

AN FPGA FOR IMPLEMENTING ASYNCHRONOUS CIRCUITS

Scott Hauck, Steven Burns, Gaetano Borriello, Carl Ebeling

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Field-programmable gate arrays are a dominant implementation medium for digital circuits, especially for glue logic. Unfortunately, they do not support asynchronous circuits. This is a significant problem because many aspects of glue logic and communication interfaces involve asynchronous elements, or require the interconnection of synchronous components operating under independent clocks. We describe Montage, the first FPGA to explicitly support asynchronous circuit implementation, and its mapping software. Montage can be used to realize asynchronous interface circuits or to prototype complete asynchronous systems, thus bringing the benefits of rapid prototyping to asynchronous design.

Keywords: Asynchronous Circuits, FPGA Architectures, Prototyping

1. Introduction

Asynchronous circuits are becoming more prevalent. Most commonly they occur in the interfaces and the glue logic that binds the components of a system. One reason for this is that asynchronous logic is adaptable to delay variations and components designed to function asynchronously can be more easily composed. Some of the differences between asynchronous and synchronous design are the arbitration mechanisms and hazard-free logic design required to ensure proper operation. As evidenced by many of the articles in this special issue, methodologies have recently been developed to effectively design large asynchronous and mixed synchronous/asynchronous systems. In fact, we are reaching the point where designers can contemplate designing complex system components or even entire systems as asynchronous logic.

Unfortunately, implementation media for asynchronous circuits and systems have not kept up with those for the synchronous world. Programmable logic devices do not include the special non-digital circuits required by asynchronous design methodologies (e.g., arbiters and synchronizers) nor do they facilitate hazard-free logic implementations. This leads to huge inefficiencies in the implementation of asynchronous designs as circuits require a variety of separate devices. This has caused most asynchronous designers to focus on custom or semi-custom integrated circuits, thus incurring greater expense in time and money. The net effect has been that optimized and robust asynchronous circuits have not become a part of typical system designs. The asynchronous circuits that must be included are usually designed in an ad-hoc manner with many underlying assumptions. This is a highly error-prone process, and causes implementations to be unnecessarily delicate to delay variations.

Field-programmable gate arrays, one of today's dominant media for prototyping and implementing digital circuits, are also inappropriate for constructing more than the simplest asynchronous interfaces. They lack the critical elements at the heart of today's asynchronous designs. Unfortunately, resolving this problem is not just a simple matter of adding these elements to the programmable array. The FPGA must also have predictable routing delay and must not introduce hazards in either the logic or routing. Furthermore, the mapping tools must also be modified to handle asynchronous concerns, especially the proper decomposition of logic to fit into the programmable logic blocks and the proper routing of signals to ensure that required timing relationships are met.

Ideally, we need an FPGA that can support both synchronous and asynchronous circuits with comparable efficiency. As a step in this direction we present Montage, an integrated system of FPGA architecture and mapping software designed to support both asynchronous circuits and synchronous interfaces. The architecture provides circuits with hazard-free logic and routing, mutual exclusion elements to handle metastability, and methods for initializing unlocked elements. The mapping software generates placement and signal routing sensitive to the timing demands of asynchronous methods. With these features, the Montage system forms a prototyping and implementation medium for asynchronous designs, providing asynchronous circuits with a powerful tool from the synchronous designer's toolbox.

The remainder of this paper is broken into four sections. In section 2 we discuss in detail why we cannot simply use standard FPGAs and mapping techniques for asynchronous circuits. In sections 3 and 4 we discuss the two aspects of Montage: its architectural design and its mapping tools, focusing on how they address the issues raised in section 2. Finally, in section 5 we conclude, and discuss some directions for future work.

2. Requirements for FPGA Support of Asynchronous Circuits

There are numerous reasons why synchronous FPGAs and mapping techniques cannot be used for asynchronous circuits. They fall into the general categories of hazards, timing constraints, stateholding elements, analog components, and decomposition. These are discussed in turn below.

2.1. Hazards

In a synchronous circuit a clock determines when a signal is sampled. The value of the signal is only important near the sampling clock edge, allowing the designer to largely ignore any extraneous signal transitions. In contrast, an asynchronous circuit is constantly sampling its signals. Because of this, any extraneous transitions (“hazards”) may cause incorrect results, and thus must be avoided.

Hazards may be inherent in a Boolean function or arise because of how it is implemented. For example, if both inputs to an XOR are allowed to change simultaneously, there is an unavoidable hazard since one input may change before the other. Thus, slight differences in signal arrival times will cause the circuit to generate spurious transitions. Asynchronous circuits either do not use elements with unavoidable hazards, or do not allow the hazardous situations to occur. However, during circuit mapping to FPGAs, especially during decomposition, the circuit logic may be altered, possibly adding hazards. Careful decomposition techniques must be used to restructure the logic so that the resulting circuit remains hazard-free. Such issues are dealt with in subsection 2.5.

