IEEE Transactions on VLSI Systems, Vol. 3, No. 4, pp. 473-482, December, 1995

Placement and Routing Tools for the Triptych FPGA

Carl Ebeling, Larry McMurchie, Scott Hauck, Steven Burns

Department of Computer Science and Engineering

University of Washington

Seattle, WA 98195

Abstract

Field-programmable gate arrays (FPGAs) are becoming an increasingly important implementation medium for digital logic. One of the most important keys to using FPGAs effectively is a complete, automated software system for mapping onto the FPGA architecture. Unfortunately, many of the tools necessary require different techniques than traditional circuit implementation options, and these techniques are often developed specifically for only a single FPGA architecture. In this paper we describe automatic mapping tools for Triptych[1]*, an FPGA architecture with improved logic density and performance over commercial FPGAs. These tools include a simulated-annealing placement algorithm that handles the routability issues of fine-grained FPGAs, and an architecture-adaptive routing algorithm that can easily be retargeted to other FPGAs. We also describe extensions to these algorithms for mapping asynchronous circuits to Montage, the first FPGA architecture to completely support asynchronous and synchronous interface applications.*

1 Introduction

Field-programmable Gate Arrays (FPGAs) are one of today's most important digital logic implementation options. An important component of an FPGA-based design environment is the automatic mapping tools necessary to effectively utilize the chips. Software is used to divide a source logic specification into logic functions that are directly implemented in the FPGA (covering/technology mapping), assign these logic functions to specific locations in the FPGA (placement), and connect the logic signals from their sources to their sinks (routing). Because of this reliance on software for generating FPGA-based designs, automatic mapping software is critical to the success of an FPGA architecture, and an FPGA architecture is only as good as the tools that map to it.

The heavy automation of the mapping process is not unique to FPGAs; Similar tools exist for mask-programmable gate arrays, standard and macro cells, and even some parts of full-custom design. However, FPGAs differ from these other technologies in one critical factor: while the logic and routing resources in an FPGA can be customized by the end-user, the amount and location of each of

---

[1] Triptych is described in the companion paper "The Triptych FPGA Architecture" [Hauck95].

the resources is fixed by the architecture. Thus, in contrast to other technologies where routers seek to minimize the number of signals in a channel, but can expand these channels to handle the required capacity, the number of routing resources in an FPGA is fixed, and a mapping solution that requires even one more wire than a given channel is designed to support is just as infeasible as a solution that overflows by many wires. While this problem is characteristic of traditional channeled gate arrays, it is more extreme for FPGAs.

Simulated annealing has been applied to the FPGA placement problem in a manner similar to the placement of standard cells [Sechen87]. While standard cell techniques are sufficient for those FPGAs that invest a large portion of their chip area in routing resources [Xilinx93], special care must be taken in FPGA architectures that seek to limit the cost of routing. For example, architectures such as the Algotronix CAL [Algotronix91] and Triptych have localized, limited routing resources, and a good placement will not only put connected logic functions together, but will also ensure that logic elements are not packed too closely for the routing to succeed. Algorithms have been developed specifically for the placement of logic in FPGAs. [Togawa94] uses a min-cut placement combined with hierarchical global routing that introduces signal congestion into the placement process. [Beetem91] uses a penalty-driven iterative improvement algorithm.

The problem of routing FPGAs bears a considerable resemblance to the problem of global routing for custom integrated circuit design. In both cases the goal is to assign signal routes to routing resources in order to minimize congestion and achieve performance goals. Both problems can be attacked by representing the routing resources as graphs and applying variants of minimum spanning tree algorithms. However, the two problems are different in several fundamental respects. Routing resources in FPGAs are discrete and finite, while they are more or less continuous in custom integrated circuits. Depending on the architecture of the FPGA and the type of routing resource (mux, pass transistor, or static antifuse), these resources may be relatively expensive. Signals compete for the same routing resources, and a circuit will not fit in a given FPGA if the congested routes cannot be resolved. For this reason FPGAs require a detailed accounting of congestion. In some sense, routing an FPGA requires integration of both global and detailed routing into a single algorithm.

Another important difference is that the global routing problem for custom ICs is rooted in an undirected graph with a Manhattan distance metric. In FPGAs, the switches are often directional, and the routing resources connect arbitrary (but fixed) locations. This distinction is important, as it prevents direct application of much of the work that has been done in custom IC routing.

By far the most common approach to global routing of custom ICs is a shortest path algorithm with obstacle avoidance [Lee61]. By itself, this technique usually yields many unrouteable nets, which

must be rerouted by hand. A multitude of rip-up and retry approaches have been proposed to remedy the deficiencies of this approach ([Kuh86], [Linsker89], [Cohn91]). In essence, rip-up and retry involves rerouting nets in congested areas. The basic problem of rip-up and retry is that the success of a route is dependent not just on the choice of which nets to reroute, but also on the order that the rerouting is done.

Most of the work to date in FPGA routing has applied variants on rip-up and retry schemes. Often specific features of a target architecture are exploited, with a resulting loss in generality. [Hill91] uses a breadth-first search while performing routes in random order. A "blame factor" is introduced to decide what routes need to be ripped up when a connection is not made. [Palczewski92] describes an application of the A* algorithm to the switchboxes in the Xilinx architecture. [Brown92] uses a global router to assign connections so that channel densities are balanced. A detailed router generates families of explicit paths within channels to resolve congestion. If some connections are unrealizable, the channel routes are ripped up and a rerouting is performed using larger families of paths.

Delay is usually factored into the standard rip-up and retry approach by ordering the nets to be routed so that critical nets are routed most directly [Brown92]. How to balance the competing goals of minimizing delay of critical paths and minimizing congestion is an open question. In [Frankle92] a slack analysis is performed to calculate upper bounds for individual source-sink connections. A rip-up and retry scheme then routes signals, increasing upper bounds as needed. Once the routing is completed, selected connections are rerouted to reduce the overall delay. Although the results of this scheme are good (delays of the final routes average only 16% higher than optimal), this scheme suffers from a dependency upon the order that the connections are routed. Also, by performing a slack analysis only at the beginning and the end of the routing process, opportunities for balancing congestion and delay are lost.

