HIGH-SPEED MULTI OPERAND ADDITION UTILIZING FLAG BITS

BY

VIBHUTI DAVE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Engineering
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

_____
Co – Adviser

Chicago, Illinois
May 2007

ACKNOWLEDGEMENT

I would like to thank my mentor Dr. Erdal Oruklu for his constant support and undue faith in me. I highly appreciate the time he has invested during my research and for the completion of this dissertation. This dissertation would not have been possible without Dr. Jafar Saniie, my advisor and his attempts to challenge me throughout my academic program, encouraging me when I was successful and pushing me to do better when I fell short. I would also like to thank Dr. Dimitrios Velenis and Dr. James Stine for their constructive criticism about my work and helping me to perform better. A special thanks to the committee members for their support and time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The goal of this research is to design arithmetic circuits that meet the challenges faced by computer architects during the design of high performance embedded systems. The focus is narrowed down to addition algorithms and the design of high speed adder architectures. Addition is one of the most basic operations performed in all computing units, including microprocessors and digital signal processors. It is also a basic unit utilized in various complicated algorithms of multiplication and division.

Various adder architectures for binary addition have been investigated, in view of a wide range of performance characteristics, which include delay, area, power, the size of the input operands, and the number of input operands. Efficient implementation of an adder circuit usually revolves around reducing the cost to propagate the carry between successive bit positions. The problem of carry propagation is eliminated by expressing addition as a prefix computation. The resulting adder circuits are called parallel prefix adders. Based on the advantages posed by the prefix scheme, a qualitative evaluation of three different existing prefix adders (Brent – Kung, Ladner – Fischer, and Kogge – Stone) has been performed. A technique to enhance the functionality of these basic designs has also been investigated enabling their utilization in integer arithmetic. The technique has been named after the new set of intermediate outputs that are generated as part of the algorithm, called the flag bits. This technique targets a new design criteria; the number of results obtained at the output. An algorithm based on flag bits has been proposed to achieve three-operand addition which has proven to have a favorable overall performance with respect to conventional multi-operand addition schemes.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Besides technological scaling, advances in the field of computer architecture have also contributed to the exponential growth in performance of digital computer hardware. The flip-side of the rising processor performance is an unprecedented increase in hardware and software complexity. Increasing complexity leads to high development costs, difficulty with testability and verifiability, and less adaptability. The challenge in front of computer designers is therefore to opt for simpler, robust, and easily certifiable circuits. Computer arithmetic, here plays a key role aiding computer architects with this challenge. It is one of the oldest sub-fields of computer architecture. The bulk of hardware in earlier computers resided in the accumulator and other arithmetic/logic circuits.

Successful operation of computer arithmetic circuits was taken for granted and high performance of these circuits has been routinely expected. This context has been changing due to various reasons. First, at very high clock rates, the interfaces between arithmetic circuits and the rest of the processor become critical. Arithmetic circuits can no longer be designed and verified in isolation. Rather an integrated design optimization is required. Second, optimizing arithmetic circuits to meet the design goals by taking advantage of the strengths of new technologies, and making them tolerant to the weakness, requires a re-examination of existing design paradigms. Finally, incorporation of higher-level arithmetic primitives into hardware makes the design, optimization and verification efforts highly complex and interrelated.

The core of every microprocessor, digital signal processor (DSP), and data-processing application-specific integrated circuit (ASIC) is its datapath. With respect to the most important design criteria; critical delay, chip size, and power dissipation, the datapath is a crucial circuit component. The datapath comprises of various arithmetic units, such as comparators, adders, and multiplier [43]. The basis of every complex arithmetic operation is binary addition. Hence, it can be concluded, that binary addition is one of the most important arithmetic operation. The hardware implementation of an adder becomes even more critical due to the expensive carry-propagation step, the evaluation time of which is dependent on the operand word length. The efficient implementation of the addition operation in an integrated circuit is a key problem in VLSI design [58].

Productivity in ASIC design is constantly improved by the use of cell-based design techniques – such as standard cells, gate arrays, and field programmable gate arrays (FPGA), and low-level and high-level hardware synthesis [13]. This asks for adder architectures which result in efficient cell-based circuit realizations which can easily be synthesized. Furthermore, they should provide enough flexibility in order to accommodate custom timing and area constraints as well as to allow the implementation of customized adders.

1.2 Goals

The following goals have been formulated for this research:

- Establish the performance criteria for an adder design which include; speed, area, power, size of input operands, number of input operands, and number of useful results obtained at the output.

- Performance evaluation of conventional adder architectures and compare them, with focus on circuit implementation.
- Study of mathematics involved behind *Flagged Prefix Adders* [8] and performance evaluation of this adder design with different prefix trees.
- Derive the logic to increase the number of input operands a prefix adder can add.
- Design a three-input binary adder utilizing the concept of flag bits, where the third operand is application-independent. The hardware that needs to be incorporated within a binary adder will be adder-independent.
- Performance evaluation of the proposed design with respect to various adder architectures in terms of delay, area, and power.
- Obtain delay and area estimates from synthesis and verify results by application of *Logical Effort* [50] to the proposed design.

1.3 Structure of the Thesis

As a starting point, the basic design criteria are discussed and their implications established in Chapter 2. Optimization techniques have been presented which are utilized for the final adder design. It is substantiated why cell-based combinational adders and their synthesis are important in VLSI design.

Chapter 3 introduces conventional adder architectures which include the Carry - Propagate, Carry - Select, Carry - Skip, and Parallel - Prefix adders [31]. The Carry-Save adder which is a multi-operand adder has also been introduced. It is this adder architecture that is used as a benchmark to evaluate the performance of the proposed technique in this thesis.

Chapter 4 gives a brief introduction to the method of logical effort and its application to digital circuits. This forms the basis of verification of results obtained via synthesis.

The theory of flagged prefix addition [6] is introduced in Chapter 5. It also discusses the applications of flagged prefix adders and the advantages presented by these designs.

Chapter 6 extends the theory of flagged prefix addition to enhance the functionality of a binary adder to three-input addition, not limiting the number of input operands to two. The necessary hardware implementation is derived, initially considering that the third input is a constant. The resulting adder designs are called *Enhanced Flagged Binary Adders (EFBA)* [17]. This is followed by a hardware optimization to enable three-input addition independent of whether the third operand is a constant or a variable. The final adder designs are referred to as *Three Input Prefix Adders (TIFPA)*.

Chapter 7 investigates the performance of conventional, flagged prefix, EFBA, and TIFPA designs theoretically as well as based on simulation results. The method of logical effort is applied to all the designs to verify synthesis results with the analytical results. Conclusions are drawn and potential for future work is presented in Chapter 8.

CHAPTER 2

DESIGN CRITERIA AND IMPLICATIONS

This chapter formulates the motivation for the research presented in this thesis by focusing on questions like: Why is the efficient implementation of binary adders important? What will be the key layout design technologies in the future, and why do cell-based design techniques, such as standard cells, gain more and more importance? Why is hardware synthesis becoming a key issue in VLSI design? How can area, delay, and power measurements of combinational circuits be estimated and optimized? How can the performance and complexity of adder circuits be modeled by taking into account architectural, circuit, layout, and technology aspects?

This chapter summarizes the techniques, advantages, and disadvantages of techniques that are utilized during the design and implementations of digital logic circuits.

2.1 Arithmetic Operations and Units

The tasks of a VLSI chip are the processing of data and the control of internal or external system components. This is typically done by algorithms which are based on logic and arithmetic operations on data items [10]. Applications of arithmetic operations in integrated circuits are manifold. Microprocessors and DSPs typically contain adders and multipliers in their datapath. Special circuit units for fast division and square-root operations are sometimes included as well. Adders, incrementers/decrementers, and comparators are often used for address calculation and flag generation purposes in controllers. ASICs use arithmetic units for the same purposes. Depending on their application, they may even require dedicated circuit components for special arithmetic

operators, such as for finite field arithmetic used in cryptography, error correction coding, and signal processing. Some of the basic arithmetic operations are listed below [58]

- Shift/extension operations
- Equality and magnitude comparison
- Incrementation / Decrementation
- Negation
- Addition/ Subtraction
- Multiplication
- Division
- Square root
- Exponentiation
- Logarithmic Functions
- Trigonometric Functions

For trigonometric and logarithmic functions as well as for exponentiation, various iterative algorithms exist which make use of simpler arithmetic operations. Multiplication, division and square root can be performed using serial or parallel methods. In both methods, the computation is reduced to a sequence of conditional additions/subtractions and shift operations. Existing speed-up techniques try to reduce the number of required addition/subtraction operations to improve their speed. Subtraction corresponds to the addition of a negated operand.

The addition of two *n*-bit binary numbers can be regarded as an elementary operation. The algorithm for negation of a number depends on the chosen number representation [42] and is usually accomplished by bit inversion and incrementation.

Increment and decrement operations are simplified additions with one input operand being constantly 1 or -1. Equality and magnitude comparison operations can also be regarded as simplified additions with only some of the respective addition flags, but no sum bits are used as outputs [58].

This short overview shows that the addition is the key arithmetic operation, which most other operations are based on. Its implementation in hardware is therefore crucial for the efficient realization of almost every arithmetic unit in VLSI. This is in terms of circuit size, computation delay, and power consumption.

Addition is a prefix problem [34], which means that each result is dependent on all input bits of equal or lower magnitude. Propagation of a carry signal from each bit position to all higher bit positions is necessary. Carry - propagate adders [31] perform this operation immediately. The required carry - propagation from the least to the most significant bit results in a considerable circuit delay, which is a function of the word length of the input operands [58].

The most efficient way to speed up addition is to avoid carry propagation thus saving the carries for later processing. This allows the addition of two or more numbers in a very short time, but yields results in a redundant number representation [20]. The redundant representation forms the basis of carry-save adders [20]. They play an important role in the efficient implementation of multi-operand addition circuits. They are very fast, their structure simple, but the potential for further optimization is minimal.

The binary carry-propagate adder therefore, is one of the most often used and most crucial building blocks in digital VLSI design. Various well-known methods exist for speeding up carry - propagation in adders, offering very different performance

characteristics, advantages and disadvantages. Instances of adder architectures targeting important design criteria are listed below.

- Delay: Kogge-Stone parallel prefix adder [32]
- Area: Carry-ripple adder [44]
- Power: 2 - level carry-skip adder [7]
- Size of input operands: Brent-Kung parallel prefix adder [5]
- Number of input operands: Carry-save multi-operand adder [42]
- Number of results at output: Flagged parallel-prefix adder [8]

The performance measure of each algorithm is also dependent on design techniques that are employed to create each circuit. The next section presents the circuit and layout design techniques.

2.2 Circuit and Layout Design Techniques

IC fabrication technologies can be classified into full-custom, semi-custom, and programmable ICs. Further distinctions are made with respect to circuit design techniques and layout design techniques, which are strongly related [58].

2.2.1 Layout-Based Design Techniques. In layout-based design techniques, dedicated full-custom layout is drawn manually for circuits designed at the transistor level. The initial design effort is very high, but maximum circuit performance and layout efficiency is achieved. Full-custom cells are entirely designed by hand for dedicated high performance units, e.g., arithmetic units. The tiled-layout technique can be used to simplify, automate, and parameterize the layout task. For reuse purpose, the circuits and layouts are often collected in libraries together with automatic generators. Mega-cells are full-custom cells for universal functions which need no parameterization. Macro-cells are

used for large circuit components with regular structure and need word-length parameterization. Datapaths are usually realized in a bit-sliced layout style, which allows parameterization of word length and concatenation of arbitrary datapath elements for logic, arithmetic, and storage functions. Since adders are too small to be implemented as macro cells, they are usually realized as data-path elements.

2.2.2 Cell-Based Design Techniques. At a higher level of abstraction, arbitrary circuits can be composed from elementary logic gates and storage elements contained in a library of pre-designed cells. The layout is automatically composed from corresponding layout cells using dedicated layout strategies, depending on the used IC technology. Cell-based design techniques are used in standard-cell, gate-array, sea-of-gates, and field-programmable gate-array (FPGA) technologies. The design of logic circuits does not differ considerably among the different cell-based IC technologies. Circuits are obtained from schematic entry, behavioral synthesis, or circuit generators (structural synthesis). Due to the required generic properties of the cells, more conventional logic styles have to be used for their circuit implementation [58].

The advantages of cell-based design techniques lie in their universal usage, automated synthesis and layout generation for arbitrary circuits, portability between tools and libraries, high design productivity, high-reliability, and high flexibility in floorplanning. This comes at the price of lower circuit performance with respect to speed and area. Cell-based design techniques are mainly used for the implementation of random logic and custom circuits for which no appropriate library component are available and custom implementation proves costly. Cell-based design techniques are widely used in the ASIC design community.

Standard Cells

Standard cells [22] [49] represent the highest performance cell-based technology. The layout of the cells is full-custom, which mandates for full-custom fabrication of the wafers. This in turn enables the combination of standard cells with custom-layout components on the same die. For layout generation, the standard cells are placed in rows and connected through intermediate routing channels. With the increasing number of routing layers and the over-the-cell routing capabilities in modern process technologies, the layout density of standard cells gets close to the density obtained from full-custom layout. The remaining drawback is the restricted use of high-performance circuit techniques [58].

Gate-arrays and sea-of gates

On gate-arrays and sea-of gates, pre-processed wafers with unconnected circuit elements are used. Thus, only metallization used for the interconnect is customized, resulting in lower production costs and faster turnaround times. Circuit performance and layout flexibility is lower than for standard cells, which in particular decreases implementation efficiency of regular structures such as macro-cells.

FPGA Cells

Field-programmable gate-arrays (FPGA) [29] are electronically programmable generic ICs. They are organized as an array of logic blocks and routing channels, and the configuration is stored in a static memory or programmed e.g., using anti-fuses. Again, a library of logic cells and macros allow flexible and efficient design of arbitrary circuits. Turnaround times are very fast making FPGAs the ideal solution for rapid prototyping. On the other hand, low circuit performance, limited circuit complexity, and high die costs

severely limit their area of application [29]. The following section gives a synopsis of the method employed for the completion of this research

2.3 Automated Circuit Synthesis and Optimization

Circuit synthesis denotes the automated generation of logic networks from behavioral descriptions at an arbitrary level. Synthesis is becoming a key issue in VLSI design for many reasons. Increasing circuit complexities, shorter development times, as well as efficient and flexible usage of cell and component libraries can only be handled with the aid of powerful design automation tools. Arithmetic synthesis addresses the efficient mapping of arithmetic functions onto existing arithmetic components and logic gates.

2.3.1 High-Level Synthesis. High-level synthesis or behavioral synthesis allows the translation of algorithmic or behavioral descriptions of high abstraction level down to Register Transfer Logic (RTL) representation, which can be processed further by low-level synthesis tools. High-level arithmetic synthesis makes use of arithmetic transformations in order to optimize hardware usage under given performance criteria. Thereby, arithmetic library components are regarded as resources for implementing the basic arithmetic operations [58].

2.3.2 Low-Level Synthesis. Low-level synthesis or logic synthesis translates an RTL specification into a generic logic network. For random logic, synthesis is achieved by establishing the logic equations for all outputs and implementing them in a logic network.

2.3.3 Data-Path Synthesis. Efficient arithmetic circuits contain very specific structures of large logic depth and high factorization degree. Their direct synthesis from logic equations is not feasible. Therefore, parameterized netlist generators using dedicated

algorithms are used instead. Most synthesis tools include generators for the basic arithmetic functions, such as comparators, incrementers, adders, and multipliers. For other important operations, such as squaring and division, usually no generators are provided and thus synthesis of efficient circuitry is not available. Also the performance of the commonly used architectures varies considerably, which often leads to sub-optimal cell-based circuit implementations [58].

2.3.4 Optimization of Combinational Circuits. The optimization of combinational circuits connotes the automated minimization of a logic netlist with respect to delay, area, and power dissipation measures of the resulting circuit, and the technology mapping. The applied algorithms are very powerful for optimization of random logic by performing steps like flattening, logic minimization [29], timing-driven factorization, and technology mapping. However the potential for optimization is limited for networks with large logic depth and high factorization degree, especially arithmetic circuits. Therefore, only local logic minimization is possible, leaving the global circuit architecture basically unchanged. Thus the realization of well-performing arithmetic circuits relies more on efficient datapath synthesis than on simple logic optimization.

2.3.5 Hardware Description Languages. Hardware description languages (HDL) allow the specification of hardware at different levels of abstraction, serving as entry points to hardware synthesis. Verilog is one of the most widely used and powerful languages. It enables the description of circuits at the behavioral and structural level. Synthesis of arithmetic units is initiated by using standard arithmetic operator symbols in the Verilog code [45] for which the corresponding built-in netlist generators are called by the synthesis tool. The advantages of utilizing Verilog over schematic entry lie in the

possibility of behavioral hardware description, the parameterizablity circuits, and portability of code. Now that it has been established how to design the schematics, the next section focuses on the design criteria for VLSI circuits.

2.4 Circuit Complexity and Performance Measures

One important aspect in design automation is the complexity and performance estimation of a circuit. At a higher design level, this is achieved by using characterization information of the high-level components to be used and by complexity estimation of the interconnect [58]. At gate-level, however, estimation is more difficult and less accurate because circuit size and performance strongly depend on the gate-level synthesis results and on the physical cell arrangement and routing.

Estimates of the expected area, speed, and power dissipation for a compiled cell-based circuit can be found as a function of the operand word length.

2.4.1 Area. Silicon area on a VLSI chip is taken up by the active circuit elements and their interconnections. In cell-based design techniques, the following criteria for area modeling can be formulated [19]:

- Total circuit complexity (GE_{total}) can be measured by the number of *gate equivalents*. (1GE ≡ 1 2-input NAND gate ≡ 4 MOSFETS)
- Circuit Area ($A_{circuit}$) is occupied by logic cells and inter-cell wiring. ($A_{circuit} = A_{cells} + A_{wiring}$)
- Total cell area (A_{cells}) is proportional to the number of transistors or GE_{total} contained in a circuit. This number is influenced by technology mapping. ($A_{cells} \alpha GE_{total}$)
- Wiring area (A_{wiring}) is proportional to the total wire length. ($A_{wiring} \alpha L_{total}$)

- Total wire length (L_{total}) can be estimated from the number of nodes and the average wire length of a node, or more accurately from the sum of cell fan-out and the average wire length of cell to cell connections. The wire lengths also depend on circuit size, circuit connectivity, and layout topology. (L_{total} α FO_{total})
- Cell fan out (FO) is the number of cell inputs a cell output is driving. Fan-in is the number of inputs to a cell, which for many combinational gates is proportional to the size of the cell. Since the sum of the cell fan-out (FO_{total}) of a circuit is equivalent to the sum of the cell fan-in it is also proportional to the circuit size. (FO_{total} α GE_{total})

2.4.2 Delay. Propagation delay in a circuit is determined by the cell and interconnection delays on the critical path. Individual cell and node values are relevant for path delays. Critical path evaluation is done by static timing analysis which involves graph-based search algorithms. Timings are also dependent on temperature, voltage, and process parameters [54].

- Maximum delay ($t_{crit\text{-}path}$) of a circuit is equal to the sum of cell inertial delays, cell output ramp delays, and wire delays on the critical path. ($t_{crit\text{-}path} = \Sigma\epsilon_{crit\text{-}path}((t_{cell} + t_{ramp}) + \Sigma\epsilon_{crit\text{-}path}\, t_{wire})$
- Cell delay (t_{cell}) depends on the transistor-level circuit implementation and the complexity of a cell. All simple gates have comparable delays. Complex gates usually contain tree-like circuit and transistor arrangements, resulting in logarithmic delay to area dependencies. (t_{cell} α $\log(A_{cell})$

- Ramp Delay (t_{ramp}) is the time it takes for a cell output to drive the attached capacitive load, which is made up of interconnect and cell input loads. The ramp delay depends linearly on the capacitive load attached, which in turn depends linearly on the fan-out of the cell. (t_{ramp} α FO_{cell})
- Wire delay (t_{wire}) is the RC-delay of a wire, which depends on the wire length. RC delays are negligible compared to cell and ramp delays for small circuits such as adders. ($t_{wire}=0$)
- A rough delay estimation is possible by considering sizes and with a small weighting factor, fan-out of the cells on the critical path. ($t_{crit\text{-}path}$ α $\Sigma\epsilon_{crit\text{-}path}$ (log (A_{cell}) + kFO_{cell})

2.4.3 Power. An increasingly important parameter for VLSI circuits is power dissipation. Peak power is a problem with respect to circuit reliability which, however, can be dealt with by careful design. On the other hand, average power dissipation is becoming a crucial design constraint in many modern applications, such as high-performance microprocessors and portable applications, due to the heat removal problems and power budget limitations.

The following principles hold for average power dissipation in synchronous CMOS circuits [30]:

- Total power (P_{total}) in CMOS circuits is dominated by the dynamic switching of circuit elements, whereas dynamic short-circuit currents and static leakage are of less importance. Thus, power dissipation can be assumed proportional to the total capacitance to be switched, the square of the supply voltage, the

clock frequency, and the switching activity α in a circuit. (P_{total} = ½ . C_{total} . V^2_{dd} . f_{clk} . α)

- Total capacitance (C_{total}) in a CMOS circuit is the sum of the capacitances from transistor gates, sources, and drains and from wiring. Thus, the total capacitance is proportional to the number of transistors and the amount of wiring, both of which are roughly proportional to circuit size. (C_{total} α GE_{total})
- Supply voltage (V_{dd}) and clock frequency (f_{clk}) can be regarded as constant within a circuit and therefore are not relevant in out circuit comparisons. (V_{dd}, f_{clk} = constant)
- The switching activity factor (α) gives a measure for the number of transient nodes per clock cycle and depends on input patterns and circuit characteristics. In many cases, input patterns to datapaths and arithmetic units are assumed to be random, which result in an average transition activity of 50% on all inputs. Signal propagation through several levels of combinational logic may decrease or increase transition activities, depending on circuit structure. (α = constant)
- For arithmetic units with constant input switching activity, power dissipation is approximately proportional to circuit size. (P_{total} α GE_{total})

2.5 Summary

Arithmetic units belong to the basic and most crucial building blocks in many integrated circuits, and their performance depends on the efficient hardware implementation of the underlying arithmetic operations. Advances in computer-aided design as well as the ever growing design productivity demands tend to prefer cell-based

design techniques and hardware synthesis, also for arithmetic components. The following chapter will discuss several adder designs that formed the basis of such arithmetic components.

CHAPTER 3

ADDER DESIGNS

Addition is the most common arithmetic operation and also serves as the building block for synthesizing all other operations. Within digital computers, addition is performed extensively both, in explicitly specified computation steps and as a part of implicit ones dictated by indexing and other forms of address arithmetic [43]. In simple ALUs due to the lack of dedicated hardware for fast multiplication and division, these latter operations are performed as sequences of additions. Subtraction is normally performed by negating the subtrahend and adding the result to the minuend. This is quite natural, given that an adder must handle signed numbers anyway.

This chapter introduces the basic principles and circuit structures used for the addition of single bits and of two or more multiple binary numbers. Binary carry - propagate addition is formulated as a prefix problem, and the fundamental algorithms and speed-up techniques for the efficient solutions of this problem have been described.

3.1 1-Bit Adders

As the basic combinational addition structure, a 1-bit adder computes the sum of m input bits of the same magnitude. It is also called an *(m,k)* counter (See Fig. 3.1) because it counts the number of 1s at the m inputs (a_{m-1}, a_{m-2}, …, a_0) and outputs a k bit sum (s_{k-1}, s_{k-2}, …, s_0), where $k = \lceil \log(m+1) \rceil$. The arithmetic equation representing the same is as follows [58]

$$\sum_{j=0}^{k-1} 2^j s_j = \sum_{i=0}^{m-1} a_i \tag{3.1}$$

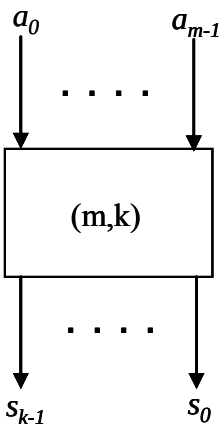

Figure 3.1. (m,k) counter

3.1.1 Half-Adder (2,2)-Counter. The half adder is a *(2,2)* counter. The most significant *sum* bit is called the *carry out*, c_{out} because it carries an overflow to the next higher bit position. Figure 3.2 depicts the logic symbol and the circuit implementation of the half-adder. The corresponding arithmetic and logic equations are as follows in equations 3.2 and 3.3 respectively.

$$2c_{out} + s = a + b$$

$$s = (a + b) \bmod 2 \tag{3.2}$$

$$c_{out} = (a + b) div 2 = 1/2(a + b - s)$$

$$s = a \oplus b$$

$$c_{out} = a \cdot b \tag{3.3}$$

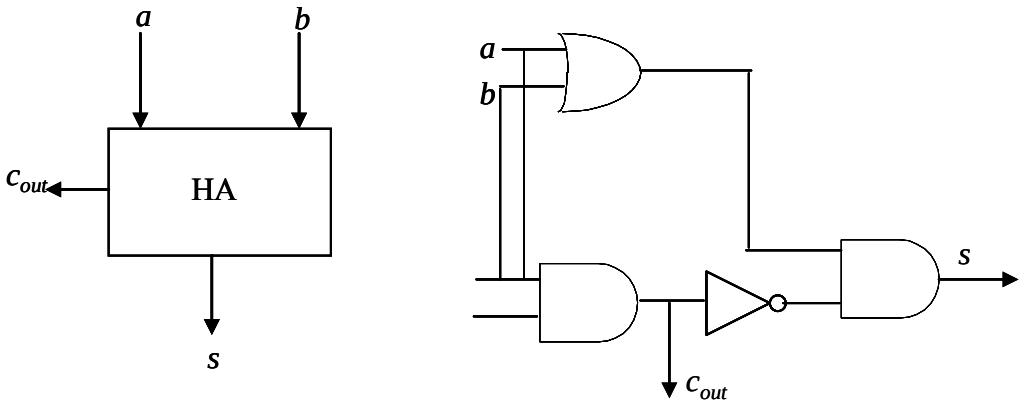

Figure 3.2. Symbol and Logic Circuit for Half Adder

3.1.2 Full Adder (3,2)-Counter. The full adder is a *(3,2)* counter. The third input is called c_{in} since it represents the carry bit from a higher bit position. Important internal signals within a full adder are the *bit generate* and the *bit propagate* signals represented by *g* and *p* respectively. The *generate* signal represents whether the carry signal, 0 or 1 will be generated within the full adder. The *propagate* signal indicates if the *carry- in* at the input of the full adder will be propagated to the *carry-out* of the full adder unchanged. The arithmetic and logic equations for all the signals encountered within a full adder are given in equations 3.4 and 3.5 respectively [58]**.**

$$2c_{out} + s = a + b + c_{in}$$

$$s = (a + b + c_{in}) \bmod 2 \tag{3.4}$$

$$c_{out} = (a + b + c_{in}) div 2 = 1/2(a + b + c_{in} - s)$$

$$g = a \cdot b$$

$$p = a \oplus b$$

$$s = a \oplus b \oplus c_{in} = p \oplus c_{in} \tag{3.5}$$

$$c_{in} = a \cdot b + ac_{in} + bc_{in} = g + p \cdot c_{in}$$

Figure 3.3 depicts the logic symbol and the implementation circuit for a full adder design utilized for the purpose of this thesis.

