USING CARRY INCREMENT ADDERS TO ENHANCE ENERGY SAVINGS WITH SPANNING-TREE ADDER STRUCTURES

By

KYLE A. PRICE

B.S. Electrical Engineering
Texas A&M Texarkana
Texarkana, Texas
2017

B.S. Mathematics
Texas A&M Texarkana
Texarkana, Texas
2017

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Master of Science
May, 2019

USING CARRY INCREMENT ADDERS TO ENHANCE ENERGY SAVINGS WITH SPANNING-TREE ADDER STRUCTURES

Thesis Approved:

Dr. James E. Stine, Jr.

Thesis Adviser

Dr. Keith Teague

Dr. Weili Zhang

Name: Kyle Price

Date of Degree: May, 2019

Title of Study: USING CARRY INCREMENT ADDERS TO ENHANCE ENERGY SAVINGS WITH SPANNING-TREE ADDER STRUCTURES

Major Field: Electrical Engineering

Abstract: Hybrid adders have provided innovation in the field of digital arithmetic. These designs take the best parts of multiple implementations and improve results in terms of area, delay, or power. This work implements a 64-bit hybrid adder using a spanning tree structure with the carry-increment algorithm. Synthesis results are obtained for 45nm technology and show promising data when compared with an existing hybrid design.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

As technology improves with the passing of time, the field of arithmetic maintains relevance for modern and future advances in processing and computation. At the core of digital arithmetic lies the fundamental operation of addition that enables improvement for datapath blocks [1]. And, implementations of more advanced functions rely on addition. This means that improvements to adder designs provide advancement to other critical areas in processor design. Therefore, the improvement of digital arithmetic design becomes crucial as feature sizes diminish to keep up with technological demands.

Addition of two strings of bits makes up the fundamental operation of any digital circuitry. Counting provides the basis for mathematics, and ,therefore, addition provides the basis for computation. Adders allow circuits to increment signals, change address locations, and perform arithmetic using hardware among other functions. These adders make up the backbone of digital circuitry. They are partially responsible for the rapid advancement of technology in the modern era.

Carry Propagate Addition (CPA) becomes more important as more complex arithmetic algorithms are considered. The adder circuit is utilized in subtraction, multiplication, and division, among other functions. As large implementations gain more utility, more adders are often included in the design. It is, therefore, critical that optimization begins at the most fundamental levels of operation and then improved in higher levels of hierarchy. An emphasis on these bottom layers of hierarchy provides a bottom-up approach for optimization, but allows the engineer freedom to spend more

time in the important top levels of design. Thus, optimization of the adder design provides improvement throughout any arithmetic design. As this design optimization occurs, the designer can focus on more important aspects of specific implementation while avoiding the time-draining tasks of lower level design.

Because adders are near the bottom of design hierarchy, they tend to be affected more by diminishing transistor size than higher level designs. However, these changes propagate as adders are used in the next level of hierarchy. Smaller transistor sizes may change the constraints of digital arithmetic design. For example, some addition algorithms may change due to smaller voltage thresholds and incomplete voltages swings [1]. Other designs may be optimized differently for area, power, or delay due to performance changes. Thus, fundamental designs should be reconsidered and re-optimized as transistor sizes continually decrease in feature size.

Early in the exploration of digital arithmetic, dedicated and novel architectures dominated advancement; however, recent improvements have led to advancements in fusing multiple designs or paradigms to create hybrid blocks [2]. These hybrid designs often take the benefits from two or more designs and seek to offset disadvantages by providing a well-rounded approach that enables solving key issues related to its implementation. Within Very-Large Scale Integration (VLSI) architectures, using hybrid designs can help simplify blocks to make certain blocks more regular and modular [1].

In fast adder designs, as well as general digital design, architectures are optimized for area, power, and delay. Generally, area and delay tend to exhibit a trade-off effect. This means that larger architectures tend to have more capability in decreasing delay due to more complex algorithms. However, area and power tend to be more directly related. This is reasonable as the circuit is more spread out with potentially longer and more connections.

Hybrid designs often combine components of each comprising design in such a way

that multiple variants can be tried and tested quickly. This provides a framework such that effects on area, delay, power, and constraints can be quickly estimated during the design process. Building upon this, a vigilant designer can optimize implementation for a specific desired result. Such modularity introduces regularity and more possibilities for customization in high levels of design flow. One important contribution to this area is the use of hybrid design in forming carry-propagate adders [2].

Carry-lookahead addition [3] revolutionized the digital adder design by increasing the speed of carry calculation by computing carries in parallel. Designers realized that a prohibiting factor in circuitry speed was the time needed for Ripple Carry Adders (RCA) to generate [4]. The carry signal is needed for calculation with more significant bits, therefore speeding this small portion of the design up could easily improve the overall design [5]. By finding ways to create a carry bit quickly, carry-propagate designs are able to calculate both the sum and carry bits of any multi-bit adder quickly. These designs utilize simple logic, but often require more space. The speed of these designs often outweighs the space and power constraints.

This thesis presents an implementation of hybrid adder design that utilizes the concepts of prefix addition and carry increment addition against a proven standard of modern addition called the Spanning Tree Adder [2]. Comparisons are made in area, power, and delay using ARM-based 12SOI GF45nm SOI technologyin order to develop an understanding of the advantages, disadvantages, and constraints of specific design choices.

This thesis is organized as follows: Chapter 2 provides insight on the importance and the background of spanning tree architecture and carry increment addition. Chapter 3 presents the design of a 64-bit hybrid adder that utilizes carry increment addition and a spanning tree architecture. Chapter 4 shares the synthesized results of the design and provides a brief analysis of this design compared to others. Finally, conclusions are presented.

CHAPTER 2

BACKGROUND

Carry-propagate adder design is generally optimized based on delay associated with the sum and carry outputs. Early adopters of digital arithmetic realized that carry-propagate addition is useful, but comes at the cost of large delay [4]. That is, the propagation of the carry signal from one cell to another throughout the entire adder created much of this delay [4].

An early solution to digital addition is called the Ripple Carry Adder (RCA) [4], seen in Figure 2.1. The RCA is formed as a concatenation of bitwise full adder cells. Each carry bit and sum bit are calculated respectively as:

$$\begin{aligned} s_k &= a_k \oplus b_k \oplus c_k \ , \\ c_{k+1} &= a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \ . \end{aligned} \tag{2.1}$$

The delay of each full adder in the RCA can be approximated using gate delay, Δ. Since the carry signal goes through 5 gates in its calculation, it is denoted as 5Δ. The sum bit goes through 6 gates, so it is 6Δ. Ripple Carry Adders are named from the need for carry signals to "ripple" as the output of one cell to the input of the next cell.

