overline{B}\overline{C}$), while the PAL structure shown contains only 3 product terms per output. Actual PALs can have many more inputs, outputs, and product terms per output, as well as latches on the outputs with feedback paths to the product term array (i.e., the output of the latch can be included in the product terms in the same way that inputs are handled). Generic Array Logics (GALs) are similar to PALs, except that they are flexible enough to be configured to implement several different types of PALs. They also can include optional inversions on the output. This is important, since it is often much easier to implement the complement of a function in a sum-of-products form and then invert it than it is to implement the uncomplemented sum-of-products form directly. For example, the function $\overline{(ABCD)}$ requires fifteen product terms to implement in sum-of-products form, while the complement requires only one product term.

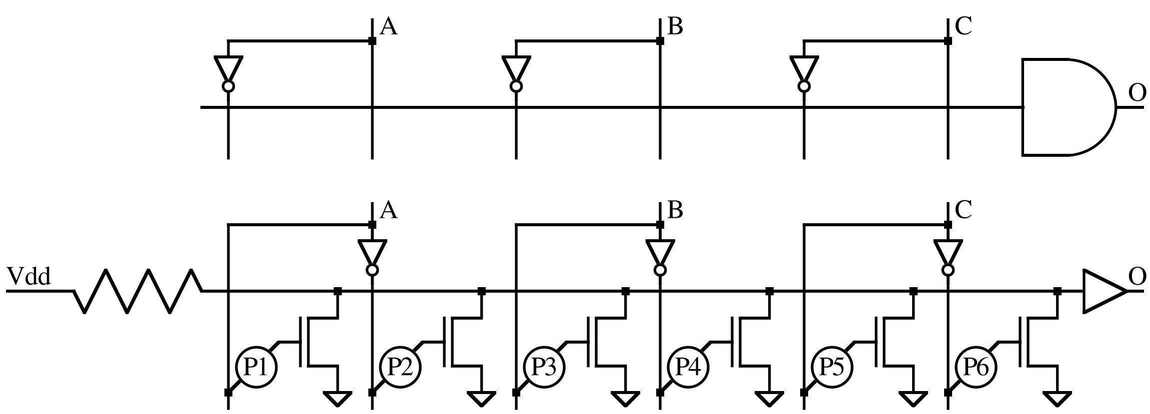


Figure 7. One portion of the PAL structure (top), and the corresponding implementation (bottom) [Wakerly94]. The circles labeled **P** are programmable connections. Note that the inverters are swapped between the two versions. In both circuits, the output value **O** is the AND of any or all vertical lines, depending on which programmable connections are enabled.

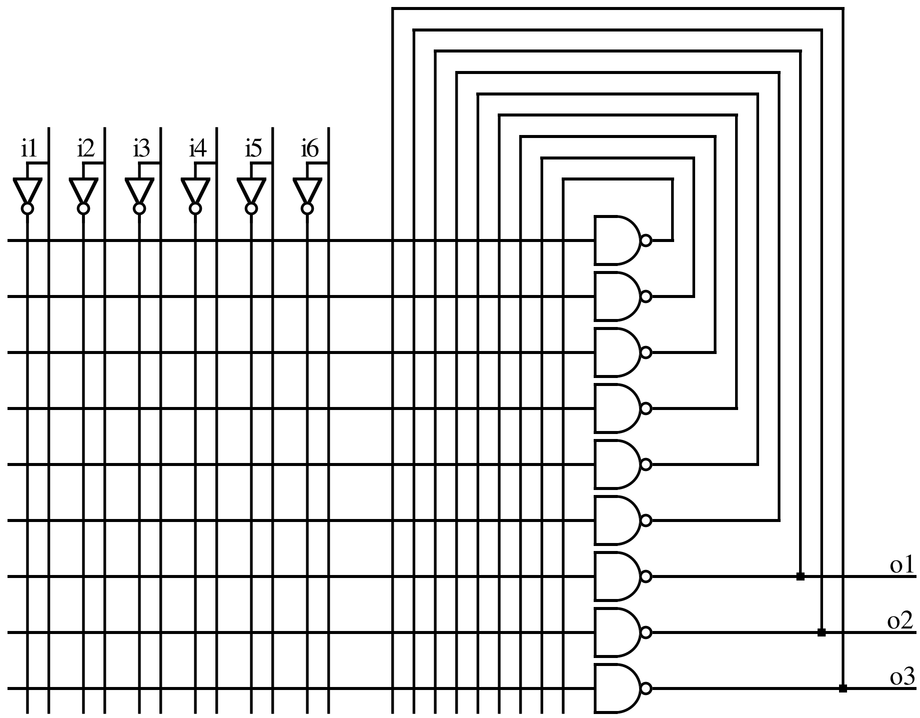


Figure 8. The PML structure [Biehl93].

One limitation of a PAL is that product terms cannot be shared between outputs. Thus, if two outputs both require the product term $\overline{A}B\overline{C}$, they would each need to generate the function with their own product terms. A different form of PLD, called a Programmable Logic Array (PLA), allows product terms to be shared between output functions (Figure 6 right). In a PLA, the AND array of product terms (the AND plane) leads to a similar OR array (the OR plane). These OR functions are implemented the same way the AND functions are, except that instead of inverting the inputs, the outputs are inverted (since the base structure computes a NOR, instead of inverting the inputs, which produces an AND, inverting the output produces an OR). Thus, the OR gates compute an OR of any or all of the input product terms. Thus, the XNOR and majority functions of three inputs can be implemented with five product terms, where a PAL structure would require eight (the product terms are $A1 = (\overline{A}BC)$, $A2 = (A\overline{B}C)$, $A3 = (AB\overline{C})$, $A4 = (\overline{A}\overline{B}\overline{C})$, $A5 = (ABC)$, and the output functions are $XNOR = (A1 + A2 + A3 + A4)$, $MAJ = (A2 + A3 + A4 + A5)$). While PLAs have more flexibility than PALs since the connections between the AND and OR gates are programmable, this flexibility results in lower performance. Primarily the performance degradation is due to the fact that in a PLA a signal must travel through two programmable connections (one in the AND plane, one in the OR plane), while in a PAL the signal goes through only one programmable connection. These connections can add significant delay to the system. Historically, PLAs were introduced before PALs, but PALs are today's most commonly used PLD [Wakerly94].

An even more general structure than the PLA is Programmable Macro Logic (PML). As shown in Figure 8, instead of separate AND and OR structures in the PLA, the PML has a single array of NAND gates, with the outputs feeding back into the array. Thus, the output of a NAND gate can be used as an input to another NAND gate in the same way as a chip input. As we know from basic logic design, a two-level NAND-NAND structure is identical to a two-level AND-OR structure, which means that the PML can implement the same structures as a PAL or PLA. However, by combining the AND and OR functionality into a single array, the user can trade off the number of AND and OR terms on a per-mapping basis, as well as choose to implement some outputs in 1, 2, 3, or more levels of logic, simply by using the proper feedback paths. However, just as in a PLA, each level must pass through a programmable connection, so multi-level paths will be significantly slower than one or two-level paths. A similar product, the ERASIC [Jenkins94], uses a single array of NOR gates instead of the PML's NAND gate structure.

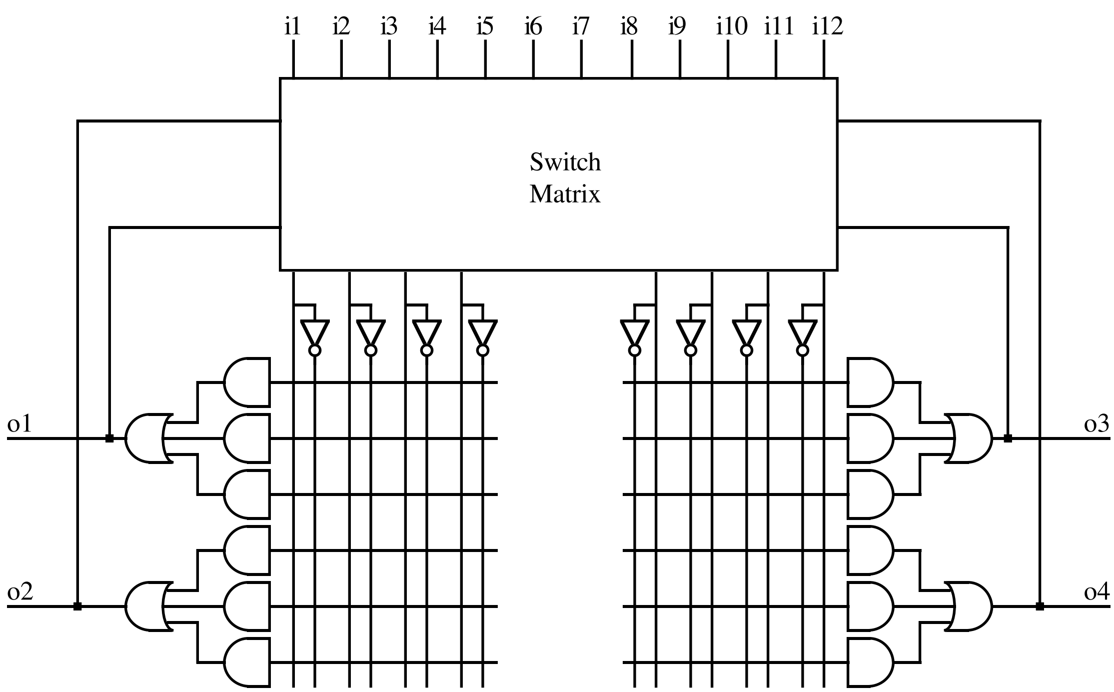


Figure 9. CPLD structure. The switch matrix provides configurable connectivity between the inputs, logic arrays, and any feedbacks from outputs [Biehl93].

While the previous structures are fine for small PLDs, they may not scale to larger arrays. Primarily, as the PLD gets larger, the number of connections to a given product term grows larger, slowing down the signal propagation. To deal with this, Complex Programmable Logic Devices (CPLDs) break the system up into several smaller product term arrays, connected together with a switch matrix (Figure 9). Thus, the CPLD can be viewed as a collection of smaller PLDs and an interconnection network. In this way, the CPLD can have much larger capacity than a single PLD, while keeping propagation delays low.

Because of the regularity and simplicity of PLDs, it is fairly easy to map to a PLD, a process that has been automated by software. These tools can take logic equations and automatically create the configuration files for the PLD. CPLDs may complicate the process by having switch matrices with restricted connectivity, as well as the requirement of partitioning the logic equations into the separate logic arrays, but the process is still much simpler than for many other technologies.

Field-programmable gate arrays

Programmable logic devices concentrate primarily on two-level, sum-of-products implementations of logic functions. They have simple routing structures with predictable delays. Since they are completely prefabricated, they are ready to use in seconds, avoiding long delays for chip fabrication. Field-Programmable Gate Arrays (FPGAs) are also completely prefabricated, but instead of two-level logic they are optimized for multi-level circuits. This allows them to handle much more complex circuits on a single chip, but it sacrifices the predictable delays of PLDs. Note that FPGAs are often considered another form of PLD, often under the heading Complex Programmable Logic Device (CPLD). In this thesis, we will use PLD to exclusively refer to the product-term oriented devices discussed in the previous section.

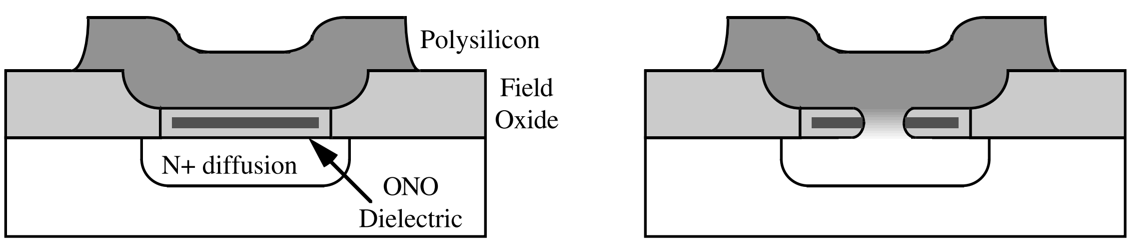


Figure 10. Actel's Programmable Low Impedance Circuit Element (PLICE). As shown at left, an unblown antifuse has an oxide-nitride-oxide (ONO) dielectric preventing current from flowing between diffusion and polysilicon. The antifuse can be blown by applying a 16 Volt pulse across the dielectric. This melts the dielectric, allowing a conducting channel to be formed (right). Current is then free to flow between the diffusion and the polysilicon [Actel94, Greene93].

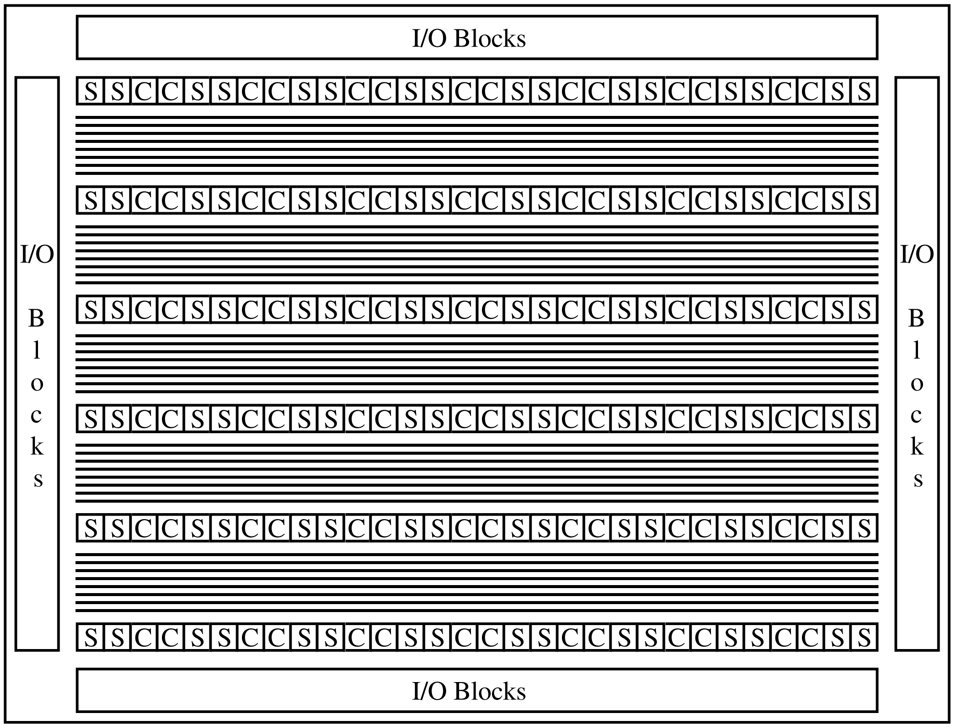


Figure 11. The Actel FPGA structure [Brown92a, Actel94]. Logic blocks (C-Modules and S-Modules) are surrounded by horizontal routing channels like an MPGA.

Just as in PLDs, FPGAs are completely prefabricated, and contain special features for customization. These configuration points are normally either SRAM cells or antifuses. Antifuses are one-time programmable devices (Figure 10), which when "blown" create a connection, while when "unblown" no current can flow between their terminals (thus, it is an "anti"-fuse, since its behavior is opposite to a standard fuse). Because the configuration of an antifuse is permanent, antifuse-based FPGAs are one-time programmable, while SRAM-based FPGAs are reprogrammable, even in the target system. Since SRAMs are volatile, an SRAM-based FPGA must be reprogrammed every time the system is powered up, usually from a ROM included in the circuit to hold configuration files. Note that FPGAs often have on-chip control circuitry to automatically load this configuration data. SRAM cells are larger than antifuses, meaning that SRAM-based FPGAs will have less configuration points than an antifuse based FPGA. However, SRAM-based FPGAs have numerous advantages. Since they are reprogrammable, their configurations can be changed for bug fixes or upgrades. Thus they provide an ideal prototyping medium. Also, these devices can be used in situations where they can expect to have numerous different configurations, such as multi-mode systems and reconfigurable computing machines. More details on such applications are included in Chapter 3. There are many different types of FPGAs, with many different structures. Instead of discussing all of them here, which would be quite involved, this chapter will present a few typical and well-known FPGAs. Details on many others can be found elsewhere [Brown92a, Rose93, Chan94, Jenkins94, Trimberger94, Oldfield95].

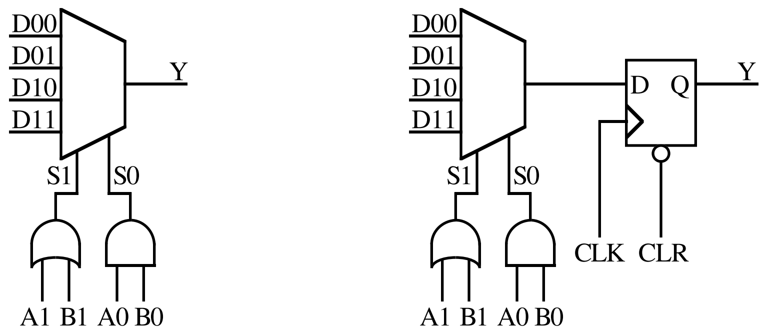


Figure 12. The Actel ACT 3 C-Module (left) and S-Module (right) [Actel94].

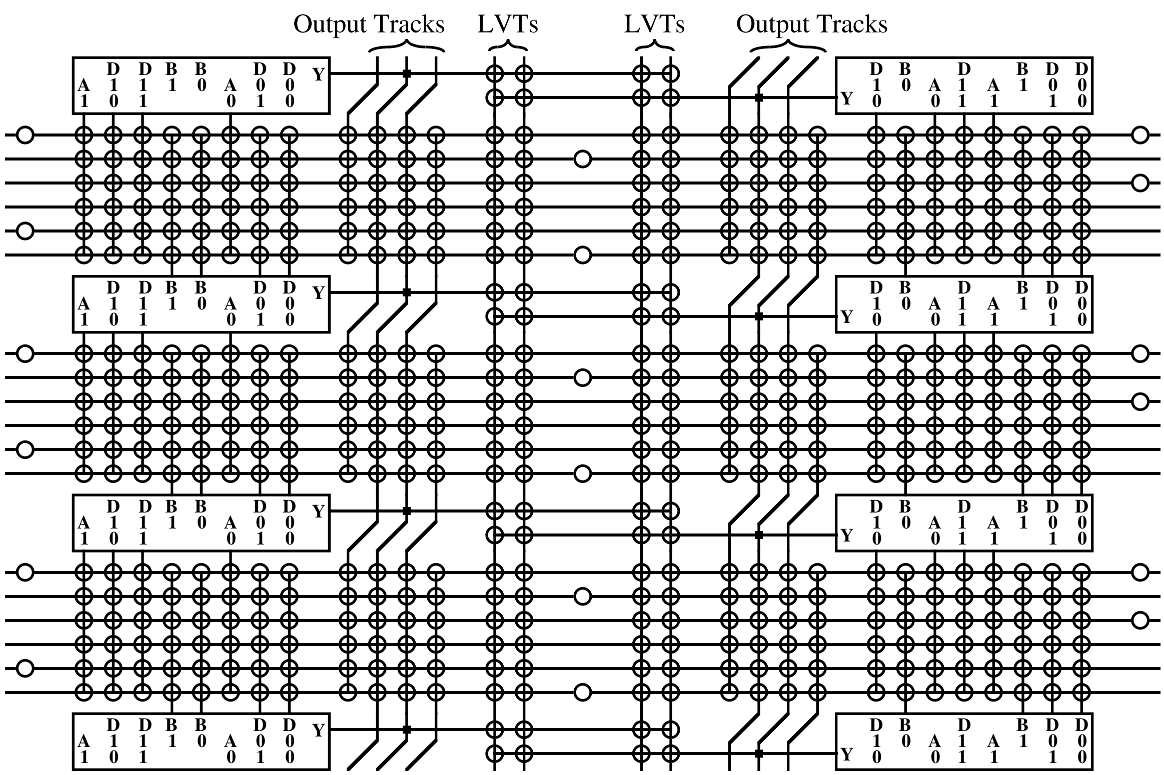


Figure 13. Actel routing structure [Actel94]. Circles are antifuses. Signal lines passing straight through an antifuse are always connected, while signals meeting at right angles in an antifuse or only touching the outside are only connected when the antifuse is blown.

One of the best known antifuse-based FPGAs is the Actel ACT series (we will discuss the third generation FPGA, the ACT 3, in this section). In many ways, the Actel FPGA is essentially a small MPGA that is ready to use instantly. The routing structure of the FPGA is very similar to an MPGA, with rows of logic cells between horizontal routing channels (Figure 11). The cells come in two types: C-Modules, for purely combinational functions, and S-Modules, for sequential logic (Figure 12). Both cells have a logic function based on a 4:1 multiplexer, though the control inputs are fed by either an AND or an OR gate. By properly assigning signals to inputs, this logic block can implement any 2-input function, and many other functions of up to eight inputs. To implement an arbitrary 2-input function, the two inputs are routed to A0 and A1, with B1=0 & B0=1, and D00-D11 are assigned either 0 or 1 to implement the desired function. To implement a function such as a 4-input AND (I1*I2*I3*I4), A1=I1, B1=0, A0=I2, A1=I3, D11=I4, D00=0, D01=0, D10=0. Many other functions are possible. In the S-Module, the logic block feeds into a flip-flop for implementing stateholding elements.

Unlike an MPGA, an FPGA is completely prefabricated, so configuration hardware must be imbedded in the system. In the Actel FPGA, this is accomplished with antifuses scattered throughout the routing topology (Figure 13). Horizontal routing channels run between the rows, and are broken into multiple segments. At the ends of these segments are antifuses, allowing the segments to either be used separately for two different signals (in which case the antifuse is left unblown), or combined together into a single longer route. Inputs to the logic cells come from signals in the channel directly above or below the cell (each input can take signals from only one of these tracks, with half coming from the upper and half from the lower). Which signal is chosen depends on which antifuse is blown. Antifuses are provided for connecting the input to any of the signals in the track, including dedicated power and ground lines for tying inputs to a constant value. The output of a cell is connected to a vertical wire (an output track) traversing the two channels above and two channels below the cell. There are also vertical wires (LVTs) traversing most of the chip. Some of these are segmented, though segments can be combined together in a similar fashion to that used for horizontal tracks. Antifuses are provided for connecting an output to any of the LVTs, and for connecting either the LVTs or the output tracks to any signal in the horizontal channels. Overall, the architecture is optimized primarily for horizontal routing, with approximately twice as many wires running horizontally as vertically. Thus, routes will primarily use the dedicated output track to reach the nearest four channels, and then route horizontally to all the signal destinations. Signals that must move further than the dedicated output will reach will have to use the LVTs, but a good logic placement will avoid this as much as possible.

Antifuses have a significant impact on both the architecture and usage of these systems. Since antifuses are much smaller than other programming technologies, an antifuse-based FPGA has many more configuration sites than an SRAM based FPGA. Thus, the number of choices in the routing structure is much higher than in other FPGAs. On the other hand, since the chips are not reprogrammable, they cannot be changed later for bug corrections, upgrades, nor used in reconfigurable systems. This means that an antifuse-based FPGA is essentially a slow, small MPGA which does not require custom fabrication. They are slower and less dense since in an MPGA the user has more control over the fabrication of the metal layers of the chip, while an antifuse-based FPGA must make the routing general enough to handle a wide range of applications. However, since they do not require custom fabrication in order to configure the chip, they are less expensive (at least at low volumes) than MPGAs, and are ready to use instantaneously.

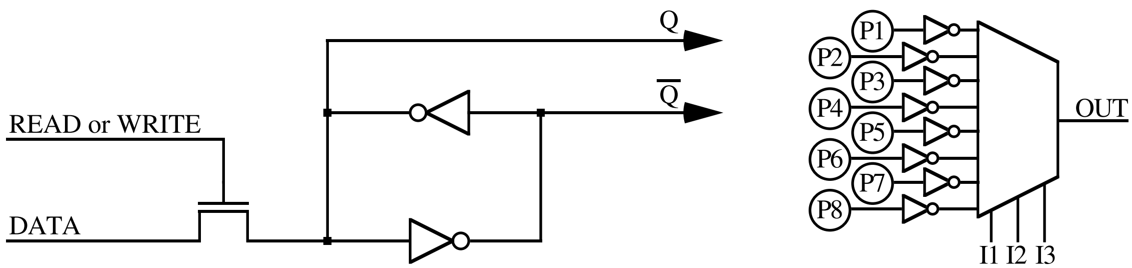


Figure 14. Programming bit for SRAM-based FPGAs (left) [Xilinx94], and a 3-input LUT (right).

In SRAM-based FPGAs memory cells are scattered throughout the FPGA instead of antifuses. As shown in Figure 14 left, a pair of cross-coupled inverters will sustain whatever value is programmed onto them. A single n-transistor gate is provided for either writing a value (the ratio of sizes between the transistor and the upper inverter is set to allow values sent through the n-transistor to overpower the inverter), or reading a value back out. The readback feature is used during debugging to determine the current state of the system. The actual control of the FPGA is handled by the Q and $\overline{Q}$ outputs. One simple application of an SRAM bit is to have the Q terminal connected to the gate of an n-transistor. If a 1 is assigned to the programming bit, the transistor is closed, and values can pass between the source and drain. If a 0 is assigned, the transistor is opened, and values cannot pass. Thus, this construct operates similarly to an antifuse, though it requires much more area. These programming bits can also be fed into decoders, allowing a few bits to control multiple features where only one feature can be active at a time. One of the most useful SRAM-based structures is the lookup table (LUT). By connecting 2^N programming bits to a multiplexer (Figure 14 right), any N-input function can be implemented. Although it can require a large amount of programming bits for large N, LUTs of up to 5 inputs can provide a flexible, powerful function implementation medium. Note that inverters may be necessary between the programming bits and the multiplexer to avoid the programming bit's value from being overwritten during operation.

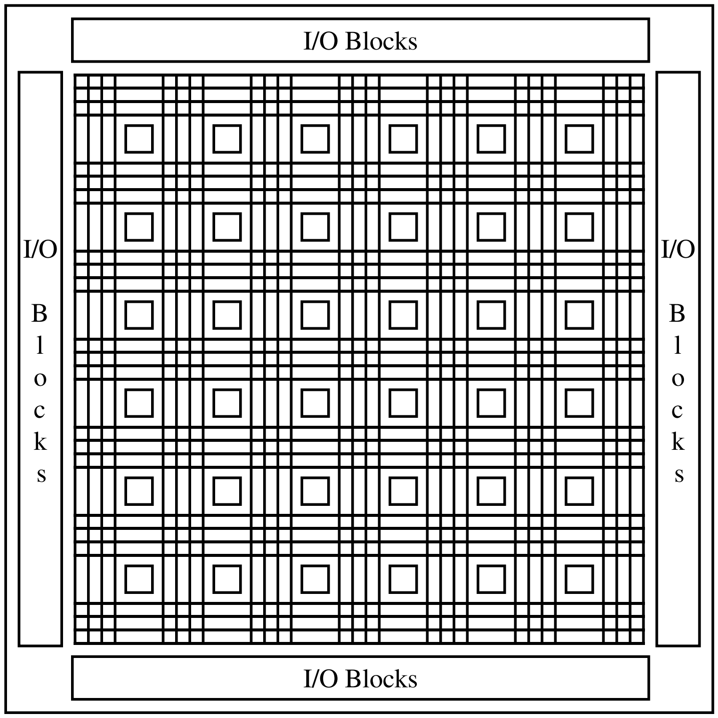


Figure 15. The Xilinx 4000 series FPGA structure [Xilinx94]. Logic blocks are surrounded by horizontal and vertical routing channels.

One of the best known of all FPGAs is the Xilinx Logic Cell Arrays (LCAs) [Trimberger93, Xilinx94]. In this section we will describe their third generation FPGA, the Xilinx 4000 series. As opposed to Actel's channel architecture, the Xilinx array is an "Island-style" FPGA [Trimberger94]. As shown in Figure 15, the logic cells are embedded in a general routing structure that permits arbitrary point-to-point communication. There is no horizontal bias in the routing (as is found in the Actel FPGAs), and the only requirement for good routing is that the source and destinations be relatively close together. Details of the routing structure are shown in Figure 16. Each of the inputs of the cell (**F1**-**F4**, **G1**-**G4**, **C1**-**C4**, **K**) comes from one of a set of tracks adjacent to the cells. The outputs are similar (**X**, **XQ**, **Y**, **YQ**), except they have the choice of both horizontal and vertical tracks. The routing structure is made up of three lengths of lines. Single-length lines travel the height of a single cell, where they then enter a switch matrix (Figure 17 right). The switch matrix allows this signal to travel out vertically and/or horizontally from the switch matrix. Thus, multiple single-length lines can be cascaded together to travel longer distances. Double-length lines are similar, except that they travel the height of two cells before entering a switch matrix (notice that only half the double-length lines enter the switch matrix, and there is a twist in the middle of the line). Thus, double-length lines are useful for longer-distance routing, traversing two cell heights without the extra delay and the wasted configuration sites of an intermediate switch matrix. Finally, longlines are lines that go half the chip height, and do not enter the switch matrix. In this way, very long-distance routes can be accommodated efficiently. With this rich sea of routing resources, the Xilinx 4000 series is able to handle fairly arbitrary routing demands, though mappings emphasizing local communication will still be handled more efficiently.

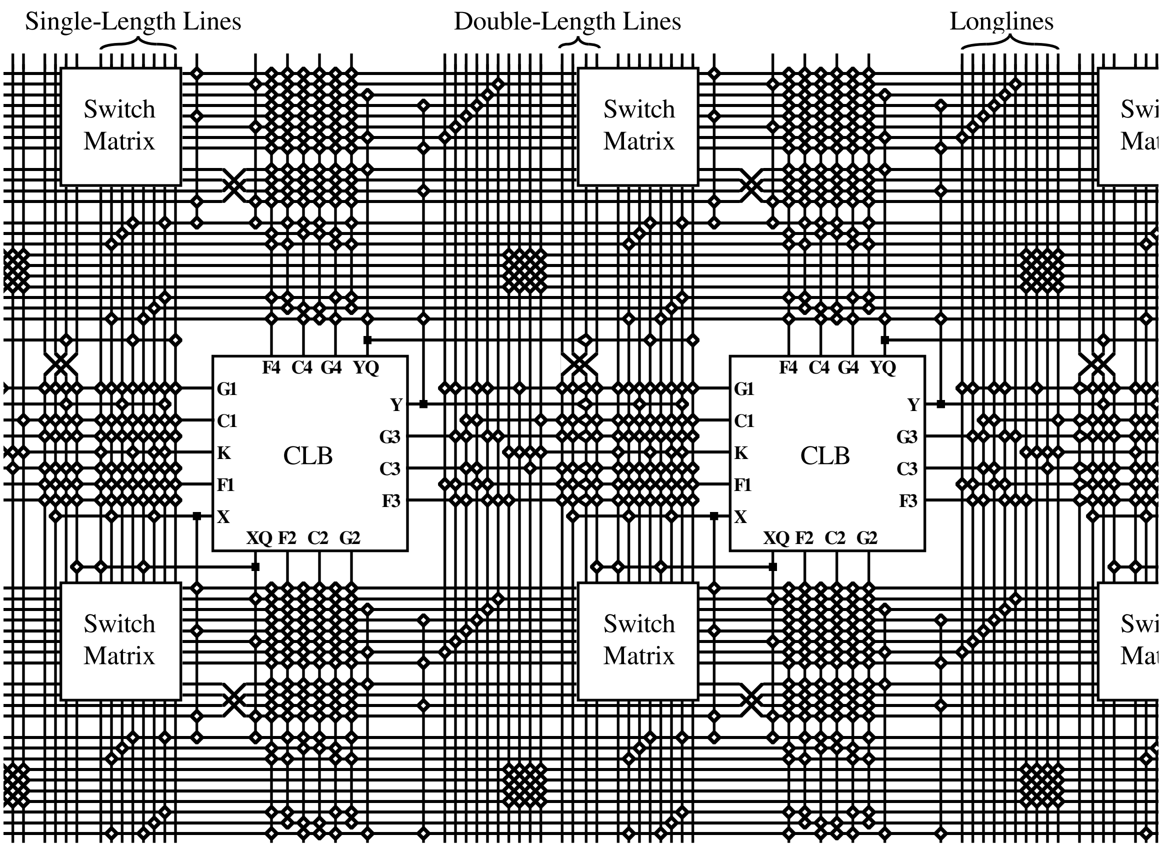


Figure 16. Details of the Xilinx 4000 series routing structure [Xilinx94]. The CLBs are surrounded by vertical and horizontal routing channels containing Single-Length Lines, Double-Length Lines, and Longlines. Empty diamonds represent programmable connections between perpendicular signal lines (signal lines touching on opposite sides of the diamonds are always connected).

As shown in Figure 17 left, the Xilinx 4000 series logic cell is made up of three lookup-tables (LUTs), two programmable flip-flops, and multiple programmable multiplexers. The LUTs allow arbitrary combinational functions of its inputs to be created. Thus, the structure shown can perform any function of five inputs (using all three LUTs, with the **F** & **G** inputs identical), any two functions of four inputs (the two 4-input LUTs used independently), or some functions of up to nine inputs (using all three LUTs, with the **F** & **G** inputs different). SRAM controlled multiplexers then can route these signals out the **X** and **Y** outputs, as well as to the two flip-flops. The inputs at top (**C1**-**C4**) provide the third input to the 3-input LUT, enable and set or reset signals to the flip-flops, and a direct connection to the flip-flop inputs. This structure yields a very powerful method of implementing arbitrary, complex digital logic. Note that there are several additional features of the Xilinx FPGA not shown in these figures, including support for embedded memories and carry chains.

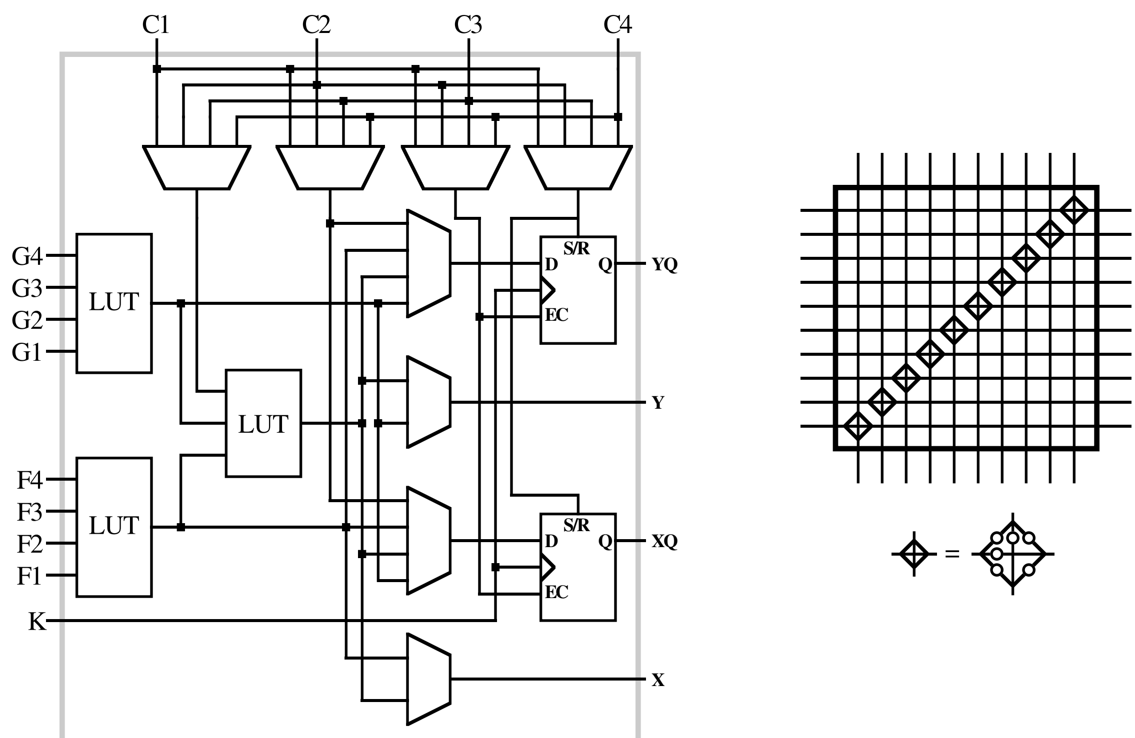


Figure 17. Details of the Xilinx CLB (left) and switchbox (top right) [Xilinx94]. The multiplexers, LUTs, and latches in the CLB are configured by SRAM bits. Diamonds in the switchbox represent six individual connections (bottom right), allowing any permutation of connections between the four signals incident to the diamond.

While many SRAM-based FPGAs are designed like the Xilinx architecture, with a routing structure optimized for arbitrary, long-distance communications, several other FPGAs concentrate instead on local communication. The "Cellular" style FPGAs [Trimberger94] feature fast, local communication resources, at the expense of more global, long-distance signals. As shown in Figure 18, the CLi FPGA [Jenkins94] has an array of cells, with a small amount of routing resources running horizontally and vertically between the cells. There is one local communication bus on each side of the cell. It runs the height of eight cells, at which point it enters a repeater. Express buses are similar to local buses, except that there are no connections between the express buses and the cells. The repeaters allow access to the express buses. These repeaters can be programmed to connect together any of the two local buses and two express buses connected to it. Thus, a small amount of global communication can be accomplished on the local and express buses, with the local buses allowing shorter-distance communications and connections to the cells, while express buses allow longer-distance connections between local buses.

While the local and global buses allow some of the routing flexibility of the Xilinx FPGA's arbitrary routing structure, there are much fewer buses in the CLi architecture than are present in the Xilinx FPGA. The CLi FPGA instead features a significant amount of local communication resources. As shown in Figure 19, each cell receives two signals from each of its four neighbors. It then sends the same two outputs (**A** and **B**) to all of its neighbors. That is, the cell one to the north will send signals **AN** and **BN**, and the cell one to the south will send **AS** and **BS**, while both will receive the same signals **A** and **B**. The input signals become the inputs to the logic cell (Figure 20).

Instead of Xilinx's LUTs, which require a large amount of programming bits per cell, the CLi logic block is much simpler. It has multiplexers controlled by SRAM bits which select one each of the **A** and **B** outputs of the neighboring cells. These are then fed into AND and XOR gates within the cell, as well as a flip-flop. Although the possible functions are complex, notice that there is a path leading to the **B** output that produces the NAND of the selected **A** and **B** inputs, and sending it out the **B** output. This path is enabled by setting the two 2:1 multiplexers to their constant input, and setting **B**'s output multiplexer to the 3rd input from the top. Thus, the cell is functionally complete. Also, with the **XOR** path leading to output **A**, the cell can efficiently implement a half-adder. The cell can perform a pure routing function by connecting one of the **A** inputs to the **A** output, and any of the **B** inputs to the **B** output, or vice-versa. This routing function is created by setting the two 2:1 multiplexers to their constant inputs, and setting **A**'s and **B**'s output multiplexer to either of their top two inputs. There are also provisions for bringing in or sending out a signal on one or more of the neighboring local buses (**NS1**, **NS2**, **EW1**, **EW2**). Note that since there is only a single wire connecting the bus terminals, there can only be a single signal sent or received from the local buses. If more than one of the buses is connected to the cell, they will be coupled together. Thus, the cell can take a signal running horizontally on an **EW** local bus, and send it vertically on a **NS** local bus, without using up the cell's logic functionality. However, by bringing a signal in from the local buses, the cell can implement two 3-input functions.

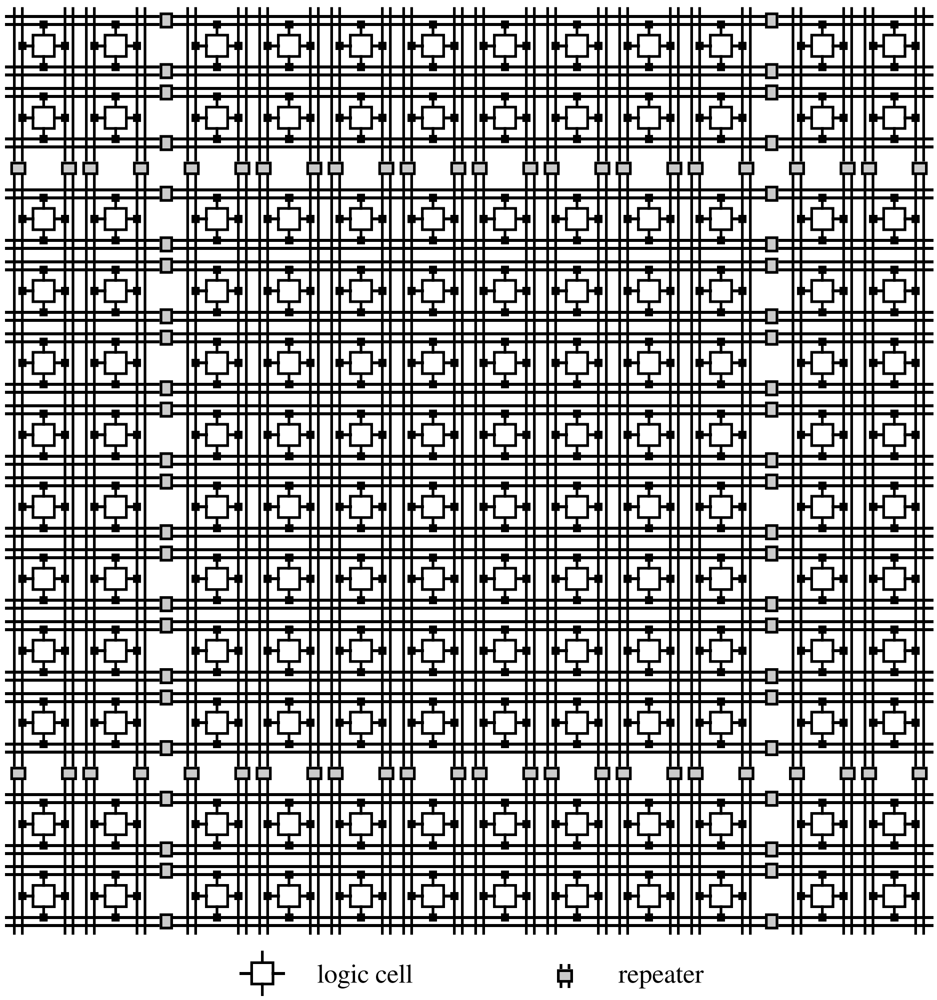


Figure 18. The CLi6000 routing architecture [Jenkins94]. One 8x8 tile, plus a single set of surrounding rows and columns, is shown. The full array has many of these tiles abutted horizontally and vertically.

The major differences between the Island style architecture of the Xilinx 4000 series and the Cellular style of the CLi FPGA is in their routing structure and cell granularity. The Xilinx 4000 series is optimized for complex, irregular random logic. They feature a powerful routing structure optimized for arbitrary global routing, and large logic cells capable of providing arbitrary 4-input and 5-input functions. This provides a very flexible architecture, though one that requires a large amount of programming bits per cell (and thus cells that take up a large amount of space on the chip). In contrast, the CLi architecture is optimized for highly local, pipelined circuits such as systolic arrays and bit-serial arithmetic. Thus, they emphasize local communication at the expense of global routing, and have simple cells. Because of the very simple logic cells there will be many more CLi cells on a chip than is found in the Xilinx FPGA, yielding a greater logic capacity for those circuits that match the FPGA's structure. Because of the restricted routing, the CLi FPGA is much harder to automatically map to than the Xilinx 4000 series, though the simplicity of the CLi architecture make it easier for a human designer to hand-map to the CLi's structure. Thus, in general cellular architectures tend to appeal to designers with appropriate circuit structures who are willing to spend the effort to hand-map their circuits to the FPGA, while the Xilinx 4000 series is more appropriate for handling random-logic tasks and automatically-mapped circuits.

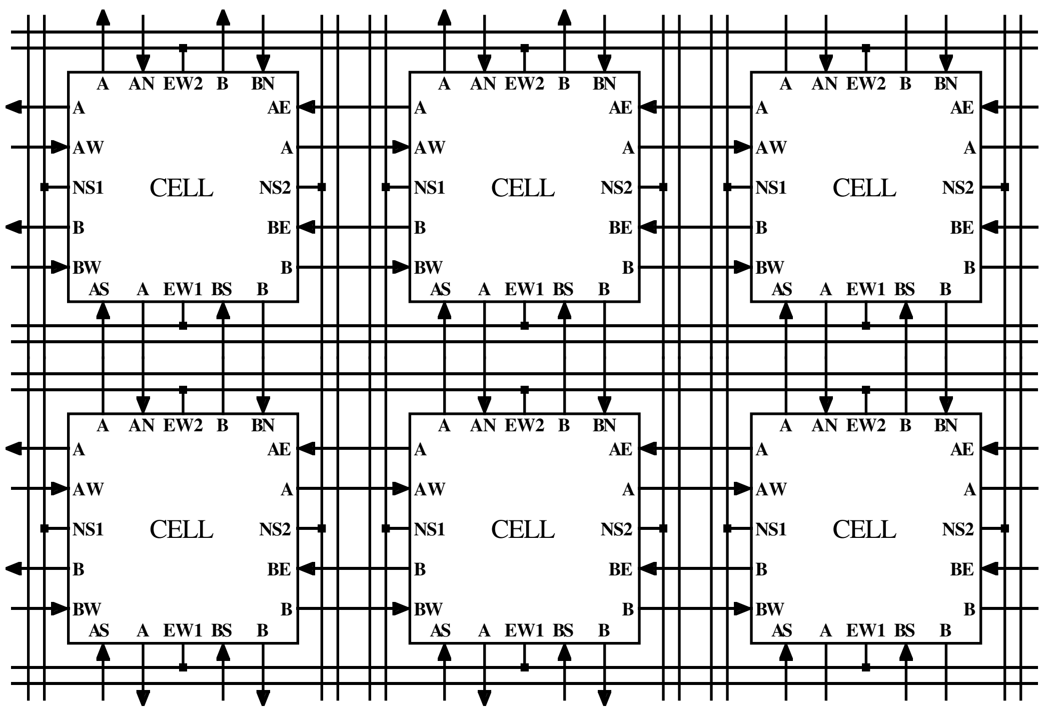


Figure 19. Details of the CLi routing architecture [Jenkins94].

Compared with technologies such as full-custom, standard cells, and MPGAs, FPGAs will in general be slower and less dense. In the case of SRAM-based FPGAs, this is due to the configuration points, which take up a significant amount of space, and add extra capacitance and resistance (and thus delay) to the signal lines. Thus, the programming bits add an unavoidable overhead to the circuit, which can be reduced by limiting the configurability of the FPGA, but never totally eliminated. Also, since the metal layers in an FPGA are prefabricated, while the other technologies custom fabricate the metal layers for a given circuit, the FPGA will have less optimized routing. This again results in slower and larger circuits. However, even given these downsides, FPGAs have the advantage that they are completely prefabricated. This means that they are ready to use instantly, while mask-programmed technologies can require weeks to be customized. Also, since there is no custom fabrication involved in an FPGA, the fabrication costs can be amortized over all the users of the architecture, removing the significant NRE's of other technologies. However, per-chip costs will in general be higher, making the technology better suited for low volume applications. Also, since SRAM-based FPGAs are reprogrammable, they are ideal for prototyping, since the chips are reusable after bug fixes or upgrades, where mask-programmed and antifuse versions would have to be discarded.

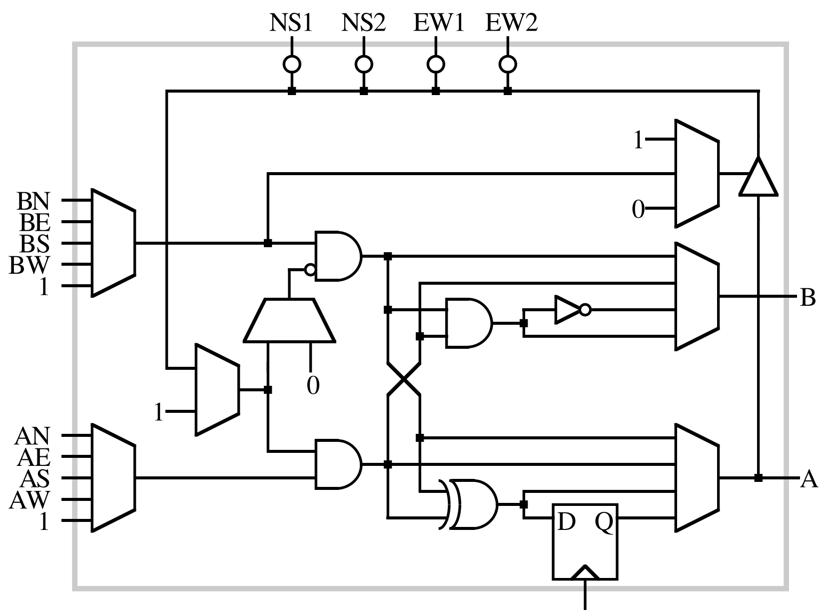


Figure 20. The CLi logic cell [Jenkins94].

Although, as we just discussed, FPGAs have an unavoidable overhead due to configurability, making them slower and larger than mask-programmed and custom technologies developed in the same processes, SRAM-based FPGAs may give the designer access to better fabrication technology than other approaches. Specifically, as a fabrication process is being readied for production use, the fabricator will produce some early designs to debug the technology. In general, the first chips off the line are test structures, and then SRAMs. Increasingly, an SRAM-based FPGA is the next chip to be fabricated on these lines. For the fabricator, SRAM-based FPGAs have the advantages that they can quickly be designed for a given technology, since they consist of a single tile replicated across the chip surface, yet are highly testable because of their programmability, and have many of the same features of full-custom designs. Thus, an FPGA can quickly be created in a technology, and used for fault location and diagnosis to isolate errors in the fabrication process. For the FPGA manufacturer (which often do not have their own fabrication lines, and thus use outside fabricators), this gives them the earliest access to a new process technology, increasing the performance and capacity of their products. Antifuse-based FPGAs do not have this same advantage. Creating the antifuses requires extra fabrication steps. This complicates the fabrication, relegating antifuse manufacture to older, lower quality fabrication processes. While antifuses are much denser than SRAM bits, which makes antifuse-based FPGAs denser than SRAM-based FPGAs created with the same process, an SRAM-based FPGA's access to new fabrication technologies may offset this cost.

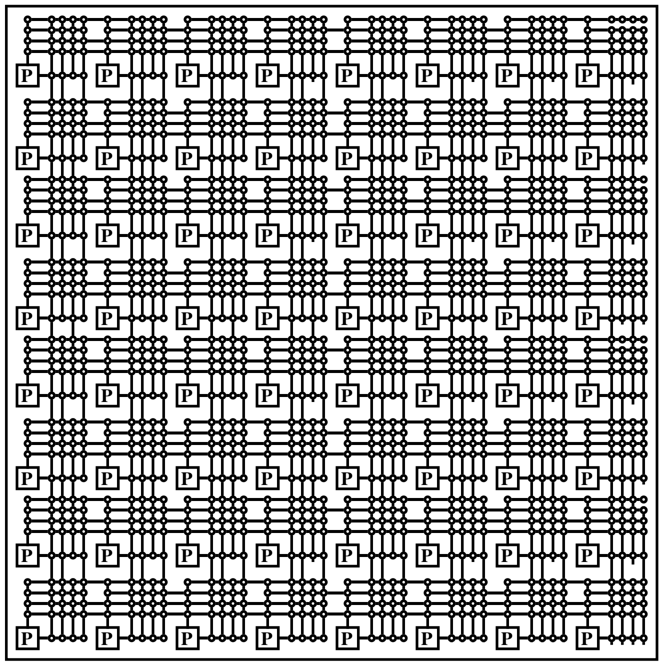


Figure 21. The Aptix FPIC architecture [Aptix93a]. The boxed P indicates an I/O pin.

A technology similar to SRAM-based FPGAs is Field-Programmable Interconnect Components (FPIC) [Aptix93a] and Devices (FPID) [I-Cube94] (we will use FPIC from now on to refer to both FPIC & FPID devices). Like an SRAM-based FPGA, an FPIC is a completely prefabricated device with an SRAM-configured routing structure (Figure 21). Unlike an FPGA, an FPIC has no logic capacity. Thus, the only use for an FPIC is as a device to arbitrarily interconnect its I/O pins. While this is not generally useful for production systems, since a fixed interconnection pattern can be achieved by the printed circuit board that holds the circuit, it can be quite useful in prototyping and reconfigurable computing (these applications are discussed in Chapter 3). In each of these cases, the connections between the chips in the system may need to be reconfigurable, or this connection pattern may change over time. In a reconfigurable computer, many different mappings will be loaded onto the system, and each of them may desire a different interconnection pattern. In prototyping, the connections between chips may need to be changed over time for bug fixes and functionality upgrades. In either case, by routing all of the I/O pins of the logic-bearing chips to FPICs, the interconnection pattern can easily be changed over time. Thus, fixed routing patterns can be avoided, hopefully increasing the performance and capacity of the prototyping or reconfigurable computing machine.

There is some question about the economic viability of FPICs. The problem is that they must provide some advantage over an FPGA with the same I/O capacity, since in general an FPGA can perform the same role as the FPIC. One possibility is providing significantly more I/O pins in an FPIC than are available in an FPGA. This can be a major advantage, since it takes a significant amount of smaller I/O chips to match the functionality of a single high-I/O chip (i.e., a chip with N I/Os requires three chips with 2/3 the I/Os to match the flexibility). However, because the packaging technology necessary for such high I/O chips is somewhat exotic, FPICs can be expensive. Another possibility is to provide higher performance or smaller chip size with the same I/O capacity. Since there is no logic on the chip, the space and capacitance due to the logic can be removed. However, even with these possible advantages, FPICs face the significant disadvantage that they are restricted to a limited application domain. Specifically, while FPGAs can be used for prototyping, reconfigurable computing, small volume products, fast time-to-market systems, and multi-mode systems, FPICs are restricted to the interconnection portion of prototyping and reconfigurable computing solutions. Thus, FPICs may never become commodity parts, greatly increasing their unit cost.

Discrete components

In the previous sections we have discussed custom fabricated and field-programmable implementation technologies for digital logic. The final alternative is to use standardized, non-programmable chips. Specifically, simple logic structures such as inverters, 2-input to 4-input logic gates, and latches can be prefabricated on individual small chips. By connecting these chips in the proper manner, the desired functionality can be implemented. This interconnection is done at the board level, either by prototyping methods such as wire-wrap and breadboarding, or by fabricating a custom circuit board to hold the parts. Performing the routing customization at the board level instead of the chip level tends to be cheaper and requires less lead time, since board fabrication is much simpler than chip fabrication. Also, since the chips are simple and standardized, they can be prefabricated, and will be quite cheap because of their small size and their economies of scale. These discrete components have been used in many different circuits, amortizing many of the costs. However, since each of the chips will hold only a small amount of logic (perhaps four 2-input functions), a large circuit will require a large amount of chips. This slows down the circuit, complicates the circuit board design, and greatly increases the circuit board size. Thus, while this approach may be cheap for small circuits, it quickly becomes unattractive for large systems.

Microcontrollers and DSPs

By prefabricating all of the components, a discrete component methodology reduces costs and avoids delays associated with custom fabrication. In the discrete component approach, all the chips are quite simple, having very restrictive functionality. However, premade components do not necessarily need to be simple. For example, microprocessors are completely prefabricated, but are some of the most complex chips currently made. Premade components can be found for many standardized, generally useful applications. Two types of premade components that merit special consideration are microcontrollers and digital signal processors (DSPs). These are complex, prefabricated parts that are widely used for many applications.

$$Out = \sum_{i=1}^{k} \left(C_i \times X_i \right)$$

Equation 1. FIR filter equation.

A microcontroller is essentially a simple microprocessor optimized for embedded control situations. The microcontroller has much the same internal structure as a microprocessor, with a datapath capable of performing complex operations and control flow dictated by a series of instructions. The major difference between a microcontroller and a microprocessor is that the microcontroller tends to be smaller, slower, and have a smaller address space, but it has more interfaces for external communications. Many microcontrollers are only 8-bit or 16-bit processors, with memories on the order of only a few kilobytes. However, because they are simple, they can be very cheap. This makes them ideal for control applications, moderating the operation of embedded systems such as automobile engines, microwave ovens, and many other applications. Since they have good external connectivity, and since the performance demands of these applications are in general relatively low, a microcontroller becomes a very cheap alternative to either a large number of discrete components, or expensive custom ASICs. The operation of the microcontroller is completely dictated by a program stored either in an internal or external PROM, making it field-programmable much in the same way as PLDs and FPGAs. However, a microcontroller is capable of much more complex control flow than an equivalent PLD or FPGA, albeit at a much slower speed, making it valuable for many applications. Also, since many applications will fit into the on-chip ROM, a microcontroller provides a complete, highly integrated processing element for many applications. Replacing a microcontroller with a standard microprocessor would require several more chips, since standard microprocessors require separate chips to implement the ROM and the external interfaces.

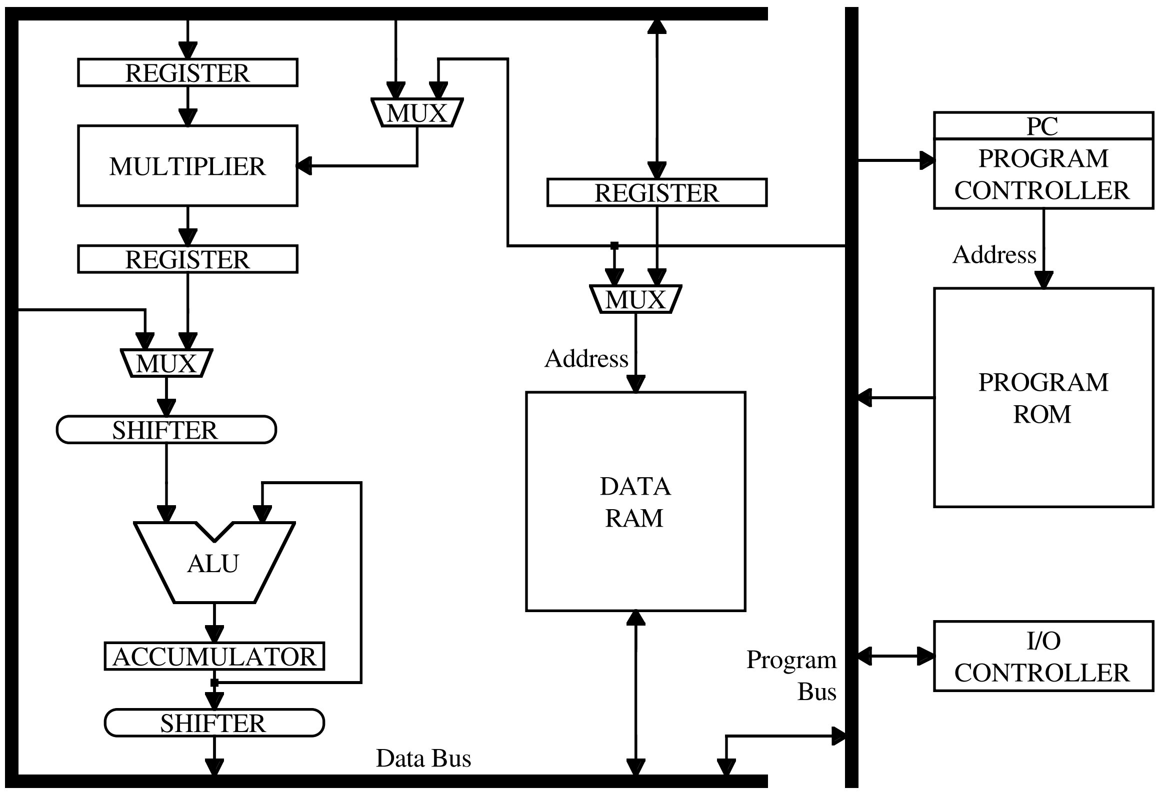


Figure 22. An example DSP architecture [Lee88]. The datapath is at left, which is optimized for high-speed multiply-accumulate computations. Shifters are provided to scale values to avoid overflows. At center is the data RAM addressing hardware, and at right is the program fetch hardware and the I/O controller.

Digital signal processors (DSPs) [Lee88, Lee89] are also quite similar to microprocessors, but instead of using slow, simple processors to reduce costs like a microcontroller, a DSP is highly optimized to provide the highest possible performance for a restricted application domain. A DSP is designed to perform multiply-accumulate operations (A := A + B*C) as fast as possible. An example of a DSP architecture is shown in Figure 22. As can be seen, the DSP includes instruction and data addressing hardware like a microprocessor, but the datapath is optimized for multiply-accumulate operations instead of the more general operations found in a standard processor. Because of the restricted application domain, the processor can be optimized to achieve high performance operations, at least for those operations that fit the DSP's computation model. This yields a programmable processor capable of handling many signal processing tasks. For example, a finite impulse response (FIR) filter computes the function shown in

Equation 1, where X_i is the input value received i steps previous. This computation fits the multiply-accumulate model quite well, and current DSPs can compute a k-input FIR filter in one step per input.

Design style summary

In the previous sections we have discussed many different methods for implementing digital logic. For traditional logic implementation tasks the choice of implementation technology is usually most constrained by capacity, performance, and price. Full-custom designs provide the highest capacity and highest performance, but at a significant price. Other implementations tend to be cheaper (at least in lower volumes), but have lower capacity and performance. In order, from highest capacity and performance to lowest, are Standard Cell, MPGA, FPGA, PLD, and discrete component implementations. The same ordering is true in price, at least at low volumes, with Standard Cell and MPGA implementations costing the most, and FPGA, PLD, and discrete component implementations costing the least. Thus, when choosing an implementation technology, the designer will use the lowest price device that can meet their capacity and performance demands. FPGA and MPGA designs are normally preferred over full-custom implementations, but they may not be powerful enough to handle the designer's needs. Note that as the volume of parts produced increases, the cost of the various technologies changes. Specifically, while FPGAs and PLDs are much cheaper than other technologies at low volumes, since they have no up-front costs for masks and other fabrication costs, their unit costs are much higher than other technologies. Full custom designs have the highest startup costs, because none of their fabrication is shared with other products, and the design process is quite complex. However, as the quantity of parts produced grows, this fixed cost is amortized over a larger number of chips. This makes the per-chip costs dominate, and a full-custom design becomes the cheapest implementation.

In some cases, the most important feature of an implementation technology is not cost, capacity, or performance, but is instead some other issue. For example, time-to-market can often be a primary factor. If a product must wait for the design and fabrication of a full-custom design, it will take many months or even years for the product to reach the market. However, if an FPGA or PLD implementation is chosen, the logic might be ready in a matter of days. Note that a hybrid approach is also possible. A product can be quickly implemented in FPGAs, and brought to market quickly. While the FPGA-based version is shipped, an MPGA or full-custom implementation can be readied. Once this design is debugged and fabricated, it will replace the FPGAs in the product being shipped. Thus, the lower per-unit costs of an MPGA or full-custom design decrease costs for later products, while the FPGA implementation allows the product to be brought to market as soon as possible.

There are many other situations where the unique abilities of FPGAs play a critical role. These are detailed in Chapter 3. As we will show, the reprogrammability of FPGAs, coupled with their ability to contain significant amounts of logic, open up many opportunities not possible in other technologies.

Overview of single FPGA mapping software

While some circuits are designed by hand, in many cases automatic mapping software is critical to logic development. This is particularly true for technologies such as FPGAs, where in general the complete mapping process is carried out by mapping software. In the following sections we will discuss some of the most important steps in mapping to LUT (lookup-table) based FPGAs. Note that the process is similar, but not identical, for other types of FPGAs. First is technology mapping, which restructures the input netlist into the logic blocks of the FPGA. Next, placement decides which specific logic blocks inside the FPGA will contain the logic functions created by technology mapping. Finally, routing determines what routing resources inside the FPGA will be used to carry the signal from where they are generated to where they are used. A more detailed treatment of all of these tasks can be found elsewhere [Venkateswaran94].

Technology mapping

The user of an FPGA provides as input a circuit specified as some interconnection of basic logic gates and functions. These functions may havemore or less inputs than the LUT in the FPGAs that will implement them. When the logic gate has too many inputs, it needs to be split into smaller functions that can fit inside the LUTs in the FPGAs. If the gates have too few inputs, several interconnected gates could be combined to fit into a single LUT, decreasing the amount of LUTs needed to handle the mapping. By reducing the number of LUTs, more logic can be fit in the same sized FPGA, or a smaller FPGA could be used. The process of restructuring the logic to best fit the logic blocks in an FPGA is called technology mapping.

There are many different methods and approaches to the technology mapping of circuits for FPGA implementation [Brown92a, Vincentelli93]. An example of this process is shown in Figure 23. The circuit at left is restructured into four 5-input LUTs, designated by the gray loops at right. Some of the logic function, such as gate **7** at left, have more inputs than the LUT can handle. Thus, the gate will be decomposed into two gates (**7** and **8** at right). Then, the gates are grouped together into LUTs, while trying to minimize the total number of LUTs required. Note that this grouping can be complex. For example, even though gates **1** and **3** each have three inputs, and thus should not fit into a single LUT with gate **2**, since input **C** is shared between the two gates a single 5-input LUT can handle all three gates. Finding this reconvergent fanout can be difficult. Also, this grouping process can cause the logic to be restructured. For example, gate **5** at left is duplicated, creating gates **5** and **6** at right. Although this seems like it would increase the hardware cost, replicating the logic can actually reduce the logic cost. In the example shown, if gate **5** was not replicated, it could not be grouped with either of its fanouts, since by grouping a gate with its fanout the gate's output is no longer available to other functions. However, by duplicating the gate, the duplicates can each be grouped with one of the fanouts, reducing the total LUT count.

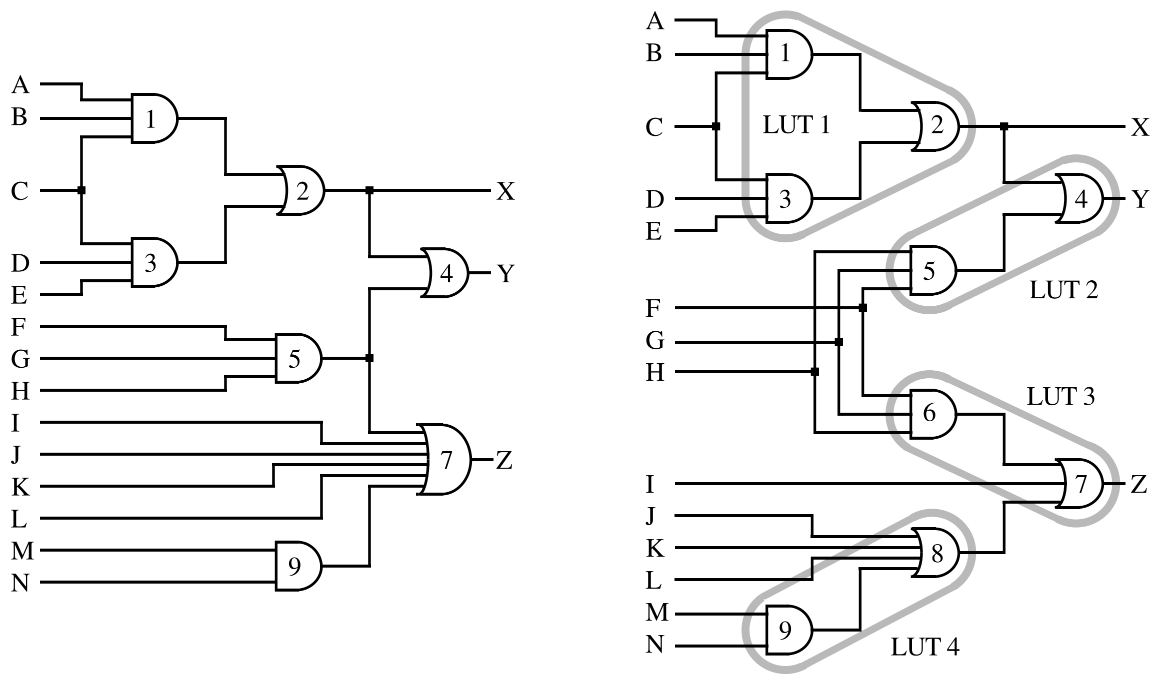


Figure 23. Example of 5-input LUT technology mapping. The input circuit (left) is restructured and grouped together into 5-input functions (right). The gray loops at right indicate individual LUTs. The numbers on the gates are for identification.

Many approaches and optimizations are possible for technology mapping. Instead of just mapping for LUT count, some algorithms optimize for performance or routeability (i.e., how easy it is to route the logic generated by technology mapping). Also, real FPGAs usually have logic blocks that are more complex than a single n-input LUT, and thus require more complex mapping algorithms. Numerous approaches to these and other mapping issues are presented in the literature.

Placement

Placement takes the logic functions formed by technology mapping and assigns them to specific logic blocks in the FPGA. This process can have a large impact on the capacity and performance of the FPGA. Specifically, routing between distant points in an FPGA requires a significant amount of routing resources.

Thus, this path will be much slower, and use up many valuable resources. Because of this, the primary goal of placement is to minimize the length of signal wires in the FPGA. To do this, logic blocks that communicate with one another are placed as close together as possible. For example, Figure 24 shows a placement of the logic functions created by technology mapping in Figure 23. Since function **2** takes the output of function **1** as an input, and shares inputs **F**, **G**, and **H** with function **3**, function **2** is placed between function **1** and function **3**. Also, **F**, **G**, and **H** are assigned to I/O blocks close to function **2**.

