To appear in *IEEE Transactions on VLSI Systems*.

High-Performance Carry Chains for FPGAs

Scott Hauck, Matthew M. Hosler, Thomas W. Fry
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
{hauck, mhosler, zaphod}@ece.nwu.edu

Abstract

Carry chains are an important consideration for most computations, including FPGAs. Current FPGAs dedicate a portion of their logic to support these demands via a simple ripple carry scheme. In this paper we demonstrate how more advanced carry constructs can be embedded into FPGAs, providing significantly higher performance carry computations. We redesign the standard ripple carry chain to reduce the number of logic levels in each cell. We also develop entirely new carry structures based on high performance adders such as Carry Select, Carry Lookahead, and Brent-Kung. Overall, these optimizations achieve a speedup in carry performance of 3.8 times over current architectures.

Introduction

Although originally intended as a way to efficiently handle random logic tasks in standard hardware systems, FPGAs have become the driving force behind a new computing paradigm. By mapping algorithms to these FPGAs significant performance benefits can be achieved. However, in order to achieve these gains, the FPGA resources must be able to efficiently support the computations required in the target application.

The key to achieving high performance hardware implementations is to optimize the circuit's critical path. For most datapath circuits, this critical path goes through the carry chain used in arithmetic and logic operations. In an arithmetic circuit such as an adder or subtractor, this chain represents the carries from bit position to bit position. For logical operations such as parity or comparison, the chain communicates the cumulative information needed to perform these computations. Optimizing such carry chains is a significant area of VLSI design and is a major focus of high-performance arithmetic circuit design.

Recently, several papers have focused on creating efficient implementations of high-performance adders in FPGAs [Hashemian94, Yu96, Xing98]. These approaches do not seek to modify the underlining architecture of the FPGA, but instead use the existing ripple carry structure of the FPGAs for their implementations. In another work, Woo modified the architecture of the FPGA to change linear cascading chains into tree structured cascade circuits [Woo95]. This modification significantly reduced the delay of cascade circuits. However, the equally important carry chains were not investigated.

In this paper, we will discuss methods of significantly improving the performance of carry computations in FPGAs by redesigning the FPGA architecture itself. In order to support datapath computations, most FPGAs already include special resources specifically optimized for implementing carry computations. However, because these resources use a relatively simple ripple carry scheme, carry computations can still be a major performance bottleneck. We will show that creating new carry schemes for the FPGA architecture can significantly improve its performance for all datapath operations with a relatively insignificant increase in chip area.

Basic Ripple Carry Cell

A basic ripple carry cell, similar to that found in the Altera 8000 series FPGAs [Altera95], is shown in Figure 1a. Mux 1, combined with the two 2-input lookup tables (2-LUTs) feeding into it, creates a 3-input lookup table (3-LUT). This element can produce any Boolean function of its three inputs. Two of its inputs (X and Y) form the primary inputs to the carry chain. The operands to the arithmetic or logic function being computed are sent in on these inputs, with each cell computing one bit position's result. The third input can be either another primary input (Z), or the carry from the neighboring cell, depending on the programming of mux 2's control bit. The potential to have Z replace the carry input is provided so that an initial carry input can be provided to the overall carry chain (useful for incrementers, combined adder/subtractors, and other functions). Alternatively the logic can

be used as a standard 3-LUT for functions that do not need a carry chain. An additional 3-LUT (not shown in the figure) is contained in each cell, which can be used to compute the sum for addition, or other functions.

Before we discuss modifications to this adder cell to improve performance, it is important to understand the role of the "Cout1" and "Cout0" signals in the carry chain. Cout1 and Cout0 are the outputs of the two 2-LUTs calculated such that Cout1 is function of A and B where Cin is true, and Cout0 is a function of A and B where Cin is false. In the case of a normal adder, Cout1 and Cout0 are given by equations 1 and 2:

$$\text{Cout1} = \text{cout}(A, B, 1) = A \textit{ AND } B \quad (1)$$

$$\text{Cout0} = \text{cout}(A, B, 0) = A \textit{ OR } B. \quad (2)$$

During carry computations the Cin input controls mux 1, which chooses which of these two signals will be the Cin for the next stage in the carry chain. If Cin is true, Cout = Cout1, while if Cin is false Cout = Cout0. Thus, Cout1 is the output whenever Cin = 1, while Cout0 is the output whenever Cin = 0. If we consider the possible combinations of values Cout1 and Cout0 can assume, there are four possibilities, three of which correspond to concepts from standard adders (Table 1). If both Cout0 and Cout1 are true, Cout is true no matter what Cin is, which is the same as the "generate" state in a standard adder. Likewise, when both Cout0 and Cout1 are false, Cout is false regardless of the state of Cin, and this combination of Cout1 and Cout0 signals is the "kill" state for this carry chain. If Cout0 and Cout1 are different, the Cout output will depend on the Cin input. When Cout0 = 0 and Cout1 =1, the Cout output will be identical to the Cin input, which is the normal "propagate" state for this carry chain. The last state, with Cout0 = 1 and Cout1 = 0, is not found in normal adders. In this state, the output still depends on the input, but in this case the Cout output is the inverse of the Cin input. We will call this state "inverse propagate".

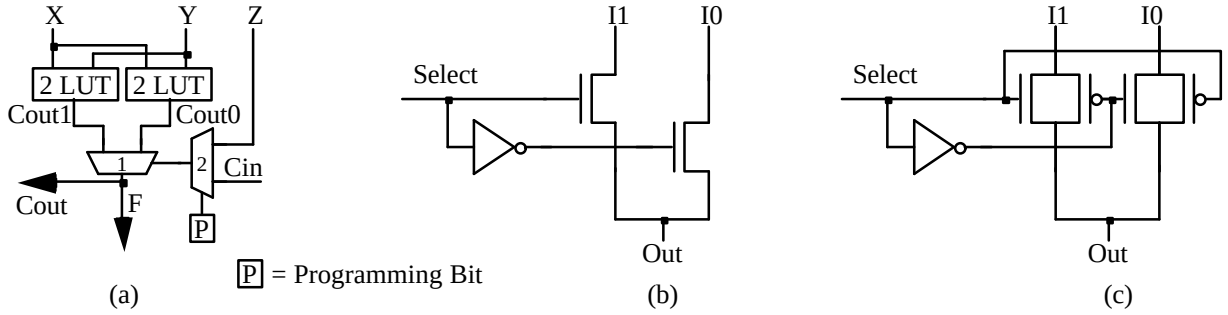

Figure 1: Carry computation element for FPGAs (a), a simple 2:1 mux implementation (b), and a slightly more complex version (c).

Cout0	Cout1	Cout	Name
0	0	0	Kill
0	1	Cin	Propagate
1	0	$\overline{Cin}$	Inverse Propagate
1	1	1	Generate

Table 1: Combinations of Cout0 and Cout1 values, and the resulting carry output. The final column lists the name for that combination.

For a normal adder, the inverse propagate state is never encountered. Thus, it might be tempting to disallow this state. However, for other computations this state is essential. For example, consider implementing a parity circuit with this carry chain, where each cell takes the XOR of the two inputs, X and Y, and the parity of the neighboring cell. If X and Y are both zero, the Cout of the cell will be identical to the parity of the neighboring cell, which is brought in on the Cin signal. Thus, the cell is in normal propagate mode. However, if X is true and Y is false,

then the Cout will be the opposite of Cin, since $(1 \oplus 0 \oplus Cin) = \overline{Cin}$. Thus, the inverse propagate state is important for implementing circuits like parity, and thus supporting this state in the carry chain we increase the types of circuits that can be efficiently implemented. In fact, by allowing an Inverse Propagate mode in the carry chain, the chain can be viewed as simply a series of 3-LUTs connected together, allowing any critical path to be implemented efficiently.

