High Performance Carry Chains for FPGAs

Matthew M. Hosler

Department of Electrical and Computer Engineering

Northwestern University

Abstract

Carry chains are an important consideration for most computations, including FPGAs, because they are often on the critical path. Current FPGAs dedicate a portion of their logic to support these demands via a simple ripple carry scheme. This thesis demonstrates how more advanced carry constructs can be integrated into FPGAs, thus providing significantly higher performance carry chains. The standard ripple carry chain is redesigned to reduce the number of logic levels in each cell. Additionally, entirely new carry structures are developed based on high performance adders such as Carry Select, Carry Lookahead, and Brent-Kung. Overall, these optimizations achieve a speedup of 3.8 times over current architectures.

Introduction

A Field-Programmable Gate Array (FPGA) is a device which can be programmed to implement almost any digital logic circuit. It is different in structure than other traditional logic circuits such as Full Custom Integrated Circuits (IC), Programmable Logic Devices (PLDs), Standard Cells, and Gate Arrays. An FPGA implements any combinational logic function by connecting many logic blocks. These logic blocks are normally arranged on the chip in a regular structure (such as a square matrix) as shown in Figure 1. Each logic block usually contains look-up tables (LUTs), flip-flops, and wires which can connect to other logic blocks. An n-input LUT can implement any function of n inputs. The output of the LUT may then connect to another logic function, to the input of a flip-flop or to the I/O devices. The configuration of each logic block is determined by SRAM bits which are connected to the circuit elements. The values of the SRAM bits are determined when the configuration data is loaded into the FPGA's internal memory cells. This

configuration data is loaded into the FPGA from some external device such as a Read-Only Memory (ROM). Therefore, the FPGA can be reconfigured an unlimited number of times just by reloading a new set of values from the external device. Additionally, the newest FPGAs allow a user to reprogram one subsection of the FPGA dynamically without changing the remaining sections of the FPGA. Thus, the FPGA can be used in systems where the hardware is changed dynamically to adapt to different user applications.

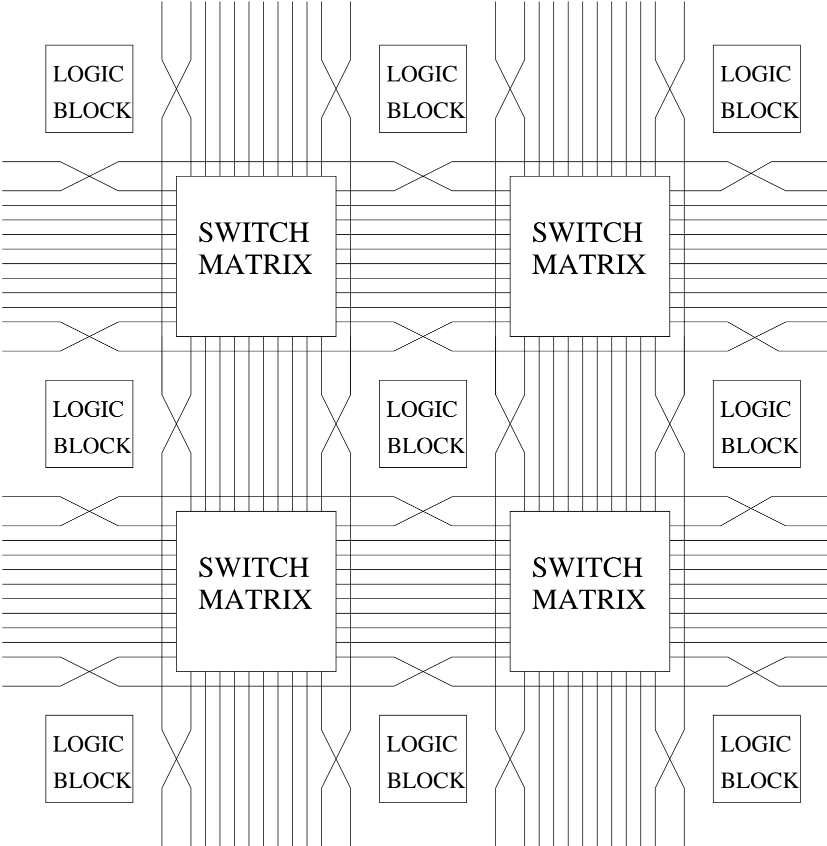


Figure 1: The logic and routing structure of a typical FPGA.

The primary difference between an FPGA and other traditional logic circuits is that the FPGA is completely fabricated before it has been customized with a design. This is different from a Full Custom design in that the Full Custom design is not fabricated until the design in complete. Standard cell designs speed up the process of designing a circuit, but also require the chip to be fabricated from a blank silicon wafer after the design is finished. Gate Arrays have a pattern of transistors and contacts that are pre-defined. Therefore, the patterns of transistors can be fabricated before the design process begins. However, the routing portion of the design, which

shows how the transistors are connected to one another, must still be configured after the design process is complete. A PLD device is completely fabricated before the design process begins, but requires programming by the user after the design phase. Since the process of programming the PLD can be done by the user, it does not have to be sent to a foundry for further fabrication. Therefore, both the FPGA and PLD have an advantage over other logic circuits because their design time is short and their chips do not have to be shipped to a foundry and be fabricated after the design process is completed.

A second advantage of an FPGA over other traditional circuits is that it is reprogrammable. A Full Custom integrated circuit can not be reprogrammed. A new design requires a completely new fabrication process with a large amount of additional cost and time. Standard Cell and Gate Array designs would also require a new fabrication process. However, a PLD could be changed simply by having the user reprogram the chip. This change could not be performed while the chip is in use, however, and would have to take place before using the chip. An FPGA, on the other hand, allows quick and simple reprogrammability. To reprogram an FPGA, one only has to send different values to the SRAM bits. Thus, the FPGA can even be reprogrammed while the chip is in use. Furthermore, some FPGAs even allow for partial reconfiguration of the chip. Thus, the FPGA gives a user the simplest and most efficient way of dynamically reprogramming a chip.

The third advantage of an FPGA is cost. The cost of a Full Custom integrated circuit is very high for the first chip, but can be amortized if millions of chips are produced. The cost of an FPGA is much smaller for the first chip. Therefore, if a small number of chips are being produced, the FPGA can be a cheaper alternative to the Full Custom design. However, if millions of chips are being produced, the Full Custom design will be the more cost effective method.

Yet the FPGA does have a disadvantage. The FPGA is fabricated before the design process has started in order to have the logic already available on the FPGA, so that the FPGA can be quickly configured to the user's specifications. Since the FPGA is pre-fabricated, the logic contained within the FPGA is not optimized for a particular design, and therefore the FPGA has a lower logic density than a Full Custom, Standard Cell, or Gate Array design. Thus, the chip area needed

to implement a design in an FPGA is greater than the chip area required for a Full Custom design. Additionally, the Full Custom design will have a higher clock speed than its FPGA counterpart, due to the fact that timing and routing issues for the Full Custom design are user optimized.

Thus, while Full Custom circuits have the advantage of having a faster clock speed, the FPGA is both cheaper for small quantities and has a shorter design time. For example, a custom fabricated circuit typically takes weeks to be fabricated, whereas an FPGA design can be customized in milliseconds. Therefore, a logic design implemented in an FPGA can reach the market more quickly. So if the speed of the logic circuit is not an issue, the FPGA may be a more desirable option.

The FPGA would be an even better logic implementation option if its logic density could be improved and its clock speed could be increased. The key to achieving high performance in any circuit, and thus improving the clock speed, is to optimize the circuit's critical path. For most datapath circuits this critical path goes through the carry chain for arithmetic and logic operations. A carry chain is a logical structure that allows one cell to pass information about a calculation to another cell. A classic example of a carry chain's use is in addition. When one adds the numbers 6 and 7 together in the decimal system, the result is 13. However, since a digit in the decimal system can only range from 0 to 9, the value 13 can not be used as the result for that column. The next column, though, is the "tens" column where each value is equivalent to ten times the first column. Therefore, the result can be recorded by placing a result of 3 in the first decimal position, and then "carrying" the remaining 10 units over to the "tens" column as 1 "tens" unit. The value of 1 that is carried to the second decimal position is then added to the calculation of that decimal position. Since values must be able to be passed or carried from one bit position to another, a logic structure known as a carry chain (see Figure 2) must be created in order to facilitate the carry.

In an arithmetic circuit such as an adder or subtractor, this carry 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.

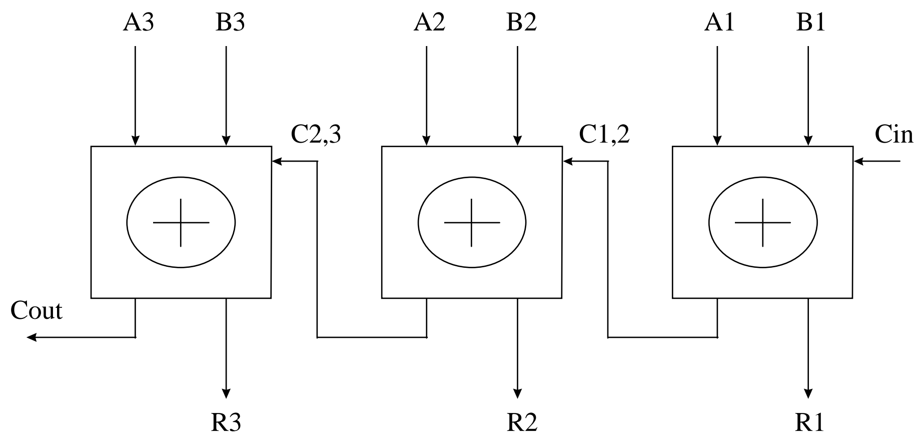


Figure 2: A simple carry chain.

In order to support datapath computations, most FPGAs include special resources specifically optimized for implementing carry computations. These resources significantly improve circuit performance with a relatively insignificant increase in chip area. However, because these resources use a relatively simple Ripple Carry scheme, carry computations can still be a major performance bottleneck. This thesis discusses methods for significantly improving the performance of carry computations in FPGAs. Thus, the clock speed of the FPGA could be increased, and the FPGA would be an even more desirable option for logic implementation.

