DESIGNING A SELF-TIMED ARITHMETIC LOGIC UNIT

BY

FERAS ALI MOHAMMED MAADI

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

APRIL 2004

UMI Number: 1420769

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1420769

Copyright 2004 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

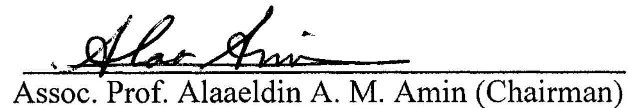
This thesis, written by

FERAS ALI MOHAMMED MAADI

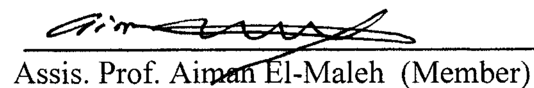
under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

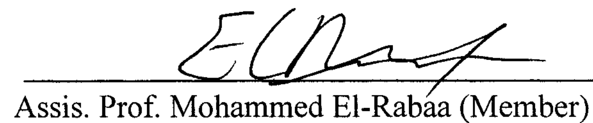
Thesis Committee



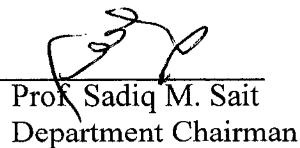
Assoc. Prof. Alaaeldin A. M. Amin (Chairman)



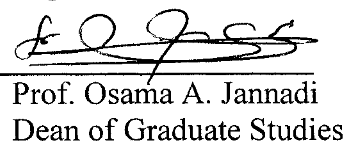
Assis. Prof. Aiman El-Maleh (Member)



Assis. Prof. Mohammed El-Rabaa (Member)



Prof. Sadiq M. Sait
Department Chairman



Prof. Osama A. Jannadi
Dean of Graduate Studies

Date May 9, 2004

Dedicated to my dear wife *Reem Salem*,

my sweetheart son *Ali*,

my lovely little daughter *Tala*

ACKNOWLEDGMENTS

Acknowledgment is due to the King Fahd University of Petroleum & Minerals (KFUPM) for supporting this research. I acknowledge the academic and computing facilities provided by the Computer Engineering Department of KFUPM.

I wish to express my appreciation to Dr. Alaaeldin Amin, my thesis advisor, for his great help, encouragement, and motivation where this thesis is never possible without his support. Dr. Amin taught me many things, but most importantly, how to do research and how to write well. I am deeply indebted to him for being my advisor and for having a complete faith in me. I would like also to thank Dr. Aiman El-Maleh and Dr. Mohammed El-Rabaa, my thesis committee members, for their comments and critical review of the thesis.

I would like to pay a heartily tribute to all of my family members and especially to my parents, who motivated me to continue and achieve higher academic goals. I would like to acknowledge the moral support and sincere prayers of my wife.

Finally, I appreciate the friendly support from all my colleagues at KFUPM, particularly, Maher Al Shareef and from my best friend Dr. Fadi Fallouh.

Contents

ACKNOWLEDGMENTS iv

LIST OF TABLES x

LIST OF FIGURES xiii

ABSTRACT (ENGLISH) xix

ABSTRACT (ARABIC) xxi

1 INTRODUCTION 1

2 ASYNCHRONOUS DESIGN METHODOLOGIES 4

2.1 Bounded Delay Models ... 7

2.1.1 Fundamental Mode Huffman Circuits ... 7

2.1.2 Non-Fundamental Mode Model ... 11

2.1.3 Burst-mode Model ... 13

2.2 Micropipelines ... 16

2.3 Delay-insensitive Model ... 21

2.4 Speed-independent Model 23

2.5 Quasi-delay-insensitive Model 24

3 FIXED-POINT ADDERS **26**

3.1 Synchronous Adders 27

3.1.1 Serial Adder 27

3.1.2 Carry-Ripple Adder 30

3.1.3 Carry Skip Adder 31

3.1.4 Manchester Carry Adder 34

3.1.5 Carry Look-Ahead Adder 37

3.1.6 Carry Select Adder 41

3.1.7 Conditional Sum Adder 42

3.2 Asynchronous Fixed-point Adders 44

3.2.1 Completion Detection 45

3.2.2 Speculative Completion 49

3.2.3 Bundled Data Based Adder 50

4 ARITHMETIC LOGIC UNITS **54**

4.1 Basic Functions of an ALU 55

4.2 Implementation of Asynchronous ALUs 56

5 ADDER-BASED SELF-TIMED ALU **64**

5.1 The Basic Adder Cell 64

5.1.1 Introduction 65

5.1.2 Double Rail Encoding 66

5.1.3 Implementation of Double Rail Encoding 70

5.1.4 Balancing the Capacitive Load 78

5.2 Completion Detection for 4-bit MCC Adder 85

5.3 Completion Detection for n-bit MCC Adder 89

5.4 The Asynchronous ALU 92

5.4.1 Arithmetic Operations 94

5.4.2 Logical Operations 98

5.4.3 The Modified Adder Cell 104

5.4.4 Shift Operations 108

5.4.5 Barrel Shifter Implementation 109

5.4.6 The Overall Design 116

5.4.7 Alternative Approach 123

5.4.7.1 The Data Path 124

5.4.7.2 Completion Detection Circuitry 127

5.4.7.3 Comparison 129

6 DESIGN VERIFICATION AND ANALYSIS 132

6.1 Delay Analysis 133

6.1.1 Worst Case Delay (WCD) 134

6.1.2 Worst Case Functionality (WCF) 136

6.2 Simulation Results 138

6.2.1 Simulation Results for Self-Timed Adder 138

6.2.2 Simulation Results for the Self-Timed ALU 139

6.2.2.1 Simulation Results under Normal Conditions 139

6.2.2.1.1 Arithmetic Operations 140

6.2.2.1.2 Logical and Shift Operations 147

6.2.2.2 Simulation Results under Other Conditions ... 150

6.2.2.3 Simulation Result for Different Process Technologies 153

6.2.3 Comparison with the Alternative Approach 154

6.2.4 Comparison with other Completion Detection Approaches 156

7 SPEED UP TECHNIQUES **161**

7.1 Carry Bypass 162

7.2 Carry Skip 164

7.3 Partial Carry Skip 166

8 CONCLUSIONS AND FUTURE WORK **169**

BIBLIOGRAPHY **172**

VITA

List of Tables

5.1 Double Rail Encoding of the Carry Signal.............................. 70

5.2 Truth Table of an Adder... 71

5.3 Discharge Paths for the Carry Nodes of MCC-1 & MCC-0.......... 74

5.4 Different Choices for Selecting Carry Nodes.......................... 79

5.5 Elmore Delays of the Carry Nodes.................................... 81

5.6 Operands at Port-A, Port-B, C_{in} for Arithmetic Operations.......... 96

5.7 Multiplexing Values at Port-A, Port-B for the Arithmetic Operations... 97

5.8 Implementing AND Operation from the Adder Resources........... 100

5.9 Implementing OR Operation from the Adder Resources............. 101

5.10 Implementing Logical Operations Using The Adder Resources..... 102

5.11 Multiplexing inputs at XOR & XNOR Gates.......................... 104

5.12 Values of F, Phi0, Phi1 During Evaluate Phase........................ 108

5.13 Implementation of Different Shift Operations........................ 111

5.14 The Control Signals of Shift Operations.............................. 113

5.15 A General Scheme for Implementing *nXn* Barrel Shifter........... 115

5.16 Status of the Control Signals for all Operations....................... 118

5.17 Saved and Added Delay Components for Arithmetic Operations... 129

5.18 Saved and Added Delay Components for Logical Operations....... 129

5.19 Saved and Added Delay Components for Shift Operations.......... 130

6.1 Simulation Results for N-bit Adder................................... 139

6.2 Delay of Major Signals (Case 2).. 145

6.3 Delay of Major Signals (Case 3).. 147

6.4 XNOR operation Signal Delays (Case 1)............................... 148

6.5 XOR operation Signal Delays (Case 2)................................. 149

6.6 Delay time of important signals under (4V,125°C) condition........ 151

6.7 Delay time of important signals under (6V,-55°C) condition......... 151

6.8 Simulation Results for (4V, 125°C) Operating Conditions........... 152

6.9 Simulation Results for (6V, -55°C) Operating Conditions........... 152

6.10 Delay time of important signals for 0.18μ CMOS at (1.8V,27°C)... 153

6.11 Simulation Results for 0.18μ CMOS at (1.8V,27°C)................. 154

6.12 Simulation Results for the Alternative Approach...................... 155

6.13 Simulation Results for the Proposed Approach........................ 155

6.14 Comparison of Completion Detection Approaches.................... 160

7.1 Propagation Delay for Different Carry Bypass Locations........... 164

7.2 Worst Case Propagation Delay for Different Speed Up Techniques of a 64-bit MCC... 167

7.3 Worst Case Delays for a 16-bit ALU with and without partial carry skip.. 168

List of Figures

2.1 A Flow Table and the Corresponding State Machine.................. 8

2.2 Implementation of a Circuit with Hazards.............................. 9

2.3 Huffman Sequential Circuit Model...................................... 10

2.4 Hollaar's Implementation of the State Diagram........................ 12

2.5 Locally Clocked Implementation... 14

2.6 Two-level AND-OR Network Implementation.......................... 15

2.7 Two equivalent transitions.. 16

2.8 A Bundled Data Interface.. 17

2.9 The Two-Phase Bundled Data Convention.............................. 18

2.10 Control and Data Path for a Micropipeline.............................. 20

2.11 Data Transfer via Transition Signaling................................. 22

2.12 An Isochronic Fork (left) and an Equivalent Speed-independent Circuit (right)... 23

2.13 Examples of Quasi-Delay-Insensitive Circuits that are not Delay-Insensitive........ 24

3.1 A Full Adder Unit........ 29

3.2 A Serial Adder........ 29

3.3 Carry Ripple Adder........ 31

3.4 Carry Skip Adder with one level skip logic........ 33

3.5 The elemental circuit for Manchester Carry Chain........ 36

3.6 A 4-bit Manchester Carry Chain........ 36

3.7 A 4-bit Manchester Carry Chain with carry skip........ 37

3.8 General Organization of Carry Select Adder........ 42

3.9 Carry Completion Adder Stage........ 46

3.10 A Generalized DCVSL Gate........ 47

3.11 General Architecture of Speculative Completion Method........ 50

3.12 Architecture of the CEA........ 53

4.1 DCVSL Sum and Carry gates used in the Self-timed ALU........ 58

4.2 An example of two-rail implementation of function F(A,B,C) with completion signal... 60

4.3 The Execution Pipe of AMULET1.. 61

4.4 The self-timed adder of AMULET1....................................... 62

5.1 A 4-bit Manchester Carry Chain.. 66

5.2 The implementation of the MCC-1 block................................ 68

5.3 The implementation of the MCC-0 block................................ 68

5.4 The Basic Slice for Implementing MCC-1............................... 73

5.5 The Basic Slice for Implementing MCC-0............................... 73

5.6 Block Diagram of a 4-bit MCC adder.................................... 77

5.7 Implementation of Propagate-Kill-Generate Block...................... 77

5.8 RC Delay Model of MCC-1, MCC-0...................................... 78

5.9 RC Delay Models of the Three Choices.................................. 80

5.10 Implementation of the XOR Gate.. 84

5.11 Implementation of the XNOR Gate...................................... 84

5.12 Completion Detection Circuit for a 4-bit MCC Adder................. 87

5.13 The Modified 4-bit MCC Adder.. 88

5.14 N-bit MCC Adder.. 89

5.15 Completion Detection Circuit For 16,32,64-bit MCC Adder........... 91

5.16 Block diagram of the self-timed ALU................................... 92

5.17 Multiplexing at Port A.. 96

5.18 Multiplexing at Port B.. 97

5.19 Multiplexing inputs of XOR and XNOR gates.......................... 103

5.20 The Modified Elemental MCC Cell...................................... 106

5.21 The Modified MCC-1.. 107

5.22 The Modified MCC-0.. 107

5.23 Interconnection of Adder and Barrel Shifter............................ 109

5.24 A 4X4 Barrel Shifter.. 112

5.25 Minimization of the 4X4 Barrel Shifter.................................. 114

5.26 Block Diagram for the Proposed 4-bit Self-Timed ALU............... 117

5.27	The Delayed Request Circuit...	119
5.28	The Delay Matching Circuit to get the FCS...	120
5.29	The Latch Circuit for the Result...	120
5.30	The Latch Circuit for the FCS...	121
5.31	The Handshake Protocol Between REQ and ACK...	122
5.32	Data path of the Alternative Approach...	126
5.33	Completion Detection Circuitry of the Alternative Approach...	128
6.1	Cascading a number of 4-bit MCC blocks...	135
6.2	Completion Detection Circuitry for a 4-bit ALU...	135
6.3	Worst case functionality...	137
6.4	Completion Detection Circuitry for an 8-bit ALU...	137
6.5	REQUEST signal VS Time...	142
6.6	Delayed Request (REQD) VS Time...	142
6.7	Sum bit F(16) VS Time...	143
6.8	FCS VS Time...	143

6.9 Latched Sum Bit LF(16) VS Time.. 144

6.10 ACK VS Time.. 144

6.11 Completion Detection Proposed by Nowick............................. 158

6.12 Completion Detection Proposed by Gustavo............................ 159

7.1 Carry Bypass Circuit... 162

7.2 Carry Bypass Circuit within MCC Blocks............................... 163

7.3 Carry Skip Technique... 165

7.4 Partial Carry Skip... 166

THESIS ABSTRACT

Name: FERAS ALI MOHAMMED MAADI

Title: DESIGNING A SELF-TIMED ARITHMETIC LOGIC UNIT

Major Field: COMPUTER ENGINEERING

Date of Degree: April 2004

With the continuous advances in VLSI technology, designing self-timed (asynchronous) digital systems has been gaining more importance. Self-timing solves several problems, e.g. worst case delay and clock skew, inherent in synchronous designs. Although self-timed circuits have several advantages over their synchronous counterparts, designing self-timed circuits is a much more difficult task. Self-timing requires a handshaking protocol between its modules. A completion signal is generated by a self-timed circuit to flag the completion of computation. The self-timed approach has been used to implement various Arithmetic Logic Units (ALUs) whose performance is largely dependent on the adder speed. Several self-timed adders have been reported in the literature using

different techniques in generating the completion signal. Hardware and delay overhead due to the added completion detection circuitry are potential disadvantages of self-timed adders. With proper implementation, however, such disadvantages can be minimized. The objective of this work is to design an efficient self-timed ALU that is based on a self-timed adder using two Manchester carry chains with completion detection circuitry.

MASTER OF SCIENCE DEGREE

King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia

April 2004

ملخص الرسالة

الاسم : فراس علي محمد معدي

عنوان الدراسة : تصميم وحدة المنطق و الحساب ذاتية التوقيت

التخصص : هندسة الحاسب الآلي

تاريخ التخرج : ابريل ٢٠٠٤

مع استمرارية التقدم و التطور في تقنية تصميم الدوائر الرقمية ذات النطاق الواسع، فان تصميم الأنظمة الرقمية ذاتية التوقيت (اللاتوافقية) قد أصبح مهما. ان فكرة استخدام طريقة ذاتية التوقيت في تصميم الدوائر و الأنظمة الرقمية قد حل مشاكل متعددة موجودة في التصاميم التوافقية مثل انحراف الساعة و انخفاض الأداء لارتباطه بأسوأ حالة تأخير.

على الرغم من أن الدوائر الرقمية ذاتية التوقيت (اللاتوافقية) تمتلك مميزات اضافية لدى مقارنتها بمنافستها من الدوائر التوافقية، الا أن تصميم الدوائر ذاتية التوقيت هي عملية أصعب من تصميم الدوائر التوافقية.

ان طريقة ذاتية التوقيت تعتمد على توليد اشارة و ذلك عند الانتهاء من تنفيذ مهة معينة و ذلك بواسطة دائرة ذاتية التوقيت. و هذه الطريقة قد استخدمت في تصميم العديد من وحدات الحساب و المنطق و التي تعتمد في أدائها بشكل كبير على سرعة و حدة الجمع، حيث أن هنالك العديد من و حدات الجمع ذاتية التوقيت و التي تستخدم طرقا مختلفة في توليد اشارة الانتهاء من أداء المهمة الحسابية. ان من سيئات

استخدام طريقة ذاتية التوقيت هي الحاجة الى دائرة ذاتية التوقيت و التي تعتبر عبء اضافي من ناحية زيادة مساحة التصميم و التي ينتج عنها نقص في أداء التصميم، الا أنه باستخدام التطبيق الصحيح لدائرة ذاتية التوقيت فانه من الممكن تقليص هذه الناحية السلبية.

ان الهدف من هذا البحث هو بناء تصميم لوحدة حساب و منطق ذاتية التوقيت و التي تعتمد على وحدة الجمع ذاتية التوقيت و ذلك باستخدام سلسلتي مانشيستر مع دائرة ذاتية التوقيت ذات كفاءة عالية.

درجة الماجستير في العلوم
جامعة الملك فهد للبترول و المعادن
الظهران- المملكة العربية السعودية
ابريل ٢٠٠٤

CHAPTER 1

INTRODUCTION

Recently, self-timed (asynchronous) circuits have received a great deal of attention due to the spectacular capacity improvements in VLSI technology. With fast switching devices, wire delay and clock skew are becoming serious problems to synchronous circuits. Self-timed circuits have solved these problems since they do not require a global clock to be distributed and thus eliminating clock skew problems and simplifying global chip routing. Moreover, self-timed circuits are lower in power consumption, higher in terms of performance, and greater in modularity than synchronous ones. However, self-timed circuits are more difficult to design than synchronous ones and they require new supporting CAD tools since the existing CAD tools can not completely fulfil the requirements of the self-timed circuits. Self-timed circuits replace the clock with a request-acknowledge protocol between modules of the circuit to control data transfer. A completion signal is

generated by the completion detection circuitry in the self-timed system to signal the completion of operation.

The self-timed approach has been utilized for implementing various self-timed arithmetic logic units (ALUs) for various self-timed processors [1, 2, 12, 29]. The performance of self-timed ALUs is influenced by the adder speed. Thus, designing an efficient self-timed adder is an important issue. Different techniques for designing self-timed adders have been reported in the literature [3, 4, 13, 14, 16, 28]. These techniques typically use completion detection logic, e.g. bundled data convention, and speculative completion. These techniques differ from each other in the way they generate the completion signal. The completion detection circuitry causes both area and performance overhead. However, a proper implementation of the self-timed circuitry may reduce these disadvantages and may enhance the performance of the adder and hence the performance of the self-timed ALU. Thus, the aim of this thesis is to design a self-timed ALU which is based on a self-timed adder using Manchester carry chain technique.

The rest of this thesis is organized as follows. Chapter 2 provides a general overview of methodologies used for designing asynchronous circuits. Chapter 3 reviews classic approaches for the design of synchronous adders as well as several self-timed (asynchronous) adder designs reported in the literature. The basic

functions of an ALU and possible implementations of various self-timed ALUs found in the literature are introduced in Chapter 4. Chapter 5 introduces the design of the proposed self-timed ALU. Simulation results of the proposed self-timed ALU are given in Chapter 6. Several speed up techniques are discussed in Chapter 7. Finally, the conclusion and future work are presented in Chapter 8.

CHAPTER 2

ASYNCHRONOUS DESIGN METHODOLOGIES

Most of today's digital systems are synchronous which assume that all signals are binary and that time is discrete. Binary signals are made to describe logic constructs by simple Boolean logic [24]. Assuming discrete time eliminates such problems as signal hazards and feedback.

Asynchronous designs, however, remove the assumption that time is discrete but keep the assumption that signals are binary. This offers potential advantages for asynchronous systems over the synchronous ones. These benefits include:

1. Reduced power dissipation, since only the portion of the circuit involved in the computation is active. Unlike asynchronous systems, in synchronous circuits the whole unit is activated by global clock [24].

2. Higher performance, the performance of synchronous system is bounded by the worst case delay in the slowest processing block where all components must wait until all computations are completed before latching the result. However, asynchronous systems allow average-case performance due to their ability to sense the completion of the computation [24].

3. No clock skew, the difference in arrival times of the clock signal at different components of the circuit is defined as clock skew. More care and effort must be paid in synchronous system to carefully distribute the global clock. By contrast, in asynchronous systems, there is no global clock and hence no need to worry about clock skew [24].

4. Ease of design, since there is no global time constraints in asynchronous circuits, designing them is easier and the modularity is greater than designing the synchronous ones where the synchronous circuits must be carefully optimized to achieve the highest clock rate [24].

5. Automatic adaptation to the physical properties, the performance of the circuit is affected by changes in temperatures, power supply voltage and the technology used in fabrication. Thus, in synchronous designs, all these factors must be taken into consideration in the worst case to adjust the clock

rate. However, in asynchronous designs, the circuit will run as quickly as the current physical properties allow due to the completion detection [24].

With the above mentioned potential advantages of asynchronous circuits over synchronous ones, asynchronous circuits have several problems as well. First, asynchronous circuits are more difficult to design in an ad hoc fashion than synchronous circuits [24]. By setting the clock rate to a long enough period, a designer of a synchronous system does not need to worry about the removal of hazards and the dynamic state of the circuit. However, a designer of asynchronous system must pay a great deal of attention to the dynamic state of the circuit and to the removal of hazards from the circuit to avoid incorrect results. Secondly, asynchronous circuits can not get benefits from existing CAD tools. Thus, it is too difficult to handle the ordering of operations of a complex asynchronous system. Finally, it is not clear that asynchronous circuits are actually any faster in practice even though most of the advantages of asynchronous circuits are towards higher performance.

Asynchronous design is a rich area of research with many different approaches to circuit synthesis. Several general classes of asynchronous circuits including bounded delay, micropipeline, delay insensitive, speed independent and quasi-delay insensitive circuits are discussed.

2.1 Bounded Delay Models

In bounded delay models, the delay of circuit elements and wires is assumed to be known or at least bounded. These models are also used for synchronous circuits. Bounded delay models include the fundamental-mode Huffman model, the non-fundamental-mode, and the burst-mode models.

2.1.1 Fundamental Mode Huffman Circuits

The fundamental mode model, also called the Huffman model, assumes that gate and wire delays are bounded or that a delay upper bound is known. The bounded-delay assumption enables the environment to control the timing of inputs so that every input change occurs only when the circuit is stable. In this model a flow table is used to express the designed circuit. There is a row for each internal state and a column for each input combination. Entries in the table represent the next state and the corresponding output. Figure 2.1 shows an example of a flow table and the corresponding state machine [24].

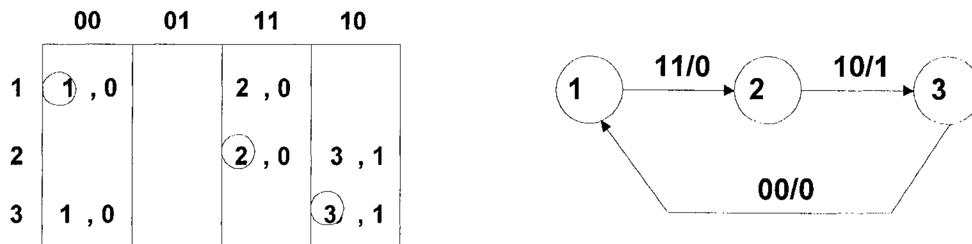


Figure 2.1: A Flow Table and the Corresponding State Machine.

In Figure 2.1, states in circles represent stable states. Since there is no clock to synchronize input arrivals, the system has to respond properly in the intermediate states due to multiple input changes. This system, for example, will not accept input "11" when its present input is "00". Instead, it will move from input "00" to either inputs "10" or "01" then it can accept input "11" and change its state accordingly. Thus, the corresponding empty entries for inputs "01" and "10" when the present state of the system is 1 must be filled with state 1 which will keep the system in state 1.

Using the fundamental-mode model to design asynchronous circuits might introduce dynamic or static hazards [24] due to single input change. Hazards cause momentary glitches on the output nodes. Thus, hazards must be eliminated for proper operation. Removal of hazards can be achieved by adding to the sum-of-products circuits additional cubes which will cover all adjacent 1's in the karnaugh

map. Consider the circuit shown in Figure 2.2 which is implemented using the sum-of-products form under the assumption that all gates have a gate delay of 1 unit. Assume that the current state of this circuit is (A, B, C) = (1, 1, 1) and the output is 1. If B is changed to 0 then the input state of the circuit will be (A, B, C) = (1, 0, 1) and the output should remain 1. But since the upper AND gate becomes false before the lower AND gate turns true, a 0 will propagate to the output causing a static-1 hazard [24]. Adding the cube AC will remove the static-1 hazard shown above.

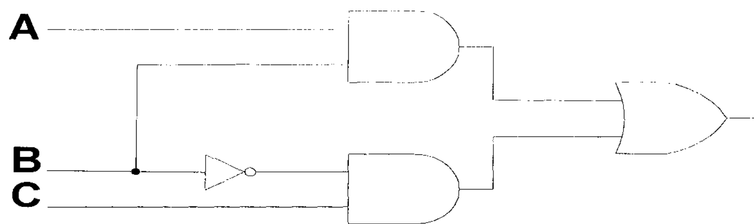


Figure 2.2: Implementation of a Circuit with Hazards.

Hazards introduced due to multiple input changes, however, can not be removed and the solution for that is to adopt a policy decision that only changes one input at a time.

Figure 2.3 shows the sequential Huffman mode model. Delay elements are placed on feedback lines to make sure that the combinational logic has settled in response to a new input before the present state entries change. Another requirement is that only one next state bit can change at a time. Hence, special schemes have to be applied for encoding the state bits, e.g. the one-hot state assignment [24].

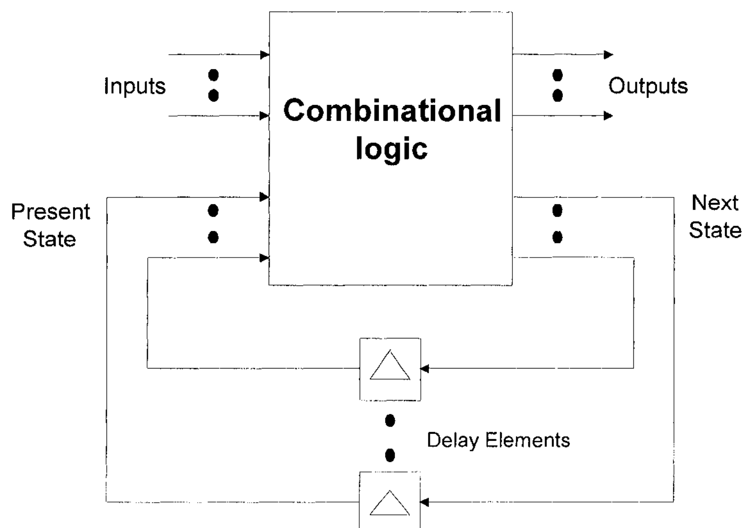


Figure 2.3: Huffman Sequential Circuit Model.

2.1.2 Non-Fundamental Mode Model

The fundamental mode model allows only one input to change at a time and no change will take place unless the circuit is stable. Such requirements can be relaxed and inputs do not have to follow the fundamental mode requirements which lead to non-fundamental mode operation. The one-hot implementation used for the fundamental mode operation can be extended to the non-fundamental mode and the sequential state machine will be implemented using this scheme.

In the non-fundamental mode, the one-hot implementation is viewed as having a simple set-reset flip-flop for each state which is set during a transition into the state and in turn resets its predecessor state flip-flop. Two cross-connected NAND gates form the set-reset flip-flop and the third NAND gate forms the transition term which represents the combinational logic needed to set the flip-flop during a transition [25]. This implementation was suggested by Hollaar and hence called Hollaar's implementation. Figure 2.4 shows a state diagram and its one-hot implementation using the method of Hollaar. This state machine has three states J, K, and L and three transition inputs R, S, and T.

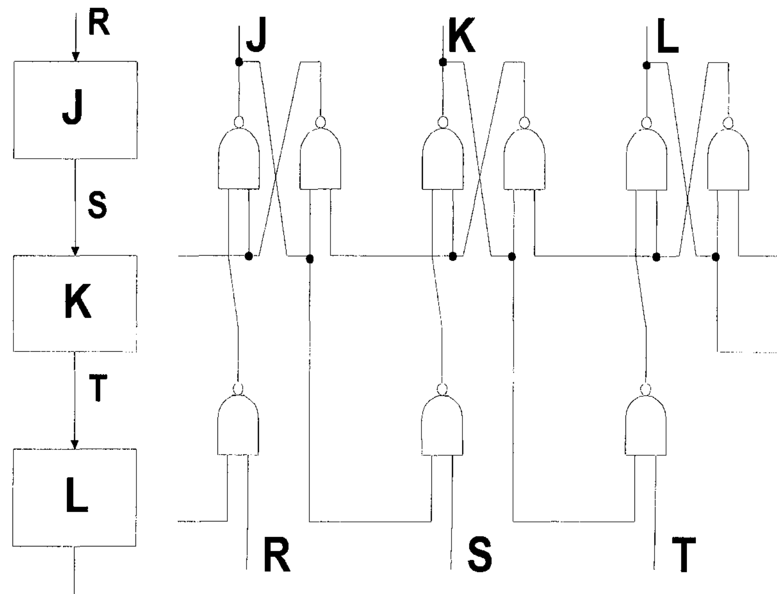


Figure 2.4: Hollaar's Implementation of the State Diagram.

Since the non-fundamental mode relaxes the requirements of the fundamental mode, three different classes of inputs are allowed by the non-fundamental model. The first class consists of inputs where changes in their values do not cause a transition from the particular state because the transition NAND gates to which these inputs are connected are disabled. The second class consists of inputs that cause a transition from a particular state where that state has only one possible successor state. Finally, the third class represents inputs that cause a transition from a particular state where that state has more than one successor [25].

2.1.3 Burst-mode Model

This design style is closer to synchronous one than the Huffman mode. Circuits are represented as a set of standard state machines, one state machine for each circuit component. For each state in the state machine there are a set of inputs (input burst) and a set of outputs (output burst). The circuit moves from a given state to another one if all input burst for that given state occurs and the specified output burst is generated accordingly. Then, the new input burst is allowed after the system has completely reacted to the previous input burst [22]. The fundamental mode assumption is also applicable for the burst-mode system but only between transitions in different input bursts. Another condition needed in the burst-mode design style is that no input burst is a subset of another input burst leaving the same state so the machine can unambiguously determine the occurrence of a complete input burst and react accordingly.

Burst-mode circuits can be implemented using the locally clocked scheme [24] or the two-level AND-OR network. Figure 2.5 shows the locally clocked implementation where a local clock is generated for each state machine independent of other state machines. The purpose of this local clock is to avoid some of the hazards found in the Huffman-mode design style. When the system is

in a stable state, the local clock gets low to allow new input burst. The new input burst passes through phase-1 dynamic latches and the corresponding outputs are generated. If the state changes as it responds to the input burst, the local clock gets high disabling the phase-1 dynamic latches and the phase-2 static latches will be enabled to allow the new state to flow back through the feedback lines and the system stabilises in a new state. By this time the local clock gets low again to allow a new input burst.

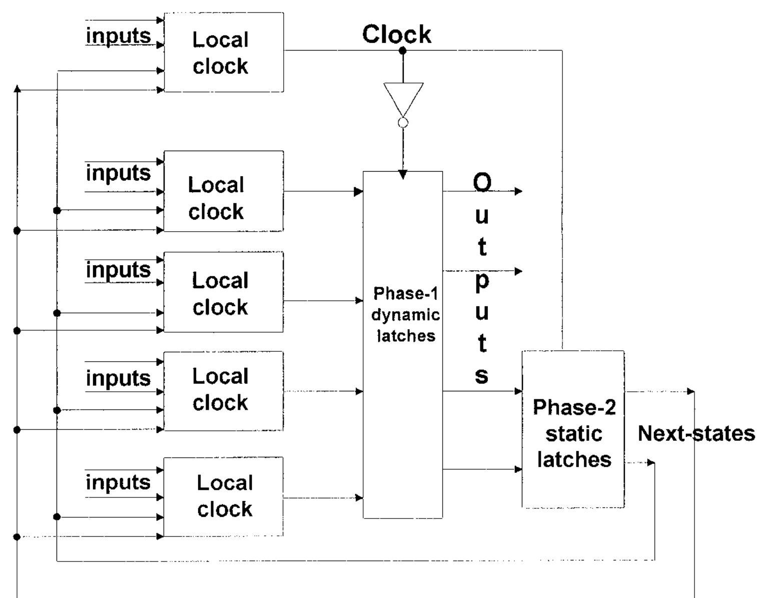


Figure 2.5: Locally Clocked Implementation.

In the two-level AND-OR network implementation, outputs and state variables are fed back as inputs to the network [22]. No explicit storage elements such as

latches are used. Static feedback is used to maintain memory. The operation of this implementation can be described as a cycle consisting of three phases. During the idle state, the machine waits for an input burst to occur. When the last input transition of the input burst arrives, the output burst takes place. State change, if required, immediately follows the output burst completing the three phase machine cycle. Figure 2.6 shows the structure used in this technique.

