PARALLEL-PREFIX STRUCTURES FOR BINARY AND MODULO $\{2^n - 1, 2^n, 2^n + 1\}$ ADDERS

By

JUN CHEN

Bachelor of Science in Information and Control
Engineering
Shanghai Jiao Tong University
Shanghai, China
2000

Master of Science in Electrical Engineering
Illinois Institute of Technology
Chicago, IL, U.S.A.
2004

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2008

PARALLEL-PREFIX STRUCTURES FOR BINARY AND MODULO $\{2^n - 1, 2^n, 2^n + 1\}$ ADDERS

Dissertation Approved:

Dr. James E. Stine

Dissertation Advisor

Dr. Louis G. Johnson

Dr. Sohum Sohoni

Dr. Hermann G. W. Burchard

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. James E. Stine, for his continuous efforts on helping me with the work through a Ph.D. degree. His enthusiasm in teaching, positive thinking, along with his strong professional strengths, guided me towards the completion of this degree.

My family's support is also greatly appreciated. This work cannot be done without the support from my parents Guangzhao Chen and Shengyuan Chen and sister Yu Chen.

I am very proud of being able to work with Ivan, my fellow at VLSI Computer Architecture Group (VCAG). Thanks also go to Johannes Grad, an alumni of Illinois Institute of Technology, who helped me much on EDA tools.

Finally, I would like to thank all the committee members, Dr. Johnson, Dr. Sohoni, Dr. Burchard and Dr. Stine. Thank you all for everything you have done to help through.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

Introduction

1.1 Motivations

To humans, decimal numbers are easy to comprehend and implement for performing arithmetic. However, in digital systems, such as a microprocessor, DSP (Digital Signal Processor) or ASIC (Application-Specific Integrated Circuit), binary numbers are more pragmatic for a given computation. This occurs because binary values are optimally efficient at representing many values [2].

Binary adders are one of the most essential logic elements within a digital system. In addition, binary adders are also helpful in units other than Arithmetic Logic Units (ALU), such as multipliers, dividers and memory addressing [3]. Therefore, binary addition is essential that any improvement in binary addition can result in a performance boost for any computing system and, hence, help improve the performance of the entire system.

The major problem for binary addition is the carry chain [4]. As the width of the input operand increases, the length of the carry chain increases. Figure 1.1 demonstrates an example of an 8-bit binary add operation and how the carry chain is affected. This example shows that the worst case occurs when the carry travels the longest possible path, from the least significant bit (LSB) to the most significant bit (MSB) [4]. In order to improve the performance of carry-propagate adders, it is possible to accelerate the carry chain, but not eliminate it. Consequently, most digital designers often resort to building faster adders when optimizing a computer architecture, because they tend to set the critical path for most computations.

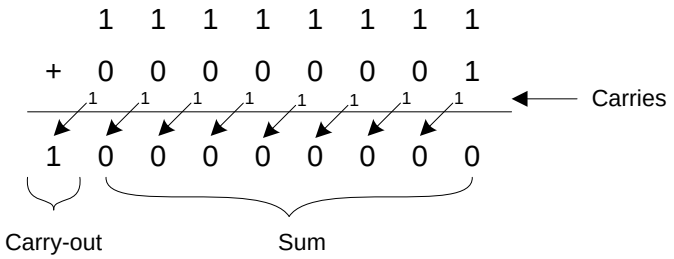


Figure 1.1: Binary Adder Example.

1.2 Carry-Propagate Adders

Binary carry-propagate adders have been extensively published, heavily attacking problems related to carry chain problem. Binary adders evolve from linear adders, which have a delay approximately proportional to the width of the adder, e.g. ripple-carry adder (RCA) [5], to logarithmic-delay adder, such as the carry-lookahead adder (CLA) [6]. There are some additional performance enhancing schemes, including the carry-increment adder [7] and the Ling adder [8] that can further enhance the carry chain, however, in Very Large Scale Integration (VLSI) digital systems, the most efficient way of offering binary addition involves utilizing parallel-prefix trees [9] [10] [11] [12] [13] [14], this occurs because they have the regular structures that exhibit logarithmic delay.

Parallel-prefix adders compute addition in two steps: one to obtain the carry at each bit, with the next to compute the sum bit based on the carry bit. Unfortunately, prefix trees are algorithmically slower than fast logarithmic adders, such as the carry propagate adders, however, their regular structures promote excellent results when compared to traditional CLA adders. This happens within VLSI architectures because a carry-lookahead adder, such as the one implemented in one of Motorola's processors [15], tends to implement the carry chain in the vertical direction instead of a horizontal one, which has a tendency to increase both wire density and fan-in/out dependence. Therefore, although logarithmic adder structures are one of the fastest adders algorithmically, the speed efficiency of the carry-lookahead adder has been hampered by diminishing returns given the fan-in and

fan-out dependencies as well as the heavy wire load distribution in the vertical path. In fact, a traditional carry-lookahead adder implemented in VLSI can actually be slower than traditional linear-based adders, such as the Manchester carry adder [16].

1.3 Algorithmic Architectures for Parallel-Prefix Binary and Modulo $2^n \pm 1$ Adders

Although parallel-prefix structures are discussed in many articles, the process utilized to create them is briefly mentioned or described in most texts. More importantly, there is still a lack of insight how the architecture impacts circuit technology. As technology advances and minimum feature sizes within integrated circuit technology get smaller, there is limited quantitative data for binary adders using nanometer range technologies. Also good circuit design depends on a balance between architecture, circuit and technology choices. If a VLSI designer ignores any one of these design aspects, it can lead to design failures. This work addresses the problem of forming and algorithmically assembling parallel-prefix architectures as well as comparing their performance under nanometer technologies.

This work also extends binary adders to perform modulo addition. Modular arithmetics find applications in a wide range of areas since the operations are the basis for Residue Number Systems (RNS) [17]. Modulo addition/subtraction and multiplication are applied to digital filters [18] in digital signal processing, cryptography [19], error detection and correction [20], as well as checksum computation in high-speed networks [21]. Therefore, utilizing efficient architectures for modulo operations can impact many fields within science and engineering.

Modulo addition, an operation with a small variation to binary addition, can also be applied with prefix architectures [22] [23] [24] [25]. Common modulo addition can even be found in memory addressing. Modulo $2^n - 1$ addition is one of the most common operations that has been put to hardware implementations because of its circuit efficiency [26]. Furthermore, modulo $2^n + 1$ addition is critical to improving advanced cryptography techniques.

As the variations of binary adders, modulo 2^n-1, 2^n and 2^n+1 adders face the same problem as binary adders do: the carry chain dominates any performance seen at the output. There are multiple ways to algorithmically implement modulo adders. The most straightforward way to perform modulo addition is using an end-around addition [27], which is usually inefficient in terms of delay. The more aggressive architectures for modulo addition utilize modified parallel-prefix structures with minimum logic levels [23] [24]. There are also schemes that have a trade-off between logic levels and structural complexity [28].

Ling [8]'s scheme is not only effective on carry-propagating adders, but also can be utilized with parallel-prefix adders. Previously, Ling's scheme has be combined with parallel-prefix adder structures, however, there is limited work done on applying Ling's scheme, especially for modulo prefix adders [29].

According to Moore's law, the number of transistors that can be put on a integrated circuit doubles every two years [30]. More functional parts can be present on a single die, leading engineers with the resource to add new and improved logic to an architecture, such as modulo addition, with dramatic performance improvements. Based on the work presented in this dissertation, it is feasible to incorporate both binary and modulo $2^n \pm 1$ adders in a single architecture. More importantly, this work also describes how to efficiently build modulo adders and combine them with traditional binary carry propagate adders.

1.4 Research Contributions

The implementation that have been developed in this dissertation help to improve the design of parallel-prefix adders and their associated computing architectures. This has the potential of impacting many application specific and general purpose computer architectures. Consequently, this work can impact the designs of many computing systems, as well as impacting many areas of engineers and science. The following list are contributions of this work.

1. Simple algorithms that describe how to build parallel-prefix trees for various operand

sizes;

2. Parallel-prefix structures embedded with carry-save notation;

3. Modulo $2^n \pm 1$ adder architectures that employ traditional prefix and Ling algorithms;

4. A better understanding of the relationship between addition and its algorithmic implementation in nanometer VLSI technology.

1.5 Dissertation Organized

This work is organized as follows. Chapter 2 reviews the basic arithmetic and background information on the implementation of binary adders. Chapter 3 illustrates the methods of building binary adders with prefix structures. This chapter also compares different types of prefix structures algorithmically. Chapter 4 offers the background of doing modulo $2^n \pm 1$ adders and discusses various implementation methods, including parallel-prefix Ling structures. A combined binary and modulo $2^n \pm 1$ adder architecture is proposed in this chapter. Chapter 5 presents results under 3 different technologies and compares the adder performance. Chapter 6 summarizes and concludes the work.

CHAPTER 2

Binary Adder Schemes

Adders are one of the most essential components in digital building blocks, however, the performance of adders become more critical as the technology advances. The problem of addition involves algorithms in Boolean algebra and their respective circuit implementation. Algorithmically, there are linear-delay adders like ripple-carry adders (RCA) [5], which are the most straightforward but slowest. Adders like carry-skip adders (CSKA) [31], carry-select adders (CSEA) [32] and carry-increment adders (CINA) [7] are linear-based adders with optimized carry-chain and improve upon the linear chain within a ripple-carry adder. Carry-lookahead adders (CLA) [6] have logarithmic delay and currently have evolved to parallel-prefix structures. Other schemes, like Ling adders, NAND/NOR adders and carry-save adders can help improve performance as well.

This chapter gives background information on architectures of adder algorithms. In the following sections, the adders are characterized with linear gate model, which is a rough estimation of the complexity of real implementation. Although this evaluation method can be misleading for VLSI implementers, such type of estimation can provide sufficient insight to understand the design trade-offs for adder algorithms.

2.1 Binary Adder Notations and Operations

As mentioned previously, adders in VLSI digital systems use binary notation. In that case, add is done bit by bit using Boolean equations. Consider a simple binary add with two n-bit inputs A, B and a one-bit carry-in c_{in} along with n-bit output S.

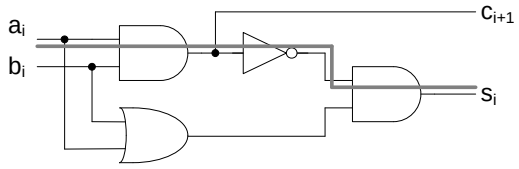


Figure 2.1: 1-bit Half Adder.

$$S = A + B + c_{in}. \tag{2.1}$$

where $A = a_{n-1}, a_{n-2}...a_0, B = b_{n-1}, b_{n-2}...b_0$.

The $+$ in the above equation is the regular add operation. However, in the binary world, only Boolean algebra works. For add related operations, AND, OR and Exclusive-OR (XOR) are required. In the following documentation, a dot between two variables (each with single bit), e.g. $a \cdot b$ denotes 'a AND b'. Similarly, $a + b$ denotes 'a OR b' and $a \oplus b$ denotes 'a XOR b'.

Considering the situation of adding two bits, the sum s and carry c can be expressed using Boolean operations mentioned above.

$$\begin{aligned} s_i &= a_i \oplus b_i \\ c_{i+1} &= a_i \cdot b_i \end{aligned} \tag{2.2}$$

The Equation (2.2) can be implemented as shown in Figure 2.1. In the figure, there is a half adder, which takes only 2 input bits. The solid line highlights the critical path, which indicates the longest path from the input to the output.

Equation (2.2) can be extended to perform full add operation, where there is a carry input.

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_i \\ c_{i+1} &= a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \end{aligned} \tag{2.3}$$

A full adder can be built based on Equation (2.3). The block diagram of a 1-bit full adder is shown in Figure 2.2. The full adder is composed of 2 half adders and an OR gate for

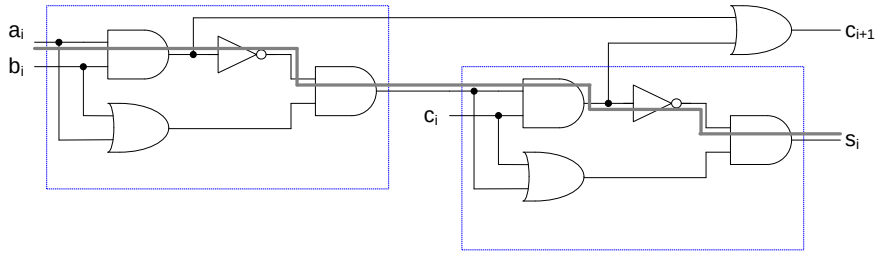


Figure 2.2: 1-bit Full Adder.

computing carry-out. The gates in Figure 2.2 do not implement directly Equation 2.3. Using Boolean algebra, the equivalence can be easily proven.

To help the computation of the carry for each bit, two binary literals are introduced. They are called carry generate and carry propagate, denoted by g_i and p_i. Another literal called temporary sum t_i is employed as well. There is relation between the inputs and these literals.

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \\ t_i &= a_i \oplus b_i \end{aligned} \tag{2.4}$$

where i is an integer and $0 \leq i < n$.

With the help of the literals above, output carry and sum at each bit can be written as

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ s_i &= t_i \oplus c_i \end{aligned} \tag{2.5}$$

In some literatures, carry-propagate p_i can be replaced with temporary sum t_i in order to save the number of logic gates. Here these two terms are separated in order to clarify the concepts. For example, for Ling adders, only p_i is used as carry-propagate.

The single bit carry generate/propagate can be extended to group version G and P. The

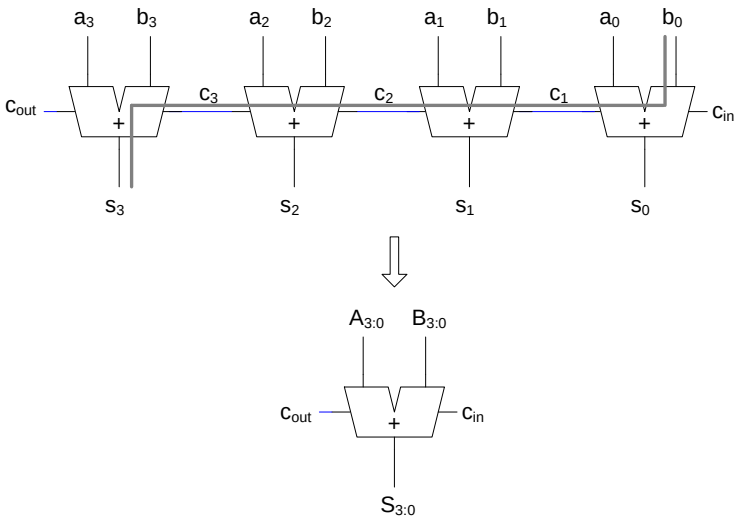


Figure 2.3: Ripple-Carry Adder.

following equations show the inherent relations.

$$\begin{aligned} G_{i:k} &= G_{i:j} + P_{i:j} \cdot G_{j-1:k} \\ P_{i:k} &= P_{i:j} \cdot P_{j-1:k} \end{aligned} \tag{2.6}$$

where $i : k$ denotes the group term from i through k. Using group carry generate/propagate, carry can be expressed as expressed in the following equation.

$$c_{i+1} = G_{i:j} + P_{i:j} \cdot c_j \tag{2.7}$$

2.2 Ripple-Carry Adders (RCA)

The simplest way of doing binary addition is to connect the carry-out from the previous bit to the next bit's carry-in. Each bit takes carry-in as one of the inputs and outputs sum and carry-out bit and hence the name ripple-carry adder. This type of adders is built by cascading 1-bit full adders. A 4-bit ripple-carry adder is shown in Figure 2.3. Each trapezoidal symbol represents a single-bit full adder. At the top of the figure, the carry is rippled through the adder from c_{in} to c_{out}.

It can be observed in Figure 2.3 that the critical path, highlighted with a solid line, is from the least significant bit (LSB) of the input (a_0 or b_0) to the most significant bit (MSB) of sum (s_{n-1}). Assuming each simple gate, including AND, OR and XOR gate has a delay of 2Δ and NOT gate has a delay of 1Δ. All the gates have an area of 1 unit. Using this analysis and assuming that each add block is built with a 9-gate full adder, the critical path is calculated as follows.

$$\begin{aligned} a_i, b_i \rightarrow s_i &= 10\Delta \\ a_i, b_i \rightarrow c_{i+1} &= 9\Delta \\ c_i \rightarrow s_i &= 5\Delta \\ c_i \rightarrow c_{i+1} &= 4\Delta \end{aligned} \tag{2.8}$$

The critical path, or the worst delay is

$$\begin{aligned} t_{rca} &= \{9 + (n-2) \times 4 + 5\}\Delta \\ &= \{4n + 6\}\Delta \end{aligned} \tag{2.9}$$

As each bit takes 9 gates, the area is simply $9n$ for a n-bit RCA.

2.3 Carry-Select Adders (CSEA)

Simple adders, like ripple-carry adders, are slow since the carry has to to travel through every full adder block. There is a way to improve the speed by duplicating the hardware due to the fact that the carry can only be either 0 or 1. The method is based on the conditional sum adder [33] and extended to a carry-select adder [32]. With two RCA, each computing the case of the one polarity of the carry-in, the sum can be obtained with a 2–1 multiplexer with the carry-in as the select signal. An example of 16-bit carry-select adder is shown in Figure 2.4. In the figure, the adder is grouped into four 4-bit blocks. The 1-bit multiplexors for sum selection can be implemented as Figure 2.5 shows. Assuming the two carry terms

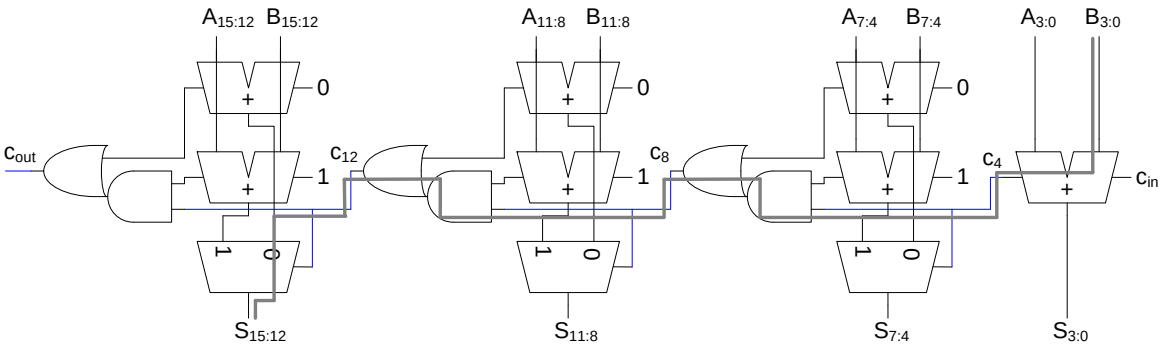


Figure 2.4: Carry-Select Adder.

are utilized such that the carry input is given as a constant 1 or 0:

$$\begin{aligned} c_k^0 &\triangleq c_k|_{c_j=0} \\ c_k^1 &\triangleq c_k|_{c_j=1} \end{aligned} \tag{2.10}$$

In Figure 2.4, each two adjacent 4-bit blocks utilizes a carry relationship

$$c_{i+4} = c_{i+4}^0 + c_{i+4}^1 \cdot c_i. \tag{2.11}$$

The relationship can be verified with properties of the group carry generate/propagate in Equation (2.7) and c_{i+4}^0 can be written as

$$\begin{aligned} c_{i+4}^0 &= G_{i+4:i} + P_{i+4:i} \cdot 0 \\ &= G_{i+4:i}. \end{aligned} \tag{2.12}$$

Similarly, c_{i+4}^1 can be written as

$$\begin{aligned} c_{i+4}^1 &= G_{i+4:i} + P_{i+4:i} \cdot 1 \\ &= G_{i+4:i} + P_{i+4:i}. \end{aligned} \tag{2.13}$$

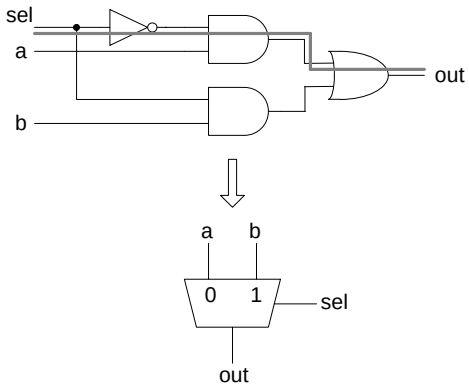


Figure 2.5: 2-1 Multiplexor.

Then

$$\begin{aligned} c^0_{i+4} + c^1_{i+4} \cdot c_i &= G_{i+4:i} + (G_{i+4:i} + P_{i+4:i}) \cdot c_i \\ &= G_{i+4:i} + G_{i+4:i} \cdot c_i + P_{i+4:i} \cdot c_i \\ &= G_{i+4:i} + P_{i+4:i} \cdot c_i \\ &= c_{i+4} \end{aligned} \tag{2.14}$$

Varying the number of bits in each group can work as well for carry-select adders. Similar to (2.10), temporary sums can be defined as follows.

$$\begin{aligned} s^0_{i+1} &= t_{i+1} \cdot c^0_i \\ s^1_{i+1} &= t_{i+1} \cdot c^1_i. \end{aligned} \tag{2.15}$$

The final sum is selected by carry-in between the temporary sums already calculated.

$$s_{i+1} = \overline{c_j} \cdot s^0_{i+1} + c_j \cdot s^1_{i+1} \tag{2.16}$$

Assuming the block size is fixed at r-bit, the n-bit adder is composed of k groups of r-bit blocks, i.e. $n = r \times k$. The critical path with the first RCA has a delay of $(4r + 5)\Delta$ from the input to the carry-out, and there are $k - 2$ blocks that follow, each with a delay of 4Δ for carry to go through. The final delay comes from the multiplexor, which has a delay

of 5Δ, as indicated in Figure 2.5. The total delay for this CSEA is calculated as

$$\begin{aligned} t_{csea} &= 4r + 5 + 4(k-2) + 5\Delta \\ &= \{4r + 4k + 2\}\Delta. \end{aligned} \tag{2.17}$$

The area can be estimated with $(2n - r)$ FAs, $(n - r)$ multiplexors and $(k - 1)$ AND/OR logic. As mentioned above, each FA has an area of 9 and a multiplexor takes 5 units of area. The total area can be estimated

$$9(2n - r) + 2(k - 1) + 4(n - r) = 22n - 13r + 2k - 2 \tag{2.18}$$

The delay of the critical path in CSEA is reduced at the cost of increased area. For example, in Figure 2.4, $k = 4$, $r = 4$ and $n = 16$. The delay for the CSEA is 34Δ compared to 70Δ for 16-bit RCA. The area for the CSEA is 310 units while the RCA has an area of 144 units. The delay of the CSEA is about the half of the RCA. But the CSEA has an area more than twice that of the RCA. Each adder can also be modified to have a variable block sizes, which gives better delay and slightly less area [34].

2.4 Carry-Increment Adders (CINA)

The delay of carry-select adders is reduced by duplicating the carry and sum part and hence, the area increase is significant. Tyagi [7] proposed a scheme that can have similar delay of carry-select adders without much increase in area. This type of adder is called a carry-increment adder. Zimmermann [35] expanded the idea to have multi-level structures.

The area can be reduced in that the carries defined in Equation (2.10) has the following relationship.

$$\begin{aligned} c_i^1 &= G_{i-1:j} + P_{i-1:j} \cdot c_j|_{c_j=1} \\ &= G_{i-1:j} + P_{i-1:j} \end{aligned} \tag{2.19}$$

These requires that c_i^1 is the carry when carry-in $c_j = 1$ and c_i^0 is the carry when $c_j = 0$. The superscript of "1" and "0" can be switched in the definition, however, the implementation

is not affected. Conversely, the carry-out equation assuming a carry-in of 0 can be written as

$$\begin{aligned} c_i^0 &= G_{i-1:j} + P_{i-1:j} \cdot c_j|_{c_j=0} \\ &= G_{i-1:j}. \end{aligned} \tag{2.20}$$

Inserting Equation (2.20) into Equation (2.19), the following relationship is established, which is the central equation for carry-increment adders.

$$c_i^1 = c_i^0 + P_{i-1:j} \tag{2.21}$$

The carry-out c_i can be also expressed with c_i^0 and c_i^1.

$$c_i = \overline{c_j} \cdot c_i^0 + c_j \cdot c_i^1 \tag{2.22}$$

Inserting Equation (2.21) into Equation (2.22), c_i can finally be rewritten as

$$\begin{aligned} c_i &= \overline{c_j} \cdot c_i^0 + c_j \cdot (c_i^0 + P_{i-1:j}) \\ &= c_i^0 + P_{i-1:j} \cdot c_j. \end{aligned} \tag{2.23}$$

As Equation (2.20) indicates, $c_i^0 = G_{i-1:j}$, the above equation agrees with Equation (2.7). A 16-bit carry-increment adder example is shown in Figure 2.6. This is a 1-level structure with fixed 4-bit block size. Incrementers, instead of full adders are used to compute the sum (i.e. the temporary sum will be incremented either by 0 or by 1). Variable block size and multi-leveling can help improve the performance of CINA [36].

The critical path is similar to that of CSEA. The difference is that the incrementers replace the multiplexors in CSEA. The incrementor can be implemented using HAs as shown in Figure 2.1. An r-bit RCA using HAs has a delay of $\{2(r-1)+3\}\Delta = \{2r+1\}\Delta$. From Equation (2.23), the carry computation is through an OR after AND, which has a delay of 4Δ. The first RCA has a delay of $(4r+5)\Delta$ from the input to the carry-out. There are $k-2$ carry generation blocks that follow, each has a delay of 4Δ.

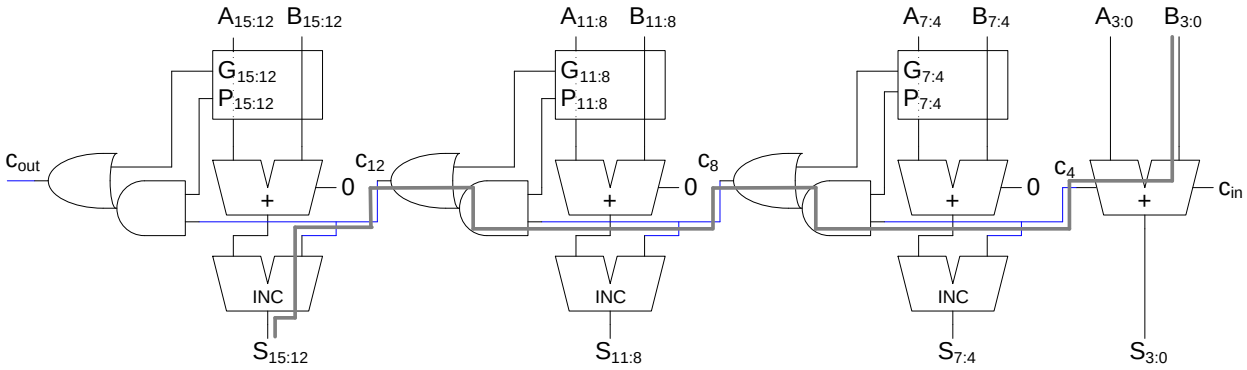


Figure 2.6: Carry-Increment Adder.

$$\begin{aligned} t_{cina} &= \{4r + 5 + 4(k-2) + 2r + 1\}\Delta \\ &= \{6r + 4k - 2\}\Delta \end{aligned} \tag{2.24}$$

The group propagate/generate is not in the critical path and does not contribute to the delay estimation. For an r-bit group propagate/generate, it takes $2r$ gates for generating single bit propagate/generate. For group propagate, one AND gate will suffice and r gates will be required for group generate. An r-bit group propagate/generate requires a total of $3r+1$ gates. The area is estimated with n-bit RCAs, $k-1$ groups of propagate/generate logic and carry generation blocks along with incrementers. Hence, the total area is estimated as

$$9n + (3r+1)(k-1) + 2(k-1) + 4r(k-1) = 16n + 3k - 7r - 3 \tag{2.25}$$

The delay of a CINA is slightly larger than a CSEA but the area is obviously reduced. The block size and the number of blocks can be varied to implement CINAs. Multi-leveling carry architectures can further improve the performance of this adder [35].

2.5 Carry-Skip Adders (CSKA)

There is an alternative way of reducing the delay in the carry-chain of a RCA by checking if a carry will propagate through to the next block. This is called carry-skip adders [31].

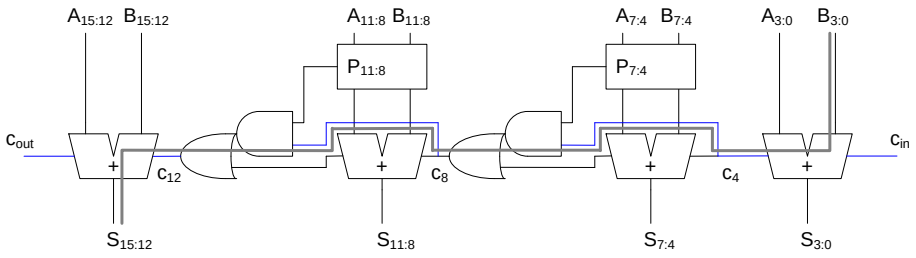


Figure 2.7: Carry-Skip Adder.

The idea of this adder is to modify the Equation (2.7), as shown in Equation (2.26).

$$c_{i+1} = \overline{P_{i:j}} \cdot G_{i:j} + P_{i:j} \cdot c_j \tag{2.26}$$

Figure 2.7 shows an example of 16-bit carry-skip adder.

The carry-out of each block is determined by selecting the carry-in and $G_{i:j}$ using $P_{i:j}$. When $P_{i:j} = 1$, the carry-in c_j is allowed to get through the block immediately. Otherwise, the carry-out is determined by $G_{i:j}$. The CSKA has less delay in the carry-chain with only a little additional extra logic. Further improvement can be achieved generally by making the central block sizes larger and the two-end block sizes smaller [37].

Assuming the n-bit adder is divided evenly to k r-bit blocks, part of the critical path is from the LSB input through the MSB output of the final RCA. The first delay is from the LSB input to carry-out, which is $4r + 5$. Then, there are $k - 2$ skip logic blocks with a delay of 3Δ. Each skip logic block includes one 4-input AND gate for getting $P_{i+3:i}$ and one AND/OR logic. The final RCA has a delay from input to sum at MSB, which is $4r + 6$. The total delay is calculated as follows.

$$\begin{aligned} t_{cska} &= \{4r + 5 + 3(k - 2) + 4r + 6\}\Delta \\ &= \{8r + 3k + 5\}\Delta \end{aligned} \tag{2.27}$$

The CSKA has n-bit FA and $k - 2$ skip logic blocks. Each skip logic block has an area of 3 units. Therefore, the total area is estimated as

$$9n + 3(k - 2) = 9n + 3k - 6 \tag{2.28}$$

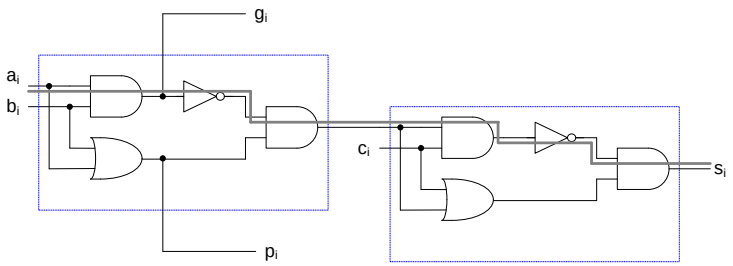


Figure 2.8: Reduced Full Adder.

2.6 Carry-Lookahead Adders (CLA)

The carry-chain can also be accelerated with carry generate/propagate logic, as in Equation (2.7). Carry-lookahead adders employ the carry generate/propagate in groups to generate carry for the next block [6]. In other words, digital logic is used to calculate all the carries at once. When building a CLA, a reduced version of full adder, which is called a reduced full adder (RFA) is utilized. Figure 2.8 shows the block diagram for an RFA. The carry generate/propagate signals g_i/p_i feed to carry-lookahead generator (CLG) for carry inputs to RFA.

The theory of the CLA is based on Equation (2.6) and (2.7). Figure 2.9 shows an example of 16-bit carry-lookahead adder. In the figure, each block is fixed at 4-bit. BCLG stands for Block Carry Lookahead Carry Generator, which generates generate/propagate signals in group form [6]. For the 4-bit BCLG, the following equations are created.

$$\begin{aligned} G_{i+3:i} &= g_{i+3} + p_{i+3} \cdot g_{i+2} + p_{i+3} \cdot p_{i+2} \cdot g_{i+1} + p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot g_i \\ P_{i+3:i} &= p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot p_i \end{aligned} \tag{2.29}$$

The group generate takes a delay of 4Δ, which is an OR after an AND, therefore, the carry-out can be computed, as follows.

$$c_{i+3} = G_{i+3:i} + P_{i+3:i} \cdot c_i \tag{2.30}$$

The carry computation also has a delay of 4Δ, which is an OR after an AND. The 4-bit BCLG has an area of 14 units [38].

The critical path of the 16-bit CLA can be observed from the input operand through 1 RFA, then 3 BCLG and through the final RFA. That is, the critical path shown in Figure 2.9 is from a_0/b_0 to s_7. The delay will be the same for a_0/b_0 to s_{11} or s_{15}, however, the critical path traverses logarithmically, based on the group size. The delays are listed below.

$$\begin{aligned} a_0, b_0 \rightarrow p_0, g_0 &= 2\Delta \\ p_0, g_0 \rightarrow G_{3:0} &= 4\Delta \\ G_{3:0} \rightarrow c_4 &= 4\Delta \\ c_4 \rightarrow c_7 &= 4\Delta \\ c_7 \rightarrow s_7 &= 5\Delta \\ a_0, b_0 \rightarrow s_7 &= 19\Delta \end{aligned} \tag{2.31}$$

The 16-bit CLA is composed of 16 RFAs and 5 BCLGs, which amounts to an area of $16 \times 8 + 5 \times 14 = 198$ units.