One last issue must be considered in this carry chain structure. In an FPGA, the cells represent resources that can be used to compute arbitrary functions. However, the location of functions within this structure is completely up to the user. Thus, a user may decide to start or end a carry computation at any place in the array. In order to start a carry chain we must program the first cell in the carry chain to ignore the Cin signal. One easy way to do this is to program mux 2 in the cell to route input Z to mux 1 instead of Cin. For situations where one wishes to have a carry input to the first stage of an adder (which is useful for implementing combined adder/subtractors as well as other circuits) this is the right solution. However, in other cases this may not be possible. The first stage in many carry computations is only a 2-input function, and forcing the carry chain to wait for the arrival of an additional, unnecessary input will only needlessly slow down the circuit's computation. This is not necessary. In these circuits, the first stage is only a 2-input function. Thus, either 2-LUT in the cell could compute this value. If we program both 2-LUTs with the same function, the output will be forced to the proper value regardless of the input, and thus either the Cin or the Z signal can be routed to mux 1 without changing the computation. However, this is only true if mux 1 is implemented such that if the two inputs to the mux are the same, the output of the mux is identical to the inputs regardless of the state of the select line. Figure 1b shows an implementation of a mux that does not obey this requirement. Since the carry chain is part of an FPGA, the input to this mux could be connected to some unused logic in another row which is generating unknown values. If that unused logic had multiple transitions which caused the signal to change quicker than the gate could react, then it is possible that the select signal to this mux could be stuck midway between true and false (2.5V for 5V CMOS). In this case, it will not be able to pass a true value from the input to the output, and thus will not function properly for this application. However, a mux built like that in Figure 1c, with both n-transistor and p-transistor pass gates will operate properly for this case. Thus, we will assume throughout this paper that all muxes in the carry chain are built with the circuit shown in Figure 1c, though any other mux implementation with the same property could be used (including tristate driver based muxes which can restore signal drive and cut series R-C chains).

Delay Model

To initially quantify the performance of the carry chains developed in this paper, a unit gate delay model will be used: all simple gates of two or three inputs that are directly implementable in one logic level in CMOS are considered to have a delay of one. All other gates must be implemented in such gates, and have the delay of the underlying circuit. Thus, inverters and 2 to 3 input NAND and NOR gates have a delay of one. A 2:1 mux has a delay of one from the I0 or I1 inputs to the output, but has a delay of two from the select input to the output due to the inverter delay (see Figure 1c). The delay of the 2-LUTs, and any routing leading to them, is ignored since this will be a constant delay for all the carry chains developed in this paper. This delay model will be used to initially discuss different carry chain alternatives and their advantages and disadvantages. Precise circuit timings are also generated using Spice on the VLSI layouts of the carry chains, as discussed later in this paper.

Optimized Ripple Carry Cell

As we discussed in an earlier section, the ripple carry design of Figure 1a is capable of implementing most interesting carry computations. However, it turns out that this structure is significantly slower than it needs to be since there are two muxes on the carry chain in each cell (mux 1 and mux 2). Specifically, the delay of this circuit is 1 for the first cell plus 3 for each additional cell in the carry chain (1 delay for mux 2 and 2 delays for mux 1), yielding an overall delay of 3n-2 for an n-cell carry chain. Note that we assume the longest path through the carry chain comes from the 2-LUTs and not input Z since the delay through the 2-LUTs will be larger than the delay through mux 2 in the first cell.

We can reduce the delay of the ripple carry chain by removing mux 2 from the carry path. As shown in Figure 2a, instead of choosing between Cin and Z for the select line to the output mux, we instead have two separate muxes, 1 and 2, controlled by Cin and Z respectively. Then, the circuit chooses between these outputs with mux 3. In this

design there is a delay of 1 in the first cell of a carry chain, a delay of 3 in the last cell (2 for mux 1 and 1 for mux 3), and a delay of only 2 for all intermediate cells. Thus, the delay of this design is only 2n for an n-bit ripple carry chain, yielding up to a 50% faster circuit than the original design.

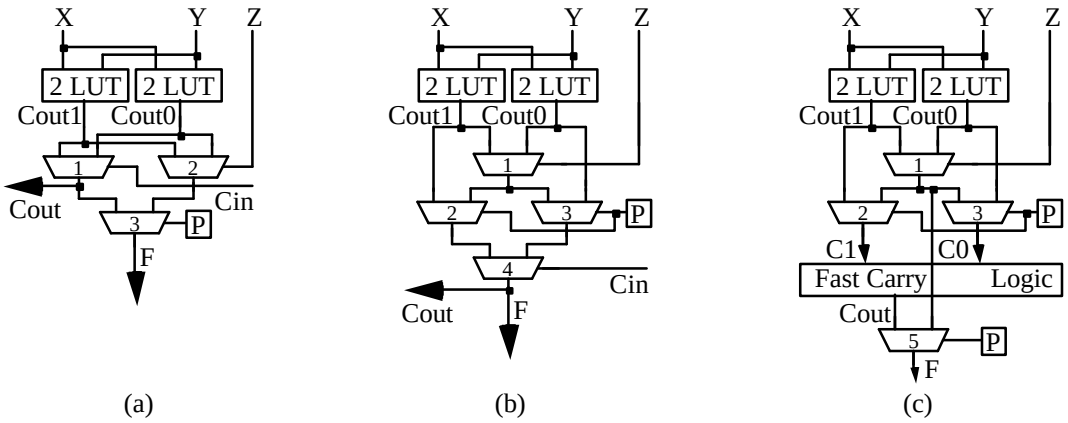

Figure 2: Carry computation elements with faster carry propagation.

Unfortunately, the circuit in Figure 2a is not logically equivalent to the original design. The problem is that the design can no longer use the Z input in the first cell of a carry chain as an initial carry input, since Z is only attached to mux 2, and mux 2 does not lead to the carry path. The solution to this problem is the circuit shown in Figure 2b. For cells in the middle of a carry chain mux 2 is configured to pass Cout1, and mux 3 passes Cout0. Thus, mux 4 receives Cout1 and Cout0, and provides a standard ripple carry path. However, when we start a carry chain with a carry input (provided by input Z), we configure mux 2 and mux 3 to both pass the value from mux 1. Since this means that the two main inputs to mux 4 are identical, the output of mux 4 (Cout) will automatically be the same as the output of mux 1, ignoring Cin. Mux 1's main inputs are driven by two 2-LUTs controlled by X and Y, and thus mux 1 forms a 3-LUT with the other 2-LUTs. When mux 2 and mux 3 pass the value from mux 1 the circuit is configured as a 3-LUT starting a carry chain, while when mux 2 and mux 3 choose their other input (Cout1 and Cout2 respectively) the circuit is configured to continue the carry chain. This design is therefore functionally equivalent to the design in Figure 1a. However, carry chains built from this design have a delay of 3 in the first cell (1 in mux 1, 1 in mux 2 or mux 3, and 1 in mux 4) and 2 in all other cells in the carry chain, yielding an overall delay of 2n+1 for an n-bit carry chain. Thus, although this design is 1 gate delay slower than that of Figure 2a, it provides the ability to have a carry input to the first cell in a carry chain, something that is important in many computations. Also, for carry computations that do not need this feature, the first cell in a carry chain built from Figure 2b can be configured to bypass mux 1, reducing the overall delay to 2n, which is identical to that of Figure 2a. On the other hand, in order to implement a n-bit carry chain with a carry input, the design of Figure 2a requires an additional cell at the beginning of the chain to bring in this input, resulting in a delay of $2(n+1) = 2n+2$, which is slower than that of the design in Figure 2b. Thus, the design of Figure 2b is the preferred ripple carry design among those presented so far.