Basic Ripple Carry Cell

A basic ripple carry cell, similar to that found in the Altera 8000 series FPGAs [Altera95], is shown in Figure 3a. Mux 1, combined with the two 2-LUTs feeding into it, creates a 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 is contained in each cell, which can be used to compute the sum for addition or other functions.

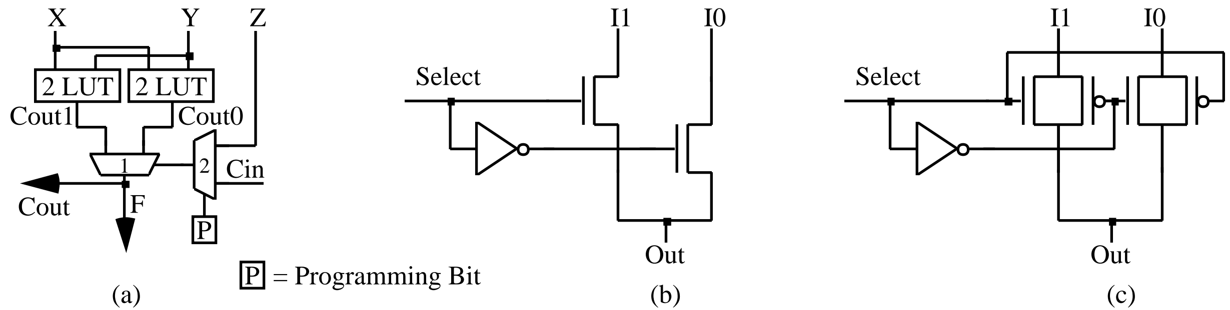


Figure 3: Carry computation element for FPGAs (a), a simple 2:1 mux implementation (b), and a slightly more complex version (c).

It is important to understand the role of the "Cout1" and "Cout0" signals in the carry chain. 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. There are four possible combinations of values that Cout1 and Cout0 can assume, 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. This state will be referred to as "inverse propagate". 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 such as parity-checkers, and therefore supporting this state in the carry chain increases the types of circuits that can be efficiently implemented.

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.

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, the first cell in the carry chain must be programmed 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 both 2-LUTs are programmed 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 3b shows an implementation of a mux that does not obey this requirement. If the select signal to this mux is stuck midway between true and false (2.5V for 5V CMOS) 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 3c, with both n-transistor and p-transistor pass gates, will operate properly for this case. Thus, it is assumed throughout this thesis that all muxes in the carry chain are built with the circuit shown in Figure 3c, though any other mux implementation with the same property could be used (including tri-state 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 thesis, a simple 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 3c). 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 thesis. This delay model will be used to initially discuss different carry chain alternatives and their advantages and disadvantages. Precise circuit timings were also generated using Spice on the VLSI layouts of the carry chains, as discussed later in this thesis.

Optimized Ripple Carry Cell

As discussed in an earlier section, the ripple carry design of Figure 3a is capable of implementing most 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 it is assumed that 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.

The delay of the ripple carry chain can be reduced by removing mux 2 from the carry path. As shown in Figure 4a, instead of choosing between Cin or Z for the select line to the output mux,

there are now two separate muxes labeled 1 and 2 which are controlled by Cin and Z, respectively. The circuit then chooses between the outputs of muxes 1 and 2 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.

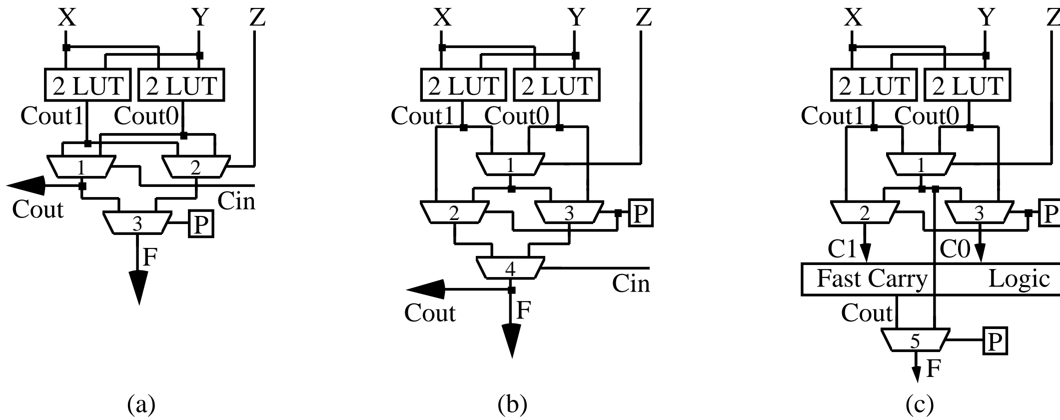


Figure 4: Carry computation elements with faster carry propagation

Unfortunately, the circuit in Figure 4a 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 4b. For cells in the middle of a carry chain mux 2 is configured to pass Cout1 and mux 3 is configured to pass Cout0. Thus, mux 4 receives Cout1 and Cout0 and provides a standard ripple carry path. However, when a carry chain begins with a carry input (provided by input Z), mux 2 and mux 3 are then configured so they 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 from Cout1 and Cout2, respectively, the circuit is configured to continue the carry chain. This design is therefore functionally equivalent to the design in Figure 3a. 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 2*n+1 for an n-bit carry chain. Thus, although this design is 1 gate delay slower than that of Figure 4a, 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 4b can be configured to bypass mux 1, reducing the overall delay to 2*n, which is identical to that of Figure 4a. On the other hand, in order to implement a n-bit carry chain with a carry input, the design of Figure 4b requires an additional cell at the beginning of the chain to bring in this input, resulting in a delay of 2*(n+1) = 2*n+2, which is slower than that of the design in Figure 4a. Thus, the design of Figure 4b is the preferred ripple carry design among those presented so far.

Dual-Rail Optimization

In all of the carry chains discussed in this thesis the primary computation element is a mux. The carry flows from previous stages in the logic to the control input of a mux, where it computes the cell's Cout value. It is important to realize that although this mux looks like a single element, there is in fact an inverter embedded inside the mux (see Figure 3c). Thus, according to the simple delay model, there is only one gate delay for a signal to go from one of the normal inputs of the 2:1 mux to the output, but there are two gate delays to go from the select input to the output. Thus, it seems possible that the carry chain delay could be decreased by moving the inverter off of the carry chain path, so that each mux has only one gate delay instead of two.

The inverter computes the inverse of the select input, which is used to select which normal input should be connected to the output. Thus, if the inverse of the control input was already available, this inverter would no longer be needed. The inverse can be generated by essentially duplicating the ripple carry chain in each cell. Instead of just computing the normal value of the Cout signal in each cell, its inverse is also computed. This is done by inverting the inputs to this mux, as shown in Figure 5. Mux 4a computes $Cout$, while mux 4b computes $\overline{Cout}$. Note that while this should speed up the propagation through intermediate cells on a carry chain, it does add an extra initial delay to the first stage. Overall, this yields a delay of n+3 for an n-bit carry chain, which is approximately twice as fast as the carry chain of Figure 4b and three times faster than the basic

ripple carry scheme. However, when the Dual Rail optimization was actually implemented in VLSI layout, the resultant Spice timing values did not produce the savings that was anticipated by this analysis. The results of the Dual Rail optimization technique will be discussed in more detail later in this thesis.

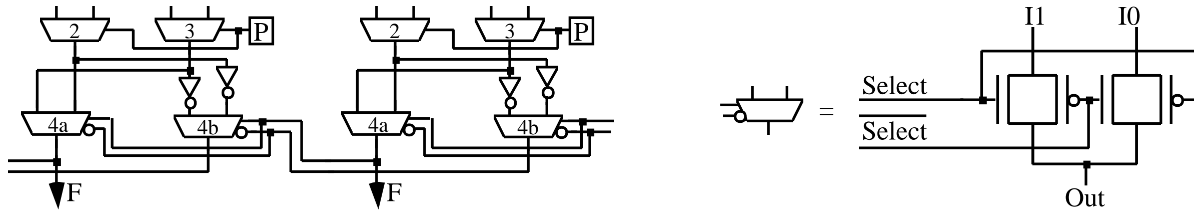


Figure 5: Dual rail optimization of the ripple carry structure. A possible implementation of the special 2:1 mux used in this mapping is shown at right. The circuit shown at left represents the carry structure for two adjacent cells, and replaces mux 4 inFigure 4b.

High-Performance Carry Logic for FPGAs

In the previous sections, methods to optimize a ripple carry chain structure for use in FPGAs were presented. 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 has been tremendous amounts of work on developing alternative carry chain schemes which overcome the linear delay growth of ripple-carry adders. Although these techniques have not yet been applied to FPGAs, this thesis will demonstrate how these advanced adder techniques can be integrated into FPGAs.

The basis for all of the high-performance carry chains developed in this thesis will be the carry cell of Figure 4c. This cell is very similar to that of Figure 4b, 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 4c is that any logic that can compute the value $Cout_i = (Cout_{i-1} * C1_i) + (\overline{Cout}_{i-1} * C0_i)$, where i is the position of the cell within the carry chain, can provide the functionality necessary to support the needs of reconfigurable carry computations. Thus, the fast carry logic unit can contain any logic structure implementing this computation. This thesis looks 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 reconfigurable logic, new circuits will have to be developed, inspired by the standard adder structures, but which are more appropriate for FPGAs. The main difference is that the carry chains must support not only the Generate, Propagate, and Kill states of an adder, but also the Inverse Propagate state. 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, for FPGAs the delay concerns 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, the carry chain not only must support the carry delay from the first to the last position, but must also consider the delay for carry computations 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 Carry Chain

The problem with a ripple carry structure is that the computation of the Cout for a 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 6. 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, two ripple carry adders are used, 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 ripple carry adders based upon the actual Cin (from mux 5). Subsequent stages continue to grow in length by one, with cells 7-10 in one block, cells 11-15 in another, and so on. Delay values showing the delay of the Carry Select carry chain relative to other carry chains will be presented later in this thesis.