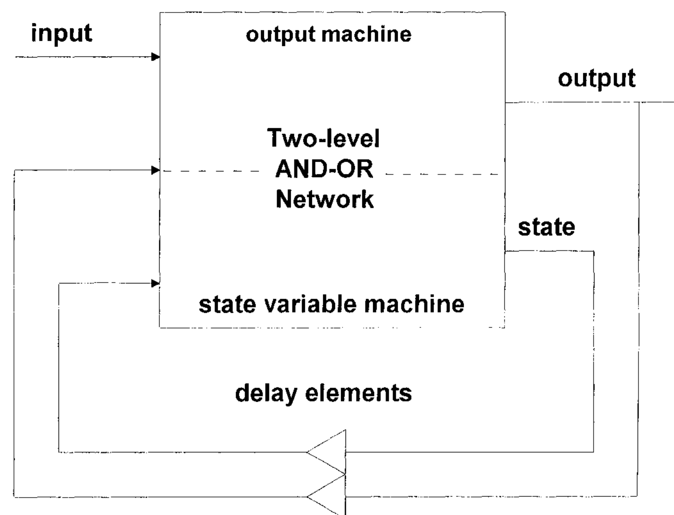


Figure 2.6: Two-level AND-OR Network Implementation.

2.2 Micropipelines

Pipelines can be classified as either clocked or event-driven. Clock-driven pipelines are activated by a global external clock whereas event-driven ones are activated by local signal events. If the input rate matches the output rate of a given pipeline such pipeline is known as an inelastic pipeline, otherwise, it is an elastic one where the rate of data may vary. In its simplest form, a pipeline takes the form of First-in-First-out (FIFO) [17]. An event-driven elastic pipeline is called micropipeline. The control part of the micropipeline is based on an event-driven scheme which relies on a form of control which uses transition (2-phase) signalling rather than the typical clocked (4-phase) signalling scheme.

In transition signalling, the rising and the falling edge of a signal have the same meaning and either one of them represents an event as shown in Figure 2.7.

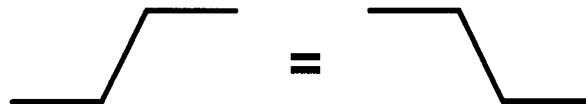


Figure 2.7: Two equivalent transitions.

The absolute high or low state of the control signal using this scheme is unimportant. Instead, the state of a control signal is considered relative to other related control signals. Since there is no need to return a control signal to some neutral state between events (as in clocked-logic), transition signalling is claimed to be better in terms of time and energy costs.

The micropipeline adopts the two-phase bundled data convention to control the data path between the sender and the receiver. There are two control wires and many data wires between the two entities. The two control wires, called request and acknowledge, use the transition signalling scheme while data wires carry boolean values. A bundled data interface between the sender and the receiver is shown in Figure 2.8.

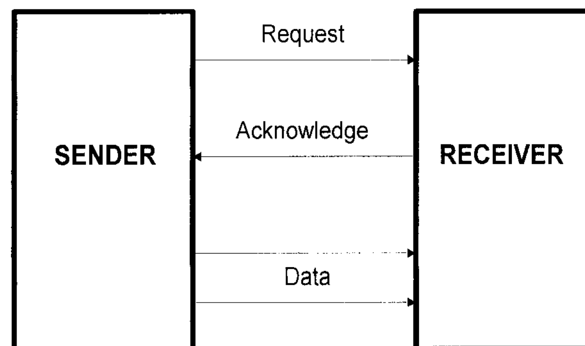


Figure 2.8: A Bundled Data Interface.

Using the two-phase bundled data protocol, there are three events per cycle that occur in the following sequence. First, during the sender's active phase, the sender puts the correct data values. Then, the sender will end its active phase by producing the request event and the sender at this time must hold the data unchanged since the receiver is in its active phase. Finally, when the receiver gets that data and is no longer in need for it, it will end its active phase by producing the acknowledge event. The two-phase bundled data convention is illustrated in Figure 2.9 [17].

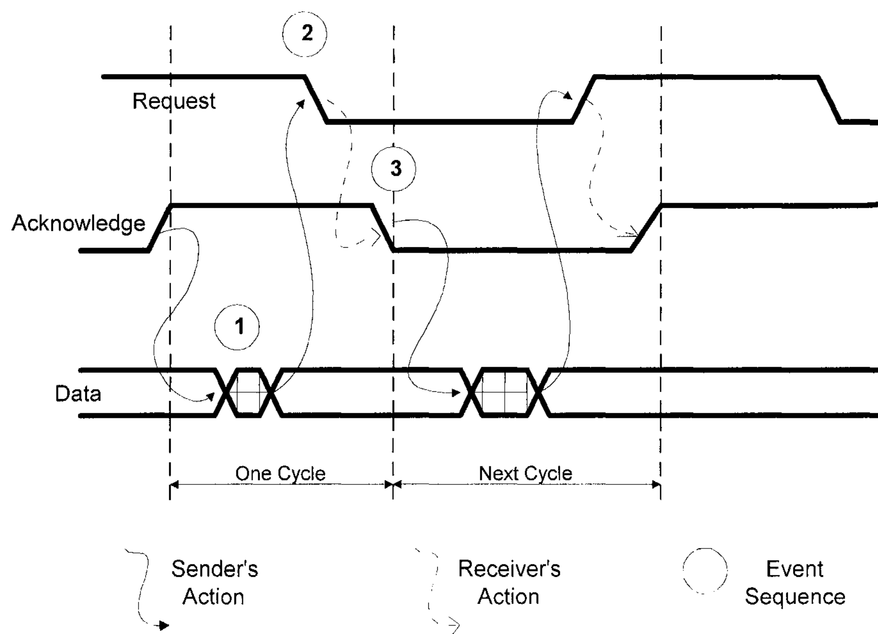


Figure 2.9: The Two-Phase Bundled Data Convention.

The data path and the control path of the micropipeline are built out of event logic modules that form different logical combinations of events. These modules include the exclusive or (XOR) or the MERGE element, the Muller C-element, the SELECT element, the TOGGLE element, and the event controlled storage element. Description of these modules is found in [17]. Since the Muller C-element is widely used in implementing micropipelines, it will be described here. The Muller C-element acts as the AND element for events. When both inputs of a Muller C-element are in the same logical state, the Muller C-element output will change to that same state. When the two inputs differ, the Muller C-element maintains its previous state.

The micropipeline consists of several adjacent stages. At the interface between stages, the request and the acknowledge signals and the data lines follow the two-phase bundled data convention shown in Figure 2.9.

The implementation structure of the control path for a micropipeline takes the form of a FIFO. A string of Muller C-elements connected in series forms the control circuit for the micropipeline. The transitions of the control signals sent by the sender are buffered in this FIFO through the acknowledge, A(in), and the request, R(in), signals. These transitions are then released by the receiver side through transitions given on A(out) and leave the FIFO through R(out). The data

path of the micropipeline can be used either to pass data only between stages, there are no logic computation block between stages, or to perform computation on data otherwise. Between the stages of the micropipeline, there are event controlled storage elements (REG) to hold the data between the stages. Each REG has two inputs, Pass (P) and Capture (C), and two outputs, Capture delayed (Cd) and Pass delayed (Pd), where Pd and Cd are the delayed versions for P and C respectively. A transition on P will pass the data to next stage and this transition will cause an equivalent transition after little delay on Pd. Moreover, a transition on C does not allow data to pass and keeps the current value of the outputs unchanged. Similarly, C is passed to Cd after some delay. The delay elements are added to the control path to match the worst-case computation times [17]. Figure 2.10 shows how the control and data path for the micropipeline are implemented.

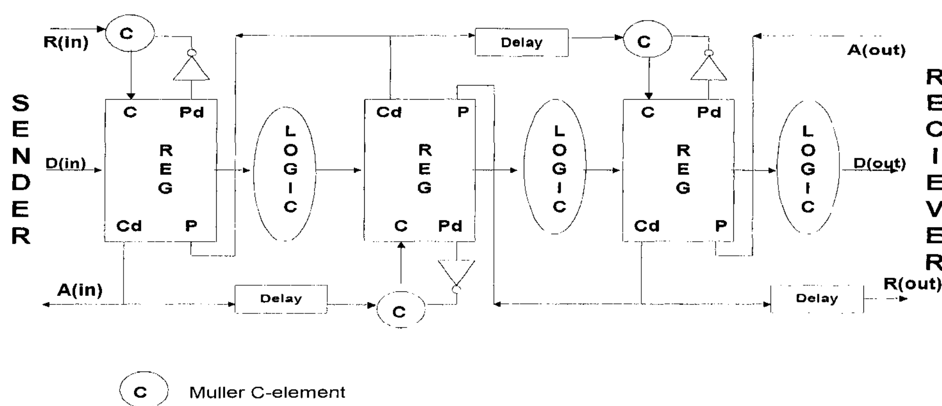


Figure 2.10: Control and Data Path for a Micropipeline.

2.3 Delay-insensitive Model

In delay-insensitive circuits, the delay model assumes that delays of both elements and wires are unbounded [24]. With a delay-insensitive model, there is no guarantee that the input will be properly received no matter how long a circuit waits. Therefore, the sender is required to wait for the completion signal produced by completion detection circuitry in the receiver before sending the next data item. Since there is no guarantee that a value of a wire will reach its proper value at any specific time, according to the delay-insensitive assumption, transition signalling scheme either two-phase or four-phase handshaking is used to pass control information. Thus, for passing control information, a request transition is sent from the sender to the receiver, and a response transition is sent back by the completion detection logic. This forms a two-phase handshaking. Other methodologies extend this to a four-phase handshaking by having a second set of request and response transitions sent in order to return the connecting wires to their original values. Likewise, to pass a bit of data using transition signalling, two wires are required from sender to receiver. For example, a transition on wire I0 indicates that the data bit (B) is a 0 and a transition on I1 indicates that the data bit (B) is 1. An additional wire is also required to send acknowledge (Ack) back to the sender as shown in Figure 2.11.

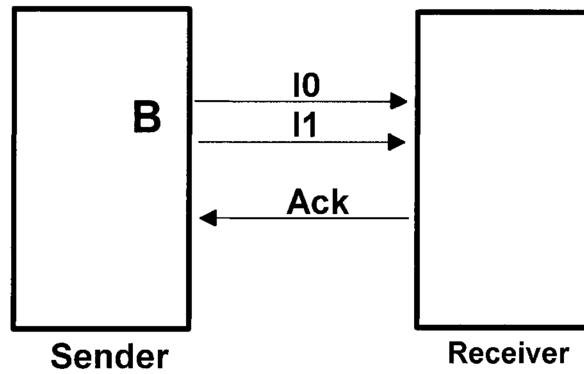


Figure 2.11: Data Transfer via Transition Signalling.

Delay-insensitive circuits can be implemented in several ways. These include delay-insensitive circuits with single-output gates, module synthesis via I-Nets, module-based compilation systems, and trace-based circuits [23, 24]. Using single-output gates such as AND, OR, and XOR for implementing delay-insensitive circuits is not reliable because hazards may be introduced. The single-output gate that can be used in delay-insensitive circuits is Muller C-element. Single-output gates methodology is not suitable for general circuit design since it allows only a very limited class of circuits to be built. The other methods are thus more commonly used since they are more practical for general computation. Finally, delay-insensitive methodologies have some benefits over the bounded-delay methodologies because they force the designers to use conventions important to good asynchronous designs such as completion signals and transition signalling.

2.4 Speed-independent Model

Speed-independent model [1], also known as the Muller model, assumes that gate delays are unbounded while there is no delay in wires (wire delays are negligible). The speed-independent model allows more implementation alternatives than delay-insensitive models but its delay assumptions are not practical to be realised in practice. The speed-independent model is identical to isochronic forks. An isochronic fork is defined as forking wires where the difference in delay between various destinations is negligible. Figure 2.12 shows how a general 2-output isochronic fork is represented in an equivalent speed-independent form.

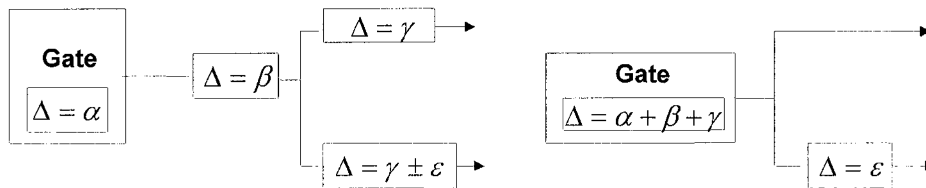


Figure 2.12: An Isochronic Fork (left) and an Equivalent Speed-independent Circuit (right).

Although speed-independent wire delay assumption might be valid in some cases, it is generally not suitable in most cases where interconnect delays are significant, e.g. field-programmable gate arrays where wire delays can dominate logic delays [24].

2.5 Quasi-delay-insensitive Model

Quasi-delay-insensitive model adopts the assumptions of delay-insensitive model that both wire and gate delays are unbounded but with the isochronic fork assumption incorporated [1]. As defined earlier, an isochronic fork is a forking interconnection in which all branches have the same wire delay. In delay-insensitive single-output gates, all ends of a fork must be sensed in order to avoid hazards. However, if the fork is isochronic, only one end of the fork can be sensed to avoid hazards. Figure 2.13 shows examples of quasi-delay-insensitive circuits that are not delay-insensitive. For the circuit of Figure 2.13(a), for example, the AND gate is guaranteed not to fire before the arrival of the two transitions on the AND gate inputs. For the circuit of Figure 2.13(b), the AND gate will never fire since one of the inputs will always be 0 and hence safely ignoring input transitions. Thus, the isochronic fork assumption prevents hazards from occurring in both cases.

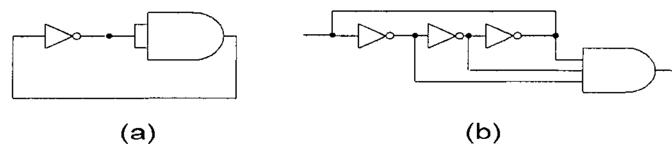


Figure 2.13: Examples of Quasi-Delay-Insensitive Circuits that are not Delay-Insensitive.

Although it is easy to handle the isochronic fork assumption, it is difficult to implement it due to difference in gate construction, variation in switching thresholds, and the different wire lengths and delays which may violate the isochronic constraint. Furthermore, isochronic forks are restricted to small localised areas to avoid the chip boundary problems since matching the delays between on-chip and off-chip destinations is impossible [24].

CHAPTER 3

FIXED-POINT ADDERS

Addition is at the heart of computer arithmetic. Accordingly, various implementations and addition techniques have been extensively studied to improve the performance of such important components. The adder is the major component in an ALU. It is not used only for addition, but it is also commonly used for subtraction, multiplication, division, address computation, and logical operations. Addition techniques can be classified as either variable-time (asynchronous) that use self-timed components as building blocks or fixed-time (synchronous). The former produces a completion signal when the result is valid whereas the latter produces valid results after a fixed delay time which is chosen to be the worst case delay time. In this section, a review for classic CMOS synchronous adders is presented. Then, reported techniques for implementing asynchronous adders will be discussed.

3.1 Synchronous Adders

The addition of two operands forms the basis of all arithmetic operations in any digital data-path. Hence, by improving the performance of the addition mechanism, direct improvements can be achieved in all arithmetic operations. Synchronous adders differ from one another in the way they handle carries. Classic implementations of synchronous adders include serial, carry ripple, carry skip, carry look-ahead, manchester carry chain, carry select, and conditional sum. These adders have been well studied and reported in the literature and several variants of them have also been reported. Speed is not the only factor in comparing adder implementations, other important factors include power consumption and layout area. A detailed comparison of these adders in terms of power consumption, speed, and layout area can be found in [26]. A review of the above mentioned adders is given in this section.

3.1.1 Serial Adder

The serial adder is the simplest, the slowest, and the most area-efficient one among all adders. It performs the addition of two n-bit operands A and B serially, that is adding a pair of bits per clock cycle. At each cycle, the full adder has three inputs

and two outputs. The three inputs are A_i, B_i, and carry-in (C_{i-1}) which is the carry-out bit resulting from adding A_{i-1} and B_{i-1}. The two outputs are the sum bit S_i and the carry-out bit (C_i). The full adder can be implemented by the following two equations:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \quad \text{and,}$$

$$C_i = A_iB_i + (A_i \oplus B_i) C_{i-1}$$

Figure 3.1 shows the implementation of the basic full adder unit used in building the serial adder. The serial adder consists of three shift register, one full adder unit, and a delay element (D_flip flop). The three shift register are to hold the two operands A and B, and the sum bits. One pair of bits is shifted out from the two operand registers A and B each clock cycle. Then the full adder unit performs the addition and the resulting sum bit is shifted in to the sum register. The carry-out bit is fed back through the delay element to be used in the next cycle. Figure 3.2 shows the complete addition unit for the serial adder. Serial adder is preferred if the issue is cost rather than performance. It is the slowest adder while its area cost is the least. The serial adder performs addition in O(n) time where n is the number of operand bits [27].

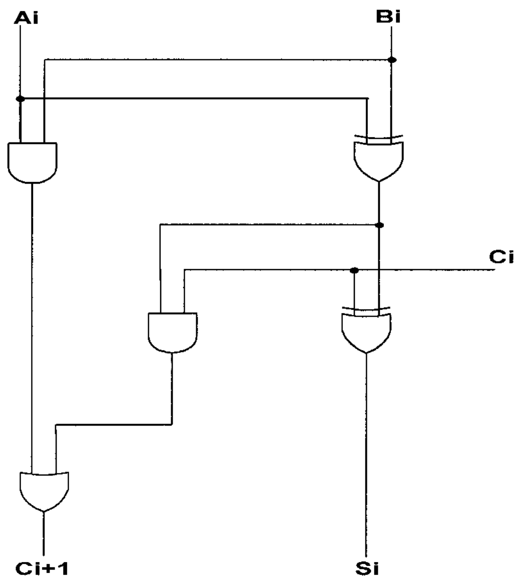


Figure 3.1: A Full Adder Unit.

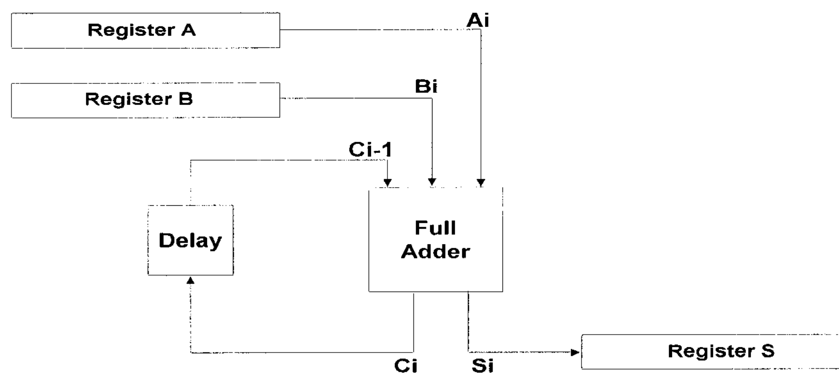


Figure 3.2: A Serial Adder.

3.1.2 Carry-Ripple Adder

An n-bit carry ripple adder is constructed from several full adder units by cascading them where the output carry from stage *i-1* feeds the carry-in input of stage *i* as shown in Figure 3.3. In carry ripple adders, the carries ripple (propagate) through the full adder units from the least significant to the most-significant stages. In terms of speed, the carry ripple adder is slightly better than the serial adder since all operand bits are fed in parallel into the adder. Thus, there is no need for delay elements for the intermediate carries. The length of the path through which the carry ripples determines the performance of the carry ripple adder. If the carry ripples through all stages, this leads to the worst case delay time [27]. However, it was proven that the average path of a carry to propagate through is much shorter than n stages. This is due to the fact that several carries are generated when the two operands are added. These carries will propagate different distances along the carry path. Studies have shown that the average ripple distance is of order log(n) [13, 27].

Although carry ripple adders are slow (O(n) time for n-bit operands) when compared to other implementations, they have a very simple and regular structure which makes them attractive for a VLSI implementation.

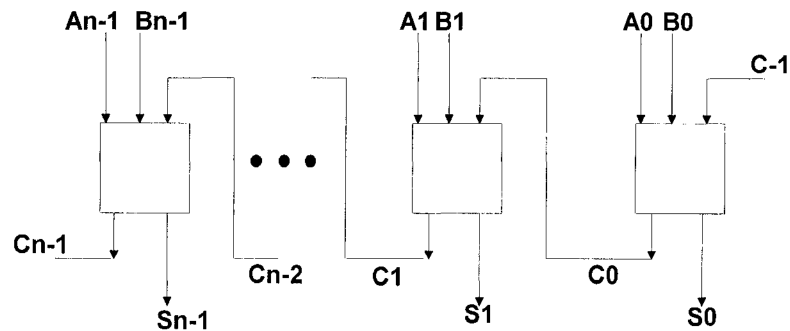


Figure 3.3: Carry Ripple Adder.

3.1.3 Carry Skip Adder

Carry skip adder shows improvement of carry propagation time as compared to carry-ripple adder. It is obtained from the carry-ripple adder by grouping several stages into blocks. These blocks need not be of the same size. Each block is equipped with carry skip logic for this particular block. A carry which will propagate through stages of the current block to finally enter the next block can actually skip (or bypass) stages of the current block. This can simply be achieved by the carry skip logic which produces a copy of that carry and passes it to the next block while the normal operation of carry propagation proceeds through stages of the block to produce the sum bits. Hence, the carry passed to the next block is either

generated by the carry skip logic of the block or produced by the block itself. Thus, this is where the carry skip adder speeds up the carry propagation over the carry-ripple adder [26, 27].

Block *i* of size m bypasses the carry to block *i+1* if and only if the following condition is satisfied:

$$(A_i + B_i)(A_{i+1} + B_{i+1}) \ldots (A_{i+m-1} + B_{i+m-1}) \Leftrightarrow (T_i)(T_{i+1}) \ldots (T_{i+m-1}) = 1,$$

where T_i is called the carry transfer or the propagate function of block i. The carry that will enter block i+1 is either the carry C_{i-1} that skips block i or is the carry C_{i+m-1} which is produced by block i. Thus, the equation for the block carry-out that implements the carry skip logic for block i is as follows:

$$\text{Block carry-out} = (T_i)(T_{i+1}) \ldots\ldots T_{i+m-1} C_{i-1} + C_{i+m-1}\,.$$

Figure 3.4 shows the carry skip adder with one level skip logic [27]. Further improvement can be obtained from the one-level carry skip adder by adding another level of carry skip logic. This extra carry skip logic groups carry skip blocks of the first level together and hence is called super-block carry skip logic where a carry can skip several blocks. Such adder, two-level carry skip adder, is faster than the

one-level carry skip adder and can achieve speeds comparable to traditional carry look-ahead adder [27].

Defining block size in carry skip adders plays an important role in determining the speed of these adders. In single-level carry skip adders, blocks at both ends of the adder are chosen to be of small sizes (few bits per block) while those in the middle are chosen to have larger sizes. Likewise, for two-level carry skip adders, the second level groups at the ends consist of small number of blocks while the number of blocks per group is larger near the middle [6]. Defining optimal block size has been studied and different algorithms to find near-optimal solutions for block sizes have been reported [7, 19, 20, 21]. In general, carry skip adders have $O(\sqrt{n})$ addition time where n is the width of the adder [27].

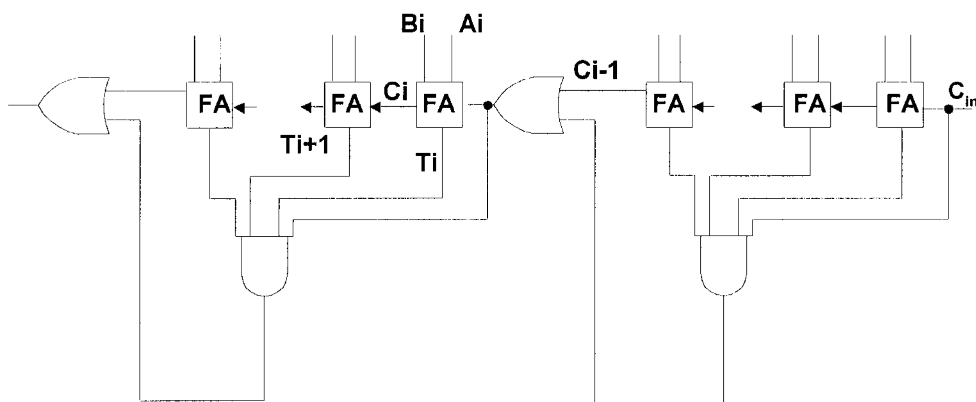


Figure 3.4: Carry Skip Adder with one level skip logic.

3.1.4 Manchester Carry Adder

The elemental circuit used to construct the carry chain in the Manchester carry adder is shown in Figure 3.5. Signal P_i is the carry propagate signal and signal G_i is the carry generate signal which are defined as follows:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

where A_i and B_i are the ith bits of the A and B operands.

The operation of the elemental circuit, shown Figure 3.5, proceeds as follows. When clock signal *(clk)* is low, the output node is precharged by the PMOS pull-up transistor. When *clk* goes high, the NMOS pull-down transistor turns on. If carry generate signal (G_i) is true, then the output node discharges. If carry propagate signal (P_i) is true, then a previous carry may be coupled to the output node, conditionally discharging it. The circuit in Figure 3.6 shows how the circuit in Figure 3.5 can be cascaded to form a 4-bit Manchester carry chain for the 4-bit Manchester carry adder block.

In Figure 3.6, if all propagate signals (P_1-P_4) are true, and carry-in signal (Cin) is high, six series NMOS transistors pull the output node low. This worst-case propagation time can be improved by bypassing the four stages if all carry propagate signals are true [6]. The additional circuitry needed to achieve this is shown in Figure 3.7. This additional circuitry consists of a dynamic AND gate, which turns on a carry bypass signal if all carry propagates are true. The circuit shown in Figure 3.7 is called the Manchester carry chain with carry skip. The Manchester carry adder with carry skip is better in performance than the regular Manchester carry adder due to the fact that carry skip through the bypass transistor is linearly proportional to the size of the block whereas the propagation delay through the serially connected NMOS transistors is proportional to the square of the size of the block [6]. As carries are generated or propagated, the sum bits are also produced. The delay of the Manchester carry adder is O(n) time where n is the width of the adder.

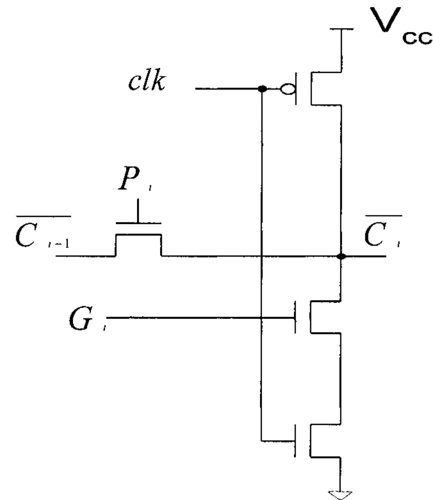


Figure 3.5: The elemental circuit for Manchester Carry Chain.

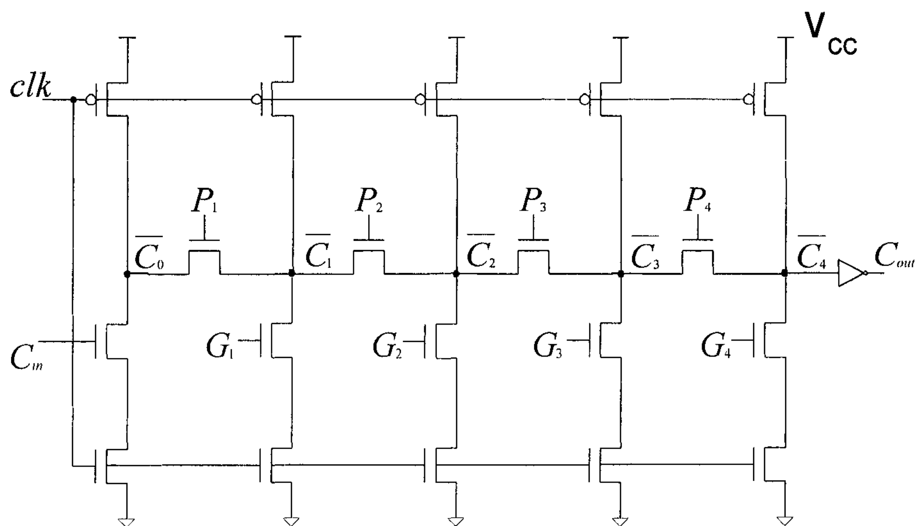


Figure 3.6: A 4-bit Manchester Carry Chain.

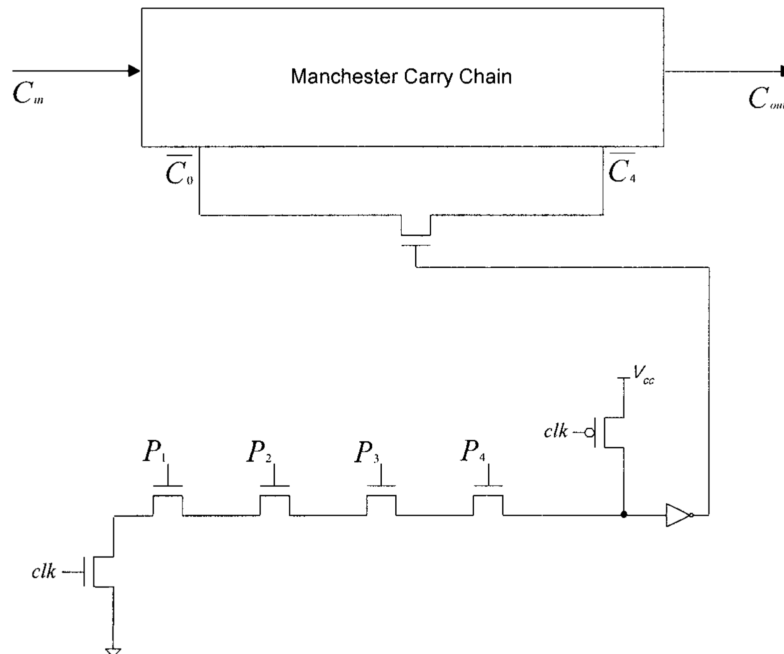


Figure 3.7: A 4-bit Manchester Carry Chain with carry skip.

3.1.5 Carry Look-ahead Adder

It is possible to determine all carries beforehand and thus eliminate the delay that would otherwise be incurred in propagating them by examining bits of both operands. The design of Carry Look-Ahead adders (CLAs) is based on this fact. A pure CLA is theoretically the fastest adder because carries can be generated independently and concurrently before they are needed as shown in the following

equations. Due to the rapid increase in fan-out and fan-in requirements as adder sizes increase, such a pure CLA is not practical for any but the smallest adders.

Let G_i, P_i, and C_i define the generation, propagation, and evaluation of a carry at stage *i* respectively where,

$$\mathbf{G_i = A_iB_i}$$

$$\mathbf{P_i = A_i \oplus B_i}$$

$$\mathbf{C_i = G_i + P_iC_{i-1}}$$

Now, using recursion in the equation for C_i for each stage yields the equations:

$$\mathbf{C_0 = G_0 + P_0C_{-1}}$$

$$\mathbf{C_1 = G_1 + P_1G_0 + P_1P_0C_{-1}}$$

$$\mathbf{C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{-1}}$$

•

•

$$\mathbf{C_i = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + P_iP_{i-1}P_{i-2}\ldots P_0C_{-1}}$$

where C_{-1} is the input carry.

It is clear from the above equations that carries can be generated independently and determined before they are actually needed to compute the sum bits. Thus, it is known as a carry look-ahead or carry prediction adder. The sum bit, S_i, is still defined by the equation:

$$\mathbf{S_i = (A_i \oplus B_i) \oplus C_{i-1}.}$$

For large adders, multilevel structures of gates with limited fan-in may be used as a solution for the fan-in problem and buffers may be used to obtain the required fan-out but this involves additional delay. Therefore, large adders use carry look-ahead in a limited fashion combining it with some other technique that reduces the fan-in and fan-out problems. In the Ripple-block CLA (RCLA) [27], the bit stages of the adder are grouped into blocks organized in such a way that carry look-ahead is used only within blocks and carries between blocks are rippled. A Block CLA (BCLA) reverses the design of the RCLA in a way that carries within blocks are rippled but those between blocks are determined by look-ahead. This design requires that the adder size be sufficiently large and the block size be sufficiently small that the performance gained from the look-ahead is not outweighed by the loss from rippling.

For very large adders, it is common to use more than one level of look-ahead. Any number of levels could be used but in practice there is a delay incurred with each level and consequently for a given adder size there is a point beyond which additional levels do not yield any improvements in performance and may produce a loss. Two levels of look-ahead have been the most common. The superblock BCLA (SBCLA) groups the bit stages of the adder into blocks which in turn are grouped into superblocks. Look-ahead is used to produce carries at both the block and superblock levels but rippling is still required. An improved version of the SBCLA, the ISBCLA, eliminates all rippling at the group level. Similarly, in a Superblock RCLA (SRCLA), the bit stages are grouped into blocks which in turn are grouped into superblocks where rippling occurs only at the superblock level. With proper choice of the block and superblock sizes, the SRCLAs generally yield better performance than the ISBCLAs [27].

Variants of pure CLA are reported in the literature. Flynn [18] describes a fully static CMOS CLA adder which saves one gate delay to reduce the number of serial transistors in the critical path over the pure CLA. Another variant of pure CLA, called binary CLA, was developed by Brent and Kung [5]. Their design depends on the reduction of carry propagation time and optimizes the area rather than the number of gates required as a measure of cost and this what makes this type of

adder well suited for VLSI implementation. Other improved variations of the basic CLA can be found in [10,11].