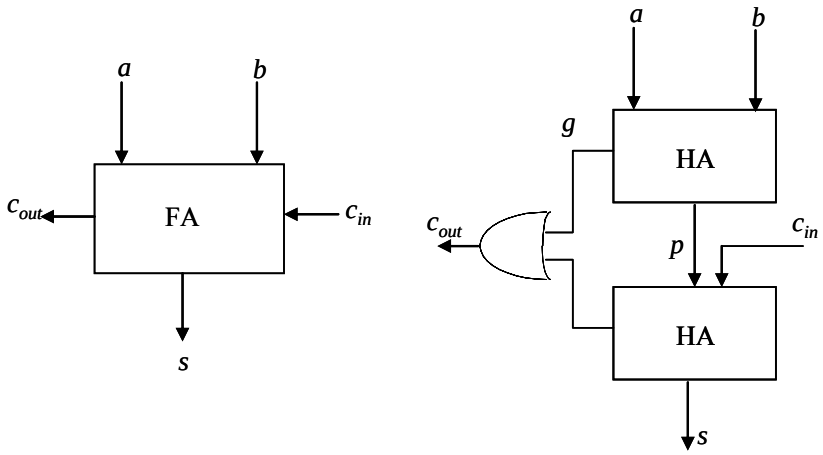

Figure 3.3. Symbol and Logic Circuit for Full Adder

3.2 Carry Propagate Adders

A carry-propagate adder adds 2 *n*-bit operands, $A=(a_{n-1}, a_{n-2}, \ldots, a_0)$ and $B=(b_{n-1}, b_{n-2}, \ldots, b_0)$ and an optional carry-in, c_{in} by performing carry-propagation. The result is an irredundant $(n+1)$-bit number consisting of the *n*-bit sum $S=(s_{n-1}, s_{n-2}, \ldots, s_0)$ and a *carry-out,* c_{out}. Equation 3.6 represents the arithmetic equations for a conventional carry-propagate adder. The logic equations are represented in equation 3.7 [58].

$$2^n c_{out} + S = A + B + c_{in}$$

$$2^n c_{out} + \sum_{i=0}^{n-1} 2^i s_i = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i + c_{in} \tag{3.6}$$

$$g_i = a_i \cdot b_i$$

$$p_i = a_i \oplus b_i \tag{3.7}$$

$$s = p_i \oplus c_i$$

$$c_{i+1} = g_i + p_i \cdot c_i; \qquad i=0, 1,\ldots, n-1$$

Also note that, $c_0=c_{in}$ and $c_n=c_{out}$.

The carry propagate adder can be implemented as a combinational circuit using *n* full adders connected in series (See Fig. 3.4) and is called a Ripple-Carry Adder.

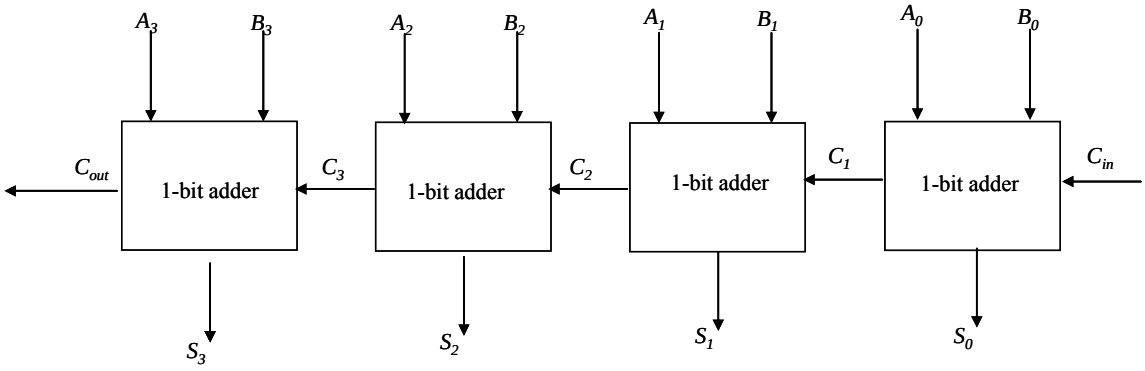

Figure 3.4. Ripple Carry Adder

The following expression depicts the latency of an *n*-bit ripple carry adder[42].

$$T_{ripple-add} = T_{FA}(a_i, b_i \rightarrow c_{i+1}) + (n-2) \times T_{FA}(c_i \rightarrow c_{i+1}) + T_{FA}(c_i \rightarrow s_i) \qquad (3.8)$$

where T_{FA} (input->output) represents the latency of a full adder on the path between its specified input and output. As an approximation to the foregoing, it can be said that the latency of a ripple carry adder is nT_{FA}.

3.3 Carry Select Adders

One of the earliest logarithmic time adder designs is based on the conditional - sum addition algorithm. In this scheme, blocks of bits are added in two ways: assuming an incoming carry of 0 or of 1, with the correct outputs selected later as the block's true carry-in becomes known. This is one of the speed-up techniques that is used in order to reduce the latency of carry propagation as seen with the ripple-carry adder. With each level of selection, the number of known output bits doubles, leading to a logarithmic number of levels and thus logarithmic time addition as opposed to the linear time addition of a carry-propagate adder. An analysis of this scheme is presented next.

The basic problem faced in speeding up carry propagation is the fast processing of a late carry input. Since this carry-in can have only two values, 0 and 1, the two possible addition results can be pre-computed and selected afterwards by the late carry-in using small and constant time. The *n*- bit adder is broken down into groups of *m* bits. Each group of *m*-bits are added utilizing what are called *m*-bit conditional adders. An *m*-bit conditional adder is shown in Figure 3.5 [20].

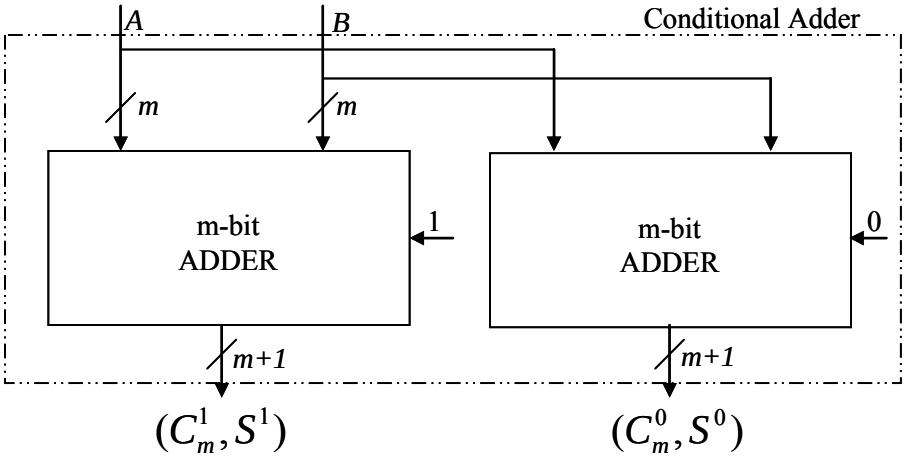

Figure 3.5. *m*-bit Conditional Sum Adder

Equation 3.9 [20] gives a mathematical representation of a conditional sum adder

$$(C_m^0, S^0) = ADD(A, B, c_0 = 0)$$
$$(C_m^1, S^1) = ADD(A, B, c_0 = 1) \tag{3.9}$$

Here, *A*, *B* and *S* are all *m*-bit vectors. C^0_m represents the *m*-bit vector of all the carry outputs assuming $c_0=0$. Similarly, C^1_m represents the *m*-bit vector of all the carry outputs assuming $c_0=1$. S^0 and S^1 represent the *m*-bit vector consisting of all the sum bits assuming $c_0=0$ and $c_0=1$ respectively. Such *m*-bit conditional adders are then combined as shown in Figure 3.6 to obtain a carry-select adder.

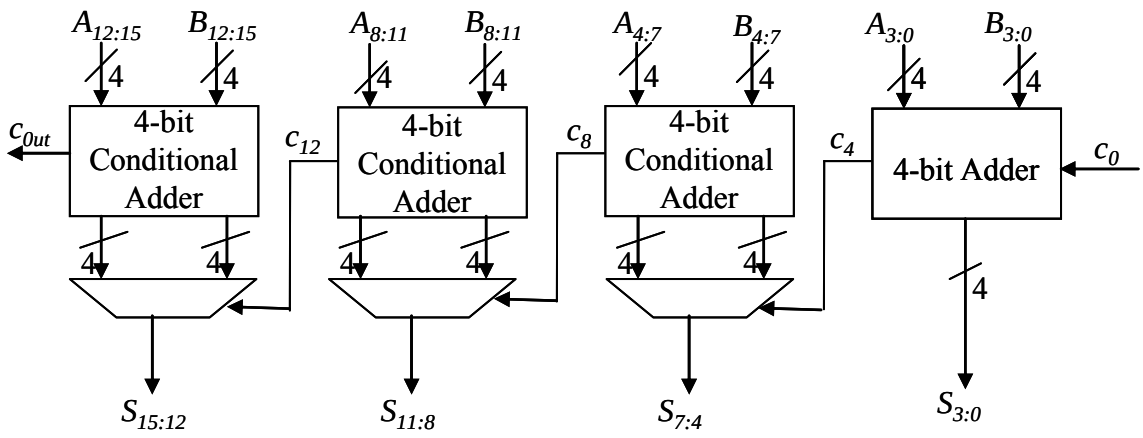

Figure 3.6. Carry-Select Adder

From Figures 3.5 and 3.6, it can be deduced that, each carry - select adder is composed of an initial full adder (**ifa**), at the LSB position, a series of full adders (**bfa**) that generate 2 sets of results for each possible value of the carry-signal. Each group within the carry-select adder will also consist of an initial full adder (**bifa**) at the group LSB, and a final carry-generator (**bcg**) at the group MSB. c_{pb} and c_{tb} denote the carry-out of the previous and the current block respectively. The logical equations for a carry select adder are given as in equations 3.10 – 3.13 [58].

Ifa

$$c_{tb} = a_0 b_0 + a_0 c_0 + b_0 c_0 \tag{3.10}$$

bifa

$$\begin{aligned} g_i &= a_i b_i \\ p_i &= a_i \oplus b_i \\ c_{i+1}^0 &= g_i \\ c_{i+1}^1 &= g_i + p_i \\ s_i^0 &= p_i \\ s_i^1 &= \overline{p_i} \\ s_i &= \bar{c}_{pb} s_i^0 + c_{pb} s_i^1 \end{aligned} \tag{3.11}$$

bfa

$$
\begin{aligned}
g_i &= a_i b_i \\
p_i &= a_i \oplus b_i \\
c_{i+1}^0 &= g_i + p_i c_i^0 \\
c_{i+1}^1 &= g_i + p_i c_i^1 \\
s_i^0 &= p_i \oplus c_i^0 \\
s_i^1 &= p_i \oplus c_i^1 \\
s_i &= \bar{c}_{pb} s_i^0 + c_{pb} s_i^1
\end{aligned}
\tag{3.12}
$$

bcg

$$c_{tb} = c_{i+1}^0 + c_{pb} c_{i+1}^1 \tag{3.13}$$

The delay of carry-select adder therefore is represented as in equation 3.14 [42].

$$T_{carry-sel} = mT_{FA} + (\frac{n}{m} - 1)T_{MUX} \tag{3.14}$$

T_{MUX} represents the delay of a multiplexer. The advantage of utilizing a carry - select adder is that the delay rises logarithmically instead of linearly. However, it also has a high hardware overhead since it requires double the number of carry-propagate adders. In addition, it also requires a set *(n/m-1)* set of multiplexers.

3.4 Carry Skip Adder

The carry - skip adder is obtained by a modification of the ripple - carry adder. The objective is to reduce the worst-case delay by reducing the number of full adder cells through which the carry has to propagate. To achieve this, the adder is divided into groups of *m* bits and the carry into group *j+1* is determined by one of the following two conditions [20]

1. The carry is propagated by group *j*. That is, the carry-out of group *j* is equal to the carry-in of that group. This situation occurs only when the sum of the inputs to that group is equal to $2^m - 1$.

2. The carry is not propagated by the group (that is, it is generated or killed inside the group).

Consequently, to reduce the length of the propagation of the carry, a skip network is provided for each group of *m* bits so that when a carry is propagated by this group, the skip network makes the carry bypass the group. The *m*-bit adder is shown in 3.7, and a network of these modules implementing an *n*-bit adder is indicated in Figure 3.8. The analytical and logical analysis is presented next.

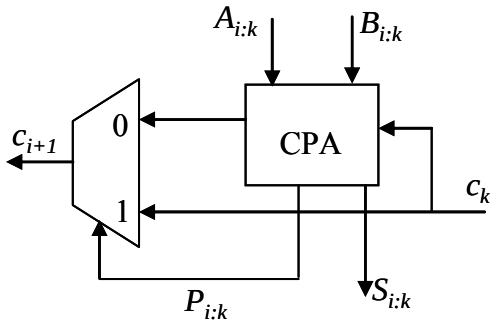

Figure 3.7. Carry-Skip Block

Carry computation for a single bit position, $c_{i+1} = \overline{p}_i g_i + p_i c_i$ can be reformulated for a group of bits as in equation 3.15

$$c_{i+1} = \overline{P}_{i:k} c'_i + P_i c_k \tag{3.15}$$

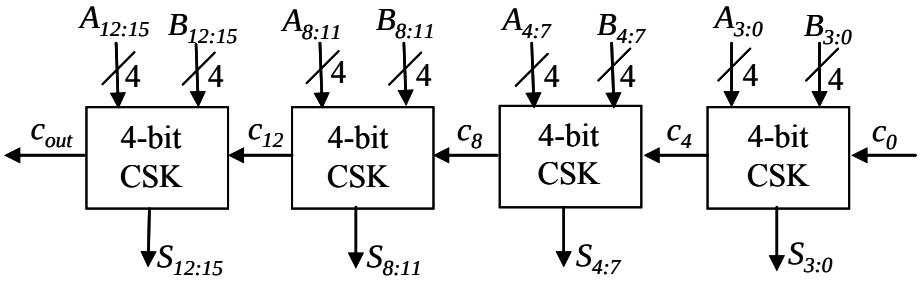

Figure 3.8. Carry-Skip Adder

where $P_{i:k}$ denotes the group propagate of the carry-propagate adder and acts as a select signal in the multiplexer structure and is given by $P_{i:k} = \underset{j=k}{\overset{i-1}{AND}} p_j$. c' is the carry - out of the partial carry propagate adder. Two cases can be distinguished [58]:

- $P_{i:k}=0$: The carry c'_{i+1} is generated within the carry propagate adder and selected by the multiplexer as carry - out c_{i+1}. The carry - in c_k does not propagate through the carry propagate adder to the carry out c_{i+1}
- $P_{i:k}=0$: The carry - in c_k propagates through the carry propagate adder to c'_{i+1} but is not selected by the multiplexer. It skips the carry propagate adder and is directly selected as the carry - out c_{i+1} instead. Thus the combinational path from the carry - in through the carry-propagate adder is never activated.

In other words, the slow carry - chain path from the carry - in to the carry - out through the carry propagate adder is broken by the adder itself or the multiplexer. The resulting carry - skip addition block, therefore is a regular carry propagate adder with a small and constant time delay from $c_k \rightarrow c_{i+1}$ which is why it speeds up carry propagation.

Note that that the multiplexer in this circuit is logically redundant, i.e., the signals c'_{i+1} and c_{i+1} are logically equivalent and differ only in signal delays. The carry - in, c_k has a reconvergent fan-out. This inherent logic-redundancy results in a false longest path which leads from the carry-in through the carry-propagate adder to the carry-out. This poses a problem in automatic logic optimization and static timing analysis. Redundancy removal techniques exist which are based on duplication of the carry-chain in the carry-propagate adder: one carry-chain computes the carry-out without a carry-in, while the other takes the carry-in for calculation of the sum bits. This scheme however signifies a considerable amount of additional logic compared to the redundant carry-skip scheme.

The worst case delay of a carry-skip adder from Figures 3.7 and 3.8 can be formulated as [7]

$$T_{CSK} = mt_c + t_{mux} + \left(\frac{n}{m} - 2\right)t_{mux} + (m-1)t_c + t_s \tag{3.16}$$

Here, T_{CSK} represents the delay of the carry-skip adder, t_{mux} is the propagation delay of the multiplexer, t_c is the delay from the inputs of the full adder to the carry-output and t_s is the delay from the inputs to the sum output. m is the size of the groups into which the adder is divided and n is the size of the input operands. As can be seen from equation 3.16, the delay of the adder depends on the size of the group m. In order to achieve minimum delay, the size of group m can be found by differentiating equation 3.16. The optimal value of m is represented below [26]

$$m_{opt} = \sqrt{t_{mux} / 2t_c n} \tag{3.17}$$

This analysis assumes that all groups are of the same size. However, this does not produce minimum delay. This is due to the fact that, for instance, carries generated in the first group have to traverse more skip networks to get to the last group than carries generated in some internal group. To determine the worst case, delays of all propagation chains need to be compared. A particular chain is initiated in group i and terminates in group j, with $j \geq i$, being propagated by the j-i-1 groups in between. Consequently, if group i, has size m_i,

$$T_{CSK} = \max_{i,j}((m_i + m_j - 1)t_c + (j - i - 1)t_{mux}) + t_{mux} + t_s \tag{3.18}$$

with $\Sigma m_i = n$. The worst case delay, therefore can be reduced by reducing the size of the groups close to the beginning and end, as illustrated in Figure 3.9.

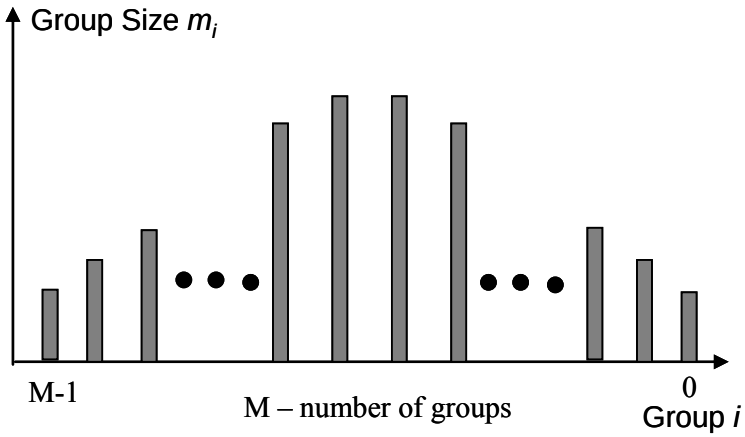

Figure 3.9. Optimal Group Sizes for Carry-Skip Adder

3.5 Carry Save Adder

A row of binary full adders can be viewed as a mechanism to reduce three numbers to two numbers to their sum. Figure 3.10 shows the relationship of a ripple-carry adder for the latter reduction and a carry-save adder for the former. The dotted lines represent the flow of the carry signals within a ripple carry adder. In the carry-save adder, the carry signals are separate outputs which are then passed on to another level of a carry-propagate adder along with the *S* bits to generate the final sum [15].

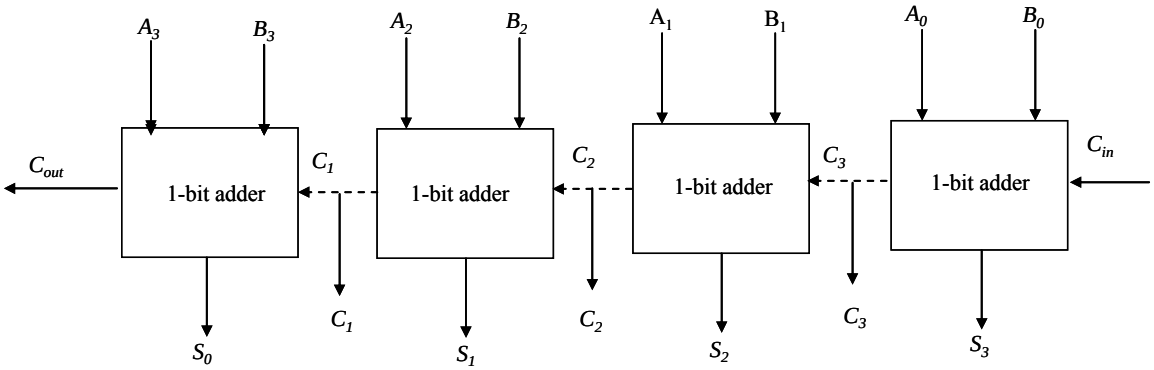

Figure 3.10. Carry-Save Adder

The basic idea is to perform an addition of three binary vectors using an array of one - bit adders without propagating the carries. The sum is a redundant *n* - digit carry - save

number, consisting of two binary numbers *S* (sum bits) and *C* (carry bits). A carry - save adder accepts three binary input operands or, alternatively, one binary and one carry - save operand. It has a constant delay independent of *n*. Mathematically [58],

$$\begin{aligned} 2C + S &= A_0 + A_1 + A_2 \\ \sum_{i=1}^{n} 2^i c_i + \sum_{i=0}^{n-1} 2^i s_i &= \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} 2^i a_{j,i} \\ 2c_{i+1} + s_i &= \sum_{j=0}^{2} a_{j,i} \end{aligned} \tag{3.19}$$

where *i=0,1,….,n-1*

3.6 Parallel Prefix Adders

The addition of two binary numbers can be formulated as a prefix problem. The corresponding parallel-prefix algorithms can be used for speeding up binary addition and for illustrating and understanding various addition principles. This section introduces a mathematical and visual formalism for prefix problems and algorithms.

3.6.1 Prefix Problems. In a prefix problem, *n* outputs ($y_{n-1}, y_{n-2}, \ldots, y_0$) are computed from *n* inputs ($x_{n-1}, x_{n-2}, \ldots, x_0$) using an arbitrary associative operator $\bullet$ as follows [21]:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_1 \bullet x_0 \\ y_2 &= x_2 \bullet x_1 \bullet x_0 \\ &\cdot \\ &\cdot \\ &\cdot \\ y_{n-1} &= x_{n-1} \bullet x_{n-2} \cdots \bullet x_1 \bullet x_0 \end{aligned} \tag{3.20}$$

The problem can also be formulated recursively:

$$\begin{aligned} y_0 &= x_0 \\ y_i &= x_i \bullet y_{i-1}; \end{aligned} \qquad i = 1,2,\ldots,n-1 \tag{3.21}$$

In other words, in a prefix problem, every output depends on all inputs of equal or lower magnitude, and every input influences all outputs of equal or higher magnitude. Due to the associativity of the prefix operator $\bullet$, the individual operations can be carried out in any order. In particular, sequences of operations can be grouped in order to solve the prefix problem partially and in parallel groups (i.e., sequences) of input bits (x_i, x_{i-1}, ….,x_k), resulting in the group variables $Y_{i:k}$. At higher levels, sequences of group variables can again be evaluated, yielding m levels of intermediate group variable $Y^l_{i:k}$. It denotes the prefix result of bits (x_i, x_{i-1}, ….,x_k) at level l. The group variables of the last level m must cover all bits from i to 0 ($Y^m_{i:k}$) and therefore represent the results of the prefix problem [58].

$$\begin{aligned} Y^{0}_{i:i} &= x_i \\ Y^{l}_{i:k} &= Y^{l-1}_{i:j+1} \bullet Y^{l-1}_{j:k}, \qquad k \le j \le i; \qquad l = 1,2,....,m \\ y_i &= Y^{m}_{i:0}; i = 0,1,..., n-1 \end{aligned} \tag{3.22}$$

Note, that for $j=i$, the group variable $Y^{l-1}_{i:k}$ is unchanged (i.e., $Y^l_{i:k} = Y^{l-1}_{i:k}$). Since prefix problems describe a combinational input-to output relationship, they can be solved by logic networks, which will be the major focus in the following text.

Various serial and parallel algorithms exist for solving prefix problems, depending on the bit grouping properties in equation 3.22. They result in very different size and delay performance measures when mapped onto a logic network.

3.6.2 Prefix Algorithms. Two categories of prefix algorithms can be distinguished; the serial prefix, and the tree-prefix algorithms. Tree-prefix algorithms include parallelism for calculation speed-up, and therefore form the category of parallel-prefix algorithms.

Equation 3.21 represents a serial algorithm for solving the prefix problem. The serial-prefix algorithm needs a minimal number of binary • operations and is inherently slow ($O(n)$).

According to equation 3.20, all outputs can be computed separately and in parallel. By arranging the operations • in a tree structure, the computation time for each output can be reduced to $O(\log n)$. However, the overall number of operations • to be evaluated and with that the hardware costs grow with ($O(n^2)$) if individual evaluation trees are used for each output.

As a tradeoff, the individual output evaluation trees can be merged (i.e., common sub-expressions be shared) to a certain degree according to different tree-prefix algorithms, reducing the area complexity to $O(n \log n)$ or even $O(n)$. Binary addition has been presented as a prefix computation next.