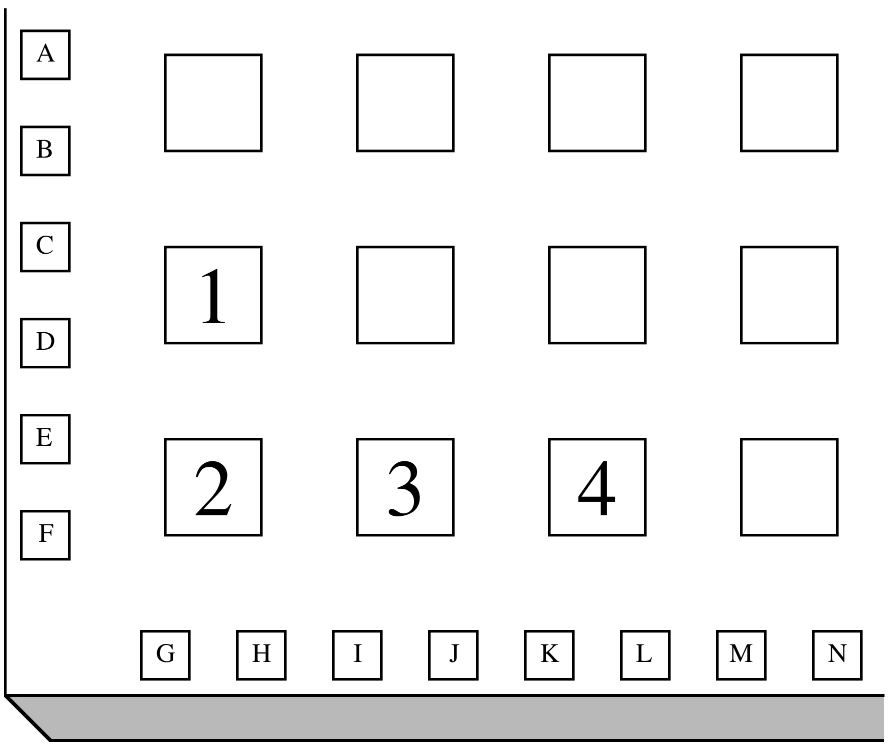


Figure 24. Placement of the circuit from Figure 23. The numbers are the number of the LUT (from the technology mapping) assigned to that logic block, while the letters are the assignment of input signals to I/O blocks.

Placement is a complex balancing act. Logic circuits tend to have a significant amount of connectivity, with many different functions communicating together. Trying to find the best 2D layout of these elements can be quite difficult, since many functions may want to be placed together to minimize communications, while only a small fraction will fit within a given region of the FPGA. Thus, the placement tool must decide which functions are most important to place together, not just to minimize the distances of communications between these functions, but to minimize the total communication in the system.

The most common technique for performing placement for FPGAs (as well as other technologies) is simulated annealing [Shahookar91]. In order to use simulated annealing to solve an optimization problem, the programmer must generate a cost function and a move function. A cost function looks at a state of the system and assigns a value to the desirability of that state, with a lower value indicating a better result. For placement, a state of the system would be an assignment of logic functions to logic blocks, and I/O connections to I/O blocks. A cost function could be the total wirelength necessary to route in this configuration (this would need to be an estimate, since exact numbers are time-consuming to calculate, and simulated annealing requires the cost metric to be quickly computed). Thus, states that have the smallest cost would require the least amount of routing, and thus would be better placements. More complex cost metrics, which take into account issues such as critical paths, are also possible. A move function is simply a method of transforming the current state of the system into a new state. Through repeated applications, this function should be capable of transforming any state of the system to any other. For placement, a move function could be to randomly pick two logic blocks in the FPGA and swap their contents.

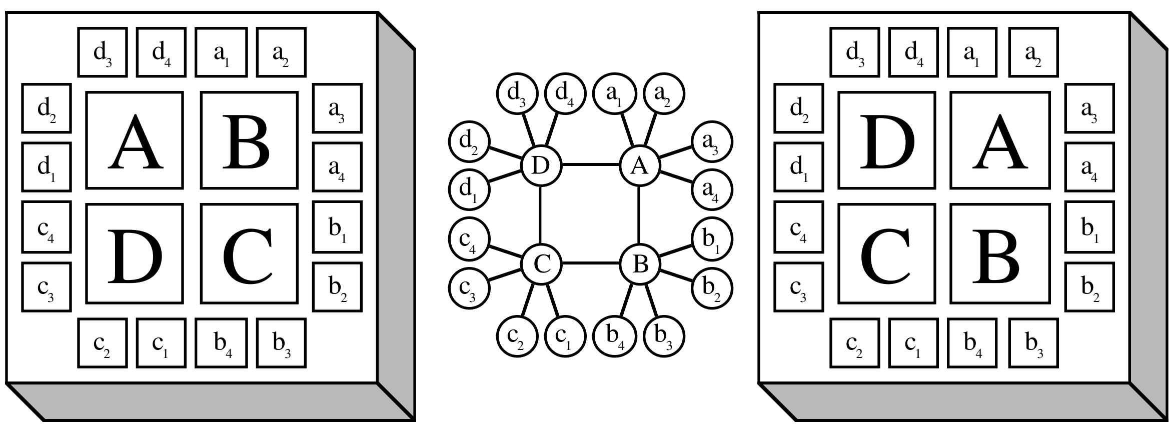


Figure 25. An example of a local minima in placement. The circuit at center, placed as shown at left, is in a local minima. No swap of logic or I/O functions will reduce the total wirelength. However, the placement at right is significantly better.

One way to perform placement once a cost and move function are defined is to first pick a random starting point. The algorithm then repeatedly applies the move function to the current state of the system, generating a new state. If this state has a lower cost than the current state, it is accepted, and replaces the current state. Otherwise the current state is retained. Thus, this algorithm will greedily accept good moves, and move into a local minimum in the cost function’s state space. The problem is that most cost functions have a huge number of local minima, many of which are much worse than the optimal placement (Figure

25). Specifically, from a given state there may be no way to swap two logic functions and reduce the cost (thus, we are in a local minima), though two or more pairs of swaps can greatly improve the placement.

Simulated annealing avoids the problem of getting caught in local minima. Like the greedy algorithm, simulated annealing takes the current state, uses the move function to generate a new state, and compares the cost metric in both states. If the move is a good move (that is, the cost function is lower for the new state than the old state) the new state replaces the current state. However, instead of rejecting all bad moves (move that increase the cost function), simulated annealing accepts some bad moves as well. In this way, the algorithm can get out of a local minima by accepting one or two bad moves. Subsequent good moves will then improve the results again, hopefully finding better results than the previous local minima.

The method of how to determine what bad moves to accept is critical. The probability of accepting a bad move is usually $\exp(-\Delta C / T)$, where ΔC is the difference between the current and the new state's cost functions, and T is the temperature, a parameter that allows more or less bad moves to be accepted. Whenever the algorithm finds a bad move, it calculates the different between the current and new state. The algorithm then randomly determines whether to accept this bad move, and it is more likely to accept moves causing small increases in the cost function than big increases in the cost function (that is, the worse the move, the less like it is to be accepted). Also, over time it gets pickier, accepting less and less bad moves. This is done by lowering the temperature parameter T. At the beginning of the annealing the algorithm accepts many bad moves, and randomly wanders around the search space. Since it always accepts good moves, it tends to stay in the portion of the search space where better states are found, but the large amount of bad moves accepted keep it from getting stuck in any one place. As time goes on, T is decreased, and the algorithm accepts less and less bad moves. As this happens, it gets harder for the algorithm to wander away from the areas in the cost function where the better states are found. Thus, it is stuck in one region of the state space based on very coarse-grain measures of goodness, and it begins to stay in parts of this region where better states are found. The algorithm continues to accept less and less bad moves, until eventually it accepts only good moves. At this point, the algorithm is zeroing in on a local minima, though this minima tends to be much better than the average local minima in the search space, since the algorithm has slowly gravitated to areas in the search space where better states are found. In this way, simulated annealing can find much better results than greedy approaches in complex search spaces.

By applying simulated annealing to a placement problem, the complex relationships between the functions in the mapping can be considered. The algorithm will slowly optimize the state, coming up with a good final placement. Note that simulated annealing can be quite time-consuming. This is because the algorithm must be allowed to accept many good and bad moves at each acceptance level, so that the algorithm can explore much of the search space. Thus, multiple hour annealing runs are not exceptional.

Routing

Routing for FPGAs is the process of deciding exactly which routing resources will be used to carry signals from where they are generated (the source) to where they are used (the destinations). Unlike many other technologies, FPGAs have prefabricated routing resources. Thus, instead of trying to limit the size of routing channels (the goal in standard cell routing), an FPGA router must work within the framework of the architecture's resources. Thus, the router must consider the congestion of signals in a channel, making sure than no more routes are made through a region than there are resources to support them. Otherwise, if too many resources are required the routing fails, while in other technologies the region could just be enlarged.

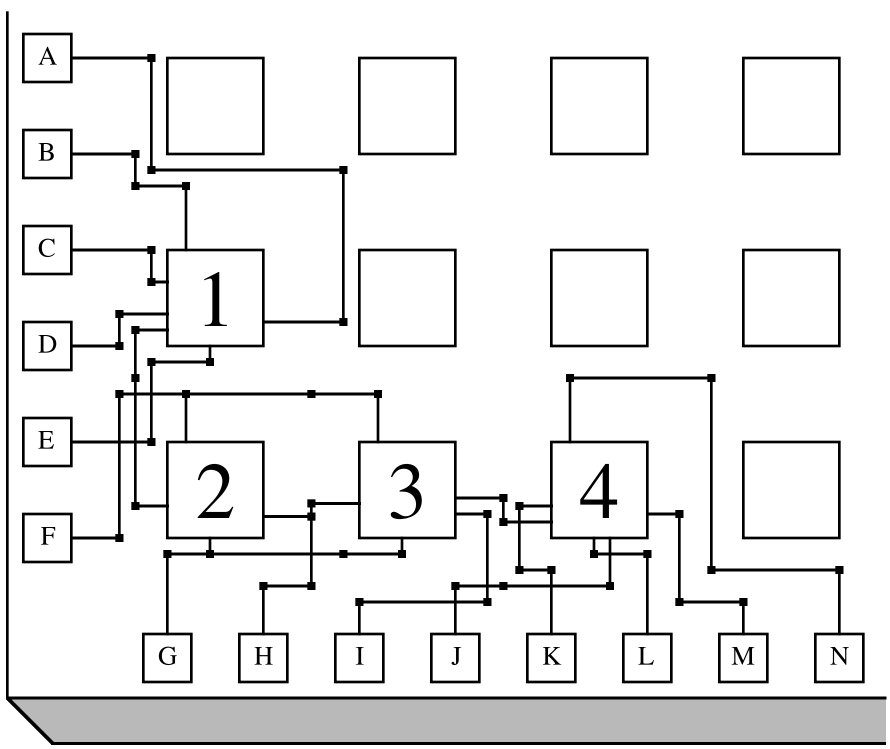


Figure 26. Routing of the placement from Figure 24. The small black squares are the configuration points used for this mapping.

An example of the routing of the placement from Figure 24 is shown in Figure 26. The router must decide which inputs and outputs of the logic blocks to connect to, which channels to route through, and how to connect through the switchboxes in the architecture. It is allowed to choose which terminal on the logic block to connect to because the logic is implemented in LUTs, and all inputs to a LUT are equivalent (assuming the programming of the LUT is modified accordingly). In deciding which channels and wires to use, and how to connect through the switchboxes, the router must ensure that there are enough resources to carry the signal in the chosen routing regions, as well as leaving enough resources to route the other signals in the system. One algorithm for performing this routing is presented in [Brown92b]. Here, the algorithm is divided into a global and a detailed router. The global router picks which routing regions the signal will move through. Thus, it will select the routing channels used to carry a signal, as well as the switchboxes it will move through. In this process, it takes care not to assign more signals to a channel than there are wires. However, the global router does not decide which specific wires to use to carry the signal. The detailed router handles this problem, making sure that it finds a connected series of wires, in the channels and switchboxes chosen by the global router, that connects from the source to all the destinations. Both algorithms worry both about congestion-avoidance, making sure that all signals can be successfully routed, as well as minimizing wirelength and capacitance on the path, attempting to optimize the performance of the circuit. By running both algorithms together, a complete routing solution can be created.

Software summary

Tools are available that automatically map a structural circuit into FPGA logic. This involves several steps. First, technology mapping restructures the logic to fit the logic blocks of the FPGA, attempting to minimize the number of blocks required. Then, placement assigns these functions to locations in the FPGA, attempting to minimize the routing required. Finally, routing determines which specific resources in the FPGA will carry the signals from source to destinations, simultaneously attempting to route the largest number of signals, and achieve the fastest implementation. Once routing is done, there is a complete description of how the circuit should be mapped to the FPGA. A simple translation takes this description and creates a programming file that, when downloaded to the FPGA, implements the desired functionality.

As discussed, mapping to a single FPGA is a highly automated process. The user is only required to specify the desired functionality, and the software automatically creates a realization. This thesis concerns systems of FPGAs used to implement circuits. While many of the issues are different for the multi-chip case than for the single chip implementations discussed here, single-chip mapping tools are still important. Specifically, several software steps, applied in series, reduce a multi-FPGA mapping into several single-FPGA mappings. Then, single-chip mapping tools are applied to the system's chips. In Chapter 9 we discuss the algorithms necessary to convert a multi-FPGA mapping into several single-FPGA mappings.

Chapter 3. Multi-FPGA System Applications

In Chapter 2 we discussed many different implementation technologies for digital logic, considering primarily how these various mediums support standard circuit designs. However, with the development of FPGAs, there are now opportunities for implementing quite different systems than were possible with other technologies. In this chapter we will discuss many of these new opportunities, especially those of multi-FPGA systems.

When FPGAs were first introduced, they were primarily considered to be just another form of gate array. While they had lower speed and capacity, and had a higher unit cost, they did not have the large startup costs and lead times necessary for MPGAs. Thus, they could be used for implementing random logic and glue logic in small volume systems with non-aggressive speed and capacity demands. If the capacity of a single FPGA was not enough to handle the desired functionality, multiple FPGAs could be included on the board, distributing the functionality between these chips.

FPGAs are more than just slow, small gate arrays. The critical feature of (SRAM-based) FPGAs is their in-circuit reprogrammability. Since their programming can be changed quickly, without any rewiring or refabrication, they can be used in a much more flexible manner than standard gate arrays. One example of this is multi-mode hardware. For example, when designing a digital tape recorder with error-correcting codes, one way to implement such a system is to have separate code generation and code checking hardware built into the tape machine. However, there is no reason to have both of these functions available simultaneously, since when reading from the tape there is no need to generate new codes, and when writing to the tape the code checking hardware will be idle. Thus, we can have an FPGA in the system, and have two different configurations stored in ROM, one for reading and one for writing. In this way, a single piece of hardware handles multiple different functionalities. There have been several multi-configuration systems built from FPGAs, including the just mentioned tape machine, generic printer interface cards with configurations for specific printers, pivoting monitors with landscape and portrait configurations, as well as others [Xilinx92, Fawcett94].

While the previous uses of FPGAs still treat these chips purely as methods for implementing digital logic, there are other applications where this is not the case. A system of FPGAs can be seen as a computing substrate with somewhat different properties than standard microprocessors. The reprogrammability of the FPGAs allows one to download algorithms onto the FPGAs, and change these algorithms just as general-purpose computers can change programs. This computing substrate is different from standard processors, in that it provides a huge amount of fine-grain parallelism, since there are many logic blocks on the chips, and the instructions are quite simple, on the order of a single five input, one output function. Also, while the instruction-stream of a microprocessor can be arbitrarily complex, with the function computed by the logic changing on a cycle by cycle basis, the programming of an FPGA is in general held constant throughout the execution of the mapping (exceptions to this include techniques of partial reconfigurability [Lysaght94, Hadley95, Jones95, Wirthlin95], which change a portion of the logic while the rest of the system continues operating). Thus, to achieve a variety of different functions in a mapping, a microprocessor does this temporally, with different functions executed during different cycles, while an FPGA-based computing machine achieves variety spatially, having different logic elements compute different functions. This means that microprocessors are superior for complex control flow and irregular computations, while an FPGA-based computing machine can be superior for data-parallel applications, where a huge amount of data must be acted on in a very similar manner. Note that there is work being done on trying to bridge this gap, and develop FPGA-processor hybrids that can achieve both spatial and limited temporal function variation [Ling93, Bolotski94, Maliniak94].

There have been several computing applications where a multi-FPGA system has delivered the highest performance implementation. An early example is genetic string matching on the Splash machine [Gokhale90]. Here, a linear array of Xilinx 3000 series FPGAs was used to implement a systolic algorithm to determine the “edit distance” between two strings. The edit distance is the minimum number of insertions and deletions necessary to transform one string into another, so the strings “flea” and “fleet” would have an edit distance of 3 (delete “a” and insert “et” to go from “flea” to “fleet”). As shown in [Lopresti91], a dynamic-programming solution to this problem can be implemented in the Splash system as a linear systolic circuit, with the strings to be compared flowing in opposite directions through the linear array. Processing can occur throughout the linear array simultaneously, with only local communication necessary, producing a huge amount of fine-grain parallelism. This is exactly the type of computation that maps well onto a multi-FPGA system. The Splash implementation was able to offer an extremely high performance solution for this application, achieving performance approximately 200 times faster than supercomputer implementations. There have been many other applications where a multi-FPGA system has offered the highest performance solution, including: mathematics applications such as long multiplication [Bertin89, Vuillemin95], modular multiplication [Cuccaro93], and RSA cryptography [Vuillemin95]; physics applications such as real-time pattern recognition in high-energy physics [Högl95], Monte Carlo algorithms for statistical physics [Monaghan93, Cowen94], second-level triggers for particle colliders [Moll95], and Heat and Laplace equation solvers [Vuillemin95]; general algorithms such as Monte Carlo yield modeling [Howard94b], genetic optimization algorithms [Scott95], stereo matching for stereo vision [Vuillemin95], hidden Markov Modeling for speech recognition [Schmit95], and genetic database searches [Lopresti91, Hoang93, Lemoine95].

One of the most successful uses for FPGA-based computation is in ASIC logic emulation. The problem is that the designers of a custom ASIC need to make sure that the circuit they designed correctly implements the desired functionality. Software simulation can perform these checks, but does so quite slowly. In logic emulation, the circuit to be tested is instead mapped onto a multi-FPGA system, yielding a solution several orders of magnitude faster than software simulation. A more detailed discussion of logic validation options, and how logic emulation fits into this process, can be found in Chapter 4.

There are many different types of FPGA-based systems. In this thesis we concentrate on multi-FPGA systems, namely those systems that contain more than one FPGA, and are constrained by the inter-chip connection topology. Before we discuss what kinds of systems fit this multi-FPGA system model, we first need to detail some of the systems that do not fall into this category. These are "topology-less" systems, systems where there are no specific, predefined connections between the FPGAs in the system. Topology-less systems are single-chip and custom multi-chip systems, which includes both the standard gate array applications of FPGAs, as well as the multi-mode hardware applications. Here the user is not constrained by any specific board topology, since the printed circuit board will be designed and fabricated after the mappings to the FPGAs have been determined, and these mapping will dictate the board interconnection pattern. Thus, if there will be more than a single FPGA in the system, the connections between these FPGAs are designed in response to the needs of the specific application. This tends to make the inter-FPGA routing problem much less significant than in the multi-FPGA domain, greatly altering the solutions adopted.

While topology-less systems have been the subject of significant amounts of work [Kuznar93, Woo93, Kuznar94a, Chan95, Huang95], it is not clear how important this domain really is. This is because many of these systems have their mapping finalized after the circuit board is sent for fabrication. In some cases, the need to get the system completed is significant enough that the board design cannot wait for the FPGA's logic to be completely specified. In others, even though a mapping may have been generated for the FPGAs before the board was designed, new functionality or bug fixes may mandate revisions to the FPGA mappings. In either case, the final mapping to the FPGAs must fit within the fixed inter-FPGA topology, and the requirements of a fixed topology move these systems into the multi-FPGA system domain. Note that some multi-FPGA systems are actually treated as a set of single-chip mappings, with each FPGA having a specific role independent of the other chips in the system, and the communication protocol between the chips is fixed. These systems would be considered topology-less, since they also avoid most multi-chip implementation issues.

As we just discussed, the primary characteristic of a multi-FPGA system is that it includes several FPGAs connected in a constrained routing topology. Within this multi-FPGA system domain, the most significant attribute is whether the system will be the target of automatic mapping software, or if it will be used primarily for hand-generated mappings. This difference can have a significant impact on the structure of the multi-FPGA system, since features that automatic-mapping software can best optimize for are often quite different from those that are appropriate for a human designer.

The ideal answer is to have all mappings to a multi-FPGA system generated by automatic-mapping software, since the hand-mapping process is often quite complex and time-consuming. Thus, if automatic-mapping software could deliver high-quality mappings quickly, there would be no reason for a human to waste time on this process. However, the problem is that current automatic-mapping software usually delivers much lower quality mappings than a human can achieve, especially on simple or highly-structured circuits. Thus, for many domains, automatic-mapping software simply produces unacceptable results, and the user is either forced to map the circuit by hand, or not use a multi-FPGA system at all.

To some extent, the applications for multi-FPGA systems can be seen as a spectrum, with highly structured (and thus easily hand-mapped) circuits at one end, and highly irregular (and thus impossible to hand-map) on the other. Currently, automatic-mapping software is only able to support the highly irregular mappings, not because the tools are somehow able to provide particularly good results for these circuits, but because the circuits are too complex for a human to do a reasonable job of hand-mapping. Thus, logic emulation tasks, where quite complex circuits are mapped to a multi-FPGA system, rely completely on automatic-mapping software. However, currently the more highly-structured applications are almost exclusively the domain of hand-mappings. One of the major research areas in multi-FPGA systems is developing automatic-mapping software that can produce higher-quality mappings. As this happens, the portion of the spectrum handled by automatic-mapping software will grow, taking over some of the mapping tasks currently the exclusive domain of the hand-mapping approach. This is a welcome development not only from the standpoint of lowering the amount of effort needed to use a multi-FPGA systems, but also because it increases the application domains of multi-FPGA systems. This is because there are many circuits that could perform well on a multi-FPGA system, but the unacceptable quality delivered by current automatic-mapping software, as well as the excessive effort necessary to hand-map these applications, forces these users to seek other solutions. Thus, while automatically-mapping circuits to a multi-FPGA system currently has a somewhat restricted application domain, as the automatic-mapping software improves this domain will become increasingly important. In this thesis we concentrate on hardware and software systems for multi-FPGA systems designed to support automatically-mapped circuits.

This distinction between multi-FPGA systems for hand-mappings and those for automatic-mappings is quite significant, and affects the entire implementation effort. When mapping by hand, the algorithm to be mapped is carefully chosen to fit the requirements of the multi-FPGA system, and may be quite different from the approach taken when the target is a microprocessor or other implementation medium. Most or all steps in the mapping are done by the user, who exerts significant effort to create the best possible mapping. The multi-FPGA system itself is designed to be simple and regular, with an emphasis on local communication, since this matches the structure of the circuits that can reasonably be hand-mapped. While this is how many of the high performance mappings have been developed for multi-FPGA systems, it limits the areas in which a multi-FPGA system can be used. In many situations, potential beneficiaries of a multi-FPGA system's performance do not have the time or the expertise to create a hand-mapping of their application, or the application may be too complex for anyone to reasonably map by hand. Thus, they will never adopt this strategy. However, hand-mapping may be necessary in some applications to achieve reasonable speedups over a microprocessor implementation, since current automatic-mapping software cannot develop truly high quality mappings.

In multi-FPGA systems designed for automatically-mapped circuits, the burden of taking a circuit from specification to implementation is handled by software (though variants that harness software for only lower-level portions of the mapping, or as assistants to a hand-mapping approach, are possible). Currently this is a fairly restricted class of systems, since the automatic mapping software often does not deliver nearly the mapping quality of hand-mappings. However, as the tools get better we should see more applications using automatic mapping software. Also, if multi-FPGA systems are to ever become commonplace, they will need to be mapped to automatically, because only a small portion of the possible users are willing and able to hand-map their circuits and algorithms.

The distinction between multi-FPGA systems for hand-mappings and systems for automatic mappings is more than a software issue. The design of the multi-FPGA system itself may be altered by how mappings are generated. For example, a system created for hand-mappings will usually have a simple topology, so that it is easier for the designer to use the system. However, complex and irregular features that automatic-mapping software can optimize for could be included in a system targeted for automatic mappings, while these features would only complicate the problem for a designer creating a mapping by hand. The pin permutations of Chapter 7 are a good example of this. While they can improve the quality of automatically generated mappings, they are probably too irregular to be used for hand-mappings.

In this thesis, we will be concerned primarily with multi-FPGA systems designed to support automatically-mapped circuits, though some of the results will also be applicable to hand-mappings as well. This bias is important, because many of the concerns of multi-FPGA systems with constrained topologies are quite different from topology-less systems. As indicated earlier, when a mapping does not need to fit into a predefined topology, the inter-FPGA routing problem is much less of a concern. Also, mapping turnaround time is a much greater concern in the systems we consider, since a fixed topology can be ready to use instantly, while custom multi-FPGA systems will need to wait for boards to be fabricated.

In the next chapter we will discussion logic validation, and how multi-FPGA systems can form an integral part of a complete validation strategy. Chapter 5 will then provide an overview of current multi-FPGA system hardware. Multi-FPGA system software is covered in Chapter 9.

Chapter 4. Logic Validation

Logic emulation is one of the most promising application domains for multi-FPGA systems. In this chapter we explore logic emulation in greater depth, and show how emulation fits into a complete logic validation methodology.

Logic validation is the process of determining whether a realization of a logic circuit exhibits the desired features. While this statement sounds simple, logic validation is in fact a fairly complex procedure, with many different possible approaches. The major issue is in defining what the "desired features" are for a given piece of logic. There are many different answers. One possibility is that the behavior of an implementation matches the behavior of some predefined specification. However, whether the specification actually matches the users' desires is not always clear, especially if the specification is quite complex. We could define "desired features" as simply whatever the users desire, but the users may not know exactly what they want. Thus, giving the users a system they can examine and test to help define their desires can also be an important part of logic validation. Some validation goals can be quite easy to define. These include the requirements that the system achieve a certain cycle time (performance), or that it avoid certain erroneous behavior, such as deadlock or other failure modes.

Another important facet of the logic validation issue is the definition of what is a "realization of a logic circuit". Logic validation will often be performed on a version of the logic somewhat abstracted from the final, physical realization. Early in the design cycle, all that may be available is a high-level specification of the circuit functionality. The designers may want to test this specification before implementing it. Catching errors in the specification can allow much simpler fixes, with much less wasted effort, than if the errors are allowed to propagate all the way to the completed implementation. In general, errors corrected earlier in the design cycle are much easier to fix than errors fixed later in the process, since as the design progresses the implementation becomes more detailed, and thus more complex. Even when the implementation is completed, there may still be the need to test an abstracted version of the circuit. Some logic validation techniques require that the circuit be simplified somewhat, either because the technique cannot handle all the details of the system, or because adequate performance can only be achieved by simplifying the logic. For example, software simulators can simulate most of the electrical behavior of a circuit, modeling the voltage levels on all signals in the system, but a much faster simulator can be built for pure functional testing, where signal voltage values are abstracted to purely "true" and "false" values.

As we will see in the following sections, there are a number of different methods for performing logic validation. They each have different strengths and goals, with different models of how user desires for the logic are expressed, and operate on different realizations of the circuit's functionality. As we will discuss in the final section of this chapter, the proper method for logic validation is not a single technique, but instead an integrated system of techniques that takes advantage of the differing strengths of each of these approaches.

Simulation

Probably the most powerful and most widely used logic validation method is software simulation. Here, the logic realization is executed in software. The software models whatever portions of the electrical and functional behavior of the elements in the logic realization the user desires, and the resulting behavior can be observed. For example, in a unit-delay functional simulator (i.e., a simulator where all electrical effects are ignored, the circuit is represented as gates, and all gates take exactly one time unit to react to new inputs), the software examines the inputs to all gates, and computes the appropriate values for each gate's output(s). By assigning different values to the inputs of the system, the user can see how the system will react to different situations. By applying sequences of inputs, the sequential behavior of the system can also be examined. Simulators need not only be unit-delay functional simulators, but can also incorporate more complex models of circuit timing and electrical behavior. For example, a simulator can include a detailed model of the capacitances on a signal line, and the drive strengths of the transistors generating this signal, to compute the predicted voltage levels on the wire at any given time.

The advantages of software simulation are that they have good observability and controllability, and are easy to use. Since each signal value must be maintained by the software to process the simulation, the software can tell the user the value of a wire at any time in the simulation, and can follow the changing value over time. In this way, the user gains a great deal of information about the system's behavior, information that might not be available in the final hardware realization. Also, since the values on the wires are simply numbers stored inside the computer, these values can quite easily be manipulated by the user. For example, the user may want to understand the behavior of the system once it is in some specific configuration. In the final logic implementation, it may be quite difficult to force the system into this state. The user of a software simulator can simply instruct the simulator to put the system into the desired state, and then observe the subsequent behavior. The user can also experiment with the system, evaluating different configurations and designs. This allows the designer to efficiently explore the design space, leading to better implementations. In terms of ease of use, software simulators can be constructed to accept any reasonable circuit description. Thus, simulators can execute high-level specifications, specifications that are quite distant from their eventual hardware implementations. In this way, users can begin testing their designs very early in the design cycle, thus saving significant effort by fixing bugs before they propagate to more detailed (and more time-consuming to modify) implementations. Also, portions of a given specification can be tested in isolation, making it easier to understand that subcircuit's behavior.

The flexibility of software-simulation is also its biggest weakness. It takes a significant amount of instructions for a standard processor to process the logic elements in the realization to be tested. This means that the software simulation is going to operate significantly slower than the real circuit. Thus, a second of circuit operation can take a software simulator minutes, hours, or more to process. Work has been done on providing specialized simulation-accelerators to speed up software simulation [Blank84, Zycad94b, Zycad94c]. Also, simulators can operate on abstracted versions of the logic, ignoring some details of the implementation such as detailed timing and voltage levels, thus providing faster simulation. While software accelerators and simulators that work on abstracted versions of the logic can give boosts in performance, it still takes a software simulation significantly longer to execute than a true hardware implementation of the circuit. This tends to relegate software simulation to those situations where either the time penalty is not severe, or the flexibility and observability of software simulation is required. These include testing subcircuits of complex designs and early testing of specifications, where short executions of the software can discover problems in the realizations, as well as executions of trouble situations where bugs have been discovered by other methods, where software simulation can help the designer understand the failure.

Prototyping

Prototyping is the process of creating a physical implementation of the circuit under validation. For example, one method of testing a circuit is to actually construct it, and then test this implementation under normal working conditions. Since one now has the real circuit, results obtained from testing it will be the most accurate, since there is no abstraction, modeling, or other hidden assumptions in the process. Thus exact performance, suitability, reliability, and other evaluations can be made. Not only can more accurate tests be performed, but the prototype can also be given to users for evaluation. This allows the users to determine whether the system will actually handle their needs and demands, something that is difficult to do with just the initial specification. Also, prototypes usually operate at or near target system speeds, allowing tests to be completed much faster than software simulation. Note that a prototype of the system can be made differently than the eventual production system. For example, techniques of breadboarding and wire-wrap allow a prototype to be created quickly. These techniques are simply methods to wire together standard components to implement the system. As such, they allow a prototype to easily be constructed, but the implementation technology (chips scattered across generic boards with individual wire connections) is different from the final product. This introduces some discrepancies between the prototype and the eventual system, introducing some inaccuracy in the modeling. One specific example is in performance, where wire-wrap and breadboard prototypes cannot handle cutting-edge performance, and thus such systems will need to be slowed down. Also, both wire-wrap and breadboarding are somewhat unreliable. Thus, the debugging process for these technologies is more complex, since the implementation technology must always be considered a potential culprit in any anomalous behavior.

While prototyping has the benefits of a high degree of accuracy and high-speed testing, it also suffers from some significant problems. First, to build a prototype one must have the system to be prototyped completely designed. Because of this in general one cannot prototype a specification or behavioral description. As a result prototyping is only useable fairly late in the design process. Also, constructing the prototype can be a costly process, both in terms of materials (since a complete implementation of a circuit may involve circuit board or ASIC fabrication) and in construction time. These prototypes are usually difficult to alter, so to avoid the costs of multiple fabrications prototyping is again relegated to very late in the design process, when (hopefully) errors will be few. Finally, it is hard to test a prototype, since there is usually little or no access to the internal state of the system, and it is difficult to control system behavior. Specifically, while a software simulator can display and alter the value of any signal in the system, this is not possible in most prototypes, making it much more difficult to detect and isolate erroneous behavior.

Even though prototypes are costly to create and difficult to use, they are the only truly accurate method of logic validation. All other methods of logic validation make abstractions and assumptions about the electrical and behavioral characteristics of actual circuits, making the results somewhat suspect. Thus, testing a prototype is an essential portion of any logic validation process, though it is usually relegated to quite late in the design process. Detection and correction of most bugs in the system is handled by other validation methods.

Emulation

Logic emulation is a method of logic validation that shares many of the advantages (and disadvantages) of both prototyping and software simulation. Like a prototype, the circuit to be evaluated is implemented in hardware so that it can achieve high performance test cycles. However, like software simulation, the emulation can easily be observed and altered to help isolate bugs. Logic emulation takes a gate-level description of a logic circuit and maps it onto a multi-FPGA system. This multi-FPGA system is a prefabricated, reprogrammable compute engine that can be configured to implement the desired circuit functionality in a matter of seconds. However, to transform the circuit description into a mapping suitable for this multi-FPGA system can take multiple hours to complete. This mapping process is usually completely automated by the emulator's system software. Once the circuit is mapped to the multi-FPGA system, the emulator provides a complete, functional implementation of the circuit that can evaluate millions of circuit cycles per second. This is orders of magnitude faster than even simulation-accelerators, since the multi-FPGA system can implement the complete circuit functionality in parallel, while accelerators simply provide one or more sequential logic evaluation processors.

Emulators provide a middle-ground between software simulation and prototyping. Compared to software simulation, an emulation executes much faster than a simulation. However, it can take a significant amount of time to map a circuit onto the emulator, and it is more difficult to observe and modify the behavior of the circuit. Thus, software simulation is a better choice for testing small subcircuits or small numbers of complete circuit cycles, where the software's flexibility and ease of use outweighs the performance penalties. Compared to a prototype, an emulation is much easier and faster to create, and it has much greater observability, controllability, and modifiability than a prototype. However, the emulation cannot run at the same speed as the target system. Thus, the emulator is a much better choice for providing a huge number of test cycles than a prototype when one expects to find bugs in the system, but it is no replacement for final checkout of the system via a prototype. For circuits that will execute software programs, an emulator can be used to debug this software much earlier in the design process than a physical prototype. This is because an emulation can be created from a high-level specification of the circuit, while prototyping must wait until the complete circuit has been designed. Simulation in general cannot be used for software development, since it is much too slow to execute enough cycles of the software. Also, just like a prototype, an emulation can be given to the end-user so that the circuit can be evaluated before the design is completed. In this way, the user can get a much better feeling for whether or not the design will fulfill the user's needs, something that is difficult with just a written specification. The emulation can be inserted into the target environment (as long as some method for reducing the performance demands on the system can be provided, such as those described in Chapter 8), and the system can be evaluated in a more realistic setting. This helps both to debug the circuit, and also to test the circuit interfaces and environment. For example, often a custom ASIC and the circuit board that will contain it will be developed simultaneously. An emulation of the ASIC can be inserted into this circuit board prototype, testing both the ASIC functionality as well as the board design.

One limitation of emulation is that it retains only the functional behavior of the circuit, which means that validation of the performance and timing features of a circuit cannot be performed on a logic emulator. Once a prototype is constructed, both logic emulation and software simulation are still valuable tools. When an error is found in a physical prototype, it can be difficult to isolate the exact cause in the circuit. An emulator can be used to reproduce the failure, since it can execute nearly as many cycles as the prototype, and the emulator's observability can be used to isolate the failure. Then, detailed testing can be performed by software simulation. Thus, logic emulation plays a complementary role to both software simulation and prototyping in logic validation.

Formal Verification

Formal verification [Yoeli90] includes techniques that can determine whether two logic realizations are identical, as well as techniques to decide whether a given logic realization could ever exhibit a certain behavior. For example, the designer of a circuit may work from a high-level specification to create an implementation of the circuit. Formal verification techniques can examine both of these realizations of the logic and determine whether they have exactly the same behavior. Note however that most formal verification techniques are restricted to functional validation (that is, they ignore the detailed timing and electrical characteristics of circuits). Thus, an actual implementation of the circuit could not be tested via formal verification, and would have to be abstracted into a functional description of the implementation. This abstraction process could be somewhat inexact, adding the potential for errors. However, ignoring this issue, formal verification techniques can tell designers whether or not they have successfully implemented a given specification. This is not the same as determining whether the implementation is actually what the user desires, since the specification may not have reflected these properly. However, if the two realizations differ, it is quite likely that there is an error in the implementation. Also, by determining that an implementation is the same as a specification, a user can then examine the specification to determine whether the implementation meets their needs. Since specifications will usually be much simpler than their implementations, this can simplify the validation process. Note that there is another approach to this process of ensuring that specification and implementation are identical: automatic synthesis methods that ensure "correct-by-construction" implementations. These systems translate a specification into an implementation by applying a set of transformations that have been formally proven correct. That is, the creator of the transformational system has examined each transformation, and proven that the circuit created by applying the transformation is always identical to the circuit to which the transformation is applied. Thus, if an implementation is derived from a specification via a "correct-by-construction" system, the designer already knows that the implementation is identical to the specification, and thus it is not necessary to apply formal verification to determine this.

There is a second portion of formal verification: the determination of whether a given realization can ever exhibit certain behaviors. For example, the designer of a resource controller may need to insure that two requesters can never access the resource at the same time (i.e., ensure mutual exclusion). The designer might use software simulation, as well as the execution of a completed prototype, to demonstrate that this requirement is met. However, in most complex systems it is impossible to exhaustively test the circuit, and thus all that the simulation and/or prototype execution can prove is that the requirement is not violated *in the situations that have been tested*. For many applications, especially safety-critical applications, this guarantee is not sufficient. Formal verification techniques hold perhaps the only solution to this problem. Formal verification methods examine a given realization of a circuit and efficiently determine either that the realization can never violate the constraint, or provide a counter-example where the circuit does violate the constraint. The counter-example is an allowable excitation of the circuit (i.e., a series of input signals that are not prohibited by the input signaling protocols) that will cause the constraint to be violated. Both of these results are valuable. In the case where the constraint is proven to be upheld, we have much greater confidence in the system than any other technique can achieve (note that this result may not be an absolute proof, since the representation of the circuit tested will at best be an abstracted version of the implementation, and this abstraction process may introduce errors). In the case where a counter-example is provided, this counter-example serves as a guide to understanding the failure mode and determining a fix.

While formal verification and "correct-by-construction" techniques offer the best hope for creating completely correct circuits, these techniques are not sufficient by themselves to handle the complete logic validation task. There are three problems with these techniques that somewhat restrict their utility. First, when applying these techniques to generate an implementation, we assume that the specification is in fact what the user desires. However, user needs and requirements are often ill-defined, and any attempt to codify them in a formal specification will inevitably introduce errors, misunderstandings, and unexpected results. Many of these problems are due to the fact that before users have actually used a system they may not know exactly what they really want, and early attempts to express these needs may be naïve. Second, these formal techniques in general require abstract mathematical models of the actual circuit behavior. These abstractions will be somewhat inexact, and there is a potential for errors both in translating a "correct-by-construction" realization into a real circuit implementation as well as in abstracting a circuit implementation into a form that formal verification techniques can examine. Finally, while formal verification techniques can efficiently handle some circuits, real circuits can be quite complex, with huge state spaces, greatly increasing the task of formal verification. These techniques, at least as developed so far, are not capable of handling complete, complex systems. Thus, formal verification techniques can be applied only to smaller subcircuits. However, even if individual subcircuits are verified to be correct, when they are composed together there is still the potential for failures due to their interactions.

Complete Logic Validation

As we have just discussed, there are numerous methods for performing logic validation, each with its own strengths and weaknesses. Current practice in general involves only software simulation during early design phases, followed by the construction and testing of a hardware prototype. Unfortunately, this ignores the fact that only formal verification can discover some kinds of unlikely, but still possible, failure modes of the circuit. Also, logic emulation can greatly improve the completeness of pre-prototype testing.

In an ideal validation environment - one where some of the current shortcomings of formal verification and logic emulation have been overcome - the methodology would take advantage of the strengths of each approach, while using multiple approaches to avoid their individual deficiencies. In early design phases, the user would be creating and modifying a specification of the circuit to be designed. Software simulation would handle most of the validation needs in this step, though formal verification techniques could be applied to discover failure modes in the specification. As this specification is subsequently refined into a complete implementation, correct-by-construction logic synthesis systems or formal verification techniques could be applied to avoid adding errors during this refinement process. At some point, software simulation will have uncovered most of the simple bugs, and further testing would only discover new failures after a large number of test cycles. At this point, emulation would become a better testing method than software simulation simply on performance grounds. Emulation may also be necessary earlier in the process to provide a platform for concurrent software development, and to provide a demonstration vehicle for end-user evaluation. Once failures are discovered via emulation, software simulation could help isolate the errors and develop the bug fixes. Eventually, the designers would have confidence that the current design is relatively bug-free. At this point, the time and expense of prototype construction are justified, and a prototype would be built. This would probably be done via a complete fabrication of the circuit, avoiding the unpredictable failures and inaccuracies of wire-wrap or breadboarding. The prototype could then be completely tested under true operating conditions. When failures occur, both emulation and software simulation could be brought to bear to isolate and correct the errors.

Under this ideal validation process several interesting things occur. Formal verification is applied to uncover failures that no other method would ever uncover. With the complexity of current systems, it is impossible to exhaustively test the circuit, and some unlikely situations would not be examined. Without formal verification, errors that occur only in these unlikely situations would only be discovered after the user is bitten by these bugs. Emulation also provides improved capabilities to the validation process. Since emulation could be applied much earlier in the design process than prototyping, many more test cycles could be completed in the early design phases. This would uncover failures earlier in the process, making the resulting debugging and correction easier and faster. Also, emulation provides a software execution platform much sooner than prototyping, allowing software development to begin much earlier in the process. It provides an early, working prototype of the system, which could be used for interface testing (such as when an emulated ASIC is inserted into the circuit board designed to contain it so that the board and board-chip interactions could be tested), as well as for early end-user evaluation. The latter is quite important, since the goal of logic validation is not just to determine whether the circuit meets the specification, but also make sure that the design is actually what the end-user really wants. Finally, emulation could delay the need for the construction of a physical prototype, saving much of the time and money that would otherwise be wasted on multiple prototype fabrications.

As we have argued, the best validation methodology is a combined methodology, taking advantage of the benefits of software simulation, emulation, prototyping, and formal verification. Software simulation helps the designers understand the design they are working on, and detect problems on a small scale. Emulation expands this capability to the complete system, giving the designers and end-users their earliest look at the complete system in operation. Prototyping provides the final sanity check, with no hidden errors due to false assumptions or incorrect abstractions of the true system behavior. Formal verification guards against the unlikely (yet still possible) error conditions, detecting problems that could be discovered via no other method. In this way, the best overall validation solution could be delivered.

In the next chapter we discuss current multi-FPGA systems, concentrating on their hardware structures. Then in Chapter 6 we present Springbok, a system for the logic emulation of board-level designs.

Chapter 5. Multi-FPGA System Hardware

In Chapter 3 we discussed the applications of multi-FPGA systems, and concentrated on logic emulation in Chapter 4. In this chapter we will explore some of the existing multi-FPGA systems themselves. There are a large number of systems that have been constructed, for many different purposes, with a wide range of structures. Note that this section is intended to illustrate only the types of systems possible, and is not meant to be an in-depth discussion of all the details of existing systems. Thus, some details of the topologies, as well as the amount of connectivity on links in the systems, have been omitted.

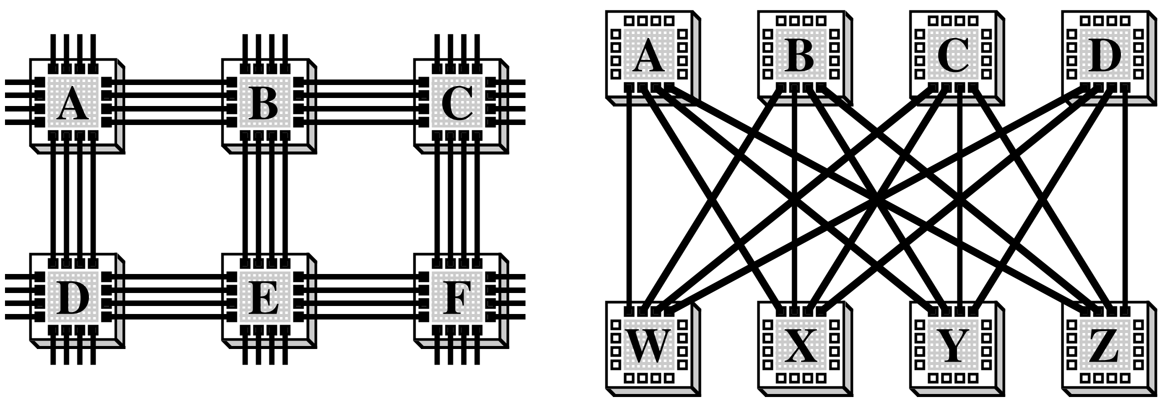


Figure 27. Mesh (left) and crossbar (right) topologies. In the crossbar, chips **A**-**D** are routing-only, while **W**-**Z** hold all the logic in the system.

The most important difference between multi-FPGA systems is in the topology chosen to interconnect the chips in the system. The most common topologies are mesh and crossbar (or bipartite graph) topologies. In a mesh, the chips in the system are connected in a nearest-neighbor pattern (Figure 27 left). These topologies have the advantage of simplicity, because of the purely local interconnection pattern, as well as easy expandability, since meshes can be grown by adding resources to the edge of the array. Numerous 2D mesh-based systems have been built [Kean92, Shaw93, Bergmann94, Blickle94, Tessier94, Yamada94], as well as 3D meshes [Sample92, Quénot94]. Linear arrays, which are essentially 1-dimensional meshes, have also been built [Gokhale90, Raimbault93, Monaghan94]. More details on mesh topologies can be found in Chapter 7.

Crossbar topologies separate the chips in the system into logic-bearing and routing-only (Figure 27 right). The logic-bearing FPGAs contain all the logic in the system, while the routing-only chips are used purely for inter-FPGA routing. Routing-only chips are connected only to logic-bearing FPGAs, and (usually) have exactly the same number of connections to all logic-bearing FPGAs. Logic-bearing FPGAs are connected only to routing-only FPGAs. The idea behind this topology is that to route between any set of FPGAs requires routing through only one extra chip, and that chip is any one of the routing-only chips. Because of the symmetry of the system, all routing-only chips can handle this role equally well. This gives much more predictable performance, since regardless of the locations of the source and destinations, the delay is the same. In a topology like a mesh, where it might be necessary to route through several intermediate chips, there is a high variance in the delay of inter-FPGA routes. There are two negative features of this topology. First, crossbar topologies are not expandable, since all routing-only chips need to connect to all logic-bearing FPGAs, and thus the system is constrained to a specific size once the connections to any specific routing-only chip are determined. Second, the topology potentially wastes resources, since the routing-only chips are used purely for routing, while a mesh can use all of its chips for logic and routing. However, since the bottleneck in multi-FPGA systems is the inter-chip routing, this waste of resources may be more than made up for by greater logic utilization in the logic-bearing chips. Also, some of the cost of the wasted resources can be avoided by using less expensive devices for the routing-only chips. Possibilities include FPICs, crossbars, or cheaper FPGAs (either because of older technology or lower logic capacity). Several pure crossbar topologies have been constructed [Chan92, Ferrucci94, Kadi94, Weiss94].

A middle-ground between the two topologies, which combines the expandability of meshes and the simpler routing of crossbars, is hierarchical crossbars [Varghese93]. As shown in Figure 28, crossbars can be stacked together hierarchically, building up multiple levels of routing chips. There are two simple crossbars in the system, one consisting of routing-only chips **E-H** and logic-bearing FPGAs **M-P**, and a second one consisting of routing-only chips **I-L** and logic-bearing FPGAs **Q-T**. Routing chips **E-L** will be called the first-level crossbars, since they connect directly to the logic-bearing FPGAs. To build the hierarchy of crossbars, the simple crossbars in the system can be thought of as logic-bearing chips in an even larger crossbar. That is, a new crossbar is built with routing-only chips and logic-bearing elements, but in this crossbar the logic-bearing elements are complete, simple crossbars. Note that the connections within this higher-level crossbar go to the routing-only chips in the simple crossbars, so first-level and second-level routing-only chips are connected together. This hierarchy can be continued, building up other crossbars with third-level and higher routing-only chips. In an N-level hierarchical crossbar, the chips are arranged as above, with routing-only chips at the *Ith* level connected to chips at the *(I+1)th* and *(I-1)th* level, where the *0th* level is the logic-bearing chips. Note that in contrast to the simple crossbar topology, in a hierarchical crossbar the logic-bearing FPGAs are not connected to all the routing-only chips (even those at the first-level). Full connectivity occurs at the top (*Nth*) level, where all *Nth*-level routing chips are connected to all *(N-1)th* level routing chips.

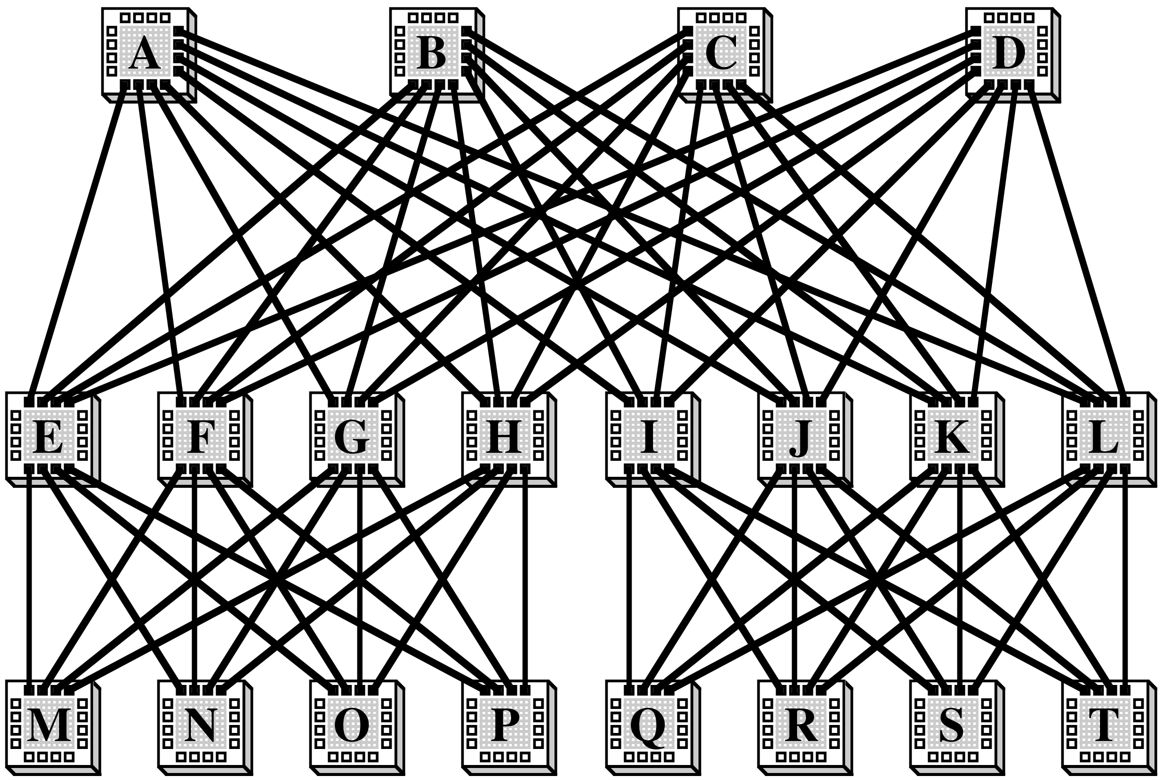


Figure 28. A hierarchy of crossbars. FPGAs **M-T** hold all the logic in the system. Chips **E-H** and **I-J** form two first-level crossbars, and chips **A-D** form a second-level crossbar.

Routing between two logic-bearing FPGAs in the system simply requires determining the level at which the source and destination share an ancestor, and then routing from the source up to one of these shared ancestors, and back down to the destination. The routing from the source to the shared ancestor requires routing through exactly one routing-only chip in the intervening levels, as does the routing from the ancestor to the destination. Because of the symmetry of the topology (and ignoring conflicts from other routes), any of the ancestors of the source (for the route up) or destination (for the route down) at a given level can be used to handle the routing, regardless of what other chips are part of the route.

As mentioned earlier, the advantage of a hierarchical crossbar topology is that it has much of the expandability of a mesh, yet has much of the simplified routing of a crossbar. Since levels of hierarchy can be added to a hierarchical crossbar, it can easily grow larger to handle bigger circuits. Levels of the hierarchy tend to map onto components of the system, such as having a complete first-level crossbar on a single board, a complete second-level crossbar contained in a cabinet, and a complete third-level crossbar formed by interconnecting cabinets [Butts91]. The routing simplicity, as shown above, demonstrates a large amount of flexibility in the routing paths, and a large amount of symmetry in the system. How easy it is to route between logic-bearing FPGAs is simple to determine, since if two logic-bearing chips are within the same first-level crossbar, then they are as easy to route between as any other pair of logic-bearing chips within that crossbar. Thus, when mapping onto the topology, the system tries to keep most of the communication between chips in the same first-level crossbar, and most of the rest of the communication between chips in the same second-level crossbar, and so on.

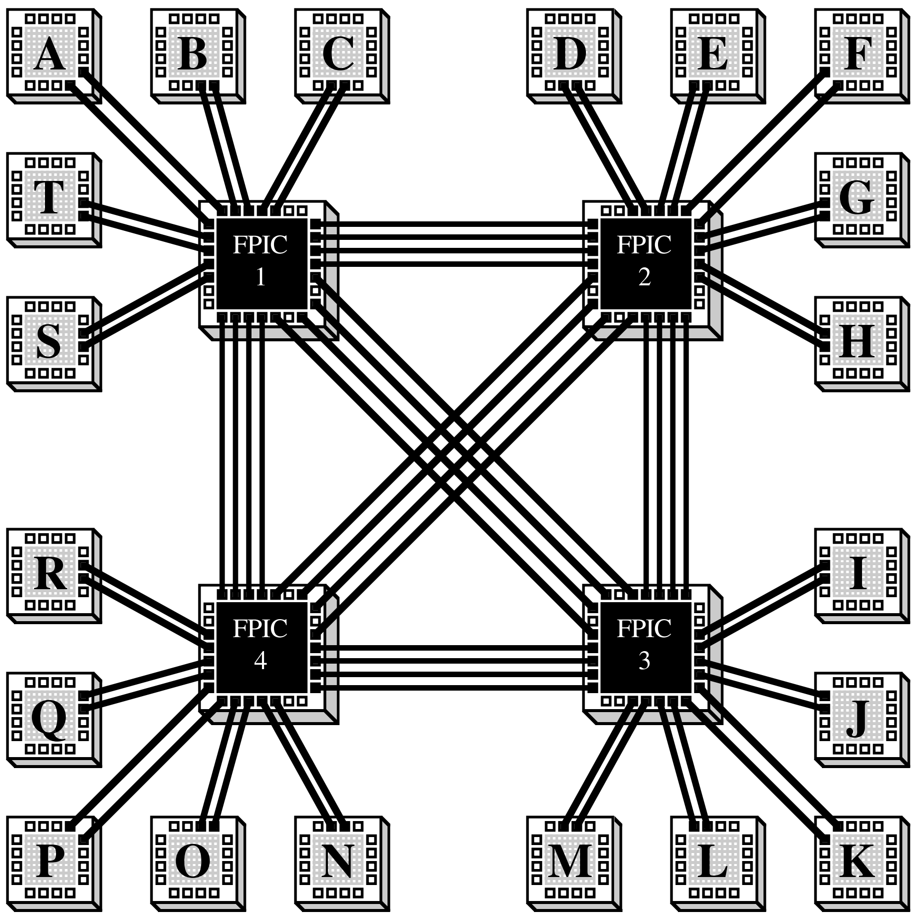


Figure 29. The Aptix AXB-AP4 topology [Aptix93b]. Logic-bearing FPGAs are connected only to routing-only FPICs, but the FPICs connect to both FPGAs and other FPICs.

There are two downsides to this topology. First, signals may have to go through many more routing-only chips than in a simple crossbar, since they could potentially have to go all the way up to the top level of the hierarchy to make a connection. However, the maximum routing distance is less than in a mesh, since the length (in chips routed through) of the maximum route in a mesh grows by $\sqrt{N}$ (where N is the number of logic-bearing FPGAs in the system), while the length in a hierarchical crossbar grows by *log(N)*. The other problem is that the hierarchical crossbar topology requires a large amount of resources to implement the routing-only chips in the system. If one could instead use the routing-only chips as logic-bearing chips, the capacity of the system might be greatly increased. However, if this extra logic capacity cannot efficiently be used, it will be of little value.

Some systems use a two-level topology, which is somewhat similar to the pure crossbar topologies. Just as in the crossbar topologies, FPGAs in the system are connected only to routing-only chips. However, unlike the pure crossbar, the routing-only chips in the system are connected to both logic-bearing and routing-only chips. That is, there is a topology of connections between the routing-only chips in the system, and the FPGAs connect to these chips. Thus, these systems are similar to multiprocessors, in that for two processing elements to communicate (the FPGAs in a two-level topology), they must send signals through a router connected to the source (an FPIC or crossbar). The signal then travels through intervening routers until it reaches the router connected to the destination, at which point it is sent to that processor (an FPGA in our case). An example of such a system is the Aptix AXB-AP4 [Aptix93b]. As shown in Figure 29, five FPGAs are connected to each FPIC, and all the FPICs are connected together. In this system, the longest route requires moving through 2 FPICs, and no intermediate FPGAs are used for routing. If instead a mesh was built out of the 20 FPGAs in this system, it potentially could require routing through many FPGAs to reach the proper destination, increasing the delay, and using up valuable FPGA I/Os. However, whether the routing flexibility of the two-level topology justifies the substantial cost of the FPICs is unclear. Other two-level systems have been constructed with a variety of topologies between the routing-only chips [Adams93, Galloway94, Njølstad94].

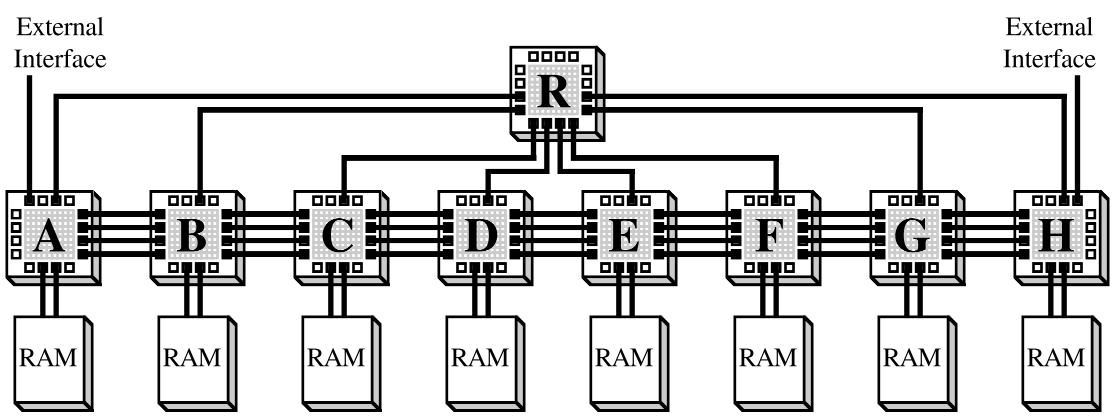


Figure 30. The Splash 2 topology [Arnold92]. The linear array of FPGAs (**A-H**) is augmented by a routing-only crossbar (**R**). Note that the real topology has 16 FPGAs in the linear array.

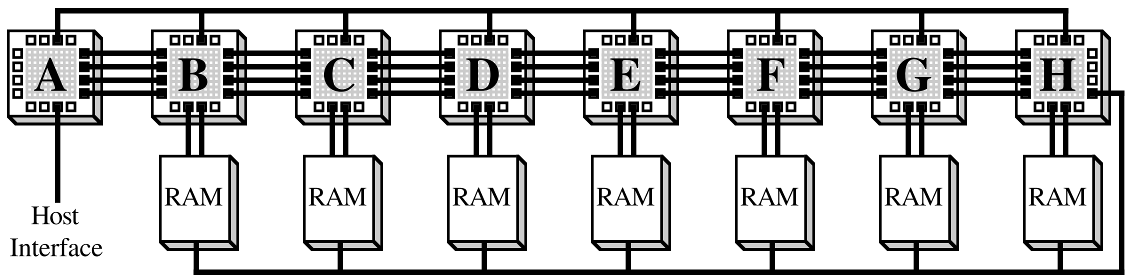


Figure 31. The Anyboard topology [Thomae91]. The linear array of FPGAs is augmented with a global bus.

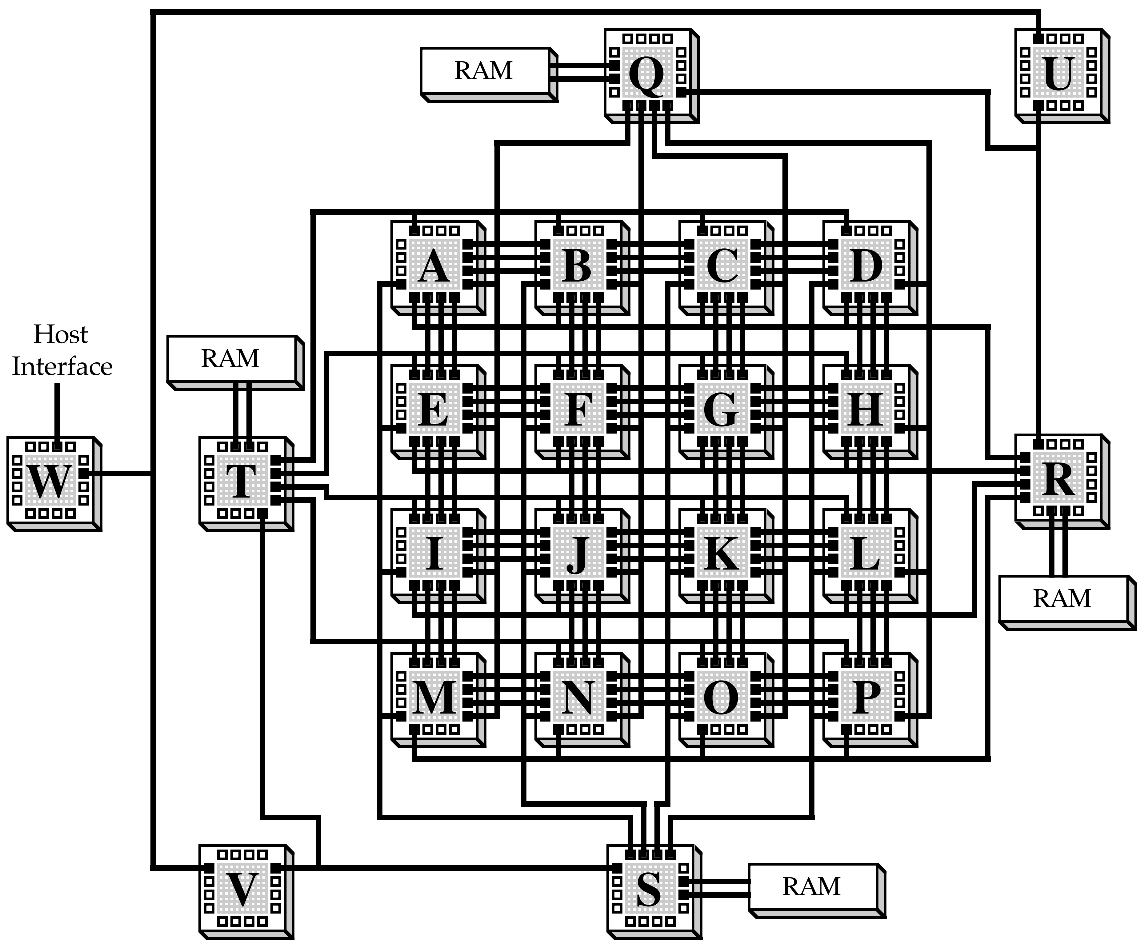


Figure 32. The DECPeRLe-1 topology [Vuillemin95]. The central 4x4 mesh (**A-P**) is augmented with global buses to four support FPGAs (**Q-T**), which feed to three other FPGAs (**U-W**).

Some systems are more of a hybrid between different topologies than a single topology. One example of this is the Splash 2 architecture [Arnold92]. The Splash 2 machine takes the linear array of its predecessor Splash [Gokhale90], and augments it with a crossbar interconnecting the FPGAs (Figure 30). In this way the system can still efficiently handle the systolic circuits that Splash was built to support, since it retains the nearest-neighbor interconnect, but the crossbar supports more general communication patterns, either in support of or as replacement to the direct interconnections. There are several other augmented linear arrays [Benner94, Box94, Darnauer94, Carrera95]; one example is the Anyboard [Thomae91], which has a linear array of FPGAs augmented by global buses (Figure 31). Another hybrid topology is the DECPeRLe-1 board [Vuillemin95], which has a 4x4 mesh of FPGAs augmented with shared global buses going to four support FPGAs (Figure 32). These are then connected to three other FPGAs that handle global routing, connections to memory, and the host interface. The DECPeRLe-0 board is similar [Bertin93].

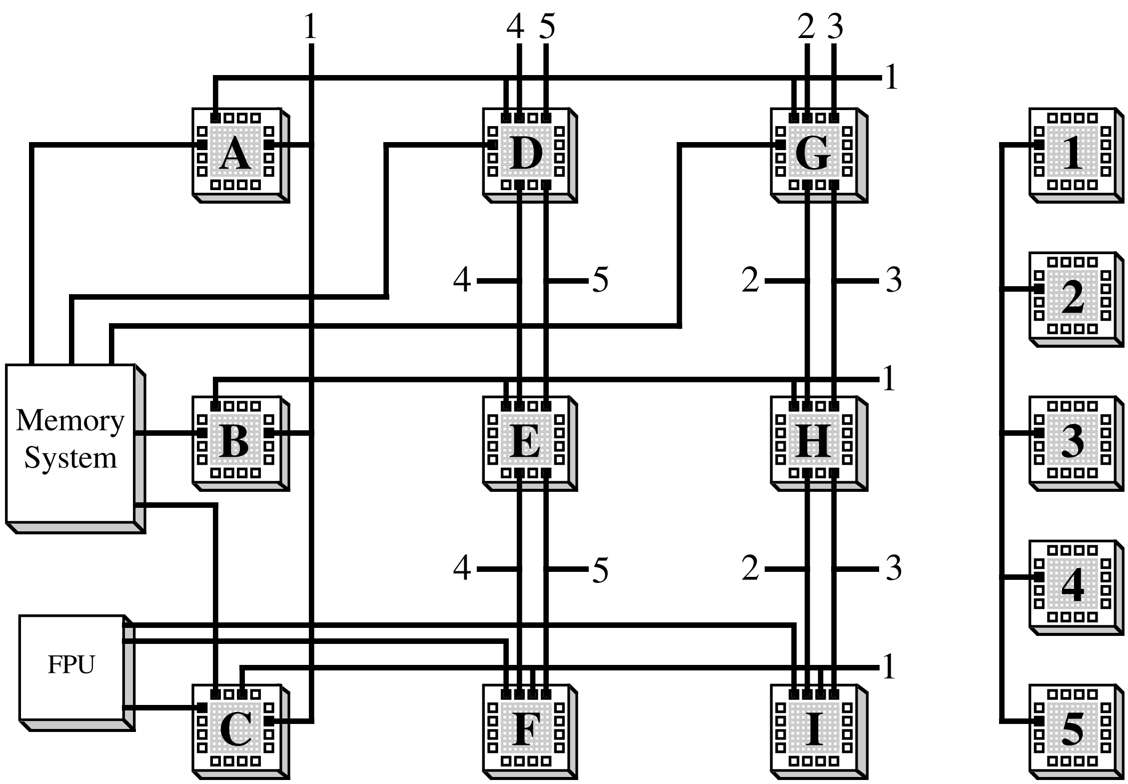


Figure 33. The Marc-1 topology [Lewis93]. The complete topology consists of two copies of the left subsystem (**A**-**I**, Memory & FPU), and one copy of the right (**1**-**5**). Numbers by themselves indicate connections to the FPGAs at right.

