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

The Triptych FPGA Architecture

Gaetano Borriello, Carl Ebeling, Scott Hauck, Steven Burns

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

Abstract

Field-programmable gate arrays (FPGAs) are an important implementation medium for digital logic. Unfortunately, they currently suffer from poor silicon area utilization due to routing constraints. In this paper we present Triptych, an FPGA architecture designed to achieve improved logic density with competitive performance. This is done by allowing a per-mapping tradeoff between logic and routing resources, and with a routing scheme designed to match the structure of typical circuits. We show that this yields a logic density improvement of up to a factor of 3.5 over commercial FPGAs, with comparable performance. We also describe Montage, the first FPGA architecture to fully support asynchronous and synchronous interface circuits.

1 Introduction

Field-programmable gate arrays (FPGAs) have quickly become an important medium for the implementation of digital logic. These arrays exploit the increasing capacity of integrated circuits to provide designers with reconfigurable logic that can be programmed on an application-specific basis. This drastically increases flexibility in both the design process and the final artifact by permitting one board-level design to perform many functions, or to be upgraded in the field.

Almost all of the FPGAs currently available - and certainly all of the dominant ones - are based on a strict separation between logic and routing resources which pervades from the architecture itself to the tools employed in mapping designs. This closely parallels the development of integrated circuit gate arrays, arrays that utilize a few metal layers for customization. A similar distinction was made between logic cells and the routing resources that interconnect them. The strict separation was too confining, eventually leading to the sea-of-gates approach. Conceptually, the difference is that in a sea-of-gates the split between logic and routing area can be made on a per-mapping basis. This permits applications with regular logic structures to more efficiently utilize silicon area, while still permitting the use of many wires (at the expense of logic) for random logic circuits.

FPGAs are at a similar point today. As with mask-programmable gate arrays, an ever larger proportion of the silicon area is being devoted to routing resources to ensure that more and more designs are routable. Furthermore, the logic cells are becoming ever more complex, attempting to perform coarser-grain functions and lighten the load on the routing resources, but often end up being under-utilized. As was the case for gate arrays, it is now time to evaluate the logic/routing tradeoff in FPGA architectures.

We have developed the Triptych FPGA architecture, which can be viewed as an FPGA in the sea-of-gates style. Triptych addresses the two fundamental efficiency problems with current FPGAs: increasing routing area and decreasing cell utilization. The innovations include the flexible allocation of logic cells to either logic or routing functions, an array structure that more closely matches the wide shallow structure of most logic functions, and fine-grain cells that can be connected to form larger structures through short, fast local wires.

The rest of the paper is divided into four major sections. Section 2 provides the details of the Triptych architecture[1] and explains the rationale underlying the design decisions. Section 3 describes some variations on the base architecture, including the first FPGA to fully support the implementation of asynchronous systems. Section 4 completes the body of the paper by presenting a methodology for comparing FPGA architectures, and demonstrates Triptych's advantages. Finally, section 5 finishes with some conclusions.

[1] Tools for automatically mapping designs to the Triptych FPGA (as well as others) are described in a companion paper "Mapping Tools for the Triptych FPGA".

2 The Triptych Architecture

The overall goal motivating the development of the architectures described in this paper was to reduce the significant cost paid for routing in standard FPGAs. The approach taken is twofold. First, instead of having a strong separation between logic and routing resources, with the percentage of each fixed in the architecture, the resources are combined in a way that allows the tradeoff of logic and routing resource on a per-mapping basis. This is done by replacing the logic blocks of standard architectures with Routing & Logic Blocks (RLBs), which perform both logic and routing tasks. The second modification is to match the structure of the logic array to that of the target circuits, rather than providing an array of logic cells embedded in a general routing structure. Most circuits are wide and shallow, with large fan-in/fanout trees (figure 1). By matching the physical structure to this logical structure, we reduce the amount of "random" routing that is otherwise required.

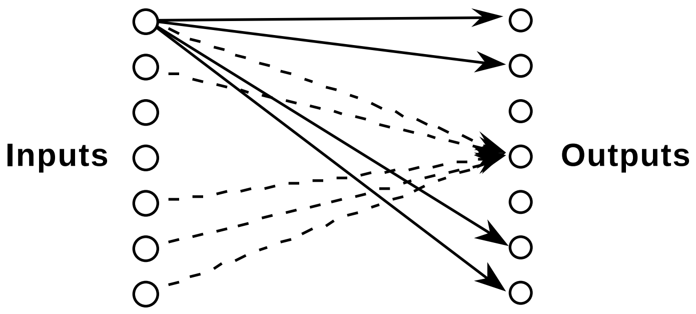

Figure 1. View of a multi-level combinational logic circuit as interleaved fanin/fanout trees.

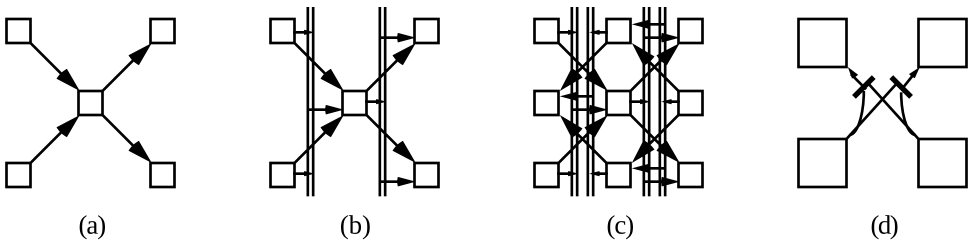

Figure 2. The overall structure of the Triptych FPGA shown in a progression of steps. The basic fanin/fanout structure (a) is augmented with segmented routing channels (b) attached to a third RLB input and output. The structure (c) is obtained by merging two copies of (b), with data flowing in opposite directions in the two copies. Shown in (d) are the connections between the two copies at diagonal crossings.

The Triptych routing structure is shown in figure 2. Short, fast connections are provided between cells in a checkerboard pattern, with signals flowing from left to right. This basic structure is augmented with segmented routing channels between the columns that facilitate larger fanout structures than is possible in the basic array. Finally, two copies of the array, flowing in opposite directions, are overlaid. Connections between the planes exist at the crossover points of the short diagonal wires. It is clear that this array does not allow arbitrary point-to-point routing like that associated with the Xilinx FPGA. However, we claim that this array matches the form of a large class of circuits, and that a mapping strategy that takes this structure into account can produce routable implementations[1]. Such an approach will use the fast diagonal connections for critical paths, while less critical signals can be routed longer distances via segments for vertical movement, and unused RLB inputs for horizontal movement.

2.1 RLB structure

A logical schematic of the Triptych RLB is show in Figure 3. As can be seen, the cell is designed to handle both function calculation and signal routing simultaneously (hence the name *routing and logic block*, RLB). It takes input from three sources and feeds them into a function block capable of computing any function of these three inputs, and the output can then be used in latched or unlatched form. The RLB's three outputs can choose from any of the three inputs and either the latched or unlatched version of the function block output. One last feature is the

[1] Descriptions of such mapping tools can be found in the companion paper "Mapping Tools for the Triptych FPGA"

loopback from the master/slave D-latch, which enables the function to be dependent on its previous value. This is included for state machine implementation, although it may be used to output both the latched and unlatched versions of the function block. Again, only one of the inputs and one of the outputs can be connected to the vertical wires; the other two of each type are connected to the local diagonal wires.