Designers do not have precise control over how logic is implemented within the logic block of an FPGA. These logic blocks must be designed such that they do not introduce any new hazards into implementations. Again, circuits may have unavoidable hazards that an implementation cannot avoid, but hazards that do not exist in the original circuit must not be introduced by the implementation. If one has only simple gates as logic elements (e.g., ANDs and XORs, as in the CFA FPGA [Concurrent91]), making them hazard-free is easy. However, look-up tables (LUTs), the element generally used in FPGAs to implement arbitrary n-input functions, are much more complex than simple gates. However, as will be shown in section 3, there are implementations of LUTs that do not introduce hazards into the logic they implement.

A less obvious concern is that FPGA routing is typically not just simple wires, but includes routing “functions”. For example, the interconnection of several wires in an FPGA will often be accomplished by a multiplexor, and this multiplexor must be hazard-free. However, it is easy to design a multiplexor which has charge-sharing. Charge-sharing allows a value to be stored on an unused capacitance, and when this value is reconnected to a signal it may momentarily alter the signal’s value. In this way, the routing functions can introduce hazards where the specification did not even have any logic.

The point to be made is that hazard avoidance in FPGAs is a subtle issue, and while it is possible to remove hazards from an FPGA architecture by careful design, an FPGA not specifically targetted to asynchronous circuits will most likely generate hazards.

2.2. Timing Constraints

All asynchronous design methodologies make assumptions about how and where delays are encountered in the resulting circuits. These circuits depend on the assumptions, and an FPGA system must meet these assumptions to properly map the circuits. Bounded-delay methodologies [Unger69][Hollaar82][Nowick91] require upper-bounds on the delays in all circuit elements, and insert extra delays into feedback or other paths in response. The magnitude of the inserted delays is any amount greater than some formula on other delays in the circuit, and thus can be left to the placement and routing tools to specify exactly. However, the FPGA system must be able to insert these delays in the circuit. Fine-grained architectures [Algotronix91][Concurrent91] generally leave many logic cells unused in their mappings, and the paths to be delayed could be fed through unused cells configured as buffers, thus delaying the signal. All FPGAs could delay signals by using more circuitous paths. However, the router must be able to efficiently find these paths. Routers usually are based on finding the shortest path under some cost metric, which can be computed efficiently. Unfortunately, finding the shortest path with delay greater than some value is a more complex problem. Other methodologies have bundled-data constraints [Brunvand89][Sutherland89], which require the delay along one path (where a path includes both logic and routing) be greater than the delay on other paths. This is a similar, but more difficult, version of the same problem, since the burden of meeting the timing constraint must be fairly shared by all segments of a path.

Quasi-Delay Insensitive methodologies [Martin90][Ebergen89] contain isochronic fork constraints. These are either symmetric, where all ends of a fork must be reached nearly simultaneously, or asymmetric, where one end of the fork must be reached before the other. While Speed-Independent methodologies [Chu87][Kishinevsky92] assume that there is no delay in any wire, in practice these can be replaced by isochronic forks. In many FPGAs, the routing

resources are very complex, with delays often greater than the logic delays. In such a system, meeting isochronic constraints can be almost impossible. While asymmetric forks can be handled simply by routing to the required earlier destination first, and then routing from there to other destinations, symmetric forks are much harder. Unless there is a relatively fast path from some shared routing resource to all fork destinations, there is very little chance that the symmetric isochronic fork assumption will be met.

The final timing constraint used is atomic, multi-output gates. Specifically, some methodologies [Brunvand89][Ebergen89] use gates with more than one output, and it is assumed that the logic for all gate outputs will react to a new input at about the same time. For example, a TOGGLE element is a one-input, two-output element where one output responds to odd input transitions, and the other to even input transitions. It is assumed that by the time one output sends an output transition, the other output has sensed the input. Thus, the environment can then send in a new transition without worrying that the unfired output hasn't sensed the previous input. A method to handle this constraint is to carefully craft a module set to guarantee that the atomic gate constraint is met [Brundvand91]. These modules will contain not only logic, but also routing designed to ensure the constraints. In order to allow this, the mapping tools must respect the structure of the modules, ensuring that all resources are assigned as dictated, and the architecture must be reasonably uniform so that a module has many potential placements.

2.3. Stateholding Elements

Synchronous circuits require some mechanism for storing information from one clock cycle to the next. Thus, FPGAs usually include D-latches in their logic blocks. Asynchronous circuits do not have any single way of storing information, but instead different methodologies use different structures. They use C-elements, asynchronous S-R flipflops, standard latches with local clocks, or even Q-flops [Rosenberger88] (locally clocked latches with metastability handling). While adding each of these elements into the standard logic cell would be expensive, all but the Q-latches can be implemented out of standard combination logic. To implement these n-input stateholding functions one can express them as an (n+1)-input combinational function, with the function's output fed back as the new input. However, the methodologies usually consider logic elements as atomic gates (discussed in subsection 2.2). Thus, we must be able to ensure that this feedback path is fast enough for a changing element to stabilize before another input arrives. In current commercial FPGAs this feedback path is routed the same as all other signals, and meeting the timing constraints can be difficult.