As RCAs become larger, the ripple effect grows. Each cell depends on every cell before it to calculate both the sum and carry bits. In modern architectures requiring *32* and *64* bit implementations, large delay in RCAs becomes problematic. Therefore, a need arises to minimize the delay in formulating the carry signal so that each full adder may output both signals more quickly. The answer to this problem was the

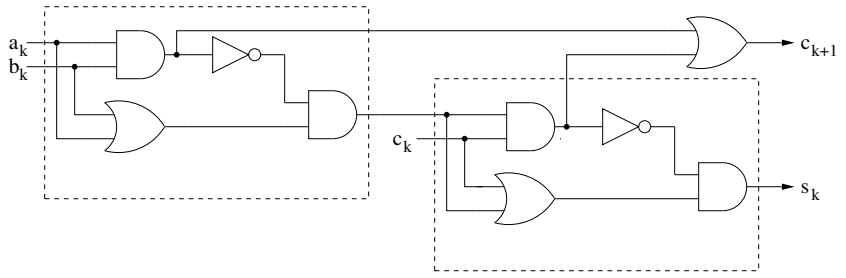

Figure 2.1: Full Adder(Adapted from [6])

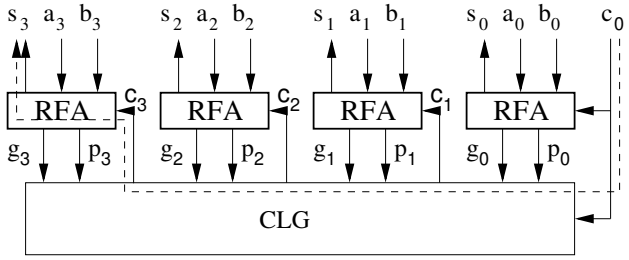

Figure 2.2: Carry Lookahead Adder(Adapted from [6])

Carry Lookahead Adder (CLA) [3].

Carry lookahead algorithms are carry-propagate adders (CPA) that minimize the delay by reducing the amount of time it takes for carry bits to be output by each cell in parallel. Consequently, breaking designs into smaller blocks or utilizing "divide and conquer" algorithms reduces unwanted delay by parallelizing carry structures [4]. For example, turning a *64*-bit adder into *8* connected *8*-bit adders greatly reduces the need for certain carry signals and, thereby, reducing the critical path or worst-case delay. These pieces of the "tree" work less dependently on each other and heavily utilize the ideas of carry lookahead generation and propagation to calculate accurate outputs without producing carry bits for each input bit.

Carry lookahead adders, as shown in Figure 2.2, are a form of carry propagate adders that are able to look ahead at the carry signal faster than the RCA. CLAs use two concepts to speed up carry calculation. The first concept is carry propagation. This determines if a carry is passed from one cell to another. By viewing the full adder in Figure 2.1, it becomes apparent that a carry signal is propagated when either (or

both) of the input signals are '1' and pass a 1 as the output of the AND-gate of the first half adder. The propagate signal only passes the previous carry signal to the next cell, as shown in reffa.fig. This means that if the carry in is '1', the carry out will also be '1'. However, a carry signal can also be generated, even if it does not propagate through the cell. This occurs when both input signals are '1' and pass that as the output of the first AND-gate. The generate and propagate signals are determined by the following Boolean equations [3]:

$$\begin{aligned} g_k &= a_k \cdot b_k \ , \\ p_k &= a_k + b_k \ . \end{aligned} \tag{2.2}$$

Generate and propagate signals ensure that the parts of the tree integrate. Both of these scenarios are noticeable when considering the possibilities of the output, c_{k+1} in the full adder circuit. When combined, these equations can be used to calculate the carry signal more quickly as:

$$c_{k+1} = g_k + p_k \cdot c_k \ . \tag{2.3}$$

Furthermore, generate and propagate signals can be grouped to provide expedient and consolidated calculation as:

$$\begin{aligned} g_{i:j} &= g_{i:k+1} + p_{i:k+1} \cdot g_{k:j} \ , \\ p_{i:j} &= p_{i:k+1} \cdot p_{k:j} \ . \end{aligned} \tag{2.4}$$

Grouped signals determine the generation or propagation of carry signals between any two given intermediate points. This proves useful in various algorithms by eliminating unneeded calculation. Furthermore, these signals can be combined with Equation 2.4 to determine various carry signals. The usage of the combined grouped equations differs according to the carry-propagate algorithm used in each specific implementation. Thus, it is important to understand particular fast adder algorithm choices. Among the more popular fast adder designs are carry-skip adders [7], carry-select adders [8], and carry-increment adders [9].

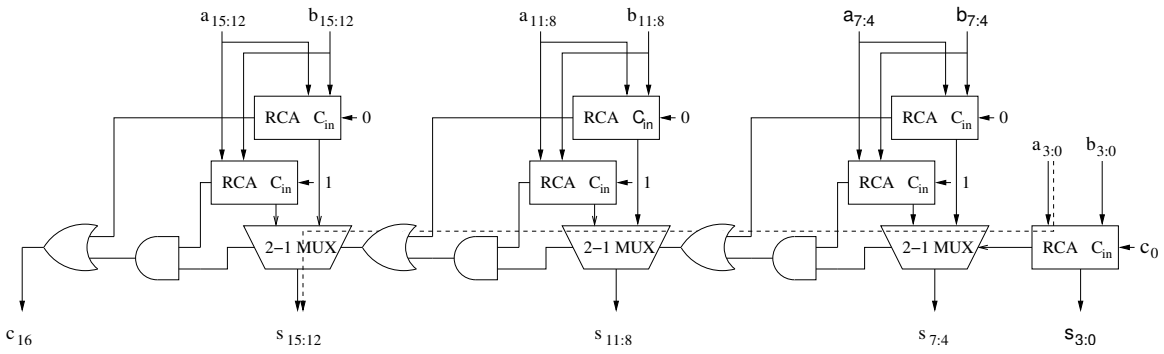

Figure 2.3: 16-Bit Carry-Select Adder(Adapted from [6])

