The SFRA: A Corner-Turn FPGA Architecture

Nicholas Weaver*
CS Department
UC Berkeley
Berkeley, CA
nweaver@icsi.berkeley.edu

John Hauser
CS Department
UC Berkeley
Berkeley, CA
jhauser@jhauser.us

John Wawrzynek
CS Department
UC Berkeley
Berkeley, CA
johnw@cs.berkeley.edu

ABSTRACT

FPGAs normally operate at whatever clock rate is appropriate for the loaded configuration. When FPGAs are used as computational devices in a larger system, however, it is better to employ fixed-frequency FPGAs operating at a high clock frequency. Such fixed-frequency arrays require pipelined interconnect structures, which are difficult to support in a traditional FPGA architecture. We have developed a novel approach, called a "corner-turn" interconnect, based on a Manhattan array of logically depopulated S-boxes with full connectivity but limited routability. This interconnect supports new polynomial-time routing techniques while maintaining conventional placement and other upstream toolflow. We have used the corner-turn interconnect to define a fixed-frequency FPGA architecture, the SFRA, that is largely compatible with the Xilinx Virtex while providing higher speed, pipelined operation. Our tools automatically repipeline designs to operate at the SFRA's intrinsic clock frequency. Since the arrays are largely compatible, we directly compare the SFRA with the Virtex on four benchmark designs. On these benchmarks, the SFRA offers higher throughput and competitive throughput per area. The SFRA routing and retiming tools also run one to two orders of magnitude faster than their Xilinx counterparts.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architectures—*Field Programmable Gate Arrays*; C.4 [**Processor Architectures**]: Design Study

General Terms

Performance, Design

*Nicholas Weaver is now at the International Computer Science Institute

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'04, February 22-24, 2004, Monterey, California, USA.
Copyright 2004 ACM 1-58113-829-6/04/0002 ...$5.00.

Keywords

FPGA Architecture, FPGA CAD, FPGA Optimization, FPGA Design Study

1. INTRODUCTION

Conventional FPGAs do not pre-determine the clock frequency at which configurations will run but rather support arbitrary clock signals up to some frequency limits. Often, the operating clock frequency is set to the maximum that a configuration allows, as determined when the design is mapped to the target FPGA. When FPGAs interface with a traditional microprocessor or other pieces of a computational system, the FPGA clock (or clocks) must be made compatible with the processor clock. In such cases, it is preferable to construct a *fixed-frequency* FPGA, which operates at a set clock rate regardless of the configuration mapped onto it.

Besides being easier to integrate, fixed-frequency FPGAs have the advantage that they can be designed to operate at a higher clock rate than is possible for normal FPGAs. Thus any computation that can be pipelined will run considerably faster on a fixed-frequency FPGA.

As interconnect times usually dominate in conventional FPGAs, fixed-frequency FPGAs generally require pipelined interconnect and routing structures. However, previous fixed-frequency arrays either introduced difficult placement problems or contained highly restrictive interconnect topologies which limited their applicability.

In order to develop fixed-frequency arrays that are compatible with conventional placement and synthesis techniques, we need routing structures where the interconnect and switches contain pipeline registers. But simply adding pipelined switches to a conventional FPGA would be impractical, as this would require far too many registers or, if not all switches include registers, potentially complicate the routing problem.

Thus we desire a switch structure that maintains the conventional Manhattan placement common to conventional FPGAs, yet which can be efficiently pipelined. We developed such an interconnect topology, which we call a "corner-turn" interconnect, based on capacity-depopulated crossbars. This interconnect supports efficient pipeline switches, fast routing algorithms, and defect tolerance, and is particularly well suited to fixed-frequency applications and coarser-grained FPGA architectures.

Using this interconnect design, we have defined a proto-

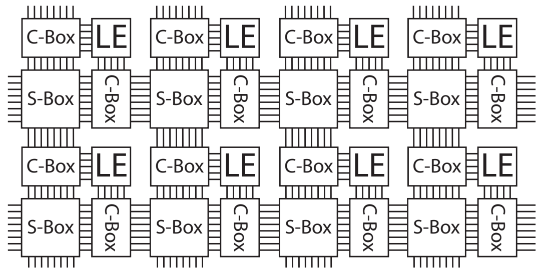

Figure 1: The classical components of a Manhattan FPGA: the LE (logic element) performs the computation, the C-box (connection box) connects the LE to the interconnect, and the S-box (switch box) provides for connections within the interconnect.

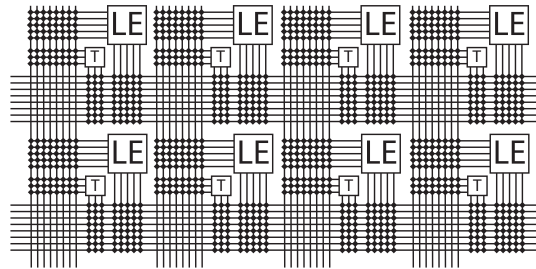

Figure 3: Our corner-turn FPGA interconnect. To simplify matters for our purposes, the C-boxes are implemented as full crossbars, while the S-boxes use capacity-depopulated crossbars to create corner turns, marked as T-boxes on this illustration.