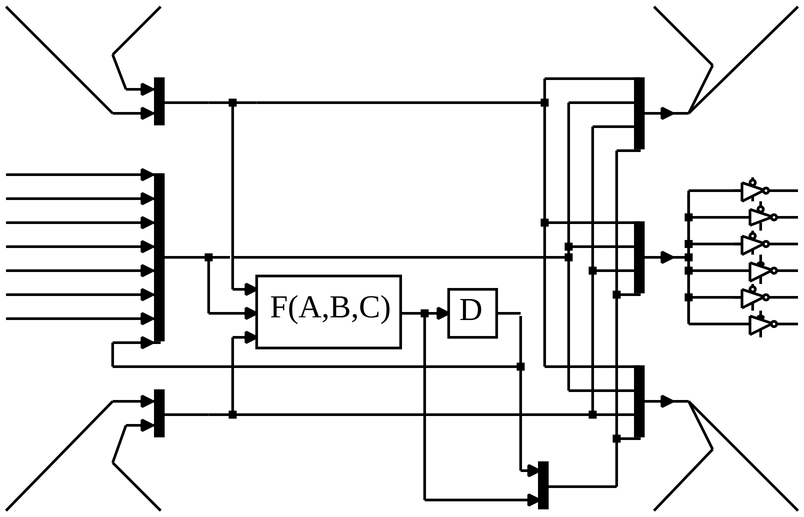

Figure 3. Triptych routing and logic block (RLB) design. The RLB consists of: 3 multiplexors for the inputs, a 3-input function block, a master/slave D-latch, a selector for the latched or unlatched result of the function, and 3 multiplexors for the outputs.

A Triptych RLB is capable of performing both function calculation and routing tasks simultaneously, which leads to several different uses of the RLB (see Figure 4). The three most obvious are: (a) a routing block with each input connected to one of the outputs; (b) a splitter with one of the inputs going to two or three of the outputs; and (c) as a function calculator with the three inputs going to the function block and the function going out the outputs. However, there are two important classes of hybrids that help produce more compact designs. The first comes from the observation that in blocks used to calculate a three-input function, the function block value will most likely not go out all three outputs, and one or two of the input signals could be sent out the unused output connection(s), as in (d). Secondly, a function of two inputs can be implemented by making the function insensitive to the third input, thus allowing the unused input to be used to route an arbitrary signal, as in (e). An important observation is that the RLBs will never need to be used for one-input functions (i.e., an inverter), since any output signal will only be used either as an input to another arbitrary function block, where the inverter could be merged into the function computed, or to an output pin, where an optional inversion can be applied.

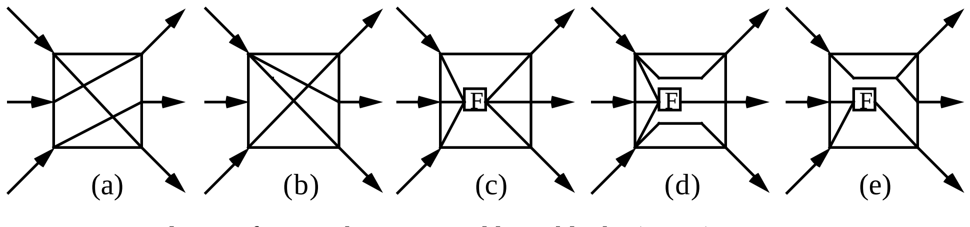

Figure 4. Five typical uses of Triptych routing and logic blocks (RLBs).

As was shown earlier, the Triptych FPGA has no global routing for moving signals horizontally. Instead, there is a heavy reliance on unused RLBs and unused portions of RLBs to perform these routing tasks.

2.2 Interconnection

The Triptych RLBs are connected by three separate interconnection schemes. The first is for horizontal interconnect, and is accomplished through the RLBs as described above. The second is for local high-speed communication between neighboring RLBs and is achieved through “diagonals”. The detailed structure of the diagonals is shown in Figure 5. They allow outputs to be sent to the four RLBs immediately above and below them; two in the next column, which flow in the same direction, and two in the same column, which flow in the opposite direction.

Diagonals are important for two reasons. Diagonals permit the construction of multilevel functions of more than three inputs without the speed penalty of general-purpose interconnect. They also allow signals to change direction so that circuits can be more tightly packed and feedback can be provided for the implementation of sequential logic.

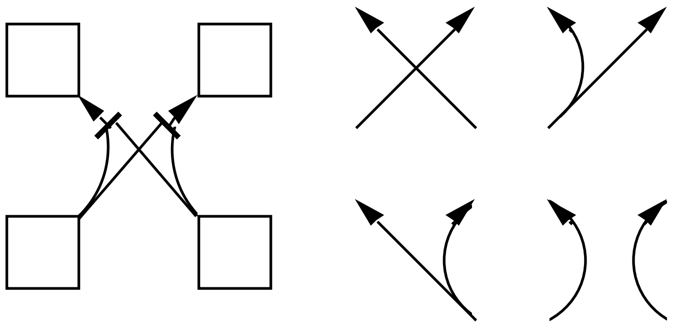

Figure 5. Schematic view of a pair of diagonals and the routing combinations they allow (implemented by a multiplexor at each diagonal input).

The third type of interconnect is used for longer range connections and large fanout nodes. It is implemented as a set of segmented "channel wires" between adjacent columns (see Figure 6) that connect middle outputs of RLBs to the middle 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-cell RLB routing and a seventh to carry a pin input. The 6 inter-cell tracks are broken up into two tracks each of 8, 16, and 32 RLB high segments.

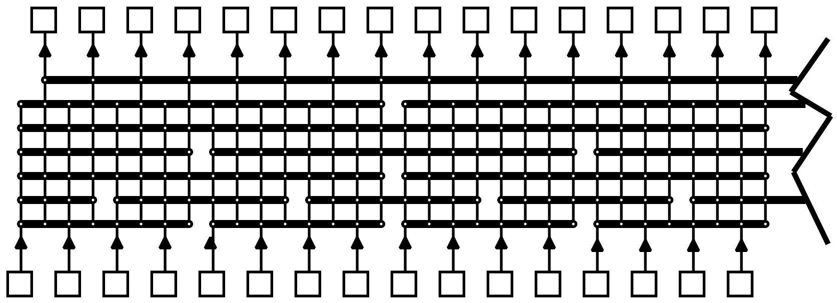

Figure 6. Top half of a segmented channel (on its side). The bottom half is a mirror image of the top.

One last important feature of the interconnect structure is how it handles the array borders. Since there are no RLBs beyond the right and left edges for the channel wires to route to, the channels on the edges tie the two directions of RLBs together. This way of handling the border cases leads to a different way of looking at the array, namely as a cylinder of RLBs. If the diagonals leading to the opposite direction of RLBs were cut except for those at the edges, the chip would appear to be a folded cylinder of RLBs. In fact, it is often helpful to think of the array as containing many smaller cylinders, with each of these cylinders implementing a smaller subcircuit. For example, a six by six square of RLBs can be broken off from the rest of the array and considered to be a cylinder three RLBs high and twelve RLBs in circumference. This is not quite true, since the vertical channel for the left and right edges of the original six by six square will be unusable on the cylinder, but it can still be a useful abstraction for mapping. In fact, the current Triptych chip is an array of 64x8 RLBs, designed to yield a 32x16 cylinder.