High-Performance Carry Logic for FPGAs

In the previous section we discussed how to optimize a ripple carry chain structure for use in FPGAs. While this provides some performance gain over the basic ripple carry scheme found in many current FPGAs, it is still much slower than what is done in custom logic. There have been tremendous amounts of work done on developing alternative carry chain schemes that overcome the linear delay growth of ripple-carry adders. Although these techniques have not yet been applied to FPGAs, in this paper we will demonstrate how these advanced adder techniques can be integrated into reconfigurable logic.

The basis for all of the high-performance carry chains developed in this paper will be the carry cell of Figure 2c. This cell is very similar to that of Figure 2b, except that the actual carry chain (mux 4) has been abstracted into a

generic “Fast Carry Logic” unit and mux 5 has been added. This extra mux is present because although some of our faster carry chains will have much quicker carry propagation for long carry chains, they do add significant delay to non-carry computations. Thus, when the cell is used as just a normal 3-LUT, using inputs X, Y, and Z, mux 5 allows us to bypass the carry chain by selecting the output of mux 1.

The important thing to realize about the logic of Figure 2c is that any logic that can compute the value

$$Cout_i = \left(Cout_{i\_1} * C1_i\right) + (\overline{Cout}_{i-1} * C0_i), \quad (3)$$

where *i* is the position of the cell within the carry chain, can provide the functionality necessary to support the needs of FPGA computations. Thus, the fast carry logic unit can contain any logic structure implementing this computation. In this paper we will look at four different types of carry logic: Carry Select, Carry Lookahead (including Brent-Kung), Variable Bit, and Ripple Carry (discussed previously). Note that because of the needs and requirements of carry chains for FPGAs, we will have to develop new circuits, inspired by the standard adder structures, but which are more appropriate for FPGAs. The main difference is that we no longer have just the Generate, Propagate, and Kill states for an adder, we must also support Inverse Propagate. These four states are encoded on signals C1 and C0 as shown in Table 1. Also, while standard adders are concerned only with the maximum delay through an entire N-bit adder structure, the delay concerns for FPGAs are more complicated. Specifically, when an N-bit carry chain is built into the architecture of an FPGA it does not represent an actual computation, but only the potential for a computation. A carry chain resource may span the entire height of a column in the FPGA, but a mapping to the logic may use only a small portion of this chain, with the carry logic in the mapping starting and ending at arbitrary points in the column. Thus, we are concerned with not just the carry delay from the first to the last position in a carry chain, but must consider the delay for a carry computation beginning and ending at any point within this column. For example, even though the FPGA architecture may provide support for carry chains of up to 32 bits, it must also efficiently support 8 bit carry computations placed at any point within this carry chain resource.

Carry Select

The problem with a ripple carry structure is that the computation of the Cout for bit position *i* cannot begin until after the computation has been completed in bit positions 0..i-1. A Carry Select structure overcomes this limitation. The main observation is that for any bit position, the only information it receives from the previous bit positions is its Cin signal, which can be either true or false. In a Carry Select adder the carry chain is broken at a specific column, and two separate additions occur: One assuming the Cin signal is true, the other assuming it is false. These computations can take place before the previous columns complete their operation, since they do not depend on the actual value of the Cin signal. This Cin signal is instead used to determine which adder’s outputs should be used. If the Cin signal is true, the output of the following stages comes from the adder that assumed that the Cin would be true. Likewise, a false Cin chooses the other adder’s output. This splitting of the carry chain can be done multiple times, breaking the computation into several pairs of short adders with output muxes choosing which adder’s output to select. The length of the adders and the breakpoints are carefully chosen such that the small adders finish computation just as their Cin signals become available. Short adders handle the low-order bits, and the adder length is increased further along the carry chain, since later computations have more time until their Cin signal is available.

A Carry Select carry chain structure for use in FPGAs is shown in Figure 3. The carry computation for the first two cells is performed with the simple ripple-carry structure implemented by mux 1. For cells 2 and 3 we use two ripple carry adders, with one adder (implemented by mux 2) assuming the Cin is true, and the other (mux 3) assuming the Cin is false. Then, muxes 4 and 5 pick between these two adders’ outputs based on the actual Cin coming from mux 1. Similarly, cells 4-6 have two ripple carry adders (mux 6 & 7 for a Cin of 1, mux 8 & 9 for a Cin of 0), with output muxes (muxes 10-12) deciding between the two based upon the actual Cin (from mux 5). Subsequent stages will continue to grow in length by one, with cells 7-10 in one block, cells 11-15 in another, and so on. Timing values showing the delay of the Carry Select carry chain relative to other carry chains will be presented later in this paper.

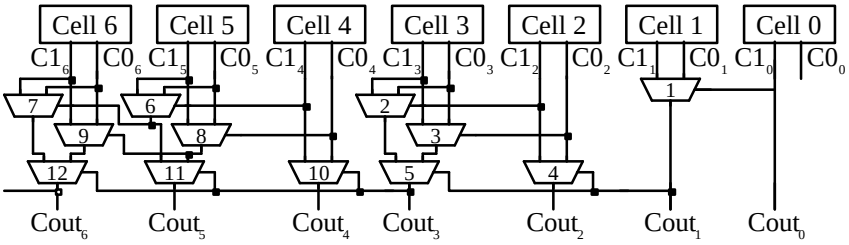

Figure 3: Carry Select structure.

Variable Block

Like the Carry Select carry chain, a Variable Block structure [Oklobdzija88] consists of blocks of ripple carry elements (Figure 4). However, instead of precomputing the Cout value for each possible Cin value, it instead provides a way for the carry signal to skip over intermediate cells where appropriate. Contiguous blocks of the computation are grouped together to form a unit with a standard ripple carry chain. As part of this block, logic is included to determine if all of the cells are in their propagate state. If so, the Cout for this block is immediately set to the value of the block's Cin, allowing the carry chain to bypass this block's normal carry chain on its way to later blocks. The Cin still ripples through the block itself, since the intermediate carry values must also be computed. If any of the cells in the carry chain are not in propagate mode, the Cout output is generated normally by the ripple carry chain. While this carry chain does start at the block's Cin signal, and leads to the block's Cout, this long path is a false path. That is, since there is some cell in the block that is not in propagate mode, it must be in generate or kill mode, and thus the block's Cout output does not depend on the block's Cin input.