Carry-select adders,as seen in Figure 2.3, prove useful through drastic improvements in speed from the RCA. The algorithm uses multiple RCAs in conjunction to quickly output sum and carry signals. As the least significant RCA calculates the first sums and carries, two RCAs, hard coded with a '0' bit and a '1' bit respectively, calculate both possibilities for the carry out of the first adder. Each of the hard coded adders send the sum outputs to a 2-1 multiplexer. As the carry out signal from the first adder becomes ready, it is used as the select signal for the multiplexer, effectively choosing which of the two equivalent carry-in values is used. Simultaneously, the hard coded adders create the group generate and propagate signals. The adder hard-coded with *0* creates the grouped generate signal while the other creates the grouped propagate signal. Consequently, the grouped generate and propagate signals are used to generate c_8 as:

$$c_8 \quad = \quad g_{7:4} + p_{7:4} \cdot c_4 \quad . \tag{2.5}$$

More bits may be added to the design by concatenating the more segments. This works seamlessly because each block outputs a carry-out signal that is primed to be used as the select signal of the next multiplexer. For large implementations, this leads to sum bits that become ready quickly and only wait on the proper carry-in bits to select which sum can be used. However, each block requires multiple RCAs. This means that area and total power dissipation increase.

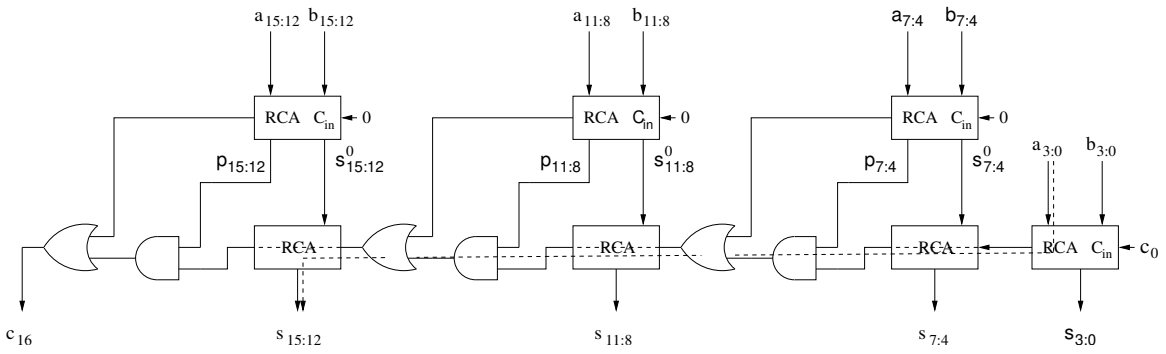

Figure 2.4: 16-Bit Carry-Increment Adder (Adapted from [6])

The carry-increment idea is an algorithmic enhancement of carry-select adders using the ideas from the carry-lookahead concept [9]. Carry-increment adders are often chosen for their moderate boost in speed and considerable improvement in power compared to many other fast adders. A carry-increment adder can be seen in Figure 2.4.

The carry-increment adder incorporates the sum and carry bits within a dual half-adder structure, as shown in Figure 2.5, simplifying the hard-coded 0 or 1 into the carry-in signal [1]. This new unit, called the Dual Half Adder (DHA) [6], generates the sum and carry bits for each position, such that:

$$\begin{aligned} s_k^0 &= a_k \oplus b_k \ , \\ s_k^1 &= \overline{a_k \oplus b_k} = \overline{s_k^0} \ , \\ c_{k+1}^0 &= a_k \cdot b_k \ , \\ c_{k+1}^1 &= a_k + b_k \ . \end{aligned} \tag{2.6}$$

Therefore, the carry-increment algorithm [9] uses the DHA structure so that carry-out or c_{k+1} can be selected between two values inside a multiplexor or mux. That is, the following relationship holds:

$$c_{k+1} = \overline{c_{in}} \cdot c_{k+1}^0 + c_{in} \cdot c_{k+1}^1 \ . \tag{2.7}$$

[1]The superscript denotes the intended hard-coded carry in

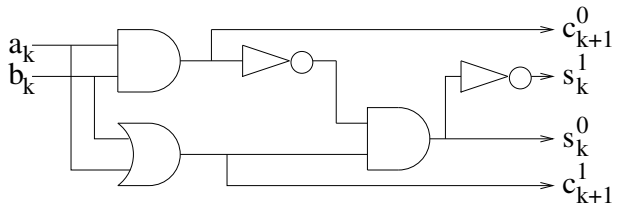

Figure 2.5: Dual Half Adder(Adapted from [6])

Using this carry-out equation and redundant forms of Boolean logic produces simpler carry-select logic that requires less hardware and eliminates the need for the multiplexing as seen in the carry-select algorithm.

$$c_{k+1} \quad = \quad c^0_{k+1} + p_{k:k-r} \cdot c_{in} \quad . \tag{2.8}$$

Each of the fast-adder designs features distinct advantages and disadvantages in their unique usage of the carry lookahead concept. These differences lay the framework for modular usage of various carry lookahead designs in hybrid addition. For this work, carry-increment adders are used to maintain low delay and improved power dissipation and area compared to the carry-select adders commonly used in spanning tree designs.

The carry-increment algorithm is often realized in its 8 bit variation by using multiple smaller, parallelized adders simultaneously. A 4-bit Ripple Carry Adder is used to generate the least significant 4 sum bits as well as a carry out. At the same time, another 4-bit ripple carry with $c_{in} = 0$ is used to create intermediate sum bits for the most significant 4 sum bits of the overall adder.

The algorithm relies on parallelization to quickly output sums while minimizing the critical path. Intermediate sum bits s^0_4, s^0_5, s^0_6, and s^0_7 are all input into consecutive half adders. The first intermediate sum is half added with the carry out bit from earlier. Each sum bit is the accurate final sum for the adder while each carry out bit is input as the carry in for the next half adder.