The prefix problem of binary carry-propagate addition computes the generation and propagation of carry signals. The intermediate prefix variables can have three different values – i.e., generate a carry 0 (or kill a carry 1), generate a carry 1, and propagate the carry-in. The variables are coded by two bits, *group generate* $G_{i:k}^{l}$, and *group propagate* $P_{i:k}^{l}$. The *generate/propagate* form a signal pair $Y_{i:k}^{l} = G_{i:k}^{l}, P_{i:k}^{l}$ at level *l.* The initial prefix signal pairs ($G_{i:i}^{o}, P_{i:i}^{o}$) corresponding to the bit generate g_i and bit propagate p_i signals have to be computed from the addition input operands in a pre processing step. The prefix signal pairs of level *l* are then calculated from the signals of level *l-1* by an arbitrary prefix algorithm using the binary operation [58]

$$(G_{i:k}^{l}, P_{i:k}^{l}) = (G_{i:j}^{l-1}, P_{i:j}^{l-1}) \bullet (G_{j:k}^{l-1}, P_{j:k}^{l-1}) \tag{3.23}$$

$$= G_{i:j}^{l-1} + P_{i:j}^{l-1} G_{j:k}^{l-1}, P_{i:j}^{l-1} P_{j:k}^{l-1}$$

In the prefix tree, there are n columns, corresponding to the number of input bits. The gates performing the $\bullet$ operation and which work in parallel are arranged in the same row, and similarly, the same gates connected in series are placed in consecutive rows. Thus, the number of rows m corresponds to the number of binary operations to be evaluated in series. The outputs of row l are the group variables $Y_{i:k}^{l} = G_{i:k}^{l}, P_{i:k}^{l}$. The generate/propagate signals from the last prefix stage ($G_{i:i}^{m}, P_{i:i}^{m}$) are used to compute the carry signals c_i. The sum bits, s_i are finally obtained from a post processing step. The parallel prefix adders therefore can be represented as shown in Figure 3.11

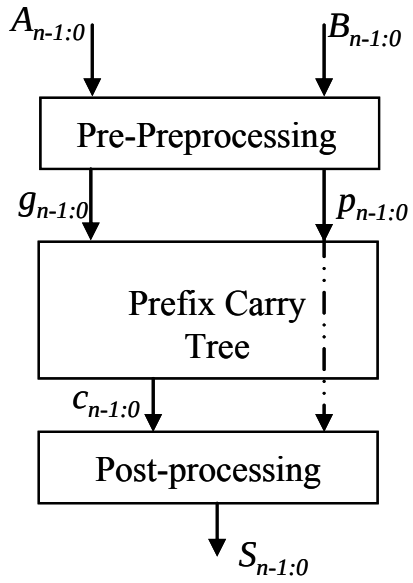

Figure 3.11. Parallel Prefix Adder

Combining equations 3.22 and 3.23 yields the following generate-propagate-based addition prefix formalism [58]:

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i \oplus b_i; \end{aligned} \qquad i = 0,1,....,n-1 \tag{3.24}$$

$$
\begin{aligned}
&(G^0_{i:i}, P^0_{i:i}) = (g_i, p_i) \\
&(G^l_{i:k}, P^l_{i:k}) = (G^{l-1}_{i:j} + P^{l-1}_{i:j} G^{l-1}_{j:k}, P^{l-1}_{i:j} P^{l-1}_{j:k}) \\
&\qquad k \le j \le i; l = 1,2,...,m
\end{aligned} \tag{3.25}
$$

$$c_{i+1} = G^m_{i:0} + P^m_{i:0} c_{in}; \qquad i = 0,1,2,....,n-1$$

$$s_i = p_i \oplus c_i; \qquad i = 0,1,2,.....,n-1 \tag{3.26}$$

$$c_{out} = c_n \tag{3.27}$$

Figures 3.12, 3.13, and 3.14 show the components that will form the parts of the pre-processing stage, the prefix tree, and the post processing stage respectively for each of the three prefix structures that are described in the following sections.

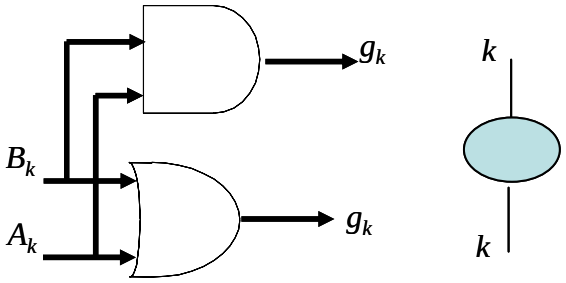

Figure 3.12. Logic and symbol for Pre-processing Gates

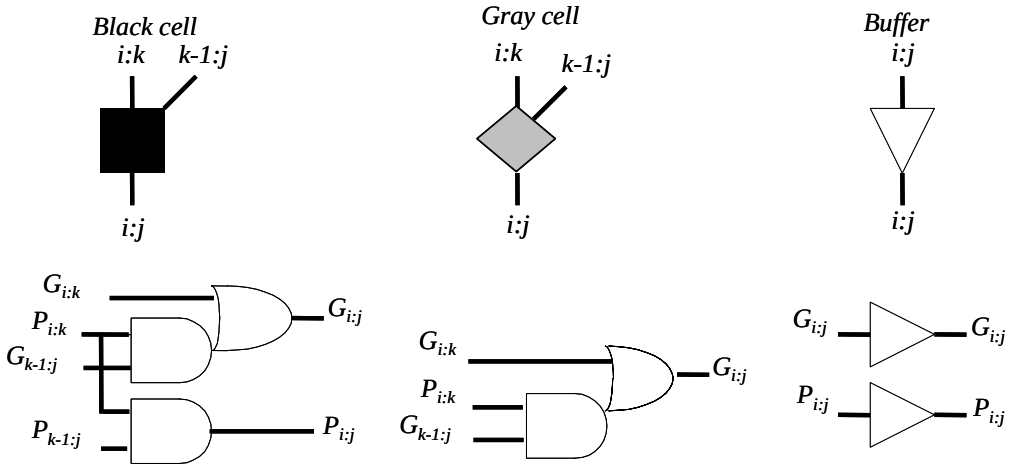

Figure 3.13. Logic and symbol for Prefix Tree Gates

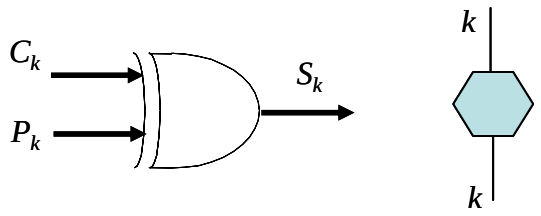

Figure 3.14. Logic and symbol for post-processing gates

It is important to note, for each structure that the bit propagate signals p_i will have to be routed through the prefix structure because they are re-used in the final step for the sum bit calculation. The pre-processing and post-processing stages remain the same in all parallel prefix adders. It is only how the carry computation takes place within the prefix tree that changes and varies among all trees.

3.6.3 Brent - Kung Adder. The Brent - Kung prefix tree [5] gives a simple and regular layout for carry computation. Their construction demonstrates that the addition of *n*-bit numbers can be performed in time $O(\log n)$, using area $O(n \log n)$. The assumptions for this circuit are as follows:

- Gates compute a logical function of two inputs in a constant time.
- An output signal can be divided or fanned out into two signals in constant time.
- Gates have a constant area.
- Wires connecting them have constant minimum width.
- At most two wires can cross at any point**.**

The signals are also assumed to travel along a wire of any length in constant time. It has also been taken into consideration that longer the wire, larger the capacitance of the wire, and thus will require a larger driver. The driver area is typically neglected and does not exceed a fixed percentage of the wire area.

The computation is assumed to be performed in a convex planar region, with inputs and outputs available on the boundary of the region. The measure of the cost of a design is the area rather than the number of gates required. The feature of the approach is to strive for regular layouts in order to design and implementation costs. Figure 3.15 shows the parallel prefix graph for the Brent – Kung design. In general, an *n*-input Brent-Kung prefix graph will have a delay of $2 \log_2 n-2$ levels and a cost of $2n-2- \log_2 n$ cells.

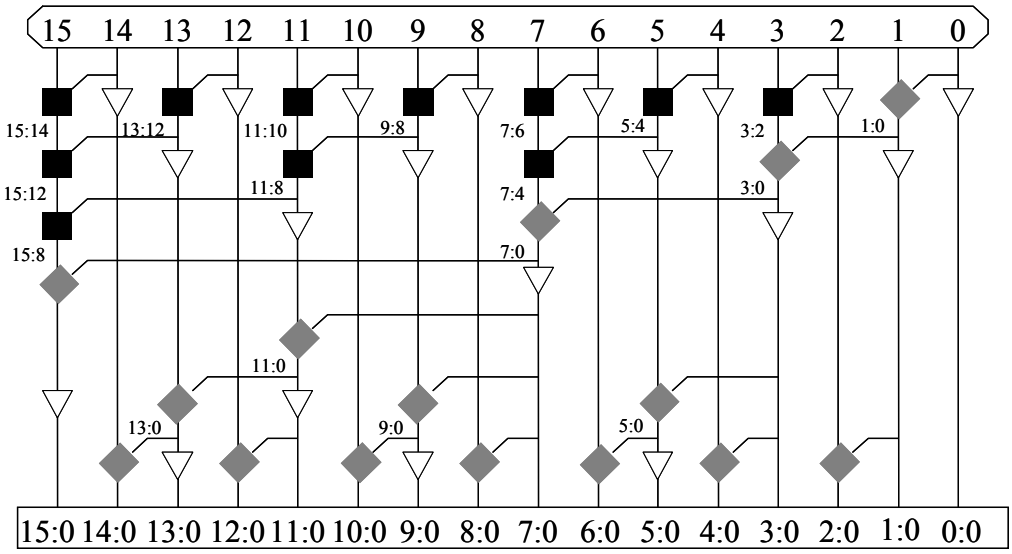

Figure 3.15. Brent – Kung Prefix Tree

3.6.4 Ladner - Fischer Adder. The Ladner - Fischer scheme exploits the associativity property of the • operator by constructing a binary tree of prefix operators. The structure is succinctly represented by the prefix graph of Figure 3.16, which shows the lateral connectivity required between nodes at each stage. Ladner - Fischer introduced the minimum depth prefix graph based on earlier theoretical work by Ofman [34]. The longest lateral fanning wires go from a node to *n/2* other nodes for an *n*-bit adder. Capacitive fan - out loads become particularly large for later levels in the graph as

increasing logical fan - out combines with increasing span of wires. Buffering inverters are added appropriately to support these large loads.

This scheme focuses on two complexity measures.

- The size, is the number of gates within the prefix tree.
- The depth is the maximum number of gates on any directed path

The following assumptions are made to describe the design criteria of the circuit

- Depth of the circuit corresponds to the computation time in a parallel computation environment.
- The size represent the amount of hardware required.

The Ladner-Fischer tree is a circuit of size, $8n + 6$ and depth $4\lceil \log_2 n \rceil + 2$ which is the same as the carry-lookahead adder [34].

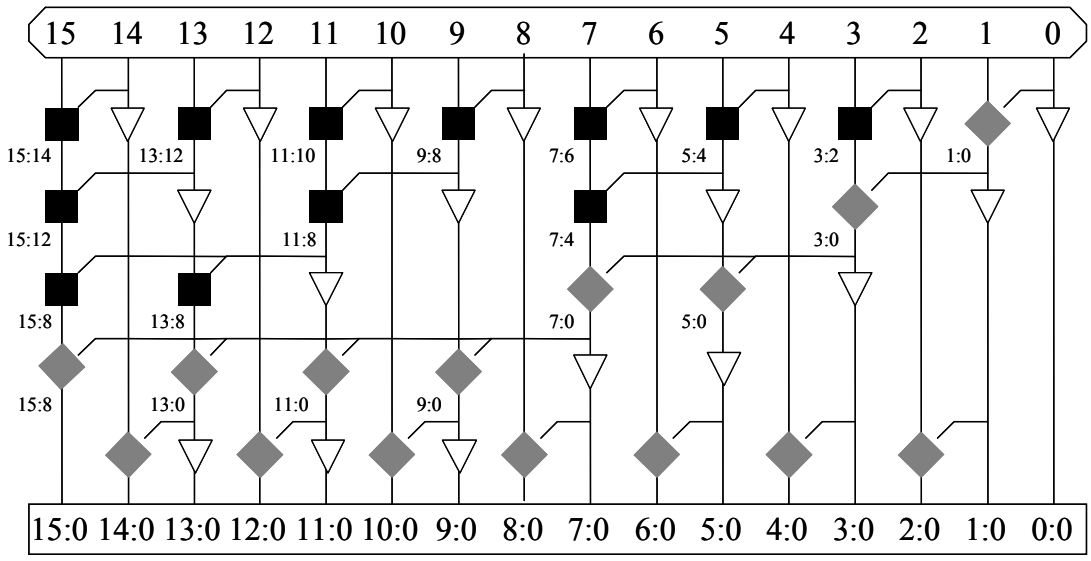

Figure 3.16. Ladner-Fischer Prefix Tree

3.6.5 Kogge - Stone Adder. The prefix tree proposed by Kogge - Stone [32] is based on the concept of recursive doubling. This involves splitting a computation of any function into two equally complex sub-functions whose evaluation can be performed

simultaneously in two separate processors. Successive splitting of each of the sub-functions spreads the computation over more processors**.**

In the case of the Kogge - Stone prefix structure, this is accomplished by spreading the computation of the *Group Generate* signals over • operations and hence the increased number of black cells as depicted in Figure 3.17. The Kogge-Stone adder limits the lateral logical fan - out at each node to unity, but at the cost of a dramatic increase in the number of lateral wires at each level. This is because there is a massive overlap between the prefix sub-terms being pre-computed. The span of the lateral wires remains the same as for the Ladner - Fischer structure, so some buffering is still usually required to accommodate the wiring capacitance, even though the logical fan - out has been minimized.

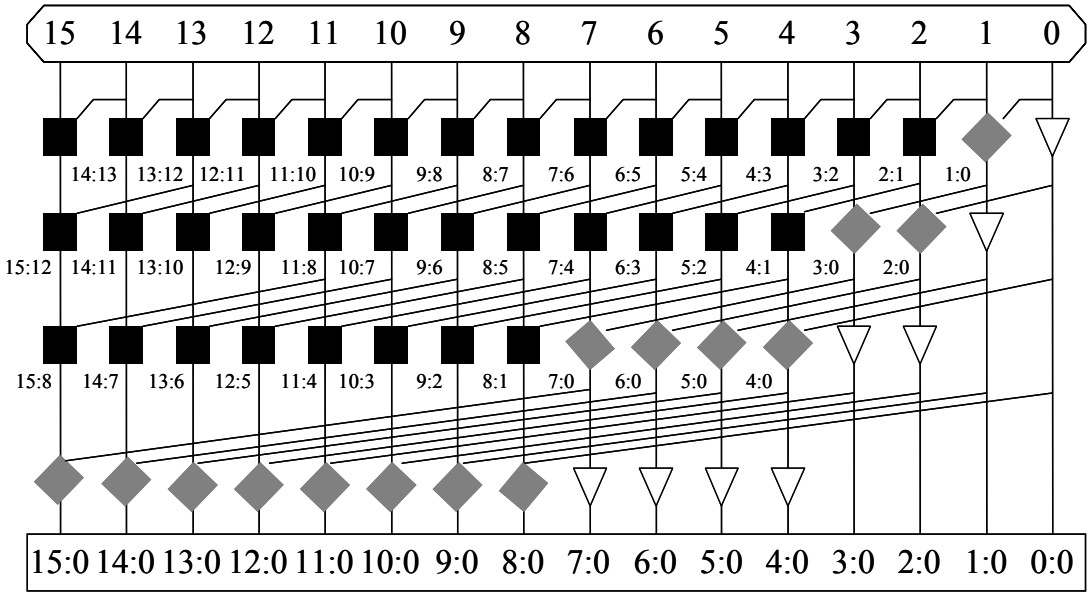

Figure 3.17. Kogge - Stone Prefix Tree

An *n* - input Kogge - Stone parallel prefix graph has a delay of $\log_2 n$ levels and a cost of $n (\log_2 n)-n+1$ cells. The Kogge-Stone parallel prefix graph represents the fastest possible implementation of a parallel prefix computation. However, its cost can be

prohibitive for a large *n*, in terms of both, the number of cells and the dense wiring between them.

3.7 Summary

This chapter summarized the concept behind binary addition, starting from a 1 – bit adder down to the parallel prefix adder designs. It also introduced a multi – operand addition technique implemented by an architecture called carry – save addition. The next chapter focuses on the method of logical effort which will be applied to each adder design that has been described in this chapter to estimate the delay along the critical path of the circuit.

CHAPTER 4

LOGICAL EFFORT

Various choices are available to design a circuit that achieves greatest speed. Several methods can be employed to have a circuit meet certain delay constraints [24]. Some of the questions that need to be answered to make an appropriate choice are [50]:

- Which of the several circuits that produce the same function will be fastest?
- How large should a logic gate's transistors be to achieve least delay?
- How many stages of logic should be used to obtain least delay?

The method of *Logical Effort* [50] is an easy way to estimate delay in a CMOS circuit. The method specifies the proper number of logic gates on a path and the best transistor sizes for the logic gates. Because the method is easy to use, it is ideal for evaluating alternatives in the early stages of a design and provides a good starting point for more intricate optimizations. This chapter describes the method of logical effort.

4.1 Delay in a Logic Gate

The method of logical effort is founded on a simple model of the delay through a single MOS logic gate. The model describes delays caused by the capacitive load that the logic gate drives and by the topology of the logic gate.

The first step in modeling delays is to isolate the effects of a particular integrated circuit fabrication process by expressing all delays in terms of a basic delay unit, τ particular to that process. τ is the delay of an inverter driving an identical inverter with no parasitics. The absolute delay, therefore becomes the product of a unitless delay of the gate *d,* and the delay unit that characterizes a given process

$$d_{abs} = d \,.\, \tau \tag{4.1}$$

The delay incurred by a logic gate is comprised of two components; a fixed part called the *parasitic delay p* and a part that is proportional to the load on the gate's output, called the *effort delay* or the *stage effort f*. The total delay, measured in units of τ , is the sum of the effort and parasitic delays [50]:

$$d = f + p \tag{4.2}$$

The effort delay depends on the load and on properties of the logic gate driving the load. The related terms utilized for these effects are: *logical effort, g* that captures the properties of the logic gate, while the *electrical effort, h* characterizes the load. The effort delay of the logic gate is the product of these two factors

$$f = g \,.\, h \tag{4.3}$$

The logical effort, *g* captures the effect of the logic gate's topology on its ability to produce output current. It is independent of the size of the transistors in the circuit. The electrical effort, *h* describes how the electrical environment of the logic gate affects the performance and how the size of the transistors in the gate determines its load-driving capability. The electrical effort is defined by [50]:

$$h = C_{out}/C_{in} \tag{4.4}$$

where C_{out} is the capacitance that loads the output of the logic gate and C_{in} is the capacitance presented by the input terminal of the logic gate. Combining equations 4.2 and 4.3 the basic equation that models the delay through a single logic gate, in units of τ can be obtained:

$$d = g \,.\, h+p \tag{4.5}$$

This equation shows that the logical effort, g and electrical effort h both contribute to delay in the same way. This formulation separates τ, g, h, and p, the four contributors to delay. The process parameter τ represents the speed of the basic transistors. The parasitic delay p expresses the intrinsic delay of the gate due to its own internal capacitance, which is largely independent of the size of the transistors in the logic gate. The electrical effort, h combines the effects of the external load, which establishes $C_{out,}$ with the sizes of the transistors in the logic gate, which establish $C_{in.}$ The logical effort, g expresses the effects of circuit topology on the delay free considerations of loading or transistor size. Logical effort, therefore proves useful because it depends only on circuit topology.

Logical effort values for basic CMOS logic gates utilized for this thesis are shown in Table 4.1 [50].

Table 4.1. Logical Effort for inputs of static CMOS gates

	Number of inputs					
Gate Type	**1**	**2**	**3**	**4**	**5**	**n**
Inverter	1					
NAND		4/3	5/3	6/3	7/3	(n + 2)/3
NOR		5/3	7/3	9/3	11/3	(2n + 1)/3
Multiplexer		2	2	2	2	2
XOR		4	12	32		

An inverter driving an exact copy of itself experiences an electrical effort of 1.

The logical effort of a logic gate tells how much worse it is at producing output current than is an inverter, given that each of its inputs may present only the same input capacitance as the inverter. Reduced output current means slower operation, and thus the logical effort is how much more slowly it will drive a load than would an inverter. Equivalently, logical effort is how much more input capacitance a gate must present in

order to deliver the same output current as an inverter. From Table 4.1, it can be concluded that larger or more complex gates have a larger logical effort. Moreover, the logical effort of most logic gates grows with the number of inputs to the gate. Larger or more complex logic gates will thus exhibit greater delay. Designs that minimize the number of stages of logic will require more inputs for each logic gate and thus have larger logical effort. Designs with fewer inputs and thus less logical effort per stage may require more stages of logic. This is a tradeoff.

The electrical effort, h is a ratio of two capacitances. The load driven by a logic gate is the capacitance of whatever is connected to its output, resulting in slowing the circuit down. The input capacitance of the circuit is a measure of the size of its transistors. Electrical effort is usually expressed as a ratio of transistor widths rather than actual capacitances. This is due to the fact that the capacitance of a transistor gate is proportional to its area. Assuming that all transistors have the same minimum length, the capacitance of a transistor gate is proportional to its width. Most logic gates drive other logic gates and therefore it is safe to express both, C_{in} and C_{out} in terms of transistor widths.

The parasitic delay of a logic gate is fixed, independent of the size of the logic gate and of the load capacitance it drives. This delay is a form of overhead that accompanies any gate. The principal contribution to parasitic delay is the capacitance of the source and drain regions of the transistors that drive the gate's output. Table 4.2 presents crude estimates of parasitic delay for a few logic gate types [50]. The parasitic delays are given as multiples of the parasitic delay of an inverter, denoted as p_{inv}. A typical value for p_{inv} is 1.0 delay units.

Table 4.2. Estimates of Parasitic Delay of Logic Gates

Gate Type	Parasitic Delay
Inverter	p_{inv}
n-input NAND	np_{inv}
n-input NOR	np_{inv}
n-way multiplexer	$2np_{inv}$
XOR, XNOR	$4p_{inv}$

The delay model of a single logic gate, as represented in Equation 4.5, is a simple linear relationship. Figure 4.1, shows this relationship graphically: delay appears as a function of electrical effort for an inverter and for a 2-input NAND gate. The slope of each line is the logical effort of the gate; its intercept is the parasitic delay. The graph shows that the total delay can be adjusted by adjusting the electrical effort or by choosing a logic gate with a different logical effort. Once a gate type has been chosen, the parasitic delay stays fixed and optimization procedures cannot be applied to reduce it.

4.2 Multistage Logic Networks

The method of logical effort reveals the best number of stages in a multistage network and how to obtain the least overall delay by balancing the delay among the stages. The notions of logical and electrical effort generalize easily from individual gates to multistage paths.

The logical effort along a path compounds by multiplying the logical efforts of all the logic gates along the path. The *path logical effort* is denoted by G. Te subscript i indexes the logic stages along the path [50]:

$$G = \prod g_i \tag{4.6}$$

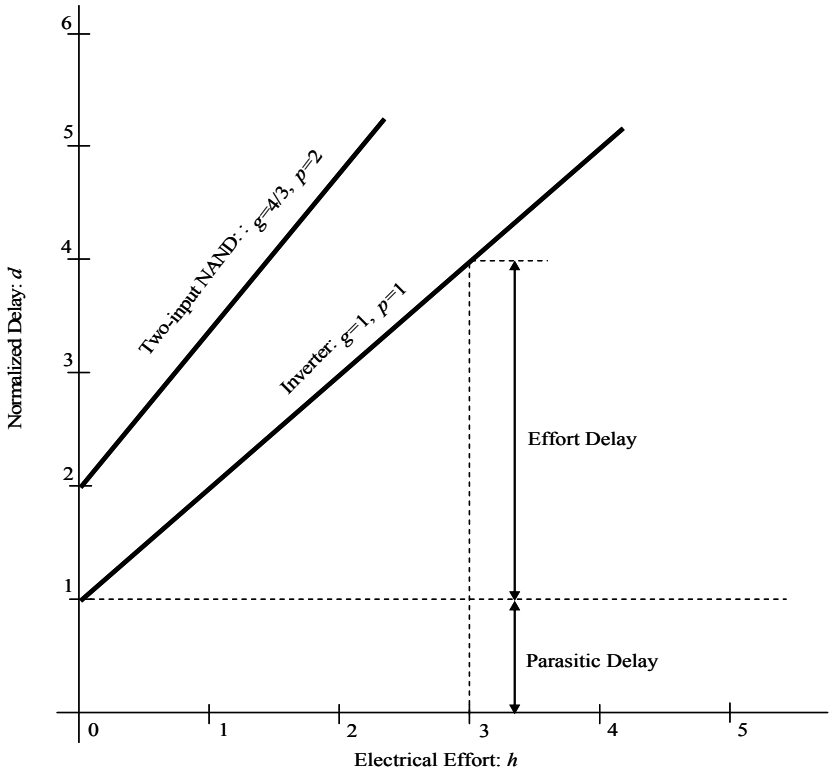

Figure 4.1. Electrical Effort vs. Delay

The electrical effort along a path through the network is simply the ratio of the capacitance that loads the last logic gate in the path to the input capacitance of the first gate in the path. H denotes the electrical effort along a path. C_{out} and C_{in}, in this case refer to the output and input capacitances of the path as a whole.

A new kind of effort, named *branching effort* is introduced to account for fanout within a network. When fanout occurs within a logic network, some of the available drive current is directed along the path being analyzed, and some is directed off that path. The branching effort, b at the output of a logic gate is defined as [50]:

$$b = \frac{C_{on-path} + C_{off-path}}{C_{on-path}} = \frac{C_{total}}{C_{useful}} \tag{4.7}$$

where $C_{on\text{-}path}$ is the load capacitance along the path being analysed and $C_{off\text{-}path}$ is the capacitance on connections that lead off the path. If the path does not branch, the

branching effort is considered as one. The branching effort along the entire path is denoted by B and is the product of the branching effort at each of the stages along the path.

$$B = \prod b_i \tag{4.8}$$

With the definitions of logical, electrical, and branching effort along a path, the *path effort F* can be defined as:

$$F = G \,.\, B \,.\, H \tag{4.9}$$

Although it is not a direct measure of delay along the path, the path effort holds the key to minimizing the delay. It is necessary to note that the path effort depends only on the circuit topology and loading and not upon the sizes of the transistors used in logic gates embedded within the network. Moreover, the effort is unchanged if inverters are added to or removed from the path because the logical effort of an inverter is one. The path effort is related to the minimum achievable delay along the delay and makes the delay calculation easy.