Extending the calculation above, the general estimation for delay and area can be derived. Assume the CLA has n-bits, which is divided into k groups of r-bit blocks. It requires $\lceil log_r n \rceil$ logic levels. The critical path starts from the input to p_0/g_0 generation, BLCG logic and the carry-in to sum at MSB. The generation of (p, g) takes a delay of 2Δ. The group version of (p, g) generated by the BCLG has a delay of 4Δ. From next BCLG, there is a 4Δ delay from the CLG generation and 4Δ from the BCLG generation to the next level, which totals to 8Δ. Finally, from c_{k+r} to s_{k+r}, there is a delay of 5Δ. Thus, the total delay is calculated as follows.

$$\begin{aligned} t_{cla} &= \{2 + 8(\lceil log_r n \rceil - 1) + 4 + 5\}\Delta \\ &= \{3 + 8\lceil log_r n \rceil\}\Delta \end{aligned} \tag{2.32}$$

An n-bit CLA requires n RFAs and a logarithmical number of BCLGs assuming each BCLG has the same block size

$$\sum_{l=1}^{log_r n} \frac{n}{r^l} \tag{2.33}$$

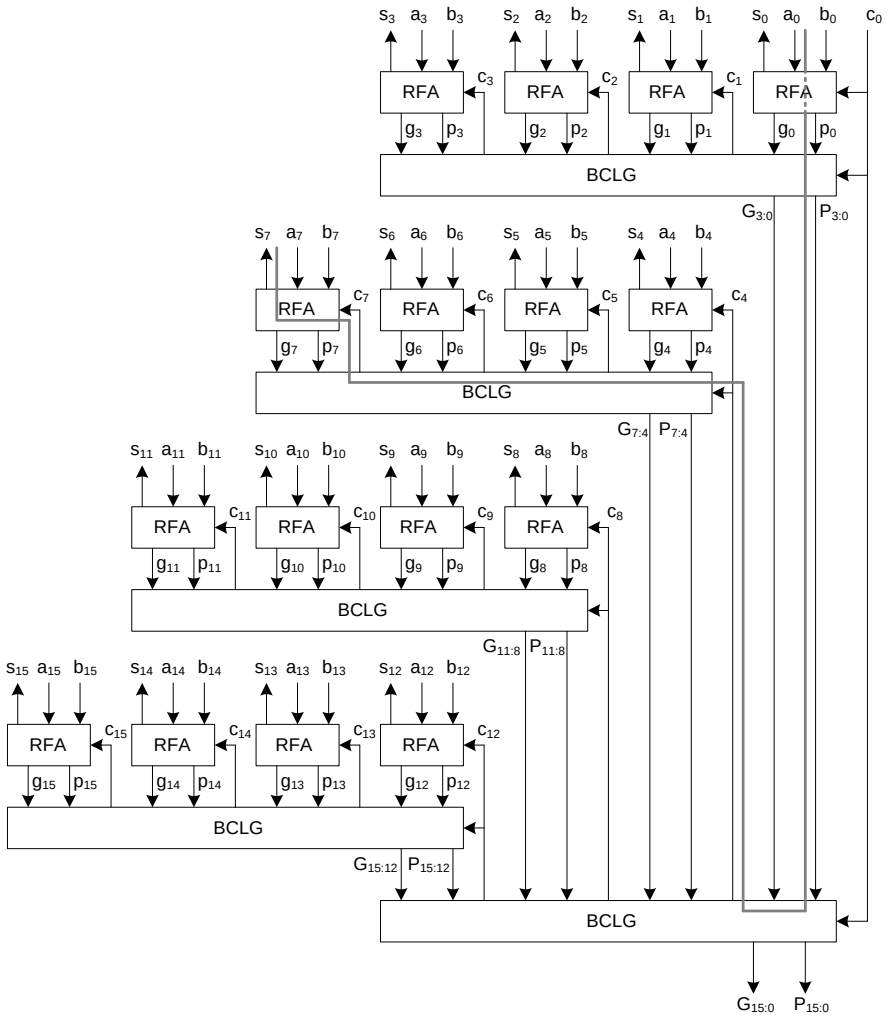


Figure 2.9: Carry-Lookahead Adder.

Each BCLG requires $\frac{(3+r)r}{2}$ gates and each RFA requires 8 gates, therefore, the total area of a CLA can be estimated as follows.

$$8n + \frac{(3+r)r}{2} \sum_{l=1}^{log_r n} \frac{n}{r^l} \tag{2.34}$$

2.6.1 Algorithmic Analysis

As seen in previous chapters, the basic adder algorithms have several algorithmic enhancements based on the length of their carry-chain. The algorithmic analysis is summarized in Table 2.1, and assumes all the adders have a width of n. Some of the adders divide these n

Table 2.1: Algorithmic Analysis

Adder Type	Delay(Δ)	Area (gates)
Ripple-Carry	$4n+6$	$9n$
Carry-Select	$4r+4k+2$	$22n-13r+2k-2$
Carry-Increment	$6r+4k-2$	$16n-7r+3k-3$
Carry-Skip	$8r+3k+5$	$9n+3k-6$
Carry-Lookahead	$3+8\lceil log_r n \rceil$	$8n+\frac{(3+r)r}{2}\sum_{l=1}^{log_r n}\frac{n}{r^l}$

bits into k groups, with a constant r-bits in each group.

2.7 Ling Adders

Although CLA structures are optimal for speeding up addition, they still can be improved. Ling adder [8] optimizes the performance by reducing fan-in of the logic gates. The idea is to utilize the property of carry generate and propagate.

$$g_i = a_i \cdot b_i \quad p_i = a_i + b_i, \tag{2.35}$$

The inherent relation can be observed as shown in the following equation.

$$g_i = g_i \cdot p_i \tag{2.36}$$

since

$$a_i \cdot b_i = a_i \cdot b_i \cdot (a_i + b_i) \tag{2.37}$$

The group carry generate can then be re-written as

$$\begin{aligned} G_{i:k} = &\quad p_i \cdot (g_i + g_{i-1} + p_{i-1} \cdot g_{i-2} + ... + p_{i-1} \cdot ... \cdot p_{k+1} \cdot g_k) \\ = &\quad p_i \cdot H_{i:k} \end{aligned} \tag{2.38}$$

where

$$\begin{aligned} H_{i:k} = &\quad g_i + g_{i-1} + p_{i-1} \cdot g_{i-2} + ... \\ &+ p_{i-1} \cdot ... \cdot p_{k+1} \cdot g_k \end{aligned} \tag{2.39}$$

Since the carry depends on a new term, it can be re-written as

$$\begin{aligned} c_{i+1} &= G_{i:k} + P_{i:k} \cdot c_k \\ &= p_i \cdot (H_{i:k} + P_{i-1:k}) \\ &= p_i \cdot d_{i+1} \end{aligned} \tag{2.40}$$

where the pseudo carry is introduced as

$$d_{i+1} = H_{i:k} + P_{i-1:k} \tag{2.41}$$

In the case of $k = -1$,

$$c_{i+1} = G_{i:-1} \tag{2.42}$$

and

$$d_{i+1} = H_{i:-1}. \tag{2.43}$$

Ling adders extract p_i, such that the fan-in of logic gates in the carry-chain is reduced. This is useful for dynamic logic since the serial stack of NMOS can be reduced by one and subsequently reduces sizing constraints. For example, consider a 4-bit group generate, such that

$$\begin{aligned} G_{i+3:i} &= g_{i+3} + p_{i+3} \cdot g_{i+2} + p_{i+3} \cdot p_{i+2} \cdot g_{i+1} \\ &\quad + p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot g_i \end{aligned} \tag{2.44}$$

while

$$\begin{aligned} H_{i+3:i} &= g_{i+3} + g_{i+2} + p_{i+2} \cdot g_{i+1} \\ &\quad + p_{i+2} \cdot p_{i+1} \cdot g_i. \end{aligned} \tag{2.45}$$

$H_{i+3:i}$ takes 7 inputs to 3 gates with a maximum fan-out of 3 whereas $G_{i+3:i}$ takes 10 inputs to 4 gates with a maximum fan-out of 4. It is obvious that the H term is much easier to obtain than the G term.

The H term shares the property of G, such that H can be derived as a group version as expressed in the following equation.

$$H_{i:k} = H_{i:j+1} + P_{i-1:j} \cdot H_{j:k} \tag{2.46}$$

Thus, the carry chain can be calculated with H instead of G. Due to the simpler circuit to computing H, the carry chain can be significantly reduced compared to a carry chain computing from G. If a long carry chain is divided with several blocks using H, the speed gain can significantly enhance the critical path, as opposed to using G in the blocks.

The sum for Ling adder can, therefore, be expressed in the following form.

$$s_i = p_i \oplus H_{i:-1} + g_i \cdot p_{i-1} \cdot H_{i-1:-1} \tag{2.47}$$

To prove Equation (2.47), there is another important relation that can be derived from Equation (2.45).

$$\begin{aligned} H_{i+3:i} &= g_{i+3} + p_{i+2} \cdot (g_{i+2} + g_{i+1} + p_{i+1} \cdot g_i) \\ &= g_{i+3} + p_{i+2} \cdot H_{i+2:i} \end{aligned} \tag{2.48}$$

since $H_{i+2:i} = g_{i+2} + g_{i+1} + p_{i+1} \cdot g_i$. Rewriting Equation (2.48), a more general and simpler expression can be derived.

$$H_{i:k} = g_i + p_{i-1} \cdot H_{i-1:k} \tag{2.49}$$

As $s_i = t_i \oplus c_i$ and $c_i = G_{i-1:-1} = p_{i-1} \cdot H_{i-1:-1}$,

$$\begin{aligned} s_i &= t_i \oplus p_{i-1} \cdot H_{i-1:-1} \\ &= \overline{t_i} \cdot p_{i-1} \cdot H_{i-1:-1} + t_i \cdot \overline{p_{i-1} \cdot H_{i-1:-1}} \\ &= (g_i + \overline{p_i}) \cdot p_{i-1} \cdot H_{i-1:-1} + p_i \cdot \overline{g}_i \cdot \overline{p_{i-1} \cdot H_{i-1:-1}} \end{aligned} \tag{2.50}$$

where $\overline{t_i} = g_i + \overline{p_i}$ and $t_i = p_i \cdot \overline{g_i}$. These two inserted equations can be easily proven by reducing the terms with a_i/b_i. Continue with Equation (2.50),

$$s_i = \overline{p_i} \cdot p_{i-1} \cdot H_{i-1:-1} + p_i \cdot \overline{g_i} \cdot \overline{p_{i-1} \cdot H_{i-1:-1}} + g_i \cdot p_{i-1} \cdot H_{i-1:-1} \tag{2.51}$$

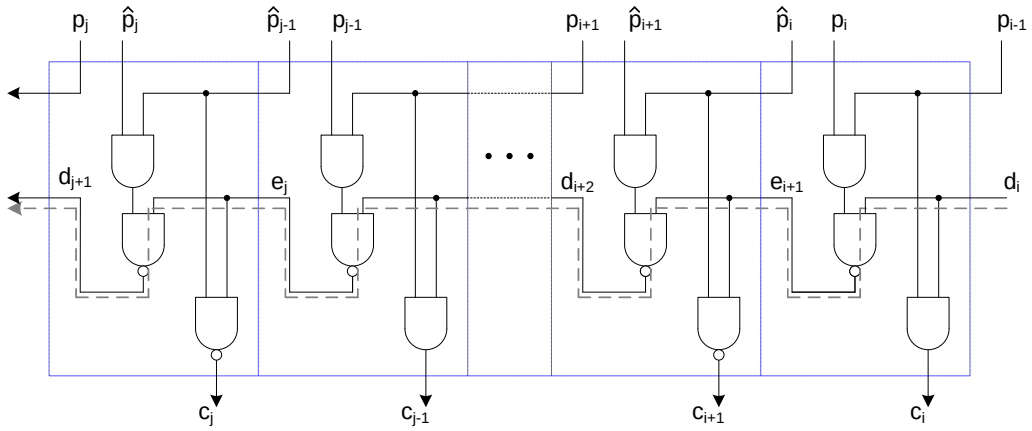


Figure 2.10: NAND Adder Carry Chain.

Adding another term $\overline{p_i} \cdot g_i$ to the Equation (2.51) also reduces the logic significantly. The added term has no effect on altering the logic since $\overline{p_i} \cdot g_i = 0$. Rewrite Equation (2.51) and applying Equation (2.49), the following equation can be derived, which is the same as Equation (2.47).

$$\begin{aligned} s_i &= \overline{p_i} \cdot (g_i + p_{i-1} \cdot H_{i-1:-1}) + p_i \cdot \overline{g_i + p_{i-1} \cdot H_{i-1:-1}} + g_i \cdot p_{i-1} \cdot H_{i-1:-1} \\ &= \overline{p_i} \cdot H_{i:-1} + p_i \cdot \overline{H_{i:-1}} + g_i \cdot p_{i-1} \cdot H_{i-1:-1} \end{aligned} \tag{2.52}$$

■

2.8 NAND/NOR Adders

Ling's equations can be further modified to enhance ripple-carry adders. This type of adders are also called NAND or NOR adders [39] [40]. In CMOS logic, NAND gates are usually faster than NOR gates while NOR adders are more suitable for domino logic. Because CMOS NOR gates have stacked PMOS transistors, they are usually slower than the NMOS transistors stacked in NAND gates. In domino logic, NOR gates has NMOS transistors in parallel, which is faster than NAND gates, where NMOS transistors are stacked in serial.

NAND/NOR adders utilize a similar idea as Ling's, but use a complemented version of

the propagate and generate from the following equations.

$$\begin{aligned} \hat{g}_i &= \overline{a_i} \cdot \overline{b_i} \\ \hat{p}_i &= \overline{a_i} + \overline{b_i} \end{aligned} \tag{2.53}$$

These equations can be formed into a new equation using the complemented values of the propagate/generate.

$$\hat{c}_{i+1} = \hat{g}_i + \hat{p}_i \cdot \overline{c_i} \tag{2.54}$$

An important property between the carry terms is

$$\hat{c}_{i+1} = \overline{c_{i+1}}. \tag{2.55}$$

This can be proven as the following procedure.

$$\begin{aligned} \overline{c_{i+1}} &= \overline{g_i + p_i \cdot c_i} \\ &= \overline{g_i} \cdot (\overline{p_i} + \overline{c_i}) \\ &= \overline{p_i} + \overline{g_i} \cdot \overline{c_i} \\ &= \hat{g}_i + \hat{p}_i \cdot \overline{c_i} = \hat{c}_i \end{aligned} \tag{2.56}$$

■

Besides Ling's pseudo-carry d, a second pseudo-carry e is introduced as following definition.

$$e_i \triangleq \hat{d}_i \tag{2.57}$$

For NAND adders, the pseudo-carries can be written as

$$\begin{aligned} d_{i+1} &= g_i + c_i \\ e_{i+1} &= \hat{g}_i + \overline{c_i} \end{aligned} \tag{2.58}$$

Applying DeMorgan's law and the relation $\hat{g}_i = \overline{p_i}$ and $\hat{p}_i = \overline{g_i}$, the complemented pseudo-carries can be expressed as

$$\begin{aligned} \overline{d_{i+1}} &= \hat{p}_i \cdot \overline{c_i} \\ \overline{e_{i+1}} &= p_i \cdot c_i \end{aligned} \tag{2.59}$$

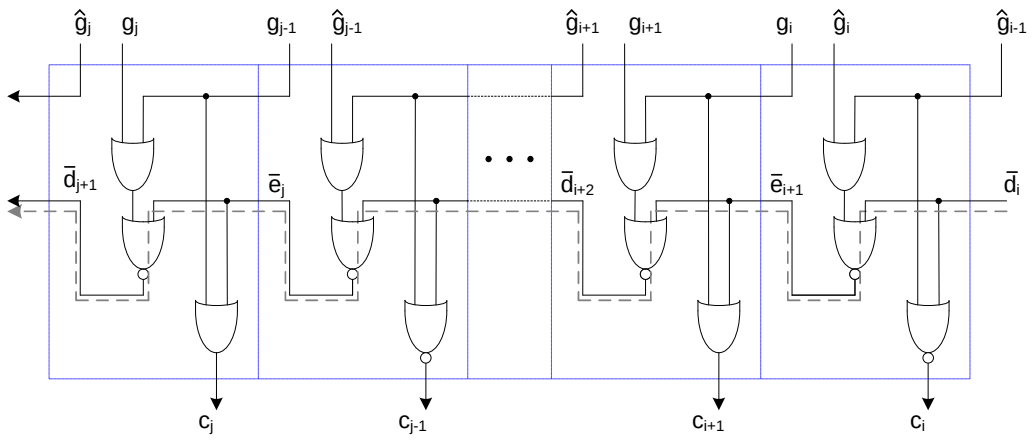


Figure 2.11: NOR Adder Carry Chain.

Insert the following relation between carry and pseudo-carries into the equations above.

$$\begin{aligned} c_i &= p_{i-1} \cdot d_i \\ \overline{c_i} &= \hat{p}_{i-1} \cdot e_i \end{aligned} \tag{2.60}$$

The NAND equations can now be rewritten as

$$\begin{aligned} \overline{d_{i+1}} &= \hat{p}_i \cdot \hat{p}_{i-1} \cdot e_i \\ \overline{e_{i+1}} &= p_i \cdot p_{i-1} \cdot d_i \end{aligned} \tag{2.61}$$

or

$$\begin{aligned} d_{i+1} &= \overline{\hat{p}_{i:i-1} \cdot e_i} \\ e_{i+1} &= \overline{p_{i:i-1} \cdot d_i} \end{aligned} \tag{2.62}$$

The carry chain in NAND adders with two pseudo-carries is shown in Figure 2.10. Thus, sum can be obtained using the following equation.

$$s_i = t_i \oplus c_i \tag{2.63}$$

or, in a different expression,

$$\begin{aligned} s_i &= t_i \oplus (p_{i-1} \cdot d_i) \\ &= \overline{t_i}(p_{i-1} \cdot d_i) + t_i(\hat{p}_{i-1} \cdot e_i) \end{aligned} \tag{2.64}$$

by referencing Equation (2.60). This can be implemented with a multiplexor with t_i as the select signal.

Similarly, the NOR adder can be derived from rewriting Equation (2.61).

$$\begin{aligned} \overline{d_{i+1}} &= \overline{\overline{\hat{p}_i \cdot \hat{p}_{i-1} \cdot e_i}} \\ \overline{e_{i+1}} &= \overline{\overline{p_i \cdot p_{i-1} \cdot d_i}} \end{aligned} \tag{2.65}$$

Further simplification can be achieved using DeMorgan's law.

$$\begin{aligned} \overline{d_{i+1}} &= \overline{g_i + g_{i-1} + \overline{e_i}} \\ \overline{e_{i+1}} &= \overline{\hat{g}_i + \hat{g}_{i-1} + \overline{d_i}} \end{aligned} \tag{2.66}$$

or

$$\begin{aligned} d_{i+1} &= g_i + g_{i-1} + \overline{e_i} \\ e_{i+1} &= \hat{g}_i + \hat{g}_{i-1} + \overline{d_i} \end{aligned} \tag{2.67}$$

Similar to the NAND adder, when computing the sum, the following equation can be utilized.

$$\overline{s_i} = t_i \oplus \overline{c_i} \tag{2.68}$$

In other words, the carry-chain has to be complemented and applied through Boolean logic to form the sm.

$$\begin{aligned} s_i &= t_i \odot \overline{c_i} \\ &= \overline{t_i}(g_{i-1} + \overline{e_i}) + t_i(\hat{g}_{i-1} + \overline{d_i}) \end{aligned} \tag{2.69}$$

These simplifications assume two carries produced after the carry tree.

$$\begin{aligned} c_i &= g_{i-1} + \overline{e_i} \\ \overline{c_i} &= \hat{g}_{i-1} + \overline{d_i} \end{aligned} \tag{2.70}$$

2.9 Parallel-Prefix Structures

To resolve the delay of carry-lookahead adders, the scheme of multilevel-lookahead adders or parallel-prefix adders can be employed [41]. The idea is to compute small group of intermediate prefixes and then find large group prefixes, until all the carry bits are computed. These adders have tree structures within a carry-computing stage similar to the carry propagate adder. However, the other two stages for these adders are called pre-computation and post-computation stages.

In pre-computation stage, each bit computes its carry generate/propagate and a temporary sum as in Equation (2.4). In the prefix stage, the group carry generate/propagate signals are computed to form the carry chain and provide the carry-in for the adder below. Equation (2.6) is repeated here as a reminder

$$\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:j} \\ P_{i:j} &= P_{i:k} \cdot P_{k-1:j} \end{aligned} \tag{2.71}$$

In the post-computation stage, the sum and carry-out are finally produced. The carry-out can be omitted if only a sum needs to be produced.

$$\begin{aligned} s_i &= t_i \oplus G_{i:-1} \\ c_{out} &= g_{n-1} + p_{n-1} \cdot G_{n-2:-1} \end{aligned} \tag{2.72}$$

where $G_{i:-1} = c_i$ with the assumption $g_{-1} = c_{in}$. The general diagram of parallel-prefix structures is shown in Figure 2.12, where an 8-bit case is illustrated.

All parallel-prefix structures can be implemented with the equations above, however, Equation (2.71) can be interpreted in various ways, which leads to different types of parallel-prefix trees. For example, Brent-Kung [9] is known for its sparse topology at the cost of more logic levels. There are several design factors that can impact the performance of prefix structures [42]

- Radix/Valency

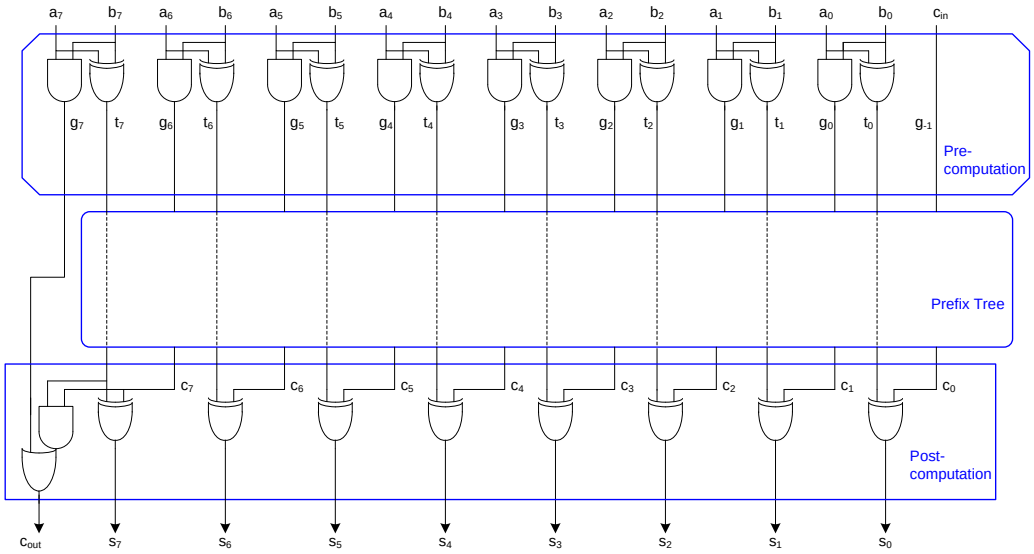


Figure 2.12: 8-bit Parallel-Prefix Structure.

- Logic Levels
- Fan-out
- Wire tracks

To illustrate a sample prefix structure, an 8-bit Sklansky [12] prefix tree is shown in Figure 2.13. Although Sklansky created this prefix structure with relationship to adders, it is typically referred to as a member of the Ladner-Fischer [13] prefix family. More details about prefix structures, including how to build the prefix structures and the performance comparison, will be described the next chapter of this dissertation.

2.10 Carry Save Adders (CSA)

Instead of performing binary addition by carry-propagating, it is possible to use redundant notation, to represent a number by using a carry-save adder [43] [44] [45]. Carry-save notation utilizes two vectors to store the carry bits and sum bits. In this way, carry-propagation is avoided but generally still required to form a final sum. Carry-save notation is usually

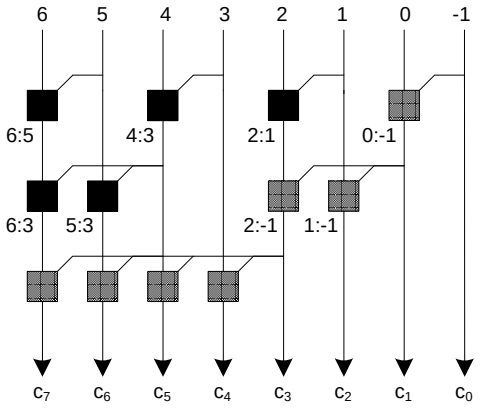


Figure 2.13: Sklansky Parallel-Prefix Examples.

used in multi-operand addition/subtraction, where multiple inputs are joined into two outputs. A 4-bit three-input carry-save adder example is shown in In Figure 2.14. The top of the figure shows the 4-bit adder bit by bit and bottom shows the vector form of the adder. Each bit is a full adder that takes three inputs and has two outputs. This can also be called as 3–2 counter. The computation at each bit for bit 0 through bit n is as follows.

$$\begin{aligned} vs_i &= d_i \oplus e_i \oplus f_i, \\ vc_i &= d_i \cdot e_i \cdot f_i \end{aligned} \tag{2.73}$$

The only exception is that

$$vc_0 = c_{in} \tag{2.74}$$

Since every bit is independent of each other and thus, there is no carry-chain. To compute the final binary sum, one more carry-propagate addition is required. The final process is,

$$\begin{aligned} S &= VC + VS \\ c_{out} &= vc_n \end{aligned} \tag{2.75}$$

Although not straight-forward, carry-save notation can also be applied to parallel-prefix structures to obtain a more balanced architecture, which will be discussed later on in this dissertation.

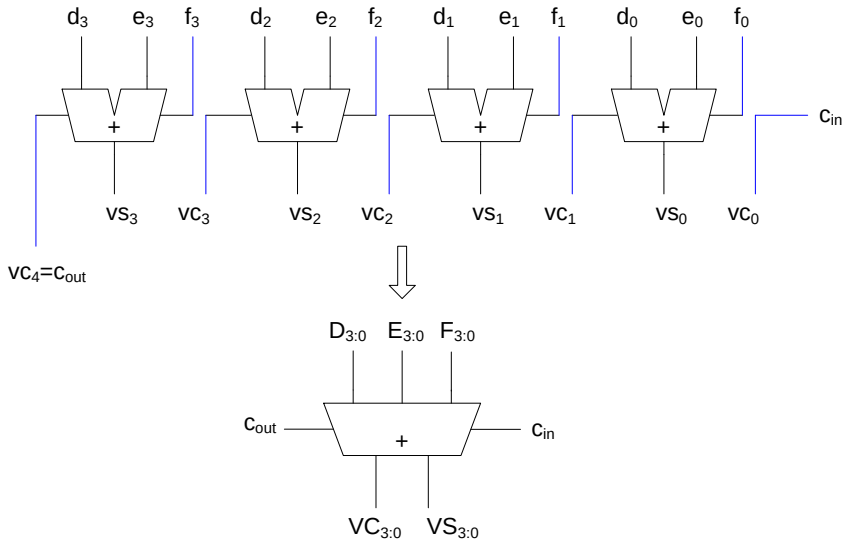


Figure 2.14: 4-bit Carry-Save Adder.

2.11 Summary

This chapter gives the background of binary carry propagate adders. Basic adder algorithms, from linear-based adders to logarithmic adders, are discussed in this chapter. Based on the defined terms and knowledge about the adders, next chapter will illustrate how to build prefix trees for use in logarithmic adders.

CHAPTER 3

Building Prefix Structures

Parallel-prefix structures are found to be common in high performance adders because of the delay is logarithmically proportional to the adder width. Such structures can usually be divided into three stages, pre-computation, prefix tree and post-computation. An example of an 8-bit parallel-prefix structure is shown in Figure 2.12

In the prefix tree, group generate/propagate are the only signals used. The group generate/propagate equations are based on single bit generate/propagate, which are computed in the pre-computation stage.

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i \oplus b_i \end{aligned} \tag{3.1}$$

where $0 \leq i \leq n$. $g_{-1} = c_{in}$ and $p_{-1} = 0$. Sometimes, p_i can be computed with OR logic instead of an XOR gate. The OR logic is mandatory especially when Ling's [8] scheme is applied. Here, the XOR logic is utilized to save a gate for temporary sum t_i.

In the prefix tree, group generate/propagate signals are computed at each bit.

$$\begin{aligned} G_{i:k} &= G_{i:j} + P_{i:j} \cdot G_{j-1:k} \\ P_{i:k} &= P_{i:j} \cdot P_{j-1:k} \end{aligned} \tag{3.2}$$

More practically, Equation (3.2) can be expressed using a symbol " $\circ$ " denoted by Brent and Kung [9]. Its function is exactly the same as that of a black cell. That is

$$(G_{i:k}, P_{i:k}) = (G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k}), \tag{3.3}$$

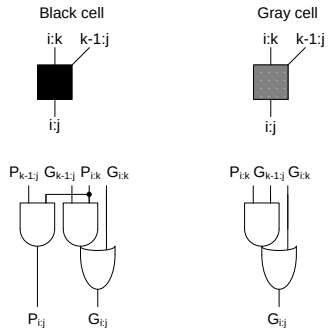


Figure 3.1: Cell Definitions.

or

$$\begin{aligned} G_{i:k} &= (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ ... \circ (g_k, p_k) \\ P_{i:k} &= p_i \cdot p_{i-1} \cdot ... \cdot p_k. \end{aligned} \tag{3.4}$$

The " $\circ$ " operation will help make the rules of building prefix structures.

In the post-computation, the sum and carry-out are the final output.

$$\begin{aligned} s_i &= p_i \cdot G_{i-1:-1} \\ c_{out} &= G_{n:-1} \end{aligned} \tag{3.5}$$

where "-1" is the position of carry-input. The generate/propagate signals can be grouped in different fashion to get the same correct carries. Based on different ways of grouping the generate/propagate signals, different prefix architectures can be created.

Figure 3.1 shows the definitions of cells that are used in prefix structures, including black cell and gray cell. Black/gray cells implement Equation (3.2) or (3.3), which will be heavily used in the following discussion on prefix trees.

3.1 Prefix Tree Family

Parallel-prefix trees have various architectures. These prefix trees can be distinguished by four major factors. 1) Radix/Valency 2) Logic Levels 3) Fan-out 4) Wire Tracks In the following discussion about prefix trees, the radix is assumed to be 2 (i.e. the number of

inputs to the logic gates is always 2). The more aggressive prefix schemes have logic levels $\lceil log_2(n) \rceil$, where n is the width of the inputs. However, these schemes require higher fan-out, or many wire-tracks or dense logic gates, which will compromise the performance e.g. speed or power. Some other schemes have relieved fan-out and wire tracks at the cost of more logic levels. When radix is fixed, The design trade-off is made among the logic levels, fan-out and wire tracks.

In the following sections, discussion on how to build a prefix tree will be presented. First, the basic elements to build a prefix tree are defined with a small example given as an introduction. After that, three representatives of the prefix family are described in detail, starting with Kogge-Stone [10], Knowles [11], followed by Brent-Kung [9]. Subsequently, Sklansky [12], Ladner-Fischer [13], Han-Carlson [14], and Harris [1] prefix trees are generally described. Finally, a new prefix scheme embedded with carry-save notation is introduced along with the explanation on how to form its structure.

3.2 Prefix Structure Synthesis

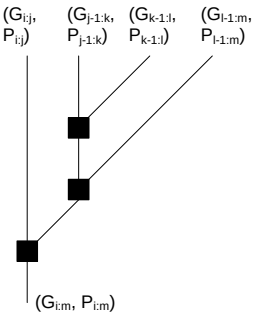
3.2.1 Taxonomy

The prefix trees can be classified using the factors mentioned above. Since the radix is fixed, the taxonomy uses the (l, f, t) [1], with each variable representing Logic Levels, Fan-out and Wire Tracks, respectively.

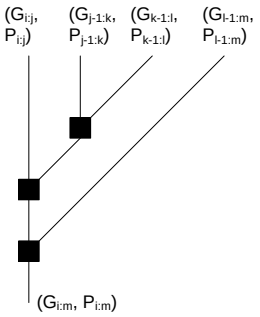
- Logic Levels: $L + l$
- Fan-out: $2^f + 1$
- Wire Tracks: 2^t

For Logic Levels, $L = log_2(n)$ for any n-bit prefix tree. l,f and t are integers between 0 and $L - 1$. Brent-Kung $(L - 1, 0, 0)$ prefix tree has the least fan-out and wiring tracks. This type of prefix tree always has a maximum fan-out of $2^0 + 1 = 2$ and maximum wire

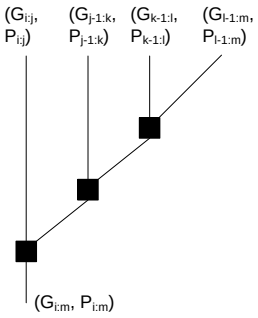
$(G_{i:j}, P_{i:j}) \circ \{[(G_{j-1:k}, P_{j-1:k}) \circ (G_{k-1:l}, P_{k-1:l})] \circ (G_{l-1:m}, P_{l-1:m})\}$



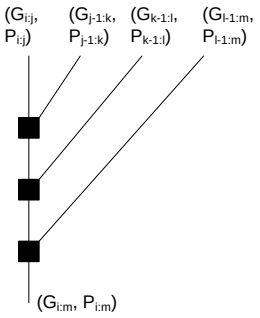
$\{(G_{i:j}, P_{i:j}) \circ [(G_{j-1:k}, P_{j-1:k}) \circ (G_{k-1:l}, P_{k-1:l})]\} \circ (G_{l-1:m}, P_{l-1:m})$



$(G_{i:j}, P_{i:j}) \circ \{(G_{j-1:k}, P_{j-1:k})] \circ [(G_{k-1:l}, P_{k-1:l}) \circ (G_{l-1:m}, P_{l-1:m})]\}$



$\{[(G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k})] \circ (G_{k-1:l}, P_{k-1:l})\} \circ (G_{l-1:m}, P_{l-1:m})$



$[(G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k})] \circ [(G_{k-1:l}, P_{k-1:l}) \circ (G_{l-1:m}, P_{l-1:m})]$

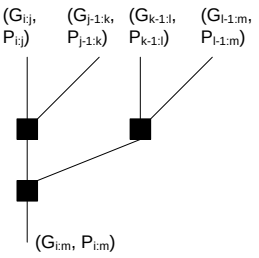


Figure 3.2: Valid Prefix Structure Synthesis.

track of $2^0 = 1$. But it requires the most logic levels among the prefix trees. For logic levels, a 16-bit Brent-Kung prefix tree has $L = 4$ and $l = L - 1 = 3$, which indicates a total of 7 logic levels is required. Kogge-Stone $(0, 0, L - 1)$ prefix tree has the least logic levels and fan-out. However, the wire tracks is 2^{L-1}, which is based on input width as L is logarithmically proportional to n. A 16-bit Kogge-Stone prefix tree has a maximum wire track of $2^{4-1} = 8$ with a dense gate structure. Sklansky $(0, L - 1, 0)$ prefix tree has the smallest number of logic levels and wiring tracks while the fan-out increases with wider inputs in the form of $2^{L-1} + 1$. A 16-bit Sklansky prefix tree has a maximum fan-out of $2^{4-1} + 1 = 9$. Knowles family $(0, f, t)$ has the least logic levels whereas the fan-out and wire tracks depend on the specific structure. Like Kogge-stone, Knowles prefix tree family use a high density gate structure. Ladner-Fischer $(L - 2, 1, 0)$ prefix tree targets at a fixed maximum fan-out which is $2^1 + 1 = 3$, compared to Sklansky prefix tree without too much compromise on logic levels. Han-Carlson $(1, 0, L - 2)$ prefix tree reduces the gate density compared to Kogge-Stone prefix tree with one more extra logic level than minimum. Its wire tracks is logarithmically proportional to input width.