The carry-out bit is combined with the group propagate signal for the four most-

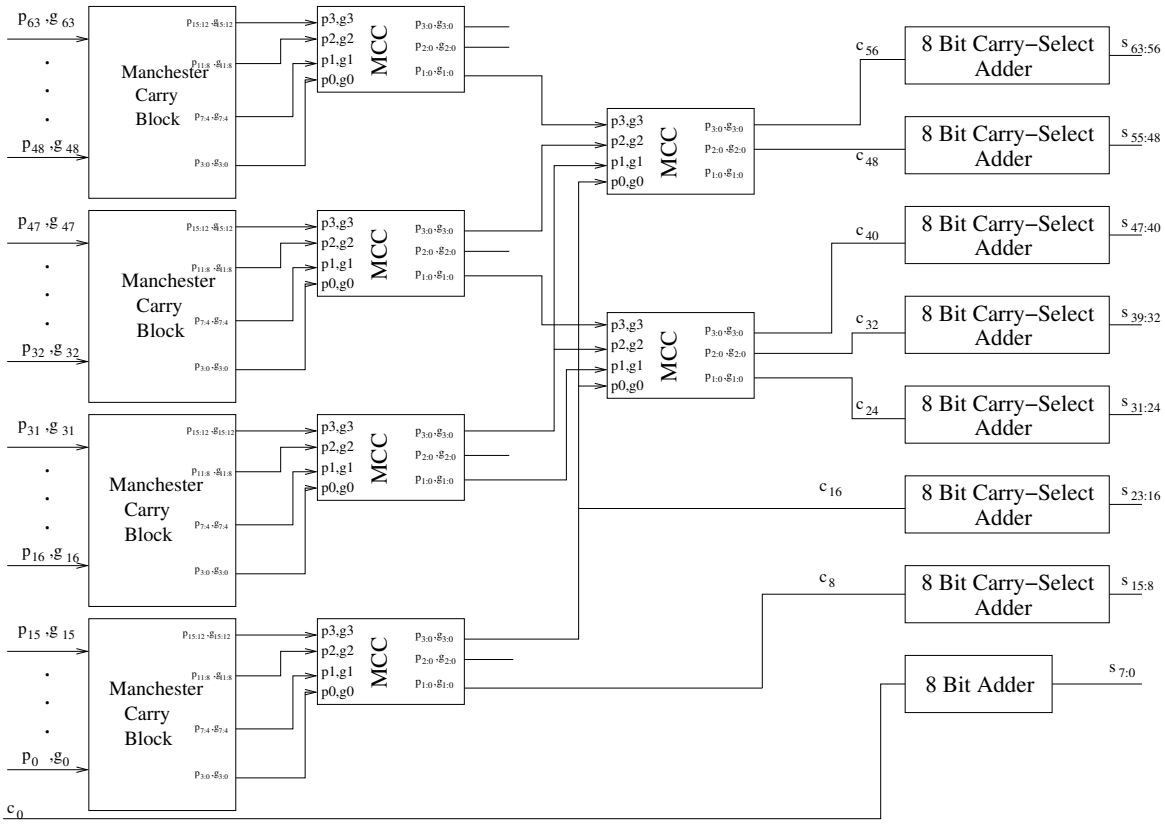

Figure 2.6: 64-Bit Spanning Tree Adder (Adapted from [2])

significant inputs using an AND-gate and then compared using an OR-gate with the carry out from the intermediary ripple-carry adder. This process generates sums quickly and passes the carry for an *8*-bit block more quickly than the standard *8*-bit ripple carry adder, allowing for faster calculation in any further adders. Beacuse the carry out bits are provided already, larger implementations can be achieved by attaching modular circuitry without the need for the initial 4-bit RCA.

The proposed design in this thesis utilizes the carry-increment algorithm; however, the design is brought together topologically using a spanning-tree architecture. An example of a spanning tree architecture can be seen in Figure 2.6 [2]. This hybridization combines the low delay and power of carry-increment with the utility and efficiency of spanning tree. The proposed architecture allows carry generation and propagation to flow unidirectionally. It also works to decrease delay as all inputs are

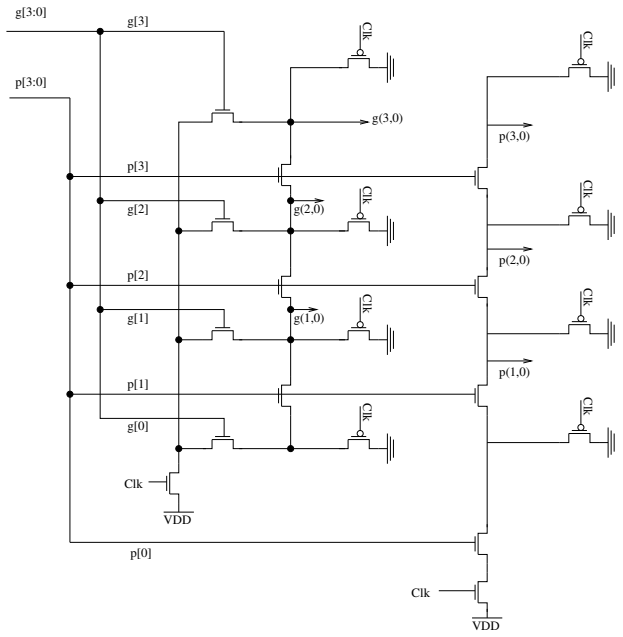

Figure 2.7: Manchester Carry Chain (Adapted from [1])

grouped and consolidated to develop the carry signals needed for each addition. This spares the strain of dealing with many input signals in architecture.

An advantage of spanning tree architectures is the regularity of signal consolidation and reduction. As a design produces carry signals in only one direction, the complexity of design facilitates minimization [2]. Furthermore, these carry signals integrate or "span" the various branches of the tree so that certain aspects of the design maybe be parallelized. These grouped signals can be transformed by combinational logic or Manchester carry chains [1], Figure 2.7, to produce intermediate signals that interact with the carry in bit of each cell as:

$$c_{k+1} \quad = \quad g_{k:0} + p_{k:0} \cdot c_{in} \quad . \tag{2.9}$$

In the Spanning Tree topology, pass-logic circuitry called Manchester carry chains are used to continually group signals so that intermediate inputs may interact with various carry signals to produce a valid carry output. Manchester carry cells are chained together to consolidate signals and develop the final output of the carry lookahead algorithm. These Manchester carry chains make up the backbone of the

spanning tree, as they only allow forward calculation of the generate and propagate signals. As the calculation occurs only forward, previous portions of the circuit do not need to wait on future calculations and may output more quickly with potentially less wiring. The speed provided from Manchester carry chains proves useful in smaller feature sizes.