3.1.6 Carry Select Adder

The classical carry select adder performs two additions. One addition assumes a "0" carry-in and the other assumes a "1" carry-in. Then, the correct sum is selected when the carry is known by the carry determination and sum selection logic. The detailed logic of the sum modules can be built from other addition techniques such as carry-ripple addition, carry-look-ahead addition, etc. Figure 3.8 shows the high level organization of a carry select adder. The delay of the carry select adder is an O($\sqrt{n}$) while its area is O(n) where n is the width of the adder. A new scheme of implementing a carry select adder was proposed by Akhilesh Tyagi [9]. This scheme generates carry bits with block-carry-in 1 from the carries of a block with block-carry-in 0 with a little extra logic. Results show that the new scheme outperforms the classical carry select adder by about 28% in area and 7% in delay.

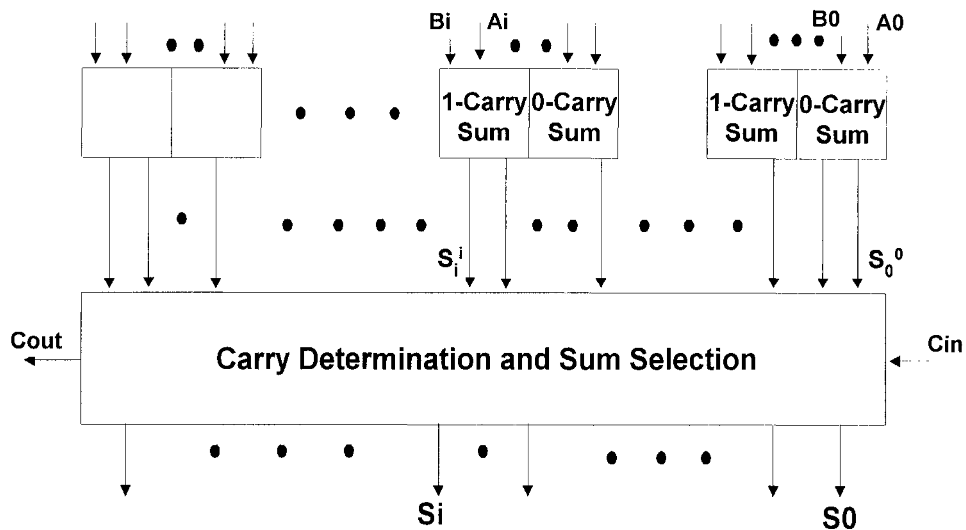


Figure 3.8: General Organization of Carry Select Adder.

3.1.7 Conditional Sum Adder

The conditional sum adder is a special case of the carry select adder. While sum selection is done at the block level in the carry select adder, it is done at the bit level in the conditional sum adder. The conditional sum adder is structured such that all possible carry and sum bits are generated in a single step then the carry bits are used to select the correct sum bits. The selection of sum bits is done through a series of

steps that is carried out on two sets of sum and carry bits. One set is generated under the assumption that there will be no carry in each bit stage while the second set is generated under the assumption that there will be a carry into each bit stage. Then, carry and sum bits for pairs of stages are generated and possible carries between the two stages of a pair are considered. Then, the stages are grouped in four and possible carries between the two pairs in each group are considered. This process continues until the size of the group is equal to the width of the adder where the correct result is available.

Norman M. Martin and Stephen P. Hufnagel reported the Conditional Sum Early Completion Adder (CSCA) [8]. The CSCA design is based on the computation of conditional sums, carries and column completion detection logic. The CSCA is not cost-effective relative to the conditional sum adder for small and moderate word length. However, the CSCA provides substantial time advantages for sufficiently large word length.

3.2 Asynchronous Fixed-point Adders

Designing large and practical asynchronous systems such as microprocessors [1, 2, 15, 29] and digital signal processors [12] is an increasingly important recent trend. The design of efficient data-path support components such as adders is critical to these systems. Asynchronous adders are generally faster than synchronous ones because they indicate when the addition result is available whereas synchronous adders address the worst case delay for carry propagation. A self-timed adder is an asynchronous adder that generates completion signal to flag completion of the addition operation. Self-timed adders enjoy an average delay time for the global sum of O(log(n)) where n is the width of the adder. Hardware overhead and an added delay due to the detection circuitry are the potential disadvantages of self-timed adders. However, a proper implementation of self-timed circuitry may reduce these disadvantages and enhance the performance of the adder.

Several approaches have been proposed for designing self-timed adders [3, 13, 14, 16, 28, 32, 34, 36]. The reported designs can be classified into three main categories depending on how the completion signal is generated. These include bundled data, completion detection, and speculative completion.

3.2.1 Completion Detection

In the completion detection approach, self-timing is explicitly introduced in the design where extra hardware circuitries are added to detect completion of the addition operation. However, the key disadvantage of this approach is the delay overhead introduced by the completion detection network. Therefore, proper implementation of completion detection circuit is essential. Self-timed adders reported in the literature using completion detection approach are mainly implemented in three different manners. These include dual-rail [14, 31, 33], delay matching [13], and differential cascode voltage switch logic (DCVSL) [28, 37].

In the dual-rail method, two wires are used to represent carry signals. One is used to propagate 0-carry while the other is used to propagate 1-carry. A 0-carry should exist when both operand bits are 0's while a 1-carry should exist when both operand bits are 1's. Thus, when either of the two cases is true, a carry / no-carry signal is immediately generated without waiting for the propagating incoming carry and this is where the speed up comes from. The carry ripple adder can be modified to support completion detection by using two carry chains; one to detect the 0-carries and the other one to detect the 1-carries. This type of adder has an average addition time of O(log n) where n is the width of the adder. The logic details of a

single stage of this adder are shown in Figure 3.9. The design of carry ripple adder with carry completion detection was reported in [14]. The design was based on ripple carry adder using only NOR gates. It was based on the indication of the presence of either a 1-carry or a 0-carry from each stage. Signals necessary for carry completion detection were automatically generated by each stage. Then, one NOR gate for each pair of stages was used for carry completion detection plus another NOR gate for the entire adder.

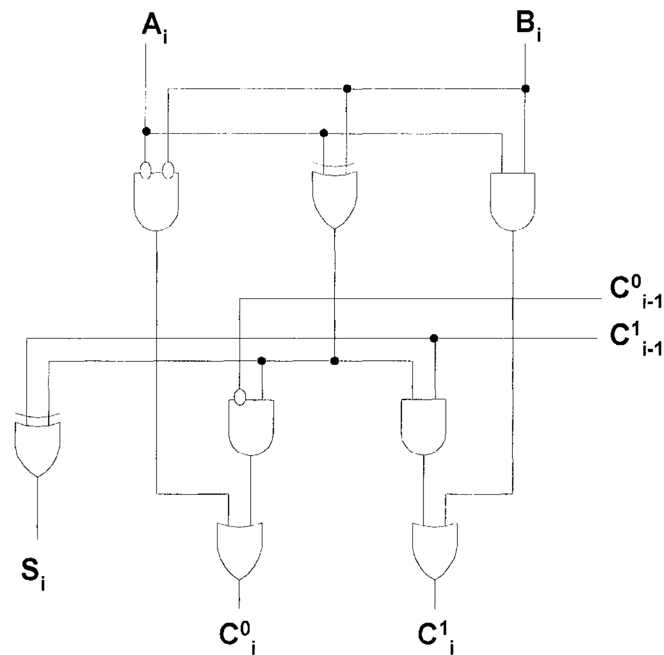


Figure 3.9: Carry Completion Adder Stage.

The second method for implementing completion detection in self-timed adders is achieved by using differential cascode voltage switch logic (DCVSL). Figure 3.10 shows a generalised DCVSL gate. When the signal I is low, both S and $\overline{S}$ are precharged high and the NMOS tree is disabled. Then, when the I signal is raised high, the NMOS tree evaluates and generates the output. During the evaluation phase, either S or $\overline{S}$ is discharged. The completion signal (DV) is generated by the OR gate. DV is low during the pre-charge phase while it turns high when computation is finished. The DCVSL blocks were used in designing the sum and the carry gates used in the self-timed ALU for a fully asynchronous digital signal processor reported by Jacobs and Brodersen [12].

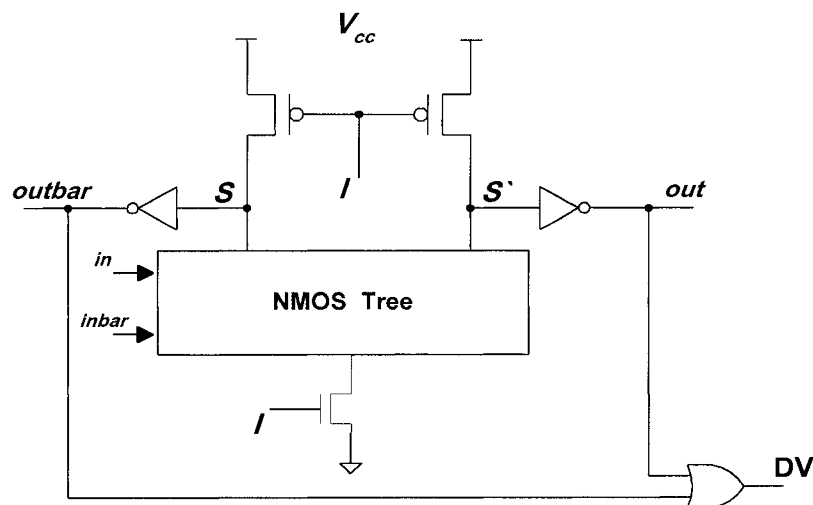


Figure 3.10: A Generalized DCVSL Gate.

Matching delay is the third method that has been reported for implementing self-timed adders. In this case, self-timed operation is achieved by carefully matching the delays of the generation of valid data and that of the generation of completion detection signal. The condition here that has to be maintained for proper operation is that the results of computation have to be valid before the generation of the completion signal. Therefore, this method requires specific knowledge and extensive engineering and simulation to ensure that the designed circuits function correctly over different ranges of process variance and environmental conditions [4]. The self-timed adder reported in [13] and called statistical carry look-ahead adder (SCLA) uses this method to generate the completion signal. In SCLA, each 4-bit block generates a completion signal during the evaluation phase through a NAND gate. This NAND gate is properly implemented to match the delays of carry propagation of different bit stages in the 4-bit block. It has different path lengths where it can be discharged through any one of the paths depending on the position of the carry propagate signal. The output of this NAND gate is low during the evaluation phase while it is high during the pre-charge phase. The final completion detection signal of SCLA is generated by combining the completion signals generated from all the 4-bit blocks during the evaluation phase via a NOR gate. The output of this NOR gate is low during the pre-charge phase while it is high when the evaluation phase is completed which flags the validity of the result.

3.2.2 Speculative Completion

Speculative completion (early completion) method was proposed by Nowick [3] for designing asynchronous adders. Speculative completion approach uses single rail data paths as in the bundled data approach. However, it differs from the bundled data approach in using several different matched delays. Usually, these delay models include a worst-case model delay and one or more speculative (i.e. early-completion) delays. Thus, speculative completion technique allows several possible speeds. In the speculative completion method, unlike existing completion detection methods, early completion detection occurs in parallel with the data path computation.

Figure 3.11 shows a general architecture of the speculative completion method. As shown in the figure, two or more delay models are used. One delay model is used for the worst-case delay while the other delay models (speculative delays) with different speeds are used for early completion. For each speculative delay, an abort detection network is used to discover the conditions for early completion and to abort early completion due to worst-case data.

Nowick [3] utilized this approach and applied it to the binary look-ahead carry adder which was reported by Brent and Kung [5]. The analysis and operation of an

asynchronous binary look-ahead adder designed with speculative completion showed an improvement up to 30% over the synchronous one.

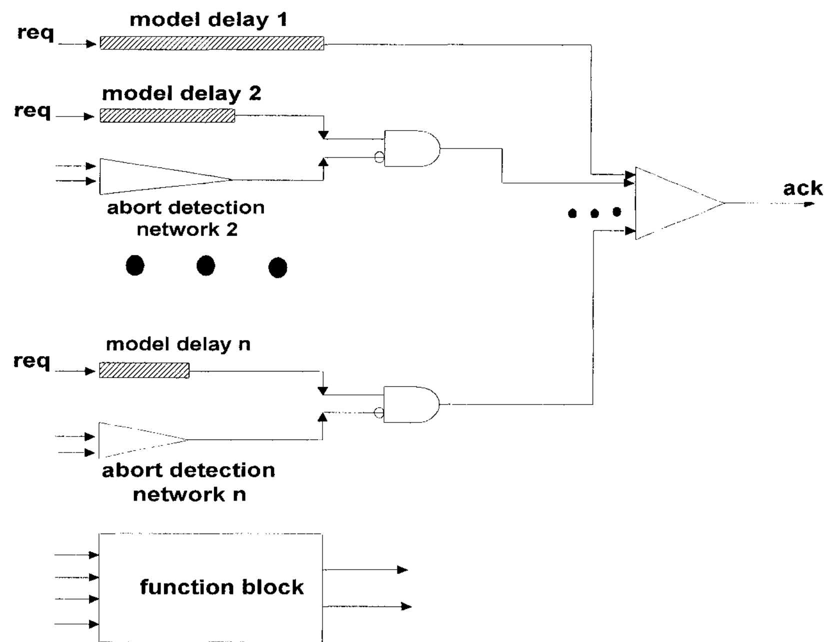


Figure 3.11: General Architecture of Speculative Completion Method.

3.2.3 Bundled Data Based Adder

This class of adders adopts the bundled data convention which is implemented in the form of a micropipeline. The control and data signals propagate together. Control signals are transitions rather than logic values where any switching of states

of a control line indicates the validity of data. The bundling of data guarantees the availability of correct result before the issuing of the control signal.

The adder reported by Ravi R. and Shih-Lien Lu [16] is a self-timed adder, called a Carry Elimination Adder (CEA), which is implemented in the form of a micropipeline that follows the bundled data convention. The CEA does vertical carry propagation instead of the common horizontal propagation through several iterations. The operation of the CEA could be formalized by the following steps. First, the two operands, the addend and augend, are loaded into two registers, SUM and CARRY. Secondly, the contents of these two registers are simultaneously bit-wise ANDed and XORed. The ANDing operation produces the carry vector that needs to be adjusted in the next iteration and the XORing operation produces the sum without any carry accounted for. Thirdly, the ANDed result is left shifted and routed back to the CARRY register and the XORed result is routed back to the SUM register. Finally, the above steps are repeated until such a time that the contents of the CARRY register are zeros which indicates that further carry propagation is unnecessary and the sum is available in the SUM register.

The architecture of the CEA shown in Figure 3.12 consists of two sections, namely the data path unit and the control unit. The control unit consists of SELECT, SYNC, and CLK-GEN blocks. All the pipeline handshaking signals, the

clock signals for the latching of data, and the select signals for the MUXes are produced by the control unit. The zero detector decides the selection of inputs for the two MUXes depending on the content of the CARRY register, wither nonzero or zero. The MUX either re-circulates the intermediate data while the adder is processing the current operands (the content of the CARRY register is nonzero) or selects new data if the adder is ready to receive it (the content of the CARRY register is zero).

To implement one bit-slice of the data path of the CEA, two double-edge-triggered-flip-flops, one XOR gate, and one AND gate are required. Since each bit of the data path is modular, scaling the CEA to any number of bits can be easily achieved. A zero detector of width equal to the number of addition bits n is also required in the data path of the CEA. The zero detector is an n-bit NOR gate which is implemented as a NAND-NOR tree.

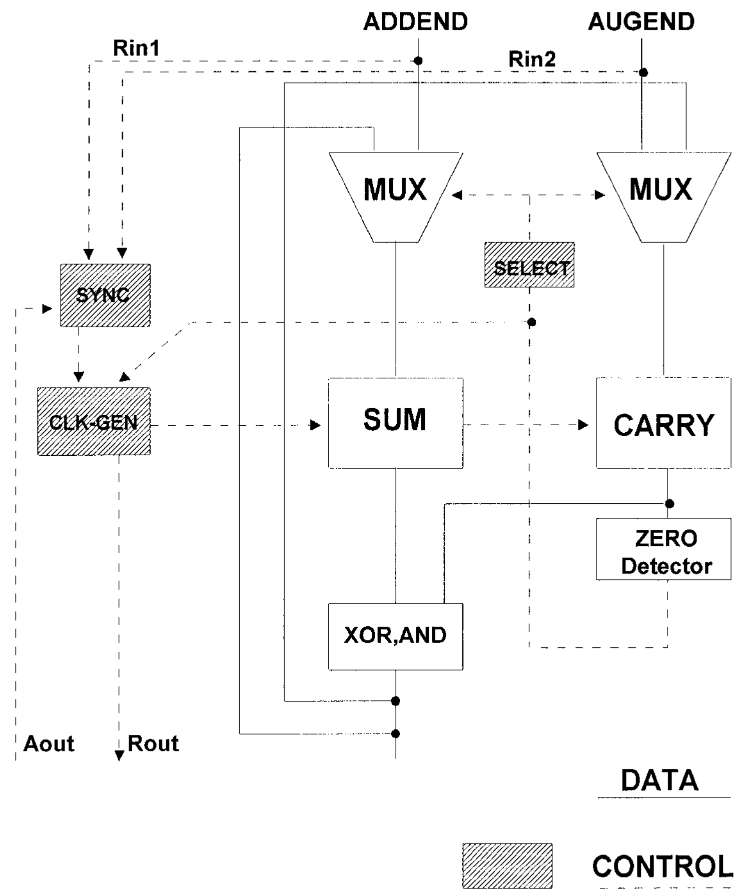


Figure 3.12: Architecture of the CEA.

CHAPTER 4

ARITHMETIC LOGIC UNITS

An arithmetic logic unit (ALU) is a digital circuit that performs a set of arithmetic operations and a set of logic operations. The ALU is found in all processors and generally operates in parallel on words. The ALUs found in the synchronous processors operate synchronously while those found in the asynchronous processors operate asynchronously. Although most of today's processor designs are synchronous, asynchronous or self-timed processors are introduced due to modern device technology with a switching delay of only a few picoseconds. Since no signal can reach further than 0.3 mm in 1ps, the use of such fast switching devices in synchronous system design presents serious timing problems, namely wire delay and clock skew [1]. As a result, asynchronous designs can replace the synchronous ones to fully enjoy the fruits of foreseeable progress in device technology.

In this section, the basic functional units of the ALU will be reviewed, and several reported implementations of asynchronous (self-timed) ALUs will be discussed.

4.1 Basic Functions of an ALU

An ALU, whether synchronous or asynchronous (self-timed), usually performs the following functions:

(a) arithmetic operations which include addition, subtraction, increment, decrement etc.

(b) logical operations which include OR, AND, XOR, etc.

(c) shift operations such as shifting left or right, arithmetic shifts, and rotations.

(d) predicate calculations on the result such as zero, positive, negative, overflow etc.

To perform the above functions [30], an ALU is typically composed of:

(a) a unit that can provide direct values of the operands, complemented values of the operands, 0, and 1.

(b) an adder unit that performs arithmetic operations such as addition, subtraction, increment, and decrement. The re-use of terms generated in the adder unit which are close to the required logical operation, disabling other terms and forcing the carry input to zero or one as needed, can be used for implementing some logical operations.

(c) a carry-in selection circuit which can select the values 0 and 1 in addition to the previously stored carry output.

(d) a circuit for the calculation and storage of predicates defined on the result.

(e) a shift unit for performing the required shift operation.

(f) buffers and output registers.

4.2 Implementation of Asynchronous ALUs

An asynchronous (self-timed) ALU can be built from asynchronous (self-timed) adders. The performance of asynchronous ALU is significantly influenced by the

adder speed. Statistical studies show that 72% of the instructions perform additions in the data path. Furthermore, branch and memory access which require calculations of effective addresses also use adders [28]. Mark A. Franklin and Tienyo Pan [28] studied and compared the effects of different designs of asynchronous adders on the performance of a simple single pipeline asynchronous DLX machine. They found that if the synchronous conditional sum adder is used in a clocked version of the DLX machine, the clock cycle would be set to 10 ns and the resulting throughput would be 100 MIPS. This is slower than its asynchronous counterpart. Many asynchronous processors [1, 12, 29] were reported in the literature. These asynchronous processors were built out of self-timed circuits with appropriate handshake protocols to eliminate the requirement for any global clock.

Gordon M. Jacobs and Robert W. Broderson [12] reported a complete microprocessor-based digital signal processor (DSP). The DSP was designed and fabricated using self-timed circuits following a four-cycle handshake protocol to provide fully asynchronous operation without the need for any global clock. The data path cells were designed using CMOS DCVSL logic family where a completion signal is generated at each stage of the data path cells. In the self-timed ALU of this DSP, the completion time of the carry propagate adder depends on the data being processed. Figure 4.1 shows a single stage of the DCVSL sum and carry

gates used in the self-timed ALU of this DSP where A_i, B_i, C_{in}, I, and DV signals are the ith bit of operand A, the ith bit of operand B, the carry-in bit, the initialization signal, and the completion signal respectively. The DV signals generated from the stages of the adder are combined using a tree of NAND-NOR gates to generate the final completion signal. The overhead in the circuitry required for generating the completion signal for this DSP was small but better design strategies could achieve greater performance.

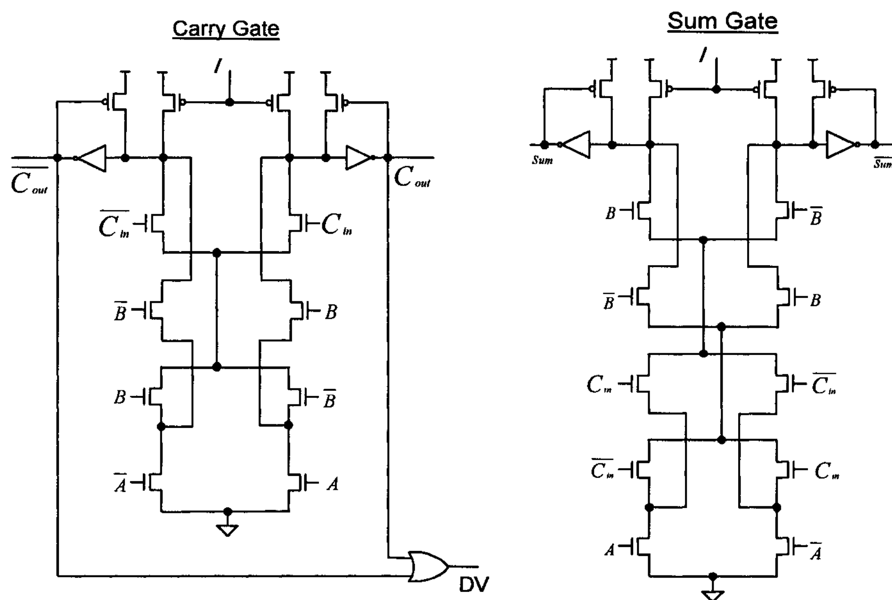


Figure 4.1: DCVSL Sum and Carry gates used in the Self-timed ALU.

Another design of a quasi-delay-insensitive microprocessor, called TITAC, which was reported by Takashi Nanya and et. al. [1] is an 8-bit microprocessor

based on the delay-insensitive model incorporating the isochronic forks assumption. The TITAC is a two-phase, event driven design. The ALU of TITAC was implemented as a two-rail, multilevel AND-OR scheme with a binary decision diagram structure for efficient signal generation. In the two-rail representation, a pair of signal lines (d1, d0) represents one bit information D according to the following convention:

$$\mathbf{D = 0 \Leftrightarrow (d1,d0) = (0,1)}$$

$$\mathbf{D = 1 \Leftrightarrow (d1,d0) = (1,0)}$$

Thus, the TITAC uses N pairs of signal lines to implement an N-bit of data. In the two-rail multilevel AND-OR implementation with the binary decision diagram structure, only the primary inputs and the AND-gate inputs must be ORed at each level, and then the OR-gate outputs are collected by a Muller C element to generate the completion signal. Figure 4.2 shows the two-rail implementation with the completion signal for the three-variable boolean function **F = AB` + A`B + C** in the form of binary decision diagram structure.

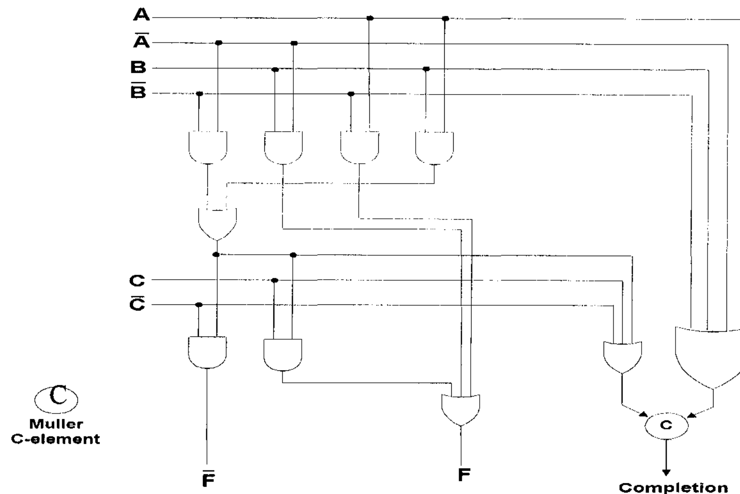


Figure 4.2: An example of two-rail implementation of function F(A,B,C) with completion signal.

Another reported implementation technique for designing asynchronous microprocessors was proposed by J. V. Woods, et. al. [29]. They developed an asynchronous implementation of the ARM microprocessor called AMULET1. The approach used in developing AMULET1 was based on Sutherland micropipelines [17]. The AMULET1 is divided into four major functional units. These include the address interface unit, register bank unit, execute unit, and data interface unit. These units operate concurrently and asynchronously. Figure 4.3 shows the major components of the execute unit of AMULET1 which has the pipeline structure. The AMULET1, like other asynchronous processors, does not constrain its ALU to produce results within a particular time and therefore may allow rare, worst-case

operation to take longer to complete. The self-timed ALU in AMULET1 was implemented as a simple ripple-through-carry adder with completion detection. Figure 4.4 shows the actual implementation of a single stage of the self-timed adder used in AMULET1. This implementation uses dynamic CMOS logic so that all the signals including the output of the NOR gate are initially zeroed. The ``done`` signals from all the stages are then combined using an AND gate tree to produce an overall completion signal which terminates the addition process. All the signals are zeroed again before the next operands are presented. ALU logical operations are allowed to complete at their faster, natural, speed by use of a separate, fixed, timing signal to indicate their completion.

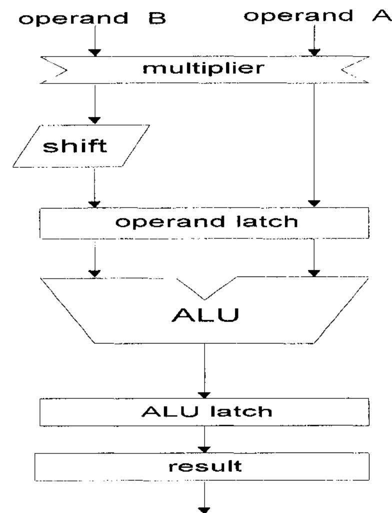


Figure 4.3: The Execution Pipe of AMULET1.

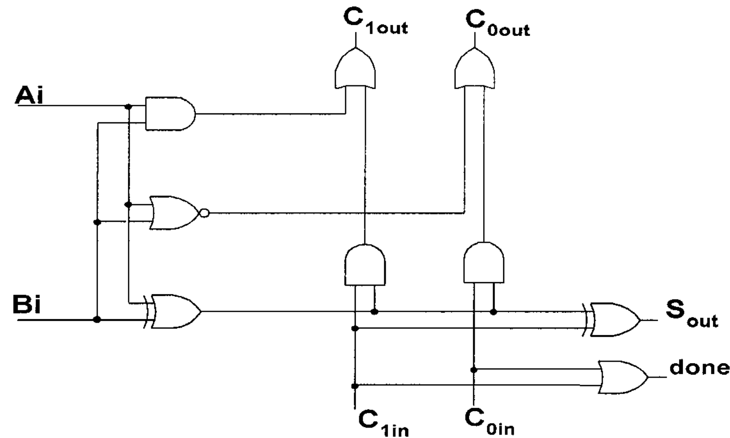


Figure 4.4: The self-timed adder of AMULET1.

By reviewing these three examples of asynchronous processors, especially their ALUs, several comments are noticed. The key feature of DCVSL family is the dual-rail coded nature. This characteristic is used to generate completion signals which indicate when a computation operation has finished. However, its major drawback is the need for a greater area for routing since it requires twice as many interconnection lines. Similarly, all the data signals in the TITAC design are represented as two-rail which means extra area. Furthermore, the number of gates required for the Verilog description of the data path of TITAC is almost twice the number required for similar synchronous processor. Finally, the completion detection circuitry of the ALU of AMULET1 is the major overhead for delay and area especially when the width of the ALU gets larger since a completion signal is generated at each stage.

The objective of this research is to design an efficient self-timed ALU based on a self-timed adder that employs dual rail encoding for the carry signals using two Manchester Carry Chains. In addition, matching delays are also employed for proper generation of completion signal. Such adder has an obvious area saving compared to implementation using DCVSL CMOS circuitry and is much faster due to proper utilization of matching delays as well as special speedup circuits. This self-timed adder is modified to allow handling other basic arithmetic and logical operations of the ALU.

CHAPTER 5

ADDER-BASED SELF-TIMED ALU

5.1 The Basic Adder Cell

As shown earlier, the adder is a basic component in any ALU since it can perform most of the required ALU functions. Thus, designing a high speed ALU requires an efficient high speed adder. The objective of this thesis is to design and characterize a self-timed ALU. For this purpose, the Manchester Carry Chain (MCC) adder with two carry chains is chosen. Among the types of adders explained earlier, the MCC adder is chosen due to its fast switching and regular structure. One carry chain is used to propagate the complements of the actual carries and will be referred to as MCC-1 while the second carry chain is used to propagate the actual carries and will be referred to as MCC-0. These two carry chains are actually the complements of one another. Completion of carry computation at the ith stage is detected when either of the ith stage of MCC-0 or MCC-1 is discharged low.

5.1.1 Introduction

Adding two n-bit operands A ($A_1 \rightarrow A_n$) and B ($B_1 \rightarrow B_n$) together with an input carry C_0 produces n sum bits S ($S_1 \rightarrow S_n$) and an output carry C_n according to the following equations:

$$S_i = P_i \oplus C_{i-1}$$

$$C_i = G_i + P_i C_{i-1}$$

$$G_i = A_i B_i$$

$$P_i = A_i \oplus B_i \text{ , for } i = 1, 2, \ldots, n$$

Where,

S_i: the ith sum bit

A_i: the ith bit of operand A

B_i: the ith bit of operand B

C_i: the carry bit of stage i

C_{i-1}: the carry bit of stage (i-1)

G_i: the ith carry generate signal

P_i: the ith carry propagate signal

The above equations suggest that n-carry bits ($C_1 - C_n$) need to be generated where the ith carry bit is generated from the $(i-1)^{th}$ carry bit. Accordingly, carry propagation is the major factor in determining the speed of addition since the sum

at stage i has to wait for the carry propagating from stage i-1. If, however, the carry bit can be computed without need for carry propagation, the sum bit can be generated directly. This situation occurs when G_i=1, i.e if and only if $A_i = B_i = 1$, in which case C_i=1 irrespective of the value of C_{i-1}.

Figure 5.1 shows the carry generation circuitry using a 4-bit MCC. The operation of the MCC adder with one carry chain was explained earlier.

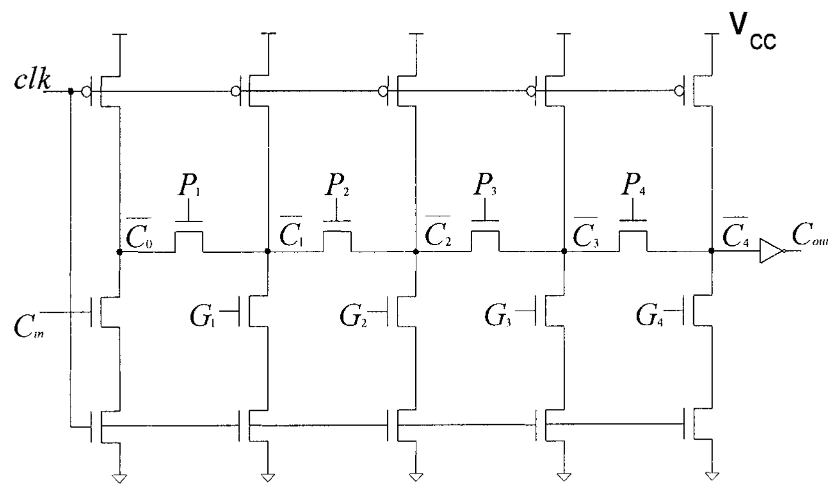


Figure 5.1: A 4-bit Manchester Carry Chain.