2. The Interdependence of Architecture and Tools

It is important when developing a new FPGA architecture to ensure that the mapping tools will be able to take advantage of it. There is a strong analogy between processor architecture and compilers. Architectural features that tools cannot handle are not useful. Thus, it is impossible to evaluate an architecture or a set of tools in isolation. They are sufficiently interdependent that they must be developed and evaluated together. An unfortunate result is that some architectural features that may be valuable in their own right may be discarded because current tools cannot support them sufficiently. With increasingly sophisticated tools, previously discounted architectural ideas may become viable.

We structured the Triptych tools to support architecture development. Both the placement and routing programs were optimized for flexibility and not performance in terms of CPU time. It was more important to be able to retarget the tools quickly to evaluate variants on the Triptych architecture than to have the fastest turn-around time for individual tool runs. We recognized a tension between generality, which allows flexibility, and specificity, which allows the tools to take advantage of specific architectural features. Thus the first requirement of the placement and routing tools was that they be specific enough to take advantage of the primary features of Triptych. Second, we required enough generality to allow changes in the design of the RLB, the local interconnect and the vertical bus structure.

Flexibility was incorporated into the placement program by isolating the architecture-specific features to the cost function. While we could have introduced an architectural analysis phase that pre-computed a cost function based on an architectural description, we instead opted for parameterizing certain aspects of the cost function while requiring other components such as local routability to be rewritten for the new architecture.

The routing resources of Triptych were described using the schematic capture system WireC [McMurchie94]. The description includes all specifics about the construction of the RLBs, the segmentation of vertical buses and diagonal connections. The output of the WireC system is a directed graph over all routing resources that includes delay information. Retargeting the router to a new architecture is a straightforward matter of modifying an existing template or creating a new one; no code modifications to the router are required. Indeed, we have retargeted the router to the Xilinx 3000 architecture and achieved very encouraging results.

3 Triptych Placement Software

The placement software for Triptych is based on a simulated annealing approach with a cost function that optimizes several different metrics. These metrics include wirelength as well as measures of the routability of the placement. Minimizing the wirelength will generally cause cells to be placed tightly in the center of the array, which almost certainly results in unrouteable nets. The cost function developed in this section assumes the three-input, three output RLB architecture described in Section 2.1 of the companion paper [Hauck95]. This architecture relies on using some number of empty cells in the array for routing. These cells must be allocated as a part of the placement process.

The wirelength calculation for the Triptych architecture requires a very different calculation than for other technologies. In particular, it is clear that the standard Manhattan distance metric is inappropriate since many of Triptych's wires run diagonally through RLBs, others run vertically in

segmented busses, and there are no horizontal wires at all. The wirelength metric must therefore include both diagonal and vertical components. The diagonal distance is computed using a modified Manhattan distance, where instead of horizontal and vertical paths, we use NE-SW and SE-NW diagonals. To account for the directionality of the RLBs, where a right-flowing RLB can reach a cell directly to the NE in one step, but a cell directly to the NW requires three steps (see Figure 1), we can use the fact that the distance along diagonals is identical from both a right-flowing RLB and the left-flowing RLB one step to the right. Thus, distances to the NE and SE are measured from the source RLB if it is right-flowing, or from the left-flowing RLB one step to the right otherwise. It is then straightforward to derive formulas for the distances between arbitrary RLBs. Vertical wires are implemented as vertical segmented channels, which means that it is more important to place cells closer in the horizontal dimension than vertically. To reflect this, all vertical distances in the cost metric are reduced by a multiplicative factor. This factor must be chosen with care. If it favors vertical routing too much, there will be too much competition for track resources. Experimentally, we have found that a factor of 2.0 is about right for the current Triptych architecture.

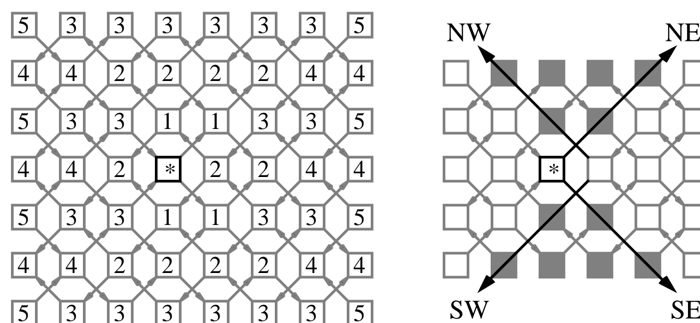

Figure 1. Distances along diagonals from the cell marked with the asterisk (at left) are calculated as shown. Distances to the northeast and southeast are calculated from right-flowing cells, while those to the northwest and southwest are calculated from left-flowing cells.

While the above metric is fine for single destination nets, multiple destination nets require more care. Specifically, although signal destinations might all be far from the source, if they are clustered together it is easier to route than if they are also very distant from each other. To reflect this, many systems use the semi-perimeter distance metric. For Triptych, this would mean that the signal length is the sum of the maximum diagonal distances along each of the NE, NW, SE, and SW directions among all of the destinations. The primary problem with the semi-perimeter metric is that a distant node can overshadow a closer node, so that an annealer will not realize that placing these closer destinations adjacent to the source is a better placement. To fix this the cost function should include the average distance from the source to all destinations. What we have done is take a hybrid approach, with 90% of a signal's distance determined by semi-perimeter, and the average distance

making up the final 10%. This yields a metric with the clustering benefit of semi-perimeter, while eliminating much of semi-perimeter's problems.

While minimizing wirelength minimizes the number of routing resources required globally, it does nothing to ensure that signals can be routed locally given the competition among signals for scarce resources. We have added two components to the cost function which address this problem: "local routability" and "density smoothing." Local routability attaches a cost to those situations where it can be determined that a signal cannot be routed given local routing resources. Each function in the RLB array requires two or three inputs, only one of which can be supplied by a vertical bus. Thus two-input functions must receive one of their signals on a diagonal from its neighbors, and three-input functions must receive two. There are four adjacent RLBs which can provide these diagonal inputs and the local routability function checks to make sure that the required input signals are either present in these RLBs or that there are sufficient route-throughs available so they could be routed. Since a right-flowing RLB uses the same diagonals for input signals as the RLB directly to its left, pairs of RLBs must be checked together. Of course, local routability finds only illegal placements that can be deduced from the immediate context. It is, moreover, a step function, as opposed to the ideal of a smoothly-changing cost function that gives best results in simulated annealing. As a result local routability has the overall effect of disallowing certain moves from consideration.