3.2.2 Synthesis Rules

The key operation in a prefix structure is described in Equation (3.4). To build a prefix structure, the key operation is that the " $\circ$ " in (3.4), which is associative but not commutative. That is, which two of the neighboring term comes to the " $\circ$ " operation does not matter but the sequence of the neighboring terms can not be altered. For example, consider the following operation.

$$(G_{i:m}, P_{i:m}) = (G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k}) \circ (G_{k-1:l}, P_{k-1:l}) \circ (G_{l-1:m}, P_{l-1:m}), \quad (3.6)$$

Only the neighboring two terms can go through a " $\circ$ " operation. The sequence of the 4 terms on the right side of the equation can not be changed. The associativity, or the valid combination of the operations are shown in Figure 3.2.

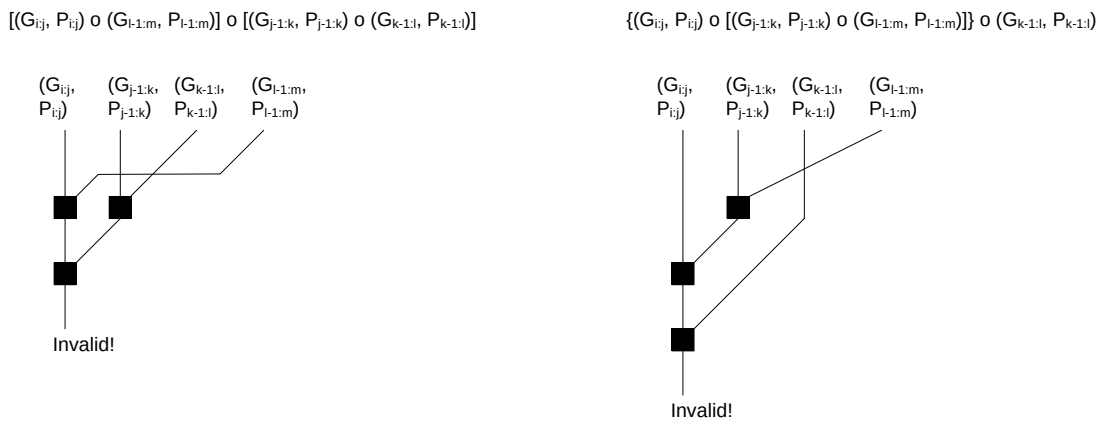


Figure 3.3: Invalid Prefix Structure Synthesis.

All the topologies in Figure 3.2 can be applied to building prefix structures. It can be observed that the last combination is the most efficient since it only take 2 logic levels while the other takes 3. This will be further discussed in the following sections.

Altering the order of the terms is not allowed because it will introduce invalid result. Two examples of invalid operations are shown in Figure 3.3.

3.2.3 Preparing Prefix Tree

The synthesis rules apply to any type of prefix tree. In this section, the methodology utilized to build fixed prefix structures is discussed. Moreover, procedure to build fixed prefix tree can be adapted to building non-fixed prefix tree with a slight modification.

In general, building prefix trees can be reduced to solving the following problems.

- How to align the bit lines.
- Where to place cells that compute group generate G and propagate P, i.e. black cells in this case (gray cells are ignored here to simplify the discussion.).
- How to connect input/output of the cells.

The solutions are based on the numbers which are power of 2 as both of the locations of the cells and wires can be related to those numbers.

To solve the problems, 3 terms are defined.

- $llevel$: logic level;
- u: maximum output bit span;
- v: maximum input bit span;

The $llevel$ refers to the logic row where group generate G and propagate P are computed. u/v are the maximum output bit span and input bit span of the logic cells. If the logic level is not the last of the prefix tree, the output of the current logic level will be the input to the next logic level. The maximum bit span sets the limit of the bit span at a certain logic level. The relations between these terms are described by the following equations

$$\begin{aligned} u &= 2^{llevel} \\ v &= 2^{llevel-1} \end{aligned} \tag{3.7}$$

The value of v is $1/2$ of the value of u. To further ease the illustration, the term $(G_{i:m}, P_{i:m})$ is briefed as $GP_{i:m}$. For example,

$$GP_{6:3} = GP_{6:5} \circ GP_{4:3} \tag{3.8}$$

which is equal to

$$\begin{aligned} G_{6:3} &= G_{6:5} + P_{6:5} \cdot G_{4:3} \\ P_{6:3} &= P_{6:5} \cdot P_{4:3} \end{aligned} \tag{3.9}$$

For this case, $llevel = 2, u = 4, v = 2$. The inputs are $GP_{6:3}$ and $GP_{4:3}$ that have a bit span of 2, as the subscripts of GP indicate. The output is $GP_{6:3}$, which has a bit span of 4.

Figure 3.4 shows an 8-bit example of an empty matrix with only bit lines and dashed boxes filled in. The inputs g_i/p_i go from the top and the outputs c_i are at the bottom. The LSB is labeled as -1 where the carry-input (c_{in}) locates. The objective is to obtain all c_i's in the form of $G_{i-1:-1}$'s, where $c_0 = G_{-1:-1}, c_1 = G_{0:-1}, c_2 = G_{1:-1}, ..., c_{n-1} = G_{n-2:-1}$

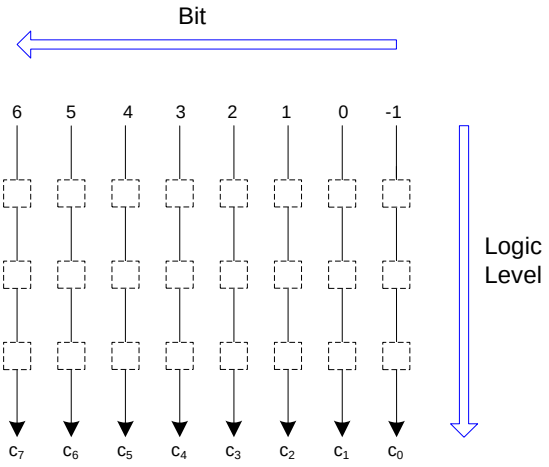


Figure 3.4: 8-bit Empty Prefix Tree.

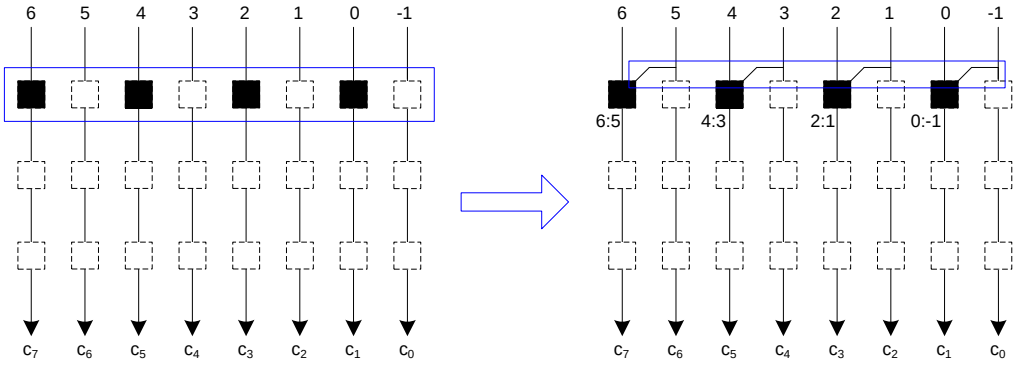


Figure 3.5: Build 8-bit Sklansky Prefix Tree: Step 1.

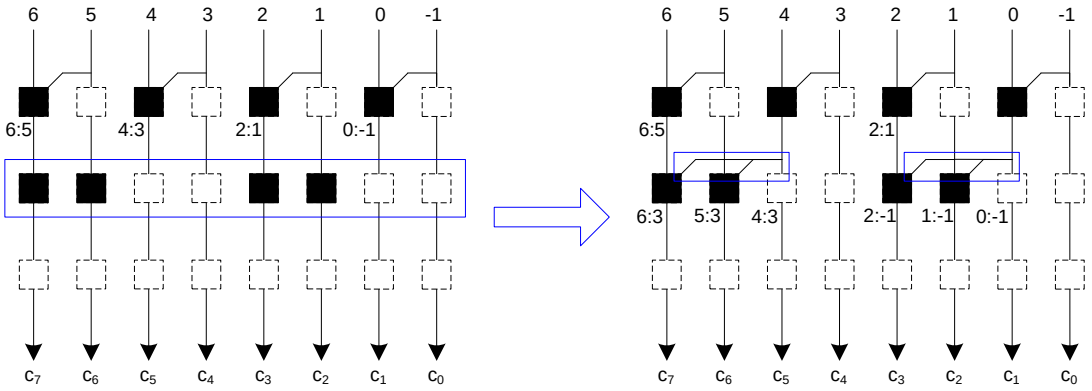


Figure 3.6: Build 8-bit Sklansky Prefix Tree: Step 2.

The way of building a prefix tree can be processed as the arrows indicate (i.e. from LSB to MSB horizontally and then from top logic level down to bottom logic level vertically).

Next, the empty matrix will be filled to compose a Sklansky prefix tree. In the first logic level, black cells are filled every other bit, which means the neighboring 2 bits will be the input to only one black cell. (For example, Kogge-Stone prefix trees, the g_i/p_i will be the input to 2 black cells.) This complies with the term mentioned above (i.e. maximum input span is 1 bit and maximum output span is $2^1 = 2$ bits). The inter-connect can be wired according the the location of the black cell and the maximum output span. The process is shown in Figure 3.5 as step 1.

The 2nd logic level can be continued, such that the black cells are placed as step 2 shown in Figure 3.6. Two black cells are grouped together and the distance between the neighboring group is 2 bits (i.e. 2 empty bit slots). The maximum input span is 2 bits and the maximum output span is $2^2 = 4$ bits, e.g. $GP_{6:3}$. In this level, the maximum fan-out is 3.

The similar fashion applies to logic level 3, where black cells are placed in a group of 4. The distance between the neighboring group is supposed to be 4 bits. As this is a 8-bit example, there exists only one such group. The maximum input span is 4 bits and the maximum output span is $2^3 = 8$ bits, e.g. $G_{6:-1}$. In this logic level, the maximum fan-out increases to 5. Finally, the prefix tree is complete when the tree computes all the carry-out

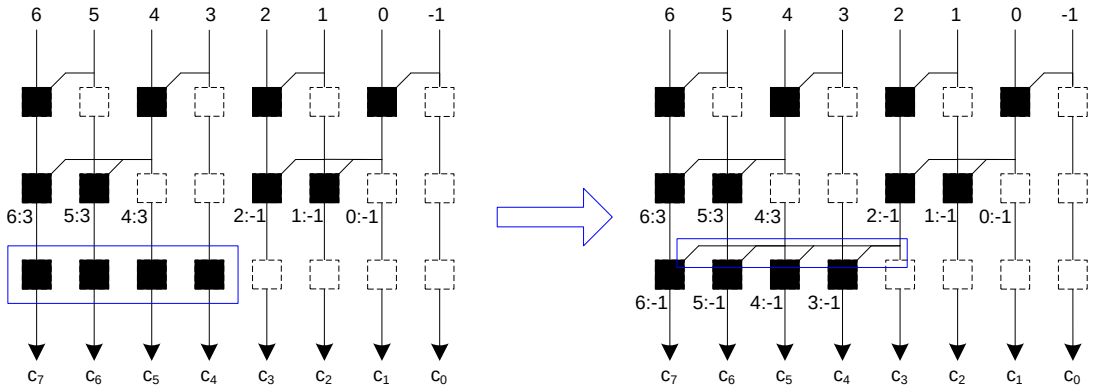


Figure 3.7: Build 8-bit Sklansky Prefix Tree: Step 3.

bits.

The example shown in Figure 3.7 is an 8-bit Sklansky prefix tree. Other types of prefix trees can be built following the similar procedure by utilizing the bit span and black cell placement according to the specific structure requirement. In the following sections, building several fixed prefix trees will be discussed with pseudo-code and examples given.

3.2.4 Kogge-Stone Prefix Tree

Kogge-Stone prefix tree is among the type of prefix trees that use the fewest logic levels. A 16-bit example is shown in Figure 3.8. In fact, Kogge-Stone is a member of Knowles prefix tree 3.10. The 16-bit prefix tree can be viewed as Knowels [1,1,1,1]. The numbers in the brackets represent the maximum branch fan-out at each logic level. The maximum fan-out is 2 in all logic levels for all width Kogge-Stone prefix trees.

The key of building a prefix tree is how to implement Equation (3.2) according to the specific features of that type of prefix tree and apply the rules described in the previous section. Gray cells are inserted similar to black cells except that the gray cells final output carry outs instead of intermediate G/P group. The reason of starting with Kogge-Stone prefix tree is that it is the easiest to build in terms of using a program concept. The example in Figure 3.8 is 16-bit (a power of 2) prefix tree. It is not difficult to extend the structure to any width if the basics are strictly followed.

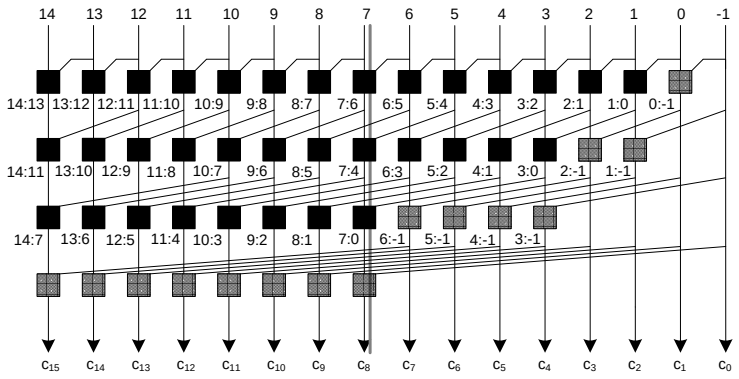


Figure 3.8: 16-bit Kogge-Stone Prefix Tree.

For the Kogge-Stone prefix tree, at the logic level 1, the inputs span is 1 bit (e.g. group (4:3) take the inputs at bit 4 and bit 3). Group (4:3) will be taken as inputs and combined with group (6:5) to generate group (6:3) at logic level 2. Group (6:3) will be taken as inputs and combined with group (10:7) to generate group (10:3) at logic level 3, and so on so forth. With this inspection, the structure can be described with the Algorithm 3.1 listed below.

Algorithm 3.1 Building Kogge-Stone Prefix Tree

$L = log_2(n)$;

for $llevel = 1; llevel \leq L; llevel ++$ **do**

$u = 2^{llevel}$; {output bit span}

$v = 2^{llevel-1}$; {input bit span}

for $i = v - 1; i < n - 1; i ++$ **do**

$GP_{i:i-u+1} = (GP_{i:i-v+1}) \circ (GP_{i-v:i-u+1})$;

end for

end for

In Algorithm 3.1, the number of logic levels is calculated first. At each logic level, the maximum input bit span and maximum output bit span are computed. Equation (3.2) is applied in the inner loop, where bit goes from bit $v - 1$ though bit $n - 1$. If any of the subscript goes less than -1, the value stays at -1. This means there is no crossing over bit

Table 3.1: Verifying the Pseudo-Code of Building a Kogge-Stone Prefix Tree

Logic Levels	u	v	Output $(i:i-u+1)$	Input1 $(i:i-v+1)$	Input2 $(i-v:i-u+1)$	Equation Mapping
1	2	1	$7:6$	$7:7$	$6:6$	$GP_{7:6} = GP_7 \circ GP_6$
2	4	2	$11:8$	$11:10$	$9:8$	$GP_{11:8} = GP_{11:10} \circ GP_{9:8}$
3	8	4	$14:7$	$14:11$	$10:7$	$GP_{14:7} = GP_{14:11} \circ GP_{10:7}$
4	16	8	$7:-1$	$7:0$	$-1:-1$	$GP_{7:-1} = GP_{7:0} \circ GP_{-1}$

-1, or the LSB boundary.

The statement in the inner for loop is applying Equation (3.2). The validity of this implementation can be verified by looking at Table 3.1. In the table, one group operation is randomly selected at each logic level. Other operations can be verified by inserting the numbers as listed in Table 3.1. The term $GP_{i:i} = GP_i$ and LSB boundary of the inputs/outputs is bit -1. Table 3.1 can also be matched against Figure 3.8 to see the correspondence.

The pseudo-code is a simplified version of the exact program. In the real program, the code should tell where are the black cells and gray cells. The program also needs control so that the LSB never goes beyond -1 and utilizes optional buffers. In Figure 3.8, there are fan-outs more than 2 because the structure is not buffered. Figure 3.9 shows a buffered 16-bit prefix tree, however, the exact number of buffers is based on the capacitance and resistance of the interconnect network [46]. Both figures indicate a wire track of 8.

The algorithmic delay is simply the number of logic levels. The area can be estimated as the number of cells in the prefix tree. To simply the calculation, all cells are counted as black cells. To understand this structure, remember that the number of gray cells always equals to $n-1$ since the prefix tree only outputs $n-1$ carries. A black cell has one more AND gate than a gray cell, and therefore, a more accurate area estimation will just subtract that $n-1$ AND gates.

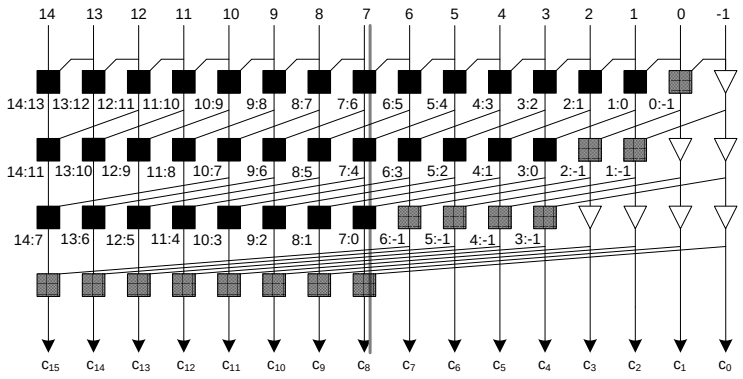


Figure 3.9: 16-bit Kogge-Stone Prefix Tree with Buffers.

The number cells for a Kogge-Stone prefix tree can be counted as follows. Each logic level has $n - m$ cells, where $m = 2^{llevel-1}$. That is, each logic level is missing m cells. That number is the sum of a geometric series starting from 1 to $n/2$ which totals to $n - 1$. The total number of cells will be $nlog_2n$ subtracting the total number of cells missing at each logic level, which winds up with $nlog_2n - n + 1$. When $n = 16$, the area is estimated as 49.

3.2.5 Knowles Prefix Tree

Knowles [11] proposed a family of prefix trees with flexible architectures. Knowles prefix trees use the fan-out at each logic level to name their family members. As mentioned in the previous section, Kogge-Stone is actually Knowles [1,1,1,1]. Figure 3.10 shows a 16-bit Knowles [2,1,1,1] prefix tree. Knowles prefix structure for 16-bit adders can be in other topologies, such as with Knowles [4,2,1,1], [4,4,2,1], [8,2,2,1], [8,4,1,1]. A 16-bit Knowles [4,2,1,1] prefix tree is illustrated in Figure 3.11, where the fan-out in logic level 3 and 4 are 2 and 4, respectively. Even different fan-out in the same logic level is allowed in Knowles prefix trees, which is called hybrid Knowles prefix tree.

The Knowles prefix tree family has multiple architectures which it can implement. However, the idea here is to clarify how to build Knowles prefix trees based on previously mentioned Kogge-Stone prefix trees, and gradually move on to more complex prefix

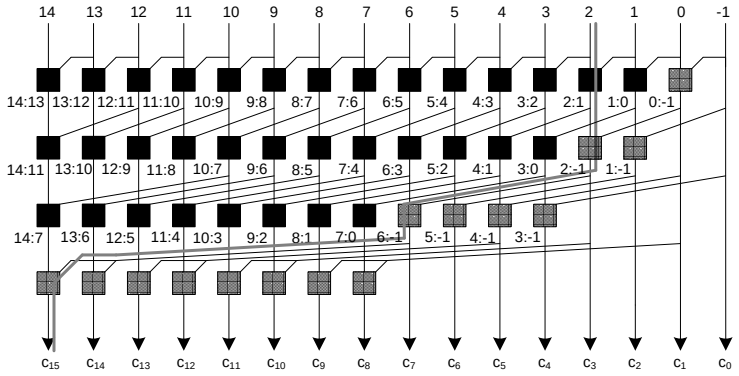


Figure 3.10: 16-bit Knowles [2,1,1,1] Prefix Tree.

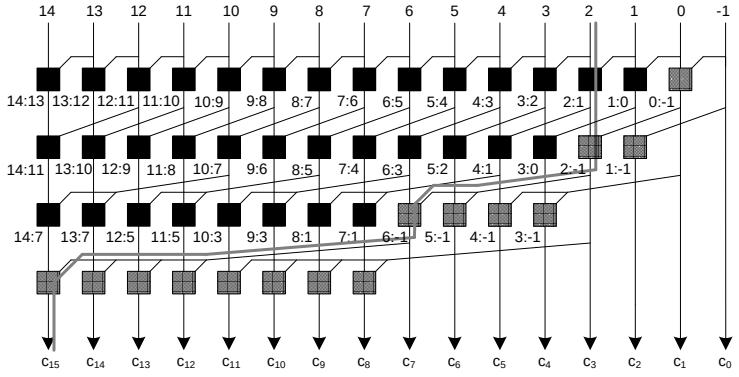


Figure 3.11: 16-bit Knowles [4,2,1,1] Prefix Tree.

structures (e.g. Brent-Kung prefix tree). To proceed without introducing much distraction, only the construction of Knowles [2,1,1,1] is described in detail. It will not be difficult to extend the algorithm once the basic concepts on the prefix trees are firmly established.

The only difference between Figure 3.10 and Figure 3.8 is at the final or logic level 4. In Knowles [2,1,1,1] prefix tree, the fan-out at logic level 4 is 3 instead of 2. To build such prefix trees, the pseudo-code made for Kogge-Stone prefix tree can be reused except for the change at the last level.

Algorithm 3.2 Building Knowels [2,1,1,1] Prefix Tree

$L = log_2(n)$;
for $llevel = 1; llevel \leq L; llevel ++$ **do**
  $u = 2^{llevel}$; {output bit span}
  $v = 2^{llevel-1}$; {input bit span}
  if $llevel == L$ **then**
    for $i = v - 1; i < n - 1; i += 2$ **do**
      $GP_{i:i-u+1} = (GP_{i:i-v+1}) \circ (GP_{i-v+1:i-u+1})$;
      $GP_{i+1:i-u+2} = (GP_{i+1:i-v+2}) \circ (GP_{i-v+1:i-u+1})$;
    end for
  else
    {same as building Kogge-Stone prefix tree}
  end if
end for

In Algorithm 3.2, the inner for loop has i increment by 2. Two GP group share one same input. An observation can be made that there is overlap at bit $i - v + 1$ in the first GP operation. This is valid and can be proved as follows, which is to prove

$$(GP_{i:j}) \circ (GP_{j:k}) = (GP_{i:j}) \circ (GP_{j-1:k}) \tag{3.10}$$

or

$$(G_{i:j}, P_{i:j}) \circ (G_{j:k}, P_{j:k}) = (G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k}) \tag{3.11}$$

The left side for G in the equation above can be rewritten as

$$\begin{aligned} G_{i:j} + P_{i:j} \cdot G_{j:k} &= G_{i:j} + P_{i:j} \cdot (G_{j:j} + P_{j:j} \cdot G_{j-1:k}) \\ &= G_{i:j} + P_{i:j} \cdot G_{j:j} + P_{i:j} \cdot P_{j:j} \cdot G_{j-1:k} \\ &= G_{i:j} + P_{i:j} \cdot g_j + P_{i:j} \cdot G_{j-1:k} \end{aligned} \tag{3.12}$$

where $G_{j:j} = g_j$. The term $G_{i:j}$ of the Equation (3.12) can be expanded as

$$G_{i:j} = g_i + p_i \cdot g_{i-1} + P_{i:i-1} \cdot g_{i-2} + ... + P_{i:j+1} \cdot g_j \tag{3.13}$$

Adding with the second term in Equation (3.12),

$$\begin{aligned} G_{i:j} + P_{i:j} \cdot g_j = \; & g_i + p_i \cdot g_{i-1} + P_{i:i-1} \cdot g_{i-2} + ... \\ & + P_{i:j+1} \cdot g_j + P_{i:j} \cdot g_j \end{aligned} \tag{3.14}$$

The last two terms in the equation above can be absorbed as one term since

$$\begin{aligned} P_{i:j+1} \cdot g_j + P_{i:j} \cdot g_j &= P_{i:j+1} \cdot g_j + P_{i:j+1} \cdot p_j \cdot g_j \\ &= P_{i:j+1} \cdot g_j \cdot (1 + p_j) \\ &= P_{i:j+1} \cdot g_j \end{aligned} \tag{3.15}$$

Inserting Equation (3.15) into Equation (3.14) and compare the right side with Equation (3.13), the following relationship is established.

$$G_{i:j} + P_{i:j} \cdot g_j = G_{i:j} \tag{3.16}$$

Therefore, Equation (3.12) becomes

$$G_{i:j} + P_{i:j} \cdot G_{j:k} = G_{i:j} + P_{i:j} \cdot G_{j-1:k}, \tag{3.17}$$

Table 3.2: Verifying the Pseudo-Code of Building a Knowles [2,1,1,1] Prefix Tree

Logic Levels	u	v	Output $(i:i-u+1)$	Input1 $(i:i-v+1)$	Input2 $(i-v+1:i-u+1)$	Equation Mapping
4	16	8	$11:-1$	$11:4$	$4:-1$	$GP_{11:-1} = GP_{11:4} \circ GP_{4:-1}$
Logic Levels	u	v	Output $(i:i-u+2)$	Input1 $(i:i-v+2)$	Input2 $(i-v+1:i-u+1)$	Equation Mapping
4	16	8	$12:-1$	$12:5$	$4:-1$	$GP_{12:-1} = GP_{12:5} \circ GP_{4:-1}$

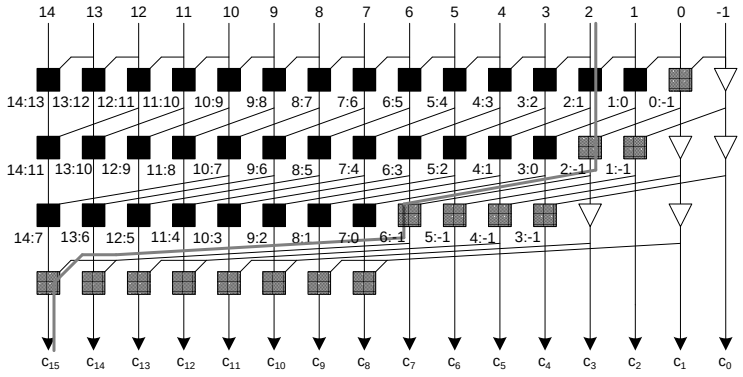


Figure 3.12: 16-bit Knowles [2,1,1,1] Prefix Tree with Buffers.

which proves the G part of Equation (3.11). The P part is easier to prove because P is computed using a AND operation. Any redundant term is absorbed by other terms.

$$\begin{aligned} P_{i:j} \cdot P_{j:k} &= P_{i:j} \cdot (p_j \cdot P_{j-1:k}) \\ &= P_{i:j} \cdot P_{j-1:k} \end{aligned} \tag{3.18}$$

where $P_{i:j} \cdot p_j = P_{i:j}$. It can be proven that overlapping is allowed even for more than 1 bit as it is allowed in prefix trees (e.g. Knowles [4,2,1,1]). ■

Figure 3.12 shows a properly buffered structure that has a maximum fan-out of 3 (i.e. $f = 1$). Both Kogge-Stone and Knowles [2,1,1,1] prefix tree have the same number of logic levels. More interestingly, they also have the same number of cells. Hence, the area for Knowles [2,1,1,1] prefix tree is also estimated as $nlog_2n - n + 1$.

3.2.6 Brent-Kung Prefix Tree

Brent-Kung prefix tree is a well-known structure with relatively sparse network. The fan-out is among the minimum as $f = 0$. So is the wire tracks where $t = 0$. The cost is the extra $L - 1$ logic levels. A 16-bit example is shown in Figure 3.13. The critical path is shown in the figure with a thick gray line.

Brent-Kung prefix tree is a bit complex to build because it has the most logic levels. To build such a structure, the pseudo-code can be composed as Algorithm 3.3.

Algorithm 3.3 Building Brent-Kung Prefix Tree

$L = log_2(n)$;
for $llevel = 1; llevel \leq L; llevel++$ **do**
  $u = 2^{llevel}$; {output bit span}
  $v = 2^{llevel-1}$; {input bit span}
  for $i = u - 2; i < n - 1; i+= u$ **do**
    $GP_{i:i-u+1} = (GP_{i:i-v+1}) \circ (GP_{i-v:i-u+1})$;
  end for
end for
for $llevel = L - 1; llevel \geq 1; llevel--$ **do**
  $u = 2^{llevel}$; {output bit span}
  $v = 2^{llevel-1}$; {input bit span}
  for $i = u + v - 2; i < n - 1; i+= u$ **do**
    $GP_{i:-1} = (GP_{i:i-v+1}) \circ (GP_{i-v:i-1})$;
  end for
end for

The algorithm deals with this prefix tree in 2 major for loops. The first for loop handles logic level by logic level from 1 up to L with the second for loop handling the rest $L - 1$ logic levels in a decremental fashion. Figure 3.13 can be divided as the top 4 logic levels and the bottom 3 logic levels. The structure starts with cells every 2 bits. The input

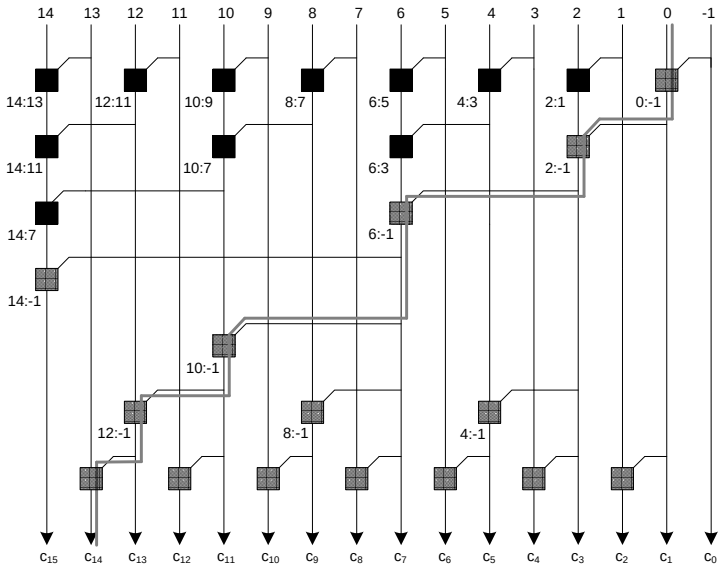


Figure 3.13: 16-bit Brent-Kung Prefix Tree.

span is 1 bit and the output span is 2 bits. At logic level 2 and 3, the distance between each cell is 4 and 8 bits, respectively. The input/output span is $2/4$ bits at logic level 2 and $4/8$ bits at logic level 3. At logic level 4, the only cell is at MSB (bit 14 from input) with input spanning 8 bits and output spanning 16 bits. By logic level 4, some carries are already generated. At logic level 5 through 7, the input bit span is decremented instead of being incremented as in the previous cases. The input bit spans at logic level 5 through 6 and 7 are 4, 2 and 1 bit, respectively. The term output span no longer applies to these $L-1$ levels since all the outputs are the final carries with the form $G_{i:-1}$. Table 3.3 lists the examples to further understand and verify the pseudo-code. The pseudo-code above does not include any buffer or tell where to insert gray cells. Figure 3.14 shows a properly buffered 16-bit prefix tree that has a maximum fan-out of 2.