Some topologies are unique [Engels91, Cox92, Erdogan92, Casselman93, Herpel93, Lewis93, Wazlowski93, Howard94a, Nguyen94, Saluvere94, Hayashi95, Högl95, Van den Bout95]. These are often machines primarily built for a specific application, and their topology is optimized for the features of that application domain. For example, the Marc-1 (Figure 33) [Lewis93] is a pair of 3x3 meshes of FPGAs (**A**-**I**), where most of the mesh connections link up to a set of FPGAs intended to be used as crossbars (**1**-**5**). While the vertical links are nearest-neighbor, the horizontal links are actually buses. There is a complex memory system attached to some of the FPGAs in the system, as well as a floating point unit. This machine architecture was constructed for a specific application - circuit simulation (and other algorithms) where the program to be executed can be optimized on a per-run basis for values constant within that run, but which may vary from dataset to dataset. For example, during circuit simulation, the structure of the gates in the circuit are fixed once the program begins executing, but can be different for each run of the simulator. Thus, if the simulator can be custom compiled on a per-run basis for the structure of the circuit being simulated, there is the potential for significant speedups. Note that this is different than logic emulation, since a special-purpose processor is mapped onto the Marc-1 system, and this processor mapping simulates the circuit, while a logic emulator directly maps the logic onto the multi-FPGA system. Because of the restricted domain of compiled-code execution, the topology was built to contain a special-purpose processor, with the instruction unit in FPGAs **A**-**C**, and the datapath in the FPGAs **D**-**I**.

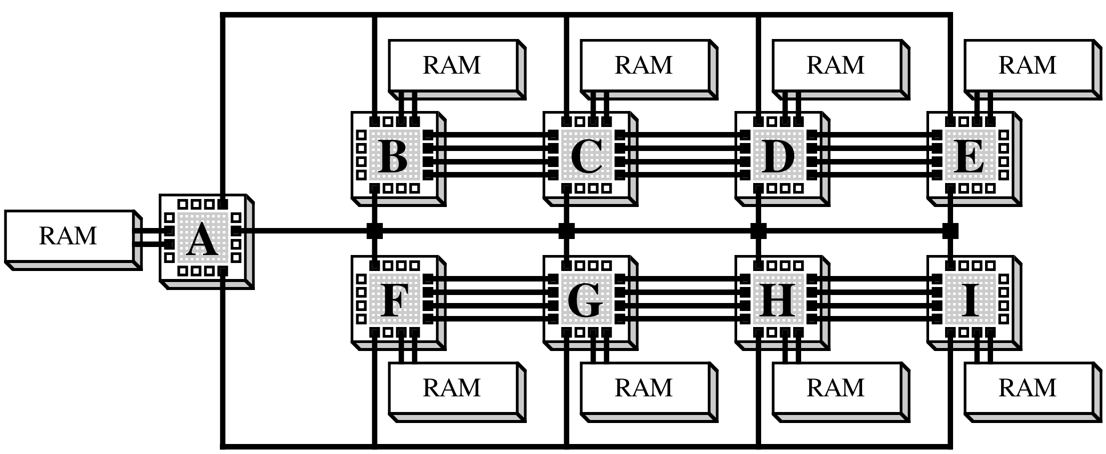


Figure 34. The RM-nc system [Erdogan92]. The linear arrays can be grown larger than the system shown.

Another example of a system optimized for a specific task is the RM-nc system [Erdogan92]. As shown in Figure 34, the system has a controller chip **A**, and two linear arrays of FPGAs **B-E** and **F-I**. Each FPGA has a local memory for buffering data. The system contains three major buses, with the outside buses going to only one linear array each, while the center bus is shared by all FPGAs in the system. This system is optimized for neural network simulation, with the controller chip handling global control and sequencing, while the FPGAs in the linear array handle the individual neuron computations. A similar system for neural-network simulation, with buses going to all FPGAs in the system, and with no direct connections between neighbors, has also been built [Eldredge94].

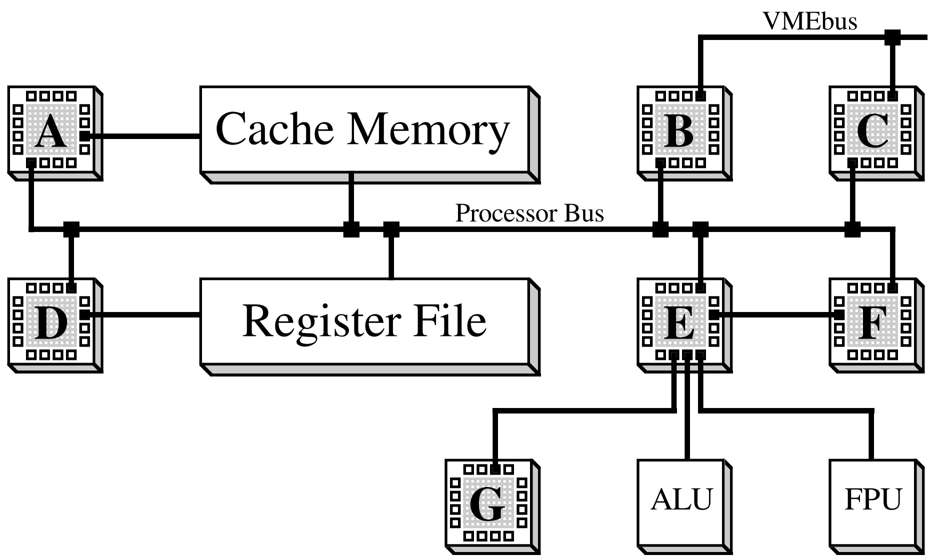


Figure 35. System similar to the Mushroom processor prototyping system [Williams91].

One of the most common domains for the development of a custom multi-FPGA system is the prototyping of computers. A good example of this is the Mushroom system [Williams91], which is a system of FPGAs, memories, and computation chips meant to be used for processor prototyping (the exact interconnection pattern is unavailable, but an approximate version is shown in Figure 35). Seven FPGAs are included in the system, and each of these has a specific role. FPGA **A** is a cache controller, **B** & **C** serve as a VMEbus interface, **D** handles register accesses, **E** is for processor control, **F** performs instruction fetches, and **G** is for tag manipulation and checking. Memories are included for caches and the register file, while an ALU and FPU chip are present for arithmetic operations. The advantage of this system is that the FPGAs can be reconfigured to implement different functions, allowing different processor structures to be explored. The arithmetic functions and memory, features that are inefficient to implement in FPGAs and which are standard across most processors, are implemented efficiently in chips designed specifically for these applications. Other systems have adopted a similar approach, including another system intended for developing application-specific processors [Wolfe88], a workstation design with FPGAs for I/O functions and for coprocessor development [Heeb93], and a multiprocessor system with FPGA-based cache controllers for exploring different multiprocessor systems and caching policies [Öner95]. A somewhat related system is the CM-2X [Cuccaro93], a standard CM-2 supercomputer with the floating-point unit replaced with a Xilinx FPGA. This system allows custom coprocessors to be built on a per-algorithm basis, yielding significant performance increases for some non-floating-point intensive programs.

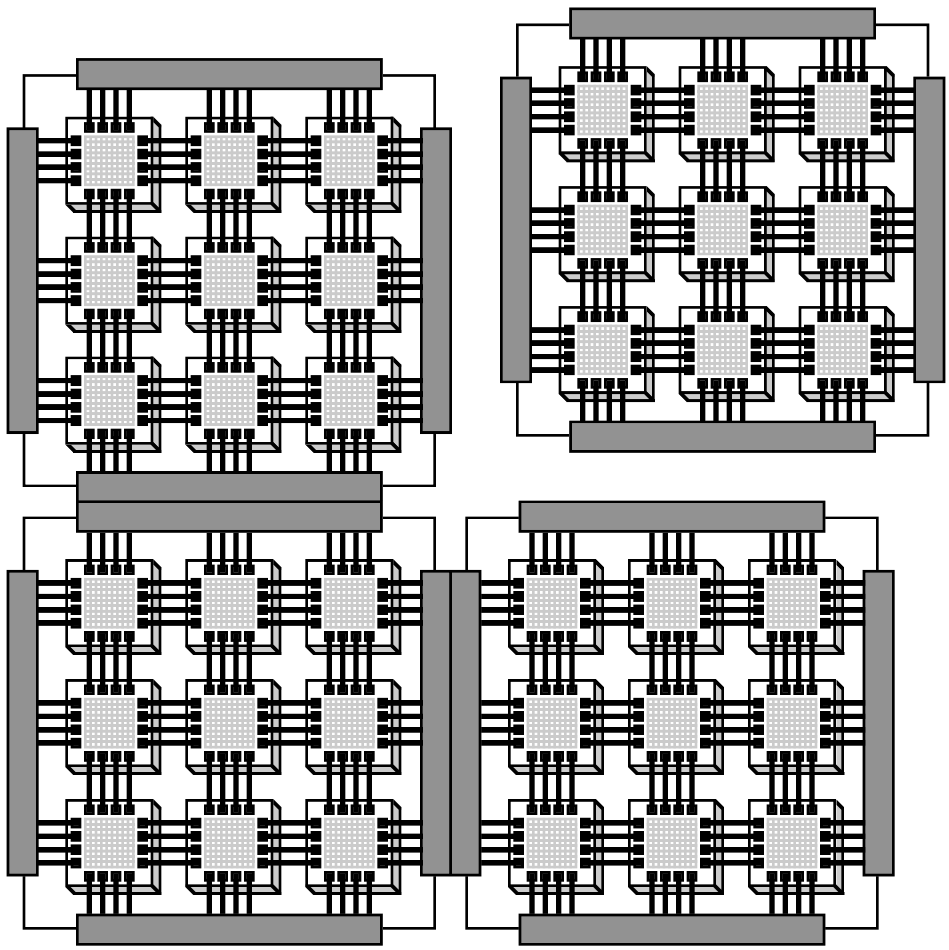


Figure 36. Expandable mesh topology similar to the Virtual Wires Emulation System [Tessier94]. Individual boards are built with edge connectors and a small amount of logic, and can be interconnected to form a larger mesh.

There are interesting features of multi-FPGA systems beyond just the interconnect topology. One of these is the ability to increase the size of the multi-FPGA system. As described previously, systems based on hierarchical crossbars [Varghese93] can grow in size by adding extra levels of hierarchy. Other systems provide similar expandability, with special interconnection patterns for multi-board systems [Amerson95, Högl95]. In the case of meshes and linear arrays, the systems can be built of a basic tile with external connectors and a small amount of on-board logic (Figure 36), and the system can be expanded by connecting together several of these boards [Thomae91, Arnold92, Shaw93, Tessier94, Drayer95]. The GERM system [Dollas94] is similar, except that the four external connectors are built to accommodate ribbon cables. In this way, fairly arbitrary topologies can be created.

Multi-FPGA systems are often composed of several different types of chips. The majority of systems are built from Xilinx 3000 or 4000 series FPGAs [Xilinx94], though most commercial FPGAs have been included in at least one multi-FPGA system. There are some system designers that have chosen to avoid commercial FPGAs, and develop their own chips. Some have optimized their FPGAs for specific applications, such as general custom-computing and logic emulation [Amerson95], emulation of telecommunication circuits [Hayashi95], or image processing [Quénot94]. Another system uses FPGAs optimized for inter-chip communication on an MCM substrate [Dobbelaere92], since MCMs will be used to fabricate the system to achieve higher density circuits. A final example is the Pegasus system [Maliniak94], which uses a hybrid processor/FPGA chip with multiple configurations. By rapidly switching configurations the hardware can be time-division multiplexed. In this way, the hardware is reused and the communication pipelined, hopefully yielding higher density mappings.

Many multi-FPGA systems include non-FPGA chips. By far the most common element to be included is memory chips. These chips are usually connected to the FPGAs, and are used as temporary storage for results, as well as general-purpose memories for circuit emulation. Other systems have included integer and/or floating point ALUs [Wolfe88, Williams91, Lewis93, Benner94], DSPs [Engels91, Bergmann94, vom Bögel94, Zycad94a], and general-purpose processors [Shaw93, Raimbault93, Benner94, Koch94, vom Bögel94, Zycad94a] to handle portions of computations where dedicated chips perform better than FPGA solutions. Another common inclusion into a multi-FPGA system is crossbars or FPICs (Chapter 2). For example, a multi-FPGA system with a crossbar or hierarchical crossbar topology requires chips purely for routing. An FPIC or crossbar chip can handle these roles. If the FPIC or crossbar has a lower unit cost, is capable of higher performance or higher complexity routing, or has a larger number of I/O pins, then it can implement the routing functions better than a purely FPGA-based solution. There have been several systems that have used crossbar chips or FPICs in crossbar [Kadi94, Weiss94] and hierarchical crossbar topologies [Varghese93], as well as hybrid crossbar/linear array topologies [Arnold92, Darnauer94, Carrera95] and other systems [Adams93, Aptix93b, Casselman93, Galloway94, Njølstad94, Högl95, Van den Bout95].

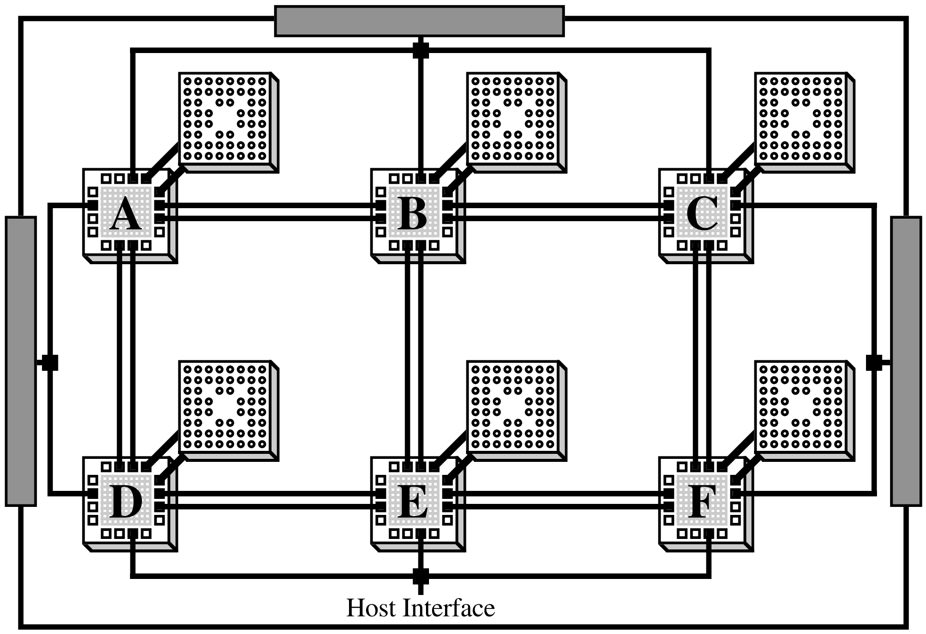


Figure 37. The MORRPH topology [Drayer95]. Sockets next to the FPGAs allow arbitrary devices to be added to the array. Buses connected to external connectors allow multiple boards to be hooked together to build an arbitrarily large system.

While adding fixed, non-FPGA resources into a multi-FPGA topology may improve quality for some types of mappings, it is possible to generate a more general-purpose solution by allowing the inclusion of arbitrary devices into the array. For example, in the MORRPH topology [Drayer95] sockets are placed next to the FPGAs so that arbitrary devices can be inserted (Figure 37). Thus, in a mapping that requires extra memory resources, memories can be plugged into the array. In other circumstances, DSPs or other fixed-function chips can be inserted to perform complex computations. In this way, the user has the flexibility to customize the array on a per-mapping basis. Other systems have adopted a similar strategy [Butts91, Sample92, Aptix93b], with a limited ability to insert arbitrary chips into the multi-FPGA system.

Some systems are more of an infrastructure for bringing to bear the best mix of resources rather than a specific, fixed multi-FPGA system. One example of this is the G800 system [Giga95]. The board contains two FPGAs, some external interface circuitry, and four locations to plug in compute modules (Figure 38). Compute modules are cards that can be added to the system to add computation resources, and can be stacked four deep on the board. Thus, with four locations that can be stacked four deep, a total of 16 compute boards can be combined into a single system. These compute boards are connected together, and to the FPGAs on the base board, by a set of global buses. Compute modules can contain an arbitrary mix of resources. Examples include the X210MOD-00, which has two medium Xilinx FPGAs, and the X213MOD-82, which has two large FPGAs, 8MB of DRAM, and 256 KB of SRAM. The users of the system are free to combine whatever set of modules they desire, yielding a system capacity ranging from only two medium FPGAs (a single X210MOD-00), to 32 large FPGAs (16 X213MOD-82s) and a significant amount of RAM. Similar systems include DEEP [vom Bögel94] and Paradigm RP [Zycad94a]. The Paradigm RP has locations to plug in up to eight boards. These boards can include FPGAs, memories, DSPs, and standard processors. In the G800 and Paradigm RP systems, cards with new functionality or different types of chips can easily be accommodated.

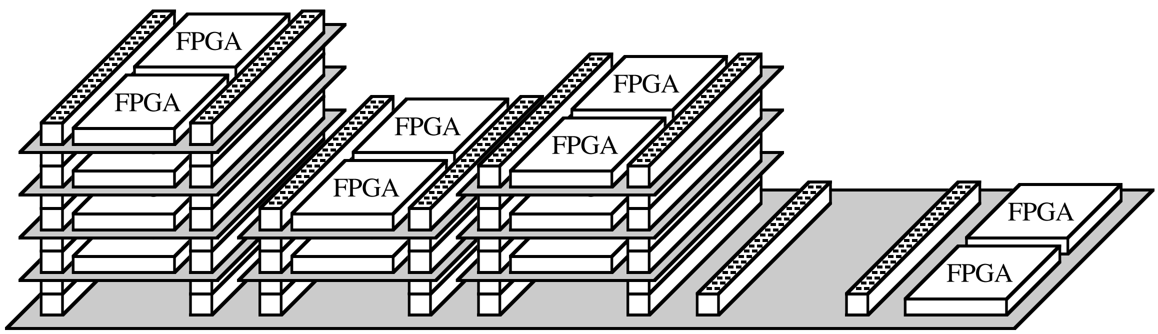


Figure 38. The G800 board [Giga95]. The base board has two FPGAs and four sockets. The socket at left holds four computing module boards (the maximum allowed in a socket), while the socket at right has none.

Even more flexible systems are possible. One example is the COBRA system [Koch94]. As shown in Figure 39, the system is made up of several types of modules. The standard *base module* has four FPGAs, each attached to an external connection at one of the board's edges. Boards can be attached together to built a larger system, expanding out in a 2D mesh. Other module types can easily be included, such as modules bearing only RAM, or a host interface, or a standard processor. These modules attach together the same way as the base module, and will have one to four connectors. One somewhat different type of module is a bus module. This module stacks other modules vertically, connecting them together by a bus.

Systems like COBRA allow an arbitrary resource mix to be brought to bear on a problem. Thus, a custom processing system can quickly be built for a given task, with FPGAs handling the general logic, while standard processors handle complex computations and control flow. Thus, the proper resource is always used for the required functionality. In Chapter 6 we will present another such system, which is optimized for the rapid-prototyping of multi-chip circuits. It offers the expandability of COBRA, with MORRPH's ability to include arbitrary devices, and was originally proposed before either of these architectures.

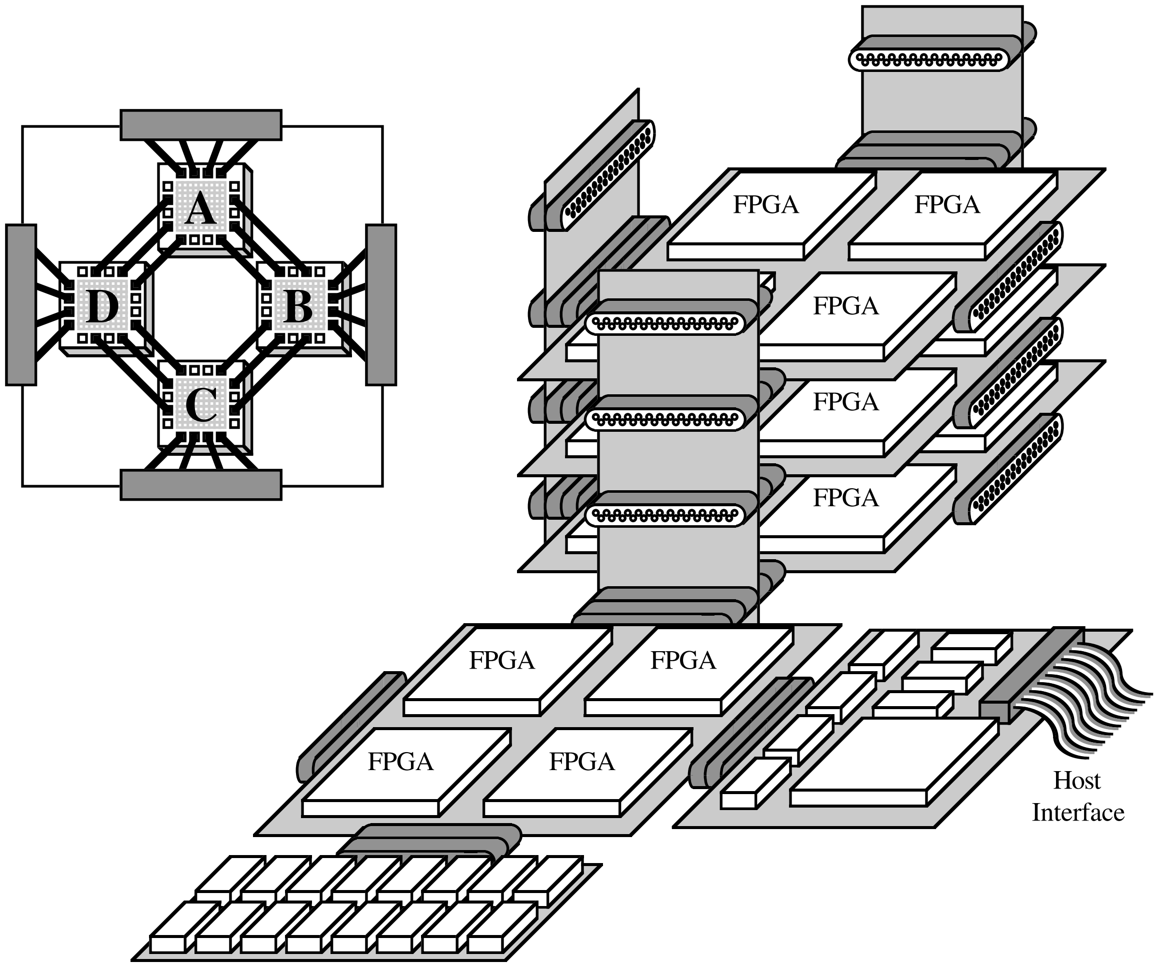


Figure 39. Base module (upper left) and example topology built in the COBRA system [Koch94]. The mapping includes a tower of three base modules surrounded by three bus modules (top), a base module (center), a RAM module (bottom), and an I/O module (right).

In the following chapters we will delve into several issues in multi-FPGA system hardware, including multi-chip system prototyping in Chapter 6, routing topologies for multi-FPGA systems in Chapter 7, and interface support for logic emulators in Chapter 8. Chapter 9 begins the discussion of software support for multi-FPGA systems.

Chapter 6. Springbok

Introduction

As pointed out in Chapter 4, logic emulation should be an important part of any logic validation environment. To perform logic emulation requires an appropriate hardware substrate and an integrated software mapping solution. In Chapter 5 we described many multi-FPGA systems. In general, they provide a constrained amount and type of resources in a fixed hardware system. For ASIC emulation tasks this is usually sufficient. However, board-level designs, designs where multiple chips are connected on a circuit board, will not fit onto these systems. The problem is that many of the chips in such systems are quite complex. Thus, if they had to be mapped onto a standard multi-FPGA system, they would require a huge amount of FPGA resources. In some cases, this process may not even be possible, since the manufacturers of these complex chips will (rightfully) be reluctant to divulge the exact implementation details of their chips. However, most or all of the chips used to construct the board-level design will be prefabricated. Thus, if we could include the actual chips into the logic emulation, this will yield a much more efficient implementation.

While some multi-FPGA systems allow the inclusion of a few arbitrary chips, most multi-FPGA systems are not designed to support a emulation style primarily dependent on premade components. This is because these systems rely primarily on FPGAs to implement the logic in the system, and the number of premade components is limited.

In the rest of this chapter we will discuss Springbok, a system to aid in the development and debugging of board-level designs. It has a flexible topology and resource mix, with the ability to include arbitrary chips into the system. While there are other multi-FPGA systems that have much of Springbok's capabilities (such as COBRA [Koch94] and MORRPH [Drayer95], described in Chapter 5), Springbok predates all of these systems [Hauck94]. An alternative approach to prototyping with a multi-FPGA system is the Aptix field-programmable circuit boards, which offer a sea-of-holes for plugging in arbitrary devices. However, the system is based around very expensive FPIC chips and multi-layer boards. Also, since the Aptix systems are based around FPIC hubs, there is no logic capacity (other than the chips added to the topology) for handling random logic, chip slow-down circuitry, or Virtual Wires multiplexing.

In the next section we discuss the proposed Springbok architecture, and then the mapping tools. We also consider Field-Programmable Interconnects (FPICs), and flexible multi-FPGA systems, which are promising approaches to circuit board testing, and compare them to the Springbok system.

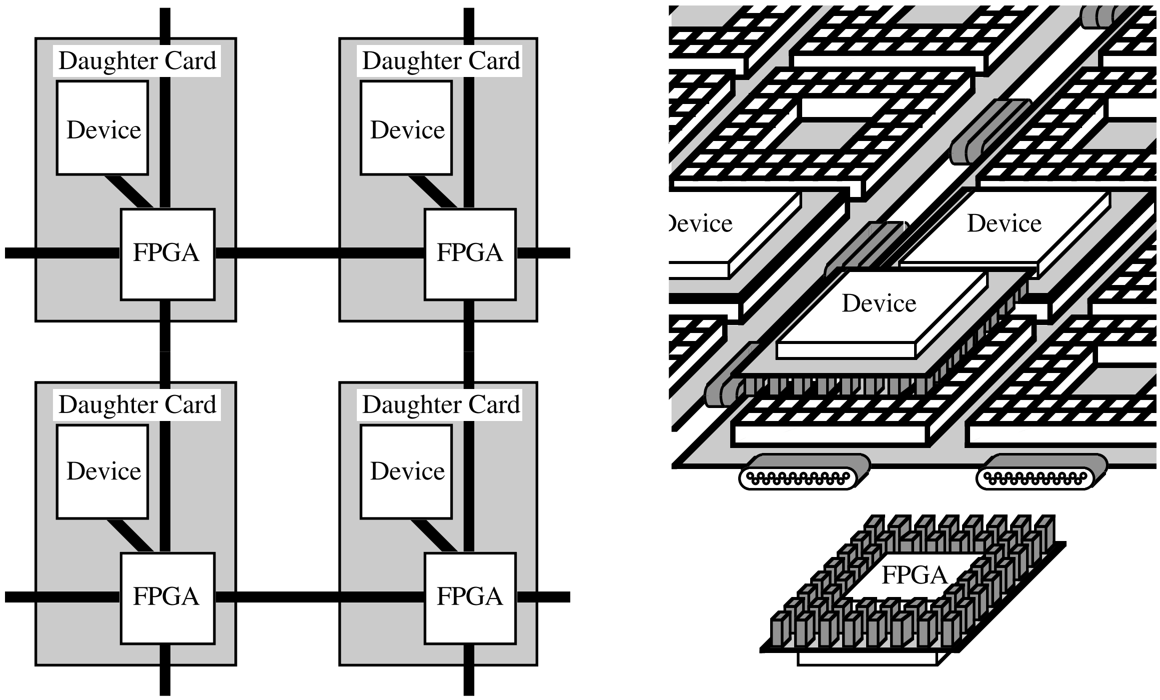


Figure 40. The Springbok interconnection pattern (left), and two connected Springbok baseplates with four daughter cards (right). The card at front is similar to the other daughter cards, but is shown upside-down.

The Springbok Architecture

The Springbok system is based on the philosophy that to develop and test board-level designs one needs a practical way to perform incremental development and testing using many of the actual chips of the final system without incurring the effort and expense of either wire-wrap or complete board fabrication. Our approach is to allow the important, complex chips comprising a design to be embedded in an FPGA-based structure, which uses these FPGAs for both the routing and rerouting of signals, as well as the implementation of random logic (Figure 40). To allow a specific circuit to be implemented in this structure, the Springbok system is comprised of a baseplate with sites for daughter cards. The daughter cards are large enough to contain both an arbitrary device on the top, as well as an FPGA on the bottom. Note that the device can be a chip, such as a processor or memory, I/O elements such as switches and LCD interfaces, or whatever else is necessary to implement the system. If necessary, daughter cards can be built that span several locations on the baseplate to handle higher space or bandwidth requirements. The daughter cards plug into the baseplate, which handles power, ground, and clock distribution, FPGA programming, and inter-daughter card routing. The baseplates include support for communicating with a host computer, both for downloading programming and for uploading data captured during prototype runs. The baseplates are constructed such that they can be connected together with each other, forming an arbitrarily large surface for placing daughter cards. The inter-daughter card routing structure is a 1-hop mesh, with the specific pin interconnections as detailed in Chapter 7. In many ways, this approach is similar both to mesh-connected multiprocessors, as well as the approach suggested in [Seitz90].

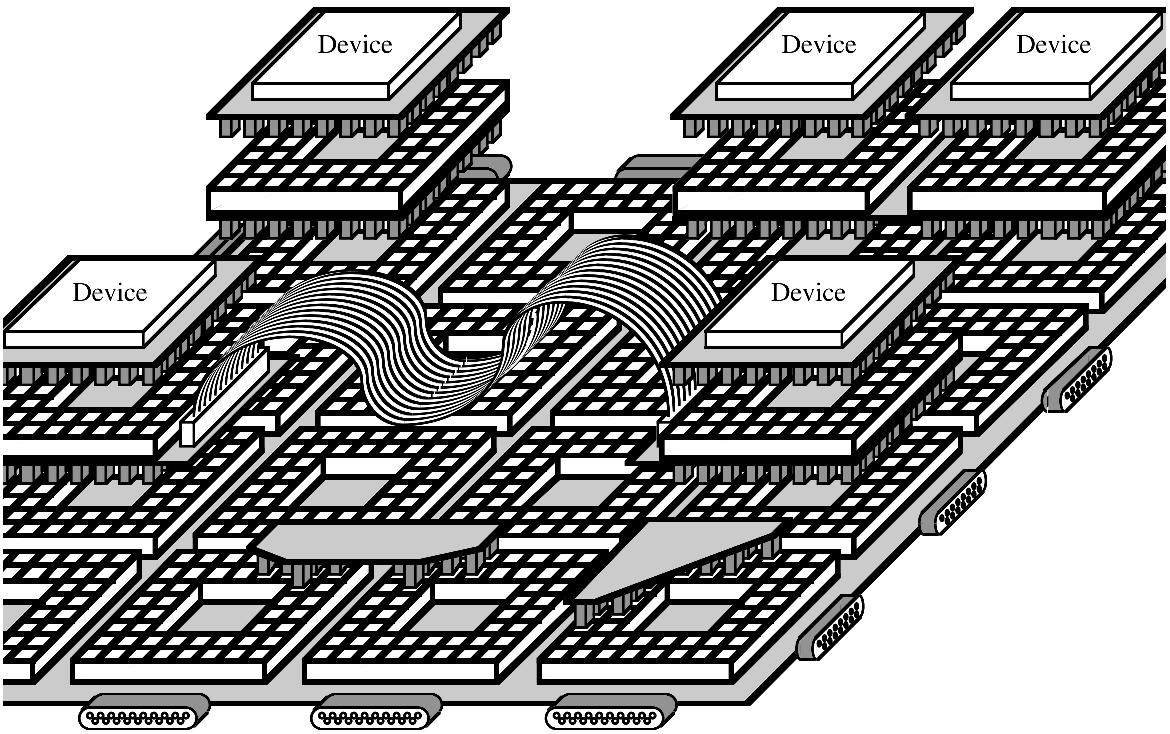


Figure 41. Non-device daughter cards and extender cards, including cards to add more FPGA logic (top left), bandwidth (double-sized card at top right), long-distance communication (middle), and edge bandwidth (bottom middle and bottom right). All but the edge bandwidth cards have FPGAs on the bottom.

An important feature of the Springbok system is the ability to insert system-specific chips on daughter cards placed into the array. This also allows us to include other, specialized daughter cards. For example, early in the design cycle the specific chips to be used to implement much of the logic may be unspecified. Thus, instead of inserting only chip-carrying daughter cards into the array, other cards with only FPGAs on them could be included. As in most other FPGA systems, there is also the potential that the simple connection scheme described above will not be able to accommodate all of the logic or routing assigned to a given location. However, as opposed to a fixed multi-FPGA system, we can insert new "extender" cards between a daughter card and the baseplate to deal with these problems (Figure 41). For example, if the logic assigned to a given FPGA simply will not fit, an extender card with another FPGA can be inserted so that it can handle some of the logic. If too many signals need to be routed along a given link in the mesh structure, an extender card spanning several daughter card positions can be added, with new routing paths included on the inserted card. Note that while most routing limitations could be dealt with via Virtual Wires (Chapter 9), added cards for routing will still serve a purpose by reducing the reliance on Virtual Wires, thus decreasing cycle time and increasing logic capacity. For signals that must go long distances in the array, sets of extender cards with ribbon cable connections can be inserted throughout the array to carry these long-distance wires. Also, at the edge of the mapping where edge effects can limit available bandwidth, dummy daughter cards which simply contain hardwired connections between their neighbors can be inserted. Thus, the Springbok approach to resource limitations is to add resources wherever necessary to map the system. In contrast, a fixed array cannot afford a failure due to resource limitations, since it would then have to redo the costly step of mapping to all of the constituent FPGAs. Thus fixed arrays must be very conservative on all resource assignments, underutilizing resources throughout the system, while Springbok simply fixes the problems locally as they arise.

Another important benefit of Springbok is how it supports hardware subroutines. In many design environments there will not be just one system developed, but instead a family of products may be built. Many of these products have subsystems shared across the entire family. For example, a company developing disk controllers would have SCSI interfaces on most of its products, as well as an interface to the host computer's bus (e.g. [Katz93]). In the Springbok model, such heavily replicated elements can be fabricated or wire-wrapped as a single daughter card, and from then on used as a hardware subroutine for all subsequent designs. One could use a similar approach in a wire-wrap domain by developing prototyping boards with these functionalities fabricated on the board. However, in such a system one must fix the number and type of these subroutines ahead of time, and this mix cannot be increased. Thus, a prototyping board designed with one SCSI interface would be useless for prototyping a two SCSI port controller, and a board for one type of computer bus could not be used for a different bus. In Springbok this is not an issue, because the number and type of daughter cards can vary from system to system, and cards with functionality not even considered in the first system of a family can be added easily in later designs.

While the construction of the Springbok system fixes many problems encountered in rapid-prototyping, there are two important concerns that remain. First, the physical wires in the target system are replaced with digital connections in Springbok. Second, while speeds achieved may be orders of magnitude faster than simulation, they will still be less than the final system speeds. The problem with the way Springbok handles wires is that not all physical wires in mapped systems are used in a purely digital, unidirectional manner. While some systems use analog subsystems, this is beyond the scope of Springbok, and any such subsystems will need to be either included as specialized daughter cards, or not handled at all. For non-unidirectional flow, signals such as buses and bi-directional signals must be handled. As shown in Figure 42, a tristate bus can be replaced by a pure logic structure [Butts91]. Note that the circuit shown at right implements a floating low bus. A floating high bus can be constructed by inverting the inputs and outputs of the circuit. Once the tristate bus is converted to the implementation shown, the gates implementing the communication can be mapped to the FPGA identically to the rest of the random logic in the system.

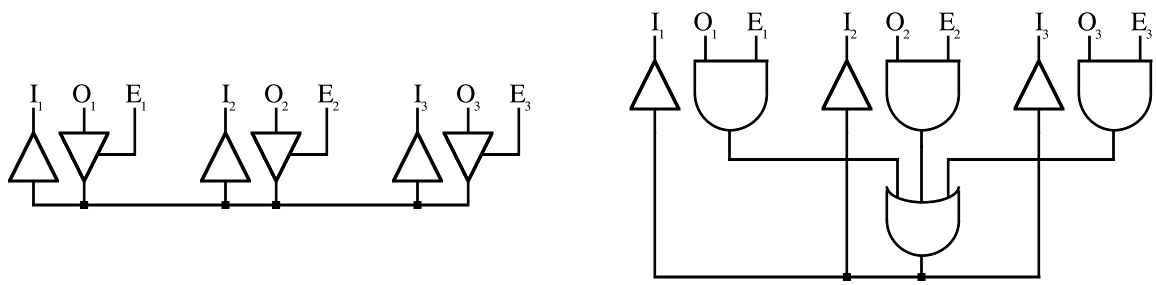


Figure 42. Method for handling bi-directional signals [Butts91]. The tristate bus at left is replaced with the logic structure at right. Note that all signals are unidirectional in the structure at right.

The second problem mentioned above, that a Springbok mapping will almost always be somewhat slower than the system being prototyped, causes several concerns. First, one of the goals of a prototype is to test the system in the actual target environment. However, other systems that interact with the circuit will be expecting certain timing assumptions to be upheld, such as those contained in a bus protocol. This same problem has been encountered by ASIC logic emulator vendors such as Quickturn, and their customers have dealt with this by building protocol-specific buffering solutions. Such an approach would work equally well with Springbok. An even better answer is detailed in Chapter 8, which presents a general solution to the logic emulator interface issue.

The second manifestation of the slow-down problem is that the chips used in the system must be slowed down as well. Many chips can simply be operated at the slower clock rate and still function properly. Springbok mappings will operate fast enough that charge leakage from dynamic nodes will not be an issue for most chips. The primary concern is for chips with phase-locked loops which can only operate at certain clock frequencies. Solutions to this problem include stalling processor chips either through manipulation of an explicit stall signal or insertion of stall instructions into the chip's instruction stream, dummy data and buffering for pipelined chips, or even replacement of very restrictive chips with slower members of the chip family or mapping it into FPGA logic.

The Springbok Mapping Tools

As is found with logic emulators, the target architecture is only half the system. Just as important as a flexible medium for implementing the desired functionality is the need for an integrated software system to map target designs onto the architecture.

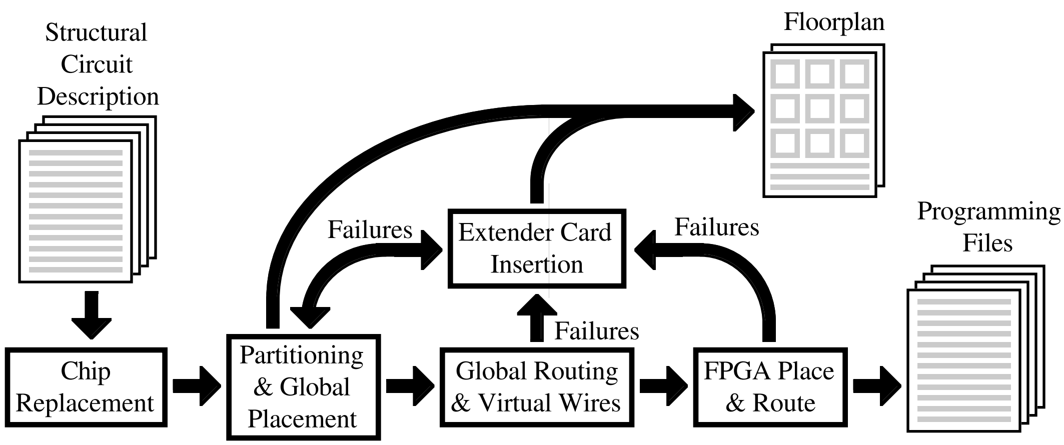


Figure 43. Springbok software flow diagram.

The overall software system flow in Springbok is shown in Figure 43. The mapping tools start with a structural description of the system to be mapped, where the system logic can either be assigned to specific chips, or possibly have portions of random logic which have not yet been assigned to chips. Through a series of steps this logic is mapped down to a form that can be placed onto the Springbok architecture, producing both a floorplan describing how daughter cards and extender cards must be placed onto baseplates, and programming files for configuring the individual FPGAs.

The first step in the mapping process is chip replacement. We expect Springbok to handle systems ranging from those with only the critical components specified (the rest left as random logic) to those with most or all of the logic implemented with specific chips. All logic not assigned to chips in the original structural description will be handled by FPGAs in Springbok, and the Springbok system may decide to assign logic mapped to chips in the description to FPGAs as well. For example, in a completely specified design, a chip that simply buffers signals might be included in the original description, but would not be necessary in the

Springbok prototype. Thus, by replacing this chip with the corresponding logic, the resulting mapping might save some hardware. Also, logic placed into a specific FPGA in the source description restricts the placement choices in the Springbok implementation, and might also be removed. Finally, since Springbok is fairly FPGA-intensive, mappings will be more compact if a certain percentage of the system is random, unassigned logic. Thus, during chip replacement the software could use a greedy algorithm to replace chips with random logic until the optimal percentage of random logic is reached. That is, greedily removing the chip with the least logic assigned to it should allow the removal of the most number of chips before reaching the target percentage of logic in the FPGAs, thus reducing the total amount of hardware. It might also be useful to break up the highest fanout chip (i.e., that chip connected to the most other chips) to ease the resulting routing congestion and delay, though how exactly to integrate these two considerations requires study. Note that the user can specify that any or all chips be left unreplaced.

The next step, partitioning and global placement, determines the overall layout of the Springbok mapping. It must decide which daughter cards to use, where to place them, and what logic will be implemented by each of these cards. Standard methods for partitioning such as k-way partitioning are difficult to use for this problem, since these algorithms ignore the locality required in mesh communication patterns, and also generally require that the number of partitions be fixed. As opposed to most current systems, the size and composition of the Springbok substrate is flexible, expanding or contracting to most efficiently map the target system. This flexibility makes the partitioning more difficult, but allows greater optimization opportunities. An overview of our partitioning approach is contained in Chapter 9, with further details in Chapter 10 and Chapter 11.

After partitioning and global placement, routing must be considered. The problem of deciding which intermediate FPGAs to route through is fairly straightforward, and is very similar to the single-FPGA routing problem. Thus, we can use much of the research on single-FPGA routing (such as [McMurchie95]) to handle this portion of the routing problem. However, this still leaves the problem of determining which individual I/O pins to use to route between the FPGAs. In order to handle pin assignment for multi-FPGA systems, we have developed a new technique that simultaneously reduces the routing resource usage, reducing both area requirements and delay, while also speeding up the mapping process (Chapter 12).

With partitioning, global placement, and routing completed, it is then necessary to place and route the individual FPGAs. For these tasks we can use one of several reasonable commercial software packages available. Some of their underlying algorithms are discussed in Chapter 2.

As shown in the software flow diagram, any mappings that fail are fixed by software that inserts extender cards into the Springbok array. Then, partitioning and global routing are rerun, maintaining as much of the previous mapping as possible while easing demands on the failing FPGAs. Note that this portion of the flow is also important for incremental alteration of a prototype. Specifically, as a system is debugged errors will need to be fixed, hopefully without requiring a completely new mapping of the circuit. By extending the partitioning and global routing steps to handle incremental alterations to fix failed FPGA mappings, we also have support for incremental alterations for bug fixes.

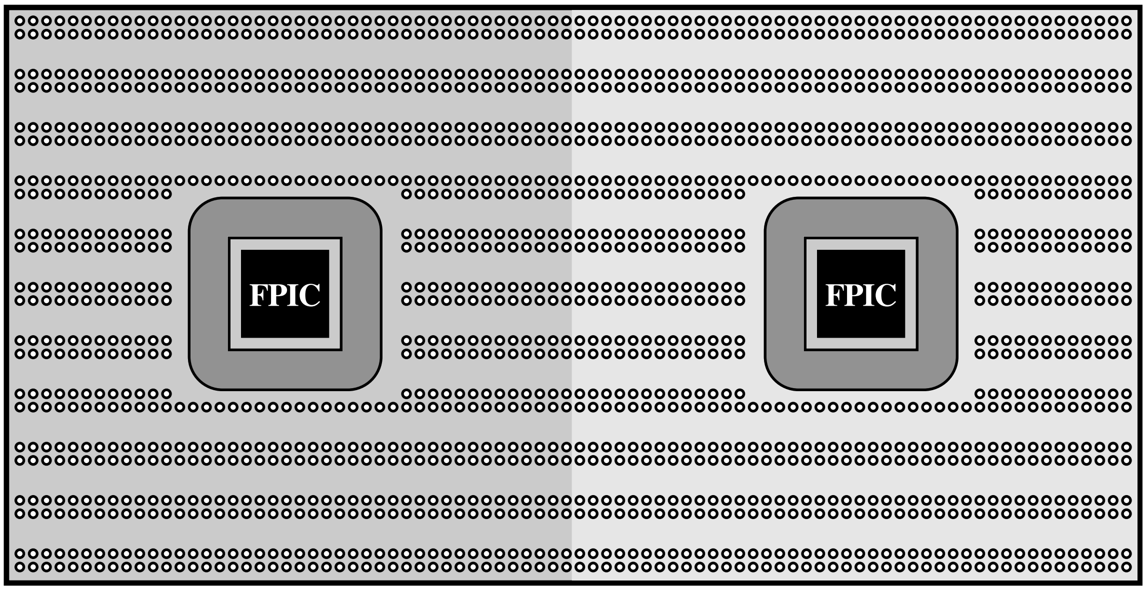


Figure 44. An FPIC based prototyping board [Aptix93a].

Springbok vs. Other Current Approaches

As mentioned earlier, another promising approach for the rapid prototyping of board-level designs is Field-Programmable Interconnects (FPICs), such as those developed by Aptix [Aptix93a]. An FPIC is a chip that can create an arbitrary interconnection of its pins. In the case of the Aptix chips, there are 936 user pins per chip, and the connections formed are passive path wire connections. This latter feature is helpful, since it means that the concerns Springbok has with bi-directional signals such as buses are easily handled with Aptix products. To use these chips for prototyping, Aptix provides circuit boards consisting of a large grid of holes for inserting arbitrary chips (Figure 44). These holes are grouped into sections, with all holes in a section leading to pins on the same FPIC. The FPIC chips communicate between themselves with direct hardwired connections. There are about 50-140 connections between each pair of Aptix FPICs (the exact number varies by the Aptix board chosen). Power and ground routing are handled by separate buses distributed throughout the grid, and jumpers can be used to connect chip pins to the power and ground lines. Clock distribution is supported with special low skew paths in the Aptix chips. Mapping to such a system is easier than for Springbok, since all that is necessary is to partition the chips into sections, minimizing wire crossings at the same time, and then route the signals through the FPICs.

There are several differences between Springbok and an FPIC approach to rapid-prototyping of board-level designs. As mentioned earlier, the software for an FPIC approach is simpler, and there is no difficulty with bi-directional signals. However, these benefits are outweighed by several problems. Most importantly, an FPIC approach is quite expensive. Since both the circuit boards necessary to handle the FPIC's high pincount package, as well as the FPICs themselves, use very aggressive technologies, they are much more expensive than standard chips and boards. In contrast, we expect the cost of a Springbok mapping to be significantly less, since it uses commodity FPGAs and simple circuit boards. Also, many Springbok mappings will use the FPGAs of the underlying routing structure to handle some of the circuit logic. Thus, much of the silicon used to implement Springbok will already be required in the target mapping, decreasing costs.

A second problem with the pure FPIC approach is that of flexibility. As stated earlier, if a mapping requires more pins than the Aptix board allows, there is no way of mapping it short of purchasing a larger board and extra FPICs, adding more expense. In contrast, the Springbok system easily expands to larger sizes, since all that is required to add more capacity is the addition of another baseplate. Thus, instead of requiring several Aptix boards in several sizes, Springbok baseplates can be used in any size mapping. Also, the Aptix boards have a fixed 50-140 pin limit on communication between FPIC sections of the grid. Again, if this number is exceeded, there is no way of mapping the prototype. In Springbok, capacity can be added to deal with any pin limitations. Also, Virtual Wires (Chapter 9), a method usable in Springbok to ease some pin limitations, cannot be directly used in an FPIC system since the FPICs do not have programmable logic with which to multiplex signals. Thus, to use Virtual Wires an FPIC system would have to add extra FPGAs, FPGAs that will also increase the number of pins in the mapping, pins which must also pay the per-pin costs.

Just as many of the Springbok FPGAs will be used to implement logic from the system being mapped, other portions of the FPGAs will be required to slow down chips. Both Springbok and FPIC mappings will operate slower than the target system. As discussed earlier, some chips cannot simply be clocked at a slower rate, but instead require special additional logic to operate correctly. In Springbok, this logic can be accommodated in the FPGAs connecting that chip into the Springbok routing structure. In an FPIC system, extra FPGAs would have to be added to handle this functionality, increasing total pin count. More importantly, these added FPGAs are not directly connected to the chip to be slowed, but instead must communicate with it through the FPIC. The FPIC introduces delays, reducing the portion of the clock period available in the FPGA slowing the chip, which makes it more difficult to properly perform these slowing functions.

A final limitation of the FPIC approach is that it does not support hardware subroutines well. As discussed earlier, in many design environments there are common subsystems used in several different circuits. In Springbok, these subsystems can be built into custom daughter cards, and then used freely in subsequent mappings. In an FPIC system one would need to develop a custom board (an activity Aptix supports with special software) which would contain the subsystem logic as well as the FPICs and the pin grid. This means not only that the resulting new board would be more complex than that necessary for the Springbok system, it also establishes a limit on both the number of such subsystems and the number of chip pins that can be used. This is because both the number of subsystems as well as the size of the pin grid is fixed on any specific Aptix board. Again, in the Springbok system, building a custom daughter card only fixes the contents of that daughter card. Springbok mappings are still free to choose the type and number of custom daughter cards to use, and the mapping can still grow to any size.

While we have spent most of this section discussing the difficulties with using a purely FPIC solution to board-level prototyping, an interesting alternative is to add FPICs into the Springbok framework. FPICs could be included on extender cards, cards which could help ease routing in hotspots of a given mapping. Also, special connections could be built into the baseplates which could lead to a centralized FPIC hub. These baseplate connections would connect a small number of pins on every daughter card position to the centralized FPIC hub. In this way, a network for more efficiently handling long-distance connections could be built without requiring many ribbon-cable extender boards scattered throughout the array. In each of these cases, the FPICs are used sparingly. Hopefully, this could yield a system with all of Springbok's advantages, while harnessing the power of FPICs to perform relatively quick, arbitrary connection patterns. Whether the added functionality is worth the increased cost is unclear, and requires further study.

As mentioned earlier (Chapter 5), there have been many multi-FPGA systems developed. However, most of these are unusable for board-level prototyping, since they do not allow the inclusion of arbitrary chips into the array, or do so to a limited extent. Some systems, such as the G800 or the Paradigm RP, allow a small amount of custom boards to be included into the system. However, many multi-chip systems will require tens or hundreds of chips to be included into the prototype, something that is beyond the capabilities of these systems.

Two systems (which were proposed after Springbok) with a lot of potential for board-level prototyping are COBRA and MORRPH (Chapter 5). Like Springbok, these systems are based around a flexible interconnection topology that allows the system to grow to an arbitrary size. The COBRA system is based around tiles with fixed resource mixes, in a fixed mesh structure. However, adding a prototyping board with sockets for arbitrary devices, as well as boards with long-distance communication links, should not be difficult. The MORRPH system is based around a single tile type, with sockets for arbitrary devices. However, there is nothing in the system construction that precludes adding boards with different functionalities and capacities. Thus, if we combine COBRA's ability to include arbitrary cards with MORRPH's ability to include arbitrary devices onto these cards, we would achieve much of the flexibility of Springbok.

Conclusions

As we have shown, Springbok is a novel approach to the rapid-prototyping of board-level designs that offers many advantages over current systems. Its flexible architecture accommodates a great range of system sizes and topologies. With the ability to solve problems as they occur, Springbok more efficiently uses its resources than fixed FPGA-based systems, which require a very conservative style. Including arbitrary devices and subsystems into the Springbok structure allows even greater efficiency and accuracy. Finally, the use of FPGAs instead of FPICs for the routing structure reduces overall costs, adds flexibility, and more easily handles the functionality necessary to interface to timing-inflexible components.

In the following chapters, we will discuss many of the issues necessary to implement Springbok. Mesh routing topologies are covered in Chapter 7, yielding topologies with higher bandwidth, lower delay, and reduced I/O pin usage. Then we discuss how the external interfaces of a prototype can be supported (Chapter 8). On the software end of multi-FPGA systems, we consider an efficient bipartitioning algorithm (Chapter 10), as well as methods for recursively applying bipartitioning to an arbitrary topology (Chapter 11). We also cover pin assignment software (Chapter 12), which handles part of the global routing step for mapping to multi-FPGA systems.

Chapter 7. Mesh Routing Topologies

Introduction

As discussed in Chapter 3, multi-FPGA systems have great potential for logic emulation and reconfigurable computing tasks. An important aspect shared by all of these systems is that they do not use single FPGAs, but harness multiple FPGAs, preconnected in a fixed routing structure, to perform their tasks. While FPGAs themselves can be routed and rerouted in their target systems, the pins moving signals between FPGAs are fixed by the routing structure on the implementation board. FPICs (Chapter 2) may remove some of the topology concerns from small arrays. However, large FPGA systems with FPICs for routing will still need to fix the topology for inter-FPIC routing.

Chapter 5 has demonstrated that there are many different possible multi-FPGA system topologies. The routing structure used in a multi-FPGA system has a large impact not only on system speed, but also on capacity and system extendibility. Crossbar topologies provide a predictable routing delay, but they sacrifice scaleability and chip utilization. Hierarchical crossbar structures have less predictable routing delays, since signals may have to pass through many FPGAs, but have improved scalability. Mesh connections are scaleable, and may have better utilization than the other structures, but have even worse routing predictability. Although mesh topologies have been criticized due to perceived pin limitations, new techniques such as Virtual Wires [Babb93, Selvidge95] and future high-I/O FPGA packages make meshes a very viable alternative.

Determining the proper routing topology for a multi-FPGA system is a complex problem. In fact, the multiprocessor community has been struggling with similar questions for years, debating the best interconnection topology for their routing networks. The necessary first step is to determine the best way to use a given routing topology, so that an honest comparison between different topologies can be performed. In this chapter, we examine mesh topologies, and present several constructs for more efficient structures. We then provide a quantitative study of the effects of these constructs, and examine their impact on automatic mapping software. Architectural studies of multi-FPGA systems based on crossbars [Chan93b] and hierarchical crossbars [Varghese93] can be found elsewhere.

Mesh Routing Structures

The obvious structure to use for a mesh topology is a 4-way interconnection (Figure 45), with an FPGA connected to its direct neighbors to the north, south, east and west. All the pins on the north side of an

FPGA are connected to the south edge of the FPGA directly to the north, and the east edge is connected similarly. In this way the individual FPGAs are stitched together into a single, larger structure, with the Manhattan distance measure that is representative of most FPGAs carried over to the complete array structure. In this and other topologies an inter-FPGA route incurs a cost in I/O pin and internal FPGA routing resources. The rest of this chapter will attempt to reduce usage of each of these resources. In this way, we not only optimize the delay in the system by shortening routes, but we also reduce the area needed to map a circuit.

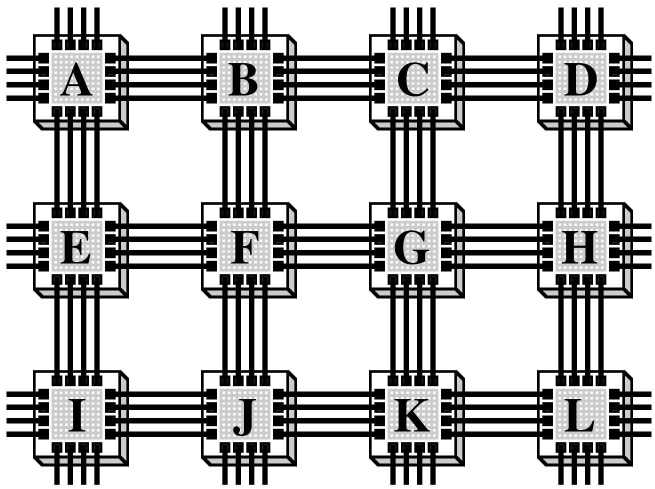


Figure 45. Basic mesh topology.

I/O Pin Usage Optimization

In order to reduce the average number of I/O pins needed to route signals, we can increase the number of neighbors connected to an FPGA. Instead of the simple 4-way connection pattern (Figure 46 top left), we can adopt an 8-way topology. In the 8-way topology (Figure 46 bottom left) an FPGA is not only connected to those FPGAs horizontally and vertically adjacent, but also to those FPGAs diagonally adjacent. A second alternative is to go to a 1-hop topology (Figure 46 right), where FPGAs directly adjacent vertically and horizontally, as well as those one step removed, are connected together. One could also consider 2-hop, 3-hop, and longer connection patterns. However, these topologies greatly increase PCB routing complexity, and wiring delays become significant.

We assume here that all connected FPGAs within a specific topology have the same number of wires connecting them, though a topology could easily bias for or against specific connections. Since the 8-way and 1-hop topologies have twice as many neighbors as the 4-way topology, and FPGAs have a fixed number of pins, each pair of connected FPGAs in these topologies have half as many wires connecting them as in the 4-way case.

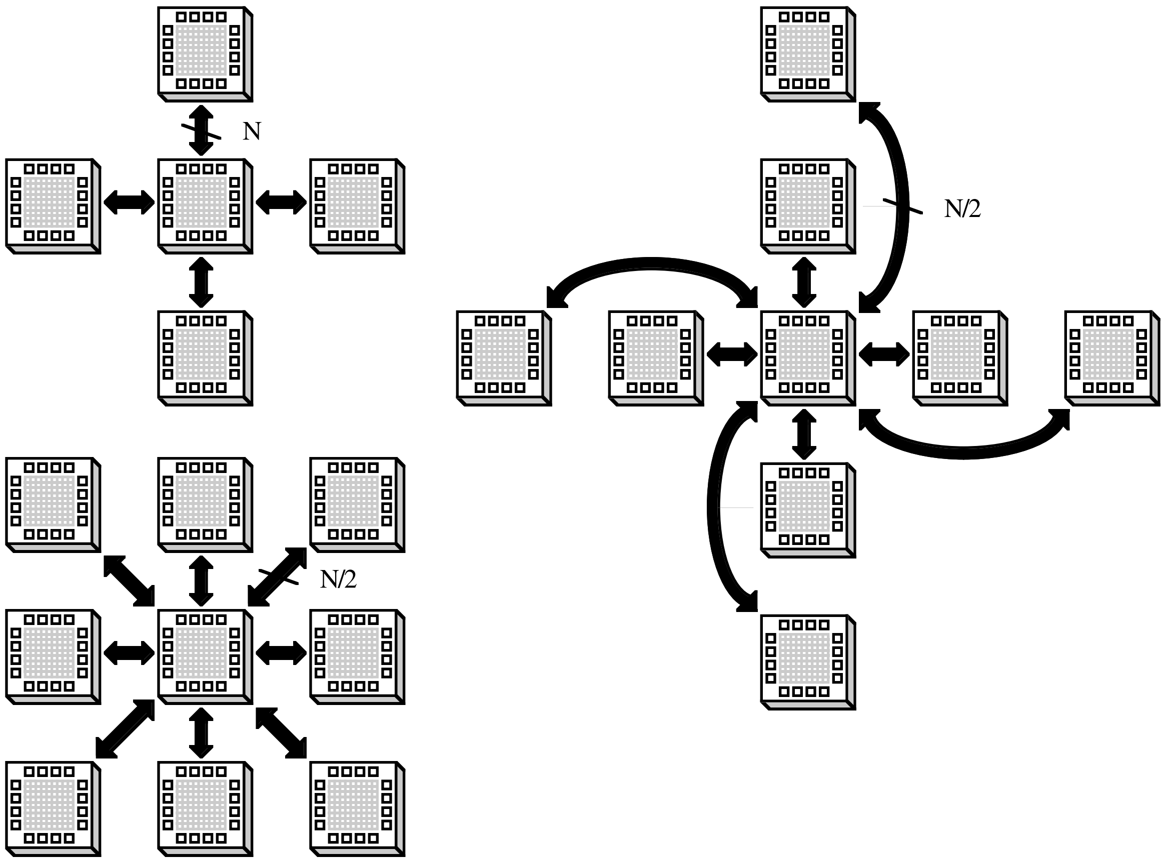


Figure 46. 4-way (top left), 8-way (bottom left), and 1-hop (right) mesh routing topologies.

Switching from a 4-way to an 8-way or 1-hop topology has two major impacts: average I/O pin usage, and bandwidth. Figure 47 shows one quadrant of a mesh under the three different topologies, with each box corresponding to an FPGA. The number indicates how many I/O connections are required to reach that FPGA from the source FPGA at the lower-left corner (shown with an asterisk). As we can see, both the 8-way and 1-hop topologies can reach more FPGAs within a given number of I/O connections than can the 4-way topology. In fact, if we consider an entire mesh instead of a single quadrant, the 8-way can reach twice as many FPGAs as the 4-way within a given number of I/O connections, and the 1-hop topology can reach three times as many FPGAs as a 4-way topology (while within a single I/O connection there are only twice as many neighbors as the 4-way, at longer distances it reaches three times or more). On average a route in a 1-hop topology requires about 40% less I/O pins than a route in a 4-way topology. Another interesting observation is that the 1-hop topology can reach almost all the same FPGAs as the 8-way topology can in the same number of I/O connections. The only exceptions to this are the odd numbered FPGAs along the diagonals from the source in the 8-way topology. What this means is that there is little benefit in using a combined 8-way & 1-hop topology, since the 1-hop topology gives almost all the benefit of the 8-way topology.

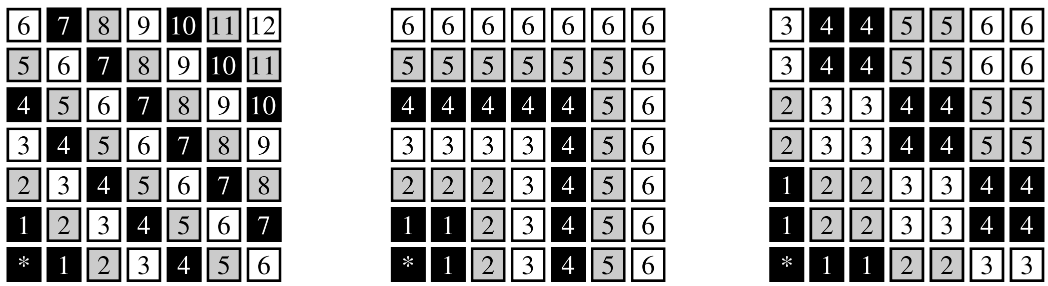


Figure 47. Distance (in number of I/O connections required) to reach FPGAs in 4-way (left), 8-way (center), and 1-hop (right) topologies. Distances are from the FPGA in the lower-left corner.

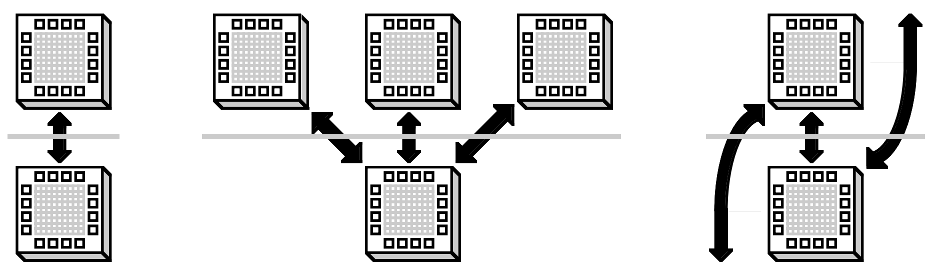
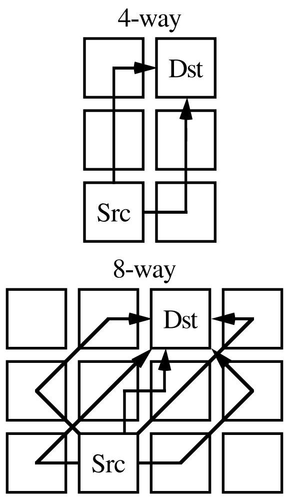


Figure 48. Bisection bandwidth in 4-way (left), 8-way (center), and 1-hop (right) topologies. Three times as many connections cross the bisecting line in the 8-way and 1-hop topologies than in the 4-way topology, though each connection contains half as many wires. This results in a 50% increase in bisection bandwidth.

One would expect the I/O pin usage optimization to come at the price of lower bandwidth in the mesh. However, this turns out not to be the case. The standard method for measuring bandwidth in a network is to determine the minimum bisection bandwidth. This means splitting the network into two equal groups such that the amount of bandwidth going from one group to the other is minimized. This number is important, since if routes are randomly scattered throughout a topology half of the routes will have to use part of this bandwidth, and thus twice the bisection bandwidth is an upper bound on the bandwidth in the system for random routes. In the mesh topologies we have presented, the minimum bisection bandwidth can be found by splitting the mesh vertically or horizontally into two halves. As shown in Figure 48, cutting each row or column in a 4-way mesh splits one pair of connected neighbors, while in an 8-way and 1-hop topology it splits 3 pairs. Since there are three times as many neighbor pairs split, though each pair has half the bandwidth (remember that the 4-way topology has half the number of neighbors, so each pair of neighbors is connected by twice the wires), the 8-way and 1-hop topologies thus have 50% more bisection bandwidth than the 4-way mesh.



8-way vs. 4-way

1.5	1.25	2.0	2.0	2.0
1.5	1.25	1.75	1.75	2.0
1.5	1.25	1.5	1.75	2.0
0.5	0.75	1.25	1.25	1.25
*	0.5	1.5	1.5	1.5

1-hop vs. 4-way

3.0	2.0	2.0	2.0	2.0
1.5	2.0	2.0	2.0	2.0
1.0	1.25	2.0	2.0	2.0
0.5	0.5	1.25	2.0	2.0
*	0.5	1.0	1.5	3.0

Figure 49. Example of the point-to-point fast bandwidth calculation in 4-way (top left) and 8-way (bottom left) meshes. The 8-way routes are only allowed to use three I/O connections, the number of I/O connections necessary to reach the destination in the 4-way topology. Also included is a complete relative bandwidth summary of 8-way vs. 4-way (center) and 1-hop vs. 4-way (right) topologies.

An alternative way to view bandwidth is point-to-point bandwidth. If we simply ask how much bandwidth is available from one specific FPGA to another, then (barring missing connections at mesh edges) all meshes have exactly the same point-to-point bandwidth. This is because there are independent paths ("independent" implying two routes don't share individual I/O connections, though they may move through the same FPGAs) from every wire leaving any source FPGA to any destination FPGA. A more interesting issue is that of fast bandwidth. Specifically, we realize that since a 4-way mesh has twice as many connections to each of its neighbors than an 8-way or 1-hop topology, the 4-way topology can send twice as many signals to that destination using a single I/O connection as can the other topologies. By extrapolation, we would expect that the 4-way topology has more bandwidth to those FPGAs two or more I/O connections away than the other topologies. However, if we allow each topology to use the same number of I/O connections (specifically, the minimum number of I/O connections necessary to reach that

FPGA in a 4-way topology), the 8-way and 1-hop topologies actually have greater fast bandwidth. As shown in Figure 49 left, if we route to an FPGA two steps north and one step east, it requires three I/O connections in the 4-way topology, and there are two independent paths between the FPGAs. If we allow the 8-way topology to use three I/O connections, it actually has five independent paths between the two FPGAs (Figure 49 bottom left). Since each path in an 8-way topology has half the bandwidth as a path in a 4-way topology, the 8-way has 25% more fast bandwidth between these FPGAs. Figure 49 right shows a complete comparison for a quadrant of the mesh, with the numbers given representing the ratio of 8-way vs. 4-way fast bandwidth (center), and 1-hop vs. 4-way fast bandwidth (right). The ratio numbers are for bandwidth between each FPGA and the FPGA at lower left. As can be seen, in all cases except the FPGAs directly adjacent vertically, horizontally, or diagonally, the 8-way and 1-hop topologies have greater fast bandwidth than the 4-way topology, up to a factor of two or more.

Thus, as we have shown, the 1-hop topology reduces average I/O pin usage by 40%, increases minimum bisection bandwidth by 50%, and has greater point-to-point fast bandwidth than the 4-way topology to almost all other FPGAs, up to three times as great.

Internal Routing Resource Usage Optimization

In this section we describe FPGA pin interconnection patterns that minimize FPGA internal routing resource usage. While most of the optimizations described here apply equally well to 4-way, 8-way, and 1-hop topologies, we'll concentrate on 4-way topologies for simplicity. Also, we'll abstract the individual FPGAs into a grid of internal routing resources, with the grid width equal to the distance between adjacent FPGA pins. Finally, we optimize for random-logic applications such as logic emulators and software accelerators. Mappings of systolic or otherwise highly regular and structured circuits may require different topologies.

As described earlier, the obvious way to build a 4-way topology is to connect all the pins on the east side of an FPGA to the pins on the west side of the FPGA directly to the east (the north edge is connected similarly). The problem with this construct is demonstrated in Figure 50 top. Because the pins connecting an FPGA to its neighbor to the east are on the opposite FPGA edge from the pins connected to the neighbor to the west, and pins connected to the south are opposite to pins connected to the north, a signal moving through several FPGAs must traverse the length or width of the intervening FPGAs. Thus, as shown in Figure 50 top, moving from the top of the FPGA at left to the FPGA at right requires a large amount of internal routing resources.