CHAPTER 3

IMPLEMENTATION

The spanning-tree architecture design connects *8* blocks of *8*-bit adders together using carry bits, c_8, c_{16}, c_{24}, c_{32}, c_{40}, c_{48}, and c_{56} to create a *64*-bit adder. And, the implementation of the *64*-bit adder is subdivided into separate *8*-bit modules. Consequently, the implementation of a *64*-bit spanning tree adder with the carry-increment algorithm is sectioned into identical *8*-bit branches for regularity. Each branch consists of four (4) parallelized *2*-bit carry-increment adders with interdependent carry lookahead chains as shown in Figure 3.1. The spanning bit is calculated with the most-significant portion using Equation 2.9 and passed to the next block.

This design is implemented in two synergistic parts. A modified carry-increment algorithm is utilized to generate intermediate sums while the spanning tree is used to turn inputs a and b into usable grouped signals that can interact with provided carry inputs to create internal carry bits and the carry out bit. As the carry bits are created, the intermediate sum values become ready for incrementation to provide accurate results. Half adders are broken up into *2*-bit intervals to make full usage of the carry signals provided in the Manchester carry portion of the spanning tree and to allow for faster computation of the sum outputs.

Since the spanning tree architecture produces the four (4) most-significant carry bits more quickly than a simple 4-bit ripple carry adder, this hybridization saves valuable time in the carry-increment process. The carry-increment algorithm provides a suitable replacement for the standard carry-select algorithm used in many spanning tree architectures as it remains similar in terms of delay while often improving overall

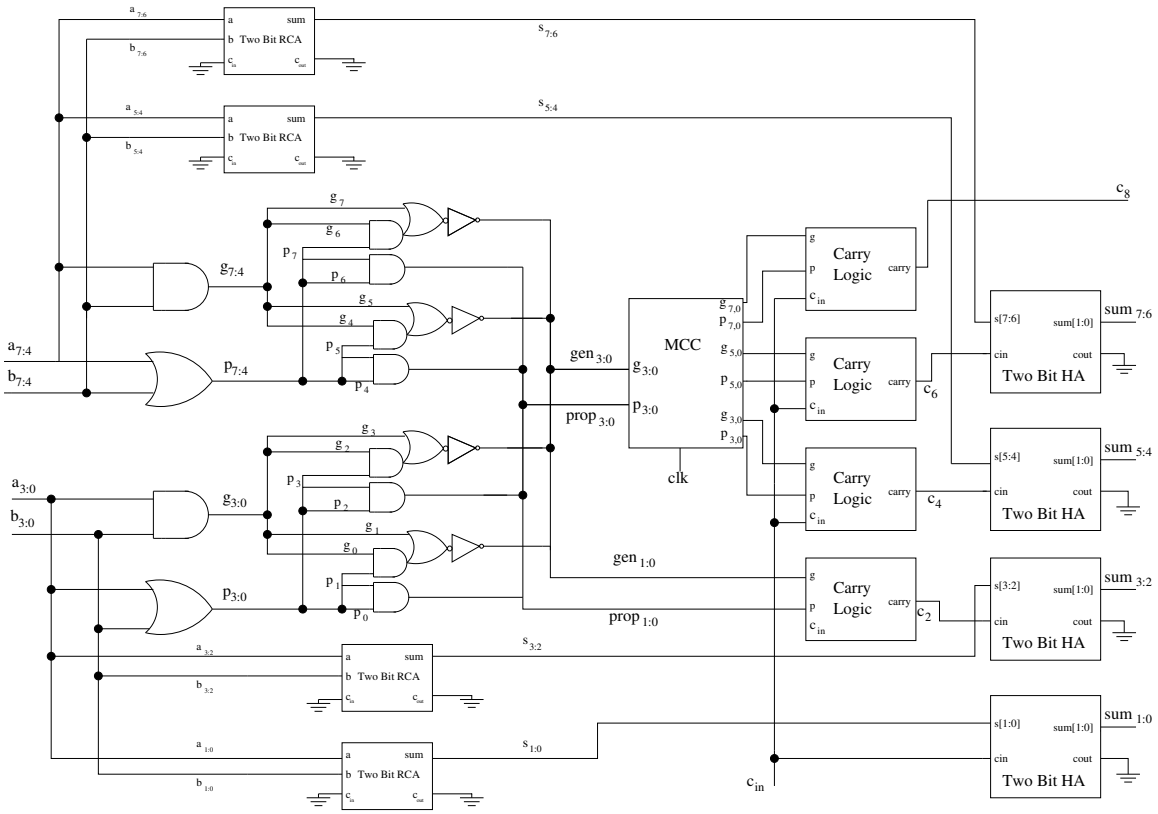

Figure 3.1: 8-Bit Spanning Tree Adder with Carry-Increment Addition

power.

As input signals enter each branch, generate and propagate signals are created using combinational logic to realize Equation 2.2. The g_i and p_i signals are then grouped in two bit intermediate group generate signals, $g_{1:0}$, $g_{3:2}$, $g_{5:4}$, and $g_{7:6}$, and intermediate propagate signals $p_{1:0}$, $p_{3:2}$, $p_{5:4}$ and $p_{7:6}$. These grouped signals are much more manageable and are used to form the carry lookahead portion of the spanning tree concept. However, the intermediate group signals can not yet be used to realize (Equation 2.9) as they simply determine the generation and propagation of a carry respective to the two grouped signals and not from the least significant bit to the current bit. Using the provided signals at this point would not provide accurate outputs. While final grouping could be achieved combinationally, the timing cost would be high and preparation of the intermediate groupings is left to the Manchester

carry chains in order to keep delay low.

The temptation may arise to realize all group signals using dynamic logic as with the original implementation in [2]. While this accomplishes much in terms of speed, care must also be taken to create outputs with full voltage swing. In small technologies, this proves particularly crucial. Smaller threshold voltages lead to smaller margins for error in output voltage swing. Initial signal grouping utilizes static logic while only one Manchester carry chain is placed in each *8*-bit branch to finally consolidate all grouped signals. This balances speed with complete output accuracy.

The exception to the Manchester carry chain is the generation of the internal carry bit, c_2. As the signals $g_{1:0}$ and $p_{1:0}$ are already of the form (Equation 2.9), this carry bit can be computed without the need for Manchester carry chains. The carry bit is then calculated using additional logic as soon as these two grouped signals become ready. However, both signals are still needed for the Manchester carry chain in order to create the fully grouped signals.