The array borders are also important because it is here that most of the chip I/O is performed. As shown in figure 7, neighboring RLBs on a chip edge share I/O pins, with the external inputs and outputs flowing on the unused diagonal connections. The control signal for bi-directional pins comes from the vertical segmented channels, with a mux (not shown) selecting one of the adjacent (non-IO) channel wires. Note that this results in 6 possible settings. With two additional configurations of purely input and purely output, three programming bits can chose the pin direction and tristate signal. Although the pin connection scheme may seem counterintuitive, it is constructed to allow arbitrary RLBs to receive two input signals, or send two output signals, while still allowing neighbors to use an I/O pin. An example of this can be found later in the traffic light controller mapping. The pins are primarily connected to diagonals instead of the vertical interconnect since at the array borders half of the diagonal connections are not used by the array, while all of the vertical segmented channels are connected. Note that the vertical segmented channels have somewhat overloaded functionality, serving both to carry inter-RLB signals, and also tristate signals to the I/O pins. At the array top and bottom, where diagonals have no neighbors to connect to, these diagonals are connected together as shown in figure 7.

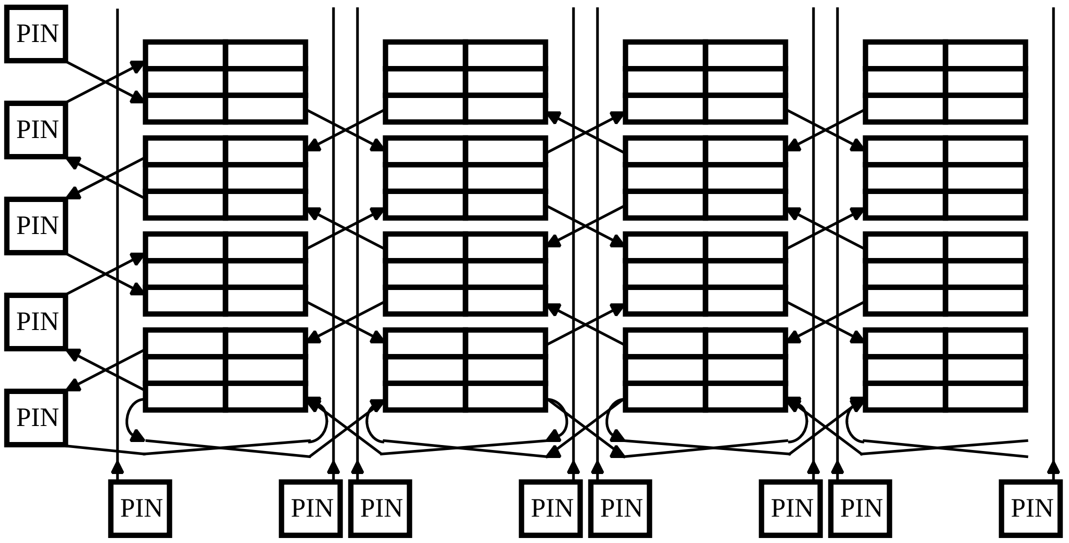

Figure 7. One corner of the Triptych array, showing pin and diagonal connections at the array periphery

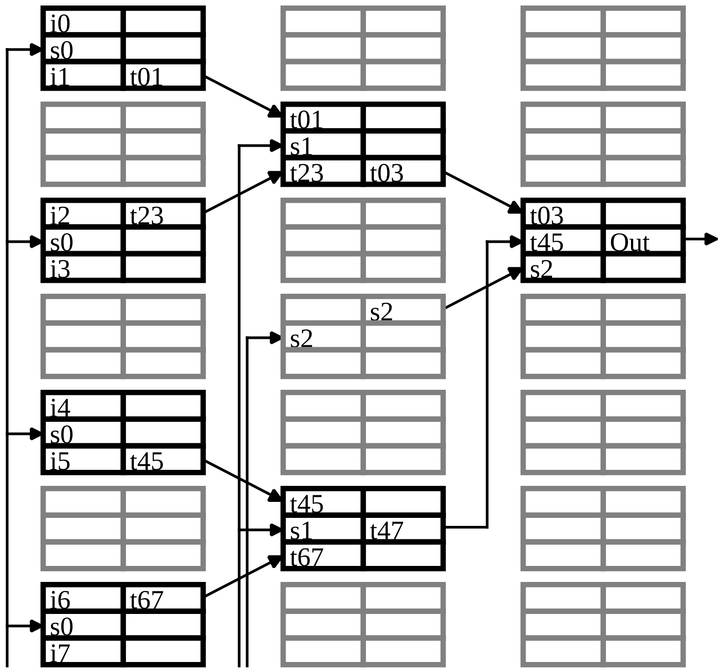

Figure 8. Triptych mapping of an 8:1 multiplexor.

2.3 Using Triptych

In this section we present two examples of circuits mapped to Triptych. The purpose of these examples is to demonstrate the constraints on routing and how multilevel logic circuits do indeed map to the physical structure provided by this FPGA. In these examples, each RLB is shown as a cell with three input entries and three output entries. Each entry indicates an incoming or outgoing signal. Note that each block may create a new signal by computing a logic function over the inputs. If a cell is unused, or only used for routing, it will be shown in gray. Diagonals, reverse diagonals and channel wires are only shown if they are actually used in the mapping.

The first example is an 8:1 multiplexor (figure 8). It has 8 data inputs, *i0* to *i7*, and three control inputs, *s0* to *s2*. The data inputs enter from the left edge, while the control bits come in on the vertical segmented channels. The output *Out* is generated in the rightmost cell. As this example shows, the diagonal connections between cells are able to handle most of the communication. However, the two intermediate values *t03* and *t47* are generated by RLBs 4 cells apart, requiring either one signal to use the vertical segmented channel, or for the signal to be routed through two intermediate RLBs. Our solution was to place the signal on the vertical segmented channel, and use an earlier RLB as a staging area to transfer the *s2* control signal from vertical segmented channel to the proper diagonal.

```
INPUTS: s1 s2 d1 st SB0 SB1
OUTPUTS: NSB0 NSB1 r1 y1 g1 r2 y2 g2 sd
NSB0 = !st * !r2;           NSB1 = !st * !g1 * !g2;
r1 = NSB0;                  r2 = !st * (SB0 * !9 + !SB0 * 9);
y1 = r2 * 51;               y2 = 53 + 45;
g1 = r2 * !51;              g2 = !st * !r2 * !y2;
sd = 12 + 45;               9  = !SB1 + !d1;
12 = !SB1 * 18;             18 = !SB0 * s2 * !st;
46 = !st * SB0;             45 = s1 * !SB1 * 46;
52 = !d1 * SB1;             51 = s2 * !SB1 + !SB0 * SB1;
53 = 52 * 46;
```

Figure 9. Factored logic equations for the traffic light controller finite state machine.

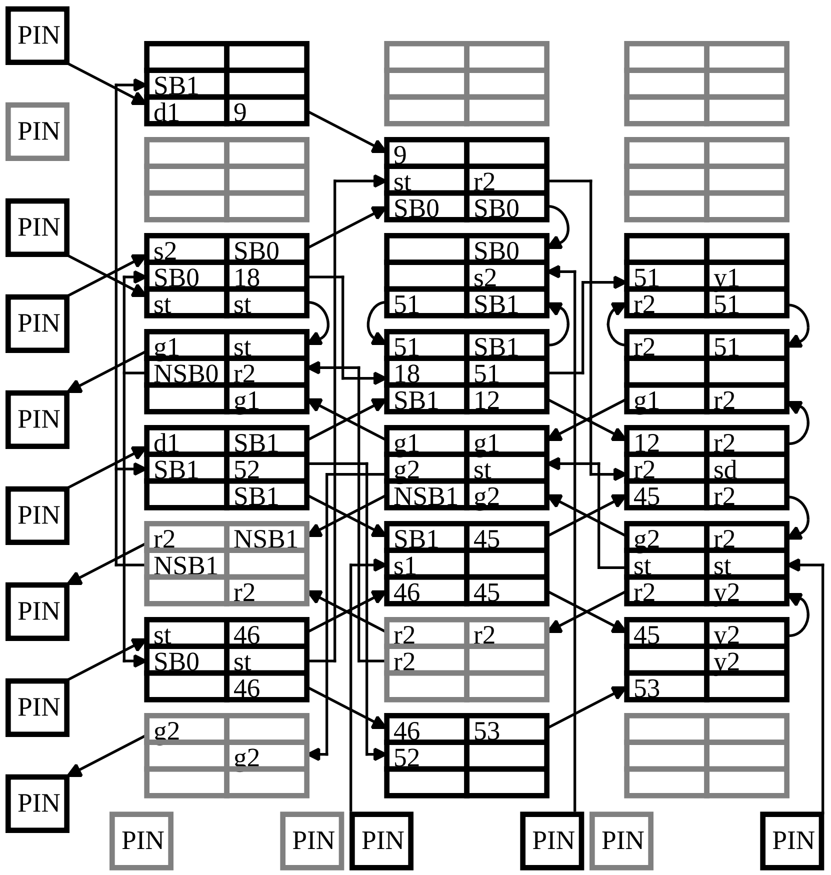

Figure 10. Triptych realization of the traffic light controller.