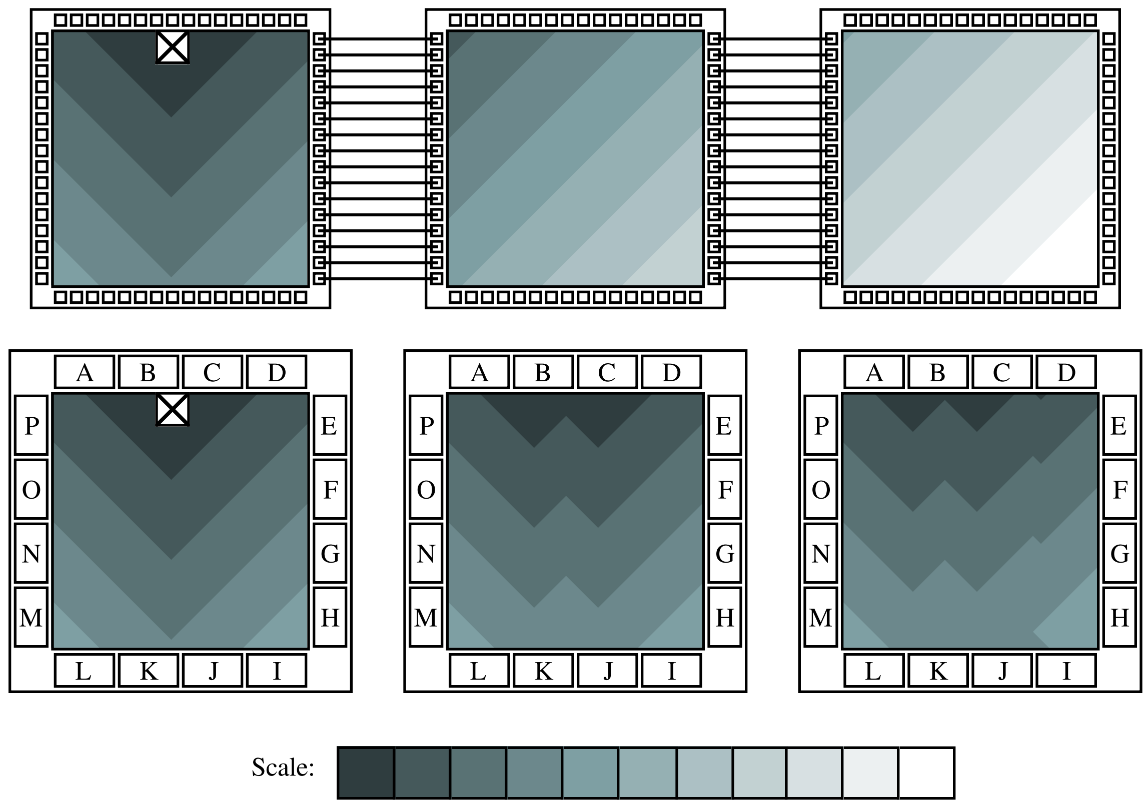


Figure 50. Distances from the X in leftmost FPGA in normal (top) and Superpin (bottom) connection pattern, on a scale from black (shortest) to white (longest). Superpin connections are given by letters, with Superpins with the same letter connected together in adjacent FPGAs.

An alternative is to scatter the pins connecting pairs of FPGAs around the edges of the FPGAs. We form groups called Superpins, and within a Superpin is one pin connected to each of that FPGA's neighbors. Thus, a Superpin in a 4-way topology has four pins, and a Superpin in an 8-way or 1-hop topology has eight pins. Within a Superpin, pins that are likely to be routed together in a mapping are grouped together. Specifically, long-distance signals will usually require pins going in opposite directions to be connected together in intermediate FPGAs. Thus, around the edge of an FPGA in a 4-way topology we order the pins N,S,E,W,N,S,E,W…, and in a 1-hop the pins are ordered NN,SS,EE,WW,N,S,E,W,NN,SS,EE,WW…, where the pin NN is connected to an FPGA two steps north of the source FPGA. In an 8-way topology a long-distance route that doesn't connect together pins going in opposite directions will instead probably connect signals 45 degrees off of opposite (e.g. S and NW or NE). Thus, the connection pattern N,S,SW,NE,E,W,NW,SE,S,N,NE,SW,W,E,SE,NW,N,S… is used, since it puts opposite pins together, and pins 45 degrees off of opposite are at most 2 pins distant.

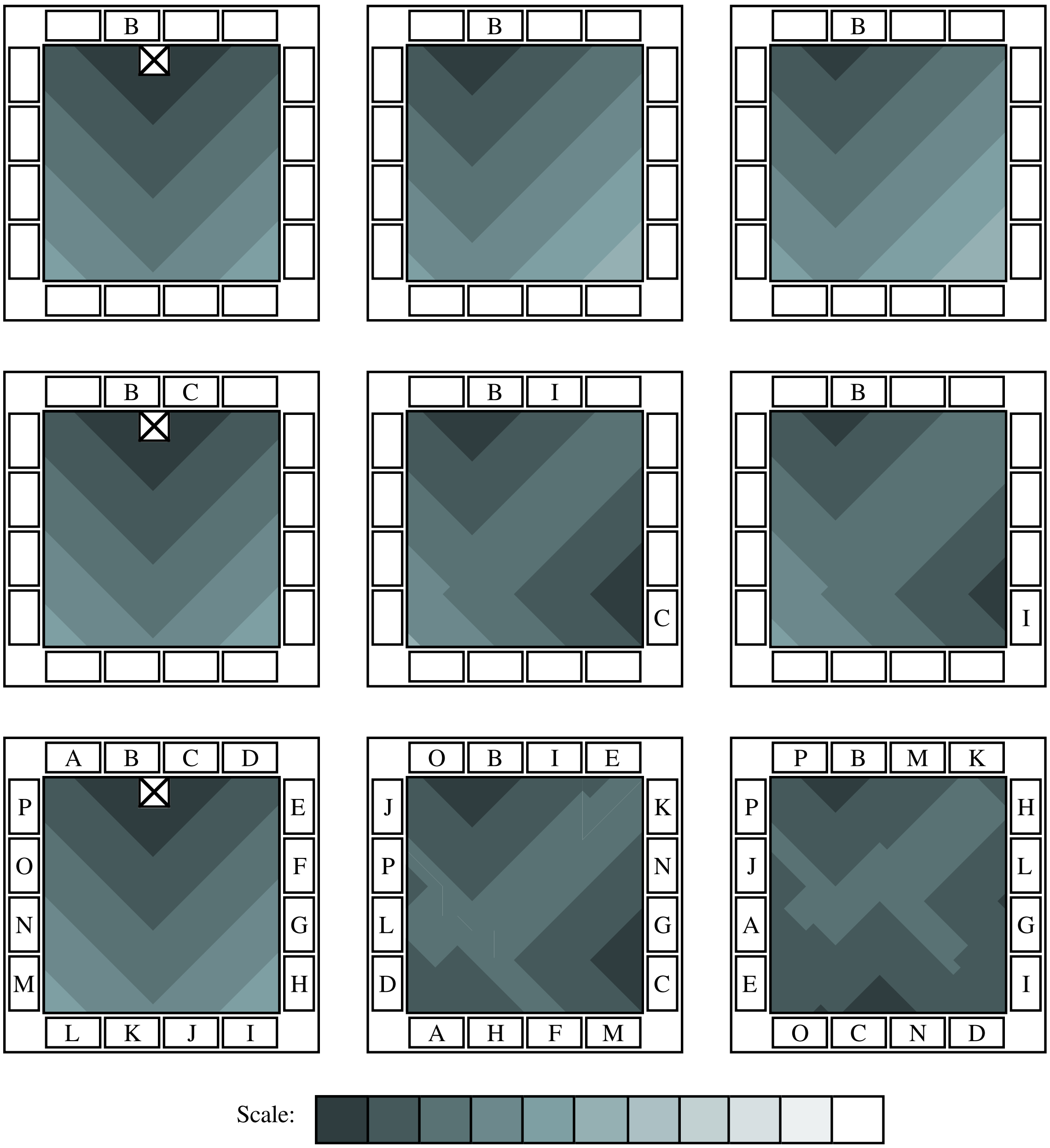


Figure 51. Intuition behind permuted Superpins. A single connection (at top) gives most of the benefit of full unpermuted Superpins. By changing connection **C** to the lower-right corner (middle), more short routes are achieved. Note that connection **I** is simply connection **C** for the middle FPGA. Bottom shows full permuted Superpins, with even shorter routes in further FPGAs. The scale ranges from black (shortest) to white (longest).

As shown in Figure 50 bottom, if we connect the Superpins together in the obvious manner, with Superpins in one FPGA connected to the corresponding Superpins in neighboring FPGAs, we get significant routing resource usage improvements. The Superpin topology almost removes incremental routing resource usage in intermediate FPGAs.

We can do better than the Superpin strategy just presented by realizing that not only can we scatter the connections between neighboring FPGAs around their edges, but we can also scatter the connections to specific sides of these FPGAs around its neighbor's edges. Put differently, instead of connecting Superpins in one FPGA to corresponding Superpins in adjacent FPGAs, we can instead permute these connections to improve routing resource usage. As shown in Figure 51 top, simply making the connection labeled "B" gives most of the benefit of the complete unpermuted Superpin pattern given in Figure 50. Thus, connecting "C" as we did in Figure 50 will give little extra benefit, since the short routes the "C" connection creates will lead to locations that already have short routes due to the "B" connection. If we instead connect "C" in the first FPGA to a location on the lower right edge of the adjacent FPGA (Figure 51 middle), we create short routes to locations that only had long routes through "B". By continuing this approach, we route Superpin connections so that not only are there short routes from one location in one FPGA to its direct neighbors, but we permute the Superpins such that all locations in the source FPGA have short routes to all locations in all other FPGAs (Figure 51 bottom).

An interesting observation is that by having two (identical) permutations in series in Figure 51 bottom, we in fact use less routing resources to reach locations in FPGAs two steps away (the rightmost FPGA) than we need for locations in adjacent FPGAs (middle FPGA in Figure 51 bottom). This effect does diminish with more permutations in series, so that average internal routing resource usage begins to increase again further away from the source, as the incremental cost of routing resources in intermediate FPGAs dominates the gain of additional permutations.

It is possible to generate even better topologies by using several different permutations in a single system. As described later in this chapter, having different permutations in different directions takes advantage of reconvergent paths by having the short routes along one path lead to different locations than the short routes along another path. However, in our experience several different permutations yield at most a 1% improvement in routing costs, and some of these benefits would probably not be realizable by automatic mapping software.

Later in this chapter we present a lower bound on the quality of permutations. Unfortunately, we do not have a simple, deterministic construction method for finding optimum permutations. However, it is fairly easy to write a simple simulated annealing program for permutations which gives very good results. Our admittedly inefficient and unoptimized annealer is less than 500 lines of C code, and has consistently found permutations within a few percent of the lower bounds. Although the runtimes are up to a few days on a Sparc 10, these times are very reasonable for the design of a fixed multi-FPGA system, something that will be done infrequently, and which takes weeks or months to complete.

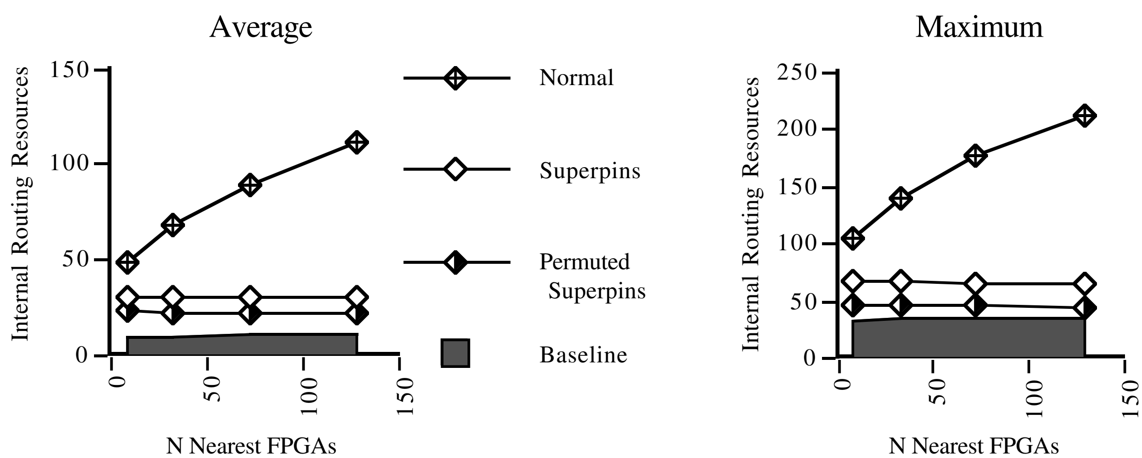
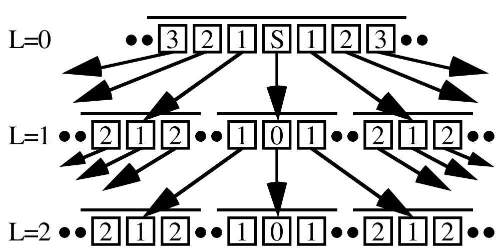


Figure 52. Average (left) and maximum (right) internal routing resource usage from each location in the source FPGA to all locations in the N nearest destination FPGAs in a 1-hop topology.

A quantitative comparison of the internal routing resource usage under the Superpin and Permuted Superpin constructs, all within a 1-hop topology, is presented in Figure 52. These graphs represent the average and maximum resource usage from every point in a source FPGA to every point in the nearest N neighbor FPGAs in the system (FPGAs are represented by grids as described earlier, with 36 pins on a side). An interesting observation is that while the Superpins have a great impact, almost totally removing incremental resource usage in intermediate FPGAs, the Permutations only decrease resource usage by about 28%. One reason for this is the theoretic lower bound ("Baseline") shown above. This lower bound comes from the observation that in any 1-hop topology, a route must use at least enough routing resources to go from the route starting point to the nearest pin on the source FPGA, plus at least one routing resource in each intermediate FPGA, plus enough resources to go between the route ending point and the closest pin on the destination FPGA. As shown, the greatest conceivable improvement over the standard Superpin pattern (assuming we stay within a 1-hop topology) is approximately 60%, and the permutations achieve almost half of this potential. However, when designing a multi-FPGA system, the benefits of permutations must be carefully weighed against the increased board layout complexity.

Bounds on Superpin Permutation Quality

As implied earlier, a "permutation" of Superpins is simply a connection of Superpins between adjacent FPGAs. One way to think about this is that a permutation forms a one-to-one relation between Superpins on adjacent FPGAs. Given this model, we can develop some bounds on how good a permutation can be. We rate permutations based on the average of the minimum distances of each Superpin on the local FPGA to each Superpin on a destination FPGA, measured in Superpin steps. We do not add any cost for inter-FPGA connections, since it is assumed that we will always make the minimum number of chip crossings, and since the permutations do not affect this crossing number. For example, two connected FPGAs with two Superpins each would have an average cost of 0.5, since half of the Superpin cross-product pairs are directly connected (0 Superpin steps away), and the other half are one step away. Note that where the Superpin steps are taken does not matter to our goodness metric. For example, if the best route between a location on FPGA A to a location on FPGA B traverses FPGA C, and uses 2 Superpin steps on A, 1 on B, and 3 on C, the overall distance is considered to be 6.



L \ N	0	1	2	3	4	5	6
0	1	2	2	2	2	2	2
1	1	4	8	12	16	20	24
2	1	6	18	38	66	102	146
3	1	8	32	88	192	360	608
4	1	10	50	170	450	1,002	1,970
5	1	12	72	292	912	2,364	5,336

Figure 53. Diagram of the lower bound calculation (left), and a table of the values (right). The numbers in the pin locations at left indicate the distance of the Superpin from the source S, with the arrows indicating direct connections. These pins form an exponentially expanding fanout tree of possible short paths.

We can generate a lower bound on how good a permutation can be by assuming each FPGA has an infinite number of Superpins. We then determine how many Superpins could possibly be reached in exactly *N* Superpin steps, given that we are moving *L* FPGAs away (that is, we travel through *L-1* intermediate FPGAs plus the destination FPGA). *L=0* indicates that we remain on the source FPGA. We use an FPGA with an infinite number of Superpins because in a finite FPGA, reconverging paths may cause less pins to be reached in a certain distance than is possible in theory. An example of this entire process is shown in Figure 53. Note that we assume that all Superpin steps are taken along the outside edge of the FPGA. While this ignores the fact that the opposite edge of the chip can be reached more quickly by moving directly across the chip than by moving around the outside, these paths will never be minimal in the permutations we use, and thus can be safely ignored. We can generate a formula for the resulting function by realizing that to reach a Superpin in exactly N steps, you must reach a Superpin in the previous FPGA in i steps, $0 \leq i \leq N$, and then move across to the destination FPGA. Since a Superpin in the previous FPGA reached in $i < N$ steps leads to two N-step neighbors (one clockwise, the other counterclockwise), but a Superpin with $i = N$ leads only to one N-step Superpin (the one directly connected to it), we have the formula $F(N,L) = F(N,L-1) + 2 * \sum_{i=0}^{N-1} F(i,L-1)$, where $F(N, L)$ is the number of locations reachable in exactly N steps when moving L FPGAs away. Note that this is equivalent to $F(N,L) = F(N,L-1) + F(N-1,L-1) + F(N-1,L)$. The boundary cases are $F(0,L) = 1$, $F(j,0) = 2 | j > 0$.

To determine the lower bound for a permutation on FPGAs with finite numbers of Superpins, we know that no permutation can reach more pins in N steps than the formula given above, and may reach less due to reconverging paths. Thus, for an FPGA with M Superpins, we get a lower bound by assuming that the first F(0, L) pins are reached in 0 steps, F(1, L) are reached in 1 step, and so on until all pins are assigned. The weighted average of these distances is the optimum value for Superpin permutations for that FPGA size and routing distance. Note that for specific permutations we would have to calculate distances from each source Superpin to each destination Superpin, but since all source pins will have the same optimum distances, the average from one source pin is identical to the average from all source pins. For a specific example, the optimums for an FPGA with 18 Superpins for L=1 is (0*1 + 1*4 + 2*8 + 3*5)/18 = 1.944, for L=2 is (0*1 + 1*6 + 2*11)/18 = 1.556, and for L=3 is (0*1 + 1*8 + 2*9)/18 = 1.444. While this lower bound is not tight for single permutations (provable by brute force search for an 8 Superpin FPGA, where no single permutation is optimum for both L=1 and L=2, though optimums for each separately exist), in our experience permutations exist that either equal or come within a few percent of the lower bound.

While the previous discussion gives a lower bound along a single path of permutations, in order to extend them to a two-dimensional mesh there are a couple observations to be made. First, a permutation must not only work well for signals going from a source FPGA A to a destination B, but also for the reverse route from B to A. However, the inverse of a permutation (the permutation seen by a route moving backwards through the original permutation) is in fact exactly as good as the original permutation. This is due to the fact that the measure of a permutation is the average distance from all sources to all sinks, which is identical to the average distance from all sinks to all sources, which is the measure of the inverse's goodness.

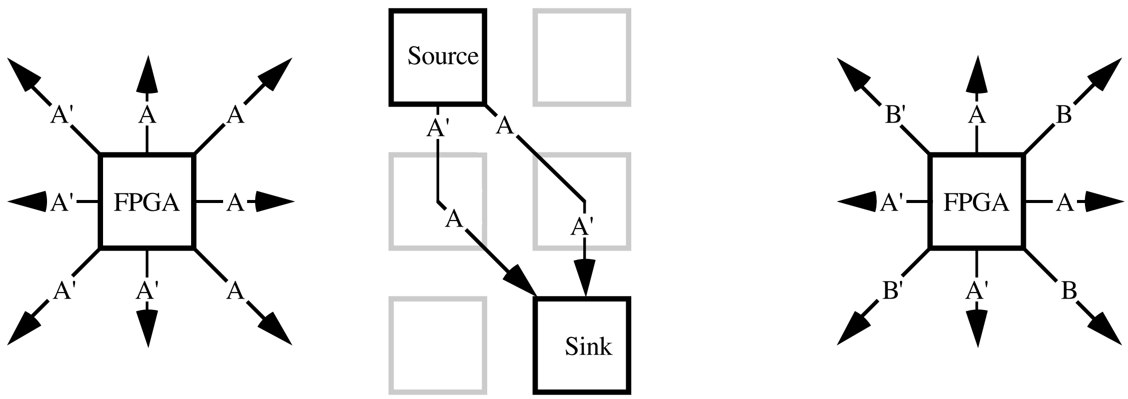


Figure 54. Permutations in 8-way meshes. The pattern at left, with a single permutation A, and its inverse A', causes some paths to use both the permutation and the inverse, as shown at center. The pattern at right, with permutations A and B, avoids this problem. The two permutations are chosen to be independent, so that permutation A and inverse A' work well with both permutation B and inverse B', and vice-versa.

The next issue is that in a two-dimensional mesh, paths do not necessarily travel in a single direction, but may be required to change direction to reach their destinations. However, as illustrated in Figure 54, using a single permutation in a mesh may make a route pass through both a permutation and its inverse. Note that this doesn't result in a total cancellation of the permutation's benefit, since a permutation followed by its inverse has at least the benefit of the inverse permutation. This can be seen by realizing that any signal could just take direct connections through the first permutation, without sideways steps in the source FPGA, and then take the minimal path through the inverse permutation, allowing sideways steps in the middle and end FPGAs. Because we do not penalize for crossing chip boundaries, the average distance through a permutation and its inverse is thus at most the average distance through the inverse permutation only. There exists a single-permutation 8-way topology that avoids the problem of paths with both a permutation and its inverse, but requires different routing from different locations (i.e., while the permutation leading south from one FPGA may be A, another FPGA might have A' leading south). Two permutations, one for the horizontal and vertical moves, and another for the diagonals, can also fix the inversion problem while keeping the same pattern in every FPGA (see Figure 54 right).

The final observation is that for some destinations, there is more than one path between two FPGAs that moves through the minimum number of intermediate FPGAs. For example, to move two FPGAs north in an 8-way mesh, a route can move through either the FPGA directly to the north, northeast, or northwest. We can use this fact to our advantage by choosing combinations of permutations such that the shorter routes through one intermediate FPGA lead to where longer routes through other intermediate FPGAs end (see Figure 55). Thus, every destination will have a short path through some intermediate FPGA, yielding a better set of permutations overall. In this way, if there are P paths between FPGAs, then there could conceivably be P times as many pins reached in N steps as predicted by the above lower bound. Note that in this case, it would actually be advantageous to have different permutations in an 8-way mesh on the two diagonals leaving an FPGA. This is so that a path leading northwest then northeast would have a different permutation order from a path leading northeast then northwest. Three permutations are sufficient, because no minimal path will move by both a horizontal (east or west) and a vertical (north or south) edge, since a single diagonal step would replace these two steps.

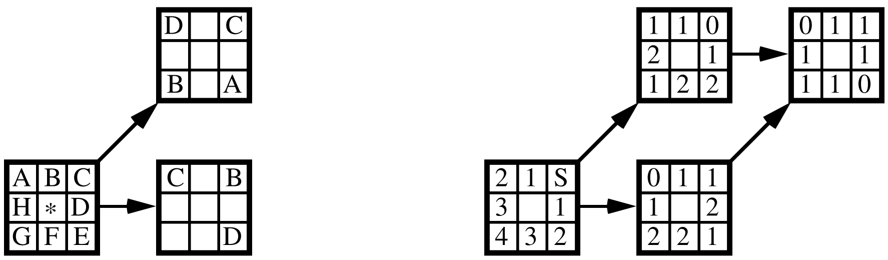


Figure 55. Example of how multiple paths in a two-dimensional permuted mesh decrease routing costs. A portion of the permutations leading northeast and east from the asterixed FPGA is shown at left. Each FPGA has 8 Superpins. At right is the resulting distances from the upper right Superpin (labeled "S") in the lower left FPGA. Notice that in the FPGA at top right there are two different Superpins 0 steps away. This is because the paths leading through different permutation orders lead to different points.

Overall Comparisons

We can make an overall comparison of all the topological improvements, both I/O pin and internal routing resource optimization, by examining inter-FPGA routing delays. As shown in Figure 56, we present the average and maximum delay from every point in a source FPGA to every point in the nearest N neighbor FPGAs in the system. The FPGAs are represented as grids with 36 pins on a side, and the delay incurred in using an FPGA's I/O pin is 30 times greater than a single internal routing resource. These numbers are similar to delays found in the Xilinx 3000 series. Note that this approximates possibly quadratic delays in internal routing resources as a linear function. As shown, an 8-way topology decreases delays by 22% over the standard 4-way topology, while a 1-hop topology decreases delays by 38%. By using the permuted Superpin pattern, the delays are decreased by an additional 25%, reducing overall delays by a total of 63%.

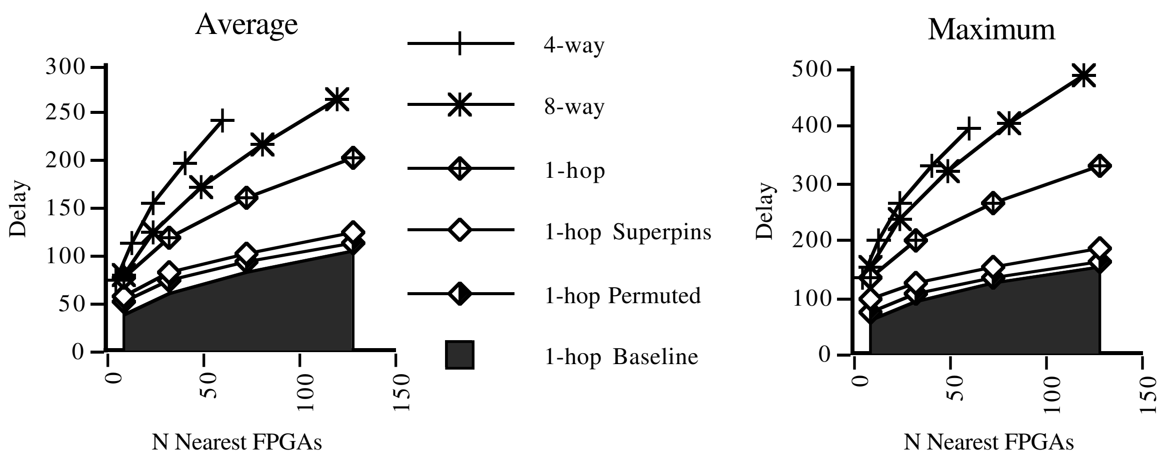


Figure 56. Graphs of average (left) and maximum (right) delay from each location in the source FPGA to all locations in the N nearest destination FPGAs.

While the above numbers give an idea of how the features decrease routing costs at different distances, they ignore the fact that we do not just route a single signal, but in fact have many signals fighting for the same resources. To measure these conflicts, we have used the router developed for the Triptych FPGA project [McMurchie95], which can be retargeted to different domains by altering a routing resource template. This router optimizes both area utilization and delay, making it a good experimental platform for this domain. As before, we abstracted the individual FPGAs to a Manhattan grid, and allowed signals to share edges in this grid. Thus, this model ignores internal routing conflicts. However, these conflicts would have the greatest impact on those topologies that use the most routing resources, especially resources nearest the FPGA center. Thus, ignoring these conflicts will in general decrease the benefit of better topologies, since they use less resources, and the resources they use are primarily at the chip edge. We also do not include signals that begin and end on the same FPGA, because these are unaffected by the inter-chip topologies.

The first two graphs (Figure 57) show the average and maximum cost for signals in each of the routing topologies, assuming a random distribution of sources and sinks of signals across a 5 by 5 array of FPGAs. Note that the same random data sets are used for all topologies at a given size, since this means that all topologies will be subjected to similar routing conditions. Again, moving between chips costs 30 times as much as a single step inside an FPGA, and the FPGAs have 36 pins on a side. The horizontal axis for both graphs is the total number of signals routed in a given trial, and eight trials are averaged together to generate each point. Trials were run at 50 signal increments until the router failed to route all signals.

There is a question of how well some of the topologies handle multi-signal buses and other situations where several signals move between the same sources and sinks. Specifically, one might expect the permutation topologies to have trouble with buses, since while there is a short path between most sources and sinks, there are few if any parallel paths. To determine if this is true, the second set of graphs (Figure 58) is for a population of 5 signal bundles with random sources and sinks, though signals within a bundle share the same source and sink.

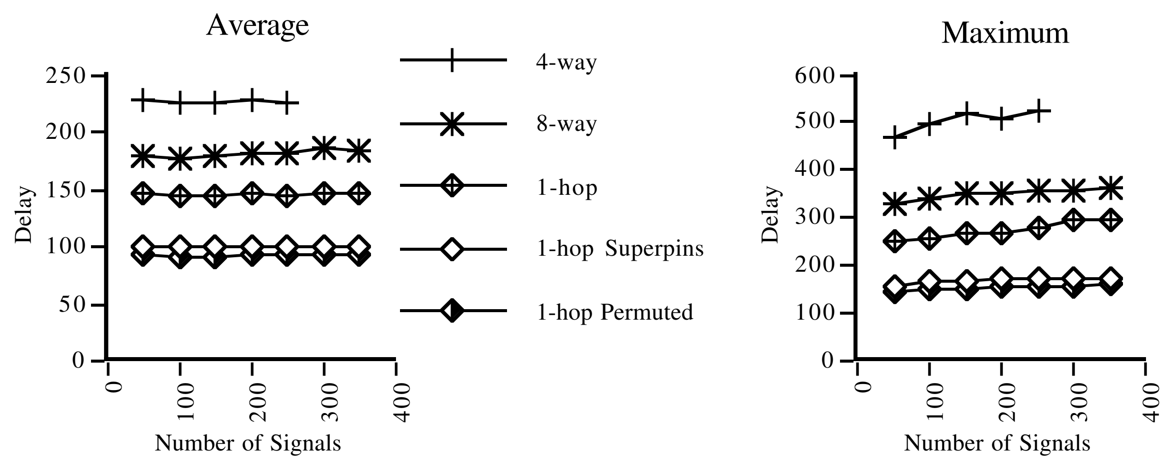


Figure 57. Average and maximum distance of signals under several topologies in a 5x5 array.

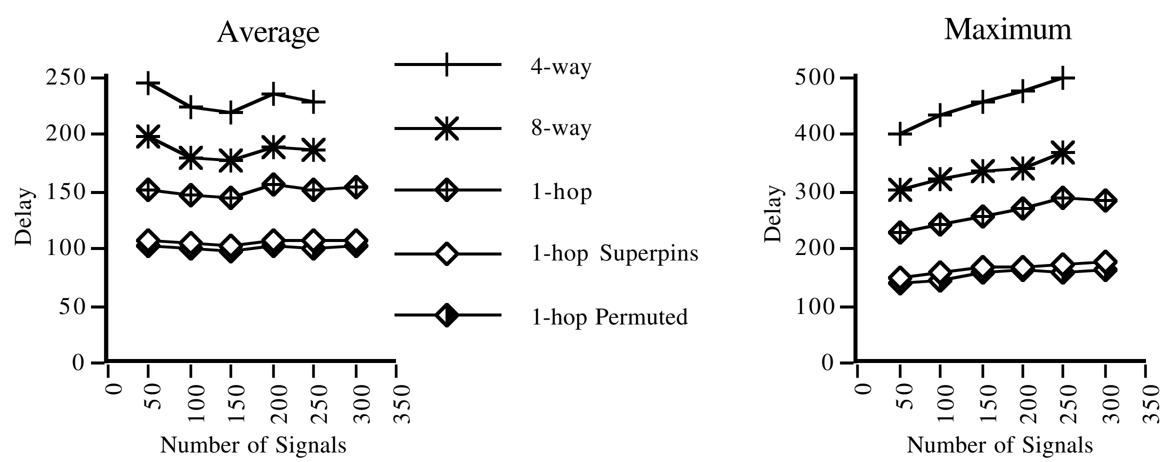


Figure 58. Average and maximum distance of signals under several topologies in a 5x5 array, with all signals in 5-signal buses.

The most striking aspect of the previous graphs is how little congestion seems to affect routing distance. Although samples were taken in 50-signal increments until the router failed, there seems to be little resulting extra routing necessary. Although the graphs of maximum lengths do show increases, this may be mostly due to the fact that a larger number of elements from a random distribution will tend to include greater extremes. The graphs for buses are less flat than the other trials, but this is probably due to the fact that each bus data set has one fifth as many random points, increasing the variance. More importantly, the benefits shown in our original graphs (Figure 56) are demonstrated in actual routing experiments. The 8-way topology is approximately 21% better than the 4-way topology, and the 1-hop is 36% better. Superpins improve the 1-hop topology by about 31%, with permutations saving an additional 5%. Also, in the first set of graphs the 8-way and 1-hop topologies successfully route 40% more signals than the 4-way topology, demonstrating the increase in minimum bisection bandwidth.

Automatic Mapping Tools

Since many multi-FPGA systems will not be used for hand mappings, but instead depend on automatic mapping tools, it is important that a routing topology not only decrease routing costs, but do so in a way that automatic tools can exploit. Since our previous comparisons involved using an automatic routing tool in the congestion examples, and since these experiments yielded distances equivalent to our previous average distance measurements, it is fairly clear that routing tools can exploit our improved topologies. As described in Chapter 12, we have developed a pin assignment tool (similar to a detailed router) for inter-FPGA routing, and the only impact of the improved topologies on this tool is the loss of a slight speed optimization opportunity. Partitioning tools are also easily adapted, since the locality needed for meshs is still the primary concern, though the number of closest neighbors is increased. Thus, automatic mappings tools for standard 4-way meshes should be able to be easily extended to the topologies presented here. More details on multi-FPGA system software can be found later in this thesis, starting with Chapter 9.

Conclusions

We have presented several techniques for decreasing routing costs in mesh interconnection schemes: 1-hop interconnections, Superpins, and Permutations. Through the retargeting of an automatic routing tool, we have demonstrated an overall improvement of more than a factor of 2. While better mesh topologies may be feasible, especially if we allow permutations to operate on individual signals instead of Superpins, theoretical lower bounds (the baseline in Figure 52) prove that there is little room for improvement. Real improvements might come from increasing the direct neighbors of an FPGA from 8 to 26 (a 3-D mesh) or more, but the Superpin and Permutation techniques would still be applicable.

The major open question is whether any mesh system makes sense, or if trees, hypercubes, crossbars, or some other general topology is a better choice. However, if this chapter is any indication, the best implementation of a given topology may not be obvious, requiring a close look at individual candidate topologies before overall topological comparisons can be completed.

Chapter 8. Logic Emulator Interfaces

Introduction

As pointed out in Chapter 4, logic emulation is an important part of the logic validation process. While current systems are capable of performing emulation for single-chip systems, Chapter 6 presents a method for raising these benefits to the system level. In this way, board-level designs can be prototyped, speeding time-to-market for these circuits. Chapter 7 delved into some of the issues of how to build multi-FPGA hardware for emulation and other applications. However, in order to handle chip-level and board-level emulation, the system must not only provide an efficient method for mapping the logic, but also a means for handling the external communications of the circuit under validation.

One of the strongest potential reasons for using logic emulation instead of simulation is that an emulation of a system might be capable of operating in the target environment of the prototype circuit. In this way, the emulation would be exercised with real inputs and outputs, providing a much more realistic evaluation of the circuit's functionality, while providing a fully functional prototype for further experimentation.

Unfortunately, while it is clear that placing the emulation into its target environment is valuable, it is unclear how this can be accomplished in general. Because the environment expects to be communicated with via some protocol, and while the final circuit will obey the protocol, it is not clear that the emulation will meet the protocol's requirements. This problem is twofold: some protocols have timing constraints faster than the emulator can support, and the process of emulation slows the prototype's clock.

Some protocols have timing assumptions much faster than FPGA-based prototypes can deliver. For example, while logic emulators can reach speeds in the hundreds of kilohertz, or even a few megahertz, many communication protocols operate in the tens of megahertz range. Thus, unless the protocol will automatically slow down to the speed of the slowest communicator, the logic emulation will be unable to keep pace with the communication protocol, and thus cannot be placed in the target environment.

Even if the protocol is slow enough that the logic emulator could keep pace, the emulation will still not meet the protocol's timing requirements. This is because a logic emulation does not run at the same speed as the original circuit, and there will inevitably be slowdown of the emulation's clock speed. For example, a 50 MHz circuit built to communicate on a channel that requires responses at 1 MHz will communicate once every 50 clock cycles. If we use logic emulation for this circuit, we might achieve a performance of 10 MHz from the emulation. However, the emulation will still be communicating only once every 50 clock cycles. This achieves a communication performance of 200 KHz, much too slow to respond to a 1 MHz channel. In general, the emulation cannot be altered to communicate in less clock cycles, both because it may be busy performing other work during this time period, and also because so altering the emulation would change its behavior, and we would not be testing the true system functionality.

One solution to this problem is to build a custom interface transducer, a circuit board capable of taking the high rate communication interface that the environment assumes and slow it down sufficiently for the logic emulation to keep pace. This approach has already been taken by Quickturn Design Systems, Inc. with its Picasso Graphics Emulation Adapter [Quickturn93], which takes video output from a logic emulation and speeds it up to meet proper video data rates. This board is essentially a frame buffer, writing out the last complete frame sent by the emulation while reading in the next frame. Although this board is adequate for the specific case of video output, it will be useless for many other communication protocols, or even for video input to the logic emulation. With the large number of communication protocols, it is clear that special-purpose transducers cannot be marketed to handle every need. Expecting the user to develop and fabricate these transducers is also impractical, since now to emulate one chip or circuit board, the user must develop other circuit boards to handle the interfaces. Thus, the user has to do extra development and debugging work to use a system meant to simplify the development and debugging process.

In this chapter, we explore an alternative solution for handling logic emulation interfaces: the development and use of a single generic, standardized interface transducer for most logic emulation interfaces. In the next section, we discuss some of the details of communication interfaces. We also describe a standard, generic transducer board. Then, in the sections that follow we present some case studies, describing how several communication protocols can be mapped to this structure. These include NTSC video, digital audio, PCMCIA, and VMEbus. Finally, we discuss some of the general techniques and limitations of this approach, as well as some overall conclusions.

Protocol Transducers

As shown in Figure 59, communication protocols can be considered to have three levels of requirements: electrical, timing, and logical. At the electrical level, the protocol specifies the proper signaling voltages, the amount of current driven onto signals, termination resistors, allowed capacitances, and other features that interact with the electrical properties of the signaling medium. At the timing level, the protocol specifies the minimum and maximum time delay between transitions on signal wires. At the logical level, the protocol specifies the order of transitions necessary to accomplish various actions, both on the sender's and on the receiver's end of the communications. To properly communicate using a given protocol a system must obey the restrictions of all levels of the specification.

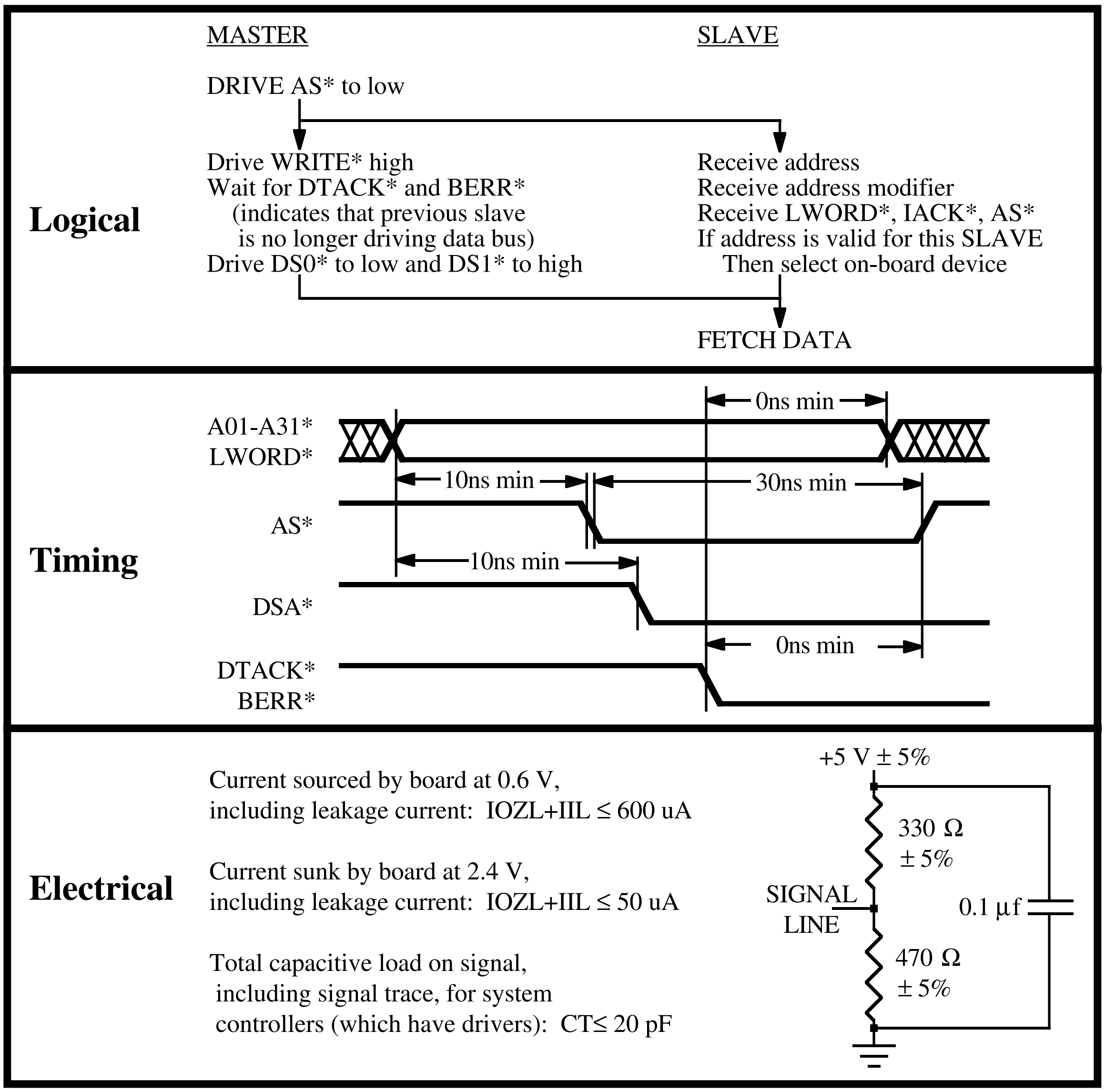


Figure 59. A communication protocol can be considered to have three levels: electrical, timing, and logical. Examples in this figure are adapted from the VMEbus specification [VMEbus85].

Most of the problems for the interfaces of logic emulations occur at the timing level. Since the circuit being emulated has been designed to communicate using the required protocols, it will already be capable of handling the logical level of the protocol. This means that the emulation will already know how to respond to any given set of input transitions, and will generate output transitions in the right order. However, the emulation will send and receive these transitions much more slowly than the final circuit is intended to, since the logic emulator will not be able to achieve the same performance as the custom hardware. The emulator may also not meet the electrical level specifications of the protocol. However, in general the electrical requirements are simple, and easily met by a small amount of interface hardware. In some cases, the protocol will perform all signaling using standard voltage values, with almost no requirements in the electrical level. This is most common when the protocol is used to communicate directly between only two systems. In this case, most programmable logic, including the logic contained in the interface transducer, will meet the electrical level specifications of the protocol. In other cases, there will be more stringent requirements on the emulation. However, most protocols will have a standard set of components that meet their electrical level specification (e.g., standard bus drivers, video input and output coders). As long as the emulator (with the help of a protocol transducer) is capable of meeting the logical and timing level specifications, the standard set of interface components can be used to meet the electrical level of the protocol.

The interface transducer for a logic emulator will be almost entirely focused on meeting the timing level of the specification. While the details of this vary greatly between protocols, there are in general two methods: slowing down the protocol, or filtering the data to reduce the amount of communication.

Many protocols are designed to allow a great deal of flexibility in the speed, cost, and quality of the systems involved in the communication. These protocols use a handshaking between the communicators, with both sides of a communication indicating both when they are ready to start or accept a communication, and also when they have finished with a communication. These protocols allow either end of a communication to slow down the actions taking place, and thus it is easy for a emulation to slow the communication sufficiently for it to keep up.

Some communications are not built with a handshake, and instead assume that any system using a given protocol is able to meet the specified timing requirements. Obviously, while the final circuit will meet these requirements, the emulation may not, and something must be done to compensate. What is necessary is to slow down the data coming into the emulation, and speed up the data leaving the emulation. In general, we wish the emulation to keep up with all incoming data streams, and not simply keep an ever increasing buffer of unreceived communications. Unfortunately, the emulation is capable of processing only a small fraction of the incoming communications. Thus, we must somehow reduce the number of communications coming into the emulation. The general approach is to throw away enough of the communications to allow the emulation to keep pace, while not throwing away any essential information. Obviously, simply throwing away all but the nth clock tick's worth of data will not normally work, since the information coming in will usually be totally corrupted. One possibility, applicable to protocols such as video input, is to throw away all but the nth packet or frame of data. In this way the emulation receives complete frames of video, though from the emulation's point of view the objects in the video are moving very fast. For other situations, we can throw away communications that do not include important data. For example, in a bus-based protocol, many of the transactions will not be intended for the emulation, and most systems can safely ignore them. Alternatively, for a voice recognition system an interface transducer can throw away the "dead air", and retain only those portions of the input surrounding high volume levels.

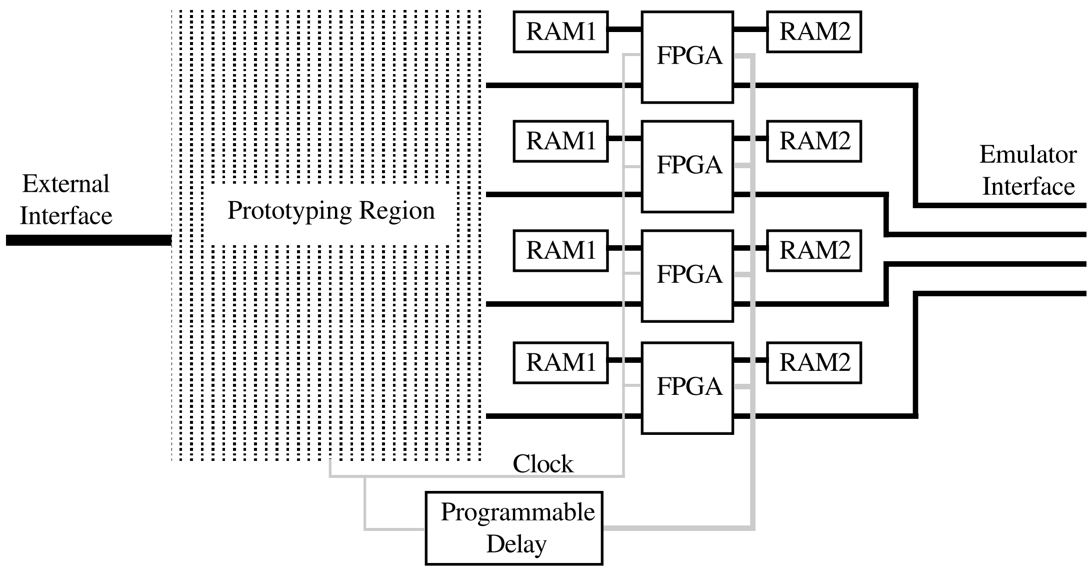


Figure 60. Interface transducer board.

The details of meeting the timing requirements of different protocols can vary greatly. However, the hardware necessary to do so can be generic and simple (see Figure 60): reconfigurable logic to perform filtering and meet timing delays, external RAM for holding intermediate results, and programmable delay lines to generate timing signals. Also important is a small prototyping area to accommodate any required standard components for meeting the electrical level requirements of the protocol. Note that there will often need to be communication between the FPGAs in the transducer so they can coordinate their actions. This can be accomplished by connecting together some of the unused wires leading from the prototyping region to the FPGAs, or from the FPGAs to the emulator interface. In general, the number of such connections will be small, since the number of actions the FPGAs will perform will be very limited. Flexibility is not only important in the number of interconnections between FPGAs, but also in the number and type of components used for a given mapping. Specifically, some interfaces will require only a single small FPGA, while others may need several large FPGAs plus a significant amount of RAM. This flexibility can be handled by using standard sockets for all chips. For example, a socket for the PQFP208 package can accommodate chips ranging from the Xilinx XC4003H (a high-I/O chip with relatively little internal logic) to the XC4025 (the largest capacity Xilinx FPGA) [Xilinx94]. In this way, a single board can be built to handle wide varieties of resource demands.

Example Mappings

In the following sections, we consider several different communication protocols, and describe how they are mapped onto the interface transducer board detailed above. This includes continuous streams of data (video and audio), as well as packetized protocols (PCMCIA and VMEbus).

Video

Video data is transmitted in a steady stream of video frames, arriving at a continuous rate (for the US's NTSC video the rate is 30 frames a second). Because the video arrives at a steady rate we must ignore some frames, throwing away enough data so that the remainder can be handled by the emulator. What remains are complete, sequential frames of video, though any motion in the pictures will seem artificially sped up to the emulator. This should be sufficient for most applications.

As mentioned earlier, to handle the constraints of a protocol we must handle the electrical, timing, and logical levels of that protocol. For video, there are standard chips available for handling the electrical characteristics of NTSC video [Philips92]. These chips take in the analog waveform containing the video data and convert it into a stream of digital data. Most video applications will use these or similar chips, and a transducer can rely on these chips to handle the electrical level of the interface.

The digital video data coming from the standard chipset arrives at a rate of 13.5 MHz, faster than most emulators can handle. To slow the video down, the protocol transducer maintains a buffer of video frames in the memories, filling up buffers as the logic emulator consumes them. In the simplest case, there needs to be only two frames worth of buffer space, one for the frame being sent to the emulator, and one for the frame being read from the external interface. In cases such as motion detection, the interface transducer may need to store several consecutive frames of video, increasing the memory demands. Because of the protocol transducer's socketed design the memory capacity can easily be increased, and the controlling logic in the FPGAs adjusted accordingly. Note that since the transducer will be writing arriving data to the memories at the same time as it is reading data to send to the emulator, there could be memory access conflicts with a single memory system. With the two memories of the proposed transducer interleaving or double-buffering can be used to avoid conflicts.

A similar mapping can handle outgoing video data as well. The primary difference is that since there is less data coming out of the emulator than the external interface expects, we must fill in the time between frames. A solution is simply to repeatedly output the last complete frame of data from the logic emulator until the next frame has been fully received.

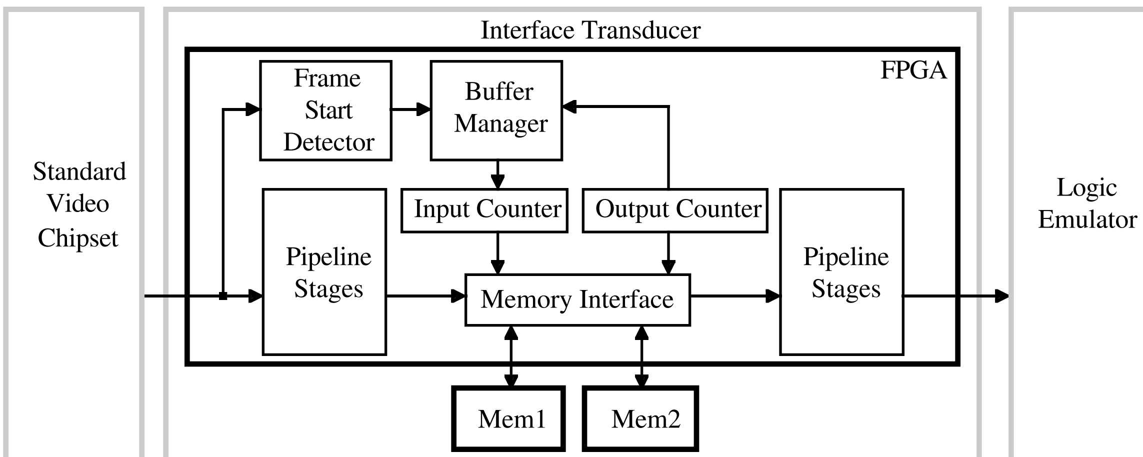


Figure 61. Logic diagram of an interface transducer mapping for incoming video data.

As we have just shown, there is relatively little logic necessary in an interface transducer to handle video data. A diagram of the logic for the input side is given in Figure 61. The logic contains a detector to find the beginning of an incoming frame. This information is passed to a buffer manager, which checks for an empty buffer in memory, and if so the incoming data frame is routed into this buffer. If there are no empty buffers, the frame is discarded. Two counters, one for incoming and one for outgoing data, index into the memory buffers. This is fed to a simple memory interface, which routes requests to the proper memory, and generates the proper read and write signals. To achieve the best performance these steps are all heavily pipelined. Since the data comes in at a steady, predictable pace, usually with no latency constraint, heavy pipelining can easily be achieved.

We implemented the interface transducer mapping as described above by specifying it in Verilog (a high-level design language). It was converted to a Xilinx 4000 mapping with completely automatic mapping tools, and easily achieved the required performance.

Audio

Even though audio is slower than video, and is well within the performance constraints of logic emulators, audio is actually more difficult to handle than video. Even though a mapping running on a logic emulator could meet the performance requirements for most audio applications, the problem is that the system under emulation was not designed to run on the logic emulator, but was in fact designed to run at a higher clock rate. Thus, it will accept or generate audio data on every nth clock cycle. Since the clock cycle on the logic emulator is slower, the system will not be keeping up with the external world.

Since the prototype on the logic emulator will not keep up with the required data rates of audio I/O, it will need an interface transducer to fix the problem. Unfortunately, unlike video, audio is not broken into frames, and there is no clear way to find data that can be ignored. For some situations, such as the input to voice recognition systems, the signal can be expected to have large gaps of silence, and a transducer could look for the "noisy" periods of time. The mapping takes all incoming data and stores it in one of the transducer's memories. The transducer looks for noisy periods, places where the signal amplitude is above some threshold. When such a period is discovered the transducer transfers the noisy period, along with a second or more of data on either side, to the next memory. We need this extra time period to catch the less noisy start and end of the signal, which is why we store all incoming data in the first memory. The data from the second memory is sent to the logic emulator, and can be padded with null data if necessary. In this way we can detect uninteresting data and throw it away.

Other, more common situations are harder to handle. In most cases we wish to preserve all of the audio data in the system. For example, if the system is generating or recording music there will always be interesting data in the audio signal, and there is no "dead air" to throw away. In such situations it is necessary is to save all of the data, perhaps to tape or disk, and use this storage as a buffer to change the speed of the data. For audio this is obviously feasible, since we already store large amounts of audio on today's systems. Thus, it may be worthwhile to augment the transducer hardware with a hard disk interface. Even in protocols and mappings where this feature is not necessary for buffering, the disk could still be used to record the data passing on the channel, serving as a debugging aid. However, for higher data rate signals (i.e., video), there may be too much data to store, and using secondary storage would not be sufficient.

PCMCIA

PCMCIA [Mori94] is a card format and bus protocol used to add peripherals to computer systems. It allows standardized memory cards, modems, disk drives, and other computer cards to be added to any compatible host, including systems from portable computers to Newton MessagePads. The interface can be moderated by standard chips such as the Intel 82365SL DF PC Card Interface Controller (PCIC) [PCIC93], which connects two PCMCIA card slots to an ExCA/ISA bus (see Figure 62).

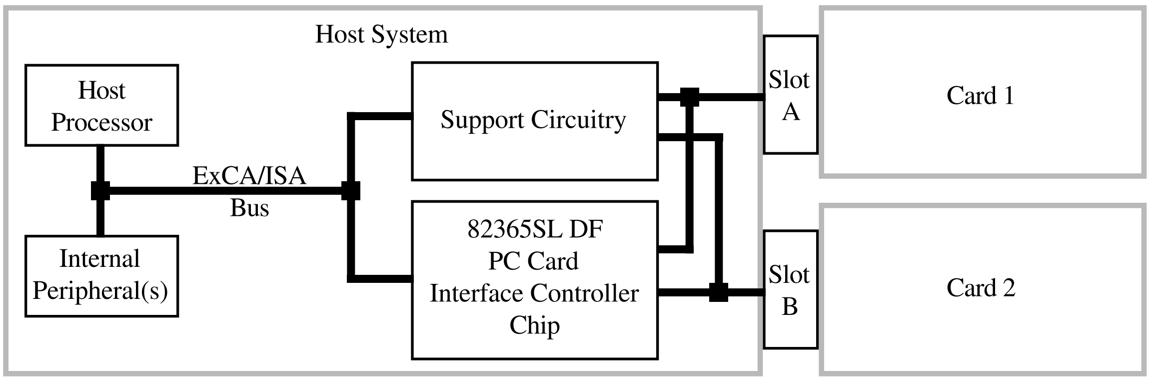


Figure 62. PCMCIA interface built with the Intel 82365SL DF PCIC chip.

Communications to the PCMCIA cards are initiated by the host processor. The PCIC chip routes these messages to the proper card slot based on the address sent by the processor. This card then has the option of asserting a WAIT signal to indicate that it needs extra time to process the current cycle. Once the card has finished processing the current cycle, either by reading in data sent by the host or by writing out requested data, it will deassert the WAIT signal. All signals are then returned to their prior (idle) state, and the system is available for another bus cycle.

Implementing a transducer for either a PCMCIA card or a PCMCIA host system is quite simple. The electrical level of the specification is handled by the Intel 82365SL DF PCIC chip plus some standard support hardware for the host side, while the card side requires little electrical adaptation. The timing level is also easy to meet because it is always obvious what subsystems are involved in a given communication, only one communication can be in progress at a time, and almost all timing constraints on the host system and the PCMCIA cards are minimum delays, not maximums. Note that conversely, the timing constraints on the PCIC chip itself are mostly maximums, making it difficult to emulate the PCIC chip itself. On the card side of the protocol, the transducer can use the WAIT signal to slow down the communication. Performing these actions inside the transducer's FPGAs is quite easy, and is well within the performance capabilities of current devices.

One important concern raised by examining the PCMCIA specification is that while it is trivial to handle the interfaces for the host system and for PCMCIA cards, it is much harder to meet the timing constraints on the controller chip itself. While most of the delays on the host and the cards are minimum delays, there are numerous maximum allowable delays on the controller chip, and these delays are on the order of tens of nanoseconds. The emulator will most likely be unable to meet the required delays, and it is unclear if there is any way to put an emulation of the controller chip into its target environment. The reason for this difference is quite simple. When we design protocols to interconnect many types of circuit boards we must realize that some of the boards will be much more complicated, or made from low-cost parts, slowing down their communications. Thus, inter-board protocols are designed to tolerate slower cards. However, when we design the interfaces to integrated circuits we expect a certain level of performance. The scope of the tasks required from these ICs are often limited and well defined, and thus we can require it to respond to external communications within a certain amount of time. Thus, meeting the external interface timings for system-level prototyping is in general simpler than for chip-level prototyping.

VMEbus Slave

The VMEbus [VMEbus85] is a backplane bus capable of supporting multiple Master (controller) boards, as well as Slave boards. One of the features that makes this system different than the others discussed so far is the fact that communication is bus-based, meaning that many of the transactions a component sees are intended for some other subsystem, and thus that component has no control over the processing of that transaction.

A VMEbus Master requests control over the bus via an arbitration process. Once it has gained control of the bus, it writes an address out onto the bus. Boards in the system are mapped into a common address space, and the address written to the bus will uniquely determine with which board the Master wishes to communicate. This board will respond to the Master's request, either reading or writing a data value onto the bus. The Master may then request additional data transfers from the same target board, either communicating several data values in the same direction at once, or performing a read-modify-write operation.

Slowing down the bus communications for an emulated slave can be difficult, primarily because the emulation will not be involved in all transactions that occur on the bus, and thus many of these communications will not give the emulation a chance to slow them. For transfers actually involving the emulation we can simply delay the interface signals to re-establish the proper minimum delays from the emulation's perspective. Communications that do not involve the emulation cannot be handled so simply, because the system does not wait for the emulation to process a transaction that is not intended for it. However, the emulation may not function correctly if these transactions proceed faster (in its slowed frame of reference) than the required minimum delays. There are two solutions to this problem. First, since the ranges of addresses a board responds to are fixed, we can program the interface transducer to simply ignore all transactions that are not intended for the emulation. A second choice is to allow the interface transducer to present a transaction to the emulation regardless of what board it is destined for, and delay it sufficiently to meet the minimum delays in the emulation's time frame. If the transaction is not destined for the emulation, this and other transactions can complete while the emulation is still receiving the original transaction. These transactions are ignored, and the interface only presents a new transaction when the last transaction has been completely received by the emulation. Since the transactions are very simple, requiring less than 100 bits, the data can easily be cached inside the FPGAs in the transducer. The emulation will not miss any transactions that are intended for it, since such a transaction would still be waiting for it when it completes the previous transaction. This second solution has the advantage that the emulation experiences transactions that are not intended for it, thus allowing the designer to test that the board does indeed ignore communications not intended for it. Note however that a emulation cannot "snoop" on the bus, attempting to overhear and capture communications not intended for it, since there will inevitably be transactions communicated on the bus that must be filtered out.

One significant problem with the VMEbus specification is that it contains several built-in time-outs. Specifically, if a board does not respond in a reasonable amount of time to a bus communication or arbitration, the system may decide to rescind the transaction. While these delays should be quite large, since these time-outs are only meant to free the bus in exceptional situations, it is possible that the emulation will operate too slowly to meet these upper bounds. In such a case, the logic that administers these time-outs will need to be modified or removed to allow proper operation of the emulation.

Conclusions

As we have shown, many logic emulator interfaces can be handled by a simple, generic interface transducer board. The board consists of FPGAs, memories, programmable delays, and perhaps an interface to a secondary storage device. This board is responsible for meeting the timing level specifications of the protocol. The electrical level of the protocol is met by standard chips where necessary, and the logical level is met in the emulator itself. The FPGAs on the transducer perform filtering and padding on the data stream, and make sure all required timing relationships are upheld. The memories are used for temporary storage and buffering, with two memories per FPGA to avoid read/write conflicts. The programmable delays are used to generate timing signals, both for the interfaces to the memories, and to help the FPGA meet the protocol's timing constraints. While the protocol transducer mappings can be somewhat involved, we have found that they can be quickly developed in a high-level language, and automatically translated into FPGA logic.

In this chapter, we described how this generic board can be used to handle NTSC video, digital audio, PCMCIA, and VMEbus. From these examples several general techniques have emerged, techniques that are applicable to many other situations. Interface transducers must somehow slow the external interface down to the speed of the logic emulator, either by delaying the communication, or filtering away data. Many protocols obey a strict handshaking, or have explicit wait or delay signals. In these cases, the incoming data rate can be slowed down to meet the emulator's speed simply by properly delaying these signals. In some cases (such as the VMEbus) data will be sent on a common medium between many subsystems, but only the sender and receiver of this data have control of the handshaking. In these situations the interface transducer can simply ignore data not intended for the emulation.

Other protocols do not have an explicit method for slowing the incoming data. Video and audio are good examples of this. In these cases it is necessary to filter away some of the data that is arriving, throwing away less interesting data, while storing the more important portions. In some cases this may require a large amount of buffer space, possibly even the use of secondary storage.

Timing constraints on individual signals must be respected by the protocol transducers. However, it is in general true for system-level prototyping that most of the timing constraints on the system are minimums, meaning that the emulator should not respond too quickly. Obviously this is not a problem. Note that if the emulation is of an individual chip, meeting the timing constraints can be much more difficult. This is because protocols are more likely to impose maximum response-time constraints on individual chips than on complete systems. The one exception to this is fault-tolerance features, features that impose a large maximum response time constraint on the system, reasoning that only a defective system will ever exceed these delays. In many cases, the emulation will meet these delays; in others, there is no choice but to disable the fault-tolerance features.

Although we did not encounter such protocols in our study, there are some types of interfaces that simply cannot be handled by any protocol transducer. Specifically, the logic emulation will run more slowly than the target system, and this will significantly increase the response-time of the system. Thus, the protocol must be able to tolerate large communication latencies. Also, there must be some method of reducing the incoming data rate. If the interface delivers data at a steady rate, and all data must be received by the system, there will be no way to reduce the data rate.

As we have shown, the design of a generic interface transducer is possible, and can meet most of the interface demands of current and future emulation systems. With a small amount of design effort, which can in general be carried out in a high-level design language, such systems can be run in their target environment, greatly increasing their utility.

Chapter 9. Multi-FPGA System Software

In Chapter 5 we presented numerous multi-FPGA systems, and described the varied roles of these systems. In the chapters that followed we described several important aspects of multi-FPGA hardware. However, hardware is not enough. In order for multi-FPGA systems to achieve widespread use, they not only require an efficient hardware implementation medium, but also a complete, automatic software flow to map circuits or algorithms onto the hardware substrate. Similar to a compiler for a standard programming language, the mapping software for a multi-FPGA system takes in a description of the circuit to be implemented, and through a series of transformations creates an implementation of the circuit in the basic instructions of the hardware system. In a multi-FPGA system, this implementation is programming files for the FPGAs in the system. This chapter will describe multi-FPGA system software in general, including some existing techniques from the literature. In the chapters that follow, we will detail the contributions of this thesis to multi-FPGA system mapping software.

Before we discuss the specific steps necessary to map onto a multi-FPGA system, it is necessary to consider some of the features of a multi-FPGA system that impact upon this software flow. One of the most important concerns in multi-FPGA systems is that while the FPGAs are reprogrammable, the connections between the FPGAs are fixed by traces on the circuit board. Thus, not all FPGA may be interconnected, and communication between FPGAs must be carried on these limited resources. If the source and destination of a route are on different FPGAs, and the FPGAs are not directly connected, this signal will need to traverse intermediate FPGA(s). This adds extra delay to the routing, and uses up multiple FPGA I/O pins. This latter constraint is the major bottleneck in many multi-FPGA systems, with I/O resource constraints limiting achieved logic utilization to 10%-20%. Thus, the primary concern of mapping tools for a multi-FPGA system is limiting the amount of I/O resources needed by a mapping. Not only does this mean that most signals should be kept within a single FPGA, but also that those signals that need to be communicated between FPGAs should be communicated between neighboring FPGAs. Thus, the mapping tools need to understand what the topology of the multi-FPGA system is, and must optimize to best fit within this routing topology. Note that some topologies use crossbars or FPICs, devices meant to ease this I/O bottleneck. However, even these chips have finite I/O resources, and the connections between them and the FPGAs are fixed. Thus the restrictions of a fixed topology occur even in these systems.

Since the routing topology is a critical feature of a multi-FPGA system, it would be tempting to come up with a topology-specific mapping solution. However, as was shown in Chapter 5, there are numerous different multi-FPGA topologies, and solutions that are only appropriate to a single topology are obviously of limited utility. Also, there are systems such as Springbok (Chapter 6), as well as others, which have a very flexible topology, with fairly arbitrary resource mixes. If the mapping solution cannot adapt to an arbitrary topology, it will not be able to meet the demands of these types of systems. Finally, as we will discuss later, incremental updates may require that only a subset of a multi-FPGA system be remapped, and this subset can be any arbitrarily complex portion. The shape of these subsets can be difficult to predetermine, and requires that the mapping tools adapt to an unpredictable topology.

Not only must the tools be topology-adaptive, but in many circumstances they must also be fully automatic. In the case of logic emulation, the user of the system has no desire to learn all the details of a multi-FPGA system, and will be unwilling to hand-optimize their complex design to a multi-FPGA system. The system is meant to speed time-to-market of the system under validation, and without an automatic mapping solution the speed benefits of an emulation will be swamped by the mapping time. Also, for a multi-FPGA system targeted to general software acceleration, the users of the system will be software programmers, not hardware designers. While there are niches where a hand-optimized solution may be the right answer, for many other domains a complete, automatic mapping system is a necessity.

The final major constraint is that the performance of the mapping system itself is an issue. A multi-FPGA system is ready to use seconds after the mapping has been developed, since all that is necessary is to download the FPGA configuration files to the system. Thus, the time to create the mapping dominates the setup time. If the mapping tools take hours or days to complete, it is difficult and time-consuming to make alterations and bug fixes to the mapping. This is especially important for rapid-prototyping systems, where the multi-FPGA system is part of an iterative process of bug detection, correction, and retesting. If the mapping time is excessive, the multi-FPGA system will be used relatively late in the design cycle (where bugs are few), which greatly limits their utility. With a mapping process that takes only minutes, it becomes possible to use the benefits of logic emulation much earlier in the design cycle, increasing the usefulness of a multi-FPGA system.

Note that there is a major tradeoff to be made between the conflicting goals of high performance mapping tools and high quality mappings, where mapping quality is primarily measured in I/O resource usage. Many of the algorithms that can create the best quality mappings (such as simulated annealing [Shahookar91]) take a significant amount of time to complete. Faster mapping tools will often achieve their speed by sacrificing mapping quality. The proper answer to this dilemma may not be a single tool that strikes some compromise between these goals, but instead a pair of tools. One tool would create the highest quality mappings, and may take multiple hours to complete. Another tool would create a mapping much more quickly, but by sacrificing mapping quality to some degree. With the pair of mapping tools, we can adopt the following methodology. In some cases the resources are not the primary concern, either because we are testing only a subset of the complete circuit, but we have the resources for the entire circuit, or because the logic emulator users decided to buy excess hardware to avoid mapping quality constraints. In these situations we use the high speed, moderate quality mapping tool. When resources are at a premium, the high quality mapper is applied, perhaps running overnight. This will create the best possible mapping, and will probably leave some excess resources scattered around the system. Users will then work with this high quality mapping, and will discover bug fixes and other modifications necessary to the mapping. Instead of iterating back to the high quality mapper, we can apply the high speed, moderate quality mapper only to the portions of the multi-FPGA system affected by the modification. In this way only a small portion of the mapping is degraded by the lower quality mapper, though hopefully the mapping will still fit within the excess resources left in the system. In this way the system is quickly back up and running. These "corruptions" of the high quality mapping are allowed to continue through the day or week. Then, when the users can afford to have the system taken down overnight or over the weekend, the high quality mapper can be rerun, developing a fresh base for further incremental alteration. In this way the user avoids constant long latency remapping steps, yet still achieves high quality mappings, although with a slight degradation. Thus, a pair of mapping systems, one high quality but slow, another high speed but lower quality (and which can work on arbitrary subsets of the system) better serves the needs of the user than a single system.