The path delay D is the sum of the delays of each of the stages of logic in the path. The sum of the *path effort delay* D_F and the *path parasitic delay* P gives the path delay D [50].

$$D = \sum d_i = D_F + P \tag{4.10}$$

The path effort delay, D_F is given by:

$$D_F = \sum g_i \,.\, h_i \tag{4.11}$$

and the path parasitic delay, P is:

$$P = \sum p_i \tag{4.12}$$

Optimizing the design of an N-stage logic network follows from the principle: *The path delay is least when each stage in the path bears the same stage effort* [50]. This minimum delay is achieved when the stage effort is:

$$\hat{f} = g_i.h_i = F^{1/N} \tag{4.13}$$

The hat over any symbol has been used to indicate an expression that achieves minimum delay. Combining these equations, the principal result of the method of logical effort is obtained [50], which is an expression for the minimum delay achievable along the path.

$$\hat{D} = N.F^{1/N} + P \tag{4.14}$$

To equalize the effort borne by each stage on a path, and therefore achieve the minimum delay along the path, appropriate transistor sizes must be chosen for each stage of logic along the path. Equation 4.13 shows that each logic stage should be designed with electrical effort

$$\hat{h}_i = \frac{F^{1/N}}{g_i} \tag{4.15}$$

From the relationship above, transistor sizes of the gates along a path can be determined. It is necessary to start at the end of the path and work backward, applying the capacitance transformation:

$$C_{in_i} = \frac{g_i}{\hat{f}} \tag{4.16}$$

This determines the input capacitance of each gate, which can then be distributed appropriately among the transistors connected to the input.

4.3 Choosing the Best Number of Stages

The delay equations of logical effort can be solved to determine the number of stages, $\hat{N}$, that achieves the minimum delay. Table 4.3 [50] provides some results. Utilizing Table 4.3 to select the number of stages that gives the least delay, may indicate that it is necessary to add stages to a network. If a path uses a number of stages that is not quite optimal, the overall delay is usually not increased very much. The table is accurate only if increasing or decreasing the number of stages by adding or removing inverters is being considered. This table assumes that stages being added or removed have a parasitic delay equal to that of an inverter.

4.4 Summary of the Method

The method of logical effort is a design procedure for achieving the least delay along a path of a logic network. The principal expressions of logical effort are summarized in Table 4.4 [50]. The procedure is

1. Compute the path effort $F=G.B.H$ along the path of the network that is being analyzed. The path logical effort G is the product of the logical efforts of the logic gates along the path. Table 4.1 needs to be utilized to obtain the logical effort of each individual gate. The branching effort B is the product of the branching effort at each stage along the path. The electrical effort, H is the ratio of the capacitance loading the last stage of the network to the input capacitance of the first stage of the network.
2. Compute the path effort $F=G.B.H$ along the path of the network that is being analyzed. The path logical effort G is the product of the logical efforts of the logic gates along the path. Table 4.1 needs to be utilized to obtain the logical

effort of each individual gate. The branching effort *B* is the product of the branching effort at each stage along the path. The electrical effort, *H* is the ratio of the capacitance loading the last stage of the network to the input capacitance of the first stage of the network.

Table 4.3. Best Number of Stages for Path Efforts

Path Effort	Best Number of Stages	Minimum Delay	Stage Effort (range)
0		1.0	
	1		0 – 5.8
5.83		6.8	
	2		2.4 – 4.7
22.3		11.4	
	3		2.8 – 4.4
82.2		16.0	
	4		3.0 – 4.2
300		20.7	
	5		3.1 – 4.1
1090		25.3	
	6		3.2 – 4.0
3920		29.8	
	7		3.3 – 3.9
14200		34.4	
	8		3.3 – 3.9
51000		39.0	
	9		3.3 – 3.9
184000		43.6	
	10		3.4 – 3.8

Table 4.4. Summary of Terms and Equations for Logical Effort

Term	Stage Expression	Path Expression
Logical Effort	g	$G = \prod g_i$
Electrical Effort	$h = C_{out}/C_i$	$H = C_{out\text{-}path}/C_{in\text{-}path}$
Branching Effort	-	$B = \prod b_i$
Effort	$f = g.h$	$F = G.B.H = \prod f_i$
Effort Delay	f	$D_F = \sum f_i$
Number of stages	1	N
Parasitic Delay	p	$P = \sum p_i$
Delay	$d = f + p$	$D = D_F + P$

3. Utilize Table 4.3 to find out the appropriate number of stages for l east delay
4. Estimate minimum delay utilizing equation 4.14. For the work presented in this thesis, different architectural approaches to a design problem are being compared and hence, the analysis will stop at this stage.
5. Add or remove stages if necessary.
6. Compute the effort to be borne by each stage utilizing equation 4.13.
7. Starting at the last logic stage in the path, work backward to compute transistor sizes for each of the logic gates utilizing equation 4.16 for each stage.

4.5 Summary

Chapter 3 and Chapter 4 form the background of this research. All adder architectures that have been investigated and incorporated with the proposed hardware to get the new algorithm for three – operand addition were presented in Chapter 3. Chapter 4 summarizes the method of logical effort that is applied to each adder design that has

been implemented in order to estimate the delay. The results that are obtained utilizing this analysis are compared with the post – implementation results in order for verification. Chapter 5 will start talking about Flagged Prefix Addition based on parallel prefix adders presented in Chapter 3.

CHAPTER 5

FLAGGED PREFIX ADDITION

In Chapter 3, it was demonstrated that the parallel-prefix scheme for binary addition is universal and the most efficient adder architectures are based on it. Furthermore, this scheme presents some additional properties which can be utilized for the implementation of special adders and related units. There are a variety of applications that require arithmetic operations such as, finding the absolute difference between two numbers or conditionally incrementing the sum of two numbers [2]. For instance, sign-magnitude adders and motion estimation algorithms in video compression applications use the former, and Reed-Solomon coders use the latter [6]. One way to perform these computations is to use a pair of adders to calculate two speculative results and select the appropriate result [35]. The scheme can be represented as shown in Figure 5.1. This approach is also called the dual adder approach and is very similar to the carry-select adder discussed in Chapter 3. The carry-select adder suffers from a large fan-out and is also known to sub - optimal in speed [16].

The design proposed in [8] focuses on the prefix type carry lookahead adders [20] due to the flexibility that the prefix trees offer in terms of fan-out loading and wire densities, both of which issues are of crucial importance in deep submicron geometry of VLSI circuits [14]. The computations mentioned above are performed by slightly modifying the prefix tree of the parallel prefix adder so as to provide a set of *flag* bits. The modified adder is called a *flagged prefix adder* [8].

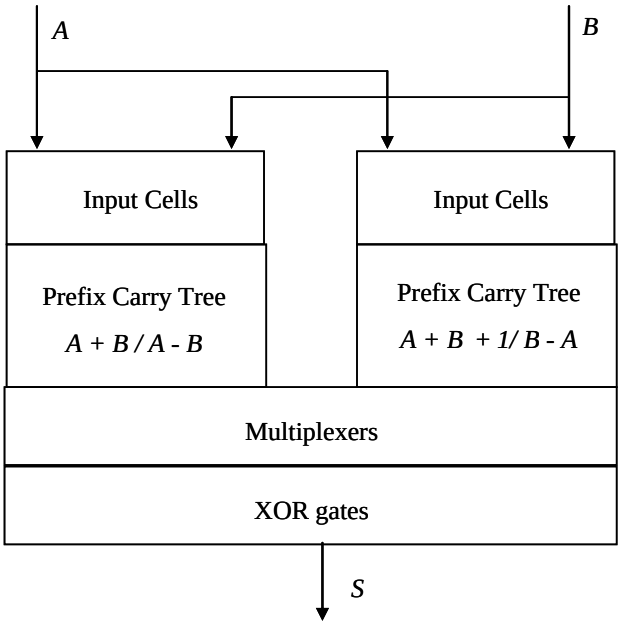

Figure 5.1. Dual Adder Design

5.1 Background Theory of Flagged Prefix Addition

The theory behind a flagged prefix adder operation rests on two simple and well known bit manipulations [2]:

- Incrementing a number by inverting a string of bits from its least significant bit (l.s.b) upwards.
- One's complementing a number by inverting all its bits.

A simple technique for incrementing a two's complement number is to invert all the bits from the l.s.b upwards upto and including the first zero [29]. The bits to be inverted may be *flagged* in parallel with the addition process by a procedure known as *Trailing – 1's bit* prediction [33]. Trailing – 1's bit prediction examines a pair of addends, *A*, and *B* and generated flag bits, *F*, to indicate those bits that should be inverted [8]. The i^{th} flag bit is defined as

$$f_i = (a_{i-1} \oplus b_{i-1}) \cdot (a_{i-2} \oplus b_{i-2}) \cdot \ldots \cdot (a_1 \oplus b_1) \cdot (a_0 \oplus b_0) = (a_{i-1} \oplus b_{i-1}) \cdot f_{i-1} \qquad (5.1)$$

The initial condition is defined with $f_0 = 1$. The example below shows how A+B+1 is derived utilizing Trailing 1's bit prediction:

A=00001001 B=01001110

0 0 0 0 1 0 0 1 +

0 1 0 0 1 1 1 0

---

0 1 0 1 0 1 1 1 $R=A+B$

---

0 0 0 0 1 1 1 1 F

0 1 0 1 1 0 0 0 $S=A+B+1=R \oplus F$

The flag bits may also be utilized to decrement the sum *A+B+1*. The flag bits indicate those bits that were inverted by the addition of the extra 1.

If all the bits of a two's complement number, *N*, are inverted, the two's complement number, *-(N+1)* is returned. This is the same as the one's complement of a number. Combining this operation with the Trailing 1's bit prediction technique give the following possibilities [8]

- $A+B$ invert none of the sum bits $\rightarrow A+B$
- $A+B$ invert fagged sum bits $\rightarrow A+B+1$
- $A+B$ invert all the sum bits $\rightarrow -(A+B+1)$
- $A+B$ invert unflagged sum bits $\rightarrow -(A+B+2)$

Replacing *B* by $-B$ = *-(B+1)* leads to the following alternative set of possibilities, where inverting the flagged sum bits decrements rather than incrementing the sum:

- $A - B$ invert none of the sum bits $\rightarrow A-B-1$
- $A - B$ invert fagged sum bits $\rightarrow A-B$
- $A - B$ invert all the sum bits $\rightarrow B-A$

- A - B invert unflagged sum bits $\rightarrow B - A - 1$

[8] describes the modifications that need to be made to the parallel prefix adder to generate all 8 results from the same adder design.

5.2 Implementation of a Flagged Prefix Adder

The flag bits $f_{i,}$ are related in a very simple manner to the *Group Propagate* signals, used in prefix adders [8]

$$f_i = P_{i-1:0} \tag{5.2}$$

Hence the flag bits are readily available from a prefix carry lookahead adder even as it calculates the final sum bits *S.* A pair of *invert enable* bits are utilized to invert the appropriate sum bits to derive the required result. The flagged prefix adder's two invert enable bits are: *cmp*, which enables the inversion of the flagged bits, and *incr* that enables the inversion of the unflagged bits. The two enable bits permit any one of the four possible results to be derived from the one addition (or subtraction), as described in Table 5.1. The block diagram of a flagged prefix adder [8] is therefore modified as shown in Figure 5.2.

Table 5.1. Selection Table for a Flagged Prefix Adder [8]

cmp	***incr***	**Result (Addition)**	**Result (Subtraction)**
0	0	*A + B*	*A – B – 1*
0	1	*A + B + 1*	*A – B*
1	0	*-(A + B + 1)*	*B – A*
1	1	*-(A + B + 2)*	*B – A – 1*

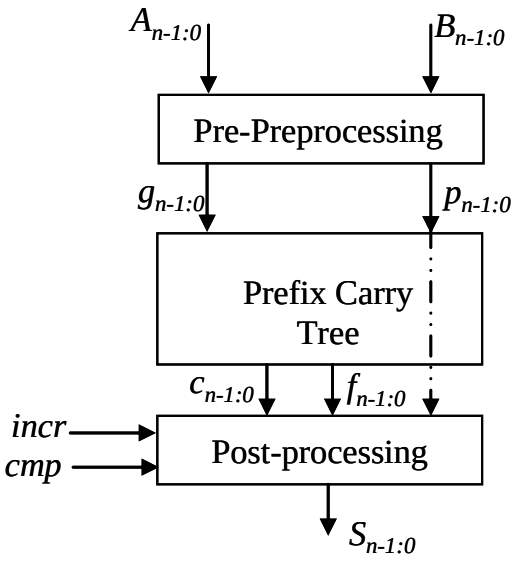

Figure 5.2. Flagged Prefix Adder

5.3 Modifications to a Prefix Adder

Three modifications to the ordinary prefix adder are required so that the alternative results may be derived from the sum [9]:

- *bit propagate* signals, p_i need to be derived from the prefix carry tree.
- *Group Propagate* signals $P_{i-1:0\ I}$ must be derived.
- a pair of *invert enable* signals must be provided to control the inversion of the sum bits.

The *bit propagate* signals are obtained in the pre-processing stage of the parallel prefix adder. However, extra buffering is required, owing to its large fan-out. The *flag bits* are the *Group Propagate* signals, $P_{i-1:0\ I}$ which are not ordinarily computed in a prefix adder. Instead the carry signals, c_i are provided by the *Group Generate* signals, $G_{i-1:0\ I}$ .The *Group Generate* signals are derived as

$$G_{i-1:0} = G_{i-1:k} + P_{i-1:j} \cdot G_{j-1:0} \tag{5.3}$$

where j is some integer, $j < i$. The i^{th} flag bit may be derived simultaneously as

$$P_{i-1:0} = P_{i-1:j} \cdot P_{j-1:0} \tag{5.4}$$

where $P_{j-1:0}$ bit is available from the same cell that provides $G_{j-1:0}$. In a parallel prefix adder, this can be easily achieved by converting the gray prefix cells to black prefix cells. A minor delay is incurred due to the extra fan-out demands on the *Group Propagate* signals, $P_{i-1:j}$. A straightforward circuit implementation of a flagged inversion cell is shown in Figure 5.3 [8]. It consists of 3 XOR gates and a multiplexer, and adds one multiplexer delay to the critical path of the adder, due to the early availability of the two enable signals and the p_i bits.

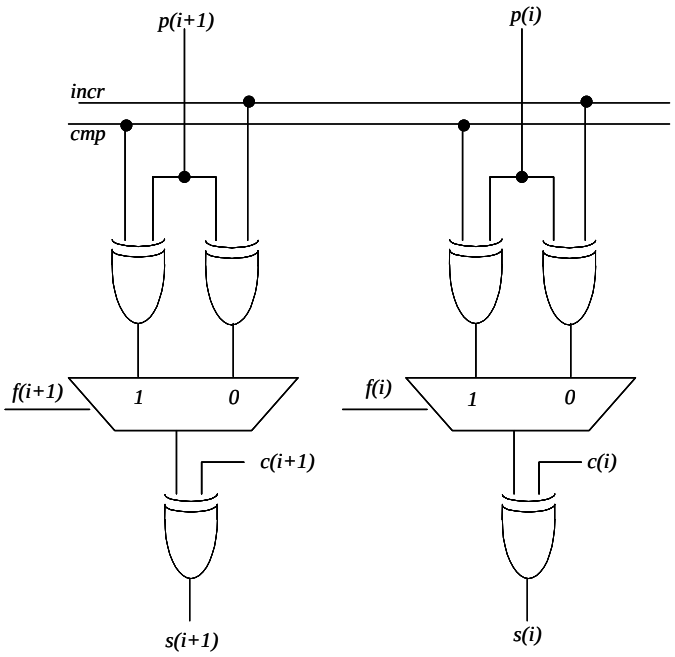

Figure 5.3. Flagged Inversion Cells

5.4 Delay Performance of a Flagged Prefix Adder

The flagged prefix adder will be approximately one XOR delay and one buffer delay slower than a standard prefix adder with the same topology prefix tree. Alternatively, it will have the same speed as a dual adder, comprising a pair of standard prefix adders whose prefix carry-tree outputs are multiplexed according to the values of

incr and *cmp* and then XOR'd with the bit propagate signals, p_i , to return the sum outputs, s_i. The differences between a flagged prefix adder and a standard prefix adder are:

- gray cells in the prefix carry tree are replaced by black cells.
- some extra output logic is added to invert the selected sum signals.
- the *incr* and the *cmp* signals need to be buffered owing to their large fan-out loads.

Black cells and gray cells have the same delay characteristics. Consequently, replacing gray cells by black cells will add no discernable delay to the prefix carry tree.

A straightforward circuit implementation of an output cell implementing the coding scheme of Table 5.1 is shown in Figure 5.4 [6].

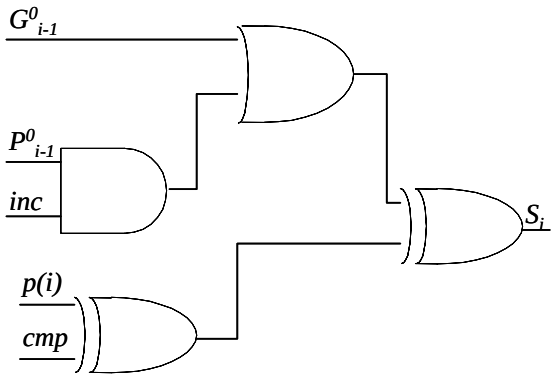

Figure 5.4. Output Cell Logic for the Flagged Prefix Adder

Referring to Figure 5.4, the output cell adds only 1 XOR delay to the critical path of a standard prefix adder, assuming *incr* and *cmp* become available before or at the same time as the *Group Generate* signals. A prefix adder's sum outputs are taken from the XOR gate combining p_i with $G_{i-1:0}$. On the other hand, the flagged prefix adder's outputs are derived by XOR'ing the standard adder's sum outputs with the output of the AND – OR combination, f_i. In CMOS, the XOR gate is slower than a complex AND – OR gate

[11] so that the only extra delay is due to the output XOR gate. The *incr* and *cmp* signals will need to be buffered owing to the large fan-out loads that must be driven. Such buffering will add one buffer delay to the flagged prefix adder's operation.

5.5 Fixed Point Arithmetic Applications

The flagged prefix adder affords the efficient implementation of a number of difficult fixed-point computations. In particular, it may be used to good effect in the following applications [6]

- Mod $2^n - 1$ addition
- Absolute value calculation
- Sign Magnitude Addition

5.5.1 Mod $2^n - 1$ Adder. Modulo adders perform the calculation [19]:

$$S = (A + B) \bmod M \tag{5.5}$$

where M is referred to as a modulus. Such adders find applications in residue number systems, cryptography, and Reed-Solomon coding. If M is of form $M = 2^w - 1$, the modulo addition may be readily implemented as an n-bit flagged prefix adder.

Modulo $2^n - 1$ adders may be designed efficiently by using an *end-around-carry* mechanism: if the sum exceeds the modulus so that $M < S < 2M$, the adder's most significant bit c_i is set high. Then the modulo reduction may be effected by reducing the carry bit, whose significance is 2^n, to +1, and using this to increment the lower n bits of the sum. A flagged prefix adder can implement this operation straightforwardly by connecting the most significant carry output, $c_n = G_{i-1:0}$, to the *incr* control input to perform a late increment when needed.

If *S=M,* all *n* sum bits are set high and should be modulo reduced or simply inverted-to yield 0. This eventuality can occur if every pair of operand bits are different. This combination should also cause the *incr* control input to invert all the submits so that the adder returns 0 instead of *M* [6].

5.5.2 Absolute Value Computation. Video motion estimation algorithms [3] are based on the summation of absolute differences between many pairs of pixel values. Hence, the flagged prefix adder, which can provide an absolute difference efficiently, would be of great utility in video compression processors.

Monochrome pixels are usually represented as *n*-bit unsigned numbers, so that their absolute difference is also an *n*-bit unsigned numbers. However, in performing the two's complement subtraction, $A - B$, the bits of *B* should be inverted, incremented and then prefixed by a '1' to obtain the correct *n+1* - bit two's complement representation of *–B.* Also, the sign of a two's complement number is given by its most significant sum bit. Hence, it appears that a *n+1* - bit flagged prefix adder is required with s_n connected to the *cmp* input, implying extra delay in obtaining the *cmp* signal.

Infact, an *n*-bit flagged prefix adder may be used as follows. Just as *B* should be prefixed by a '1', *A* should also be prefixed by a '0' : thus s_n is obtained as

$$\begin{aligned} s_n &= a_n \oplus b_n \oplus c_n \\ &= 0 \oplus 1 \oplus c_n \end{aligned} \tag{5.6}$$

If $c_n = 0$, $s_n = 1$, the difference is negative and should be two's complemented to return the positive result *–S;* if $c_n = 1$, $s_n = 0$, the difference is positive and the two's complement of *B* should be completed by incrementing the initial sum $S = A + -B$. This pair of operations is covered by the *cmp* input. Hence, absolute differences may be calculated on

a flagged prefix adder by inverting $G_{n-1:0}$ output of an *n*-bit flagged prefix adder and connecting it to the *cmp* input [6].

5.5.3 Sign Magnitude Addition. Sign-magnitude arithmetic [27] has a number of advantages over two's complement arithmetic:

- Multiplication and division operations may be performed on the magnitudes and signs of the operands independently over one another.
- They dynamic range is symmetric about 0.
- Overflow is simpler to detect.

However, implementation difficulties of sign – magnitude adders have led to this number system's decline in favor of two's complement arithmetic. Specifically, absolute differences are required, implying a two-step subtraction process because the sign of a difference is unpredictable from the signs of the operands. The flagged prefix adder readily overcomes this problem so that sign-magnitude arithmetic circuitry can attain similar performance to two's complement arithmetic circuitry.

The first task in performing a sign-magnitude addition is working out whether an addition or a subtraction is implied by the signs of the operands and the arithmetic operation. A "true addition" is performed if the signs of the operands are opposite and a subtraction is specified [6]. Otherwise, a "true subtraction" is performed:

- True addition: the result magnitude is given by the sum of the operands magnitudes.
- True subtraction: the result magnitude is given by the absolute difference of the operands' magnitudes.

A sign-magnitude adder thus should be capable of computing both sums and absolute differences equally well – the flagged prefix adder offers a structure with this characteristic [6]. True subtraction may be performed in the same way as the absolute difference computation described earlier, in which the bits of B are inverted, and the most significant carry signal used to perform a two's complement operation if the result is negative. Note that if the initial difference is complemented, the sign of the result is altered. True addition may be incorporated by conditionally inverting the B inputs for true subtractions only, and by letting the most significant output denote an overflow rather than a request that the result be two's complemented. Figure 5.6 [6] presents the block diagram for the full circuit.

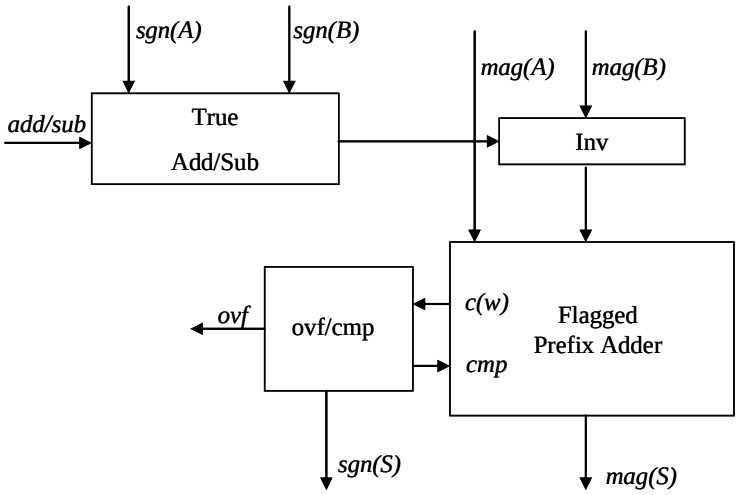

Figure 5.5. Block Diagram of a Sign Magnitude Adder

5.6 Summary

The theory presented in this chapter introduces the concept of flag bits that have been utilized in order to achieve three – operand addition during the research. The following chapter is dedicated to the derivation of the logic equations to obtain necessary flag bits to enable three – operand addition along with its hardware implementation.

CHAPTER 6

THREE INPUT ADDITION

Multi-operand addition is implicit in both multiplications and computation of vector inner products. It is also an operation that is required with certain addressing modes within a microprocessor in order to access the correct memory location. It is such applications that led to the advent of high performance multi-operand adders [15]. Instead of utilizing a series of binary adders in cascade to perform the operation, designers came up with hardware to achieve multi-operand addition with short critical paths and minimal carry propagation. The following section discusses some basic muti-operand addition schemes.

6.1 Carry - Save Adders

The carry - save adder avoids carry propagation by treating the intermediate carries as outputs instead of advancing them to the next higher bit position, thus saving the carries for later propagation. The sum is a redundant *n* – digit carry - save number, consisting of two binary numbers *S* (sum bits) and *C* (carry bits). A carry – save adder accepts three binary input operands or alternatively, one binary and one carry – save operand. It is realized by a linear arrangement of full adders as shown in Figure 6.1 and has a constant delay [58].

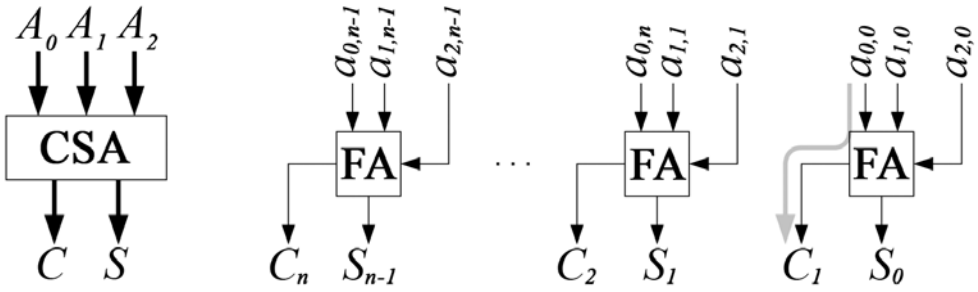

Figure 6.1. Symbol and Schematic of Carry – Save Adder

The arithmetic equations representing the carry – save operation are as given in equations 6.1 – 6.3 [58].

$$2C + S = A_0 + A_1 + A_2 \tag{6.1}$$

$$\sum_{i=1}^{n} 2^i c_i + \sum_{i=0}^{n-1} 2^i s_i = \sum_{j=0}^{2} \sum_{i=0}^{n-1} 2^i a_{j.i} \tag{6.2}$$

$$2c_{i+1} + s_i = \sum_{j=0}^{2} a_{j,i} \quad ; \quad i = 0, 1, \ldots, n-1 \tag{6.3}$$

6.2 Multi - Operand Adders

Multi-operand adders are used for the summation of m n - bit operands $A_0, \ldots, A_{m-1}$ $(m>2)$ yielding a result, S in irredundant number representation with $(n + \lceil \log m \rceil)$ bits. The arithmetic equation for this adder design is given as in equation 6.4 [58]

$$S = \sum_{j=0}^{m-1} A_j \tag{6.4}$$

An m-operand adder can be realized either by serial concatenations of (m-1) carry-propagate adders (ripple carry adders) as shown in Figure 6.2 or by (m-2) carry – save adders followed by a final carry – propagate adder (See Figure 6.3).