The delay is estimated as the number of logic levels (i.e. L). The total number of cells can be calculated in the following way. In the first $log_2 n$ logic levels, the number of cells is a geometric series. For example, in the 16-bit prefix tree, at logic level 1 through 4, there are 8, 4, 2, 1 cell at each level. The sum of this series is $n-1$. For the rest of the logic levels, there only exist $gray$ cells. The total number of $gray$ cells is $n-1$ for any prefix tree

Table 3.3: Verifying the Pseudo-Code of Building a Brent-Kung Prefix tree

Logic Levels	u	v	Output $(i:i-u+1)$	Input1 $(i:i-v+1)$	Input2 $(i-v:i-u+1)$	Equation Mapping
1	2	1	$6:5$	$6:6$	$5:5$	$GP_{6:5} = GP_6 \circ GP_5$
2	4	2	$10:7$	$10:9$	$8:7$	$GP_{10:7} = GP_{10:9} \circ GP_{8:7}$
3	8	4	$6:-1$	$6:3$	$2:-1$	$GP_{6:-1} = GP_{6:3} \circ GP_{2:-1}$
4	16	8	$14:-1$	$14:7$	$6:-1$	$GP_{14:-1} = GP_{14:7} \circ GP_{6:-1}$
Logic Levels	u	v	Output $(i:-1)$	Input1 $(i:i-v+1)$	Input2 $(i-v:-1)$	Equation Mapping
5	8	4	$10:-1$	$10:7$	$6:-1$	$GP_{10:-1} = GP_{10:7} \circ GP_{6:-1}$
6	4	2	$12:-1$	$12:11$	$10:-1$	$GP_{12:-1} = GP_{12:11} \circ GP_{10:-1}$
7	2	1	$1:-1$	$1:1$	$0:-1$	$GP_{1:-1} = GP_1 \circ GP_{0:-1}$

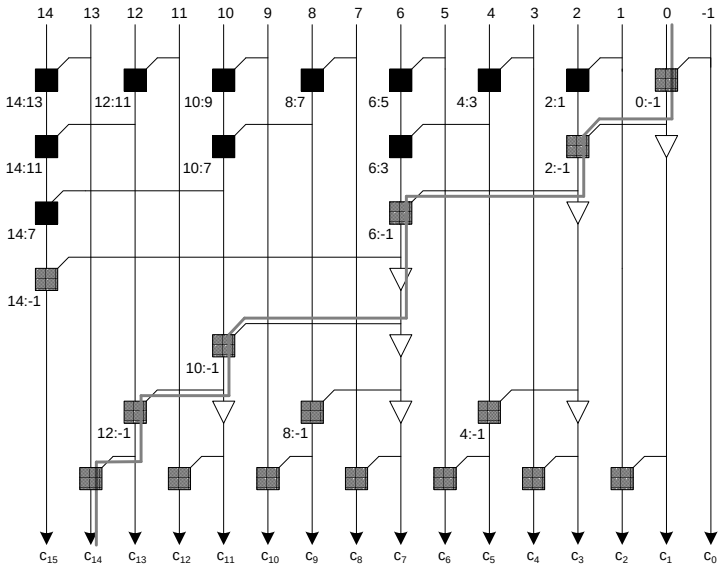


Figure 3.14: 16-bit Brent Kung Prefix Tree with Buffers.

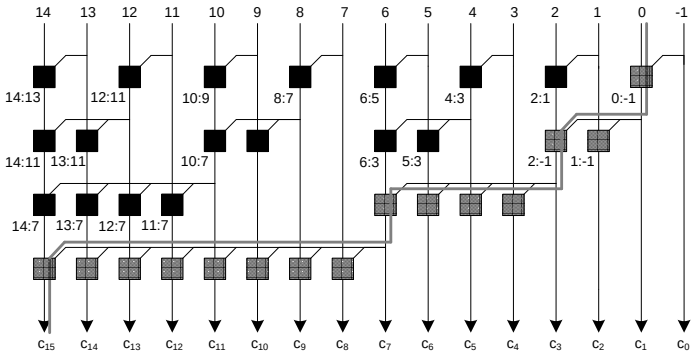


Figure 3.15: 16-bit Sklansky Prefix Tree.

as mentioned before. However, in the previous $log_2 n$ logic levels, the prefix tree contains $log_2 n$ gray cells. The sum of cells is $2(n-1) - log_2 n$. When $n = 16$, the number of cells required is 26.

3.3 Other Prefix Trees

3.3.1 Sklansky Prefix Tree

Sklansky prefix tree takes the least logic levels to compute the carries. Plus, it uses less cells than Knowles [2,1,1,1] and Kogge-Stone structure at the cost of higher fan-out. Figure 3.15 shows the 16-bit example of Sklansky prefix tree with critical path in solid line.

For a 16-bit Sklansky prefix tree, the maximum fan-out is 9 (i.e. $f = 3$). The structure can be viewed as a compacted version of Brent-kung's, where logic levels is reduced and fan-out increased. A similar pseudo-code listed for Brent-Kung prefix tree can be used to generate a Sklansky prefix tree. However, the maximum input span is still a power of 2, relating with the number of logic levels. The difference is that one more for loop is required to account for the multiple fan-out (e.g. at logic level 2 through 4 in Figure 3.15, where the cells are placed in group of 2,4 and 8, respectively).

The number of logic levels is $log_2 n$. Each logic level has $n/2$ cells as can be observed in Figure 3.15. The area is estimated as $(n/2)log_2 n$. When $n = 16$, 32 cells are required.

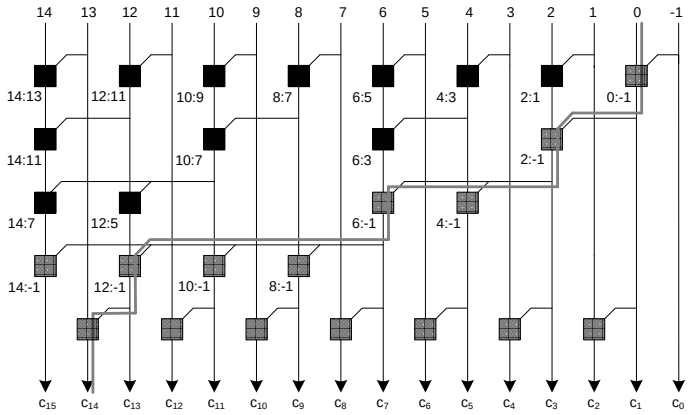


Figure 3.16: 11-bit Ladner-Fischer Prefix Tree Synthesis.

3.3.2 Ladner-Fischer Prefix Tree

Sklansky prefix tree has the minimum logic levels, and uses less cells than Kogge-Stone and Knowles prefix trees. The major problem of Sklansky prefix tree is its high fan-out. Ladner-Fischer prefix tree is proposed to relieve this problem.

To reduce fan-out without adding extra cells, more logic levels have to be added. Figure 3.16 shows a 16-bit example of Ladner-Fischer prefix tree.

Ladner-Fischer prefix tree is a structure that sits between Brent-Kung and Sklansky prefix tree. It can be observed that in Figure 3.16, the first two logic levels of the structure are exactly the same as Brent-Kung's. Starting from logic level 3, fan-out more than 2 is allowed (i.e. $f > 0$). Comparing the fan-out of Ladner-Fischer's and Sklansky's, the number is reduced by a factor of 2 since Ladner-Fischer prefix tree allows more fa-nout one logic level later than Sklansky prefix tree.

Building a Ladner-Fischer prefix tree can be seen as a relieved version of Sklansky prefix tree. For a structure like Figure 3.16, a extra row of cells are required to generate the missing carries.

The delay for the type of Ladner-Fischer prefix tree is $log_2 n + 1$. The first and last logic level take $n/2$ and $n/2 - 1$ cells. In between, there are $log_2 n - 1$ logic levels, each having $n/4$ cells. Summing up the cells, $n/2 + n/2 - 1 + (n/4)(log_2 n - 1)$, which is equal to

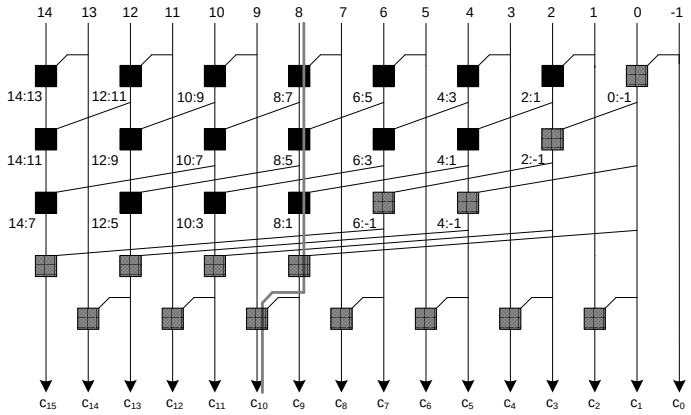


Figure 3.17: 16-bit Han-Carlson Prefix Tree.

$(n/4)log_2n + 3n/4 - 1$. When $n = 16$, total cells required is 27.

3.3.3 Han-Carlson Prefix Tree

The idea of Han-Carlson prefix tree is similar to Kogge-Stone's structure since it has a maximum fan-out of 2 or $f = 0$. The difference is that Han-Carlson prefix tree uses much less cells and wire tracks than Kogge-Stone. The cost is one extra logic level.

Han-Carlson prefix tree can be viewed as a sparse version of Kogge-Stone prefix tree. In fact, the fan-out at all logic levels is the same (i.e. 2). The pseudo-code for Kogge-Stone's structure can be easily modified to build a Han-Carlson prefix tree. The major difference is that in each logic level, Han-Carlson prefix tree places cells every other bit and the last logic level accounts for the missing carries. Figure 3.17 shows a 16-bit Han-Carlson prefix tree, ignoring the buffers. The critical path is shown with thick solid line.

This type of Han-Carlson prefix tree has $log_2n + 1$ logic levels. It happens to have the same number cells as Sklansky prefix tree since the cells in the extra logic level can be move up to make the each of the previous logic levels all have $n/2$ cells. The area is estimated as $(n/2)log_2n$. When $n = 16$, the number is 32.

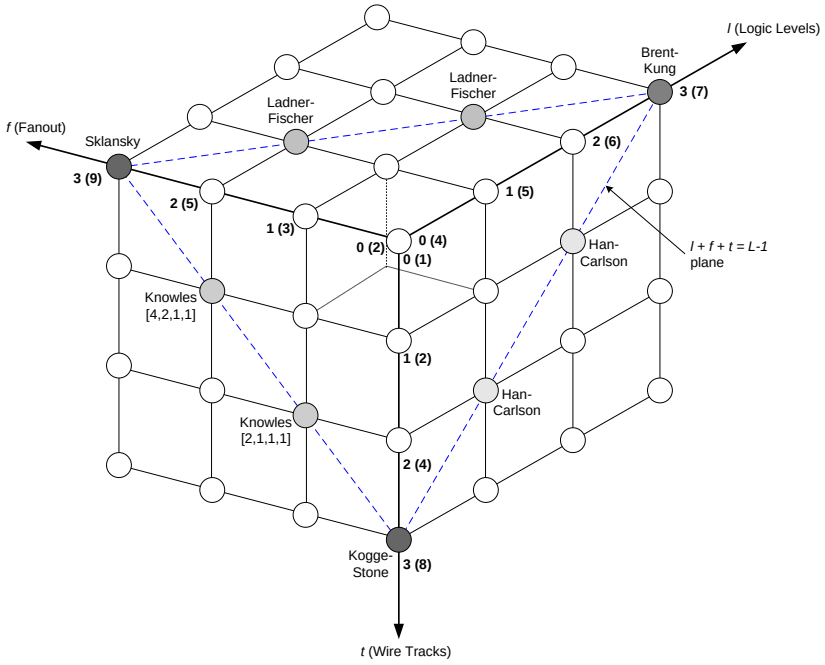


Figure 3.18: Taxonomy of 16-bit Prefix Tree (Adapted from [1]).

3.3.4 Harris Prefix Tree

The idea from Harris about prefix tree is to try to balance the logic levels, fan-out and wire tracks. Harris proposed a cube to show the taxonomy for prefix trees in Figure 3.18, which illustrates the idea for 16-bit prefix trees [1]. In the figure, all the prefix trees mentioned above are on the cube, with Sklansky prefix tree standing at the fan-out extreme, Brent-Kung at the logic levels extreme, and Kogge-Stone at the wire track extreme.

The balanced prefix structure is close to the center of cube (i.e.when $n = 16$, $l = 1$, $f = 1$ and $t = 1$ or represented in short by $(1, 1, 1)$). The logic levels is $2^4 + 1 = 5$, maximum fan-out is $2^f + 1 = 3$ and wire track is $2^t = 2$. The diagram is shown in Figure 3.19 with critical path in solid line. Observation can be made that there is bit overlap in logic level 4 similar to Knowles $[2, 1, 1, 1]$. The overlap is valid for producing correct carries as it has be proven for Knowles $[2, 1, 1, 1]$.

For $n \geq 16$, $(1, 1, 1)$ will not be sufficient to build a prefix tree. More logic levels, or fan-out, or wire tracks need to be added. For example, when $n = 32$, the prefix tree can be

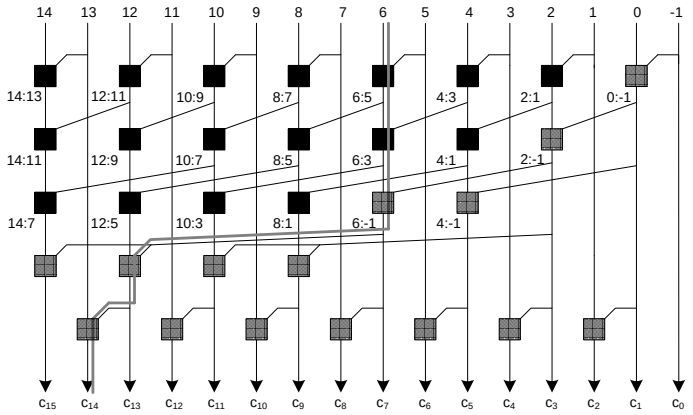


Figure 3.19: 16-bit Harris Prefix Tree.

Table 3.4: Algorithmic Analysis

Types	Logic Levels	Area	Fan-out	Wire tracks
Brent-Kung	$2log_2n - 1$	$2n - log_2n - 2$	2	1
Kogge-Stone	log_2n	$nlog_2n - n + 1$	2	$n/2$
Ladner-Fischer	$log_2n + 1$	$(n/4)log_2n + 3n/4 - 1$	$n/4 + 1$	1
Knowles$[2, 1, 1, 1]$	log_2n	$nlog_2n - n + 1$	3	$n/4$
Sklansky	log_2n	$(n/2)log_2n$	$n/2 + 1$	1
Han-Carlson	log_2n	$(n/2)log_2n$	2	$n/4$
Harris	$log_2n + 1$	$(n/2)log_2n$	3	$n/8$

in the form of $(1, 1, 2)$,$(1, 2, 1)$ and $(2, 1, 1)$. Like Ladner-Fischer and Han-Carlson prefix tree illustrated in the previous sections, Harris prefix tree has $log_2n + 1$ logic levels. It needs the same number of cells required for Han-Carlson and Sklansky prefix tree, which is $(n/2)log_2n$.

3.3.5 Algorithmic Analysis for Prefix Trees

Unfolding the algorithms mentioned, prefix trees can be built structurally either by HDL or schematic entry. Table 3.4 summarizes the prefix trees' parameters, including logic levels, area estimation, fan-out and wire tracks.

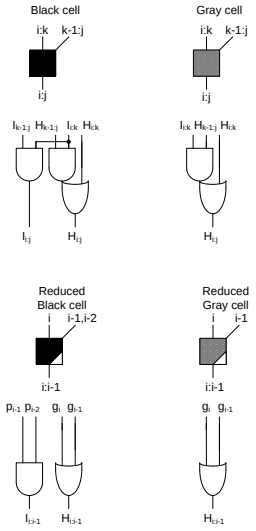


Figure 3.20: Cell Definition for Ling's Parallel-Prefix Tree.

3.4 Parallel-Prefix Ling Structures

As mentioned in Chapter 2, similar to CLA's, Ling adders have a faster critical path than CLA's, and parallel-prefix adders are based on the same idea as CLA's. Ling's scheme can apply to parallel-prefix structure as well. To build a Ling parallel-prefix adder, starting from pre-computation stage, carry-generate/propagate are repeated as follows.

$$g_i = a_i \cdot b_i \quad p_i = a_i + b_i, \tag{3.19}$$

In the equation above, p_i can no longer be mixed with XORing a_i and b_i.

In the parallel-prefix tree stage, unlike regular parallel-prefix adder, Ling adder has its own group generate/propagate terms H and I. This requires a redefinition of cells to build Ling type of parallel-prefix adders. Figure 3.20 shows the new cell definitions. The relationship between H/I pair and G/P can be described in the following equations.

$$\begin{aligned} H_{i:k} &= g_i + G_{i-1:k} \\ I_{i:k} &= P_{i-1:k-1} \end{aligned} \tag{3.20}$$

With a bit span of one, the H/I can start from p/g, which will be the first logic level of the

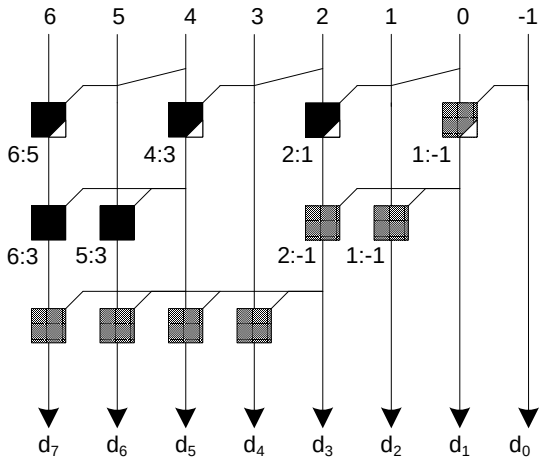


Figure 3.21: 8-bit Sklansky Prefix Tree with Ling's Scheme.

prefix tree.

$$\begin{aligned} H_{i:i-1} &= g_i + g_{i-1} \\ I_{i:i-1} &= p_{i-1} \cdot p_{i-2} \end{aligned} \tag{3.21}$$

As for $i = 0$, $i - 2 = -2$ which is a non-existent bit, this corresponds to $I_{0:-1}$. In the real application, there is no such worry since $I_{0:-1}$ is not used. In regular prefix tree, even $P_{0:-1}$ is not needed since at bit $i = 0$, there is always a gray cell, which means only G is computed.

The terms H/I have the similar recurrence relation as G/P.

$$\begin{aligned} H_{i:k} &= H_{i:j} + I_{i:j} \cdot H_{j-1:k} \\ I_{i:k} &= I_{i:j} \cdot I_{j-1:k} \end{aligned} \tag{3.22}$$

These equations are essentially the same as those for regular parallel-prefix structure, which means that same cells can be used to implement the equations. In actuality, starting from logic level 2, the prefix tree with Ling's scheme has no physical difference than the regular prefix tree. The difference is merely logical, i.e. Ling's prefix tree produce pseudo-carry $H_{i:-1}$ or d_{i+1}, while regular prefix tree produce real carry $G_{i:-1}$ or c_{i+1}.

Ling's scheme also makes the final post-computation stage unique. To compute the

sum, an equation presented earlier can be used

$$s_i = p_i \oplus H_{i:-1} + g_i \cdot p_{i-1} \cdot H_{i-1:-1}, \tag{3.23}$$

which concludes the computation.

3.5 Prefix Tree with Carry-Save Notation

3.5.1 Carry-Save Notation

Carry-save notation utilizes the redundant notation to store the sum and carry-out. This idea can be incorporated in the prefix tree. There are two steps required for this scheme: use carry-save notation and then add the notation with carry lookahead equations.

Start from the first step with the carry-save formula.

$$\begin{aligned} s_i &= a_i \oplus b_i = t_i \\ c_i &= a_i \cdot b_i = g_i \end{aligned} \tag{3.24}$$

Figure 3.22 shows a n-bit example how to align the t_i and g_i. Here temporary sum t_i is set apart from carry-propagate p_i. The carry-in bit is illustrated as the following equation indicates.

$$g_{-1} = c_{in}. \tag{3.25}$$

Using a carry-save notation, the sum and carry can be expressed into a prefix structure. The temporary sum t_i''s are obtained by XOR gates as normal binary add operation does.

$$\begin{aligned} t'_0 &= t_0 \oplus g_{-1} \\ t'_1 &= t_1 \oplus g_0 \\ &... \\ t'_{n-1} &= t_{n-1} \oplus g_{n-2} \end{aligned} \tag{3.26}$$

Using the carry lookahead Equation (2.5) to compute $c'_{i+1}, i = 0, 1, ..., n-1$.

$$c'_{i+1} = g'_i + p'_i \cdot c'_i \tag{3.27}$$

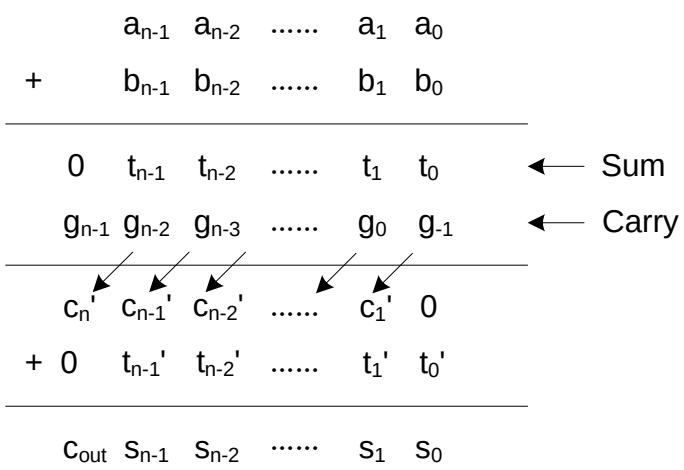


Figure 3.22: n-bit Binary Adder with Carry-Save Notation.

where

$$\begin{aligned} g_i' &= t_i \cdot g_{i-1} \\ p_i' &= t_i + g_{i-1} \end{aligned} \tag{3.28}$$

The sum s_i is the result of XORing t_i' and c_i', i.e.

$$s_i = t_i' \oplus c_i' \tag{3.29}$$

Finally, the carry-out can be derived as

$$c_{out} = g_{n-1} \oplus c_n' \tag{3.30}$$

or, in another expression,

$$c_{out} = g_{n-1} + t_{n-1} \cdot c_{n-1}. \tag{3.31}$$

This equation differs drastically from previous carry-out equations, however, it can easily be proven using simple definitions. The following proof documents this relationship.

Proof

$$\begin{aligned} c_{out} &= g_{n-1} \oplus c_n' \\ &= g_{n-1} \oplus (t_{n-1} \cdot c_{n-1}) \\ &= g_{n-1} \cdot \overline{t_{n-1} \cdot c_{n-1}} + \overline{g_{n-1}} \cdot (t_{n-1} \cdot c_{n-1}) \end{aligned} \tag{3.32}$$

When $g_{n-1} = 0$

$$c_{out} = t_{n-1} \cdot c_{n-1}. \tag{3.33}$$

This result is the same as Equation (3.31) indicates when $g_{n-1} = 0$. If $g_{n-1} = 1$,

$$c_{out} = \overline{t_{n-1} \cdot c_{n-1}}. \tag{3.34}$$

When $g_{n-1} = 1$, t_{n-1} has to be 0 because of the relation in Equation (3.37). Thus

$$c_{out} = \overline{0} = 1, \tag{3.35}$$

which agrees with Equation (3.31) when $g_{n-1} = 1$. Hence the expression in Equation (3.30) computes exactly the same outcome as Equation (3.31). ■

3.5.2 Carry-Save Notation and Application to Prefix Tree

By looking at Figure 3.22, the carry-save notation, i.e. sum and carry can be directly input to a prefix tree. Figure 3.23 shows a 16-bit Sklansky prefix tree adapted to carry-save notation without considering the MSB carry output.

From Figure 3.23, the structure is observed to be a shifted right 1-bit version compared to Figure 3.15. The outputs are c_i's and the logic for MSB in Figure 3.15 can be saved. However, this is at the cost of extra carry-save logic and most importantly, the number of logic levels stays the same. Plus, the assumption is to ignore the MSB carry output. If the carry output is to be considered, either the same structure as Figure 3.15 or some extra logic out of the prefix tree is required. In summary, the saving of directly applying the carry-save notation to a prefix structure is not general. In fact, carry-save notation is not targeting at reducing the logic of those prefix trees with minimum logic levels.

Further observation needs to be made so that the carry-save notation can help in a general sense. In Figure 3.22, there is no carry input at bit 0 as $c_{in} = g_{-1}$, which is aligned

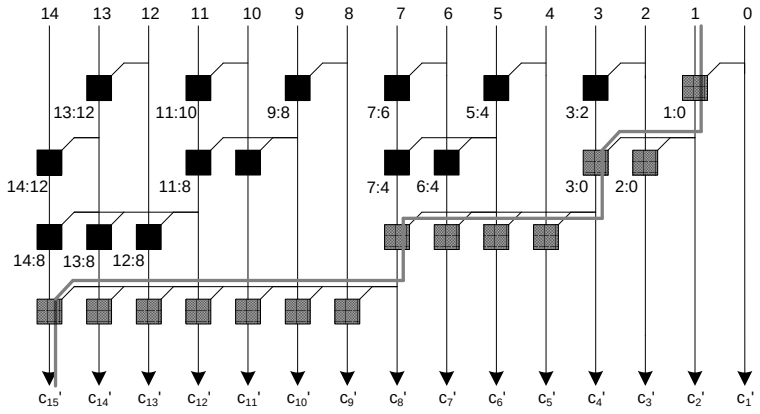


Figure 3.23: 16-bit Sklanksy Prefix Tree with Carry-Save Notation.

with t_0, i.e. $c'_0 = g_{-2} = 0$. Use Equation (2.5)

$$\begin{aligned} c'_1 &= g'_0 + p'_0 \cdot c'_0 \\ &= t_0 \cdot g_{-1} + (t_0 + g_{-1}) \cdot 0 \\ &= t_0 \cdot g_{-1} \end{aligned} \tag{3.36}$$

The important element here is that

$$t_i \cdot g_i = 0, \tag{3.37}$$

since $(a_i \cdot \overline{b_i} + \overline{a_i} \cdot b_i) \cdot (a_i \cdot b_i) = 0$. This will be applied to the following computation. Proceed to c'_2

$$\begin{aligned} c'_2 &= g'_1 + p'_1 \cdot c'_1 \\ &= t_1 \cdot g_0 + (t_1 + g_0) \cdot (t_0 \cdot g_{-1}) \\ &= t_1 \cdot g_0 + t_1 \cdot t_0 \cdot g_{-1} \end{aligned} \tag{3.38}$$

The normal carry c_1 can be computed as $c_1 = g_0 + t_0 \cdot g_{-1}$, c'_2 can be written in terms of c_1.

$$c'_2 = t_1 \cdot c_1 \tag{3.39}$$

Similarly,

$$\begin{aligned} c'_3 &= g'_2 + p'_2 \cdot c'_2 \\ &= t_2 \cdot g_1 + (t_2 + g_1) \cdot (t_1 \cdot g_0 + t_1 \cdot t_0 \cdot g_{-1}) \\ &= t_2 \cdot g_1 + t_2 \cdot t_1 \cdot g_0 + t_2 \cdot t_1 \cdot t_0 \cdot g_{-1} \\ &= t_2 \cdot c_2 \end{aligned} \tag{3.40}$$

where $c_2 = g_1 + t_1 \cdot g_0 + t_1 \cdot t_0 \cdot g_{-1}$. By simply expanding the equation, c'_{n-1} and c'_n can be computed as

$$\begin{aligned} c'_{n-1} &= t_{n-2} \cdot c_{n-2} \\ c'_n &= t_{n-1} \cdot c_{n-1} \end{aligned} \tag{3.41}$$

This can be generalized as

$$c'_i = t_{i-1} \cdot c_{i-1} \tag{3.42}$$

The fastest parallel-prefix tree computes the carry using $\lceil log_2 n \rceil$ levels of logic assuming n-bit addition and 2-input gates only. For example, a 16-bit addition take 4 logical levels to obtain the carry in the most significant bit by using the parallel-prefix structure such as Sklansky, Kogge-Stone or Knowles $[2, 1, 1, 1]$. One more observation can be made that from bit 7 to bit 15, logic level is increased to 4. That is, the increasing point of logic level are at bit i and $i = 2^j - 2$, where $j = 1, 2, ..., \lceil log_2 n \rceil - 1$.

As Equation (3.42) indicates, c'_i can take advantage when the previous carry c_{i-1} is available. Assuming a 16-bit addition, computing the carry at bit 15 takes the most logic levels (i.e. 4). On the other hand, using a carry-save notation will not help reduce the logic levels anywhere below 4, since the carry at bit 14 take 4 logic levels, as well.

A simple application of Equation (3.42) cannot help reduce logic levels except for the $2^m + 1$ bit additions (m is an integer) where the number of bits is at the boundary of logic level increase. For example, a 9-bit addition normally using Sklansky parallel-prefix structure takes $\lceil log_2 9 \rceil = 4$ logic levels or gray/black cells as shown on the left of Figure 3.25.

Example: 85 (hex) + 66 (hex) + 1 = EC (hex)

Bit		7	6	5	4	3	2	1	0	-1
	a_i	1	0	0	0	0	1	0	1	
	b_i	0	1	1	0	0	1	1	0	
+	c_{in}									1
	g_i	0	0	0	0	0	1	0	0	1
	t_i	1	1	1	0	0	0	1	1	0
	c_i	0	0	0	0	1	1	1	1	0
	t_i'	1	1	1	0	1	0	1	0	
	c_i'	0	0	0	0	0	1	1	0	

Figure 3.24: 8-bit Example of Carry-Save Add.

Using carry-save notation takes 3 logic levels plus 1 AND gate, saving 1 OR gate compared to normal addition. This is shown on the right part of Figure 3.25. c_8' will need its counter-part t_8' and this will be discussed more soon. Therefore, the carry-save notation must utilize Equation (3.42) inter-digitally to achieve a more balanced prefix structure.

3.5.3 Adding Binary Carry-Save Numbers

This section illustrates the validity of using a prefix adder that utilizes carry-save notation. The most important equation derived from the previous section is Equation (3.42). The normal carry-lookahead equation can be changed to the following form.

$$\begin{aligned} c_i &= g_{i-1} + t_{i-1} \cdot c_{i-1} \\ &= g_{i-1} + c_i' \end{aligned} \tag{3.43}$$

This means that computing c_i takes one more OR operation than computing c_i'. This can be exploited to optimize the critical path from input to sum. As proven in the previous section, the saving here doesn't apply to the path from input to carry-out since the carry-save notation takes the same effort as normal carry-lookahead adder does. To get the correct

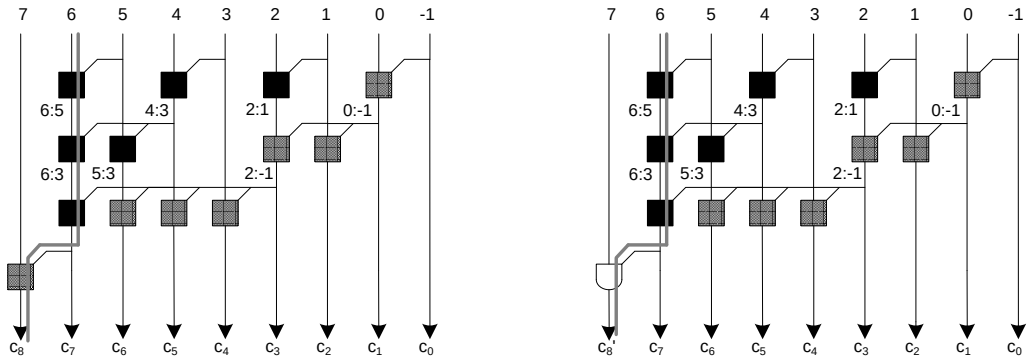


Figure 3.25: 9-bit Sklansky Parallel-Prefix Adder Comparison.

sum with c'_i, t'_i is required.

$$t'_i = t_i \oplus g_{i-1} \tag{3.44}$$

Then final sum can be derived as Equation (3.29).

The XOR operation to get t'_i indicates an extra gate to compute the right sum. However, it can be inserted into pre-computation stage, which is not in the critical path. Next section will discuss more about this.

An 8-bit carry-save add example is shown in Figure 3.24. In the figure, two numbers $A = a_7, a_6, ..., a_0$ and $B = b_7, b_6, ...b_0$ are added with carry c_{in}. t_i's and g_i's are obtained using Equation (2.4). In this case, $g_{-1} = 1$ since $g_{-1} = c_{in}$. c_i's are derived with simple observation regarding to A, B and c_{in}. c'_i's and t_i's are computed with Equation (3.42) and Equation (3.44) respectively. It is easy to verify that bitwise XORing t_i's with c_i's has the same result as bitwise XORing t'_i's with c'_i's. They both end up with binary string 11101100 which is EC in hex.

3.5.4 Incorporating Parallel-Prefix Structure with Carry-save Notation

Simple carry-save notation will not help the prefix tree as described earlier. However, carry-save notation can help optimize the prefix tree with a combination of carries of c_i and c'_i. This is applicable to the prefix trees with reduced wire tracks or fan-out at the cost of more logic levels. Such type of prefix trees include Brent-Kung, Han-Carlson, Ladner-Fischer or

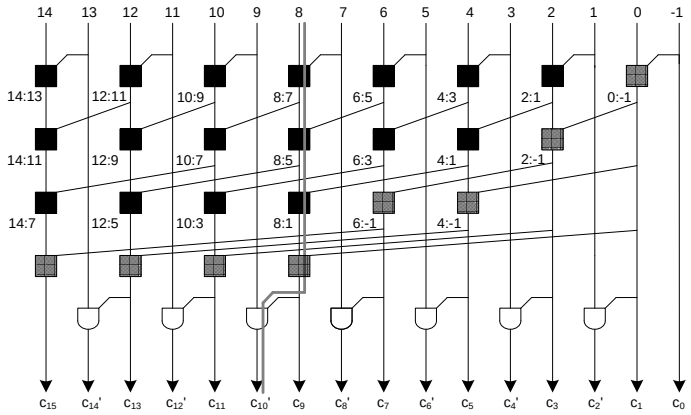


Figure 3.26: 16-bit Han-Carlson Parallel-Prefix Structure with Carry-Save Notation.

Table 3.5: Comparison of selected parallel-prefix structures

Type	Logic Levels	Max Fan-out	Wire Tracks
Brent-Kung	$2log_2n - 1$	2	1
Sklansky	log_2n	$n/2 + 1$	1
Han-Carlson	$log_2n + 1$	2	$n/4$
Modified Han-Carlson	$log_2n + 0.5$	2	$n/4$
Ladner-Fischer	$log_2n + 1$	$n/4 + 1$	1
Modified Ladner-Fischer	$log_2n + 0.5$	$n/4 + 1$	1

Harris's structures. The idea is to have inter-digitated c_i and c_i' so that Equation (3.42) can be utilized. In this way, not every bit needs to be in carry-save notation. In the positions where c_i''s are generated, t_i''s are required for the sum. In the rest positions, g_i/t_i can still function as in normal prefix trees. It will be illustrated that the difference happens at the final logic level. Figure 3.17 shows that a 16-bit Han-Carlson prefix tree takes 5 levels of logic. The final logic level is to reduce dense gates in those fast prefix trees which take 4 logic levels as mentioned.