Density smoothing addresses the inadequacies of local routability. This component is designed to prevent routing congestion from an over concentration of functions in one part of the array. The metric itself consists of looking at small windows of RLBs, three cells on a side, and counting the number of "pegs" in this region. A "peg" is a used RLB input, which means a two-input function has two pegs, a three-input function has three pegs, and an unused RLB has no pegs. To ensure that the "holes" (RLB inputs unfilled by a peg, which represents a routing opportunity) are evenly spread, the penalty is the square of the number of pegs above a threshold in a window, summed across all windows. The squaring is necessary to penalize peg hot-spots more than smooth peg distributions. The threshold is required so that a small circuit mapped onto an array will not be spread throughout the array. Note also that we examine every unique window in the FPGA, which means many windows overlap. To avoid edge effects, windows are also allowed to move beyond the chip edge, with the virtual cells beyond the chip boundaries assumed to have as many pegs as the overall average. This is important, because if we either did not allow windows to move beyond the chip edge, or assumed that the virtual cells had no pegs, large numbers of pegs (and the associated logic) would congregate at the edge. Similarly, if we assumed the virtual cells were completely filled with pegs, pegs would strongly avoid the edge. Both cases would tend to build up wavefronts of pegs in the chip, with high-peg rows and columns alternating with low-peg rows and columns, yielding extremely bad placements.

Delay is introduced into the cost function by performing a path analysis prior to the start of the annealing process. All paths that start and end at I/O pins or latches are considered, and the maximum length path (where length is the number of logic levels along the path in the source mapping) containing each signal is determined. All source-sink distances are then weighted relative to the critical path and the contribution of their lengths to the overall cost function are scaled accordingly.

4 Triptych Routing Software

Our approach to routing for Triptych is based on an iterative approach to global routing of custom integrated circuits developed by Nair [Nair87]. This approach differs in several aspects from most forms of rip-up and retry. Only one net is ripped up at a time, but every net is ripped up and rerouted on every iteration, even if the net does not pass through a congested area. In this way nets passing through uncongested areas can be diverted to make room for other nets currently in congested regions. Nets are ripped up and rerouted in the same order every iteration. Our routing algorithm differs from Nair's primarily in the construction of the cost function and the handling of delay.

The algorithm can be described as two interacting parts: a signal router, which routes one signal at a time using a shortest-path algorithm, and a global router, which calls the signal router to route all signals, adjusting the resource costs in order to achieve a complete routing. The signal router uses a breadth-first search to find the shortest path given a congestion cost and delay for each routing resource. The global router dynamically adjusts the congestion penalty of each routing resource based on the demands signals place on that resource. During the first iteration of the global router there is no cost for sharing routing resources, and individual routing resources may be used by more than one signal. However, during subsequent iterations the penalty is gradually increased so that signals in effect negotiate for resources. Signals may use shared resources that are in high demand if all alternative routes utilize resources in even higher demand; other signals will tend to spread out and use resources in lower demand. The global router reroutes signals using the signal router until no more resources are shared. The use of a cost function that *gradually* increases the penalty for sharing is a significant departure from Nair's algorithm, which assigns a cost of infinity to resources whose capacity is exceeded.

In addition to minimizing congestion, the signal router ensures that the delay of all signal paths stays within the critical path delay. For multiple sinks, low congestion cost can be achieved by a minimum Steiner tree, but this can result in long delays. Low delay can be achieved by a minimum-delay tree, but this may mean competition by many signals for the same routing resources. To achieve a balance, the signal router uses the relative contribution of each connection in the circuit (i.e. source-sink pair)

to the overall delay of the circuit to determine how to trade off congestion and delay. A slack ratio is computed for each connection in the circuit as the ratio of the delay of the longest path using that connection to the delay of the circuit's longest (i.e. most critical) path. Thus, every connection on the longest path has a slack ratio of 1, while connections on the least critical paths have slack ratios close to 0. The inverse of the slack ratio gives the factor by which the delay of a path can be expanded before the circuit is slowed down.

The key idea behind the signal router is that connections with a slack ratio close to 1 will be assigned greater weight in negotiating for resources and consequently will be routed directly (i.e. using a minimum-delay route) from source to sink. Connections with a small slack ratio will have less weight and pay more attention to congestion-avoidance during routing. A net with multiple sinks (which corresponds to several connections with varying slack ratios) will be routed using a combined strategy, and will not be constrained to either an overall minimum Steiner tree or minimum-delay tree route. The slack mechanism provides a smooth tradeoff between these two extremes.

4.1 Terminology

The routing resources in an FPGA and their connections are represented by the directed graph $G = (V,E)$. The set of vertices V corresponds to the electrical nodes or wires in the FPGA architecture, and the edges E to the switches that connect these nodes. Associated with each node n in the architecture is a constant delay d_n and a congestion cost c_n determined by the competition among signals for n.

Given a signal i in a circuit mapped onto the FPGA, the signal net N_i is the set of terminals including the source terminal s_i and sinks t_{ij}. N_i forms a subset of V. A solution of the routing problem for signal i is the directed routing tree RT_i embedded in V and connecting s_i with all its t_{ij}.

4.2 Congestion-based Router

We will first present a pure congestion-based routing algorithm in this subsection, and then extend it to optimize delay in the next subsection. The cost of using a given node n in a route is given by

$$c_n = (b_n + h_n) * p_n \qquad (1)$$

where b_n is the base cost of using n, h_n is related to the history of congestion on n during previous iterations of the global router, and p_n is related to the number of other signals presently using n. A reasonable choice for b_n is the intrinsic delay d_n of the node n, since minimizing the delay of a path in general minimizes the number of routing resources of a path.

The h_n and p_n terms are motivated by the routing problems in Figures 2 and 3. Figure 2 shows a *first order congestion* problem. We need to route signals 1, 2, and 3 from their sources S_1, S_2, and S_3 to their respective sinks D_1, D_2, and D_3. The arcs in the graph represent partial paths, with the associated

costs in parentheses. Ignoring congestion, the minimum cost path for each signal would use node B. If a simple obstacle-avoidance routing scheme is used to eliminate congestion, the order in which the signals are routed now becomes important. If the signals are routed in the order (3, 2, 1), signal 3 will route through B, 2 through A, and 1 will be unrouteable. Other orderings will be routeable, but the total routing cost will be a minimum only if we start with signal 2.