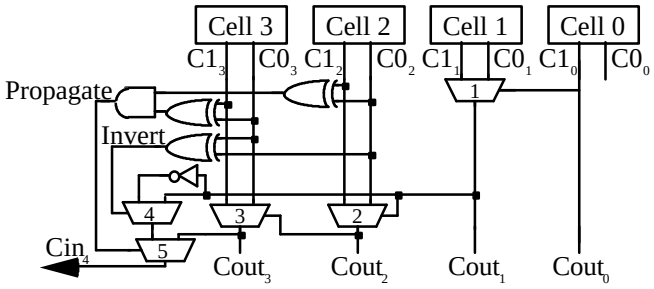

Figure 4: The Variable Block carry structure. Mux 1 performs an initial two stage ripple carry. Muxes 2 through 5 form a 2-bit Variable Block block. Mux 5 decides whether the Cin signal should be sent directly to Cout, while mux 4 decides whether to invert the Cin signal or not.

A major difficulty in developing a version of the Variable Block carry chain for inclusion in an FPGA's architecture is the need to support both the propagate and inverse propagate state of the cells. To do this, we compute two values. First, we check to see if all the cells are in some form of propagate mode (either normal propagate or inverse propagate) by ANDing together the XOR of each stage's C1 and C0 signals. If so, we know that the Cout function will be equal to either *Cin* or $\overline{Cin}$. To decide whether to invert the signal or not, we must determine how many cells are in inverse propagate mode. If the number is even (including zero) the output is not inverted, while if the number is odd the output is inverted. The inversion check can be done by looking for inverse propagate mode in each cell and XORing the results. To check for inverse propagate, we only look at the C0 signal from each cell. If this signal is true, the cell is in either generate or inverse propagate mode, and if it is in generate mode the inversion signal will be ignored anyway (we only consider inverting the Cin signal if all cells are in some form of propagate mode). Note that for both of these tests we can use a tree of gates to compute the result. Also, since we ignore the inversion signal when we are not bypassing the carry chain we can use C1 as the inverse of C0 for the inversion signal's computation, which avoids the added inverter in the XOR gate.

The organization of the blocks in the Variable Block carry structure bears some similarity to the Carry Select structure. The early stages of the structure grow in length, with short blocks for the low order bits, building in length further in the chain in order to equalize the arrival time of the carry from the block with that of the previous

block. However, unlike the Carry Select structure, the Variable Block adder must also worry about the delay from the Cin input through the block's ripple chain. Thus, after the carry chain passes the midpoint of the logic, the blocks begin decreasing in length. This balances the path delays in the system and improves performance. The division of the overall structure into blocks depends on the details of the logic structure and the length of the entire computation. We use a block length (from low order to high order cells) of 2, 2, 4, 5, 7, 5, 4, 2, 1 for a normal 32 bit structure. The first and last block in each adder is a simple ripple carry chain, while all other blocks use the Variable Block structure. Delay values of the Variable Block carry chain relative to other carry chains will be presented later in this paper.

Carry Lookahead and Brent-Kung

There are two inputs to the fast carry logic in Figure 2c: $C1_i$ and $C0_i$. The value of $C1_i$ is programmed by the LUTs so that it contains the value that $Cout_i$ should have if Cin_i is true. Similarly, the value of $C0_i$ is programmed by the LUTs so that it contains the value that $Cout_i$ should have if Cin_i is false. We can combine the information from two stages together to determine what the Cout of one stage will be given the Cin of the previous stage. For example,

$$C1_{i,i-1} = \left(C1_{i-1} * C1_i\right) + \left(\overline{C1_{i-1}} * C0_i\right) \tag{4}$$

$$C0_{i,i-1} = \left(C0_{i-1} * C1_i\right) + \left(\overline{C0_{i-1}} * C0_i\right), \tag{5}$$

where $C1_{x,y}$ is the value of $Cout_x$ assuming that $Cin_y = 1$. This allows us to halve the length of the carry chain, since once these new values are computed a single mux can compute $Cout_i$ given Cin_{i-1}. In fact, similar rules can be used recursively, halving the length of the carry chain with each application. Specifically,

$$C1_{i,k} = \left(C1_{j-1,k} * C1_{i,j}\right) + \left(\overline{C1_{j-1,k}} * C0_{i,j}\right) \tag{6}$$

$$C0_{i,k} = \left(C0_{j-1,k} * C1_{i,j}\right) + \left(\overline{C0_{j-1,k}} * C0_{i,j}\right), \tag{7}$$

assuming $i > j > k$. The digital logic computing both of these functions will be called a concatenation box. The Brent-Kung carry chain [Brent82] consists of a hierarchy of these concatenation boxes, where each level in the hierarchy halves the length of the carry chain, until we have computed $C1_{i,0}$ and $C0_{i,0}$ for each cell i. A single level of muxes at the bottom of the Brent-Kung carry chain can then use these values to compute the Cout for each cell given a Cin. The Brent-Kung carry chain is shown in Figure 5.

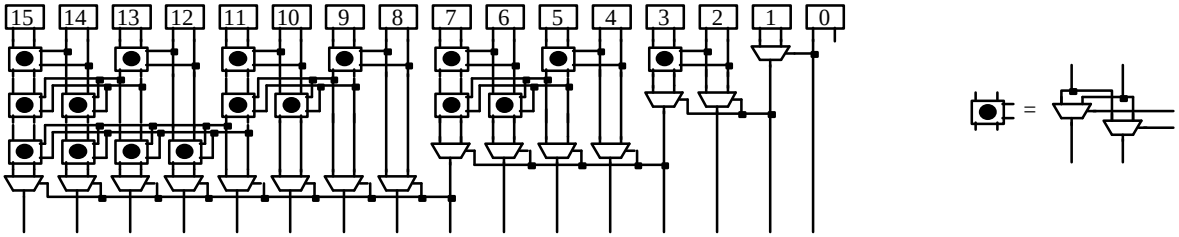

Figure 5: The 3-Level, 16 bit Brent-Kung structure. At right are the details of the concatenation block. Note that once the Cin has been computed for a given stage, a mux is used in place of a concatenation block.

The Brent-Kung adder is a specific case of the more general Carry Lookahead adder [Sklansky60]. In a Carry Lookahead adder a single level of concatenation combines together the carry information from multiple sources. A typical Carry Lookahead adder will combine 4 cells together in one level (computing $C1_{i,i-3}$ and $C0_{i,i-3}$), combine four of these new values together in the next level, and so on.

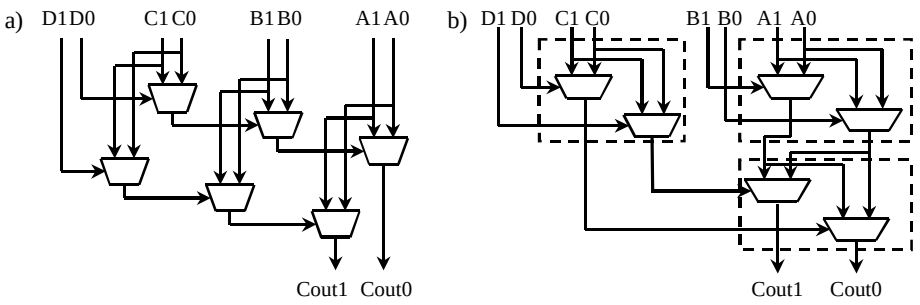

Figure 6: Concatenation boxes. (a) A 4-cell concatenation box, and (b) its equivalent made up of only 2-cell concatenation boxes.