The two resulting *adder arrays* are very similar with respect to their logic structure, hardware requirements, as well as length of the critical path. The major difference is the unequal bit arrival time at the last carry – propagate adder. While in the carry – save adder array, bit arrival times are balanced, higher bits arrive later than lower bits in the carry – propagate adder array which, however is exactly how the final adder expects them. This holds true if the ripple carry adders are used for carry propagation through out.

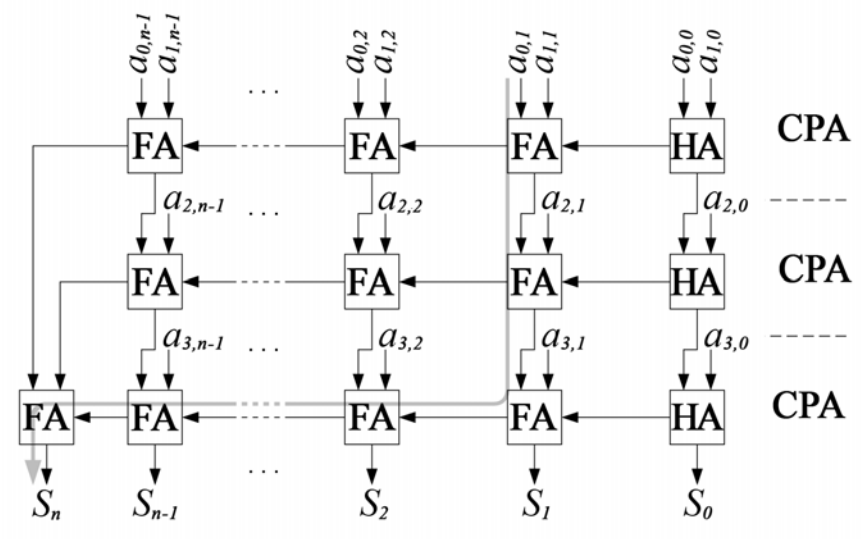

Figure 6.2. Four – Operand Carry – Propagate Adder Array

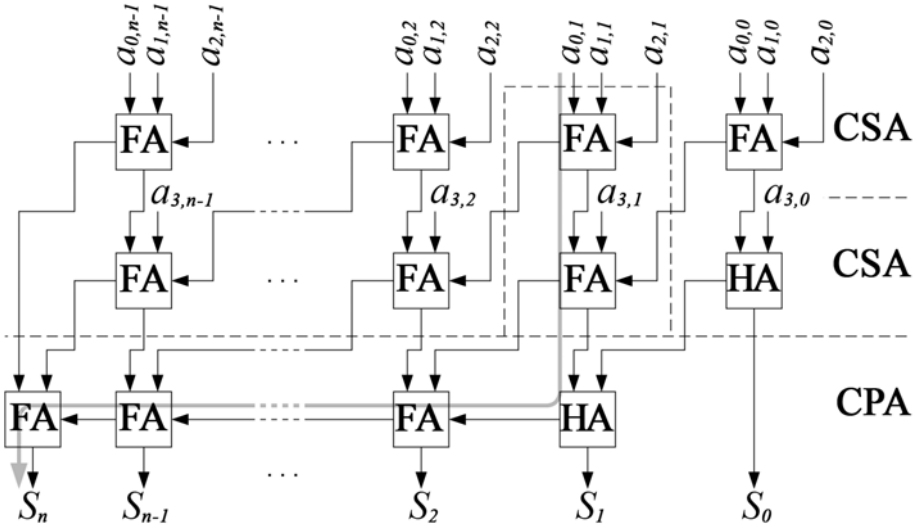

Figure 6.3. Four – Operand Carry – Save Adder Array with Final CPA

Speeding up the operation of the carry propagate adder array is not efficient because each ripple carry adder has to be replaced by some faster adder structure. On the other hand, the balanced bit arrival profile of the carry – save adder array allows for massive speed – up since it utilizes the carry ripple adder only in the last stage as shown in Figure 6.4 [58].

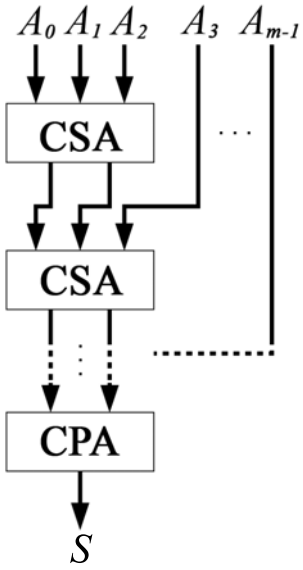

Figure 6.4. Typical Array Adder Structure for Multi - Operand Addition

A single bit slice of the carry – save array from Figure 6.2 is a 1 – bit adder called the *(m,2)* compressor. It compresses *m* input bits down to two sum bits *(c,s)* by forwarding (*m-3)* intermediate carries to the next higher bit position (Figure 6.4). The arithmetic equation is given as follows:

$$2(c + \sum_{l=0}^{m-4} c_{out}^{l}) + s = \sum_{i=0}^{m-1} a_i + \sum_{l=0}^{m-4} c_{in}^{l} \tag{6.5}$$

No horizontal carry propagation occurs within a compressor circuit, i.e., c_{in}^{l} only influences $c_{out}^{k>l}$. An *(m,2)* compressor can be built from *(m-2)* full adders or from smaller compressors. Note that the full adder can also be regarded as a (3,2) compressor. Again, cells can be arranged in tree structures for speed up.

Adder trees are carry save adders composed of tree structured compressor circuits. Tree adders are multi – operand adders consisting of a CSA tree and a final CPA. A fast final CPA can be utilized to get fast multi – operand adder circuits.

Some general remarks on multi – operand adders are formulated [16]

- Array adders have a highly regular structure which is of advantage for both netlist and layout generators.
- An *m*-operand adder accommodates (*m-1*) carry inputs.
- The number of full adders does only depend on the number of operands and bits to be added but not on the adder structure. However the number of half adders as well as the amount and complexity of interconnect wiring depends on the chosen adder configuration.

The following sub – section develops and derives a technique to achieve multi – operand addition utilizing regular binary adders. The first step during this process is to assume that one of the operands is a constant. This is then followed by deriving the necessary logic that allows the third binary operand to be any arbitrary number. The technique has been derived based on the concept of flagged addition as described in Chapter 5.

6.3 Flag Logic Computation

In order to derive the necessary flag logic equations, assume that there are three input operands, *A*, *B*, and *M*. The goal is to achieve the operation, $A + B \pm M$. Starting from the basic building block in arithmetic systems, a full adder takes three bits, a_k, b_k, and c_k and produces two outputs : a sum bit R_k and a carry bit c_{k+1}. The traditional logic equations for a FA are [48]

$$R_k = a_k \oplus b_k \oplus c_k$$
$$c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \tag{6.6}$$

Assuming that the output *R* needs to be added with a third number, the full adder equations can be rewritten assuming R_k is the value that needs to be augmented by M_k, such that:

$$S_k = R_k \oplus M_k \oplus c_k$$
$$c_{k+1} = R_k \cdot M_k + R_k \cdot c_k + M_k \cdot c_k \tag{6.7}$$

Utilizing these equations, the new function, called a flag function, can be computed such that: $F_k = M_k \oplus c_{k+1}$ where F_k is the flag. The flag function is utilized such that it determines whether the current value is flagged to change. Consequently, this structure can be formulated by developing flag equations based on speculative elements of the constant [48]:

$$c_{k+1} = \begin{cases} R_k \cdot c_k & if \;\; M_k = 0 \\ R_k + c_k & if \;\; M_k = 1 \end{cases}$$
$$F_k = \begin{cases} c_k & if \;\; M_k = 0 \\ \overline{c_k} & if \;\; M_k = 1 \end{cases} \tag{6.8}$$

Similar to the conditional sum adder, the flag bit is chosen based on either a 1 or a 0 for the present and the previous bit of the third operand. Utilizing the equations above, new flag equations can be computed. In other words, two bits of the constant are examined to determine whether or not the carry bit affects the current position. Assume that $M_k = 0$ and $M_{k-1} = 1$, equation 6.9 can be used obtain the flag logic [48]

$$c_{k+1} = R_k \cdot c_k \quad \exists \quad M_k = 0$$
$$c_{k+1} = R_k \cdot c_{k-1} \quad \exists \quad M_{k-1} = 1$$
$$\therefore \;\; c_{k+1} = R_k \cdot F_k \tag{6.9}$$
$$\therefore \;\; c_k = R_{k-1} \cdot \overline{F_{k-1}}$$

Table 6.1 [48] shows the complete set if equations for both the flag bits and the carry bits, F_k and c_k.

Table 6.1. Flag and Carry Logic Based on Third Operand [48]

M_k	M_{k-1}	c_k	F_k
0	0	$R_{k-1} \bullet F_{k-1}$	$R_{k-1} \bullet F_{k-1}$
0	1	$R_{k-1} + \overline{F_{k-1}}$	$R_{k-1} + \overline{F_{k-1}}$
1	0	$R_{k-1} \bullet F_{k-1}$	$\overline{R_{k-1} \bullet F_{k-1}}$
1	1	$R_{k-1} + \overline{F_{k-1}}$	$\overline{R_{k-1}} + F_{k-1}$

The generation of flag bits for the third operand is complicated due to the fact that the value of R_k is produced from the addition of two numbers a_k and b_k. Therefore, not only is the computation based on the bits from the constant but also from the carry produced from the binary adder. The relationship between the result bits R_k and the carry bits c_k can be re-written as [48]

$$R_{k-1} = \begin{cases} a_{k-1} \oplus b_k & if \;\; c_k = 0 \\ a_{k-1} \oplus b_{k-1} & if \;\; c_k = 1 \end{cases} \tag{6.10}$$

In other words Table 6.1 can be re written based on the value of the carry assuming $R_{k-1} = a_{k-1} \oplus b_{k-1}$ as shown in Equation 6.10. Table 6.2 shows the new logic equations for the flag based on the third operand M_k.

Table 6.2. Flag Logic utilizing Carry from the Prefix Tree [48]

M_k	M_{k-1}	$F_k (c_k = 0)$	$F_k (c_k = 1)$
0	0	$R_{k-1} \bullet F_{k-1}$	$\overline{R_{k-1}} \bullet F_{k-1}$
0	1	$R_{k-1} + \overline{F_{k-1}}$	$\overline{R_{k-1} \bullet F_{k-1}}$
1	0	$\overline{R_{k-1} \bullet F_{k-1}}$	$R_{k-1} + \overline{F_{k-1}}$
1	1	$\overline{R_{k-1}} \bullet F_{k-1}$	$R_{k-1} \bullet F_{k-1}$

In order for the equations to be computed properly, some initial conditions are required to guarantee correct computation. Similar to other algorithms that examine two bits at a time, the least significant bit assumes that $M_{-1} = 0$. The following initial conditions are assumed [48]

$$R_{-1} = M_{-1} = F_{-1} = 0 \tag{6.11}$$

The following example represents the exact operation of the circuit

A=00001001 B=01001110 M=00111001

0	0	0	0	1	0	0	1	A
0	1	0	0	1	1	1	0	B
0	0	1	1	1	0	0	1	M
0	1	0	1	0	1	1	1	*R*
1	1	0	0	1	1	1	1	*F*
1	0	0	1	0	0	0	0	*S=R+M*

The following sections describe the implementation of the flag logic for constant addition in binary adders and for three input addition in prefix adders.

6.4 Constant Addition

Based on the flag logic equations derived in the previous equation, the hardware was designed to achieve constant addition. Constant addition is a term used to describe addition of three operands, where one of the three operands is a constant. The resulting adder designs are called Enhanced Flagged Binary Adders (EFBA) [17].

Enhanced flagged binary adders incorporate extra hardware allowing the third operand to be any constant [17]. The underlying technique is to generate flag bits but the computation of flag bits gets more complicated due to the dependence on the bits of the

third operand. Computation of flag bits for this new adder design depends on the carry outputs, readily available from each of the selected adder designs.

Flag bits are computed according to the logic equations in Table 6.2. A straightforward example implementation of flagged inversion cells is shown in Figure 6.5 [17]. The order of the basic gates in the first stage will change depending on the constant that is being added. As can be observed from this logic that the cells only consist of very basic gates with a multiplexer at every bit position choosing the appropriate flag bit depending on the carry out at the corresponding bit position.

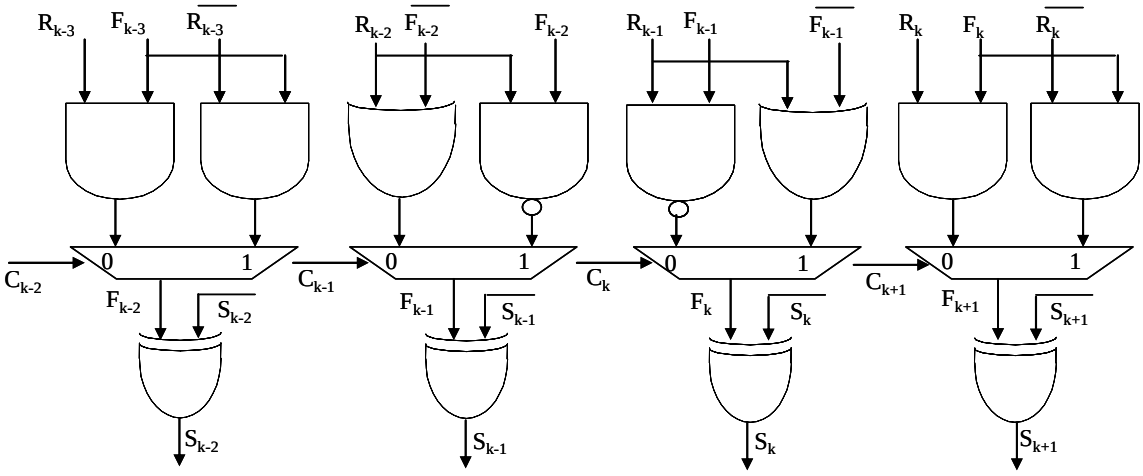

Figure 6.5. Flag Inversion Cells for Constant Addition

6.5 Three Input Addition

It can be deduced observing Figure 6.5 and Table 6.2, that a minimum of four logic gates are required to compute the flag bits based on eight different combinations of the constant and carry bits. The four logic gates and their inputs are summarized in Table 6.3. Tables 6.4 summarizes the combinations of the constant and carry bits that select one of the four logic gates mentioned in Table 6.3 to compute the flag bit in each successive position. The FIC will therefore be a multi-level structure as shown in Figure 6.6.

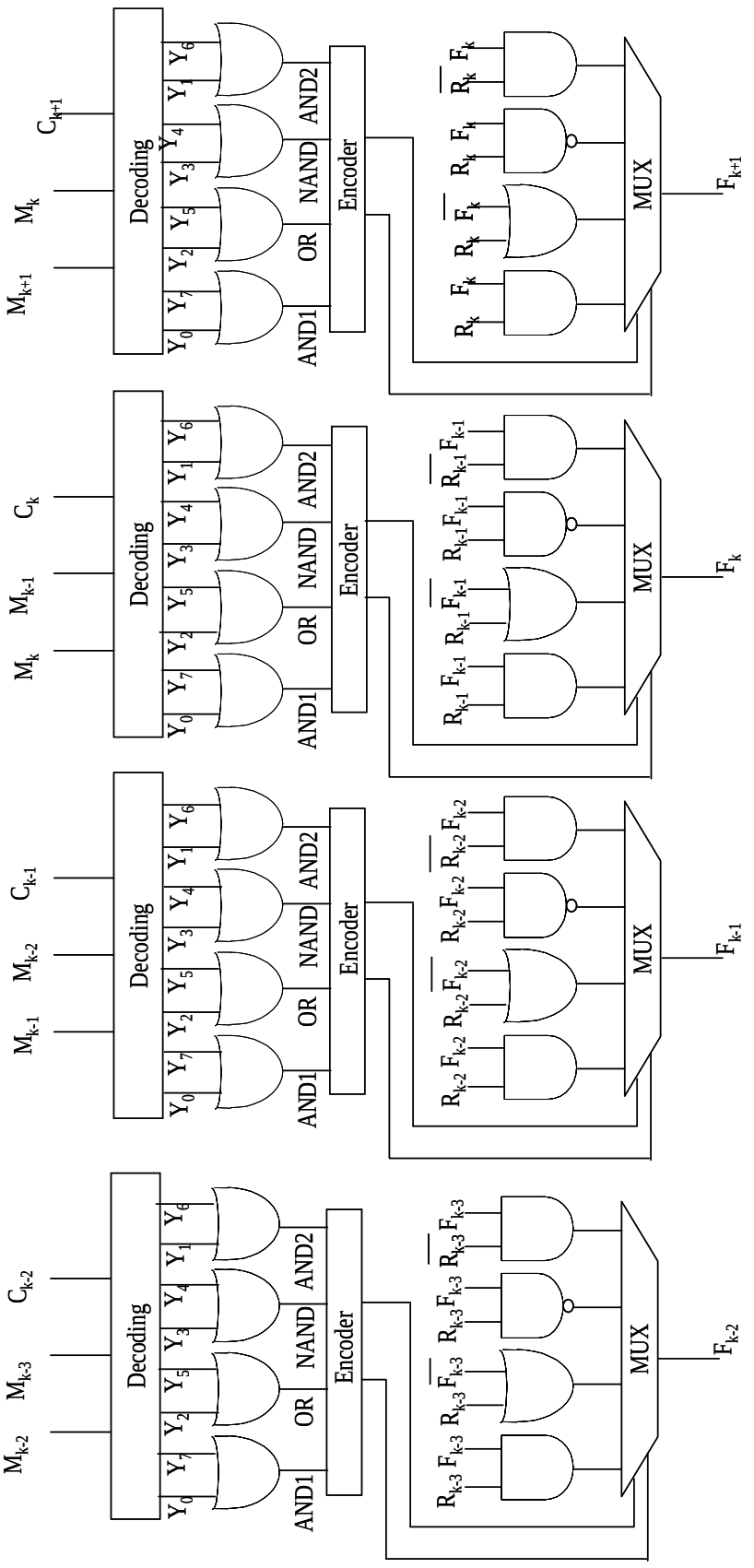

Figure 6.6. Flag Inversion Cells for Three - Input Addition

Table 6.3. Minimum Flag Logic Gates

Gate	*Inputs*
AND1	R_{k-1}, F_{k-1}
OR	$R_{k-1}, \overline{F_{k-1}}$
NAND	R_{k-1}, F_{k-1}
AND2	$\overline{R_{k-1}}, F_{k-1}$

Notice, that the flag bit computation for each bit position depends on the flag bit computed in the preceding position, thereby causing a rippling effect, making this the critical path of the circuit.

Table 6.4. Logic Gate Combinations

M_k	M_{k-1}	C_k	*Gate*
0	0	0	AND1
1	1	1	
0	1	0	OR
1	0	1	
0	1	1	NAND
1	0	0	
0	0	1	AND2
1	1	0	

In order to understand the impact that the additional hardware will have on the adder designs, a detailed gate count has been presented in the following section.

6.6 Gate Count

A mathematical analysis is provided in this section to further understand the operation and performance of flagged binary adders in terms of gate count. This analysis gives an idea of the impact that can be expected on the area of a regular adder enabling the understanding of the trade off between the extra hardware and the added flexibility.

The XOR, AND, OR, and INVERTER are all considered as single gates for the presented analysis. The multiplexer is considered as a separate complex gate entity.

For the normal prefix adder, consider, *n* as the word length and *w* as the multiplying factor of the prefix adder architecture used. *w* is the factor, determining the number of prefix cells for a given architecture. The gate count (*gc*) used in an *n*-bit parallel prefix adder (gc_{pp}) can be summarized as in equation 6.12.

$$gc_{pp} = n(p \text{ and } g \text{ cells}) + w(\text{black prefix cells}) + (n\text{-}1)(\text{gray prefix cells}) + n(\text{XOR}) \quad (6.12)$$

The *p* and *g* cells are utilized to calculate the *bit propagate* and the *bit generate* signals at the input level and therefore contribute to a total of *2n* AND/OR gates. The black and the gray prefix cells are a part of the prefix tree, each comprising of 3 and 2 AND/OR gates respectively. The final level is a group of XOR gates to calculate the final sum. For a Brent-Kung adder, $w=n$ [6], for the Ladner-Fischer prefix adder, $w=n/2\ log_2 n/4$ [6], and for the Kogge-Stone architecture, it has been proved that $w= n\ log_2 n/4$ [6]. The gate count will increase for a flagged prefix adder (gc_{fpp}) capable of performing basic increment decrement operations and hence equation 6.12 is modified to give

$$gc_{fpp} = n(p \text{ and } g \text{ cells}) + w(\text{black prefix cells}) + (n\text{-}1)(\text{black prefix cells}) + n(\text{MUX}) + 3n\ (\text{XOR}) \quad (6.13)$$

Recall from Chapter 5, that the gray prefix cells are converted to black prefix cells and hence there are no terms in equation 6.13 accounting for the gray cells. The extra set of multiplexers is included in the flagged inversion cells and also the XOR gates are increased due to the implementation of the logic as seen in Figure 5.3.

Following the same set of equations, the gate count for an EFBA (gc_{epp}) adder is given by:

$$gc_{epp} = n(p \text{ and } g \text{ cells}) + w(\text{black prefix cells}) + (n\text{-}1)(\text{gray prefix cells}) + n(\text{MUX}) + 2n(\text{XOR}) + 2n(\text{AND/OR}) \quad (6.14)$$

No modifications are made to the prefix tree in this case and therefore, the prefix cells stay the same as in the original prefix adder. However, the last stage consists of the flagged inversion cells which comprise of a pair of basic AND/OR gates at each bit position to compute the flag bits according to Table 6.2. The multiplexers are utilized to select the appropriate flag bit depending on the carry signal. In addition to the *n* XOR gates required to compute the sum of two input operands, an additional set of *n* XOR gates are required to compute the final result, i.e., the sum of three input operands.

For a carry-skip adder, the gate count calculation is more complicated since it depends on the number of groups [26], [28] within the adder, and the size of each group. In this paper, the following group sizes have been utilized.

Wordlength	Group Size
16	3-5-5-3
32	3-5-8-8-5-3
64	2-4-5-6-7-8-8-7-6-5-4-2

Consider m_i as the group size which implies that each group comprises m_i full adder full adder (FA) modules. Each FA in turn consists of 2 XOR gates, 2 AND gates and 1 OR gate, a total of 5 basic gates. Also, each group includes AND gates in order to compute the *Group Propagate* output. Recall from Chapter 3, (See Figure 3.7) each carry skip module also consists of a multiplexer and therefore each adder will consist of the same number of multiplexers as there are groups. It is also important to realize that when the group size exceeds 4 bits, it is necessary to compute the *Group Propagate* utilizing

multi-level AND gates, in order to ensure, that the fan-in of any of the gates does not exceed four. For a flagged carry-skip adder, an additional set of AND gates are a part of the carry-skip module to enable the correct computation of flag bits according to $P_{i:k} = \underset{j=k}{\overset{i-1}{AND}} p_j$. On the other hand, for an enhanced flagged carry-skip adder, the gate count is only increased by *2n* AND gates and *n* MUXs which are a part of the flagged inversion cells of Figure 6.5.

Similar to the carry-skip adder, the carry-select adder is also divided into groups. Assume, the *n*-bit adder is divided into groups of size, *w* bits. Each adder design consists of a *m-bit* conditional adder for each group of *m* bits and (*n/m*)*-1* MUXs. The conditional adder module in turn consists of two *m*-bit FAs, thereby leading to a total of *10m* gates in each module. The total number of gates therefore, equals

$$gc_{COS} = (10m)(n/m)(\text{AND/OR}) + (n/m)\text{-}1\ (\text{MUX}) \tag{6.15}$$

Once the adder is modified to incorporate the enhanced adder, the gate count is modified to

$$gc_{eCOS} = (10m)(n/m)(\text{AND/OR}) + (n/m)\text{-}1\ (\text{MUX}) + 2n(\text{AND/OR}) + n(\text{MUX}) + n(\text{XOR}) \tag{6.16}$$

The preceding discussion was based on the assumption that the third operand is a constant. In case, the third operand is a variable the flag logic changes according to Figure 6.6. This hardware is incorporated into the parallel prefix architectures and shows the presence of a 3-8 decoder that consists of 3 inverters and 8 AND gates at each bit position. This is followed by a stage of 4 OR gates and then an encoder comprising of 2 inverters and 2 OR gates. The final stages consist of 4 AND/OR gates and a multiplexer.

This path is replicated *n* times for an *n* bit adder. Therefore the, final gate count equations for a three-input flagged prefix adder, gc_{tipp}

$$gc_{tipp} = n(p \text{ and } g \text{ cells}) + k(\text{black prefix cells}) + (n-1)(\text{gray prefix cells}) + 5n(\text{INV}) + 18n(\text{AND/OR}) + n(\text{MUX}) + 2n(\text{XOR}) \quad (6.17)$$

Table 6.5 summarizes the gate count for each adder design. These designs comprise of the conventional, flagged, EFBA, and TIFPA versions for three different operand sizes.

It can be observed that area is expected to increase minimally in the new design. The Kogge-Stone adder suffers the maximum increase in area due to the highest number of black prefix cells within the prefix carry tree. Also due to the very intricate and complex wiring within the structure, it does not prove to be very area efficient. On the other hand, the carry skip adder, when modified to perform constant addition, is expected to be the most area efficient adder.