Figure 9 shows the factored logic equations and Figure 10 the corresponding Triptych implementation for the ubiquitous traffic light example. This example shows that circuit mappings can be very compact if the individual logic blocks are correctly placed. The inputs and outputs of this circuit are all connected at the left and right of the array, except for three signals that use the pin input track of the vertical channels. In this example 16 RLBs are used to compute logic functions, 3 RLBs are used only for routing, and 5 RLBs are left unused (however, these 5 RLBs could be used in neighboring circuits). Also, this circuit is assumed to be placed along the left edge of the chip, so the vertical tracks at that edge are used to connect RLBs in the same column. This is about as compact a Triptych layout as can be achieved for a random logic function.

Resource Used	Delay
RLB	1.6ns
Function Block	additional 2.2ns
Channel Wire	2.5-3.7ns

Table 1. Speed of important features, estimated using HSPICE with parameters for the 1.2µm CMOS n-well process available from MOSIS.

2.4 Triptych Performance

The speed of a path in a Triptych mapping can be calculated from the numbers given in table 1. For example, a path using 4 RLBs, 2 for routing and 2 for function calculation, and 1 channel wire would take 13.3 to 14.5 nanoseconds (4×1.6 + 2×2.2 + 2.5 to 3.7 = 13.3 to 14.5). Note that being able to use such a simple speed calculation method is due both to the simplicity of the interconnect and also to the design philosophy of “independence of paths”. Simply put, “independence of paths” means that gate logic is used in place of switch logic in much of the FPGA, making signal propagation insensitive to the amount of fanout contained in the current mapping.

3 Architectural Variations

The structure detailed above differs little from Triptych’s initial conception. With the experience we have gained with the original Triptych architecture, with placement and routing tools, and with asynchronous and interfaced synchronous circuits, we have extended this architecture. These extensions can be grouped into two classes, 4-input 4-output RLBs, and asynchronous support, with the architecture for asynchronous circuits named “Montage”.

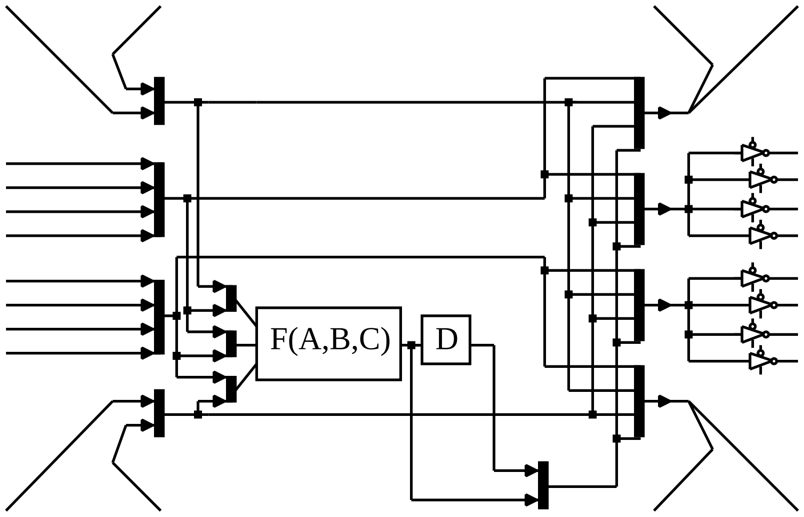

Figure 11. A 4-input, 4-output RLB. The segmented channel input is split into two separate inputs, to allow for greater routeability. Muxes are also added to the function block inputs to choose three of the four RLB inputs for function calculation

3.1 4-Input RLBs

The major bottleneck that restricts circuit densities in the Triptych array is the 3-input limit on RLBs. If we decompose circuits to mostly 3-input logic functions, there is no space to route unrelated signals through these RLBs, requiring placements to leave large gaps of unused cells for routing. If we decompose circuits into more 2-input functions, we greatly increase the number of function blocks required to handle the logic. The solution we have adopted is to add a fourth input and fourth output to each RLB (figure 11), with this signal connected to the vertical segmented channels. While this does increase the amount of logic and programming bits in each RLB, this area penalty is more than made up for by denser circuit mappings.

Although it is tempting to also increase the function block to 4 inputs, this would require a much larger increase in RLB area, and we would again face problems in routing signals. However, one possible extension is to build the architecture with a 4-input lookup-table, but only use 4-input functions on the circuit's critical path(s). In this way, we retain the routing advantage of extra RLB input, yet can greatly speed up critical paths by reducing their logic depth. Similarly, the basic Triptych architecture built with 3-input 3-output RLBs could only use 3-input functions on the critical paths. However, the switch from 3-input to 2-input functions entails a much higher increase in the number of lookup tables required than does the switch from 4-input to 3-input RLBs, and is a less attractive option.

3.2 Montage

While Triptych, as well as most other commercial FPGAs, is optimized to handle synchronous circuits, circuits whose sequential behavior is moderated by clocked latches on all feedback paths, there is little support in the FPGA community for asynchronous circuits. Unfortunately, an asynchronous designer cannot simply make do with a synchronous FPGA because the constraints on the implementation medium are much more stringent for asynchronous circuits. Note that work has been done on mapping asynchronous circuits to standard FPGAs [Brunvand91], but have difficulty handling the required timing assumptions and mutual exclusion operations. More troublesome than this is the fact that synchronous interfaces, a significant portion of the "glue logic" that is widely considered as the strongest candidate for mapping to FPGAs, have requirements very similar to asynchronous circuits -- requirements not met by today's FPGAs.

Asynchronous circuits and synchronous interfaces have no reference clock which synchronizes all of the incoming signals, and thus are sensitive to every signal transition on their inputs. Thus, any medium implementing these circuits must be carefully designed to avoid all hazards. While in practice this is not difficult, it is a step that must be taken to handle these classes of circuits, but which is safely ignored when implementing synchronous circuits. The most troublesome portion of designing to avoid hazards is the proper choice and implementation of the logic element. It turns out that the Triptych function block (figure 14) is in fact hazard-free. This is because during a single input change the paths to the corresponding programming bits for the before and after states briefly conflict, and no other paths are connected. A similar situation occurs during a multiple-input change, with the before, after, and any possible intermediate states possibly conflicting. While the intermediate states might cause a hazard, this is an unavoidable (function) hazard, and represents a fault in the original circuit. One other important feature of this function block is the fact that every internal node is always driven to a valid value, avoiding any concern for charge-sharing. Other parts of the circuit could suffer from charge-sharing, and thus the layout of the Montage architecture has been carefully constructed to avoid large undriven capacitances.