The carry-save notation can be applied to Han-Carlson prefix tree as follows. Instead of having a last row of gray cells, a row of AND gates are placed in that logic level to produce c_i''s instead of c_i's, and thus, saving one OR logic. As the neighboring c_i is available,

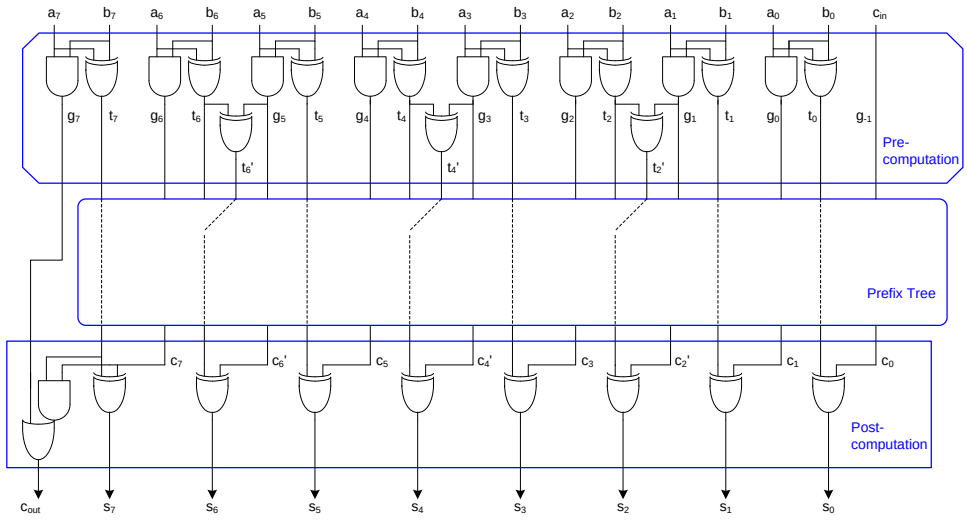


Figure 3.27: 8-bit Parallel-Prefix Adder with Carry-Save Notation.

the next c'_{i+1} can be computed immediately by using Equation (3.42). The modified Han-Carlson structure is illustrated in Figure 3.26. Only the even bits except bit 0 have carry $c'_i, i = 2, 4, ..., 14$. These carries will be XORed with t'_i's to get the right sum bits.

For odd bits and bit 0, the sum is calculated as follows.

$$s_i = t_i \cdot c_i \tag{3.45}$$

For even bits except bit 0, an alternative is used to compute the sum.

$$\begin{aligned} s_i &= t'_i \oplus c'_i \\ &= t_i \oplus g_{i-1} \oplus c'_i \end{aligned} \tag{3.46}$$

To adopt the relation above, the pre-computation will have extra XOR gates. But these gates will not incur extra logic delay since in the critical path, c'_i is expected to arrive later than t'_i, especially when bits go near MSB. Figure 3.27 shows an 8-bit example of parallel-prefix adder incorporating with carry-save notation.

As the final sum is either bit XORing t_i/c_i pair or XORing t'_i/c'_i and these pairs never interfere with each other, the sum computed with interdigitated c_i and c'_i will be always correct. Table 3.5 summarizes selected parallel-prefix trees qualitatively in terms of logic levels, fan-out and wire tracks.

3.6 Summary

The algorithmic construction of parallel-prefix adders are described in detail in this chapter. Simple algorithms and examples are given to build the understanding of parallel-prefix structures. Based on prefix trees, Ling's algorithm has been applied. In the end, a prefix architecture embedded with carry-save notation is proposed.

CHAPTER 4

Modulo $2^n \pm 1$ Adders

Modular arithmetics find applications in a wide range of areas since the operations are the basis in Residue Number Systems (RNS) [17]. Modulo addition/subtraction and multiplication are applied to digital filters [18] in digital signal processing, cryptography [19], error detection and correction [20], as well as checksum computation in high-speed networks [21].

4.1 Modulo $2^n - 1$ Adders

Modulo 2^n-1 addition is one of the most common operations that has been put to hardware implementations because of its circuit efficiency [26]. There are several ways of doing modulo $2^n - 1$ addition. The basic idea is to add the carry-out to the sum as in the fashion of end-around add. The most straight-forward scheme takes two steps, add and increment, which results in more delay but less complex circuit. Carry-select adder can be used for this type of implementation. An alternative way of doing modulo addition is to utilize the carry-lookahead equations and use carry-generates/propagates signals recursively to get the final result in just one step. In this method, modified parallel-prefix structure is employed to achieve the fastest architecture.

Many techniques can help modulo addition [7] [1] [47] [48]. This section reviews the publications about modulo $2^n - 1$ adders and proposes improvement over the current work in publication. The scheme employed here is based on Ling's equations [8] and the methods are extended to modulo $2^n - 1$ addition with parallel-prefix structure. The end-around adder structure is also implemented with NAND adders. For less distraction, only

double representation of zero modulo $2^n - 1$ addition is discussed.

4.1.1 Background

Modulo Operation

Modular arithmetic is to get the remainder of M dividing by A, or to a reduce number A to subtract the modulus M until $A < M$. The operation can be written as

$$A \; mod \; M,$$

or briefed as

$$|A|_M$$

. Modulo adders simply perform addition followed by the modulo operation.

$$S = |A + B|_M. \tag{4.1}$$

Modular arithmetic has the property that

$$S = |A + B|_M = |A|_M + |B|_M. \tag{4.2}$$

For the modulo $2^n - 1$,

$$S = |A + B|_{(2^n-1)}. \tag{4.3}$$

Or

$$S = A + B + c_{out}. \tag{4.4}$$

The equation above can be implemented in hardware by using a end-around adder as mentioned in Figure 4.1.

Review of Binary Addition Basics

The binary addition is the base of modulo addition. Brief review and the key equations are repeated for modulo addition. The major difference at the inputs between binary and

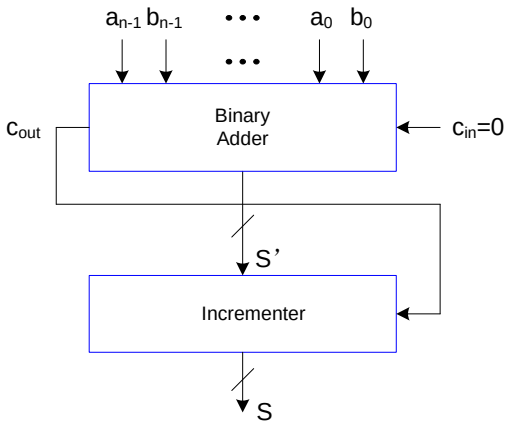


Figure 4.1: General Block Diagram Modulo $2^n - 1$ Adder.

modulo adders is that modulo adders generally do not have carry-in. For binary addition, various techniques can be utilized as mentioned in Chapter 2. The binary add operation with carry-in is to obtain a carry and sum. For a single bit addition, sum and carry can be computed as follows.

$$\begin{aligned} s &= a \oplus b \oplus c_{in} \\ c_{out} &= a \cdot b + b \cdot c_{in} + a \cdot c_{in} \end{aligned} \tag{4.5}$$

For an n-bit addition, $S = A+B+c_{in}$, where $A = \{a_{n-1}, a_{n-2}, ...a_0\}, B = \{b_{n-1}, b_{n-2}, ...b_0\}$. Remember that the relationship between carry and carry generate/propagate is as Equation (4.6) indicates.

$$\begin{aligned} g_k &= a_k \cdot b_k \\ p_k &= a_k + b_k \\ c_{k+1} &= g_k + p_k \cdot c_k \end{aligned} \tag{4.6}$$

Group version of carry generate/propagate and their relation with carry is expressed in Equation (2.71).

Modulo adders can be built using any type of adders. The most efficient ways employ the generate/propagate terms. The cells defined in Figure 4.2 are utilized for modulo

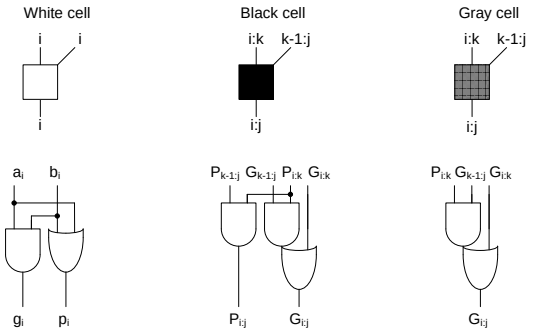


Figure 4.2: Cell Definitions.

adders. The white cell is to compute g_i and p_i.

Implementing Modulo $2^n - 1$ Adders

As mentioned before, the most straight-forward adders (i.e. end-around adders take two steps). In the first step, the sum and carry-out are obtained by adding the inputs with no carry-in. In the second step, the sum is incremented with carry-out. This is a simple way of doing modulo $2^n - 1$ addition. This type of adders can be improved with a $smart$ incrementer.

Parallel-prefix structures can be utilized for modulo $2^n - 1$ adders as well. There are two categories of architectures present in publication. One is called the full parallel-prefix structure [22] and the other is called the reduced parallel-prefix structure [35]. The following presentation on modulo $2^n - 1$ adders starts with parallel-prefix structure and ends with end-around adders.

4.1.2 Full Parallel-Prefix Structure

The first idea to combine the two steps of modulo addition to a single step is proposed in [22]. This is done by inserting the carry-out in the first step into the carry equation to generate a real carry at bit 0. Then traverse rest of the bits. The required cells are defined in Figure 4.2. The white cells are for pre-computation.

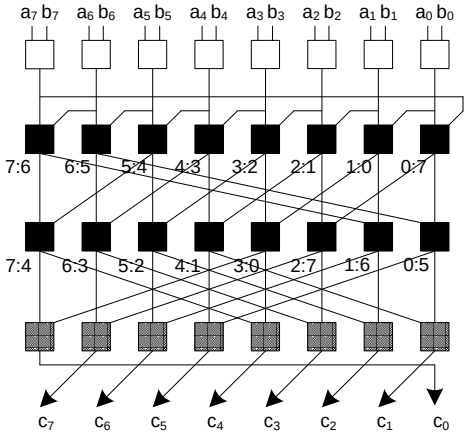


Figure 4.3: Modulo $2^n - 1$ Adder with the Full Parallel-Prefix Structure.

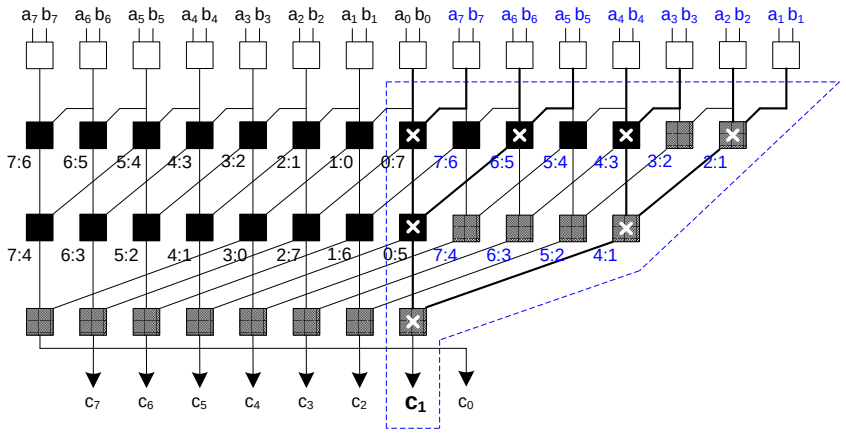


Figure 4.4: Another Way to Look at Modulo $2^n - 1$ Adder with the Full Parallel-Prefix Structure.

In the first step, $c_0 = 0$, therefore,

$$\begin{aligned} c_{out} &= c_{n-1} \\ &= g_{n-1} + p_{n-1} \cdot g_{n-2} + ... + p_{n-1} \cdot p_{n-2} ... p_1 \cdot g_0. \end{aligned} \tag{4.7}$$

In the second round, $c_0 = c_{out}$, Equation (4.6) is used in the following pattern.

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_0 \\ &= g_0 + p_0 \cdot g_{n-1} + p_0 \cdot p_{n-1} \cdot g_{n-2} + ... \\ &\quad + p_0 \cdot p_{n-1} ... p_1 \cdot g_0 \end{aligned} \tag{4.8}$$

The first and the last terms in the equation above can be combined to one term, i.e. $g_0 + p_0 \cdot p_{n-1} ... p_1 \cdot g_0 = g_0$. Thus,

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot g_{n-1} + p_0 \cdot p_{n-1} \cdot g_{n-2} + ... \\ &\quad + p_0 \cdot p_{n-1} \cdot p_{n-2} ... p_2 \cdot g_1 \\ &= g_{|n+0|_n} + p_{|n-1+1|_n} \cdot g_{|n-2+1|_n} + ... \\ &\quad + p_{|n-1+1|_n} \cdot p_{|n-2+1|_n} ... p_{|1+1|_n} \cdot g_{|0+1|_n} \end{aligned} \tag{4.9}$$

The subscripts for p and g start from the current bit, then go around to the next right bit, and so forth. If the current bit is already LSB, subscripts go to MSB until all the subscripts are traversed. The subscripts indicate the end-around add for the modulo adder. From the previous expression, c_1 can be rewritten as in the following equation.

$$c_1 = g_{|n+0|_n} + \sum_{j=0}^{n-2} (\prod_{k=j+1}^{n-1} p_{|k+1|_n}) \cdot g_{|j+1|_n} \tag{4.10}$$

The rest carries can be written as

$$\begin{aligned} c_2 &= g_{|n+1|_n} + \sum_{j=0}^{n-2} (\prod_{k=j+1}^{n-1} p_{|k+2|_n}) \cdot g_{|j+2|_n} \\ &\quad ... \\ c_{n-1} &= g_{|n+n-2|_n} + \sum_{j=0}^{n-2} (\prod_{k=j+1}^{n-1} p_{|k+n-1|_n}) \cdot g_{|j+n-1|_n} \end{aligned} \tag{4.11}$$

In general, the carries are expressed as follows

$$c_{i+1} = g_{|n+i|_n} + \sum_{j=0}^{n-2}(\prod_{k=j+1}^{n-1} p_{|k+i+1|_n}) \cdot g_{|j+i+1|_n} \tag{4.12}$$

where $0 \leq i \leq n-2$.

As the equations indicate above, the modulo $2^n - 1$ addition can be completed in a single step. This idea was originally applied to Carry-Lookahead Adder (CLA). It is easy to adapt the idea to fast parallel-prefix structures since they share the same equations [23]. For example, a 4-bit modulo $2^n - 1$ adder utilizes the following group generates/propagates to get the c_{out}.

$$\begin{aligned} c_{out} &= c_4 \\ &= G_{3:2} + P_{3:2} \cdot G_{1:0}. \end{aligned} \tag{4.13}$$

Then c_1 is computed as

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_4 \\ &= g_0 + p_0 \cdot G_{3:2} + p_0 \cdot P_{3:2} \cdot G_{1:0} \\ &= g_0 + p_0 \cdot g_3 + p_3 \cdot p_0 \cdot g_2 + p_3 \cdot p_2 \cdot p_0 \cdot g_1 \\ &= G_{0:3} + P_{0:3} \cdot G_{2:1} \end{aligned} \tag{4.14}$$

where $G_{0:3} = g_0 + p_0 \cdot g_3, P_{0:3} = p_0 \cdot p_3$. This indicates the exact idea of an end-around addition. Similarly, c_2 and c_3 can be derived as

$$c_2 = G_{1:0} + P_{1:0} \cdot G_{3:2} \tag{4.15}$$

$$c_3 = G_{2:1} + P_{2:1} \cdot G_{0:3}. \tag{4.16}$$

An 8-bit full parallel-prefix structure is shown in Figure 4.3. The structure has more black cells and wires when compared to regular parallel-prefix structures. A Sklansky [12] Parallel-Prefix Tree is shown in Figure 2.13.

Figure 4.4 shows another way of looking at 8-bit modulo $2^n - 1$ addition. With duplicated white cells, the wires now are always feeding forward while in Figure 4.3, some

wires are going backward. According to Figure 4.4, computation of each carry requires all of the input bits. For example, c_0 takes all the bits $a_0\ b_0, a_7\ b_7, ...a_1\ b_1$. In the triangle, the cells used to generate c_1 are marked with an 'x' in them and wires are highlighted with bold lines.

4.1.3 Reduced Parallel-Prefix Structure

The full parallel-prefix structure computes modulo addition with a small number of stages. However, it has the disadvantages of dense gates and congestive wire tracks. It is important to understand that even though there are many different parallel-prefix algorithms, not every architecture can utilize the algorithm presented above.

Zimmermann [35] presented the idea of being capable of utilizing any type of parallel-prefix tree, by slightly changing the prefix tree and adding one logic level at the bottom of the tree. Figure 4.5 illustrates the block diagram for prefix tree stage. An 8-bit modified Sklansky prefix tree is shown as an example in Figure 4.6, which can be inserted into the box of Figure 4.5. Using a parallel-prefix tree, a new carry block c'_i $(0 < i < n)$ can be derived. Then, c_{out} can be inserted backward as c_0 to complete the carry computation.

$$c_{i+1} = G_{i:0} + P_{i:0} \cdot c_{out} \tag{4.17}$$

The final row of gray cells in Figure 4.5 implements the equation above.

The modified prefix tree can be viewed as one-bit right shifted (towards LSB) version of regular prefix tree since there is no carry input. The general objective here is to reduce the density of gates and wires with the flexibility of choosing any type of desired prefix tree. The cost is one extra level of logic and large fan-out at this level.

4.1.4 Parallel-Prefix Ling Structures for Modulo $2^n - 1$ Adders

The original end-around addition was proposed for performing modulo $2^n - 1$ adders. A carry-select incrementer (CSEI) [27] can be such a solution, where a conventional adder is

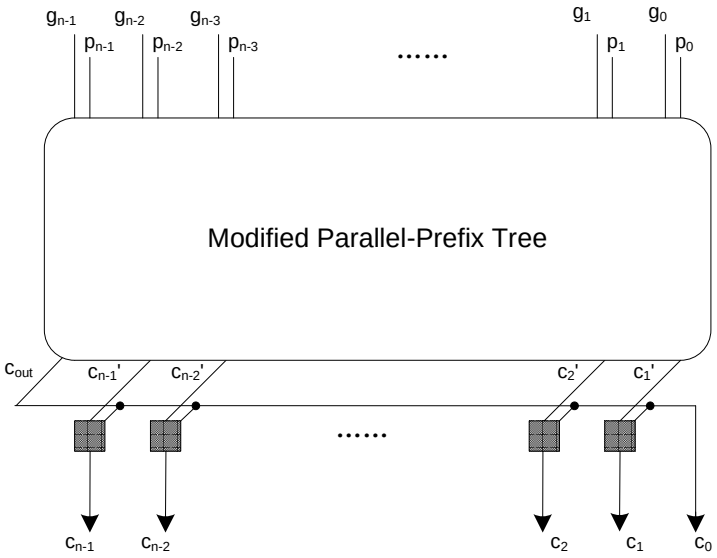


Figure 4.5: Modulo $2^n - 1$ Adder with the Reduced Parallel-Prefix Structure.

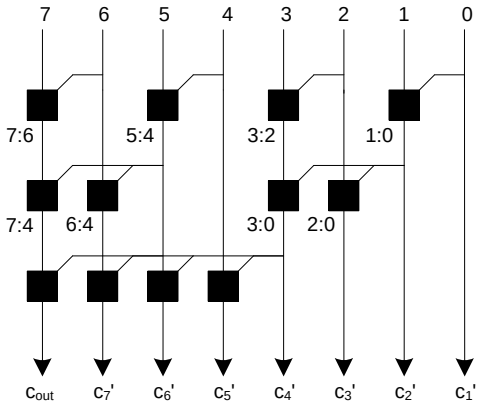


Figure 4.6: The Reduced Sklansky Parallel-Prefix Tree.

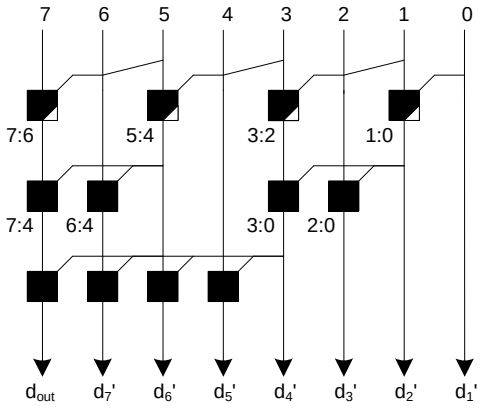


Figure 4.7: 8-bit Sklansky with Ling's Scheme for Modulo $2^n - 1$ Adders.

followed by an incrementer implemented by a tree of AND gates and a set of multiplexors. Taking the effort of two steps, the resulted implementation is slower, and consumes more area and power than parallel-prefix structures.

To build a parallel-prefix Ling modulo $2^n - 1$ adder, starting from pre-computation stage, carry-generate/propagate are required as indicated in Equation (4.6). In the parallel-prefix tree stage, unlike regular parallel-prefix adder, Ling adders have their own group generate/propagate terms H and I. This requires a redefinition of cells to build parallel-prefix Ling adders. Figure 3.20 shows the new cell definitions. Starting from a bit span of one, the equations in Chapter 3 are modified slightly to indicate the difference where carry-in is absent for modulo adders.

$$\begin{aligned} H_{i+1:i} &= g_{i+1} + g_i \\ I_{i+1:i} &= p_i \cdot p_{i-1} \end{aligned} \tag{4.18}$$

At this level, the reduced cells in Figure 3.20 are used. Similar to binary prefix trees, for $i = 0$, $i - 1 = -1$ which is a non-existent bit, which corresponds to a not used term $I_{1:0}$. An example is shown in Figure 4.7, where an 8-bit Sklansky prefix tree with Ling's scheme is illustrated. The tilted lines in the first logic level indicate the Ling's implementation of Equation 4.18.

The recurrence terms H/I have been discussed as (Equation 3.22) in Chapter 3. The

only difference is that there is no carry for modulo adders, therefore, Ling's prefix tree produces pseudo-carry $H_{i:0}$ or d_{i+1}.

Like the regular reduced parallel-prefix structure, Ling's scheme also requires an extra logic level to turn d'_i to d_i. In regular parallel-prefix structure, c_{out} needs to be inserted as c_0, as Equation (4.17) does. The operation required for Ling's structure is to get c_{out} from pseudo-carry d_{out}.

$$\begin{aligned} c_{out} &= p_{n-1} \cdot d'_n, \\ &= p_{n-1} \cdot d_{out}. \end{aligned} \tag{4.19}$$

The parallel-prefix tree stage is shown in Figure 4.8 where d_{out} is ANDed with p_{n-1} to generate c_{out} and this term is inserted as d_0 at LSB. The last logic level computes the pseudo-carry at each bit using the following equation.

$$d_{i+1} = H_{i:0} + I_{i:0} \cdot d_0 \tag{4.20}$$

If both sides of Equation (4.20) are ANDed with p_i, the equation will become exactly the same as Equation (4.17).

Ling's scheme also makes the final post-computation stage unique by computing the sum as

$$s_i = p_i \oplus d_{i+1} + g_i \cdot p_{i-1} \cdot d_i. \tag{4.21}$$

In the case of $i = 0$, $s_0 = p_0 \oplus d_1 + g_0 \cdot d_0$. Equation (4.21) is an alternative representation of Equation (3.42) in terms of computing the sum as it has been proven in Chapter 2.

4.1.5 Carry-Select Incrementer

General Implementation

Parallel-prefix structure is fast, but end-around add is still the simplest way of doing modulo $2^n - 1$ addition. This method offers an alternative to compute the add instead of using the

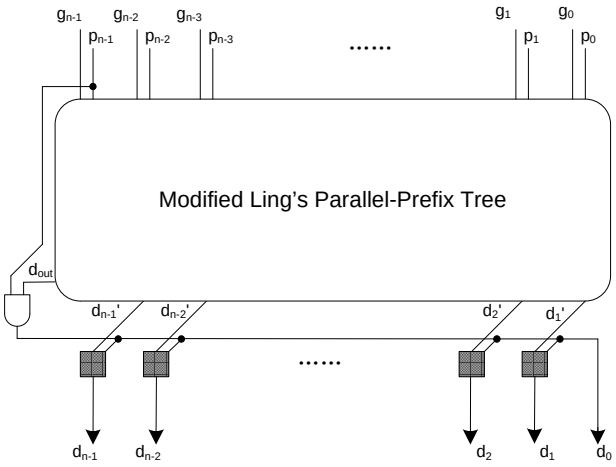


Figure 4.8: Modulo $2^n - 1$ Adder with the Reduced Parallel-Prefix Ling Structure.

congestive full parallel-prefix structure. A carry-select incrementer (CSEI) can be a good candidate for such implementation [27].

Since the carry-out can either be 1 or 0, CSEI can be employed to choose the S or $S + 1$ using multiplexors (MUX). An incrementer is supposed to increment the input by 1, i.e. $S = A + 1$. In the modulo addition, it is required such that $S = A + c_{out}$, where $A = \{a_{n-1}, a_{n-2}, ...a_1, a_0\}$. Since c_{out} only has the LSB that is either 1 or 0, the following relationship can be utilized.

For $0 \leq i \leq n - 1$,

$$\begin{aligned} g_i &= a_i \cdot b_i = 0 \\ p_i &= a_i + b_i = a_i \\ c_{i+1} &= g_i + p_i \cdot c_i = a_i \cdot a_{i-1} \cdot ...a_0 \cdot c_{out} \end{aligned} \tag{4.22}$$

The critical path of the incrementer logic is the path to generate the most significant bit of carry for the MUX to choose the modulo sum, $c_{n-1} = a_{n-2} \cdot a_{n-3} \cdot ...a_0 \cdot c_{out}$. This can be implemented with $\lceil log_m(n-1) \rceil$ logic levels if only m-input AND gates are used. For example, assuming only 2-input AND gates are available, a 32-bit incrementer requires $\lceil log_2(31) \rceil = 5$ levels of logic.

Besides carry, the inputs to the MUX are simply the first stage sum bit S' and its com-

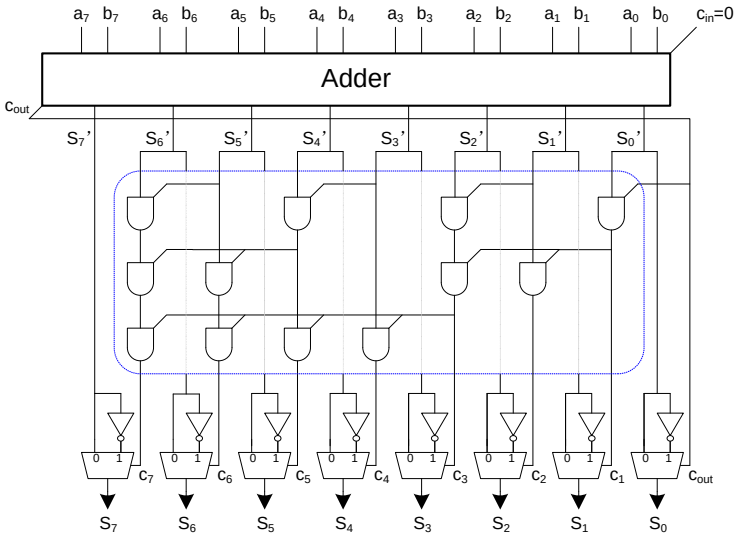


Figure 4.9: 8-bit Carry-Select Modulo $2^n - 1$ Adder.

plement. An 8-bit example of this type of modulo $2^n - 1$ adders is shown in Figure 4.9. The adder at the top of the figure can be of any type, including the parallel-prefix trees. The increment logic is in the rounded rectangle and its structure is similar to parallel-prefix trees. There is no large fan-out from the c_{out} as can be observed from the figure.

Carry-Select Incrementer with NAND Adder

Ling's equations not only can help to improve parallel-prefix tree, they can also enhance ripple-carry adders. This type of adders are also called NAND or NOR adders [39]. In CMOS logic, NAND gates are usually faster than NOR gates while NOR adders are more suitable for domino-logic. Here only NAND adders are introduced for modulo $2^n - 1$ addition since the circuit will be implemented with CMOS standard-cell library, where in CMOS circuits, NAND gates are usually faster then NOR gates because of less slow PMOS in series.

Recall from Chapter 2 that the NAND equations are

$$\begin{aligned} \overline{d_{i+1}} &= \hat{p}_i \cdot \hat{p}_{i-1} \cdot e_i \\ \overline{e_{i+1}} &= p_i \cdot p_{i-1} \cdot d_i \end{aligned} \tag{4.23}$$

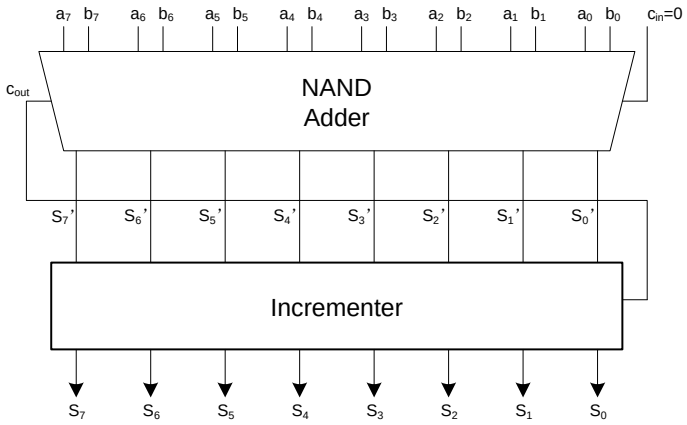


Figure 4.10: CSEI with NAND Adder.

or

$$\begin{aligned} d_{i+1} &= \overline{\hat{p}_{i:i-1} \cdot e_i} \\ e_{i+1} &= \overline{p_{i:i-1} \cdot d_i} \end{aligned} \tag{4.24}$$

The modulo adder is constructed as the end-around add, i.e. normal addition followed by an incrementer. Grad has proven that this type of adders are much faster than traditional ripple-carry adders [40] due to the reduction form AOI gates to NAND gates. An 8-bit example of modulo $2^n - 1$ adder using this type of scheme is shown in Figure 4.10.

4.1.6 Summary

For modulo $2^n - 1$ adders with prefix tree implementation, since every carry-out bit takes the full effort of the all logic levels, it makes sense to have higher fan-out, i.e. radix greater than 2, to compute the carries. Higher fan-out structure can utilize Naffziger's [49] method, which is an improved fan-out of 4 adder based on Ling's idea.

Radix 2 results, which can be compared with binary parallel-prefix adders, will be presented in Chapter 5. Various implementations with prefix trees, including the full and the reduced parallel-prefix trees, and carry-select incrementer with NAND Adders are compared in the following chapter.

4.2 Modulo $2^n + 1$ Adders

4.2.1 Background

Modulo $2^n + 1$ adders are very similar to modulo $2^n - 1$ adders. As mentioned in the previous chapter, modulo $2^n + 1$ add can be expressed as follows.

$$S = |A + B|_{2^n+1}. \tag{4.25}$$

The diminished-one system is used here where a number A is represented as $A' = A-1$ and the value 0 is not used. This type of addition has the following relationships.

$$\begin{aligned} A + B &= S \\ (A' + 1) + (B' + 1) &= S' + 1 \\ A' + B' + 1 &= S' \end{aligned} \tag{4.26}$$

Modulo $2^n + 1$ addition is formulated as

$$|A' + B' + 1|_{2^n+1} = \begin{cases} A' + B' + 1 - (2^n + 1) & \text{if } A' + B' + 1 \geq 2^n + 1 \\ A' + B' + 1 & \text{otherwise} \end{cases} \tag{4.27}$$

Equation (4.27) can be reduced to the following expression, which is easy to implement in digital logic.

$$|A' + B' + 1|_{2^n+1} = \begin{cases} |A' + B'|_{2^n} & \text{if } A' + B' \geq 2^n \\ A' + B' + 1 & \text{otherwise} \end{cases} \tag{4.28}$$

This can be written in the expression similar to modulo $2^n - 1$ adders.

$$S' = A' + B' + \overline{c_{out}}. \tag{4.29}$$

That is, when $A'+B' < 2^n$, the sum of $A'+B'$ is incremented. Otherwise, the sum remains as it is. The equation above can be adapted to compute modulo $2^n + 1$ add in hardware by using a end-around adder as illustrated in Figure 4.11.

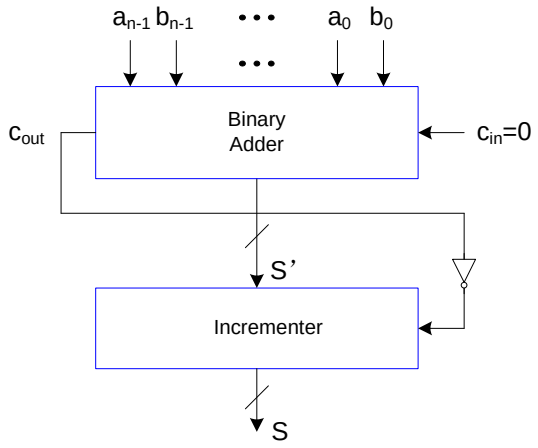


Figure 4.11: General Block Diagram Modulo $2^n + 1$ Adder.

It can be observed that in Figure 4.11 the only difference between modulo $2^n + 1$ adder and modulo $2^n - 1$ adder is the inverter that takes c_{out} as input. In this end-around adder, c_{out} needs to be inverted before going to the incrementer.

The ways of building a modulo $2^n + 1$ can also be divided into three categories. One utilizes the reduced parallel prefix tree with an extra logic level at the bottom [35]. A second method uses the similar idea as the full parallel prefix tree. The third one is the end-around adder with any type of adder followed by an incrementer.

4.2.2 Reduced Parallel-Prefix Structure

To compute modulo 2^n+1 add, parallel-prefix structure has to be modified for the operation. There is no easy way of doing modulo $2^n + 1$ add with a full parallel-prefix structure as that for modulo $2^n - 1$ add. One efficient way of doing this type of add is to employ the structures of Zimmermann, i.e. having an extra level at the end of a prefix tree to compute the exact carry for each bit. Then the carries go ahead for the final sum computation.

