

1999

On the Use of Undefined Logic Values in Digital VLSI

William D. Armitage
University of Rhode Island

Follow this and additional works at: https://digitalcommons.uri.edu/oa_diss

Terms of Use

All rights reserved under copyright.

Recommended Citation

Armitage, William D., "On the Use of Undefined Logic Values in Digital VLSI" (1999). *Open Access Dissertations*. Paper 788.
https://digitalcommons.uri.edu/oa_diss/788

This Dissertation is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

ON THE USE OF UNDEFINED LOGIC VALUES IN DIGITAL VLSI
BY
WILLIAM D. ARMITAGE

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
ELECTRICAL ENGINEERING

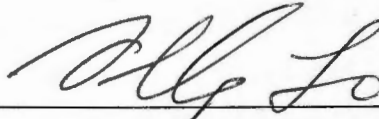
UNIVERSITY OF RHODE ISLAND

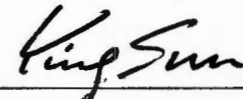
1999

DOCTOR OF PHILOSOPHY DISSERTATION
OF
WILLIAM D. ARMITAGE

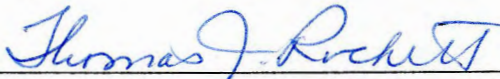
APPROVED:

Dissertation Committee
Major Professor









DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1999

Abstract

This dissertation addresses both the consequences and advantages of the fact that all digital logic implementations are analog in reality. Although, in the ideal sense, all digital signals exist at either a logic 0 or a logic 1, in practice signals are generally between these two extreme values. There is a poorly-defined zone (which we denote as ϕ) near the midpoint of the logic range where a logic level is not recognizable as a 0 or 1 beyond a reasonable doubt. Variations in design and fabrication exacerbate this uncertainty. We introduce the concept of *zoned binary*, which has three states $\{ 0, \phi, 1 \}$, and arbitrarily define ϕ as consisting of the logic voltage range between $1/3V_{dd}$ and $2/3V_{dd}$, although the designer is free to set the boundary at any other levels appropriate to the specific implementation. It is pointed out that there are many physical causes why a logic value might be in the ϕ zone, including insufficient time to settle to a static value, wire and device defects, and noise. It is noted that current techniques focus on avoidance, or detection of and dealing with effects. We introduce the idea of an unknown value as *information*, and suggest that it can be used to enhance performance. We design and test a detector for ϕ , and proceed to apply it to rudimentary practical problems such as interconnect difficulties, and to more demanding applications such as asynchronous systems and communications error correction. A new logic family - Binary Plus logic - is proposed, designed and validated, in both static and dynamic versions. Its applicability to completion-detection requirements of asynchronous circuitry is shown, and an asynchronous stage is designed, fabricated and tested. The detection of ϕ in a received communications bit is interpreted as an *error location* method. It is shown that this information can be used with techniques well documented in the literature to enhance the error

correction capability of existing error-control coding schemes. A 9-bit simple parity-based circuit capable of correcting received bits in the ϕ state is designed, fabricated and shown to perform properly.

Acknowledgments

I would like to thank my advisor, Professor Jien-Chung Lo, for his understanding, guidance, insightful suggestions, and, above all, his patience during my studies at the University of Rhode Island. His many practical recommendations, an occasional gentle “push” when needed, and his willingness to share his own experiences with paper preparation - all were instrumental in my success.

I am deeply grateful to Professor James Daly, Professor Edmund Lamagna and Professor Ying Sun for serving on my thesis committee, and to Professor Bala Ravikumar for serving as my defense Chair. Each of them has given freely of his time, and has shown extraordinary flexibility in adapting to my at times “strange” schedule. I greatly appreciate their support.

Special thanks go to Phyllis Golden, without whose occasional reminders and timely assistance I might not have made it through this program.

Preface

Throughout my long and checkered career in the technology field, I have always been fascinated with unknowns. Whether they were statistical “missing values”, “missing inputs” in neural networks, or other instances of “knowing that something was not known”, I was interested in how the knowledge of their existence affected how the problem was approached, and possibly affected the validity of the results.

When taking ELE447 and ELE537 with Professor James Daly, I obtained practical, and occasionally frustrating, experience in dealing with a new kind of “unknown” - logic values that were not recognizable as either a zero or one. Trying to adjust the design of a circuit so as to minimize the time it spent in this unknown area, and thus delivered results faster, occupied serious time in design lab.

When the topic of this dissertation (among other possible topics) was suggested to me, I found that it captured my interest immediately. Although I could find no previous work directly addressing the topic, there was a reasonable body of literature in areas that would be affected by this work. It quickly became clear that unknown values in CMOS VLSI circuitry was an area that should be viewed in a positive way, rather than something to be avoided. Attempting by design to avoid an uncertain logic level (as I had spent so much time in the lab doing) was not at all the same thing as detecting it and using the information.

The idea of maintaining the integrity of the “unknown” state through the function of the gate led to the development of a new logic family, Binary Plus logic, and to its dynamic version, Centered Binary Plus logic. This family is equivalent to classic binary logic in terms of the functions realized, but has the added advantage (hence the “Plus”) of being able to recognize and deal with inputs in an uncertain logic range

in an way appropriate to the binary function implemented by the gate. While the family should certainly be useful in dealing with inputs that are genuinely unknown, it was also shown to have great potential as a completion-indicating construct, and hence had obvious use in the area of asynchronous systems. Using the Centered Binary Plus logic family, a rudimentary 4-bit ripple carry adder was designed and fabricated. Testing has shown that the adder takes advantage of many input data patterns to produce significant completion time savings.

Unknown inputs are often the result of a defect or noise in transmission of the data from another place (on the chip, within the computer, or in the world) to the circuit. Current techniques for combating communications errors focus on error-control coding. It is well established in the literature that if the *location* of an error can be independently (of the error-control coding scheme) determined, correction capabilities are greatly enhanced - for example, a distance-4 code can correct three errors whose locations are known, as opposed to only one when it has to determine for itself the location of the error. Another example is the simple 1-bit parity code, which is, by itself, capable of detecting one error but correcting none. Using uncertain logic values as error location identifiers, a simple 1-bit parity scheme can *correct* one-bit errors. As part of this work, a 9-bit parity-based communications input register was developed, fabricated and tested. This circuit can identify an uncertain bit, and use the parity relationship in the transmitted word to correct it.

Contents

Abstract	ii
Acknowledgments	iv
Preface	v
Table of Contents	vii
List of Tables	xi
List of Figures	xiii
1 Introduction	1
1.1 Motivation	2
1.1.1 Asynchronous system design	3
1.1.2 Communications error correction	4
1.2 Organization of the dissertation	5
2 Undefined logic values in digital VLSI	6
2.1 Defining Terms	6
2.2 Existing approaches to avoid undefined values	7
2.2.1 What can cause undefined values?	7
2.2.2 What are the effects of undefined values?	16
2.2.3 How are they combated?	19
2.3 An undefined value as information	23

2.4	Summary	25
3	Binary Plus logic	26
3.1	The detector	26
3.1.1	Not a new “value”	27
3.1.2	Required products of the detection process	28
3.2	Development of Binary Plus concepts	28
3.2.1	A small step	29
3.2.2	Complete “Binary Plus” concept	30
3.3	Binary Plus logic specifications	33
3.3.1	Complementary logic	33
3.3.2	Intuitive development	35
3.3.3	Formal development	39
3.3.4	Binary Plus and races	42
3.4	Summary	44
4	Design and Implementation	46
4.1	Detector design	46
4.1.1	The design equations	48
4.2	Binary Plus gate design	52
4.2.1	Internal versus external complemented inputs	54
4.3	Rudimentary applications	55
4.3.1	Warnings of potential problems	56
4.3.2	The detector revisited as a decoder	57
4.4	Introduction to the proof-of-concept circuit	59
4.4.1	Overall view	60
4.4.2	Layout	61
4.4.3	Pinouts	62
4.4.4	Test board	64
4.5	Binary Plus component experiments	64
4.5.1	Circuit descriptions	65

4.5.2	Testing results	67
4.6	Summary	69
5	Centered Binary Plus logic	73
5.1	Static versus dynamic logic in VLSI design	73
5.2	Asynchronous systems - current status and requirements	75
5.2.1	Overview	75
5.2.2	Implications for input set sensitivity	76
5.2.3	Globally asynchronous locally synchronous systems	78
5.2.4	Currently used methods for completion detection	81
5.2.5	Interstage requirements	82
5.3	Centered Binary Plus logic	83
5.3.1	Precharge is to V_h	84
5.3.2	Inherent speed enhancement	86
5.3.3	Elimination of races	87
5.3.4	Detection of invalid inputs and defects	87
5.3.5	Granularity	88
5.3.6	Control and handshaking	89
5.4	Comparison with other GALS self-clocking methods	93
5.5	Fabricated 4-bit ripple-carry adder experiment	94
5.5.1	Ripple-carry adder	95
5.5.2	Testing strategy	99
5.5.3	Testing results	100
5.6	Summary	102
6	Communications applications	104
6.1	Hardware and error detection/correction	105
6.2	Error-control coding	106
6.2.1	Channel models and errors	106
6.2.2	Distance	111
6.2.3	Simple parity code	112

6.2.4	SEC and SEC/DED codes	112
6.3	Error location with zoned binary detector	112
6.3.1	An easy case: the unidirectional channel	113
6.4	Error correction strategies for ϕ errors	114
6.4.1	Strategy for simple parity codes	115
6.4.2	Extension of strategy to DED codes	115
6.4.3	Extension to SEC/DED codes	115
6.4.4	Extension to the general model	116
6.5	Implementation example: simple parity code	117
6.6	The detector once again revisited as a decoder	119
6.7	Partial utilization: some gain at lower cost	119
6.7.1	Code-independent advantage	120
6.7.2	Simple set to zero with uniform distribution of erasure errors .	120
6.7.3	Simple set to most probable value with asymmetric distribu-	
	tion of erasure errors	123
6.7.4	Possible enhancements	123
6.7.5	Special case: Bridge detection and correction for bus commu-	
	nications	125
6.8	Comparison with classic method	126
6.9	Fabricated 9-bit parity-based corrector experiment	128
6.9.1	Actual design topology	128
6.9.2	Functional unit topology	129
6.9.3	Testing results	132
6.10	Summary	133
7	Summary and conclusions	137
7.1	Future work	139
	List of References	141
	Bibliography	146

List of Tables

3.1	Division of $V_{ss} \Rightarrow V_{dd}$ Logic Range into Zones	26
3.2	Implied Value and Signal	27
3.3	Relationship of Output Signals from Detector	28
3.4	2-Input OR Gate Truth Table	30
3.5	Implied Value and Signal	31
3.6	Binary Plus 2-Input OR Gate Truth Table	31
3.7	Binary Plus 2-Input AND Gate Truth Table	32
3.8	Binary-Plus NOT Gate Truth Table	32
3.9	Binary-Plus 2-Input XOR Gate Truth Table	33
3.10	Binary Plus 2-Input OR Gate Truth Table	35
3.11	Relationship of Output Signals	35
3.12	<i>fet</i> Network States vs. Zoned Output	36
3.13	Relationship of Output Signals, Including Inverted	38
4.1	Inverter Pair Behavior	48
4.2	Binary Plus Inverter Truth Table	55
4.3	Input Pinouts	64
4.4	Output Pinouts	65
4.5	Power Supply Pinouts	66
4.6	Test Results: Binary Plus Inverter Pair	67
4.7	Test Results: 2-input Centered Binary Plus logic AND and OR Gates	69
5.1	Ripple-Carry Adder Performance Summary	78
5.2	Results of Gate-Level Simulation of 4-bit Ripple Carry Adder	100
5.3	Timings of Adder Cycle Time Across Input Patterns	103