However, while a combining factor of 4 is considered optimal for a standard adder, in FPGAs combining more than two values in a level is not advantageous. The problem is that although the logic to concatenate N values together grows linearly for a normal adder, it grows exponentially for a reconfigurable carry chain. For example, to concatenate three values together for a normal adder we have the equation:

$$C_{x,z} = G_x + \left(P_x * C_{x-1,z}\right), \tag{8}$$

where P_x and G_x are the propagate and generate values of the current cell. However, to concatenate three values together for a reconfigurable carry chain together we have the equation:

$$C1_{w,z} = \left(C1_{x-1,z} * C1_{w,x}\right) + \left(\overline{C1_{x-1,z}} * C0_{w,x}\right) \tag{9}$$

$$= \left(\left(C1_{y-1,z} * C1_{x-1,y}\right) + \left(\overline{C1_{y-1,z}} * C0_{x-1,y}\right)\right) * C1_{w,x}$$

$$+ \left(\left(C1_{y-1,z} * \overline{C1_{x-1,y}}\right) + \left(\overline{C1_{y-1,z}} * \overline{C0_{x-1,y}}\right)\right) * C0_{w,x}. \tag{10}$$

An alternative way to see why combining 4 cells together in one level is bad for FPGAs is to consider how this combining would be implemented. Figure 6a shows a concatenation box that takes its input from 4 different cells. Figure 6b then shows how a 4-cell concatenation box can be built using only three 2-cell concatenation boxes. This second method of creating a 4-cell concatenation box is really the equivalent of a 2-Level Carry Lookahead adder using 2-cell concatenation boxes. Using the simple delay model discussed earlier, the delay for the 4-cell concatenation box in Figure 6a is 6 units since the signal must travel through 3 muxes. The delay for the 4-cell concatenation box equivalent found in Figure 6b, however, is only 4 units since the signal must travel through only 2 muxes. Thus, a 4-cell concatenation box is never used since it can always be implemented with a smaller delay using 2-cell concatenation boxes. Therefore, the Brent-Kung structure is the best approach.

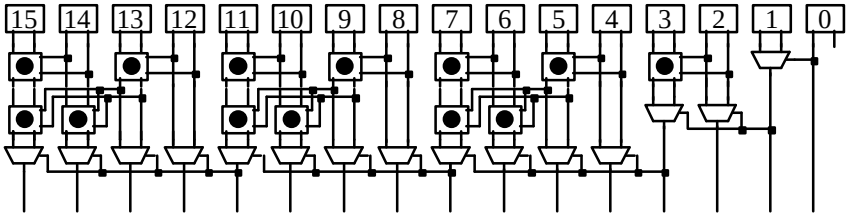

Figure 7: A 2-Level, 16 bit Carry Lookahead structure.

Another option in Carry Lookahead adders is the possibility of using less levels of concatenation than a Brent-Kung structure. Specifically, a Brent-Kung structure for a 32-bit adder would require 4 levels of concatenation. While this allows Cin_0 to quickly reach $Cout_{31}$, there is a significant amount of delay in the logic that computes the individual $C1_{i,0}$ and $C0_{i,0}$ values. We can instead use fewer levels than the complete hierarchy of the Brent-Kung

adder and simply ripple together the top-level carry computations of smaller carry-lookahead adders. Specifically, if we talk about an N-level Carry Lookahead adder, that means that we only apply N levels of 2-input concatenation units. A 2-Level, 16 bit Carry Lookahead carry chain is shown in Figure 7.

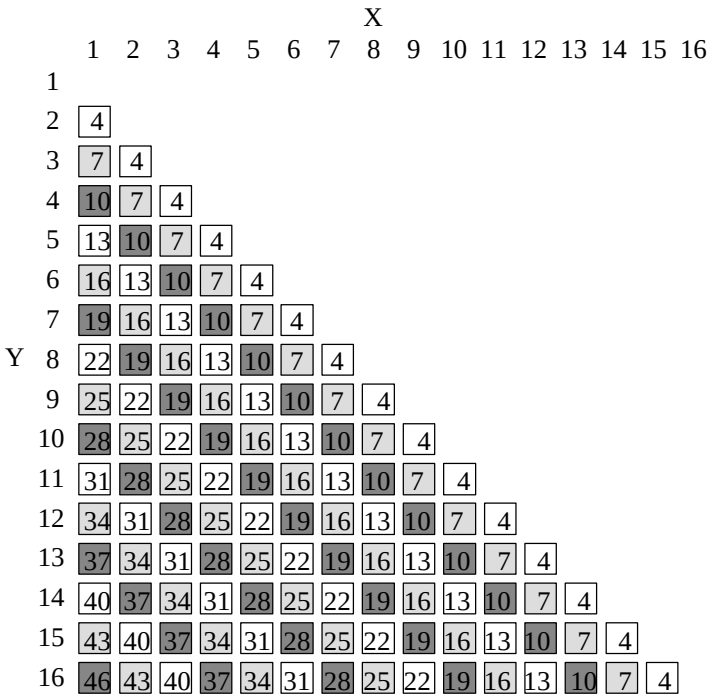

Figure 8: The delays of a Basic Ripple carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Carry Chain Performance

In order to compare the carry chains developed in this paper, we computed the performance of the carry chains of different lengths. The delay is computed from the output of the 2-LUTs in one cell to the final output (F) in another using the simple delay model discussed earlier in the Delay Model section. This simple model calculates a delay based on the number of gates that must be traversed by a signal. Precise circuit timings are discussed later in this paper. Figures 8 and 9 show the delays of a carry chain starting at Cell X and ending at Cell Y for the Basic Ripple and Brent-Kung carry chains, respectively. These figures show how the delay patterns are different for each carry chain. One important issue to consider is what delay we should use to compare carry chain performance. While the carry chain structure is dependent on the length of the carry computation supported by the FPGA (such as the Variable Block segmentation), the user may decide to use any contiguous subsequence of the carry chain's length for their mapping. To deal with this, we assume that the FPGAs are built to support up to a 32-bit carry chain, and record the maximum carry chain delay for any length L carry computation within this structure. That is, since we do not know where the user will begin their carry computation within the FPGA architecture, we measure the worst case delay for a length L carry computation starting at any point in the FPGA. Note that this delay is the critical path within the L-bit computation, which means carries starting and ending anywhere within this computation are considered.

Figure 10 shows the maximum carry delays for each of the carry structures discussed in this paper, as well as the basic ripple carry chain found in current FPGAs. These delays are based on the simple delay model that was discussed earlier. More precise delay timings from VLSI layouts of the carry chains will be discussed later. As can be seen, the best carry chain structure for short distances is different from the best chain for longer computations, with the basic ripple carry structure providing the best delay for length 2 carry computations, while the Brent-Kung structure provides the best delay for computations of four bits or more. In fact, the ripple carry structure is more than twice as fast as the Brent-Kung structure for 2-bit carry computations, yet is approximately