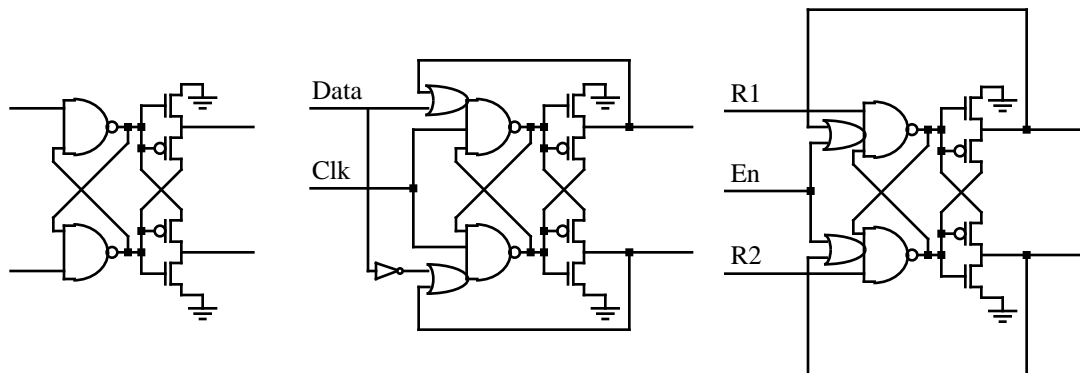


Figure 1. A mutual exclusion element (left), and a synchronizer (center) and enabled arbiter (right) built from it.

It is also important to consider the starting state of the FPGA. After an SRAM-based FPGA is programmed, or after an antifuse-based FPGA is powered up, the programming will be established, but the signal values may be incorrect. In a synchronous circuit, we can simply set the latches to some preset state (a feature provided in many FPGAs), and wait for the circuit to stabilize before starting the clock. Unfortunately, an asynchronous circuit has no global clock to stall, and the stateholding functions often have no latches to preload. While one could require the circuit logic to have an explicit reset signal, this would require a large amount of extra logic. An alternative is to provide an underlying mechanism to hold stateholding functions at a preset value until the circuit settles.

2.4. Analog Components

In many asynchronous circuits there are elements used to reliably sample a signal at a given point (a synchronizer), or to determine which of two signals arrives first (an arbiter). What is special about these elements is that while the elements may take an arbitrarily long time to respond, the responses must always be correct and hazard-free. For example, an arbiter has 2 inputs and 2 outputs, and it raises an output when the corresponding input is raised, while ensuring that at most one output is raised at a time. If an output is raised, it will not be lowered until the corresponding input is lowered. Unfortunately, this behavior cannot reliably be implemented in a purely digital circuit. Thus, we cannot use the standard digital logic elements provided in FPGAs to map these elements. One solution is to include a mutual exclusion element (figure 1) into the FPGA architecture. With the addition of appropriate digital logic around it, the element can reliably perform synchronization and arbitration functions.

2.5. Decomposition

In order to map a synchronous circuit into an FPGA, it is necessary to restructure it so that its basic elements fit in the FPGA's logic elements. This process is called decomposition or technology mapping. For LUT-based FPGAs, one breaks all logic elements into individual gates with no more inputs than the LUT can handle. For other FPGA logic blocks, this may require changing what types of gates are used as well. For example, an FPGA whose only logic element is a NAND cannot implement an XOR directly. Instead, the XOR would be replaced with an equivalent sum-of-products form, with the proper number of inputs per gate, which can then be implemented by NAND gates.

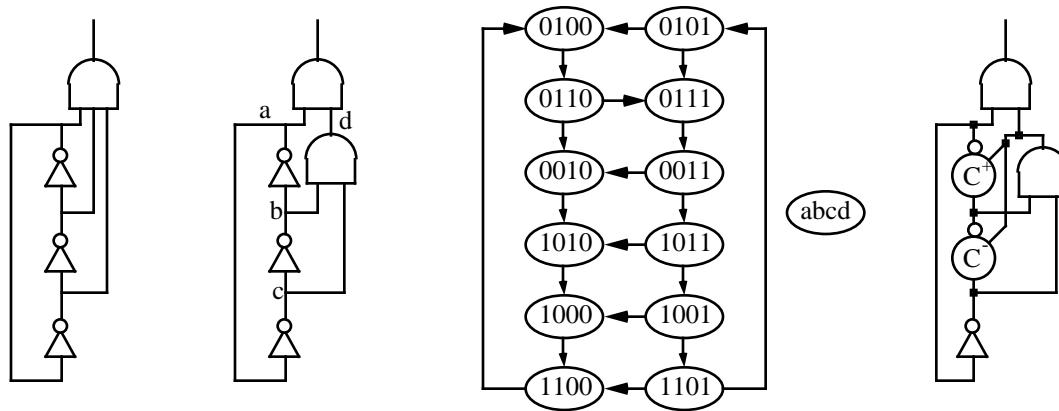


Figure 2. Example of an incorrect decomposition (center left) of circuit (left). A correct decomposition is at right.