6.1	Comparison with Classic Parity Checker	128
6.2	Truth Table for PE_{out}	132
6.3	All Inputs in Valid Ranges and Parity = "Even"	133
6.4	All Inputs in Valid Ranges and Parity = "Odd"	133
6.5	Some Inputs in ϕ Range and Parity = "Even"	135
6.6	Some Inputs in ϕ Range and Parity = "Odd"	136

List of Figures

2.1	Simple Circuit with Inherent Race	9
2.2	Physical Bridge (Short) Between Two Bus Lines	11
2.3	Resistive Equivalence of Bridge (Short) Between Two Bus Lines	11
2.4	“Open” in Bus Line	12
2.5	Resistive Equivalent to Open in Transmission Line	12
2.6	Simple NAND Circuit	13
2.7	NAND Circuit with Shorted Transistor	14
2.8	Resistive Network: Shorted NAND	14
2.9	NAND Gate with Open Transistor	15
2.10	CMOS Inverter Transition	16
2.11	CMOS Inverter Transition across Fabrication Runs	18
2.12	Input to Two Inverters	19
2.13	Two Inverter Chains to XOR	19
2.14	Operation of Two Inverter Chains to XOR	19
2.15	Non-Pipelined Circuit (Block Diagram)	21
2.16	Pipelined Circuit (Block Diagram)	21
3.1	Prevention of Output Based on Uncertain Inputs	29
3.2	Complementary Logic	34
3.3	Binary Plus Gate	36
3.4	Binary Plus Gate with Float Centering	37
3.5	Binary Plus Gate including Complemented Inputs	38
4.1	Detector: Simple Block Diagram	47
4.2	Detector: with Varied Transition Voltage Inverters	47

4.3	Basic Inverter Design	48
4.4	OR Created with Inverters and a NAND	53
4.5	OR Created with Inverters and a Device Level NAND	53
4.6	Binary Plus OR Gate	54
4.7	Binary Plus OR Gate with Float Centering	54
4.8	Example of Multi-Zoning	56
4.9	Forcing a Zone onto a Floating Line	58
4.10	Inoperative Sensor Encoding	59
4.11	Circuit Layout	62
4.12	Binary Plus Inverter Pair	65
4.13	Centered Binary Plus logic 3-input OR Gate	66
4.14	Centered Binary Plus logic 2-input AND Gate	67
4.15	Centered Binary Plus logic 3-input AND Gate	68
4.16	Pinout Schematic	71
4.17	Test Board Schematic	72
5.1	Dynamic NAND Gate	74
5.2	Ripple-Carry Adder	77
5.3	Three-Stage Pipeline	79
5.4	GALS Three-Stage Pipeline	80
5.5	Expanded GALS Three-Stage Pipeline	80
5.6	Weak Transistor Precharge	85
5.7	Precharge Using V_h Supply	86
5.8	Adder with Completion Signal	89
5.9	Adder Including Precharge Cycle	90
5.10	Adder Including Enable Controls	92
5.11	Centered Binary Plus logic Full Adder	95
5.12	Centered Binary Plus 4-Bit Ripple Carry Adder	97
5.13	Centered Binary Plus logic Full Adder with $NONE_{fa}$	98
6.1	Symmetric Error Model	107
6.2	Ideal Asymmetric Error Model	107
6.3	Binary Erasure Error Model	108

6.4	General Channel	109
6.5	Symmetric Channel with Erasures	110
6.6	Transmission of ϕ	110
6.7	Illustrative Correction System	118
6.8	Input Bit Pre-Processing ($\phi \Rightarrow 0$)	124
6.9	Post-Processing for Two Erasures after SEC/DED Checker	125
6.10	Physical Bridge (Short) Between Two Adjacent Bus Lines	126
6.11	Distance-3 Correction System for Adjacent Bus Line Bridges	127
6.12	9-bit Implemented Correction System (4 bits shown)	129
6.13	Implemented Correction System (Bit-Slice View)	130
6.14	Selection Circuit for 9-bit Parity-Based Corrector	131

Chapter 1

Introduction

“... Yu, I shall instruct you about knowledge. To acknowledge what is known as known, and what is not known as not known is knowledge.”

Confucius, Lun Yu, Chapter 2, Verse 17

Digital logic constitutes the heart of so many of the technological improvements that have been introduced to society during the last thirty years. Personal computer systems, hardware that employs embedded processors, controllers for all sorts of previously “manual” devices - these and many more depend on digital logic for their operation.

Digital communications have likewise increased greatly, especially during the growth explosion of the Internet during the last five years.

In today’s comparatively technology-savvy world, it is likely that more people than not know words like “binary”, and can identify the concept as having to do with two states, perhaps can even specify it as the “zero or one idea.”

Binary circuitry as an electronic dichotomy, however, is an abstraction. Digital logic, as implemented in a practical sense, is not, strictly speaking, digital in nature. Although future concepts such as quantum computers and networks[1] may be based on phenomena that can be interpreted as true dichotomies, CMOS digital fabrications today are inherently analog in implementation.

1.1 Motivation

Design rules, including Boolean algebra, assume a set of two possible values: $\{0, 1\}$, but, in reality, these values do not have an equivalent voltage level in a circuit, except in the ideal sense.

Values inside a CMOS “digital” circuit are, in actuality, a continuum. Ranging from the primary supply voltage, V_{dd} , down to “ground”, V_{ss} , it is easy to classify voltage levels near V_{dd} or V_{ss} , but neither easy nor reliable to interpret a voltage near the midpoint of that range as “belonging” to a binary 0 or a binary 1, for, as shall be explained, the boundary between the upper and lower halves of this range is not a reliable one - between fabrication runs or even within a single circuit. The area near the midpoint of the range is therefore a region of uncertainty, in which a value cannot be reliably assigned to a member of the binary dichotomy. Common practice, we shall see, is to design so as to maximize the occurrence of the “easy to assign” values and minimize, insofar as possible, those which cannot be clearly assigned to one binary value or the other.

There are many physical causes for the existence of intermediate, undefined logic values near the midpoint of the logic range. The classic response is to use other methods, *not related to the existence of undefined values themselves*, to make their occurrence less likely or to find the causes and eliminate them. So circuits that exhibit undefined values at their output - because they have not had time to settle - are given enough time to settle in the worst-case condition. Manufacturing defects that might cause undefined values are addressed by extensive and sophisticated testing techniques. Problems that might develop in high-reliability systems are addressed by fault-tolerance techniques. Undefined values occurring during data transmission are detected and/or corrected using error-control coding methods.

In all of the classic approaches, undefined values are treated as a problem that *might* occur, and should be designed, tested or coded around in such a way that they will tend to be taken care of if they *do* occur. An undefined value, when it resolves itself into the incorrect binary value, is thus treated as merely a case of the “wrong” valid binary value. For example, an undefined value in data transmission

may resolve itself to its proper, "as transmitted", value, providing no error, or as the opposite, "incorrect" value, in which case the error detection/correction capabilities of the code checker are responsible for finding the problem and dealing with it.

The classic approaches make no effort to specifically detect the presence of undefined values. In so doing, they discard *information* which could potentially be useful in correcting the problem.

This work will address this region of uncertainty, showing that its existence - once detected and systematically treated - can be exploited in a number of useful applications, of which two - asynchronous system design and communications error correction - will be examined more closely.

1.1.1 Asynchronous system design

As processors scale down in feature size, but up in speed, absolute size and complexity, new problems develop. "Global clock propagation" (getting the synchronizing, lock-step control signal everywhere on the processor at roughly the same time) is becoming a greater and greater concern. One author, in discussing the future of processor design, made the observation that "the percentage of the die that can be reached in a few clock cycles is decreasing at an alarming rate." [2] Others agree, observing that while "local" interconnect time (the time for signals to propagate within an individual logic block) is actually decreasing due to decreased feature sizes, global interconnects require new approaches to avoid being a barrier to processor speed. [3]

As more and more processors "go mobile", power consumption also becomes a critical problem. Even in non-mobile applications, power consumption must be dissipated in the form of heat, a pressing design problem in itself. In CMOS circuits, power use tends to be proportionally related to clock speed. A CMOS circuit uses power only when the charge state of a circuit is changing, and states change only as a result of the clock changing. A slower clock equals less power use. Already this approach is used in portable systems today, with the aim of prolonging battery charge life. But as applications require more and more speed, this method will be squeezed between the demands of the application and the need to conserve power

and reduce the need for circuit cooling. Other methods need to be implemented to allow greater effective processing speed while keeping power use under control.[4]

Types of asynchronous systems which we will explore in this work eliminate the need for a global clock signal. Asynchronous concepts such as GALS (Globally Asynchronous Locally Synchronous)[5] limit synchronizing clock signals to the local logic block level. Additionally, when a local logic block “has no work to do,” it stops and consumes no power. We will show (and demonstrate in practice) the applicability of using detection of our uncertain logic level to GALS-based asynchronous systems.

1.1.2 Communications error correction

While it is easy to think of communications in the “macro” sense - between computers on a network, for example - we must also remember that much more communication occurs on a “micro” level - among circuits on a printed circuit board, or even among different processing elements within a single-chip microprocessor.

Data bits are continuously flowing inside a microprocessor, and elements such as noise and even radiation can create occasional errors. It is important that these errors be able to be (1) detected and, (2) if possible, corrected before serious system degradation occurs.[6]

Error-control coding - a method of encoding information bits in a group of bits also containing checking information that can be used to detect and sometimes correct errors - is the predominant method of protecting systems from data corruption errors. Merely detecting an error in a single bit using these techniques is a very simple task, utilizing what is known as a simple parity code. Designing and implementing a coding scheme that can *correct* errors is far more complex and costly, as it requires that the *bit location* of the error be identified. Much of the “overhead” of an error correction code goes into locating the error. It is well established in the literature that, if a method can be separately implemented (over and above the error-control coding scheme in use) to identify by other means the location of errors, the correction capability of a standard error-control code can be greatly enhanced.[6, 7, 8, 9, 10]

Detecting that a given bit is “uncertain” can be used as an error location technique. This information can then be utilized as described in the literature to provide superior correction capabilities.

1.2 Organization of the dissertation

Chapter 2 examines the region of uncertainty - its causes and effects - and discusses the means typically used to “avoid or evade” the consequences. We also introduce the concept of the unknown as knowledge.

Chapter 3 introduces the central concept of this work, Binary Plus logic, examining it from a theoretical standpoint and proving its validity.

Chapter 4 addresses the design and implementation of the “Binary Plus” logic family. Design equations for a simple detector for undefined logic values are derived, and rudimentary applications are discussed. The overall organization of a proof-of-concept integrated circuit fabricated as part of this work is described, and specific testing data for the detector and Binary Plus logic elementary gates are presented.