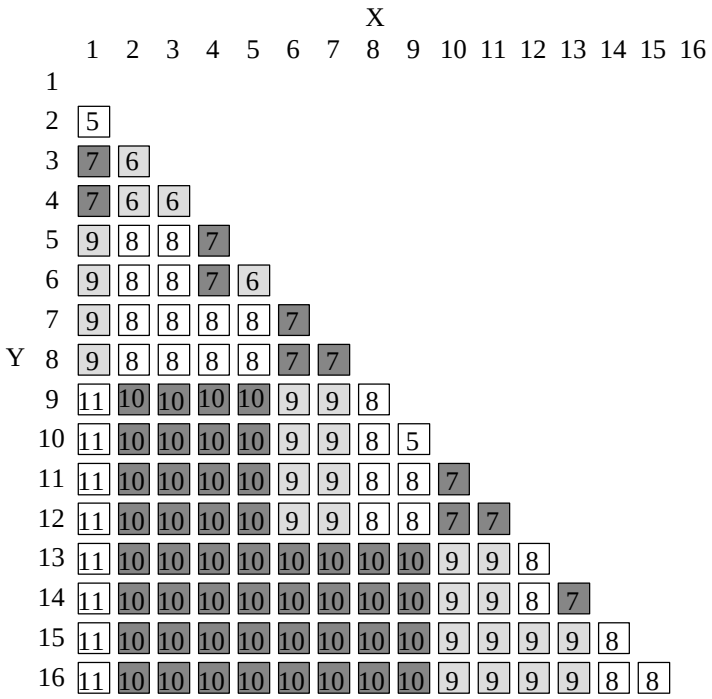

Figure 9: The delays of a Brent-Kung carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

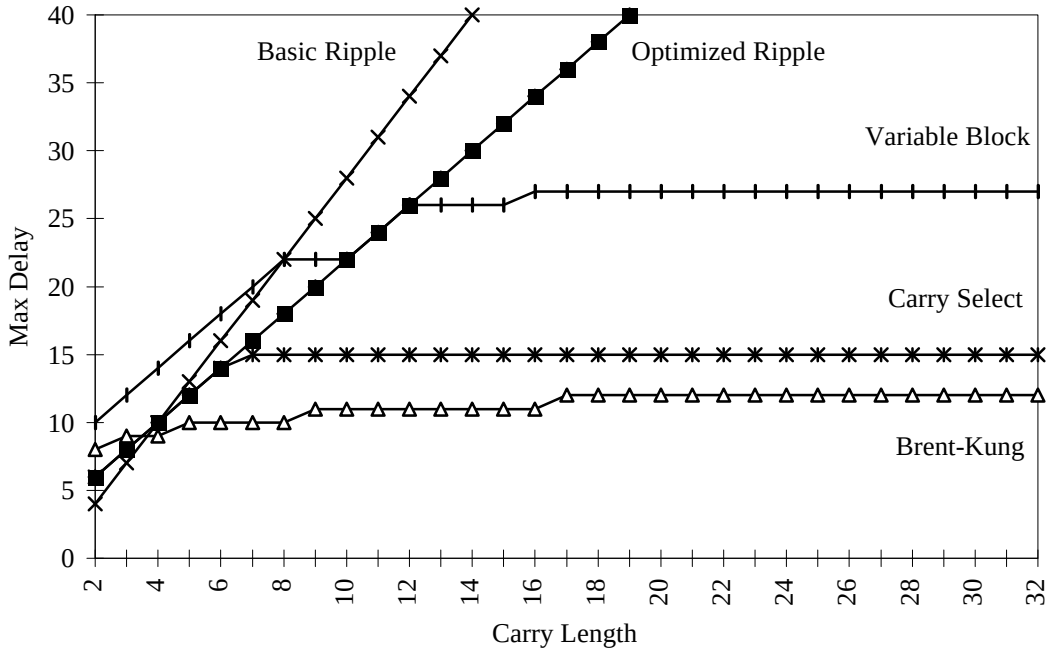

Figure 10: A comparison of the various carry chain structures. The delays represent the maximum delay for a N-bit adder placed anywhere within a 32-bit carry structure.

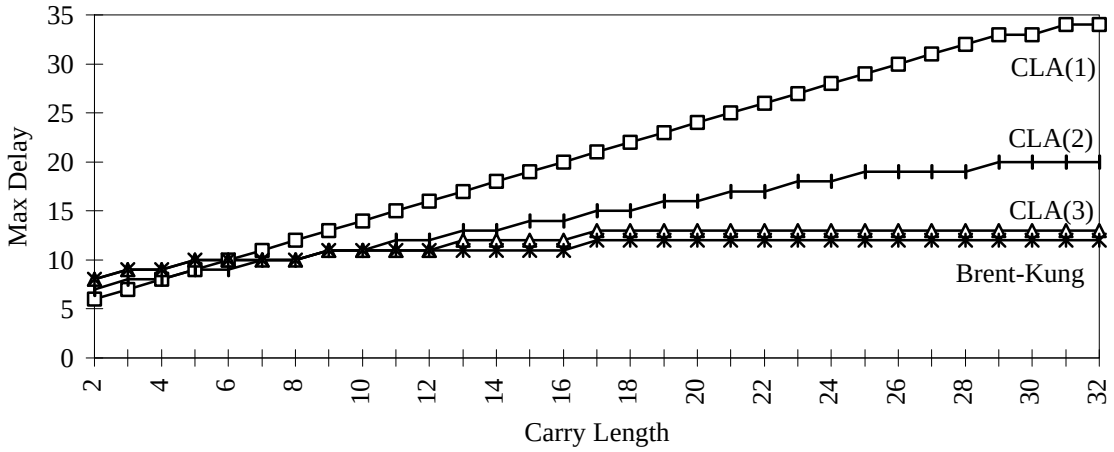

Figure 11: A comparison of Carry Lookahead structures. CLA(i) represents a Brent-Kung style Carry Lookahead structure with only *i* levels of concatenation boxes.

eight times slower for 32 bit computations. However, short carries are often not that critical, since they can be supported by the FPGA's normal routing structure and will tend not to dominate the performance of the overall system. Therefore, we believe that the Brent-Kung structure is the preferred structure for FPGA carry computations, and that it is capable of providing significant performance improvement over current FPGA carry chains.

In this paper we also considered other Carry Lookahead adder designs which do not use as many levels of concatenation boxes as a full Brent-Kung adder. However, as can be see in Figure 11, the other carry structures provide only modest improvements over the Brent-Kung structure for short distances, and perform significantly worse than the Brent-Kung structure for longer carry chains.

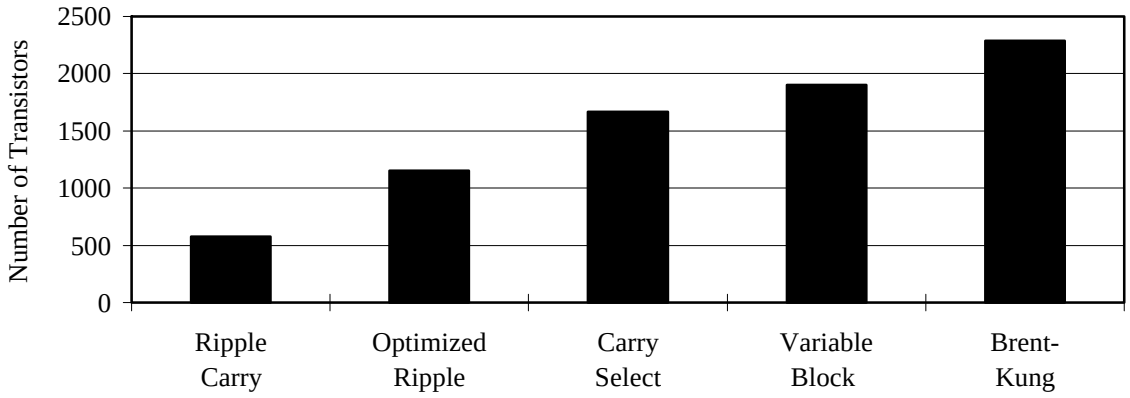

Figure 12: The transistor counts of the Basic Ripple, Optimized Ripple, Carry Select, Variable Block, and Brent-Kung carry chains.