The carries in the modulo $2^n + 1$ adder can be generalized as

$$c_i = \begin{cases} \overline{G_{n-1:0}} & \text{if } i = 0 \\ G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:0}} & 1 \leq i \leq n-1 \end{cases} \tag{4.30}$$

For example, an 8-bit modulo $2^n + 1$ adder computes carries as the following equations indicate.

$$c_8 = G_{7:0}. \tag{4.31}$$

The c_0 through c_7 will base on c_8 or $\overline{c}_8$.

$$\begin{aligned}
c_0 &= \overline{G_{7:0}} = \overline{c_8} \\
c_1 &= g_0 + p_0 \cdot \overline{G_{7:0}} \\
c_2 &= G_{1:0} + P_{1:0} \cdot \overline{G_{7:0}} \\
c_3 &= G_{2:0} + P_{2:0} \cdot \overline{G_{7:0}} \\
c_4 &= G_{3:0} + P_{3:0} \cdot \overline{G_{7:0}} \\
c_5 &= G_{4:0} + P_{4:0} \cdot \overline{G_{7:0}} \\
c_6 &= G_{5:0} + P_{5:0} \cdot \overline{G_{7:0}} \\
c_7 &= G_{6:0} + P_{6:0} \cdot \overline{G_{7:0}}
\end{aligned} \tag{4.32}$$

The general form is simply expressed as

$$c_{i+1} = G_{i:0} + P_{i:0} \cdot \overline{c_{out}}. \tag{4.33}$$

There is no bit -1 since no carry-in is required. Figure 4.12 shows the block diagram of Zimmermann's idea of diminished-one modulo $2^n + 1$ adder. The number of logic levels needed for this type of modulo $2^n + 1$ adder is $log_2(n) + 1$ assuming the radix is 2.

Similar to modulo $2^n - 1$ adder, there is no congested wires or dense gates compared to a full parallel prefix version. Any type of parallel-prefix tree can be used. The modification is the same as it is required for modulo $2^n - 1$ adders. The cost is the extra level of gray cells. However, such effort is not very obvious since a full parallel-prefix tree implementation is far more complex than that of modulo $2^n - 1$ adders.

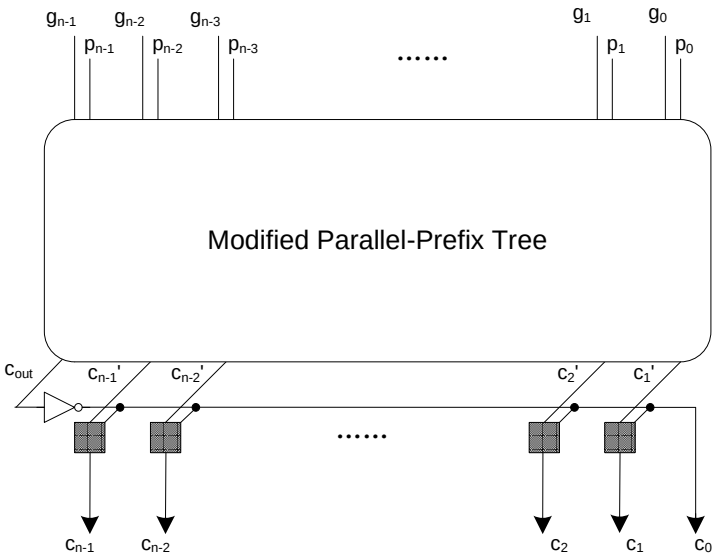


Figure 4.12: Modulo $2^n + 1$ Adder with the Reduced Parallel-Prefix Structure.

4.2.3 Parallel-Prefix Ling Structures for Modulo $2^n + 1$ Adders

Ling's scheme can be applied to modulo $2^n + 1$ adders in the similar way as the reduced parallel-prefix structures for modulo $2^n - 1$ adders. The idea is applicable for full parallel-prefix structure for modulo $2^n + 1$ adders [50]. However, due to the complexity of the full parallel-prefix structure, the benefit tend to diminish when Ling's equations are utilized, especially for wide adders (i.e. 64-bit or larger). As there is an inverted carry-in for modulo $2^n + 1$ adders, the same logic will be there in Ling's reduced prefix tree, i.e.

$$d_{i+1} = H_{i:0} + \overline{I_{i:0} \cdot d_0} \tag{4.34}$$

Figure 4.13 shows the block diagram. A quick observation can be made that the only difference between this prefix tree and the one for modulo $2^n - 1$ adders is that the inverter at the lower left, where carry-out is generated.

4.2.4 Full Parallel-Prefix Structure

Diminished-one modulo $2^n + 1$ adder can be implemented with the least logic levels (i.e. $log_2(n)$ levels). The scheme was proposed by Vergos [51] et al. The idea is to modify

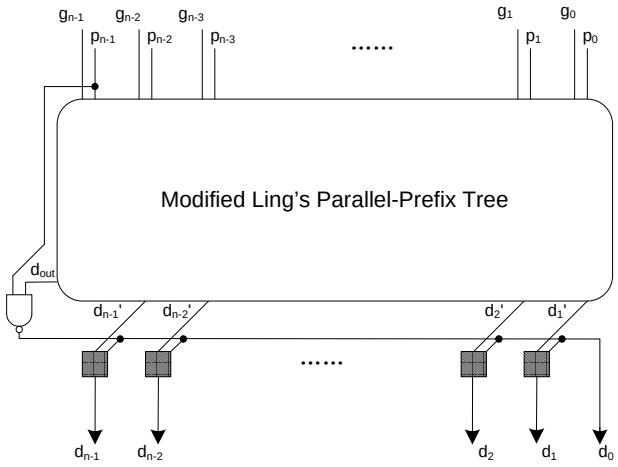


Figure 4.13: Modulo $2^n + 1$ Adder with the Reduced Parallel-Prefix Ling Structure.

the Equation (4.30) utilizing the properties of group carry-generate/propagate in order to eliminate the extra logic level.

To ease the explanation of the scheme, the symbol " $\circ$ " mentioned in Chapter 3 is addressed, where the symbol is used with (G, P) pairs.

$$(G_{i:k}, P_{i:k}) = (G_{i:j}, P_{i:j}) \circ (G_{j-1:k}, P_{j-1:k}). \tag{4.35}$$

Or $G_{i:j}$ and $P_{i:j}$ can be expressed separately as follows.

$$\begin{aligned} G_{i:k} &= (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ ... \circ (g_k, p_k) \\ P_{i:k} &= p_i \cdot p_{i-1} \cdot ... \cdot p_k \end{aligned} \tag{4.36}$$

To simplify Equation (4.30), two theorems have to be applied.

Theorem 1. $c_i = G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:i}}$, for $1 \leq i \leq n-1$.

Proof. From (4.30)

$$\begin{aligned} c_i &= G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:0}} \\ &= G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:i} + P_{n-1:i} \cdot G_{i-1:0}} \\ &= G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:i}} \cdot (\overline{P_{n-1:i}} + \overline{G_{i-1:0}}) \\ &= G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:i}} \cdot \overline{P_{n-1:i}} + P_{i:0} \cdot \overline{G_{n-1:i}} \cdot \overline{G_{i-1:0}} \end{aligned} \tag{4.37}$$

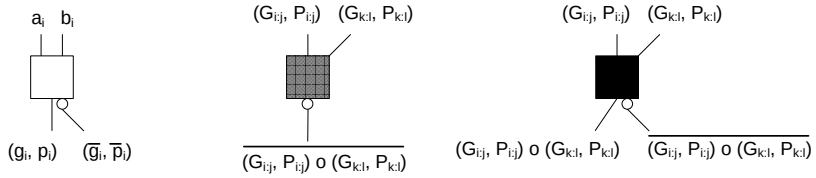


Figure 4.14: Additional Cell Definitions.

If $G_{i-1:0} = 1, c_i = 1$;

If $G_{i-1:0} = 0$,

$$\begin{aligned} c_i &= P_{i-1:0} \cdot \overline{G_{n-1:i}} \cdot \overline{P_{n-1:i}} + P_{i:0} \cdot \overline{G_{n-1:i}} \cdot 1 \\ &= P_{i-1:0} \cdot \overline{G_{n-1:i}} \end{aligned} \tag{4.38}$$

Considering both of the above cases of $G_{i-1:0}$, the following relationship can be established.

$$c_i = G_{i-1:0} + P_{i-1:0} \cdot \overline{G_{n-1:i}}. \tag{4.39}$$

■

According to **Theorem 1** and using the $\circ$ operation, an 8-bit diminished-one modulo $2^n + 1$ add in (4.32) becomes

$$\begin{aligned}
c_0 &= \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0)} \\
c_1 &= (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1)} \\
c_2 &= (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2)} \\
c_3 &= (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3)} \\
c_4 &= (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4)} \\
c_5 &= (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6) \circ (g_5,p_5)} \\
c_6 &= (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7) \circ (g_6,p_6)} \\
c_7 &= (g_6,p_6) \circ (g_5,p_5) \circ (g_4,p_4) \circ (g_3,p_3) \circ (g_2,p_2) \circ (g_1,p_1) \circ (g_0,p_0) \circ \overline{(g_7,p_7)}
\end{aligned} \tag{4.40}$$

Equation (4.40) cannot be implemented with the minimum logic levels (i.e. $log_2(8) = 3$ logic levels). For example, in computing c_1, it takes at least 3 ($\lceil log_2(7) \rceil$) logic levels to compute $\overline{(g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1)}$. To do another $\circ$ operation with (g_0, p_0), one more level has to be included. Computing c_2 has the same problem. Only c_4 has no such problem because both the non-inverting and inverting logic have 4 terms which can be computed within 2 logic levels and c_4 can be derived with one more level.

To illustrate the full parallel structure for modulo $2^n + 1$, group version of $\hat{g}$ and $\hat{p}$ defined in Equation (2.53), $\hat{G}$ and $\hat{P}$ are introduced. Similar to Equation (4.36), there is relationship as the following equations illustrate.

$$\begin{aligned} \hat{G}_{i:j} &= (\hat{g}_i, \hat{p}_i) \circ (\hat{g}_{i-1}, \hat{p}_{i-1}) \circ ... \circ (\hat{g}_j, \hat{p}_j) \\ \hat{P}_{i:k} &= \hat{p}_i \cdot \hat{p}_{i-1} \cdot ... \cdot \hat{p}_j \end{aligned} \tag{4.41}$$

Another theorem needs to be applied in order to compute modulo $2^n + 1$ addition with the minimum logic levels.

Theorem 2. $(G_{i:j}, P_{i:j}) \circ \overline{(G_{k:l}, P_{k:l})} = \overline{(\hat{G}_{i:j}, \hat{P}_{i:j}) \circ (G_{k:l}, P_{k:l})}$.

Proof.

On the left side of the equation, the following relationship can be observed.

$$(G_{i:j}, P_{i:j}) \circ \overline{(G_{k:l}, P_{k:l})} = G_{i:j} + P_{i:j} \cdot \overline{G_{k:l}} \tag{4.42}$$

On the right of the equation, the expression can be deduced as follows.

$$\begin{aligned} \overline{(\hat{G}_{i:j}, \hat{P}_{i:j}) \circ (G_{k:l}, P_{k:l})} &= \overline{\hat{G}_{i:j} + \hat{P}_{i:j} \cdot G_{k:l}} \\ &= \overline{\hat{G}_{i:j}} \cdot (\overline{\hat{P}_{i:j}} + \overline{G_{k:l}}) \end{aligned} \tag{4.43}$$

Since

$$\begin{aligned}
\overline{\hat{G}_{i:j}} &= \overline{\hat{g}_i + \hat{p}_i \cdot (\hat{g}_{i-1} + ... + \hat{p}_{j+2} \cdot (\hat{g}_{j+1} + \hat{p}_{j+1} \cdot \hat{g}_j)...)} \\
&= \overline{\overline{p_i} + \overline{g_i} \cdot (\overline{p_{i-1}} + ... + \overline{g_{j+2}} \cdot (\overline{p_{j+1}} + \overline{g_{j+1}} \cdot \overline{p_j})...)} \\
&= p_i \cdot (g_i + p_{i-1} \cdot (g_{i-1} + ... \cdot (g_{j+2} + p_{j+1} \cdot (g_{j+1} + p_j))...) \\
&= g_i + p_i \cdot g_{i-1} + P_{i:i-1} \cdot g_{i-2} + ... + P_{i:j+2} \cdot g_{j+1} + P_{i:j} \\
&= G_{i:j+1} + P_{i:j} \qquad (4.44)
\end{aligned}$$

and

$$\begin{aligned}
\overline{\hat{P}_{i:j}} &= \overline{\overline{g_i} \cdot \overline{g_{i-1}} \cdot ... \cdot \overline{g_{j+1}} \cdot \overline{g_j}} \\
&= g_i + g_{i-1} + ... + g_{j+1} + g_j \qquad (4.45)
\end{aligned}$$

The right side of Equation (4.43) can now be expanded as

$$\begin{aligned}
\overline{\hat{G}_{i:j}} \cdot (\overline{\hat{P}_{i:j}} + \overline{G_{k:l}}) &= (G_{i:j+1} + P_{i:j}) \cdot (\overline{\hat{P}_{i:j}} + \overline{G_{k:l}}) \\
&= G_{i:j+1} \cdot \overline{\hat{P}_{i:j}} + P_{i:j} \cdot \overline{\hat{P}_{i:j}} + G_{i:j+1} \cdot \overline{G_{k:l}} + P_{i:j} \cdot \overline{G_{k:l}} \qquad (4.46)
\end{aligned}$$

And, simplifying the first term of Equation (4.46) using the property that $g_k + g_k \cdot X = g_k$, X can be any term or multiple terms.

$$\begin{aligned}
G_{i:j+1} \cdot \overline{\hat{P}_{i:j}} &= (g_i + p_i \cdot g_{i-1} + ... + P_{i:j+2} \cdot g_{i+1}) \cdot (g_i + g_{i-1} + ... + g_j) \\
&= g_i + p_i \cdot g_{i-1} + ... + P_{i:j+2} \cdot g_{i+1} + (g_i + p_i \cdot g_{i-1} + ... + P_{i:j+2} \cdot g_{i+1}) \cdot g_j \\
&= g_i + p_i \cdot g_{i-1} + P_{i:i-1} \cdot g_{i-1} ... + P_{i:j+2} \cdot g_{i+1} \\
&= G_{i:j+1} \qquad (4.47)
\end{aligned}$$

Inserting the result above, right side of Equation (4.46) becomes

$$\begin{aligned}
&= G_{i:j+1} + P_{i:j} \cdot \overline{\hat{P}_{i:j}} + P_{i:j} \cdot \overline{G_{k:l}} \\
&= g_i + p_i \cdot g_{i-1} + ... + P_{i:j+2} \cdot g_{i+1} + P_{i:j} \cdot (g_i + g_{i-1} + ... + g_j) + P_{i:j} \cdot \overline{G_{k:l}} \qquad (4.48)
\end{aligned}$$

Some of the terms in the equation above can be absorbed.

$$\begin{aligned}
g_i + P_{i:j} \cdot g_i &= g_i \\
p_i \cdot g_i + P_{i:j} \cdot g_{i-1} &= p_i \cdot g_{i-1} \cdot (1 + P_{i-1:j}) = p_i \cdot g_{i-1} \\
P_{i:i-1} \cdot g_{i-2} + P_{i:j} \cdot g_{i-2} &= P_{i:i-1} \cdot g_{i-2} \cdot (1 + P_{i-2:j}) = p_i \cdot g_{i-1} \\
&\ldots \\
P_{i:j+2} \cdot g_{j+1} + P_{i:j} \cdot g_{j+1} &= P_{i:j+2} \cdot g_{i+1} \cdot (1 + P_{j+1:j}) = P_{i:j+2} \cdot g_{j+1} \qquad (4.49)
\end{aligned}$$

Equation (4.46) is reduced as

$$\begin{aligned}
&= g_i + p_i \cdot g_{i-1} + \ldots + P_{i:j+2} \cdot g_{i+1} + P_{i:j} \cdot g_j + P_{i:j} \cdot \overline{G_{k:l}} \\
&= G_{i:j} + P_{i:j} \cdot \overline{G_{k:l}} \qquad (4.50)
\end{aligned}$$

■

By applying **Theorem 2**, Equation (4.32) for the 8-bit example can be transformed as

$$\begin{aligned}
c_0 &= \overline{(g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0)} \\
c_1 &= \overline{(\overline{p_0}, \overline{g_0}) \circ (g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1)} \\
c_2 &= \overline{(\overline{p_1}, \overline{g_1}) \circ (\overline{p_0}, \overline{g_0}) \circ (g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2)} \\
c_3 &= \overline{(\overline{p_2}, \overline{g_2}) \circ (\overline{p_1}, \overline{g_1}) \circ (\overline{p_0}, \overline{g_0}) \circ (g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3)} \\
c_4 &= (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0) \circ \overline{(g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4)} \\
c_5 &= (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1) \circ \overline{(\overline{p_0}, \overline{g_0}) \circ (g_7, p_7) \circ (g_6, p_6) \circ (g_5, p_5)} \\
c_6 &= (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ \overline{(\overline{p_1}, \overline{g_1}) \circ (\overline{p_0}, \overline{g_0}) \circ (g_7, p_7) \circ (g_6, p_6)} \\
c_7 &= (g_6, p_6) \circ (g_5, p_5) \circ (g_4, p_4) \circ (g_3, p_3) \circ \overline{(\overline{p_2}, \overline{g_2}) \circ (\overline{p_1}, \overline{g_1}) \circ (\overline{p_0}, \overline{g_0}) \circ (g_7, p_7)}
\end{aligned} \qquad (4.51)$$

Using the Equation (4.51), the 8-bit modulo $2^n + 1$ adder can be constructed with the full parallel-prefix structure as shown in Figure 4.15.

As Figure 4.15 shows, the adder has the minimum logic levels of $log_2 8 = 3$. The structure is more complicated than its modulo $2^n - 1$ counterpart. Each bit follows Equation

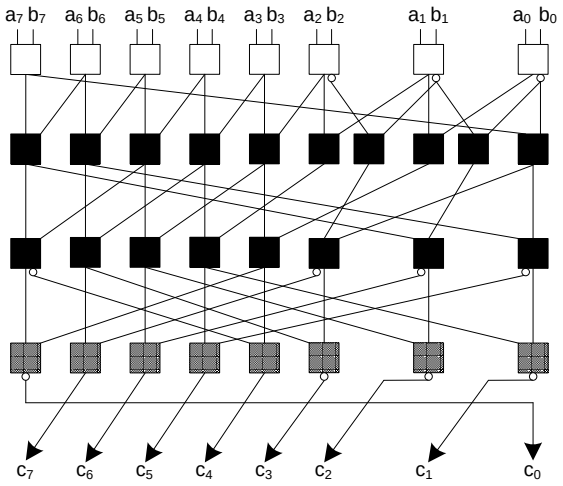


Figure 4.15: Modulo $2^n + 1$ Adder with the Full Parallel-Prefix Structure.

(4.51). Take c_1 for example, in level 1, the top-right black cell unites $(\overline{p_0}, \overline{g_0})$ and (g_7, p_7). In level 2, the black cell unites the result from above and $(g_6, p_6) \circ (g_5, p_5)$. In level 3, the gray cell unites the previous result with $(g_4, p_4) \circ (g_3, p_3) \circ (g_2, p_2) \circ (g_1, p_1)$ and then inverts the output. This is exactly the same c_1 as described in Equation (4.51).

4.2.5 Carry-Select Incrementer

Modulo $2^n + 1$ adder can be implemented using end-around add as described in a similar fashion of modulo $2^n - 1$ adder. As Equation (4.29) indicates, the end-around add for modulo $2^n + 1$ adder only needs an inverter at the c_{out} of the first adder. Figure 4.16 shows an 8-bit example of modulo $2^n + 1$ adder implemented with a Carry-Select Incrementer. Compared to modulo $2^n - 1$ adder, the bold inverter near top-right is the only extra logic that is required for the Carry-Select Incrementer.

4.3 Combination of Binary and Modulo $2^n \pm 1$ Adder

Reviewing binary and modulo $2^n \pm 1$ adder architectures, it can be found that the prefix tree can be applied to all these adders. Modulo adders are an extension of binary adders. The

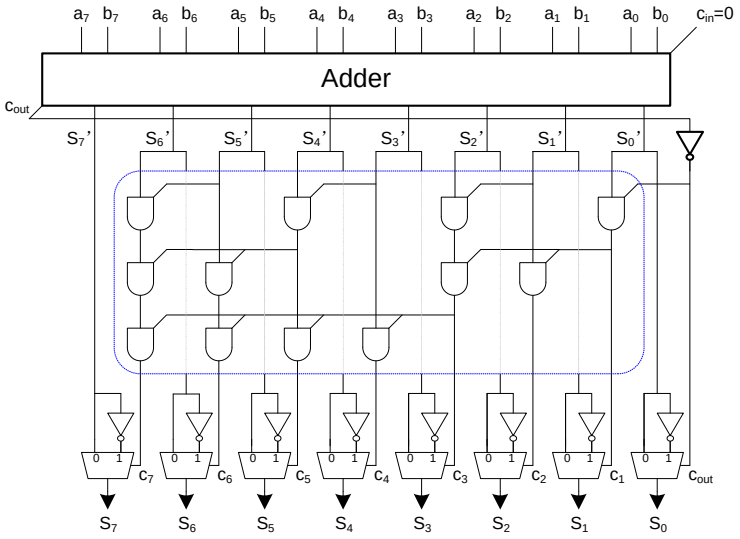


Figure 4.16: 8-bit Carry-Select Modulo $2^n + 1$ Adder.

reduced parallel-prefix structure applies to both modulo $2^n - 1$ and $2^n + 1$ adders, however, with the only difference of one inverter, as shown in Figure 4.5 and 4.12. If the input of the last gray cell row in Figure 4.5 comes from the c_{in} instead of c_{out}, the structure functions exactly as a binary adder does. Controlling the carry input to the last gray cell row with a multiplexor can accommodate the functions of binary and modulo $2^n \pm 1$ add. Figure 4.17 shows the prefix tree part of such adders.

In Figure 4.17, g_i/p_i comes from the pre-computation stage. The selections 0, 1 and 2 in the multiplexor are for modulo $2^n + 1$, modulo $2^n - 1$ and binary addition, respectively. Utilizing Equation (4.4) and (4.29) and the general equation for binary addition, the combined-function adder can be formulized as the following equation shows.

$$S = \begin{cases} A+B+\overline{c_{out}} & \text{modulo } 2^n + 1 \text{ add} \\ A+B+c_{out} & \text{modulo } 2^n - 1 \text{ add} \\ A+B+c_{in} & \text{binary add} \end{cases} \tag{4.52}$$

Inserting this structure between pre- and post-computation, the adder architecture is complete. The modified parallel-prefix tree does not handle the carry-input. This is the only difference between this special prefix tree and that solely for binary adder. The carry input

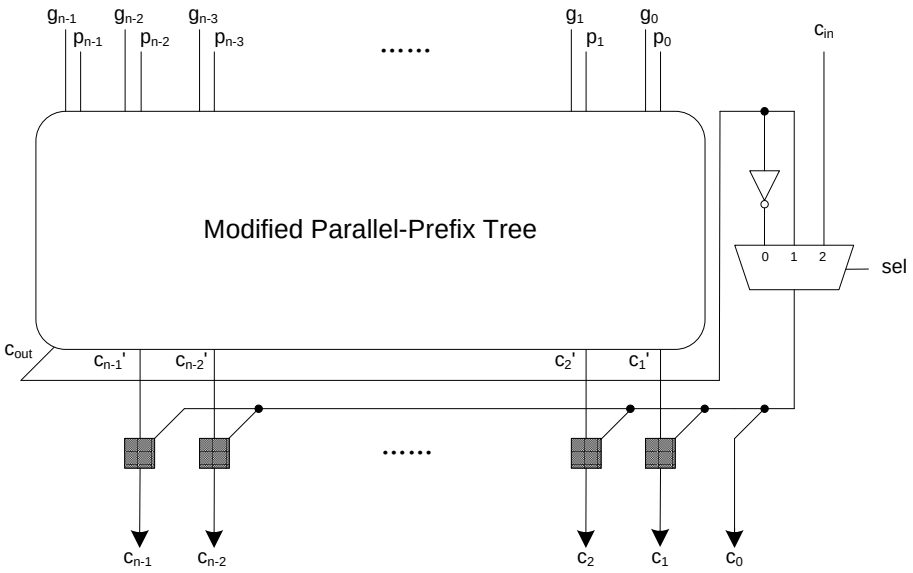


Figure 4.17: Combined Binary and Modulo $2^n \pm 1$ Adders.

is handled at the last row of gray cells. This agrees with the associativity of the synthesis rule. The prefix tree can be modified from any type of normal binary prefix tree, e.g. Brent-Kung.

To expedite the carry-chain, Ling's scheme can be employed. The architecture mentioned above can be easily modified using Ling's equation as shown in Figure 4.18, which is similar to Figure 4.17. For modulo addition, the carry input is from the carry output as described previously. The difference is that the pseudo carries replace the regular carries at the output of the rectangle box and the final row of gray cells. The AND gate at the left is to compute the carry-out, which performs exactly the same function as previously discussed modulo $2^n - 1$ adder with Ling's scheme. The For binary add operation, c_{in} can be the input as the pseudo-carry to the last row of gray cells as well. By referencing Equation (2.43) and (3.21), the of following relationship can be established.

$$d_0 = H_{-1} = g_{-1} = c_{in} \tag{4.53}$$

This corresponds to the 2nd selection of the multiplexor.

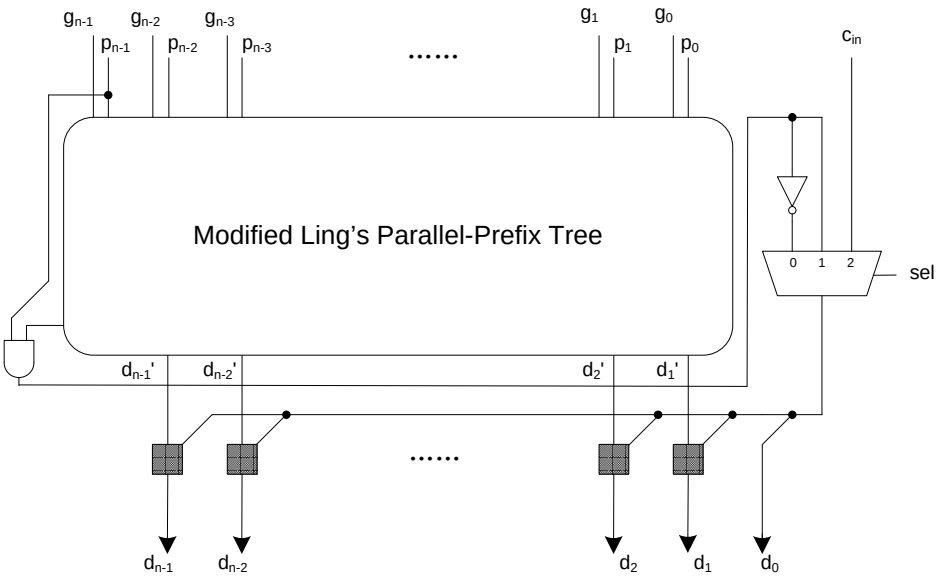


Figure 4.18: Combined Binary and Modulo $2^n \pm 1$ Adders using Ling's Scheme.

4.4 Summary

This chapter reviews the previous works done for modulo $2^n \pm 1$ adders. The implementation choice can be divided into three categories: the full parallel-prefix structures, the reduced parallel-prefix structures and carry-select incrementers. Based on the reduced parallel-prefix structures, which is a compromise between the fast full parallel-prefix structures and slow carry-select incrementers, Ling's algorithm is applied to these structures. The objective is to speed up the carry chain without loosing the structure regularity. Finally, a combined architecture for traditional binary and modulo $2^n \pm 1$ adders is proposed. This new architecture is extended from the reduced parallel-prefix structures with a multiplexor controlling the carry input.

CHAPTER 5

Results of Parallel-Prefix Structures and Comparisons

To see how the architecture impacts the circuit, quantitative data extracted from placed and routed circuits are desired, however, the implementation of parallel-prefix adders and modulo $2^n \pm 1$ adders with state-of-the-art technology is limited in publications. To address this problem, the implementation methodology shown in Figure 5.1 is employed. First, binary and modulo $2^n \pm 1$ addition are all formalized with Boolean equations. Modulo 2^n addition is the same as binary addition except that the carry-in/out are not required. After the formalization, multiple structures of the adders, especially the parallel-prefix structures are built as described in Chapter 3. Then the architectures are built with HDL and put through standard-cell based design flow to collect the data.

The methodology adopted is to quantify the algorithmic implementation. Several adder architectures were created and analyzed using a commercial-based VLSI design flow. To see the impact of the technology, this work includes TSMC 180nm, Cadence GSCL 90nm and a 45nm design kit called FreePDK [52]. FreePDK is an open source, variation aware Process Design Kit (PDK), based on Scalable CMOS (SCMOS) design rules, which is provided by North Carolina State University. With these kits, universities are able to perform System-on-Chip (SoC) implementations and statistical circuit analysis through commercial front-end and back-end tools, without violating the intellectual property (IP) controls. As the PDK is free, it can be modified and redistributed. In this way, research can be promoted to close the gap between industry and education facilities.

All designs start with Verilog HDL models, which are automatically generated with Perl scripts. The Perl scripts implement the algorithms of building the parallel-prefix structures,

with the extension to Ling's structure, modulo $2^n \pm 1$ adders, and the combined adder. The adders to be compared have the size of $8, 16, 32, 64$ and 128-bit. The HDL models are verified using testbench written in Verilog and put to synthesis with Synopsys Design Compiler, and placed and routed with Cadence Design Systems SoC Encounter. The parameters to be observed are Delay, Area, Power, Leakage Power, Energy and EDP (Energy Delay Product). Delay is derived from the placed and routed layout. Area is estimated by calculating the total area of all the gates composing the adder with Synopsys tool. Power numbers were generated from a $100,000$-vector VCD file and analyzed back through SoC Encounter. Energy is the product of power and delay. EDP [42] is the product of energy and delay, which is a good metric when design is targeting at low power without much compromise in delay.

5.1 Binary Parallel-Prefix Adders

A total of seven different types of parallel-prefix adders, along with their Ling versions and those architectures that can be embedded with carry-save notation are implemented. The selection includes the architectures proposed by Brent-Kung, Sklansky, Kogge-Stone, Han-Carlson, Knowles [1,1,1,1], Ladner-Fischer and Harris. Carry-save notation can be only applied to Brent-Kung, Han-Carlson, Ladner-Fischer and Harris prefix trees. Hence, there are 4 types of carry-save parallel-prefix adders being implemented. Figure 5.2, 5.3 and 5.4 show the results of the implementations. In the figures, the adder types are briefed as BK, Sk, KS, HC, Kn, LF and Ha, respectively. Ling's versions will have a prefix of $Ln-$ and carry-save version with a prefix $CS-$.

For 180nm technology, Ling's scheme can generally help to reduce the delay compared to the regular parallel-prefix structure. Carry-save notation demonstrates some savings in delays over regular structures and Ling's scheme has about the same delays. Ling's implementation has the largest area as it can be seen in Figure 5.2. Ling's scheme also consumes more power and has more leakage power than regular and carry-save implementations. In

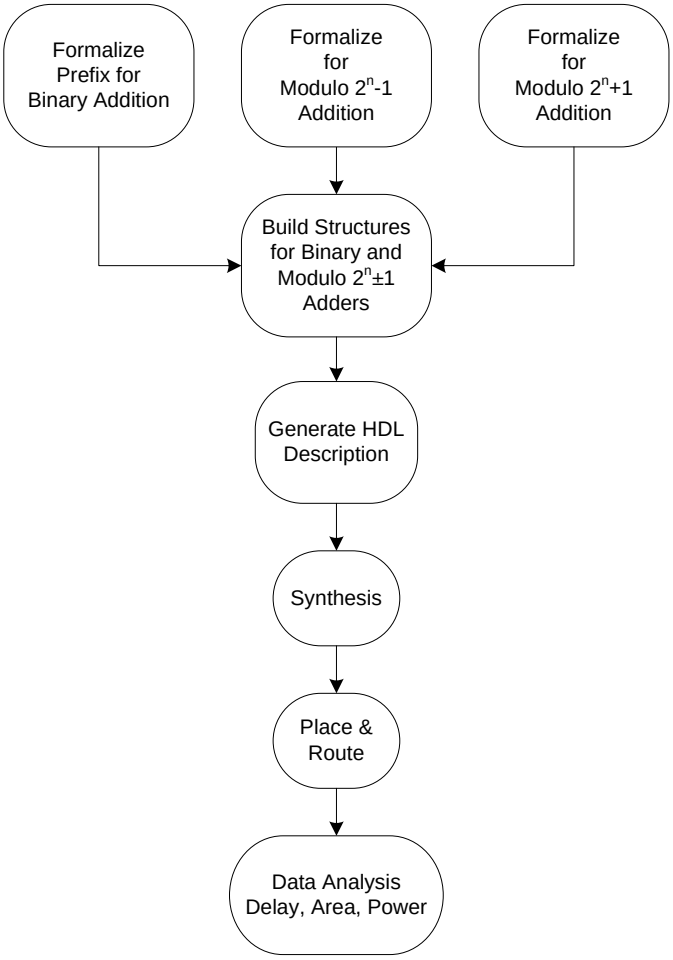


Figure 5.1: Methodology of This Work.

terms of EDP, Ling's version is no better than the regular and carry-save versions. Regular parallel-prefix structure has almost the same EDP as that with carry-save notation.

For 90nm technology, again, Ling's scheme helps with delay. Carry-save notation shows little better delay than Ling's version. Similar to 180nm technology, Ling's implementation takes more area, showing more power consumption and leakage power. The best EDP rests in either regular or carry-save parallel-prefix structure.

For 45nm technology, carry-save notation embedded architecture shows apparently less delay than regular implementation while Ling's scheme demonstrates even better numbers in delay. Unlike the cases for 180nm and 90nm technology, Ling's scheme is not the most area intensive. Ling's parallel-prefix structure consumes less power and has less leakage. In the category of EDP, Ling's scheme is better than both regular and carry-save parallel-prefix structure.

5.2 Modulo $2^n - 1$ Adders with Prefix Implementation

The results for modulo $2^n - 1$ adders include the reduced parallel-prefix structure and the full parallel-prefix structure. The end-around type of adders are implemented and data are retrieved. However, they are much slower than parallel-prefix structures with large area and power consumption and hence, the results are not included in the comparison. The reduced parallel-prefix structure include Brent-Kung, Sklansky, Kogge-Stone, Han-Carlson, Knowles [1,1,1,1], Ladner-Fischer and Harris, which are briefed as BK, Sk, KS, HC, Kn, LF and Ha, respectively in Figure 5.5, 5.6, 5.7. The Ling's version has a prefix of $Ln-$. The full parallel-prefix structure is briefed as FP.