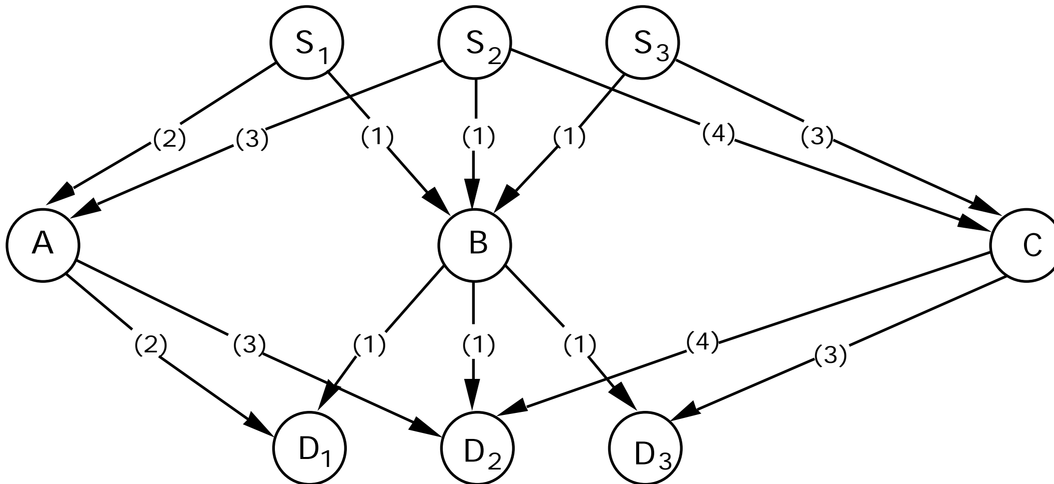

Figure 2. First order congestion

The first-order congestion of Figure 2 can be solved using the p_n factor in our cost function (assuming for the time being $h_n = 0$). During the first iteration of the global router, p_n is initialized to one, thus no penalty is imposed for the use of n regardless of how many signals occupy n. During subsequent iterations, this penalty is gradually increased, depending on how many signals share n. In the first iteration therefore, all three signals share B. During some later iteration signal 1 will find that a route through A gives a lower cost than through the congested node B. During an even later iteration signal 3 will find that a route through C gives a lower cost than through B. This scheme of negotiation for routing resources depends on a relatively gradual increase in the cost of sharing nodes. If the increase is too abrupt, signals may be forced to take high cost routes that lead to other congestion. Just as in the simple obstacle-avoidance scheme, the ordering would become important.

Figure 3 shows an example of *second order congestion*. Again, we need to route three signals, one from each source to the corresponding sink. Let us first consider this example from the standpoint of obstacle-avoidance with rip-up and retry. Assume that we start with the routing order (1, 2, 3). Signal 1 routes through B, and signals 2 and 3 share node C. For ripup and retry to succeed, both signals 1 and 2 would have to be rerouted, with signal 2 rerouted first. Because signal 1 does not use a congested node, determining that it needs to be rerouted will be difficult in general.

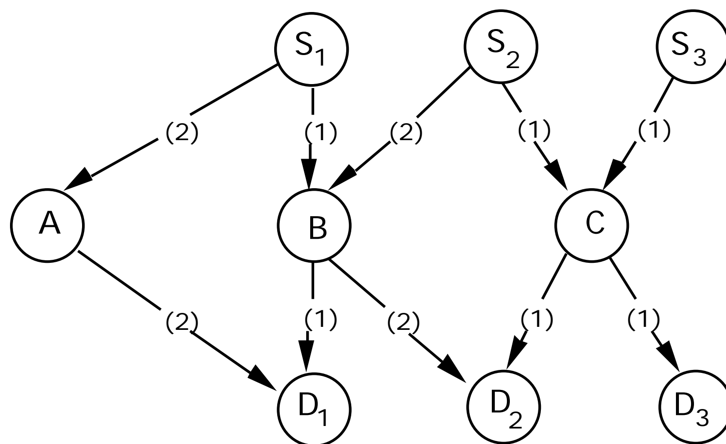

Figure 3. Second order congestion

This second-order congestion problem cannot be solved using p_n alone. During the first iteration signal 1 is routed through *B* and signals 2 and 3 through *C*. During subsequent iterations, the cost of sharing node *C* increases. Signal 3 has no alternative. Signal 2 could share node *B* with signal 1, but the cost of that route will always be greater than the route through *C*. This is because the cost of the path from S_2 to D_2 via *B* is greater than the corresponding path through *C*, and the cost of sharing *B* and *C* is the same. Therefore, signal 2 never attempts the path through *B*.

The term h_n overcomes this problem. Each iteration that node *C* is shared, h_n is increased slightly. After enough iterations, the route through *C* will become more expensive for signal 2 than the route through *B*. Once *B* is shared by both signals 1 and 2, signal 1 will be rerouted through *A*, and the congestion will be eliminated. The effect of h_n is to permanently increase the cost of using congested nodes so that routes through other nodes are attempted. The addition of this term to account for the history of congestion of a node is another distinction between our algorithm and Nair's.

The congestion-based routing algorithm is described in detail in Figure 4. The signal router loop starts at step 2. The routing tree RT_i from the previous global routing iteration is erased and initialized to the signal source. A loop over all sinks t_{ij} of this signal is begun at step 5. A breadth-first search for the closest sink t_{ij} is performed using the priority queue *PQ* in steps 7-12. Fanouts *n* of node *m* are added to the priority queue at $c_n + P_{im}$, where P_{im} is the cost of the path from s_i to *m*.

After a sink is found, all nodes along a backtraced path from the sink to source are added to RT_i (steps 13-16), and this updated RT_i is the source for the search for the next sink (step6). In this way, all locations on routes to previously found sinks are used as potential sources for routes to subsequent sinks. This is similar to Prim's algorithm for determining a minimum spanning tree over an

undirected graph. In our algorithm the minimum path from the tree RT_i to the closest sink will be found. The branch leading to this sink may start from an intermediate (neither source nor sink) node. This algorithm for constructing the routing tree is identical to an algorithm suggested by [Takahishi80] for constructing a Steiner tree embedded in an undirected graph. The quality of the Steiner points chosen by the algorithm is an open question for directed graphs. Finding optimum (or even near-optimum) Steiner points is not essential for successful routing because the global router can adjust costs to eliminate congestion and complete routes.