Chapter 5 extends the Binary Plus logic family to its dynamic version - Centered Binary Plus logic - and shows its applicability to the design of asynchronous systems. A simple asynchronous logic stage on the fabricated circuit is described and test data presented.

Chapter 6 considers the use of uncertain logic levels in data communications - both within a circuit and between circuits and devices. It is shown that the approach, by providing error location information, can enable limited error correction capabilities where only error detection is possible using error-control coding alone. A parity-based uncertainty error detector/corrector implemented on the fabricated circuit is described and test data presented.

Chapter 7 summarizes the work, and suggests further research areas.

Chapter 2

Undefined logic values in digital VLSI

2.1 Defining Terms

In digital logic a binary 1 is represented by a logic level 1, which is chosen by convention to be a value nominally equal to the power supply voltage V_{dd} . This voltage, typically five volts in the early days of VLSI development, may still be five volts in some circuitry, but can be less than one volt in advanced circuits today. A binary 0 is represented by a logic level 0, which is chosen by convention to be a value nominally equal to power supply ground, or V_{ss} , which we will define equal to zero volts.

In practice, values merely near V_{dd} are also considered to represent a binary 1, and those near V_{ss} a binary 0. The question therefore arises: how near V_{dd} and V_{ss} need signals be in order to be a binary 1 and 0, respectively? Although a simple question, it has no simple answer.

To aid in our understanding, let us define a term V_h :

$$V_h = V_{dd}/2$$

It would be easy (and tempting) to refer to all values $< V_h$ as binary 0 and all values $> V_h$ as binary 1. Theoretically, as $V_{ss} \Rightarrow V_{dd}$ is a continuum in the physical sense, values exactly equal to V_h are of such low likelihood that they can be said to not exist, and therefore there is no ambiguity. However, logic design is an eminently

practical process, and matters discussed later explain why such an ideal “point of division” is impractical and unreliable.

For practical reasons, we shall see, a “buffer” must be defined around V_h , such that all values outside the range of that buffer can be reliably counted on to default to binary 1 or binary 0. As a study of the precise size and statistical reliability of such a buffer is beyond the scope of this work, we shall err on the conservative side and divide the $V_{ss} \Rightarrow V_{dd}$ interval into three equal intervals, resulting in a definition of $1/3 V_{dd}$ to $2/3 V_{dd}$ for our undefined area. In short, we shall specify that, for the purpose of this work:

The voltage level interval $1/3 V_{dd}$ to $2/3 V_{dd}$ shall be defined as the “uncertain”, “undefined” or “invalid” logic level interval. That is, values in this voltage range shall be deemed to be neither logic level 1 nor logic level 0, but instead a level that cannot be reliably distinguished as to its proper binary value.

As implied above, no claim is made that this represents an ideal or even a reasonable division of the $V_{ss} \Rightarrow V_{dd}$ voltage range into truly valid and invalid sub-ranges. But it does provide a standard and a target for design and simulation of circuits illustrating the principles in this work.

2.2 Existing approaches to avoid undefined values

2.2.1 What can cause undefined values?

It should be clear that one cause of an undefined value is a normal transition from one logic level to the other. These changes are clearly not discontinuous, but transition through the undefined region near V_h on their way from one valid value to the other. Although good design practices emphasize as quick a transition as possible, it is inevitable that every circuit segment in transition will spend at least some time in this undefined region. We recognize, of course, that this “uncertainty” is a momentary - a *transient* - phenomenon - waiting “a little longer” will result in a valid logic level

(0 or 1). There can be other causes of a transient visit to V_h . But the key term here is “transient”: the undefined status is dynamic. Given time, the circuit will resolve itself into a *steady state* valid level.

There are, however, causes that can result in a steady state undefined value. No matter how long we wait, the observed circuit value will never become a valid 0 or 1.

We’ll now look at both of these circumstances.

Circuit Delays - insufficient time to “settle out”

In CMOS circuits, no power is used in steady state conditions. Despite this principle, power consumption is one of the most urgent and continuing problems in CMOS design. Power consumed must be dissipated in the form of heat, necessitating special cooling arrangements. Laptop and handheld system battery life is inversely proportional to power consumption.

Power is used only during transitions from one logic state to another, and consists primarily of the charging and discharging of parasitic capacitance that is the inevitable result of placing independent conductors - and parallel elements of active devices like transistors - within very small distances of each other and other layers of the integrated circuit. As it is a normal design goal to run the circuit as fast as possible, this translates into as many logic transitions as possible per second, and, as an undesirable side effect, into increased power consumption. In fact, to achieve theoretically maximum speed, a circuit would potentially be in transition virtually all of the time.

During this charging and discharging of parasitic capacitance, logic levels transition from one state to another. During some of this time, inescapably, circuit output levels (and, consequently, input levels to following stages) are in this undefined area near V_h . In fact, the maximum clock speed at which a circuit stage may be run is determined by how long it takes the slowest signal in the worst case to leave this area and become recognizably a logic 0 or logic 1. We see, therefore, that the need to allow sufficient time for each value to reach defined levels - to leave the undefined

region near V_h and become distinguishably steady state - is the *de facto* determinant in the practical maximum clock speed of a circuit.

This cause of transient undefined logic levels is certainly the most common.

Races - may transition through V_h more than once

Due to differing delay times in paths within a circuit segment, the output value of that segment may transition through V_h multiple times. This condition is known variously as a “race” or as a “hazard”.[11, 12, 13] A simple example of a circuit with an evident race is shown in Figure 2.1:

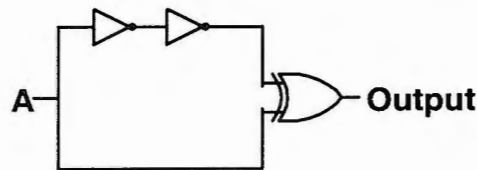


Figure 2.1: Simple Circuit with Inherent Race

In the static sense, the Output from this circuit will always be a logic level of 0. In the dynamic sense, however, it is clear that when A changes from $0 \Rightarrow 1$ or from $1 \Rightarrow 0$, the change takes longer to arrive at the Exclusive OR gate through the chain of two inverters than via the direct line. Thus, there is a small period of time during which one input to the gate differs from the other; yielding a logic level of 1 at the output.

The danger posed by races has little to do with the undefined region in V_h , however. The very fact of a “spurious” transition to a valid logic level may, when the signal is used as input to a sequential circuit, result in improper operation. We will return to the matter of races later in this work.

Noise

Another cause of dynamic values in the invalid range is noise[14]. Signal degradation or noise injected into a circuit may have the effect of causing logic levels to enter the undefined area near V_h . Of course, it may also cause a momentary transition to an

incorrect logic value in a valid range ($0 \Rightarrow 1$ or $1 \Rightarrow 0$). In this sense, it is similar in effect to a race. Noise may appear on the inputs to the circuit, or even on power supply lines, including V_{ss} and V_{dd} . Noise is by definition a transient phenomenon.

Defects

Under normal circumstances, a properly designed integrated circuit should never exhibit static logic values near V_h . However, fabrication problems or, less frequently, failures during service may result in defects affecting signal integrity, resulting in logic levels near V_h . [15] Such faults may be hard - caused by a permanent defect - or soft - caused by a sporadic event such as a radiation particle strike. [16] One type of defect, a bridge, is most likely to occur in data transmission busses. Another type, an open, may occur anywhere, but is most likely where minimum-width features are being used. Additionally, opens or shorts may also occur in active devices (transistors) on an integrated circuit [17, 18]; we'll refer to these problems collectively as "device faults".

Bridges

In an integrated circuit, a single transmission line typically transmits a single binary value - logic 0 or 1 - from one part of the circuit to another. As digital data is usually made up of several bits (a data word in modern microprocessors, for example, may be 16, 32 or 64 bits in width), several transmission lines must run in parallel to carry the full word of data. Thus is created a situation in which several transmission lines run for (comparatively) long distances in parallel paths. To minimize parasitic capacitance, these lines typically are composed of minimum width metal features. To minimize consumption of valuable silicon "real estate", they are usually spaced apart the minimum allowed by the fabricating technology being used. The significant proportion of space on many integrated circuits taken up by these data routing busses, combined with their minimum feature separation, results in a high feature "density" that increases the probability that a conducting defect will result in a resistive "short" between two (or more) adjacent lines, as illustrated in Figure 2.2.

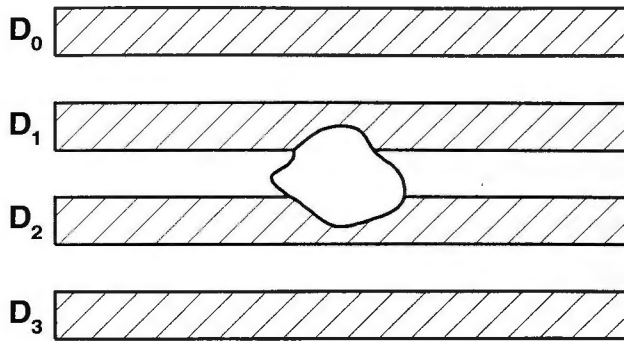


Figure 2.2: Physical Bridge (Short) Between Two Bus Lines

This fabrication defect could be a result of any of several problems, including contamination of the substrate during processing or a defect on the photo negative, or equivalent, used to form the features on the substrate. If the defect is of conducting material, the effect is to form a resistive short between data lines D_1 and D_2 , as shown in Figure 2.3.

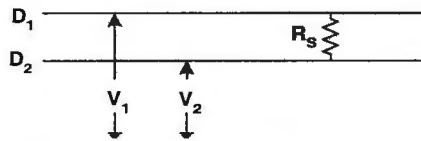


Figure 2.3: Resistive Equivalence of Bridge (Short) Between Two Bus Lines

The effect of this resistive short between data lines 1 and 2 on logic levels V_1 and V_2 depends on the “intended values” of V_1 and V_2 (V_{i1} and V_{i2} , respectively), as well as the resistance of R_S . Clearly, when $V_{i1} = V_{i2}$, there is likely to be no ill effect. When, however, $V_{i1} \neq V_{i2}$, the actual voltage values appearing as V_1 and V_2 will usually differ from their intended values, depending on the parameters of all circuitry attached to those two lines and, not insignificantly, the value of the shorting resistance, R_S . As R_S decreases,

$$|(V_1 - V_2)| \Rightarrow 0 \text{ volts}$$

, until, if R_S achieves a “dead short” ($R_S = 0\Omega$), V_1 and V_2 will exhibit close to the same value. If circuit parameters are reasonably similar for D_1 and D_2 , as is

particularly likely for a bus, the resulting value of both V_1 and V_2 are likely to be close to V_h for low values of R_S .

Opens

If a problem resulting from a bridge can be thought of as a "fight for possession" between two voltage sources, an open could be characterized as an absence of voltage sources. An open occurs under conditions of a break in a transmission line, as illustrated in Figure 2.4.