Unfortunately, while decomposition for synchronous circuits is well understood, these techniques are not sufficient for all asynchronous methodologies. For synchronous circuits, operations such as De Morgan's Law, associativity, and boolean minimization can all be applied. For bounded-delay methodologies [Hollaar82][Nowick91] many of these techniques can be used, especially purely algebraic operations [Unger69][Kung92]. For other methodologies, particularly the Quasi-Delay Insensitive [Martin90][Ebergen89] and Speed-Independent [Chu87][Kishinevsky92] methodologies, even these are suspect. For example, consider the circuit in figure 2 left. It contains a ring-oscillator of three inverters, and a 3-input AND gate attached to the inverter outputs. In this circuit the AND gate will never fire. If we use standard decomposition techniques to map this to an FPGA with 2-input LUTs, the AND gate will be broken into two cascaded AND gates (shown at center left). Note that this resynthesis is one of those allowed for both synchronous and bounded-delay circuits. As we can see from the state graph at center right, this circuit can reach state 1011, where the top AND gate might become TRUE. Thus, this decomposition is incorrect. A correct decomposition is shown at right. While the original circuit is not useful since it never generates an output, it is representative of a large number of situations in asynchronous circuits where a gate is partially but not completely activated. That is, there are many situations where a gate comes within one input transition of an output change, but the circuit changes some other gate input. In fact, a circuit could reach all states within one input of the gate firing without actually firing the output. In such a situation any simple decomposition can fail, since there is some new signal transition introduced that is unsensed by the rest of the circuit. What is necessary is to do a more complex resynthesis of the circuit, ensuring that no gate transition is unsensed. Unfortunately, we are currently aware of no work that addresses this problem.

Technology mapping must also deal with hazards in an FPGA's logic elements. While a logic element might be able to implement a given specification, it is possible for the element to introduce new hazards. For example, an FPGA could include a 4-input AND-OR element, with the logic function $(a * b) + (c * d)$. Such an element could implement the function $(a * c) + (\bar{c} * d)$ with the aid of an inverter and proper signal routing. However, this circuit has a hazard for the transition $acd \rightarrow a\bar{c}d$. Thus, unless the circuit to be mapped does not allow this transition to happen, the circuit cannot be implemented with this element. Techniques exist for handling such situations in bounded-delay circuits [Siegel93].

3. The Montage Architecture

As discussed earlier, asynchronous circuits are not well served by current FPGA architectures. Asynchronous logic implementations must consider hazards, synchronization and arbitration of events, and strict adherence to the timing assumptions of the design methodologies. Unfortunately, these issues are not addressed in current FPGAs. Some of the elements required cannot be implemented in the standard digital logic found in these devices. In addition, the logic and routing elements must be designed more carefully to avoid hazards, since in asynchronous circuits every transition is important. Finally, routing resources must have predictable, optimizeable delays to help meet timing assumptions.

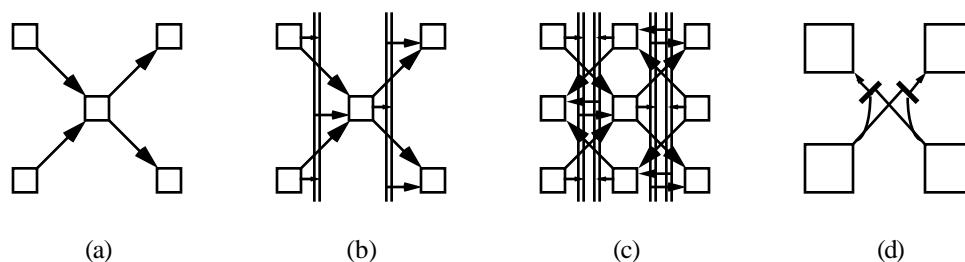


Figure 3. The overall structure of the Montage FPGA shown in a progression of steps. The basic fanin/fanout structure (a) is augmented with segmented routing channels (b) attached to a third RLB input and output. The structure (c) is obtained by merging two copies of (b), with data flowing in opposite directions in the two copies. Shown in (d) are the connections between the two copies at diagonal crossings.

The Montage FPGA is a version of the Triptych architecture designed to handle synchronous interface and asynchronous circuits. Since much of Montage is identical to Triptych, we direct readers wishing more information on the architecture to [Hauck92]. Like Triptych, Montage is an SRAM-based FPGA, which have the advantage over antifuse-based FPGAs of allowing the chip to be programmed for delay testing without permanently configuring it. Note that while we discuss a specific instance of the Montage architecture in this paper, we are currently considering architectural variations, including alteration of the vertical interconnect and increases to look-up table size.

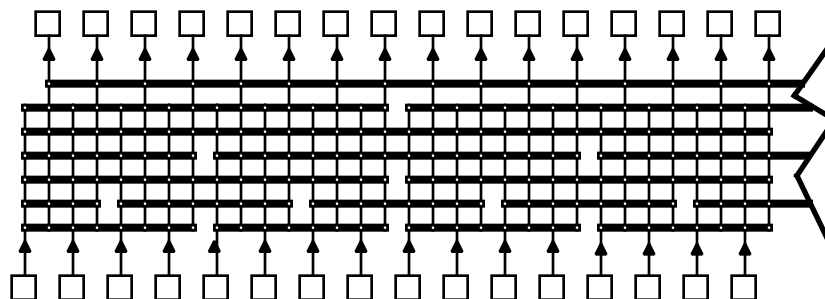


Figure 4. Top half of a segmented channel (on its side). The bottom half is identical to the top.