Manchester carry chains consolidate intermediate grouped signals to provide usable "full group" signals. The design uses pass logic to perform the functions:

$$\begin{aligned} g_{3,0} &= g_{3:2} + p_{3:2} \cdot g_{1:0} \ , \\ p_{3,0} &= p_{3:2} \cdot p_{1:0} \ , \\ g_{5,0} &= g_{5:4} + p_{5:4} \cdot g_{3:0} \ , \\ p_{5,0} &= p_{5:4} \cdot p_{3:2} \cdot p_{1:0} \ , \\ g_{7,0} &= g_{7:6} + p_{7:6} \cdot g_{5:0} \ , \\ p_{7,0} &= p_{7:6} \cdot p_{5:4} \cdot p_{3:2} \cdot p_{1:0} \ . \end{aligned} \tag{3.1}$$

The Manchester carry design is grouped together in a 4-bit output implementation as larger combinations tend toward instability and diminished voltage swing. The pass logic in the Manchester carry chain can be replaced with additional logic as in the

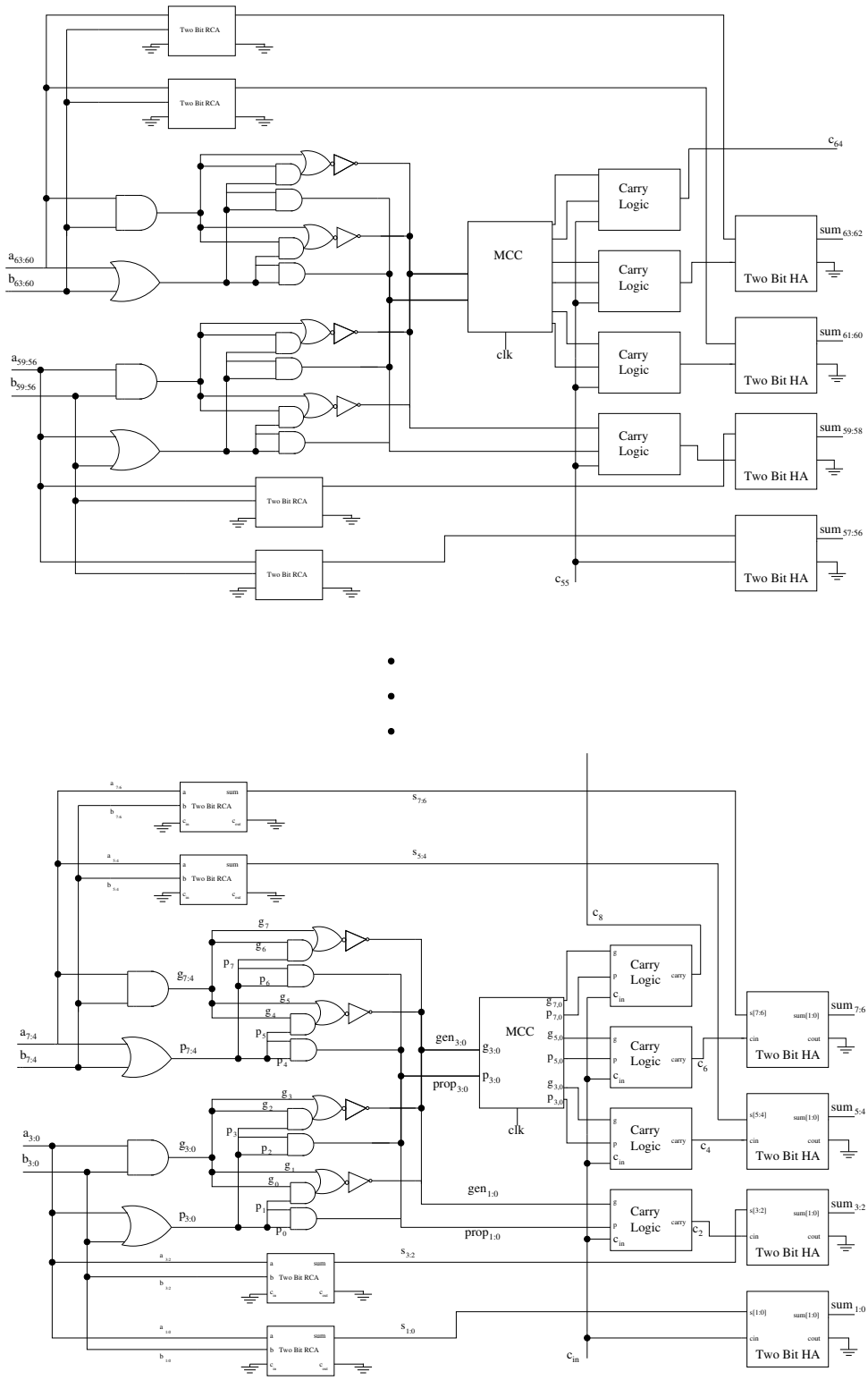

Figure 3.2: 64-Bit Spanning Tree Adder with Carry-Increment Addition

case of the previous grouping circuitry; however, this causes extra delay.

The final step of the carry lookahead stage is the construction of carry bits for use in the second stage of carry increment. Corresponding full group signals are combined with the carry in bit of each *8*-bit module to determine carry values for c_4, c_6, and c_{out} bits. Signals c_4 and c_6 are used in the carry increment algorithm while c_{out} is passed as the c_{in} of the next branch. This stage is where the spanning-tree architecture becomes useful. As each branch relies on the c_{out} of the previous branch, a carry tree is formed in only the forward direction without the need for backward propagation. In addition, each internal carry bit is calculated as:

$$\begin{aligned} c_2 &= g_{1:0} + p_{1:0} \cdot c_{in} \ , \\ c_4 &= g_{3:0} + p_{3:0} \cdot c_{in} \ , \\ c_6 &= g_{5:0} + p_{5:0} \cdot c_{in} \ , \\ c_8 &= g_{7:0} + p_{7:0} \cdot c_{in} \ . \end{aligned} \tag{3.2}$$

The beginning of the carry-increment algorithm occurs in parallel with the carry-lookahead algorithm so that minimal calculation occurs after the carry bits are ready. As a *4*-bit ripple carry adder outputs the sum after carry-lookahead execution, two *2*-bit ripple carry adders are used in parallel instead. These adders generate intermediate sum bits, s_k^0 for use in carry incrementation. The carry-in and carry-out values of each adder are accounted for in the carry-lookahead algorithm. Thus, the carry in of each adder is set to 0 and the carry out is disconnected. Including these would have the effect of doubling the carry out value for each 2-bit adder. This process is used for four 2-bit blocks intermediate outputs simultaneously to represent each 8-bit branch.