Another consideration when choosing a carry chain structure is the size of the circuit. Figure 12 shows the number of transistors that are used in the design of the Basic Ripple, Optimized Ripple, Carry Select, Variable Block, and Brent-Kung carry chains. The transistor counts here are based on a CMOS implementation of the inverting tri-state mux. One concern with the Brent-Kung structure is that it requires four times more transistors to implement than the basic ripple carry. However, in typical FPGAs the carry structure occupies only a tiny fraction of the chip area, since the programming bits, LUTs, and programmable routing structures dominate the chip area. Therefore, the increase in chip area required by the higher performance carry chains developed in this paper is relatively insignificant, yet the performance improvements can greatly accelerate many types of applications. The area and

performance of the high performance carry chains with respect to those of the basic ripple carry chains will be discussed further in the next section of this paper.

Layout Results

The results of the simple delay model described earlier suggest that the Brent-Kung carry chain has the best performance of any of the carry chains. However, the performance results used to make this decision are based only on the simple delay model, which may not accurately reflect the true delays. The simple delay model does not take into account transistor sizes or routing delays. Therefore, in order to get more accurate comparisons the carry chains were sized using logical effort [Sutherland90], layouts were created, and timing numbers were obtained from Spice for a 0.6 micron process. Only the most promising carry chains were chosen for implementation. These include the basic ripple carry, which can be found in current FPGAs, as well as the new Optimized Ripple and Brent-Kung carry chains.

Carry Chain	32-bit delay (ns)	3-LUT delay (ns)
Basic Ripple Carry	23.4	1.6
Optimized Ripple	18.7	2.5
Brent-Kung	6.1	2.1

Table 2: A comparison of the delays of different structures for (a) a 32-bit carry, and (b) a non-carry computation of a function, f(X,Y,Z).

Table 2 shows the delays of a 32-bit carry for the carry chains that were implemented. Notice that the delay for basic ripple carry chain is 23.4ns, and the delay for the optimized ripple carry chain is 18.7ns resulting in a speedup of 1.25 times over the basic ripple carry chain. Furthermore, the delay for the Brent-Kung carry chain is only 6.1ns. Thus, the best carry chain developed here has a delay 3.8 times faster than the basic ripple carry chain used in industry. Table 2 also shows the delays of the FPGA cell assuming that the cell is programmed to compute a function of 3 variables and avoid the carry chain (as shown by Mux 5 in Figure 2c). The delay for the basic ripple carry chain in this case is 1.6ns, while the delay for the Brent-Kung carry chain is 2.1ns. Thus, the Brent-Kung implementation does slow down non-carry operations, but only by a small amount.

Carry Chain	Area (λ^2)	% Increase for Chimaera FPGA	% Increase for General-Purpose FPGA
Basic Ripple Carry	171368	0	0
Optimized Ripple	394953	1.3	0.18
Brent-Kung	1622070	8.5	1.18

Table 3: Areas of different carry chain implementations.

Table 3 shows the area of these carry chains as measured from the layouts. One item to note is the size of the Brent-Kung carry chain. Its size is shown as 9.47 times larger than the basic ripple carry chain. This number should be viewed purely as an upper bound, since the layout of the Basic Ripple Carry was optimized much more than the Brent-Kung layout. We believe that further optimization of the Brent-Kung design could reduce its area by 600,000 square lambda, yielding only a factor of 5 size increase over the Basic Ripple Carry scheme.

A more accurate comparison of the size implications of the improved carry chains is to consider the area impact of including these carry chains in an actual FPGA. We have conducted such experiments with the Chimaera FPGA [Hauck97], a special-purpose FPGA which has been carefully optimized to reduce the amount of chip area devoted to routing. As shown in Table 3, replacing the basic ripple carry structure in the Chimaera FPGA with the Brent-Kung structure results in an area increase of 8.5%. Our estimates of the area increase on a general-purpose FPGA such as the Xilinx 4000 [Xilinx96] or Altera 8000 FPGAs, where the more complex routing structure consumes a much greater portion of the chip area, is that the Brent-Kung structure would only increase the total chip area by

1.2%. This is based upon increasing the portion of Chimaera's chip area devoted to routing up to the 90% of chip area typical in general-purpose FPGAs.

Using the Carry Chain

Thus far, we have explained why high performance carry chains should be used in FPGAs. Now we will explain where the carry chain is located in the FPGA and how it is programmed. In our design, the carry chain is row-based and unidirectional as shown in Figure 13. There is exactly one carry chain per row, and it spans the entire length of that row. The carry chains in different rows are not interconnected. However, normal FPGA routing could connect these carry chains if a larger carry chain is needed.

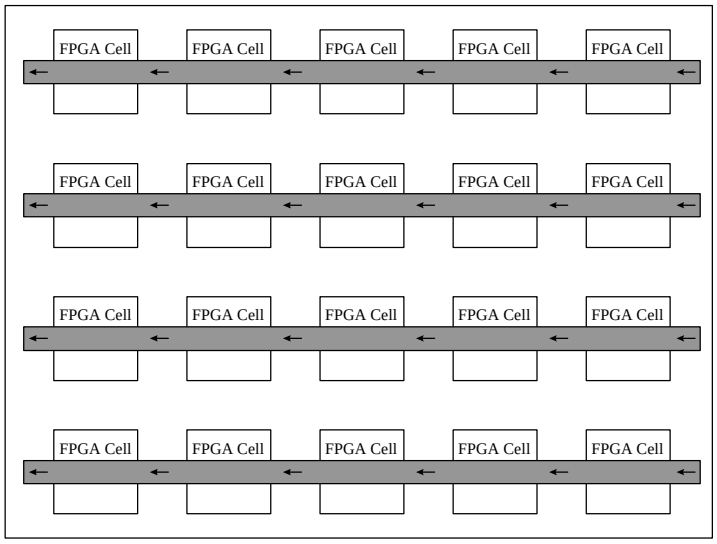

Figure 13: The location of the carry chains (shaded region) within the FPGA.

The Brent-Kung carry chain that we designed is an n-bit carry chain where n is a power of 2. The carry chain is placed in one row of the FPGA, and it interfaces with the FPGA cells in that row. Each FPGA cell connects to a different part of the carry chain. Since the Brent-Kung carry chain is not uniform, the carry chain logic seen by each FPGA cell will be different. Figure 14 shows two cells of the Chimaera FPGA and the portion of the Brent-Kung carry chain contained within them.

One additional feature of the carry chain is that it can be broken into smaller, independent carry chains at any point by programming the LUTs. Recall that one LUT produces Cout1, which is the value of Cout if Cin is equal to 1. Similarly, the other LUT produces Cout0, which is the value of Cout if Cin is equal to 0. In order to break the carry chain we program the LUTs so that both Cout1 and Cout0 have the same value. In this case, Cout has the same value regardless of the value of Cin, and the original carry chain has been segmented into smaller, independent carry chains.