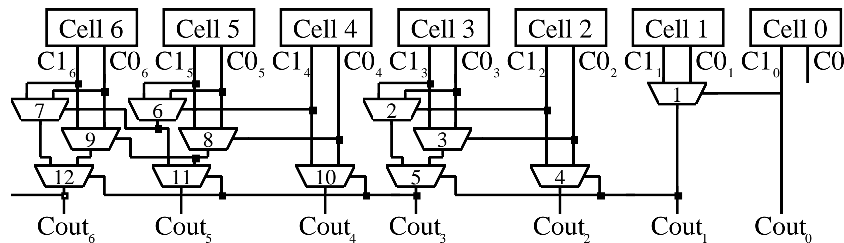


Figure 6: Carry Select structure.

Variable Block Carry Chain

Like the Carry Select carry chain, a Variable Block structure [Oklobdzija88] consists of blocks of ripple carry elements. 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.

A major difficulty in developing a version of the Variable Block carry chain (see Figure 7) for inclusion in an FPGA's architecture is the need to support both the propagate and inverse propagate state of the cells. Unfortunately, this required that significant changes to the Variable Block adder structure be made. The new structure requires two new values to be computed: a propagate signal and an invert signal. First, the cells are checked to see if they 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, the Cout function will be equal to either Cin or $\overline{Cin}$. To decide whether to invert the signal or not, the number of cells that are in inverse propagate mode must be determined. 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, only the C0 signal from each cell is considered. If this signal is true, the cell is in either generate or inverse propagate mode. If it is in generate mode the inversion signal will be ignored anyway, since the Cin signal is only inverted if all cells are in some form of propagate mode. Note that for both of these tests a tree of gates can be used to compute the result. Also, since the inversion signal is ignored when the carry chain is not bypassed, C1 can be used 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. Block lengths (from low order to high order cells) of 2, 2, 4, 5, 7, 5, 4, 2, 1 for a 32 bit structure was used. 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 thesis.

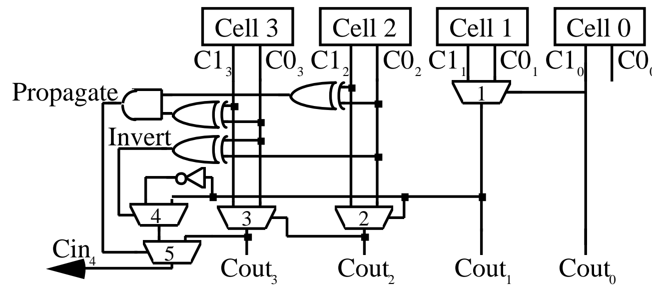


Figure 7: The Variable Block carry structure. Mux 1 performs an initial two stage ripple carry. Muxes 2 through 5 form a 2-bit Variable Block block. Mux 5 decides whether the Cin signal should be sent directly to Cout, while mux 4 decides whether to invert the Cin signal or not.

Carry Lookahead and Brent-Kung Carry Chains

There are two inputs to the fast carry logic in Figure 4c: $C1_i$ and $C0_i$. The values of $C1_i$ have already been generated by the LUTs. If Cin_i is 1, the output of the mux, $Cout_i$ is $C1_i$. If Cin_i is 0, the output of the mux, is $C0_i$. The information represented by $C1_i$ and $C0_i$ can be combined together to determine what the Cout of two stages will be if the Cin of the first stage is given. For example, $C1_{i,i-1} = (C1_{i-1} * C1_i) + (\overline{C1_{i-1}} * C0_i)$ and $C0_{i,i-1} = (C0_{i-1} * C1_i) + (\overline{C0_{i-1}} * C0_i)$, where $C1_{x,y}$

is the value of $Cout_x$ assuming that $Cin_y = 1$. The length of the carry chain can now be halved, 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, $C1_{i,k} = (C1_{j-1,k} * C1_{i,j}) + (\overline{C1_{j-1,k}} * C0_{i,j})$ and $C0_{i,k} = (C0_{j-1,k} * C1_{i,j}) + (\overline{C0_{j-1,k}} * C0_{i,j})$, 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 $C1_{i,0}$ and $C0_{i,0}$ has been computed for each cell *i*. A string of muxes at the bottom of the Brent-Kung carry chain can then use the values precomputed by the concatenation boxes to compute the Cout for each cell when its Cin is given The Brent-Kung carry chain is shown in Figure 8.

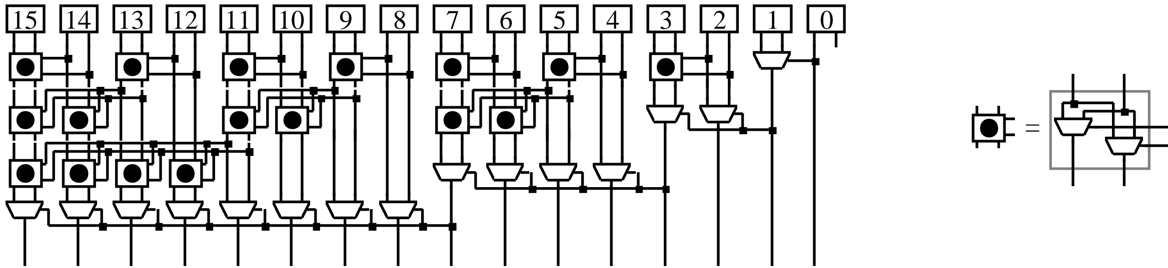


Figure 8: The 16 bit Brent-Kung structure. At right is the details of the concatenation block. Note that once the Cin has been computed for a given stage, a simple mux can be used in place of a concatenation block.

The Brent-Kung adder is a specific case of the more general Carry Lookahead adder. 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. However, while a combining factor of 4 is considered optimal for a standard adder, in a reconfigurable system 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 the following equation is used:

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

Since this computation is more than twice as complex as the computation needed to concatenate two cells together, one can conclude that concatenating pairs is preferable over concatenating 3 cells together. However, it is not immediately clear whether the concatenation of cells in groups of 4 would be a better approach.

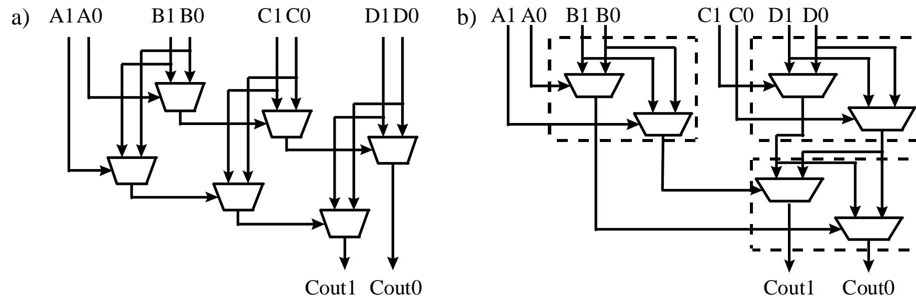


Figure 9: Concatenation boxes. (a) a 4-cell concatenation box, and (b) its equivalent made up of only 2-cell concatenation boxes.

Figure 9a shows a concatenation box that takes its input from 4 different cells. Figure 9b then shows how a 4-cell concatenation box can be built using 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 9a is 3 units since the signal must travel through 3 muxes. The delay for the 4-cell concatenation box equivalent found in Figure 9b, however, is only 2 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 in a 2-Level Carry Lookahead structure.

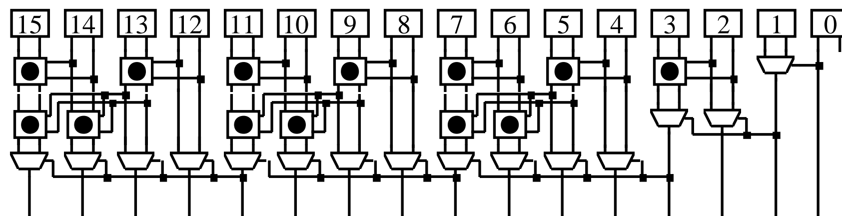


Figure 10: A 2-Level, 16 bit Carry Lookahead structure.

Another option in Carry Lookahead adders is the possibility of using less levels of concatenation than in 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 $C1_{i,o}$ and $C0_{i,0}$ values. Fewer levels than the complete hierarchy of the Brent-Kung adder can be used, if one simply ripples together the top-level carry computations of smaller carry-Lookahead adders. Specifically, a N-level Carry Lookahead adder would be the name for N levels of 2-input concatenation units. A 2-Level Carry Lookahead adder is shown in Figure 10. Delay values showing the delay of the Brent-Kung and Carry Lookahead carry chains relative to other carry chains will be presented next.

Carry Chain Performance

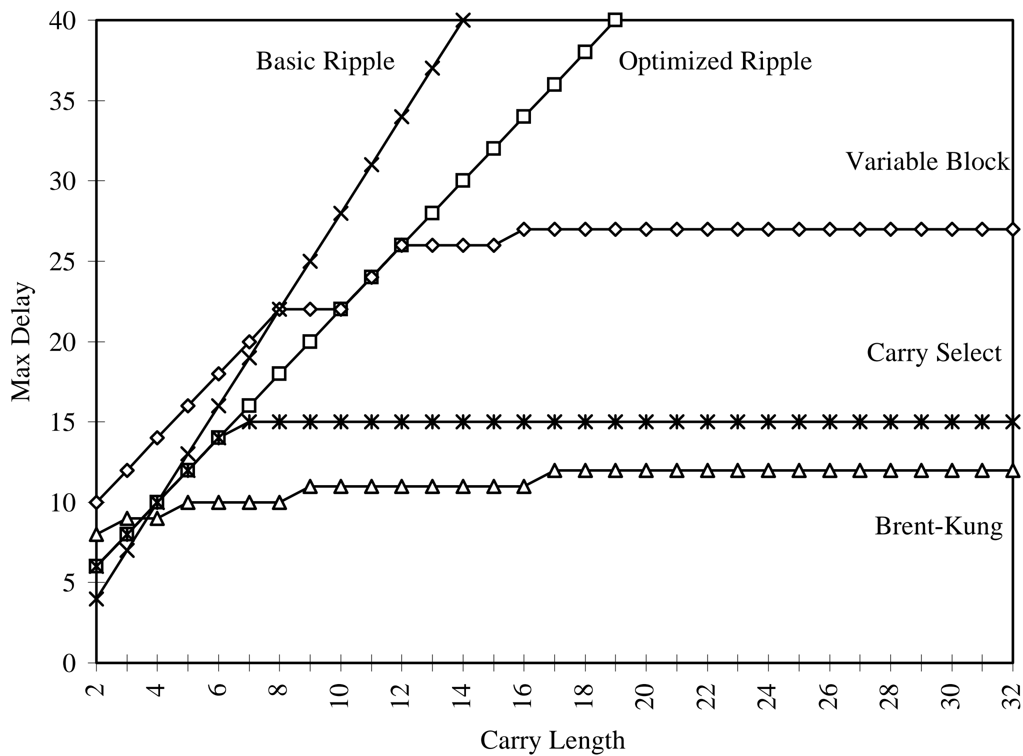


Figure 11: A comparison of the various carry chain structures.