Finally, the intermediate sum bits and the prepared carry lookahead outputs undergo incrementation. Each carry signal is paired with corresponding intermediate sum bits and enters a 2-bit half adder. Each of the half adders form the final step of

the carry-increment algorithm.

Sum bits for each signal are generated and the carry outputs are disconnected as any useful carry signals have already been generated and passed to the next branch. This demonstrates the advantage of hybridizing the carry-increment algorithm with spanning tree architecture. The final combinational logic from the carry-increment algortihm is not needed since the spanning tree quickly outputs the carry out for each blocks, thereby diminishing logic levels needed and improving delay. The sum bits from each half-adder are output as the final sum bits for each branch and the complete design. Each sum bit is subsequently found as:

$$s_k \quad = \quad s_k^0 \oplus c_k \quad . \tag{3.3}$$

Additional *8*-bit branches are connected modularly by connecting carry out signals between branches as described above and supplying the corresponding a and b inputs. Connecting carry signals are found as:

$$\begin{aligned}
c_8 &= g_{7:0} + p_{7:0} \cdot c_{in} \quad , \\
c_{16} &= g_{15:8} + p_{15:8} \cdot c_8 \quad , \\
c_{24} &= g_{23:16} + p_{23:16} \cdot c_{16} \quad , \\
c_{32} &= g_{31:24} + p_{31:24} \cdot c_{24} \quad , \\
c_{40} &= g_{39:32} + p_{39:32} \cdot c_{32} \quad , \\
c_{48} &= g_{47:40} + p_{47:40} \cdot c_{40} \quad , \\
c_{56} &= g_{55:48} + p_{55:48} \cdot c_{48} \quad , \\
c_{64} &= g_{63:56} + p_{63:56} \cdot c_{56} \quad .
\end{aligned} \tag{3.4}$$

More 8-bit blocks may be concatenated as needed. A modular 64-bit design can be seen in Figure 3.2. Results and analysis are provided for the 64-bit variation below.

CHAPTER 4

RESULTS AND ANALYSIS

System on Chip (SoC) design flow for the proposed adder design is realized in Figure 4.1. First, Hardware Description Language (HDL) and test vectors are written,and then the logic is verified. Power files were generated from the verification software. The design was synthesized and results were obtained [10]. Each design flow step is detailed below.

To design the proposed architecture, HDL is first implemented, where HDL is a text-based code used to describe and design hardware. HDL is heavily used in both academia and commercially due to its ease in design and simulation [1]. The programmer is able to implement a desired hardware implementation by describing the logic functions of the associated design. This description takes on a code-like structure and the developer can quickly create testable models.

Two commonly used HDLs are Verilog and VHDL. The proposed implementation is created using Verilog. *8*-bit blocks were created as Verilog modules and combined together to form the *64*-bit design. Within each block, logic components are created using submodules that represented the desired logic for each component. Appropriate inputs and outputs are added to the design along with internal signals called "wires". Because the carry out bits from each block spanned between modules, these connecting signals were designated as wires. After compilation, the design logic is verified using ModelSim® software.

After creating the design in HDL, the logic must be tested. Mentor Graphics Corporation (MGC) ModelSim® is a software that receives Verilog files as an input

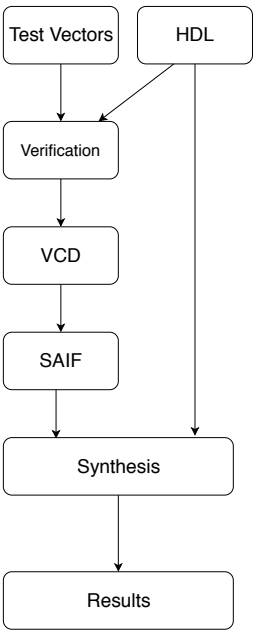

Figure 4.1: VLSI System on Chip (SoC) Design Flow

along with a file containing test values as input for the Verilog design. These test vectors are carefully chosen to show critical inputs and outputs of the design. Critical inputs may be those representing greatest delay or certain internal conditions that must be checked for the design. For the proposed implementation, vectors are chosen to determine proper operation at values for each bit as well as the critical delay condition and proper carry conditions between blocks. Further test vectors are created to ensure that the most significant bits were output correctly after going through the carry lookahead portion of the spanning tree. When the Verilog and test vector files are ready, simulation demonstrates the various outputs across time for the design using the test vectors. The outputs are carefully verified to determine correct operation of the proposed design. During this step, files can be generated that represent power conditions in the design. These files are called Value Changed Dump (VCD) files, and prove useful in determining accurate power or energy readings for the synthesized design.

VCD files determine the quantity of vector changing, and therefore a relatively

accurate power model, that the Verilog design goes through with the test vectors provided. However, this file also includes a large amount of information for large designs or vectors. It can be converted into a consolidated formatted called the Switching Activity Interchange Format file. This file contains the switching information for the design signals along with signal duration. While a layout can be synthesized from only the Verilog code, the power results of the layout tend to be much more accurate when using the SAIF file.

Standard cells are the building blocks of synthesized implementations. While the Verilog model describes the logic of the design, the standard cells are used to create the layout. These cells usually utilize predetermined height and width characteristics to lay the design out in a standard format [1]. A benefit of using standard cells is that they exploit regularity to quickly "place and route" a given design. What could take days or weeks in custom logic design can take minutes in standard cell implementation. The disadvantage of this is that the design may not be completely optimized in the same way a custom design might be. Standard cell supply lines and input/output signals often use the same metal between cells for the sake of regularity. However, standard cells need direction to be placed.

Standard cells can be used alongside HDL files to create layout for a given design. This process is known as synthesis. Synthesis places the needed components and gates as standard cells and routes them together using lines of metal. The placement and routing may be internally optimized for area, delay, or power. Based on this optimization, results can be obtained from the design so that the designer may understand the associated constraints. The implemented design is ultimately optimized for delay. While the results are viewable at this point, some other considerations are also necessary.