For 180nm technology, the full parallel-prefix adders have the best delay in almost cases. Ling's scheme helps to reduce delay in most adders. The full parallel-prefix adders have larger area and consumes more power in general. The reduced parallel-prefix Ling adders take more area and consume more power than the regular parallel-prefix adders. The best EDP is in the reduced parallel-prefix adders, e.g. the 64-bit Han-Carlson prefix

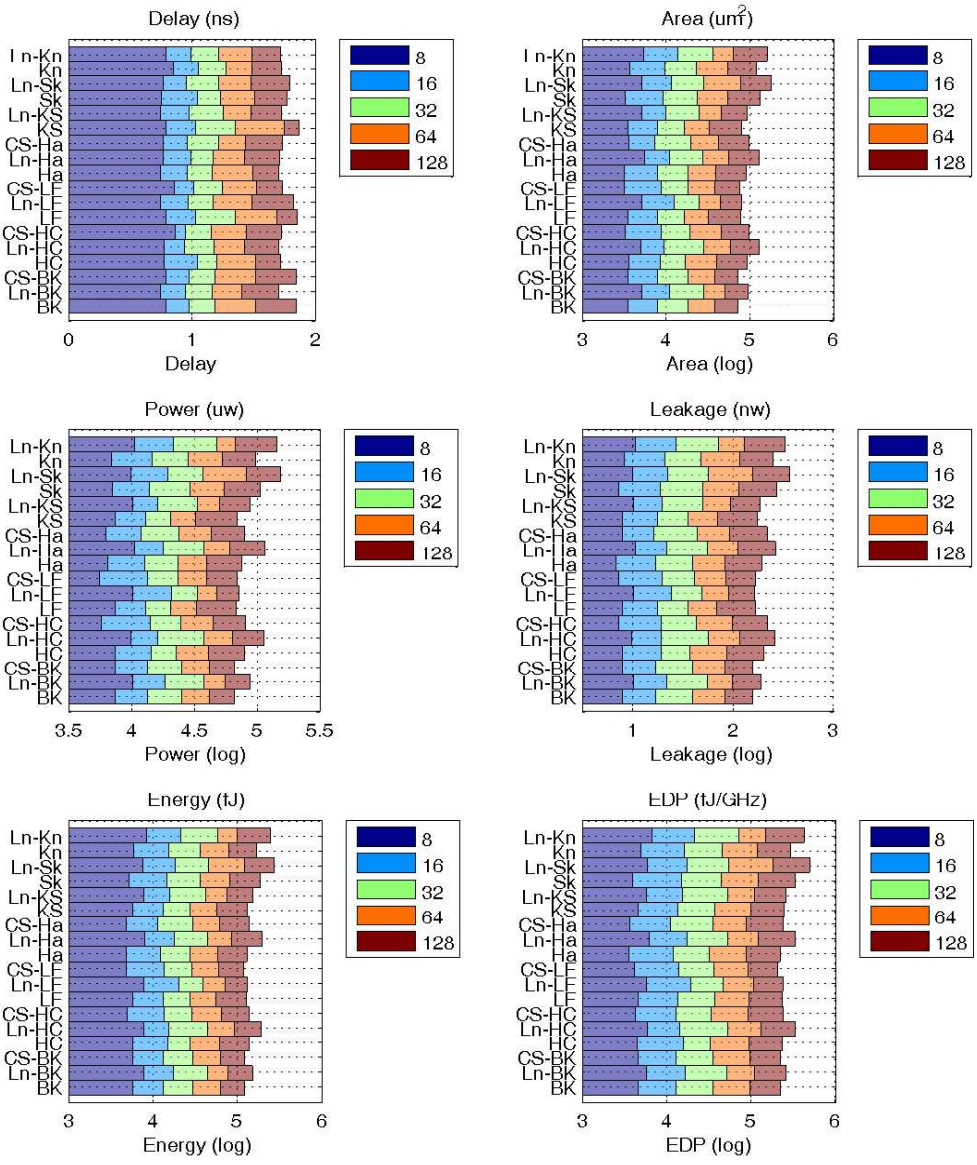


Figure 5.2: TSMC 180nm Technology for Parallel Prefix Adders.

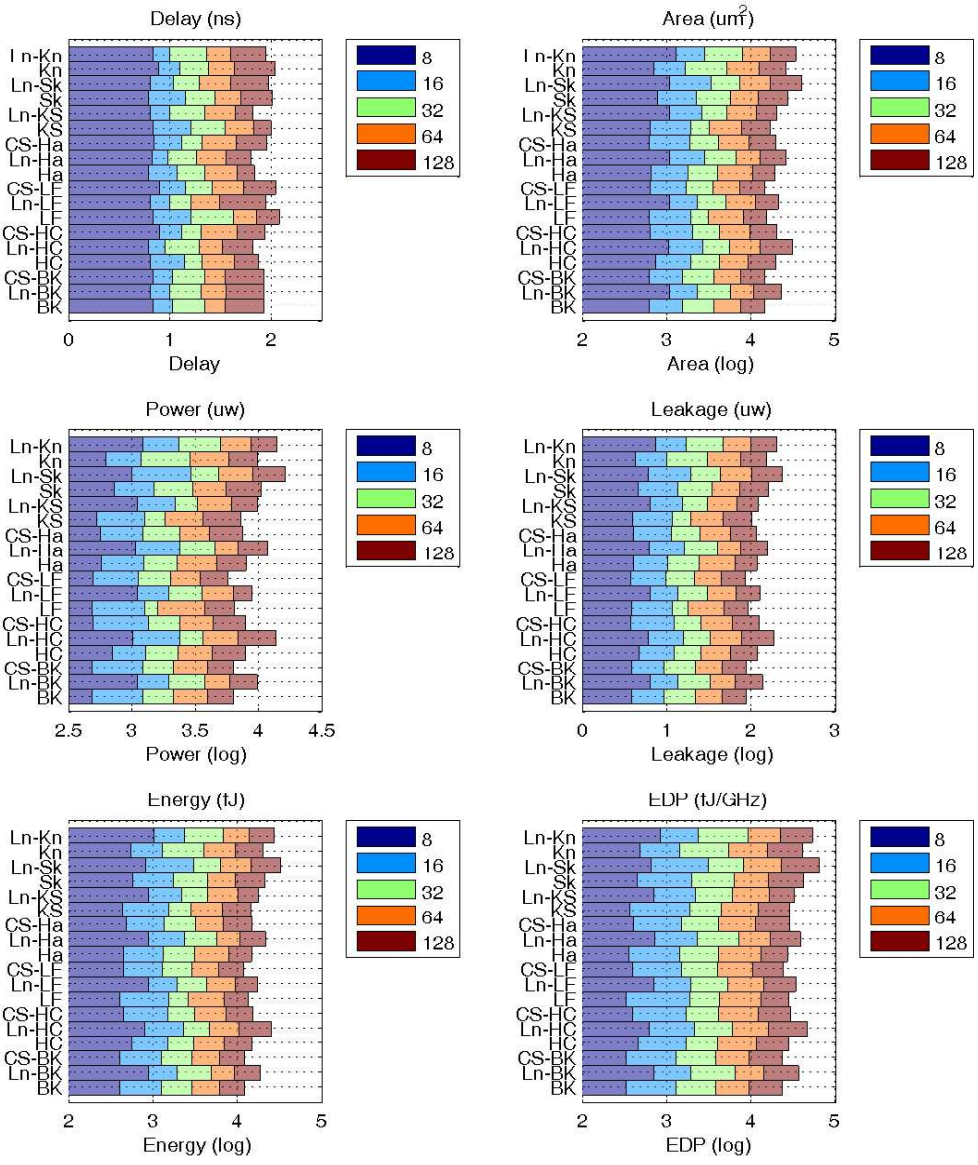


Figure 5.3: Cadence GSCL 90nm Technology for Parallel Prefix Adders.

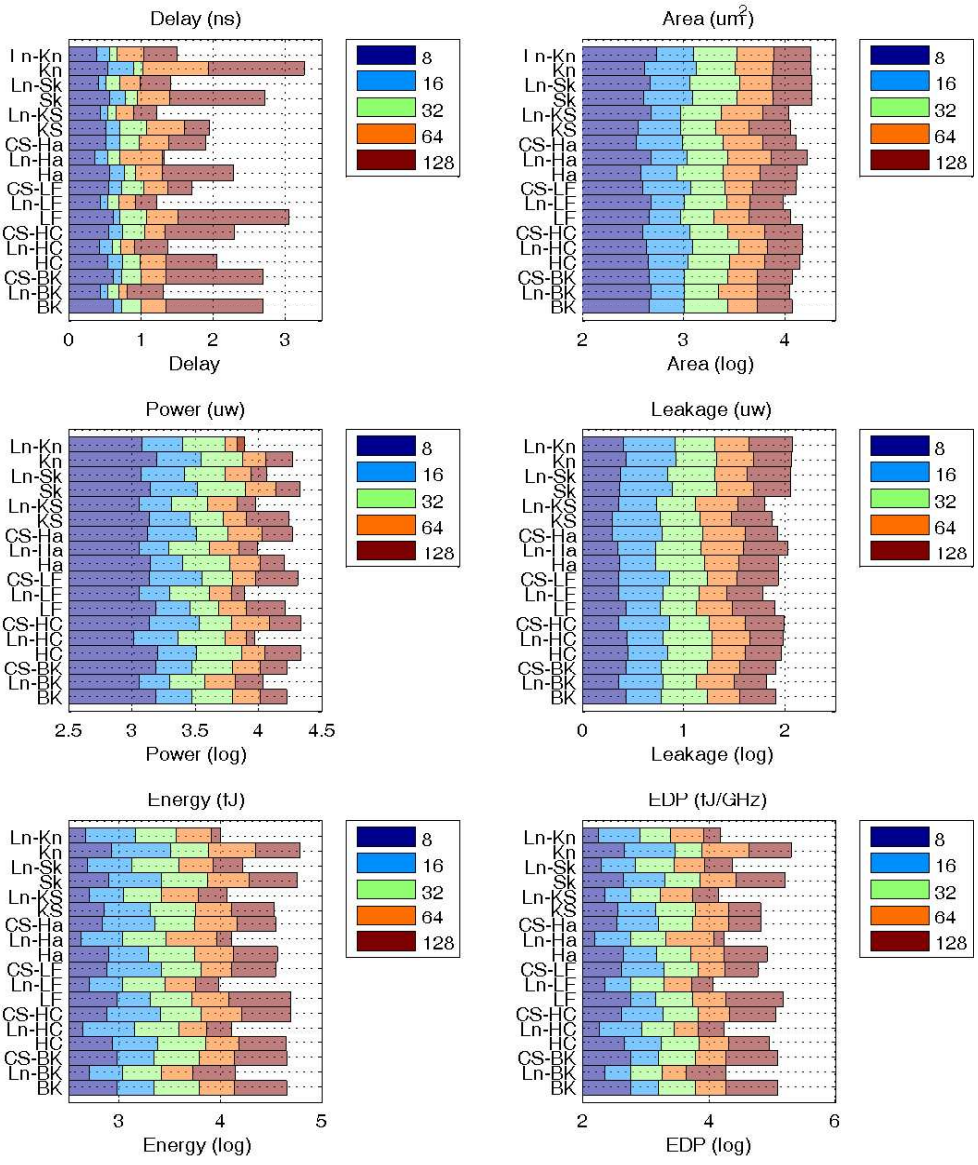


Figure 5.4: FreePDK 45nm Technology for Parallel Prefix Adders.

adder.

For 90nm technology, the full parallel-prefix adders have the best delay. Ling's scheme again helps the regular structure to reduce delay. The cost of the full and Ling parallel-prefix adders is the area, power consumption and leakage. The reduced parallel-prefix adders, like Brent-Kung's structures, have the best EDP in general.

For 45nm technology, interestingly, Ling's scheme has better delay than the full parallel-prefix adders. Generally, Ling and the full parallel-prefix adders have larger area. Ling's version tends to have less power consumption than non-Ling's applications. As it can be observed in EDP category, Brent-Kung reduced parallel-prefix Ling adders are the best.

5.3 Modulo $2^n + 1$ Adders with Prefix Implementation

The results for modulo $2^n + 1$ adders are presented in the same fashion as modulo $2^n - 1$ adders. The implementations include Brent-Kung (BK), Sklansky (Sk), Kogge-Stone (KS), Han-Carlson (HC), Knowles [1,1,1,1] (Kn), Ladner-Fischer (LF), Harris (Ha), the Ling's reduced parallel-prefix adders (with a prefix $Ln-$), and the full parallel-prefix structure (FP). Figure 5.8, 5.9 and 5.10 illustrate the results.

The results are similar to those for modulo $2^n + 1$ adders. For 180nm technology, the full parallel-prefix adders have the best delay. The reduced parallel-prefix Ling adders are among the second best. In terms of area, power and leakage, the regular reduced parallel-prefix adders are better than their Ling versions and the full parallel-prefix adders. The number of EDP indicate that Ling's scheme is no better than regular prefix structures.

For 90nm technology, the best delay is at the full parallel-prefix adders in general. Ling's scheme does not help to improve delay in as many cases as it does for modulo $2^n - 1$ adders. The full parallel-prefix adders take the most area, consume most power and have the greatest leakage. The reduced parallel-prefix Ling adders are ranked the second and the regular reduced parallel-prefix adders, the third. As a result, the regular reduced parallel-prefix adders have the best EDP than the other two types of adders.

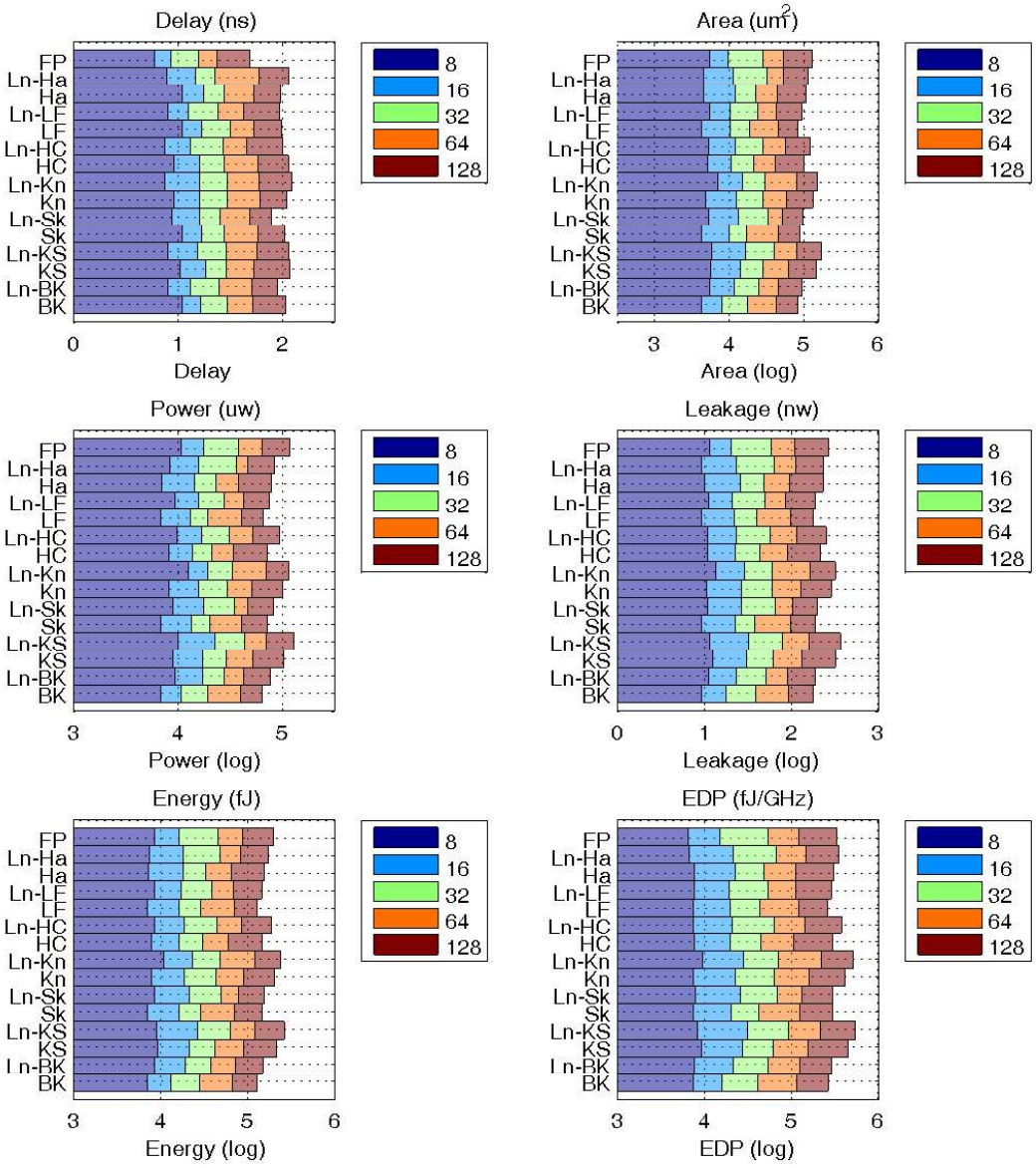


Figure 5.5: TSMC 180nm Technology for Modulo $2^n - 1$ Adders.

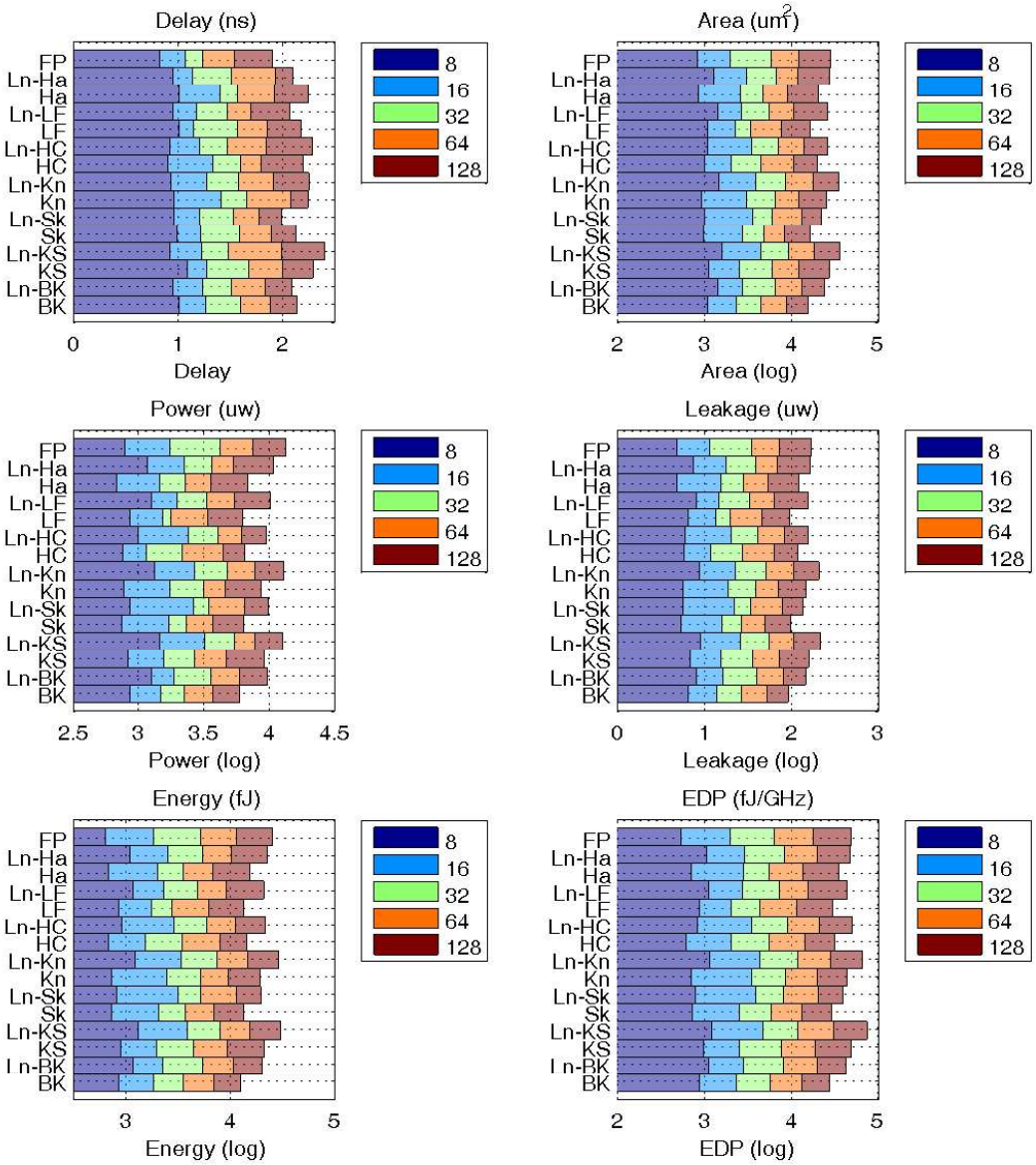


Figure 5.6: Cadence GSCL 90nm Technology for Modulo $2^n - 1$ Adders.

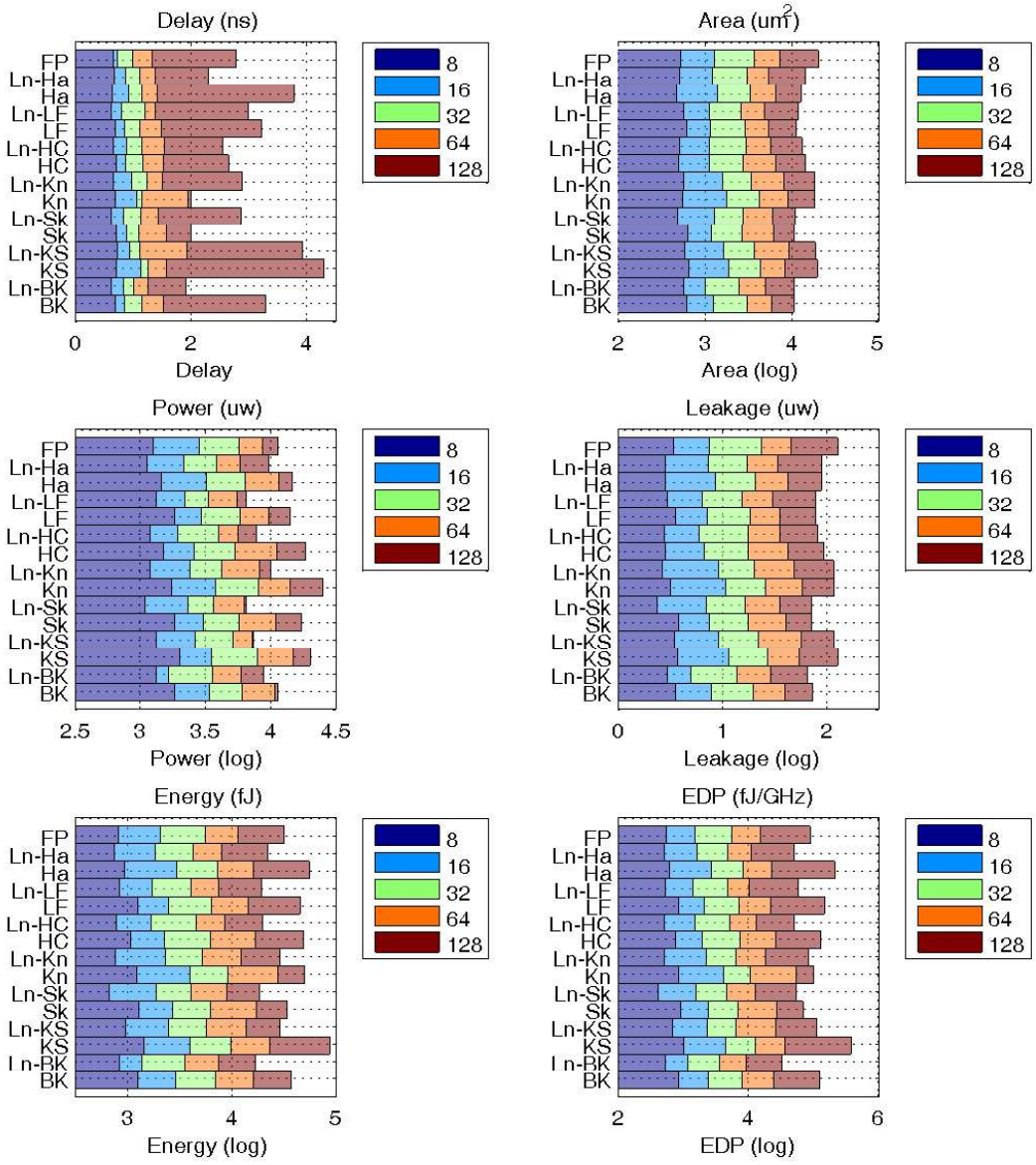


Figure 5.7: FreePDK 45nm Technology for Modulo $2^n - 1$ Adders.

For 45nm technology, Ling's reduced parallel-prefix adders show their advantages in delay. Ling's scheme also helps to reduce area, power consumption, even in some cases, leakage. The full parallel-prefix adders are no long the fastest. Ling's reduced parallel-prefix adders wind up with better EDP than the full and the regular reduced parallel-prefix adders.

5.4 A Combination of Modulo $2^n \pm 1$ and Binary Adders with Prefix Implementation

The combination of modulo $2^n \pm 1$ and binary adder is implemented using the reduced parallel-prefix structure, which can be easily applied to modulo $2^n \pm 1$ and binary adders. The selected structures are modified version of Brent-Kung (BK), Sklansky (Sk), Kogge-Stone (KS), Han-Carlson (HC), Knowles [1,1,1,1] (Kn), Ladner-Fischer (LF), Harris (Ha) along with their Ling's counterparts. Figure 5.11, 5.12 and 5.13 show the results.

For 180nm technology, Ling's scheme shows the advantage in delay. However, the regular structures are more area and power efficient, e.g. Brent-Kung's structure. The best EDP occurs at the regular reduced parallel-prefix structure as can be observed from the figure.

For 90nm technology, Ling's scheme does not help much in terms of delay. As for area and power, the regular structures show the similar advantage as those for 180nm technology. The regular structures are better in EDP than Ling's structures.

For 45nm technology, the improvement in delay of Ling's scheme is obvious, e.g. Knowles parallel-prefix Ling adder has superior savings in delay. Ling's structures sometimes consume more area but in most cases, less power. The EDP category also demonstrates the benefit of using Ling's approach.

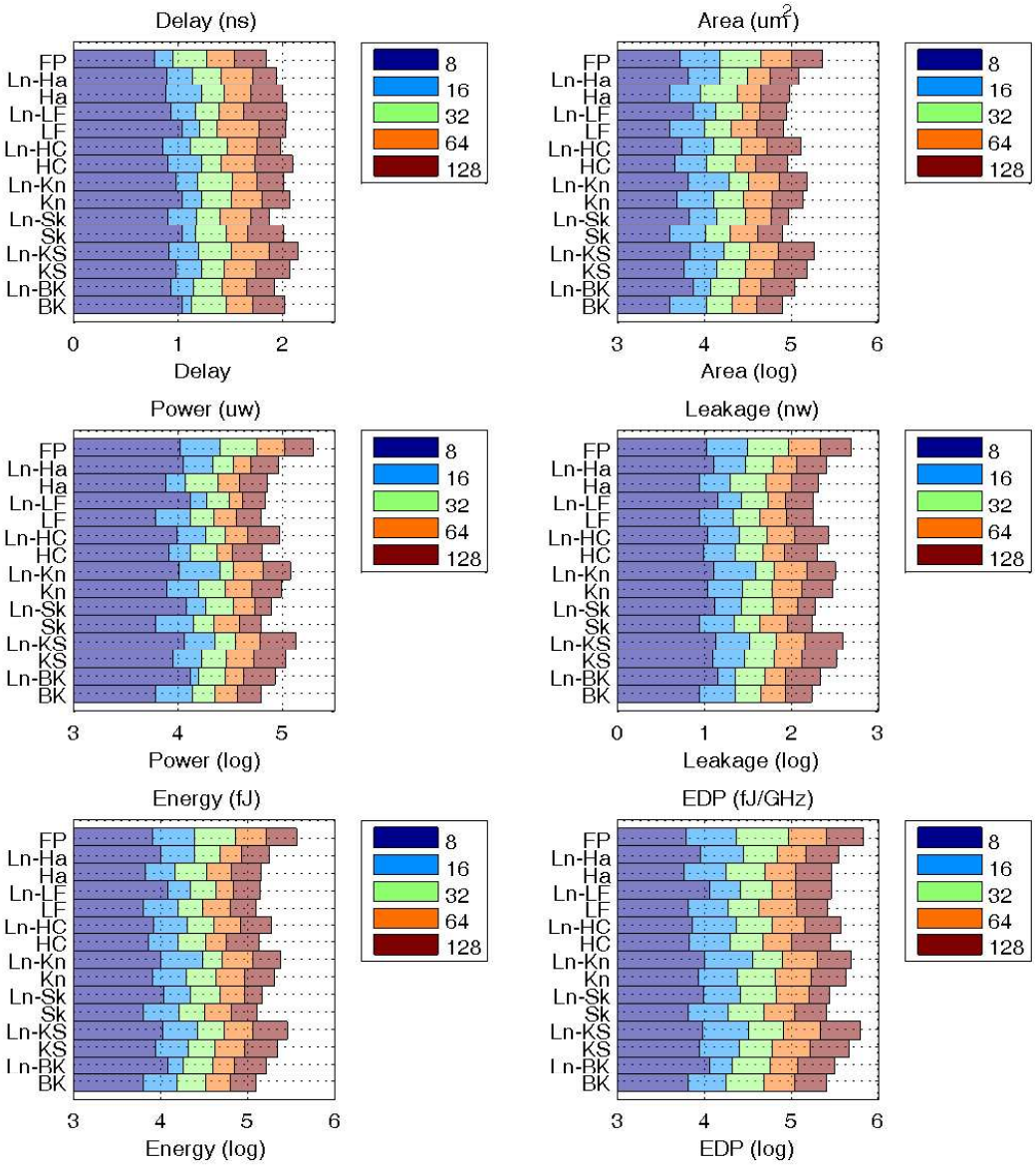


Figure 5.8: TSMC 180nm Technology for Modulo $2^n + 1$ Adders.

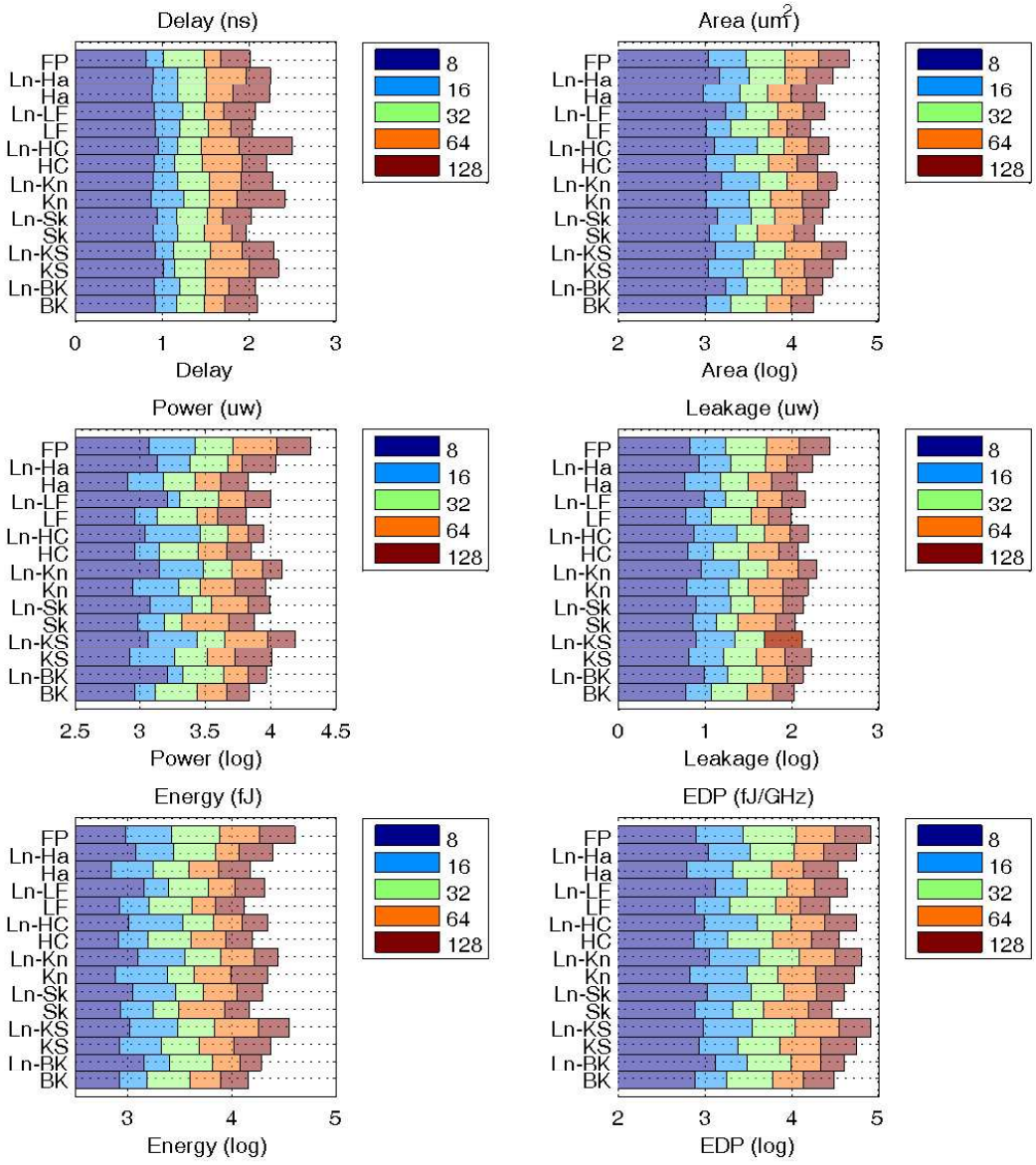


Figure 5.9: Cadence GSCL 90nm Technology for Modulo $2^n + 1$ Adders.