The Montage global routing structure is identical to the Triptych routing structure, with diagonal connections between local cells, augmented with vertical segmented channels (figure 3). This structure has proven to be effective for mapping general synchronous circuits. It is even better suited to asynchronous circuits, where one expects to find more tightly connected subcircuits, and less random global routing. Also shared with Triptych is the philosophy of allowing mappings to fix the tradeoff between logic and routing resources by having logic blocks capable of performing routing functions.

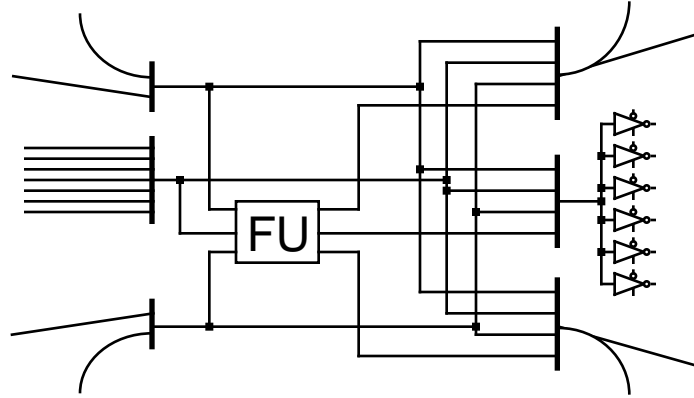


Figure 5. Montage routing and logic block (RLB) design. The RLB consists of 3 multiplexers for the inputs, a functional unit, 3 multiplexers for the outputs, and tristate drivers for the segmented channels.

Montage's short, diagonal connections are used for most routing, providing fast signal propagation. The vertical segmented channels are used for longer range connections and large fanout nodes. They are implemented as a set of segmented "channel wires" (figure 4) that connect the center outputs of RLBs to the center inputs of RLBs flowing in the same direction in the next column. Needless to say, this flexibility leads to slower signal propagation, and speed-critical designs will avoid using the vertical channels for critical paths. There are 7 tracks in a vertical channel, with 6 handling inter-RLB routing and a seventh to carry a pin input. The 6 inter-RLB tracks are broken up into two tracks each of 8, 16, and 32 RLB high segments. The basic Montage array is 64 RLBs high by 16 wide.

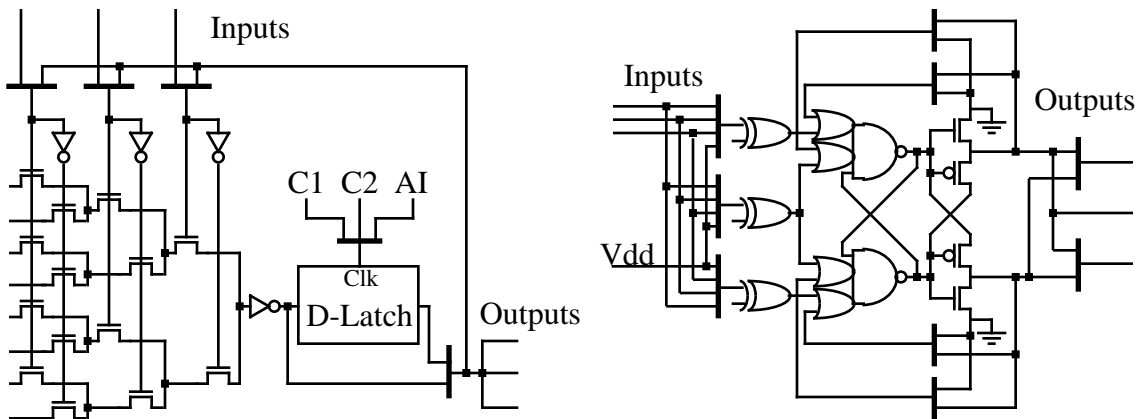


Figure 6. The two types of functional units: the logic block (left) and the arbiter unit (right).

A Montage RLB (figure 5) has three inputs and three outputs, and a functional unit (FU) which operates on the inputs. There are two different types of functional units. The first is a logic block, which implements logic functions and stateholding elements. As shown in figure 6, the logic block has a look-up table capable of implementing any function of 3 inputs. The switch logic function block shown was chosen because it does not suffer from charge sharing. This is important because asynchronous circuits require very clean signals, with absolutely no extraneous transitions. The function output can be fed through a d-latch. This d-latch can be configured with one of two clocks in synchronous mode (allowing two independently clocked synchronous circuits to coexist on a chip), or with a choice of initialization state in asynchronous mode. In the asynchronous initialization mode the latch is set to a value during programming. The latch holds the function output to this value until the circuit stabilizes, at which point the latch is bypassed. Each RLB can choose independently how to use the d-latch, so a single circuit can have two separately clocked synchronous circuits, asynchronous elements initialized with the built-in circuitry, and unlatched logic blocks. Note that any one of the three logic block inputs can be replaced with a feedback line carrying the function's output value. This feature allows asynchronous state-holding elements to be built. This is done by expressing the state-holding function of n inputs as a combinational function of $(n+1)$ inputs, where the extra input is the function's previous value. In this way a single logic block can implement any 3-input

combinational function, or a 2-input stateholding function such as an asynchronous S-R flipflop or a Muller C-Element.