In order to compare the carry chains developed in this thesis, the performance of the carry chains of different lengths is computed. The delay is computed from the output of the 2-LUTs in one cell to the final output (F) in another. One important issue to consider is what delay to measure. 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, it is assumed that the

FPGAs are built to support up to a 32 bit carry chain, and the maximum carry chain delay for any length L carry computation within this structure is then recorded. That is, since it is not known where the user will begin their carry computation within the FPGA architecture, the worst case delay for a length L carry computation starting at any point in the FPGA is measured instead. 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.

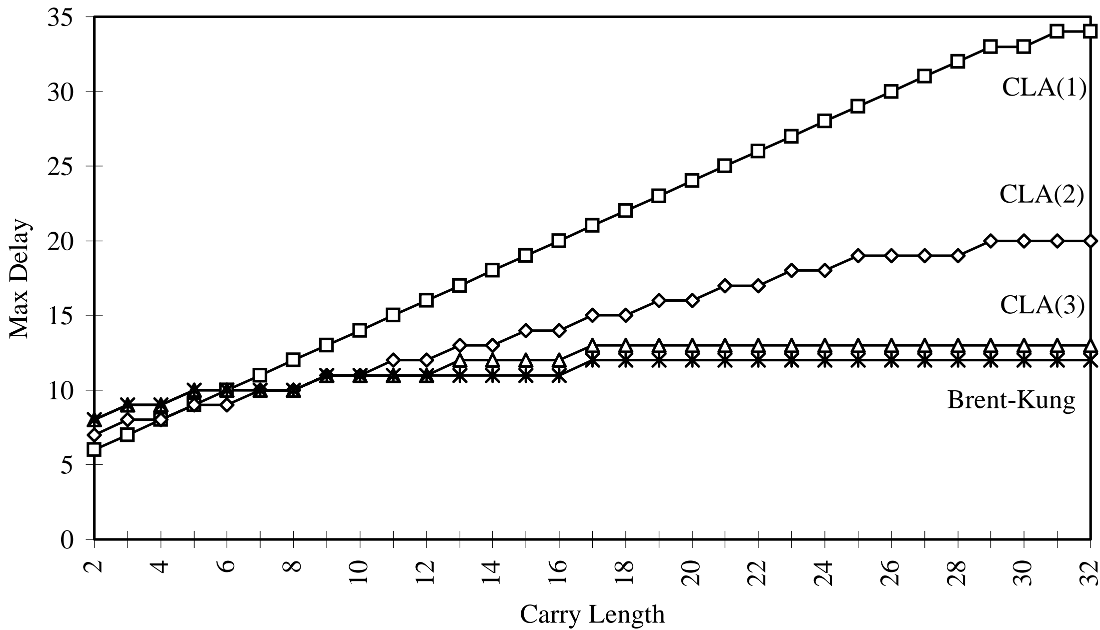


Figure 12: A comparison of Carry Lookahead structures.

Figure 11 shows the maximum carry delays for each of the carry structures discussed in this thesis, 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 implementations 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 six 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 eight times slower for 32 bit computations. However, short carries are not as critical, since they can usually be supported by the FPGA's normal routing

structure. Thus, the short carries are less likely than the 32 bit carries to dominate the performance of the overall system. Therefore, the Brent-Kung structure is the preferred structure for FPGA carry computations, since it is capable of providing significant performance improvement over current FPGA carry chains.

This thesis also considers other types of Carry Lookahead adder designs which do not use as many levels of concatenation boxes as a full Brent-Kung adder. However, as can be seen from Figure 12, 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.

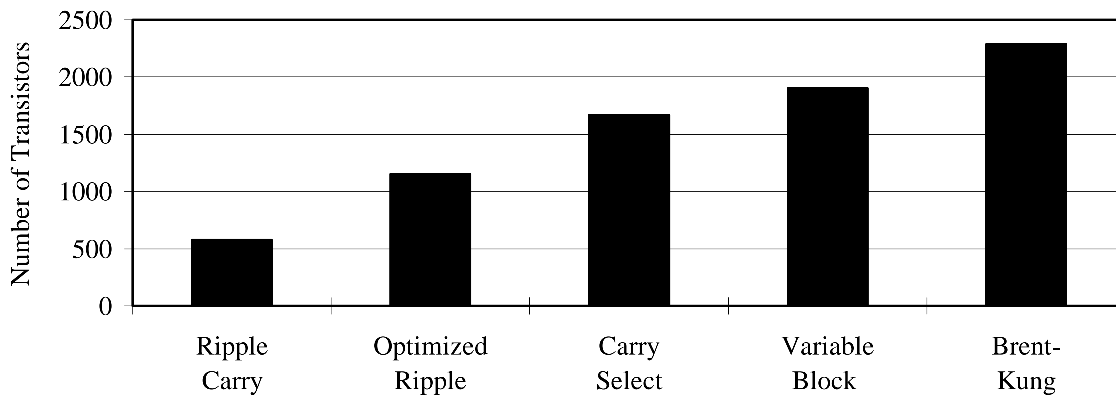


Figure 13: The Transistor counts of the Ripple Carry, Optimized Ripple, Carry Select, Variable Bit, and Brent-Kung carry chains.

Another consideration when choosing a carry chain structure is the size of the circuit. Figure 13 shows the number of transistors that are used in the design of the simple Ripple Carry, Optimized Ripple Carry, Carry Select, Variable Block, and Brent-Kung carry chains. The transistor counts here are based on a CMOS implementation of the tri-state mux, which has 8 transistors, and is shown in Appendix B. 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 ripple 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 thesis 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 simple Ripple Carry chains will be discussed further in the next section of this thesis.

Layout Results

Carry Chain	32-bit delay (ns)	3-LUT delay (ns)
Ripple Carry (Mux)	23.4	1.6
Ripple Carry (Complex Logic)	25.4	1.9
Optimized Ripple	18.7	2.5
Dual Rail Optimized Ripple	21.2	2.7
Brent-Kung	6.1	2.1
Dual Rail 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).

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 simple Ripple Carry, which can be found in current FPGAs, as well as the new Optimized Ripple and Brent-Kung carry chains. Additionally, Dual Rail Brent-Kung and Dual Rail Optimized Ripple carry chains were also implemented in VLSI to determine whether the dual rail optimization can increase performance. Diagrams showing the VLSI layouts can be found in Appendix C.

Table 2 shows the delays of a 32-bit carry for the carry chains that were implemented. Notice that the delay for simple Ripple Carry chain is 23.4ns, and the delay for the Brent-Kung carry chain is 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. One item to note is that two versions of the simple Ripple Carry chain were created. The first version used muxes to implement the design, while the second version used complex gates (see Appendix B for the transistor diagram). The delay of the Mux version was 23.4ns while the delay for the Complex Logic version was 25.4ns. Thus it appears that the Mux implementation is somewhat faster than the Complex Logic version of the design.

Another item to note is that the delay of the Dual Rail Ripple carry chain is 21.2ns while the delay for the Optimized Ripple carry chain is only 18.7ns. Thus, the application of the Dual Rail signaling protocol actually increased the delay of the Optimized Ripple carry chain by 13.4%. For the Brent-Kung design, the delay was 6.1ns, and for the Dual Rail Brent-Kung design, the delay was also 6.1ns. Thus, the Dual Rail signaling protocol did not reduced the delay of the Brent-Kung carry chain. Therefore, the timing results seem to indicate that the dual rail optimization yields little or no improvement. Appendix D contains timing numbers for variable length carries of the various carry chains.

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 4c). The delay for the simple 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.

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 simple Ripple carry chain. This number should be viewed purely as an upper bound, since the layout of the simple 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.

Carry Chain	Area	% Increase for Chimaera FPGA	% Increase for General-Purpose FPGA
Ripple Carry (Mux)	171,368	0	0
Ripple Carry (Complex Logic)	226,859	0.3	0.04
Optimized Ripple	394,953	1.3	0.18
Dual Rail Optimized Ripple	484,640	1.8	0.25
Brent-Kung	1,622,070	8.5	1.18
Dual Rail Brent-Kung	1,256,044	6.3	0.88

Table 3: Areas of different carry chain implementations.

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 simple 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.

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 ripple carry structure found in current FPGAs several innovative techniques were developed. 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 delays. Also, a Dual Rail signaling protocol was investigated.

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 demonstrated in this thesis, the demands of FPGA-based carry chains are different than standard adders, especially because of variable length carries and the "inverse propagate" cell state. Thus, standard high performance adder carry chains can not be directly taken and embedded into current FPGA architectures.

In this thesis, novel high performance carry chain structures appropriate to reconfigurable systems were developed. These include implementations of the Carry Select, Variable Block, and Carry Lookahead (including Brent-Kung) adders. A carry chain was produced that is up to a factor of 3.8 times faster than current FPGA structures while maintaining the flexibility of current systems. This provides a significant performance boost for the implementation of future FPGA-based systems.