While shared resources exist (global router) [1]
    Loop over all signals i (signal router) [2]
        Rip up routing tree RT_i [3]
        RT_i s_i [4]
        Loop until all sinks t_{ij} have been found [5]
            Initialize priority queue PQ *to* RT_i at cost 0 [6]
            Loop until new t_{ij} is found [7]
                Delete lowest cost node m from PQ [8]
                Loop over fanouts n of node m [9]
                    Add n to PQ at cost $c_n + P_{im}$ [10]
                End [11]
            End [12]
            Loop over nodes n in path t_{ij} to s_i (backtrace) [13]
                Update c_n [14]
                Add n to RT_i [15]
            End [16]
        End [17]
    End [18]
End [19]

Figure 4. Congestion-based routing algorithm

4.3 Congestion/Delay Router

To introduce delay into the congestion-based router, we redefine the cost of using node n for routing from s_i to t_{ij} as

$$C_n = A_{ij}\, d_n + (\, 1 - A_{ij}\,)\, c_n \qquad (2)$$

where c_n is defined in eq. (1) and A_{ij} is the slack ratio

$$Aij = Dij\, /\, D_{max} \qquad (3)$$

where D_{ij} is the longest path containing the arc $(\, s_i,\, t_{ij}\,)$, and D_{max} is the maximum over all paths, i.e. the critical path delay. Thus, $0 < A_{ij}$ 1.

The first term of equation (2) is the delay-sensitive term, while the second term is congestion-based. Equations (2) and (3) are the keys to providing the appropriate mix of minimum-cost and minimum-delay trees. If a particular source-sink pair lies on the critical path, then $A_{ij} = 1$, and the cost it sees for using node n is simply the delay term; hence a minimum-delay route will be used, and

congestion will be ignored. If a source-sink pair belongs only to a path whose delay is much smaller than the critical path, its A_{ij} will be small, and the congestion term will dominate, resulting in a route which avoids congestion at the expense of extra delay.

In Appendix 1 we show that if h_n is bounded by d_n, then eq. (2) guarantees a worst case path delay equal to the minimum delay route of the critical path. That is, in this situation the algorithm achieves the fastest implementation allowed by the placement. In practice, h_n is allowed to increase gradually until a complete route is found. For very congested circuits, h_n will exceed d_n, but as we show experimentally in Section 6, the algorithm comes very close to this bound in practice.

A_{ij} 1 for all signals i and sinks j [1]
While shared resources exist (global router) [2]
    Loop over all signals i (signal router) [3]
        Rip up routing tree RT_i [4]
        RT_i s_i [5]
        Loop over all sinks t_{ij} in decreasing A_{ij} order [6]
            PQ RT_i at costs $A_{ij}d_n$ for each node n in RT_i [7]
            Loop until t_{ij} is found [8]
                Delete lowest cost node m from PQ [9]
                Loop over fanouts n of node m [10]
                    Add n to PQ at cost $A_{ij}\,d_n + (1 - A_{ij})\,c_n + P_{im}$ [11]
                End [12]
            End [13]
            Loop over nodes n in path t_{ij} to s_i (backtrace) [14]
                Update c_n [15]
                Add n to RT_i [16]
            End [17]
        End [18]
    End [19]
    Calculate path delays and A_{ij}'s (eq. 3) [20]
End [21]

Figure 5. Congestion/Delay routing algorithm

Figure 5 contains the details of the delay-based routing algorithm. The A_{ij}'s are initialized to 1 (step 1). Thus during the first iteration the global router finds the minimum-delay route for every signal. In a manner similar to the congestion-based router, a priority queue implementation of a breadth first search is performed to find sinks. For nodes already in RT_i the cost is just the delay term (step 7); for all others it is the sum of the delay and congestion terms (step 11) as given earlier in equation (2). The net effect is that nodes that are already in the (partial) routing tree will not have a congestion component. Having already been allocated to signal i they are “free” from a congestion point of view.

Sinks are routed in order of decreasing A_{ij} (step 6). Intuitively, sinks with the highest slack ratios (and thus the most time-critical) should have the most importance in determining the tree structure,

while sinks with a low slack ratio will have more flexibility. While this heuristic works well in practice, one can come up with examples where sinks with lower A_{ij} should be routed first because of their proximity to good paths to sinks with higher A_{ij}. The general problem of finding a minimum spanning tree subject to delay constraints is a difficult problem. Finding the true minimum spanning tree, given a set of delay constraints and congestion values, is not crucial to this algorithm, just as finding the optimum Steiner points is not crucial to the congestion-based algorithm. If we "encourage" the routing tree to meet the delay constraints through the use of the A_{ij} s, the global router will attempt to adjust the congestion values to converge on a routing tree that has no shared nodes.

At the end of each iteration (step 20) the path delays and A_{ij} are recalculated. The global router completes when no more shared resources exist. Note that by recalculating the A_{ij} each iteration, we keep a tight reign on the critical path. Over the course of iterations, the critical path increases only to the extent required to resolve congestion. This approach is fundamentally different from other schemes ([Brown92], [Frankle92]) which attempt to resolve congestion first, then reduce delay by rerouting critical nets.