5.1.2 Double Rail Encoding

Another case where the carry of stage i can be computed without need for carry propagation is not covered by the circuitry of Figure 5.1. This case occurs when A_i and B_i are both zeros in which case C_i is 0 irrespective of the value of

C_{i-1}. This case is designated as the carry kill condition $K_i = \overline{A_i}\,\overline{B_i} = 1$. Thus, the two cases where C_i can be directly computed independent of the value of C_{i-1} occur when there is either carry generate condition (G_i=1) or a carry kill condition (K_i=1). Both of the carry generate and carry kill cases can be used to improve the average speed of the adder. This can be accomplished by using two MCC's, where one MCC (MCC-0) propagates the carries and detects the carry kill conditions, if any, while the other MCC (MCC-1) propagates the complements of the carries and detects the carry generate conditions if any. Once carry computation is complete, the corresponding nodes of both MCC's should have complementary values. The carry generate condition (Gi=1) is used to speed up computation of node voltages of one chain, while the carry kill condition (Ki=1) is used to speed up computation of node voltages of the other chain. Carry computation at stage *i* is complete when the corresponding nodes of the two MCC's have complementary values. Therefore, two MCC's are used to cover the above two conditions and the resultant adder is known as MCC adder with two carry chains. The MCC that detects the carry generate condition is referred to as the MCC-1 and the one which detects the carry kill condition is referred to as the MCC-0. Figures 5.2 and Figure 5.3 show the MCC-1 and MCC-0 circuitry respectively. The carry generate signals (G_i's) are used to discharge the nodes of

MCC-1 whenever a carry-generate (G_i=1) is detected. The carry kill signals (K_i's) are used to discharge the nodes of MCC-0 whenever a carry-kill condition (K_i=1) is detected.

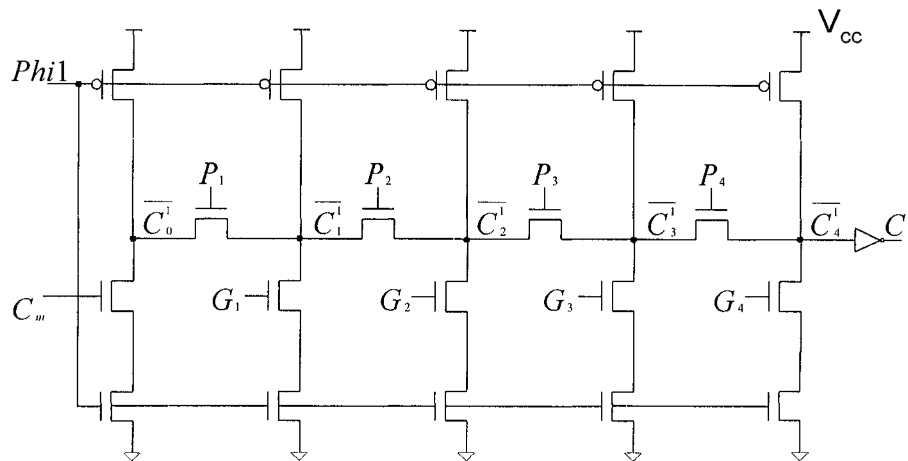


Figure 5.2: The implementation of the MCC-1 block.

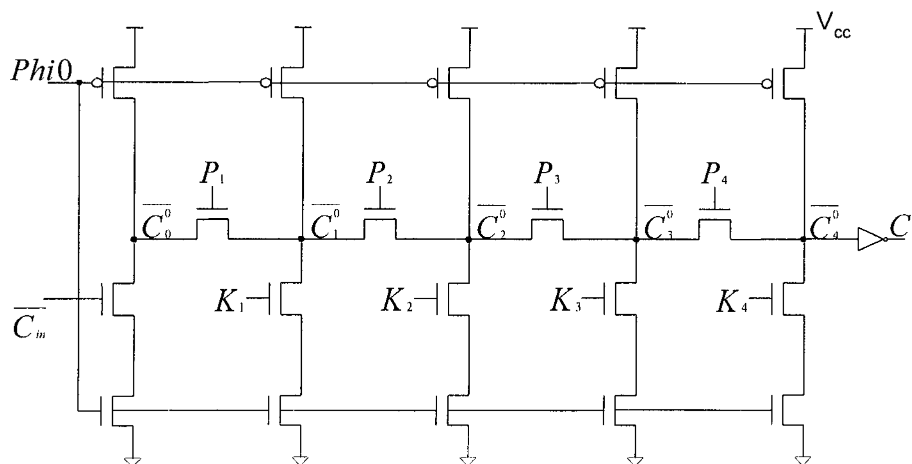


Figure 5.3: The implementation of the MCC-0 block.

Referring to Figures 5.2 and 5.3, three groups of important signals are identified:

1- The carry propagate signals P_i where $P_i = A_i \oplus B_i$. When $P_i=1$, $G_i=K_i=0$, in which case the carry is propagated from stage (i-1) to stage i in both MCC-1 and MCC-0, i.e $\overline{C_i^x} = \overline{C_{i-1}^x}$ for x = 0, 1.

2- The carry generate signals G_i where $G_i = A_iB_i$. When $G_i=1$, $\overline{C_i^1} = 0$ irrespective of the value of $\overline{C_{i-1}^1}$.

3- The carry kill signals K_i where $K_i = \overline{A_i}\,\overline{B_i}$. When $K_i=1$, $\overline{C_i^0} = 0$ irrespective of the value of $\overline{C_{i-1}^0}$.

From the above equations, it can be observed that the signals P_i, G_i, and K_i, are directly generated from the two input operands A and B after a constant delay.

The two carry chains, MCC-1 and MCC-0, are used for carry propagation/generation. The carry signals generated by the 2 carry chains can be considered as the double-rail encoding [6] of the corresponding carry signals. (See Table 5.1)

Table 5.1: Double Rail Encoding of the Carry Signal

$\overline{C_i}$	$\overline{C_i^1}$	$\overline{C_i^0}$
0	0	1
1	1	0
Spacer	1	1
Not allowed	0	0

The first carry chain (MCC-1) computes the $\overline{C_i^1}$ signals, while the second (MCC-0) computes the $\overline{C_i^0}$ signals. Thus, the MCC-1 is used for propagation/generation of the complement of the actual carry denoted as $\overline{C_i^1}$ while the MCC-0 is used for propagation/generation of the actual carry denoted as $\overline{C_i^0}$.

5.1.3 Implementation of Double Rail Encoding

Consider the truth table shown in Table 5.2 which lists the values of P_i, G_i, K_i, C_i, and $\overline{C_i}$ for all combinations of A_i, B_i, C_{i-1}. It is observed that, for

each combination of the inputs A_i, B_i, C_{i-1}, exactly one of P_i, G_i or K_i is equal to 1. Moreover, P_i=1 for 50% of the possible input combinations, while G_i=1 for 25% of the possible input combinations and K_i=1 for the remaining 25% of the possible input combinations. Again, the following two equations can be derived from Table 5.2 through the K-map:

$$C_i = G_i + P_i C_{i-1} \qquad (1)$$

$$\overline{C_i} = K_i + P_i \overline{C_{i-1}} \qquad (2)$$

Table 5.2: Truth Table of an Adder.

A_i	B_i	C_{i-1}	P_i	G_i	K_i	C_i	$\overline{C_i}$
0	0	0	0	0	1	0	1
0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	1
1	1	0	0	1	0	1	0
0	0	1	0	0	1	0	1
0	1	1	1	0	0	1	0
1	0	1	1	0	0	1	0
1	1	1	0	1	0	1	0

Equations 1 and 2 can be rewritten using the double-rail encoded carry signals (C_i^0 and C_i^1) shown in Table 5.1 as follows:

$$C_i^1 = G_i + P_i C_{i-1}^1 \quad (3)$$

$$C_i^0 = K_i + P_i C_{i-1}^0 \quad (4)$$

The complement of equation (3), i.e $\overline{C_i^1}$, can be implemented by the domino precharge slice shown in Figure 5.4 which is the basic slice in implementing MCC-1 shown in Figure 5.2. Thus, MCC-1 propagates $\overline{C_{i-1}^1}$ when P_i=1 or generates $\overline{C_i^1}$=0 when G_i=1 where in both cases the complement of the actual carry is propagated or generated accordingly. Similarly, the complement of equation (4) can be implemented by the domino precharge slice shown in Figure 5.5 which is the basic slice in implementing MCC-0 shown in Figure 5.3. Thus, MCC-0 propagates $\overline{C_{i-1}^0}$ when P_i=1 or generates $\overline{C_i^0}$=0 when K_i=1 where in both cases the actual carry is propagated or generated accordingly.

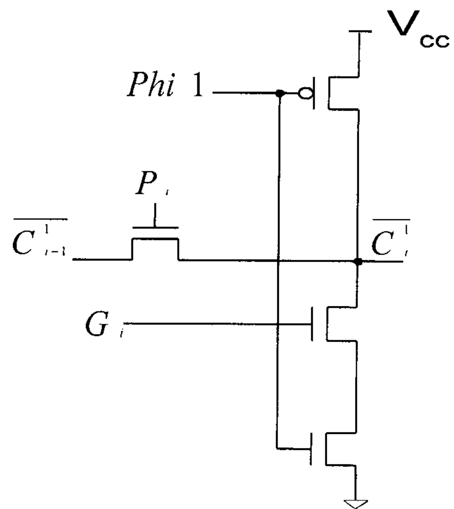


Figure 5.4: The Basic Slice for Implementing MCC-1.

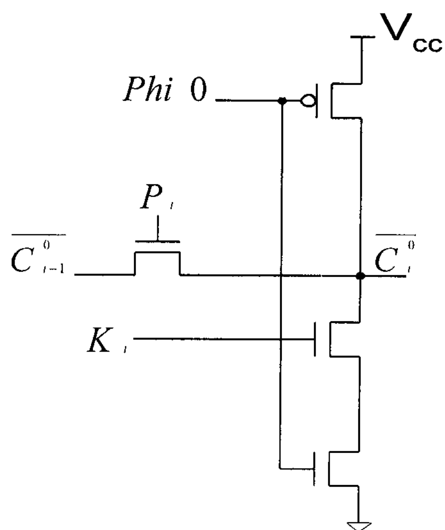


Figure 5.5: The Basic Slice for Implementing MCC-0.

Thus, the carry generate (G_i=1) and the carry kill (K_i=1) are used to speed up carry computation for MCC-1 and MCC-0 since they provide faster discharge paths for the evaluation of the carry node voltages through only two NMOS transistors. Table 5.3 shows the longest and the shortest discharge path for the $\overline{C_i^1}$ ($\overline{C_i^0}$) nodes of the MCC-1 (MCC-0) shown in Figure 5.2 (Figure 5.3).

Table 5.3: Discharge Paths for the Carry Nodes of MCC-1 & MCC-0.

Node	**Longest discharge path (P$_1$=P$_2$=P$_3$=P$_4$=1)**	**Shortest discharge path When G$_i$ (K$_i$) = 1**
$\overline{C_1^x}$	3	2
$\overline{C_2^x}$	4	2
$\overline{C_3^x}$	5	2
$\overline{C_4^x}$	6	2

For MCC-1 and MCC-0, the longest discharge path is six NMOS transistors while the shortest discharge path is only two NMOS transistors. Since the delay through a chain of n transistors is $O(n^2)$, it is not recommended to increase the number of transistors in the discharge path beyond 6. Wider adders will be constructed using these 4-bit MCC units. Hence, the choice of MCC 4-bit as a building block for

wider adders is acceptable since the longest discharge path through NMOS transistors will not exceed six transistors. The MCC-1 and MCC-0 are designed using dynamic CMOS domino technology and the use of an inverter at the final carry stage allows cascading of any number of such 4-bit MCC for wider carry chain [10].

Figure 5.6 shows the block diagram for a 4-bit MCC adder with the two Manchester carry chains (MCC-1, MCC-0). The P_i, G_i, and K_i signals are directly generated from the operands A and B after a constant delay as shown in Figure 5.7. The addition process proceeds as follows:

1- During the precharge phase (Phi=0), all carry nodes ($\overline{C_i^1}$ and $\overline{C_i^0}$) are pre-charged high through the pull up PMOS transistors. During this phase, the P_i, G_i, and K_i signals are also evaluated.

2- During the evaluate phase, the Phi signal becomes high and one of the two carry signals $\overline{C_i^1}$ or $\overline{C_i^0}$ is discharged for each node (i =1, 2, 3, 4).

Thus, complements of the actual carries $\overline{C_i^1}$ are propagated through the MCC-1 while the actual carries $\overline{C_i^0}$ are propagated through the MCC-0.

Either the carry signals of MCC-1 or those of MCC-0 can be used to generate the sum bits (S_1, S_2, S_3, S_4) through the use of either XOR or XNOR respectively.

However, this causes the carry signals to drive high capacitive loads (XOR/XNOR) as compared to inverters which results in slowing down carry computation on both chains. If the sum bits (S_1, S_2, S_3, S_4) are generated using only carry bits of MCC-1, this will cause MCC-1 to be much slower than MCC-0 due to heavier load. Thus, it is advantageous, in terms of speed, to balance the capacitive load on MCC-1 and MCC-0. Ideally, each MCC should contribute 2 carry signals to compute 2 of the 4-sum bits. Each carry node of MCC-1 or MCC-0 used to generate a sum bit must be buffered through an inverter where the output of the inverter is used to generate the sum bit. This results in both reducing the capacitive loads on the carry nodes of MCC-1 and MCC-0 and balancing this load between both chains.

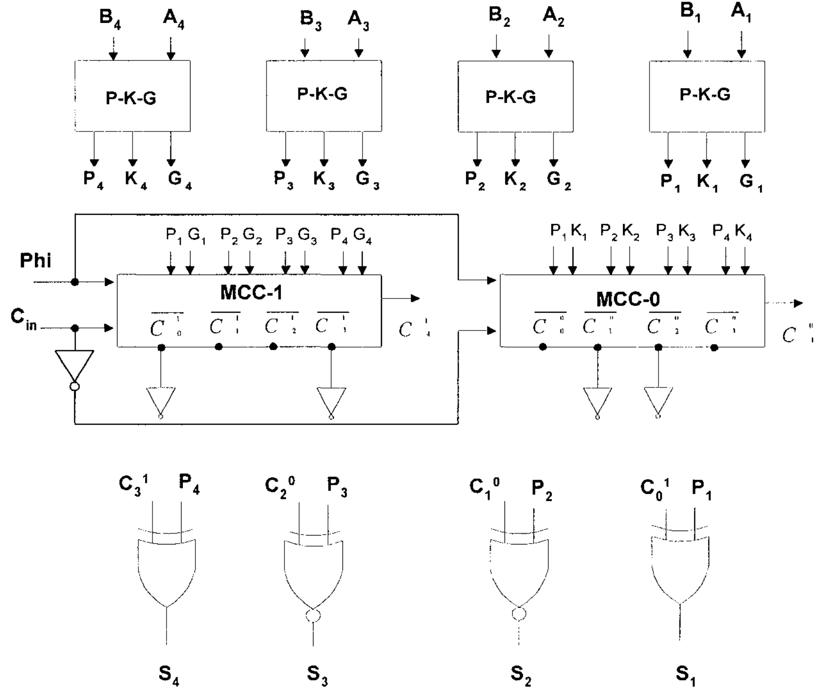


Figure 5.6: Block Diagram of a 4-bit MCC adder.

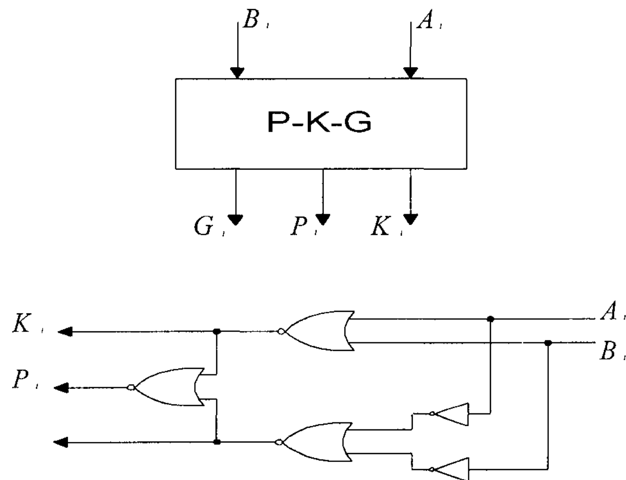


Figure 5.7: Implementation of Propagate-Kill-Generate Block.

5.1.4 Balancing the Capacitive Load

To balance the capacitive load on the carry nodes, we use two carry signals from MCC-1 and two carry signals from MCC-0 to generate the sum bits. However, to be more accurate the following RC delay analysis is performed.

The MCC-1 and MCC-0 shown in Figures 5.2 and 5.3 respectively can be modeled by an *RC* chain as shown in Figure 5.8 where it is assumed that each transistor has an ON-resistance of *R* and that each carry node has a capacitive load of *C*.

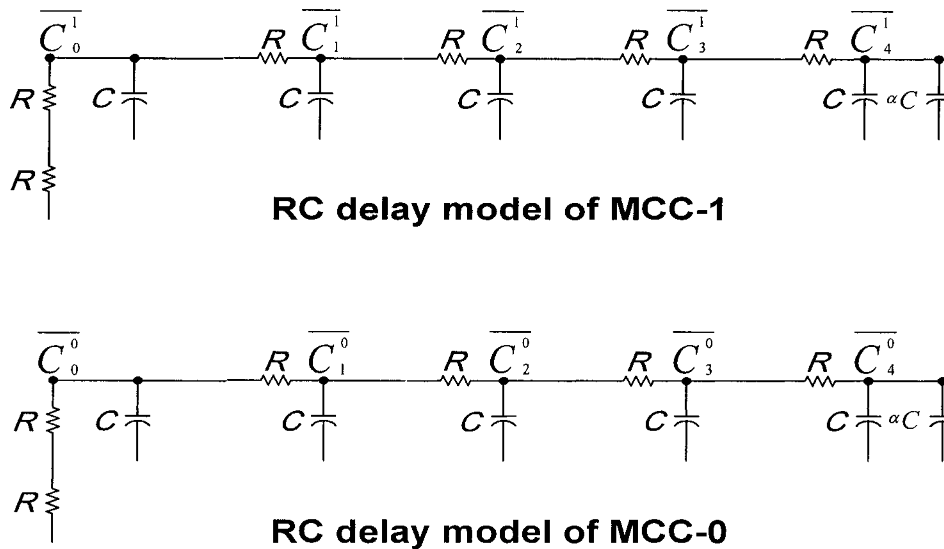


Figure 5.8: RC Delay Model of MCC-1, MCC-0.

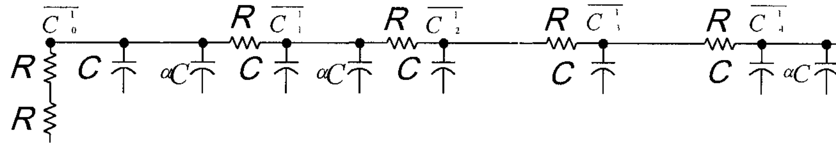
According to the model shown in Figure 5.8, any carry node selected to generate a sum bit should drive an inverter which in turn should increase the capacitive load on that carry node. This extra capacitive load is assumed to be αC. Based on this, the delay at each carry node of MCC-1 and MCC-0 can be calculated using the Elmore delay model [4]. Three different ways are possible to generate the sum bits using some carry nodes of MCC-1 and some carry nodes of MCC-0 as shown in Table 5.4.

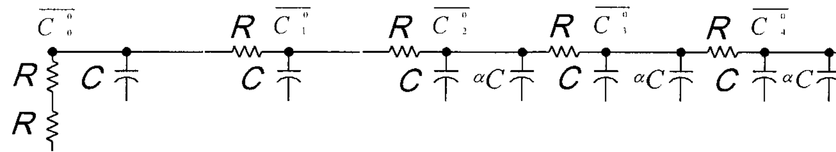
Table 5.4: Different Choices for Selecting Carry Nodes.

	Selected carry nodes from MCC-1		Selected carry nodes from MCC-0	
1st choice	$\overline{C_0^1}$	$\overline{C_1^1}$	$\overline{C_2^0}$	$\overline{C_3^0}$
2nd choice	$\overline{C_0^1}$	$\overline{C_2^1}$	$\overline{C_1^0}$	$\overline{C_3^0}$
3rd choice	$\overline{C_0^1}$	$\overline{C_3^1}$	$\overline{C_1^0}$	$\overline{C_2^0}$

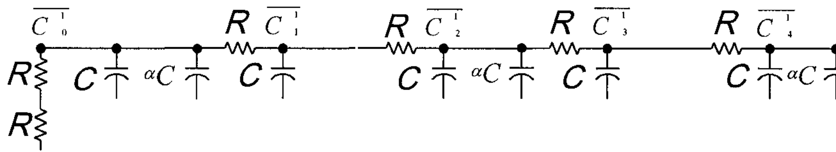
The *RC* delay models and the corresponding Elmore delay calculations for the choices of Table 5.4 are shown in Figure 5.9 and Table 5.5 respectively.

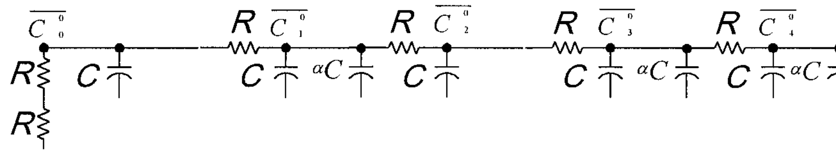
RC delay models of MCC-1 and MCC-0 of the first choice



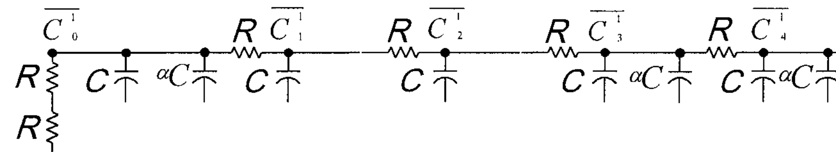


RC delay models of MCC-1 and MCC-0 of the second choice





RC delay models of MCC-1 and MCC-0 of the third choice



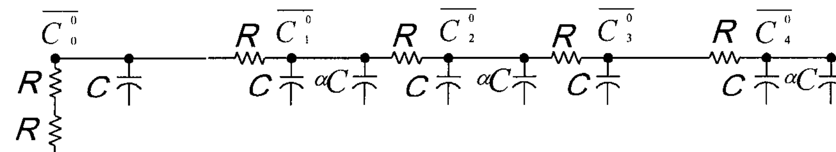


Figure 5.9: RC Delay Models of the Three Choices.

Table 5.5: Elmore Delays of the Carry Nodes.

First Choice			
Node of MCC-1	**Elmore delay**	**Node of MCC-0**	**Elmore delay**
$\overline{C_0^1}$	$10RC + 6\alpha RC$	$\overline{C_0^0}$	$10RC + 6\alpha RC$
$\overline{C_1^1}$	$14RC + 8\alpha RC$	$\overline{C_1^0}$	$14RC + 9\alpha RC$
$\overline{C_2^1}$	$17RC + 9\alpha RC$	$\overline{C_2^0}$	$17RC + 12\alpha RC$
$\overline{C_3^1}$	$19RC + 10\alpha RC$	$\overline{C_3^0}$	$19RC + 14\alpha RC$
$\overline{C_4^1}$	$20RC + 11\alpha RC$	$\overline{C_4^0}$	$20RC + 15\alpha RC$
Second Choice			
Node of MCC-1	**Elmore delay**	**Node of MCC-0**	**Elmore delay**
$\overline{C_0^1}$	$10RC + 6\alpha RC$	$\overline{C_0^0}$	$10RC + 6\alpha RC$
$\overline{C_1^1}$	$14RC + 8\alpha RC$	$\overline{C_1^0}$	$14RC + 9\alpha RC$
$\overline{C_2^1}$	$17RC + 10\alpha RC$	$\overline{C_2^0}$	$17RC + 11\alpha RC$
$\overline{C_3^1}$	$19RC + 11\alpha RC$	$\overline{C_3^0}$	$19RC + 13\alpha RC$
$\overline{C_4^1}$	$20RC + 12\alpha RC$	$\overline{C_4^0}$	$20RC + 14\alpha RC$

Table 5.5: Elmore Delays of the Carry Nodes (Continued).

Third Choice			
Node of MCC-1	**Elmore delay**	**Node of MCC-0**	**Elmore delay**
$\overline{C_0^1}$	$10RC + 6\alpha RC$	$\overline{C_0^0}$	$10RC + 6\alpha RC$
$\overline{C_1^1}$	$14RC + 8\alpha RC$	$\overline{C_1^0}$	$14RC + 9\alpha RC$
$\overline{C_2^1}$	$17RC + 10\alpha RC$	$\overline{C_2^0}$	$17RC + 11\alpha RC$
$\overline{C_3^1}$	$19RC + 12\alpha RC$	$\overline{C_3^0}$	$19RC + 12\alpha RC$
$\overline{C_4^1}$	$20RC + 13\alpha RC$	$\overline{C_4^0}$	$20RC + 13\alpha RC$

From the above table, it is observed that the third choice is the best one in terms of balancing the capacitive load on the carry nodes of MCC-1 and MCC-0. Thus, the inversions of the carry nodes $\overline{C_0^1}$ and $\overline{C_3^1}$ are used via two XOR gates to generate the sum bits S_1 and S_4 respectively while the inversions of the carry nodes $\overline{C_1^0}$ and $\overline{C_2^0}$ are used via two XNOR gates to generate the sum bits S_2 and S_3 respectively as shown in Figure 5.6. Therefore, it is assumed that the generation of the sum bits from adding two 4-bit operands A and B using two

MCC's (MCC-1, MCC-0) will be implemented according to the following equations:

For MCC-1,

$$S_i = A_i \oplus B_i \oplus C^1_{i-1} \quad \text{, for } i=1, 4$$

$$C^1_i = G_i + P_i C^1_{i-1} \quad \text{, for } i=1, 4$$

For MCC-0,

$$S_i = A_i \oplus B_i \odot C^0_{i-1} \quad \text{, for } i=2, 3$$

$$C^0_i = K_i + P_i C^0_{i-1} \quad \text{, for } i=2, 3$$

It should be noted that the initial carry C_{in} is fed into MCC-1 while its complement is fed into MCC-0 through an inverter. Figures 5.10 and 5.11 show the implementation of the XOR and the XNOR gates respectively.

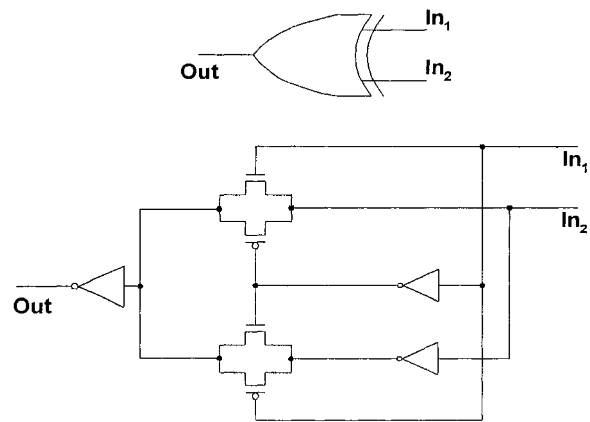


Figure 5.10: Implementation of the XOR Gate.

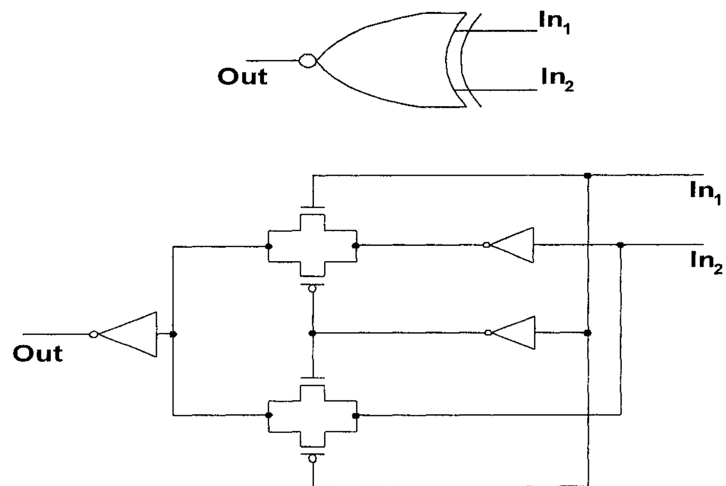


Figure 5.11: Implementation of the XNOR Gate.

5.2 Completion Detection for 4-bit MCC Adder

Carry propagation is the major delay factor for adders and, hence, for adder-based ALUs. The distance that a carry has to propagate through depends on the operands being added [2, 5, 8]. Therefore, designing an asynchronous adder will improve the average delay of an adder and, hence, the average delay of adder-based ALU. To achieve asynchronous average speed addition, a completion detection circuitry must be included to indicate/signal completion of operation [2, 11]. Since the completion detection circuitry represents hardware overhead, it is essential that such circuit be as efficient as possible in terms of area as well as delay.

The Final Completion Signal (FCS) should be asserted after computation is done. Since carry propagation is the major delay factor for an operation to be completed, it can be used to signal the completion of an operation. The two Manchester carry chains, MCC-1 and MCC-0, can be easily used to detect completion of the carry propagation process. It has been shown that the average carry propagation distance through the MCC is O(log(n)) where n is the adder width [1,3,6,12]. Since both MCC's are initially precharged, carry computation at the ith node is complete when either of its 2 corresponding carry nodes ($\overline{C_i^1}, \overline{C_i^0}$) is discharged low. Thus, it is possible to generate the FCS from all stages of an

adder using the carry nodes on both MCC-1 and MCC-0 [9]. Such scheme is accurate in generating the FCS after all carry bits are properly computed. However, if all carry nodes are used to generate the FCS, the completion detection circuitry will be unacceptably large for even small adder widths. Furthermore, this would increase the capacitive load on all MCC nodes thus slowing down carry propagation. Instead of using all carry nodes to generate the FCS, we have opted to only use the middle carry nodes $\overline{C_2^1}$ and $\overline{C_2^0}$ for this purpose. Using the middle carry nodes of MCC-1 and MCC-0 to generate the FCS is reasonable since the average distance for carry propagation through the MCC is known to be O(log(n)) where n is the width of the MCC. In addition to reducing the capacitive load on the carry nodes, this approach will maintain the average delay of the FCS to be within the average delay of the MCC as well as reducing the area needed for the completion detection circuitry. For proper self-timing, the FCS should be asserted only after all sum and carry bits are computed to indicate validity of result. Moreover, the delay between the last computed result and FCS should be minimized. Validation of the FCS signal requirements has been carried out and verified through simulation using SPICE as will be shown in details in the next chapter.

Figure 5.12 illustrates how the FCS signal is generated. During the pre-charge phase, $\overline{C_2^1}$ and $\overline{C_2^0}$ are pulled high and the CS signal is set low. The CS passes through the delay matching circuit and the FCS is set low. During the evaluate phase, however, either $\overline{C_2^1}$ or $\overline{C_2^0}$ but not both goes low causing the CS to turn high and, hence, the FCS to turn high. Details of the delay matching circuit will be explained in the next chapter.

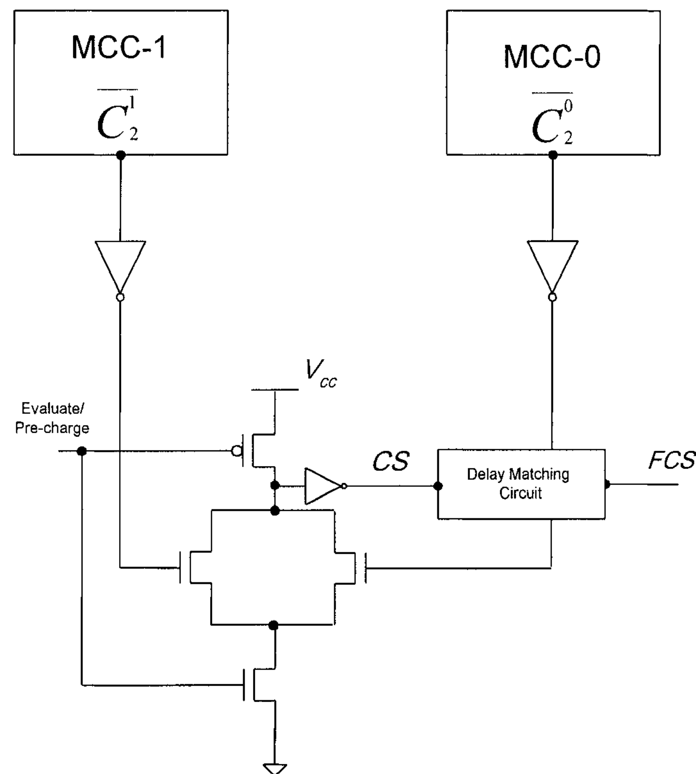


Figure 5.12: Completion Detection Circuit for a 4-bit MCC Adder.

Based on the above discussion and referring back to Figure 5.6, an extra inverter is needed at the carry node $\overline{C^1_2}$ of MCC-1. Thus, three carry nodes of the MCC-1 drive three inverters while two carry nodes of the MCC-0 drive only two inverters. Therefore, the input carry C_{in} is used instead of the carry node $\overline{C^1_0}$ of the MCC-1 to evaluate the sum bit S_1 since the value of C_{in} is equivalent to the inversion value of $\overline{C^1_0}$. This modification balances the load on both MCC-1 and MCC-0 where two carry nodes on each chain drive two inverters as shown in Figure 5.13.