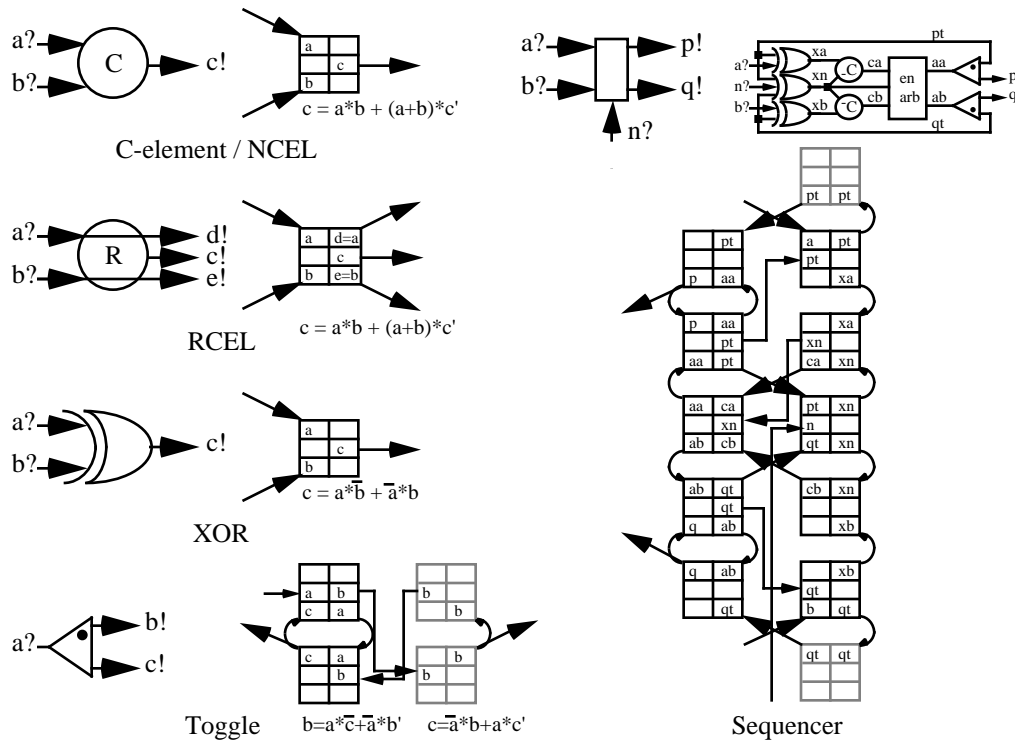


Figure 7. Ebergen's basic elements, mapped to the Montage FPGA. Included at top right is a circuit diagram for the implementation of the Sequencer. Other elements have logic equations for the outputs.

The second type of functional unit is an arbiter block. This block is capable of implementing an arbiter, an enabled arbiter, or a synchronizer. They can also be combined with logic blocks to form more complex functions such as Q-flops. All inputs are completely permutable and invertable. Although we expect these blocks to be used infrequently, the roles they serve in asynchronous circuits are essential, and are not implementable in standard digital logic. Thus, they must appear as special, built-in blocks in any FPGA which hopes to implement asynchronous circuits, but which does not allow mappings to program circuits at the transistor level (for an example of an antifuse-based FPGA which might allow sufficient transistor-level programming to implement an arbiter, see [Marple92]). As an example of Montage's power, all of Ebergen's basic elements [Ebergen89] can be mapped as shown in figure 7. Since elements in gray are only used for routing, and can easily be used for logic from other circuit elements, the C-element, NCEL, RCEL, and XOR can all be mapped into a single RLB, a Toggle requires two RLBs, and the Sequencer ten RLBs. The Sequencer is the only element including an arbiter block, which is used in the center left RLB of the Sequencer mapping. Since approximately eight Montage RLBs can be fit into the space of a single Xilinx CLB [Xilinx92] (the basic tile of one of the most popular current FPGAs, which can implement at most 2 functions in a single CLB), these are very efficient mappings. Larger hand-mappings are shown in figure 8, including Martin's Fair Arbiter [Martin90], and a Sutherland-style FIFO [Sutherland89].

Currently we plan to have a 15:1 ratio between the number of logic blocks and arbiter blocks, as shown in figure 9. This number was chosen based on the relative infrequency of arbiters and synchronizers in typical asynchronous circuits. Since we found that typical Triptych mappings used at least 25% of their RLBs for routing only, jobs which the arbiter RLBs in Montage are equally capable of handling, we believe that most unused arbiters will be absorbed into this factor. However, we have taken care to ensure arbiter blocks occupy the same amount of area as logic blocks, allowing easy alteration of the arbiter mix in Montage implementations.