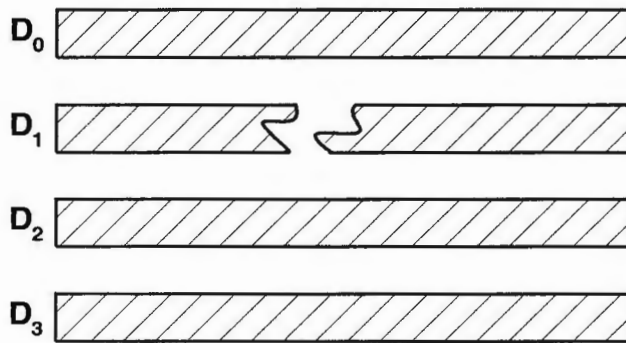


Figure 2.4: "Open" in Bus Line

Unlike a bridge, an open may affect only one transmission line, although in a bus structure, it has the potential for "opening" two or more adjacent lines. Since no interconnection is made with any other bus line, all such defects can be viewed as independent. Also, not all opens are total - a small amount of conductive material may still connect the two segments, which results in the open appearing as a resistor. In the most general case, then, an open can be diagrammed as shown in Figure 2.5.

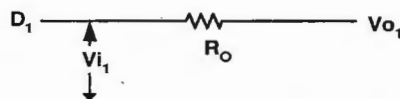


Figure 2.5: Resistive Equivalent to Open in Transmission Line

Also unlike a bridge, voltage levels on the driving side of the defect are not much affected. In Figure 2.5, V_{i1} will not be significantly affected by the break at R_O - except, perhaps, that those areas of the circuit will operate more quickly, as a result

of being disconnected from the load and parasitic capacitance associated with the circuitry “downstream” from the defect. Clearly, if R_O is low, there will be little or no impact on V_{O1} , while as $R_O \Rightarrow \infty$, V_{O1} approaches independence of V_{i1} . In this case, V_{O1} can take on any value at all, even one outside the normal logic range; V_{O1} is said to be “floating”.

Device faults

Transistor defects may result in the equivalent of an open or a short. Consider the simple 2-input NAND gate and its truth table in Figure 2.6.

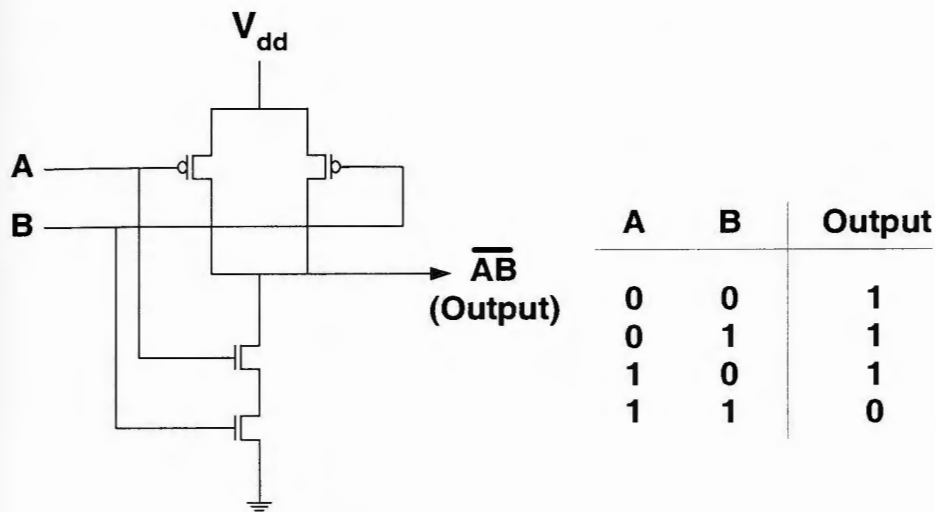


Figure 2.6: Simple NAND Circuit

Note that if we make the pfet transistor attached to the A input “shorted out” (Figure 2.7), there is always an effective connection between the source and the drain, as shown in Figure 2.7.

Note that in Figure 2.7 the value for the output of the circuit in the case that $A=1$ and $B=1$ is not obvious. In the normal NAND gate in Figure 2.6, both nfet transistors conducted and neither pfet transistor conducted, so a V_{ss} (Ground) logic level was connected to the output. With the defect of Figure 2.7, however, there is always a conducting path from V_{dd} to the output, so the circuit reduces in this case to the resistive network shown in Figure 2.8, where R_1 is the resistance exhibited

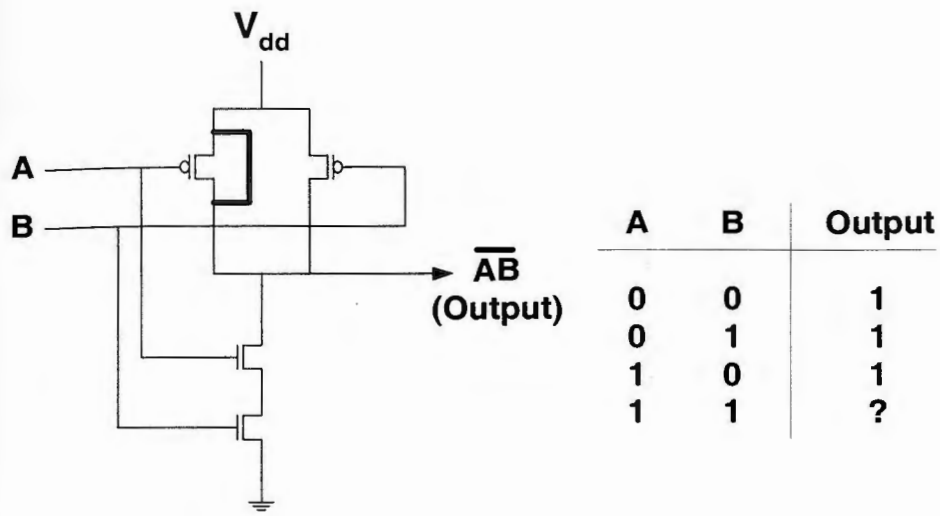


Figure 2.7: NAND Circuit with Shorted Transistor

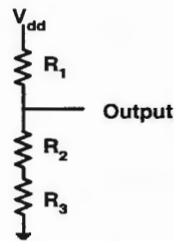


Figure 2.8: Resistive Network: Shorted NAND

by the shorted pfet transistor, and R_2 and R_3 are the resistances exhibited by the conducting nfet transistors. In this circumstance, the voltage presented at the output can be approximately determined by

$$\text{Output} = V_{dd} \left(\frac{R_2 + R_3}{R_1 + R_2 + R_3} \right)$$

and may or may not be in the vicinity of V_h .

When a transistor is open, results are different, and similar to a transmission line open, as shown in Figure 2.9.

It is most likely that the “floating” value shown for $A=0, B=1$ will actually simply maintain the last value displayed by the output, at least until the charge dissipates,

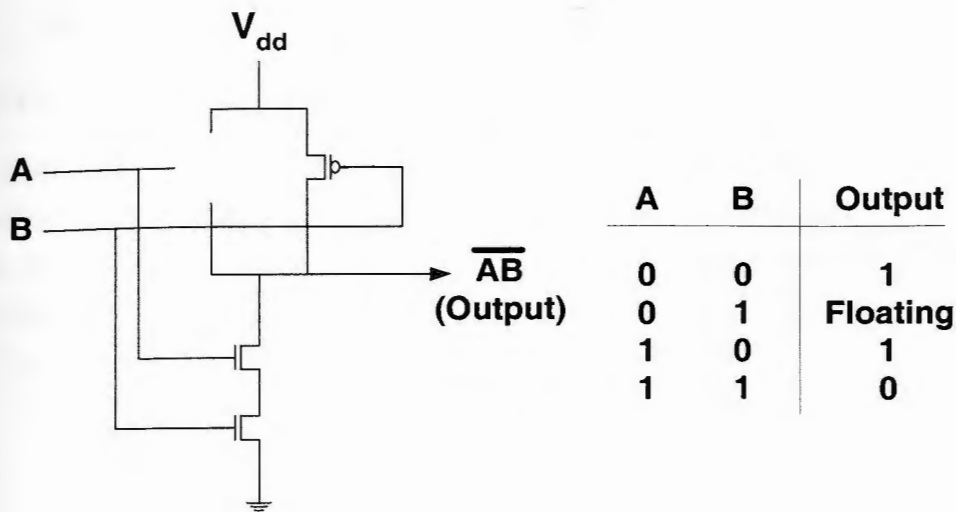


Figure 2.9: NAND Gate with Open Transistor

although a “race” condition could alter this. As an example of this hazard, consider a previous input/output set of $A=1, B=0 / \text{Output}=1$. If the transition to $A=0, B=1$ was not instantaneous, but instead went through the state $A=1, B=1 / \text{Output}=0$, then the output would likely continue to be 0 even after the input state changed to $A=0, B=1$.

Imperfect inputs to circuit

We have examined causes of the output of a combinational circuit falling in the undefined area around V_h . It is important to note that, when this happens, it can become a cause of the same phenomenon in later circuitry, as the output from a circuit is usually used as an input to another. Therefore, it is conceivable that an external input level presented to a circuit may fall in the area not clearly defined as a logic 0 or 1.

Other causes of an input signal falling in this area include electronic faults external to the integrated circuit, such as a bridge or open in a printed circuit board (PCB) or multi-chip module (MCM) transmission line, or noise. It might also be due to a problem at an original data source, such as a transmitting sensor having lost power or malfunctioning.

2.2.2 What are the effects of undefined values?

Conversion to either logic 1 or 0

These intermediate logic levels are considered non-desirable, and circuit design is intended to minimize their occurrence and persistence. As the input voltage to an inverter, for example, increases from 0 to 1, the output remains high until the input nears (ideally) V_h , and then makes as rapid a transition as possible to a low output state. The graph in Figure 2.10 illustrates this behavior.

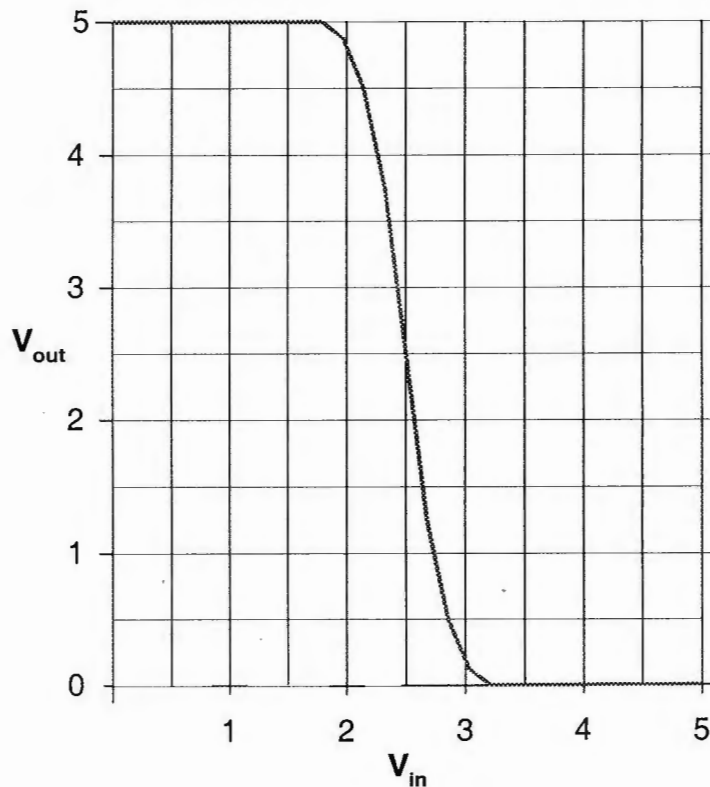


Figure 2.10: CMOS Inverter Transition

Note that an input of $1/3 V_{dd}$ (1.67 volts in Figure 2.10) does not result in an output of $2/3 V_{dd}$, as would be expected if this was a linear function. Instead, the output is very close to a logic 1. The design and technology used in the fabrication of CMOS integrated circuits makes this effect consistent. It is desirable to have as

sharp a transition as possible - an instantaneous transition from an output of 1 to an output of 0 (a square wave) might be desirable, although unattainable.