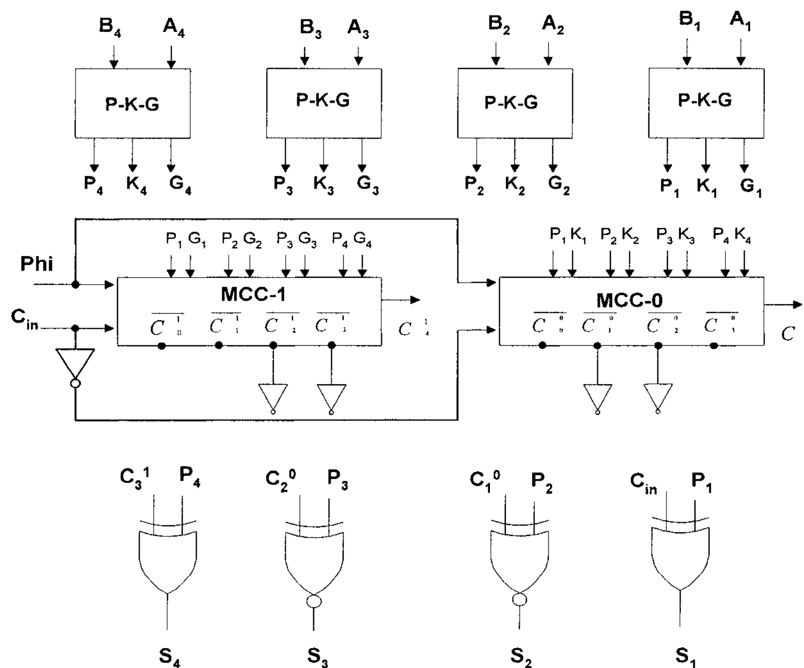


Figure 5.13: The Modified 4-bit MCC Adder.

5.3 Completion Detection for n-bit MCC Adder

MCC n-bit adder will be built using (n/4) MCC 4-bit adder blocks. It is assumed that n is in general divisible by 4. For wider adders, C_4^1 is fed into MCC-1 of the next block as an initial carry and is used to evaluate the sum bit S_5 while C_4^0 of the first block is fed into MCC-0 of the next block as an initial carry and so on. Figure 5.14 shows how an n-bit MCC adder is constructed using 4-bit adder blocks.

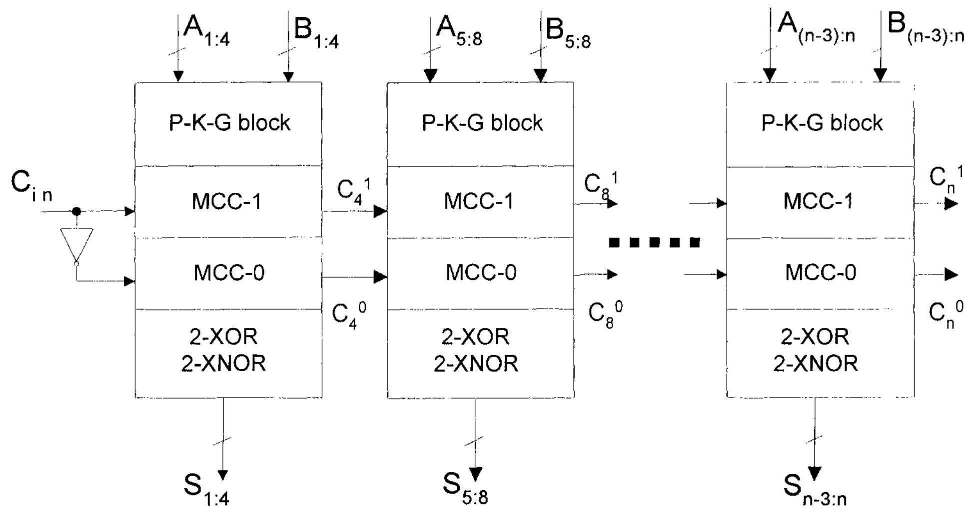
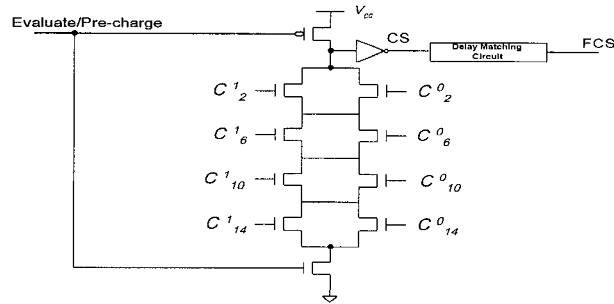


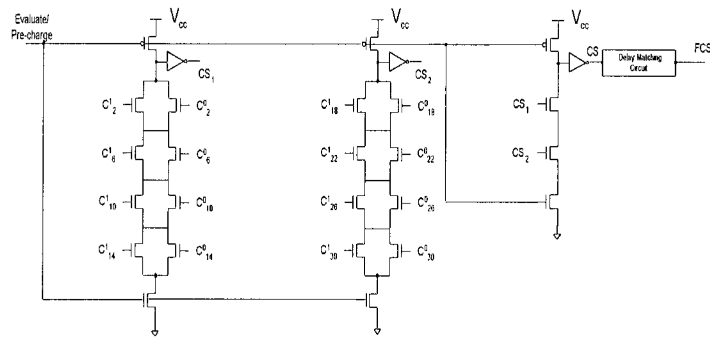
Figure 5.14: N-bit MCC Adder.

The middle carry nodes of the MCC-1's and the MCC-0's are used to generate the FCS for the n-bit MCC adder. Figure 5.15 (a, b, c) shows how the FCS is generated for a 16, 32, 64-bit self-timed MCC adder respectively. Thus, one level

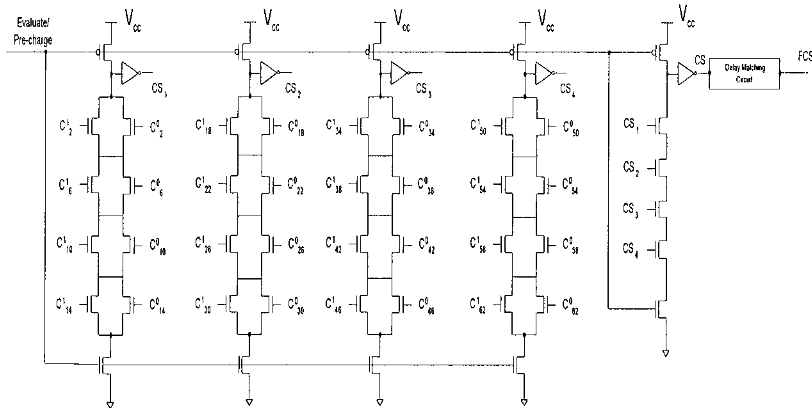
of completion detection is needed to generate the FCS for 16-bit MCC adder. However, tow levels of completion detection are required to generate the FCS for 32 and 64-bit MCC adder. Therefore, for an n-bit MCC adder, $\left|\log_4(n/4)\right|$ levels of completion detection are required to generate the FCS.



(a) : FCS for 16-bit MCC adder.



(b) : FCS for 32-bit MCC adder.



(c) : FCS for 64-bit MCC adder.

Figure 5.15: Completion Detection Circuit for 16, 32, 64-bit MCC Adder.

5.4 The Asynchronous ALU

Generally, any ALU should perform arithmetic, logical, and shift operations. Thus, the proposed self-timed ALU should have the ability to perform these operations. Arithmetic operations include addition, subtraction, increment, decrement, and arithmetic complement. Logical operations include ANDing, ORing, XORing, XNORing, and logical complement. Shift operations include logical shift right and left, arithmetic shift right and left, rotate right and left, and transfer. Figure 5.16 shows the block diagram of the proposed self-timed ALU with the required two n-bit operands A and B as inputs, the request signal (REQ), the acknowledge signal (ACK), the control signals, and the latched version (LS) of the resulting sum bits S. The sum bits S give the required result of any arithmetic or logical operation.

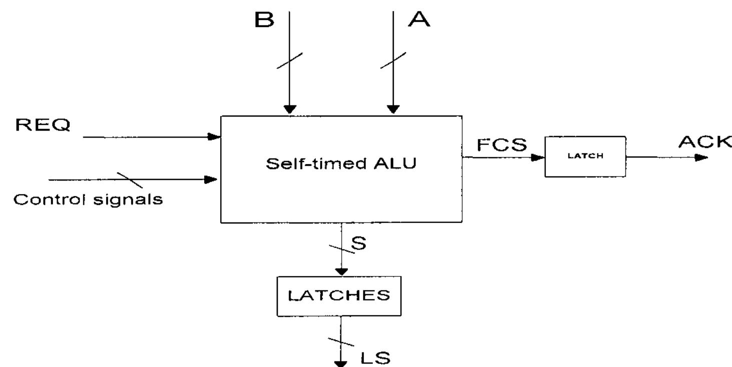


Figure 5.16: Block diagram of the self-timed ALU.

Adders can be modified to allow performing other arithmetic and logical operations. Normally, the sum S of two operands A and B follows the boolean functions:

$$\mathbf{S_i = A_i \oplus B_i \oplus C_{i-1}}$$

$$\mathbf{C_i = A_iB_i + ((A_i \oplus B_i)C_{i-1})}$$

From the above two equations, it can be shown that by forcing some terms to either logic 0 or logic 1, other logical operations can be performed. Implementation of other arithmetic operations, however, is mainly achieved through manipulating the way of presenting the operands and the input carry to the adder.

Referring back to Figure 5.13, the addition of two operands A and B with sum S follows the boolean functions:

$$S_i = A_i \oplus B_i \oplus C_{i-1}^1 \quad \text{, for i=1, 4}$$

$$C_i^1 = G_i + P_i C_{i-1}^1 \quad \text{, for i=1, 4}$$

$$S_i = A_i \oplus B_i \odot C_{i-1}^0 \quad \text{, for i=2, 3}$$

$$C_i^0 = K_i + P_i C_{i-1}^0 \quad \text{, for i=2, 3}$$

In this section, we are going to present the proposed modifications required to allow the adder of Figure 5.13 to perform other arithmetic and logical operations. We will also show how shift operations can be implemented by means of an ***nxn*** barrel shifter.

5.4.1 Arithmetic Operations

For arithmetic operations, all numbers are assumed to be in 2's complement format. Arithmetic operations other than addition include subtraction, incrementing, decrementing, and arithmetic complement. These operations can be easily implemented by proper presentation of operands to the adder of Figure 5.13.

1. **Subtraction:** the subtraction of two operands A and B can be represented as:

$A - B = A + \overline{B} + 1$, where $\overline{B}$ is the logical complement of B. This is accomplished by feeding operand A at port A of the adder, feeding the logical complements of operand B at port B, and setting the carry-in (C_{in}) to 1. Similarly, it is possible to implement $B - A$. However, the operation $(-A - B)$ can not be implemented

directly using the adder shown in Figure 5.13 as it would require a correction factor of +2 which can not be achieved through the initial carry in.

2. **Increment:** incrementing A is a unary operation. This operation can be implemented by feeding operand A to port A of the adder, setting port B to 0, and setting the carry-in to 1 which will increment operand A by one using the adder cell.

3. **Decrement**: decrementing A (i.e. A-1) is a unary operation. This can be achieved by feeding operand A to port A of the adder, feeding all 1's to port B, and setting the carry-in to 0.

4. **Arithmetic Complement:** arithmetic complement of operand A is its two's complement. The two's complement of A can be obtained by inverting the bits of A and adding 1 to it. Thus, arithmetic complement of operand A can be achieved by feeding the logical complement of operand A at port A, setting port B to 0, and setting the carry-in to 1.

Table 5.6 shows how port A and port B of a 4-bit adder and the initial carry in C_{in} are set to perform subtraction, increment, decrement, and arithmetic complement.

Table 5.6: Operands at Port-A, Port-B, C_{in} for Arithmetic Operations.

Operation	**Port A**	**Port B**	**C_{in}**
Addition (A+B)	A	B	0
Subtraction (A-B)	A	$\overline{B}$	1
Increment (A+1)	A	0000	1
Decrement (A-1)	A	1111	0
Arithmetic complement $\overline{A}$	$\overline{A}$	0000	1

Since different operands are fed into the two input ports (port A, port B), a set of 2x1 multiplexers (MUX's) are needed at the inputs of the two ports to perform the arithmetic operations. From Table 5.7, it can be seen that either the operand A or its complement $\overline{A}$ need to be passed to port A. For port B, however, either operand B itself, its complement $\overline{B}$, an all 0's value or an all 1's value need to be passed to port B depending on the required operation. This can be accomplished through the use of multiplexers as shown in Figures 5.17 and 5.18.

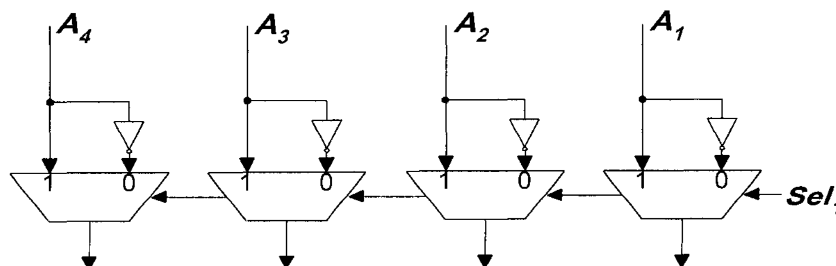


Figure 5.17: Multiplexing at Port A.

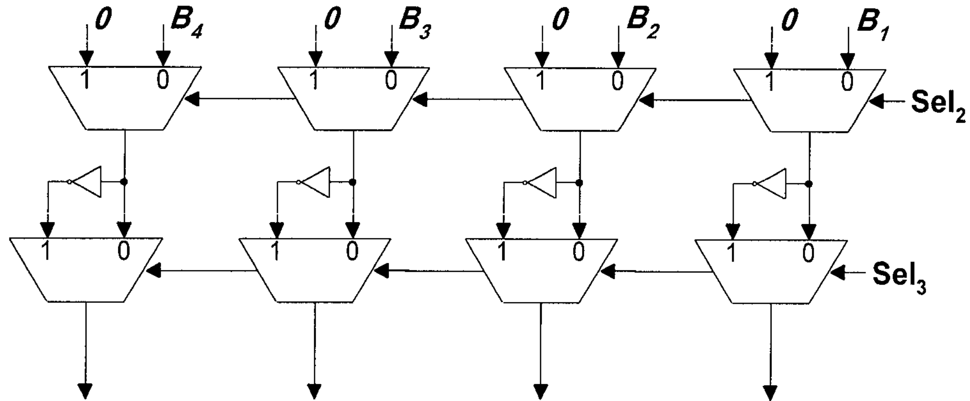


Figure 5.18: Multiplexing at Port B.

Table 5.7: Multiplexing Values at Port-A, Port-B for the Arithmetic Operations.

Operation	**Sel1**	**Sel2**	**Sel3**	**Selected operand at port A**	**Selected operand at port B**
Addition	1	0	0	Operand A	Operand B
Subtraction	1	0	1	Operand A	$\overline{B}$
Increment	1	1	0	Operand A	Zeros
Decrement	1	1	1	Operand A	Ones
Arithmetic complement	0	1	0	$\overline{A}$	Zeros

5.4.2 Logical Operations

To compute the required logical operations, e.g. AND, OR …, some modifications to the basic adder logic (Figure 5.13) are necessary. This section discusses how these operations are computed using the basic developed adder logic and the logic modifications required in this case. The logical operations to be incorporated, AND, OR, XOR, XNOR, and logical complement, can be derived from the basic adder cell shown in Figure 5.13 as follow:

1. **<u>Exclusive OR:</u>** the XOR and the XNOR gates that generate the sum bits can be utilized to provide the exclusive OR function of the two operands A and B. Recall that:

For MCC-1,

$$S_i = A_i \oplus B_i \oplus C^1_{i-1} \text{ , for i=1, 4}$$

For MCC-0,

$$S_i = A_i \oplus B_i \odot C^0_{i-1} \text{ , for i=2, 3}$$

From the above two equations, it is noted that forcing carry nodes of MCC-0 to be zero ($\overline{C^{0}_{i-1}}$=0) and forcing carry nodes of MCC-1 to be one ($\overline{C^{1}_{i-1}}$=1), the sum bits S will give the required result A⊕B.

2. Exclusive NOR: the XOR and the XNOR gates that generate the sum bits can be utilized to provide the exclusive NOR function of the two operands A and B. Again recall that:

For MCC-1,

$$S_i = A_i \oplus B_i \oplus C^{1}_{i-1} \quad \text{, for i=1, 4}$$

For MCC-0,

$$S_i = A_i \oplus B_i \odot C^{0}_{i-1} \quad \text{, for i=2, 3}$$

From the above two equations, it is noted that forcing carry nodes of MCC-0 to be one ($\overline{C^{0}_{i-1}}$=1) and forcing carry nodes of MCC-1 to be zero ($\overline{C^{1}_{i-1}}$=0), the sum bits S will give the required result A⊙B.

3. ANDing: the sum bits, (S_1, S_2, S_3, S_4), should give the result of ANDing operands A and B. The generate signals $G_i = A_iB_i$ generated by the P-K-G block can be utilized to implement the AND operation. If G_i is XORed with zero, the

result will be A_iB_i. Referring back to Figure 5.13, by passing G_i instead of P_i to the two XOR gates and forcing carries of MCC-1 to be one, the sum bits, S_1 and S_4, will actually give the required result A_iB_i for i=1, 4. Thus, a 2x1 MUX is needed at one input of both XOR gates to pass either P_i or G_i while the other input of both XOR gates will pass the complement of the carries of MCC-1. Now, the sum bits, S_2 and S_3, should give the required result A_iB_i for i=2, 3 via the two XNOR gates. It is observed that $P_i \odot K_i$ is equivalent to A_iB_i according to the truth table shown at Table 5.8.

Table 5.8: Implementing AND Operation from the Adder Resources.

$\mathbf{A_i}$	$\mathbf{B_i}$	$\mathbf{K_i}$	$\mathbf{P_i}$	$\mathbf{P_i \odot K_i}$	$\mathbf{A_iB_i}$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	1	0	0
1	1	0	0	1	1

Thus, by XNORing P_i and K_i, the sum bits S_2 and S_3 will actually give the required result A_iB_i for i=2, 3 respectively.

4. <u>ORing:</u> the sum bits, (S_1,S_2,S_3,S_4), should give the result of ORing operands A and B. The kill signal $K_i = \overline{A_i}\overline{B_i}$ generated by the P-K-G block can be utilized to implement the OR operation. If K_i is XNORed with zero, the result will be A_i+B_i. Referring back to Figure 5.13, by passing K_i instead of P_i to the two XNOR gates and forcing carries of MCC-0 to be one, the sum bits, S_2 and S_3, will actually give the required result A_i+B_i for i=2, 3. Thus, a 2x1 MUX is needed at one input of both XNOR gates to pass either P_i or K_i while the other input of both XNOR gates will pass the complement of the carries of MCC-0. Now, the sum bits, S_1 and S_4, should give the required result A_i+B_i for i=1, 4 via the two XOR gates. It is observed that $P_i \oplus G_i$ is equivalent to A_i+B_i according to the truth table shown at Table 5.9.

Table 5.9: Implementing OR Operation from the Adder Resources.

A_i	B_i	G_i	P_i	$P_i \oplus G_i$	A_i+B_i
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	1

Thus, by XORing P_i and G_i, the sum bits S_1 and S_4 will actually give the required result A_i+B_i for i=1, 4 respectively.

5. **Logical Complement:** logical complement of operand A can be achieved through the adder since the inversion of operand A is available at port A. Then, setting port B to zero and applying the conditions for OR-ing the contents of ports A and B; i.e. forcing carries of MCC-0 to be one and forcing carries of MCC-1 to be zero; will pass the inversion of operand A to the sum bits (S_1, S_2, S_3, S_4).

Table 5.10 summarizes the requirements to implement the logical operations mentioned above using the adder shown at Figure 5.13.

Table 5.10: Implementing Logical Operations Using The Adder Resources.

Operation	**Port A**	**Port B**	**Carry Nodes of MCC-1**	**Carry Nodes of MCC-0**
XOR	A	B	Forced to 1	Forced to 0
XNOR	A	B	Forced to 0	Forced to 1
AND	A	B	Forced to 1	Forced to 0
OR	A	B	Forced to 0	Forced to 1
Logical complement	$\bar{A}$	0000	Forced to 0	Forced to 1

Since the above mentioned logical operations are generated via 2 XOR and 2 XNOR gates that are used to generate the sum bits S, a set of 2x1 MUX's are

needed at the inputs of the XOR and the XNOR gates to feed the proper values. Figure 5.19 shows how the XOR and XNOR gates that are used to generate the sum bits S are utilized to generate other logical operations according to the selection lines (D, E) as shown in Table 5.11. It should be noted that when carries of MCC-1, *(MCC-0)* are forced to be zeros, carries of MCC-0, *(MCC-1)* must be forced to ones to activate the completion signal (CS) and, hence, the Final Completion Signal (FCS).

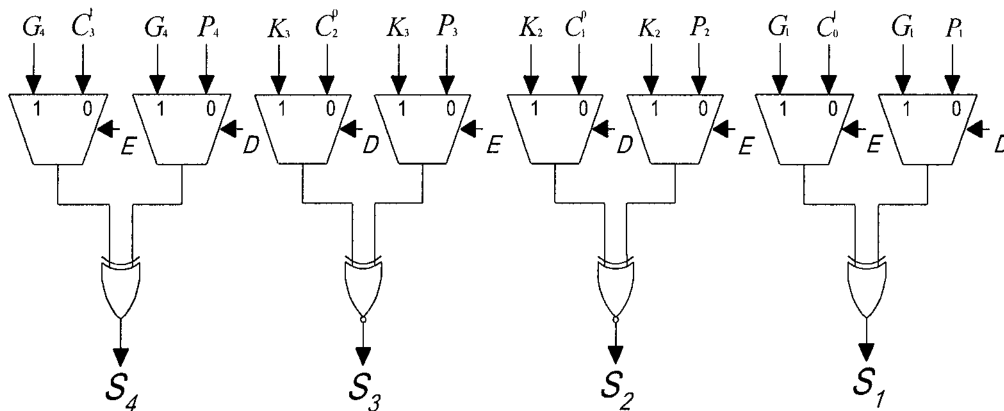


Figure 5.19: Multiplexing inputs of XOR and XNOR gates.

Table 5.11: Multiplexing inputs at XOR & XNOR Gates.

Operation	**D**	**E**	**Selected inputs of XOR gates for i=1,4**	**Selected inputs of XNOR gates for i=2,3**
Arithmetic operations	0	0	(C^1_{i-1}, P_i)	(C^0_{i-1}, P_i)
XOR	0	0	$(C^1_{i-1} = 0, P_i)$	$(C^0_{i-1} = 1, P_i)$
XNOR	0	0	$(C^1_{i-1} = 1, P_i)$	$(C^0_{i-1} = 0, P_i)$
AND	1	0	$(C^1_{i-1} = 0, G_i)$	(K_i, P_i)
OR	0	1	(G_i, P_i)	$(C^0_{i-1} = 0, K_i)$
Logical complement	0	1	(G_i, P_i)	$(C^0_{i-1} = 0, K_i)$

5.4.3 The Modified Adder Cell

In this part, the basic adder cell will be modified to allow performing different arithmetic and logical operations. As mentioned before, an adder can be used to implement logical and arithmetic operations. Arithmetic operations do not require any changes to the basic structure of the adder cell since they only require feeding of proper operands and carry-in values according to the required operation. The implementation of logical operations, however, requires some modification to the

adder cell specially in the two Manchester carry chains whose nodes are required to be forced either to zeros or ones. It should be noted that if carries on one MCC are forced to zero then carries on the other MCC must be forced to one and vice versa in order to activate the completion detection process.

The elemental circuit of the MCC that can force a carry node to zero or one is shown in Figure 5.20. It is similar to the elemental circuit used for building the MCC's except that the control signal F is OR-ed with G_i or K_i. To force the MCC nodes to 0's, F is set to logic 1 during the evaluate phase. To force these nodes to logic 1 state, the precharge signal Phi is locked to a logic 0 state which maintains the carry nodes at logic 1 irrespective of the values of G_i or K_i. It should be noted that if the nodes of one of the MCC's are forced to 1's, the nodes of the other MCC must be forced to 0's. This implies that the 2 MCC's can not use the same precharge signal Phi. Thus, two precharge signals, Phi1 and Phi0, are needed for MCC-1 and MCC-0 respectively.

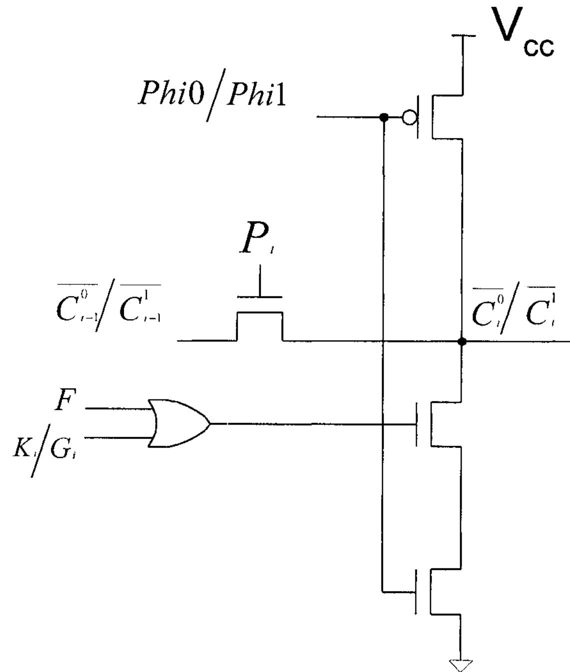


Figure 5.20: The Modified Elemental MCC Cell.

Figure 5.21 and Figure 5.22 show the implementations of MCC-1 and MCC-0 after modifications respectively. The same signal F will be used for both MCC-1 and MCC-0 because signal F is low for arithmetic operations but for logical operations signal F is used to force nodes of either MCC-1 or MCC-0 to zero but not both. Signals F, Phi_0, and Phi_1 are generated via a decoder according to the implemented operation. Table 5.12 lists the values of F, Phi_0, and Phi_1 during the evaluate phase for arithmetic and logical operations.

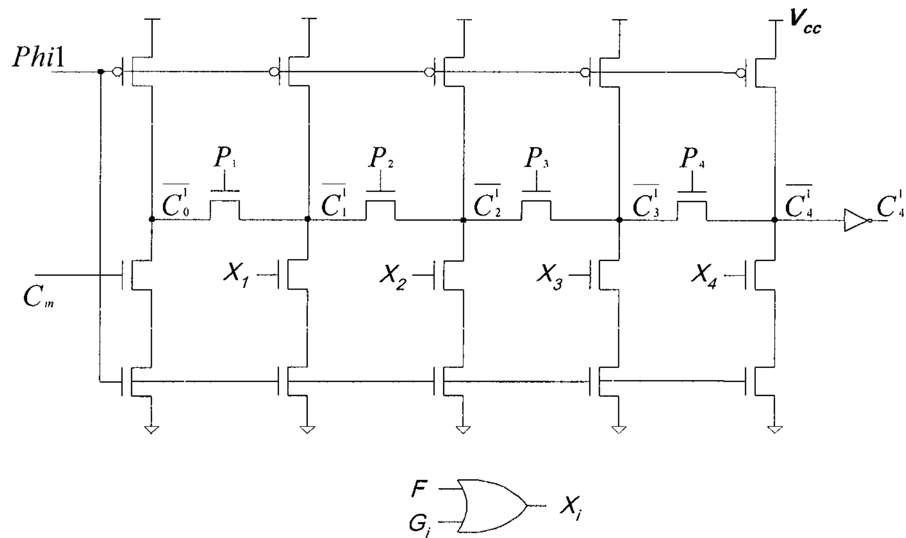


Figure 5.21: The Modified MCC-1.

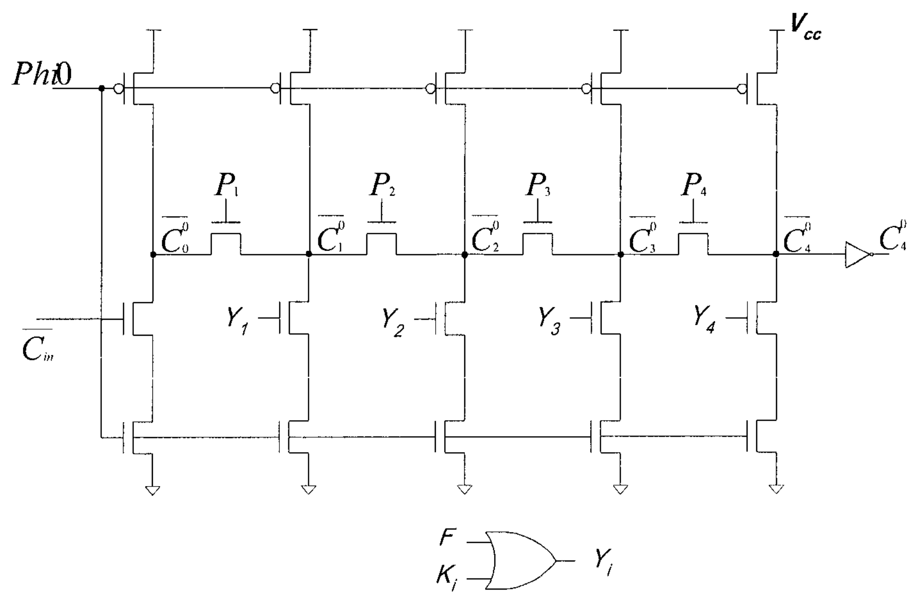


Figure 5.22: The Modified MCC-0.

Table 5.12: Values of F, Phi0, Phi1 During Evaluate Phase.

Operation	**F**	**Phi_0**	**Phi_1**
Arithmetic operations	0	1	1
XOR	1	1	0
XNOR	1	0	1
AND	1	1	0
OR	1	0	1
Logical Complement	1	0	1

5.4.4 Shift Operations

To allow for shift as well as rotation operations, a barrel shifter block is used at the output of the modified adder as shown in Figure 5.23. Shift operations include various types of shifts, arithmetic and logical, left or right as well as rotation operations. The adder sum bits are the inputs of the barrel shifter. The output bits of the barrel shifter (F_1, F_2, F_3, F_4) are the final output result of the asynchronous ALU for all operations (arithmetic, logical, shift). The operand to be shifted or rotated is applied at port A while port B is set to zero.

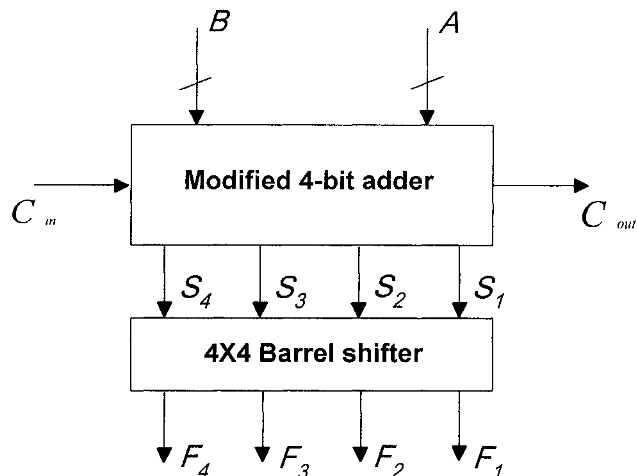


Figure 5.23: Interconnection of Adder and Barrel Shifter.

5.4.5 Barrel Shifter Implementation

The commonly used shift operations are one-bit shifts. These include arithmetic shift left, arithmetic shift right, logical shift left, logical shift right, rotate left and rotate right. An $n \times n$ barrel shifter can implement shift operations on an n-bit operand. In addition to shift and rotate operations, the barrel shifter should have a direct transfer operation to pass the result of the arithmetic and logical operations directly to the output without any shifts. In logical shift left, the bits are shifted by one bit position to left and a zero is appended at the rightmost significant bit position. For logical shift right, the bits are shifted by one bit position to the right

and a zero is appended at the leftmost significant bit position. For arithmetic shift operations, the situation is different because of the sign bit. Since numbers are represented in two's complement format, the sign bit must be maintained for arithmetic shifts left or right. Hence, arithmetic shift left is similar to logical shift left except that the leftmost sign bit is left unchanged. Likewise, arithmetic shift right is similar to logical shift right except that the sign bit is propagated to the right by one bit position. Rotation to left or right is similar to logical shift left or right except that the most significant bit replaces the least significant bit or the least significant bit replaces the most significant bit respectively. Table 5.13 summarizes these shift operations where S is the shifter input and F is its output.

Table 5.13: Implementation of Different Shift Operations.

Shift operation	Notation
Logical shift left	$(F_n,F_{n-1},\ldots,F_2) \leftarrow (S_{n-1},S_{n-2},\ldots,S_1)$, $F_1 \leftarrow 0$
Logical shift right	$(F_{n-1},F_{n-2},\ldots,F_1) \leftarrow (S_n,S_{n-1},\ldots,S_2)$, $F_n \leftarrow 0$
Arithmetic shift left	$(F_n,F_{n-1},\ldots,F_2) \leftarrow (S_n,S_{n-2},\ldots,S_1)$, $F_1 \leftarrow 0$
Arithmetic shift right	$(F_{n-1},F_{n-2},\ldots,F_1) \leftarrow (S_n,S_{n-1},\ldots,S_2)$, $F_n \leftarrow S_n$
Rotate left	$(F_n,F_{n-1},\ldots,F_2) \leftarrow (S_{n-1},S_{n-2},\ldots,S_1)$, $F_1 \leftarrow S_n$
Rotate right	$(F_{n-1},F_{n-2},\ldots,F_1) \leftarrow (S_n,S_{n-1},\ldots,S_2)$, $F_n \leftarrow S_1$
Transfer	$(F_n,F_{n-1},\ldots,F_1) \leftarrow (S_n,S_{n-1},\ldots,S_1)$

Figure 5.24 shows a *4×4* barrel shifter that performs seven shift operations; i.e.; logical shift left, logical shift right, arithmetic shift left, arithmetic shift right, rotate left, rotate right, and transfer operations. Table 5.14 shows the control signal associated with each shift operation.