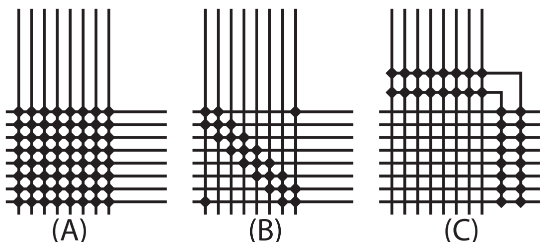

Figure 2: (A) A crossbar switch-box. (B) A physically depopulated crossbar. (C) A capacity-depopulated crossbar.

type fixed-frequency FPGA architecture, the SFRA,[1] which has CLBs that are nearly identical to the Xilinx Virtex [16] but with the new registered routing interconnect. Thus, except for routing and retiming, the SFRA is largely design- and tool-compatible with the Xilinx Virtex. Like other fixed-frequency architectures, all designs must be either repipelined or C-slow retimed to match the FPGA's delay model. Unlike previous fixed-frequency architectures, the SFRA architecture works with the conventional Xilinx synthesis, mapping, and placement tools.

We have created a complete architecture and toolflow for the SFRA, as well as layout for portions of the interconnect. Our resulting array is of comparable size to previous fixed-frequency arrays, and the toolflow for routing and retiming is an order of magnitude faster than the Xilinx routing tool, requiring slightly more than a minute to route and retime a large benchmark. Additional details about this work are available [14].

2. THE CORNER-TURN TOPOLOGY

Manhattan-structured FPGAs, as commonly visualized in Figure 1, consist of 3 main pieces: logic elements (LEs) that perform the actual computation, connection boxes (C-boxes) which connect the logic elements to the general interconnect, and switch boxes (S-boxes) used to route signal in the general interconnect. Parameters of the architecture include the number of signals in each channel (the channel capacity), and the number of bits routed as a unit (the bitwidth or granularity).

If routability was the only criteria, the C-boxes and S-boxes would both be full crossbars, as in Figure 2(A), enabling the C-boxes to use any wire as an input or output and the S-boxes to connect any input to output. Unfortunately, crossbars, especially for the S-boxes, are usually too expensive as they require $O(N^2)$ switches to implement for a channel capacity of N bits.

The common solution, as shown in Figure 2(B) is to create a physically depopulated crossbar. Such crossbars are constructed by removing switches until a desired balance between cost and routability is achieved. There are several methods for constructing these depopulations, ranging from the classic diagonal pattern, ad-hoc removal of switches, to analytic techniques based on multiple routing trials [9]. These depopulations usually enable every wire to be routed between the two channel, but with limited connectivity between the channels. Thus each particular signal can only be routed to particular destination wires.

Most conventional FPGAs use physically depopulated crossbars to implement both the C-boxes and S-boxes. Unfortunately, such structures are not very suitable for creating a fixed-frequency FPGA. If most or all of the connections in the S-boxes are registered, this would require a prohibitive number of registers. If only a small subset of the S-box connections were registered, this would complicate an already difficult routing problem.

An alternate approach is to create a capacity-depopulated crossbar, as in Figure 2(C). The desire is to maintain the any-to-any connectivity of a crossbar, while sacrificing the capacity that can be routed. Thus instead of attempting to depopulate connectivity, this strategy depopulates capacity. In this approach, a limited number of total signals can be routed between the two channels.

By using these capacity-depopulated switchpoints in a Manhattan FPGA, we create what we call a "corner-turn" interconnect, as seen in Figure 3. The corner-turn name comes from the observation that this interconnect form

[1] "SFRA" means either "Synchronous and Flexible Reconfigurable Array" (based on its fixed-frequency routing architecture) or "San Francisco Reconfigurable Array" (based on the numerous "No Left Turn" signs in the city of San Francisco).

prefers straight lines, with each "turn" from horizontal to vertical or vertical to horizontal being a very limited resource. In the construction we studied, the C-boxes are implemented as full crossbars, with each input and output connected to both the associated horizontal and vertical channels. The S-boxes are replaced with T-boxes, capacity-depopulated crossbars with the capacity described as the number of turns supported. Each turn is able to route a signal from the horizontal to vertical channel direction and a separate signal from the vertical to horizontal channel.

For simplicity, we assume that the C-boxes are full crossbars. This assumption makes it easy to perform fast detailed routing at the cost of some efficiency. As discussed later, we are confident the C-boxes could also be depopulated without sacrificing fast detailed routing, but we have deferred this issue to later research.

Unlike conventional switchpoints, there are several advantages to a corner-turn topology that result from the use of reduced-capacity S-boxes instead of physically depopulated S-boxes:

Fast Routing: As detailed in Section 4, both global and detailed routing use fast, polynomial-time heuristics. Detailed routing uses known techniques, while global routing uses new algorithms. No step is worse than $O(N^2)$ in the worst case.

Pipelined Switchpoints: If a conventional crossbar or depopulated crossbar were to use pipelined switchpoints, it would require numerous pipeline registers. A corner-turn switchpoint, however, requires only two registers for each turn, a substantial savings. This savings is particularly useful given our goal to implement a fixed-frequency FPGA.

Potential Defect Tolerance: Any resource that can be exchanged for another offers the potential for defect tolerance. It is obvious that with an additional turn and spare wires, stuck-off failures or breaks in the T-boxes and routing channels can be overcome by simply remapping the wires in question from a defective connection to a working connection.

The problem is how to route a signal onto a stuck-on routing resource when that fault would otherwise cause interferance. If the interconnect uses a classical braiding pattern, many stuck-on switchpoints in the C-boxes can not be avoided, as exploiting the stuck-on point will require using a wire with registers in different locations. However, if all wires in a channel have breaks, rebuffering, or registers at the same locations, then a large number of stuck-on and stuck-off faults in the C-boxes can be routed around by transposing connections.