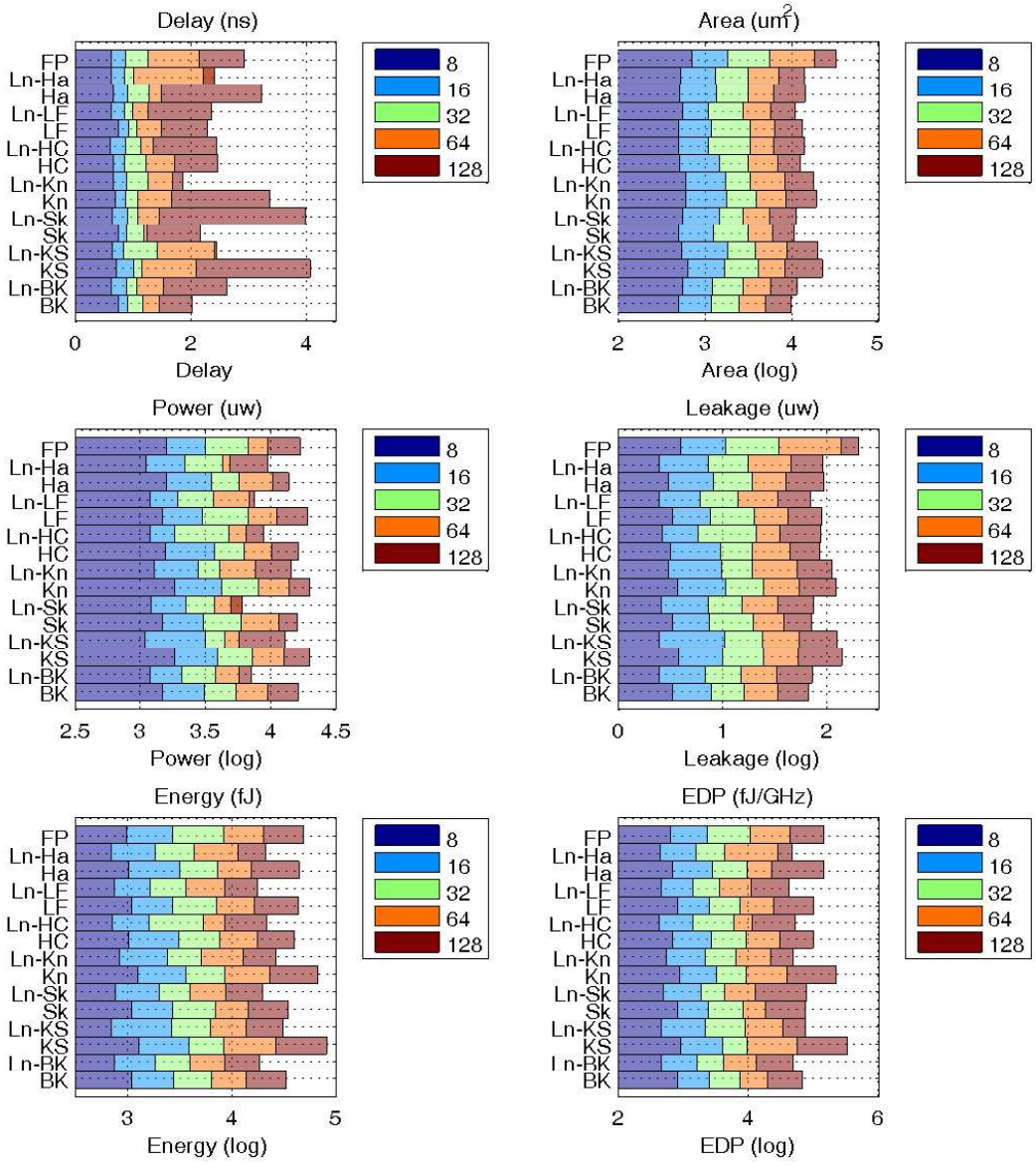


Figure 5.10: FreePDK 45nm Technology for Modulo $2^n + 1$ Adders.

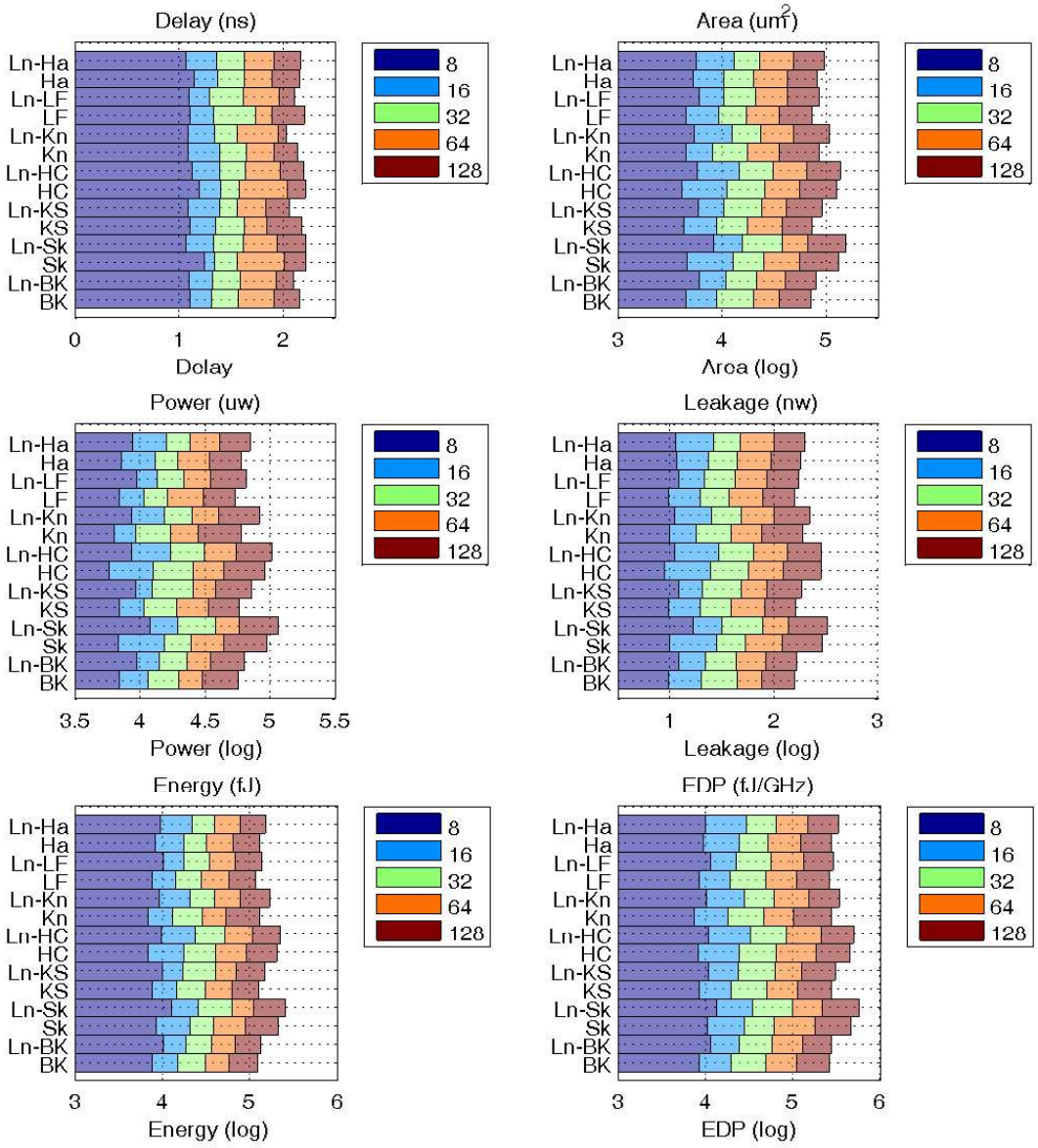


Figure 5.11: TSMC 180nm Technology for Modulo $2^n \pm 1$ and Binary Prefix Adders.

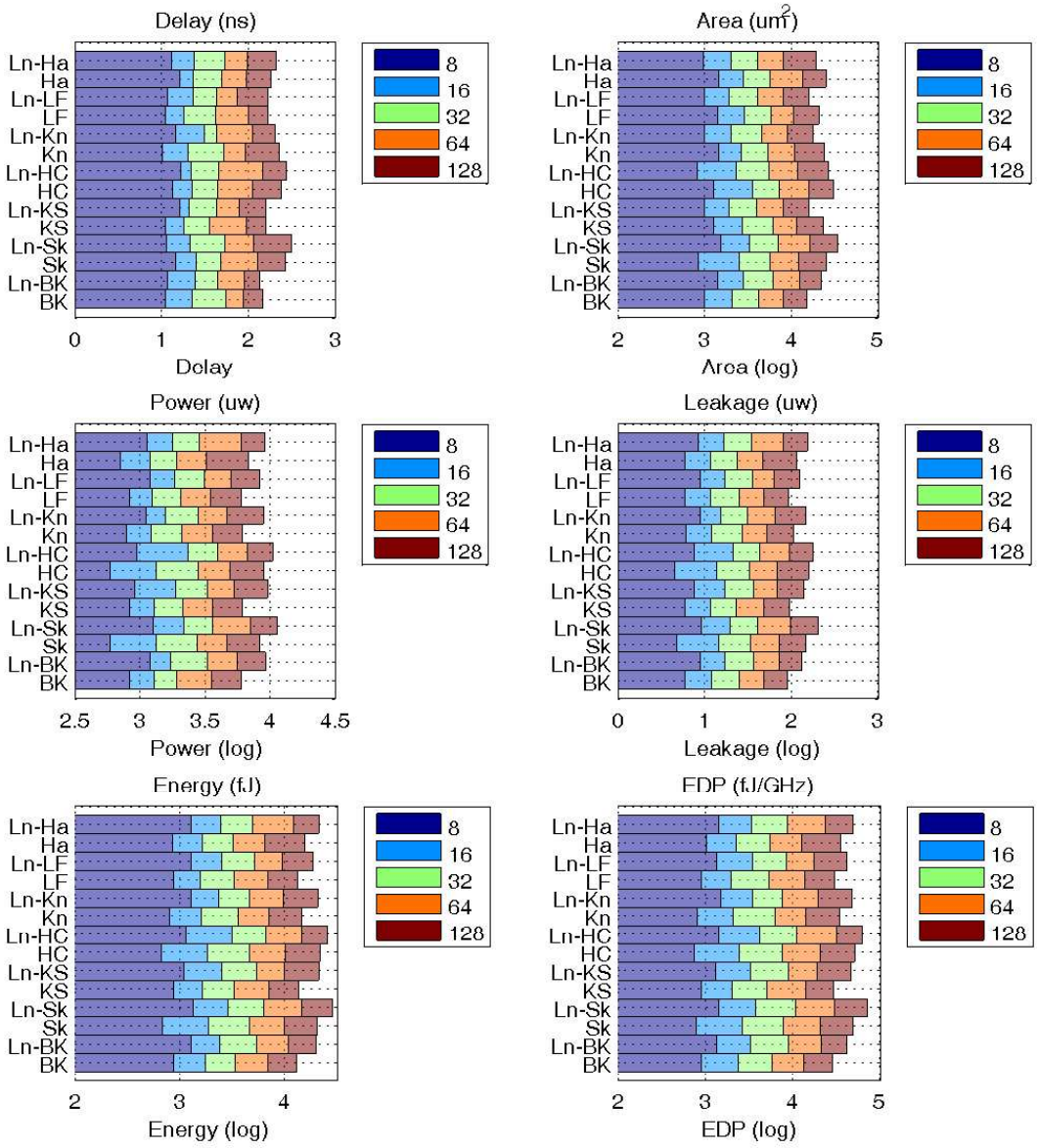


Figure 5.12: Cadence GSCL 90nm Technology for Modulo $2^n \pm 1$ and Binary Prefix Adders.

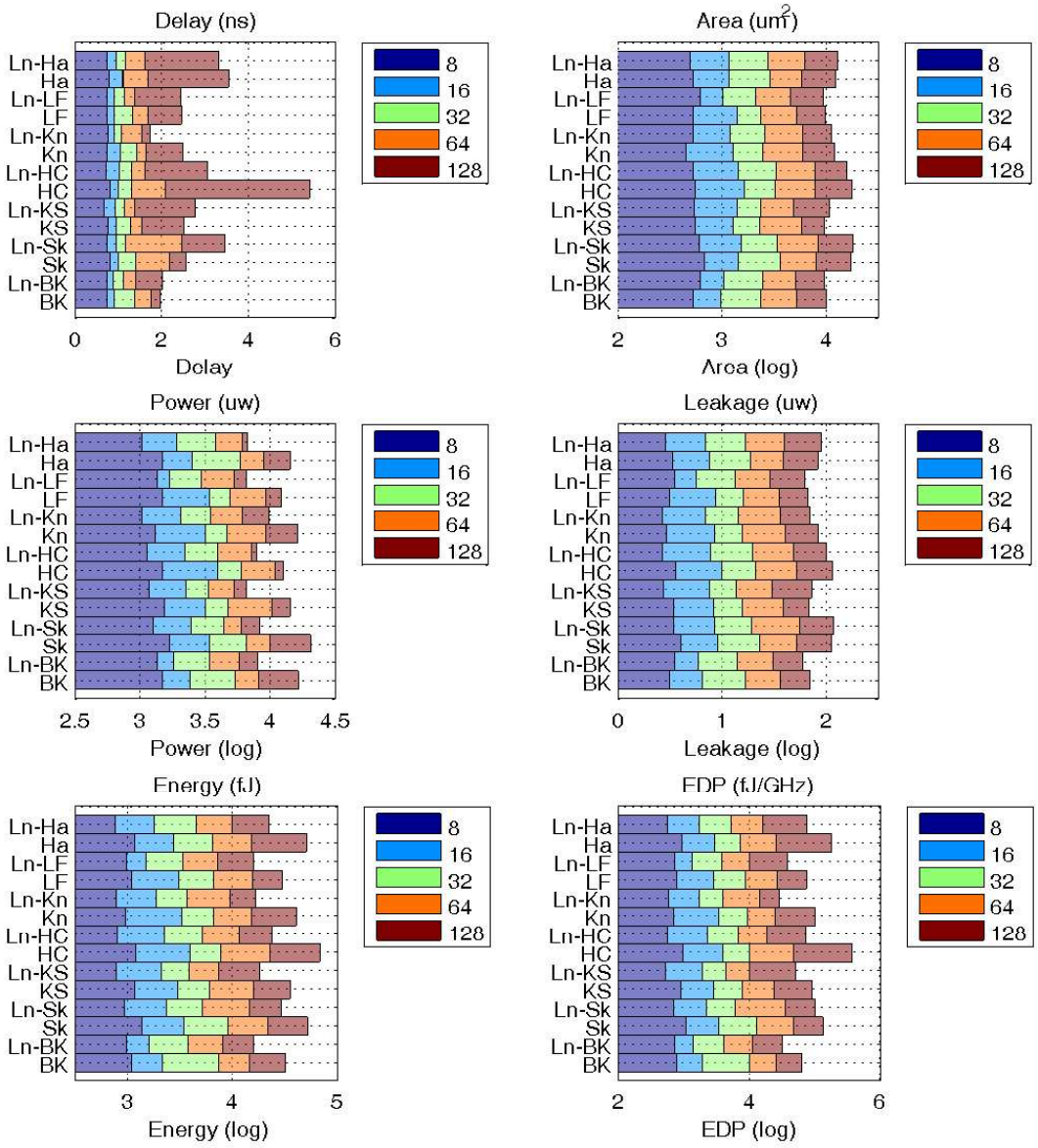


Figure 5.13: FreePDK 45nm Technology for Modulo $2^n \pm 1$ and Binary Prefix Adders.

5.5 Summary

The results indicate that there is no best adder architecture for all technology. For a certain technology, the choice of architecture is important for performance.

Ling's scheme can generally help to improve critical path so it usually demonstrates better delay. Carry-save notation is another potential way of improving the parallel-prefix adders. For 180nm and 90nm technologies, carry-save notation offers better EDP. For 45nm, Ling's scheme is the best at EDP.

For modulo $2^n - 1$ adders, the full parallel-prefix structures offer the best delay except the cases under 45nm technology. The full parallel-prefix structures are more area intensive and consume more power compared to the reduced parallel-prefix structures. Ling's scheme is the second best in terms of delay. For 45nm, Ling's scheme has the best EDP.

The full parallel-prefix structures are not always the fastest for modulo $2^n + 1$ adders, especially the cases under 90nm and 45nm technologies. The full parallel-prefix structures for modulo $2^n + 1$ require many more gates and congestive wires than the reduced parallel-prefix structures, which leads to their inefficiency in performance for wide adders. Ling's algorithm can help to reduce delay. Furthermore, Ling's scheme has the best EDP under 45nm technology.

As for the combined-function adder, the results agree with those from modulo $2^n \pm 1$ adders, i.e. Ling's scheme is helpful as can be observed from the numbers in delay. For 180nm and 90nm technologies, Ling's structure consumes more area, power and has larger EDP than regular structures. For 45nm technology, Ling's structures consumes the least power and have the best EDP.

CHAPTER 6

Conclusion

Parallel-prefix structures have been a hot topic because of their regular structure and logarithmic delay. However, few of the publications mentioned the construction of prefix structures and implementations with the state-of-the-art technology, i.e. technology in the nanometer range. Hence, there is still a lack of view how different parallel-prefix adders will behave in modern technology. The description on how parallel-prefix structures can contribute to modulo adders is also limited.

6.1 Dissertation Summary

This work starts with a review of basic adder architectures. Linear adders and non-linear adders are discussed with algorithmically analysis. The trade-offs between delay and area is presented. After that, parallel-prefix structures along with some other adder schemes are introduced, which offers a broad insight in how to build an adder.

After the review, the details of building the parallel-prefix structures for binary or 2^n adders are illustrated. To understand the properties of the architectures and their variants, design heuristics and simplified pseudo code for constructing the adders are provided. Ling's scheme is applied to parallel-prefix structure to see the potential improvement. Combining with carry-save notation, a new parallel-prefix structure is proposed. The parallel-prefix structure embedded with carry-save notation is a more balanced structure considering the trade-off factors for parallel-prefix adders. The theory is that the more balanced the structure is, the adder is more likely to have better overall performance.

For modulo $2^n \pm 1$ adders, the implementation of parallel-prefix structures can be cate-

gorized into the full parallel-prefix and the reduced parallel-prefix type of adders. The full parallel-prefix adders are algorithmically faster while the reduced parallel-prefix adders are less complicated. However, the reduced parallel-prefix adders have the flexibility to be easily modified from the existing parallel-prefix architectures and Ling's scheme can be applied to help enhance the performance.

Finally, quantitative results are provided with 180nm, 90nm and 45nm standard-cell libraries. All the parallel-prefix adders, along with their Ling versions and carry-save versions are implemented. Modulo $2^n \pm 1$ adders in the full and the reduced parallel-prefix structures are implemented. A combo adder, which is capable of handling with both binary and modulo $2^n \pm 1$ addition, is also implemented with parallel-prefix structure. Important design parameters, including delay, area, power, energy and EDP are listed and compared. The results suggest that no single type of architecture is the best for all technologies. However, the results offer enough insights of which type of adders is the best for a given technology. In general, the quantitative results agree with the qualitative analysis.

The selected results within FreePDK 45nm technology are summarized in Table 6.1, 6.2 and 6.3 to further clarify the comparison of binary, modulo $2^n - 1$ and modulo $2^n + 1$ parallel-prefix adders. The naming convention is the same as in Chapter 5, i.e. Brent-Kung, Sklansky, Kogge-Stone, Han-Carlson, Knowles [1,1,1,1], Ladner-Fischer and Harris are briefed as BK, Sk, KS, HC, Kn, LF and Ha, respectively. Ling's versions will have a prefix of $Ln-$ and carry-save version with a prefix $CS-$. The best numbers are highlighted in bold fonts.

For binary parallel-prefix adders, it is apparent that Ling's scheme can help the carry-chain problem in parallel-prefix structures. For example, Brent-Kung parallel-prefix Ling structure has the best delay with a little more power consumption than minimum. For modulo $2^n \pm 1$ adders, the reduced parallel-prefix structures that have better delay than the full parallel-prefix structures. Ling's scheme always helps to reduce the power, up to over 50%, while the delay is not always better. However, the benefit is more obvious for Ling's

Table 6.1: Results for 64-bit Parallel-Prefix Adders within FreePDK 45nm Technology

Types	Area (um^2)	Delay (ns)	Power (uw)	EDP (fJ/GHz)
BK	5,246	1.344	10,163	18,357
Ln-BK	5,287	**0.809**	6,543	**4,282**
CS-BK	5,246	1.344	10,163	18,357
HC	6,243	1.339	11,141	19,974
Ln-HC	6,651	0.914	7,881	6,584
CS-HC	6,189	1.331	11,982	21,226
LF	4,355	1.513	7,945	18,187
Ln-LF	4,482	0.923	**6,110**	5,205
CS-BK	4,738	1.368	9,357	17,510
Ha	5,627	1.297	10,225	17,200
Ln-Ha	7,225	1.297	7,001	11,777
CS-Ha	5,994	1.385	10,496	20,133
KS	**4,347**	1.602	7,942	20,384
Ln-KS	5,952	0.900	6,708	5,433
Sk	7,497	1.391	13,594	26,302
Ln-Sk	7,373	0.985	8,578	8,322
Kn	7,575	1.933	11,263	42,084
Ln-Kn	7,646	1.036	7,699	8,263

structures when EDP is taken into consideration.

6.2 Future Directions

Parallel-prefix structure is attractive for adders because of its logarithmic delay. The influence of design trade-offs can be easily observed from adder designs. In this work, the radix is fixed as 2. However, higher radix may be desirable in some applications as indicated by [49]. With higher radix, the prefix tree will be more sparse, allowing savings in area and power. With Doran's formula [39], the hybrid ripple-carry and prefix adders may have even better performance. Another direction can lead to the combination of Ling's idea with carry-save notation. Considering all the possibilities, there are still plenty to explore for adder design.

Current adder design is getting more complex and adders are having wider applications

Table 6.2: Results for 64-bit Modulo $2^n - 1$ Parallel-Prefix Adders within FreePDK 45nm Technology

Types	Area (um^2)	Delay (ns)	Power (uw)	EDP (fJ/GHz)
BK	5,792	1.521	10,584	24,485
Ln-BK	4,943	**1.251**	5,907	**9,244**
KS	8,145	1.569	14,761	36,338
Ln-KS	9,155	1.916	7,189	26,393
Sk	6,145	1.567	10,803	26,526
Ln-Sk	5,960	1.436	6,226	12,839
Kn	8,845	1.934	14,070	52,626
Ln-Kn	7,971	1.503	8,088	18,272
HC	6,488	1.520	11,078	25,594
Ln-HC	5,701	1.532	5,613	13,175
LF	5,354	1.493	9,565	21,321
Ln-LF	**4,815**	1.376	**5,426**	10,274
Ha	6,488	1.408	11,343	22,487
Ln-Ha	5,370	1.380	5,754	10,958
FP	7,269	1.327	8,625	15,189

Table 6.3: Results for 64-bit Modulo $2^n + 1$ Parallel-Prefix Adders within FreePDK 45nm Technology

Types	Area (um^2)	Delay (ns)	Power (uw)	EDP (fJ/GHz)
BK	**4,945**	1.454	9,355	19,778
Ln-BK	5,618	1.516	5,663	13,016
KS	8,277	2.091	12,470	54,522
Ln-KS	8,627	2.399	5,696	32,784
Sk	5,943	**1.246**	11,518	17,881
Ln-Sk	5,469	1.458	5,979	12,709
Kn	8,507	1.663	13,792	38,142
Ln-Kn	8,214	1.676	7,603	21,358
HC	6,775	1.721	10,082	29,861
Ln-HC	6,053	1.342	6,407	11,539
LF	6,292	1.484	11,083	24,407
Ln-LF	5,612	1.262	6,826	**10,872**
Ha	6,116	1.491	10,225	22,731
Ln-Ha	7,010	2.393	**4,822**	27,615
FP	17,888	2.143	9,345	42,919

than ever before. Quantitative analysis using certain technology is required instead of simple gate model for performance analysis. Corner simulation is also desirable for comprehensive characterization of adder architectures. Custom design techniques can be employed for implementing adders. However, this kind of task is usually more time-consuming. Therefore, automated custom design flow may be worthwhile when design cost is justified.

6.3 In Summary

The goals of this work are to clarify the construction of parallel-prefix structures, with the extension to modulo $2^n \pm 1$ adders, and to study the mutual influence of modern technology and algorithmic architectures. As presented in previous sections, the objectives have been achieved.

BIBLIOGRAPHY

[1] D. Harris, "A taxonomy of parallel prefix networks," in *Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, Nov. 2003, pp. 2213–2217.

[2] D. Goldberg, "What every computer scientist should know about floating point arithmetic," *ACM Computing surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[3] J. Chen and J. E. Stine, "Optimization of bipartite memory systems for multiplicative divide and square root," *48th IEEE International Midwest Symp. Circuits and Systems*, vol. 2, pp. 1458–1461, 2005.

[4] S. Winograd, "On the time required to perform addition," *J. ACM*, vol. 12, no. 2, pp. 277–285, 1965.

[5] R. K. Richards, *Arithmetic Operations in Digital Computers*. D. Van Nostrand Co., Princeton, N.J., 1955.

[6] A. Weinberger and J. Smith, "A logic for high-speed addition," *National Bureau of Standards*, no. Circulation 591, pp. 3–12, 1958.

[7] A. Tyagi, "A reduced area scheme for carry-select adders," *IEEE Trans. Computers*, vol. 42, no. 10, pp. 1163–1170, Oct. 1993.

[8] H. Ling, "High speed binary adder," *IBM Journal of Research and Development*, vol. 25, no. 3, pp. 156–166, 1981.

[9] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[10] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence relations," *IEEE Trans. Computers*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[11] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symp. Comp. Arith.*, June 2001, pp. 277–281.

[12] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. Electronic Computers*, vol. EC-9, pp. 226–231, June 1960.

[13] R. Ladner and M. Fischer, "Parallel prefix Computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[14] T. Han and D. Carlson, "Fast area-efficient VLS Adders," in *Proc. 8th Symp. Comp. Arith.*, Sept. 1987, pp. 49–56.

[15] A. Naini, D. Bearden, and W. Anderson, "A 4.5ns 96b CMOS adder design," in *Proc. IEEE Custom Integrate Circuits Conference*, vol. 38, no. 8, Apr. 1965, pp. 114–117.

[16] T. Kilburn, D. B. G. Edwards, and D. Aspinall, "Parallel addition in digital computers: a new fast carry circuit," in *Proc. IEE*, vol. 106, pt. B, Sept. 1959, p. 464.

[17] N. Szabo and R. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*. McGraw-Hill, 1967.

[18] W. K. Jenkins and B. J. Leon, "The use of residue number systems in the design of finite impulse response digital filters," *IEEE Trans. Circuits and Systems*, vol. 24, no. 4, pp. 171–201, Apr. 1977.

[19] X. Lai and J. L. Massey, "A proposal for a new block encryption standard," in *Advances in Cryptology - EUROCRYPT'90, Berlin, Germany: Springer-Verlag*, 1990, pp. 389–404.

[20] S. S.-S. Yau and Y.-C. Liu, "Error correction in redundant residue number systems," *IEEE Trans. Computers*, vol. C-22, no. 1, pp. 5–11, Jan. 1973.

[21] F. Halsall, *Data Communications, Computer Networks and Open Systems*. Addison Wesley, 1996.

[22] C. Efstathiou, D. Nikolos, , and J. Kalamatianos, "Area-time efficient modulo $2^n - 1$ adder design," *IEEE Trans. Circuits and System-II*, vol. 41, no. 7, pp. 463–467, 1994.

[23] L. Kalamboukas, D. Nikolos, C. Efstathiou, H. T. Vergos, , and J. kalamatianos, "High-speed parallel-prefix modulo $2^n - 1$ adders," *IEEE Trans. Computers*, vol. 49, no. 7, special issure on computer arithmetic, pp. 673–680, July 2000.

[24] C. Efstathiou, H. T. Vergos, and D. Nikolos, "Fast parallel-prefix modulo $2^n + 1$ adders," *IEEE Trans. Computers*, vol. 53, no. 9, pp. 1211–1216, Sept. 2004.

[25] H. T. Vergos, C. Efstathiou, and D. Nikolos, "Modulo $2^n \pm 1$ adder design using select-prefix blocks," *IEEE Trans. Computers*, vol. 52, no. 11, pp. 1399–1406, Nov. 2003.

[26] V. Paliouras and T. Stouraitis, "Novel high-radix residue number system multipliers and adders," in *Proc. 1999 IEEE Int'l Symp. Circuits and Systems VLSI (ISCAS '99)*, 1999, pp. 451–454.

[27] S. Bi, W. J. Gross, W. Wang, A. Al-khalili, and M. N. S. Swamy, "An area-reduced scheme for modulo $2^n - 1$ addition/subtraction," in *Proc. 9th International Database Engineering & Application Symp.*, 2005, pp. 396–399.

[28] R. Zimmermann, "Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication," in *Proc. 14th IEEE Symp. Computer Arithmetic*, 1999, pp. p.158–167.

[29] J. Chen and J. E. Stine, "Enhancing parallel-prefix structures using carry-save notation," *51st Midwest Symp. Circuits and Systems*, pp. 354–357, 2008.

[30] G. E. Moore, “Cramming more components onto integrated circuits,” in *Electronics*, May 1965, pp. 25.5.1–25.5.4.

[31] M. Lehman and N. Burla, “Skip techniques for high-speed carry propagation in binary arithmetic units,” *IRE Trans. Electron. Comput.*, pp. 691–698, Dec. 1961.

[32] O. J. Bedrij, “Carry-select adder,” *IRE Trans. Electron. Comput.*, pp. 340–346, June 1962.

[33] J. Sklansky, “Conditional sum addition logic,” *IRE Trans. Electron. Comput.*, pp. 226–231, June 1960.

[34] V. G. Oklobdzija, B. Zeydel, H. Dao, S. Mathew, and R. Krishnamurthy, “Energy-delay estimation technique for high-performance microprocessor VSLI adders,” *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH-16’03)*, p. 272, June 2003.

[35] R. Zimmermann, “Binary adder architectures for cell-based vlsi and their synthesis,” Ph.D. dissertation, ETH Dissertation 12480, Swiss Federal Institute of Technology, 1997.

[36] R. Zimmermann and H. Kaeslin, “Cell-based multilevel carry-increment adders with minimal AT- and PT-products.”

[37] S. Majerski, “On determination of optimal distributions of carry skips in adders,” *IEEE Trans. Electron. Comput.*, pp. 45–58, Feb. 1967.

[38] J. E. Stine, *Digital Computer Arithmetic Datapath Design Using Verilog HDL*. Kluwer Academic, 2004.

[39] R. W. Doran, “Variants of an improved carry-look-ahead-sum adder,” *IEEE Trans. Computers*, vol. 37, no. 9, pp. 1110–1113, 1988.

[40] J. Grad, "Analysis and implementation of binary addition in nanometer cmos technology," Ph.D. dissertation, Department of Electrical Engineering: Illinois Institute of Technology, May 2006.

[41] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2004.

[42] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman, "Robust energy-efficient adder topologies," in *18th IEEE International Symposium on Computer Arithmetic*, June 2007, pp. 16–28.

[43] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A System Perspective*. Addison-Wesley, 1985.

[44] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electron. Comput.*, pp. 389–400, Sep. 1961.

[45] ——, "A study of redundant number representations for parallel digital computers," Ph.D. dissertation, University of Illinois, Urbana, Jan. 1960.

[46] I. Sutherland, B. Sproull, and D. Harris, *Logic Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[47] P. M. Seidel and G. Even, "Delay-optimized implementation of IEEE floating-point addition," *IEEE Trans. Computers*, vol. 53, no. 2, pp. 97–113, Feb. 2004.

[48] N. Burgess, "Prenormalization rounding in IEEE floating-point operations using a flagged prefix adder," *IEEE Trans. VLSI Systems*, vol. 13, pp. 266–277, February 2005.

[49] S. Naffziger, "A subnanosecond 0.5μm 64b adder design," in *Proc. IEEE Int'l Solid-State Circuits Conf.*, 1996, pp. 362–363.

[50] H. T. Vergos and C. Efstathiou, "Efficient modulo $2^n + 1$ adder architectures," *Integration VLSI J. (2008)*, p. doi:10.1016/j.vlsi.2008.04.004, 2008.

[51] H. T. Vergos, C. Efstathiou, and D. Nikolos, "Diminished-one modulo $2^n + 1$ adder design," *IEEE Trans. Computers*, vol. 51, no. 12, pp. 1389–1399, 2002.

[52] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An open-source variation-aware design kit," in *Proc. IEEE International Microelectronic System Education*, 2007, pp. 173–174.

VITA

Jun Chen

Candidate for the Degree of

Ph.D.

Dissertation: PARALLEL-PREFIX STRUCTURES FOR BINARY AND MODULO $\{2^n - 1, 2^n, 2^n + 1\}$ ADDERS

Major Field: Electrical and Computer Engineering

Biographical:

Education: Completed the requirements for the Ph.D. degree with a major in Electrical and Computer Engineering at Oklahoma State University in December, 2008.

Experience: Dr. Jun Chen received his Master's Degree in Electrical and Computer Engineering from Illinois Institute of Technology in 2004. He worked as a Firmware Engineer from 2000 to 2002 in ALi (Shanghai) Corp., China. He graduated from Shanghai Jiao Tong University with his Bachelor's Degree in 2000.

Professional Memberships: IEEE Student Member

Name: Jun Chen Date of Degree: December, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: PARALLEL-PREFIX STRUCTURES FOR BINARY AND MODULO $\{2^n - 1, 2^n, 2^n + 1\}$ ADDERS

Pages in Study: 124 Candidate for the Degree of Ph.D.

Major Field: Electrical and Computer Engineering

Adders are the among the most essential arithmetic units within digital systems. Parallel-prefix structures are efficient for adders because of their regular topology and logarithmic delay. However, building parallel-prefix adders are barely discussed in literature. This work puts emphasis on how to build prefix trees and simple algorithms for building these architectures. One particular modification of adders is for use with modulo arithmetic. The most common type of modulo adders are modulo $2^n - 1$ and modulo $2^n + 1$ adders because they have a common base that is a power of 2. In order to improve their speed, parallel-prefix structures can also be employed for modulo $2^n \pm 1$ adders. This dissertation presents the formation of several binary and modulo prefix architectures and their modifications using Ling's algorithm. For all binary and modulo adders, both algorithmic and quantitative analysis are provided to compare the performance of different architectures. Furthermore, to see how process impact the design, three technologies, from deep submicron to nanometer range, are utilized to collect the quantitative data .

ADVISOR'S APPROVAL: _____