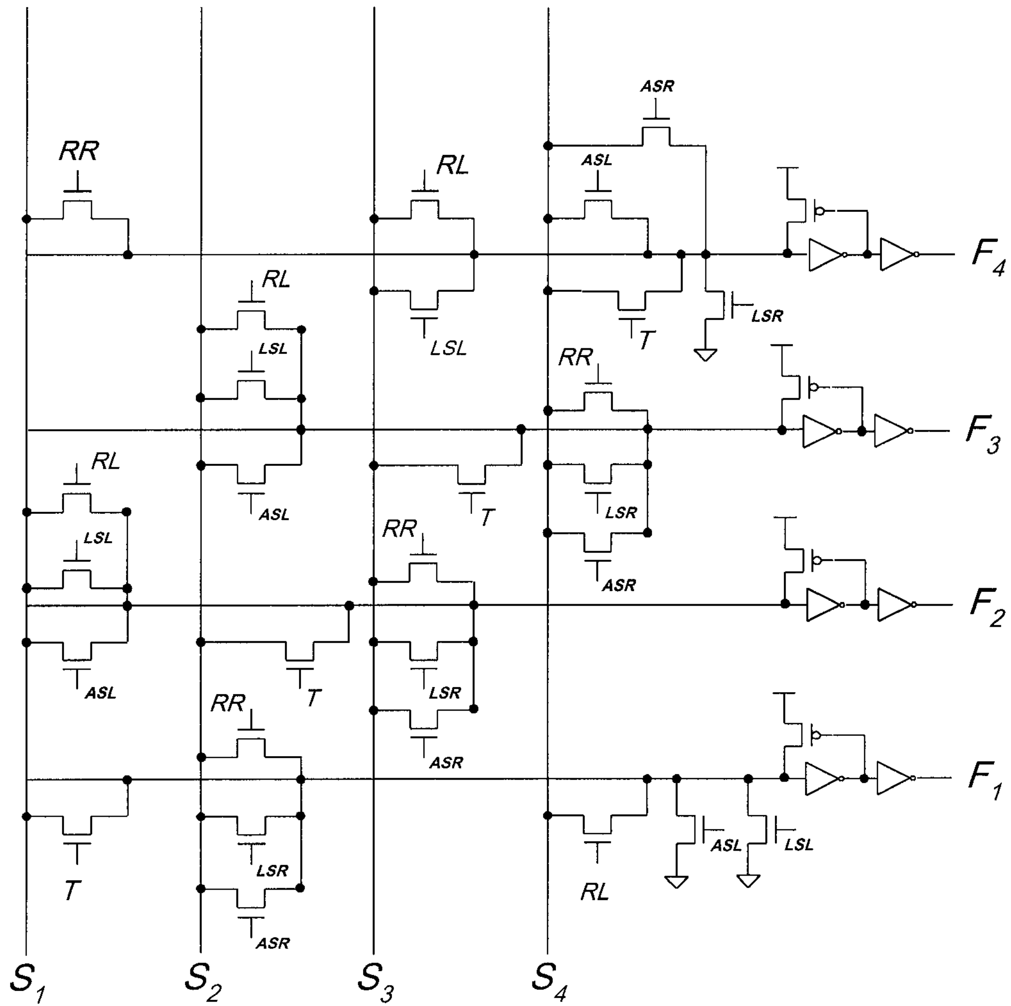


Figure 5.24: A 4X4 Barrel Shifter.

Table 5.14: The Control Signals of Shift Operations.

Shift operation	**Control signal**
Logical shift left	LSL
Logical shift right	LSR
Arithmetic shift left	ASL
Arithmetic shift right	ASR
Rotate left	RL
Rotate right	RR
Transfer	T

For the *4×4* barrel shifter shown in Figure 5.24, the input word is (S_1, S_2, S_3, S_4) and the output word is (F_1, F_2, F_3, F_4). During any of the shift operations listed in Table 5.14, only the corresponding control signal is activated while the others are disabled. The barrel shifter in Figure 5.24 can be further minimized into an equivalent barrel shifter as shown in Figure 5.25. Minimization is possible since parallel NMOS transistors connecting the same nodes can be replaced by a single NMOS transistor whose gate is driven by a signal which is the OR_ing of the signals driving the individual NMOS transistors.

Unlike the adder, it is not possible to build an n-bit shifter by cascading a number of smaller size shifters. From the barrel shifter shown in Figure 5.25, it is

possible to derive the general scheme of an n×n barrel shifter implementation. An n×n barrel shifter is an n×n matrix where each transistor location is referred to by its corresponding row and column indices. For example, the notation [NMOS(1,8)] means that there is NMOS transistor that connects input S_1 to output F_8 if its gate is enabled. Moreover, the notation [NMOS(GND,7)] means that there is NMOS transistor that discharges output F_7 if its gate is enabled. Table 5.15 summarizes the locations NMOS transistors for implementing an n×n barrel shifter and the control signals that drive their gates.

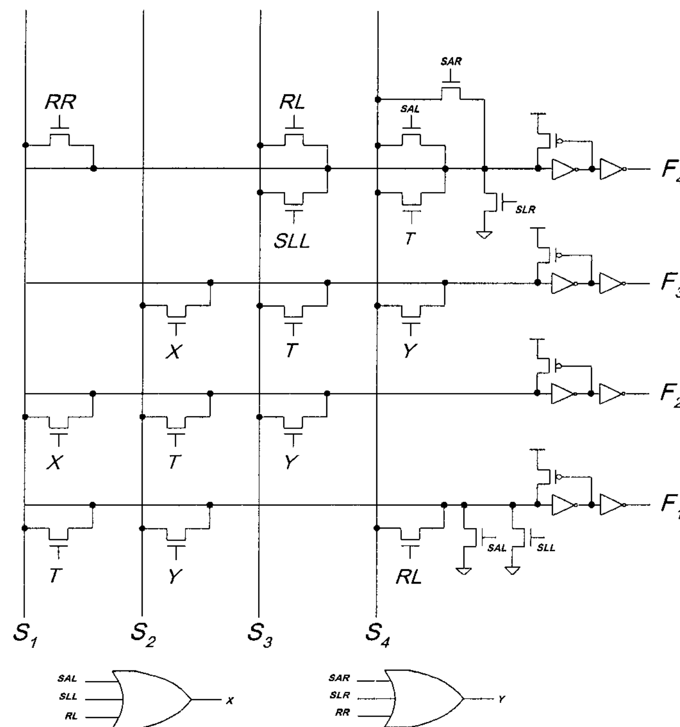


Figure 5.25: Minimization of the 4X4 Barrel Shifter.

Table 5.15: A General Scheme for Implementing nXn Barrel Shifter.

Transistor locations	**Gate driven by:**	**#of NMOS**
NMOS(1,2), NMOS(2,3), ……, NMOS(i,i+1), ………, NMOS(n-2,n-1).	OR-ing(ASL,LSL,RL)	n-2
NMOS(2,1), NMOS(3,2), ….., NMOS(i,i-1), ………, NMOS(n,n-1).	OR-ing(ASR,LSR,RR)	n-1
NMOS(1,1), NMOS(2,2), ……, NMOS(i,i), ………., NMOS(n-1,n-1).	T	n-1
NMOS(1,n).	RR	1
NMOS(n,1)	RL	1
NMOS(GND,1)	ASL	1
NMOS(GND,1)	LSL	1
NMOS(n-1,n).	RL	1
NMOS(n-1,n).	LSL	1
NMOS(n,n)	ASL	1
NMOS(n,n)	ASR	1
NMOS(n,n)	T	1
NMOS(GND,n)	LSR	1

5.4.6 The Overall Design

In this section, the overall design of the proposed self-timed ALU is described. The proposed self-timed ALU implements 16 different basic operations. These operations include addition, subtraction, increment, decrement, arithmetic complementation, AND, OR, XOR, XNOR, logical complement, logical shift left, logical shift right, arithmetic shift left, arithmetic shift right, rotate left, and rotate right. Thus, the op-code field of these operations consists of 4 bits. The op-code field is fed into a decoder where different control signals are set according to the operation to be implemented. Figure 5.26 illustrates a 4-bit block of the proposed self-timed ALU that can be used in building wider ALU. Values of the control signals generated by the decoder are given in Table 5.16.

The proposed self-timed ALU is assumed to be initially in the pre-charge mode where a request signal (REQ) is low and the FCS signal is low. When new operands become valid at the input of the ALU, the REQ signal is asserted high and the ALU goes into the evaluate phase after the P_i, K_i, and G_i signals are evaluated. Thus, the REQ signal passes through a delay circuit to generate the delayed request signal (REQD) as shown in Figure 5.27. This amount of the delay should match the delay required to generate P_i, K_i, and G_i signals (two NOR gates and one inverter), the delay required to pass the proper inputs at the ALU ports

(two MUX's and one inverter), and the delay required to set the control lines of the decoder for the ALU (assumed to be equivalent to two NAND gates).

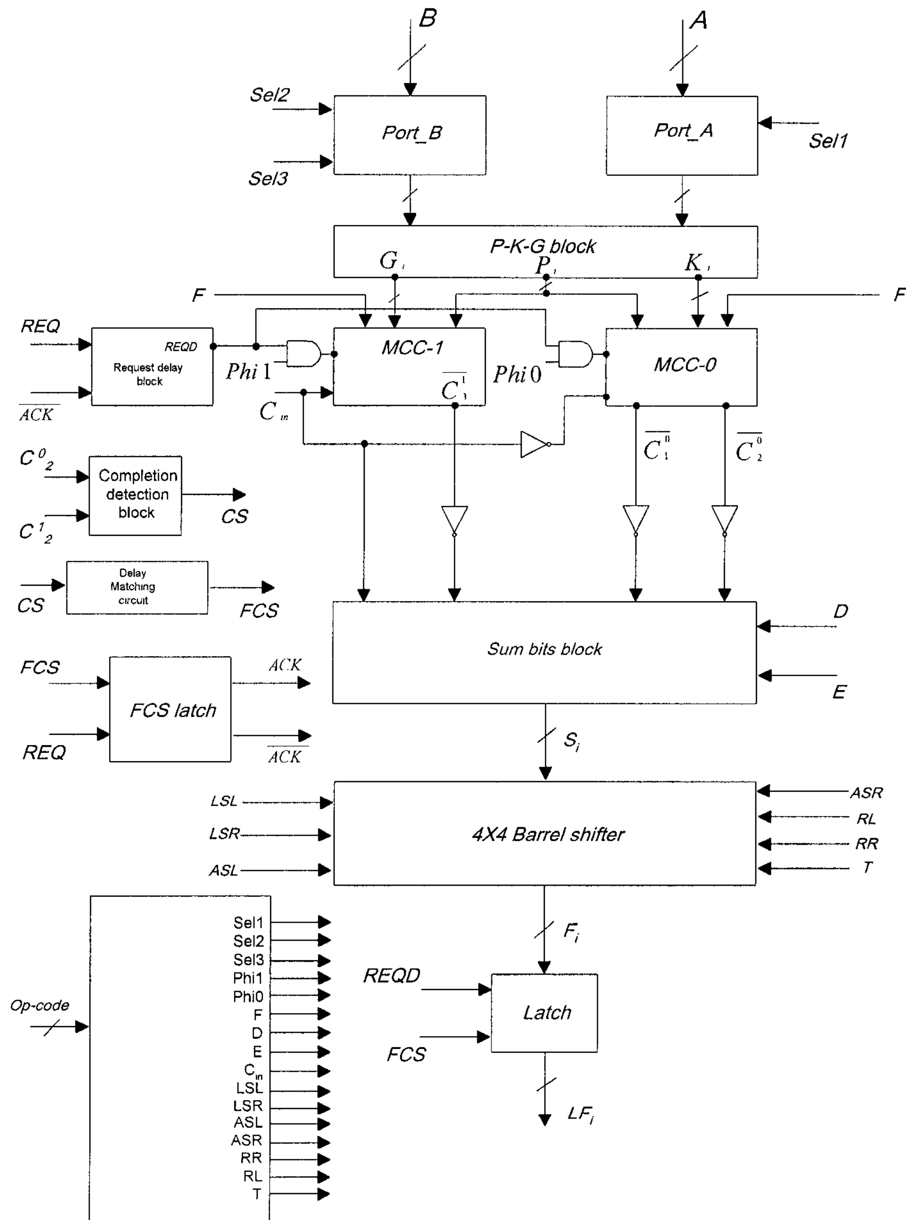


Figure 5.26: Block Diagram for the Proposed 4-bit Self-Timed ALU.

Table 5.16: Status of the Control Signals for all Operations.

OPERATION	Sel1	Sel2	Sel3	Phi1	Phi0	F	D	E	C_{in}	LSL	LSR	ASL	ASR	RL	RR	T
Addition	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
Subtraction	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1
Increment	1	1	0	1	1	0	0	0	1	0	0	0	0	0	0	1
Decrement	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1
Arithmetic Complement	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0	1
AND	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1
OR	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	1
XOR	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1
XNOR	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1
Logical Complement	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	1
Logical Shift Left	1	1	0	1	0	1	0	1	1	1	0	0	0	0	0	0
Logical Shift Right	1	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0
Arithmetic Shift Left	1	1	0	1	0	1	0	1	1	0	0	1	0	0	0	0
Arithmetic Shift Right	1	1	0	1	0	1	0	1	1	0	0	0	1	0	0	0
Rotate Left	1	1	0	1	0	1	0	1	1	0	0	0	0	1	0	0
Rotate Right	1	1	0	1	0	1	0	1	1	0	0	0	0	0	1	0

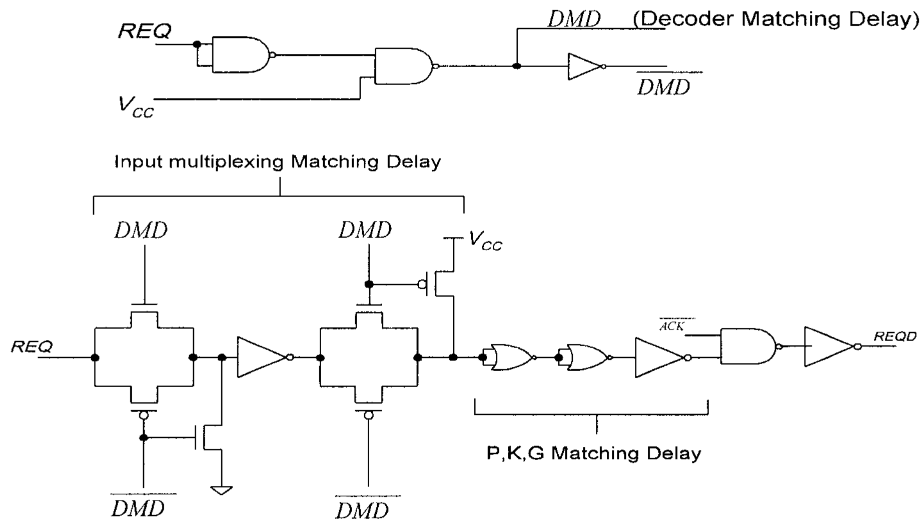


Figure 5.27: The Delayed Request Circuit.

During the evaluate phase, the carry nodes of MCC-1 and MCC-0 are evaluated to compute the result of an operation and finally the FCS signal turns high when computation is completed. To maintain the bundle data constraint for different process technologies, the CS should pass through a delay matching circuit to get the FCS. This delay is equivalent to the delay needed to compute the final result (Mux delay, XOR delay, Barrel shifter delay) as shown in Figure 5.28.

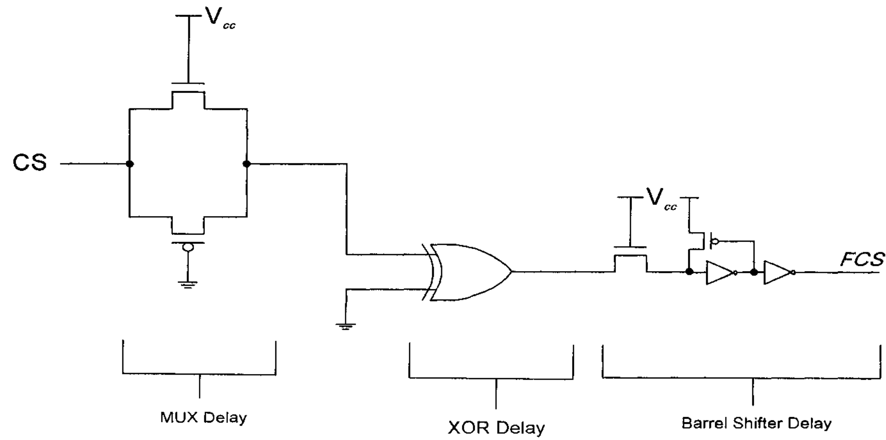


Figure 5.28: The Delay Matching Circuit to get the FCS.

When the FCS signal turns high, the result is valid and, hence, it and its complement are latched using the latch circuit shown in Figure 5.29. When the REQD signal is high and the ith data bit F_i is valid, the F_i is latched when the FCS is high.

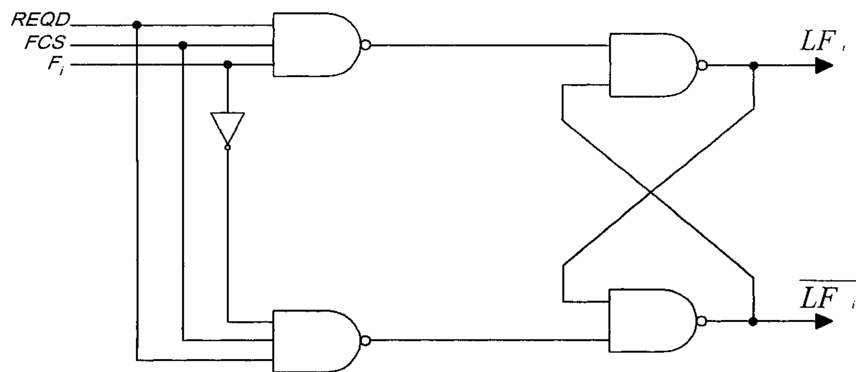


Figure 5.29: The Latch Circuit for the Result.

Figure 5.30 shows the latch circuit for latching the FCS. As soon as the FCS turns high, it is latched and known as ACK.

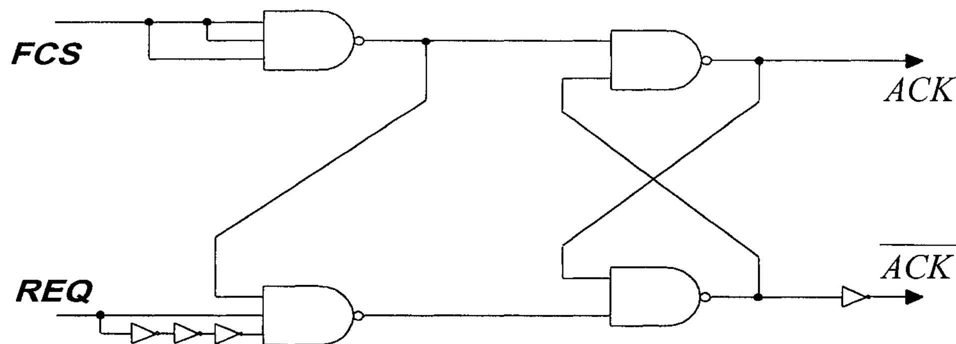


Figure 5.30: The Latch Circuit for the FCS.

When the ACK turns high, the REQD signal turns low and the ALU enters the pre-charge mode to be ready for the execution of the next operation. Thus, the ACK signal is initially set high. When the REQ signal turns high, the ACK signal turns low and the ALU is placed in the evaluate phase. Figure 5.31 shows the handshake protocol between the REQ and the ACK.

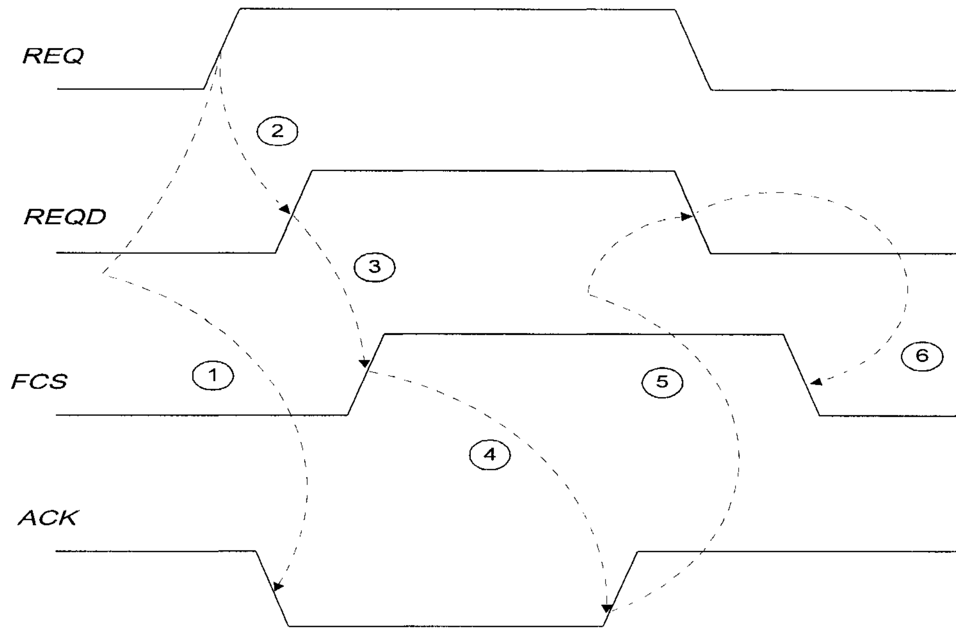


Figure 5.31: The Handshake Protocol between REQ and ACK.

The handshake protocol in Figure 5.31 works as follows:

1- The ACK is reset as soon as the REQ is received for another operation.

2- The Delayed request (REQD) is generated. This delay allows P_i, K_i, G_i to be evaluated before starting the evaluation phase.

3- The evaluation phase starts and the results are computed. When the result is valid the FCS is asserted and the result is latched.

4- The FCS is latched and causes the ACK to turn high.

5- The ACK resets the REQD to be ready for the pre-charge phase.

6- Pre-charge phase starts and resets the FCS to be ready for another operation.

5.4.7 Alternative Approach

In the proposed STALU, the results of arithmetic, logical, and shift operations are generated as the output of common XOR/XNOR gates through proper multiplexing at their inputs. Furthermore, the results of all operations pass through the barrel shifter before being latched. The operand multiplexing and the barrel shifter increase the overhead delay beyond what is needed for the required operation. For example, in arithmetic operations, two extra delay components are added to the delay required by the arithmetic operation; namely the delay of the 2x1 multiplexers before the XOR/XNOR gates and the barrel shifter delay.

Likewise, logical operations can be directly generated from the P-K-G circuits. Implementing them according the proposed STALU, however, will add four extra delay components on top of the delay required by the logical operation itself. These delays are; the delay of 2 NMOS pull down transistors of the MCC's, the delay of the 2x1 multiplexers, the delay of the XOR/XNOR gates, and the barrel shifter delay.

Similarly, shift operations can be implemented by passing the operand directly to the barrel shifter after a delay that matches the decoder delay. However, generating shift operations through the proposed STALU adds four extra delay

components; namely the delay of two 2x1 multiplexers at the input ports, the delay of 2 NMOS pull down transistors of the MCC's, the delay of 2x1 multiplexers, and the delay of XOR/XNOR gates.

Accordingly, an alternative approach would generate the results of the three types of operations through three different paths. This approach is discussed in this section and compared to the proposed STALU discussed earlier in terms of delay and hardware cost. Simulation results of both approaches will be given in Chapter 6 for comparison.

5.4.7.1 The Data Path

The data path of the alternative STALU design is shown in Figure 5.32 where three separate paths are used for each type of operation (arithmetic, logical, and shift). The delay path to generate arithmetic operations consists of the delay required to generate the REQD signal, the delay required to calculate the carry signals of MCC's, the delay required to generate the sum bits at the XOR/XNOR outputs, and the delay of a 2x1 multiplexer to pass the valid result to the data latch.

For logical operations, the delay path consists of the delay needed to generate the REQD signal, the delay of 5x1 multiplexer to select the required logical

operation, and the delay of two 2x1 multiplexers to pass the correct logical result to the data latch.

Considering shift operations, the delay path consists of the delay needed to generate the decoder signals (DMD), the barrel shifter delay, and two 2x1 multiplexers to pass the required shift result to the data latch.

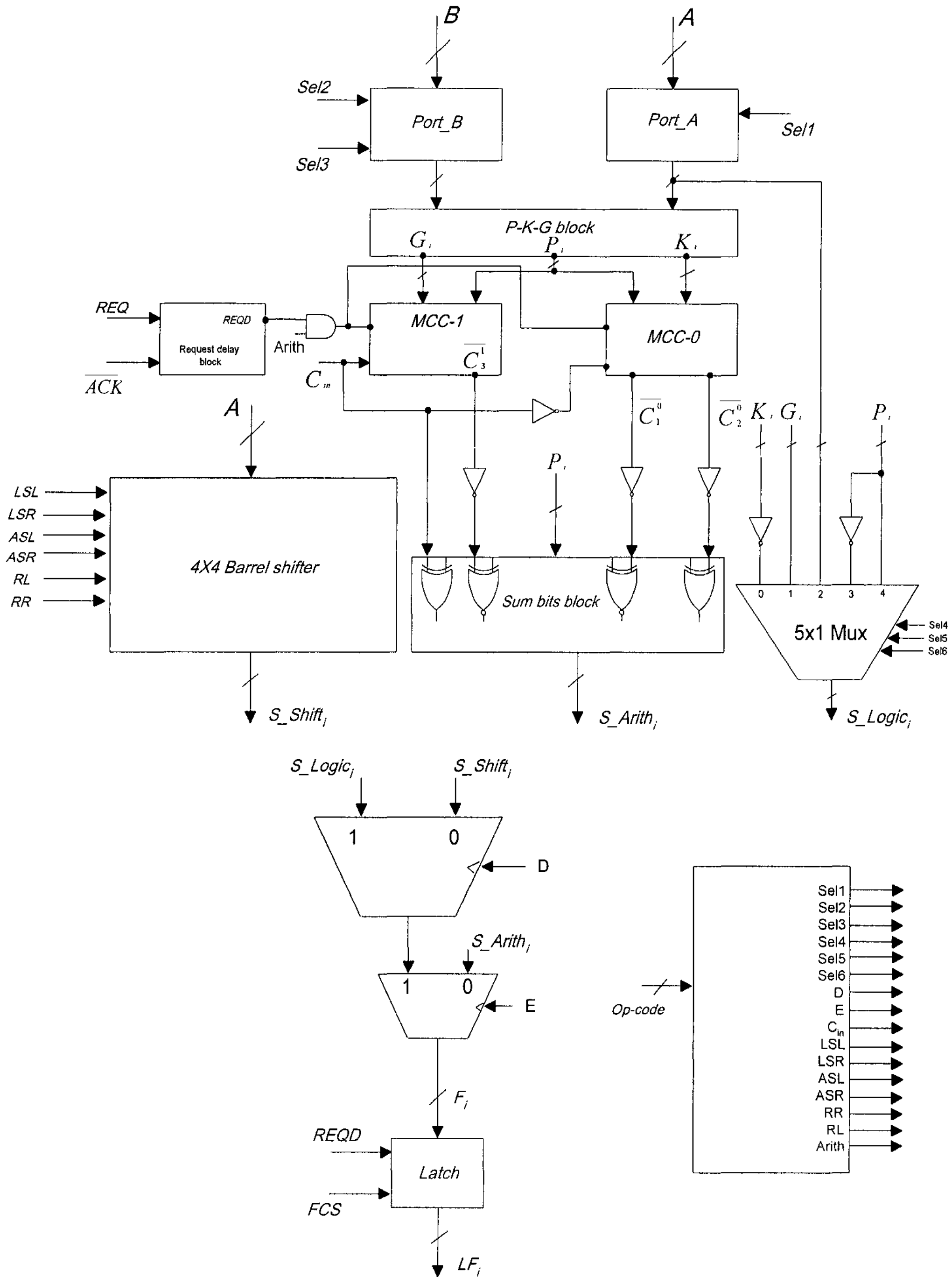


Figure 5.32: Data path of the Alternative Approach.

5.4.7.2 Completion Detection Circuitry

Since the alternative approach has three different paths for implementing the three types of operations, a separate completion signal needs to be generated for each type of operations. The combined STALU completion signal (FCS) would have to be generated from 3 signals. Figure 5.33 shows how the FCS for the alternative STALU is generated from three different signals; namely FCS_Arith, FCS_Logic, and FCS_Shift. The three signals FCS_Arith, FCS_Logic, and FCS_Shift signal the validity of results for Arithmetic, Logical, and Shift operations respectively.

The FCS_Arith is generated from a circuit similar to that generating the FCS of the previous design where the CS signal is passed through an XOR gate to match the delay that generates the sum bits. The REQD signal is used to generate the FCS_Logic through a series of 3 NMOS transistors to match the delay of the 5x1 multiplexer used to select the required logical operation. The DMD signal is used to generate the FCS_Shift through an NMOS transistor that matches the delay of the barrel shifter. Finally, one of these signals (FCS_Arith, FCS_Logic, FCS_Shift) is selected to be latched as ACK through two 2x1 multiplexers which are equivalent to the delay of the two 2x1 multiplexers on the data path used to select the proper output (S_arith_i, S_Logic_i, S_Shift_i) to be latched .

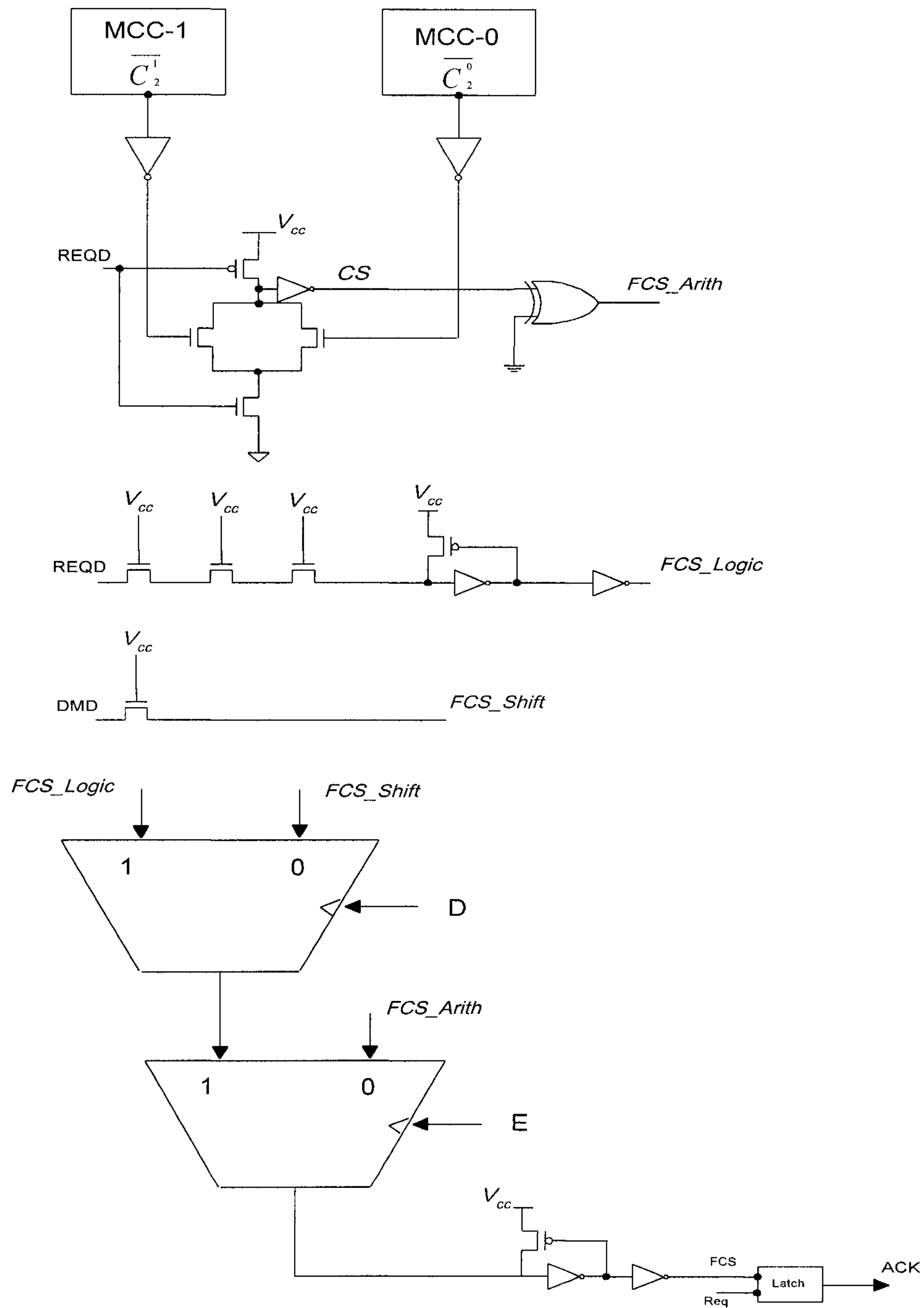


Figure 5.33: Completion Detection Circuitry of the Alternative Approach.