Future Work

Future work in this area could include a study of the dual rail signaling protocol. On paper, this technique appears to halve the delay of the carry chains it was applied to. However, Spice timings of the VLSI layouts show virtually no advantage to using the dual rail techniques. A study which explains this discrepancy would be interesting. Additional optimization of the VLSI layouts

would also be helpful in providing a more accurate comparison of the areas of the new carry chains and the simple Ripple Carry chain.

Acknowledgments

Special thanks to my advisor Scott Hauck for his assistance with this thesis and to Thomas Fry for his preliminary work in this area. This research was funded in part by the Defense Advanced Research Projects Agency and the National Science Foundation.

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.

[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.

[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,* pg 90-96, 1995.

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

Appendix A: Carry Logic and the FPGA Cell Structure

When carry chains are designed for an FPGA, inverters can be added within the design in various places in order to optimize the design. While adding inverters to a typical logical circuit might cause problems with the logical correctness of the design, inverters can be added to the FPGA without out causing this problem. The reason why an FPGA can support the addition of inverters when a typical circuit cannot is because the FPGA contains LUTs. Recall that an n-input LUT can produce any function of n variables. Thus, if inverters are added to the structure of the FPGA, one can just reprogram the LUT to produce an inverted function of the input variables instead. Thus, inverters can be added to the FPGA without a problem.

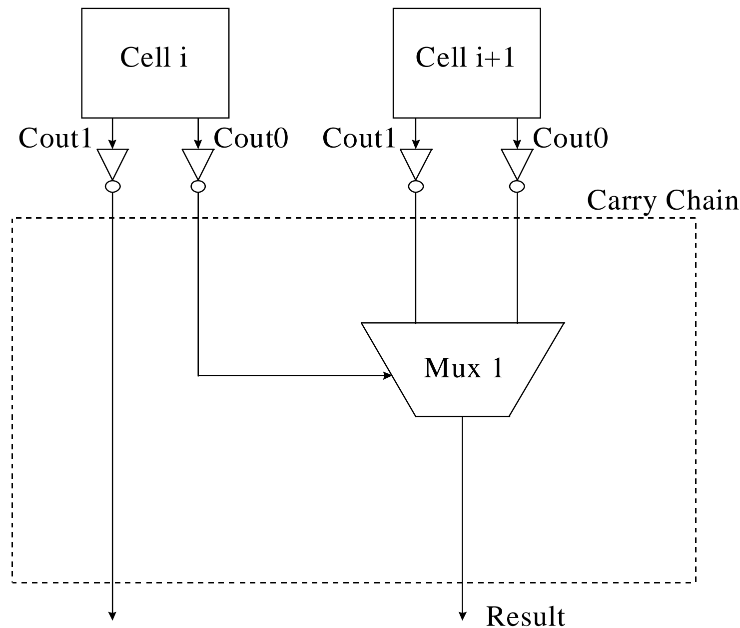


Figure 14: Logical incorrectness caused by adding inverters before a simple carry chain

Unfortunately, the addition of extra inverters in a FPGA cell could cause logical problems for the carry chains within that cell. Figure 14 shows a simple carry ripple scheme in which inverters were placed before all of the inputs to the carry chain. Unfortunately, the carry chain will not just produce an inverted output. Instead, the inversion of the Cout0 signal of the left LUT will cause the select line of Mux 1 to be inverted. The inversion of Mux 1's select line will cause Mux 1 to choose the wrong input, and therefore the output of Mux 1 will be incorrect. Thus, the inverters

in this example cause the carry chain to function incorrectly, instead of just inverting the outputs of the carry chain.

However, it is possible to fix this problem so that inverters can be added to the FPGA and so that the carry chain will still function properly. First, the problem must be simplified slightly. The problem assumes that there are chains of inverters that are placed within an FPGA cell either before of after the carry chain. Because two inverters in series produce a logical result equivalent to 0 inverters, any chain of inverters can be reduced to the logical equivalent of 0 inverters or 1 inverter. If there are the equivalent of 0 inverters in the FPGA cell, then there is no problem. Thus, there are only 2 cases to consider. Case 1 is that there is the equivalent of 1 inverter before the carry chain. Case 2 is that there is the equivalent of 1 inverter after the carry chain. Both cases will be discussed. Note that the solutions to these two cases can also be combined, allowing inverters to appear both before and after the carry chain.

First, Case 1 will be considered. As was discussed above, an inverted signal entering the carry chain will cause the select lines of a mux to choose the wrong input. Therefore, inverted inputs can not be allowed to enter the carry chain. However, the problem can still be solved. As you will recall, the two 2-LUTs in Figure 4c produce signals labeled Cout1 and Cout0. These outputs are generated by the 2-LUTs based on a user-programmable function of X and Y. Therefore, the LUTs can just be reprogrammed by the user to produce $\overline{Cout1}$ and $\overline{Cout0}$ instead of $Cout1$ and $Cout0$, respectively. Then when the logical inversion takes place before the carry chain, the inputs to the carry chain will still be equivalent to Cout1 and Cout0.

Now Case 2 will be considered. In this case, 1 inverter is added to the output of the carry chain. One initial solution might be to just reprogram the LUTs to output $\overline{Cout1}$ and $\overline{Cout0}$ so that the inversions cancel out. Unfortunately, this solution does not work, because if the inputs to the carry chain are inverted (as the result of changing the LUT outputs), then the select inputs of the muxes would again be inverted, causing the muxes to choose the wrong inputs and causing logical incorrectness. The solution to this problem however is to just reprogram the LUTs in a different manner. Instead of having the LUTs output Cout1 and Cout0, they are instead programmed to

output $\overline{Cout0}$ and $\overline{Cout1}$, respectively. Note that the outputs of the LUTs are both inverted and exchanged. The LUT that was previously outputting Cout1 is now generating the inversion of Cout0, and vice versa. Now, the carry chain works properly again. Inverting the inputs to the carry chain causes the select lines of the muxes to choose the wrong inputs. However, by switching the inputs also, the muxes end up choosing the correct input after all. Therefore, all of the outputs of the carry chain are now inverted. However, since there is one logical inverter after the carry chain, the final solution is equivalent to the original solution.

The rules in Case 1 and Case 2 can then be applied together to handle any structure of inverters. For example, if there are inverters both before and after the carry chains, then first Case 1 is applied to the cells to negate the inverters before the carry chain. Thus, $Cout1$ and $Cout0$ are inverted. Then Case 2 is applied to the cells so that the outputs of the LUT, $\overline{Cout1}$ and $\overline{Cout0}$ (as produced by Case 1), are inverted and switched. Thus, the final output of the LUTs for the case of inverters before and after the carry chain is Cout0 and Cout1, respectively. Therefore, any number of inversions may be placed before or after the carry chain without affecting its logical correctness.

Appendix B: Transistor diagrams

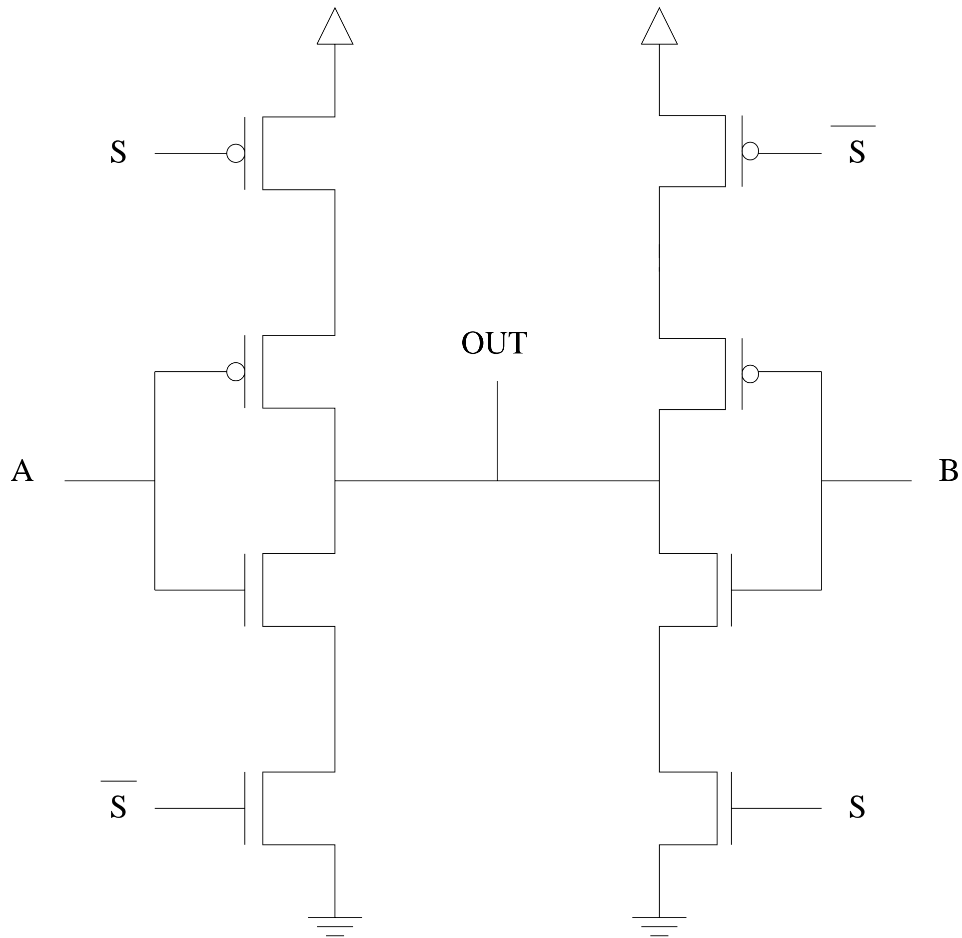


Figure 15: The transistor diagram of the inverting tri-state mux that was used in this thesis. A and B are the two inputs to the mux, and S and $\overline{S}$ are the select lines.