Multi-FPGA System Software Flow

The input to the multi-FPGA mapping software may be a description in a hardware description language such as Verilog or VHDL, a software programming language such as C or C++ [Agarwal94, Wo94, Galloway95, Iseli95], or perhaps a structural circuit description. A structural circuit description is simply a representation of a circuit where all the logic is implemented in basic gates (ANDs, ORs, latches, etc.), or in specific, premade parts (i.e., microprocessors, memories, etc.). Programming language descriptions (Verilog, VHDL, C) differ from structural descriptions in that the logic may be described more abstractly, or "behaviorally", with the functions described by what needs to be done, not by how it should be implemented (note that structural descriptions may be found in higher-level languages as well, but these portions can be handled identically to complete structural descriptions). Specifically, an addition operation in a behavioral description would simply say that two values are added together to form a third number, while a structural description would state the exact logic gates necessary to perform this computation. To implement the behavioral descriptions, there are automatic methods for converting such descriptions into structural circuit descriptions. Details of such transformations can be found elsewhere [McFarland90].

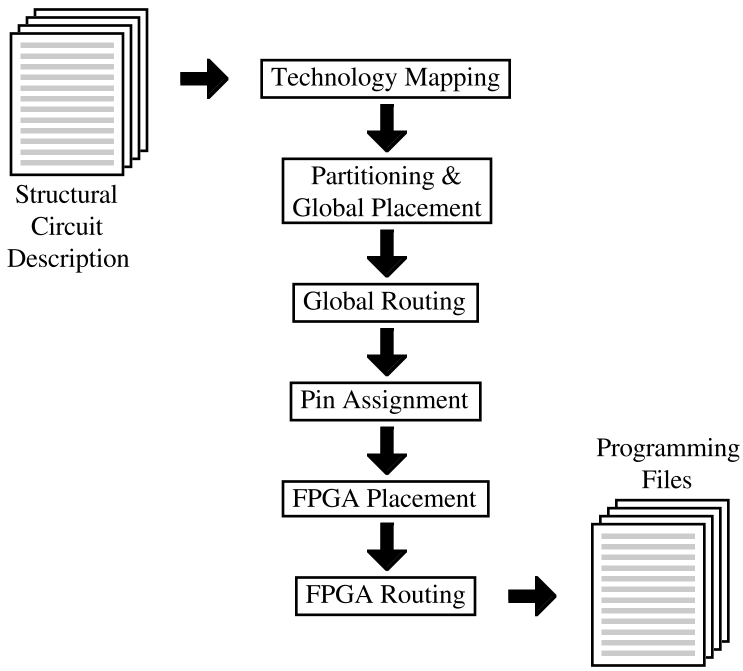


Figure 63. Multi-FPGA system mapping software flow.

To convert from a structural circuit description to a multi-FPGA realization requires a series of mapping steps (Figure 63). For elements assigned to specific chips (memories, microprocessors, etc.), it is assumed that the implementation hardware will include such devices, and there is little processing necessary to map these elements. If such chips are not available, the logic must be converted into basic gates so that it can be handled similarly to the rest of the random logic in the source description. This requires that there be a description of the logic in the chip, and this description will be inserted to replace the chip in question.

To map the random logic requires technology mapping, partitioning & global placement, global routing, and FPGA placement and routing. FPGA placement and routing, as well as technology-mapping, are identical to the tools used for single FPGAs, and were described in Chapter 2. In the pages that follow, we will first give a brief overview of the process. A more in-depth discussion of each of the steps appears later in this chapter.

The first mapping step is technology mapping. Technology mapping restructures the logic to best fit the logic blocks in the FPGAs. This primarily requires the grouping together of small gates to form larger functions that better use a single function block's resources, though it may be necessary to split high fanout gates, resynthesize logic, and duplicate functions to produce the best implementation.

After technology mapping, partitioning takes the single input circuit description and splits it into pieces small enough to fit into the individual FPGAs in the system. The partitioner must ensure not only that the partitions it creates are small enough to fit the logic capacity of the FPGAs, but must also ensure that the inter-FPGA routing can be handled within the constraints of the multi-FPGA routing topology. In general global placement, the process of assigning partitions to specific FPGAs in the system, is combined with the partitioning stage. Otherwise, it is unclear where a given partition resides within the multi-FPGA topology, and thus it is difficult to properly optimize the interpartition connectivity.

After partitioning and global placement, global routing handles the routing of inter-FPGA signals (i.e., the signals that need to be communicated between partitions). This phase can be broken up into abstract global routing and detailed global routing (pin assignment). Abstract global routing (hereafter referred to simply as global routing) determines through which FPGAs an inter-FPGA signal will be routed. Pin assignment then decides which specific I/O pins on each of the FPGAs will carry the inter-FPGA signals.

Once partitioning, global placement, global routing, and pin assignment are completed, all that is left is the placement and routing of the individual FPGAs in the system. When this is completed, there are now configuration files prepared for each FPGA in the system. Downloading these files to the multi-FPGA system then customizes the multi-FPGA system, so there will then be a completed realization of the desired functionality.

Partitioning and Global Placement

As mentioned earlier, the user of a multi-FPGA system specifies the desired functionality as a single, large structural circuit. This circuit is almost always too large to fit into a single FPGA, and must instead be split into pieces small enough to fit into multiple FPGAs. When the mapping is split up, there will some signals that will need to be communicated between FPGAs, because two or more logic elements connected to this signal reside on different FPGAs. This communication is a problem for a multi-FPGA system, because the amount of I/O resources on the FPGAs tends to be used up long before the logic resources are filled.

Because I/O resources are the primary limitation on logic capacity in a multi-FPGA system, the primary goal of the partitioner, the tool that splits logic up into FPGA-sized partitions, is to minimize the communication between partitions. There have been many partitioning algorithms developed which have as a primary goal the reduction of inter-partition communication, and which can split a mapping up into multiple pieces. Unfortunately, these algorithms are not designed to work inside a fixed topology. Specifically, most multi-way partitioning algorithms, algorithms that break a mapping into more than 2 parts, assume that there is no restriction on which partitions communicate. They only seek to minimize the total amount of that communication, measured as either the total number of nets connecting logic in two or more partitions (the net-cut metric), or the total number of partitions touched by each of these cut nets (the pin-cut metric). These are reasonable goals for when the partitioner is run before the chips are interconnected, and is used in cases where someone is building a custom multi-FPGA system for a specific application (note that Chapter 11 discusses some fixed topologies where these methods are also appropriate). However, most multi-FPGA systems are prefabricated, with the FPGAs connected in a routing topology designed for general classes of circuits, and these connections cannot be changed. Thus, some FPGAs may be interconnected in the topology, and others will not, and the partitioner needs to understand that the cost of sending a signal between pairs of FPGAs depends on which FPGAs are communicating.

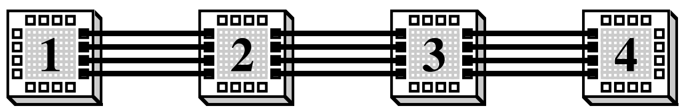


Figure 64. Example system for topological effects on multi-FPGA system partitioning.

For example, consider a linear array of 4 FPGAs, numbered **1**-**4** (Figure 64). A signal communicated between FPGAs **1** & **2** will consume two I/Os, one on each FPGA, since the FPGAs are directly connected. A signal between FPGAs **1** & **4** will consume six I/Os, two between **1** & **2**, two between **2** & **3**, and two between **3** & **4**. Thus, when partitioning onto this linear array of FPGAs, it is better to have two signals being communicated, one between **1** & **2**, and one between **2** & **3**, than it is to have a single signal communicated between **1** & **4**. Thus, in order to do a reasonable job of partitioning onto a multi-FPGA system, the partitioner needs to understand the topology onto which it is partitioning. For this reason the step of global placement, the assigning of partitions to specific FPGAs in the system, is often done simultaneously with partitioning. Otherwise it is hard for a partitioner to understand the cost of communication within a given topology if it does not know where the partitions it is creating lie within this topology.

As mentioned earlier, the best method for mapping onto a multi-FPGA system is probably a pair of mapping approaches, one high quality but slow mapper, and another high speed mapper that can work on arbitrary subsets of the system. For the high quality approach we believe a system such as that developed by Roy and Sechen [Roy93] is the proper approach. This algorithm applies simulated-annealing onto a circuit being mapped to a multi-FPGA topology, and performs simultaneous partitioning and global placement. During its processing it knows which FPGA a given logic element will be assigned to, and it performs routing of the inter-FPGA signals to determine how expensive a given connection pattern is, and to look for severe congestion points in the system. In this way the partitioner optimizes to a specific multi-FPGA topology, and can attempt to find the global optimum for this mapping situation. Unfortunately, simulated annealing is a very slow process, especially when it is necessary to also perform routing as part of the cost metric. Thus, while this algorithm provides an ideal method to perform mapping onto a multi-FPGA system when mapping time is not an issue, it is too slow to be used for incremental updates and local bug fixes, or for situations that are not resource constrained. Note that Vijayan has presented an algorithm with a similar goal [Vijayan90]. Unfortunately, the algorithm is exponential in the number of partitions, making it unsuitable for most complex multi-FPGA systems, which may have tens to hundreds of FPGAs.

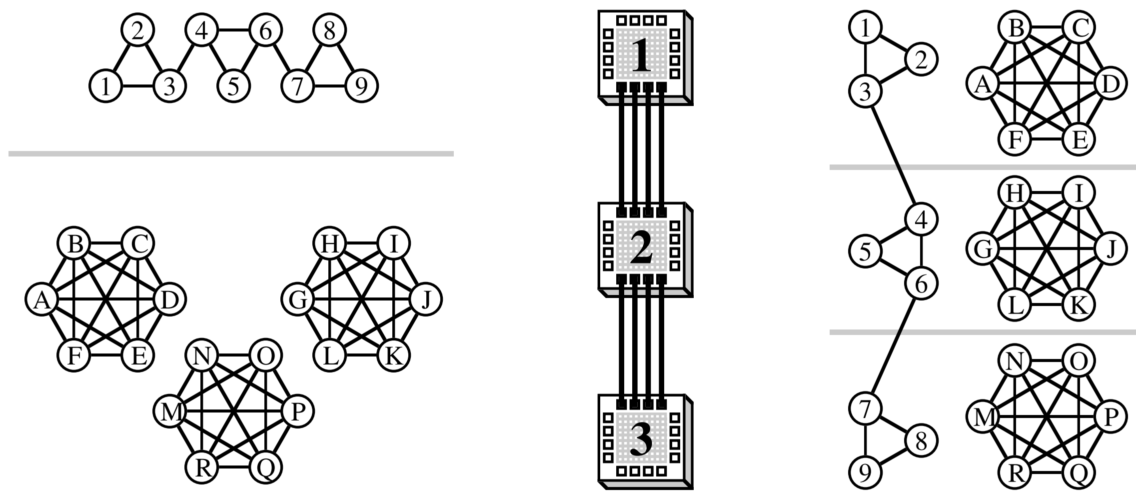


Figure 65. An example of the suboptimality of recursive bipartitioning. We are partitioning to the multi-FPGA system at middle, and the best first cut is shown at left (gray line). The next partitioning would have to cut nine signals. We can instead partition as shown at right, cutting a total of two signals.

The partitioning work presented in this thesis has focused on the other half of the partitioning problem. It details a partitioning algorithm that is fast, yet still develops good quality partitions, and is capable of working on an arbitrary subset of a multi-FPGA system. Such an algorithm would be appropriate in non-resource constrained situations (such as when a user has decided to spend extra on hardware in order to use faster mapping algorithms), as well as for incremental updates during testing and debugging, where mapping speed can be a major issue. The algorithm is based on the concept of iterative bipartitioning. Specifically, one can take a mapping and partition it into two parts. If one then reapply bipartitioning to these parts, one gets four parts. This process can be continued until the system is broken up into as many pieces as there are FPGAs in the system.

There are two problems with iterative bipartitioning onto a multi-FPGA system: it is greedy, and it ignores the multi-FPGA topology. It is greedy because the first bipartitioning attempts to find the best possible bipartitioning. While it may find a good way to split a mapping if it is only going to be broken into two parts, it may be a poor choice as a starting point for further cuts. The first split may require only a small amount of communication between the two halves of the system, but later cuts may require much more communication. We may have been better served having a somewhat larger initial cut, which could let subsequent cuts require much less communication. An example of this is shown in Figure 65.

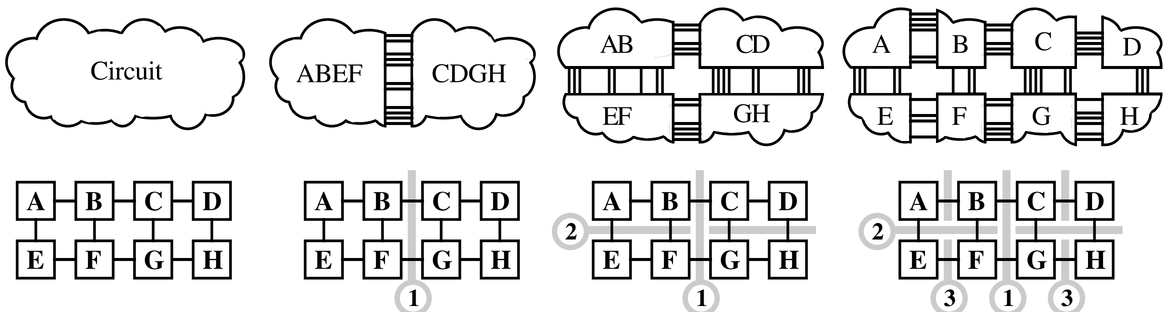


Figure 66. Example of iterative bipartitioning. The circuit (top left) is partitioned onto the multi-FPGA topology (bottom left) in a series of steps. Each partitioning corresponds to the most critical bottleneck remaining in the multi-FPGA system, and after each partitioning the placement of the logic is restricted to a subset of the topology (labeling on circuit partitions).

The greediness of iterative bipartitioning can be taken advantage of in order to map onto a specific multi-FPGA topology. Specifically, the multi-FPGA topology itself will have some bottleneck in its topology, some place where the expected communication is much greater than the routing resources in the multi-FPGA topology. For example, going back to the linear array of FPGAs discussed earlier (Figure 64), it is clear that if the number of wires connecting adjacent FPGAs is the same throughout the system, then the wires between the middle FPGAs, numbers **2** & **3**, will be the most heavily used, since with an equal number of FPGAs on either side of this pair of FPGAs, the communication demand should be the highest. Not all multi-FPGA systems will be linear arrays of FPGAs, but within most of them will be some set of links that are the critical bottleneck in the system. If we iteratively partition the circuit being mapped onto a topology such that the first cut in the circuit falls across this critical bottleneck, with the logic in one partition placed on one side of the bottleneck, and the other partition on the other side of the bottleneck, then this greedy early cut is performed exactly where the best cut is necessary. We can continue this process, performing partitionings corresponding to the remaining bottlenecks in the system. In this way we take advantage of the greedy nature of iterative bipartitioning, performing the early, greedy cuts where necessary to relieve the most significant bottlenecks in the topology, while later cuts (which are hurt by the greediness of earlier cuts) are performed where lower quality cuts can be tolerated, since they correspond to locations that are not critical bottlenecks in the system. An example of this process is shown in Figure 66. While this does not totally avoid the suboptimality of iterative bipartitioning, it does help limit it.

In order to implement such a partitioner, it is necessary both to develop an efficient bipartitioning algorithm, as well as an algorithm to find the critical bottlenecks in the multi-FPGA system. In Chapter 10 we discuss a bipartitioning algorithm that is fast and develops very high quality partitionings. Then in Chapter 11 we discuss an algorithm that can look at a topology and determine where the critical bottlenecks are in the multi-FPGA system. Thus, this algorithm can be used to determine the order in which bipartitionings should be iteratively performed to best map onto the multi-FPGA topology. Note that these bipartitionings might not be equipartitionings (that is, they might not attempt to split the circuit into two partitions each holding half of the circuit), but may in fact be unbalanced. Specifically, if the critical bottleneck in a multi-FPGA system splits the topology into one part containing 80% of the logic capacity, and the other part containing 20% of the logic capacity, then the bipartitioning of the circuit will attempt to form a split with one side having 80% of the logic, and the other having 20% of the logic. Also, after one or more bipartitionings there may be information about logic placement that needs to be transmitted between partitions. For example, in Figure 66 during the last partitioning, if logic *X* in subcircuit **AB** is connected to logic in subcircuit **CD**, then when partitioning subcircuit **AB** we need to know that *X* should be in partition **B**. Otherwise, there will be an additional signal that needs to be communicated from FPGA **A** to **B** because of the required connection to logic in subcircuit **CD**. There are standard techniques, such as those described in [Dunlop85], for handling such situations. What happens is that signals that cross earlier cuts (such as that between **AB** and **CD**) are represented by terminals, where this terminal is a logic node permanently assigned to the partition next to the cut. So in the case described before, there would be a logic node permanently assigned to partition **B**, which is connected to the signal going to *X*. *X* is still free to be assigned to partition A, but if it does so the algorithm will realize that the net from *X* to the terminal is cut, and thus will properly take into account this extra cost when optimizing. Note that similar techniques can be used to handle fixed logic and fixed external connections, situations where certain signals must reside in a specific FPGA or else they will contribute to the inter-FPGA communication costs in the multi-FPGA system.

Global Routing

Global routing is the process of determining through which FPGAs to route inter-FPGA signals. Note that the related problem of determining which specific pins to use to carry a signal is often handled separately;

it will be discussed later in this chapter. In cases where a signal needs to connect between only two directly connected FPGAs, this step is usually simple. No intermediate FPGA is needed to carry this signal, and the routing can be direct. However, in some cases the wires that directly connect these FPGAs may be used up by other signals. The signal will need to take alternate routes, routes that may lead through other FPGAs. For long-distance routing, there may be many choices of FPGAs to route through. A global router attempts to choose routes in order to make the most direct connections, thus minimizing delay and resource usage, while making sure that there are sufficient routing resources to handle all signal.

Routing is a problem in numerous domains, including standard cells, gate arrays, circuit board layout, and single FPGAs. Global routing for multi-FPGA systems is similar to routing for single FPGAs. Specifically, in an FPGA the connections are fixed and finite, so that the router cannot add resources to links that are saturated, and the connections may not obey strict geometrical distance metrics. The same is true for a multi-FPGA system. In the cases of standard cells and gate arrays, the size of routing channels can (usually) be increased, and in circuit board routing extra layers can be added. Thus, in order to handle multi-FPGA routing, we can use many of the techniques from single-FPGA routing.

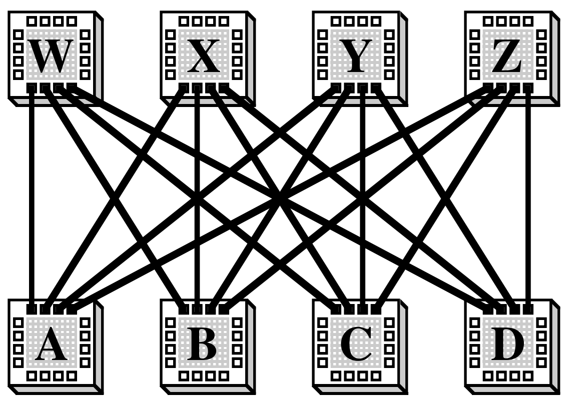


Figure 67. Crossbar routing topology. The chips at top are used purely for routing, while the FPGAs at bottom handle all the logic in the system.

There have been some routing algorithms developed specifically for multi-FPGA systems. These algorithms focus on crossbar topologies (Figure 67). In a crossbar topology, a signal that needs to be routed between FPGAs will always start and end at logic-bearing FPGAs, since routing-only chips have no logic in them. Thus, to route any signal in the system, regardless of how many FPGAs it needs to connect to, requires routing through exactly one other chip, and that chip can be any of the routing-only chips. This is because each routing-only chip connects to every logic-bearing FPGA, and the routing-only chips have exactly the same connectivity. Thus, routing for a crossbar topology consists simply of selecting which routing-only chip an inter-FPGA signal should route through, and a routing algorithm seeks only to route the most number of signals. Note that this does require some effort, since some assignments of signals to routing-only chips allow much less communication than others. For example, assume that each of the connections between chips in Figure 67 consist of 3 wires, and assume that we are attempting to route three-terminal wires, with connections evenly distributed between the logic-bearing FPGAs. If we route signals between FPGAs **ABC** through W, **ABD** through X, **ACD** through Y, and **BCD** through Z, we will be able to route three signals through each routing-only chip, for a total of 12 signals. At this point, no further routing can be performed, even of two-terminal wires. We can instead route one **ABC**, one **ABD**, one **ACD**, and one **BCD** wire through each of the routing-only chips, achieving a total of 16 routed signals.

There have been several algorithms proposed for the routing of signals in crossbar topologies. [Mak95] presents an algorithm that is optimal for 2-terminal routing, as well as proof that routing for multi-terminal nets is NP-Complete. The routing algorithm is based on the idea that if too many routes are assigned to a given routing-only chip, there must be some routing-only chip that is underutilized. Otherwise, there would be no possible routing for this mapping. Given this fact, the routes going through these two chips can be balanced, so that there is almost an identical number of nets going to each of these routing-only chip from each logic-bearing chip. Thus, to perform the routing for a crossbar topology, simply assign all of the nets to a single routing-only chip, and then iteratively balance the demand on routing-only chips until the routing is feasible. Since the problem of routing multi-terminal nets in a crossbar topology is NP-Complete, heuristic algorithms have been proposed [Butts91, Kadi94]. These greedily route signals through routing-only chips based on their current utilization. Because of this, they may not always find a routing solution in situations where such a solution exists, even in the case of purely 2-terminal routing. Thus, there may be some benefit in combining a heuristic approach to multi-terminal net routing with the optimal approach to 2-terminal net routing, though it is not clear how this could be done.

Another interesting approach to the global routing phase for multi-FPGA systems is the concept of Virtual Wires [Babb93, Selvidge95]. Virtual Wires attempts to deal with the I/O bottleneck in multi-FPGA systems by time-division multiplexing all external communications. As we mentioned earlier, most logic emulation systems achieve only 10%-20% logic utilization because of I/O limitations. Also, these systems tend to run at clock frequencies ten times or more slower than the maximum clock frequency of the FPGAs. For example, many FPGAs can easily achieve 30-50 MHz operation, while logic emulations using these chips achieve only 1-5 MHz at best.

The Virtual Wires approach is based on the observation that there are extra logic resources, and higher clock rates, available to use to ease the I/O congestion. Imagine that we generate a clock rate at twice the frequency of the emulation clock. This clock toggles a 2:1 multiplexer at each output, and a 1:2 demultiplexer at each input to the FPGAs in the system. This extra logic may take up 10% of the logic capacity of the FPGAs. However, since there is logic capacity and clock frequency available to perform these operations, the logic inside the FPGA sees an interface with twice the number of I/O pins, since two signals can go out each output, and two signals can come in each input, in a single emulation clock cycle. Although 10% of the logic capacity is consumed in doing this multiplexing and demultiplexing, we have room to spare in the FPGA. Thus, we should now be able to fit twice as much inside each FPGA, achieving perhaps 20%-40% utilization of the logic resources. The amount of multiplexing can be increased until the logic capacity and the multiplexing overhead use up the entire chip area. In this way, significantly greater logic capacities can be achieved.

As one might expect, there are many details that must be taken care of to fit a circuit into the model described above. Primarily, the order in which signals are sent from chip to chip must be carefully controlled, since when a signal is sent from one chip to another its value must be completely determined, and may not be changed later in the clock cycle. Methods for performing this analysis, architectures for best performing this multiplexing, and methods for transforming certain circuit features into the well-formed synchronous logic necessary for this approach have all been dealt with in the Virtual Wires system [Babb93, Dahl94, Tessier94, Selvidge95]. The important point with this system is that by applying these techniques, they have been able to increase the external connectivity of the FPGAs by a factor of almost eight [Tessier94].

Applying Virtual Wires to a multi-FPGA system has a significant impact on the mapping software. Primarily, Virtual Wires allows the mapping to a specific FPGA to trade off useful logic capacity with I/O resources. That is, if we apply greater multiplexing to the I/O pins, there is greater I/O capacity, but less logic capacity (this capacity is consumed by the logic necessary to perform the multiplexing). Similarly, less multiplexing yields greater usable logic capacity, but lower I/O capacity. This means that when partitioning onto a system using Virtual Wires, I/O and logic capacities of the partitions cannot be treated separately. However, these types of constraints have already been dealt with in the literature with the ratio-cut partitioning metric [Wei89]. This metric removes strict size bounds on partitions, and instead has a unified cost metric that combines size balancing and I/O limiting features. More details on the ratio-cut metric can be found in Chapter 11. While the ratio-cut metric may not be exactly the right metric to apply in this situation, it shows that these concerns can be added to standard partitioning metrics, including the Fiduccia-Mattheyses variant of the Kernighan-Lin algorithm described in this thesis (Chapter 10).

Another issue of concern with Virtual Wires is global routing. While in most systems it is simply necessary to minimize congestion on each link, making sure that no more signals are passed between FPGAs than there are wires to carry them, global routing under Virtual Wires introduces timeslices into the problem. Specifically, since the wires will be time-division multiplexed, several signals can be sent on a single wire during an emulator clock cycle. Thus, there are multiple "timeslices" on a wire, where each timeslice is a single tick of the faster clock used to multiplex the wires. Unfortunately, not all signals will be ready to be sent during any specific timeslice, since a signal can only be sent when all its inputs have arrived, and its logic has been given enough time to compute its value. However, the arrival of the inputs is dependent upon how they are routed, and which timeslice they are assigned to. Thus, the global router must understand the Virtual Wires model, and properly optimize the communication in both the normal spatial domain (which wire to use), and the Virtual Wires timeslice domain. Techniques for handling this routing have been developed as part of the Virtual Wires project.

Pin Assignment

Pin assignment is the process of deciding which I/O pins to use for each inter-FPGA signal. Since pin assignment occurs after global routing, which FPGAs long-distance routes will pass through has already been determined. Thus, if we include simple buffering logic in these intermediate FPGAs (this buffering logic can simply be input and output pads connected by an internal signal), all inter-FPGA signals now move only between adjacent FPGAs. Thus, pin assignment does not do any long-distance routing, and has no concerns of congestion-avoidance or too few resources.

The problem of pin assignment has been studied in many other domains, including routing channels in ASICs [Cai91], general routing in cell-based designs [Yao88, Cong91], and custom printed circuit boards (PCBs) [Pförtner92]. These approaches assume that the terminals of the logic can be placed into a restricted set of locations, and they attempt to minimize the inter-cell or inter-chip routing distances. Unfortunately, these approaches are unsuitable to the multi-FPGA pin assignment problem for several reasons. The most obvious is that while the cell, channel, and PCB problems seek to minimize the routing area between the logic elements, in a multi-FPGA system the routing between pins in the system is fixed. Thus, the standard approaches seek to optimize something that in the multi-FPGA problem is fixed and unchangeable. The standard approaches also tend to ignore the details of the logic, assuming that pin assignment has no effect on the quality of the logic element implementations. In the multi-FPGA problem, the primary impact of a pin assignment is on the quality of the mapping to the individual FPGAs, which correspond to logic elements in the other approaches. Thus, standard algorithms would ignore the routeability of the individual FPGAs, the most important factor to optimize in multi-FPGA pin assignment.

Finally, only specific sets of pins in a multi-FPGA system are connected, and the distances of routes leaving neighboring pin locations may not obey any standard geometric relationship, both of which are departures from the model assumed by the standard pin assignment methods. Because of these differences, there is no obvious way to adapt existing pin assignment approaches for other technologies to the multi-FPGA system problem.

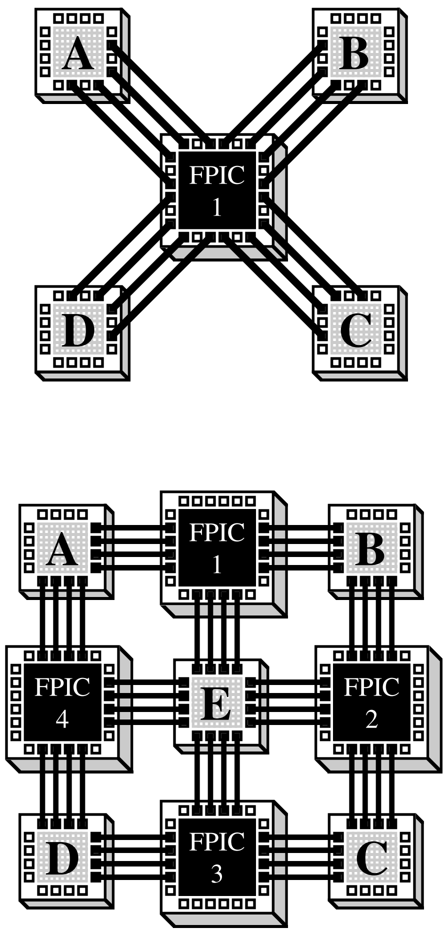


Figure 68. Multi-FPGA topologies with simple pin assignments.

There are existing methods of pin assignment for certain multi-FPGA topologies. The most obvious is for systems connected only through a single FPIC, crossbar chip, or routing-only FPGA (Figure 68 top left). In these situations, it is assumed that the routing chip can handle equally well any assignment of signals to its I/O pins, and thus the only consideration for pin assignment is the routeability of the logic-bearing chips. Since no logic-bearing chips are directly connected, because all routing is through the single routing chip, we can place these FPGAs independently. These placement runs can be almost completely unconstrained, allowing inter-FPGA signal pins the freedom to be assigned to any I/O pin going to the routing chip. Once the individual placements have completed, the pin assignments created by the placement tools induce a pin assignment onto the routing chip. That is, pins on the routing chip must be assigned such that they lie on the opposite end of the board trace connected to the appropriate pin on the FPGAs. In general this is a trivial problem, since there are one-to-one connections between FPGA pins and routing chip pins. Since the routing chip is capable of handling any pin connection pattern equally well, the routing chip need only be configured to handle this connection pattern, and the mapping to the multi-FPGA system is complete. A similar approach works for two-level topologies (Chapter 5) where there are multiple routing chips, but no two logic-bearing chips are directly connected, and no logic-bearing chip is connected to more than one routing chip (Figure 68 right). Again, the logic-bearing chips can be placed independently, and this induces a pin assignment onto the routing-only chips.

In a multi-FPGA topology where no two logic-bearing chips are directly connected, but with logic-bearing FPGAs connected to two or more routing-only chip (Figure 68 bottom left), the algorithm must be slightly modified. Depending on which I/O pin on an FPGA a signal is assigned to, it may go to a different routing-only chip. However, its ultimate destination may only be connected to one of these routing-only chips, and so the choice of which chip to route through is critical. Even if the destination of the route is connected to the same chips as the source, if the placements of these two FPGAs independently choose different routing-only chips to send the signal through, there will need to be an additional connection between these routing-only chips. Such connections may not exist, or may be too heavily congested to handle this routing. Thus, the placements cannot be free to choose the routing-only chip to which to connect a signal. However, this is not a problem. The global routing step determines which chips to route inter-FPGA signals through, and is capable of picking which routing-only chip is the best to use for any given route. Thus, all that is necessary is to make sure the pin assignment respects these connections. This is easy to accomplish, since it is clear which I/O pins on a logic-bearing FPGA are connected to which routing-only chip. This establishes constraints on the placement tool, telling it to assign individual logic pins to certain subsets of the chip's I/O pins. In this way, we can again place all the FPGAs independently, and let their placement induce a pin assignment on the routing-only chips.

As described in [Chan93b], there are some topologies that can avoid the need to have global routing determine the routing-only chip to route through, and instead everything can be determined by the placement tool. The simplest such topology is a crossbar topology with two logic-bearing and two routing-only chips (Figure 69 left). The pin connection pattern of the logic-bearing chips has alternating connections to the two different routing-only chips (Figure 69 right), so that each connection to one routing-only chip is surrounded by connections to the other routing-only chip.

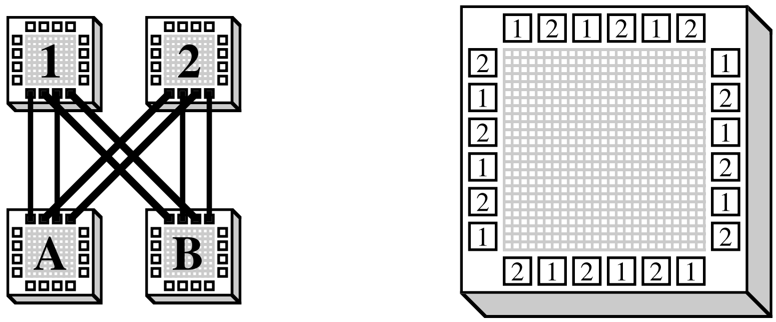


Figure 69. Topology for integrated global routing and pin assignment (left), with the FPGAs at bottom used for logic, and the FPGAs at top for routing only. The detailed pin connections of the logic-bearing FPGAs are given at right. The numbers in the pin positions indicate to which routing-only chip that pin is connected.

In a topology such as the one described, the logic-bearing FPGAs can be placed independently, without any restriction on the routing-only chip the signals must move through. The inter-FPGA routing pins can be placed into any I/O pin connected to a routing-only chip, but they are not constrained to any specific routing-only chip. While this gives the placement tool the flexibility to create a high quality placement, the source and destination pins of an inter-FPGA signal may end up connected to different routing-only chips. Obviously, this is not a valid placement, since there are no inter-routing chip wires to connect these terminals. However, since the problem terminals connect to different routing-only chips, and since the neighboring I/O pin positions go to different routing-only chips, moving one of the logic pins to one of the neighboring I/O pins fixes the routing problem for this signal. While this move may require moving the logic pin that already occupies the neighboring I/O pin, an algorithm for efficiently handling this problem is presented in [Chan93b] (hereafter referred to as the *Clos routing algorithm*). It will require moving pins at most to a neighboring I/O pin location, and can fix all routing problems in the topology presented above. Since this is only a slight perturbation, the placement quality should largely be unaffected. The alternate solution, where global routing determines which routing-only chip a signal should be routed through, can have a much more significant impact. Assume that in the optimum placement of the logic in the chip, where inter-FPGA signals are ignored, a set of 10 logic pins end up next to each other in the FPGA's I/O pins. Under the method just presented, these pins might be slightly shuffled, but will still occupy pins within one I/O pin position of optimum. Under the global router solution, the router might have assigned these 10 pins to be routed through the same routing-only chip. This would require these I/O connections to be scattered across every other pin position (since only half the pins in the region go to any given routing-only chip), forcing some of the connections a significant distance from the optimum.

While the Clos routing algorithm requires somewhat restricted topologies, it can be generalized from the topology in Figure 69. Specifically, there can be more than two logic-bearing chips, and more than two routing-only chips. The main requirements are that the logic-bearing chips are connected only to routing-only chips, the routing-only chips only to logic-bearing chips, and there are the same number of connections between each pair of logic-bearing and routing-only chips. The pin connections are similar to those shown in Figure 69 right, with connections interspersed around the logic-bearing FPGA's periphery. Specifically, if there are N routing-only chips in the system, then there are N-pin regions on the logic-bearing FPGA's periphery that contain connections to each of the N routing-only chips. In this way, when the pin assignment is adjusted after placement, a pin need only move within this group to find a connection to the appropriate routing-only chip, yielding a maximum movement of $N-1$ pin positions from optimal.

The pin assignment methods described above are adequate for systems where logic-bearing FPGAs are never directly connected. However, as we showed in Chapter 5, there are many multi-FPGA systems where logic-bearing FPGAs are directly connected. An algorithm for handling such systems is described in Chapter 12. It analyzes the logic assigned to each chip in the system, and determines the best way to assign the pins to create a good placement for all the FPGAs in the system. While it can handle routing-only chips reasonably, the techniques just discussed may deliver better mappings. The reason is that our pin assignment algorithm determines a specific pin assignment for all logic pins in the system based on somewhat abstracted information about the logic and FPGA architecture. The algorithms discussed in this section are able to use more detailed information on the FPGA architecture and logic to create a better pin assignment, but only for restricted multi-FPGA topologies. The best solution to the pin assignment problem may be a combined approach, which uses the algorithms from this section where appropriate, while relying on the algorithm from Chapter 12 for the rest of the system.

We can create a combined pin assignment algorithm as follows. Partitioning, global placement, and global routing are run normally. The pin assignment algorithm from Chapter 12 is then applied, creating a specific assignment for all pins in the system. This includes a specific assignment for pins that should best be assigned by one of the algorithms from this section, since this helps create a better pin assignment for all pins in the system. We then remove the assignments for those pins that can be handled by the algorithms presented in this section, namely those pins that need to connect to a routing-only chip. In cases where the pins are not interspersed as necessary for the Clos routing algorithm, the pins are restricted to the I/O pins connected to the routing-only chip the global router has picked for this pin to communicate with. Thus, it has the freedom to pick any pin going to the proper routing-only chip. For I/O pins arranged in the topology needed for the Clos routing algorithm, the logic pins going to these routing-only chips are allowed to be assigned to any of the I/O pins going to any of the routing-only chips. Thus, we are ignoring both the assignment created by the pin assignment algorithm, as well as the routing creating by the global router, for signals that can best be handled by the Clos routing algorithm. The FPGAs are then placed independently, and the Clos routing algorithm is applied to the necessary logic pins. In this way, we get the benefits of all three algorithms, applied where appropriate. Note that the pin assignment tool does extra work assigning pins going to routing-only chips, and the global router does extra work routing signals that will be handled by the Clos routing algorithm. However, this allows the tools to factor in the requirements of these signals when routing and assigning the other signals in the system, without causing any restrictions to these pins.

Placement And Routing

In many ways, the placement and routing tools necessary to map to the FPGAs in a multi-FPGA system are identical to standard single-FPGA placement and routing tools. Just as with single-FPGA systems, the placement and routing tools for a multi-FPGA system seek to fit a circuit into the logic and routing resources of the FPGA, managing resource conflicts while trying to create the highest performance mapping. It must be capable of accepting a predefined pin assignment, which assigns logic pins to I/O locations on the chip. If it also allows the restriction of pins to a subset of the possible I/O pin locations, it is possible to use the algorithms of the previous section to handle signals going to routing-only chips.

The only major difference between placement and routing for individual FPGAs and placement and routing of FPGAs in a multi-FPGA system is the need for fast mapping turnaround. For a system using a single FPGA, the quality of the mapping is often much more important than the amount of time it takes to create the mapping. Thus, high quality but low performance algorithms such as simulated annealing are the standard methods for these tasks. However, multi-FPGA systems are often used interactively, with numerous changes to the logic for bug fixes and augmented functionality. Since these changes are occurring in an interactive environment, and there may be hundreds of FPGAs in the multi-FPGA system, the time to place and route the FPGAs can be a serious concern. Also, many systems will use only a small fraction of the available resources, often using only 10%-20% of the logic resources. Thus, quality can be somewhat ignored, and high speed, moderate quality algorithms may be more appropriate. Because the placement and routing of each of the FPGAs can be performed independently, since all dependencies between the FPGAs are handled by the global routing and pin assignment algorithms, parallel execution of single-FPGA mapping runs is critical. If the mapping runs can be performed in parallel, on different machines connected to a shared network, then the multi-FPGA system can be back up and running much more quickly. This raises the issue of distributed processing and load balancing on multiple workstations. However, since the parallelism is quite coarse-grain (i.e., the unit of work is the placement and routing of a complete FPGA, a complex and time-consuming activity), the support software should be quite simple.

One final issue is the need to allow for future modifications when an FPGA is placed and routed. Specifically, in working with a design on a multi-FPGA system, the user may need to add test probes to the system, probes that allow the user to determine what is happening on a given signal in the system. This corresponds to a new signal in the multi-FPGA system, a signal that must be routed from the test point to an external connection. In order to handle these requirements smoothly, the placement and routing tools should attempt to leave extra resources available in the system so that these test signals can be accommodated in the extra resources. Also, the placement and routing tools should be able to create a new mapping for the FPGA, which includes the new signal, without the need to create the mapping from scratch. If the earlier mapping was created with adding future signals as a goal, then the remapping should be able to reuse most of this previous mapping, saving significant mapping time. It is not clear how well these goals can be accomplished, and there may be many interesting research problems involved in developing such mapping tools for multi-FPGA systems.

Summary

As we have pointed out in this chapter, one of the keys to widespread use of multi-FPGA systems is a complete, automatic method for mapping circuits onto the system. Such a system should be fast, so that the system can be used interactively. It should adapt to the multi-FPGA topology, so that it can be used on many different systems, on flexible topologies, and for iterative updates.

The mapping software consists of six steps: technology-mapping, partitioning & global placement, global routing, pin assignment, FPGA placement, and FPGA routing. Three of the steps (technology-mapping, FPGA placement, and FPGA routing) are currently supported by standard tools. Global routing is similar to routing for single FPGAs, and similar techniques can be applied. That leaves two major areas of concern: partitioning & global placement, and pin assignment. In Chapter 10 we discuss methods for performing logic bipartitioning efficiently and effectively. Then, Chapter 11 presents a method for recursively applying bipartitioning onto an arbitrary multi-FPGA topology. These chapters suggest an algorithm to handle the partitioning & global placement of multi-FPGA system mapping. Finally, Chapter 12 describes a pin assignment tool for arbitrary multi-FPGA systems. By combining the work contained in the following chapters with existing tools and techniques, a complete automatic-mapping system for multi-FPGA systems can be developed. Such a system would be fast, efficient, and topology-adaptive.

Chapter 10. Bipartitioning

Introduction

As detailed in Chapter 9, automatic-mapping tools for multi-FPGA systems are important to their success. A major part of this process is a fast partitioning algorithm that effectively minimizes inter-partition communication. In this chapter we present such a tool. Specifically, we investigate how best to quickly split logic into two parts. Along with the techniques discussed in Chapter 11, this forms a multi-way partitioning algorithm capable of optimized to an arbitrary, fixed multi-FPGA system topology.

Logic partitioning is also a critical issue for digital logic CAD in general. Effective algorithms for partitioning circuits enable us to apply divide-and-conquer techniques to simplify most of the steps in the mapping process. For example, standard cell designs can be broken up so that a placement tool need only consider a portion of the overall design at any one time, yielding higher quality results, in a shorter period of time. Also, large designs must be broken up into pieces small enough to fit into multiple devices.

For all of these tasks, the goal is to minimize the communication between partitions while ensuring that each partition is no larger than the capacity of the target device. While it is possible to solve the case of unbounded partition sizes exactly [Cheng88], the case of balanced partition sizes is NP-complete [Garey79]. As a result, numerous heuristic algorithms have been proposed.

In a 1988 survey of partitioning algorithms [Donath88] Donath stated "there is a disappointing lack of data comparing partitioning algorithms", and "unfortunately, comparisons of the available algorithms have not kept pace with their development, so we cannot always judge the cost-effectiveness of the different methods". This statement still holds true, with many approaches but few overall comparisons. This chapter addresses the bipartitioning problem by comparing many of the existing techniques, along with some new optimizations. It focuses primarily on those approaches that build on the Kernighan-Lin, Fiduccia-Mattheyses (KLFM) algorithm [Kernighan70, Fiduccia82].

One of the surprising results to emerge from this study is that by appropriately applying existing techniques an algorithm based upon KLFM can produce results better than the current state-of-the-art. In Table 1 we present the results of our algorithm (Optimized KLFM), along with results of three of the best current methods (Paraboli [Riess94], EIG1 [Hagen92], and Network Flow [Yang94]), on a set of standard benchmarks [MCNC93]. Note that the EIG1 algorithm is meant to be used for ratio-cut partitioning, not mincut partitioning as presented here.

Table 1. Quality comparison of partitioning methods. Values for KLFM and Optimized KLFM[1] are the best of ten trials. The EIG1 and Paraboli results are from [Riess94] (though EIG1 was proposed in [Hagen92]), and the Network Flow results are from [Yang94]. All tests require partition sizes to be between 45% and 55% of the total circuit sizes.

Mapping	Basic KLFM	**Optimized KLFM**	EIG1	Paraboli	Network Flow
s38584	243	**52**	76	55	47
s35932	136	**46**	105	62	49
s15850	105	**46**	215	91	67
s13207	105	**62**	241	91	74
s9234	65	**45**	227	74	70
Geom. Mean	118.8	**49.8**	156.5	73.1	60.3
Normalized	2.386	**1.000**	3.143	1.468	1.211

The results show that our algorithm produces significantly better solutions than the current state-of-the-art bipartitioning algorithms, with the nearest competitor producing results 21% worse than ours (thus, our algorithm is 17% better). Our algorithm is also fast, taking at most 7 minutes on the largest examples. Note that bipartitioning with replication has shown some promising results (all of the algorithms in the table do not use replication). [Kuznar94a, Kuznar94b] has reported results only 7-10% worse than ours. However, these results have no cap on the maximum partition size, while all other trials have a maximum partition size of 55% of the logic. In fact, some of Kuznar et al's runs have partitions of size 60% or larger. As will be demonstrated, allowing a partitioning algorithm to use a larger maximum partition size can greatly reduce the cutset size. Also, their work does not include primary inputs connected to both partitions as part of the cutset, while the cutsizes reported for the other approaches, including ours, do include such primary inputs in the cutset.

In the rest of this chapter we discuss the basic KLFM algorithm and compare numerous optimizations to the basic algorithm. This includes methods for clustering and unclustering circuits, initial partition creation, extensions to the standard KLFM inner loop, and the effects of increasing the maximum allowed partition size and the number of runs per trial.

---

1 Optimized KLFM includes recursive connectivity clustering, presweeping, per-run clustering on gate-level netlists, iterative unclustering, random initialization, and fixed 3rd-level gains. Each of these techniques is described later in this chapter.

Although the work described in this chapter is applicable to many situations, it has been biased by the fact that we are targeting multi-FPGA systems. One part of this is that the time it takes to perform the partitioning is important, and is thus a primary concern in this work. In tasks such as ASIC design, we can afford to allow the partitioner to run for hours or days, since it will take weeks to create the final implementation. In contrast, a multi-FPGA system is ready to use seconds after the mapping has been completed, and users demand the highest turnaround time possible. If the mapping software can complete a mapping in minutes instead of hours, the user can use the multi-FPGA system more interactively and thus more efficiently. Thus, there is significant interest in using an efficient partitioning method, such as KLFM partitioning, as opposed to more brute force approaches such as simulated annealing, which can take multiple hours to complete. Targeting our partitioning work towards multi-FPGA systems has several other impacts, which will be discussed later in this paper.

Methodology

In our work we have integrated numerous concepts from the bipartitioning literature, along with some novel techniques, to determine what features make sense to include in an overall system. We are primarily interested in Kernighan-Lin, Fiduccia-Mattheyses based algorithms, though we do include some of the spectral partitioning approaches as well. Note that there is one major omission from this study: the use of logic replication (i.e., the duplication of nodes to reduce the cutset). This is primarily because of uncertainty in how to limit the amount of replication allowed in the multi-FPGA partitioning problem. We leave this aspect to future work.

Table 2. Sizes of example circuits.

Mapping	s38584	s35932	s15850	s13207	s9234	s5378
Nodes (gates, latches, I/Os)	22451	19880	11071	9445	6098	3225

The best way to perform this comparison would be to try every combination of techniques on a fixed set of circuits, and determine the overall best algorithm. Unfortunately, we consider such a large number of techniques that the possible combinations reach into the thousands, even ignoring the ranges of numerical parameter settings relevant to some of these algorithms. Instead, we use our experience with these algorithms to try and choose the best possible set of techniques, and then try inserting into this mix each technique that was not chosen. In some cases, where it seemed likely that there would be some benefit of examining multiple techniques together and exploiting synergistic effects, we also tested those sets of techniques. In the comparisons that follow we always use all the features of the best mix of techniques found except where specifically stated otherwise.

The 6 largest circuits from the MCNC partitioning benchmark suite [MCNC93] are used as test cases for this work (Table 2). One of the largest, s38417, was not used because it was found to be corrupted at the storage site. While these circuits have the advantage of allowing us to compare with other existing algorithms, the examples are a bit small for today's partitioning tasks (the largest is less than 25,000 gates) and it is unclear how representative they are for bipartitioning. We hope that in the future a standard benchmark suite of real end-user circuits, with sizes ranging up to the hundreds of thousands of gates, will be available to the community.

```
Create initial partitioning;
While cutsize is reduced {
   While valid moves exist {
      Use bucket data structures to find unlocked node in each
         partition that most improves/least degrades cutsize when
         moved to other partition;
      Move whichever of the two nodes most improves/least degrades
         cutsize while not exceeding partition size bounds;
      Lock moved node;
      Update nets connected to moved nodes, and nodes connected to
         these nets;
   } endwhile;
   Backtrack to the point with minimum cutsize in move series just
      completed;
   Unlock all nodes;
} endwhile;
```

Figure 70. The Fiduccia-Mattheyses variant of the Kernighan-Lin algorithm.

Basic Kernighan-Lin, Fiduccia-Mattheyses Bipartitioning

One of the best known, and most widely extended, bipartitioning algorithms is that of Kernighan and Lin [Kernighan70], especially the variant developed by Fiduccia and Mattheyses [Fiduccia82]. Pseudo-code for the algorithm is given in Figure 70. It is an iterative-improvement algorithm, in that it begins with an initial partition and iteratively modifies it to improve the cutsize. The *cutsize* is the number of nets connected to nodes in both partitions, and is the value to be minimized. The algorithm moves a node at a time, moving the node that causes the greatest improvement, or the least degradation, in the cutsize. If we allowed the algorithm to move any arbitrary node, it could decide to move the node just moved in the previous iteration, returning to the previous state. Thus, the algorithm would be caught in an infinite loop, making no progress. To deal with this, we lock down a node after it is moved, and never move a locked node. The algorithm continues moving nodes until no unlocked node can be moved without violating the size constraints. It is only after the algorithm has exhausted all possible nodes that it checks whether it has improved the cutset. It looks back across all the intermediate states since the last check, finding the minimum cutsize. This allows it to climb out of local minima, since it is allowed to try out bad intermediate moves, hopefully finding a better later state. After it moves back to the best intermediate state, it unlocks all nodes and continues. Once the algorithm fails to find a better intermediate state between checks it finishes with the last chosen state.

One important feature of the algorithm is the bucket data structure used to find the best node to move. The data structure has an array of lists, where each list contains nodes in the same partition that cause the same change to the cutset when moved. Thus, all nodes in partition 1 that increase the cutset by 5 when moved would be in the same list. When a node is moved, all nets connected to it are updated. There are four situations to look for: 1) A net that was not in the cutset that now is. 2) A net that was in the cutset that now is not. 3) A net that was firmly in the cutset that is now removable from the cutset. 4) A net that was removable from the cutset that is now firmly in the cutset. A net is “firmly in the cutset” when it is connected to two nodes, or a locked node, in each partition. All other nets in the cutset are “removable from the cutset”, since they are connected to only one node in one of the partitions, and that node is unlocked. Thus, the net can be removed from the cutset by moving that node. Each of these four situations means that moving a node connected to that net may have a different effect on the cutsize now than it would have had if it was moved in the previous step. All nodes connected to one of these four types of nets are examined and moved to a new list in the bucket data structure if necessary.

The basic KLFM algorithm can be extended in many ways. We can choose to partition before or after technology-mapping. We can cluster circuit nodes together before partitioning, both to speed up the algorithm’s run-time, and to give some better local optimization properties to the KLFM’s primarily global viewpoint. We also have a choice of initial partition creation methods, from completely random to more intelligent methods. The main search loop can be augmented with more complex cost metrics, possibly adding more lookahead to the choice of nodes to move. We can uncluster the circuit and reapply partitioning, using the previous cut as the initial partitioning of the subsequent runs. Finally, we can consider how these features are improved or degraded by larger or smaller maximum partition sizes, and by multiple runs. In this chapter, we will consider each of these issues in turn, examining not only how the different approaches to each step compare with one another, but also how they combine together to form a complete partitioning solution.

Clustering and Technology-Mapping

One of the most common optimizations to the KLFM algorithm is clustering, the grouping together of nodes in the circuit being partitioned. Nodes grouped together are removed from the circuit, and the clusters take their place. Nets that were connected to a grouped node are instead connected to the cluster containing that node. Clustering algorithms are applied to the partitioning problem both to boost performance, and also to improve quality. The performance gain is due to the fact that since many nodes are replaced by a single cluster, the circuit to be partitioned now has fewer nodes, and thus the problem is simpler. Note that the clustering time can be significant, so we usually cluster the circuit only once, and if several independent runs of the KLFM algorithm are performed we use the same clustering for all runs. The ways in which clustering improves quality are twofold. First, the KLFM algorithm is a global algorithm, optimizing for macroscopic properties of the circuit. It may overlook more local, microscopic concerns. An intelligent clustering algorithm will often focus on local information, grouping together a few nodes based on local properties. Thus, a smart clustering algorithm can perform good local optimization, complementing the global optimization properties of the KLFM algorithm. Second, it has been shown that the KLFM algorithm performs much better when the nodes in the circuit are connected to at least an average of 6 nets, while nodes in circuits are typically connected to between 2.8 and 3.5 nets [Goldberg83]. Clustering should in general increase the number of nets connected to each node, and thus improve the KLFM algorithm. Note that most algorithms (including the best KLFM version we found) will partition the clustered circuit, and then use this as an initial split for another run of partitioning, this time on the unclustered circuit. Several variations on this theme will be discussed in a later section.

The simplest clustering method is to randomly combine connected nodes. The idea here is not to add any local optimization to the KLFM algorithm, but instead to simply exploit KLFM's better results when the nodes in the circuit have greater connectivity. A maximum random matching of the circuit graph [Bui89] can be formed by randomly picking pairs of connected nodes to cluster, and then reclustering as necessary to form the maximum number of disjoint pairs. Unfortunately, this is complex and time-consuming, possibly requiring $O(n^3)$ time [Galil86]. We chose to test a simpler algorithm (referred to here as *random clustering*), that should generate similar results while being more efficient and easier to implement. Each node is examined in random order and clustered with one of its neighbors (note that a node connected to a neighbor by N nets is N times as likely to be clustered with that neighbor). A node that was previously the target of a clustering is not used as a source of a clustering, but an unclustered node can choose to join a grouping with an already clustered node. Note that with random clustering a separate clustering is always generated for each run of the KLFM algorithm.

Numerous more intelligent clustering algorithms exist. *K-L clustering* [Garbers90] (not to be confused with KL, the Kernighan-Lin algorithm) is a method that looks for multiple independent short paths between nodes, expecting that these nodes should be placed into the same partition. Otherwise, each of these paths will have a net in the cutset, degrading the partition quality. In its most general form, the algorithm requires that two nodes be connected by k independent paths (i.e., paths cannot share any nets), of lengths at most $l_1 ... l_k$ respectively, to be clustered together. Checking for K-L connectedness can be very time-consuming, especially for longer paths. The biggest problem is high fanout nets, which are quite common in digital circuits. Specifically, if we are looking for potential nodes to cluster, and the source node of the search is connected to a clock or reset line, most of the nodes in the system are potential candidates, and a huge number of paths need to be checked. However, since huge fanout nets are the most likely to be cut in any partitioning, we can accelerate the algorithm by ignoring all nets with fanout greater than some constant. Also, if $l_1 = 1$, then the potential cluster-mates are limited to the direct neighbors of a node (though transitive clustering is possible, with A and C clustered together with B because both A and C are K-L connected with node B, while A and C are not directly K-L connected). In our study of K-L clustering we ignored all nets with fanout greater than 10, and used $k = 2$, $l_1 = 1$, $l_2 = 3$. The values of maximum considered fanout and l_1 were chosen to give reasonable computation times. While [Garbers90] recommends $k = 3$, $l_1 = 1$, $l_2 = 3$, $l_3 = 3$, we have found that this yielded few clustering opportunities (this will be discussed later), and the parameters we chose gave the greatest clustering opportunities with reasonable run time. Using $l_2 = 4$ would increase the clustering opportunities, but would also greatly increase run time.

A much more efficient clustering algorithm, related to K-L clustering, has been proposed [Roy93] (referred to here as *bandwidth clustering*). In this method, each net e in the circuit provides a bandwidth of $1/(|e|-1)$ between all nodes connected to it, where $|e|$ is the number of nodes or clusters connected to that net. All pairs of nodes that have a total bandwidth between them of more than 1.0 are clustered. Thus, nodes must be directly connected by at least two 2-terminal nets to be clustered, or a larger number of higher fanout nets. This clustering is similar to K-L clustering with $k = 2$, $l_1 = 1$, $l_2 = 1$, though it requires greater connectivity if the connecting nets have more than two terminals. Transitive clustering is allowed, so if the bandwidth between A&C is zero, they may still be clustered together if A&B and B&C each have a bandwidth of greater than 1.0 between them. There is an additional phase (carried out after all passes of recursive clustering, discussed below) that attempts to balance cluster sizes.

A clustering algorithm similar to bandwidth clustering, but which does not put an absolute lower bound on the necessary amount of bandwidth between the nodes, and which also considers the fanout of the nodes involved, has also been tested. It is based upon work done by Schuler and Ulrich [Schuler72], with several modifications. We will refer to it as *connectivity clustering*. Like random clustering, each node is examined in a random order and clustered with one of its neighbors. If a node has already been clustered it will not be the source of a new clustering attempt, though more than two nodes can choose to cluster with the same node. Nodes are combined with the neighbor with which they have the greatest connectivity.

Connectivity is defined in Equation 2. $Bandwidth_{ij}$ is the total bandwidth between the nodes (as defined in bandwidth clustering), where each *n*-terminal net contributes 1/(*n*-1) bandwidth between each pair of nodes to which it is connected. In this method nodes are more likely to be clustered if they are connected by many nets (the $bandwidth_{ij}$ in the numerator), if the nodes are small (the $size_i$ & $size_j$ in the denominator), and if most of the nodes' bandwidth is only between those two nodes (the $fanout_i$ - $bandwidth_{ij}$ & $fanout_j$ - $bandwidth_{ij}$ terms in the denominator). While most of these goals seem intuitively correct for clustering, the reason for the size limits is to avoid large nodes (or subsequent large clusters in recursive clustering, defined below) attracting all neighbors into a single huge cluster. Allowing larger nodes to form huge clusters early in the clustering will adversely affect the circuit partitioning.

$$connectivity_{ij} = \frac{bandwidth_{ij}}{size_i * size_j * \left(fanout_i - bandwidth_{ij}\right) * \left(fanout_j - bandwidth_{ij}\right)}$$

Equation 2. Connectivity metric for connectivity clustering.

While all the clustering techniques described so far have been bottom-up, using local characteristics to determine which nodes should be clustered together, it is possible to perform top-down clustering as well. A method proposed by Yeh, Cheng, and Lin [Yeh92] (referred to here as *shortest-path clustering*) iteratively applies a partitioning method to the circuit until all pieces are small enough to be considered clusters. At each step it considers an individual group at a time, where a group contains all nodes that have always been on the same side of the cuts in all prior partitionings. The algorithm then iteratively chooses a random source and sink node, finds the shortest path between those nodes, and increases the flow on these edges by 0.1. The flow is a number used in computing net lengths, where the current net length is *exp*(10**flow*). Before each partitioning, all flows are set to zero. When the flow on a net reaches 1.0, the net is part of the cutset. Once there is no uncut path between the random pairs of nodes chosen in the current iteration, the algorithm is finished with the current partitioning. In this way, the algorithm proceeds by performing a large number of 2-terminal net routings on the circuit graph, with random source and sink for each route, until it exhausts the resources in the system. Note that the original algorithm limits the number of subpartitions of any one group. Since this is not an important issue for our purposes, it was not included in our implementation. Once the algorithm splits up a group into subpartitions, the sizes of the new groups are checked to determine if they should be further subdivided. For our purposes, the maximum allowable cluster size is equal to (total circuit size)/100, which is half the maximum partition size variation. There are several alterations that can be made to this algorithm to boost performance, details of which can be found in Chapter 11.

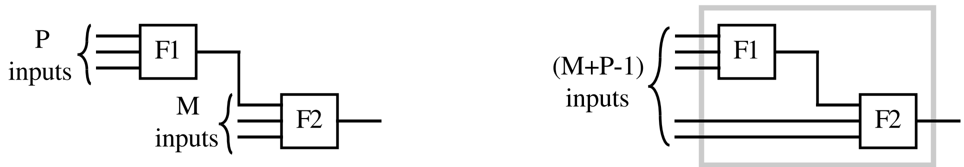


Figure 71. Example for the discussion of the size of logic functions. The *P*-input and *M*-input functions cascaded together (left) are fit into a *(M+P-1)* input LUT (right).

Before describing the last clustering method, it is necessary to discuss how to calculate the size of a logic node in the circuit being clustered. For our application (multi-FPGA systems), we are targeting FPGAs such as the Xilinx 3000 series [Xilinx94], where all logic is implemented by lookup-tables (LUTs). A LUT is a logic block that can implement any function of N variables, where N is typically 4 or 5. Since we will be partitioning circuits before technology-mapping (the reasons for this will be discussed later), we cannot simply count the number of LUTs used, since several of the gates in the circuit may be combined into a single LUT. An important aspect of a LUT-based implementation is that we can combine an M-input function with a P-input function that generates one of the M inputs into an (M+P-1)-input function (see Figure 71). The reason that it is an (M+P-1)-input function, and not an (M+P)-input function, is that the output of the P-input function no longer needs to be an input of the function since it is computed inside the LUT. A 1-input function (inverter or buffer) requires no extra inputs on a LUT. We can therefore say a logic node of P inputs uses up P-1 inputs of a LUT, and thus the size of a P-input function is (P-1), with a minimum size of 0. Any I/O nodes (i.e., external inputs and outputs) have a cost of 0. This is because if size keeps an I/O node out of a partition in which it has neighbors (i.e., nodes connected to the same net as the I/O node), a new I/O must be added to each partition to communicate the signal across the cut. Thus, moving an I/O node to a partition in which it has a neighbor never uses extra logic capacity. Although latches should also have a size of 0, since most FPGAs have more than sufficient latch resources, for simplicity we treat them identically to combinational logic nodes.

The last clustering technique we explored is not a complete clustering solution, but instead a preprocessor (called *presweeping*) that can be used before any other clustering approach. The idea is that there are some nodes that should always be in the same partition. Specifically, one of these nodes has a size of zero, and that node can always be moved to the other node's partition without increasing the cut size. The most obvious case is an I/O node from the original circuit which is connected to some other node N. This I/O node will have a size of zero, will be connected to one net, and moving the I/O node to node N's partition can only decrease the cut size (the cut size may not actually decrease, since another node connected to the net between N and the I/O node may still be in that other partition). Another situation is a node R, which is connected to exactly two nets, and one of these two nets is a 2-terminal net going to node *S*. Again, node *R* will have a size of zero, and can be moved to *S*'s partition without increasing the cutsize. The presweeping algorithm goes through the circuit looking for such situations, and clusters together the involved nodes (*R* with *S*, or *N* with the I/O node). Note that presweeping can be very beneficial to some clustering algorithms, such as K-L and Bandwidth, since such algorithms may be unable to cluster together the pairs found by presweeping. For example, an I/O node connected to only one net will never be clustered by the K-L clustering algorithm. Since the presweeping clustering should never hurt a partitioning (except due to random variation), presweeping will always be performed in this study unless otherwise stated.

Table 3. Quality comparison of clustering methods. Values are minimum cutsize for ten runs using the specified clustering algorithm, plus the best KLFM partitioning and unclustering techniques. Source mappings are not technology-mapped. The "No Presweep" column is connectivity clustering applied without first presweeping. All other columns include presweeping.

Mapping	Random	K-L	Bandwidth	Connectivity	Shortest-Path	No Presweep
s38584	177	88	112	57	50	59
s35932	73	86	277	47	45	70
s15850	70	90	124	60	59	65
s13207	109	94	87	73	72	79
s9234	63	79	56	52	51	65
s5378	84	78	88	68	67	66
Geom. Mean	89.7	85.6	108.7	58.8	56.5	67.1

Results for the various clustering algorithms are presented in Table 3 and Table 4. The shortest-path clustering algorithm generates the best results, with connectivity clustering performing only about 4% worse. In terms of performance, the shortest-path algorithm takes more than twice as long as the connectivity clustering algorithm. This is because clustering with the shortest-path algorithm takes more than 15 times as long as the connectivity approach. Shortest-path clustering would thus be even worse compared to connectivity clustering if the partitioner does not share clustering between runs. As we will show later, it is sometimes a good idea not to share clusterings. Because of the significant increase in run time because of shortest-path clustering, with only a small increase in quality, we use the connectivity algorithm for all of our other comparisons. We can also see that presweeping is a good idea, since connectivity clustering without presweeping does about 14% worse in terms of cutsize, while taking about 13% longer.

Table 4. Performance comparison of clustering methods. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm, plus the best KLFM partitioning and unclustering techniques. The "No Presweep" column is connectivity clustering applied without first presweeping. All other columns include presweeping.

Mapping	Random	K-L	Bandwidth	Connectivity	Shortest-Path	No Presweep
s38584	2157	2041	2631	1981	4715	2183
s35932	3014	1247	2123	2100	3252	2114
s15850	780	500	871	643	1354	713
s13207	648	428	629	549	1279	696
s9234	326	266	460	333	669	416
s5378	120	147	223	181	447	189
Geom. Mean	710.4	526.5	824.4	667.6	1412.5	751.5

One surprising result is that K-L clustering only does slightly better than random clustering, and Bandwidth clustering actually does considerably worse. The reason for this is that these clustering algorithms seem to require technology-mapping, and the comparisons in the tables are for non-technology-mapped circuits. Technology-mapping for Xilinx FPGAs is the process of grouping together logic nodes to best fill a CLB (an element capable of implementing any 5-input function, or two 4-input functions). Thus, it combines several basic gates into a single CLB. The reason that K-L and Bandwidth clustering perform poorly on non-technology-mapped (gate-level) circuits is that there are very few clustering opportunities for these algorithms. Imagine a sum-of-products implementation of a circuit. In general, any specific AND gate in the circuit will be connected to two or three input signals and some OR gates. Any AND gates connected to several of the same inputs will in general be replaced by a single AND gate. The OR gates are connected to other AND gates, but will almost never be connected to the same AND gate twice. The one possibility, an OR gate connected to an AND gate's output as well as producing one of that AND gate's inputs, is a combinational cycle, and usually not allowed. Thus, there will be almost no possibility of finding clusters with Bandwidth clustering, and few K-L clustering opportunities. While many gate-level circuits will not be simple sum-of-products circuits, we have found that there are still very few clustering opportunities for the K-L and Bandwidth algorithms.

Unfortunately, technology-mapping before partitioning is an extremely poor idea. In Table 5, columns 2 through 4 shows results for applying the various clustering algorithms to the Xilinx 3000 technology-mapped versions of the circuits (note that only four of the examples are used, because the other examples were small enough that the size of a single CLB was larger than the allowed partition size variation). Column 5 (“No Tech Map”) has the results for connectivity clustering on gate-level (non-technology-mapped) circuits. The results show that technology-mapping before partitioning almost doubles the cutsize. The K-L and Bandwidth clustering algorithms do perform almost as well as the connectivity clustering for these circuits, but we are much better off simply partitioning the gate-level circuits. This has an added benefit of speeding up technology-mapping as well, since we can technology-map each of the partitions in parallel. Note that we may increase the logic size by partitioning before technology-mapping, because there are fewer groupings for the technology-mapper to consider. However, in many technologies (especially multi-FPGA systems) the amount of logic that can fit on the chip is constrained much more by the number of I/O pins than on the logic size, and thus decreasing the cutsize by a factor of two is worth a small increase in logic size. This increase in logic size is likely to be small since the gates that technology-mapping will group into a CLB share signals, and are thus likely to be placed into the same partition.

Table 5. Quality comparison of clustering methods on technology-mapped circuits. Values are minimum cutsize for ten runs using the specified algorithm. The values in the column marked “Unclusterable” are the results of applying Connectivity clustering to technology-mapped files, but allowing the algorithm to uncluster the groupings formed by the technology-mapping. Note that only the four largest circuits are used, because technology-mapping for the others causes clusters to exceed allowed partition size variation.