5.4.7.3 Comparison

The alternative approach is compared to the proposed one in terms of delay and hardware cost. For arithmetic operations, Table 5.17 shows the delay components saved versus those added by the alternative approach data path as compared to the proposed STALU. It shows that the alternative approach data path is faster than the proposed one since it saves the delay of the barrel shifter.

Table 5.17: Saved and Added Delay Components for Arithmetic Operations.

Saved components	Added components
• Delay of 2x1 multiplexer before the XOR/XNOR stage. • The barrel shifter delay.	• Delay of 2x1 multiplexer to pass the arithmetic result to the data latch.

For logical operations, Table 5.18 shows the delay components saved and added by this approach as compared to the previous approach.

Table 5.18: Saved and Added Delay Components for Logical Operations.

Saved components	Added components
• Delay of two pull down NMOS transistors of the MCC. • Delay of 2x1 multiplexer before the XOR/XNOR stage. • Delay of XOR/XNOR stage. • The barrel shifter delay.	• Delay of 5x1 multiplexer to select the required logical operation. • Delay of two 2x1 multiplexers to pass the result to the data latch.

Again, the alternative approach data path is faster than the proposed approach in generating the logical results.

For shift operations, Table 5.19 shows the delay components saved and added by this approach as compared to the proposed approach. It shows that the alternative approach data path is faster than the proposed design in generating the shift results.

Table 5.19: Saved and Added Delay Components for Shift Operations.

Saved components	Added components
• Delay of two 2x1 multiplexers at the input ports. • Delay needed to calculate the propagate, generate, and kill signals. • Delay of two pull down NMOS transistors of the MCC. • Delay of 2x1 multiplexer before the XOR/XNOR stage. • Delay of XOR/XNOR stage.	• Delay of two 2x1 multiplexers to pass the result to the data latch.

In terms of hardware cost, the data path of both approaches have the same components except that the alternative approach requires additional four 5x1 multiplexers for every block of 4-bits to select the 4-bit result of the required

logical operation. Moreover, the completion detection circuitry of the alternative approach uses two more 2x1 multiplexers required to select one of the three final completion signals (FCS_Arith, FCS_Logic, FCS_Shift) as an ACK signal.

When the alternative approach is compared to the proposed one, simulation results show that the speed improvement is not exceeding 2% for arithmetic and logical operations while it is 44% for shift operations. Details of these results are given in Chapter 6.

While the alternative approach executes simple instructions faster, the proposed approach will execute more complex instructions a lot faster. For example, an add-shift instruction which is frequently used in multiplication and division operations can be performed in one cycle using the proposed approach compared to two cycles in the second. This will cause the overall performance of the proposed approach to generally be higher than that of the second particularly for more complex instructions. Moreover, the hardware cost for the proposed approach is less than that of the second. Accordingly, the proposed approach is selected and all simulation results and analysis are based on it.

CHAPTER 6

DESIGN VERIFICATION AND ANALYSIS

One of the major potential advantages of self-timed (asynchronous) design is the promise of high performance. To achieve high performance, one must design a fast self-timed circuit with good average case performance and a fast completion detection circuit. The proposed self-timed ALU satisfies the above two conditions since the Manchester Carry Chain Adder with two carry chains is the major unit in the proposed ALU where the average speed of this type of adders is known to be $O(\log n)$ where n is the adder width. Moreover, the completion detection circuit of the proposed ALU is fairly efficient and simple as compared to others since the middle stage of every block is only used to generate the final completion detection signal (FCS).

In this chapter, delay analysis is performed in order to decide the minimum length of the proposed ALU that can operate correctly. SPICE simulation results for 0.8μ and 0.18μ CMOS process technologies are discussed. Finally, the proposed completion detection circuit is compared to other existing completion detection schemes in terms of delay and area overhead.

6.1 Delay Analysis

In this section, delay analysis is performed on the proposed ALU to guarantee its proper operation, i.e bundling constraint and, hence, latching the FCS (ACK) after latching the correct result. At this time, let us define a time parameter called *acknowledge slack* **(T_{slk})** as the period from the time output data are computed to the time the FCS signal is asserted (**T_{slk}** = time FCS is asserted – time output data are valid). For proper operation, the bundled data constraint **($T_{slk} > 0$)** must be maintained.

The delay analysis of the proposed ALU is performed under various conditions of voltages and temperatures using SPICE to verify the bundled data constraint under the following two conditions:

1- Operation under worst case delay condition. This occurs when all carry propagates P_i's are 1's. This condition is referred to as *worst case delay.*

2- Operation under worst case functionality condition. Here, operands are chosen to achieve the fastest possible generation of the final completion signal and the slowest output result. This condition occurs when all propagate signals of the consecutive 4-bit blocks are 1's except the final middle stage that has true carry generate (G_{n-2}=1) or carry kill (K_{n-2}=1) condition. This condition is referred to as the *worst case functionality.*

6.1.1 Worst Case Delay (WCD)

The worst case delay occurs when all the propagate signals P_i's are ones. Thus, for an n-bit ALU where (n/4) 4-bit blocks of MCC are cascaded, the slowest sum bit is S_n. The final completion signal (FCS) needs to be checked for the bundled data constraint versus S_n. Thus, for proper operation, the delay path of the FCS has to be verified to be longer than the delay path of S_n.

Figure 6.1 illustrates a number of cascaded 4-bit MCC blocks. When all the propagate signals P_i are 1's, all the carry nodes follow the longest discharge path. After reaching the last MCC block, the carry nodes of this block are discharged low starting by the carry node $\overline{C_{n-4}}$ then $\overline{C_{n-3}}$ and so on. By discharging the carry node $\overline{C_{n-2}}$, the final completion signal for all blocks can be evaluated through the completion detection circuitry. The carry node $\overline{C_{n-1}}$ is used to generate the sum bit S_n. Thus, the discharge delay difference between the two carry nodes $\overline{C_{n-2}}$ and $\overline{C_{n-1}}$ is only one NMOS pass transistor. For proper operation, the completion detection circuitry should compensate for a delay more than one NMOS transistor. Referring to Figure 6.2, the delay through the completion detection circuitry for a 4-bit ALU is a stack of two pull down NMOS

transistors and an inverter. This delay is more than enough to guarantee valid operation for the proposed ALU of 4-bit even for different fabrication processes.

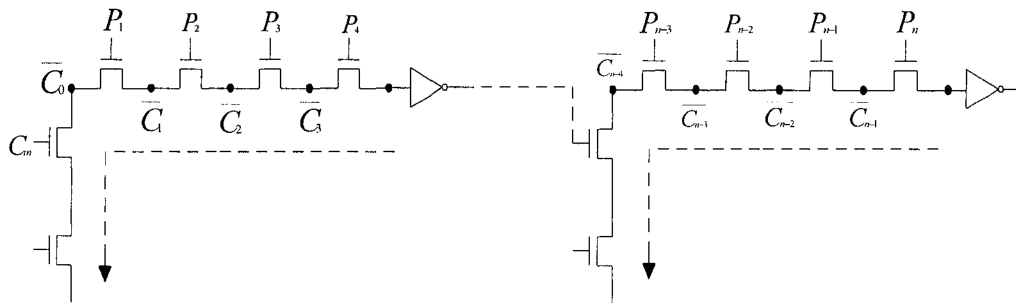


Figure 6.1: Cascading a number of 4-bit MCC blocks.

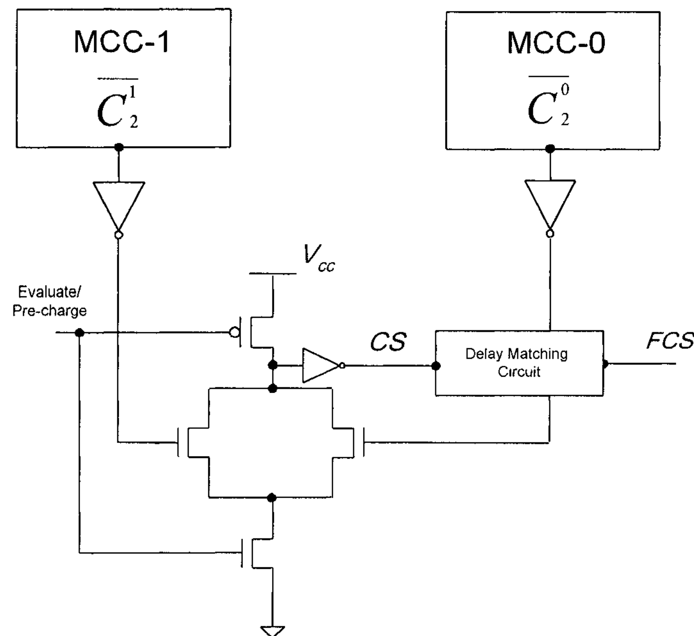


Figure 6.2: Completion Detection Circuitry for a 4-bit ALU.

6.1.2 Worst Case Functionality (WCF)

Figure 6.3 shows two consecutive 4-bit MCC blocks for an 8-bit ALU. Let the middle carry node $\overline{C_i}$ be discharged low either through the two pull down NMOS transistors or through the pass NMOS transistor. Let the middle carry node $\overline{C_{i+4}}$ be discharged low through the two pull down NMOS transistors. Let the carry propagate signals P_i, P_{i+1}, and P_{i+2} to be ones and P_{i+3} to be zero. When the carry node $\overline{C_i}$ is discharged low, then the carry nodes $\overline{C_{i+1}}$, $\overline{C_{i+2}}$, and $\overline{C_{i+3}}$ are discharged low. Thus the difference in delay discharge path between the carry nodes $\overline{C_i}$ and $\overline{C_{i+3}}$ is the delay of two NMOS pass transistors, an inverter, and a stack of three NMOS pull down transistors. Therefore, the delay through the completion detection circuitry should be greater than this delay for proper functionality. Figure 6.4 shows the completion detection circuitry for an 8-bit ALU. The delay through this circuitry to generate the FCS is a stack of 5 NMOS pull down transistors and an inverter. For correct functionality for an 8-bit ALU, this delay should be more than the delay through two NMOS pass transistors, an inverter, and a stack of three NMOS pull down transistors. This condition has to be validated through SPICE simulation.

The above argument is also applicable for a 16-bit, 32-bit, and 64-bit ALU. The completion detection circuitry for a 16-bit ALU is similar to that for an 8-bit ALU. The bundled data constraint for a 32, 64-bit ALU is more relaxed than that for an 8-bit ALU since the delay needed to evaluate their FCS's is longer than that for an 8-bit ALU.

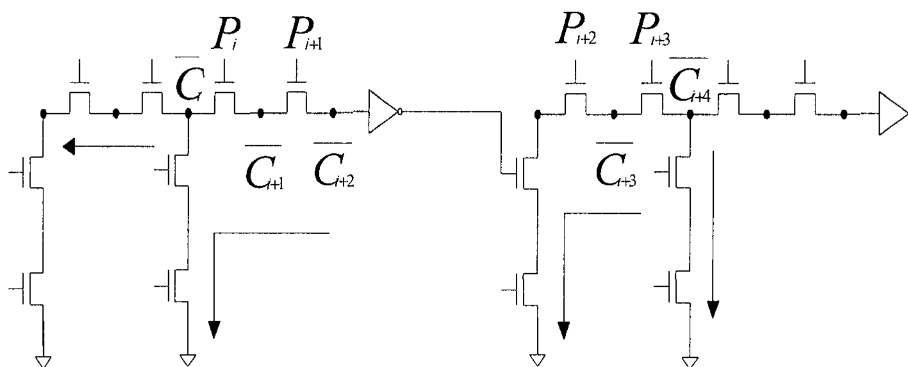


Figure 6.3: Worst case functionality.

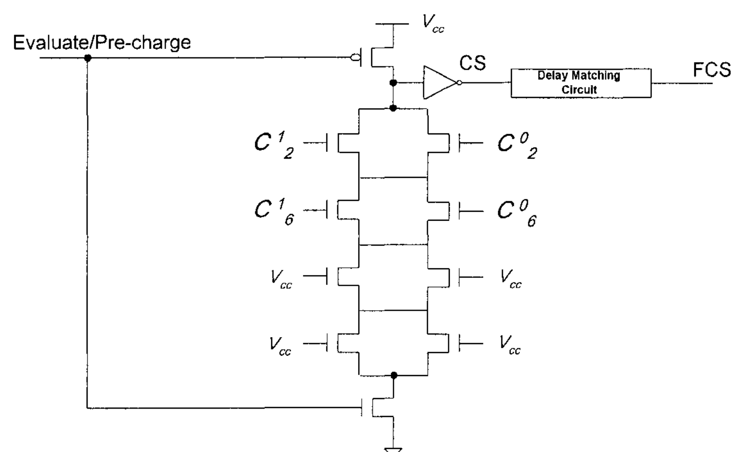


Figure 6.4: Completion Detection Circuitry for an 8-bit ALU.

6.2 Simulation Results

Firstly, the self-timed adder version of the proposed design has been SPICE simulated for 4, 8, 16, 32-bit adder size using a 0.8μ CMOS technology to verify the bundled data constrain. Secondly, the proposed self-timed ALU has been SPICE simulated for the worst case delay and the worst case functionality using a 0.8μ CMOS technology. A 16-bit self-timed ALU has been simulated under several conditions of voltage and temperature. Simulation results for 27°C and Vcc=5V, 125°C and 4V, and -55°C and 6V will be discussed. Thirdly, the design has also been simulated using a 0.18μ process technology. Fourthly, simulation results for the alternative approach are reported for comparison with results of the proposed design. Finally, the proposed self-timing scheme is compared with other existing self-timing schemes in terms of delay and area overhead.

6.2.1 Simulation Results for Self-Timed Adder

The proposed self-timed adder is SPICE simulated for different adder sizes 4, 8, 16, 32-bit using a 0.8μ CMOS technology under normal operating condition (27°C, 5V). SPICE simulations have shown that adders with sizes > 8-bit work robustly satisfying the bundled data constraint ($T_{slk} > 0$). For the 4-bit (8-bit) adder, however, the bundled data constraint was marginally passed under the WCD

(WCF) condition with the FCS asserted almost simultaneously ($T_{slk} = 0$) with the slowest sum bit. Table 6.1 summarizes these results.

Table 6.1: Simulation Results for N-bit Adder.

Adder Size	T_{slk}	Condition
4-bit	0.0 ns	WCD
8-bit	0.0 ns	WCF
16-bit	0.1 ns	WCF
32-bit	0.4 ns	WCF

6.2.2 Simulation Results for the Self-Timed ALU

A 16-bit self-timed ALU is SPICE simulated for critical cases under different operating conditions using a 0.8μ CMOS technology.

6.2.2.1 Simulation Results under Normal Conditions

Here, a 16-bit self timed ALU is simulated under normal operating conditions where the temperature is 27°C and the power supply voltage is 5 volts. Since the addition operation is the most important among other operations, it will be simulated under both the worst case delay and the worst case functionality

conditions. Then, simulations for other operations, e.g. logical and shift operations are carried out for analysis.

6.2.2.1.1 Arithmetic Operations

Three different cases have been simulated. These mainly model the worst case delay and the worst case functionality. The worst case delay occurs when all the carry propagate signals (P_i) are 1's where all carries have to propagate through all the stages of either MCC-1 or MCC-0. The worst case functionality occurs when either all the carry kills K_i or all the carry generates G_i of the middle points of the cascaded MCC-0's or MCC-1's are 1's while the other carry propagates P_i are 1's.

CASE 1: (worst case delay)

The operands used here are $A=(AAAA)_{16}$, operand $B=(5555)_{16}$, and $C_{in}=1$. In this case all carry propagates P_i are ones and the initial carry C_{in} propagates through all the stages of MCC-1, since the $C_{in}=1$, which results in the worst case delay. Initially, all carry nodes are pre-charged high. When the REQUEST signal is asserted high, after some delay, the delayed request signal (REQD) is also asserted high where the evaluation process starts. The carry nodes of MCC-1 are discharged low and the sum bits $S=(0000)_{16}$ are generated and

passed through the barrel shifter where the valid sum bits are denoted as $F=(0000)_{16}$. The final completion signal is evaluated and asserted high signalling the validity of the data. Finally, the valid data and the final completion signal are latched as $LF=(0000)_{16}$ and ACK respectively. When the ACK signal is asserted high, the delayed request signal (REQD) is reset low and the carry nodes of MCC-1 and MCC-0 are placed in the pre-charge mode to be ready for another operation. Thus, each operation passes through two phases, namely, the evaluation phase and the pre-charge phase. The delay from the REQUEST signal asserted high to the time of latching the final completion signal (ACK) is the evaluation phase delay. However, the delay from the ACK signal asserted high to the time the FCS discharge low is the pre-charge phase delay.

For this example, the simulation results are plotted for the major signals, namely, REQUEST, REQD, the final sum bit F(16), the final completion signal, the latched data bit LF(16) of F(16), and the ACK signal as shown in Figures 6.5 – 6.10 respectively.

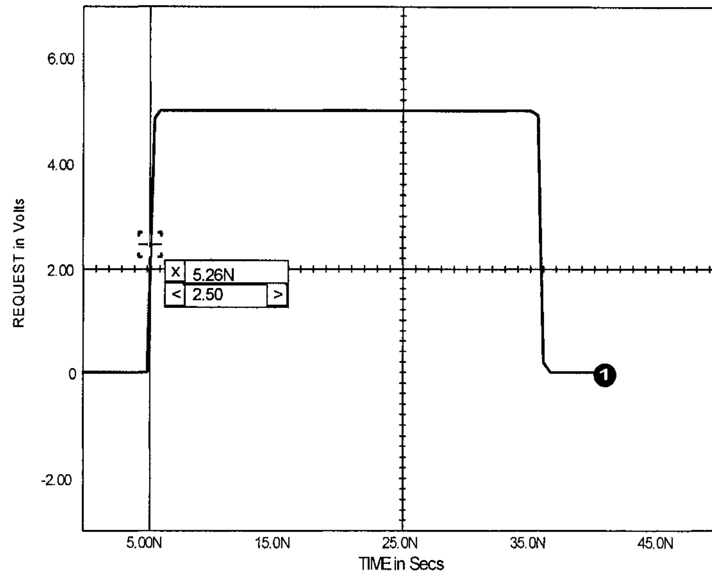


Figure 6.5: REQUEST signal VS Time.

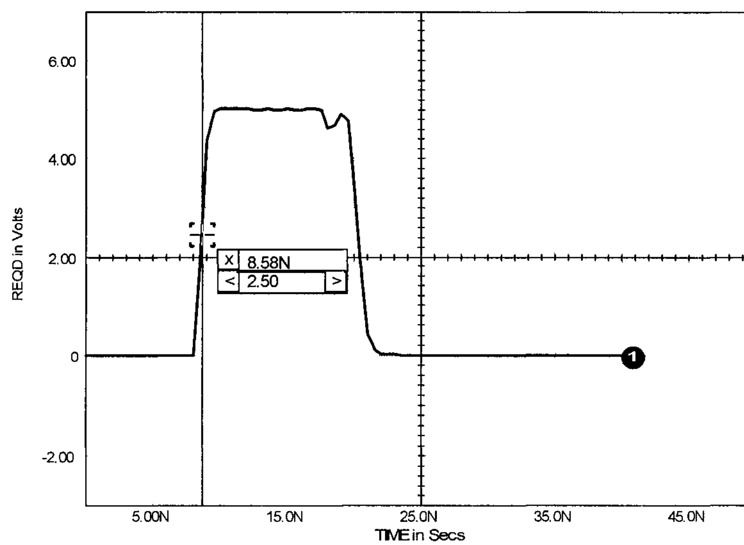


Figure 6.6: Delayed Request (REQD) VS Time.

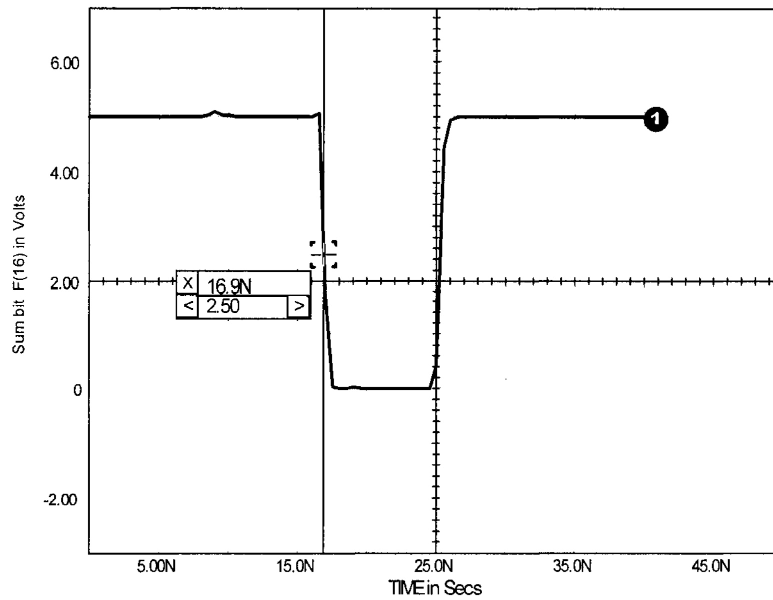


Figure 6.7: Sum bit F(16) VS Time.

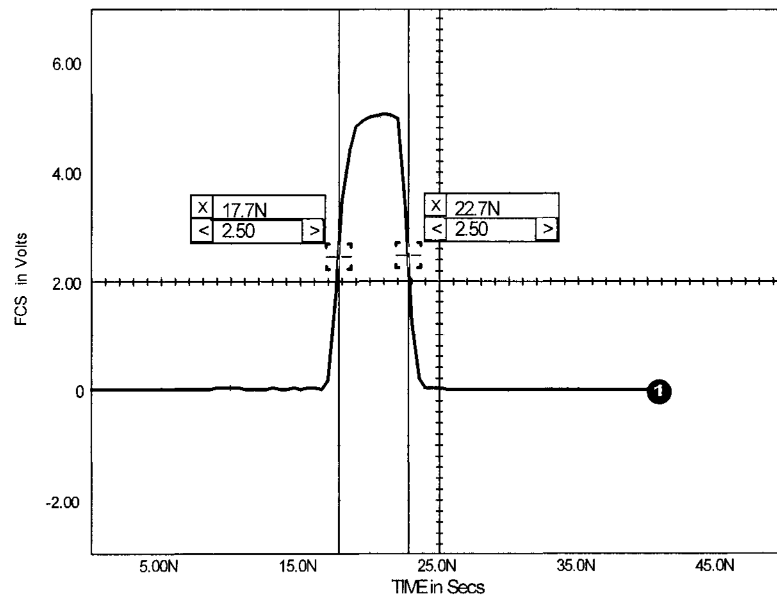


Figure 6.8: FCS VS Time.

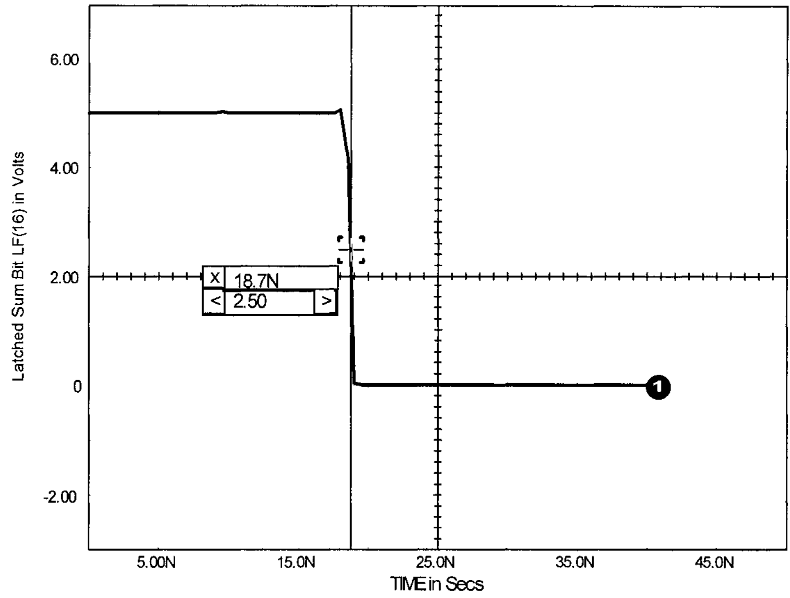


Figure 6.9: Latched Sum Bit LF(16) VS Time.

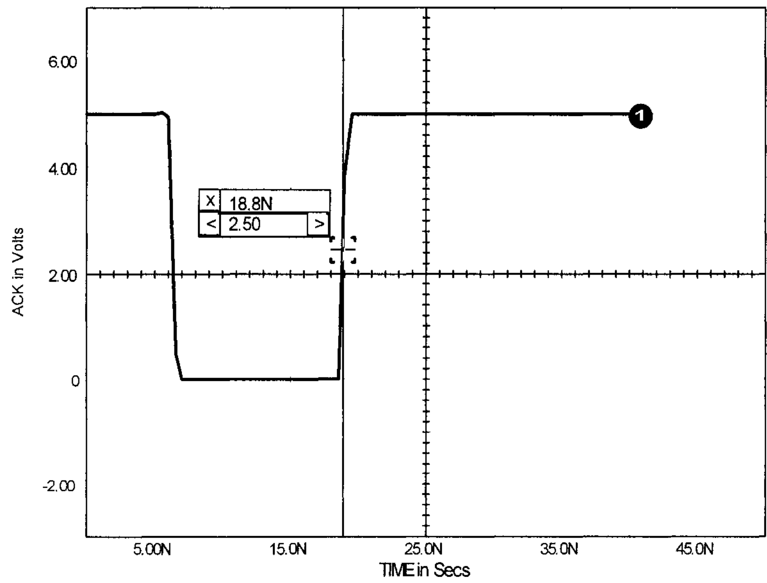


Figure 6.10: ACK VS Time.

From the above plots, it is clear that the final completion signal is asserted high to indicate the validity of the sum bits after the slowest sum bit F(16) is valid. Likewise, the ACK signal is asserted high after the result bits are latched. The evaluation delay time is 13.54 NS while the precharge delay time is 3.9 NS. Thus, the over all cycle time for this scenario is 17.44 NS.

CASE 2 : (worst case delay)

In this case, the above scenario is repeated with the difference that the initial carry in C_{in} is set low. Accordingly, the complement of the C_{in} propagates through all the stages of the cascaded MCC-0's which again results in worst case delay. For this example, the simulation results for the major signals, namely, REQD, the slowest sum bit F(15), the final completion signal, the latched data bit LF(15) of F(15), and the ACK signal are summarized in Table 6.2.

Table 6.2: Delay of Major Signals (Case 2).

Signal	Request	REQD	F(15)	FCS	LF(15)	ACK
Delay (NS)	5.26	8.58	16.2	17.4 (High) 22.4 (Low)	18.1	18.7

From the above results, it is clear that the final completion signal is asserted high to indicate the validity of the result bits after the slowest sum bit F(15) is valid. Likewise, the ACK signal is valid after latching the result bits. The evaluation delay time is 13.44 NS while the precharge delay time is 3.7 NS. Thus, the over all cycle time for this scenario is 17.14 NS.

CASE 3 : (worst case functionality)

This case simulates the scenario of worst case functionality with operands $A=(5555)_{16}$, operand $B=(8888)_{16}$, and $C_{in}=0$. The result of this operation is $F=(DDDD)_{16}$. The middle nodes of the cascaded MCC-0's are discharged low through the short path since the corresponding carry kills (K_i's) are ones. These carry kills are propagated through the stages of MCC-0's since the carry propagates (P_i's) for other nodes are 1's. In this scenario, the final completion signal is generated via the fastest path while the slowest sum bit F(14) is generated via the slowest possible path. For this example, the simulation results for the major signals, namely, REQD, the slowest sum bit F(14), the final completion signal, the latched data bit LF(14) of F(14), and the ACK signal are summarized at Table 6.3.

Table 6.3: Delay of Major Signals (Case 3).

Signal	Request	REQD	F(14)	FCS	LF(14)	ACK
Delay (NS)	5.26	8.58	12.2	13.0 (High) 18.0 (Low)	13.8	14.3

From the above results, it is clear that the final completion signal is asserted high to indicate the validity of the result bits after the slowest sum bit F(14) becomes valid. Likewise, the ACK signal is valid after latching the result bits. The evaluation delay time is 9.04 NS while the precharge delay time is 3.7 NS. Thus, the over all cycle time for this scenario is 12.74 NS.

6.2.2.1.2 Logical and Shift Operations

Logical and shift operations have fixed propagation delays since there is no carry propagation through the MCC-0's nor the MCC-1's. During the evaluate phase, the carry nodes of MCC-1's (MCC-0's) are forced low in order to activate the completion signal. The logical and shift operations may be divided into two groups. The first group consists of these operations that force the carry nodes of MCC-0 low while the second group consists of these operations that force the carry nodes of MCC-1 low. One case study for each group is demonstrated via a simulation example. The results of each example are applicable to other operations belonging to the same group.

CASE 1: (carry nodes of MCC-1 are forced low)

The operations OR, XNOR, LOGICAL COMPLEMENT, LSL, LSR, ASL, ASR, RL, and RR are similar in terms of having the same delay since, during the evaluation phase, carry nodes of MCC-1 are forced low while carry nodes of MCC-0 are forced high. Therefore, the delay simulation results of one operation are applicable to other similar operations. The XNOR operation is simulated with operands $A=(AAAA)_{16}$,$B=(AAF0)_{16}$, and the simulation results for major signals are summarized in Table 6.4.

Table 6.4: XNOR operation Signal Delays (Case 1).

Signal	Request	REQD	F(16)	FCS	LF(16)	ACK
Delay (NS)	5.26	8.58	11.7	13.0 (High) 18.0 (Low)	13.6	14.3

It is clear that the bundled data constraint is maintained when final completion signal is asserted. Accordingly, the ACK signal is valid after latching the result bits. Since there is no carry propagation through the MCC-0's or MCC-1's, all the result data bits are available at the same time. The evaluation delay time is 9.04 NS while the precharge delay time is 3.7 NS. Thus, the overall delay time

for this scenario is 12.74 NS and it is the same for all of the above mentioned operations.

CASE 2: (carry nodes of MCC-0 are forced low)

The operations AND, and XOR are similar in terms of having the same delay since, during the evaluation phase, carry nodes of MCC-1 are forced high while carry nodes of MCC-0 are forced low. Therefore, the simulation results of one operation are applicable to the other operation. The XOR operation has been simulated with operands $A=(AAAA)_{16}$, $B=(AAF0)_{16}$ and the simulation results for major signals are summarized in Table 6.5.

Table 6.5: XOR operation Signal Delays (Case 2).

Signal	Request	REQD	F(15)	FCS	LF(15)	ACK
Delay (NS)	5.26	8.58	11.4	12.9 (High) 17.9 (Low)	13.8	14.2

It is clear that the bundled data constraint is maintained when final completion signal is asserted. Likewise, the ACK signal is asserted after latching the result bits. Since there is no carry propagation through the MCC-0's or MCC-1's, all the result bits are available at the same time. The evaluation delay time is

8.94 NS while the precharge delay time is 3.7 NS. Thus, the over all cycle time for this scenario is 12.64 NS and it is the same for all of the above mentioned operations.

6.2.2.2 Simulation Results under Other Conditions

In this section, the proposed self timed ALU is simulated for a size of 16-bits using SPICE under extreme operating conditions of voltage and temperature (corner points) such as (4V, 125°C) and (6V, -55°C). All of the above cases (arithmetic, logical and shift operations) have been verified under these corner operating conditions. The simulation results are examined for the bundling constraint where the final completion signal must be asserted after the slowest data bit becomes valid. Tables 6.6 and 6.7 show the delay time of the slowest data bit, the latched data bit, the final completion signal, and the acknowledge (ACK) signal for operating conditions (4V, 125°C) and (6V, -55°C) respectively.

Table 6.6: Delay time of important signals under (4V, 125°C) condition.

Operation	Scenario	Delay of the slowest data bit	Delay of the latched slowest data bit	Delay of final completion signal	Delay of ACK signal
Arithmetic	Case 1	21.04NS	24.04NS	22.34NS	24.54NS
	Case 2	19.74NS	23.04NS	21.94NS	24.04NS
	Case 3	12.34NS	15.34NS	13.64NS	15.94NS
Logical and shift	Case 1	11.24NS	14.44NS	13.34NS	15.54NS
	Case 2	11.14NS	14.54NS	13.54NS	15.64NS

Table 6.7: Delay time of important signals under (6V, -55°C) condition.

Operation	Scenario	Delay of the slowest data bit	Delay of the latched slowest data bit	Delay of final completion signal	Delay of ACK signal
Arithmetic	Case 1	6.54NS	7.24NS	6.94NS	7.54NS
	Case 2	5.94NS	7.04NS	6.74NS	7.54NS
	Case 3	3.96NS	4.94NS	4.33NS	5.04NS
Logical and shift	Case 1	3.94NS	4.94NS	4.61NS	5.34NS
	Case 2	3.75NS	5.04NS	4.59NS	5.34NS

Tables 6.8 and 6.9 summarize the simulation results for operating conditions (4V, 125°C) and (6V, -55°C) respectively.