Asymmetric Routing Channels: In Hallschmid [6] and elsewhere, it has been argued that asymmetric routing channels, where the horizontal and vertical capacities are significantly different, may have significant advantages. Although Betz [1] argues this is not the case for rectangualre arrays, hotspots will occur in non-rectangular arrays which are embedded in larger systems. As a related observation, if a design following Rent's Rule is mapped to a Manhattan FPGA using common placement techniques, more interconnect will be required for larger FPGAs.

Because the corner-turn switches maintain any-to-any connectivity, it is easy to construct and route arrays where the routing channels closer to the periphery have fewer wires, or where the channel size is increased for larger arrays. Such arrays do not affect the routability beyond the capacity limits, as each channel is detail-routed independently. As the bulk of any FPGA lies in the wiring and switching, this strategy offers potentially large area savings.

Finally, as seen in Section 7, our bit-oriented FPGA based on the corner turn topology is substantially larger than commercial FPGAs although competitive with previous fixed-frequency FPGAs. This disadvantage disappears when the array's basic operations occur on larger quantities. Thus for arrays that operate on 8-bit or 16-bit bit quantities such as those used by Chameleon [2], Matrix [10] and others, a corner-turn interconnect may be desirable even when pipelined interconnect is not needed, as the interconnect still supports fast routing and potential defect tolerance.

The placement and upstream tool problems are essentially the same for both conventional Manhattan FPGAs and corner-turn FPGAs, although some minor changes in the placement and synthesis cost functions are desirable as slight misalignments in datapath elements will have a more significant effect on routing.

3. OTHER FIXED-FREQUENCY FPGAS

There have been several previous fixed-frequency arrays. In general, these arrays either have operated on restricted designs or have imposed significant new tool problems. The earliest examples include Garp [7], which coupled a fixed-frequency reconfigurable functional unit with a microprocessor. Garp's interconnect fabric supported only limited connectivity to maintain the target clock cycle. The RaPiD [5] architecture provided a coarse-grained array of functional units in a general interconnect. All long signals were explicitly pipelined to support high clock speeds. Similarly, PipeRench [4] was limited to largely feed-forward designs.

The most general fixed-frequency array has been the HSRA [13]. This array had a pipelined H-tree for the routing structure, ensuring that all designs would run at the target clock frequency. The HSRA included retiming chains, programmable-delay shift registers, on all inputs to balance interconnect delays.

A significant limitation of the H-tree structure is that it introduces a new placement problem. Instead of regular Manhattan placement, the HSRA structure relies on recursive bipartitioning. Although less of a concern for random logic, this limitation can be significant when designs use datapath synthesis techniques.

4. ROUTING FOR CORNER-TURN NETWORKS

We perform routing for the corner-turn FPGA in two stages, a global routing step, which assigns signals to particular channels and turns, and a detailed routing step, which performs wire assignment within each channel. Since each channel is independent, for detailed routing we use the well-known greedy channel-packing technique, an $O(N \log N)$ operation, with N being the number of signals within each channel.

Global routing uses new techniques developed for this style of interconnect. As only a limited number of turns are available, global routing attempts to minimize the number of turns taken by each signal. Not coincidentally, ff every signal uses only a single turn, this also minimizes the delay of all signals as they traverse the minimum distance and

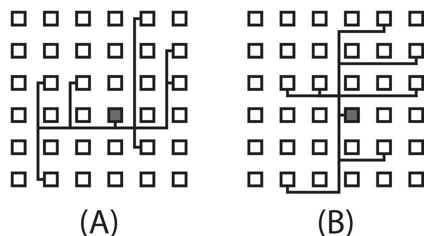

Figure 4: The two alternatives considered by fanout routing a signal from the dark square to several destinations: (A) start horizontal and branch vertically; (B) start vertically and branch horizontally.

pass through the minimum number of switches necessary to route the design.

Our global routing occurs in five distinct passes: 1) direct routing of any signal which does not require switching; 2) routing nets with high fanout to share switches; 3) "pushrouting" phase which attempts to route all remaining nets using a single turn; 4) zig-zag routing phase which attempts to discover two-turn routes; and 5) probabilistic rip-up-and-reroute cleanup phase which attempts to rip-up previously routed nets to route new nets.

Direct routing, the first phase, selects all nets which can be routed without using a turn. These nets are assigned to the appropriate channels and removed from further consideration. This step is linear in the number of signals to be routed.

The second phase, fanout routing, attempts to route large fanout nets and maximize the sharing of turns. Since later phases route all point-to-point connections independently, this phase is necessary to enable sharing of limited turn resources between destinations. This phase begins by first sorting all remaining nets by their degree of fanout. Then, starting with the highest fanout net, all nets of fanout greater than four are routed.[2]

Two possible routes are considered: starting horizontally with vertical fanout and starting vertically with horizontal fanout (Figure 4). At each switchpoint, a limited number of turns are available for this routing step to preserve some freedom for the later phases of routing. If only one route is available, that route is selected. If both routes are available, the route that uses the least number of turns is selected. If no route is available, the net is routed by later steps. Any routes selected in this phase are locked down and unperturbed by later routing steps. Due to the initial sort, this requires $O(N \log N)$ time, with N being the number of signals remaining.

After fanout routing, the bulk of the routing uses a technique we call "pushrouting" that considers all nets as individual point-to-point signals. All unrouted nets are first sorted to give a higer priority to short nets (which become less routable in later phases) and nets on the critical path. Pushrouting observes that for every net, there are only two possible one-turn routes, making it feasible to perform a comprehensive search to see if a route for a group of nets exist.