6.7 Summary

This chapter is dedicated to the hardware implementation of the proposed design. It presents the logic design for the flag logic and also described how it is incorporated into binary adders. The following adders have been implemented and designed

- Conventional Brent – Kung Parallel Prefix Adder
- Conventional Ladner – Fischer Parallel Prefix Adder
- Conventional Kogge – Stone Parallel Prefix Adder
- Flagged Brent – Kung Parallel Prefix Adder
- Flagged Ladner – Fischer Parallel Prefix Adder
- Flagged Kogge – Stone Parallel Prefix Adder

Table 6.5. Gate Count for All Adder Implementations

Adder Designs	16 bits		32 bits		64 bits	
	Gate Count	Mux Count	Gate Count	Mux Count	Gate Count	Mux Count
Brent – Kung	123	0	254	0	510	0
Ladner – Fischer	129	0	305	0	705	0
Kogge – Stone	179	0	449	0	1089	0
Carry – Skip	86	3	170	5	329	10
Carry – Select	160	3	320	7	640	15
Flagged BK	141	16	285	32	573	64
Flagged LF	144	16	336	32	768	64
Flagged KS	192	16	480	32	1152	64
Flagged CSK	148	19	298	37	583	74
Enhanced BK	174	16	350	32	702	64
Enhanced LF	177	16	401	32	1281	64
Enhanced KS	222	16	542	32	1281	64
Enhnaced CSK	134	19	266	37	520	74
Enhanced COS	208	19	416	39	832	79
Three-Input BK	526	16	1054	32	2110	64
Three-Input LF	529	16	1105	32	2305	64
Three-Input KS	579	16	1249	32	2689	64

- Flagged Carry – Skip Adder
- Enhanced Brent – Kung Parallel Prefix Adder
- Enhanced Ladner – Fischer Parallel Prefix Adder
- Enhanced Kogge – Stone Parallel Prefix Adder
- Enhanced Carry – Skip Adder
- Enhanced Conditional – Sum Adder
- Three – Input Brent Kung Parallel Prefix Adder

- Three – Input Ladner – Fischer Parallel Prefix Adder
- Three – Input Kogge – Stone Parallel Prefix Adder

The conventional designs refer to the adders capable of adding two operands. The flagged binary adder fall in the category of adders capable of performing increment and decrement operations. The enhanced flagged binary adders, have been incorporated with the new hardware and are capable of constant addition. Finally, the three – input prefix adders perform multi – operand addition. The next chapter focuses on the delay analysis and simulation results for each adder design.

CHAPTER 7

ANALYSIS AND SIMULATION

The goal of this chapter is to look at each adder design individually and investigate the performance analytically by applying the method of logical effort. In addition, each adder circuit is also implemented in order to obtain simulation results. Both results are compared in order to better understand the operation of the proposed technique and also to verify the simulation results with the theoretical results.

7.1 Logical Effort

Logical Effort is a method described in Chapter 4 to estimate the critical delay of a circuit based on fan-out and gate size. This section applies the method of logical effort to the three prefix tree architectures, carry skip, carry select adders, and to the modified versions described in Chapters 5 and 6.

Recall the structure of a prefix tree. The black cell computes the Group *Generate* ($G_{i:j}$) and the *Group Propagate* ($P_{i:j}$) signals utilizing 4 inputs, $G_{i:k,}$ $G_{k-1:,j,}$ $P_{i:k,}$ and $P_{k-1:j}$. These inputs are denoted as the upper and lower *generate* and *propagate* signals, *gu*, *gl*, *pu*, and *pl* respectively [25]. The post-processing stage comprises of XOR gates to calculate the final sum. Table 7.1 [25] tabulates the logical effort for the three basic building blocks within the parallel prefix adder. The logical effort, LE and the parasitic delay, PD of each of these cells has been calculated considering an inverting CMOS design.

The method of logical effort can be applied to calculate the critical delay of each adder design based on which and how many blocks fall in the critical path of the circuit.

The total delay is the sum of the logical effort along the critical path and the parasitic delays of the each block on that path [50].

Table 7.1. Logical Effort and Path Delays for Adder Blocks

Cell	Term	Values
Bitwise cell	LE_{bit}	9/3
	PD_{bit}	6/3
Black cell	$LE_{blackgu}$	4.5/3
	$LE_{blackgl}$	6/3
	$LE_{blackpu}$	10.5/3
	$LE_{blackpl}$	4.5/3
	PD_{blackg}	7.5/3
	PD_{blackp}	6/3
Gray cell	$LE_{graypgu}$	4.5/3
	LE_{graygl}	6/3
	LE_{graypu}	6/3
	PD_{gray}	7.5/3
Buffer	LE_{buf}	½
Sum XOR	LE_{xor}	9/3
	PD_{xor}	9/3 + 12/3

The logical effort for the Brent-Kung (LE_{BK}), Ladner-Fischer (LE_{LF}), and Kogge-Stone (LE_{KS}) adders is calculated utilizing equations 7.1, 7.2, and 7.3 respectively. The parasitic delays, PD_{BK}, PD_{LF}, PD_{KS} are given by 7.4, 7.5, and 7.6. *n* represents the size of the adder design.

$$LE_{BK} = LE_{bit} + LE_{graypu} + (n\text{-}1)\,(LE_{graygl}+LE_{buf}) + LE_{graygl}+LE_{buf} + (n\text{-}2)\,(LE_{graygl}+LE_{buf}) + LE_{xor} \quad (7.1)$$

$$LE_{LF} = LE_{bit} + LE_{graypu} + (n\text{-}2)\,(LE_{graygl}+LE_{buf}) + LE_{graygl}+LE_{buf} + (n\text{-}2)\,(LE_{graygl}+LE_{buf}) + LE_{xor} \quad (7.2)$$

$$LE_{KS} = LE_{bit} + LE_{blackpu} + LE_{blackpl} + (n\text{-}2)\,(LE_{blackpu} + LE_{blackpl}) + LE_{graypu} + LE_{xor} \tag{7.3}$$

$$PD_{BK} = PD_{bit} + (m\text{-}1)\,(PD_{gray}) + PD_{buf} + (n\text{-}2)\,(PD_{gray}) + PD_{xor} \tag{7.4}$$

$$PD_{LF} = PD_{bit} + (n\text{-}2)\,(PD_{gray}) + PD_{gray} + (n\text{-}2)\,(PD_{gray}) + PD_{gray} + PD_{xor} \tag{7.5}$$

From Chapter 3, recall the structure of a carry select adder. The delay of a carry select adder is given by equation 3.14 [52] [53] which is repeated here for convenience. *n* represents the size of the operand and *m* is the size of each module the adder is broken down to.

$$T_{carry-sel} = mT_{FA} + (\frac{n}{m} - 1)T_{MUX} \tag{7.6}$$

The full adder structure that has been utilized for this design in represented in Figure 7.1.

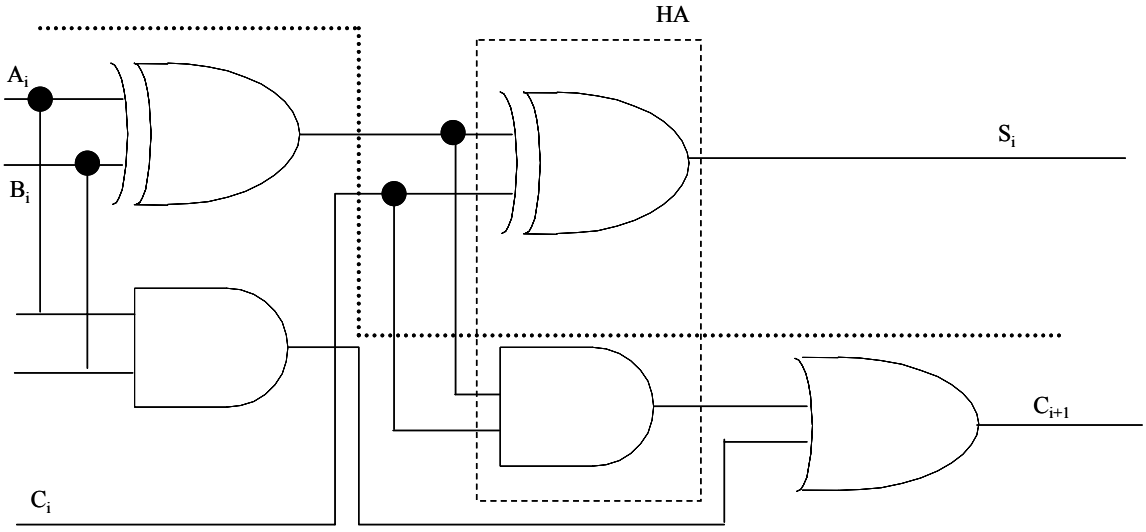

Figure 7.1. Full Adder used within Carry-Skip, Carry-Select, and Carry-Save Adders [20]

Table 7.2 provides the logical effort and path delays for the gates that have been utilized within the full adder circuit, and multiplexers, and flagged inversion cells. MUXn, represents an *k*-way multiplexer with *k* inputs.

Table 7.2. Logical Effort and Path Delays for Gates within Adder Blocks [25]

Cell	Term	Values
AND2	LE_{AND2}	4/3
	PD_{AND2}	2
AND3	LE_{AND3}	5/3
	PD_{AND3}	3
OR2	LE_{OR2}	5/3
	PD_{OR2}	2
OR4	LE_{OR4}	9/3
	PD_{OR4}	4
NAND2	LE_{NAND2}	4/3
	PD_{NAND2}	2
MUXk	LE_{MUX}	2
	PD_{MUXn}	2k
Inverter	LE_{INV}	1

Each carry select adder has been divided into groups of 4 bits each for 16, 32, and 64 bit adder designs. Therefore, the logical effort (LE_{COS}) and the path delay (PD_{COS})can be computed utilizing equations 7.7 and 7.8 respectively.

$$LE_{COS} = 4.2.LE_{XOR} + 4.2.LE_{AND2} + 4.LE_{OR2} + (n/m-1)LE_{MUX2} \quad (7.7)$$

$$PD_{COS}=4.PD_{XOR} + 4.PD_{AND2} + 4.PD_{OR2} + (n/m-1)\, PD_{MUX2} \quad (7.8)$$

The carry-skip adder has a similar structure with an additional multiplexer in the critical path. Following the same line as for a carry select adder, the logical effort and path delay equations are presented in 7.9 and 7.10 respectively. Each group is 4 bits long in order to make sure that the fan-in for the AND gate that is utilized to calculate the *Group Propagate* signals does not exceed 4. The critical delay of a carry-skip adder

is $(2m-1)t_c + (\frac{n}{m}-1)t_{mux} + t_s$, where t_c and t_s are the critical delays to the carry and the sum bit respectively. Hence the logical effort and the path delays can be approximated as

$$LE_{CSK} = 7.2.2.LE_{XOR} + 7.2.2LE_{AND2} + 7.2LE_{OR2} + (n/4\text{-}1)LE_{MUX} \quad (7.9)$$

$$PD_{COS} = 7.PD_{XOR} + 7.PD_{AND2} + 7.PD_{OR2} + (n/4\text{-}1)\, PD_{MUX} \quad (7.10)$$

Table 7.3 represents the FO4 delays as estimated by the application of logical effort for all conventional adder designs that have been implemented in this thesis.

Table 7.3. Logical Effort Estimates for Conventional Adder Designs

Adder	Logical Effort		
	16 bits	32 bits	64 bits
Brent – Kung	9.4	11.4	15.4
Ladner – Fischer	9	11	14
Kogge – Stone	7.6	9	11.8
Carry – Select	15.83	31.83	43.83
Carry – Skip	52.92	65.92	77.92

For the flagged binary adders, it is necessary to calculate the delay for the flagged inversion cells as presented in Chapter 5. Based on figure 5.3, the logical effort and path delay equations can be represented in 7.11 and 7.12 respectively.

$$LE_{FIC} = 2.LE_{XOR} + LE_{MUX} + LE_{XOR} \quad (7.11)$$

$$PD_{FIC} = PD_{XOR} + PD_{MUX} + PD_{XOR} \quad (7.12)$$

Table 7.4 summarizes the critical delays of flagged binary adders as estimated by the application of logical effort.

Table 7.4. Logical Effort Estimates for Flagged Adder Designs

Adder	Logical Effort		
	16 bits	32 bits	64 bits
Brent – Kung	15.15	17.15	21.15
Ladner – Fischer	14.75	16.75	19.75
Kogge – Stone	13.35	14.75	17.55
Carry – Skip	58.67	71.67	83.67

In order to estimate the delay for three – input addition the flagged inversion cell is broken down for 1 bit as shown in Figure 7.2.

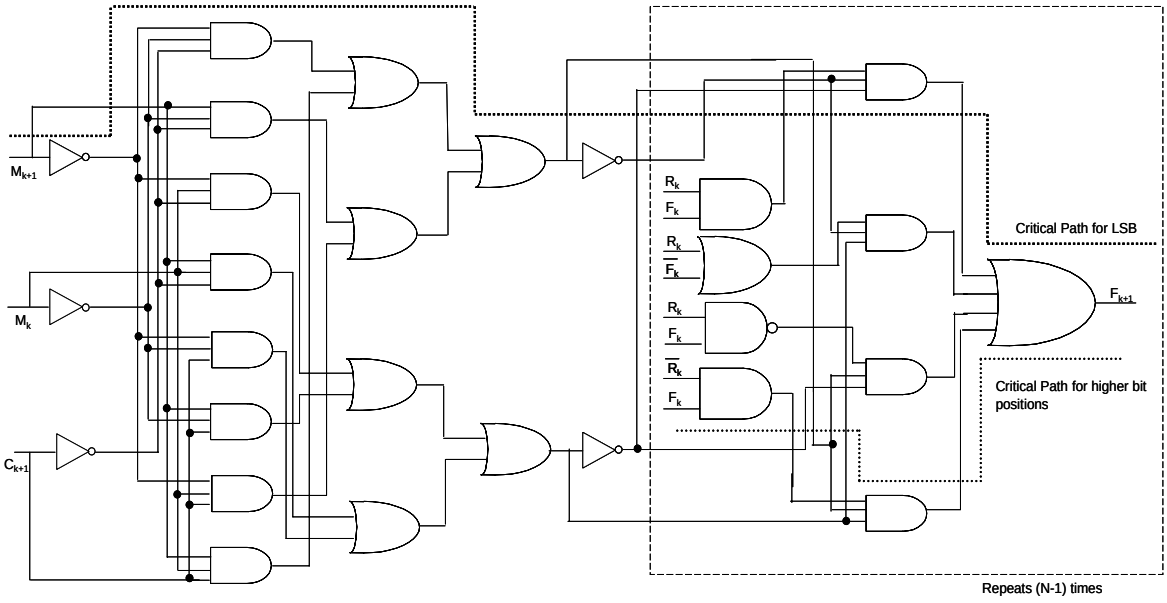

Figure 7.2. One Bit FIC for Three – Input Addition

The prefix structure's critical path will change once, the extra hardware is incorporated within the design to enable three-input addition. In order to compute the delay for the flag logic, an expanded view of the flagged inversion cell for one bit position has been shown in Figure 7.2. The critical path has been marked. It comprises of 3 input AND gates (AND3), 2 input OR gates (OR2), inverters (inv) and a 4input OR gate (OR4). The critical path of this block will consist of 2 components. The first component accounts for

the delay to compute the flag bit in the LSB position. The flag bit then ripples through the remaining, (*n-1*) stages. The critical path for (*n-1*) stages has also been shown in Figure 7.2. Following the same methodology as presented in [8], equations 7.13 and equations 7.14 represent the logical effort and the parasitic delay for the FIC respectively.

$$LE_{inv} + 4\,(LE_{AND3}) + LE_{OR2} + LE_{OR2} + LE_{inv} + 2(LE_{AND3}) + LE_{OR4} + (n\text{-}1)\,[LE_{OR2} + LE_{AND2} + LE_{NAND2}] + LE_{OR4} \tag{7.13}$$

$$PD_{inv} + PD_{AND3} + PD_{OR2} + PD_{OR2} + PD_{inv} + PD_{AND3} + PD_{OR4} + (n\text{-}1)\,[PD_{AND2} + PD_{AND3} + PD_{OR4}] \tag{7.14}$$

The path delay of the FIC is added to the critical delay of the prefix adder to get the total delay of the circuit. In order to compare this delay with that of a CSA, the logical effort for a CSA has also been calculated. Refer to Figure 7.3 which is a dot diagram. The full adder and half adder structures utilized within the CSA are shown in Figure 7.1 [1].

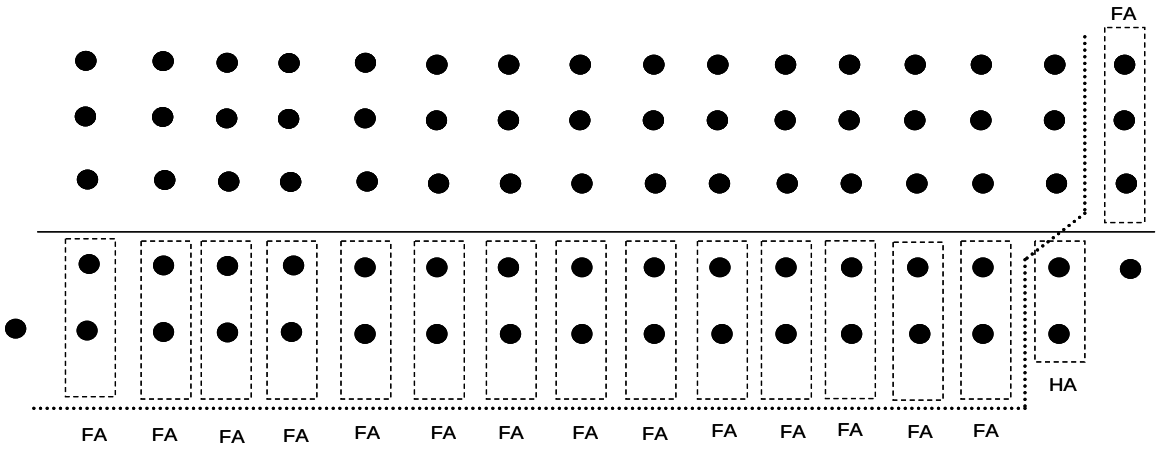

Figure 7.3. Dot Diagram of a Carry – Save Adder

The sum bit for the LSB position is obtained utilizing a FA. The second bit for the sum is then calculated utilizing a half adder. After the first two stages, the rest of the sum

is computed utilizing a carry ripple adder. The ripple carry adder (RCA) in the last stage of a CSA has a significant impact on the critical path of the design.

Based on Figure 7.3, the LE and PD are represented in equations 7.15 and 7.16. Here, $LE_{RCA(N-2)}$ represents the LE of a RCA with operand size *n-2*.

$$LE_{XOR} + LE_{AND2} + LE_{XOR} + LE_{AND2} + LE_{OR2} + LE_{XOR} + LE_{AND2} + LE_{RCA(n-2)} \quad (7.15)$$

$$PD_{XOR} + PD_{AND2} + PD_{OR2} + PD_{XOR} + PD_{RCA(n-2)} \quad (7.16)$$

The final delays for the TIFPA and three-input CSA designs are tabulated in Table 7.5.

Table 7.5. Logical Effort Estimates for Three – Input Flagged Adder Designs

Adder	Logical Effort		
	16 bits	32 bits	64 bits
Brent – Kung	93.48	159.48	305.48
Ladner – Fischer	93.08	159.08	305.08
Kogge – Stone	90.68	157.08	302.48
Carry – Skip	83.08	200.167	405.5

In the case of the flagged prefix structures, the flag bit ripples from one stage to another. From Figure 6.6 from Chapter 6, it can be observed that the flag bit needs to ripple through three CMOS gates in order to compute the flag bit in the successive positions. In the case of the CSA, the last stage is always a RCA. The RCA is implemented utilizing, XOR which form a part of the critical path in every bit position. This causes the delay to worsen as the operand size increases.

From Table 7.5, it can be concluded that the total delay to compute the sum based on logical effort calculations will not depend on the prefix tree. This is attributed to the fact that prefix adders have a very small delay compared to the delay imposed by the flag logic. Certain assumptions such as, all inputs arrive at the same time with equal drive,

and uniform gate size simplify the computations. Although logical effort analysis does not highlight the difference in delay, synthesis results show a clearer difference of the impact the structure of a prefix tree will have on the final design.

7.2 Simulation Results

The binary adders described in the previous sections are implemented and modified to be able to perform increment and decrement operations. The adders are also modified to incorporate the flag logic according to Table 6.2, enhancing the functionality of flagged adders to perform constant addition. Each adder was designed for 16-bit, 32-bit and 64-bit operand sizes. An analysis was performed on all adders with regards to, area, delay, and power. The designs are implemented in the TSMC 0.18μm technology System-on-Chip design flow to investigate the power, area, and delay tradeoffs. Synthesis is performed with Cadence Build Gates and Encounter [22]. The nominal operating voltage is 1.8V and simulation is performed at T=25^{o}C. Layouts are generated for each adder design and parasitically extracted to obtain numbers for area, delay and power.

7.2.1 Conventional Adder Architectures. The results for regular adders, without any modifications are tabulated as shown in Table 7.6 at the end of the chapter for 16, 32 and 64 bit designs. The frequency of operation is obtained by driving the circuit with the worst case input and verifying correct operation at the highest number by gradually increasing the operating frequency.

The Ladner-Fischer prefix tree aims at reducing the depth of the tree in order to compute the carry signals. The complexity measure for this case is the gate count and speed. However it does not perform well in terms of the capacitive fan out load. Contrary to this is the Brent-Kung design that addresses the fan-out restrictions, but the logical

depth of the tree is increased. Therefore, the Brent-Kung tree has a more regular structure, which is easy to implement in terms of chip design and wiring density. Another approach is the Kogge-Stone design that limits the lateral logic fan-out to unity at each node, but increases the number of lateral wires at each level. This accounts for the fact that the Kogge-stone adder has the highest value for the multiplying factor *w* among the three prefix designs selected. Maximum increase in area is observed with the Kogge-Stone adders with increase in operand size. However, it should also be noted that the Kogge-Stone adder has the best performance in terms of speed.

The objective of the carry-skip adder is to reduce the worst-case delay by reducing the number of FA modules through which the carry has to propagate [52]. Therefore, the adder consists of small groups of ripple carry adders, modified to include the skip network shown in Chapter 3. This leads to attractive regular structures, but unlike the prefix adders, the carry signals are not generated in parallel, leading to a less attractive performance in terms of speed. The carry- select adder consists of a pair of carry-propagate adders for each group, leading to significant area consumption.

7.2.2 Flagged Adder Architectures. The regular adder architectures are modified to perform operations presented in Chapter 5. Area, delay and power estimates are also obtained for these designs. A comparison has also been made between conventional designs and the flagged architectures, to study the impact of the additional hardware on the critical path delay and area consumption.

For the prefix adders, the gray cells are converted to black prefix cells, increasing the gate count by unity for each cell. Also, the inclusion of the flagged inversion cells will account for the rise in area consumption. As the bit count increases, the number of

prefix cells within the prefix tree also increase. For a carry-skip adder, the increase in area can be attributed to the fact that the maximum fan in has been limited to 4. As the number of bits increases, the number of AND gates required in order to obtain all the necessary *Group Propagate* signals also increases.

The delay of the flagged prefix adders increases almost linearly with the increase in operand size. For, a carry skip adder, this does not hold true. Again, the multi-level AND gate inclusion accounts for a higher increase in delay compared to the prefix tree architectures.

The carry-select adder has not been incorporated with the flag logic since it proves to be an inefficient design. Table 7.7 summarizes area, delay and power measurements for the flagged binary adders for 3 different operand sizes.

The impact of the additional hardware is analyzed by looking at the difference in area and delay between a regular adder and flagged binary adder. The increase in area is more noticeable for the Kogge-Stone adder than it is for the Brent-Kung or Ladner-Fischer designs. Also notice the significant impact on area for the carry-skip design. Although the gate count increase is higher for the prefix trees, the layout shows a major difference for the carry-skip adder. The variable group sizes utilized within the design is responsible for the reduction in power consumption, but leads to a negative impact on area consumption. The impact on delay proves to be insignificant after inclusion of the flagged inversion cells within each design. This particularly holds true for the prefix adder designs. The critical path is affected minimally, compared to the flexibility incorporated within the design. The Kogge-Stone structures prove to be the fastest, but the tradeoff is the area consumption. It can also be observed, that the power consumption

for these structures is the highest probably due to the high switching speed within the circuit.

7.2.3 Enhanced Adder Architectures. A similar analysis is performed to study the impact of the flagged inversion cells for constant addition. Numerical results have been provided in Tables 7.8 for each operand size. Increase in operand size, causes a significant increase in area for all designs including the carry-select adder. This is due to the flagged inversion cells that consist of *2n* basic gates, along with *n* multiplexers.

The increase in delay is seen to be highest for the carry-skip and the carry-select adder. This is due to the fact that unlike the prefix structures, the carries are not obtained in parallel. The adders are divided into groups and hence, each group waits for the preceding group to generate the carry. The flagged inversion cells for the constant addition fall in the critical path of the design since the flag bit computation depends on the carry bits, thereby having a more significant impact on the delay compared to the flagged binary addition. A comparison was also made between the enhanced flagged binary adders and the carry–save adders to see the performance improvement of this design over the conventional method.

In terms of area, it can be noticed, that the prefix designs have an edge over the carry-save adders, although the Kogge-Stone adder consumes maximum area with increase in operand size. Again, this can be attributed to the number of black prefix cells and the intricate wiring. For the carry select adder, the area consumed due to the pair of adders that are required to compute the sum for each possible value of carry.

In terms of delay, again, the enhanced binary adders have a favorable performance to that of a carry-save adder. In fact, the prefix designs prove to be the most

efficient in terms of delay. The carry-skip and carry-select adders, however lie at a disadvantage due to the unavailability of the carry bits in parallel.

The proposed technique, therefore, utilizes regular adder designs, incorporating minimal logic within these architectures, to introduce the flexibility to deal with three operands at the same time. The choice of the constant determines the flag bits, and therefore can have an impact on the overall speed of the circuit. However, as discussed earlier, the constant presented in this paper gives a decent estimate of the delay after simulation due to the realization of maximum possible gates to compute the final sum. If the constant chosen results in fewer gates at the last level of the design, better delay and area results can be expected.