4.4 Enhancements

Several enhancements can increase the speed of the algorithm without adversely affecting the quality of the route. One enhancement is to introduce the A* algorithm into the breadth-first search loop. A* uses lower bounds on path lengths to bound the breadth-first search. A* can be applied to the congestion/delay router by tabulating the cost of minimum-delay routes from every node to all the potential sinks. We modify line [11] of figure 5, so that fanout n of node m is added to the priority queue PQ at cost

$$A_{ij}\, d_n + (1 - A_{ij})\, c_n + P_{im} + D_{nj} \qquad (4)$$

where D_{nj} is the cost of the minimum-delay route from n to sink j. To make things clearer, we expand c_n in equation (2), and use the intrinsic node delay d_n for the base cost b_n, to yield

$$A_{ij}\, d_n + (1 - A_{ij})\ (d_n + h_n) * p_n + P_{im} + D_{nj} \qquad (5)$$

Since p_n 1, the first two terms sum to at least d_n. The cost to reach node m, P_{im}, is at least D_{im}. Thus, D_{ij} (= d_n.+ D_{im} + D_{nj}) is a lower bound to the queue entry and thus the total cost of the route. During the first iteration congestion is ignored, and D_{ij} is the total cost of the route. The net result is that during the first iteration the search reaches only those nodes on minimum-delay paths. By careful scheduling the search can be made linear in the number of nodes along a minimum-delay path. As iterations progress, increasing p_n and h_n cause this lower bound to prove less and less

useful, and the search expands. As a result, the number of events introduced onto the priority queue (and the CPU time required) grows, but still remains less than a full breadth-first search.

Another enhancement is to route only the signals involved in congested (i.e. shared) nodes. This is a significant departure from Nair's original algorithm. If one examines the routing problems in figures 2 and 3 with this modification, the same result is obtained as if one had rerouted all signals during every iteration. To date we have not seen any cases where rerouting only signals involved in congested nodes resulted in a lower quality route. While the number of iterations increases, the total running time decreases.

5 Montage Software

In order to apply the mapping tools described in Sections 2 and 3 to Montage, some extensions are necessary. Specifically, the placement and routing tools must ensure isochronic fork constraints on some wires, and logic and arbiter functions must be placed into logic and arbiter RLBs respectively. The latter constraint is easily accomplished by correctly placing the logic and arbiter functions into RLBs during placement initialization, and from then on only considering annealing moves between two logic RLBs, or two arbiter RLBs, but never between a logic and an arbiter RLB.

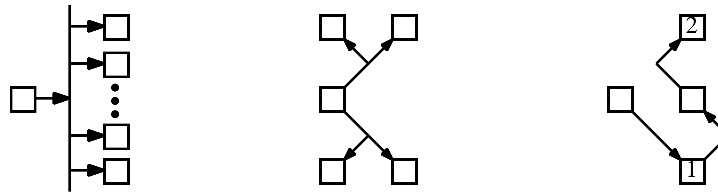

Figure 6. Placement of a symmetric isochronic fork on an interconnect line (left) and on diagonals (center), as well as an asymmetric fork (right) reaching destination "1" before destination "2".

A more difficult requirement is for the placement and routing tools to ensure the isochronic constraints. An isochronic fork is a multi-destination wire where the circuit requires that a transition on the wire reach all destinations simultaneously (a symmetric fork), or reach one end before another (an asymmetric fork). For the placer, we require that all destinations of a symmetric isochronic fork be placed such that the constraint can be met. Specifically, the destinations must be able to share a single vertical segmented channel, or the diagonals from a shared neighbor RLB. In order to incorporate this requirement into the annealer's cost function, we could simply add a penalty for all isochronic forks that do not meet this constraint. Unfortunately, forks with large numbers of destinations will rarely happen to line up as a proper isochronic fork, and the annealer has little chance of meeting all the constraints. Our solution is to extend the fork penalty to recognize when a

fork constraint is getting close to being met. Specifically, the penalty is decreased when two or more terminals are positioned such that the constraint can be met between those pins, with larger locally correct groups decreasing the penalty even more. In this way the annealer is encouraged to get closer and closer to a proper placement, while the graduated cost metric also allows it to try different fork positionings. In practice, fork constraints are almost always met, and do not greatly impact circuit routability and density.

Routing of symmetric isochronic forks also requires special handling. Specifically, we cannot simply attempt to reach each fork destination individually, since they may take paths inconsistent with the isochronic assumption. Instead, we check the placement of the destinations of an isochronic fork and determine all valid fork points. For example, if the isochronic fork ran to exactly two different destinations, and they were 1 cell apart in the same column, the fork point could either be on a shared vertical segmented channel, or either of the two RLBs that are direct neighbors of these two destinations. Then, instead of routing to the function blocks of the destinations, we instead route to these fork points. Once a fork point is reached, we calculate the cost for routing from this fork point to each of the destinations of the fork, add that to the cost of the current route up to this point, and insert it back into the queue. When we finally reach one of these complete routes in the queue we know that this is the preferred route, and accept it. In this way we can directly extend all of the work on delay optimization and congestion avoidance to isochronic fork routing without any extra special-casing.

Placing asymmetric isochronic forks, forks where one destination must be reached before another, simply requires that the distance metric be extended to properly reflect the resulting routing. Since we will route the signal through the earlier destination and then on to the later destination, we simply treat this segment between terminals as a separate signal, with the earlier end as the new signal's source, and the later end as the sink. A similar extension works for the router, where the only addition necessary is to ensure that the new signal is routed not from the function block output of the earlier destination, but instead from wherever the signal enters the earlier destination's RLB.

6 Results

In this section we present the results of mapping circuits from the PREP FPGA benchmarks [PREP92] and the logic synthesis benchmarks [ILSW93] to the Triptych architecture. All circuits were synthesized and technology-mapped using SIS [SIS92] and covering was performed using the table-lookup (TLU) architecture primitives in SIS. The 8x64 Triptych array (8 columns, 64 rows) was used for all the experiments. There are 8 vertical, unidirectional, segmented buses. The RLBs used are the 4-input, 4-output variety described in Section 3.1 of the companion Triptych architecture

paper. These RLBs have two diagonal and two vertical bus inputs, three of which are chosen as function inputs.

Results for the PREP benchmarks are shown in table 1. *Reps* is the maximum number of repetitions of the benchmark circuit that will fit into the 8x64 array. This number was determined by performing a congestion-based placement and a congestion-based route on a range of repetitions to determine a maximum. This insures a dense circuit, and is therefore a good test to determine how well the router can optimize for delay when congestion is high. *RLBs* denotes how many RLBs are used solely for logic. *Logic Levels* is the number of logic levels on the critical path. For sequential circuits the critical path is defined as the longest path (in terms of logic levels) between any two flip-flops (corresponding to the "internal frequency" measurement defined by PREP). For combinational circuits the critical path is defined as the longest path between an input and an output pad.