For each net, if one of the possible two routes has a free turn, the net is assigned to the available turn. Otherwise, a depth-first search is conducted, beginning at the two possible turns, to determine if a series of adjustments exists that can provide a free location for the net.

Due to the limited freedom of routing, pushrouting is comprehensive if it is the only technique employed. However, because high fanout nets are locked, there may be some minor variation in routability due to the order that nets are routed. Since this process performs a depth-first search for each net routed, it is possible to require visiting all previously routed nets in the process, making the maximum running time of this step $O(N^2)$, with N representing the number of signals remaining to route.

After pushrouting, the remaining nets are "zig-zag" routed. Each net is examined to see if there exists a two-turn route for it. As longer nets have more possible routes, the previous pushrouting phase attempts to route short nets first, ensuring that any remaining nets are more easily routed during this phase. This phase is linear in the number of nets, although the number of turns that need to be examined depends on the average length of each net.

A final phase is a probabilistic rip-up-and-reroute step. This step is the most costly for each net, although it is usually limited in scope. This process iterates over the set of unrouted nets until no further progress can be made. For each net still not routed, it is first determined whether it can be pushrouted or zig-zag routed. If not, the two possible switches involved in a one-turn route are examined.

All possible nets that could be unrouted to enable this net to route are examined by a breath first search. One of these nets is probabilistically selected, with preference given to long nets that are more likely to be zig-zag routable. For each net, this process may require $O(N)$ steps in the worst case, with N being the number of signals in the design, but in practice usually considerably less. As this technique is only suitable for routing the last few nets, the number of iterations is capped.

In experimenting with the number of hardware turns as a parameter, we have observed that the fanout and pushrouting are the most successful phases. Although there are some cases where the zig-zag and probabilistic rip-up steps can route the last few nets, they are far less successful at finding routes because these steps require more turns per signal to complete.

5. FIXED-FREQUENCY RETIMING

Any fixed-frequency FPGA must either restrict the user's designs to meet the array's pipeline requirements or must automatically transform designs to meet these constraints. For a given design, registers can be moved using retiming [8] to determine if the design can meet the constraints.

If a design fails to meet the constraints, it must be modified to match the array's timing model. A feed-forward design can be automatically *repipelined*, but any design with a feedback loop presents difficulty. A common technique is *C-slow retiming* [8]. The only difference between repipelining and C-slow retiming is where the registers are added. Repipelining simply adds k registers to every design input before attempting to retime the design, wherease C-slow retiming replaces every register with C registers before re-

[2]Four is simply a parameter we determined by experimentation, as fanout nets are locked during later routing steps.

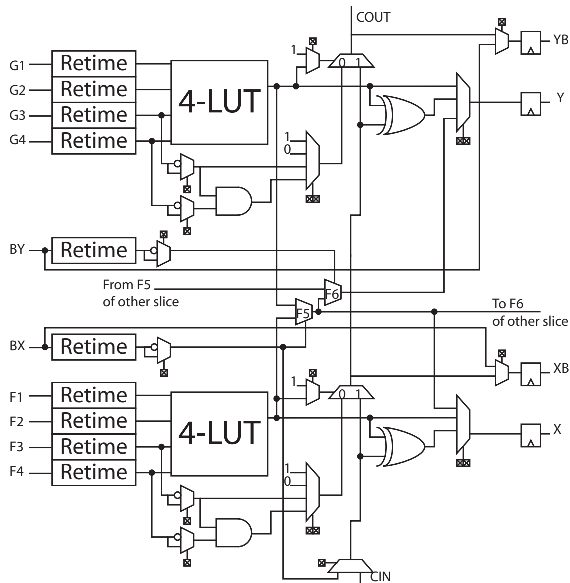

Figure 5: The Slice used by the SFRA

timing. Once retimed, a C-slowed design operates on C independant data streams in a round-robin fashion. Because we wish to accept arbitrary designs containing feedback, not just feed-forward pipelines, our tool only performs C-slowing.

In either case, the retiming process is used to determine the minimum number of registers which must be added to the design. For most fixed-frequency architectures (including the SFRA) there are no global constraints in this process. Rather, local constraints are used to mandate that every connection is appropriately registered to meet the architecture's requirements. These constraints can be solved in $O(N^2)$ time (with N the number of connections in the design) using the Bellman-Ford shortest paths algorithm.

The retiming process will converge if the design has sufficient registers and provides a placement for these registers. We have observed that, if such a solution exists, the process converges quickly. Thus instead of performing all N iterations required by the Bellman-Ford algorithm, the process can be halted considerably earlier. A binary search can be used to find the minimum k or C required for a design.

The retiming technique is the same for all fixed-frequency architectures; the only significant modification we make is to integrate retiming with routing. We first perform an initial C-slow retiming based on an approximate delay model. This retiming is used to prioritize signals that are on the critical path so that critical signals will have no registers added beyond the minimum necessary to account for routing delays. It is also used to measure the success of routing in minimizing additional delays. After detailed routing, a final retiming is performed, using the precise delay model of the target architecture.

6. THE SFRA: A CORNER-TURN ARCHITECTURE

To compare the benefits and costs of a fixed-frequency, corner-turn architecture, and to take advantage of high quality commercial tools, we have defined a corner-turn architecture with effectively the same CLB as the Xilinx Virtex FPGA [16].