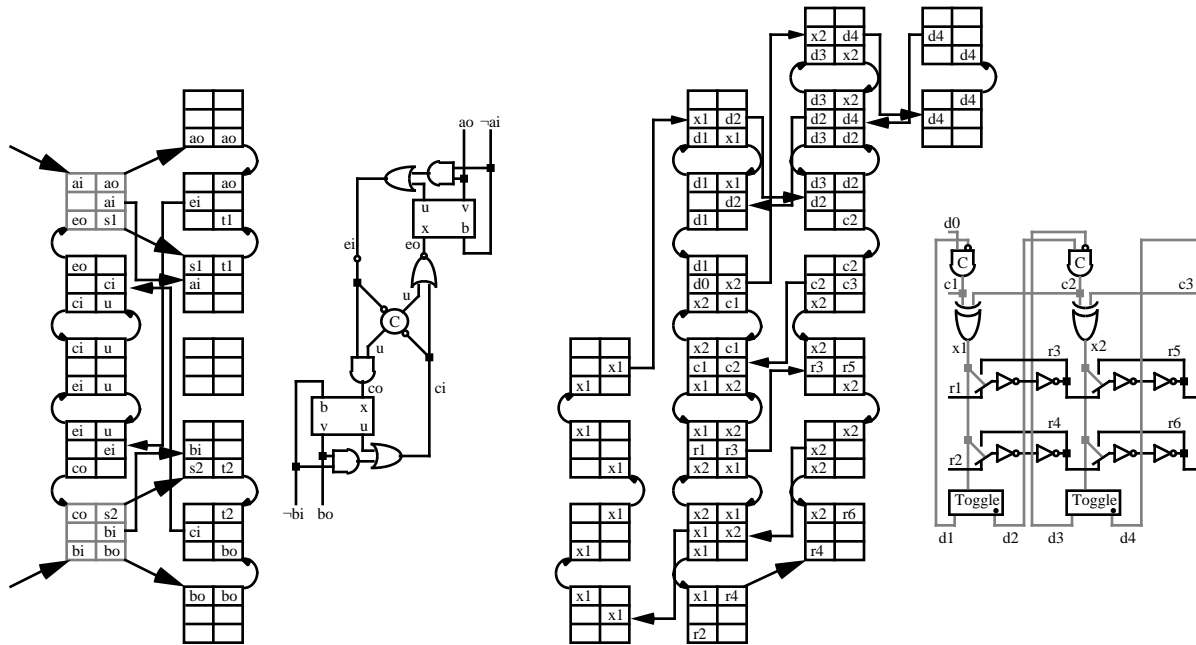


Figure 8. Two example circuits: (left) Martin’s fair arbiter [Martin90], built with two synchronizers (arbiter blocks have grey outlines), and (right) Sutherland’s micropipelined FIFO [Sutherland89]. Note that although only two levels of the FIFO are shown, the mapping fits together for longer FIFOs.

An important point to be made about the architecture is how Montage handles bundled data, inserted delays for bounded-delay circuits, and isochronic forks. For bundled data and inserted delays, the Montage routing structure’s simplicity makes it easier to design a router which ensures that signals to be delayed take longer paths. Also, since Montage mappings will typically leave up to 25% of the logic blocks unused, these unused logic blocks can serve as inserted delays by configuring the blocks as buffers. For isochronic forks, there are different implementations for the two types of isochronic forks (figure 10). For asymmetric isochronic forks, forks where one end must be reached before the other, the signal is routed to the critical end of the fork, and then back out from that block to the other end of the fork. Thus, the dual routing and logic nature of a Montage RLB ensures that the signal reaches one end before the other. For symmetric isochronic forks, forks where all ends must be reached simultaneously, the ends of the fork are placed either off the same interconnect line, or off diagonals flowing from a shared source RLB. In this way, the isochronic fork depends on the delays of very localized elements, delays which can easily be checked during initial chip verification.

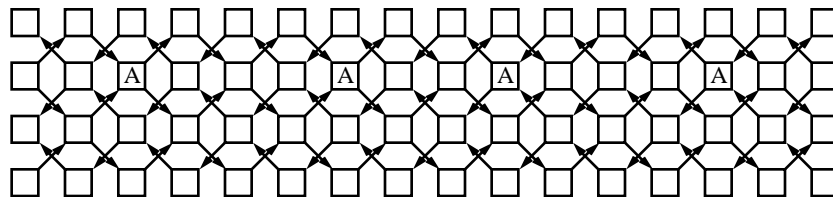


Figure 9. Distribution of arbiter (labelled “A”) blocks throughout the Montage array. The complete array is built by stacking vertically the tile shown above.

4. Montage Mapping Software

As part of the Triptych project we have developed placement and routing tools to support the FPGA architectures. These include a simulated annealing placement program and an iterative router that optimizes both area and delay. While the Montage architecture shares much in common with Triptych, and the Triptych mapping tools can largely be applied to Montage, there are some extensions necessary. Specifically, the placement and routing tools must ensure isochronic fork constraints, and logic and arbiter functions must be placed in 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.

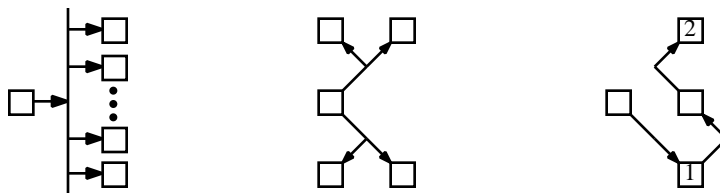


Figure 10. Placement of a symmetric isochronic fork on an interconnect line (left) and on diagonals (center), as well as an asymmetric fork (right) reaching destination “1” before destination “2”.