7.2.4 Three-Input Adder Architectures. Table 7.9 tabulates the results for 16, 32 and 64 bit binary numbers as operands. It can be seen that the TIFPA designs have a favorable performance compared to the CSA. It is observed, that the area increases by approximately 3.5 times for all architectures when the operand size is doubled. Delay measurements increase by approximately 1.7 times with increase in operand size. The increase in power with change in operand size is significant for prefix structures compared to the carry-save adder. This can be due to the high fan in and significant wiring complexity associated with the prefix trees. The depth of the Kogge Stone adder is the least among all three adder architectures resulting in it being the fastest design. The Ladner Fischer adder has fewer levels compared to the Brent-Kung adder, leading in lower delay numbers. Brent Kung adders however, have lower numbers of area making it a good choice where area is a constraint. A trade-off between area and delay can be observed for the Kogge-Stone flagged prefix adder, which is observed to be the fastest

design among all adder architectures and consumes maximum area. The Ladner-Fischer tree provides a good compromise in terms of area and speed.

7.2.5 Graphical Results. In order to get a better understanding of the results provided in this thesis graphical results are generated for comparison. Figures 7.4 – 7.12 show how the critical parameters for every adder design varies as the operand size changes. Figures 7.13 – 7.21 show results for each operand size and carious adder implementations.

7.3 Summary

This chapter summarized results of all adder designs that are implemented for the purpose of completing this research. Each adder design was analyzed based on the results obtained from logical effort. Each adder architecture is implemented utilizing Verilog as the hardware description language. Synthesis tools are utilized in order to obtain area, delay and power estimates for every adder architecture. Results obtained from logical effort are in sync with the results obtained from synthesis. Chapter 8 describes the conclusions and proposes future work in this area.

Table 7.6. Post – Layout Estimates for Conventional Adder Architectures

Adders/Parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)
Brent – Kung	0.276	0.27	5.63E-04	0.8551	0.39	2.35E-03	1.8843	0.55	3.97E-03
Ladner – Fischer	0.276	0.16	5.81E-04	0.8583	0.25	2.30E-03	1.8871	0.38	4.11E-03
Kogge – Stone	0.496	0.04	7.78E-04	1.6146	0.15	3.31E-03	3.701	0.29	6.21E-03
Carry – Skip	0.259	0.33	4.27E-04	0.6914	0.61	9.05E-04	1.6432	0.79	2.64E-03
Carry – Select	0.327	0.13	6.04E-04	1.3367	0.33	9.47E-04	2.707	0.62	2.81E-03

Table 7.7. Post – Layout Estimates for Flagged Adder Architectures

Adders/Parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)
Brent – Kung	0.2803	0.273	7.06E-04	0.8917	0.394	2.94E-03	2.0871	0.555	4.27E-03
Ladner – Fischer	0.2821	0.165	7.27E-04	0.9004	0.257	3.10E-03	2.1691	0.386	4.66E-03
Kogge – Stone	0.5362	0.045	9.92E-04	1.9146	0.158	4.02E-03	3.9163	0.294	7.14E-03
Carry – Skip	0.3746	0.42	6.13E-04	0.8499	0.871	1.08E-03	1.9247	1.34	3.12E-03

Table 7.8. Post – Layout Estimates for Enhanced Adder Architectures with Constant Addition

Adders/Parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)
Brent – Kung	0.2921	0.29	8.08E-04	0.9112	0.411	3.05E-03	2.1142	0.61	4.35E-03
Ladner – Fischer	0.2933	0.19	8.34E-04	0.9273	0.273	3.14E-03	2.2397	0.43	4.82E-03
Kogge – Stone	0.5462	0.08	1.12E-03	2.0375	0.27	4.10E-03	4.1183	0.39	6.47E-03
Carry – Skip	0.3881	0.41	7.72E-04	0.8736	1.01	1.17E-03	2.0667	1.701	3.48E-03
Carry – Select	0.4972	0.2	9.03E-04	1.5347	0.937	1.28E-03	3.2581	1.517	3.15E-03

Table 7.9. Post – Layout Estimates for Three - Input Adder Architectures

Adders/Parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)
Brent – Kung	0.3271	0.33	1.62E-03	1.1889	0.59	4.28E-03	3.2788	1.02	6.03E-03
Ladner – Fischer	0.3183	0.24	1.53E-03	1.1928	0.48	4.55E-03	3.4739	0.85	6.49E-03
Kogge – Stone	0.5957	0.13	2.59E-03	2.2606	0.45	5.27E-03	6.4265	0.77	7.83E-03
Carry – Save	0.3448	0.3	1.17E-03	1.2131	0.57	2.06E-03	2.4016	1.019	2.77E-03