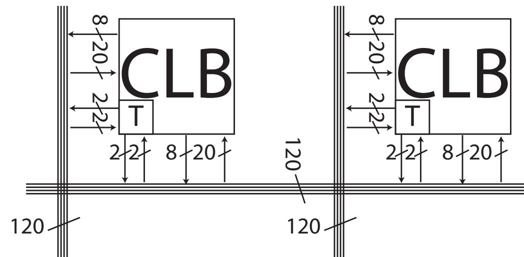

Figure 6: The CLB embedded in the routing network

This architecture, the SFRA, is generally compatible with the Virtex, albeit with some design restrictions. Designs must use a single global clock to permit retiming and the fixed-frequency model. Because of the semantic changes required by C-slowing, both resets and clock enables must be expressed as combinational logic rather than using the built-in primitives. Similarly, the use of LUTs as RAMs or SRL16s is forbidden.[3]

The SFRA CLB contains two *slices* that are derived from the Xilinx Virtex's slice. Figure 5 shows the SFRA's slice. Due to the semantic restrictions, there is no longer any support for LUT-as-RAM or register clock enables and resets. Thus the CE and SR inputs, and their associated logic, are removed as unnecessary. The XQ and YQ outputs are also not needed, as the explicit registers in the initial design are converted to retiming elements during the C-slowing process. To accommodate variable delays, all inputs are assumed to have retiming chains. All outputs are registered, with the carry chain registered every 4 CLBs.

During our experiments, we observed that, although the BX and BY inputs and the YB and XB outputs are required for design compatibility, they are seldom used in our benchmark designs. With the SFRA's semantic restrictions, the BX input is only used for the F5 multiplexer and to provide an external source for the carry chain, while the BY input is only used for the F6 multiplexer. Similarly, the XB and YB outputs are only used for tapping the carry chain as part of a multiplier. Hence we modified these I/Os to allow their use as turns by the router. In addition to the turns formed by the BX/BY inputs of each slice, we provide two additional turns per CLB. The array thus contains an effective six turns per CLB, enough to pushroute easily all our benchmarks.

The interconnect itself (illustrated in Figure 6) contains 120 wires in each horizontal and vertical channel. Although significantly more than are needed for our benchmarks, the channel capacity is comparable to the wiring resources in the Virtex part. All inputs are sourced from both the horizontal and vertical channels, with each input able to be sourced from any wire. Similarly, all outputs can be driven onto any wire in the channel, and an output can be directed onto both the horizontal and vertical channels. There are no restrictions to the allowed fanout in this interconnect.

[3]In our experiments with the LEON synthesized core, these restrictions added less than 10% to the CLB count.

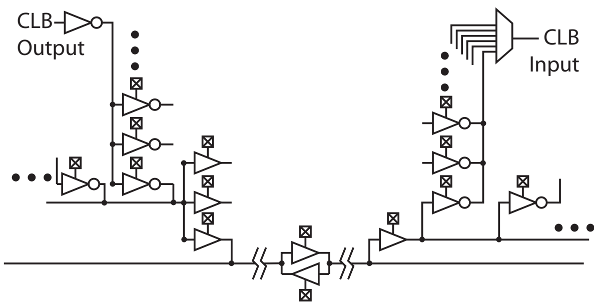

Figure 7: The basic circuits of the SFRA's interconnect. Each output is driven onto a series of 40 tristates, with each tristate driving an output wire. Every output tristate in the same row drives onto the same output wire. The output wires can then be driven onto one of three different interconnect wires. The general interconnect wires are broken and rebuffered every 3 CLBs, with bidirectional registers every 9 CLBs.

Every interconnect wire is broken every three CLBs with a bidirectional buffer, and every nine CLBs with a bidirectional register. These breaks are staggered evenly in the traditional braided fashion. The breaks help detailed routing by providing small segments, and they increase performance by rebuffering and pipelining connections.

Because all registers, switches, and outputs are pipelined, the SFRA is a fixed-frequency architecture. The critical path is from the output of a LUT or turn register, through the output buffer, across up to nine CLBs of interconnect, through the input buffer, and into the next register.

7. THE SFRA LAYOUT

We have constructed the layout for the routing channel and CLB I/Os in a 180 nm process to evaluate the area and performance of this proposed architecture. Within each C-box (Figure 7), the output buffers consist of a set of initial drivers, tristate buffer switches which route signals onto intermediate wires, and a set of output drivers that drive each intermediate output onto one of three general interconnect wires.[4] The input buffers take signals from the general interconnect and drive onto a set of local wires, where tristate drivers are used to select individual signals onto a final level of hierarchy. A multiplexer selects the final signal. All drivers include associated state bits to store configurations. The two turns per CLB are implemented as additional input and output buffers associated with the CLB.

This strategy was chosen because it creates a hierarchical network: The output drivers only drive a small number of tristates, which only drive short segments of wires before being buffered onto the general interconnect. The wires in the routing channel only directly drive the local input rebuffers, enabling long communication within the target cycle. The rebuffering and the tri-state design on the input buffers allows this interconnect to support arbitrary fanout. The diagram in Figure 7 presents only a simplified view of the actual implementation.

There are two different versions of the pieces used in the layout, a version where the routing channel's pitch is two signals in the tile, and one where it is three, that have different aspect ratios. Both use a row/column style addressing to ensure than only one point is actually active. These addresses are driven directly from configuration bits, not decoders, as general decoders require more area than six-transistor SRAM cells. Also, the routing channel includes both the bit-lines used to load configurations and the configuration cells itself.

Metal-1 and metal-2 layers are used for local signals, with metal-1 being used for signals parallel to the routing channel and metal-2 used for perpendicular signals. Metal-3 is used for the local wires in each routing channel, effectively the lower end of the hierarchy. Metal-4 is used for the upper level. Metal-5 and Metal-6 are used as a bridge, allowing the vertical channel's signals to be routed over active horizontal interconnect. Also included in the routing channels is a power distribution grid.