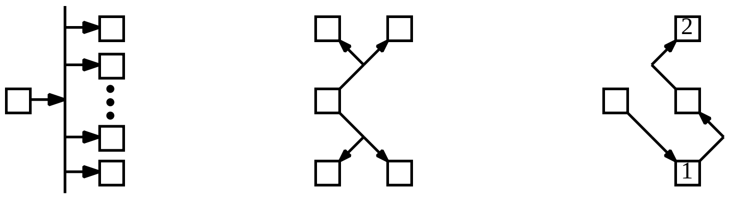

Figure 12. Placement of a symmetric isochronic fork on an interconnect line (left) and on diagonals (center), as well as an asymmetric fork (right) reaching destination "1" before destination "2".

Another feature of the Montage FPGA that is very important for the support of asynchronous circuits is the overall routing structure. Some asynchronous design methodologies, including [Martin89, Ebergen89], make assumptions about the delays incurred in wires. Specifically, they assume that some wires are asymmetric isochronic forks, where a signal must reach one destination of the fork before it reaches the other, or symmetric isochronic forks, where the signal must reach all destinations at the same time. In many FPGAs, routing delays are very unpredictable, and it would be difficult (if not impossible) to ensure the isochronic constraints in these FPGAs. In Triptych, the isochronic constraint is very easy to meet. For asymmetric isochronic forks, we simply route a signal to the earlier destination's RLB, and route out from that RLB to the later destination. In this way, we ensure the isochronic constraint. For symmetric isochronic forks, there are several RLB placements that respect the isochronic constraint. For example, a signal that moves from a vertical segmented channel directly into the function blocks of adjacent RLBs will have very little skew between the different destinations. Isochronic forks can also take a shared diagonal to two RLBs, or move from an intermediate RLB to its four neighbors on diagonal connections, again with very little skew. Thus, the Montage FPGA, by directly duplicating the Triptych routing structure, provides a medium for guaranteeing isochronic constraints. Note that this does put some constraints on how circuits can be mapped.

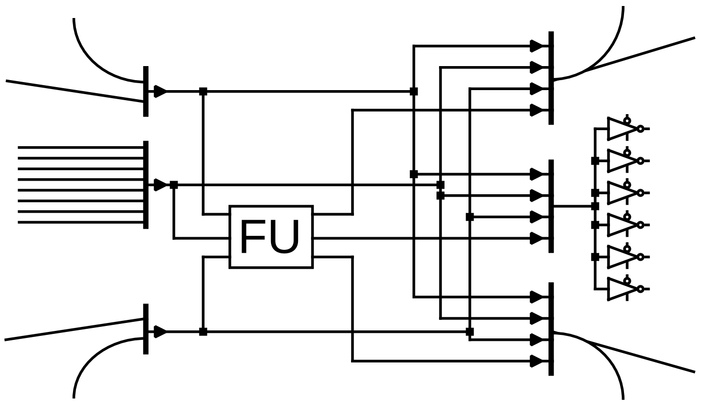

Figure 13. The Montage RLB. This differs from the standard Triptych RLB by removing the feedback path, as well as abstracting the function calculation unit into a separate entity.

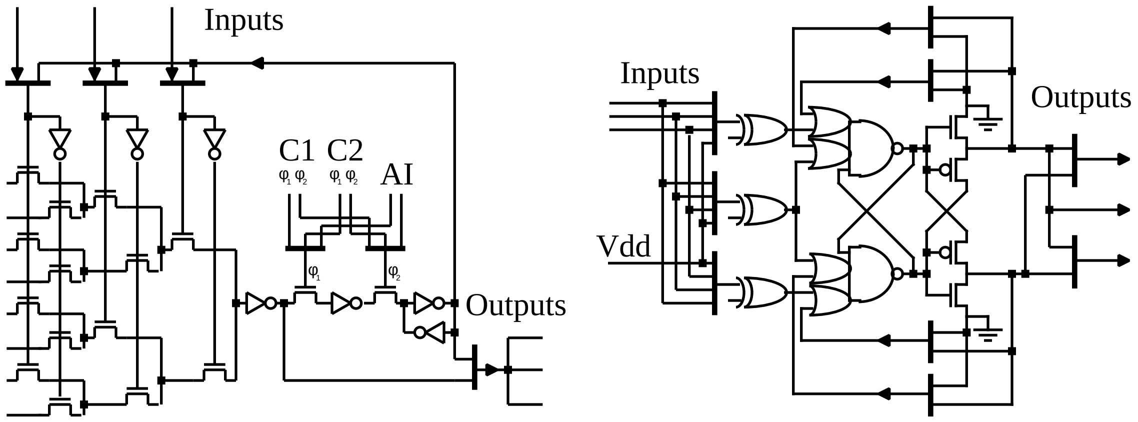

Figure 14. The two types of functional units: the logic block (left) and the arbiter unit (right).

Two requirements for Montage have altered the structure of the basic Triptych RLB: the need for multiple clocks for interfaced synchronous circuits, and the need to implement and initialize asynchronous stateholding elements. Handling multiple clocks is simply a matter of having more than one set of clock lines to the latches, which in fact is what is done for Montage. Asynchronous stateholding elements are more troublesome, because they require a very tight feedback path for the previous state. Specifically, elements such as an asynchronous S-R latch or a Muller C-Element (an element whose output is 1 when all inputs are 1, 0 when all inputs are 0, and held to its previous value in all other cases) can be implemented as a combinational function with one input being the element’s current output. However, we must ensure that this path from output to input is faster than other paths in the system so that the element stabilizes before a new input can arrive. While the feedback path in Triptych could probably handle these requirements, it would mean that all of the RLBs with asynchronous stateholding elements would be unable to receive a signal from the vertical segmented channels, greatly restricting the routing available in Montage.

To support asynchronous stateholding elements in Montage we have adopted the structure shown in figure 14, with a very fast feedback path which does not interfere with any of the route-through capability of the RLB. In order to initialize these stateholding elements, we use Triptych’s ability to load a value onto its D-latch to initialize this path. During startup, all feedback paths are held to the loaded value, and once enough time has passed for the entire FPGA to settle to a proper start-state, the feedback path is switched to the non-latched path, allowing the FPGA to run freely. The combination of the feedback path and the initialization scheme means asynchronous stateholding elements such as an S-R flipflop or a 2-input C-element can be implemented in a single cell, yielding very compact mappings.

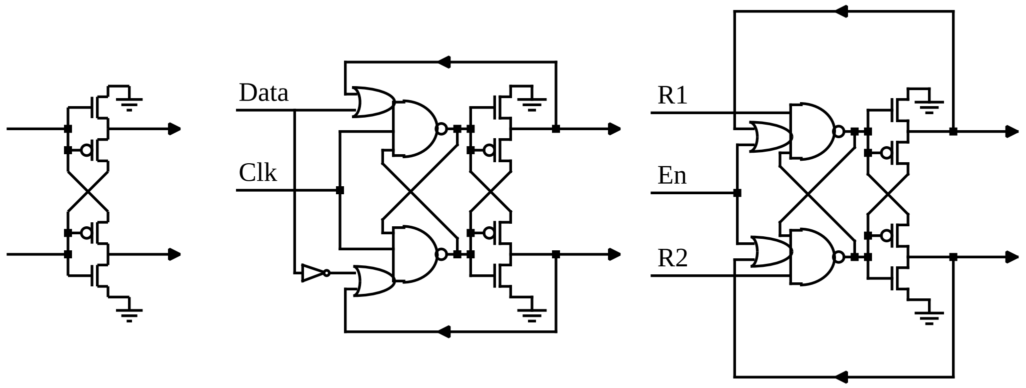

Figure 15. A mutual exclusion element (left), and a synchronizer (center) and enabled arbiter (right) built from it.