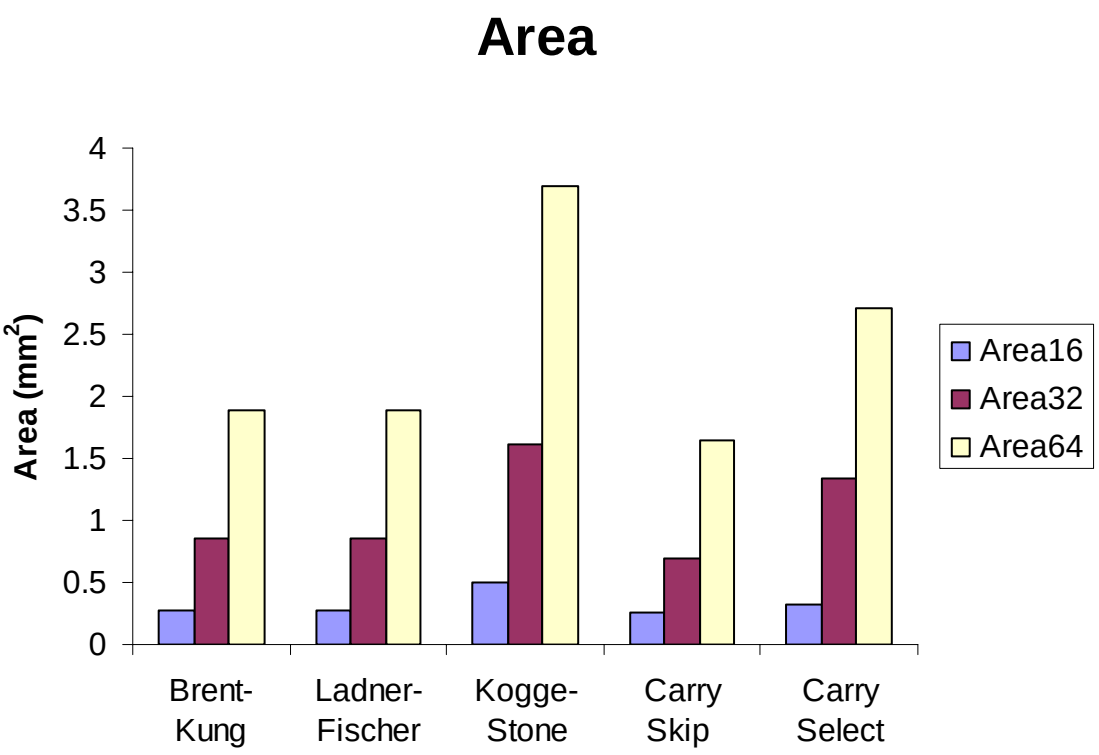

Figure 7.1. Area Results for Conventional Adder Designs

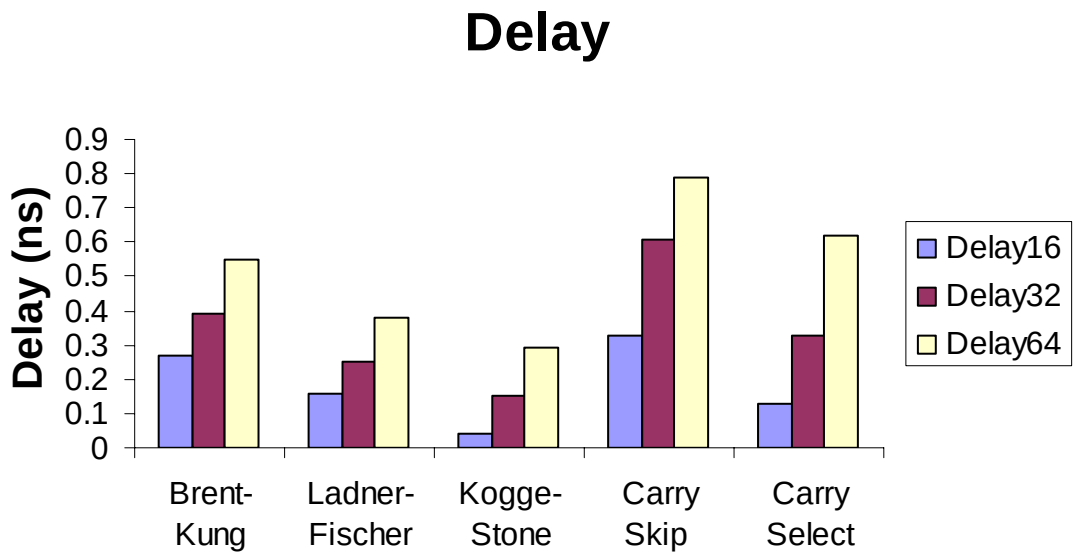

Figure 7.2. Delay Results for Conventional Adder Designs

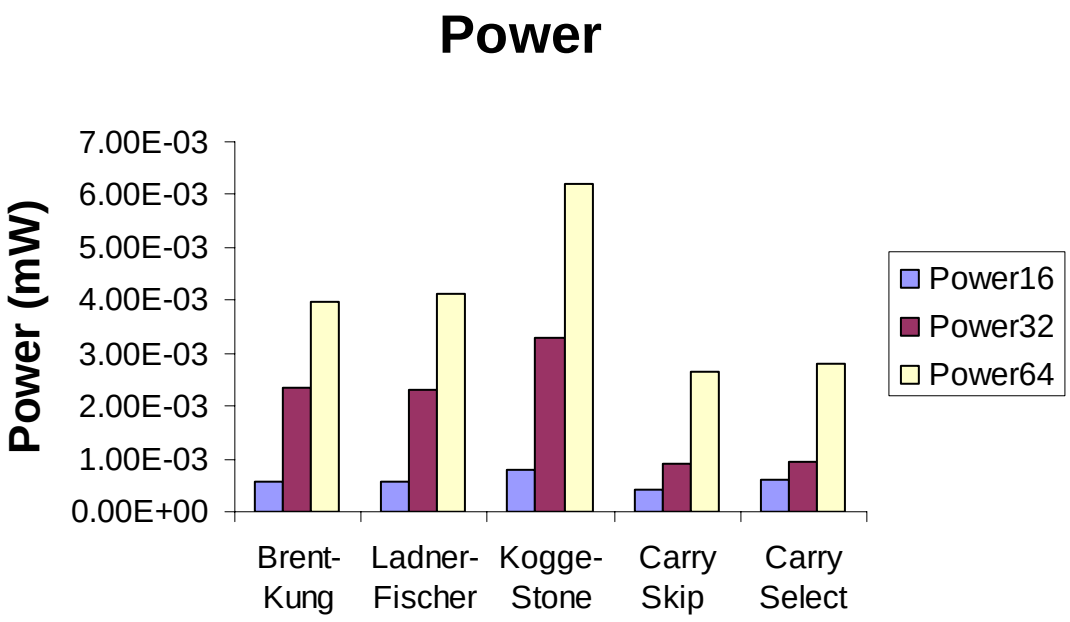

Figure 7.3. Power Results for Conventional Adder Designs

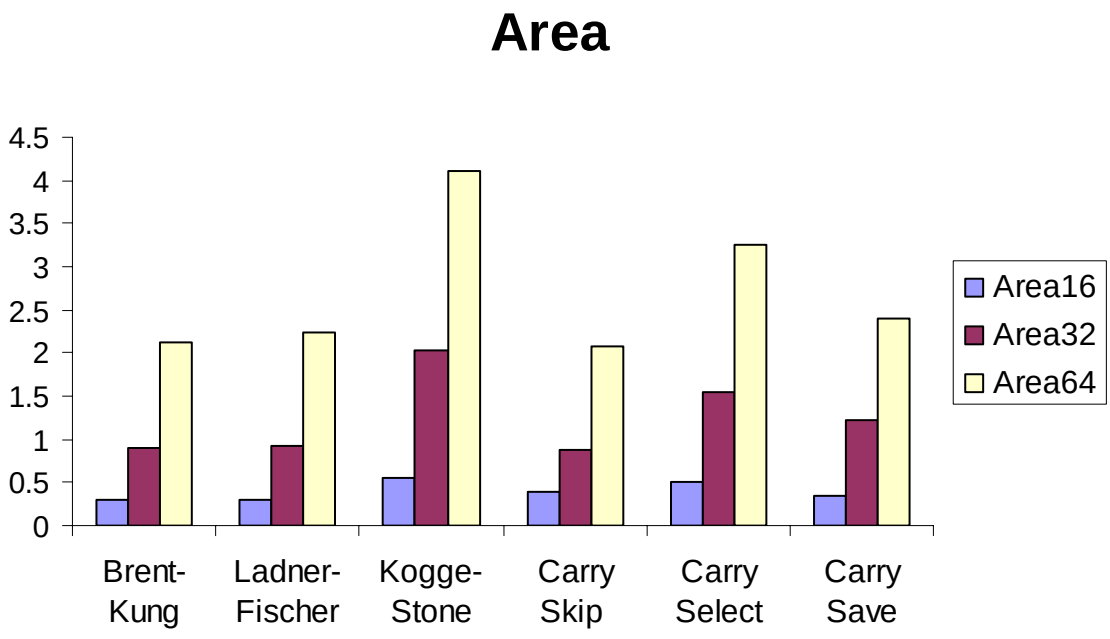

Figure 7.4. Area Results for Enhanced Adder Designs

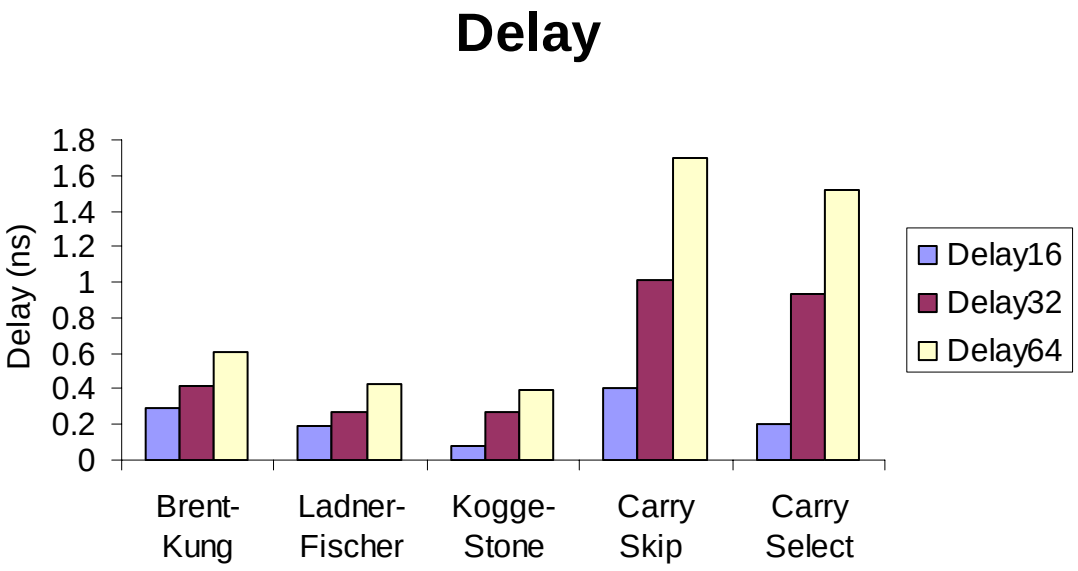

Figure 7.5. Delay Results for Enhanced Adder Designs

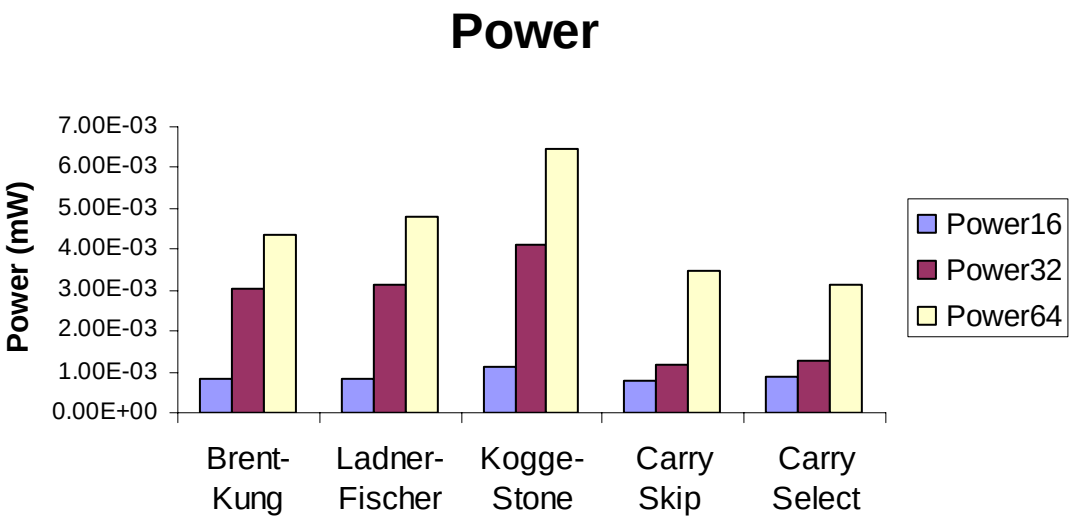

Figure 7.6. Power Results for Enhanced Adder Designs

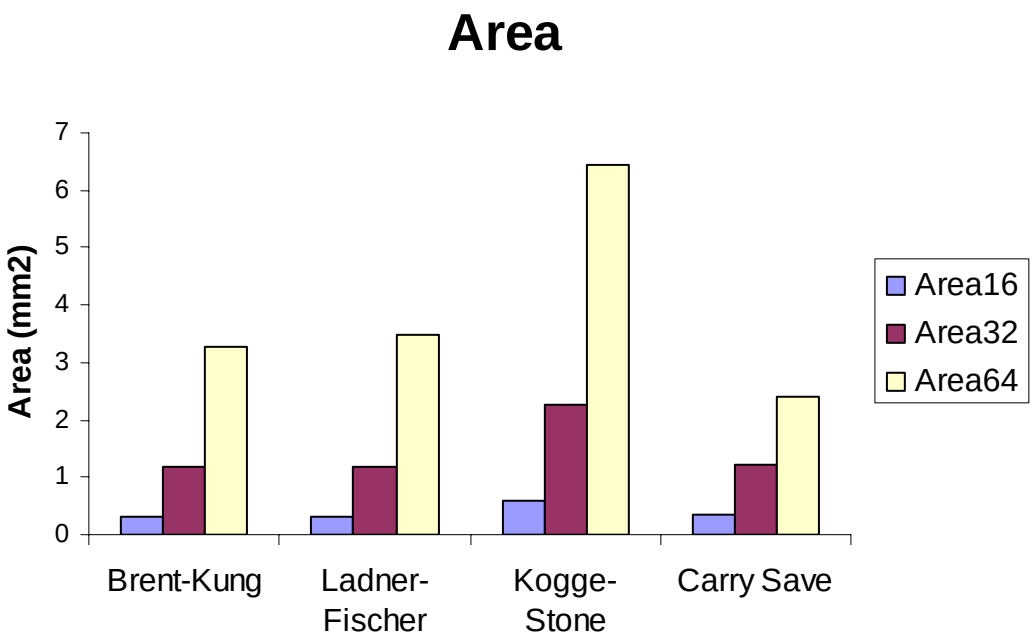

Figure 7.7. Area Results for Three - Input Adder Designs

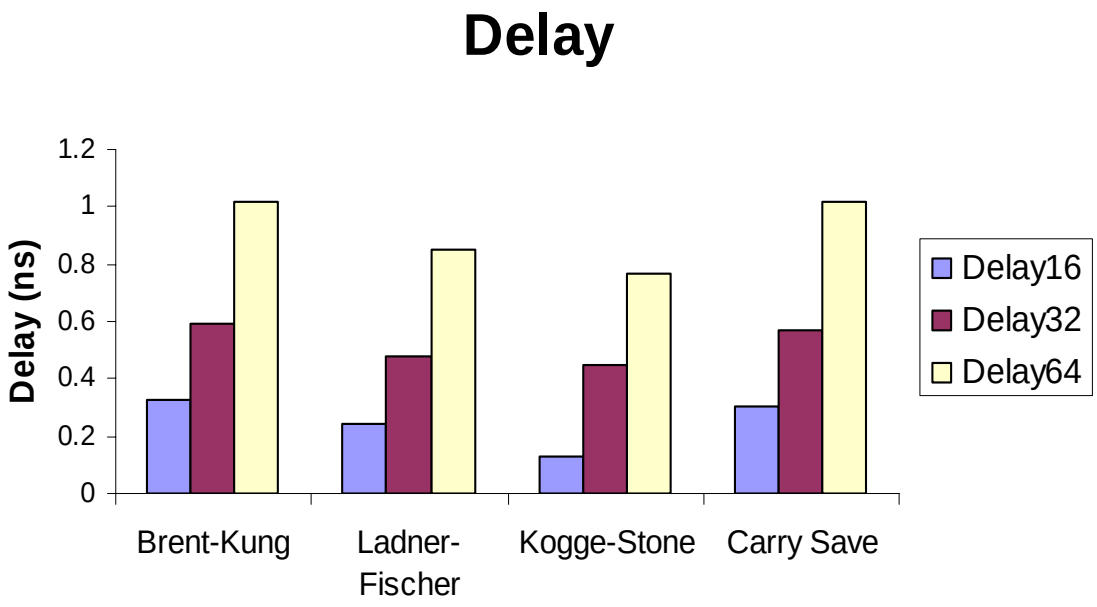

Figure 7.8. Delay Results for Three - Input Adder Designs

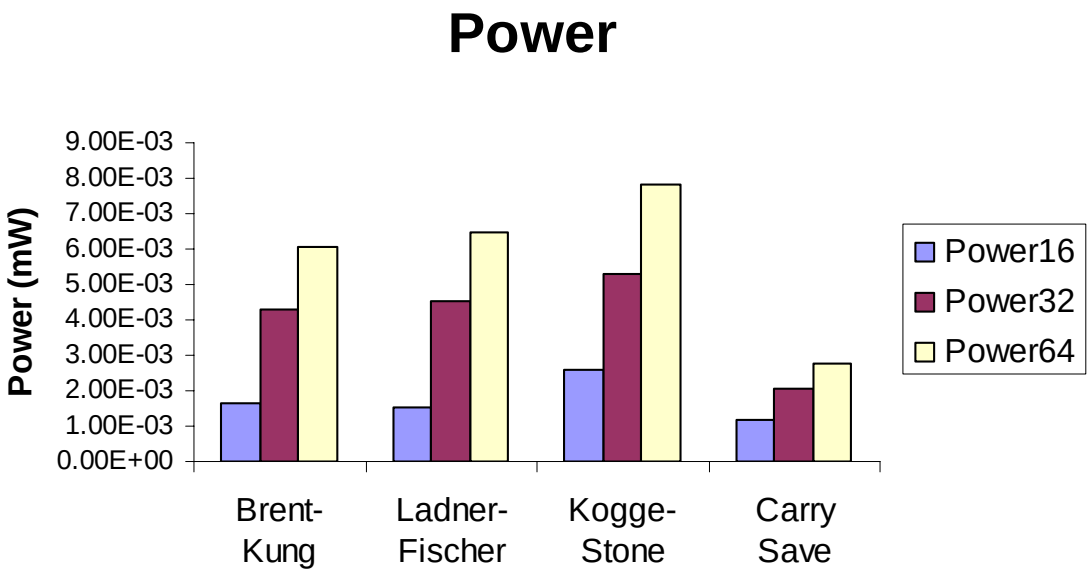

Figure 7.9. Power Results for Three - Input Adder Designs

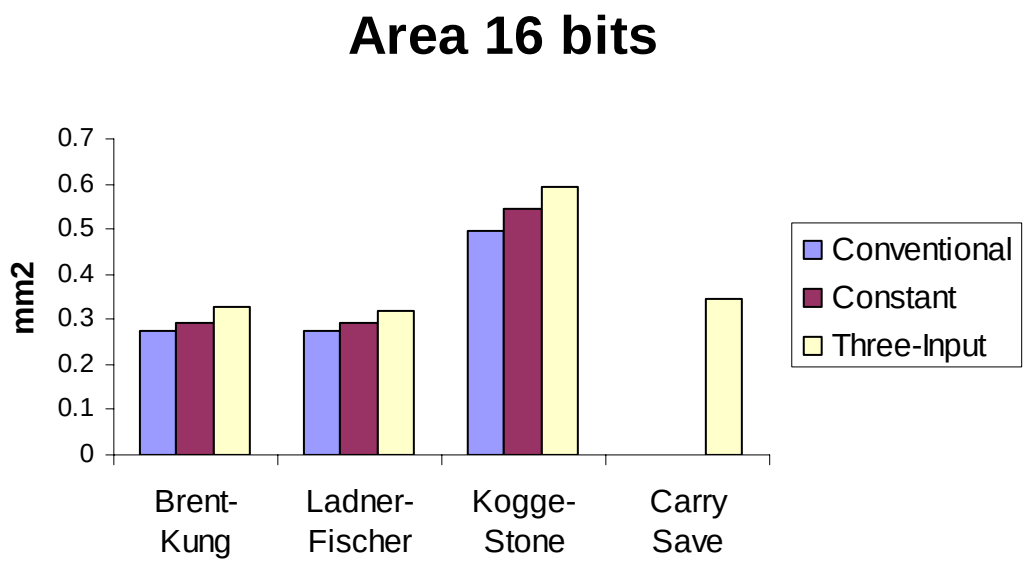

Figure 7.10. Area Results for 16 – bit Designs

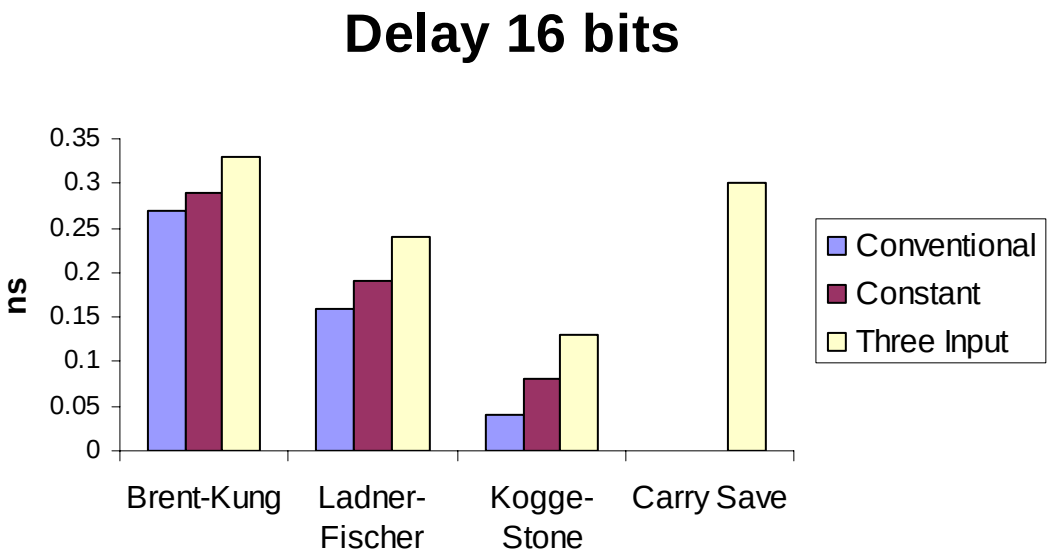

Figure 7.11. Delay Results for 16 – bit Designs

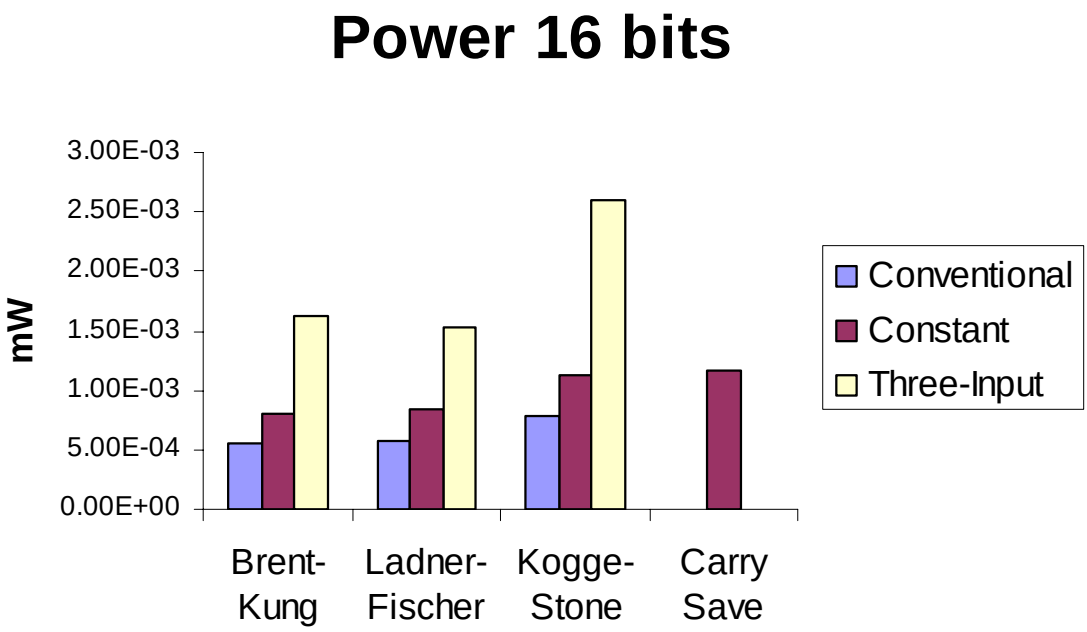

Figure 7.12. Delay Results for 16 – bit Designs

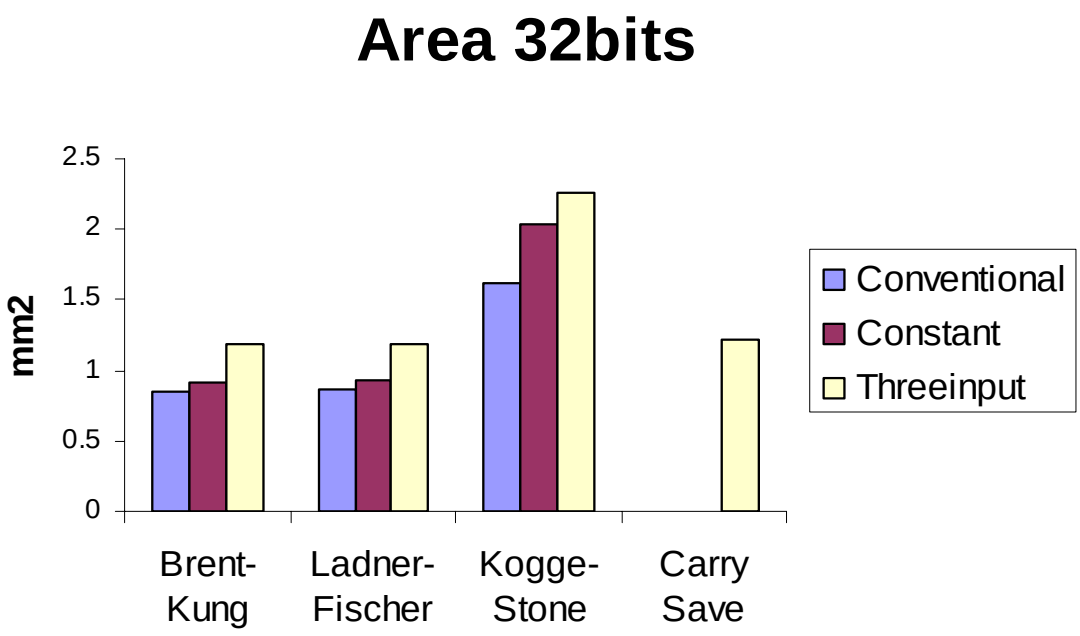

Figure 7.13. Area Results for 32 – bit Designs

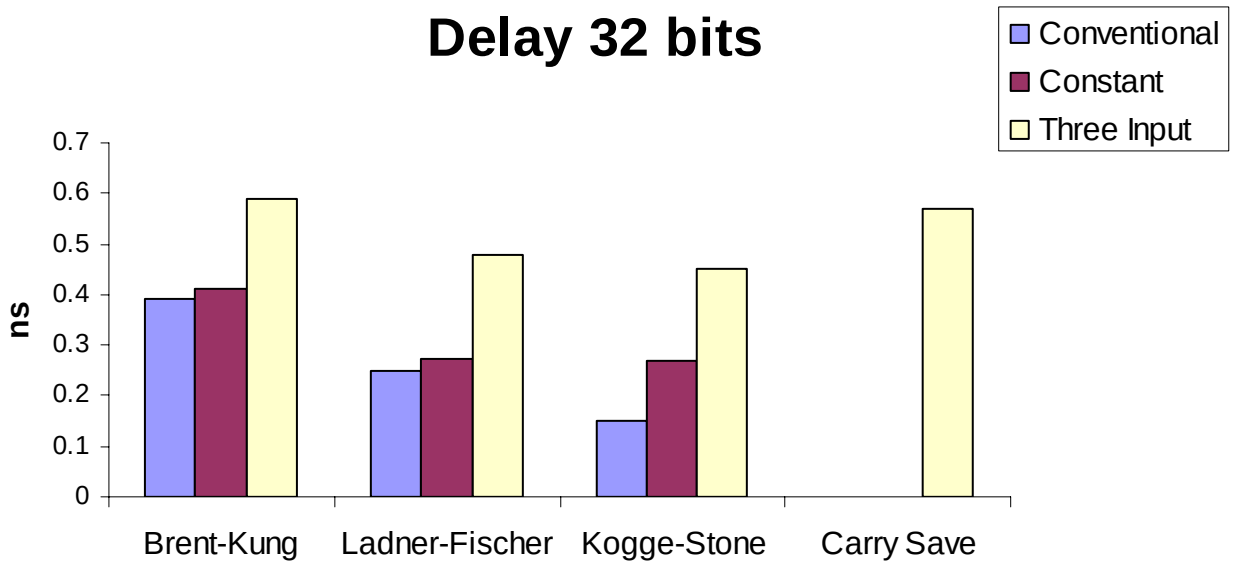

Figure 7.14. Delay Results for 32 – bit Designs

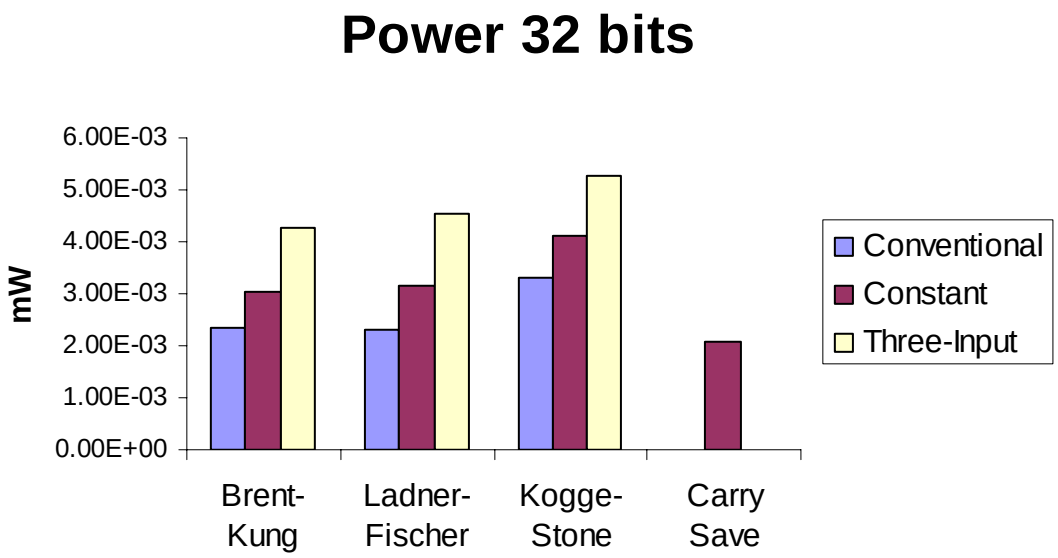

Figure 7.15. Power Results for 32 – bit Designs

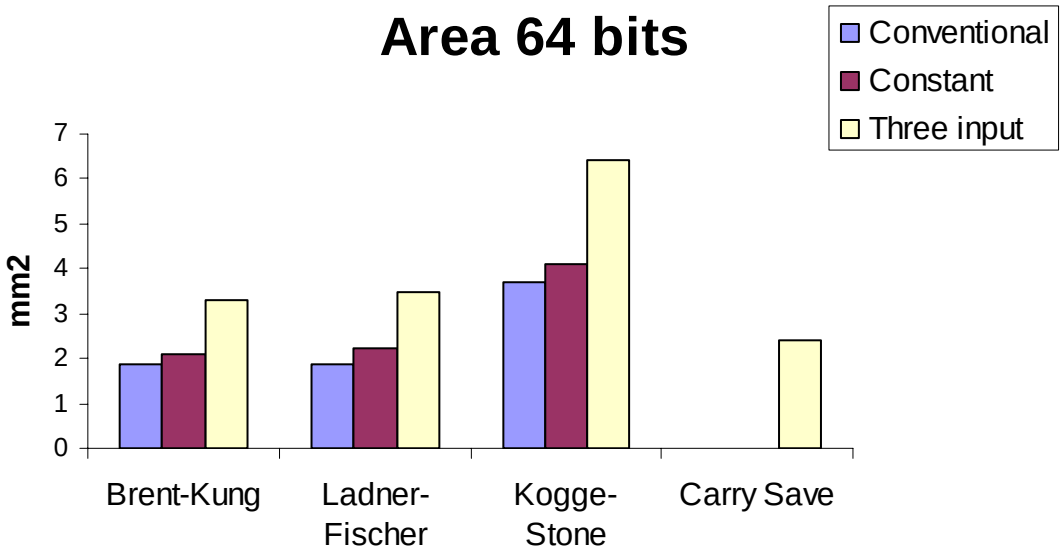

Figure 7.16. Area Results for 64 – bit Designs

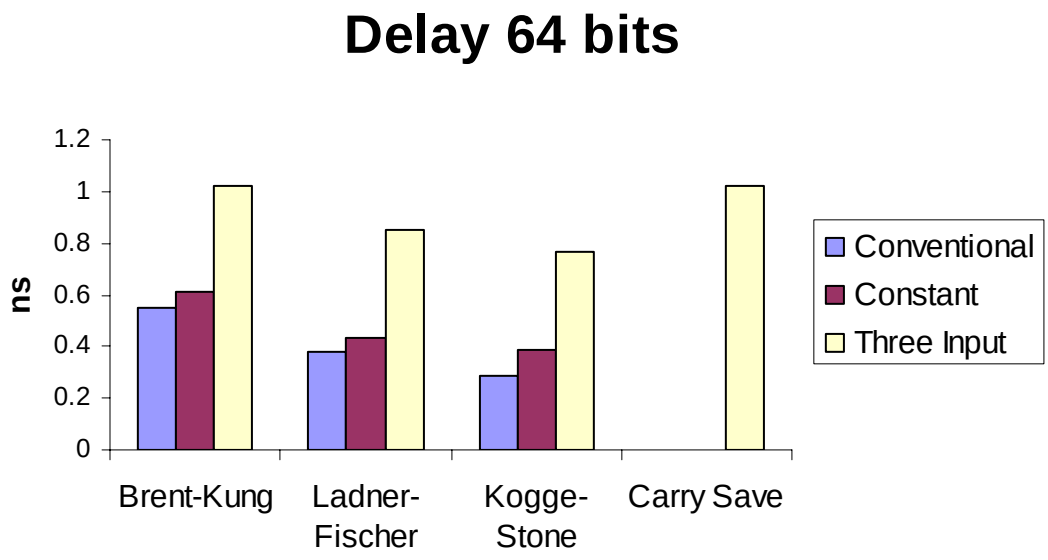

Figure 7.17. Delay Results for 64 – bit Designs

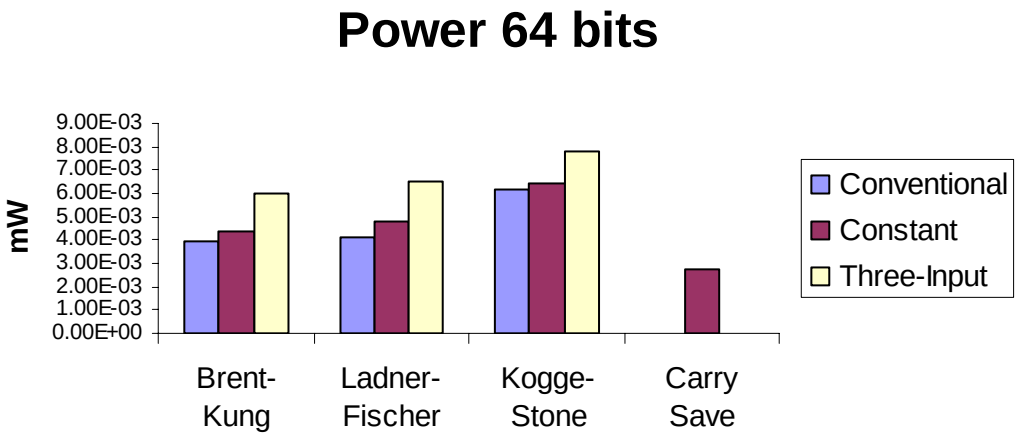

Figure 7.18. Power Results for 64 – bit Designs

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Prefix adder architectures capable of three – operand addition for cell based design and their synthesis have been designed and investigates in this thesis. Binary adders capable of constant addition have also been presented and their performance investigated. The design is possible due to the generation of a new set of intermediate outputs called *"flag"* bits. The research items and the results of this work can be summarized as follows

- Qualitative and quantitative comparisons of carry – skip, carry – select, and prefix adders for cell based designs have been carried out.
- This thesis presents an algorithm to compute an intermediate set of outputs called flag bits within a regular adder to make it capable of handling three operands at a time.
- This design can be used as a replacement to carry-save adders with the possibility of having the third operand as a constant or a variable binary number.
- The proposed technique has a comparable performance to the conventional multi-operand adder.
- It eliminates the need to have dedicated adder units to perform the operation since the new logic is incorporated within a regular adder.
- The Kogge-Stone adder has a favorable performance in terms of speed, with the trade off being high power and area consumption.

- A good compromise could be reached utilizing the Ladner-Fischer prefix architecture to achieve good area and power numbers.

Finally, the following outlook and topics for future work can be formulated:

- The hardware will be optimized by gate sizing in order to achieve better performance results.
- Flag bits will be introduced to a Zero – Deficiency Adder design proposed in [57]. This is predicted to result in delay and power efficient circuits compared to the Kogge – Stone which is the most delay efficient prefix adder so far.
- Power saving techniques proposed in [40] [12] [30] will be employed to save on power consumption
- The adder will be implemented in practical application designs like decimal Arithmetic and multiplier units to study the impact and performance gain that can be projected.

BIBLIOGRAPHY

[1] Abu-Khater, I.S., R.H. Yan, A. Bellaouar, and M.I. Elmasry M.I. "A 1-V Low-Power High-Performance 32-Bit Conditional Sum Adder." Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium, 1994. 66-67.

[2] Beaumont-Smith, A., N. Burgess, S.Cui, and MLiebelt . "Gaas Multiplier and Adder Designs for High-Speed DSP Applications." Signals, Systems & Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on, 1997. 1517-21 vol.2.

[3] Beaumont-Smith, A., N. Burgess, S.Lefrere, and C.C.Lim. "Reduced Latency IEEE Floating-Point Standard Adder Architectures." Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on, 1999. 35-42.

[4] Becker, B., R. Drechsler, and P. Molitor. "On the Generation of Area-Time Optimal Testable Adders." Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 14.9 (1995): 1049-66.

[5] Brent, R.P., and H.T. Kung. "A Regular Layout for Parallel Adders." Computers, IEEE Transactions on C-31.3 (1982): 260-64.

[6] Burgess, N. "The Flagged Prefix Adder and Its Application in Integer Arithmetic." The Journal of VLSI Signal Processing 31.3 (2002): 263-71.

[7] Burgess, N., "Accelerated Carry-Skip Adders with Low Hardware Cost." Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on, 2001. 852-56 vol.1. Vol. 1.

[8] Burgess, N., "The Flagged Prefix Adder for Dual Addition." In Proceedings of SPIE-The International Society for Optical Engineering, 1998. 567-75. Vol. 3461.

[9] Burgess, N., and S. Knowles. "Efficient Implementation of Rounding Units." Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on, 1999. 1489-93 Vol. 2.

[10] Cavanagh, J. J. F., Digital Computer Arithmetic: Design and Implementation. McGraw-Hill, 1984.

[11] Chandrakasan, A., Low Power Digital CMOS Design. Kluwer, Nowell, 1995.

[12] Chandrakasan, A.P., and R.W. Brodersen. "Minimizing Power Consumption in Digital CMOS Circuits." Proceedings of the IEEE 83.4 (1995): 498-523.

[13] Chen, Y., W.K. Tsai, and F.J. Kurdahi. "Layout Driven Logic Synthesis System." Circuits, Devices and Systems, IEE Proceedings- 142.3 (1995): 158-64.

[14] Cheng, S.W., H.-C. Chen, D.H.C. Du, and A Lim. "The Role of Long and Short Paths in Circuit Performance Optimization." Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE, 1992. 543-48.

[15] Cortadella, J., and J.M. Llaberia. "Evaluation of A+B=K Conditions without Carry Propagation." Computers, IEEE Transactions on 41.11 (1992): 1484-88.

[16] Dadda, L., and V. Piuri. "Pipelined Adders." Computers, IEEE Transactions on 45.3 (1996): 348-56.

[17] Dave, V., E. Oruklu, and J. Saniie. "Performance Evaluation of Flagged Prefix Adders for Constant Addition." Electro/information Technology, 2006 IEEE International Conference on, 2006. 415-20.

[18] De Gloria, A., and M. Olivieri. "Statistical Carry Lookahead Adders." Computers, IEEE Transactions on 45.3 (1996): 340-47.

[19] Efstathiou, C., D. Nikolos, and J. Kalamatianos. "Area-Time Efficient Modulo 2^N-1 Adder Design." Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on] 41.7 (1994): 463-67.

[20] Ercegovac, M., and Tomas Lang. Digital Arithmetic. Kauffmann, 2004.

[21] Fich, F. E., "New Bounds for Parallel Prefix Circuits." Proceedings of the fifteenth annual ACM symposium on Theory of computing, 1983. 100-09.

[22] Grad, J., and J.E. Stine. "A Standard Cell Library for Student Projects." Microelectronic Systems Education, 2003. Proceedings. 2003 IEEE International Conference on, 2003. 98-99.

[23] Guyot, A., B. Hochet, and J.-M. Muller. "A Way to Build Efficient Carry-Skip Adders." Computers, IEEE Transactions on C-36.10 (1987): 1144-52.

[24] Han, T., and D.A. Carlson. "Fast Area Efficient VLSI Adders." In proceedings. Eighth Computer Arithmetic Symposium, 1987. 49-56.

[25] Harris, D., and I. Sutherland. "Logical Effort of Carry Propagate Adders." Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on, 2003. 873-78 Vol.1. Vol. 1.

[26] Hobson, R.F., "Optimal Skip-Block Considerations for Regenerative Carry-Skip Adders." Solid-State Circuits, IEEE Journal of 30.9 (1995): 1020-24.

[27] Hodges, D., H. Jackson, and R. Saleh. Analysis and Design of Digital Integrated Circuits in Deep Submicron Technology. Mc Graw Hill, 2004.

[28] Kantabutra, V., "Designing Optimum One-Level Carry-Skip Adders." Computers, IEEE Transactions on 42.6 (1993): 759-64.

[29] Katz, R., Contemporary Logic Design. Benjamin Cummings, 1997.

[30] Kitahara, T., and R.K. Brayton. "Low Power Synthesis Via Transparent Latches and Observability Don't Cares." Circuits and Systems, 1997. ISCAS '97., Proceedings of 1997 IEEE International Symposium on, 1997. 1516-19 vol.3. Vol. 3.

[31] Knowles, S., "A Family of Adders." Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on, 2001. 277-81.

[32] Kogge, P.M, and H.S Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations." IEEE Transactions on Computers 22.8 (1973): 783-91.

[33] Koren, I., Computer Arithmetic Algorithms. Prentice Hall, 1993.

[34] Ladner, R.E, and M.J Fischer. "Parallel Prefix Computation." Journal of ACM 27.4 (1980): 831-38.

[35] Lindkvist, H., and P. Andersson. "Dynamic CMOS Circuit Techniques for Delay and Power Reduction in Parallel Adders." Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on, 1995. 121-30.

[36] Lindkvist, H., and P. Andersson. "Techniques for Fast CMOS-Based Conditional Sum Adders." Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on, 1994. 626-35.

[37] Lo, J.C., "Correction to a Fast Binary Adder with Conditional Carry Generation"." Computers, IEEE Transactions on 47.12 (1998): 1425.

[38] Lynch, T., and E.E. Swartzlander, Jr., "A Spanning Tree Carry Lookahead Adder." Computers, IEEE Transactions on 41.8 (1992): 931-39.

[39] Micheli, G. De., Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.

[40] Nagendra, C., M.J. Irwin, and R.M. Owens. "Area-Time-Power Tradeoffs in Parallel Adders." Circuits and Systems II: Analog and Digital Signal Processing,

IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on] 43.10 (1996): 689-702.

[41] Najm, F.N., "A Survey of Power Estimation Techniques in VLSI Circuits." Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 2.4 (1994): 446-55.

[42] Parhami, B., Computer Arithmetic: Algorithms and Hardware Designs. Oxford Univerity Press, 2000.

[43] Patterson, D., and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface. Morgan Kauffmann Publishers, 2005.

[44] Rabaey, J., A. Chandrakasan, and B. Nikolic. Digital Integrated Circuits: A Design Perspective. Prentice Hall, 2003.

[45] Sagdeo, Vivek. The Complete Verilog Book. Springer, 1998.

[46] Sklansky, J., "Conditional Sum Addition Logic." IRE Transactions on Electron Computers EC-9.6 (1960): 226-31.

[47] Srinivas, H.R., and K.K. Parhi. "A Fast VLSI Adder Architecture." Solid-State Circuits, IEEE Journal of 27.5 (1992): 761-67.

[48] Stine, J.E., C.R. Babb, and V.B. Dave. "Constant Addition Utilizing Flagged Prefix Structures." Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on, 2005. 668-71 Vol. 1.

[49] Stine, J.E., J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Iliev, and N. Jachimiec. "A Framework for High-Level Synthesis of System on Chip Designs." Microelectronic Systems Education, 2005. (MSE '05). Proceedings. 2005 IEEE International Conference on, 2005. 67-68.

[50] Sutherland, I., Bob Sproull, and David Harris. Logical Effort: Designing Fast CMOS Circuits. Morgan Kauffmann, 1999.

[51] Turrini, S., "Optimal Group Distribution in Carry-Skip Adders." In Proceedings of the Ninth Computer Arithmetic System. Santa Monica, CA, 1989. 96-103.

[52] Tyagi, A., "A Reduced-Area Scheme for Carry-Select Adders." Computers, IEEE Transactions on 42.10 (1993): 1163-70.

[53] Tyagi, A.,"A Reduced-Area Scheme for Carry-Select Adders." Computers, IEEE Transactions on 42.10 (1993): 1163-70.

[54] Wei, B.W.Y., and C.D. Thompson. "Area-Time Optimal Adder Design." Computers, IEEE Transactions on 39.5 (1990): 666-75.

[55] Weste, Neil, and David Harris. CMOS VLSI Design. Addison Wesley, 2005.

[56] Xu, M., and F.J. Kurdahi. "Layout-Driven High Level Synthesis for Fpga Based Architectures." Design, Automation and Test in Europe, 1998., Proceedings, 1998. 446-50.

[57] Zhu H, Chun-Kuan Cheng and Ronald Graham. "Constructing Zero-Deficiency Parallel Prefix Circuits of Minimum Depth." ACM Transactions on Design Automation of Electronic Systems V.N (2005).

[58] Zimmerman, R. "Binary Adder Architectures for Cell Based VLSI Design." Swiss Federal Institute of Technology, 1997.