We completed the layout for the input and output buffers, including the I/Os necessary to implement the corner turn. This layout, outlined in Figure 8, includes all the configuration bits, bit-lines to load configurations in parallel, and the rebuffering which transfers local signals to and from the metal-4 routing channels and the buffers used to break the interconnect at a regular interval. We have not yet implemented the interconnect registers, but include a reasonable space for it in our layout. We have also not implemented the CLB itself, but have left a more-than-sufficient space.

The resulting area is approximately 160,000 μm^2. This is roughly 3.9 times larger than the Xilinx Virtex E's CLB tile (with interconnect), but only 1.5x larger than the HSRA's tile. The gap in the tile for the logic cell is sufficient to accommodate the entire Virtex E CLB tile, suggesting that the SFRA's CLB plus retiming registers could easily fit in this space.

However the SFRA tile is substantially larger than the approximately 5 $M\lambda^2$ Xilinx CLB. This is due primarily to the cost of all the switches needed to implement the input and output buffers, as every input or output needs to be connected to every wire in the routing channel. Unlike the Xilinx, the SFRA has pipelined interconnect and switch points.

Simulation of the circuits suggest that the design can run at 300 MHz, including "clk→Q" and setup time, based on the critical path of an output register, through an output buffer, across 9 CLBs of interconnect, through an input buffer, and to an input register.

8. THE COMPLETE SFRA TOOLFLOW

The complete corner-turn toolflow begins with the Xilinx tools that are used for design entry, placement, and mapping. The mapped output, an `.ncd` file, is converted to Xilinx's textual representation, `.xdl`, before being loaded into our back-end router and retiming tool.

We chose Xilinx's placement tool for several reasons. Although the ideal cost-function for a corner-turn array is different, the Xilinx's hex lines and longlines create a similar cost function where direct horizontal and vertical alignments offer substantial benefits over slight misaligment. Also, by

[4]This minor restriction on outputs (no two outputs from the same CLB can be routed onto the same group of three wires) has effectively no effect on routing.

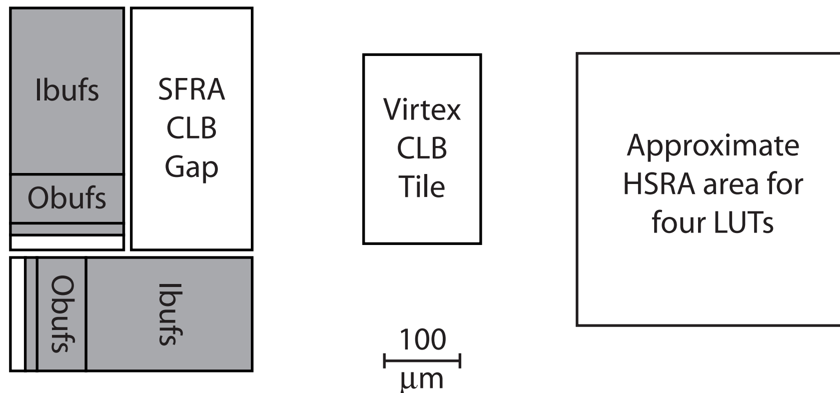

Figure 8: The area required for the layout of the tile. The input buffers and output buffers include the I/Os for the turn as well as the CLB, plus the configuration bits required. The wires for the channels are located over the I/O buffers for the CLB. This area is compared with that of the Virtex E FPGA (determined by die measurement) and a scaled four 4-LUT tile from the HSRA.

using existing placements, we can provide quantitative comparisons of routing time on identical designs.

Our back-end tool first performs an initial retiming, as described in Section 5, to determine which nets are on the critical path. The initial retiming uses a set of estimated delays based on best-case routing. This information is used both to gauge the precision of the router and to guide the routing process. Nets that are on the critical path receive priority during subsequent routing to minimize the delays incurred.

Global routing proceeds, as outlined in Section 4, with priority given to short nets and nets on the critical path. The priority of short nets improves the preformance of the zig-zag routing step by insuring that longer nets, with more flexibility, will be more likely to be routed during this phase.

For detailed routing we simply use modified greedy routing within each channel. All nets in a channel are first sorted by the endpoint before being routed. Each net is assigned to the first available routing track that minimizes the number of registers crossed and the number of excess routing slots used. Because of the inital sort, this step requires $O(N \log(N))$ time in the worst cas, with N being the number of signals in each channel. Finally, a second retiming pass is performed using the actual delays produced by the router.

9. EVALUTING THE SFRA

To evaluate our proposed architecture and toolflow, we used four benchmarks: AES encryption [11], Smith/Waterman (S/W) sequence matching [12], a synthetic datapath, and the LEON 1 microprocessor core [3]. The Xilinx benchmarks originally targeted the 250 nm Spartan II. To compensate for the process differences, the Xilinx clock rates are scaled by 1.4 (40%).

The AES, Smith/Waterman, and synthetic datapath designs are heavily optimized for the Virtex architecture. These applications are very appropriate for aggressive C-slowing, as most usages attempt to optimize throughput. These three benchmarks were hand-mapped, and for placement we have included both hand-placed and automatically placed versions. Thus we can differentiate between placement either by hand or through simulated annealing and the subsequent effects on the router. The three benchmarks stress different aspects, as AES uses bitwise operations and table lookups, Smith/Waterman is almost entirely 16-bit arithmetic operations, and the synthetic datapath is typical of the core of a 32-bit processor, including register file, shifter, and ALU.