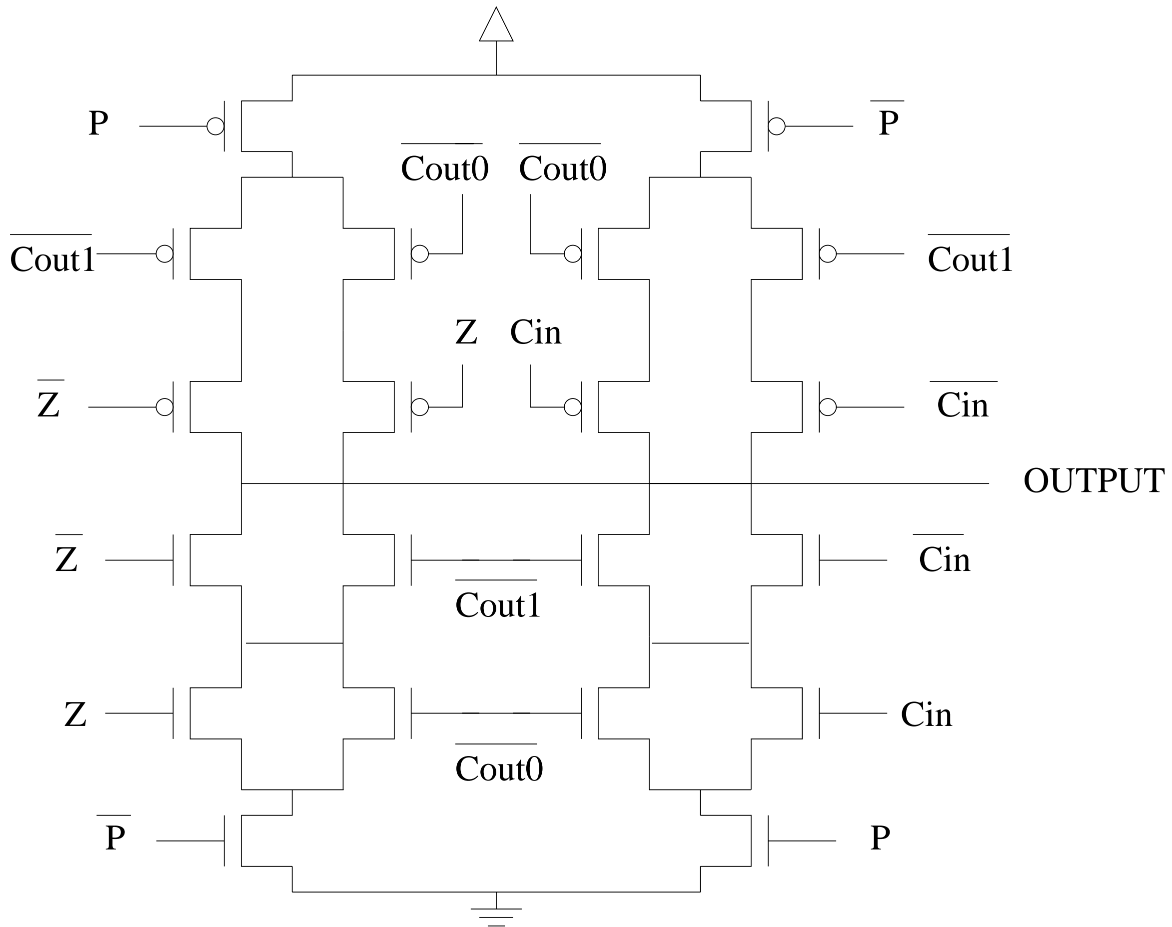


Figure 16: The transistor diagram of the simple Ripple Carry adder that was implemented using complex logic. Cout1 and Cout0 are the outputs of the two 2-LUTs. Cin is the carry input, Z is the third input, and P is the programming bit which determines whether a carry or 3-LUT function occurs.

Appendix C: Printouts of VLSI Layouts

Figure 17: The VLSI layout of the simple Ripple Carry adder implemented using muxes.

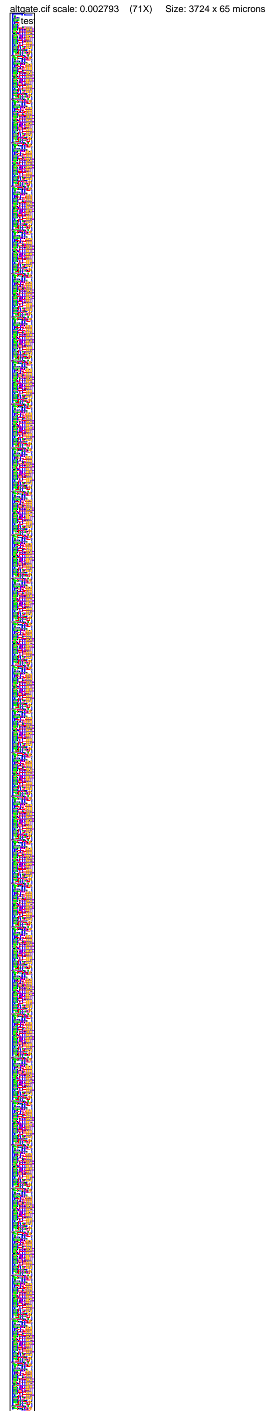


Figure 18: The VLSI layout of the simple Ripple Carry adder implemented using complex logic.

Figure 19: The VLSI of the Optimized Ripple carry chain.

Figure 20: The VLSI layout of the Dual Rail Optimized Ripple carry chain.

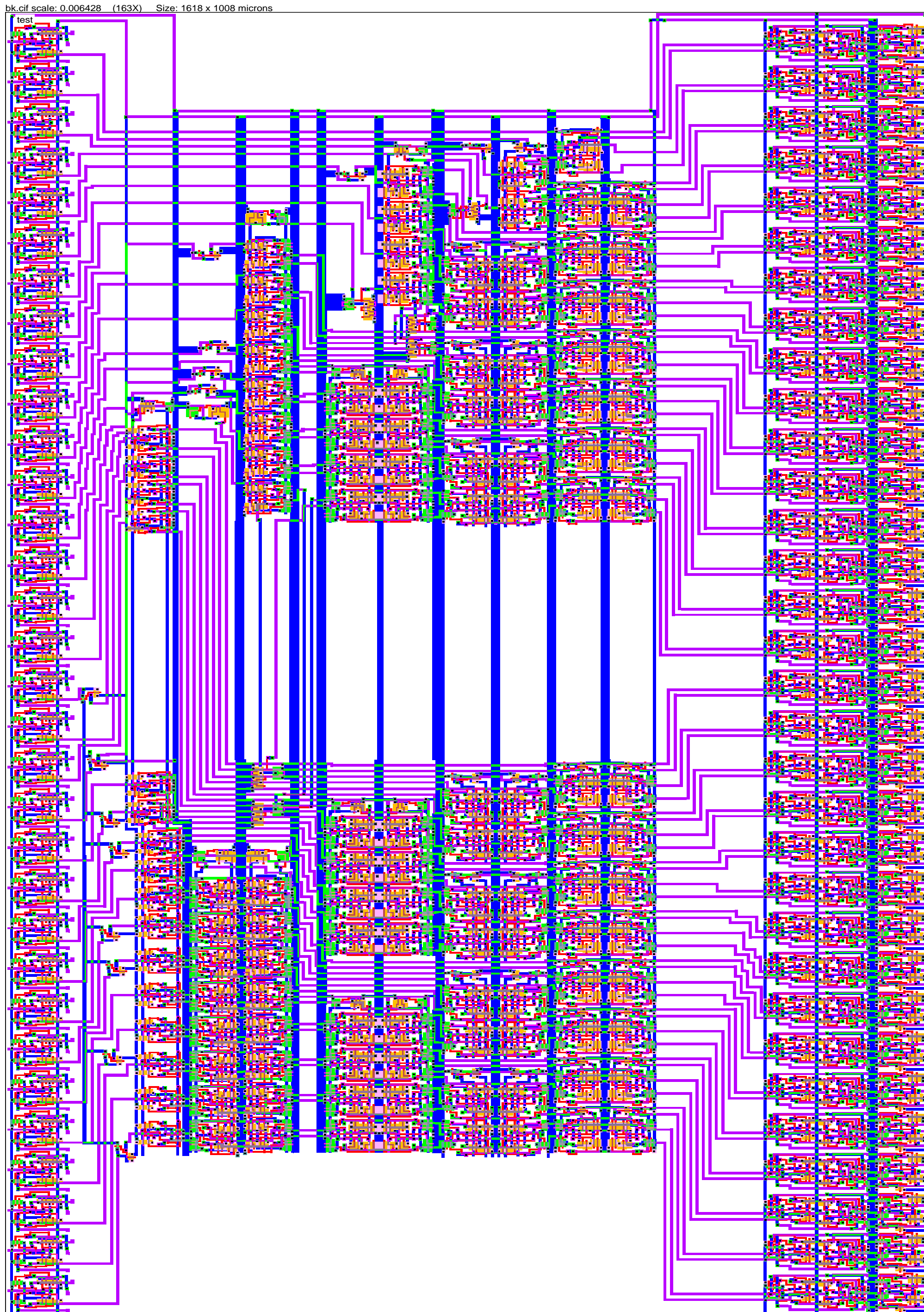


Figure 21: The VLSI layout of the Brent-Kung carry chain. The top and bottom sections are the implementation of the muxes shown Figure 4c. The middle section is the actual Brent-Kung carry chain and corresponds to the Fast Carry Logic section shown in Figure 4c.