An input signal in the area very close to V_h , then, is placed in an effective position of unstable equilibrium as it and its "descendents" pass through successive stages of circuitry. If the first "stage" it encounters doesn't convert it to a logic 0 or 1, one of the following stages almost certainly will. It is therefore virtually guaranteed that an input to a set of successive inverters and gates will eventually be effectively converted to a logic level of 0 or 1.

But what determines which value that input (or its descendents) eventually takes on, and is it reliable?

Appears random overall - but really determined by fabrication conditions

In an ideal world, any value below V_h would tend toward a logic 0, and any value greater than V_h would tend toward a logic 1. Only a signal falling exactly at the infinitely small point V_h on the continuum from V_{ss} to V_{dd} would have an indeterminate fate. As the world of microelectronic fabrication is indisputably practical, rather than ideal, such is not the case. Minor differences in the process used to form elements across a wafer's surface make it inevitable that no two inverters, for example, will be truly identical. On a more general scale, differences in measured electronic parameters between different fabrication runs can provide clear proof of the inaccessibility of consistent device behavior near V_h . The graph in Figure 2.11, which is shown on a 1.1 volt to 3.9 volt x-axis for clarity, might represent area between parallel transition curves for two different inverters in 5-volt 2.0 micron CMOS; in fact, they are the extremes of transition curves for the *same* inverter across 31 fabrication runs, all of which were considered within tolerance by the foundry.

Note that even for truly identical inverters, Figure 2.11 shows clearly that the variation in electronic parameters for different fabrication runs alone provides for a range of 2.38 to 2.68 volts (fully 6% of a 5-volt scale) in driving voltage at the inverter transition point (vertical dotted lines on the graph). Considered another way, a driving voltage of exactly V_h (2.5 volts on this 5 volt scale) could yield as

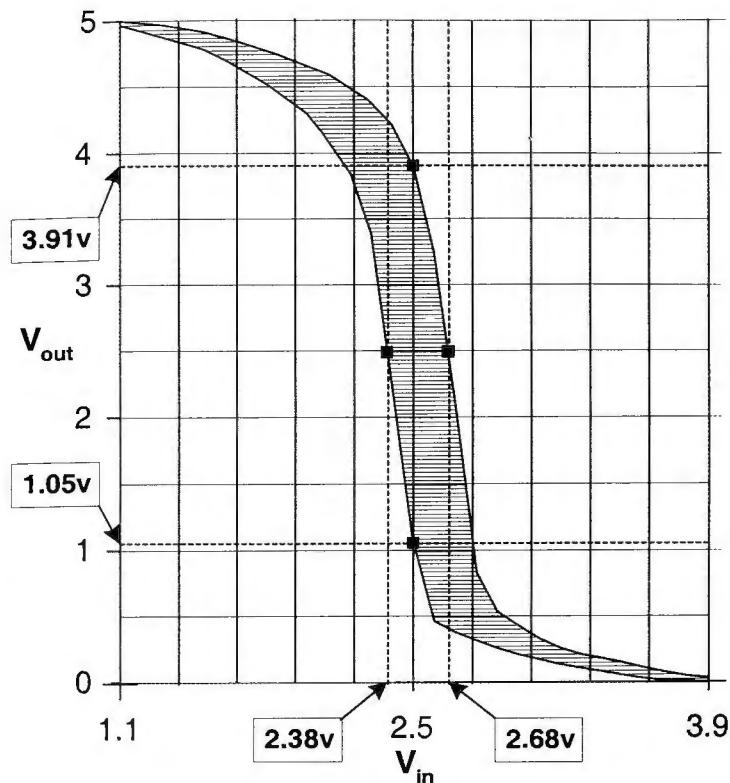


Figure 2.11: CMOS Inverter Transition across Fabrication Runs

little an output as 1.05 volts or as much as 3.91 volts (horizontal dotted lines on the graph).

Furthermore, consider the simple circuit segment in Figure 2.12.

If the input A is very close in value to V_h , we cannot even be certain that the values at the outputs of the two inverters will be or tend to the same logic level (0 or 1). Discrepancies of this type can clearly lead to unplanned behavior by the overall circuit. Consider the more specific example in Figure 2.13.

Logic would dictate that the output from the circuit in Figure 2.13 would always be zero, as the inputs to the Exclusive OR gate would always be identical. But consider the following case of an input value close to V_h ($V_{dd} = 5$ volts in this example). Due to fabrication differences, the chains of A1a through A1d and A2a through A2d may not come down on the same side of our unstable equilibrium. This

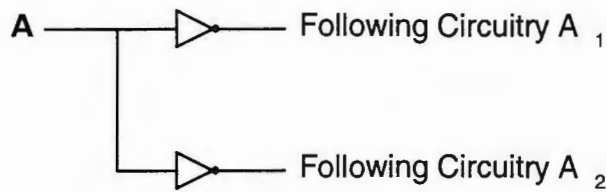


Figure 2.12: Input to Two Inverters

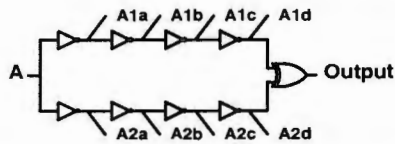


Figure 2.13: Two Inverter Chains to XOR

is illustrated in Figure 2.14.

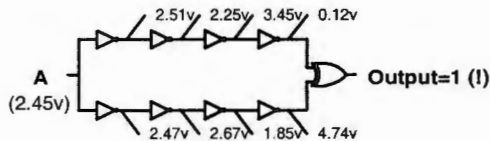


Figure 2.14: Operation of Two Inverter Chains to XOR

Although the example in Figure 2.14 is clearly contrived, it illustrates the potential dangers inherent in logic levels close to V_h .

2.2.3 How are they combated?

The effect of the problems described above is to make it desirable - even imperative - to avoid these effects.

The specific method(s) used to minimize the effects of uncertain logic levels, of course, depends on which of the causes applies. We shall see, however, that all have one characteristic in common: the aim of minimizing (ideally, eliminating) the occurrence of these conditions.

When circuit delay is cause

The approaches here can be summed up by the phrase, "give it more time." But in today's optimized and pipelined circuitry, there are a variety of techniques available to do this. The reader is referred to design texts [19, 20, 21, 14] for a full understanding of these methods, a few of which we will briefly summarize here.

Decrease clock rate to allow sufficient time

The simplest and most obvious approach is to slow the clock rate governing the circuit. With more time, the signals in the "problem segment" have an opportunity to "settle", resolving themselves into a set of valid logic levels. As a practical matter, however, as high a circuit speed as possible is highly desirable for competitive reasons, so other remedies are pursued when possible.

Optimize circuit elements for speed

Significant attention is paid in VLSI texts to consideration of circuit delays - their causes and design techniques to minimize them. The primary cause of delays is the charging and discharging of the parasitic capacitance which is a natural and inevitable consequence of placing conducting and semi-conducting elements in close proximity to each other. Beyond the parasitic capacitance, the effective resistance of both active (such as transistors) and passive circuit elements through which the capacitance must be charged or discharged is critical in determining the delay.

One obvious approach is to increase the size of the "driving" transistors, thereby decreasing its effective resistance, and enabling the more rapid charging or discharging of the capacitance of the circuit. This may be more complex than it appears, however, since increasing the size of the driving transistor(s) also increases the amount of parasitic capacitance in the circuit "feeding" the gate of the driving transistors, resulting in a slowdown in that segment of the circuit. There is therefore a "balancing act" inherent in the optimization of circuit elements.

Redesign pipelined circuits to redistribute delays

Modern circuits are frequently pipelined to increase speed. Briefly, pipelining as a technique takes a large, long delay, circuit (with a necessarily low clock speed) such as that illustrated in block form in Figure 2.15, and breaks up the work of the

circuit into several smaller circuits, each of which run at a much higher clock speed, as illustrated in Figure 2.16.

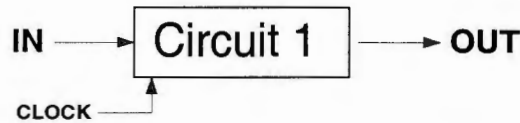


Figure 2.15: Non-Pipelined Circuit (Block Diagram)

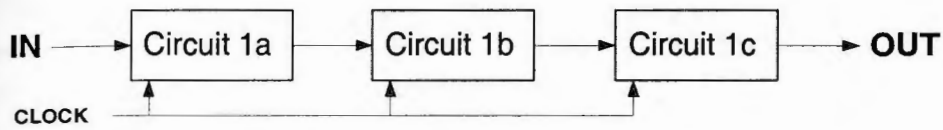


Figure 2.16: Pipelined Circuit (Block Diagram)

While a given piece of data takes as long (usually longer) to get through the circuit (latency), several other pieces of data are being processed through the pipeline simultaneously, resulting in a much higher throughput. In an ideal partitioning of the work of Circuit 1 above into Circuits 1a, 1b and 1c, the delay of each of the three pipeline “stages” would be one third the delay of the original, non-pipelined circuit, yielding a throughput of three times the original circuit. The attainment of such an ideal is unlikely in practice, however, but it is crucial to balance the pipeline stages as evenly as possible, as the maximum clock speed of the entire pipeline is determined by the worst-case delay of the slowest pipeline stage.

When race (hazard) is the cause

As mentioned earlier, races are already considered a potentially serious problem, not because the circuit spends more time in an undefined state, but because the transition through it to a valid logic state (although not necessarily the desired one) can occur more than once while a final value is being arrived at.

For our purposes, primarily concerned with problems resulting from the existence of undefined logic levels, this cause is not much different from the situation discussed above where simple circuit delay is the cause. Given time, a combinational circuit

subject to race conditions will eventually settle into a final, valid state. Nonetheless, we wish to make note of the fact that races in sequential or dynamic circuits can be a serious problem producing spurious results; we shall return to the subject later in this work.

When noise is the cause

Efforts in this area center on making the *noise margin* as great as possible. Weste and Eshraghian[14] describe noise margin as a parameter that “permits one to determine the allowable noise voltage on the input of a gate so that the output will not be affected”, and go on to recommend design goals in which “the transfer characteristic should switch abruptly.” A transition voltage near the midpoint of the logic range (near V_h) is also desirable; while, for example, increasing the voltage at which the transition takes place may raise the “low” noise margin, it will simultaneously *lower* the “high” noise margin, rendering the gate asymmetrically sensitive to noise.