The final benchmark, the LEON microprocessor core, is a fully synthesized SPARC-compatible core. It was synthesized using Synplify, with clock enables suppressed. To allow C-slow retiming, the resulting EDIF was edited to replace hardware resets with explicit logic. Although it is not sensible to aggressively retime a processor core,[5] the LEON core is very representative of designs that use HDL synthesis and simulated annealing. Being over 6000 LUTs, it is a substantial design.

All these benchmarks were placed using the Xilinx placement tools (version 4.1) with maximum effort selected. The Xilinx router times are also for maximum effort, as we have noticed performance drawbacks when lower effort is used. To provide an effective comparison, we also utilized our own C-slow retiming tool we developed for the Virtex [15]. This tool can effectively double the throughput on our benchmarks when targeting the Xilinx Virtex. We therefore do not just compare with what the Xilinx toolflow currently supports, but also potential improvements. All tools were run on a 550-MHz Pentium III with 256 MB of memory running Windows 2000.

Our tool for retiming Virtex designs takes considerably longer as general retiming requires a larger set of constraints and $O(N^2)$ memory compared with linear memory usage when targeting the SFRA. Our Virtex-targeting tool also has an $O(N^3)$ operation that could be reduced to $O(N^2 \log(N))$ in a more sophisticated implementation, with

[5]However, a 2-slow or 3-slow retiming produces a useful multithreaded architecture.

Benchmark	Xilinx Routing Time	Custom Xilinx Retiming Time	SFRA Routing Time	SFRA Toolflow Time
AES (hand placed)	405 s	73 s	2 s	6 s
AES (autoplaced)	542 s	77 s	2 s	6 s
Smith/Waterman (hand placed)	104 s	178 s	2 s	8 s
Smith/Waterman (autoplaced)	102 s	169 s	2 s	9 s
Synthetic Datapath (hand placed)	284 s	47 s	2 s	5 s
Synthetic Datapath (autoplaced)	307 s	50 s	2 s	6 s
LEON (autoplaced)	432 s	hours	11 s	62 s

Table 1: Time taken by the tools to route and retime the benchmarks, for both the Xilinx Virtex and the SFRA. The SFRA toolflow time includes both routing and retiming, while the Xilinx retiming time excludes the $O(N^3)$ Dijkstra's step used to calculate the W and D matrixes (see text for explanation). The SFRA router minimizes the number of turns required for each connection, and all connections use minimum-length wires.

Benchmark	Adjusted Xilinx Clock Rate	SFRA C-slow Factor	Expected Speedup for 300-MHz SFRA	Area-Normalized Speedup
AES (hand placed)	67 MHz	24-slow	4.4x	1.1x
AES (autoplaced)	67 MHz	27-slow	4.4x	1.1x
Smith/Waterman (hand placed)	66 MHz	31-slow	4.4x	1.1x
Smith/Waterman (autoplaced)	60 MHz	37-slow	5.0x	1.3x
Synthetic Datapath (hand placed)	77 MHz	21-slow	3.9x	1.0x
Synthetic Datapath (autoplaced)	70 MHz	23-slow	4.2x	1.1x
LEON (autoplaced)	38 MHz	67-slow	7.9x	2.2x

Table 2: Throughput and Throughput/Area results for the unoptimized Virtex benchmarks when compared with a 300-MHz SFRA.

N being the number of computational elements. To be fair, we excluded the time required to perform this operation when reporting our tool times. If this step were included in our results, the time required to retime designs for the Virtex would double.

Table 1 shows the vast differences in tool performance between targeting the Virtex and the SFRA. Wherease the Xilinx router may require minutes, the SFRA router completes in a couple of seconds. Even the 6000-LUT LEON core can be processed through the complete toolflow in slightly more than a minute, with routing requiring only 12 seconds. Similarly, for retiming, the Xilinx requires a couple of minutes to several hours, while the SFRA toolflow, including two retiming passes, is an order of magnitude faster (and requires considerably less memory, too).

In all cases, global congestion is considerably less than the number of wires available in the routing channel. It may be possible to reduce the size of the SFRA's routing channels without impacting performance. We do not measure detailed congestion as the detailed router uses the entire channel to maximize performance.

Table 2 compares the unoptimized Xilinx designs with the SFRA. Not only do the SFRA's tools run considerably faster, but the resulting designs also support higher throughput. The expected speedup for a 300-MHz SFRA is substantial. Unfortunately, the limited quality of an academic FPGA design becomes apparent if throughput is normalized for area. We see speedups of 3x to 8x when comparing with the unoptimized versions, but when normalized for area this is reduced to 1.0x (no speedup) to 2.2x.

Of particular interest is the greater sensitivity of the SFRA to poor placement. AES, although regular when viewed at a macro level, is composed of highly irregular bit-mixing operations, preventing the SFRA from significantly benefiting from a hand-placed datapath. Smith/Waterman, on the other hand, is composed of aligned arithmetic operations. With automatic placement, the cumulative effect of small misalignments becomes substantial. Note that the degraded placement only affects latency (as reflected by a higher C-slow factor); the throughput remains the same.

Table 3 repeats the experiments with our best automatically optimized Xilinx implementations. Although the SFRA is still faster for all benchmarks, the performance wins are substantially reduced. Speedups are now only 2.0 to 4.6x, and when normalized for area become 0.5x (a slowdown) to 1.2x, or within about a factor of 2 in the worst case.

10. OPEN QUESTIONS

Given that Virtex is a highly optimized, fifth-generation commercial product, the fact that we are within a factor of two is good for this stage of our research. Note that this factor of two is *after* we used our custom C-slow retiming tool to automatically reoptimize the Virtex designs. Compared with the tools provided by Xilinx, the SFRA offers comparable thoughput at a fraction of the tool time.