Mapping	K-L	Bandwidth	Connectivity	No Tech Map	Unclusterable
s38584	169	159	120	57	60
s35932	155	157	143	47	53
s15850	86	90	87	60	60
s13207	118	119	116	73	72
Geom. Mean	127.7	127.9	114.7	58.5	60.9

It is fairly surprising that technology-mapping has such a negative effect on partitioning. There are two possible explanations: 1) technology-mapping produces circuits that are somehow hard for the KLFM algorithm to partition or 2) technology-mapping creates circuits with inherently much higher minimum cutsizes. There is evidence that the second reason is the underlying cause, that technology-mapped circuits simply cannot be partitioned as well as gate-level circuits, and that it is not simply due to a poor partitioning algorithm. To demonstrate this, we use the fact that the technology-mapped circuits for the Xilinx 3000 series contain information on what gates are clustered together to form a CLB. This lets us to consider the technology-mapping not as a permanent restructuring of the circuit, but instead simply as another clustering preprocessor. We allowed our algorithm to partition the circuit with the technology-mapped files, with connectivity clustering applied on top, then uncluster to basic gates and partition again. The results are shown in the final column of Table 5. Although the results for this technique are slightly worse than pure connectivity clustering, they are is still much better than the permanently technology-mapped versions. The small example circuit (Figure 72) demonstrates the problems technology-mapping can cause. There is a balanced partitioning of the circuit with a cutsize of 2, as shown in gray at left. However, after technology-mapping (CLBs are shown by gray loops), the only balanced partitioning puts the smaller CLBs in one partition, the larger CLB on the other. This split has a cutsize of 5.

The effects of technology mapping on cutsize have been examined previously by Weinmann [Weinmann94], who determined that technology-mapping before partitioning is actually a good idea, primarily for performance reasons. However, in his study he used only a basic implementation of Kernighan-Lin (apparently not even the Fiduccia-Mattheyses optimizations were applied), thus generating cutsizes significantly larger than what our algorithm produces, with much slower performance. Thus, the benefits of any form of clustering would help the algorithm, making the clustering provided by technology-mapping competitive. However, even these results report a 6% improvement in arithmetic mean cutsize for partitioning before technology-mapping, and the difference in geometric mean is actually 19% [2].

Figure 72. Example of the impact of technology-mapping on partitioning quality. The circuit s27 is shown (clock, reset lines, and I/O pins are omitted). At left is a balanced partition of the unmapped logic, which has a cutsize of 2. Gray loops at right indicate logic grouped together during technology-mapping. The only balanced partitioning has the largest group in one partition, the other two in the other partition, yielding a cutsize of 5.

---

[2] Throughout this chapter we use geometric instead of arithmetic means because we believe improvements to partitioning algorithms will result in some percentage decrease in each cutsize, not a decrease of some constant number of nets in all examples. That is, it is likely that an improved algorithm would reduce cutsizes for all circuits by 10%, and would not reduce cutsizes by 10 nets in both large and small examples. Thus, the geometric mean is more appropriate.

Unclustering

When we use clustering to improve partitioning, we will usually partition the circuit, uncluster it, and partition again. There are several ways to uncluster. Most obviously, we can either choose not to uncluster at all (*no unclustering*), or we can completely remove all clustering in one step (*complete unclustering*). However, there are better alternatives. The important observation is that while clustering we can build a hierarchy of clusters by recursively applying a clustering method, and then uncluster it in a way that exploits this hierarchy. In *recursive clustering*, after the circuit is initially clustered we reapply the clustering algorithm again upon the already clustered circuit. Clusters are never allowed to grow larger than half the allowed partition size variation. Recursive clustering continues until no more clusters can be formed. While we are clustering we remember what clusters are formed at each step, with clusters formed in the *ith* pass forming the *ith* level of a clustering hierarchy.

There are two ways to take advantage of the clustering hierarchy formed during recursive clustering. The most obvious method is that after partitioning completes (that is, when a complete pass of moving nodes fails to find any state better than the results of the previous pass) we remove the highest level of the clustering hierarchy, leaving all clusterings at the lower levels alone, and continue partitioning. That is, subclusters of clusters at the highest level, as well as those clusters that were not reclustered in the highest level, will remain clustered for the next pass. This process repeats until all levels of the clustering have been removed (note that clustering performed by presweeping is never removed, since there is nothing to be gained by doing so). In this way, the algorithm performs very coarse-grain optimization during early passes, very fine grain optimization during late passes, as well as medium-grain optimization during the middle passes. This algorithm, which we will refer to here as *iterative unclustering*, is based on work by Cong and Smith [Cong93].

An alternative to iterative unclustering is *edge unclustering*. This technique is based on the observation that at any given point in the partitioning there is likely to be some fine-grained, localized optimization, and some coarse-grained, global optimization that should be done. Specifically, those nodes that are very close to the current cut should be very carefully optimized, while nodes far from the cut need much less detailed examination. The edge unclustering algorithm is similar to iterative unclustering in that it keeps unclustering the highest levels of clustering remaining in between runs of the KLFM partitioning algorithm. However, instead of removing all clusters at a given level, it only removes clusters that are adjacent to the cut (i.e., those clusters connected to edges that are in the cutset). In this way, we will end up eventually unclustering all clusters next to the cut, while other clusters may remain together. When there are no more clusters left adjacent to the cut, we completely uncluster the circuit and partition one final time with KLFM.

Table 6. Quality comparison of unclustering methods. Values are minimum cutsize for ten runs using the specified algorithm. Source mappings are not technology-mapped, and are clustered by presweeping and connectivity clustering.

Mapping	Single-level Clustering		Recursive Clustering			
	No Uncluster	Complete Uncluster	No Uncluster	Complete Uncluster	Iterative Uncluster	Edge Uncluster
s38584	95	77	167	88	57	56
s35932	157	156	90	75	47	46
s15850	77	67	123	84	60	62
s13207	101	79	119	89	73	72
s9234	68	61	105	54	52	58
s5378	79	68	125	70	68	68
Geom. Mean	92.4	80.1	119.3	75.6	58.8	59.7

Table 7. Performance comparison of unclustering methods. Values are run times on a SPARC-IPX for ten runs using the specified algorithm. Source mappings are not technology-mapped, and are clustered by presweeping and connectivity clustering.

Mapping	Single Clustering		Recursive Clustering			
	No Uncluster	Complete Uncluster	No Uncluster	Complete Uncluster	Iterative Uncluster	Edge Uncluster
s38584	1220	1709	1104	1784	1981	2023
s35932	1224	1664	1359	1798	2100	2127
s15850	380	491	301	485	643	646
s13207	375	525	282	429	549	572
s9234	219	283	145	262	333	335
s5378	104	144	82	132	181	162
Geom. Mean	411.4	557	338.9	533.6	667.6	664.8

As the results in Table 6 and Table 7 show, using recursive clustering and a hierarchical unclustering method (iterative or edge unclustering) has a significant advantage. The methods that do not uncluster are significantly worse than all other approaches, by up to more than a factor of two. Using only a single clustering pass plus complete unclustering yields a cutsize 36% larger than the best unclustering (iterative), and even complete unclustering of a recursively clustered mapping yields a 29% larger cutsize. The difference between the two hierarchical unclustering methods is only 1.5%, with three mappings having smaller cutsizes with edge unclustering, and two mappings having smaller cutsizes with iterative unclustering. Thus, it appears that the difference between the two approaches is slight enough to be well within the margins of error of this survey, with no conclusive winner. In this survey, we use iterative unclustering except where explicitly stated otherwise.

Initial Partition Creation

KLFM is an iterative-improvement algorithm that gives no guidance on how to construct the initial partitioning that is to be improved. As one might expect, there are many ways to construct this initial partitioning, and the method chosen has an impact on the results.

The simplest method for generating an initial partition is to just randomly create one (*random initialization*) by randomly ordering the clusters in the circuit (initial partition creation takes place after clustering), and then finding the point in this ordering that best balances the total cluster sizes before and after this point. All nodes before this point are in one partition, and all nodes after this point are in the other partition.

Table 8. Quality comparison of initial partition creation methods. Values are minimum cutsize for ten runs using the specified algorithm.

Mapping	Random	Seeded	Breadth-first	Depth-first
s38584	57	57	57	56
s35932	47	47	47	47
s15850	60	60	60	60
s13207	73	75	80	74
s9234	52	68	52	52
s5378	68	79	80	78
Geom. Mean	58.8	63.4	61.4	60.2

An alternative to this is *seeded initialization*, which is based on work by Wei and Cheng [Wei89]. The idea is to allow the KLFM algorithm to do all the work of finding the initial partitioning. It randomly chooses one cluster to put into one partition, and all other clusters are placed into the other partition. The standard KLFM algorithm is then run with the following alterations: 1) partitions are allowed to be outside the required size bounds, though clusters can not be moved to a partition that is too large, and 2) at the end of the pass, it accepts any partition within size bounds instead of a partition outside of the size bounds. Thus, the KLFM algorithm should move clusters related to the initial "seed" cluster over to the small partition, thus making all nodes that end up in the initially 1-cluster partition much more related to one-another than a randomly generated partitioning.

Table 9. Performance comparison of initial partition creation methods. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm.

Mapping	Random	Seeded	Breadth-first	Depth-first
s38584	1981	1876	1902	2033
s35932	2100	2053	2090	2071
s15850	643	604	613	584
s13207	549	531	561	533
s9234	333	302	319	325
s5378	181	186	177	173
Geom. Mean	667.6	641.0	652.5	647.5

We can also generate an initial partitioning that has one tightly connected partition by *breadth-first initialization*. This algorithm again starts with a single node in one of the partitions, but then performs a breadth-first search from the initial node, inserting all nodes found into the seed node's partition. Once the seed partition grows to contain as close to half the overall circuit size as possible the rest of the nodes are placed into the other partition. To avoid searching huge fanout nets such as clocks and reset lines, which would create a very unrelated partition, nets connected to more that 10 clusters are not searched. *Depth-first initialization* can be defined similarly, but should produce much less related partitions.

Results for these initial partition construction techniques are shown in Table 8 and Table 9. The data shows that random is actually the best initialization technique, followed by depth-first search. The "more intelligent" approaches of seeded and breadth-first do 7% and 4% worse than random, respectively, and the differences occur only for the three smaller mappings. There are three reasons for this. First, recursive clustering and iterative unclustering seem to be able to handle the larger circuits well, regardless of how the circuit is initialized. With larger circuits there are more levels of hierarchy and the algorithms consistently get the same results. For smaller mappings there are fewer levels and much greater variance in results. Since there are thus many potential cuts that might be found when partitioning these smaller circuits, getting the greatest variance in the starting point will allow greater variety in results, and better values will be found (as will worse, but we only accept the best value of ten runs). Thus, the more random starting points perform better (random and depth-first initialization). Also, the more random the initial partitioning, the easier it is for the partitioner to move away from the initial partitioning. Thus, the partitioner is not trapped in a potentially poor partitioning, and can generate better results. Some of these effects can be seen in Figure 73 , which contains the distribution of results for ten runs of both random (black bars) and seeded (gray bars) initialization. As can be seen, there is little variation in the larger circuits. There is greater variation for smaller circuits, and for the random algorithm in general. Also, the random algorithm seems to be finding very different results from the seeded initialization, since few of the same cuts are found by the two algorithms for smaller circuits.

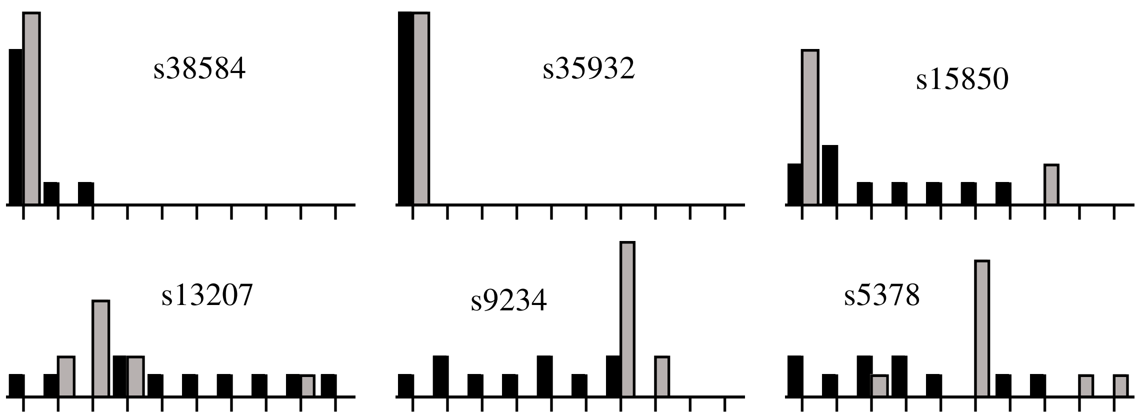


Figure 73. Distribution of results from partitioning with random (black bars) and seeded (gray bars) initialization. The *ith* bar from the left represents the *ith* best cutsize found by either algorithm, and the height indicates how many different runs of the algorithm (out of ten) achieved that result. There is little variation in the larger circuits. There is more variation in smaller circuits, and more variation in random initialization results in general.

While the previous discussion of initial partition generation has focused on simple algorithms, we can in fact use more complex, complete partitioning algorithms to find initial partitions. Specifically, there exists a large amount of work on “spectral” partitioning methods (as well as others) that constructs a partitioning from scratch. We will consider here the IG-Match [Cong92], EIG1 and EIG-IG [Hagen92] spectral partitioning algorithms. One important note is that these algorithms are designed to optimize for the ratio-cut objective [Wei89], which does not necessarily generate balanced partitions. However, we obtained the programs from the authors and altered them to generate only partitions with sizes between 49% and 51% of the complete circuit size, the same allowed partition size variation used throughout this chapter. These algorithms were applied to clustered circuits to generate initial partitionings. These initial partitionings were then used by our KLFM partitioning algorithm.

Table 10. Quality comparison of spectral initial partition creation methods. IG-Match [Cong92], EIG1 and EIG-IG [Hagen92] are spectral partitioning algorithms, used here to generate initial partitions. Entries labeled "n/a" are situations where the algorithm failed to find a partitioning within the required partition size bounds. Some of the spectral algorithms may move several clusters from one side of the cut to the other at once, missing the required size bounds (required only for our purposes, not for the ratio-cut metric for which they were designed). "All Spectral" is the best results from all three spectral algorithms.

Mapping	Random	EIG1	EIG1-IG	IG-Match	All Spectral
s38584	57	57	57	57	57
s35932	47	47	47	47	47
s15850	60	60	96	96	60
s13207	73	111	82	82	82
s9234	52	54	54	n/a	54
s5378	68	78	78	n/a	78
Geom. Mean	58.8	65.0	66.8	n/a	61.8

Table 11. Performance comparison of spectral initial partition creation methods. Values are total CPU seconds on a SPARC-IPX for the clustering, initialization, and partitioning algorithms combined. Values marked with "*" do not include the time for the failed IG-Match runs. "All Spectral" is the combined times for all three spectral partitionings.

Mapping	Random	EIG1	EIG1-IG	IG-Match	All Spectral
s38584	1981	336	445	1207	1988
s35932	2100	444	463	540	1447
s15850	643	89	102	206	397
s13207	549	79	95	152	326
s9234	333	42	56	n/a	98*
s5378	181	26	39	n/a	65*
Geom. Mean	667.6	102.3	127.8	n/a	365.2*

As the results show (Table 10 and Table 11), the algorithms (when taken as a group, under “All Spectral”) produce fairly good results, but are still 5% worse than random initialization. They do have the advantage of faster run times (including the time to perform spectral initialization on the clustered circuits), since they do not require, and cannot use, multiple partitioning runs. Also, as we will show in the next section the spectral approaches are competitive when our KLFM algorithm is restricted to the same runtimes as the spectral algorithm. However, the extra complexity of the spectral algorithm, along with their inability to take advantage of multiple runs, make us conclude that spectral initialization is not worthwhile.

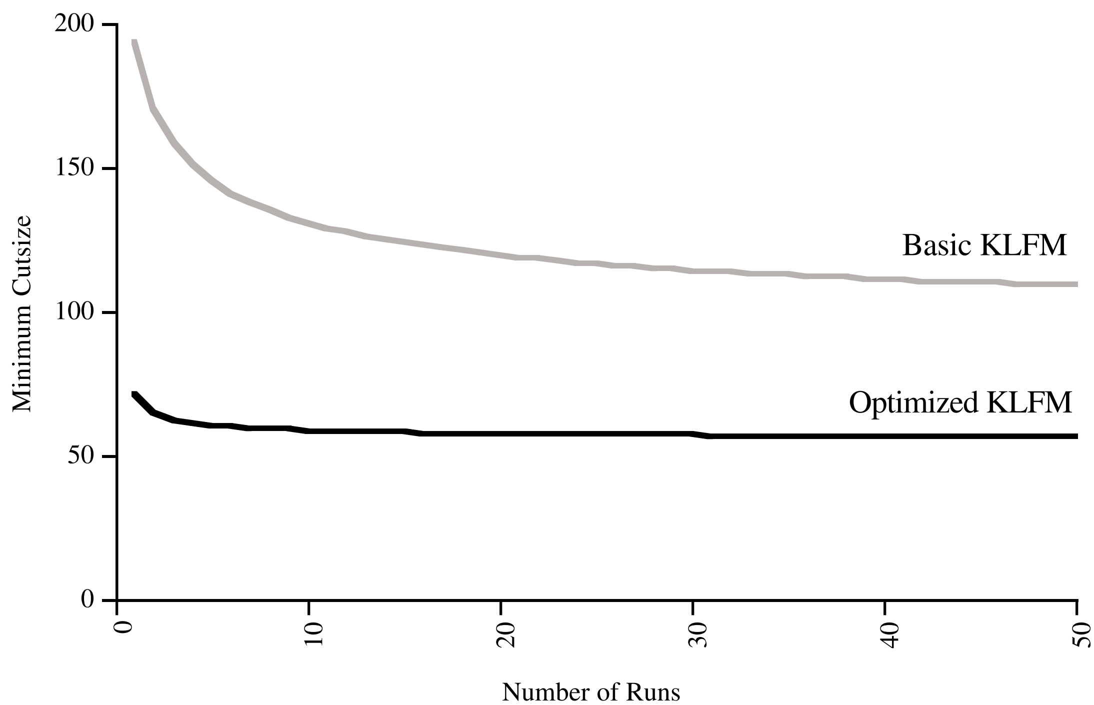


Figure 74. Graphs of cutsizes for different numbers of runs of both basic KLFM, and our optimized version of KLFM. Values shown are the geometric means of the results for all 6 test circuits.

Multiple Runs

While all of our tests have involved ten separate runs of the algorithm under consideration, and we retain the best result of these ten runs, we can consider using more or less runs per test. Basic KLFM is notoriously variable from run to run, and using multiple runs (up to even a hundred or more) is essential for achieving good results. To test how our algorithm responded to multiple runs, we ran one hundred runs of our best algorithm. We then used these datasets to determine the expected best cutsize found by an arbitrary number of runs of the algorithm. The results are shown in Figure 74, as well as similar results for the basic KLFM algorithm. As can be seen, not only does our optimized algorithm generate better results than the base KLFM algorithm, but it also has much less variability than the original algorithm, thus requiring fewer runs to be performed in general. Multiple runs are still valuable, since running the algorithm twice produces results 10% better on average than only a single run, and ten runs produces results 18% better than a single run. However, there are significantly diminished returns from further runs. Twenty runs produce results only 2% better than ten runs, and the best values found from all one hundred runs are only 5% better than those produced from ten runs. It is unclear exactly how many runs should be used in general, since for some situations a 2% improvement in cutsize is critical, while for others it is performance that is the primary concern. We have chosen to use ten runs for all of the tests presented in this chapter unless stated otherwise.

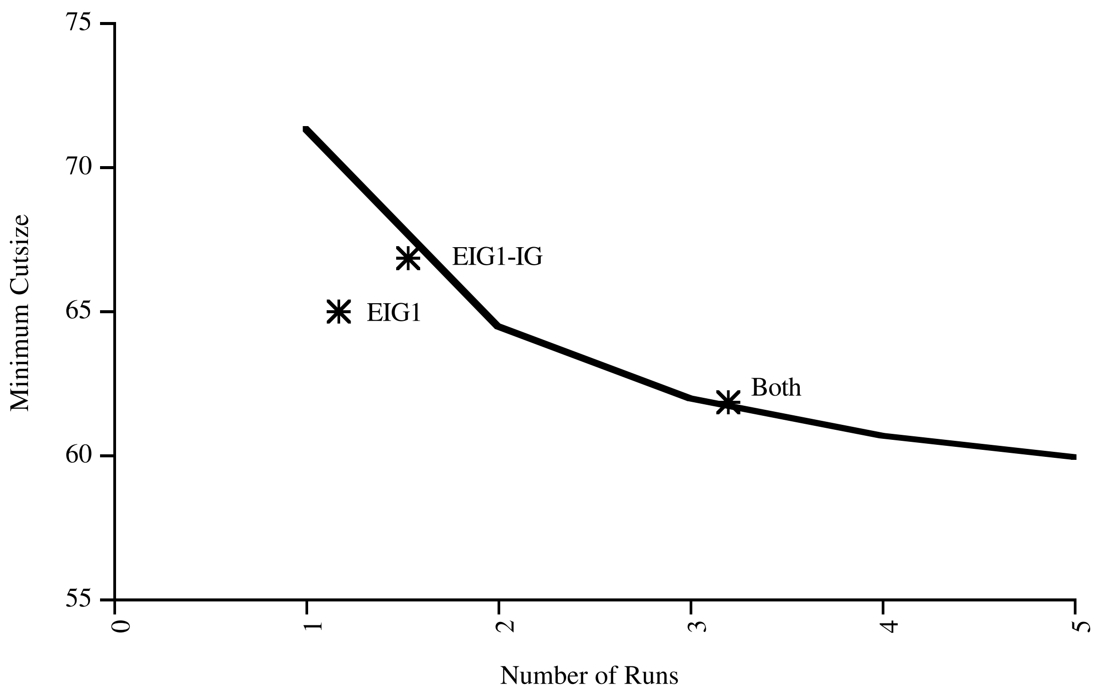


Figure 75. Graphs of cutsizes for different numbers of runs of our optimized version of KLFM versus the spectral initialization approaches. The spectral approaches are placed horizontally so that they line up to the equivalent amount of time spent on the optimized KLFM algorithm. Values shown are the geometric means of the results for all 6 test circuits.

Figure 75 shows details of the optimized KLFM runs from Figure 74, as well as the results from the spectral initialization approaches discussed earlier. The spectral results are placed horizontally so that they line up to the equivalent amount of time spent on the optimized KLFM algorithm. That is, in the time it takes to run both EIG1 and EIG1-IG initialized runs of our optimized KLFM algorithm approximately 3.2 runs of the optimized KLFM algorithm with random initializations could be performed. Note that the time to perform clustering is factored into the time to perform the first run of the optimized KLFM algorithm. As can be seen, the spectral approaches are somewhat more time efficient than our optimized algorithm, and thus might be useful in extremely time critical situations. However, multiple runs of our optimized algorithm can be run in parallel, performing better than the spectral approaches, and with slightly more time sequential runs of our optimized algorithm on a single processor produce better quality than the spectral approaches. Because of this, and also because the spectral approaches are much more complex than the optimized algorithm (since the spectral approaches perform spectral initialization, a complex process, and then run our entire optimized algorithm as well), we will use random initialization for our optimized algorithm.

Higher-Level Gains

The basic KLFM algorithm evaluates node moves purely on how much the move immediately affects the cutsize. However, there are often several possible moves that have the same effect on the cutsize, but these moves may have very different ramifications for later moves. Take for example the circuit in Figure 76 left. If we move either **B** or **E** to the other partition, the cutsize remains the same. However, by choosing to move **B**, we can reduce the cutsize by one by then moving **A** to the other partition. If we move **E**, it will take two further moves (**C** and **D**) to remove the newly cut three-terminal net from the cutset, and this would still keep the cutsize at 2 because of the edge from **C** to the rest of the logic.

To deal with this problem, and give the KLFM algorithm some lookahead ability, Krishnamurthy proposed *higher-level gains* [Krishnamurthy84]. If a net has n unlocked nodes in a partition, and no locked nodes in that partition, it contributes an *nth*-level gain of 1 to moving a node from that partition, and an *(n+1)th*-level gain of -1 to moving a node to that partition. The first-level gains are identical to the standard KLFM gains, with a net currently uncut giving a first-level gain of -1 to its nodes, and a net that can be uncut by moving a node **A** gives a first-level gain of 1 to node **A**. The idea behind this formulation is that an *nth*-level gain of 1 indicates that by moving N nodes, including the node under consideration, we can remove a net from the cutset. An *(n+1)th*-level gain of -1 means that by moving this node, we can no longer remove this net by moving the n nodes connected to the net in the other partition. Moves are compared based on the lowest-order gain in which they differ. So a node with gains (-1, 1, 0) (1st-level gain of -1, 2nd-level of 1, 3rd-level of 0) would be better to move than a node of (-1, 0, 2), but worse to move than a node of (0, 0, 0). To illustrate the gain computation better, we give the examples in Figure 76 right. Net **123** has one node in the left partition, giving a 1st-level gain of 1 for moving a node out of this partition, and a 2nd-level gain of -1 for moving a node to this partition. It has two nodes in the right partition, giving a 2nd-level gain of 1 for moving a node from this partition, and a 3rd-level gain of -1 for moving a node to this partition. Thus, node **1** has a gain vector of (1,0,-1), and nodes **2** and **3** have gains of (0,0,0), since the 2nd-level gains of 1 & -1 cancel each other. This makes sense, because after moving either node **2** or **3** you have almost the same situation for net **123** as the current state. Note that if node **3** were locked, node **2** would have a gain vector of (0,-1,0), and node **1** would have a gain vector of (1,0,0), since there is no longer any contribution to the gain vector of net **123** from the state of the right partition. For net **45** there is a 2nd-order gain of 1 for moving nodes out of the left partition, and a 1st-order gain of -1 for moving nodes into the right partition, giving nodes **4** and **5** a gain vector of (-1,1,0). If node **4** was locked, then node 5 would have a gain vector of (-1,0,0), since there is no longer any contribution to the gain vector of net **45** from the state of the left partition. Net **678** is similar to **45**, except that it has a 3rd-order, not a 2nd-order, gain of 1. So, we can rank the nodes (from best to move to worst) as **1**, **23**, **45**, **678**, where nodes grouped together have the same gains. If we do move **1** first, **1** would now be locked into the other partition, and nodes **2** and **3** would have a 1st-level gain of -1, and no other gains. Thus, they would become the worst nodes to move, and node **4** or **5** would be the next candidate.

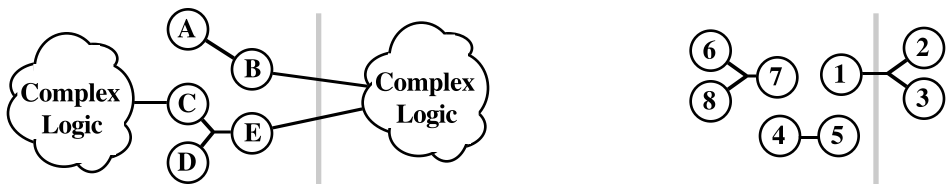


Figure 76. Examples for the higher-level gains discussion.

Note that the definition of *nth*-level gains given above is slightly different than Krishnamurthy's. Specifically, in Krishnamurthy's definition the rule that gives an *nth*-level gain to a net with *n* unlocked nodes in a partition is restricted to nets that are currently in the cutset. Thus, nets **678** and **45** would both have gains (-1, 0, 0). However, as we have seen, allowing nth-level gains for nets not in the cutset allows us to see that moving a node on **45** is better than moving a node on **678**, since it is easier to then remove **45** from the cutset than it is **678**. Also, this definition handles 1-terminal nets naturally, while Krishnamurthy requires no 1-terminal nets to be present in the circuit. A 1-terminal net with our definitions would have a 1st-level gain of 1 for having only one node in the starting partition, but a 1st-level gain of -1 because there are no nodes in the other partition, yielding an overall 1st-level gain of 0. Note that 1-terminal nets are common in clustered circuits, occurring when all nodes connected to a net are clustered together.

Table 12. Quality comparison of higher-level gains. Numbers in column headings are the highest higher-level gains considered. Note that a fixed gain-level of 1 is identical to KLFM without higher-level gains. Values are minimum cutsize for ten runs using the specified algorithm.

Mapping	Dynamic	Fixed				
		1	2	3	4	20
s38584	57	57	57	57	57	57
s35932	49	47	49	47	47	47
s15850	60	64	62	60	60	60
s13207	75	77	77	73	73	73
s9234	52	56	52	52	52	52
s5378	66	71	70	68	68	68
Geom. Mean	59.2	61.2	60.4	58.8	58.8	58.8

Table 13. Performance comparison of higher-level gains. Numbers in column headings are the highest higher-level gains considered. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm.

Mapping	Dynamic	Fixed				
		1	2	3	4	20
s38584	1904	1606	1652	1981	2078	3910
s35932	2321	1830	1862	2100	2297	2766
s15850	630	509	518	643	678	956
s13207	551	425	446	549	572	815
s9234	338	252	250	333	355	466
s5378	186	130	134	181	185	241
Geom. Mean	677.2	524.5	536.4	667.6	703.8	990.8

There is an additional problem with using higher-level gains on clustered circuits: huge runtimes. The KLFM partitioning algorithm maintains a bucket for all nodes with the same gains in each partition. Thus, if the highest fanout node has a fanout of N, in KLFM without higher-level gains there must be $2*N+1$ buckets per partition (the N-fanout node can have a total gain between $+N$ and $-N$). If we use M-level gains (i.e., consider higher-level gains between 1st-level and *Mth*-level inclusive), we would require $(2*N+1)^M$

different buckets. In unclustered circuits this is fine, since nodes will have a fanout of at most 5 or 6. Unfortunately, clustered circuits can have nodes with fanout on the order of hundreds. This causes not only a storage problem, but also a performance problem, since the KLFM algorithm will often have to perform a linear search of all buckets of gains between occupied buckets, and buckets will tend to be sparsely filled. We have found two different techniques for handling these problems. First, the runtimes are acceptable as long as the number of buckets is reasonable (perhaps a few thousand). So, given a specific bound N on the largest fanout node (which is fixed after every clustering and unclustering step), we can set M to the largest value that requires less than a thousand buckets be maintained. This value is recalculated after every unclustering step, allowing us to use a greater number of higher-level gains as the remaining cluster sizes get smaller. We call this technique *dynamic gain-levels*. An alternative to this is to exploit the sparse nature of the occupied gain buckets. That is, among nodes with the same 1st- and 2nd-level gains, there will be few different occupied gain buckets. What we can do is perform the dynamic gain-level computation to determine the number of array locations to use, but each of these array locations is actually a sorted list of occupied buckets. That is, once the dynamic computation yields a given M, all occupied gain buckets with the same first M gains will be placed in the list in the same array location. In this way, circuits with large clusters, and thus very sparse usage of the possible gain levels, have only 2 or 3 gain-levels determining the array location, while circuits with small or no clusters, and thus more dense usage of the smaller possible gain locations, have more of their gain orders determining the array locations. In this latter technique, called *fixed gain-levels*, the user can specify how many gain-levels the algorithm should consider, and the algorithm automatically adapts its data structures to the current cluster sizes.

As shown in Table 12 and Table 13, using more gain levels improves the results, but only up to a point. Once we consider gains up to the 3rd level, we get all the benefits of up to 20 gain levels. Thus, extra gain levels beyond the 3rd level only serve to slow down the algorithm, up to a factor of 50% or more. Dynamic gain-levels produces results between those of 2-level and 3-level fixed gains. This is to expected, since at high clustering levels the dynamic algorithm uses only 2 gain levels, though once the circuit is almost totally unclustered it expands to use several more gain-levels. In this survey we use fixed, 3-level gains.

Dual Partitioning

During partitioning, the goal is to minimize the number of nets in the cutset. Because of this, it seems odd that we move nodes from partition to partition instead of moving nets. As suggested by Yeh, Cheng, and Lin [Yeh91], we can combine both approaches in a single partitioning algorithm. The algorithm consists of Primal passes, which are the basic KLFM outer loop, and Dual passes, which are the KLFM outer loop, except that nets are moved instead of nodes. In this way, the Dual pass usually removes a net from the cutset at each step, though this may be more than balanced by the addition of other nets into the cutset. Just as in the KLFM algorithm, a single Primal or Dual pass moves each node or net once, and when no more objects can be moved the state with the lowest cutsize is restored. Primal and Dual passes are alternated, and the algorithm ends when two consecutive passes (one Primal, one Dual, in either order) produce no improvement. When performing unclustering, we start with a Primal pass after each unclustering.

While the concept of moving nets may seem straightforward, there are some details to consider. First, when we move a net, we actually move all nodes connected to that net to the destination partition. Nodes already in that partition remain unlocked, while moved nodes are locked. Because we are moving nets and not nodes, the bucket data structure holds nets sorted by their impact in the cutsize, not nodes. An odd situation occurs when a net is currently in the cutset. Since it has nodes in each partition, it is a candidate to be moved to either partition. Also, because we are moving nets and not nodes, it is unclear how to apply higher-level gains to this problem, so higher-level gains are only considered in the Primal passes.

One of the problems with the Dual partitioning passes is that they are excessively slow. When we move a net, it not only affects the potential gain/loss of moving neighboring nets (where two nets are neighbors if they both connect to a shared node), it can affect the neighbor's neighbors as well. The gain of moving a net is the sum of the gain of removing the net from the cutset (1 if the net is currently cut, 0 otherwise), plus gains or losses from adding or removing neighboring nets from the cutset (by moving a node connected to a neighboring net, we may add or remove that net from the cutset). Thus, when we move a net, we may add or remove a neighboring net to or from the cutset. That neighbor's neighbors may have already expected to add or remove the neighbor from the cutset, and their gains may need to be recalculated. In a recursively clustered circuit, or even in a circuit with very high fanout nets (such as clocks and reset lines), most of the nets in the system will be neighbors or neighbors of neighbors. Thus, each move in a Dual pass will need to recalculate the gains of most of the nets in the system, taking a significant amount of time.

The solution we adopted is to ignore high fanout nets in the Dual pass. In our study, we do not consider moving high fanout nets (those nets connected to more than 10 nodes) since it is unlikely that moving a high fanout net will have a positive effect on the cutsize. We also do not consider the impact of cutting these high fanout nets when we decide what nets to move. Thus, when a neighbor of this net is moved, we do not have to recalculate the gains of all neighbors of this high fanout net, since these nets do not have to worry about cutting or uncutting the high fanout net. Note that this makes the optimization inexact, and at the end of a Dual pass we may return to what we feel is the best intermediate state, but which is actually worse than other states, including the starting point for this pass. To handle this, we re-evaluate the cutsize at this state, and only accept it if it is in fact better than the original starting point. Otherwise, we backtrack to the starting point. In our experience the cutsize calculation is almost always correct.

Table 14. Quality comparison of Dual partitioning. Values are minimum cutsize for ten runs of the specified algorithm. The data does not include the largest circuit due to excessive runtimes.

Mapping	No Dual Passes	Dual Passes
s35932	47	47
s15850	60	62
s13207	73	75
s9234	52	51
s5378	68	67
Geom. Mean	59.2	59.5

Table 15. Performance comparison of Dual partitioning. Values are total CPU seconds on a SPARC-IPX for ten runs of the specified algorithm. The data does not include the largest circuit due to excessive runtimes.

Mapping	No Dual Passes	Dual Passes
s35932	2100	10695
s15850	643	19368
s13207	549	8415
s9234	333	8293
s5378	181	5445
Geom. Mean	667.6	9532.5

Data from testing the Dual partitioning passes within our best algorithm is shown in Table 14 and Table 15. As can be seen, there is little difference in the quality of the two solutions, and in fact using the Dual passes actually degrades the quality slightly. The Dual passes also slow overall algorithm runtimes by a factor of over 14 times, even with the performance enhancements discussed previously. Obviously, without any signs of improvement in partitioning results, there is no reason to suffer such a large performance degradation.

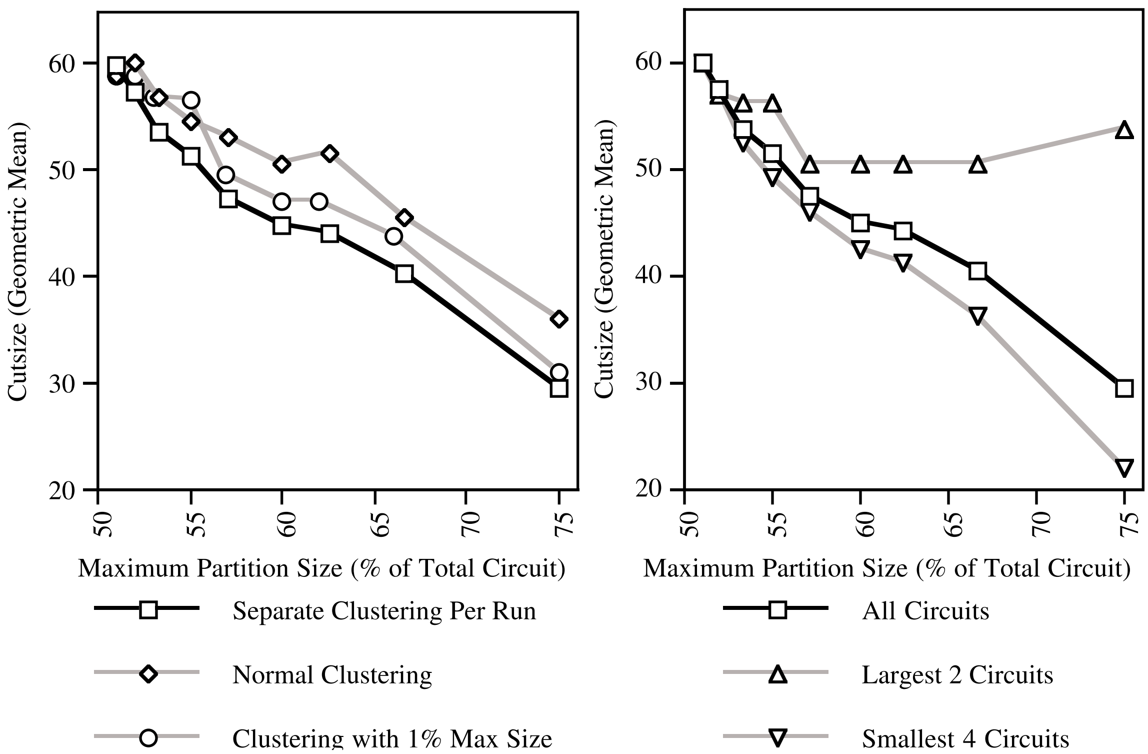


Figure 77. Graphs of partitioning results as the maximum allowed partition size is increased. At left are the results for separate clustering calculations for each run of the algorithm ("Separate Clustering Per Run"), one clustering for each partition size ("Normal Clustering"), and one clustering for each partition size, plus a maximum cluster size of 1% of the total circuit ("Clustering with 1% Max Size"). At right we have more detail on the "Separate Clustering Per Run", with the results for all circuits, plus one line for just the largest 2 circuits, and one for the smallest 4. Both "Largest 2" and "Smallest 4" lines are scaled to have the same value as "All Circuits" for the leftmost data point.

Partition Maximum Size Variation

Variation in the allowed partition size can have a significant impact on partitioning quality. In partitioning, we put limits on the sizes of the partitions so that the partitioner cannot place most of the nodes into a single partition. Allowing all nodes into a single partition obviously defeats the purpose of partitioning in most cases, since we are usually trying to divide the problem into manageable pieces. The variance in partition size defines the range of sizes allowed, such as between 45% and 55% of the entire circuit. There are two incentives to allow as much variance in the partition sizes as possible. First, the larger the allowable variation, the greater the number of possible partitionings. With more possible partitionings, it is likely that there will be better partitionings available, and hopefully the partitioner will generate smaller cutsizes. The second issue is that there needs to be enough variance in partition sizes to let each node move between partitions. If the minimum partition size plus the size of a large node is greater than the maximum partition size then this node can never be moved. This will artificially constrain the placement of this node to the node's initial partition assignment, which is often a poor choice. While we might expect that the size of the nodes in the graph being partitioned will be small, and thus not require a large variation in partition sizes, we will usually cluster together nodes before partitioning, greatly increasing the maximum node size. A smaller partition variation will limit the maximum cluster size, limiting the effectiveness of clustering optimizations. In general, we will require that the maximum cluster size be at most half the size of the allowable variation in partition sizes. In this way, if we have maximum-sized clusters as move candidates from both partitions, at least one of them will be able to move.

Conflicting with the desire to allow as much variation in partition sizes as possible is the fact that the larger the variation, the greater the wastage of logic resources in a multi-chip implementation, particularly a multi-FPGA system. Specifically, when we partition to a system of 32 FPGAs, we iteratively apply our bipartitioning algorithm. We split the overall circuit in half, then split each of these partitions in half, and so on until we generate a total of 32 subpartitions. Now, consider allowing partition sizes to vary between 40% and 60% of the logic being split. On average, it is likely that better partitions exist at points where the partition sizes are most unbalanced, since with the least amount of logic in one partition there is the least chance that a net is connected to one of those nodes, and thus the cutsize is likely to be smaller. This means that many of the cuts performed may yield one partition containing nearly 60% of the nodes, and the other containing close to 40%. Thus, after 5 levels of partitioning, there will probably be one partition containing $.6^5 = .078$ of the logic. Now, an FPGA has a fixed amount of logic capacity, and since we need to ensure that each partition fits into an individual FPGA, all FPGAs must be able to hold that amount of logic. Thus, for a mapping of size N, we need a total FPGA logic capacity of $32*(.078*N) = 2.488*N$, yielding a wastage of about 60%. In contrast, if we restrict each partition to between 49% and 51%, the maximum subpartition size is $.51^5 = .035$, the required total FPGA logic capacity is $1.104*N$, and the wastage is about 10%. This is a much more reasonable overhead and we will thus restrict the partition sizes considered in this chapter to between 49%-51% of the total logic size. Note that by a similar argument we can show that partitioning algorithms that lack strong control over partition sizes, such as ratio-cut algorithms [Wei89], are unsuitable for our purposes.

As we just discussed, the greater the allowed variation in partition sizes, the better the expected partitioning results. To test this out, we applied our partitioning algorithm with various allowed size variations. The results are shown in Figure 77, and contain all of the optimizations discussed in this chapter, except: the "Clustering with 1% Max Size" only allows clusters to grow to 1% of the total circuit size, while the others allow clusters to be as large as half the allowed partition size variation (that is, maximum partition size - 50%). The "Separate clustering" line does not share clusterings, while the other lines share one clustering among all runs with the same partition size bound. As is shown, the achieved geometric means of the six circuits decreases steadily as we increase the maximum partition size. However, how we perform clustering has an impact on achieved quality, and the difference is greater for larger allowed partition sizes. Specifically, when the maximum allowed partition size is 51%, using the same clustering for all runs of the algorithm produces results as good as using separate clusterings for each run. Using a shared clustering is also faster than separate clustering, at least when all runs are performed sequentially. However, as the allowed partition size gets larger, it becomes important to use multiple different clusterings. Note that while each of these runs is performed with the connectivity clustering algorithm, the algorithm randomly chooses nodes as starting points of clusters, and thus different runs will produce somewhat different clusterings.

The reason why a single clustering does poorly for larger partition sizes is that it reduces the value of multiple runs, with almost all runs producing identical results. Specifically, as the allowed partition size grows, the allowed cluster size grows as well. When a partition is only allowed to be at most 51% of the total circuit size, no cluster can contain more than 1% of the circuit, and there will be at least 100 clusters. When the maximum partition size is 75%, a cluster can be 25% of the circuit size, and there will be relatively few top-level clusters. Thus, when partitioning is performed with this few clusters, all of the different runs will get the same results before the first unclustering, even though we create the initial partitionings randomly. Since the algorithm is totally deterministic, all of these runs will produce the same values. In fact, for partition sizes greater than 60% all ten runs of the algorithm for each circuit with shared clusterings produced the same results, and only s15850 had more than one result for a maximum partition size of 60%. To deal with this, we also ran the algorithm with a maximum cluster size of 1% of the total circuit size regardless of the maximum partition size. This technique is successful not only in better using multiple partition runs, with many different results being generated for a circuit with a specific maximum partition size, but also produces results that are up to 14% lower than the normal clustering results. However, this is still not as good as separate clusterings per run, which produces results up to 9% lower than clustering with a fixed maximum size. Because of this, for partition maximum sizes larger than 51% we use separate clusterings for each run of the partitioner.

While increasing the maximum partition size can produce lower cutsizes, most of this gain is due to improvement on the smaller circuits, while the larger circuits sometimes actually have worse results as the size variation increases. The line "Largest 2 Circuits" in Figure 77 right is the geometric mean of only the largest two circuits, s38584 and s35932, while "Smallest 4 Circuits" represents the other test cases. These lines have been scaled to be identical to the "All Circuits" value at the leftmost data-point, so that if the benefit of increasing the maximum partition size was uniform across all circuits, the three lines should line up perfectly. However, our algorithm does worse on the larger circuits as the maximum partition size increases, with the geometric mean actually increasing at the rightmost trials. The true optimum cutsize cannot get larger with larger maximum partition sizes, since when increasing the allowed partition size, the algorithm could still return a partitioning that satisfies the smaller partition bounds. Thus, the results should never increase as the maximum partition size increases, and should in general decrease. We are forced to conclude that our algorithm is unable to exploit the larger partition size bounds for the larger circuits, and in fact gets sidetracked by this extra flexibility.

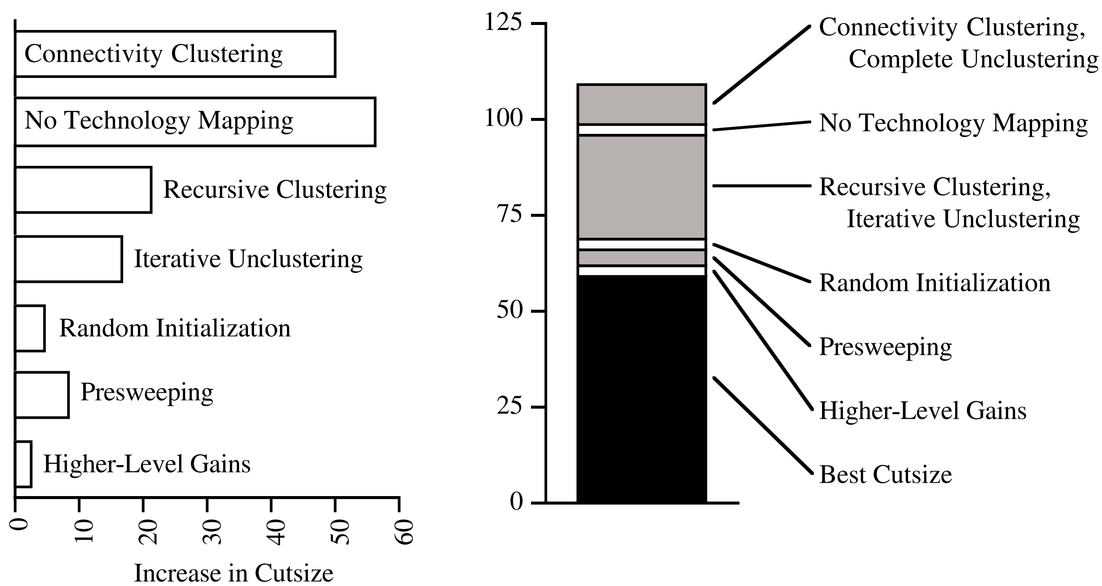


Figure 78. Two methods of determining the contribution of individual partitioning techniques to the overall results. At left are the results of comparing our best algorithm vs. taking the specified technique and replacing it with the worst alternative in this chapter. At right are the resulting cutsizes after starting with the worst algorithm, then iteratively adding the technique that gives the greatest improvement at that point.

Overall Comparison

While throughout this chapter we have discussed how individual techniques impact an overall partitioning algorithm, it is natural to wonder which of these techniques is the most important, and how much of the cutsize improvement is due to any specific technique. We have attempted to answer this question in two ways. First, we can take the comparisons we have made throughout this chapter, and bring them together into a single graph (Figure 78 left). Here we show the difference between the cutsizes generated by our best algorithm and the cutsizes generated with the same algorithm, except the specified technique has been replaced with the worst alternative considered in this chapter. For example, the “Connectivity Clustering” line is the difference between our best algorithm, which uses Connectivity clustering, and the best algorithm with Bandwidth clustering used instead. Note that the alternative used for iterative unclustering is complete clustering, not no unclustering, since complete unclustering is a very commonly used technique when any clustering is applied.

Our second comparison was made by starting with an algorithm using the worst choice for each of the techniques, and then iteratively adding whichever of the best techniques gives the greatest improvement in cutsize. Specifically, we ran the worst algorithm, and then ran it several more times, this time with one of the best techniques substituted into the mix. Whichever technique reduced the overall cutsize the most was inserted into the algorithm. We then tried running this algorithm several more times again, this time with both that best technique inserted, as well as each of the other techniques inserted one at a time. This process was repeated until all techniques were inserted. The resulting cutsizes, and the technique that was added to achieve each of these improvements, are shown in Figure 78 right. The initial, worst algorithm used was basic KLFM with seeded initialization and technology-mapped files.

As shown in the graphs in Figure 78, the results are mixed. Both of the comparisons show that connectivity clustering, recursive clustering, and iterative unclustering have a significant impact, presweeping has a modest impact, and both random initialization and higher-level gains cause only a small improvement. The results are somewhat mixed on technology-mapping, with the right comparison indicating only a small improvement, while the left comparison indicates a decrease in cutsize of almost a factor of two.

The graphs in Figure 78 give the illusion that we can pinpoint which individual techniques are responsible for what portion of the improvements in cutsizes. However, it appears that cutsize decreases are most likely due more to synergy between multiple techniques than to the sum of individual techniques. In Figure 79 we present all of the data used to generate Figure 78 right. The striped bar at left is the cutsize of the worst algorithm. The other groups of bars represent the cutsizes generated by adding each possible unused technique to the best algorithm found in the prior group of bars. Thus, the leftmost group of 6 bars represent 6 possible techniques to add to the worst algorithm, and the group of 5 bars just to the right represent the 5 possible additions to the best algorithm from the leftmost group of bars.

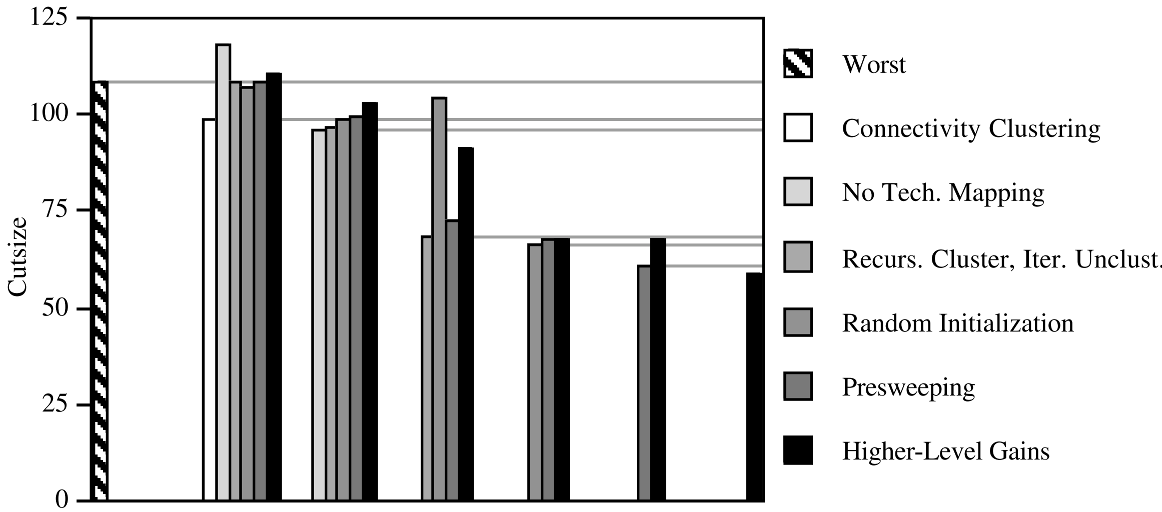


Figure 79. Details of the comparison of individual features. The bar at left is the cutsize of the worst algorithm. Each group of bars is the set of all possible improvements to the algorithm. Gray horizontal lines show the cutsize of the best choice in a given group of bars.

The important observation to be made from Figure 79 is that any specific technique can have a radically different impact on the overall cutsize depending on what other techniques are used. For example, if we take the worst algorithm and apply it to non-technology mapped files, the resulting cutsizes increase (degrade) by about 9%; once we add connectivity clustering to the worst algorithm we then see an improvement of 3% by working on non-technology mapped files. In fact, Figure 79 shows cases where we degrade the cutsize by applying random initialization, presweeping, or higher-level gains, even though all of these techniques are used in our best algorithm, and the cutsizes would increase if we removed any of these techniques. The conclusion to be reached seems to be that it is not just individual techniques that generate the best cutsizes, but it is the intelligent combination of multiple techniques, and the interactions between them, that is responsible for the strong partitioning results we achieve.

Pseudocode and Computational Complexity

In this section we present pseudocode for our algorithm, including the individual optimizations, as well as a discussion of the computational complexity of the algorithm. The overall algorithm flow is given in Figure

80. The first steps of the algorithm, reading in the input file and building the data structures, can be done in linear time. This will include constructing the pointers between nodes and signals in the circuit. Presweeping (see Figure 81) can also be done in linear time.

```
Read in gate-level netlist;
Build internal data structures;
                     // Presweeping and recursive clustering
Presweep circuit;
do {
   connectivity clustering;
} until no new clusters formed;
                     // Initialize circuit
Create random initial partitioning;
                     // Perform partitioning with 3 gain levels
repeat 10 times {
   do {
      perform KLFM inner loop;
      remove highest level of clusters;
   } until no clusters remain;
   perform final KLFM inner loop on totally unclustered circuit;
}
return best of 10 results;
```

Figure 80. Optimized Kernighan-Lin, Fiduccia-Mattheyses algorithm.

```
For each node N {
   if N is an I/O node {
      cluster N with any neighboring node;
   } else if N is a 1-input function {
      if any neighbor M is connected to N by a 2-terminal net {
         cluster N and M;
      }
   }
}
```

Figure 81. Presweeping.

Recursive connectivity-clustering (Figure 82) is more complex. In each clustering pass each node is examined, and for each highest-level cluster every signal connected to it is examined, as well as every terminal on that signal (where a terminal exists at each connection between a node and a signal). Since every signal has at least one terminal, one pass takes O(C*T) time, where T is the number of terminals, and C is the number of clusters in the highest level of the hierarchy at the beginning of the pass (where in the first pass all nodes are at the same level, including clusters formed by presweeping). Note that we do not need to examine nodes and clusters at lower levels, since such elements were not clustered during the last pass of the algorithm. Any element not clustered in one pass will never be clustered in subsequent passes, since the only reason not to cluster an element is that it has no neighbors small enough to cluster with, and this will not change in subsequent passes. Thus, if there are C candidates for clustering in one pass, there can be at most C/2 candidates in the next pass, and will probably be much less due both to clusters formed by three or more elements in one pass, as well as elements which cannot be clustered. Thus, in recursive connectivity clustering, with N = the number of nodes in the circuit, the first pass will take O(N*T), the second O(N/2*T), the third O(N/4*T), and so on. Since N*T + N/2*T + N/4*T + N/8*T + ... = 2N*T, the entire recursive connectivity clustering algorithm is O(N*T). Gatelevel netlists have only a few terminals per gate (usually two or three inputs and one output), and thus T is only a constant factor larger than N. Thus, the entire clustering process is $O(N^2)$.

```
For each cluster N at the highest level {
   If N has not yet been clustered during this pass {
      for each node or cluster M in circuit {
         bandwidth(M) = 0;
      }
      for each signal S connected to N {
         for each node or cluster X connected to S {
            bandwidth(X) = bandwidth(X) + bandwidth due to S;
         }
      }
      find neighbor Y with highest connectivity(Y, N) where
         size(Y) + size(N) <= maximum cluster size;
      if Y exists {
         cluster N and Y;
      }
   }
}
```

Figure 82. Connectivity clustering.

The random initialization algorithm is shown in Figure 83. The random assignment of numbers to elements can be done in linear time: place the elements in any order in an array. Then, randomly pick any element (including the last) and swap it with the element in the last position. Ignore this element and repeat with the second to last position. Continuing this process, a random, non-duplicated ordering can be determined in linear time. Since the other steps in the initialization simply make linear passes through this array, the time for the process is O(E), where E is the number of elements in the clustered circuit. This is obviously at worst O(N).

The KLFM algorithm, with 3rd-level gains, is shown in Figure 84. In [Fiduccia82] it was shown that the basic KLFM algorithm, without iterative unclustering and higher-level gains, is O(N). Addition of iterative unclustering increase the runtime to at most O(N log N) since there can be at most log N levels in the clustering hierarchy, and the KLFM algorithm is run once per clustering level. Note that the algorithm may actually do better than this, because at higher levels of the hierarchy there is less work to do, and an argument such as that applied for the clustering algorithm may improve the runtimes to possibly even O(N). It is unclear what the addition of our optimized higher-level gains structure does to the runtimes. A naïve implementation of the bucket structure could require a linear search through a 3-dimensional table, where each dimension can be as large as the number of nodes in the input circuit. This would yield an $O(N^3 \log N)$ algorithm. However, in our experience the optimized data structures presented here avoid much of this performance degradation, and the algorithm operates nearly as fast as the version without higher-level gains. If the higher-level gain bucket data structure does prove to be a limiting factor, higher-level gains can be removed from the algorithm with only a 4% quality degradation.

```
Max = total number of elements in clustered circuit;
Without duplication, randomly assign numbers 1..Max to elements;
below = 0;
above = total circuit size;
For (I = 1 to Max-1) {
   below = below + size(element(I));
   above = above - size(element(I));
   balance(I) = absolute value(above - below);
}
J = value of I that minimizes balance(I);
For (K = 1 to Max) {
   if K > I {
      assign element(K) to partition 1;
   } else {
      assign element(K) to partition 0;
   }
}
```

Figure 83. Random initialization.

All the other steps in the algorithm are trivial with the proper data structures. Thus, the dominant part of asymptotic complexity of this algorithm is the recursive clustering step. Ignoring higher-level gains, the overall algorithm is $O(N^2)$. It is likely that under reasonable conditions the algorithm with 3rd-level gains is also $O(N^2)$. If the algorithm is run multiple times, such as when the algorithm is recursively applied, the runtime is $O(\max(N^2, T{*}N \log N))$, where T is the number of partitioning runs performed.

Conclusions

There are numerous approaches to augmenting the basic Kernighan-Lin, Fiduccia-Mattheyses partitioning algorithm, and the proper combination is far from obvious. We have demonstrated that technology-mapping before partitioning is a poor choice, significantly impacting mapping quality. Clustering is very important, and we found that Connectivity clustering performs well, though Shortest-path clustering is a reasonable alternative. Recursive clustering and a hierarchical unclustering technique help take advantage of the full power of the clustering algorithm, with iterative unclustering being slightly preferred to edge unclustering. Augmenting the basic KLFM inner loop with at least 2nd- and 3rd-level gains improves the final results, while Dual passes are not worthwhile, and greatly increase run times. Finally, when the allowed maximum partition size is greater than 51% of the total circuit size, creating a clustering on a per-run basis produces better results than shared clustering. The table in the chapter introduction shows that applying all of these techniques generates results at least 17% better than the state-of-the-art in partitioning research.

```
Initialize bucket data structures (using 3 levels of gains)
While cutsize is reduced {
   While valid moves exist {
      Use buckets to find unlocked node in each partition that has
         the best gain (using 3 levels of gains);
      Move whichever of the two nodes has the best gain while not
         exceeding partition size bounds;
      Lock moved node;
      Update nets connected to moved nodes, and nodes connected to
         these nets;
   } endwhile;
   Backtrack to the point with minimum cutsize in move series just
      completed;
   Unlock all nodes;
} endwhile;
```

Figure 84. The KLFM inner loop, augmented with 3rd-level gains.

This chapter has included several novel techniques, or efficient implementations of existing work. We have started from the base work of Schuler and Ulrich [Schuler72] to develop an efficient, effective clustering method. We have also created the presweeping clustering pre-processor to help most algorithms handle small fanout gates. We have shown how shortest-path clustering can be implemented efficiently. We developed the edge unclustering method, which is competitive with iterative unclustering. Finally, we have extended the work of Krishnamurthy [Krishnamurthy84], both to allow higher-order gains to be applied to nets not in the cutset, and also to give an efficient implementation, even when the circuit is clustered.

Beyond the details of how exactly to construct the best partitioner, there are several important lessons to be learned. As we have seen, the only way to determine whether a given optimization to a partitioning algorithm makes sense is to actually try it out, and to consider how it interacts with other optimizations. We have shown that many of the optimizations had greater difficulty working on clustered circuits than on unclustered circuits, yet clustering seems to be important to achieve the best results. Also, many of the clustering algorithms seem to assume the circuit will be technology-mapped before partitioning, yet technology-mapping the circuit will greatly increase the cutsize of the resulting partitionings. However, it is quite possible to reach a different conclusion if we use only the basic KLFM algorithm, and not any of the numerous enhancements proposed since then. By using the basic KLFM algorithm, cutsizes are huge, and subtle effects can be ignored. While a decrease of 10 in the cutsize is not significant when cutsizes are in the hundreds, it is critical when cutsizes are in the tens. Thus, it is important that as we continue research in partitioning we properly place new concepts and optimizations in the context of what has already been discovered.

Chapter 11. Logic Partition Orderings

Introduction

Chapter 10 discussed methods of efficiently splitting a logic circuit into two parts. In order to harness these techniques for multi-FPGA systems, some method of breaking a circuit into more than two parts is necessary. There has been a significant amount of work done on multi-way partitioning . However, these works have primarily focused on problems where there are no restrictions on how the partitions are interconnected (that is, there is no reason to prefer or avoid connections between any pair of partitions). Unfortunately, in many multi-FPGA systems only a subset of the FPGAs are connected, and routing between FPGAs not directly connected will use many more external resources than routing between connected FPGAs. Works that have handled the limited connectivity problem take a significant amount of time [Roy93], possibly even exponential in the number of partitions [Vijayan90].

Our approach to the multi-FPGA partitioning problem is to harness the work on standard bipartitioning, such as that discussed in Chapter 10, as well as multi-way partitioning algorithms for some restricted situations. We do this by recursively applying the bipartitioning algorithms to the circuit until it is cut into the required number of pieces. While this approach is greedy, and thus if applied unwisely can produce poor results (Chapter 9), we can use the greediness to our advantage. If the first "greedier" cuts of the logic correspond to the most critical bottlenecks in the multi-FPGA system, then optimizing these cuts more than subsequent cuts is the correct thing to do. In fact, this method of iterative bipartitioning has already been applied to partitioning of logic within a single ASIC to simplify placement [Suaris87]. In this approach, the chip is divided recursively in half, alternating between horizontal and vertical cuts.

One issue the multi-FPGA system partitioning problem raises that ASIC partitioning does not is how to find the critical routing bottlenecks in the system. In an ASIC, all routing occurs on a flat surface, and a cut through the center of the chip represents the routing bottleneck. A multi-FPGA system can have an arbitrary topology, and it may not be obvious where the critical bottleneck is. What is necessary is to find a partition ordering which specifies the critical bottlenecks in the multi-FPGA system and determines the order of cuts to make in any logic being mapped to that topology. In many cases we can handle this by requiring that the designer of the multi-FPGA system explicitly specify the proper cuts to make. However, there are several issues in multi-FPGA systems that argue for an automatic method for finding these bottlenecks.

The primary argument for automatic methods to find critical bottlenecks is the current trend towards more flexible multi-FPGA system architectures. Instead of fixed systems with a finite mix of resources, many current systems are offering much greater flexibility (see Chapter 5). For some, it is primarily the ability to connect together small systems to handle much larger problems, possibly with the additional capability of adding other non-FPGA chips. Other systems have a fixed connection pattern between daughter card or chip locations, but allow a large variety of types and capacities of resources to be inserted. There is also the possibility of much greater flexibility, with systems that allow very complex and customized topologies to be built (such as our Springbok system, described in Chapter 6). While rules for static partition orderings might be generated in some cases, the flexibility of current and future systems forces us to adopt an automatic approach to the partition ordering problem. A software system capable of automatically adapting to an arbitrary multi-FPGA topology also becomes a valuable tool for people building their own multi-FPGA system. Since the tool automatically adapts to the topology, it can be used by designers of systems who have little or no knowledge of partitioning techniques.

Another trend is towards faster automatic mapping turnaround times. Just as was found for software compilation, a faster turnaround time from specification to implementation allows a more interactive use of a multi-FPGA system, making these systems more attractive. One way to speed up the mapping process is to retain most of a previous mapping after a small change, and simply update incrementally. Even when starting from scratch, if the mapping process fails because it uses too much of some resource, it is better to remap only that part of the system where the failure occurred (possibly with neighboring parts that have resources to spare) than restart the entire mapping process. In either case, the area(s) to be remapped are likely to be very irregular. Even if it is easy for the designers to determine a partition ordering for the entire system, precomputing a partition ordering for all possible subsets is very difficult. Again, the solution to this problem is to develop an automatic algorithm to determine partition orderings.

In the rest of this chapter we discuss our solution to a problem we believe has not yet been investigated: determining how to iteratively apply 2-way and N-way partitioning steps to best map a circuit onto chips in a fixed topology.

Partition Ordering Challenges

Before we discuss solutions, we need to define some terms. In this chapter, the *topology* is the FPGAs and connections built into the multi-FPGA system. A *partition* is a subset of the topology where all FPGAs are on the same side of all cuts made in the system. A partition may be broken into *groups* by a candidate cut, with each group being a set of FPGAs from the partition which are still connected after the cut is applied.

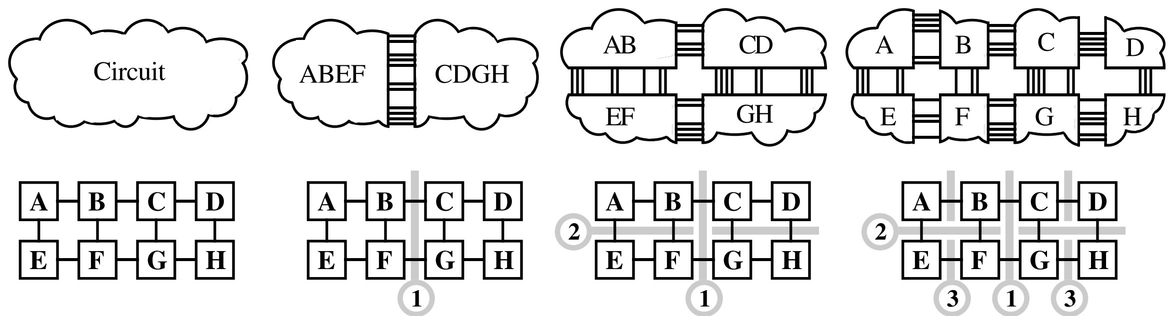


Figure 85. Example of iterative bipartitioning. The circuit (top left) is partitioned onto the multi-FPGA topology (bottom left) in a series of steps. Each partitioning corresponds to the most critical bottleneck remaining in the multi-FPGA system, and after each partitioning the placement of the logic is restricted to a subset of the topology (labeling on circuit partitions).

We can approach the problem of determining a partitioning order as a partitioning problem on the multi-FPGA topology itself. That is, we recursively split up the topology graph, attempting to optimize some cost metric on the routing resources cut. Then, we split the logic graph in the same manner, restricting the logic on each side of the split to the available capacity on either side of the corresponding cut of the topology, while attempting to keep the number of signals cut in the logic graph to be less than the amount of routing resources crossing the cut in the topology. An example of this is in Figure 85.

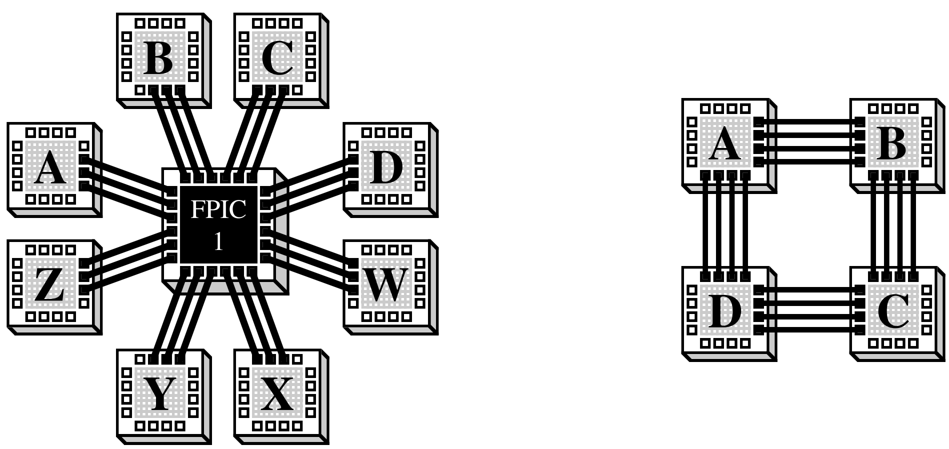


Figure 86. Example topology for the discussion of connectedness of partitions (left), and for the discussion of multiple partition creation (right).

There are two important issues that must be considered in any partition ordering computation: partitions must be connected, and a cost metric for evaluating cuts must be determined. The issue of connected groupings is demonstrated in Figure 86 left. The topology shown has a central chip **R** that acts as a routing hub, and eight other FPGAs connected only to **R**. If we apply a standard partitioning algorithm to this topology, it is likely that the partition that does not include **R** will include two or more FPGAs. For example, one likely cut has **A**-**D** on one side, and **W**-**Z** plus **R** on the other. The problem with this cut occurs when we try to partition the logic according to this cut scheme. If we assume all the connections between **R** and the other FPGAs contain 10 wires, then when we do the cut described above we will allow 40 nets to be cut. However, this ignores the fact that once the **A**-**D** partition is broken up into the individual FPGAs, there may need to be connections between these FPGAs. However, there are no wires within the **A**-**D** partition to handle these connections, and the wires needed to make these connections may have already been allocated for signals to the **W**-**Z** partition during the first split. What is necessary is to require that any partitioning of the circuit produce connected partitions. Thus, there will be at least one possible path within a partition to carry signals between its FPGAs (ensuring that there are enough wires is the responsibility of the cost metric for evaluating partitionings, discussed below). Thus, in a bipartitioning of the topology in Figure 86 left, all but one of the FPGAs, including **R**, would have to be in one partition, and the other FPGA in the other partition (better methods for this topology will be discussed later).