Bench #	Reps	RLBs	Logic Levels	Mapped Delay	Place Time	Placed Delay	M-P Delay Increase	Route Iterations	Route Time	Routed Delay	P-R Delay Increase
1	10	248	2	14.0	27.6	23.3	66%	78	67.5	23.3	0.0%
2	4	223	6	40.0	11.6	57.3	38%	28	19.5	57.3	0.0%
3	7	252	7	46.5	10.7	65.1	40%	52	19.9	66.6	2.3%
4	2	193	14	92.0	10.6	123.3	33%	52	34.6	125.3	1.6%
5	4	244	16	105.0	11.5	150.2	43%	46	25.6	155.9	3.8%
6	8	248	16	105.0	7.8	134.2	28%	108	36.5	138.0	2.8%
7	5	275	10	66.0	16.0	92.8	41%	65	46.1	97.2	4.7%
9	5	234	7	46.5	13.3	61.6	33%	31	20.4	62.6	1.6%

Table 1. Results of the PREP benchmarks on an 8x64 Triptych array

Mapped Delay is the minimum delay after covering, which is the delay obtained by assuming every block on the critical path can be reached via a single diagonal from the previous block. Note that this delay may in fact not be feasible since circuits usually have multiple critical paths, and nearest neighbor connections for all of these signals may not be possible.

Place Time is the time taken to perform placement (in CPU minutes on a 90MHz HyperSPARC processor). *Placed Delay* is the minimum delay after placement, which is the delay of a minimum delay route for every signal on the longest path. This number is the best that the router can possibly do with a given placement. *M-R Delay Increase* is the % dilation of the critical path in going from covering to placement. *Route Time* is the time to perform a complete route and *Route Iterations* is the number of iterations the router took to reach a successful routing using the delay-based router described earlier. *Routed Delay* is longest delay path after routing. *P-R Delay Increase* is the % dilation of the critical path in going from placement to routing.

The first point to notice is that the delay-based placement and delay-based route achieved the same density as the congestion-based placement and route. In other words, introducing delay into both the placement and routing cost functions still allowed routeable circuits. Also, a number of benchmarks

achieve approximately 50% RLB utilization, and those which achieved significantly less than 50% (e.g. #4) would have required significantly more than 50% utilization to fit another repetition into the array.

The average increase of the critical path in going from covering to placement is 40%, while the average increase of the critical path in going from placement to routing is only 2.1%. It is apparent that the router is able to achieve close to the minimum delay allowed by the placement. The delay performance of the placement program, on the other hand, is much less than optimal and improvements to the placement algorithm may reap significant improvements.

Bench #	RLBs	Logic Levels	Total Fanouts	Mapped Delay	Place Time	Placed Delay	M-P Delay Increase	Route Iterations	Route Time	Routed Delay	P-R Delay Increase
ex1(S)	150	8	425	53.1	10.3	76.1	44%	36	41.5	79.7	4.7%
keyb(S)	150	10	417	66.0	12.8	95.4	45%	28	34.3	102.9	7.8%
C880(C)	152	14	420	100.3	7.6	153.8	31%	8	12.1	159.4	3.6%
clip(C)	155	11	440	80.8	13.6	126.0	56%	9	14.1	131.0	3.9%
C1908(C)	159	15	436	106.8	11.3	161.9	52%	16	18.8	171.3	5.8%
mm9b(S)	163	14	428	85.5	16.2	128.2	50%	28	29.1	128.3	0.0%
bw(C)	169	7	472	54.7	12.3	89.6	64%	11	19.8	98.1	9.4%
s832(S)	173	11	487	72.5	11.7	117.0	62%	25	32.1	118.1	1.0%
s820(S)	176	10	488	66.0	10.4	112.8	71%	35	47.0	114.7	1.7%
x1(C)	192	7	522	54.7	11.1	84.1	54%	12	18.8	94.7	12.4%
s953(S)	220	10	627	66.0	15.0	101.7	54%	63	60.2	103.8	2.1%
s1423(S)	235	30	707	196.0	13.9	265.2	35%	38	34.1	271.0	2.1%
s1(S)	246	13	694	85.5	16.6	129.5	51%	(did not converge)			
styr(S)	295	12	830	79.0	20.2	120.0	52%	(did not converge)			
planet(S)	296	12	845	79.0	13.9	109.0	38%	(did not converge)			
planet1(S)	297	12	848	79.0	20.6	107.3	36%	(did not converge)			

Table 2. Results of mapping selected circuits from the benchmark suite of the 1993 International Logic Synthesis Workshop.

Table 2 shows the results of mapping circuits chosen from the benchmark suite of the 1993 International Logic Synthesis Workshop (ILSW). Circuits were chosen if they mapped into 150-300 RLBs and if the number of I/Os did not exceed the number available on an 8x64 array (148). The maximum density obtainable using these benchmarks is slightly less than 50%. The PREP Benchmarks obtain somewhat better density, because they are replications of smaller circuits and therefore are more local.

These results suggest that the upper limit on the number of RLBs used for logic is approximately 250, or about 50% of the total available (512). The average increase of the critical path in going from covering to placement is 51%. The average increase of the critical path in going from placement to routing is 4.5%. Both these figures are somewhat larger than for the PREP Benchmarks, due again to the decreased locality of the ILSW benchmarks.

7 Summary and Future Work

In this paper we have presented placement and routing tools that take advantage of the Triptych architecture. Placement and routing is particularly difficult for this architecture because of the limited routing resources available. The results show that we can achieve approximately 50% utilization of RLBs across a range of benchmark circuits. That is, about half the RLBs are allocated to logic functions and the remainder to routing. Although hand-placement and routing of regular circuits and small control circuits achieved a better utilization, this is to be expected, The Triptych architecture was designed with the assumption that some RLBs would be used for routing only, or possibly left empty for routability. The comparisons in our companion paper show, for example, that with 50% logic utilization, Triptych still achieves about twice the density of the Xilinx 3000 architecture. Moreover, utilization of large FPGAs is often substantially less than 100%.

We have also presented the modifications required to adapt these placement and routing tools for implementing asynchronous circuits on Montage. These algorithms have implications beyond that of this single architecture. Asynchronous circuits are of great interest but present constraints beyond that of standard synchronous circuits. By developing automatic tools to support the placement and routing of asynchronous circuits, we demonstrate that supporting asynchronous design is feasible, and the techniques can be applied to other FPGA architectures as well.