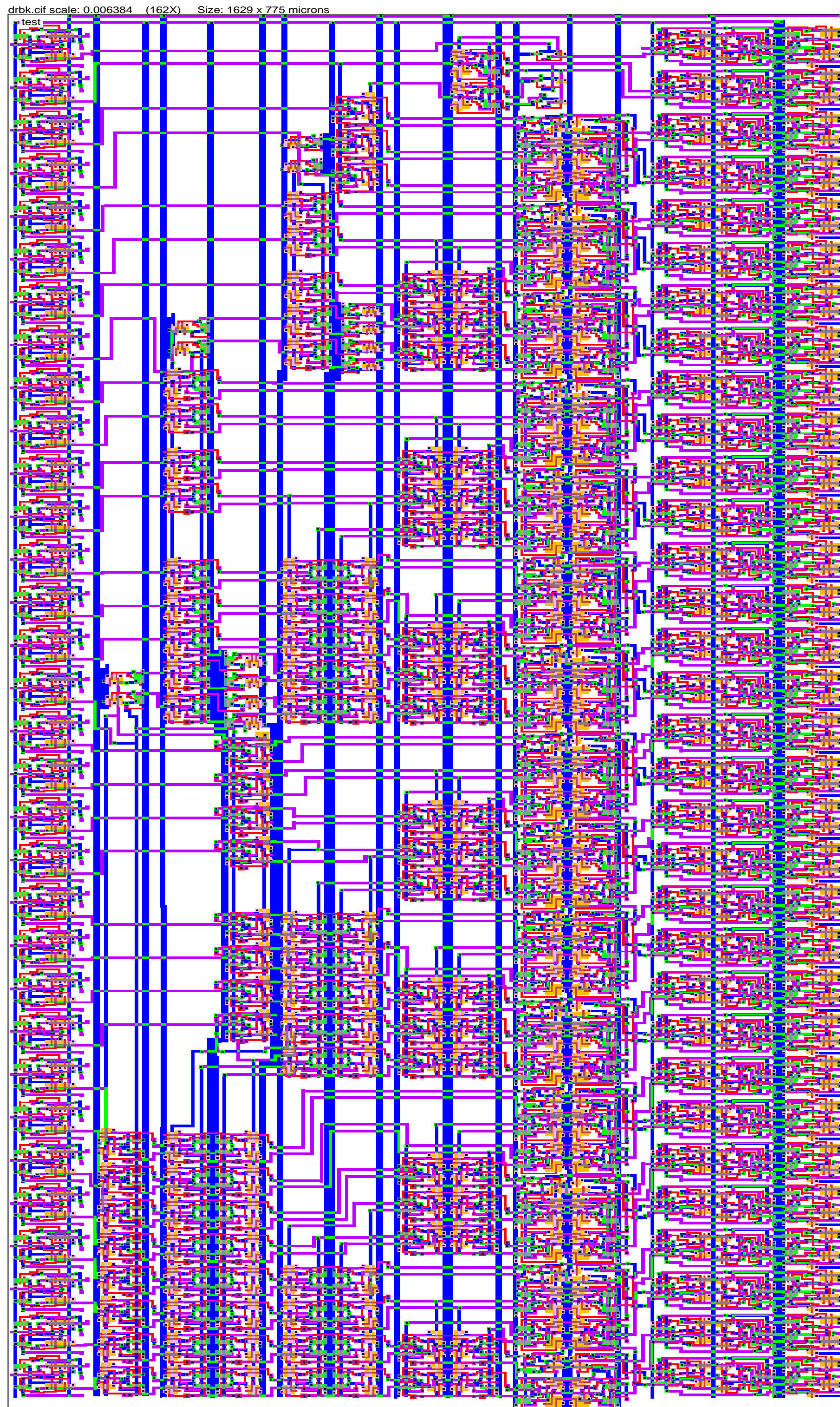


Figure 22: The VLSI layout of the Dual-Rail Brent-Kung carry chain.

Appendix D: Timing Values for Variable Length Carries

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	4																													
3	7	4																												
4	10	7	4																											
5	13	10	7	4																										
6	16	13	10	7	4																									
7	19	16	13	10	7	4																								
8	22	19	16	13	10	7	4																							
9	25	22	19	16	13	10	7	4																						
10	28	25	22	19	16	13	10	7	4																					
11	31	28	25	22	19	16	13	10	7	4																				
12	34	31	28	25	22	19	16	13	10	7	4																			
13	37	34	31	28	25	22	19	16	13	10	7	4																		
14	40	37	34	31	28	25	22	19	16	13	10	7	4																	
15	43	40	37	34	31	28	25	22	19	16	13	10	7	4																
16	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4															
17	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4														
18	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4													
19	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4												
20	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4											
21	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4										
22	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4									
23	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4								
24	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4							
25	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4						
26	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4					
27	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4				
28	82	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4			
29	85	82	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4		
30	88	85	82	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4	
31	91	88	85	82	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4
32	94	91	88	85	82	79	76	73	70	67	64	61	58	55	52	49	46	43	40	37	34	31	28	25	22	19	16	13	10	7	4

Figure 23: The delays of a Basic Ripple carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	5																														
3	6	6																													
4	8	8	7																												
5	8	8	7	6																											
6	8	8	8	8	7																										
7	10	10	10	10	9	7																									
8	10	10	10	10	9	7	6																								
9	10	10	10	10	9	8	8	7																							
10	10	10	10	10	10	10	10	9	7																						
11	12	12	12	12	12	12	12	11	9	7																					
12	12	12	12	12	12	12	12	11	9	7	6																				
13	12	12	12	12	12	12	12	11	9	8	8	7																			
14	12	12	12	12	12	12	12	11	10	10	10	9	7																		
15	12	12	12	12	12	12	12	12	12	12	12	11	9	7																	
16	14	14	14	14	14	14	14	14	14	14	14	13	11	9	7																
17	14	14	14	14	14	14	14	14	14	14	14	13	11	9	7	6															
18	14	14	14	14	14	14	14	14	14	14	14	13	11	9	8	8	7														
19	14	14	14	14	14	14	14	14	14	14	14	13	11	10	10	10	9	7													
20	14	14	14	14	14	14	14	14	14	14	14	13	12	12	12	12	11	9	7												
21	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	13	11	9	7											
22	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	11	9	7										
23	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	11	9	7	6									
24	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	11	9	8	8	7								
25	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	11	10	10	10	9	7							
26	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	12	12	12	12	11	9	7						
27	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	14	14	14	14	14	13	11	9	7					
28	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	15	13	11	9	7				
29	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	17	15	13	11	9	7			
30	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	17	15	13	11	9	7	6		
31	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	17	15	13	11	9	8	8	7	
32	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	17	15	13	11	10	10	10	9	7

Figure 24: The delays of a Carry Select carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	5																														
3	7	6																													
4	9	8	6																												
5	10	10	10	10																											
6	12	12	12	12	6																										
7	14	14	14	14	8	6																									
8	16	16	16	16	10	8	6																								
9	16	16	16	16	13	11	11	11																							
10	16	16	16	16	15	13	13	13	6																						
11	17	17	17	17	17	15	15	15	8	6																					
12	19	19	19	19	19	17	17	17	10	8	6																				
13	21	21	21	21	21	19	19	19	12	10	8	6																			
14	21	21	21	21	21	19	19	19	15	13	11	11	11																		
15	21	21	21	21	21	19	19	19	17	15	13	13	13	6																	
16	21	21	21	21	21	19	19	19	19	17	15	15	15	8	6																
17	22	22	22	22	22	21	21	21	21	19	17	17	17	10	8	6															
18	24	24	24	24	24	23	23	23	23	21	19	19	19	12	10	8	6														
19	26	26	26	26	26	25	25	25	25	23	21	21	21	14	12	10	8	6													
20	28	28	28	28	28	27	27	27	27	25	23	23	23	16	14	12	10	8	6												
21	28	28	28	28	28	27	27	27	27	25	23	23	23	19	17	15	13	11	11	11											
22	28	28	28	28	28	27	27	27	27	25	23	23	23	21	19	17	15	13	13	13	6										
23	28	28	28	28	28	27	27	27	27	25	23	23	23	23	21	19	17	15	15	15	8	6									
24	28	28	28	28	28	27	27	27	27	25	25	25	25	25	23	21	19	17	17	17	10	8	6								
25	28	28	28	28	28	27	27	27	27	27	27	27	27	27	25	23	21	19	19	19	12	10	8	6							
26	28	28	28	28	28	27	27	27	27	27	27	27	27	27	25	23	21	19	19	19	15	13	11	11	11						
27	28	28	28	28	28	27	27	27	27	27	27	27	27	27	25	23	21	19	19	19	17	15	13	13	13	6					
28	28	28	28	28	28	27	27	27	27	27	27	27	27	27	25	23	21	19	19	19	19	17	15	15	15	8	6				
29	28	28	28	28	28	28	28	28	28	28	28	28	28	28	26	24	22	21	21	21	21	19	17	17	17	10	8	6			
30	28	28	28	28	28	28	28	28	28	28	28	28	28	28	26	24	22	21	21	21	18	19	17	17	17	13	11	11	11		
31	28	28	28	28	28	28	28	28	28	28	28	28	28	28	26	24	22	21	21	21	20	19	17	17	17	15	13	13	13	6	
32	28	28	28	28	28	28	28	28	28	28	28	28	28	28	26	24	22	21	21	21	21	19	17	17	17	16	14	14	14	10	10