Conclusions

One of the critical performance bottlenecks in most systems is the carry chains contained in many arithmetic and logical operations. Current FPGAs optimize for these elements by providing some support specifically for carry computations. However, these systems rely on relatively simple ripple carry structures which provide much slower performance than current high-performance carry chain designs. With the advent of reconfigurable computing, and the demands of implementing complex algorithms in FPGAs, the slowdown of carry computations in FPGAs is an even more crucial concern.

In order to speed up the carry structure found in current FPGAs we developed several innovative techniques. A novel cell design is used to reduce the delay through the cell to a single mux by moving the decision of whether to

use the carry chain off of the critical path. This results in approximately a factor of 1.25 speedup over current FPGA carry delays.

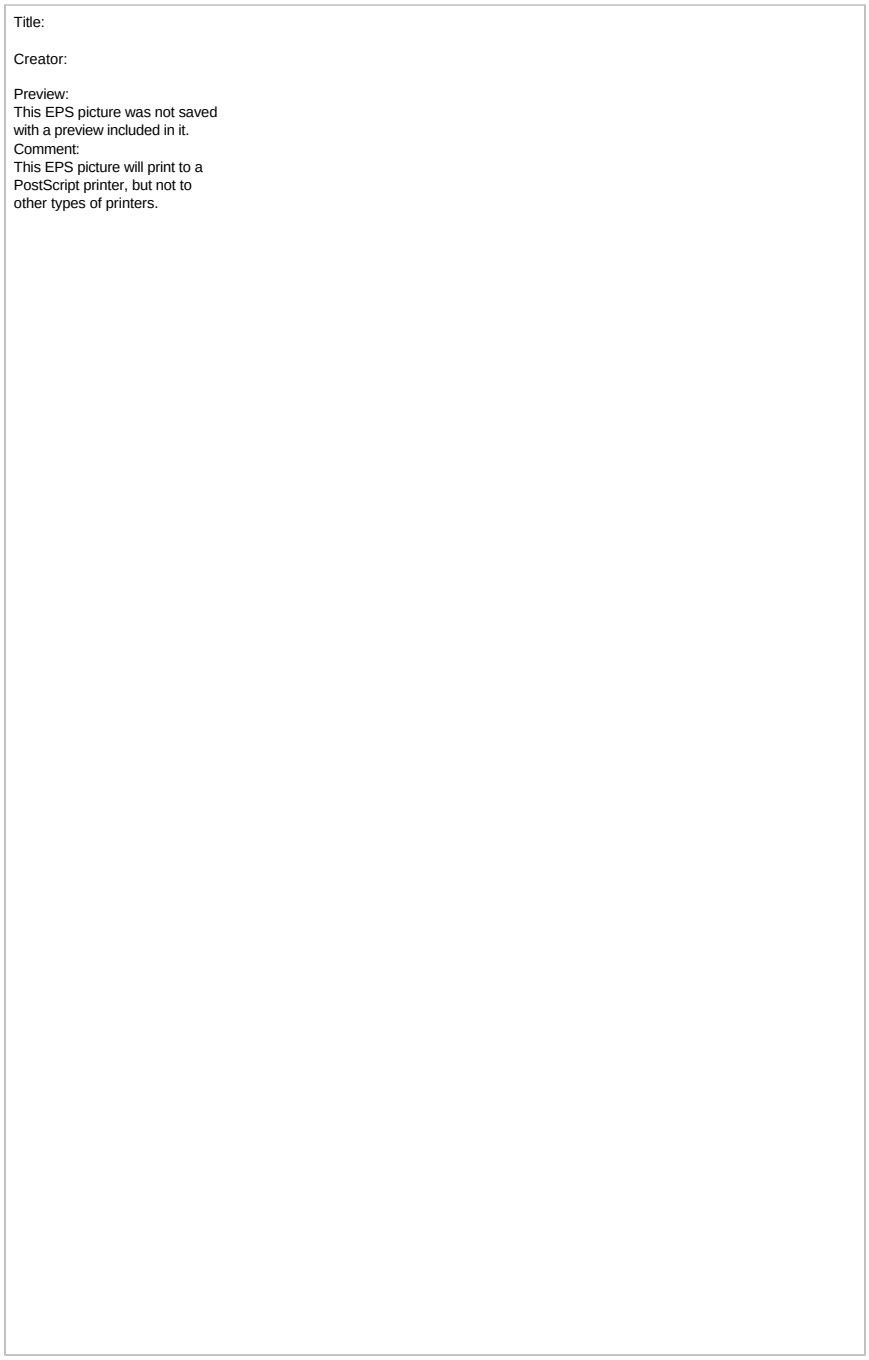

Figure 14: Two cells from the Chimaera FPGA, including the most complex bitslice of the Brent-Kung adder. The adder logic is the isolated logic in the upper right of each cell (4 blocks in the right cell, 5 in the left). Metal3 routing (not shown) for other purposes occupies the empty space in the adder. The logic for the entire adder represents approximately 8% of the chip area.

High performance adders are not limited to simple ripple carry schemes, and in fact rely on more advanced formulations to speed up their computation. However, as we demonstrated in this paper, the demands of FPGA-based carry chains are different than standard adders, especially because of the "inverse propagate" cell state. Thus, we cannot directly take standard high performance adder carry chains and embed them into current FPGA architectures.

In this paper we developed novel high performance carry chain structures appropriate to reconfigurable systems. These include implementations of Carry Select, Variable Block, and Carry Lookahead (including Brent-Kung) adders. We have been able to produce a carry chain that is up to a factor of 3.8 faster than current FPGA structures while maintaining all the flexibility of current systems. This provides a significant performance boost for the implementation of future FPGA-based systems.

Acknowledgments

This research was funded in part by DARPA contract DABT63-97-C-0035 and NSF grant CDA-9703228.

References

[Altera95] *Data Book*. San Jose, CA: Altera Corp., 1995.

[Brent82] R. P. Brent, H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Transactions on Computers*, Vol. C-31, No. 3, March 1982.

[Hashemian94] R. Hashemian, "Algorithm and Design Procedure for High Speed Carry Select Adders using FPGA Technology", *Proceedings of the 37th Midwest Symposium on Circuits and Systems*, Vol. 1, 1994.

[Hauck97] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[Oklobdzija88] V. G. Oklobdzija, E. R. Barnes, "On Implementing Addition in VLSI Technology", *Journal of Parallel and Distributed Computing*, Vol. 5, No. 6, pp. 716-728, December 1988.

[Sklansky60] J. Sklansky, "Conditional Sum Addition Logic", *IRE Transactions on Electronic Computers*, Vol. EC-9, No. 6, pp. 226-231, June 1960.

[Sutherland90] I. E. Sutherland, R. F. Sproull, *Logical Effort: Designing Fast MOS Circuits*. Palo Alto, CA: Sutherland, Sproull, and Associates, 1990.

[Woo95] N.-S. Woo, "Revisiting the Cascade Circuit in Logic Cells of Lookup Table Based FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 90-96, 1995.

[Xilinx96] *The Programmable Logic Data Book.* San Jose, CA: Xilinx Corp., 1996.

[Xing98] S. Xing, W. H. Yu, "FPGA Adders: Performance Evaluation and Optimal Design", *IEEE Design and Test of Computers*, Vol. 15, No. 1, pp. 24-29, 1998.

[Yu96] W. H. Yu, S. Xing, "Performance Evaluation of FPGA Implementations of High-Speed Addition Algorithms", *Proceedings of SPIE*, Vol. 2914, 1996.