Our future work will focus on improving the Triptych placement algorithm to increase both logic density and reduce circuit delay. Both the local-routability and density-smoothing can be improved by using a different approach that combines the two measurements into a single component that avoids the problems with both. For example, a window of six RLBs could be examined -- a pair of "input" RLBs and their four common "output" RLBs. A *probability* of local routing success could be computed based on the number of signals required by the input RLBs, the number of input signals immediately available at the output RLBs, and the number of route-throughs available. Assigning a probability avoids the problem associated with the current step function form of the local routability component. Formulated in this way, density smoothing is performed based on local need rather than some global estimated requirement. The probabilities could easily be precomputed from the specifics of the architecture and accessed very quickly at run time. This would allow the cost function to be retargeted automatically to a new architecture.

The Triptych architecture is an example where significant gains can be achieved by taking advantage of improved place and route tools that utilize routing resources more efficiently. While the tools we have described here are specific in some respects to the Triptych architecture, we expect the ideas presented here to be applicable in different forms to other FPGA architectures. Indeed we have already retargeted the routing algorithm to other FPGAs with very promising results. The challenge

remains for FPGA architects and tool designers to work together in the future to produce the best overall solutions.

Acknowledgments

Thanks to Darren Cronquist, David Hubbell, David Song, and Elizabeth Walkup, who helped in the development of mapping strategies for the Triptych FPGA. This research was funded in part by the Defense Advanced Research Projects Agency under Contract N00014-J-91-4041. Carl Ebeling was supported in part by an NSF Presidential Young Investigator Award. Steven Burns was supported in part by an NSF Young Investigator Award.

References

[Algotronix91] Algotronix Limited, “CAL1024 Preliminary Datasheet”, 1991.

[Beetem91] J. Beetem, "Simultaneous Placement and Routing of the LABYRINTH Reconfigurable Logic Array", International Workshop on Field-Programmable Logic and Applications, Oxford, 1991, pp. 232-243.

[Brown92] S. Brown, J. Rose, Z. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 5, May 1992, pp. 620-628.

[Cohn91] J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing," *IEEE Journal of Solid-State Circuits*, vol. 26, March 1991, pp. 330-342.

[Frankle92] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," in *Proc. 29h Design Automation Conference*, June 1992, pp. 536-542.

[Hauck95] S. Hauck ...., “The Triptych FPGA Architecture,” *IEEE Transactions on Computer-Adied Design,* current issue.

[Hill91] D. Hill, "A CAD System for the Design of Field Programmable Gate Arrays," in *Proc. 28t Design Automation Conference*, June 1991, pp. 187-192.

[ILSW93], "LGSynth93 Benchmark Set," 1993 International Logic Synthesis Workshop, May 1993.

[Kuh86] E. Kuh, M. Marek-Sadowska, "Global Routing," in *Layout Design and Verification*, T. Ohtsuki, Ed., Elsevier Science Publishers B. V. (North-Holland), 1986, pp. 169-198.

[Lee61] C. Lee, "An Algorithm for Path Connections and its Applications," *IRE Trans. Electron. Comput.*, vol. EC-10, 1961, pp. 346-365.

[Linsker84] R. Linsker, "An Iterative-Improvement Penalty-Function-Driven Wire Routing System," *IBM Journal of Research and Development*, vol. 28, Sept. 1984, pp. 613-624.

[McMurchie94] L. McMurchie and C. Ebeling, "WireC Tutorial and Reference Manual," UW Dept. of CS&E TR# 94-09-09, Sept. 1994.

[Nair87] R. Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, March 1987, pp. 165-172.

[Palczewski92] M. Palczewski, "Plane Parallel A* Maze Router and its Application to FPGAs," in *Proc. 29th Design Automation Conference*, June 1992, pp 691-697.

[PREP92] "PREP PLD Benchmark Suite #1," Programmable Electronics Performance Corp., Oct. 1992.

[Sechen87] C. Sechen, K. Lee, "An Improved Simulated Annealing Algorithm for Row-Based Placement," Proc. IEEE International Conference on Computer Aided Design, Nov. 1987, pp. 478-481.

[SIS92] Sentovich et al., "SIS: A System for Sequential Circuit Synthesis," Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, Dept. of EE & CS, Univ. of California, Berkeley, CA, May 1992.

[Togawa94] N. Togawa, M. Sato, T. Ohtuski, "A Simultaneous Placement and Global Routing Algorithm for Field-Programmable Gate Arrays," presented at FPGA94, Berkeley, 1994.

[Walkup92] E. Walkup, S. Hauck, G. Borriello, C. Ebeling, "Routing-directed Placement for the Triptych FPGA," Proceedings of FPGA92, Berkeley, CA, 1992.

[Xilinx93] Xilinx, Inc., "XACT Development System Reference Guide," Jan. 1993.

Appendix 1. Proof of Delay Bound

Theorem: If $h_n <= d_n$ for all nodes, then the delay of any signal path routed by the congestion/delay algorithm is bounded by D_{max}, the delay of the longest minimum-delay path in the circuit.

Proof sketch: When the algorithm terminates successfully, the p_n term in eq. (1) is 1 and can be ignored. Let *R* be the most critical routed path and *S* be the shortest delay route for *R*. The cost of *S* is given by:

$$
\begin{aligned}
C_S &= \sum_{n \in S} c_n \\
&= \sum_{n \in S} (A_{ij} d_n + (1 - A_{ij})(d_n + h_n)) \\
&= \sum_{n \in S} d_n + \sum_{n \in S} (1 - A_{ij}) h_n
\end{aligned}
$$

Since $h_n \le d_n$,

$$
\begin{aligned}
C_S &\le D_{ij} + \sum_{n \in S} (1 - A_{ij}) d_n \\
&\le D_{ij} + (1 - A_{ij}) D_{ij} \\
&\le D_{ij} + (D_{MAX} - D_{ij}) D_{ij} / D_{MAX}
\end{aligned}
$$

Since $D_{ij} / D_{MAX} \le 1$,

$$
\begin{aligned}
C_S &\le D_{ij} + D_{MAX} - D_{ij} \\
&\le D_{MAX}
\end{aligned}
$$

The cost of *R* must be less than the cost of *S*, thus the delay of *R* must be less than the cost of *S* which is less than D_{max}.