An important question is what further improvements to our architecture could result in better efficiency. There are three significant open questions remaining for the corner-turn architecture: 1) Can the area be substantially reduced by depopulating the input and output C-boxes? 2) Would less channel capacity be sufficient? 3) How much savings are achieved when the corner-turn interconnect is used for coarser-grained arrays?

The first question arises from the following observation: We utilized full crossbars for the input and output buffers because of the speed and simplicity of the one-dimensional greedy router. If we began to depopulate the full crossbars which formed the C-boxes, this obviously would result in vast area savings. However, depopulating the C-boxes complicates detailed routing, as it is no longer possible to use simple greedy channel packing. Intuition suggests that because the routing of each channel would still be independent, there should almost certainly be fast heuristics that can operate on most designs. The wiring in the channel could be specifically designed to work with a particular heuristic. Additionally, the ability to transpose most LUT inputs should further ease the routing process.

Conversely, is it possible to reduce the channel size used by the SFRA? The current C-boxes are considerably more flexible than those employed by the Virtex, suggesting it may be possible to use this flexibility to create narrower routing channels. Similarly, it should be possible to reduce the capacity near the periphery to save area while maintaining full capacity near the center of the array. This would reduce the total size of the array by reducing the size of the C-boxes.

The final question is again related to the use of full crossbars to implement the C-boxes. Word-oriented crossbars use far fewer switchpoints as fewer wires need to be directly connected. Unfortunately, there are no comparable coarse-granularity FPGAs with mature toolflows that the corner-turn strategy can be compared against. Thus we can not effectively gauge the savings that might occur with this approach.

11. CONCLUSIONS

We have created a new FPGA interconnect architecture, the corner-turn architecture, that offers several advantages over conventional interconnects, including placement compatibility, fast routing, efficiently pipelineable switchpoints, and the potential for defect tolerance. This interconnect style can be quickly routed using fast, polynomial-time heuristics that can route even large, 6000-LUT benchmarks in just a few seconds. A corner-turn interconnect should be well suited to fixed-frequency or coarse-grained FPGAs.

Using this interconnect style, we defined a prototype fixed-frequency FPGA, the SFRA, that is largely compatible with the Xilinx Virtex FPGA. Our tools operate an order of magnitude faster than the Xilinx tools and the resulting designs operate with a higher throughput when operating on substantial designs. Although our array's area is considerably larger than the Xilinx Virtex, it is comparable to previous fixed-frequency FPGAs.

It is left to further research to prove that the area can be reduced by compacting the C-boxes without losing the advantages of the corner-turn architecture.

12. ACKNOWLEDGMENTS

Many thanks to Eylon Caspi for his advice on the semantics of retiming. This work is partially sponsored by Xilinx and the California MICRO program.

13. REFERENCES

[1] V. Betz and J. Rose. Effect of the prefabricated routing track distribution on fpga area-efficiency, 1998.

Benchmark	Adjusted Xilinx Clock Rate and C-slow Factor	SFRA C-slow Factor	Expected Speedup for 300 MHz SFRA	Area Normalized Speedup
AES (hand placed)	147 MHz 5-slow	24-slow	2.0x	.5x
AES (autoplaced)	123 MHz 5-slow	27-slow	2.4x	.6x
Smith/Waterman (hand placed)	120 MHz 3-slow	31-slow	2.5x	.6x
Smith/Waterman (autoplaced)	117 MHz 3-slow	37-slow	2.5x	.6x
Synthetic Datapath (hand placed)	127 MHz 3-slow	21-slow	2.4x	.6x
Synthetic Datapath (autoplaced)	123 MHz 3-slow	23-slow	2.4x	.6x
LEON (autoplaced)	64 MHz 2-slow	67-slow	4.6x	1.2x

Table 3: Performance results for automatically C-slowed applications when compared with an 300 MHz SFRA.

[2] Chameleon systems, http://www.chameleonsystems.com/.

[3] J. Gaisler. LEON SPARC-compatable processor, http://www.gaisler.com/leonmain.html.

[4] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *International Symposium on Computer Architecture*, pages 28–39, 1999.

[5] C. E. D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, 1996. Springer-Verlag.

[6] P. Hallschmid and S. J. E. Wilton. Detailed routing architectures for embedded programmable logic ip cores. In *Ninth international symposium on Field programmable gate arrays*, pages 69–74. ACM Press, 2001.

[7] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Gate Arrays for Custom Computing Machines*, pages 12–21. IEEE, April 1997.

[8] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*, March 1993.

[9] G. Lemieux, P. Leventis, and D. Lewis. Generating highly-routable sparse crossbars for plds. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 155–164, February 2000.

[10] E. Mirsky and A. DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proceedings of the IEEE Symposium on Field-Programmable Gate Arrays for Custom Computing Machines*. IEEE, April 1996.

[11] NIST. Federal information processing standards (FIPS) publication 197: Advanced encryption standard, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[12] T. Smith and M. Waterman. Identification of common molecular subsecquences, 1981.

[13] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. HSRA: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.

[14] N. Weaver. *The SFRA: A Fixed-Frequency FPGA Architecture*. PhD thesis, University of California at Berkeley, 2003. http://www.cs.berkeley.edu/ nweaver/ nweaver_thesis.pdf.

[15] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek. Post-placement c-slow retiming for the xilinx virtex fpga. In *Proceedings of the Eleventh ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, 2003.

[16] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex Series FPGAs*, 1999.