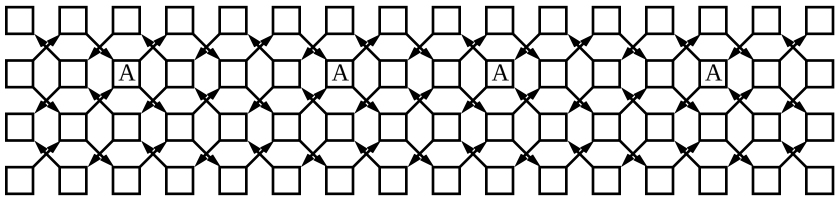

Figure 16. Distribution of arbiter (labeled "A") blocks throughout the Montage array. The complete array is built by stacking vertically the tile shown above.

The final necessary feature for Montage was some way of performing robust arbitration and synchronization. For both synchronous interfaces and asynchronous circuits there is a need to reliably determine the arrival order of two signals (an arbiter) and sample a signal at a given time (a synchronizer). While standard combinational logic can approximate these behaviors, there is always a chance that the output of such as approximation is incorrect, or possibly even oscillating. Correct implementations can be built with the aid of a mutual exclusion element, an element whose correct functioning depends on analog effects. As shown in figure 15 left, the element has two inputs and two outputs, with an output raised in response to its corresponding input. What is special about this element is the guarantee that at most one of the outputs will be raised at a time, without visible oscillations or hazards.

To support arbitration and synchronization in Montage we could attempt to insert a mutual exclusion element into each cell, or possibly place one by each I/O pad. However, each of these solutions would entail a large area increase, and the I/O pad solution hurts circuit mappings where the arbitration or synchronization occurs in the middle of the circuit instead of at the edges. The solution we adopted is to choose a small subset of the RLBs in the Montage array and replace the function block with an arbiter unit (figures 15 and 16). The arbiter was carefully designed to have about the same size as the logic it replaces, as well as use the same number of programming bits, meaning that a logic RLB and an arbiter RLB can have the same footprint. Also, since the Triptych/Montage FPGA family expects to have a portion of the RLBs used only for routing, and since an arbiter RLB has exactly the same routing capabilities as a logic RLB, mappings that do not use arbiters or synchronizers will have little or no area penalty due to the inclusion of arbiter RLBs.

3.3 Using Montage

To illustrate how well typical asynchronous elements map to Montage, we present all of the basic elements used in Ebergen's synthesis methodology [Ebergen89] in figure 17. Most of the elements fit compactly into a single RLB, and can be initialized with the built-in circuitry. The Toggle element is a two-output element, and thus required two RLBs (with one extra signal that must be routed outside this mapping). The Sequencer, which is Ebergen's two-phase mutual-exclusion element, requires 10 RLBs, primarily for a two-phase to four-phase converter.

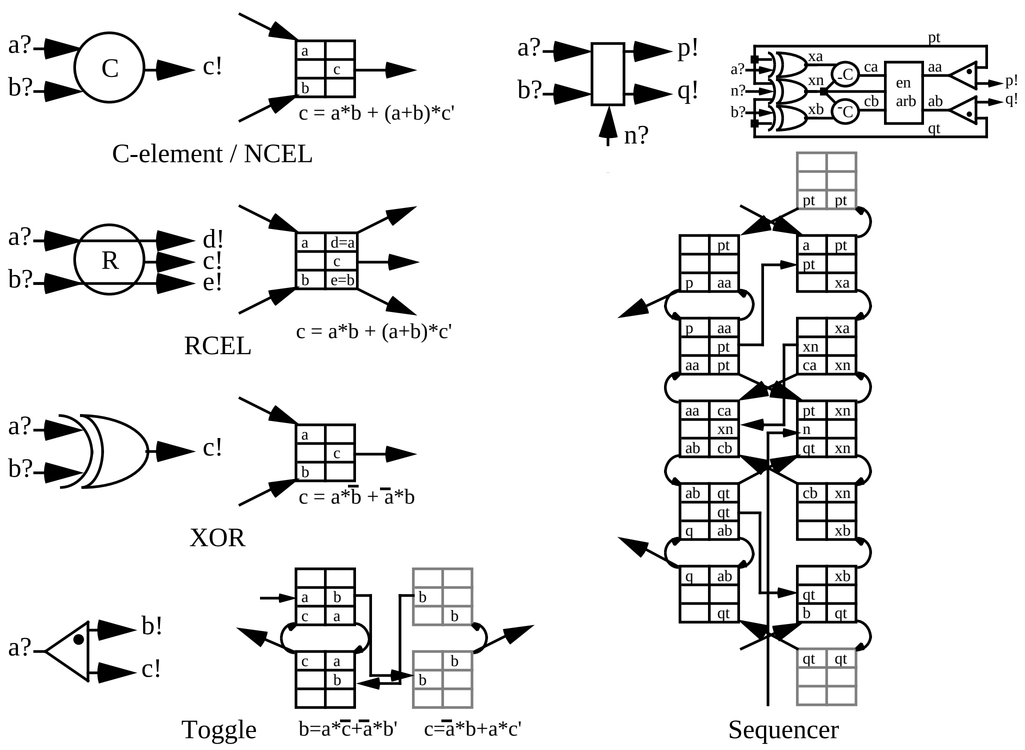

Figure 17. Ebergen's basic elements, mapped to the Montage FPGA. Included at top right is a circuit diagram for the implementation of the Sequencer. Other elements have logic equations for the outputs.

4 Area comparisons with other FPGAs

In this section, we present a comparison of the area efficiency of four different FPGA architectures, based on several representative examples. The FPGAs compared are CLI's CFA [Concurrent91], Algotronix's CAL1024 [Algotronix91], the Xilinx 3000 series [Xilinx91], and Triptych (with 3-input, 3-output RLBs). The example circuits were chosen across a wide range of domains, rather than a specific one for which an architecture might have been optimized (e.g., bit-serial computations). The circuits were mapped by hand and only optimized for area. Automatic tools were not used as this would make the comparison as much about the CAD algorithms as the architectures themselves. Even after eliminating these two factors, domain and CAD support, performing an architectural comparison between FPGAs is difficult because of complicating factors due to particular implementations. The most significant of these include the manufacturing process, the design rules employed, and the quantity and quality of designer effort. These factors make the straightforward approach of comparing actual chip area inadequate for purposes of architectural comparison. Simply using bits per cell also fails because it assumes that all FPGAs have equal per bit logic complexity, which is not true in current FPGAs.

	CFA	CAL	Xilinx	Triptych
Normalization factor	0.45	0.45	9.1	1.0

Table 2. Area normalization factors. Triptych is used as the baseline.