When defect is the cause

A defect differs significantly from delay-based causes in that additional time will likely do little to change the result - the final resting state of the circuit may lie in the undefined area near V_h . Approaches toward mitigating this problem vary according to whether the defect is “hard” - caused by a manufacturing defect or later permanent damage - or “soft” - a temporary result of a event such as the strike of an alpha-particle. To this dichotomy we must add for completeness aging-based defects, such as the development of an open in a transmission line due to conductor migration and use-caused device shorts and opens.[22] This last class of defects resembles hard errors in their permanence, but differ in that they were not present at time of manufacture.

Hard manufacture-time error: testing procedures must detect

It is the aim of modern testing procedures to detect hard errors as part of the manufacturing/testing process. There are many testing methods which may be used

to confirm proper operation of a circuit, including boundary scan (a form of edge-pin testing), current-sensing (a higher than designed supply current may indicate a short in the circuit), and methods for getting to the “innards” of a fabricated circuit prior to final processing and packaging, such as “guided probe”, “electron-beam” and “bed-of-nails” testing.[23, 24, 25, 26, 27, 28, 29, 17, 18, 30] It is pointed out, however, that defects that are not strong enough to produce a logic error during testing (such as one that produced an intermediate logic level but one that barely resolves itself to the right value) cannot be detected with many standard tests.[31] It has been known for some time to test designers working with analog circuits that digital testing techniques accounted only for “catastrophic faults”, and not for the “out-of-specification” faults that occur as often.[32] Later work[33] pays some attention to analog effects of such faults in digital circuit testing.

Post-manufacture error: error-checking circuitry must detect and correct

Errors that are transient, or permanent errors that develop after the circuit is put into service, must be detected while normal operations are in progress. Simple techniques, such as including a parity bit in RAM arrays, may be used, or complex fault-tolerant methods applied.[34, 35, 36] All such approaches have costs associated with them, and what may be appropriate for a restricted subset of uses (long mission, high reliability applications such as a space probe) may not be cost-effective for most uses.

When imperfect input is the cause

Additional circuitry must be added to detect and sometimes correct this condition. The same fault-tolerant on-chip methods to detect error (dual-rail encoding and similar fault-tolerant methods) can be used between chips or assemblies.

2.3 An undefined value as information

We have seen that there are clear causes for undefined logic levels. Knowing that a logic level is undefined could be an indicator of one of these specific causes, dependent

on the environment and circumstances. Indeed, we must consider knowledge of an undefined logic level as information; in brief, the information is that *we do not know the proper value that this circuit is indicating*. Yet current practice is effectively to throw this information away - to never detect it and, instead, to avoid its occurrence (and/or its effects) to the extent possible. We design as if it's not there, and do what we need to do to increase the probability that the circuit comes down on the correct side of V_h . In circuits for high reliability applications, we have seen earlier, the possibility of incorrect results is accepted, and complex methods for detecting and correcting it (double rail encoding and the like) are employed where the cost can be justified.

How could such knowledge (that a value is in the undefined range) be of use in CMOS circuits? Earlier in this chapter, we looked at some of the causes that would result in a value being in this range. By specifying appropriate constraints, we should be able, in a practical sense, to use the existence of the condition of uncertain value to infer the active presence of the corresponding cause. For example:

- In a tested and “known good” circuit, information that a result is undefined could be used as an indication that more time is needed to allow the result to settle, or that a circuit failure has occurred.
- During operation of a tested and “known good” circuit designed to receive data (from an external source or from another area of the integrated circuit via bus lines), undefined values can be an indication of a transmission line or other failure, and point to the bits in which the failure exists.
- During the initial post-manufacture testing of a circuit, where both (1) adequate time has been provided for signal values to “settle” and (2) value injection functions of the testing equipment have been verified, the presence of an undefined logic level where none should exist can serve as an indicator of a physical defect.

The question occurs: what is required to fulfill the promise inherent in these uses of undefined logic levels as information? We can say immediately that two clear

requirements exist:

- A theoretical foundation must be established for the reliable and robust use of this information, unless it already exists in the literature.
- The condition must be detectable. There must be circuitry implemented at appropriate locations (dependent on the desired detection capabilities) to detect when a logic level is valid or invalid.
- Once a detection scheme has been implemented, appropriate circuitry must be present to make use of this new information in a meaningful and practical way.

We will consider these requirements in later chapters.

2.4 Summary

We have defined what we mean when we say a logic level is uncertain, undefined or invalid, and have provided for the purposes of this work a range $1/3V_{dd} \Rightarrow 2/3V_{dd}$.

We have further surveyed several causes of logic levels in this uncertain range, and briefly discussed measures typically taken in response to their potential existence.

It should be clear, notably, that design methods used to address this problem are of an “evade and avoid” character. There is no effort in the design to detect the condition; on the contrary, they seem to be considered a nuisance - a form of “non-information”, and therefore something to be minimized or corrected.

We briefly discussed the potential the detection of undefined values has for use in VLSI circuitry, based only on the inference that if the condition exists, a cause (or causes) is indicated.

In the following chapters, we shall consider not only the inference of the cause from the condition, but also other uses for this information.

Chapter 3

Binary Plus logic

In this chapter we shall define theoretically a new logic family, which we shall call “Binary Plus” logic. This family is similar to existing binary logic in that it is based on two valid values. It enhances the binary concept by adding the detection of undefined logic levels - states in which the true binary value cannot be reliably determined - and using that information to add capabilities unavailable to pure binary logic circuitry.

3.1 The detector

We begin by specifying the requirements for a functional unit to detect the presence of an undefined value.

Specific circuitry is needed to somehow measure the logic level on the input and make a determination as to within which range it falls, in accordance with Table 3.1:

Range	Zone
$V_{ss} \Rightarrow 1/3V_{dd}$	Valid 0
$1/3V_{dd} \Rightarrow 2/3V_{dd}$	Uncertain
$2/3V_{dd} \Rightarrow V_{dd}$	Valid 1

Table 3.1: Division of $V_{ss} \Rightarrow V_{dd}$ Logic Range into Zones

We can say that the boundaries between the zones are robust. They might vary significantly, while still maintaining confidence that, for example, an input on

or near the $1/3V_{dd}$ boundary will never be interpreted as a valid 1. We must, of course, remind ourselves of an earlier stated point - that there is no reason why these boundaries could not be set closer to (or farther from) V_h . Provided that they are not set excessively close to V_h , robustness should still be present. [What constitutes “excessively close”, in the presence of noise and other factors, must be left for the specific implementation designer.]

3.1.1 Not a new “value”

It should be noted here that dividing the $V_{ss} \Rightarrow V_{dd}$ range into three, rather than two, zones might be seen as creating a third “value” in a heretofore 2-value, or binary, scheme. Although that theme has been applied - to create ternary logic - this is not what we seek to do here. Ternary logic, in carrying three rather than two values in each signal, actually suffers from a worse form of the same uncertainty problem as CMOS binary logic circuitry. There are *two* zones of uncertainty in ternary logic - between the first and second values, and between the second and third.

The third zone we seek to create in the $V_{ss} \Rightarrow V_{dd}$ voltage continuum does not represent a new value. Instead, it establishes a signal of the existence of a condition. This signal can be conceptualized as an interdependent yet separate signal, as shown in Table 3.2:

Value	Binary (Value)	Uncertain (Signal)
V_{ss}	0	No
V_h	?	Yes
V_{dd}	1	No

Table 3.2: Implied Value and Signal

One advantage of this approach is that the two pieces of information (binary value and uncertainty signal) are encoded within one physical line. We will refer to a line carrying such a logic level to a detector as carrying *Zoned Binary* data. It is, in reality, no different than any line carrying binary data - it differs in that it is used as input to a detector designed to “decode” it.

3.1.2 Required products of the detection process

For the purposes of this work, we will now define a signal RDY such that:

$$RDY = \overline{\text{Uncertain}}$$

Conceptually, RDY, when true, indicates that the input value is in one of the two valid binary zones: $\{ < 1/3V_{dd}, > 2/3V_{dd} \}$.

We also wish to define signals which indicate the presence of a valid “0” and a valid “1”, effectively splitting the RDY signal into two: RDY_0 and RDY_1 . We shall see in Chapter 4 that it is most efficient to implement and use these signals as inverted forms. We therefore define signals XH and XL as follows:

- XH takes on a value of 0 only when the input to the detection circuitry is a valid 1. XH has a value of 1 under all other conditions.
- XL takes on a value of 1 only when the input to the detection circuitry is a valid 0. XL has a value of 0 under all other conditions.

We summarize in Table 3.3 the interrelationship of the signals we wish to be able to obtain from an input.

Input	RDY	XH	XL
0	1	1	1
Uncertain	0	1	0
1	1	0	0

Table 3.3: Relationship of Output Signals from Detector

We have defined signals that may be used to provide various sorts of detection of undefined values. We now proceed to develop the use of this detection information in Binary Plus logic.

3.2 Development of Binary Plus concepts

We require a more precise operational description of Binary Plus logic, which we give here:

- the logic is still two-valued, or binary, and
- logic gates are implemented so as to maintain the integrity of the additional zoned binary signal through the function of the logic gate to the output; that is, outputs become valid only when valid inputs constitute a sufficient Boolean condition for a known output, and are invalid at all other times.

3.2.1 A small step

We take a small step in the direction of Binary Plus logic by considering a rudimentary use of our detection capabilities as applied to binary logic.

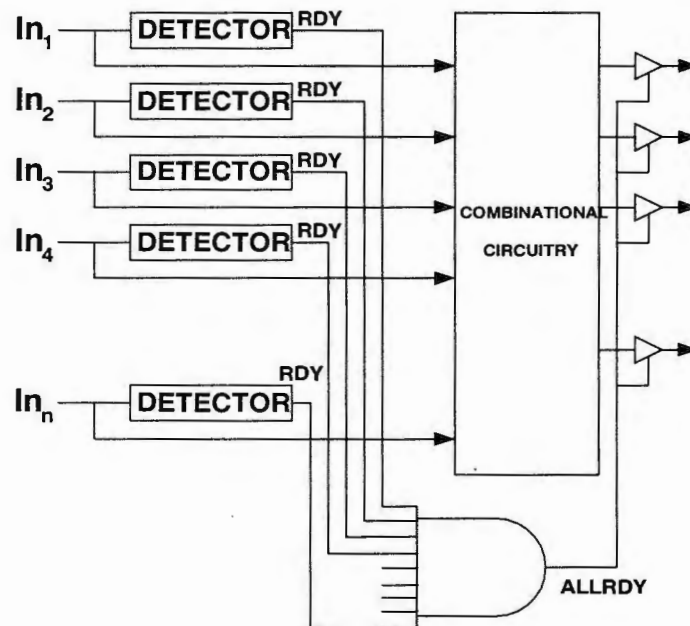


Figure 3.1: Prevention of Output Based on Uncertain Inputs

In Figure 3.1, we have placed tri-state buffers on the output(s) of the combinational circuitry that uses the inputs. Controlling the buffers with the ANDed RDY signals of our detectors, we prevent erroneous signals from being passed on to later circuits. We have satisfied, in a basic way, our requirement that the outputs be valid only when needed inputs are valid. In fact, *all* inputs must be valid in this case

in order that outputs become valid. Clearly this is a contrived example, *and* an imperfect one, too, for:

- the circuit has a clear hazard, in that the output tri-state buffers will likely be enabled *before* the newly valid inputs have had time to flow through the combinational logic block and reach their static value,
- efficiencies are disregarded, as in many implementations, not all inputs are critical to the output, depending on the values of those inputs at any given time, and
- we do not know what the outputs of the circuit will be when the tri-state buffers are *not* enabled, as they will be left floating.