In the introduction, we discussed finding the critical bottlenecks in the system, and partitioning the logic accordingly. However, we didn't define what a critical bottleneck is. For our purposes, we define the critical bottleneck as that cut through the system that most restricts the routing. That is, assume that we take a very small circuit and map it randomly to our topology. If it fits, we repeat this with a slightly larger circuit, until we can no longer route the circuit. This will yield at least one set of edges that are completely saturated with signals, and which splits the circuit into two or more pieces. Since these edges are the first that become cut, they are probably the edges that will most restrict the mapping of a circuit to this topology, and represent where the most care should be taken during partitioning. The critical bottleneck is within this set of edges. We can evaluate a cut in a topology (which splits it into two parts) and determine its criticality. If L_1 is the total logic capacity of all FPGAs on one side of a cut, and L_2 is the capacity on the other side, then out of $(L_1+L_2)^2$ signals, $2*L_1*L_2$ signals will cross this cut in a randomly mapped circuit (The factor of 2 is because we treat the source and destination separately, so a route from A to B is different that a route from B to A). Thus, $(2*L_1*L_2)/(L_1+L_2)^2$ is the percentage of all nets crossing this cut. Since (L_1+L_2) is a constant for a given topology, we can ignore it. Let $wire_{12}$ be the number of edges in the topology crossing the cut. Thus, the cut that will saturate the earliest is the cut with the least edges to handle the most routing. To define the criticality of a cut, we use the ratio of $wire_{12}$ to the percentage of nets crossing this edge (without the constant terms), which is $wire_{12}/(L_1*L_2)$. Note that this is the standard ratiocut metric [Wei89]. The lower the ratiocut value, the more critical the cut in the topology.

Basic Partition Ordering Algorithm

As we discussed earlier, what we want is an algorithm that will determine how to partition logic onto a given topology. It needs to find a cut in the topology that reflects the most constrained routing (assuming random placement of logic), while ensuring that the partitions formed are connected. There is an existing algorithm that provides much of this functionality. As proposed by Yeh, Cheng, and Lin [Yeh92], we can partition a graph by iteratively choosing random start and end points of a route. We then find the shortest path between these two points, and use up a small fraction of the resources on this path to handle this route. Distances are *exp(10*flow/cap)*, where *cap* is the capacity of the edge, and *flow* is the amount of that capacity already used by previous routes. A net whose capacity has been used up is removed from the system, and no longer connects between any of its terminals. The iteration of choosing random points and then routing between them is repeated until there are no longer any resources available to route between the next random pair of points. At this stage, we have broken the circuit up into at least two groups, where a group is a set of all nodes still connected together. Note that there may in fact be more than two partitions created. For example, if we are partitioning three FPGAs connected by only a three-terminal wire, the system will be broken into three partitions. If we are partitioning four FPGAs in a mesh (Figure 86 right), it may be broken into four partitions, since the resources on all four edges may be used up.

There are several modifications that we have made to this basic algorithm. First, using the edge length in the original algorithm, namely *exp(10*flow/cap)*, yields real-valued path lengths. During the shortest-path calculation, we have a queue of currently found shortest paths. We remove the shortest of these paths, and add the neighbors of this path back into the queue, as long as that neighbor has not yet been reached by a shorter path. Since the edge lengths are real values, we need to use a tree data structure for the queue, resulting in *O(log n)* time for each insertion and deletion. To fix this, we have changed the algorithm to use an edge length of $round(exp_2(10*flow/cap))$. Since we round the edge lengths, all path lengths are integers. This, plus the fact that the maximum length of an edge is 2^{10} (or 1024), means we can very efficiently implement the shortest path calculation. Instead of a tree used as a queue of current shortest paths, we have an array of lists. Since the path we are extending at each step is the smallest still in the queue, and since the maximum path we can add to the list is at most 1024 longer than the current path, we only need keep 1025 separate active queues (see Figure 87). Thus, we can implement the queue as an array of 1025 elements (with element j representing paths of length j+i*1025, for i≥0). In this way, insertions *O(1)*. Deletions are also *O(1)* since at most we must look at each queue location for the next element, and there are a constant number of locations.

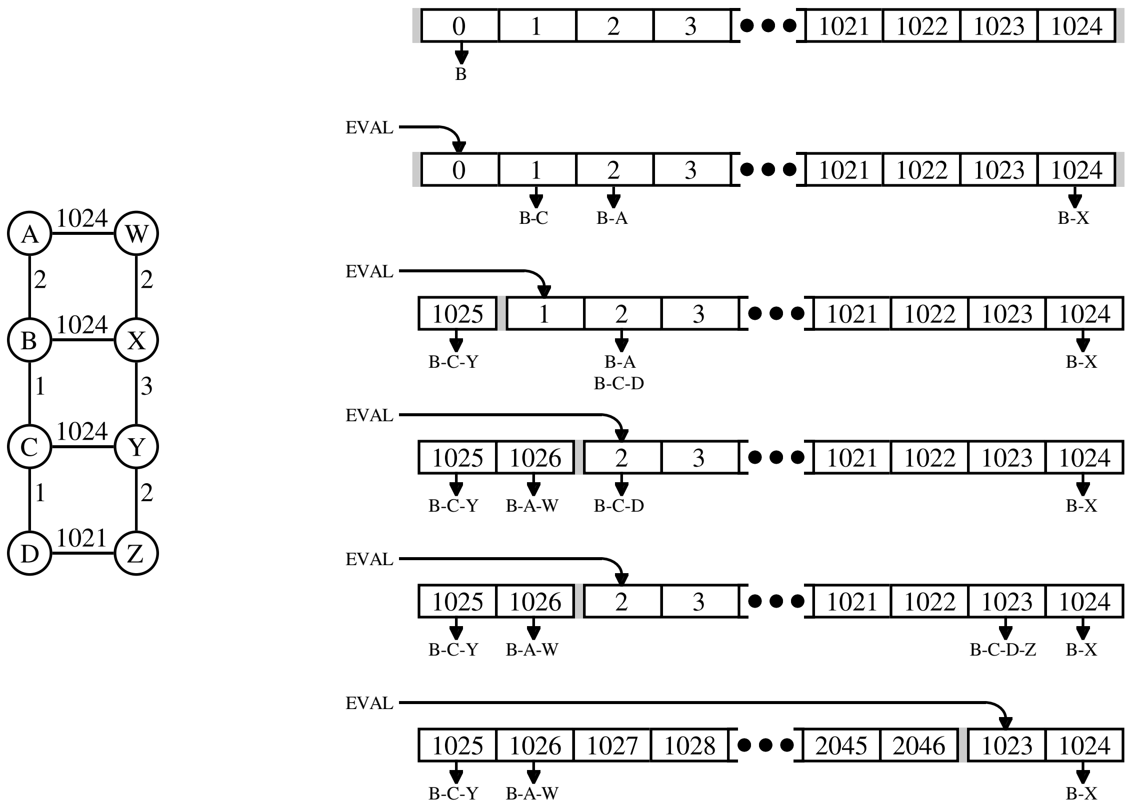


Figure 87. Queue formulation of the shortest-path algorithm. In attempting to find the shortest path from **B** to **Z** in figure at left (number on edges are edge lengths), the queue states are shown in order at right, from top (earliest) to bottom (latest). In the first step, the source is inserted in the 0 bucket. Then iteratively the shortest path is removed, and the neighbors are added to the appropriate bucket. Notice that the 0 bucket is reused as the 1025 bucket, 1 as 1026, and so on. In the final step, the shortest path from **B** to **Z** is found in bucket 1023, and the algorithm finishes.

Another issue is the distribution of sources and sinks of random routes in the algorithm. Not all chips in a multi-FPGA system are the same, and the choice of source-destination pairs needs to reflect this. Most obviously, if a routing-only FPGA or FPIC is included in the topology, there should be no sources or sinks assigned to that chip. The capacity of logic-bearing chips should also be considered. If one FPGA in the system has twice the capacity of another, then twice as much logic will likely be assigned to it, and twice as many random routes should start and end there. Thus, the distribution of random routes should be directly related to the logic capacities of the chips in the system. Note that if a route starts and ends in the same FPGA it can be ignored. One final consideration is the handling of external inputs and outputs. In many multi-FPGA systems there will be specific ports that are used for communication with the environment, or with non-FPGA chips within the multi-FPGA system. Note that for our purposes FPICs are treated as FPGAs with zero logic capacity, so "non-FPGA chips" refers to memories, microprocessors, and other devices that cannot be assigned random logic nor can be used to route inter-FPGA signals. To handle the routing demands caused by these external connections, the FPGAs to which they are connected have their capacities increased, which increases the portions of random routes beginning or ending in that chip.

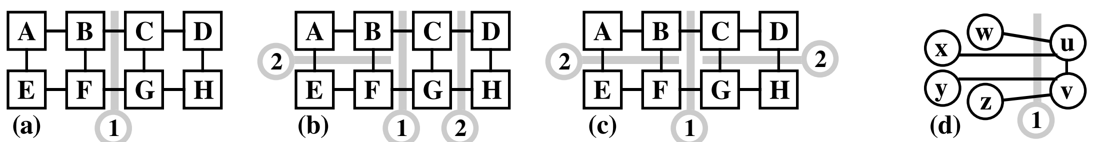


Figure 88. Examples of multiple cuts in a topology (cuts are in gray, with lower numbers indicating earlier cuts). If the initial cut (a) is used to split the entire system, uncoordinated subcuts (b) may occur. Keeping the system together would generate more coordinated cuts (c). An example circuit is also shown (d).

Now that we know how to perform a single cut of the topology, the question arises of how to recursively apply this algorithm to generate multiple cuts in the system. The obvious approach is to use the cut generated in each step to break the topology into two separate topologies, and handle these halves independently. However, consider the system shown in Figure 88a. If we use the first cutline generated (gray line labeled "1") to split the topology, then the two halves will independently choose either a horizontal or vertical cutline (Figure 88b). There are two problems with this. First, there is the issue of terminal propagation [Dunlop85]. When we perform the split between **AB** and **EF**, there often needs to be information transferred between both sides of the previous cut. For example, say that we are mapping the circuit in Figure 88 d, and have already partitioned **u** and **v** onto **CDGH**, and **w**-**z** onto **ABEF**. It should be obvious that **w** and **x** should be partitioned into the same group, so that **u** can be placed in the adjacent FPGA on the other side of the first cut (for example **w** and **x** can be put into **B**, and **u** into **C**). If this is not done, one or more extra vertical connections will be necessary. However, with the partition ordering specified in Figure 88b, there is no way for the partitioner to figure this out, since **u** and **v** have not yet been partitioned into the top or bottom half of the system. However, if we use the partition ordering shown in Figure 88 c, **u** and **v** will be assigned to either the top or bottom half of the topology, and terminal propagation algorithms [Dunlop85] can be applied to properly group together **w** and **x**. A second issue is multi-section algorithms [Suaris87, Bapat91]. These algorithms can often take two or more intersecting cuts, such as those in Figure 88c, and perform them simultaneously (both cuts labeled 2 in the figure would be considered as one single cut, so all three cuts shown could be done simultaneously). These algorithms would be able to recognize a situation such as the mapping in Figure 88d, and handle it correctly.

The answer to the previous issues is to perform similar cuts on either side of a previous cut simultaneously. There is a simple way to accomplish this in our current algorithm. Instead of breaking the topology up into two independent halves after a cut is found, we instead keep the topology together, and restrict the random routes we consider. That is, we only consider random routes that start and end in the same side of all previous cuts. So in Figure 88a, after the cut labeled 1 has been performed, the topology is kept intact. Routes between **A**, **B**, **E**, and **F** would be considered, but no routing between FPGAs on both sides, such as between **A** and **C**, would be generated. However, the shortest paths found in the algorithm could cross the previous cut. So, if a large number of routes had already used some of the vertical wires **AE** and **BF**, a route from **B** to **F** could choose to go through **C** and **G**. In this way, the two sides of the cut will tend to affect each other, and generating similar cuts in both partitions is likely. Note that it is possible that edges moving across a previous cut will become saturated before others, and we might expect to find this previous bottleneck again. However, our stopping condition on the iteration is when there is no path between the source and sink of a random route. Since we never consider routes between FPGAs on both sides of a cut, the iteration continues until at least one of the partitions is split.

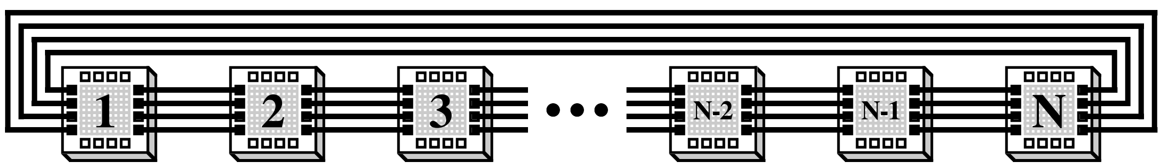


Figure 89. Ring topology, one of the most symmetric topologies.

As was alluded to earlier, it is possible (and quite likely in topologies such as tori and rings) that the sectioning found will split a partition into more than two groups. For example, consider a ring of N FPGAs where FPGA i is connected to FPGAs $(i+1)$ and $(i-1)$, with FPGA N connected to FPGA 1 (see Figure 89). If all the connections and FPGAs have the same capacities, there is no obvious point to cut the topology, and in fact many or all of the connections may be cut in one partitioning step. If we allow partitions to be broken into multiple groups, we would then need an algorithm that can do multi-way partitioning with routing constraints between partitions, the very thing this algorithm is built to avoid. The solution to this is to detect cases where a partition is being broken into multiple groups, and combine groups together until no partition is broken into more than two groups.

To combine groups together, our algorithm selects the two largest groups (based on the total logic capacities of the FPGAs in the group) as seeds. It then iteratively selects the largest remaining subgroup, and combines it with whichever of the two seeds has the greatest ratio-cut with this group (i.e., largest $wire_{ij}/(size_i * size_j)$, where i is the largest remaining group, and j is one of the two seeds). Note that if the largest remaining group isn't connected to either of the seeds, we ignore it and move on, returning to it only once a group is successfully combined with a seed. The reasoning behind this combining algorithm is as follows: we start with the largest groups, since this will hopefully allow us to best balance the sizes of the two final groups created. We merge based on ratio cut, since we are trying to minimize the overall ratio cut. We do not combine unconnected groups, since (as discussed in the chapter introduction) all partitions created must be connected.

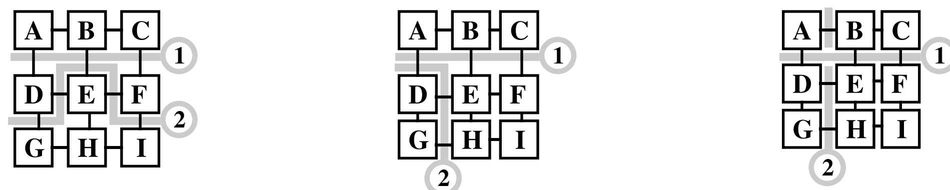


Figure 90. A failure of Global combining (left), and a comparison of ratio-cut metrics (center and right).

This combining algorithm can be turned into either a Global or a Local version. The Global version ignores partitions, and considers the entire topology at once. The groups considered are the sets of connected FPGAs after only applying the final cut. The Local version works on one partition at a time, and the groups considered are the sets of FPGAs in a partition that are connected after all the cuts (including the current one) are applied. The Global version produces better results, since it considers the entire topology at once, but is not guaranteed to successfully combine groups. For example, if we are partitioning the topology in Figure 88a, and have already found cut #1, we could end up breaking both partitions into multiple groups. The Local algorithm would ignore global concerns, and could generate the results in Figure 88b. While the Global algorithm has a good chance of generating the partitioning in Figure 88c, it might also fail to produce a good partitioning. One possibility is that it would simply find cut line #1 again, and the algorithm would make no progress. A second possibility is demonstrated in Figure 90 left. As shown, the second cut breaks the topology into two halves, and could be found by the Global algorithm. However, with the pre-existing cut #1, this breaks the lower partition into three groups, and is thus a failure. The Local algorithm will always ensure that a partition is broken into at most two groups.

A final issue in the basic algorithm is that of controlling the randomness of the process. Since we use a random process to select the cuts to make in the system, it is possible that any specific cut selected by the algorithm will be a poor choice. This is particularly true if the Local combining process is used. To deal with this, we perform multiple runs of the algorithm for each cut, and select the best at each step. Specifically, we make ten passes of the cut selection algorithm, as well as the combining algorithms (if necessary), to find the first cut. The best cut from all these runs is selected, and used as the starting point for ten runs to find the second cut. Each cut is selected from ten individual runs, and the best is used as the starting point for subsequent passes. Since we are trying to find the best ratio-cut partitioning, we could evaluate a cut based on a multi-partition ratio-cut metric [Chan93a]. This cost is shown in Equation 3.

$$\frac{1}{(k-1)}\sum_{i=1}^{k}\left(\frac{RoutingCapacity_i}{LogicCapacity_i}\right)$$

Equation 3. Multi-partition ratio-cut metric.

K is the current number of partitions, $RoutingCapacity_i$ is the amount of routing capacity connected between partition i and any other partition, and $LogicCapacity_i$ is the total logic capacity of partition i. The lower the ratio-cut, the better. One problem with this formulation is that it tends to penalize larger numbers of partitions. For example, the cuts shown in Figure 90 center have a ratio-cut cost of $(3\text{-}1)^{-1}(3/3 + 3/2 + 4/4) = 1.75$, and the cuts in Figure 90 right have a cost of $(4\text{-}1)^{-1}(3/2 + 2/1 + 3/2 + 4/4) = 2.0$. Thus, the cuts in Figure 90 center are preferred by the ratio-cut metric, but the cuts in Figure 90 right are actually better for our purposes. The reason for this is that the standard ratio-cut metric tends to favor less partitions, while for our purposes it is usually better to break a topology into a larger number of partitions, which reduces the total number of cuts necessary. To fix this, we divide the ratio-cut metric given above by k (the number of partitions) since this will tend to favor more partitions. This results in costs of 1.75/3 = .5833 for Figure 90 center, and 2.0/4 = .5 for Figure 90 right, which indicates that Figure 90 right has the preferred cuts. One final piece is necessary: it is possible in the ten runs that some cuts will be generated directly from the partitioning algorithm, some will result from the Global combining algorithm, and some from the Local combining algorithm. The Global algorithm produces better results than the Local algorithm, and results that need no combining are better than results from the Global algorithm. Thus, we always prefer results generated by the partitioning algorithm above any combining algorithm results, and we prefer results of the Global algorithm above results from the Local algorithm. This preference takes precedence over the ratio-cut metric.

Algorithm Extensions

While the algorithm described so far is capable of handling arbitrary partitioning situations, there are two extensions that can be made to greatly improve the results. These are parallelizing cuts, and clustering for multi-way partitioning.

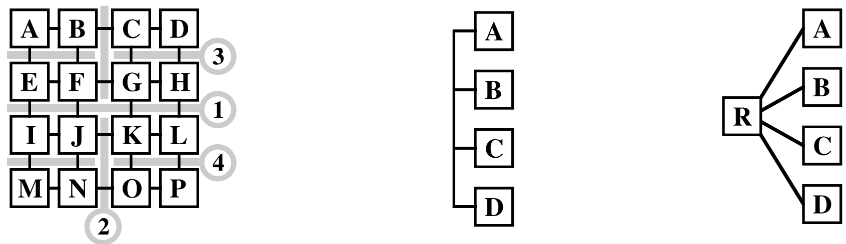


Figure 91. Example of parallelizeable cuts (left), and multi-way partitioning opportunities (center and right).

In parallelizing cuts, we observe that two or more cuts can be performed in parallel if they do not cut the same partition into more than two groups. For example, in Figure 91 left, given the set of cuts shown we would have to run four partitionings in series. We cannot combine cuts 1 and 2 since they cut the initial partition (the entire system) into four groups. We cannot combine cuts 2 and 3 since they cut partition **A-H** into four groups. We do not consider combining cuts that are not sequential since that would disrupt the order of partitionings. However, the last potential pair to combine, namely cuts 3 and 4, can be combined since they do not both cut the same partition. By combining these cuts, we only have to perform three cuts in series, potentially speeding up the partitioning process in a multiprocessing environment. To parallelize cuts, we consider each pair of sequential cuts, from earliest to latest, and combine all pairs that do not cut the same partition. Note that a similar algorithm could parallelize cuts to allow a quadrisection algorithm [Suaris87] to be applied. Quadrisection breaks a partition into four separate groups, while handling the connectivity constraints between the partitions. So, in Figure 91 left we could combine cuts 1 and 2 together, further accelerating the partitioning process.

The final extension we made to our algorithm is clustering to enable multi-way partitioning. As we mentioned earlier, the reason we do not use standard multi-way partitioning algorithms for the multi-FPGA partitioning problem is that we normally have to optimize for inter-FPGA routing capacity constraints, while standard multi-way algorithms do not allow inter-partition constraints. What these algorithms do optimize for is either the total number of nets connecting logic in two or more partitions (the net-cut metric), or the total number of partitions touched by each of these cut nets (the pin-cut metric). For example, if one net touches partitions **A** and **B**, and another touches **A**, **B**, and **C**, then the net-cut metric yields a cost of 2, and the pin-cut metric yields a cost of 5. It turns out that there are places in some topologies where a multi-way partitioning under the net-cut or pin-cut metric is exactly the right solution. For example, in the topology in Figure 91 center, which has only a set of buses connecting the individual FPGAs, there are no specific routing capacities between individual FPGAs. The only thing that is important is minimizing the number of nets moving between the partitions, since each net uses up one of the inter-FPGA buses, regardless of which or how many FPGAs it touches. In this situation, the net-cut metric is correct, and we can partition into all of the FPGAs simultaneously. Note that if there were other wires between a subset of the FPGAs in Figure 91 center we could not perform this 4-way partitioning, since there would be capacity constraints between individual FPGAs (most multi-FPGA systems with buses also have other connections, so we will ignore the net-cut model for multi-way partitioning). In the topology in Figure 91 right, there are four FPGAs connected to only a purely routing chip **R**. In this situation, it is necessary to limit the number of wires going between **R** and the other FPGAs, but there is no other limitation on the number or connectivity of the inter-FPGA nets. For example, if the wires in the topology had a capacity of one, it would make no difference if there was one net connecting all four partitions, or two connections between two different FPGAs (i.e., **A**-**B** and **C**-**D**). In this situation, which is common in two-level topologies (Chapter 5), a multi-way partitioning based on the pin-cut model is the proper way to partition.

We have the following algorithm for finding multi-way partitioning opportunities. Before we do any partitioning, we search for multi-way partitioning opportunities. We remove all routing-only nodes from the system, and find all maximally connected subcomponents of the topology. We examine in order (from smallest to largest in total capacity) each subcomponent, and examine if it can be multi-way partitioned. Specifically, for multi-way partitioning to be performed, this subcomponent must be connected to only one routing-only node. Also, removal of this routing node from the complete topology must break the topology into at least three connected components (if it didn't, the k-way partitioning that could be performed here would be 1-way or 2-way, and thus best left to the normal algorithm to discover). If these constraints can be met, we group together the original component, the routing node, and all but the largest of the other components found in the complete topology minus the one routing-only node. This cluster becomes a new node, which replaces all nodes being grouped, and all edges that were incident to grouped nodes are connected to the new node instead. The cluster node's logic capacity is equal to the total logic capacity of all nodes being grouped. This process of clustering multi-way partitioning opportunities continues until no more clusterings can be performed. During the partitioning process, once one of these clusters is broken off into a partition by itself, or with only other routing-only chips, we add the multi-way partitioning of this cluster to the list of partitionings to be performed. The cluster node is replaced with the nodes it clusters, and the algorithm is allowed to perform further partitioning of the nodes that were in the cluster. Note that if the first cut found causes a clustered node to be unclustered next, we make this multi-way partitioning the first cut performed.

Experiments

All of the experiments in this section were performed with the algorithms described in this chapter. The topologies are a mix of a set of existing multi-FPGA systems, as well as a few hypothetical mesh structures included to demonstrate some specific features of our algorithm. All processing occurred on a SPARC-10. For each cut, we chose the best of ten runs. The flow increment (the amount of capacity used up by each random route) was ten percent of the average capacity of the wires in the system, after all wires connecting the same destinations were combined.

Figure 92. The DECPeRLe-1 board. Connections from **U**,**V**,**W**, and **X** going off to the right connect to each of **A**-**P**. Some connections are not shown.

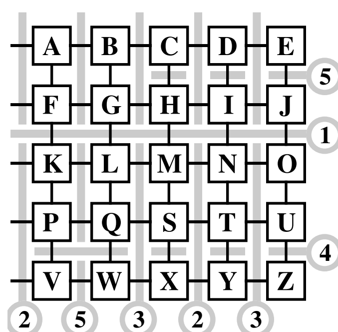


Figure 93. The NTT board. Connections going off the left side connect to the corresponding FPGA on the right side. All chips still connected are cut by the 6th cut line. Note that there is 50% more capacity on the horizontal wires than on the vertical ones.

The first three figures demonstrate our algorithm on three different current topologies. Figure 92 is the DECPeRLe-1 board [Vuillemin95]. Note that the connections dangling off of **U**-**X** are actually buses connecting to all the FPGAs **A**-**P** in the center. As can be seen, the algorithm first partitions three times through the left half, which are crossbars connected throughout the system plus the host interface. It then partitions up the main mesh twice (center), and finishes both halves in the last two cuts.

In Figure 93 is the NTT board [Yamada94], which is a mesh with wrap-around connections from left to right. While most of the cuts are reasonable, note cut #5. The algorithm actually attempted to do both a horizontal and a vertical cut at once. The partition **ABFG** is actually split several times in this step, but the combining algorithm reduces it to a single cut. Note that cut #6 is not depicted - it splits all remaining 2-FPGA partitions in half.

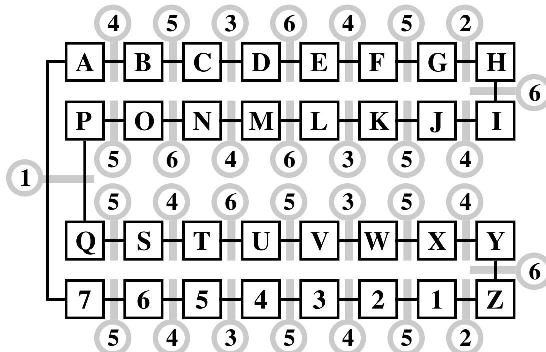


Figure 94. The Splash topology. Note that the cuts are somewhat imbalanced because of limited resources between FPGAs **A** and **7**.

In Figure 94 we partition the Splash board [Gokhale90]. Note that while there are 68 connections between most of the neighboring FPGAs, there are only 35 connections between **A** and **7**. Because of this, the cuts made in steps 2 and 3 are shifted over one FPGA from an even split, and it takes a total of 6 steps to subdivide the entire system. If we change the topology, putting 68 connections between **A** and **7**, the algorithm completes the partitioning in 5 steps, performing even splits in each step. Note that if we did not apply our parallelization of cuts technique to the original topology, it would actually require 17 separate partitionings.

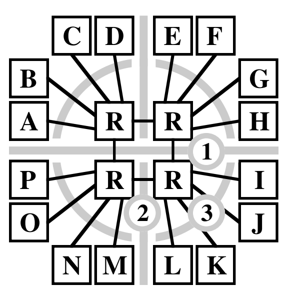


Figure 95. Demonstration of the clustering to discover k-way partitioning opportunities.

To demonstrate our method for finding k-way partitioning opportunities, we ran our algorithm on the topology in Figure 95 . This two-level topology is a 2x2 mesh of routing FPGAs, with each routing FPGA connected to four logic-bearing FPGAs. The algorithm clusters together the logic FPGAs connected to each routing FPGA. The first two cuts separate the routing FPGAs, and then the four 4-way partitionings can all be accomplished simultaneously (the latter was done with the help of the parallelization routines).

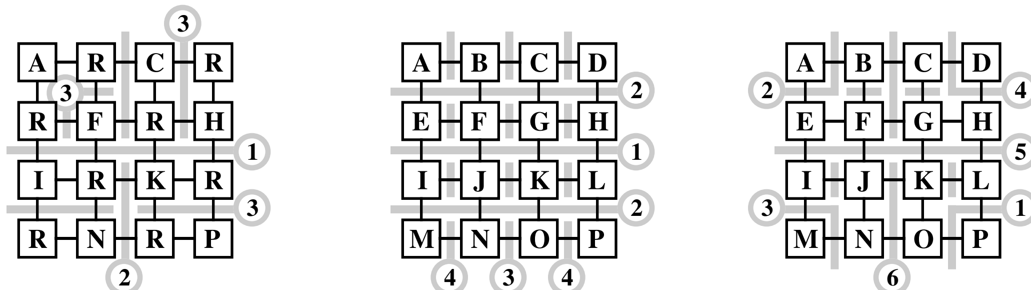


Figure 96. Demonstrations of how the algorithm reacts to different topological parameters. At left, the array is half routing-only chips (those marked with R). At center, a normal array with three times as much routing capacity on the horizontal edges as on the vertical edges. At right, all edges have equal capacity, but the corner four FPGAs have ten times as much logic capacity as the others (to simulate external connections).

As a final test of our system, we took a 4x4 mesh and varied some of the parameters we have discussed previously. Figure 96 left demonstrates routing-only nodes (labeled “R”). As can be seen, since there is no need to separate a logic-bearing node from routing-only nodes, the algorithm stops after 3 steps. In Figure 96 center, we show a topology where the horizontal edges have 3 times the capacity as the vertical edges. As might be expected, the algorithm adjusts by partitioning the vertical edges before it cuts the horizontal ones, since the excess capacity makes the horizontal edges much less critical. Finally, Figure 96 right assigns the same routing capacity to all edges, but gives the four corner FPGAs 10 times as much logic capacity as the other FPGAs (to simulate external connections for example). As can be seen, the algorithm decides it needs to deal with the connections to these FPGAs first, and isolates them in the first four cuts. It then cuts the rest of the topology. Note that the four unlabeled gray lines are cuts from step #7, and the remaining uncut edges are all cut in step #8.

Pseudocode and Computational Complexity

In this section we present the pseudocode for the logic partition ordering algorithm, as well as its computational complexity. The overall algorithm is given in Figure 97 .

```
read in topology;
group nets connected to same set of chips, incrementing capacity;
cluster for multi-way partitioning;

While two logic-bearing FPGAs are in the same partition {
   compute probabilities;
   repeat 10 times {
      saturate network;
      if more than two groups are formed {
         apply global combining;
         if global combining failed {
            apply local combining;
         }
      }
   }
   select best of 10 cuts;
   if cluster is isolated {
      select multi-way partitioning of cluster;
      remove clustering;
   }
}
parallelize cuts;
```

Figure 97. Logic partition ordering algorithm.

```
remove routing-only nodes from topology;
find maximally connected components;
replace routing-only node;
foreach component C, smallest to largest {
   if C is connected only to one routing-only node R {
      remove node R from topology;
      if there are 3 or more maximally connected components {
         cluster R and all but largest connected component;
      }
      replace node R;
      restart clustering algorithm on clustered topology;
   }
}
```

Figure 98. Clustering algorithm for multi-way partitioning.

In [Yeh92] the algorithm that formed the basis of this work was presented, and it was shown that it has a complexity of O(R*M*N log N). N is the number of nodes and M the number of nets in the graph being partitioned. Note that for our purposes, M is actually the number of different types of nets, where all nets connecting exactly the same set of nodes are in the same category. Multiple nets in the same category in a multi-FPGA systems simply increase the capacity on the single edge that represents the entire category. R is the number of flow increments necessary to saturate the largest-capacity net in the circuit. This flow increment can easily be modified to always keep the value of R constant, and thus R can be ignored. The N log N is the time to find the shortest path between the random points in the original algorithm's real-valued edge length model. Because of our reformulation of the edge lengths, which allows constant-time access to the queue, the shortest-path search can be done in O(T), where T is the number of terminals in the graph (one terminal exists at each connection of nets to nodes in the graph). In any reasonable multi-FPGA system the number of terminals will be only a small constant factor larger than the number of nets in the system (almost all nets will be 2-terminal nets, with only a few large fanout nets). Thus, one cut in the system can be found in $O(M^2)$. All other portions of the algorithm are dominated by this runtime.

```
total_probability = 0.0;
foreach group G {
   cum = total capacity of nodes in group;
   cum_prob(G) = 0.0;
   foreach node N in group G
      cum_prob(G) = cum_prob(G) + (cum - capacity(N))*capacity(N);
   total_probability = total_probability + cum_probability(G);
}
foreach group G
   probability(G) = cum_probability(G) / total_probability;
```

Figure 99. Algorithm for computing probabilities for random route distribution. A group is the set of nodes that have been on the same side of all previous cuts. Probability(G) is the portion of random routes that should occur within a given group, and is based on the capacity of the nodes in that group. Note that a group with only one logic-bearing node will have a 0 probability of routes.

```
set flow on all edges to 0;
set length on all edges to 1;
do forever {
   randomly pick group G based on probabilities;
   randomly pick source node S in group G based on capacities;
   randomly pick different destination node D in group G
      based on capacities;
   find shortest-path P from S to D using queue structure;
   if no path exists
      return topology;
   increment flow on all edges in P;
   recompute edge length on all edges in P;
}
```

Figure 100. Network saturation algorithm.

The combining algorithm will have to be run multiple times to create a complete partitioning ordering for a multi-FPGA system. While for many situations there will only need to be O(log N) cuts in the multi-FPGA system, it is possible to need up to O(N) cuts. This is true for a linear array of FPGAs, where each of the

N-1 links in the array will need to be cut separately. Thus, the complete runtime for this algorithm is $O(N*M^2)$. While a cubic complexity may not be ideal, one must realize that N and M are unlikely to grow that quickly in future. While FPGAs will increase in internal capacity and external I/Os, neither of these numbers increases N nor M (remember that more nets going to the same location simply increase the flow on the edge in the graph being partitioned, and greater capacity simply alters the distribution of random sources and sinks of routes). Thus, improvements in multi-FPGA systems are likely to come mostly from improved chips, while the number of FPGAs and the types of routing are unlikely to change much in the future.

```
partition topology based only on current cuts;
find largest group L, second-largest group S;
repeat until no groups other than L and S remain {
   find largest group G other than L and S which is connected
      to L and/or S;
   L_ratio = bandwidth(L, G) / (size(L) * size (G));
   S_ratio = bandwidth(S, G) / (size(S) * size (G));
   if S_ratio >= L_ratio
      combine S and G;
   else
      combine L and G;
   if size(S) > size (L)
      swap(L, S);
}
```

Figure 101. Global combining algorithm.

```
foreach group G in topology, ignoring current cut
   apply global combining to group G only;
```

Figure 102. Local combining algorithm.

```
max = total number of cuts in partition ordering;
base = 1;
target = 2;
while next <= max {
   if cut(base) and cut(next) do not subdivide the same group {
      combine cut(base) and cut(next);
      next = next + 1;
   } else {
      base = next;
      next = next + 1;
   }
}
```

Figure 103. Cut parallelization algorithm.

Conclusions

In this chapter we have considered applying standard partitioning algorithms to multi-FPGA systems. We have detailed an algorithm that determines the order to perform bipartitioning by finding the critical bottlenecks, while ensuring that all partitions created are connected. We have also detailed a method for increasing the parallelism, and decreasing the required run time, of partitioning in a multiprocessing environment. This technique is also capable of finding multi-sectioning opportunities. Finally, we have included a method of determining when multi-way partitioning can be used. In this way, we have developed an integrated method for best harnessing the numerous existing bipartitioning, multi-sectioning, and multi-way partitioning algorithms. The algorithm is efficient, and handles arbitrary topologies and heterogeneous FPGAs.

As mentioned earlier, an automatic method for generating partition orderings has several benefits. For current and future multi-FPGA systems, it allows a large amount of flexibility and extendibility to be built into the system, since the software can cope with arbitrary topologies. Thus, small systems can be grouped into even larger machines, arbitrary chips and connections can be introduced, and the topology itself can be dynamically modified. Automatic mapping software can be generated that requires little user intervention, since the designer is not required to determine the location and ordering of cuts to be made. Also, failure recovery and incremental update capabilities can be included easily, since the software can partition to an arbitrary subset of the full system.

By combining the logic partition ordering algorithm from this chapter with the bipartitioning work from Chapter 10, a complete logic partitioning algorithm for multi-FPGA systems can be developed. It is fast and efficient, and automatically adapts to an arbitrary topology. In Chapter 12 we discuss a topology-adaptive algorithm for pin assignment, which handles part of the inter-FPGA routing problem for multi-FPGA systems.

Chapter 12. Pin Assignment

Introduction

A theme running throughout this thesis has been the need for the highest performance automatic mapping solution possible, especially one that can adapt to arbitrary topologies. In Chapter 10 we presented a fast and efficient algorithm for logic bipartitioning. Chapter 11 takes this algorithm and automatically applies it to arbitrary multi-FPGA system topologies. In this chapter we present a topology-adaptive pin assignment algorithm. By applying this algorithm, the overall mapping quality is improved, and the time it takes to place and route the FPGAs in the system is reduced. Current systems can require placement and routing times of seventeen hours or more on a uniprocessor system.

In this chapter we will examine the global routing stage of the automatic mapping process. It is important to remember during the following sections the context we are working in: we are mapping to a fixed structure of FPGAs, where wiring between the pins of the FPGAs are fixed, and while the logic has been partitioned to the individual FPGAs, these individual FPGA mappings have not yet been placed nor routed.

Global Routing For Multi-FPGA Systems

The global routing phase of mapping to multi-FPGA systems bears a lot of similarity to routing for individual FPGAs. Just as in single FPGAs, global routing needs to route on a fixed topology, with strictly limited resources, while trying both to accommodate high density mappings and minimize clock periods.

The obvious method for applying single-FPGA routing algorithms to multi-FPGA systems is to view the FPGAs as complex entities, explicitly modeling both internal routing resources and pins connected by individual external wires (Figure 104 left). A standard routing algorithm would then be used to determine both which intermediate FPGA to use for long distance routing (i.e., a signal from FPGA **A** to **D** would be assigned to use either FPGA **B** or **C**), as well as which individual FPGA pins to route through. Unfortunately, this approach will not work. The problem is that although the logic has already been assigned to FPGAs during partitioning, the placement of logic into individual logic blocks will not be done until the next step, FPGA placement. Thus, since there is no specific source or sink for the individual routes, standard routing algorithms cannot be applied.

The approach we will take here is to abstract entire FPGAs into single nodes in the routing graph, with the arcs between the nodes representing bundles of wires. This solves the unassigned source and sink problem mentioned above, since while the logic hasn't been placed into individual logic blocks, partitioning has assigned the logic to the FPGAs. It also simplifies the routing problem, since the graph is much simpler, and similar resources are grouped together (i.e., all wires connecting the same FPGAs are grouped together into a single edge in the graph). Unfortunately, the routing algorithm can no longer determine the individual FPGA pins a signal should use, since those details have been abstracted away. For example, in Figure 104 left, logic function **a** in FPGA **A** may best be placed in the lower right corner, and logic function **b** in FPGA **B**'s lower left corner. Thus, a connection between **a** and **b** would best be placed on one of the lower wires connecting these FPGAs. A global routing algorithm could not determine this fact. It is this problem, the assignment of interchip routing signals to FPGA I/O pins, that this chapter addresses.

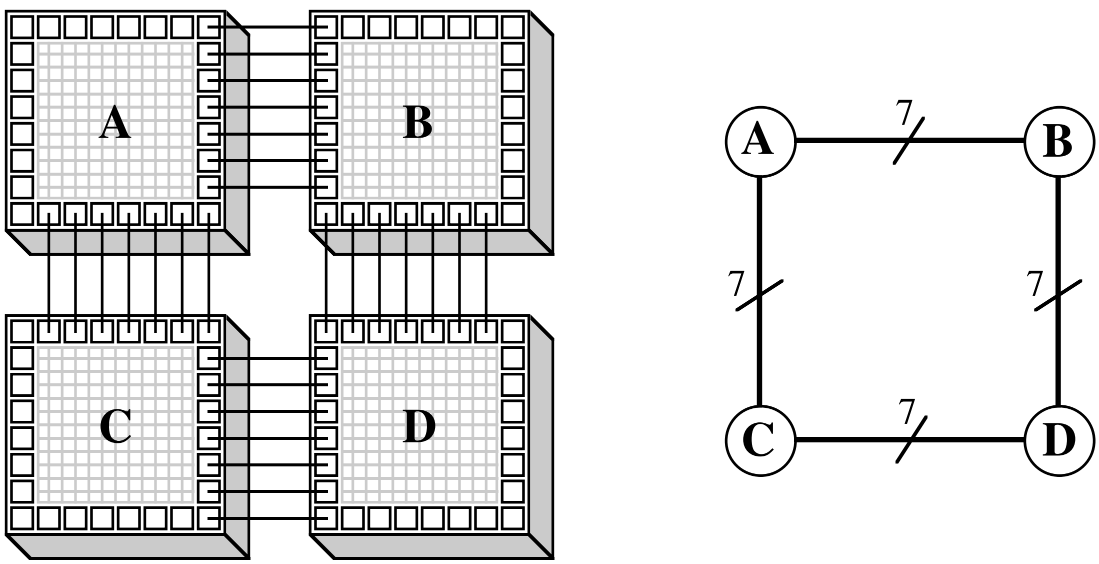


Figure 104. Two views of the inter-FPGA routing problem: as a complex graph including internal resources (left), and a more abstract graph with FPGAs as nodes (right).

Pin Assignment For Multi-FPGA Systems

Currently, the only work done on multi-FPGA pin assignment has been for the restricted case where no two logic-bearing FPGAs are directly connected (Chapter 9). For crossbars, hierarchical crossbars, and two-level topologies, where routing-only chips handle all connections between logic-bearing FPGAs, these algorithms works well. However, many systems map logic to most or all of their FPGAs (Chapter 5), and are beyond the scope of this algorithm. Also, as pointed out in Chapter 9, there is no way to adapt the pin assignment techniques from other technologies (such as ASIC routing channels, general routing for cell-based designs, and printed circuit board routing). This is because these algorithms optimize for features that are fixed in the multi-FPGA system domain, yet ignore the issues critical to multi-FPGA systems.

One solution to the pin assignment problem is quite simple: ignore it. After global routing has routed signals through intermediate FPGAs, those signals are then randomly assigned to individual pins. While this simple approach can quickly generate an assignment, it gives up some optimization opportunities. A poor pin assignment can not only result in greater delay and lower logic density, but can also slow down the placement and routing software, which must deal with a more complex mapping problem.

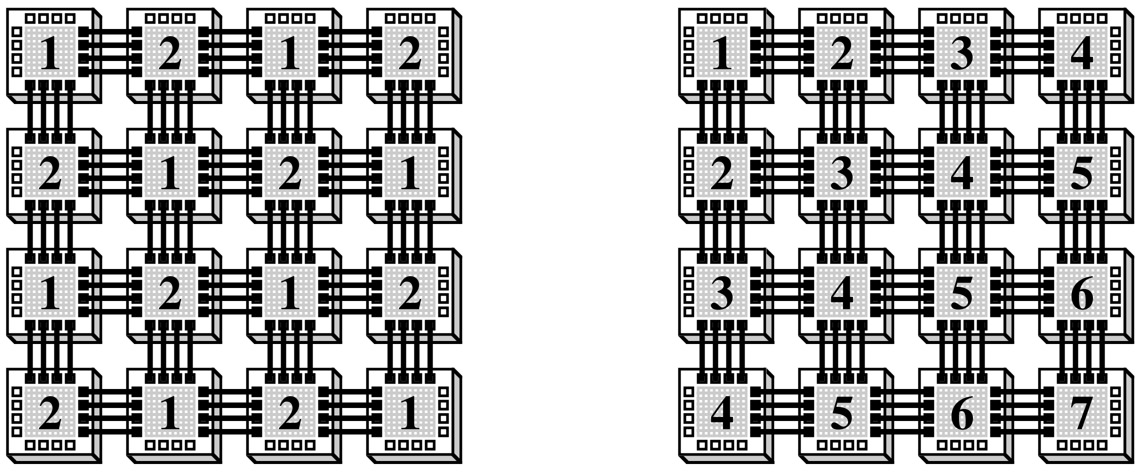


Figure 105. Checkerboard (left) and wavefront (right) pin assignment placement orders.

A second approach is to allow the FPGA placement tool to determine its own assignment. This requires that the placement tool allow the user to restrict the locations where an I/O pin can be assigned (a feature available in tools such as the Xilinx APR and PPR placement and routing tools [Xilinx94]). With such a system, I/O signals are restricted to only those pin locations that are wired to the proper destinations. Once the placement tool determines the pin assignment for one FPGA, this assignment is propagated to the attached FPGAs. It is important to note that this does limit the number of placement runs that can be performed in parallel. Specifically, since the assignment from one FPGA is propagated to adjacent FPGAs only after that entire FPGA has been placed, no two adjacent FPGAs can be placed simultaneously. Since the placement and routing steps can be the most time-consuming steps in the mapping process, achieving the greatest parallelism in this task can be critical. An algorithm for achieving the highest parallelism during placement, while allowing the placement tool to determine the pin assignment, is to find a minimum graph coloring of the FPGAs in the routing graph. Since the structure of a multi-FPGA system is usually predefined, the coloring can be precomputed, and thus the inefficiency of finding a graph coloring is not important (techniques introduced later in this chapter will be able to handle topologies without a predefined structure). Then, all the FPGAs assigned the same color can be placed at the same time, since any FPGAs that are adjacent cannot be assigned the same color. For example, in a four-way mesh (every FPGA is connected to the FPGA directly adjacent horizontally and vertically), the FPGAs could be placed in a checkerboard pattern, with half handled in the first iteration, and half in the second (Figure 105 left). Note that since the pin assignment will be determined during placement, the FPGA routing can be run in parallel with other placements or routings.

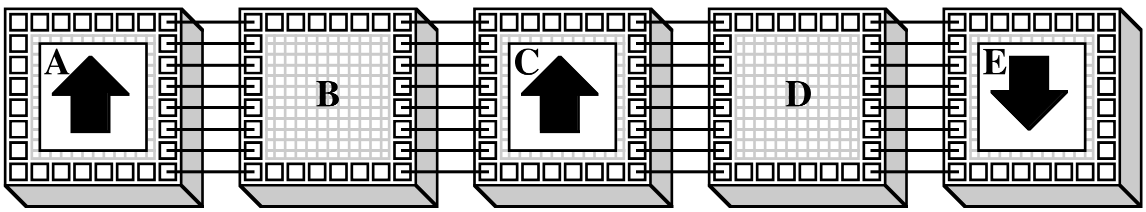


Figure 106. Example problem with the Checkerboard algorithm.

One problem with the maximum parallelism (or “checkerboard”) method just described is that while the pin assignment for the FPGAs placed first will be very good, since the placer is mostly free in its placement choices, other FPGAs may be placed fairly poorly. For example, consider the mapping of a systolic circuit to a linear array of FPGAs (see Figure 106). If allowed to map freely, the algorithm will map in a certain manner (the up arrows), or the mirror image around the horizontal (the down arrow). Since the individual FPGAs are mapped independently, some will choose one orientation, while others will choose the other. The problem is that there is a good chance that one of the unmapped FPGAs will be left with some neighbors in one orientation, and others in the other orientation (FPGA D). These FPGAs will have their mapping task complicated by the pin assignment imposed on them by their neighbors.

There is an alternative approach to the checkerboard algorithm, which trades longer mapping run times for better results. The idea is to make sure that FPGAs are not mapped completely independently of one another. In the first step, a single FPGA is placed. The FPGA that is most constrained by the system’s architecture (i.e., by special global signals or external interface connections) is normally chosen, since it should benefit most from avoiding extra pin constraints. In succeeding steps, neighbors of previously placed FPGAs are chosen to be placed next, with as many FPGAs placed as possible without simultaneously placing interconnected FPGAs. For example, in a 4-way mesh, where the first FPGA routed is in the upper-left corner, the mapping process would proceed in a diagonal wave from upper-left to lower-right (Figure 105 right). In this way, mappings “grow” from a single “seed” assignment, and will be more related to one another than in the checkerboard approach, hopefully easing the mapping tasks for all of the FPGAs.

Unfortunately, even this "wavefront" placement order may not generate good pin assignments. Most obviously, while the individual FPGA placements attempt to find local optimums, global concerns are largely ignored. For example, while pairs of signals that are used together in an FPGA will be placed together in the mapping, the fact that two of these pairs are used together in a neighbor will not be discovered, and these pairs may be assigned pin locations very distant from each other.

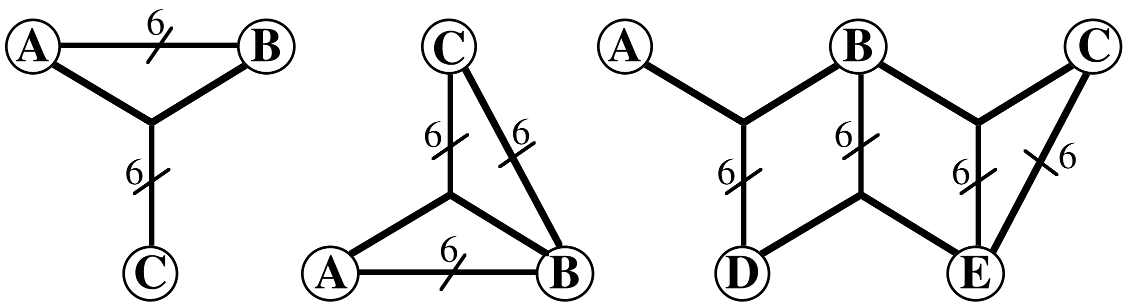


Figure 107. Examples of FPGA topologies that cannot be handled by sequential placement pin assignment techniques. Each of these topologies can be found as subcomponents in current systems, with all of these occurring in [Lewis93], and all but the topology at right in both [Thomae91, Vuillemin95].

Even worse than the speed and quality problems of both the checkerboard and wavefront methods (hereafter referred to jointly as "sequential placement methods") is the fact that there are some topologies that cannot be handled by these methods. Specifically, when the wires that connect FPGA pins are allowed to connect more than two FPGAs there is the potential for conflicts to arise, conflicts that can cause assignment failures with sequential placement methods. For example, consider the topology in Figure 107 left. Assume that the logic we wish to map to this topology has one connection between **A** and **C**, one between **B** and **C**, and six between **A** and **B** (Note that while this and other examples in this paragraph are specifically crafted to best show the underlying problems, each of these specific examples represent more general situations, and most wire and mapping connection counts can be changed without fixing these problems). Since all three FPGAs in this topology are directly connected, both the wavefront and checkerboard approach would place these FPGAs sequentially. If FPGA **A** is placed first, it could assign five of its connections to **B**, as well as its single connection to **C**, on the three-destination wires connecting **A**, **B**, and **C**. This means that there is no longer any way to route the connection between **B** and **C**, and the assignment fails. The same thing can happen when FPGA **B** is placed first. In the specific case of the topology at left, we could place FPGA C first, and avoid the conflicts described previously. Unfortunately, topologies such as the one in Figure 107 center have no order that is guaranteed to work. Assume that for this topology we wish to map a circuit with one connection between **A** and **C**, seven between **A** and **B**, and seven between **B** and **C**. In this situation at least one connection between each pair of FPGAs must be placed on a three-destination wire. However, regardless of the order, the first FPGA placed can use all of the three-destination wires for its own signals, causing the assignment to fail.

Instead of trying to find a placement order that will work, we could instead try to restrict the choices available to the placement tools. For example, in the mapping to the topology in Figure 107 center we know that each FPGA pair must have at least one connection assigned to the three-terminal wires. We could ensure that the placement tool generates a successful mapping by reserving wires for these signals. That is, if we placed FPGA **B** first, one of the three-destination wires would be reserved for a connection between **A** and **C** by disallowing the placement of signals from **B** onto the corresponding pin location. However, not only is this decision arbitrary, generating lower quality mappings since we have no method to pick a good wire to reserve, but it can also require significant effort to determine which and how many wires to reserve. For example, in the topology in Figure 107 right, a connection between **B** and **E** could be assigned either to a **BCE** or a **BDE** wire. If we do nothing to reserve space for it, connections between other FPGAs can fill these wires, making the assignment fail. We cannot reserve space on both wires, since all the capacity in the system may be required to successfully handle the mapping. However, to determine which wire to reserve requires the system to examine not only FPGA **B**'s and **E**'s connections, but also **D**'s, **C**'s, and **A**'s connections, since congestion on one wire can ripple through to the wires between **B** and **E**. Thus, determining the wires to reserve for one FPGA's connections can involve examining most or all FPGAs in the system. Note that while some of these topologies may seem contrived, topologies similar to the one in Figure 107 center are fairly common, and arise when a local interconnect is augmented with global connections, connections that include adjoining FPGAs.

Force-Directed Pin Assignment For Multi-FPGA Systems

As we have shown, pin assignment via sequential placement of individual FPGAs can be slow, cannot optimize globally, and may not work at all for some topologies. What is necessary is a more global approach which optimizes the entire mapping, while avoiding sequentializing the placement step. In the rest of this chapter, we will present one such approach and then give a quantitative comparison with the approaches presented earlier.

Intuitively, the best approach to pin assignment would be to place all FPGAs simultaneously, with the individual placement runs communicating with each other to balance the pin assignment demands of each FPGA. In this way a global optimum could be reached, and the mapping of all FPGAs would be completed as quickly as any single placement could be accomplished. Unfortunately, tools to do this do not exist, and even if they did, the communication necessary to perform this task would be prohibitive. Our approach is similar to simultaneous placement, but we will perform the assignment on a single machine within a single process. Obviously, with the placement of a single FPGA consuming considerable CPU time, complete placement of all FPGAs simultaneously on a single processor is impractical, and thus simplification of the problem is key to a workable solution.

Our approach is to use force-directed placement of the individual FPGAs [Shahookar91]. In force-directed placement, the signals that connect logic in a mapping are replaced by springs between the signal's source and each sink, and the placement process consists of seeking a minimum net force placement of the logic ("net force" indicates that forces in opposite directions cancel each other. Note that the spring force is calculated based on the Manhattan distance). To find this minimum net force configuration, and thus minimize wirelength in the resulting mapping, the software randomly chooses a logic block and moves it to its minimum net force location. This greedy process continues until a local optimum is found, at which point the software accepts the current configuration.

Force-directed placement may seem a poor choice for pin assignment, and is generally felt to be inferior to simulated annealing for FPGA placement. Two reasons for this are the difficulty force-directed placement has with optimizing for goals other than wirelength, and the inaccuracy of the spring approximation to routing costs. However, as will be shown, force-directed placement can handle all of the optimization tasks involved in pin assignment, and the spring metric is the key to efficient handling of multi-FPGA systems.

As implied earlier, we will not simply place individual FPGAs, but will in fact use force-directed placement simultaneously on all FPGAs in the system. We make two alterations to the basic force-directed algorithm: first, we do not restrict logic blocks to non-shared, discrete locations, but instead allow them to be freely placed into any location in the FPGA with no regard to congestion or physical logic block boundaries (however, I/O pins ARE constrained to exact pin locations, and I/Os cannot share a single pin location). Second, we assume that all logic blocks are always at their optimal positions. While the second alteration is simply a change in the movement order of nodes, the first change could cause a significant loss in accuracy. However, the resulting degradation of the local optimum will turn out to be more than made up for by the ability to seek the global optimum. Also, while this assumption allows logic blocks to share physical locations, something that cannot be done in valid FPGA mappings, our force-directed placement is used only to determine I/O pin assignments, and the logic blocks will be placed in a later step by standard FPGA placement tools.

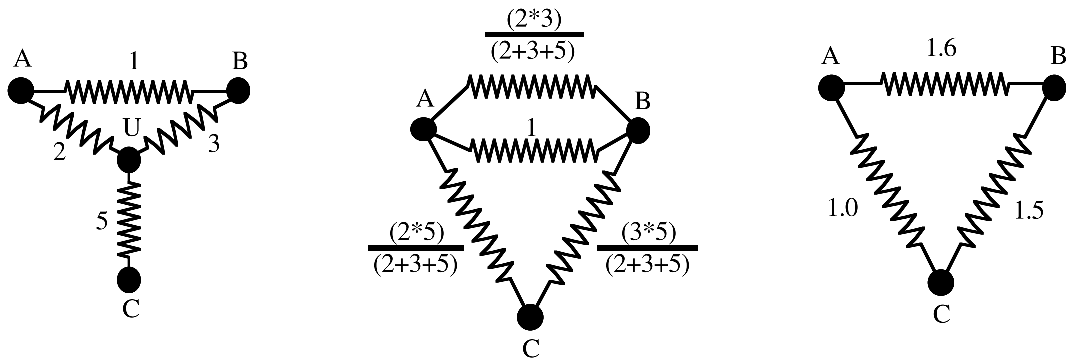


Figure 108. Example of spring simplification rules. Source circuit at left has logic node U replaced at center, and any springs created in parallel with others are merged at right.

By making the two assumptions just given, we now have the opportunity to greatly simplify the mapping process. We can examine the system of springs built for the circuit mapping, and use the laws of physics to remove nodes corresponding to logic functions, leaving only I/O pins. As shown at the end of this chapter, as well as in the example of Figure 108, the springs connected between an internal logic node and its neighbors can be replaced with a set of springs connected between the node's neighbors while maintaining the exact same forces on the remaining nodes. By repeatedly applying these simplification rules to the logic nodes in the system, we end up with a mapping consisting only of I/O pins, with spring connections that act identically to the complete mapping they replace. In this way, we simplify the problem enough to allow the pin assignment of a large system of FPGAs to be performed efficiently. Note that this spring replacement process can take a significant portion of the algorithm's run time, primarily because the replacement time is dependent upon the number of springs connected to the node being removed, and the removal of a node causes all of its neighbors to become connected. This effect can be mostly alleviated by removing nodes in increasing order of the number of springs connected to that node.

There are some details of the force-directed algorithm that need to be considered here. First, there are two ways to perform individual node moves in the algorithm. A simple approach is to pick a node and swap it with whichever node yields the least overall net force. A more complex approach is to perform a ripple move, where a node is moved to its optimum location, and any node occupying this location is then moved. This continues until the destination of the current node being moved is empty, at which point the move as a whole is evaluated. If it is a beneficial move, it is accepted; if not, it is rejected. Note that during this process the nodes already moved are flagged, and are not allowed to move again until a complete ripple move is over. While it is tempting to go with the simplicity of the swap move, there are some topologies for which swap moves yield poor results. Specifically, consider an FPGA with a total of 3 three-terminal wires (Figure 109), one to A and B, one to B and C, and one to A and C. Now consider a mapping onto this structure with a separate connection to each of the neighbor FPGAs. In this situation it is impossible to make a correct swap move, since no two connections have the same two wires as possible assignments.

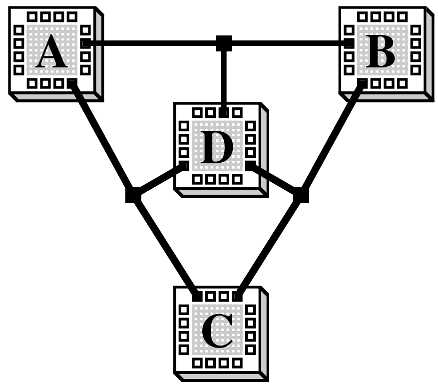


Figure 109. Topology for demonstrating the advantage of ripple moves instead of swap moves.

Another issue with the algorithm has to do with the starting position. While we would like to construct an initial, correct assignment for each wire, this can be very complex. As we indicated in the discussion of the topology in Figure 107 right, determining onto which wire a signal can be placed can require the examination of most or all connections in the system. An easier answer is to start with all wires unassigned, and always accept all valid ripple moves that start with an unassigned connection. In this way the mechanisms already required to perform moves of previously assigned pins can be easily extended to handle unassigned pins as well.

Efficiency in the calculation of the individual moves is another important problem. To determine the best possible destination of a node during a ripple move it is necessary to calculate the change in net force for assigning the connection to each wire that connects the FPGAs the signal moves between. This is necessary because there is not necessarily any relationship between the location of an I/O pin on one FPGA and the location of the pin the wire goes to on another FPGA (in fact, as described in Chapter 7, there is some advantage to be gained by scattering pin connections). Because of this, it is unlikely that there would be any method for pruning the search space of possible move destinations. This can be very troubling, especially because the node replacement method described above causes nodes to be connected to most or all other nodes in the same FPGA. However, as shown in Equation 4, the standard force equation (at left) can be reformulated so that the information from all springs connected to a node can be combined at the beginning of a move, and the calculation of the net force at each location requires only a small amount of computation. Another method for speeding up the process is to remember which connections have been starting points of ripple moves that were determined not to be beneficial, and thus not moved. Then, the algorithm will not try these moves again until another pin on one or more of that connection's FPGAs has been successfully moved.

$$\sum_{i=1}^{n} SpringConst_i \times \left(pos_i - pos_{node}\right) = \left(\sum_{i=1}^{n} SpringConst_i \times pos_i\right) - \left(\sum_{i=1}^{n} SpringConst_i\right) \times pos_{node}$$

Equation 4. Reformulation of the standard force equation.

A final extension necessary is software support to help avoid the problems found in the sequential approaches on some topologies. Just as the topologies in Figure 107 caused problems for the other approaches, they could cause difficulties with the force-directed approach. Imagine that we have the topology in Figure 107 left, and there are six connections between **A** and **C**, and one connection between **A** and **B**. At some point, all the three-destination wires will be occupied by connections between **A** and **C**. If we begin a ripple move with the connection between **A** and **B**, its best position might be on one of the three-destination wires. Unfortunately, while these wires must be occupied by the connections between **A** and **C**, and cannot accept the connection between **A** and **B**, this fact will not be determined until 5 of the 6 connections between **A** and **C** have been moved as part of this ripple move. At that point all of the wires will have connections assigned to them which have been flagged as part of the current ripple move, and the final connection between **A** and **C** will have no allowable assignment. While we can undo the ripple move, there is no way to tell the connection from **A** to **B** that it needs to try one of the two-terminal wires.

Our solution to the problems with the topologies in Figure 107 is to add the notion of equivalence classes to the system. That is, every wire in the system is a member of exactly one equivalence class, and each class contains only those wires that connect exactly the same FPGAs. For example, the topology in Figure 107 center would have three equivalence classes, one for the two-terminal wires between **A** and **B**, another for the two-terminal wires between **B** and **C**, and a final class for the three-terminal wires connecting **A**, **B**, and **C**. For each equivalence class we maintain a count of the number of wires within the equivalence class that have no connection assigned to them. Then, whenever a connection wishes to be assigned to a wire in an equivalence class, it can only do so if either the equivalence class has an unassigned wire, or if one of the (unflagged) connections already assigned to the class can be moved to another equivalence class (moving this connection to another class requires the same check to be performed on that class). Note that much of this information can be cached for greater efficiency, since if at one point in a ripple move an equivalence class cannot accept a connection, it cannot accept a connection at any future point during that move. Thus, in the previous example where it took several moves to determine that the connection between **A** and **B** shouldn't be moved onto a three-terminal wire in Figure 107 left, this fact would be immediately apparent from the equivalence classes. This would allow the search for possible moves to be pruned, and the connection would be forced to move to one of the two-terminal wires between **A** and **B**, the only legal assignments for that connection.

Table 16. Quantitative speed comparison table. Numbers in parentheses are normalized to the force-directed algorithm's results. Time is the time on multiple processors, while Uniproc. is the time on a single machine.

System	Splash		NTT		DECPeRLe-1	
Mapping	DNA Comparator		Telecom		Calorimeter	
Force - Time	34:25		8:45		29:14	
Uniproc.	12:05:12		19:31		2:43:18	
Wave - Time	5:35:40	(9.7530)	20:50	(2.3810)	2:11:33	(4.5000)
Uniproc.	13:24:58	(1.1100)	22:36	(1.1580)	3:10:02	(1.1637)
Checker - Time	1:07:00	(1.9467)	12:22	(1.4133)	1:51:24	(3.8107)
Uniproc.	12:37:36	(1.0447)	21:55	(1.1230)	2:55:24	(1.0741)
Rand - Time	1:00:18	(1.7521)	8:18	(0.9486)	30:47	(1.0530)
Uniproc.	17:45:56	(1.4698)	20:59	(1.0751)	2:56:44	(1.0823)

As mentioned earlier, our force-directed approach is capable of handling most of the optimization issues necessary for pin assignment. First, since little information about the individual FPGA architectures is necessary beyond the location of the I/O pins, it is very easy to port this software to different FPGAs. In fact, our current system allows different FPGA architectures and sizes to be intermingled within a single multi-FPGA system. Note that the locations defined in the software's chip descriptions need not be exactly the same as the real chip's geometry. In fact, our definitions of the Xilinx chips used in the comparisons below space pins at the corners closer together than others to reflect the lower expected congestion at these locations. We can also accommodate crossbar chips and FPICs, chips where minimizing wirelength is unnecessary, by setting the spring constant of all springs within such chips to zero. The assignments to pins going to the crossbar can also be removed after the pin assignment is complete. In this way, connections that require pin assignment can be assigned by our tool, while connections to the crossbar can be left unconstrained. More details on this can be found in Chapter 9.

Table 17. Continuation of quantitative speed comparison from Table 16. Numbers in parentheses are normalized to the force-directed algorithm's results. Time is the time on multiple processors, while Uniproc. is the time on a single machine.

System		Virtual Wires		Marc-1	
Mapping		Palindrome		Logic Sim. Processor	
Force -	Time	13:18		26:36	
	Uniproc.	1:44:37		3:38:11	
Wave -	Time	22:01	(1.6554)	Failure	
	Uniproc.	1:27:34	(0.8370)		
Checker -	Time	13:56	(1.0476)	Failure	
	Uniproc.	1:28:19	(0.8442)		
Rand -	Time	10:37	(0.7982)	26:38	(1.0013)
	Uniproc.	1:28:35	(0.8467)	4:36:32	(1.2674)

Table 18. Routing resource usage comparison table. Numbers in parentheses are normalized to the force-directed algorithm's results.

System	Splash	NTT	DECPeRLe-1	Virtual Wires	Marc-1
Force	143690.9	7292.2	45159.4	41255.4	102728.6
Wave	146424.8	7345.6	45661.2	41111.8	Failure
	(1.0190)	(1.0073)	(1.0111)	(0.9965)	
Checker	149860.5	7714.6	46522.5	41900.7	Failure
	(1.0429)	(1.0579)	(1.0302)	(1.0156)	
Random	156038.2	7854.8	48713.4	42225.3	108236.1
	(1.0859)	(1.0772)	(1.0787)	(1.0235)	(1.0536)

To optimize for delay, spring constants can be altered. Although our tool currently does not optimize critical paths, and assigns a spring constant of one to all non-crossbar springs, the tool could easily be extended to increase the spring constant on critical paths. This would cause critical path I/Os to be clustered closer together at the chip edge, improving performance.

Another important consideration is support for predefined pin assignments for some special connections. Specifically, special connections such as clocks, reset signals, fixed global communication lines, and host interfaces need to be assigned to specific pin locations. Handling these constraints in the pin assignment tool is trivial, since the support necessary to flag nodes that participate in the current ripple move can be extended to permanently freeze a connection in a specified position. Note also that any special chip features in an FPGA, such as the source of global distribution lines, can be handled similarly by defining a special pin location at the special feature site, and always preassigning the feature node to that point. For signals within FPGAs that are globally distributed via special resources, resources that make wirelength minimization irrelevant, the corresponding springs can simply be removed, or have their spring constants set to zero. Finally, there are other limited resources within an FPGA that might require special processing. For example, in Xilinx FPGAs [Xilinx94] there are a limited number of horizontal longlines which can be used as wide multiplexers or wired ANDs. To handle these features, the corresponding nodes in the circuit graph could be left unremoved, and force-directed placement could be applied to these elements as well (though these resources would of course require different potential positions than the pin locations). By doing this, a more accurate assignment could be performed without greatly affecting runtimes. Also, for the specific case of the Xilinx longlines, where the horizontal portion of the Manhattan distance is unimportant (the longlines stretch the width of the chip), these nodes could be specially marked, and the force calculation could be set up to properly ignore the horizontal component.