	CFA	CAL	Xilinx	Triptych
Technology feature size	1.0	1.5	1.25	1.2
Scaling method	Linear	Linear	Quadratic	—
Normalization factor	1.2	0.8	0.92	1.0

Table 3. Speed normalization figures and the data used to obtain them. A 1.2µm process is used as a baseline.

	size	Triptych area	CFA factor	CAL factor	Xilinx factor
Systolic					
Single-flow string compare (# of bits per character)	2	7	3.1	1.1	3.9
	4	10	2.9	1.1	3.6
	8	16	2.7	1.0	3.7
Double-flow string compare (# of bits per character)	2	20	2.3	1.0	2.8
	4	24	2.4	1.0	3.4
	8	36	2.2	1.0	3.8
FIFO (width in bits)	8	58	—	1.1	2.4
	16	90	—	1.0	2.3
	32	154	—	1.0	2.3
	64	282	—	.9	2.3
Arithmetic					
Counter (width in bits)	8	24	.9	.9	2.7
	16	48	.9	.9	2.7
	32	96	.9	.9	2.7
Adder (width in bits)	8	40	1.6	.9	3.7
	16	80	1.6	.9	3.7
	32	160	1.6	.9	3.7
Linear-growth bit-parallel					
Comparator (width in bits)	8	8	2.3	.9	4.5
	16	16	2.3	.9	4.6
	32	32	2.4	.9	4.6
	64	64	2.4	.9	4.6
Shift Register (number of bits)	8	8	2.8	2.3	4.5
	16	16	2.7	2.3	4.6
	32	32	2.7	2.3	4.6
	64	64	2.7	2.3	4.6
Exponential-growth bit-parallel					
Decoder (inputs:outputs)	2:4	4	—	1.0	4.5
	3:8	10	—	1.0	3.6
	4:16	22	—	1.1	3.3
	5:32	46	—	1.4	3.6
Multiplexor (inputs:outputs)	4:1	3	2.7	1.7	4.7
	8:1	8	2.1	1.6	4.0
	16:1	18	2.1	1.7	3.8
	32:1	41	2.0	1.8	3.4
Random Logic/FSM					
s27		11	—	—	4.2
s208		59	—	—	2.9
traffic light controller		24	—	—	3.0
Geometric Means			2.1	1.2	3.5

Table 4. Normalized area of circuits hand-mapped to four FPGAs. For each circuit, the number of Triptych cells used and the normalized size of the mappings for the other three FPGAs is listed (e.g., the Triptych s27 mapping is 11 cells, while the Xilinx mapping is 4.2 times as big, or the equivalent of 46 Triptych cells).

Appendix A contains a methodology for comparing the area-efficiency of FPGA architectures which compensates for most of the effects just mentioned. This approach allows us to normalize the area of an FPGA tile (a logic block an its associated routing) so that mappings to different FPGAs can be compares. The resulting scaling factors are contained in table 2. Normalization is also necessary for speed comparisons. We use the delay values provided by the manufacturers (Xilinx values are for the 3020-50 [Schlag91]), and scale them based on the feature sizes of the processes used. As shown in [Vuillamy91], FPGA logic speed scales linearly with process size, while routing delay scales quadratically. However, since the division of FPGA timings into routing-based and logic-based delays is not always clear, we have scaled CFA linearly from 1.0µm to 1.2µm, and Xilinx quadratically from 1.25µm to 1.2µm, both of which are generous to the given FPGAs. For CAL, which has no long-distance routing, all delays were scaled linearly. Table 3 shows the normalization factors calculated for the FPGAs in question.

4.1 Summary of mapping results

Table 4 shows the results of area comparisons using this methodology for a wide range of circuits. They are collected into the following general categories: systolic, arithmetic, linear-growth bit-parallel, exponential-growth bit-parallel, and finite state machines (random logic). The systolic circuits are two versions of a string comparison circuit that computes the "edit distance" of two strings [Lipton85], the two versions differing on whether both strings move or if one is fixed in place, as well as a FIFO [Guibas82]. The arithmetic circuits chosen are a counter with a 4-stage carry look-ahead and an adder with a 2-stage carry look-ahead. The linear-growth bit-parallel circuits (circuits whose interconnection tends to scale linearly with problem size) include a simple equal/not-equal comparison and a shift register, while the exponential-growth bit-parallel circuits are a decoder and a multiplexor. Finally, for the random logic/FSMs, we chose the two smallest ISCAS benchmarks (s27 and s208), as well as the traffic light controller example. Table 5 shows the results of delay comparisons for the circuits of Table 4. Note that these numbers are for circuits mapped to optimize for area, not delay.

	size	Triptych speed	CFA factor	CAL factor	Xilinx factor
Systolic					
Single-flow string compare (# of bits per character)	2	14.6	2.7	1.8	.9
	4	20.9	2.2	1.7	.6
	8	33.5	1.3	1.6	.6
Double-flow string compare (# of bits per character)	2	26.3	1.9	1.1	.8
	4	31.7	1.6	1.0	.8
	8	64.2	.9	.7	.6
FIFO (controller speed only)	—	34.0	—	3.0	1.5
Arithmetic					
Counter (width in bits)	8	21.2	1.6	1.5	1.0
	16	32.0	1.6	1.6	1.3
	32	53.6	1.6	1.6	1.5
Adder (width in bits)	8	42.3	1.6	1.9	.9
	16	73.9	1.4	1.8	.9
	32	137.1	1.3	1.7	.9
Linear-growth bit-parallel					
Comparator (width in bits)	8	30.4	1.6	1.4	.9
	16	60.8	1.2	1.3	.9
	32	121.6	1.3	1.3	.9
	64	243.2	1.2	1.3	.9
Exponential-growth bit-parallel					
Decoder (inputs:outputs)	2:4	7.9	—	1.3	1.0
	3:8	7.9	—	2.3	1.3
	4:16	19.0	—	1.5	.7
	5:32	24.0	—	1.9	.8
Multiplexor (inputs:outputs)	4:1	7.6	3.5	3.2	1.7
	8:1	13.9	2.7	3.1	1.5
	16:1	17.7	3.0	3.9	1.7
	32:1	26.5	2.9	4.3	1.5
Geometric Means			1.7	1.7	1.0

Table 5. Delay of circuits hand-mapped to 4 FPGAs, in nanoseconds, using normalized delay values. The Triptych column contains actual speed numbers, while the other columns are the factor by which the given FPGA's mapping is slower (or faster) than Triptych's.

Even under the model given above, which for several reasons is biased towards the other FPGAs, Triptych does quite well in both area and speed. For all but the counter, whose CFA mapping is only 10% smaller, Triptych mappings are consistently smaller than CFA mappings, quite often by a factor of 2 or more, and Xilinx mappings are usually three to four times larger. As to CAL mappings, the best are only about 10% smaller than Triptych mappings, with the worst being more than twice as large. At the same time, delays in these circuits are almost always worse, up to a factor of more than 4 in the larger multiplexors. All of these factors indicate that Triptych is a viable architecture. This is especially true since the only FPGA surveyed that comes close to matching Triptych's area efficiency is CAL, which has no global routing facilities. This absence implies that larger, more complex circuits than those surveyed would be more difficult to map to CAL than to Triptych. They will be larger and slower, due to more cells being utilized for routing.

Several more points need to be made about these comparisons. First, the Xilinx mappings are the only ones for which we did not perform the assignment of routing resources. This was due to the complexity of hand-mapping Xilinx routing, and it was simply assumed that the circuit could be routed without leaving unreachable CLBs. This is not always the case in the 3000 series, where large designs often achieve only 50% cell utilization due to the scarcity of resources. Thus, the area numbers for Xilinx should be considered as a lower bound, and the mappings may actually require significantly more space. Since we did not do routing, we have no direct method for measuring the speed of Xilinx circuits. However, by using a routing delay model for Xilinx [Schlag91], we were able to compute estimated routing delay, and these numbers were presented in table 5. Second, since all the circuits were mapped by hand, we were able to use only small examples. No comparisons of large, complex random logic circuits are presented. Third, since FPGAs normally come in families, with the amount of cells in a chip variable, the issue of I/O blocks has been ignored and the circuits mapped to an unbroken plane of cells with no chip edges (except for a few circuits who actually map easier along a chip boundary, in which case an arbitrarily long boundary was assumed).