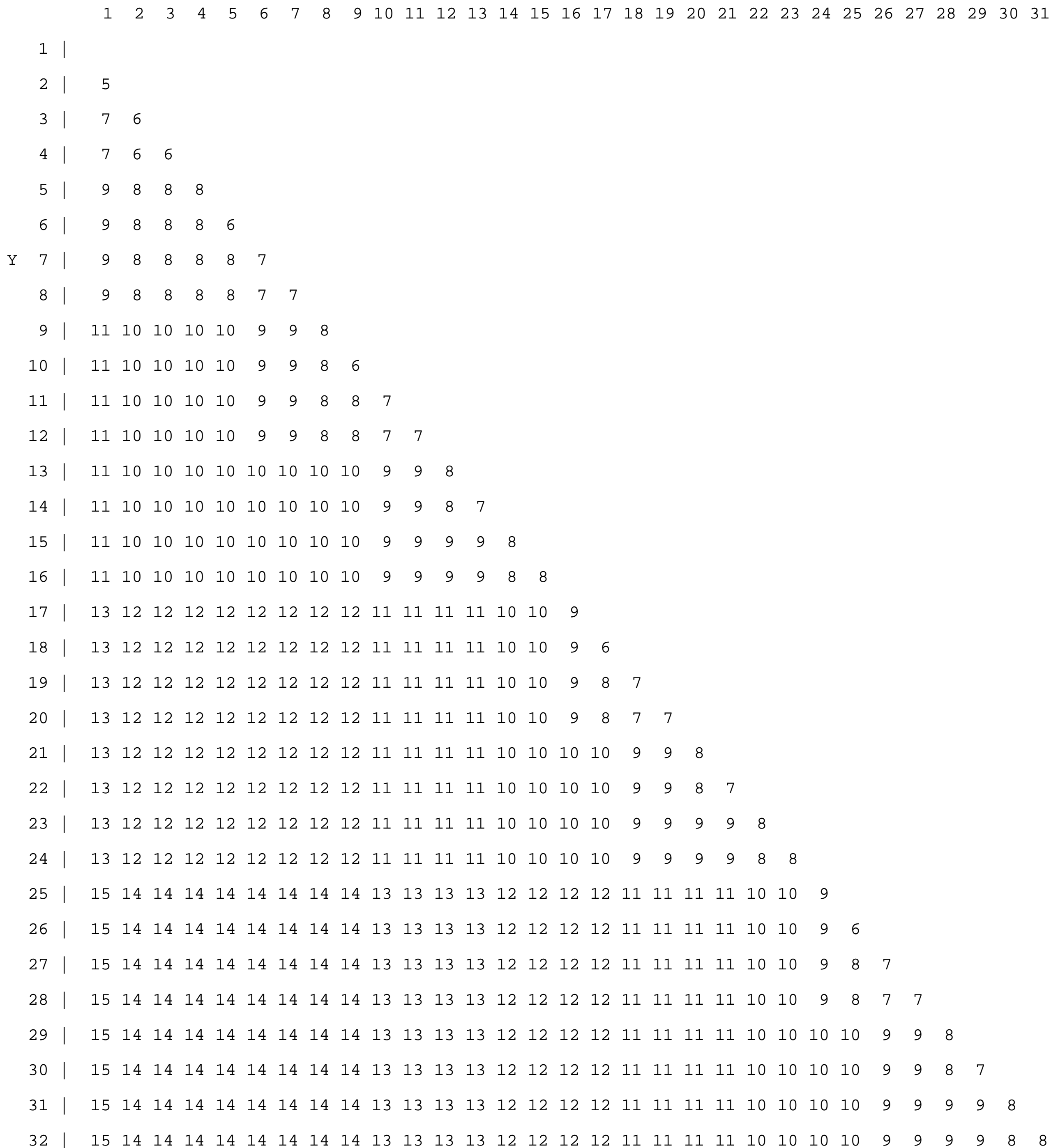
Figure 25: The delays of a Variable Block carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	5																														
3	7	6																													
4	7	6	6																												
5	9	8	8	7																											
6	9	8	8	7	6																										
7	11	10	10	9	8	7																									
8	11	10	10	9	8	7	6																								
9	13	12	12	11	10	9	8	7																							
10	13	12	12	11	10	9	8	7	6																						
11	15	14	14	13	12	11	10	9	8	7																					
12	15	14	14	13	12	11	10	9	8	7	6																				
13	17	16	16	15	14	13	12	11	10	9	8	7																			
14	17	16	16	15	14	12	13	11	10	9	8	7	6																		
15	19	18	18	17	16	15	14	13	12	11	10	9	8	7																	
16	19	18	18	17	16	15	14	13	12	11	10	9	8	7	6																
17	21	20	20	19	18	17	16	15	14	13	12	11	10	9	8	7															
18	21	20	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6														
19	23	22	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7													
20	23	22	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6												
21	25	24	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7											
22	25	24	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6										
23	27	26	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7									
24	27	26	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6								
25	29	28	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7							
26	29	28	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6						
27	31	30	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7					
28	31	30	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6				
29	33	32	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7			
30	33	32	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6		
31	35	34	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	
32	35	34	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6

Figure 26: The delays of a 1-Level Carry Lookahead carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	6																														
3	7	6																													
4	7	6	6																												
5	9	8	8	7																											
6	9	8	8	7	6																										
7	9	8	8	8	8	7																									
8	9	8	8	8	8	7	7																								
9	11	10	10	10	10	9	9	8																							
10	11	10	10	10	10	9	9	8	6																						
11	11	10	10	10	10	9	9	8	8	7																					
12	11	10	10	10	10	9	9	8	8	7	7																				
13	13	12	12	12	12	11	11	10	10	9	9	8																			
14	13	12	12	12	12	11	11	10	10	9	9	8	6																		
15	13	12	12	12	12	11	11	10	10	9	9	8	8	7																	
16	13	12	12	12	12	11	11	10	10	9	9	8	8	7	7																
17	15	14	14	14	14	13	13	12	12	11	11	10	10	9	9	8															
18	15	14	14	14	14	13	13	12	12	11	11	10	10	9	9	8	6														
19	15	14	14	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7													
20	15	14	14	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7	7												
21	17	16	16	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8											
22	17	16	16	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	6										
23	17	16	16	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7									
24	17	16	16	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7	7								
25	19	18	18	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8							
26	19	18	18	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	6						
27	19	18	18	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7					
28	19	18	18	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7	7				
29	21	20	20	20	20	19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8			
30	21	20	20	20	20	19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	6		
31	21	20	20	20	20	19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7	
32	21	20	20	20	20	19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10	10	9	9	8	8	7	7

Figure 27: The delays of a 2-Level Carry Lookahead carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.



Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	5																														
3	7	6																													
4	7	6	6																												
5	9	8	8	8																											
6	9	8	8	8	6																										
7	9	8	8	8	8	7																									
8	9	8	8	8	8	7	7																								
9	11	10	10	10	10	9	9	8																							
10	11	10	10	10	10	9	9	8	6																						
11	11	10	10	10	10	9	9	8	8	7																					
12	11	10	10	10	10	9	9	8	8	7	7																				
13	11	10	10	10	10	10	10	10	10	9	9	8																			
14	11	10	10	10	10	10	10	10	10	9	9	8	7																		
15	11	10	10	10	10	10	10	10	10	9	9	9	9	8																	
16	11	10	10	10	10	10	10	10	10	9	9	9	9	8	8																
17	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9															
18	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	6														
19	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	8	7													
20	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	8	7	7												
21	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	8											
22	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	8	7										
23	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8									
24	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8	8								
25	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	9							
26	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	9	6						
27	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	9	8	7					
28	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	9	8	7	7				
29	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	10	10	9	9	8			
30	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	10	10	9	9	8	7		
31	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8	
32	15	14	14	14	14	14	14	14	14	13	13	13	13	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8	8

Figure 28: The delays of a 3-Level Carry Lookahead carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1																															
2	5																														
3	7	6																													
4	7	6	6																												
5	9	8	8	7																											
6	9	8	8	7	6																										
7	9	8	8	8	8	7																									
8	9	8	8	8	8	7	7																								
9	11	10	10	10	10	9	9	8																							
10	11	10	10	10	10	9	9	8	5																						
11	11	10	10	10	10	9	9	8	8	7																					
12	11	10	10	10	10	9	9	8	8	7	7																				
13	11	10	10	10	10	10	10	10	10	9	9	8																			
14	11	10	10	10	10	10	10	10	10	9	9	8	7																		
15	11	10	10	10	10	10	10	10	10	9	9	9	9	8																	
16	11	10	10	10	10	10	10	10	10	9	9	9	9	8	8																
17	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9															
18	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	6														
19	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	8	7													
20	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	8	7	7												
21	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	8											
22	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	5	7										
23	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8									
24	13	12	12	12	12	12	12	12	12	11	11	11	11	10	10	10	10	9	9	9	9	8	8								
25	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9							
26	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	7						
27	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	9	8					
28	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	10	10	9	9	8	8				
29	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	11	11	11	11	10	10	9			
30	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	11	11	11	11	10	10	9	8		
31	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	11	11	11	11	10	10	10	10	9	
32	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	11	11	11	11	11	11	11	11	10	10	10	10	9	9

Figure 29: The delays of a Brent-Kung carry chain which starts at Cell X and ends at Cell Y using the theoretical delay model.

Y \ X	1	4	8	12	16	20	24	28	32
1	2.4								
2	3.3								
3	3.4								
4	3.4	2.8							
5	3.9	3.6							
6	3.9	3.6							
7	3.9	3.6							
8	4.2	3.8	3.5						
9	4.5	4.2	3.8						
10	4.5	4.2	3.8						
11	4.5	4.2	3.8						
12	4.6	4.2	4.0	3.8					
13	4.6	4.2	4.0	3.9					
14	4.6	4.3	4.3	4.3					
15	4.6	4.3	4.3	4.3					
16	5.0	4.5	4.5	4.5	4.2				
17	5.8	5.5	5.4	5.4	4.8				
18	6.0	5.6	5.5	5.5	5.3				
19	6.0	5.6	5.5	5.5	5.3				
20	6.0	5.6	5.5	5.5	5.3	4.0			
21	6.0	5.6	5.5	5.5	5.3	4.0			
22	6.0	5.6	5.5	5.5	5.3	4.1			
23	6.0	5.6	5.5	5.5	5.3	4.1			
24	6.0	5.6	5.5	5.5	5.3	5.1	4.3		
25	6.0	5.6	5.5	5.5	5.3	5.3	4.7		
26	6.0	5.6	5.6	5.6	5.3	5.3	4.7		
27	6.0	5.6	5.6	5.6	5.3	5.3	4.7		
28	6.0	5.6	5.6	5.6	5.3	5.3	4.7	4.2	
29	6.0	5.6	5.6	5.6	5.3	5.3	4.7	4.6	
30	6.0	5.6	5.6	5.6	5.3	5.3	4.7	4.7	
31	6.0	5.6	5.6	5.6	5.3	5.3	4.7	4.7	
32	6.1	5.7	5.6	5.6	5.3	5.3	4.7	4.7	3.8

Figure 30: The delays in nanoseconds of a Brent-Kung carry chain which starts at Cell X and ends at Cell Y. The delays are generated using Spice on a VLSI layout.