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].

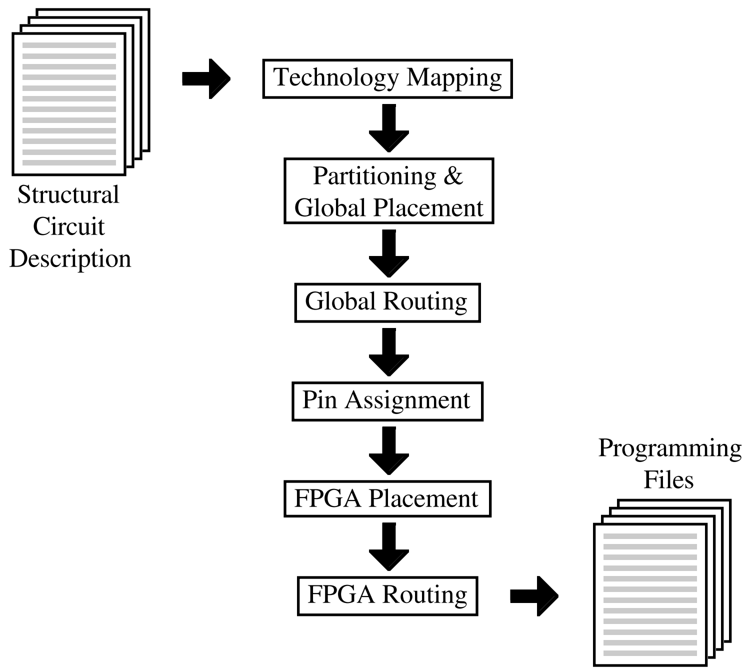


Figure 63. Multi-FPGA system mapping software flow.

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.

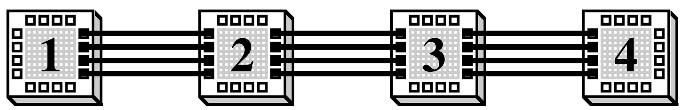


Figure 64. Example system for topological effects on multi-FPGA system partitioning.

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.

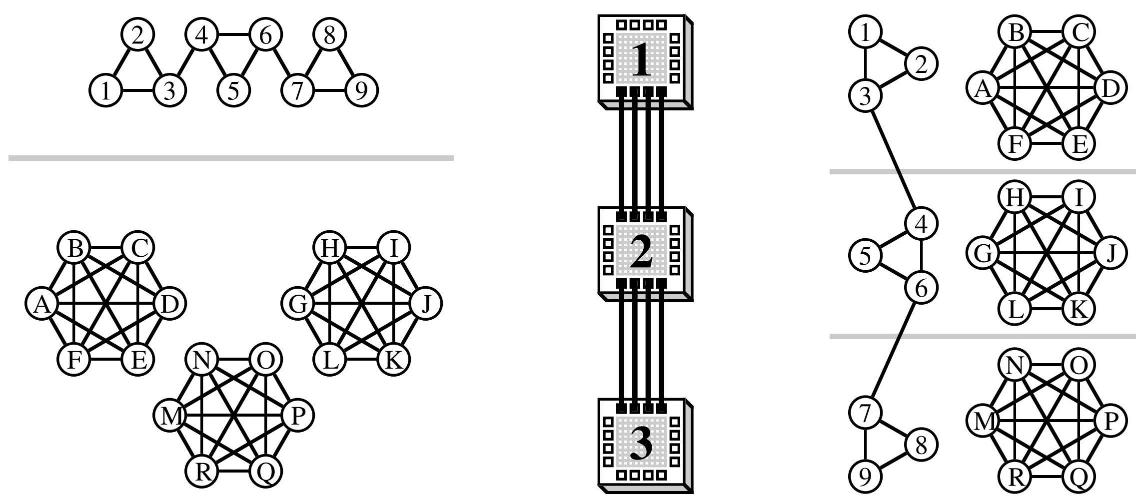


Figure 65. An example of the suboptimality of recursive bipartitioning. We are partitioning to the multi-FPGA system at middle, and the best first cut is shown at left (gray line). The next partitioning would have to cut nine signals. We can instead partition as shown at right, cutting a total of two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

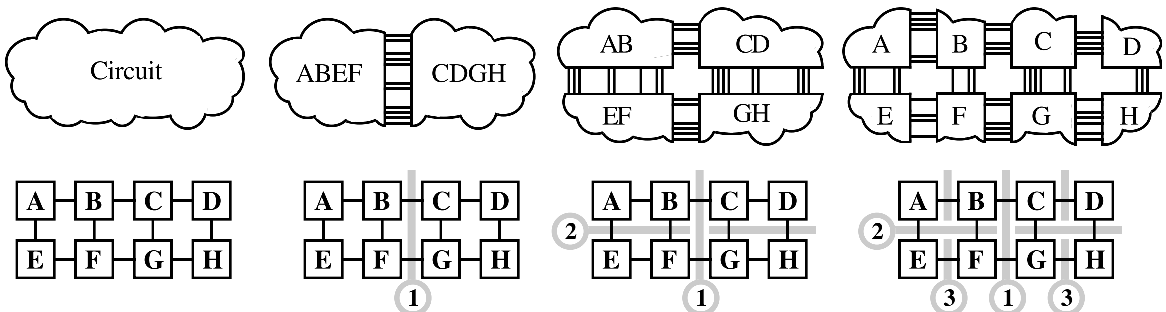


Figure 66. Example of iterative bipartitioning. The circuit (top left) is partitioned onto the multi-FPGA topology (bottom left) in a series of steps. Each partitioning corresponds to the most critical bottleneck remaining in the multi-FPGA system, and after each partitioning the placement of the logic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