5 Conclusions

Current FPGA architectures are designed around the philosophy that the architecture must provide a general, arbitrary routing structure, with a strict distinction between logic and routing resources. Unfortunately, this imposes a significant cost, with 90% or more of the chip area dedicated to routing resources. Triptych is a step away from this philosophy, with logic and routing integrated into RLBs, letting the tradeoff between logic and routing be determined on a per-mapping basis. These resources are also optimized for local communication, and are designed to map the structure of typical circuit mappings.

The Triptych philosophy results in an architecture with much better density than standard commercial FPGAs for hand-mapped circuits. Triptych mappings are up to 3.5 times denser than other architectures, and deliver comparable performance. We have also written placement and routing tools, which are described in a companion paper, for the Triptych architecture that are able to take advantage of the features of Triptych. These tools achieve a logic density of 50% across a range of PREP and logic synthesis benchmark circuits. While this density is somewhat less than that of the hand-mapped, regular circuits described here, it reflects our philosophy which shares the RLBs between logic and routing.

Montage takes the Triptych architecture a step further, with an FPGA that completely supports the mapping of asynchronous and interfaced synchronous circuits. This yields an attractive prototyping medium to the researcher and developer of many types of circuits, and demonstrates that FPGAs can be applied to a much broader class of designs than is currently considered.

Acknowledgments

Thanks to Christopher Hébert and the other students of CSE568, Winter quarter 1991 for contributing to the early design phases of the Triptych layout. 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.

[Brunvand91] E. Brunvand, "Implementing Self-Timed Systems with FPGAs", *International Workshop on Field-Programmable Logic and Applications*, Oxford, 1991.

[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.

[Furtek91] F. Furtek, Personal communication, October 1991.

[Guibas82] L. J. Guibas and F. M. Liang, "Systolic Stacks, Queues, and Counters", Second MIT Conference on Advanced Research in VLSI, 1982.

[Kean91] T. Kean, Personal communication, October 1991.

[Lipton85] R. J. Lipton and D. Lopresti, "A Systolic Array for Rapid String Comparison", Chapel Hill Conference on VLSI, 1985.

[Martin89] A. J. Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits", in *UT Year of Programming Institute on Concurrent Programming,* C. A. R. Hoare, Ed. MA: Addison-Wesley, pp. 1-64, 1989.

[Schlag91] M. Schlag, P. K. Chan, and J. Kong, "Empirical Evaluation of Multilevel Logic Minimization Tools for a Field Programmable Gate Array Technology", International Workshop on Field-Programmable Logic and Applications, Oxford, 1991.

[Singh90] S. Singh, J. Rose, D. Lewis, K. Chung, and P. Chow, "Optimization of Field-Programmable Gate Array Logic Block Architecture for Speed", Proceedings of the IEEE Custom Integrated Circuits Conference, May 1990.

[Trimberger91] S. Trimberger, Personal communication, October 1991.

[Vuillamy91] J. Vuillamy, Z. G. Vranesic, and J. Rose, "Performance Evaluation and Enhancement of FPGAs", International Workshop on Field-Programmable Logic and Applications, Oxford, September 1991.

[Xilinx91] Xilinx, Inc., "The Programmable Gate Array Data Book", 1991.

Appendix A: Area comparison methodology

In this paper we presented a comparison of the area efficiency of four different FPGA architectures, based on several representative examples. Automatic tools were not used as this would make the comparison as much about the CAD algorithms as the architectures themselves. Even after eliminating CAD support, performing an architectural comparison between FPGAs is difficult because of complicating factors due to particular implementations. The most significant of these include the manufacturing process, the design rules employed, and the quantity and quality of designer effort. These factors make the straightforward approach of comparing actual chip area inadequate for purposes of architectural comparison. Simply using bits per cell also fails because it assumes that all FPGAs have equal per bit logic complexity, which is not true in current FPGAs (see discussion of table 2 below).

Our approach to architectural comparison is based on the premise that we can normalize cell area (the basic tiling structure in the FPGA) based on the size of the programming bits. There are two assumptions underlying this claim. First, a programming bit should occupy the same area if two different architectures are implemented on the same fabrication line, using the same design rules, and with the same amount of designer expertise and effort. It is not obvious that this is true, since the programming bits may be designed quite differently in different FPGAs. In the case of Triptych, the bits are 7-transistor pseudo-static shift-register cells interconnected in a scan-path, while Xilinx and CAL use 5-transistor static RAM cells (CFA's bit cell implementation is unspecified). However, even with these differences, we claim that the programming logic would take up similar area, due to the fact that the Triptych design can avoid ratioed transistors and use minimum-size devices (there is no feedback inverter to overpower through a pass-gate), and that it does not have the extra area for addressing and decode logic.

The second assumption is that the rest of the cell's logic and routing will scale proportionally to the programming bit cell size. This second assumption can be stated as the following relationship:

$$\frac{\text{total cell area in process } \alpha}{\text{total RAM bit area in process } \alpha} = \frac{\text{total cell area in process } \beta}{\text{total RAM bit area in process } \beta} \qquad (1)$$

There are at least two ways in which this assumption may not hold true. First, speed impacts area, and one could design a very small cell using minimum-size devices that would be unreasonably slow. Since speed of programming is not a major issue, programming bit cells are almost always made as small as possible, so decreasing the logic size could not decrease the programming bit size, and the ratio given above would change. This means that any area comparison must also include a speed comparison of the circuit mappings in the sample. We present such a comparison below. Second, programming bit cells are the easiest and most advantageous cell components to optimize, due to the lack of speed requirements and their high degree of replication, and thus receive a much higher share of optimization efforts in less optimized designs. As the programming bits shrink, the number of bit-equivalents the cell occupies increases. Therefore, less optimized implementations will compare less favorably to more highly optimized implementations (this is in fact the case for Triptych, thus handicapping it somewhat in the comparison).

Given these assumptions, we compare cell areas using the following normalization to a hypothetical standard process and design effort, Π:

$$\text{total cell area in process } \Pi \quad \times \quad \frac{1}{\text{single RAM bit area in process } \Pi} \qquad (2)$$

Note that this is simply the area of the cell in terms of the area of a single RAM bit. Since we assume the area of a RAM bit is the same for all architectures, then the normalized area of a cell is the same for all processes. Equation 2 can be rewritten to yield:

$$= \ \# \text{ of RAM bits per cell} \quad \times \quad \frac{\text{total cell area in process } \Pi}{\text{total RAM bit area in process } \Pi} \qquad (3)$$

And finally, by applying equation 1, we obtain equation 4, which consists of two values easily obtained from specific FPGA implementations (where α is whatever process, design rules, and designer effort were used to implement that particular FPGA). The value of the ratio at right is the inverse of the percentage of chip area dedicated to programming bits. This has the advantage of not requiring knowledge of the actual size of the FPGA's logic cell or programming bit, values that commercial vendors may not be willing to divulge.

$$= \ \# \text{ of RAM bits per cell} \quad \times \quad \frac{\text{total cell area in process } \alpha}{\text{total RAM bit area in process } \alpha} \qquad (4)$$

Equations 2 to 4 allow us to scale all FPGA cell areas to our hypothetical process Π. Table A1 shows the results of applying equation 4 to the four FPGAs, with Triptych as the baseline for comparison.

	CFA	CAL	Xilinx	Triptych
RAM bits per cell	18	18	201	26
Percentage of cell area for bits	40%	40%	22%	26%
Normalization factor	0.45	0.45	9.1	1.0

Table A1. Area normalization figures calculated by applying equation 4, and the data used to obtain them. Triptych is used as the baseline. The numbers for CFA, CAL, and Xilinx were obtained from [Furtek91, Kean91, Trimberger91] respectively.