Comparison of Pin Assignment Approaches

To compare the various pin assignment approaches, we tested each approach on mappings for five different current systems. These include a systolic DNA comparison circuit for Splash [Lopresti91], a telecommunications circuit for the NTT board [Yamada94], a calorimeter circuit for DECPeRLe-1 [Vuillemin95], a palindrome circuit for the Virtual Wires Emulation system [Tessier94], and a RISC processor configured for logic simulation on the Marc-1 system [Lewis93]. The pin assignment systems compared are: “random”, which randomly assigns connections to wires; “checkerboard” and “wavefront”, which use sequential placement runs to do pin assignment; and “force”, which uses the force-directed pin assignment technique described above. The results include both mapping time and resulting wirelength (Table 16, Table 17, and Table 18). The time is CPU time on a SPARC-10, and includes the time to perform pin assignment as well as individual FPGA place and routes, and assumes enough machines are available to achieve maximum parallelism. Uniprocessor time, the time it takes to complete all of these tasks on a single machine, is also included. Note that the sequential placement techniques only sequentialize the placement step, while routing is allowed to be performed in parallel with subsequent placements and routings. Also, these placement attempts are begun as soon as possible, so that for example in the checkerboard algorithm, we do not need to wait to begin placing FPGAs of one color until all FPGAs of the previous color are placed, but instead can proceed with a placement once all of its direct neighbors in the previous color(s) have been completed. Also, the random approach is assumed to be able to generate a pin assignment in zero time, so its time value is simply the longest time to place and route any one of the individual FPGAs. The wirelength is determined by summing up all source to sink delays for all signal in the FPGAs. While this is not exact, since some portions of a multi-destination route will be included more than once, it gives a reasonable approximation of the routing resource usage of the different systems.

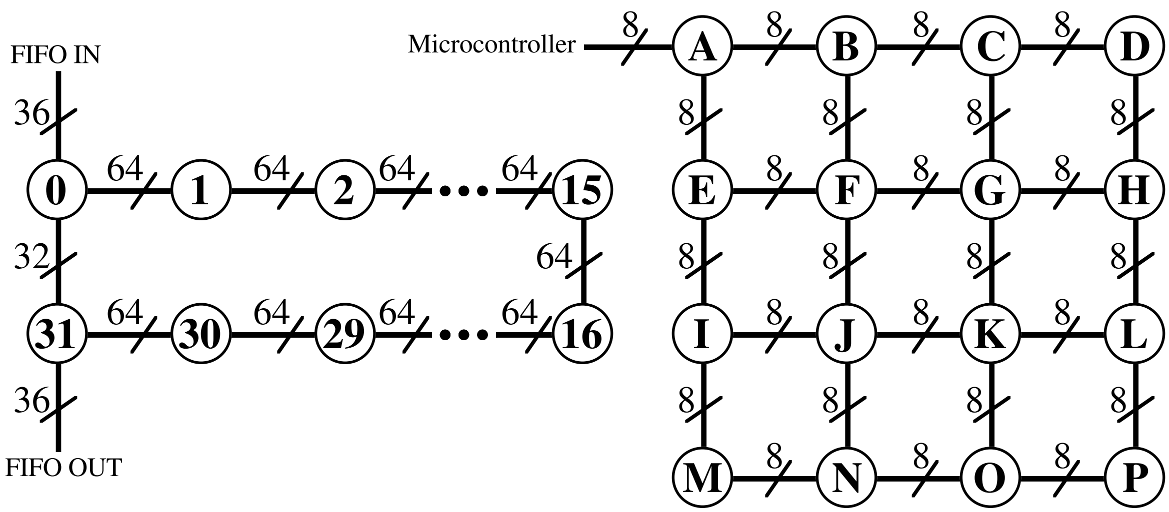


Figure 110. The Splash (left) and Virtual Wires (right) topologies. Note that system features not used in the example mappings, as well as global communication wires and clocks, are omitted.

The Splash system [Gokhale90] is a linear array of Xilinx 3090 FPGAs (Figure 110 left), with memories attached to some of the communication wires. As expected, Random does poorly on the average wirelength, with Checkerboard and Wavefront each doing successively better. However, the Force-Directed approach does the best, since it is able to optimize the mapping globally. More surprising is the fact that the Force-Directed approach is actually faster than all the other approaches, including the Random approach, in both parallel and uniprocessor times. The reason for this is that the poor pin assignment generated by the random approach significantly increases the routing time for some of its FPGAs, up to a factor of more than 4 times as long. Note that the time to perform the force-directed pin assignment does not dominate for either this or any other mapping. For the DNA mapping it is only 9% of the overall runtime, for Palindrome and Telecom it is 6%, and for Calorimeter it is 8%, though for the Marc-1 mapping it is 31%.

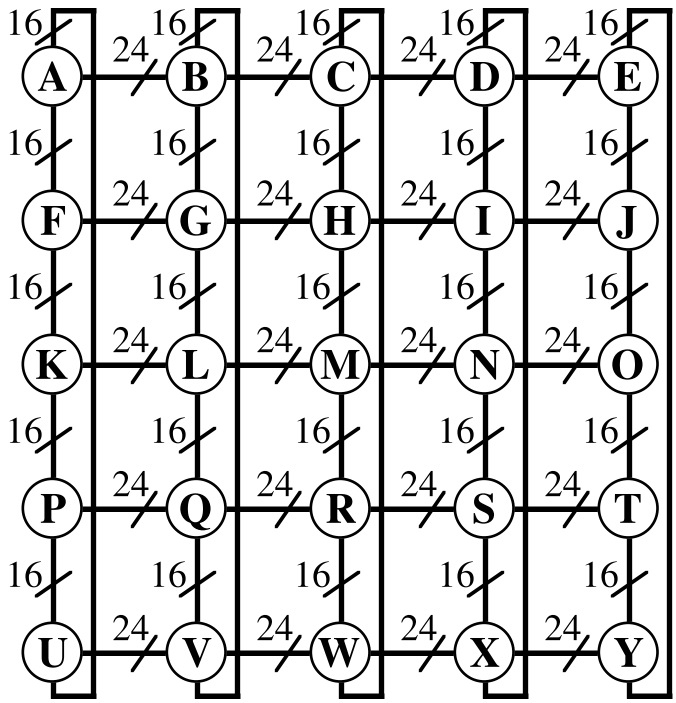


Figure 111. The NTT topology.

The results are almost identical for the DECPeRLe-1 board [Vuillemin95], a four by four mesh of Xilinx 3090 FPGAs with seven additional support FPGAs, and the NTT board [Yamada94], a five by five mesh of 3042 Xilinx FPGAs with wrap-around connections on the top and bottom edges (Figure 111). However, the Checkerboard approach takes much longer on the DECPeRLe-1 board than it did on the Splash board, making it only about 15% faster than the Wavefront approach. The reason is that the DECPeRLe-1 board, which has 23 FPGAs, is only 19-colorable, leaving very little parallelism for the Checkerboard algorithm to exploit.

The Virtual Wires board [Tessier94] is a 4-way mesh of Xilinx 4005 FPGAs, with the connections in a given direction scattered around the FPGA's edge (Figure 110 right). The results for this mapping are less favorable than for the DNA comparator and the calorimeter, with the Wavefront method actually generating a slightly better assignment than the Force-Directed approach. However, we believe the reason for this poor performance is the special techniques applied to the mapping prior to pin assignment. The Virtual Wires system [Babb93, Selvidge95] attempts to overcome FPGA I/O limitations by time-division multiplexing the chip outputs. Thus, each chip I/O is connected to a shift chain, which is connected to the actual I/O signals of the mapping. Unfortunately, the choice of which wires to combine is done arbitrarily, without regard to the mapping structure. Thus, instead of combining signals that are close together in the logic, signals from very distant portions of an FPGA's mapping may be merged. Because of this, there is very little structure left in the mapping, and there is little pin assignment can do to improve things. We believe that a more intelligent combining strategy, possibly driven by the spring constants generated by our spring simplification approach, would generate better mappings, mappings that can also be helped by good pin assignments.

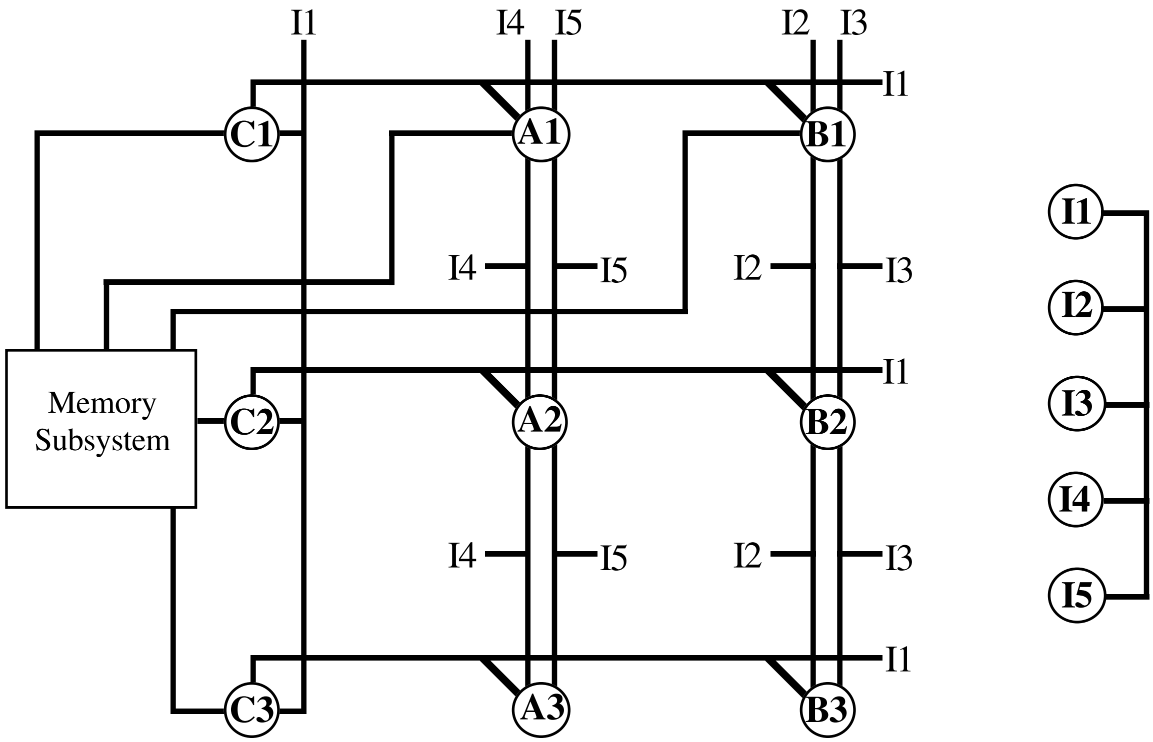


Figure 112. Schematic of the Marc-1 board. The entire board consists of two copies of the subsystem at left, and one copy of the subsystem at right. Labels without circles indicate connections to the corresponding FPGA circle. Note that system features not used in the example mappings, as well as global communication wires and clocks, are omitted.

The Marc-1 board [Lewis93] is a complex system of Xilinx 4005 FPGAs, as shown in Figure 112. As mentioned earlier, this board includes all topologies from Figure 107, topologies that make mapping failures possible in sequential placement approaches regardless of placement order. We attempted to use these sequential approaches four times each on this system, but each time it failed because of resource conflicts (note that even had they succeeded, the topology is only 5-colorable, and the Wavefront method would sequentialize 9 placement runs, which in both cases would greatly increase runtimes). However, our force-directed approach easily handled this topology, and generated a good assignment. Note that while the force-directed result is somewhat less of an improvement than the other examples led us to expect, this may be due to the fact that only 51% of the logic pins are assignable, with the others fixed by memory interfaces and other fixed global signals.

Table 19. Unconstrained pin assignment distances. Numbers in parentheses are normalized to the force-directed algorithm's results. The NTT results show the "optimal" value is worse than what we achieve. This is due to the random nature of the placement process.

Splash	138899.9	(0.9667)
NTT	7380.7	(1.0121)
DECPeRLe-1	37373.3	(0.8276)
Virtual Wires	39789.1	(0.9645)
Marc-1	91267.4	(0.8884)

While the previous comparisons demonstrate the advantages of the force-directed approach over the simple approaches described earlier, it would be useful to find out how close to optimal the force-directed approach is. In general, we can expect that the best a pin assignment can do is to equal the placement qualities the placement and routing tools can achieve without fixed pin assignments. That is, if we place and route the individual FPGAs and only require that logic pins be assigned to chip pins going to the correct neighbor (though we do not require that connected logic pins on neighboring FPGAs match up, that they occupy connected FPGA pins), this is probably as good as any pin assignment can hope to achieve, and probably better than is achievable. We did exactly this experiment, and the results are in Table 19. As can be seen, our pin assignments achieve results within 3.5% of "optimal" on half of the examples, with the geometric mean of the lower bound results being 93% of the force-directed results. In one case our results are even better than the lower bound results; however, we believe this is due mainly to the random nature of the placement process. It is unclear exactly how much can be inferred from these results, especially since it is not clear any approach can achieve results as good as those shown in Table 19. One important observation to note is that most likely not much improvement (at least in routing resource usage) can be achieved beyond what our algorithm provides.

Pseudocode and Computational Complexity

In this section we present pseudocode for the force-directed pin assignment algorithm, as well as a discussion of the computational complexity. The overall algorithm is presented in Figure 113.

```
read in topology and circuit description;
                     // Apply spring reduction rules
foreach FPGA F {
   foreach logic node L in F, from low to high connectivity {
      T = total spring constants of springs incident to L;
      foreach pair of L's neighbors N and M {
         spring(N, M) = spring(N, M)
            + (spring(L, N)*spring(L, M))/T;
      }
      foreach of L's neighbors N
         remove spring(N, L);
   }
}

create equivalence classes;
while moves possible {
   randomly pick IO signal S;
   perform ripple move for signal S;
   if S was originally unassigned, or ripple move is beneficial
      accept ripple move;
   else
      undo ripple move;
}
```

Figure 113. The force-directed pin assignment algorithm.

```
S = IO signal starting ripple move;
Compute constants from Equation 4 for each FPGA connected to S;
foreach wire W connecting between S's FPGAs {
   if equivalence classes allow S to be assigned to W {
      compute net force of S assigned to W;
   }
}
pick W with lowest net force;
temporarily assign S to W;
alter equivalence class so that S cannot be moved again in ripple;
if W is unoccupied {
   return change in force of assigning S to W;
} else {
   perform ripple move for signal assigned to W;
   return change in force of assigning S to W plus rest of ripple;
}
```

Figure 114. Algorithm for applying ripple moves.

Unfortunately, it is difficult to quantify the computational complexity of the force-directed pin assignment algorithm. The primary problem is bounding the number of iterations for the algorithm to find a minimum net-force state. It is possible to believe that degenerate situations exist where the system will take a significant number of moves to find a local minimum. Also, it is possible for a single ripple-move to move every I/O node in the system. Specifically, imagine a system that only has three-terminal wires, and a mapping with two-terminal connections. A connection between FPGAs A and B might move to a three-terminal wire between A, B, and C. This will displace a connection from B to C, which goes to a three-terminal wire between B, C, and D, displacing a connection from C to D. Thus, the worst-case complexity is significant. Also, the spring reduction process can be $O(N^3)$ in the worst-case. Specifically, imagine a mapping to an FPGA where every logic and I/O node is connected to a single signal (such as a reset or clock signal that does not use the FPGA's global distribution resources). When the first node is replaced, all pairs of nodes in the FPGA will become connected. Thus, every subsequent step in the process will have to modify a spring between each pair of nodes in the FPGA, resulting in $O(N^3)$ performance. Note that it is unlikely that such a situation will occur, since most huge-fanout nets will be handled by global routing resources, and thus will be removed from consideration by the spring reduction algorithm. For real circuits, the runtimes should be more on the order of linear or quadratic because of the sparse, localized connection patterns. Thus, while the worst-case complexity of this algorithm is quite large, and hard to bound, the program runs efficiently on current systems, and should scale reasonably to future machines.

Conclusions

As we have shown in this chapter, pin assignment for multi-FPGA systems is a difficult problem that has not been previously studied. We presented some approaches for using existing tools, as well as a new force-directed algorithm that can help improve mapping quality. Placement and routing with the force-directed approach is almost always faster than the random approach, by up to 43%, and almost always delivers superior routing resource usage, by up to an 8% decrease in total resource usage in the entire system. We have also shown that this is within at least 7% of optimal. Given these results, it is clear that pin assignment improves the mapping process for multi-FPGA systems.

Several possible extensions have been mentioned earlier for improving mapping quality, and these are worth further study. More importantly, the spring cost metric is not necessarily tied to force-directed placement, but could instead be used within a simulated annealing algorithm. While we believe simulated annealing would not yield significant improvements, since the cost metric is so abstracted from the real situation, and since simulated annealing is so time-consuming, a study of its actual benefits should be performed. More interesting is the approach mentioned above for improving Virtual Wires mappings by guiding wire combining by the spring forces our system generates. We hope that such an approach would not only improve the overall placements achieved by Virtual Wires, but also show the benefits of good pin assignments in this domain.

Spring Replacement Rules

As discussed earlier, we wish to reduce the complexity of the force-directed placement algorithm by replacing all springs touching non-I/O nodes with equivalent springs only involving I/O nodes. To do this, we iteratively remove individual internal nodes from the system, and replace the attached springs with new springs between the removed node's neighbors.

Since we are using a Manhattan distance metric, the forces along the X and Y dimension are independent, and are given in Equation 5.

$$F_x = C \times (X_1 - X_2) \quad F_y = C \times (Y_1 - Y_2)$$

Equation 5. Spring forces in the X and Y direction.

C is the spring constant, and (X_1, Y_1) and (X_2, Y_2) are the locations of the nodes connected by the spring. There are two simplification rules necessary for our purposes: parallel spring combining, and node removal. For parallel springs, springs which connect the same two endpoints, the springs can be merged into a single spring whose spring constant is the sum of the parallel springs.

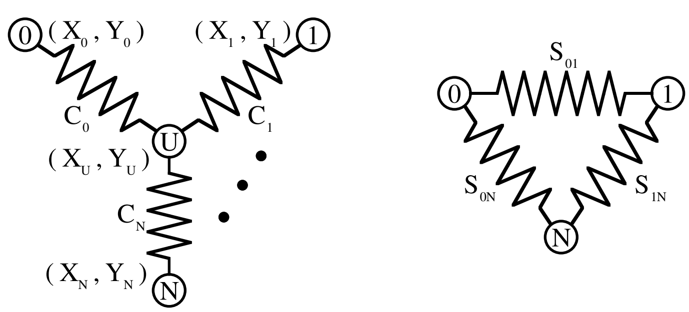


Figure 115. Spring system before node **U** is removed (left), and after (right).

In node removal, we remove an internal node, and replace the springs connected to that node with springs connected between the node's neighbors. As shown in Figure 115, we have node **U** with $N+1$ neighbors labeled **0**…**N** (note that if **U** has only one neighbor, the node and the spring can simply be removed, since it will never exert force on its neighbor). Node i is located at (X_i, Y_i), and is connected to node **U** by a spring with spring constant C_i. As discussed earlier, we assume that internal nodes are always placed at their optimal location, which is the point where the net force on the node is zero. Thus, we can calculate the location of node **U** as shown in Equation 6 and Equation 7 (note that from now on we will work with only the X coordinates of all nodes. Similar derivations can be found for the Y coordinates as well).

$$0=\sum_{i=0}^{n} C_i \times \left(X_i - X_U\right) = \left(\sum_{i=0}^{n} C_i \times X_i\right) - X_U \times \sum_{j=0}^{n} C_j$$

Equation 6. Total forces on node **U**, assuming **U** is placed at its optimal location.

$$X_U = \frac{\sum_{i=0}^{n} C_i \times X_i}{\sum_{j=0}^{n} C_j}$$

Equation 7. Optimal X position of node **U** expressed in terms of its neighbors' positions.

To replace the springs connected to node **U**, we must make sure the new springs provide the same force as the old springs. So, we start with the force equation from Equation 5, and substitute in the location found in Equation 7. The results are Equation 8-Equation 11.

$$F_k = C_k \times (X_U - X_k) = \frac{C_k \times \left(\sum_{i=0}^{n} C_i \times X_i\right)}{\sum_{j=0}^{n} C_j} - C_k \times X_k$$

Equation 8. Substitution of Equation 7 into Equation 5.

$$F_k = \frac{C_k \times \left(\sum_{i=0}^{n} C_i \times X_i\right) - C_k \times X_k \times \sum_{l=0}^{n} C_l}{\sum_{j=0}^{n} C_j}$$

Equation 9. Simplification of Equation 8.

$$F_k = \frac{\left(\sum_{i=0}^{n} C_k \times C_i \times X_i\right) - \left(\sum_{l=0}^{n} C_k \times C_l \times X_k\right)}{\sum_{j=0}^{n} C_j}$$

Equation 10. Simplification of Equation 9.

$$F_k = \sum_{i=0}^{n} \left(\frac{C_k \times C_i}{\sum_{j=0}^{n} C_j} \times \left(X_i - X_k \right) \right)$$

Equation 11. Simplification of Equation 10.

From Equation 11 it is now clear how to replace the springs incident on node **U**. We can replace all of these springs, and insert a new spring between each pair of neighbors of node **U**. The new spring between nodes *I* and *K* will have a spring constant S_{ik} as given in Equation 12.

$$S_{jk} = \frac{C_k \times C_i}{\sum_{j=0}^{n} C_j}$$

Equation 12. Spring constants for spring replacement, derived from Equation 11.

Chapter 13. Conclusions and Future Work

Multi-FPGA systems is a growing area of research. They offer the potential to deliver high performance solutions to general computing tasks, especially for the emulation of digital logic. However, to realize this potential requires a flexible, powerful hardware substrate and a complete, high quality and high performance automatic mapping system.

In the last five to ten years there have been a large number of multi-FPGA systems built and proposed. Work has also been done on creating software algorithms to support these machines. However, in general these hardware and software systems have been created in an ad hoc fashion. The hardware systems are grab-bags of appealing features with little investigation into what is truly best for the applications they will support. For software, individual algorithms have been proposed, but with little insight into which approaches are the most important, and how these pieces can be combined into a complete solution.

The primary goal of this thesis has been to offer a disciplined look at the issues and requirements of multi-FPGA systems. This includes an in-depth study of some of the hardware and software issues of multi-FPGA systems, especially logic partitioning and mesh routing topologies, as well as investigations into problems that have largely been ignored previously, including pin assignment and architectural support for logic emulator interfaces. This work points out the challenges of multi-FPGA systems, and presents solutions to many of them.

In Chapter 6 we presented Springbok, a novel rapid-prototyping system for board-level designs that offers many advantages over current systems. Its flexible architecture accommodates a great range of system sizes and topologies. With the ability to solve problems as they occur, Springbok uses its resources more efficiently than do fixed FPGA-based systems, which require a very conservative style. Including arbitrary devices and subsystems into the Springbok structure allows even greater efficiency and accuracy. Finally, the use of FPGAs instead of FPICs for the routing structure reduces overall costs, adds flexibility, and more easily handles the functionality necessary to interface to timing-inflexible components.

To design Springbok, as well as many other multi-FPGA systems, requires a hard look at the hardware structures used to build them. Chapter 7 presented several techniques for decreasing routing costs in mesh interconnection schemes: 1-hop interconnections, Superpins, and Permutations. These topologies reduce I/O and internal routing resource usage, increase bandwidth, and reduce delays in the system.

Including an emulation into its target environment is one of the most important benefits of emulation over simulation. However, with current systems the user is required to develop an application-specific protocol transducer to allow the emulation to communicate with its environment. In Chapter 8 we demonstrated that many logic emulator interfaces can be handled by a simple, generic interface transducer board. While the protocol transducer mappings can be somewhat involved, we have shown that they can be quickly developed in a high-level language, and automatically translated into FPGA logic. Thus, with a small amount of design effort, an emulation can be run in its target environment, greatly increasing its utility.

To be successful and widely used, a multi-FPGA system needs more than just a flexible hardware system. Also necessary is automatic mapping software that maps circuits onto this structure. This software needs to be fast, fully automatic, and capable of producing high-quality results. Chapter 10 presented a survey of bipartitioning techniques, mainly those based on the Kernighan-Lin, Fiduccia-Mattheyses algorithm. We have discussed many optimizations to the algorithm, including several new approaches, which have yielded a fast and efficient bipartitioning algorithm that is significantly better than the current state-of-the-art.

Chapter 11 presented an algorithm for automatically applying bipartitioning to an arbitrary multi-FPGA system. It analyzes the multi-FPGA system topology, and recursively applies bipartitioning, as well as multi-way partitioning and multi-sectioning where appropriate, as part of a fast and automatic method for mapping to a multi-FPGA system.

Pin assignment was covered in Chapter 12. Our algorithm adapts to an arbitrary multi-FPGA system topology. It creates better routing than current systems, which results in faster mapping runtimes, and higher quality mappings.

By combining the algorithms discussed in this thesis with those commercially available, a complete automatic-mapping system can be developed. It offers fast runtimes, with good quality results, and adapts to an arbitrary topology. In this way, a single system can be applied to the numerous different topologies proposed, including topologies with flexible interconnection patterns.

This thesis has provided an in-depth investigation into most of the issues in multi-FPGA hardware and software. As a result, the hardware and software structures of multi-FPGA systems have been optimized, yielding faster and higher-quality systems. Bringing these advantages to bear on board-level prototyping, such as in the Springbok system, gives logic designers a powerful new tool for system-level debugging.

One of the most important aspects of the work contained in this thesis is that the approaches given here are applicable not just to Springbok, but also to multi-FPGA systems in general. The routing topologies can improve many multi-FPGA systems, and the generic interface support is useful for existing logic emulators. The software algorithms provide architecture-adaptive tools to optimize to arbitrary topologies, delivering improved quality, in much less time, than current mapping software.

While the techniques discussed can help many different systems, there are other opportunities to improve multi-FPGA systems, and to achieve the full promise of reconfigurable logic. There are many outstanding issues in both the hardware and software constructs of FPGA-based system.

While this and other works have focused on some portions of the mapping software flow, there is still much yet to be done. One example is the partitioning of digital logic onto multi-FPGA systems. This thesis has included a survey of many bipartitioning techniques, resulting in significant quality and mapping speed improvements over current techniques. However, in many cases multi-way partitioning algorithms make more sense than bipartitioning techniques. Thus, a similar study investigating the many multi-way partitioning algorithms, which integrates the techniques discussed in this thesis, has the potential to provide even better results. Also, while iterative bipartitioning can map to an arbitrary topology, its greedy nature may generate substandard results. Alternatives are possible, including extensions of quadrisection and other techniques that simultaneously perform multiple cuts. However, these techniques must be modified to handle the different possible topologies found in multi-FPGA systems.

Another important issue in the mapping tools for multi-FPGA systems is the time the software tools take to create a mapping. With current software taking up to a full day to complete a mapping, there are many applications that do not take advantage of multi-FPGA systems because of the mapping delays. There are several ways to combat this issue. The fast partitioning and pin assignment tools contained in this thesis are one such step. Another possibility is to examine the other parts of the mapping software and develop new algorithms optimized for this domain. Specifically, current systems use technology-mapping, placement, and routing software originally created for single-chip systems. However, single-chip systems have much different constraints than multi-FPGA systems. The FPGAs in a current multi-FPGA system can expect to have only a small fraction of their logic capacity used, but all of their I/O pins will be consumed. Tools that understand this domain may be able to perform their optimizations much faster than tools optimized for single-chip mapping situations.

Another possibility for improving the performance of multi-FPGA system mapping tools is to adopt and extend the two-phase methodology proposed in this thesis's partitioning chapter. That is, it is not always necessary to achieve the best possible mapping, and a lower quality, but faster, mapping algorithm may be a better choice. Cases where more than ample resources are available, and optimal performance is not critical, are obvious situations for applying poor but fast software. However, even when the highest performance is required, only using high-quality but slow software may not be the best answer. Specifically, when using the best possible mapping software the initial mapping steps (such as partitioning) will usually need to be run on a single processor, and will take a long time. Once these steps are done, multiple independent place and route processes can be run on different machines, speeding up the mapping process. However, there is no reason that these processors must remain idle in the early mapping phases. One possibility is to run the fast, but lower quality, mapping tools concurrently with the high quality, but slow, algorithms. These faster algorithms could quickly generate a mapping, and have the emulation system running the mapping much sooner than the other approach. While the emulation will run more slowly than the higher quality mapping produced by the slower software, the emulator will be making headway as the slower software completes. Once the improved mapping is completed, it is relatively simple to pause the emulator, save the current state of the emulation, and load the improved mapping. By properly setting the stateholding elements in the new mapping, the high-quality mapping can resume where the slower mapping ended. In this way, the emulator can be used more efficiently, with shorter down times, and the user gets answers much more quickly than current systems can achieve. However, there is much research to be done on determining how to quickly create these lower-quality mappings.

One can also speed up the software by minimizing the amount of remapping necessary. Specifically, if a minor upgrade or bug fix is made to an input circuit, it should be possible to reuse much of the previous mapping. Since much of the circuit remains the same, a fast way to create a new mapping is to just remap that portion of the circuit that has changed. While our architecture-adaptive partitioning and routing tools are capable of working on arbitrary subsets of a multi-FPGA system, there are several other pieces necessary to achieve a true incremental update facility. Specifically, the synthesis tools that generate the mapping need to modify only those portions that have changed, or else the modification will alter the entire mapping, and the incremental facilities of subsequent tools will be useless. Also, technology-mapping, placement, and routing tools can also be given an incremental-update capability, yielding even greater performance increases.

While the ability of our algorithms to adapt to an arbitrary topology can be important as part of an incremental update mechanism, this adaptability also opens up another opportunity. Since the tools can optimize for any architecture, the tool suite becomes an ideal system for performing architecture exploration. In this thesis, we examined mesh routing topologies based on totally random routing requirements. While this made it possible to get some idea of the tradeoffs in mesh architectures, it also limited the study's scope. Most obviously, real circuits are not completely random, and although our study took into account some of the non-randomness of circuits (such as buses), better results could be achieved by using actual circuits in the study. Thus, a toolset that can automatically adapt to an arbitrary topology is the key to better understanding how multi-FPGA systems should be constructed. This would not only yield comparisons of different optimizations to a given type of topology (i.e., comparing meshes to meshes, and crossbars to crossbars), but also allow comparisons between different types of topologies (i.e., meshes to crossbars). In this way, we can continue the search for better architectures, resulting in higher quality systems.

There are many other opportunities for work in multi-FPGA systems and reconfigurable hardware. One possibility is to push forward with the deployment of systems such as Springbok. Although Springbok holds a lot of promise for the prototyping of board-level designs, there is still much work necessary to bring it to fruition. Detailed design of the hardware, especially in the connector technology to enable Springbok's flexibility, as well as communication and debugging support to help the user understand the operation of their prototype, is key to a complete implementation. The software tools will also need to be extended. Most obviously, there is currently no algorithm for assigning daughter cards to specific locations to achieve the highest-quality mapping. That is, while the input specification for a circuit may require a specific mix of daughter cards, there still needs to be an algorithm that decides where on the baseplate these daughter cards should be placed to minimize communication and improve performance. While simulated annealing should be able to handle this task, this would be a very slow answer to the problem. It is likely that alternative placement algorithms, such as force-directed placement, or possibly combining daughter card placement with the partitioning algorithm (assigning daughter cards to different partitions as well as logic nodes), could produce nearly equal quality results in much less time. There is also the issue of how and when to insert extender cards into the array. As each of the software mapping algorithms run, they may encounter resource limitations that cause significant decreases in mapping quality, or even cause the mapping to fail. A way to solve these problems in the Springbok system is to add extender cards with extra resources to ease these bottlenecks. However, how best to integrate this feature into current algorithms is an open question.

Looking beyond Springbok, there are many other opportunities in reconfigurable hardware. One of the most promising is combining together reconfigurable logic and standard processors. While we have discussed many applications where FPGAs can provide the highest performance implementation, most situations fit better onto standard processors. In current systems, there is usually a strict separation of the two resources. If a problem fits well onto FPGAs, it is mapped onto an array of FPGAs. If not, it must be run on a separate, standard workstation. This separation ignores the numerous applications where a mixed approach might be better. Normally, an algorithm goes through a series of phases. Often this might involve some complex initialization protocol, a regular inner loop that consumes most of the computation time, and another complex final cleanup step. The complexity of the starting and ending phases require the flexibility of standard processors, while the inner loop could be best implemented in FPGA logic

The answer to this problem is to integrate together the two types of resources. One way is to construct a mixed FPGA-microprocessor system. By combining several off-the-shelf FPGAs and processors, a single system can provide efficient mappings for many different types of algorithms. The processors in the system provide complex control flow for external interaction, complex mathematical computations, and irregular initialization and cleanup. The FPGAs provide huge amounts of fine-grain parallelism for fast bit-manipulations, large data-parallel operations, and custom datapaths. Thus, merging the two resources together could achieve much greater performance than either type of resource separately can provide.

A more exciting possibility is to add reconfigurable logic into the processors themselves. Specifically, for many applications there is a small set of operations that would greatly improve their performance, but which are not useful widely enough to be included in a general-purpose processor. By including reconfigurable logic into the processors, these instructions can be built on a per-application basis, with different mappings to the reconfigurable logic for different programs (or even different sections of a single program). Thus, we would have a general-purpose method of providing special-purpose processors. This logic may not look anything like current FPGA logic. The reconfigurable logic will need to be optimized to this application domain, opening up a very interesting area for future architectural investigations. Also, the processors themselves may need to be modified to accept this new type of resource. However, the most critical area of research for these types of systems is in determining how compilers can be made to recognize and optimize for the opportunities provided by this added logic. Good compiler support for this application is critical to achieving widespread benefit from the reconfigurable logic.

This logic need not be solely a method of adding extra instructions to the processor's instruction-set. A different model for using these resources is FPGAs used as a generic coprocessor system, where the reconfigurable logic is viewed more as a separate processing element, perhaps with a different control stream, than as just another unit in the processor. In current machines, coprocessors are typically provided for floating-point arithmetic, graphics acceleration, and possibly data encryption. These coprocessors are used not only because they can speed up these applications, but also because these applications are perceived to be important enough to merit special-purpose hardware. Many other algorithms could benefit from special-purpose coprocessors, but are used so infrequently that providing this custom hardware is impractical. A solution to this problem is to add a generic, reconfigurable coprocessor to the system. While it may not be able to achieve the same performance as a custom chip, a reprogrammable coprocessor can economically provide improved performance to many different algorithms. The coprocessor will be shared across many different algorithms, since each algorithm can have its own configuration. Thus, although each algorithm individually is not of wide enough utility to merit a special-purpose coprocessor, by sharing the same generic coprocessor with many different algorithms the hardware becomes practical.

Such processor-FPGA hybrids, even if they prove practical, are still many years away. In that time there are several changes in the reconfigurable logic domain that are important to anticipate. First, the primary bottleneck of current systems may (at least partially) be alleviated. While today's systems suffer from a significant I/O bottleneck, this need not always be so extreme. There currently are much higher-I/O packages available, such as those used by FPICs to achieve 1,000 pin chips. Although these technologies are quite expensive now, with relatively few applications taking advantage of the extra connectivity, as more applications migrate to these technologies the cost should greatly decrease. Thus, it is quite likely in the near future that FPGAs may see a nearly fourfold increase in the available I/O pins. Also, there are techniques such as Virtual Wires that can also help alleviate the I/O bottleneck. Time-division or voltage-division multiplexing can greatly increase the available bandwidth even in today's I/O-bound chips. Thus, while the I/O bottleneck may never completely disappear, there is great potential to ease the current I/O crunch.

If the potential for increasing the external bandwidth of current FPGAs actually is realized, it may require a complete re-evaluation of our current systems. While today's software and hardware constructs are primarily built to minimize the required inter-chip routing, with more I/O bandwidth other issues become more critical. Primarily, the performance of the mapping becomes an even greater concern, and algorithms such as min-cut partitioning may give way to more critical path oriented approaches. Investigating these issues will become an even greater concern in the future.

Another change in the current multi-FPGA system equation is in the types of circuits handled, or at least in the perception of what these circuits look like. Most current systems either are hand-mapped, and thus highly aware of the peculiarities of a given mapping, or automatically mapped, where the input circuit is assumed to be purely random logic. However, even in our current mappings, the circuits are not truly random. Much of the current logic is regularly structured datapaths. By treating these datapaths as random logic, the automatic mapping software throws away many potential optimizations. While optimizing for random logic may have been the right answer for initial system construction, since if you can map random logic you can handle anything, future systems may need to be more circuit-aware. Specifically, the automatic-mapping software should be able to recognize highly regular datapaths - or require that the user highlight these components - and take advantage of the regularity to provide much better mapping quality. Techniques such as bit-slicing or pipelining at chip boundaries can both improve the performance while decreasing the inter-chip bandwidth. Memory elements in input circuits can also require special processing. While many multi-FPGA systems include separate memory chips, and thus memory structures in the circuit can be handled by these elements, the circuit may need to be restructured to take advantage of these facilities. Since the memories in the hardware may differ from those specified in the circuit, resynthesis and repacking into the available memories will be important to achieve the highest quality mapping. Such circuit-sensitive optimizations should become an ever more important part of a complete automatic mapping solution.

Although there is much work to be done on the hardware and software aspects of multi-FPGA systems, and reprogrammable logic in general, the potential payoffs are significant. With ever faster mapping tools, as well as hardware systems that simplify the job of the mapping algorithms, we may be able to create mappings in the same time it takes us now to compile programs. With such a system, it may be as easy to implement an algorithm in hardware as it is now to run them on a processor. Thus, software programmers will then have access to the power and performance to which the hardware designers are accustomed.

These advanced software processes, as well as improved hardware systems, will push logic emulation further and further into the currently exclusive domain of software simulation. Where today a designer will only use emulation when no other solution will suffice, with fast mapping software logic emulation becomes much more competitive. While very small, short tests will always be the domain of software simulation, testing of large subsystems can much more quickly be performed on a logic emulator if the mapping tools are fast enough. With board-level emulation systems similar benefits will also be achieved for multi-chip designs.

Higher quality software may also push FPGA-based custom-computing machines into the mainstream. Where large supercomputers currently hold dominance, multi-FPGA systems may become an equal partner. While there are some algorithms for which a supercomputer will always be the best answer, likewise there are algorithms for which only the fine-grain parallelism of a multi-FPGA system will suffice. However, the key to making FPGA-based supercomputers generally useful is automatic mapping software that can deliver high enough quality to rival software compilers for traditional programming languages.

Finally, we may see reconfigurable logic making inroads into the workstation and PC workplaces. While this logic may bear only passing similarity to current FPGAs, it will have many of the software and hardware issues of FPGAs. With the ability to add custom instructions to standard processors, as well as truly generic coprocessors for accelerating many types of algorithms, reconfigurable logic may provide a significant performance boost for many situations.

List of References

[Actel94] *FPGA Data Book and Design Guide,* Sunnyvale, CA: Actel Corp, 1994.

[Adams93] J. K. Adams, H. Schmitt, D. E. Thomas, "A Model and Methodology for Hardware-Software Codesign", *International Workshop on Hardware-Software Codesign,* 1993.

[Agarwal94] L. Agarwal, M. Wazlowski, S. Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 101-110, 1994.

[Altera93] *Data Book,* San Jose, CA: Altera Corp, 1993.

[Amerson95] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, G. Snider, "Teramac - Configurable Custom Computing", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Aptix93a] Aptix Corporation, *Data Book*, San Jose, CA, February 1993.

[Aptix93b] Aptix Corporation, *Data Book Supplement*, San Jose, CA, September 1993.

[Arnold94] B. Arnold, "Mega-issues Plague Core Usage", *ASIC & EDA*, pp. 46-58, April, 1994.

[Arnold92] J. M. Arnold, D. A. Buell, E. G. Davis, "Splash 2", *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316-322, 1992.

[Babb93] J. Babb, R. Tessier, A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 142-151, 1993.

[Bapat91] S. Bapat, J. P. Cohoon, "Sharp-Looking Geometric Partitioning", *Euro-DAC*, pp. 172-176, 1991.

[Benner94] T. Benner, R. Ernst, I. Könenkamp, U. Holtmann, P. Schüler, H.-C. Schaub, N. Serafimov, "FPGA Based Prototyping for Verification and Evaluation in Hardware-Software Cosynthesis", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 251-258, 1994.

[Bergmann94] N. W. Bergmann, J. C. Mudge, "Comparing the Performance of FPGA-Based Custom Computers with General-Purpose Computers for DSP Applications", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 164-171, 1994.

[Bertin93] P. Bertin, "Mémoires actives programmables : conception, réalisation et programmation.", *Ph.D. Thesis, Université Paris 7 UFR D'INFORMATIQUE,* 1993.

[Bertin89] P. Bertin, D. Roncin, J. Vuillemin, "Introduction to Programmable Active Memories", in J. McCanny, J. McWhirter, E. Swartzlander Jr., Eds., *Systolic Array Processors*, Prentice Hall, pp. 300-309, 1989.

[Biehl93] G. Biehl, "Overview of Complex Array-Based PLDs", in H. Grünbacher, R. W. Hartenstein, Eds., *Lecture Notes in Computer Science 705 - Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Berlin: Springer-Verlag, pp. 1-10, 1993.

[Blank84] T. Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design", *IEEE Design & Test,* Vol. 1, No. 3, pp. 21-39, 1984.

[Blickle94] T. Blickle, J. König, L. Thiele, "A Prototyping Array for Parallel Architectures", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 388-397, 1994.

[Bolding93] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, R. Wille, "The Chaos Router Chip: Design and Implementation of an Adaptive Router", *Proceedings of the IFIP Conference on VLSI*, pp. 311-320, 1993.

[Bolotski94] M. Bolotski, A. DeHon, T. F. Knight Jr., "Unifying FPGAs and SIMD Arrays", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Box94] B. Box, "Field Programmable Gate Array Based Reconfigurable Preprocessor", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 40-48, 1994.

[Brown92a] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, *Field-Programmable Gate Arrays*, Boston, Mass: Kluwer Academic Publishers, 1992.

[Brown92b] S. Brown, J. Rose, Z. G. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays", *IEEE Transactions on Computer-Aided Design,* Vol. 11, No. 5, pp. 620-628, May 1992.

[Bui89] T. Bui, C. Heigham, C. Jones, T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms", *Design Automation Conference*, pp. 775-778, 1989.

[Butts91] M. Butts, J. Batcheller, "Method of Using Electronically Reconfigurable Logic Circuits", *U.S. Patent 5,036,473*, July 30, 1991.

[Cai91] Y. Cai, D. F. Wong, "Optimal Channel Pin Assignment", *IEEE Transaction on Computer-Aided Design*, Vol. 10, No. 11, pp. 1413-1424, Nov. 1991.

[Carrera95] J. M. Carrera, E. J. Martínez, S. A. Fernández, J. M. Chaus, "Architecture of a FPGA-based Coprocessor: The PAR-1", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Casselman93] S Casselman, "Virtual Computing and the Virtual Computer", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 43-48, 1993.

[Chan95] P. K. Chan, M. D. F. Schlag, J. Y. Zien, "Spectral-Based Multi-Way FPGA Partitioning", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 133-139, 1995.

[Chan94] P. K. Chan, S. Mourad, *Digital Design Using Field Programmable Gate Arrays,* Englewood Cliffs, New Jersey: PTR Prentice Halls, 1994.

[Chan93a] P. K. Chan, M. Schlag, J. Y. Zien, "Spectral K-Way Ratio-Cut Partitioning and Clustering", *Symposium on Integrated Systems*, pp. 123-142, 1993.

[Chan93b] P. K. Chan, M. D. F. Schlag, "Architectural Tradeoffs in Field-Programmable-Device-Based Computing Systems", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 152-161, 1993.

[Chan92] P. K. Chan, M. Schlag, M. Martin, "BORG: A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays", *Proceedings of the 1st International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp. 47-51, 1992.

[Cheng88] C. K. Cheng, T. C. Hu, "Maximum Concurrent Flow and Minimum Ratio-cut", *Technical Report CS88-141*, University of California, San Diego, December, 1988.

[Cong93] J. Cong, M. Smith, "A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design", *Design Automation Conference*, pp. 755-760, 1993.

[Cong92] J. Cong, L. Hagen, A. Kahng, "Net Partitions Yield Better Module Partitions", *Design Automation Conference*, pp. 47-52, 1992.

[Cong91] J. Cong, "Pin Assignment with Global Routing for General Cell Designs", *IEEE Transaction on Computer-Aided Design*, Vol. 10, No. 11, pp. 1401-1412, Nov. 1991.

[Cowen94] C. P. Cowen, S. Monaghan, "A Reconfigurable Monte-Carlo Clustering Processor (MCCP)", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 59-65, 1994.

[Cox92] C. E. Cox, W. E. Blanz, "GANGLION - A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier", *IEEE Journal of Solid-State Circuits,* Vol. 27, No. 3, pp. 288-299, March, 1992.

[Cuccaro93] S. A. Cuccaro, C. F. Reese, "The CM-2X: A Hybrid CM-2 / Xilinx Prototype", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 121-130, 1993.

[Dahl94] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, A. Agarwal, "Emulation of the Sparcle Microprocessor with the MIT Virtual Wires Emulation System", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 14-22, 1994.

[Darnauer94] J. Darnauer, P. Garay, T. Isshiki, J. Ramirez, W. W.-M. Dai, "A Field Programmable Multi-chip Module (FPMCM)", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 1-10, 1994.

[Dobbelaere92] I. Dobbelaere, A. El Gamal, D. How, B. Kleveland, "Field Programmable MCM Systems - Design of an Interconnection Frame", *First International ACM/SIGDA Workshop on Field Programmable Gate Arrays,* pp. 52-56, 1992.

[Dollas94] A. Dollas, B. Ward, J. D. S. Babcock, "FPGA Based Low Cost Generic Reusable Module for the Rapid Prototyping of Subsystems", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 259-270, 1994.

[Donath88] W. E. Donath, "Logic Partitioning", in *Physical Design Automation of VLSI Systems*, B. Preas, M. Lorenzetti, Editors, Menlo Park, CA: Benjamin/Cummings, pp. 65-86, 1988.

[Drayer95] T. H. Drayer, W. E. King IV, J. G. Tront, R. W. Conners, "MORRPH: A MOdular and Reprogrammable Real-time Processing Hardware", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Dunlop85] A. E. Dunlop, B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits", *IEEE Trans. on CAD*, Vol. CAD-4, No. 1, pp. 92-98, January 1985.

[Eldredge94] J. G. Eldredge, B. L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180-188, 1994.

[Engels91] M. Engels, R. Lauwereins, J. A. Peperstraete, "Rapid Prototyping for DSP Systems with Microprocessors", *IEEE Design & Test of Computers*, pp. 52-62, June 1991.

[Erdogan92] S. S. Erdogan, A. Wahab, "Design of RM-nc: A Reconfigurable Neurocomputer for Massively Parallel-Pipelined Computations", *International Joint Conference on Neural Networks*, Vol. 2, pp. 33-38, 1992.

[Fawcett94] B. Fawcett, "Applications of Reconfigurable Logic", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 57-69, 1994.

[Ferrucci94] A. Ferrucci, M. Martín, T. Geocaris, M. Schlag, P. K. Chan, "EXTENDED ABSTRACT: ACME: A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Fiduccia82] C. M. Fiduccia, R. M. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions", *Design Automation Conference,* pp. 241-247, 1982.

[Galil86] Z. Galil, "Efficient Algorithms for Finding Maximum Matching in Graphs", *ACM Computing Surveys*, Vol. 18, No. 1, pp. 23-38, March, 1986.

[Galloway95] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[Galloway94] D. Galloway, D. Karchmer, P. Chow, D. Lewis, J. Rose, "The Transmogrifier: The University of Toronto Field-Programmable System", *Second Canadian Workshop on Field-Programmable Devices*, 1994.

[Garbers90] J. Garbers, H. J. Prömel, A. Steger, "Finding Clusters in VLSI Circuits", *International Conference on Computer-Aided Design*, pp. 520-523, 1990.

[Garey79] M. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: Freeman, 1979.

[Giga95] Giga Operations Corp, "G800 RIC", Berkeley, CA, 1995.

[Gokhale90] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, P. Olsen, "Splash: A Reconfigurable Linear Logic Array", *International Conference on Parallel Processing,* pp. 526-532, 1990.

[Goldberg83] M. K. Goldberg, M. Burstein, "Heuristic Improvement Technique for Bisection of VLSI Networks", *International Conference on Computer Design,* pp. 122-125, 1983.

[Greene93] J. Greene, E. Hamdy, S. Beal, "Antifuse Field Programmable Gate Arrays", *Proceedings of the IEEE,* Vol. 81, No. 7, pp. 1042-1056, July, 1993.

[Hadley95] J. D. Hadley, B. L. Hutchings, "Design Methodologies for Partially Reconfigured Systems", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Hagen92] L. Hagen, A. B. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering", *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 9, pp. 1074-1085, September, 1992.

[Hauck94] S. Hauck, G. Borriello, C. Ebeling, "Springbok: A Rapid-Prototyping System for Board-Level Designs", *ACM/SIGDA 2nd International Workshop on Field-Programmable Gate Arrays*, February, 1994.

[Hayashi95] K. Hayashi, T. Miyazaki, K. Shirakawa, K. Yamada, N. Ohta, "Reconfigurable Real-Time Signal Transport System Using Custom FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Heeb93] B. Heeb, C. Pfister, "Chameleon: A Workstation of a Different Color", in H. Grünbacher, R. W. Hartenstein, Eds., *Lecture Notes in Computer Science 705 - Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Berlin: Springer-Verlag, pp. 152-161, 1993.

[Herpel93] H.-J. Herpel, N. Wehn, M. Gasteier, M. Glesner, "A Reconfigurable Computer for Embedded Control Applications", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 111-120, 1993.

[Hoang93] D. T. Hoang, "Searching Genetic Databases on Splash 2", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 185-191, 1993.

[Högl95] H. Högl, A. Kugel, J. Ludvig, R. Männer, K. H. Noffz, R. Zoz, "Enable++: A Second Generation FPGA Processor", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Hollis87] E. E. Hollis, *Design of VLSI Gate Array ICs,* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[Howard94a] N. Howard, "Defect-Tolerant Field-Programmable Gate Arrays", *Ph.D. Thesis, University of York, Department of Electronics,* 1994.

[Howard94b] N. Howard, A. Tyrrell, N. Allinson, "FPGA Acceleration of Electronic Design Automation Tasks", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 337-344, 1994.

[Huang95] D. J.-H. Huang, A. B. Kahng, "Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 140-145, 1995.

[I-Cube94] I-Cube, Inc., "The FPID Family Data Sheet", Santa Clara, CA, February 1994.

[Iseli95] C. Iseli, E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[Jenkins94] J. H. Jenkins, *Designing with FPGAs and CPLDs,* Englewood Cliffs, New Jersey: PTR Prentice Hall, 1994.

[Jones95] C. Jones, J. Oswald, B. Schoner, J. Villasenor, "Issues in Wireless Video Coding Using Run-time-reconfigurable FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Kadi94] M. Slimane-Kadi, D. Brasen, G. Saucier, "A Fast-FPGA Prototyping System That Uses Inexpensive High-Performance FPIC", *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Katz93] R. Katz, P. Chen, A. Drapeau, E. Lee, K. Lutz, E. Miller, S. Seshan, D. Patterson, "RAID-II: Design and Implementation of a Large Scale Disk Array Controller", *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pp.23-37, 1993.

[Kean92] T. Kean, I. Buchanan, "The Use of FPGAs in a Novel Computing Subsystem", *First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp. 60-66, 1992.

[Kernighan70] B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning of Electrical Circuits", *Bell Systems Technical Journal*, Vol. 49, No. 2, pp. 291-307, February 1970.

[Koch94] G. Koch, U. Kebschull, W. Rosenstiel, "A Prototyping Environment for Hardware/Software Codesign in the COBRA Project", *Third International Workshop on Hardware/Software Codesign*, September, 1994.

[Krishnamurthy84] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers,* Vol. C-33, No. 5, pp. 438-446, May 1984.

[Kuznar94a] R. Kuznar, F. Brglez, B. Zajc, "Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect", *Design Automation Conference*, pp. 238-243, 1994.

[Kuznar94b] R. Kuznar, F. Brglez, B. Zajc, "A Unified Cost Model for Min-cut Partitioning with Replication Applied to Optimization of Large Heterogeneous FPGA Partitions", *European Design Automation Conference*, 1994.

[Kuznar93] R. Kuznar, F. Brglez, K. Kozminski, "Cost Minimization of Partitions into Multiple Devices", *Design Automation Conference*, pp.315-320, 1993.

[Lee89] E. A. Lee, "Programmable DSP Architectures: Part II", *IEEE ASSP Magazine*, Vol.6, No.1, pp. 4-14, January, 1989.

[Lee88] E. A. Lee, "Programmable DSP Architectures: Part II", *IEEE ASSP Magazine*, Vol.5, No.4, pp. 4-19, October, 1988.

[Lemoine95] E. Lemoine, D. Merceron, "Run Time Reconfiguration of FPGA for Scanning Genomic DataBases", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Lewis93] D. M. Lewis, M. H. van Ierssel, D. H. Wong, "A Field Programmable Accelerator for Compiled-Code Applications", *ICCD '93*, pp. 491-496, 1993.

[Ling93] X.-P. Ling, H. Amano, "WASMII: a Data Driven Computer on a Virtual Hardware", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 33-42, 1993.

[Lopresti91] D. P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays", *Advanced Research in VLSI 1991: Santa Cruz*, pp. 139-152, 1991.

[Lysaght94] P. Lysaght, J. Dunlop, "Dynamic Reconfiguration of FPGAs", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 82-94, 1994.

[Mak95] W.-K. Mak, D. F. Wong, "On Optimal Board-Level Routing for FPGA-based Logic Emulation", *Design Automation Conference*, 1995.

[Maliniak94] L. Maliniak, "Hardware Emulation Draws Speed from Innovative 3D Parallel Processing Based on Custom ICs", *Electronic Design*, May 30, 1994.

[McFarland90] M. C. McFarland, A. C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE,* Vol. 78, N0. 2, pp. 301-318, February 1990.

[McMurchie95] L. E. McMurchie, C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 111-117, 1995.

[MCNC93] MCNC Partitioning93 benchmark suite. E-mail benchmarks@mcnc.org for ftp access.

[Moll95] L. Moll, J. Vuillemin, P. Boucard, "High-Energy Physics on DECPeRLe-1 Programmable Active Memory", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 47-52, 1995.

[Monaghan94] S. Monaghan, J. E. Istiyanto, "High-level Hardware Synthesis for Large Scale Computational Physics Targetted at FPGAs", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 238-248, 1994.

[Monaghan93] S. Monaghan, C. P. Cowen, "Reconfigurable Multi-Bit Processor for DSP Applications in Statistical Physics", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 103-110, 1993.

[Mori94] M. T. Mori, *The PCMCIA Developer's Guide*, Sunnyvale, CA: Sycard Technology, 1994.

[Nguyen94] R. Nguyen, P. Nguyen, "FPGA Based Reconfigurable Architecture for a Compact Vision System", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 141-143, 1994.

[Njølstad94] T. Njølstad, J. Pihl, J. Hofstad, "ZAREPTA: A Zero Lead-Time, All Reconfigurable System for Emulation, Prototyping and Testing of ASICs", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 230-239, 1994.

[Oldfield95] J. V. Oldfield, R. C. Dorf, *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems,* New York: John Wiley & Sons, Inc., 1995.

[Öner95] K. Öner, L. A. Barroso, S. Iman, J. Jeong, K. Ramamurthy, M. Dubois, "The Design of RPM: An FPGA-based Multiprocessor Emulator", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 60-66, 1995.

[PCIC93] *82365SL DF PC Card Interface Controller (PCIC),* Santa Clara, CA: Intel Corporation, April 1993.

[Pförtner92] T. Pförtner, S. Kiefl, R. Dachauer, "Embedded Pin Assignment for Top Down System Design", *European Design Automation Conference*, pp. 209-214, 1992.

[Philips92] *Desktop Video Data Handbook,* Sunnyvale, CA: Philips Semiconductors, 1992.

[Quénot94] G. M. Quénot, I. C. Kraljic, J. Sérot, B. Zavidovique, "A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 91-100, 1994.

[Quickturn93] Quickturn Design Systems, Inc., "Picasso Graphics Emulator Adapter", 1993.

[Raimbault93] F. Raimbault, D. Lavenier, S. Rubini, B. Pottier, "Fine Grain Parallelism on a MIMD Machine Using FPGAs", *IEEE Workshop on FPGAs for Custom Computing Machines,* pp. 2-8, 1993.

[Riess94] B. M. Riess, K. Doll, F. M. Johannes, "Partitioning Very Large Circuits Using Analytical Placement Techniques", *Design Automation Conference*, pp. 646-651, 1994.

[Rose93] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proceedings of the IEEE,* Vol. 81, No. 7, pp. 1013-1029, July 1993.

[Roy93] K. Roy, C. Sechen, "A Timing Driven N-Way Chip and Multi-Chip Partitioner", *International Conference on Computer-Aided Design*, pp. 240-247, 1993.

[Saluvere94] T. Saluvere, D. Kerek, H. Tenhunen, "Direct Sequence Spread Spectrum Digital Radio DSP Prototyping Using Xilinx FPGAs", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 138-140, 1994.

[Sample92] S. P. Sample, M. R. D'Amour, T. S. Payne, "Apparatus for Emulation of Electronic Hardware System", *U.S. Patent 5,109,353*, April 28, 1992.

[Schmit95] H. Schmit, D. Thomas, "Implementing Hidden Markov Modelling and Fuzzy Controllers in FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Schuler72] D. M. Schuler, E. G. Ulrich, "Clustering and Linear Placement", *Design Automation Conference*, pp. 50-56, 1972.

[Scott95] S. D. Scott, A. Samal, S. Seth, "HGA: A Hardware-Based Genetic Algorithm", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53-59, 1995.

[Seitz90] C. L. Seitz, "Let's Route Packets Instead of Wires", *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pp. 133-138, 1990.

[Selvidge95] C. Selvidge, A. Agarwal, M. Dahl, J. Babb, "TIERS: Topology IndependEnt Pipelined Routing and Scheduling for VirtualWire™ Compilation", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 25-31, 1995.

[Shahookar91] K. Shahookar, P. Mazumder, "VLSI Cell Placement Techniques", *ACM Computing Surveys*, Vol. 23, No. 2, pp. 145-220, June 1991.

[Shaw93] P. Shaw, G. Milne, "A Highly Parallel FPGA-based Machine and its Formal Verification", in H. Grünbacher, R. W. Hartenstein, Eds., *Lecture Notes in Computer Science 705 - Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Berlin: Springer-Verlag, pp. 162-173, 1993.

[Suaris87] P. R. Suaris, G. Kedem, "Standard Cell Placement by Quadrisection", *ICCD*, pp. 612-615, 1987.

[Tessier94] R. Tessier, J. Babb, M. Dahl, S. Hanono, A. Agarwal, "The Virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environment", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Thomae91] D. A. Thomae, T. A. Petersen, D. E. Van den Bout, "The Anyboard Rapid Prototyping Environment", *Advanced Research in VLSI 1991: Santa Cruz*, pp. 356-370, 1991.

[Trimberger94] S. M. Trimberger, *Field-Programmable Gate Array Technology*, Boston: Kluwer Academic Publishers, 1994.

[Trimberger93] S. Trimberger, "A Reprogrammable Gate Array and Applications", *Proceedings of the IEEE,* Vol. 81, No. 7, pp. 1030-1041, July 1993.

[Van den Bout95] D. E. Van den Bout, "The XESS RIPP Board", in J. V. Oldfield, R. C. Dorf, *Field-Programmable Gate Arrays : Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems,* pp. 309-312, 1995.

[Varghese93] J. Varghese, M. Butts, J. Batcheller, "An Efficient Logic Emulation System", *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 171-174, June 1993.

[Venkateswaran94] R. Venkateswaran, P. Mazumder, "A Survey of DA Techniques for PLD and FPGA Based Systems", *Integration, the VLSI Journal,* Vol. 17, No. 3, pp. 191-240, 1994.

[Vijayan90] G. Vijayan, "Partitioning Logic on Graph Structures to Minimize Routing Cost", *IEEE Trans. on CAD*, Vol. 9, No. 12, pp. 1326-1334, December 1990.

[Vincentelli93] A. Sangiovanni-Vincentelli, A. El Gamal, J. Rose, "Synthesis Methods for Field Programmable Gate Arrays", *Proceedings of the IEEE,* Vol. 81, No. 7, pp. 1057-1083, 1993.

[VMEbus85] *The VMEbus Specification*, Tempe, Arizona: Micrology pbt, Inc., 1985.

[vom Bögel94] G. vom Bögel, Petra Nauber, J. Winkler, "A Design Environment with Emulation of Prototypes for Hardware/Software Systems Using XILINX FPGA", in R. W. Hartenstein, M. Z. Servít, Eds., *Lecture Notes in Computer Science 849 - Field-Programmable Logic: Architectures, Synthesis and Applications*, Berlin: Springer-Verlag, pp. 315-317, 1994.

[Vuillemin95] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, 1995.

[Wakerly94] J. F. Wakerly, *Digital Design: Principles and Practices*, Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[Wazlowski93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, "PRISM-II Compiler and Architecture", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9-16, 1993.

[Wei89] Y.-C. Wei, C.-K. Cheng, "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning", *International Conference on Computer-Aided Design*, pp. 298-301, 1989.

[Weinmann94] U. Weinmann, "FPGA Partitioning under Timing Constraints", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford: Abingdon EE&CS Books, pp. 120-128, 1994.

[Weiss94] R. Weiss, "MCM+4 FPGAs = 50,000 gates, 4048 flip-flops", *EDN*, pp. 116, April 28, 1994.

[Weste85] N. Weste, K. Eshraghian, *Principles of CMOS VLSI Design*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1985.

[Williams91] I. Williams, "Using FPGAs to Prototype New Computer Architectures", in W. Moore, W. Luk, Eds., *FPGAs*, Abingdon, England: Abingdon EE&CS Books, pp. 373-382, 1991.

[Wirthlin95] M. J. Wirthlin, B. L. Hutchings, "A Dynamic Instruction Set Computer", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995.

[Wo94] D. Wo, K. Forward, "Compiling to the Gate Level for a Reconfigurable Co-processor", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 147-154, 1994.

[Wolfe88] A. Wolfe, J. P. Shen, "Flexible Processors: A Promising Application-Specific Processor Design Approach", *21st Annual Workshop on Microprogramming and Microarchitecture,* pp. 30-39, 1988.

[Woo93] N.-S. Woo, J. Kim, "An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementations", *Design Automation Conference,* pp. 202-207, 1993.

[Xilinx94] *The Programmable Logic Data Book*, San Jose, CA: Xilinx, Inc., 1994.

[Xilinx92] *The Programmable Gate Array Data Book*, San Jose, CA: Xilinx, Inc., 1992.

[Yamada94] K. Yamada, H. Nakada, A. Tsutsui, N. Ohta, "High-Speed Emulation of Communication Circuits on a Multiple-FPGA System", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Yang94] H. Yang, D. F. Wong, "Efficient Network Flow Based Min-Cut Balanced Partitioning", to appear in *International Conference on Computer-Aided Design*, 1994.

[Yao88] X. Yao, M. Yamada, C. L. Liu, "A New Approach to the Pin Assignment Problem", *Design Automation Conference*, pp. 566-572, 1988.

[Yeh92] C.-W. Yeh, C.-K. Cheng, T.-T. Lin, "A Probabilistic Multicommodity-Flow Solution to Circuit Clustering Problems", *International Conference on Computer-Aided Design*, pp. 428-431, 1992.

[Yeh91] C.-W. Yeh, C.-K. Cheng, T.-T. Y. Lin, "A General Purpose Multiple Way Partitioning Algorithm", *Design Automation Conference*, pp. 421-426, 1991.

[Yoeli90] M. Yoeli, *Formal Verification of Hardware Design*, Los Alamitos, CA: IEEE Computer Society Press, 1990.

[Zycad94a] *Paradigm RP*, Fremont, CA: Zycad Corporation, 1994.

[Zycad94b] *Paradigm ViP*, Fremont, CA: Zycad Corporation, 1994.

[Zycad94c] *Paradigm XP*, Fremont, CA: Zycad Corporation, 1994.

Curriculum Vitae

Education

Ph.D., Computer Science & Engineering, University of Washington, 1995.
Thesis: *Multi-FPGA Systems and Rapid-Prototyping of Board-Level Designs*.
Advisors: Gaetano Borriello and Carl Ebeling.
M.S., Computer Science & Engineering, University of Washington, 1992.
B.S., Electrical Engineering & Computer Science, University of California - Berkeley, 1990.

Research Interests

Multi-FPGA Systems, FPGA Architectures and CAD tools, Rapid-Prototyping, Asynchronous Circuit and VLSI Design, Parallel Processing and Parallel Programming Languages.

Awards

AT&T Bell Laboratories Graduate Fellowship
Berkeley Honors Society
National Merit Finalist

Employment

Northwestern University, Evanston, IL

10/95 -	Assistant Professor.

University of Washington, Seattle, WA

3/91- 9/95	Research Assistant. Investigated multi-FPGA systems, rapid-prototyping, asynchronous design. Developed FPGA architectures. Helped write Orca-C portable parallel programming language.
10/90 - 3/91	Teaching Assistant. Intro. to A.I. & Intro. to Operating Systems

University of California - Berkeley, Berkeley, CA

1/90 - 5/90	Reader. Graded for CS170 - Intro. to Computer Science Theory
1/87 - 5/89	Senior Engineering Aid. Participated in the design and development of the PICASSO user interface for the POSTGRES database project.

I.B.M. - T. J. Watson Research Center, Hawthorne, NY

6/89 - 12/89	Temporary Employee - Co-op. Built a framework and toolkit for a set of X11-based Common Lisp programming tools. Wrote a data object inspector, and helped write a call graph/data-flow graph inspector and debugger. System was distributed commercially by Lucid, Inc. as *XLT*.

Bell Communications Research (Bellcore), Morristown, NJ

6/88 - 8/88	Technical Summer Intern. Helped with a B-ISDN feasibility study.

Publications

Journal Articles

S. Hauck, S. Burns, G. Borriello, C. Ebeling, “An FPGA For Implementing Asynchronous Circuits”, *IEEE Design & Test of Computers*, Vol. 11, No. 3, pp. 60-69, Fall, 1994.

S. Hauck, “Asynchronous Design Methodologies: An Overview”, *Proceedings of the IEEE*, Vol. 83, No. 1, pp. 69-93, January, 1995.

G. Borriello, C. Ebeling, S. Hauck, S. Burns, “The Triptych FPGA Architecture”, *IEEE Transactions on VLSI Systems*, December, 1995.

C. Ebeling, L. McMurchie, S. Hauck, S. Burns, “Mapping Tools for the Triptych FPGA”, *IEEE Transactions on VLSI Systems*, December, 1995.

S. Hauck, G. Borriello, “An Evaluation of Bipartitioning Techniques”, submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Patents and Book Chapters

S. Hauck, G. Borriello, S. Burns, C. Ebeling, “Field-Programmable Gate Array for Synchronous and Asynchronous Operation”, U.S. Patent 5,367,209, November 22, 1994.

J. A. Brzozowski, S. Hauck, C.-J. H. Seger, “Chapter 15: Design of Asynchronous Circuits”, in J. A. Brzozowski, C.-J. H. Seger, *Asynchronous Networks*, Springer-Verlag, 1995.

Conference and Symposium Papers

S. Hauck, G. Borriello and C. Ebeling, “TRIPTYCH: An FPGA Architecture with Integrated Logic and Routing”, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pp. 26-43, March, 1992.

S. Hauck, G. Borriello, C. Ebeling, “Mesh Routing Topologies for Multi-FPGA Systems”, *International Conference on Computer Design*, pp. 170-177, October, 1994.

S. Hauck, G. Borriello, “Logic Partition Orderings for Multi-FPGA Systems”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 32-38, February, 1995.

S. Hauck, G. Borriello, “An Evaluation of Bipartitioning Techniques”, *Chapel Hill Conference on Advanced Research in VLSI*, March, pp. 383-402, 1995.

Workshop Papers

C. Ebeling, G. Borriello, S. Hauck, D. Song, and E. A. Walkup, “Triptych: A New FPGA Architecture”, *Oxford Workshop on Field-Programmable Logic and Applications*, September, 1991. Also appearing in W. Moore, W. Luk, Eds., *FPGAs*, Abingdon, England: Abingdon EE&CS Books, pp. 75-90, 1991.

E. A. Walkup, S. Hauck, G. Borriello, and C. Ebeling, “Routing-directed Placement for the Triptych FPGA”, *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, February, 1992.

S. Hauck, G. Borriello, S. Burns and C. Ebeling, “Montage: An FPGA for Synchronous and Asynchronous Circuits*”, 2nd International Workshop on Field-Programmable Logic and Applications*, August, 1992. Also appearing in H. Grunbacher, R. W. Hartenstein, Eds., *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Berlin: Springer-Verlag, pp. 44-51, 1993.

S. Hauck, G. Borriello, C. Ebeling, “Mesh Routing Topologies for FPGA Arrays”, *ACM/SIGDA 2nd International Workshop on Field-Programmable Gate Arrays*, February, 1994.

S. Hauck, G. Borriello, C. Ebeling, “Springbok: A Rapid-Prototyping System for Board-Level Designs”, *ACM/SIGDA 2nd International Workshop on Field-Programmable Gate Arrays*, February, 1994.

S. Hauck, G. Borriello, “Pin Assignment for Multi-FPGA Systems (Extended Abstract)”, *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 11-13, April, 1994.

Technical Reports

S. Hauck, G. Borriello, "Pin Assignment for Multi-FPGA Systems", *University of Washington, Dept. of Computer Science & Engineering Technical Report #94-04-01*, April 1994.

S. Hauck, G. Borriello, C. Ebeling, "Achieving High-Latency, Low-Bandwidth Communication: Logic Emulation Interfaces", *University of Washington, Dept. of Computer Science & Engineering Technical Report #95-04-04*, January 1995.