3.2.2 Complete “Binary Plus” concept

The simplistic approach to ensuring that results have been generated using valid data that we discussed in section 3.2.1 can be extended to a far more powerful implementation.

We shall first develop an understanding of what it means when we say that “outputs become valid only when needed inputs are valid.” As an example, consider the truth table of a basic 2-input OR gate, as shown in Table 3.4.

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.4: 2-Input OR Gate Truth Table

We note that a 1 on either input (by extension, *any* input on an OR gate of more than two inputs) is a fully sufficient condition for a 1 appearing at the output. Conversely, a logic level of 0 must be applied to *all* inputs of the OR gate in order for a 0 to appear at the output.

To understand how these characteristics will point toward a better understanding of the Binary Plus concept, let us first, for clarity, extend Table 3.2 by defining our notation for zoned binary, as shown in Table 3.5.

Value	Binary (Value)	Uncertain (Signal)	Zoned Representation
V_{ss}	0	No	0
V_h	?	Yes	ϕ
V_{dd}	1	No	1

Table 3.5: Implied Value and Signal

We will be using the notational symbol ϕ to represent our uncertain zone in a zoned binary representation. It is important to remember, however, that this is not a true third value, but is instead *shorthand* for the combination of an unknown value and a known signal.

Now we expand the truth table of Table 3.4 to include new possibilities on the input, as shown in Table 3.6.

A	B	OR
0	0	0
0	ϕ	ϕ
0	1	1
ϕ	0	ϕ
ϕ	ϕ	ϕ
ϕ	1	1
1	0	1
1	ϕ	1
1	1	1

Table 3.6: Binary Plus 2-Input OR Gate Truth Table

Note the behavior we have specified for the gate when one or more of the inputs in ϕ . When one input is 1, it matters not whether the other input is 0, 1 or ϕ . The other input is *no longer critical*. As a 1 on *any* input of an OR gate is a necessary and sufficient condition for a 1 on the output, we do not have to be concerned whether the other input is even *known*.

There are two factors that separate this example from the rudimentary data application illustrated in Figure 3.1, and which therefore define the concept of Binary Plus:

- The concept of critical inputs for logic functions is taken into account in determining whether the output of the function can be considered valid. To rephrase, we take advantage of logic functions that do not require complete data for a valid output.
- The output of the function is also *zoned binary*.

Similarly, the Binary Plus AND gate also takes advantage of this conditional criticality of data inputs, as shown in Table 3.7.

A	B	AND
0	0	0
0	ϕ	0
0	1	0
ϕ	0	0
ϕ	ϕ	ϕ
ϕ	1	ϕ
1	0	0
1	ϕ	ϕ
1	1	1

Table 3.7: Binary Plus 2-Input AND Gate Truth Table

For completeness, we define the *Binary Plus* NOT in Table 3.8.

A	NOT
0	1
ϕ	ϕ
1	0

Table 3.8: Binary-Plus NOT Gate Truth Table

It should be mentioned that there are functions for which no advantage of conditional input criticality can be obtained. For example, consider the *Binary Plus* exclusive OR (XOR) table in Table 3.9.

A	B	XOR
0	0	0
0	ϕ	ϕ
0	1	1
ϕ	0	ϕ
ϕ	ϕ	ϕ
ϕ	1	ϕ
1	0	1
1	ϕ	ϕ
1	1	0

Table 3.9: Binary-Plus 2-Input XOR Gate Truth Table

Although the Binary Plus XOR gate maintains the integrity of the invalid input signal, it can derive no performance advantage from input value patterns, as *all* inputs are always critical in an XOR gate.

3.3 Binary Plus logic specifications

Before formulating the method we will use to create Binary Plus gates, it will be useful to review some basic topics in VLSI CMOS design. We can then proceed to develop the basic implementation theory of the Binary Plus logic family.

In doing so, we must remember that, for inputs in the valid ranges, the operation of such gates must be exactly equivalent to its implemented Boolean function. For inputs not in one of the two valid ranges, the gate must behave differently: taking the logic function being implemented into effect, the gate must return a valid output or an output reliably within the invalid range, preferably as close to V_h as possible.

We shall first develop the specification intuitively for understanding. We shall then more formally extend the design technique to the general or complex gate.

3.3.1 Complementary logic

In standard CMOS complementary circuit design, the *pfet* network for a logic function is the complement, or dual, of the *nfet* network. The arrangement of these

networks is shown in Figure 3.2. The *pfet* network connects the output to V_{dd} when the inputs warrant a logic 1 output; its complement, the *nfet* network, connects the output to Ground (V_{ss}) when the inputs warrant a logic 0 output.

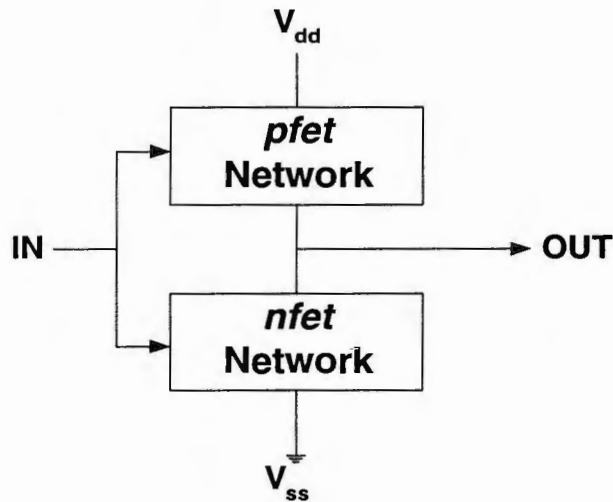


Figure 3.2: Complementary Logic

In Boolean logic, saying that the inputs do not warrant a logic 1 output is the same as saying they do warrant a logic 0 output - the output from a binary gate is a dichotomy. Therefore the *pfet* network and *nfet* network are true complements of each other.

Binary Plus: not quite complementary

For convenience, Table 3.6 is reprinted as Table 3.10. This Binary Plus OR gate truth table shows the required zoned binary output for each possible input state.

Since Binary Plus gates must exhibit a *three state* output, it follows that the *pfet* network and *nfet* network in such a gate cannot be true complements of each other. Yet the same Boolean logic function must be realized. How are we to implement a gate in the face of this seeming contradiction?

A	B	OR
0	0	0
0	ϕ	ϕ
0	1	1
ϕ	0	ϕ
ϕ	ϕ	ϕ
ϕ	1	1
1	0	1
1	ϕ	1
1	1	1

Table 3.10: Binary Plus 2-Input OR Gate Truth Table

3.3.2 Intuitive development

In our rudimentary example in Figure 3.1, we used the RDY signals from the detectors that receive the input for “pre-processing”. We shall now rely on the other signals - XH and XL - we specified for our detector outputs. Table 3.3 is reproduced here as Table 3.11 for reference.

Input	RDY	XH	XL
0	1	1	1
ϕ	0	1	0
1	1	0	0

Table 3.11: Relationship of Output Signals

If we now consider the *pfet* and *nfet* networks separate entities whose function is to pull up or down, respectively, the output line, a solution is possible. Table 3.12 specifies the conditions in the *pfet* and *nfet* networks which must be met in order that specified outputs will appear.

Remembering that a logic 0 input to the base of a *pfet* will cause it to conduct, we wish to apply inputs of logic level 0 to the *pfet* network *only* when that level results from a valid input to the circuit - that is, when the input driving the detector is in the valid logic 1 range. Examining Table 3.11, we see that output “XH” meets this requirement. Output “XL” does not, as it will display a logic 0 when the input

Output (A+B)	<i>pfet</i> network	<i>nfet</i> network
0	Not Conducting	Conducting
1	Conducting	Not Conducting
ϕ	Not Conducting	Not Conducting

Table 3.12: *fet* Network States vs. Zoned Output

is *either* 1 or ϕ . Therefore, we must connect "XH" outputs to the *pfet* network.

Similarly, noting that a logic 1 input to the base of a *nfet* will cause it to conduct, we wish to apply inputs of logic level 1 to the *nfet* network *only* when that level results from a valid input to the circuit - that is, when the input driving the detector is in the valid logic 0 range. Examining Table 3.11, we see that output "XL" meets this requirement. Output "XH" does not, as it will display a logic 1 when the input is *either* 0 or ϕ . Therefore, we must connect "XL" outputs to the *nfet* network.

Figure 3.3 shows this modification.

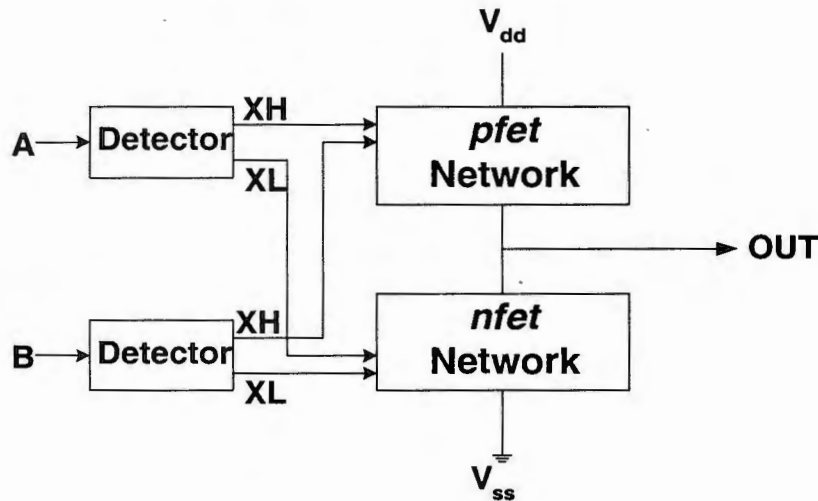


Figure 3.3: Binary Plus Gate

It is clear that we have handled the conditions under which the output should be 0 or 1. However, the output line *floats* when the conditions for an output of 1 or 0 are not present - resulting in *neither* the *pfet* network nor the *nfet* network conducting. The result of this would be that the gate would tend to display the last valid 0 or 1

output level. To ensure this does not occur when an output state of ϕ is appropriate, we can “center” the output when it would otherwise be floating, creating the circuit shown in Figure 3.4.