A more difficult requirement is for the placement and routing tools to ensure isochronic constraints. For the placer, we require that all destinations of a symmetric isochronic fork be placed so 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 so 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 allowing it to try different fork positionings. In practice, fork constraints are almost always met.

Routing of these 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 to 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 both of these destinations. Then, instead of routing to the function blocks of the destinations, we route to these fork points. The Triptych router routes individual source-sink pairs by using a straightforward shortest-path algorithm, which maintains a queue of the neighbors of short paths found so far, and repeatedly removes the shortest neighbor from this queue. Once a fork point is reached we calculate the cost for routing from this point to each of the destinations of the symmetric isochronic 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 our work on performance 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 destinations as a separate signal. A similar extension works for the router, with the addition that the new signal is routed not from a function block output, but instead from wherever the signal enters the earlier destination's RLB.

By leveraging off of the work we have done for Triptych, we have developed in a fairly short time an integrated toolset for doing placement and routing of asynchronous circuits with isochronic forks. We have not yet extended these tools to handle bundled data nor inserted delays.

5. Conclusions and Future Work

As discussed in this paper, the Montage system represents an integrated FPGA architecture and software mapping system for implementing asynchronous circuits. The architecture provides many of the features necessary for proper asynchronous designs, including hazard-free logic and routing, elements capable of implementing and initializing asynchronous stateholding elements, mutual-exclusion elements, and a simple, coherent routing structure that accommodates the timing constraints of asynchronous methodologies. Our mapping software provides much of the support necessary to map Quasi-Delay-Insensitive circuits to the Montage architecture, including placement and routing tools that respect isochronic fork assumptions.

The development of an FPGA for asynchronous circuits opens up several new avenues of exploration. The entire process of mapping for FPGAs must be re-evaluated for this domain. Most obviously, placement algorithms must

take into account the constraints generated by bundled data and inserted delays, and routers must ensure these constraints are met. Decomposition tools must also be developed for properly breaking down circuit elements into sizes accommodated by the target FPGAs.

Acknowledgments

This research was funded in part by the Defense Advanced Research Projects Agency under Contract N00014-J-91-4041. Gaetano Borriello and Carl Ebeling were supported in part by NSF Presidential Young Investigator Awards. Steven Burns was supported in part by an NSF Young Investigator Award.

References

- [Algotronix91] Algotronix Limited, "CAL1024 Preliminary Datasheet", 1991.
- [Brunvand89] E. Brunvand, R. F. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits", *Proceedings of ICCAD'89*, pp. 262-265, 1989.
- [Brunvand91] E. Brunvand, "Implementing Self-Timed Systems with FPGAs", International Workshop on Field-Programmable Logic and Applications, Oxford, 1991.
- [Chu87] T. A. Chu, "Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications", M.I.T. Tech. Rep. MIT/LCS/TR-393, June 1987.
- [Concurrent91] Concurrent Logic, Inc., "CFA6006 Field Programmable Gate Array", March 1991.
- [Ebergen89] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, Centre for Mathematics and Computer Science, Amsterdam CWI Tract 56, 1989.
- [Hauck92] S. Hauck, G. Borriello, C. Ebeling, "Triptych: An FPGA Architecture with Integrated Logic and Routing", Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems, March 1992.
- [Hollaar82] L. A. Hollaar, "Direct Implementation of Asynchronous Control Units", *IEEE Transactions on Computers*, vol. C-31, No. 12, pp. 1133-1141, Dec. 1982.
- [Kishinevsky92] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, V. I. Varshavsky, "On Self-Timed Behavior Verification", *Proceedings of TAU'92*, March 1992.
- [Kung92] D. S. Kung, "Hazard-non-increasing Gate-level Optimization Algorithms", *Proceedings of ICCAD'92*, pp. 631-634, 1992.
- [Marple92] D. Marple, L. Cooke, "An MPGA Compatible FPGA Architecture", First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays, Berkeley, 1992.
- [Martin90] A. Martin, "Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits". In C. Hoare, "UT Year of Programming Institute on Concurrent Programming", Addison-Wesley, Reading, MA, 1990.
- [Nowick91] S. M. Nowick, D. L. Dill, "Synthesis of Asynchronous State Machines Using a Local Clock", in *Proceedings of ICCD*, pp. 192-197, 1991.
- [Rosenberger88] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, T.-P. Fang, "Q-Modules: Internally Clocked Delay-Insensitive Modules", *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1005-1018, 1988.
- [Siegel93] P. Siegel, G. De Micheli, D. Dill, "Automatic Technology Mapping for Generalized Fundamental-Mode Asynchronous Designs", *Proceedings of DAC'93*, 1993.
- [Sutherland89] I. Sutherland, "Micropipelines", *CACM*, Volume 32, Number 6, June 1989.
- [Unger69] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York NY: Wiley-Interscience, 1969.
- [Xilinx92] Xilinx, Inc., "The Programmable Gate Array Data Book", 1992.