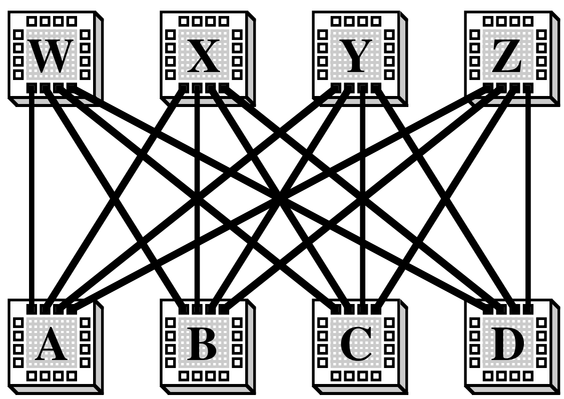


Figure 67. Crossbar routing topology. The chips at top are used purely for routing, while the FPGAs at bottom handle all the logic in the system.

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.

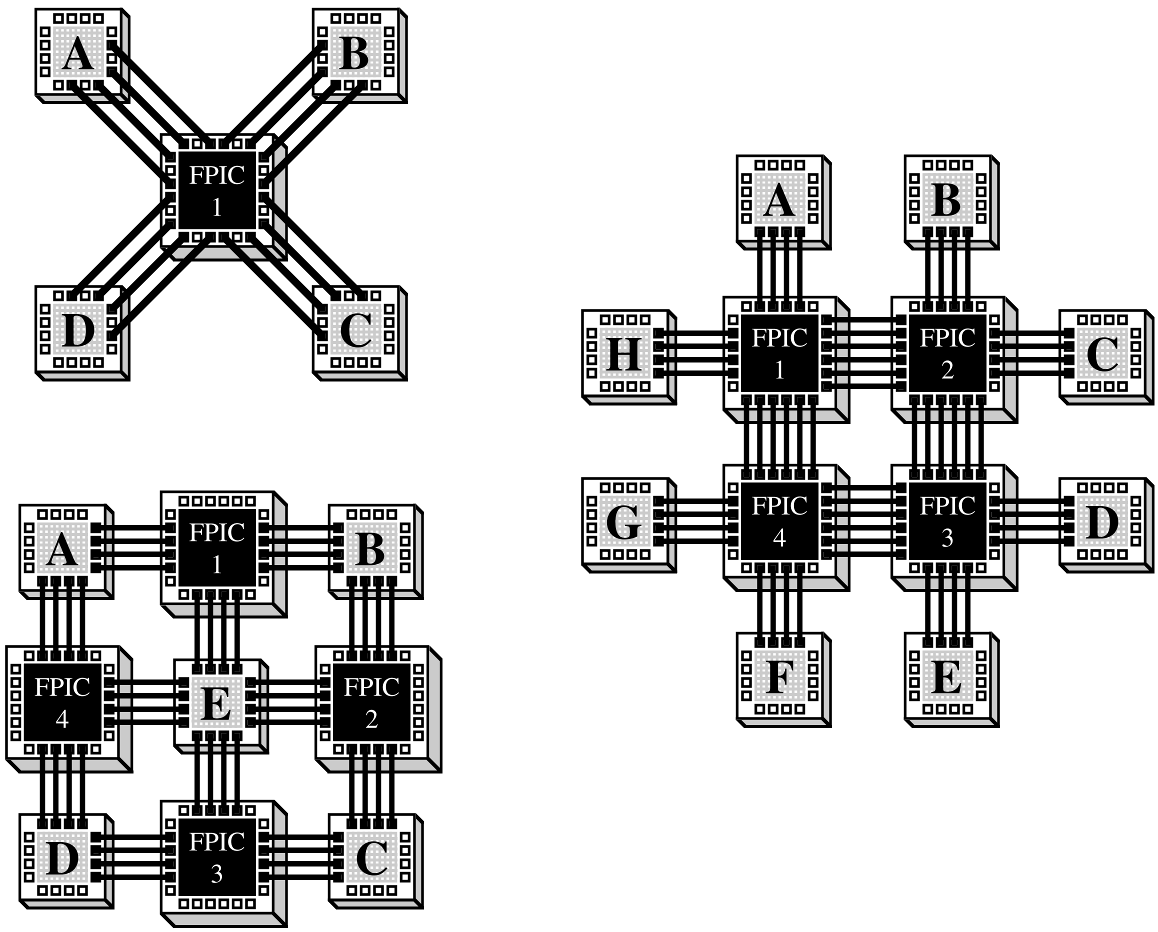
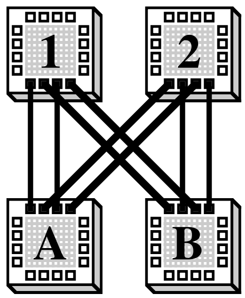


Figure 68. Multi-FPGA topologies with simple pin assignments.

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.



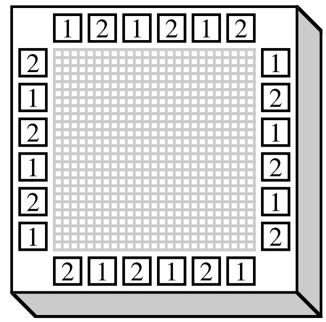


Figure 69. Topology for integrated global routing and pin assignment (left), with the FPGAs at bottom used for logic, and the FPGAs at top for routing only. The detailed pin connections of the logic-bearing FPGAs are given at right. The numbers in the pin positions indicate to which routing-only chip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