Table 6.8: Simulation Results for (4V, 125°C) Operating Conditions.

Operation	Scenario	Evaluate Time	Pre-charge Time	Over all Time
Arithmetic	Case 1	24.54NS	7.0NS	31.54NS
	Case 2	24.04NS	7.0NS	31.04NS
	Case 3	15.94NS	6.9NS	22.84NS
Logical and Shift	Case 1	15.54NS	6.9NS	22.44NS
	Case 2	15.64NS	7.0NS	22.64NS

Table 6.9: Simulation Results for (6V, -55°C) Operating Conditions.

Operation	Scenario	Evaluate Time	Pre-charge Time	Over all Time
Arithmetic	Case 1	7.54NS	2.1NS	9.64NS
	Case 2	7.54NS	2.0NS	9.54NS
	Case 3	5.04NS	2.0NS	7.04NS
Logical and Shift	Case 1	5.34NS	2.2NS	7.54NS
	Case 2	5.34NS	2.2NS	7.54NS

From the above tables, it is clear that the bundled data constraint is met under different operating conditions.

6.2.2.3 Simulation Result for Different Process Technologies

Simulation results of the previous scenarios for the proposed 16-bit ALU using a 0.18μ CMOS technology and the same channel width used for a 0.8μ CMOS technology at (1.8V, 27°C) are shown in Table 6.10. The objective of this part is to show that the proposed design is robust and process independent. The results shown in Table 6.10 justify our design and show that it is correctly functional even for different process technology. Table 6.11 summarizes the simulation results for the 16-bit self-timed ALU for this process technology.

Table 6.10: Delay time of important signals for 0.18μ CMOS at (1.8V, 27°C).

Operation	Scenario	Delay of the slowest data bit	Delay of the latched slowest data bit	Delay of final completion signal	Delay of ACK signal
Arithmetic	Case 1	8.04NS	9.04NS	8.54NS	9.54NS
	Case 2	7.44NS	8.94NS	8.44NS	9.44NS
	Case 3	4.45NS	5.54NS	4.94NS	5.94NS
Logical and shift	Case 1	3.99NS	5.34NS	4.84NS	5.74NS
	Case 2	3.89NS	5.44NS	4.84NS	5.94NS

Table 6.11: Simulation Results for 0.18μ CMOS at (1.8V, 27°C).

Operation	Scenario	Evaluate Time	Pre-charge Time	Over all Time
Arithmetic	Case 1	9.54NS	2.2NS	11.74NS
	Case 2	9.44NS	2.1NS	11.54NS
	Case 3	5.94NS	2.1NS	8.04NS
Logical and Shift	Case 1	5.74NS	2.2NS	7.94NS
	Case 2	5.94NS	2.1NS	8.04NS

6.2.3 Comparison with the Alternative Approach

As discussed in Chapter 5, the alternative approach is faster than the proposed one. However, the proposed design was selected based on its ability to perform complex instructions, such as multiplication, faster. In this section, simulation results for the alternative approach are given and compared to the proposed ones.

The alternative approach is simulated using a 0.8m CMOS technology under normal operating conditions (5V, 27°C). It is simulated for the same scenarios of arithmetic operations used for the proposed design; i.e. CASE1, CASE2, and CASE3. Another two simulation runs are conducted for logical and shift operations. Table 6.12 shows the simulation results for the alternative approach.

These include the Slowest Data Bit (SDB), FCS, the latched data, ACK, pre-charge, and the over all delay time.

For comparison purposes, the previously reported results for the proposed design are summarized in Table 6.13.

Table 6.12: Simulation Results for the Alternative Approach.

Scenario	SDB(ns)	FCS(ns)	Latched Data(ns)	ACK(ns)	Pre-charge(ns)	Over all Cycle(ns)
CASE 1	16.2	17.4	18.3	18.7	3.8	17.24
CASE 2	15.4	17.2	17.8	18.3	4.1	17.14
CASE 3	11.9	12.7	13.6	13.8	4.1	12.64
Logic Op.	11.3	11.9	12.4	13.2	4.6	12.54
Shifts	8.18	8.54	9.75	9.90	2.4	7.04

Table 6.13: Simulation Results for the Proposed Approach.

Scenario	SDB(ns)	FCS(ns)	Latched Data(ns)	ACK(ns)	Pre-charge(ns)	Over all Cycle(ns)
CASE 1	16.9	17.7	18.7	18.8	3.9	17.44
CASE 2	16.2	17.4	18.1	18.7	3.7	17.14
CASE 3	12.2	13.0	13.8	14.3	3.7	12.74
Logic Op.	11.7	13.0	13.6	14.3	3.7	12.74
Shifts	11.4	12.9	13.8	14.2	3.7	12.64

Thus, comparing the results of Tables 6.12 and 6.13, it is observed that the alternative approach is faster than the proposed one. While the speed improvement for arithmetic and logical operations is not exceeding 2%, it is 44% for shift operations. However, the proposed design has the ability to perform complex operations, such as multiplication, in one cycle while the alternative approach can perform them in two cycles. Thus, the proposed design is faster than the alternative approach in performing complex instruction. Therefore, all simulation results and analysis are based on the proposed design.

6.2.4 Comparison with other Completion Detection Approaches

In this section, a comparison study is conducted on different existing completion detection approaches and compared to the proposed approach in terms of efficient T_{slk} and hardware overhead. The proposed approach is compared to the approach reported by Nowick [38] and the approach reported by Gustavo [35]. The comparison study is based on simulating a 32-bit self-timed adder using eight consecutive blocks of 4-bit of MCC-1 and MCC-0 using a 0.8 CMOS technology under normal operating conditions (5V, 27°C).

The completion detection circuit reported by Nowick is shown in Figure 6.11. This approach depends on using all the carry nodes of MCC-1's and MCC-0's after inversion to generate the FCS (Done). For comparison purposes, the 32-bit self-timed adder using Nowick approach is simulated for cases that reflect the maximum and minimum T_{slk}.

The completion detection circuit reported by Gustavo is shown in Figure 6.12. Again, this approach depends on using all the carry nodes of MCC-1's and MCC-0's after inversion to generate the FCS (Done). $Comp_i$ signal results from OR-ing C_i^1 of MCC-1 with C_i^0 of MCC-0. The 32-bit self-timed adder using Gustavo approach is simulated for cases that reflect the maximum and minimum T_{slk}.

Table 6.14 compares the simulation results for the three approaches. It is observed that the proposed approach is the best in terms of hardware overhead and efficient $T_{slk.}$ In addition, it has the smallest pre-charge time.

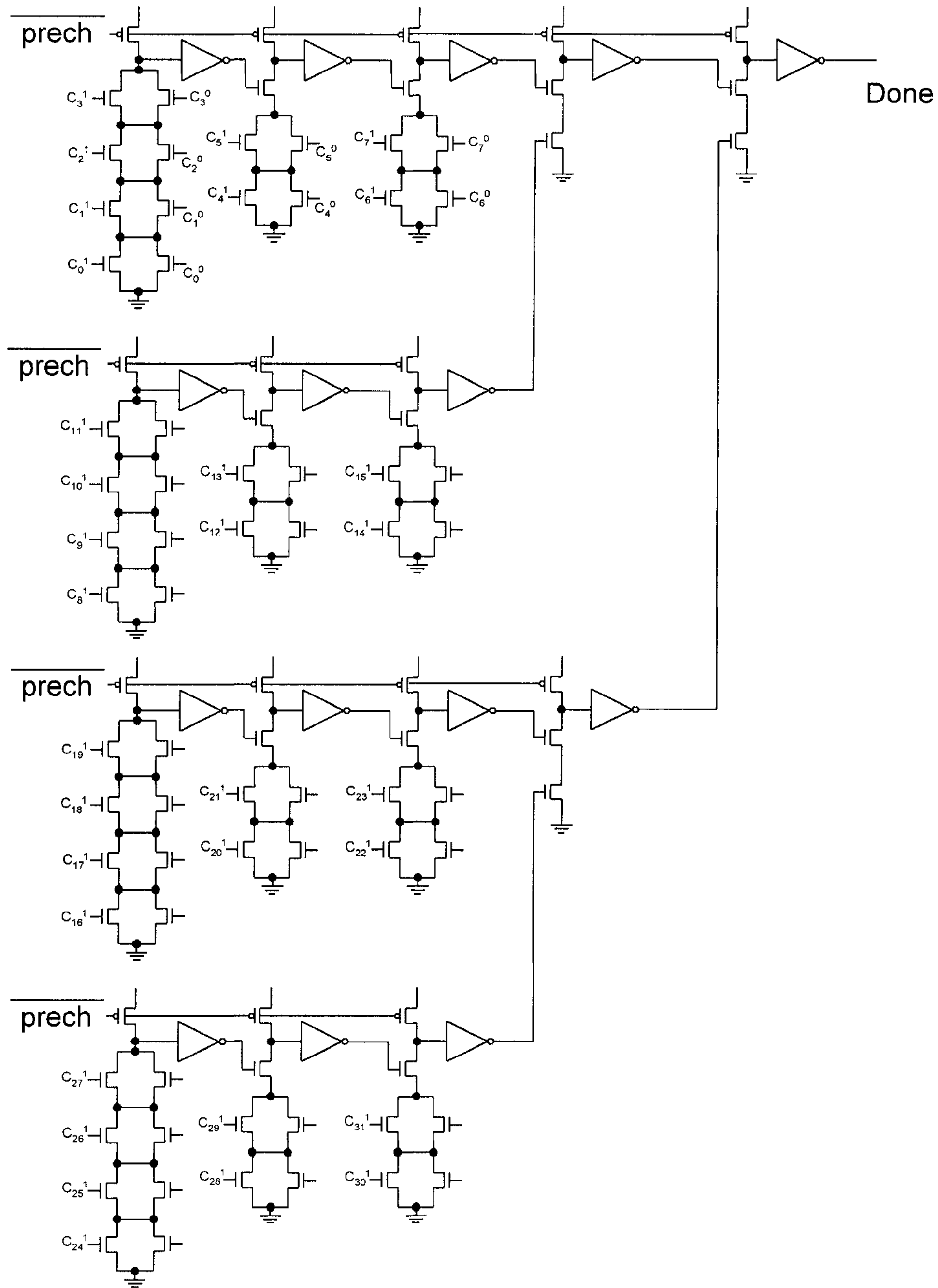


Figure 6.11: Completion Detection Proposed by Nowick.

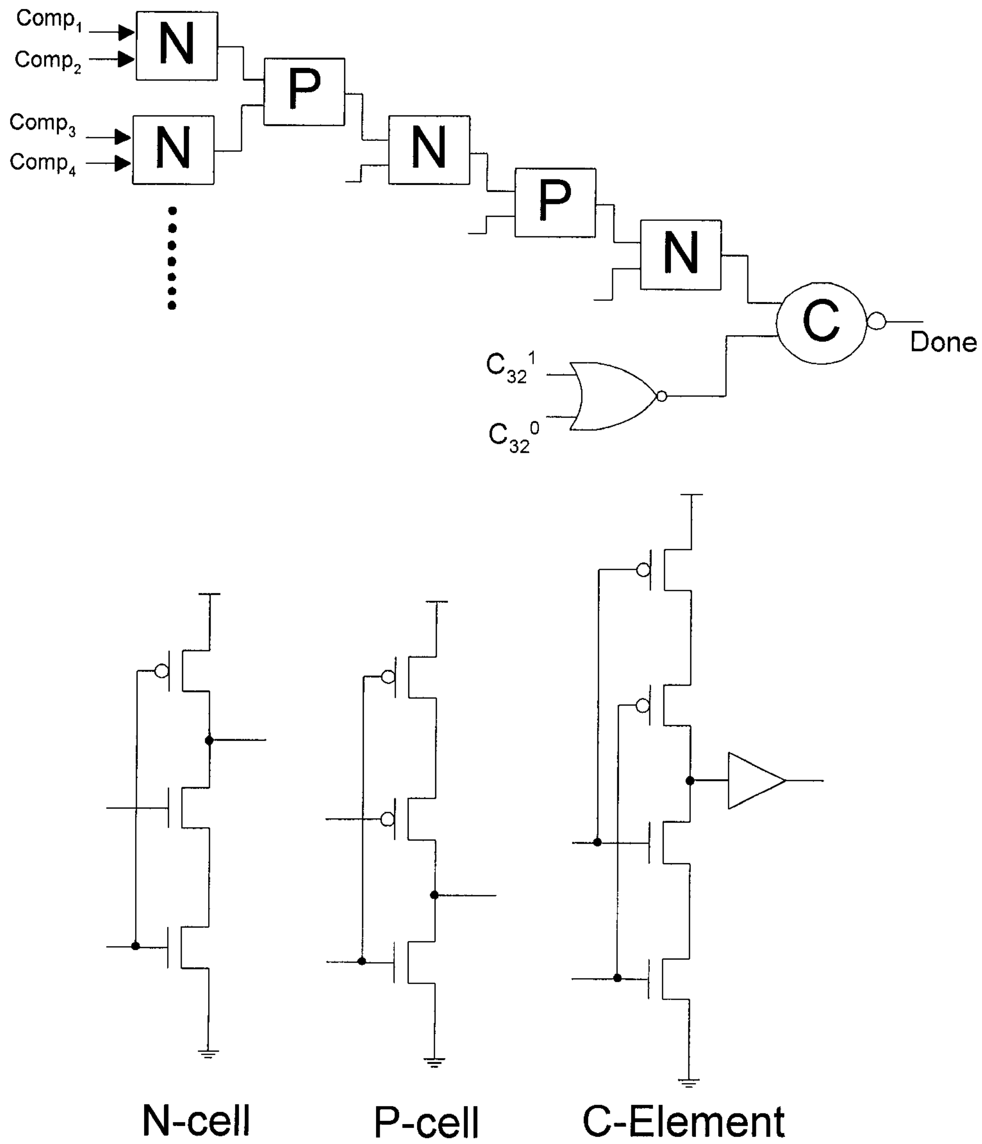


Figure 6.12: Completion Detection Proposed by Gustavo.

Table 6.14: Comparison of Completion Detection Approaches.

Criteria	Our Approach	Nowick	Gustavo
Min Tslack	0.4 ns	0.5 ns	1.6 ns
Max Tslack	1.0 ns	1.6 ns	2.1 ns
Precharge-Time	1.6 ns	3.0 ns	5.3 ns
# of Transistors	56	123	229

CHAPTER 7

SPEED UP TECHNIQUES

The total delay of any operation carried out by the proposed ALU consists of two components. The first is a fixed delay component which consists of the delay required to evaluate the propagate, kill, and generate signals, the XOR/XNOR gate delay required to evaluate the sum bits and the final completion signal, and the delay required to latch the sum bits and the FCS. The second is the variable delay component needed to propagate the carry signals through the MCC's. This delay part is the major delay component as the ALU width gets larger while the first delay component is independent of the ALU width. Improving the second delay component improves the overall performance of the ALU.

This chapter presents several possible speed-up techniques that may be used to improve the carry propagation through the MCC's. These techniques are evaluated and SPICE-simulated using a 0.8μ CMOS technology for a 64-bit ALU.

The used speed-up techniques include carry bypass, carry skip, and partial carry skip.

7.1 Carry Bypass

The carry bypass circuit within a 4-bit MCC block shown in Figure 7.1 improves the worst case carry propagation delay, i.e. when P_1, P_2, P_3, P_4 are all ones. SPICE simulation of a 64-bit MCC under worst case condition using the circuit of Figure 7.1 shows a propagation delay of 14.15 NS compared with a propagation delay of 17.45 NS without the carry bypass logic.

Figure 7.1: Carry Bypass Circuit.

This carry bypass circuit can be placed between any two adjacent 4-bit MCC blocks at four different tapping points as explained below.

The bypass circuits connects the carry node $\overline{C_j}$ of the ith MCC block with the carry node $\overline{C_j}$ of the $(i+1)^{th}$ MCC block as shown in Figure 7.2. This scenario is repeated between every two adjacent 4-bit MCC blocks. SPICE simulations of a 64-bit MCC under worst case delay condition using the circuit of Figure 7.2 (for j=1, 2, 3, 4) report the propagation delays shown in Table 7.1.

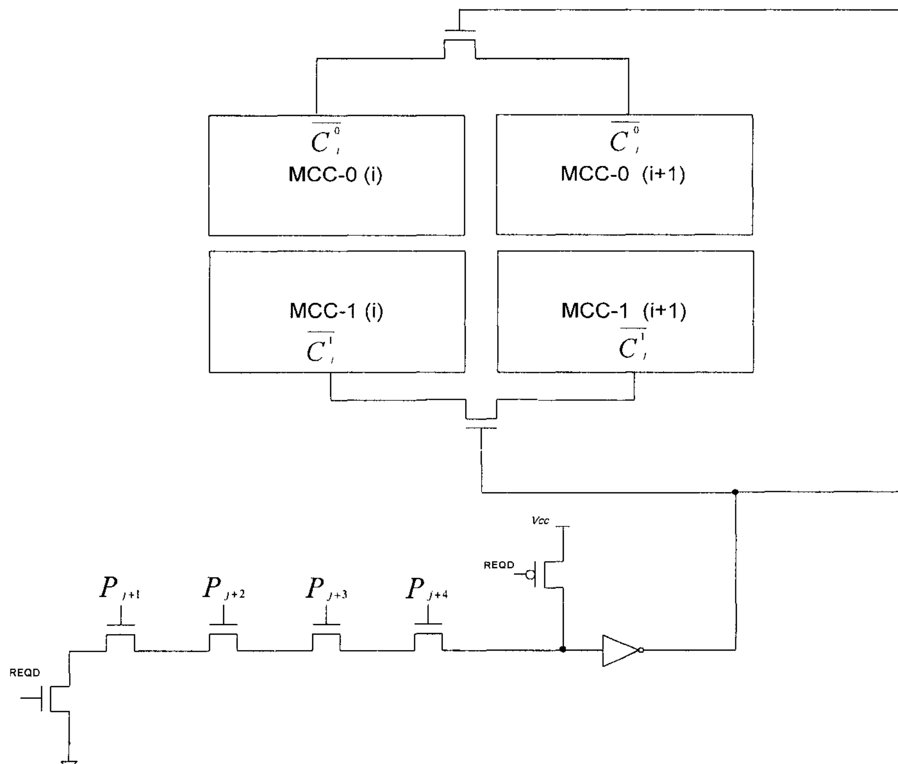


Figure 7.2: Carry Bypass Circuit within MCC Blocks.

Table 7.1: Propagation Delay for Different Carry Bypass Locations.

j	**Carry Bypass Location**	**Propagation delay**
1	$\overline{C_j}$ - $\overline{C_j}$	11.25 NS
2	$\overline{C_j}$ - $\overline{C_j}$	10.95 NS
3	$\overline{C_j}$ - $\overline{C_j}$	10.25 NS
4	$\overline{C_j}$ - $\overline{C_j}$	10.15 NS

7.2 Carry Skip

The carry skip technique, discussed at chapter 4, allows the input carry to skip certain adder stages under certain conditions. The length that a carry can skip varies but it is recommended not to exceed 4 stages. Carry skip can be implemented into the proposed ALU for both Manchester Carry Chains namely MCC-1 and MCC-0 as shown in Figure 7.3.

Figure 7.3: Carry Skip Technique.

Figure 7.3 shows that the carry-in signal (Y) will skip the four stages of the MCC when all the carry propagate signals (Pi's) are ones and will cause the carry node $\overline{C_4}$ to discharge low through the two NMOS pull down transistor path. The SPICE simulation for a 64-bit MCC for the worst case delay condition using the circuit of Figure 7.3 reports a propagation delay of 6.15 NS which is 64% improvement due to the skip logic.

7.3 Partial Carry Skip

Partial carry skip is similar to the carry skip with the difference that the input carry skips only three stage of the MCC instead of four stages as shown in Figure 7.4. In this case, the carry skip logic is effective whenever (P_1, P_2, P_3) = (1, 1, 1). Although the input carry skips only three stages in this case, the number of combinations that satisfies the skip condition (P_1, P_2, and P_3 are all ones) is more than the combinations that satisfy the condition for the full carry skip (P_1, P_2, P_3, and P_4 are all ones). The SPICE simulation for a 64-bit MCC for the worst case delay condition using the circuit of Figure 7.4 reports a propagation delay of 8.65 NS which is 50% improvement due to the skip logic. Table 7.2 summarizes the simulation results for the above speed up techniques for comparison.

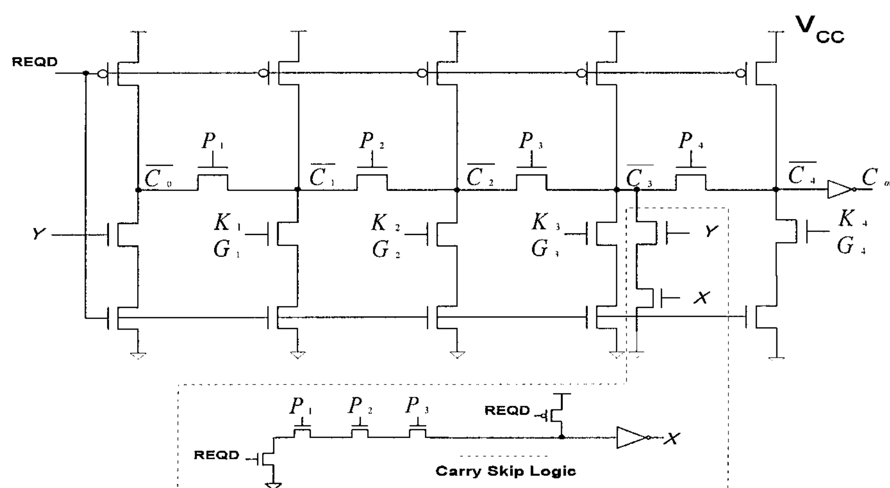


Figure 7.4: Partial Carry Skip.

Table 7.2: Worst Case Propagation Delay for Different Speed Up Techniques of a 64-bit MCC.

No.	Speed up Technique	Propagation Delay (Simulation)
1	No speed up	17.45 NS
2	Carry bypass	
	Bypass within the MCC block	14.15 NS
	$\overline{C_1}$ - $\overline{C_1}$ Carry Bypass	11.25 NS
	$\overline{C_2}$ - $\overline{C_2}$ Carry Bypass	10.95 NS
	$\overline{C_3}$ - $\overline{C_3}$ Carry Bypass	10.25 NS
	$\overline{C_4}$ - $\overline{C_4}$ Carry Bypass	10.15 NS
3	Carry Skip	6.15 NS
4	Partial Carry Skip	8.65 NS

From the above table, it is noticed that the carry skip is the best among others where the carry propagation delay is improved by 64%. Furthermore, the carry propagation delay is improved by 50% for the partial carry skip. Moreover, it is improved by 40% for the carry bypass ($\overline{C_4}$ - $\overline{C_4}$). Thus, carry skip techniques show better improvement for the worst case carry propagation delay when compared to carry bypass techniques.

Although carry skip has better improvement on the worst case propagation delay when compared to partial carry skip, cases that satisfy the partial carry skip condition are more. Thus, a simulation study is conducted on both techniques to evaluate the average gain in speed for both techniques. A comprehensive simulation runs are conducted on an 8-bit MCC for both techniques and the gain is calculated for every case. The simulation study shows that the average gain in speed when the carry skip is involved is 4.7% while it is 7.9% for the partial carry skip. Hence, the partial carry skip technique has better average gain than the carry skip one.

The 16-bit self-timed ALU with partial carry skip has been simulated with full carry propagation on MCC-1's. Table 7.3 shows the simulation results for the 16-bit ALU with and without the partial carry skip and the gain in speed.

Table 7.3: Worst Case Delays for a 16-bit ALU with and without partial carry skip.

	Delay without carry skip	**Delay with partial carry skip**	**% Gain**
Carry propagation	9.15 NS	6.05 NS	33.8
The slowest data bit	11.55 NS	8.15 NS	29.4
The slowest latched data bit	13.95 NS	11.05 NS	20.7
Final completion signal	13.05 NS	10.25 NS	21.4
ACK	14.95 NS	12.05 NS	19.4

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

A self-timed ALU designed around a self-timed adder has been presented. The carry signals of the self-timed adder are double rail encoded through the use of two complementary Manchester Carry Chains (MCCs). The middle carry nodes of the two MCCs are used to detect completion signal. This scheme results in efficient generation of the final completion signal with minimum delay.

Different size self-timed adders (4, 8, 16 and 32-bit) were SPICE simulated using 0.8μ CMOS technology parameters. Simulations were run under the worst case delay and the worst case functionality conditions. SPICE simulations have shown that adders with sizes > 8-bits work robustly satisfying the bundled data constraint ($T_{slk} > 0$). For the 8-bit adder, however, the bundled data constraint was marginally met under the worst case functionality condition with the final completion signal asserted almost simultaneously ($T_{slk} = 0$) with the slowest sum bit.

The self-timed adder was augmented to implement other logic and arithmetic operations. Four 4-bit blocks of the self-timed adder were cascaded to build a 16-bit self-timed ALU. A 16-bit barrel shifter is involved to implement shift operations. The 16-bit ALU was SPICE simulated using 0.8μ CMOS technology parameters. SPICE simulations have shown that the ALU works robustly satisfying the bundled data constraint under various temperature and power supply voltage combinations. The ALU was designed such that relative delays are process tracking so as to ensure proper operation for different technologies. Accordingly, SPICE simulations of the self-timed ALU using 0.18μ CMOS technology parameters were run and the ALU was found to function robustly satisfying the bundled data constraint providing that the proposed design is process tracking due to the proper implementation of the completion detection circuit.

The comparison of the proposed ALU with two other reported self-timed designs was conducted. Simulation results have shown that our design to be the most efficient in terms of area and minimized T_{slk}.

Several speed up techniques were investigated and SPICE simulated. These include carry bypass, carry skip and partial carry skip. Simulation studies have shown that the partial carry skip is the best in terms of average gain in speed.

Possible future works include evaluation of an overall asynchronous processor using the proposed approach with sizable instruction set as compared to a synchronous processor. Furthermore, multi-level carry skip logic may be investigated.

BIBLIOGRAPHY

[1] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor," IEEE Design & Test of Computers, pp 50-63, 1994.

[2] Jose A. Tierno, Alian J. Martin, Drazen Borkovic, and Tak K. Lee, "A 100-MIPS GaAs Asynchronous Microprocessor," IEEE Design & Test of Computers, pp 43-49, 1994.

[3] S. M. Nowick, "Design of a low-latency asynchronous adder using speculative completion," IEE Proceedings-Computers and Digital Techniques, 143(5) : 301-307, September 1996.

[4] Ted E. Williams, and Mark A. Horowitz, "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider," IEEE Journal of Solid-State Circuits, 26(11) : 1651-1661, November 1991.

[5] Richard P. Brent, and H. T. Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, C-31(3) : 260-264, March 1982.

[6] Pak K. Chan, and Martine D. F. Schlag, "Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip," IEEE Transactions on Computers, 39(8) : 983-992, August 1990.

[7] Alain Guyot, Bertrand Hochet, and J. M. Muller, "A Way to Build Efficient Carry-Skip Adders," IEEE Transactions on Computers, C-36(10) : 1144-1152, October 1987.

[8] Norman M. Martin, and Stephen P. Hufnagel, "Conditional-Sum Early Completion Adder Logic," IEEE Transactions on Computers, C-29(8) : 753-756, August 1980.

[9] Akhilesh Tyagi, "A Reduced-Area Scheme for Carry-Select Adders," IEEE Transactions on Computers, 42(10) : 1163-1170, October 1993.

[10] T. P. Kelliher, R. M. Owens, M. J. Irwin, and T. T. Hwang, "ELM - A Fast Addition Algorithm Discovered by a Program," IEEE Transactions on Computers, 41(9) : 1181-1184, September 1992.

[11] R. W. Doran, "Variants of an Improved Carry Look-Ahead Adder," IEEE Transactions on Computers, 37(9) : 1110-1113, September 1988.

[12] Gordon M. Jacobs, and Robert W. Brodersen, "A Fully Asynchronous Digital Signal Processor Using Self-Timed Circuits," IEEE Journal of Solid-State Circuits, 25(6) : 1526-1537, December 1990.

[13] Alessandro De Gloria, and Mauro Olivieri, "Statistical Carry Lookahead Adders," IEEE Transactions on Computers, 45(3) : 340-347, March 1996.

[14] D. Salomon, "A Design for an Efficient NOR-gate only Binary Ripple Adder with Carry-Completion-Detection Logic," The Computer Journal, 30(3) : 283-285, 1987.

[15] W. F. Richardson, and E. Brunvand, "Architectural considerations for a self-timed decoupled processor," IEE Proceedings-Computers and Digital Techniques, 143(5) : 251-257, September 1996.

[16] Ravi Ramachandran, and Shih-Lien Lu, "Efficient Arithmetic Using Self-Timing," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 4(4) : 445-453, December 1996.

[17] Ivan E. Sutherland, "Micropipelines," Communications of the ACM, 32(6) : 720-738, June 1989.

[18] Nhon T. Quach, and Michael J. Flynn, "High-Speed Addition in CMOS," IEEE Transactions on Computers, 41(12) : 1612-1615, December 1992.

[19] Vitit Kantabutra, "Accelerated Two-Level Carry-Skip Adders - A Type of Very Fast Adders," IEEE Transactions on Computers, 42(11) : 1389-1393, November 1993.

[20] Vitit Kantabutra, "Designing Optimum One-Level Carry-Skip Adders," IEEE Transactions on Computers, 42(6) : 759-764, June 1993.

[21] Richard F. Hobson, "Optimal Skip-Block Considerations for Regenerative Carry-Skip Adders," IEEE Journal of Solid-State Circuits, 30(9) : 1020-1024, September 1995.

[22] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick, "Synthesis of 3D Asynchronous State Machines," in Proceedings of ICCAD, pp. 346-350, 1992.

[23] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang, "Q-Modules: Internally Clocked Delay-Insensitive Modules," IEEE Transactions on Computers, 37(9) : 1005-1018, September 1988.

[24] Scott Hauck, "Asynchronous Design Methodologies: An Overview," Proceedings of the IEEE, 83(1) : 69-93, January 1995.

[25] Lee A. Hollaar, "Direct Implementation of Asynchronous Control Units," IEEE Transactions on Computers, C-31(12) : 1133-1141, December 1982.

[26] Chetana N., Mary Irwin, and Robert M. Owens, "Area-Time-Power Tradeoffs in Parallel Adders," IEEE Transactions on Circuits and Systems, 43(10) : 689-702, October 1996.

[27] Amos R. Omondi, Computer Arithmetic Systems, Prentice Hall (UK), 1994.

[28] Mark A. Franklin and Tienyo Pan, "Performance Comparison of Asynchronous Adders," Computer and Communication Research Center, Washington University, 1994.

[29] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple, "AMULET1: An Asynchronous ARM Microprocessor," IEEE Transactions on Computers, 46(4) : 385-397, April 1997.

[30] Francois Anceau, The Architecture of microprocessors, Addison-Wesley (UK), 1986.

[31] J. Escriba, and J.A. Carrasco, "Self-timed Manchester chain carry propagate adder," Electronics Letters, Vol. 32, No. 8, pp. 708-710, April 1996.

[32] S. Perri, P. Cosonello, G. Cocorullo, G. Cappuccino, and G. Staino, "VLSI Implementation of a fully static CMOS 56-bit self-timed adder using overlapped execution circuits," The 8th IEEE International Conference on Electronics, Circuits and Systems, 2001.(ICECS 2001), pp. 723-727, vol. 2.

[33] Fu-Chiung Cheng, Stephen H. Unger, and Michael Theobald, "Self-timed carry-lookahead adders," IEEE Transactions on Computers, pp. 659-672, July 2000.

[34] A. De Gloria, and M. Olivieri, "Completion-detecting carry select addition," IEE Proceedings. Computers and Digital Techniques, pp. 93-100, March 2000.

[35] Gustavo A. Ruiz, "Evaluation of Three 32-bit CMOS adders in DCVS logic for self-timed circuits," IEEE Journal of Solid-State circuits, pp. 604-613, vol. 22, No. 4, April 1998.

[36] P. Corsonello, S. Perri, and G. Cocorullo, "A 56-bit self-timed adder for high speed asynchronous datapath," IEEE Transactions on Computer, pp. 37-41, 1999.

[37] J. Won, and K. Choi, "Self-timed statistical lookahead adder using multiple-output DCVSL," IEEE Trans. On Computer, pp. 560-563, 1999.

[38] Steven M. Nowick, Kenneth Y. Yun, Peter A. Beerel, and Ayoob E. Dooply, "Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders," IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, Netherlands, 1997.

VITA

- Feras Ali Mohammed Maadi
- Born in Doha, Qatar, 14 November 1969.
- Received Bachelor Degree in Computer Engineering from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia in September 1992.
- Joined Computer Engineering Department, KFUPM, as a Part Time Student in 1993.
- Started working on his thesis in January 2002 and successfully defended his work in June 2003.
- Received Master of Science Degree in Computer Engineering from KFUPM, Dhahran, Saudi Arabia in April 2004.