The first major consideration for obtaining results is that of the topographical model versus Wire Load Model (WLM). WLM model synthesizes a rudimentary de-

sign based on the concept that the load comes from the wires. This looks at the design from an overall perspective. While WLM remains an option for large technology feature sizes, it overlooks the fine detail needed for small transistor technologies. Topographical mode includes considerations for cell interconnects and looks at the nuanced details of each cell for more realistic results.

Finally, it is important to understand technology sizing options. An admirable goal is to develop the design and obtain results for the smallest technology possible. This is so the design keeps up with modern constraints and the fast pace of industry. The proposed design was first designed in 14nm technology. However, synthesis would only correctly place layout using WLM mode. As these results were inaccurate, the design was resynthesized using a 45nm process in topological mode and compared to a design of the same technology size.

The 64-bit adder design is designed using RTL-level Verilog and synthesized in a Global Foundries 12SOI 45nm technology using an ARM standard-cell library. All designs are synthesized using Synopsys' Design Compiler (DC) topographical mode, optimized for minimum delay. DC topographical mode is desirable over the typical wire load model (WLM) in order to ensure a stronger correlation to the area, power consumption, and delay of a post-layout physical implementation. Even more accuracy is achieved in using topographical mode over WLM when synthesized in smaller technologies due to circuit density compared to wire size. After synthesis, the design is compared to a spanning tree adder design utilizing the carry-select algorithm as in [2]. Results for 45nm area, power, and delay are shown in Table 4.1.

	# Cells	Area [um^2]	Delay [ps]	Power [mW]
Proposed Architecture	1099	1601.43	146.29	0.8445
Carry-Select Spanning Tree	1253	1983.83	135.70	1.0100

Table 4.1: Results for the Proposed 64-Bit Design in 45nm Technology

As carry-increment emphasizes relatively low delay and improved power, the results of Table 4.1 are somewhat expected. The low delay of the proposed design keeps computation fast while reducing the area and dissipated power of the circuit. While the proposed design is slower than the compared design by about 9.4ps (a 7% difference), the total power dissipation of the carry-increment design is improved by 16.4% compared to the carry-select design. Additionally, the proposed design features 154 less cells and an overall area improvement of 19.3%. The delay stays comparable to the other design while area and power greatly improve.

As the design creates internal carry signals for every other bit, more capabilities are added by implementing this design. Pairing this with a decrease in power provides a robust, but low-cost option for 64-bit addition. As carry signals are often needed for other functions, this could prove useful in other implementations

Area and power improvements in 45nm for the proposed design are due to the extra ripple carry adders needed for the carry-select algorithm. As each block of 4 bits needs a `2:1` multiplexer and two 4-bit ripple carry adders, the device area quickly accrues. These adders compute in parallel with both each other and the carry lookahead. The result is accumulated more quickly.

For this work's implementation, much of the calculation can occur in linked half adders, which explains the similar delay without the cost of power dissipation. Much of the critical path for both designs lies within the carry lookahead portions of each branch. Additional area can be attributed to the usage of multiple internal carry bits compared to the smaller amount of capability in the carry-select adder. This area disadvantage may be counteracted by using larger branches in a spanning tree, but come at the cost of delay.

CHAPTER 5

CONCLUSIONS

As device sizes tend to decrease, it becomes more important to look at the building blocks of commonly used designs. General improvements to basic digital arithmetic units can have large impacts when considered cumulatively. Additionally, modular designs such as hybrid adders allow for fast and scalable implementations in quickly changing environments. Modules for these hybrid designs hold differing advantages in terms of desired constraints such as area, delay, and power.

Design choice becomes a matter of advantage in time versus advantage in power. These choices should not be made lightly, however. As adder circuitry is used often in modern processing, these advantages accumulate on a large scale.

It should be noted that highly customized results may be obtained through the usage of differing branches in modular spanning tree designs. The designer may choose to optimize for a particular constraint as in this paper, but a balance may be found by the use of alternation or hybridization. For example, a spanning tree architecture that alternates between blocks of carry-increment algorithm and carry-select algorithm could provide a balanced performance boost in both power and speed respectively.

Results for the proposed design show an encouraging improvement in total power while remaining competitive in speed. As device power becomes more important in smaller technologies, these results prove more useful. Further combining of fast adder algorithms with spanning tree architecture may show improvements for application-specific implementations, but the proposed design provides a baseline for general

performance enhancement.

BIBLIOGRAPHY

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[2] T. Lynch and E. E. Swartzlander, Jr., "A spanning tree carry lookahead adder," *IEEE Transactions on Computers*, vol. 41, pp. 931–939, Aug 1992.

[3] A. Weinberger and J. L. Smith, "A one-microsecond adder using one-megacycle circuitry," *IRE Transactions on Electronic Computers*, vol. EC-5, pp. 65–73, June 1956.

[4] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[5] S. Winograd, "On the time required to perform multiplication," *J. ACM*, vol. 14, pp. 793–802, Oct. 1967.

[6] J. E. Stine, "Basic adder structures and the lookahead concept." ECE4243 Lecture Notes, 2019.

[7] M. Lehman and N. Burla, "Skip techniques for high-speed carry-propagation in binary arithmetic units," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 691–698, Dec 1961.

[8] O. J. Bedrij, "Carry-select adder," *IRE Transactions on Electronic Computers*, vol. EC-11, pp. 340–346, June 1962.

[9] A. Tyagi, "A reduced area scheme for carry-select adders," in *Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 255–258, Sep. 1990.

[10] K. Price and J. E. Stine, "Using carry increment adders to enhance energy savings with spanning-tree adder structures." Submitted to 2019 Midwest Symposium on Circuits and Systems, 2019.

VITA

Kyle Price

Candidate for the Degree of

Master of Science

Thesis: USING CARRY INCREMENT ADDERS TO ENHANCE ENERGY SAVINGS WITH SPANNING-TREE ADDER STRUCTURES

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Alamosa, Colorado, United States of America on June 8, 1995.

Education:

Received the B.S. degree from Texas A&M-Texarkana, Texarkana, Texas, United States of America, 2017, in Electrical Engineering

Received the B.S. degree from Texas A&M-Texarkana, Texarkana, Texas, United States of America, 2017, in Mathematics

Completed the requirements for the degree of Master of Science with a major in Electrical Engineering from Oklahoma State University in May, 2019.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
January 2018 - May 2019