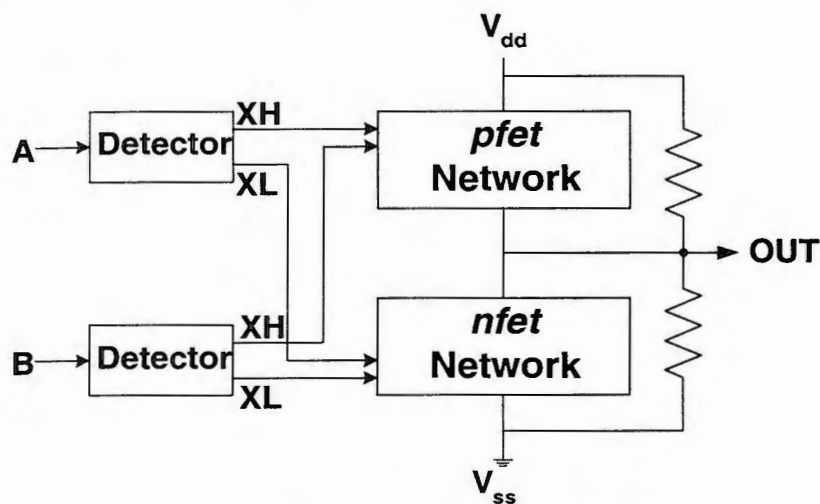


Figure 3.4: Binary Plus Gate with Float Centering

The effect of the resistors that “center” the output value in event of a floating condition can be simulated in CMOS circuitry using weak, always-conducting transistors. A disadvantage of this approach is that these weak devices are always conducting, resulting in continuous power dissipation, not a desirable condition. We shall see in Chapter 5 how a “dynamic” approach alleviates this problem.

A note on complemented inputs

In the preceding development, we have said that the “XH” outputs of the detector should be used as inputs to the *pfet* network, as that output, in contrast to the “XL” output, displays a logic 0 (needed to make a *pfet* conduct) only when the input to the detector is a valid 1. If, however, it is desired to create a complex gate in which some of the inputs must be inverted within the gate and used in that form, the approach must be adjusted, as shown in Figure 3.5.

To make it clear why we now route the complemented “XH” outputs to the *nfet* network and the complemented “XL” outputs to the *pfet* network, we now

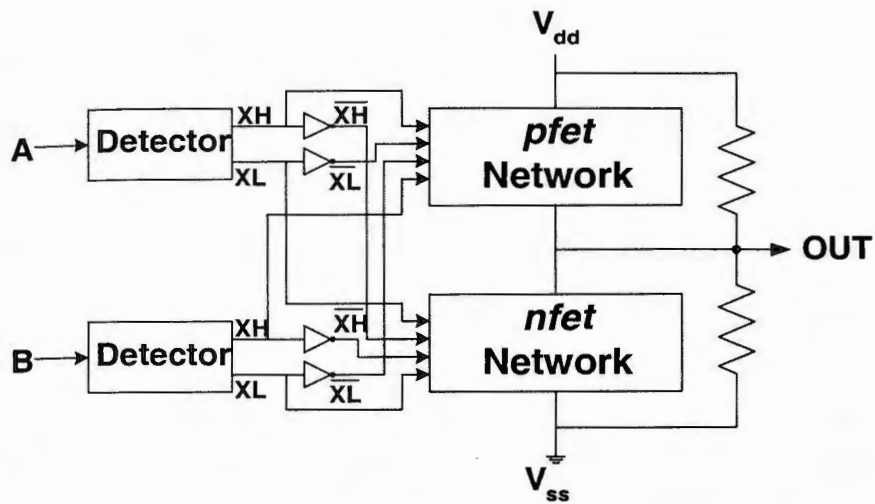


Figure 3.5: Binary Plus Gate including Complemented Inputs

expand Table 3.11 to show the internally complemented values of Figure 3.5, yielding Table 3.13.

Input	RDY	XH	XL	\overline{XH}	\overline{XL}
0	1	1	1	0	0
ϕ	0	1	0	0	1
1	1	0	0	1	1

Table 3.13: Relationship of Output Signals, Including Inverted

It is now obvious that the criteria for selecting the output to be used as input to the *pfet* network is reversed by internal complementing. That is, it is the complemented XL that takes on a value of logic 0 unambiguously, and should therefore be used as input to the *pfet* network. By the same reasoning, it is the complemented XH which should be used as input to the *nfet* network.

Elimination of races

As no Binary Plus logic gate can display any valid logic level on its output until the inputs have reached a *necessary and sufficient condition* for that output (which implies that the later arrival of a previously unknown input *cannot* change the output),

and provided that all such inputs shall be, in turn, zoned binary inputs conditioned by previous Binary Plus or equivalent "protected" sources, it follows - and will be proven later in this chapter - that races cannot occur in properly functioning Binary Plus logic stages.

3.3.3 Formal development

We begin by defining "zoned binary" more formally:

Definition 3.1 *Zoned binary is the combination of a binary value and a signal, carried on the same line. The binary values are 0 and 1, and the signal, which is asserted when the value reaches an indeterminate state between 0 and 1, the width of which is determined by the implementer, is termed ϕ , and represents that the value is unknown.*

We now proceed to define Binary Plus logic.

Definition 3.2 *A Binary Plus logic gate is one that accepts zoned binary inputs ($\{ 0, \phi, 1 \}$), and delivers outputs that are (1) logic level 1 when the set of valid inputs constitutes a sufficient condition for an output of 1 under the implemented Boolean function, (2) logic level 0 when the set of valid inputs constitutes a sufficient condition for an output of 0 under the implemented Boolean function, and (3) ϕ under all other conditions.*

Before we can proceed to Binary Plus gate construction, we must define the term "similarly constructed conventional binary gate":

Definition 3.3 *A "similarly constructed conventional binary gate" is a conventional binary gate whose pfet and nfet networks have been designed under the assumption that the inputs will be inverted.*

We are now ready to define gate construction in the form of a theorem.

Theorem 3.1 *A Binary Plus gate constructed by connecting the “XH” outputs of the input detectors (for complemented inputs the “XL” outputs of the detectors) to the inputs of a pfet network equivalent to the pfet network for a similarly constructed conventional binary gate generating the same Boolean function, and the “XL” outputs of the input detectors (for complemented inputs the “XH” outputs of the detectors) to the inputs of an nfet network equivalent to the nfet network for a similarly constructed conventional binary gate generating the same Boolean function, and in which a centering method is used to set floating outputs to ϕ , will display outputs appropriate to the implemented Boolean function when valid inputs constitute a sufficient condition for that output under the Boolean function, and will display a ϕ output in all other cases.*

Proof: Suppose that there is a Binary Plus logic gate that, when the “high” detector outputs (“XH” for normal and “XL” for internally complemented) are connected to a pfet network equivalent to the pfet network for a similarly constructed conventional binary gate generating the same Boolean function, and the “low” detector outputs (“XL” for normal and “XH” for internally complemented) are connected to an nfet network equivalent to the nfet network for a similarly constructed conventional binary gate generating the same Boolean function, and a centering method is used to ensure that floating outputs are brought to ϕ , does *not* display the proper zoned binary output. Then either (1) the pfet network is pulling the output high when the Boolean function does not specify it, (2) the pfet network is not pulling the output high when the Boolean function does specify it, (3) the nfet network is pulling the output low when the Boolean function does not specify it, (4) the nfet network is not pulling the output low when the Boolean function does specify it, (5) the output is not being set to ϕ when neither conditions for a logic 1 output nor a logic 0 output are met, or (6) the output is being set to ϕ when sufficient conditions for a logic 1 output or a logic 0 output are being met.

If (1), and since the “XH” inputs (“XL” inputs for complemented inputs) are identical to those in a similarly constructed conventional binary gate generating the same Boolean function, then the pfet network is conducting when the pfet network

of a similarly constructed conventional binary gate would not. Therefore the *pfet* network is not equivalent to the *pfet* network in a similarly constructed conventional binary gate generating the same Boolean function, which contradicts the initial assumption.

If (2), and since the "XH" inputs ("XL" inputs for complemented inputs) are identical to those in a similarly constructed conventional binary gate generating the same Boolean function, then the *pfet* network is failing to conduct when the *pfet* network of a similarly constructed conventional binary gate would. Therefore the *pfet* network is not equivalent to the *pfet* network in a similarly constructed conventional binary gate generating the same Boolean function, which contradicts the initial assumption.

If (3), and since the "XL" inputs ("XH" inputs for complemented inputs) are identical to those in a similarly constructed conventional binary gate generating the same Boolean function, then the *nfet* network is conducting when the *nfet* network of a similarly constructed conventional binary gate would not. Therefore the *nfet* network is not equivalent to the *nfet* network in a similarly constructed conventional binary gate generating the same Boolean function, which contradicts the initial assumption.

If (4), and since the "XL" inputs ("XH" inputs for complemented inputs) are identical to those in a similarly constructed conventional binary gate generating the same Boolean function, then the *nfet* network is failing to conduct when the *nfet* network of a similarly constructed conventional binary gate would. Therefore the *nfet* network is not equivalent to the *nfet* network in a similarly constructed conventional binary gate generating the same Boolean function, which contradicts the initial assumption.

If (5), since a centering method is being used to set all floating outputs to ϕ , therefore the output line must not be floating. If this is true, then either or both of the *pfet* network and the *nfet* network are conducting when input conditions do not warrant it. See (1) and (3) above for refutation.

If (6), since a centering method is being used that can set only floating outputs to ϕ , therefore the output line must be floating. If this is true, then either the *pfet*

network or the *nfet* network are not conducting when input conditions warrant it. See (2) and (4) above for refutation.

Q.E.D.

3.3.4 Binary Plus and races

We wish to prove that combinational blocks of Binary Plus logic, as defined, are free from races (hazards). We begin by defining the input conditions that must exist:

Definition 3.4 *A Binary Plus compatible source is a source of a single binary value, encoded in zoned binary, in which the source remains in the ϕ zone until its final, valid value is known, at which point it transitions to that value and remains there for the duration of the Binary Plus evaluation phase.*

Intuitively, the requirement for a Binary Plus compatible source would be satisfied by a tri-stated binary source, in which the tri-state buffer is not enabled until the value it will release to the Binary Plus logic block is static, and which employs a circuit mechanism to ensure that floating outputs to the logic stage are “centered” to ϕ . The term *Binary Plus evaluation phase* will be defined shortly.

We proceed to define a *Binary Plus logic stage* and a *Binary Plus evaluation phase*:

Definition 3.5 *A Binary Plus logic stage is a combinational logic block, consisting solely of Binary Plus gates, and obtaining all inputs from Binary Plus compatible sources.*

Definition 3.6 *A Binary Plus evaluation phase defines the time period over which a Binary Plus logic stage evaluates its inputs from Binary Plus compatible sources and produces outputs. Prior to its start, all input sources, outputs and intermediate results must be at ϕ . The phase begins when the first valid input is released into the stage and ends when all outputs from the stage reach valid values.*

Theorem 3.2 *A properly operating Binary Plus gate, operating on inputs from Binary Plus compatible sources, is free from internal races over the duration of its Binary Plus evaluation phase.*

Proof: Suppose that there exists a properly functioning Binary Plus gate, operating on inputs from Binary Plus compatible sources, that, over the duration of a Binary Plus evaluation phase, exhibits an output race - that is, its output changes from ϕ to a valid binary value and then changes back to ϕ or through ϕ to the opposite binary value. We know by definition of a Binary Plus compatible source that no input value will change from a valid binary value to ϕ or the opposite valid binary value. We also know by the definition of a Binary Plus gate that the initial transition of the output from ϕ to a valid binary value will occur *only* when a necessary and sufficient condition for that output in a similarly constructed conventional binary gate generating the same Boolean function has been reached, turning on conductivity of either the *pfet* or *nfet* network. As only a change of a critical gate input from a valid binary value to some other state (ϕ or the opposite valid binary value) could cause a *pfet* or *nfet* network to *stop* conducting, thereby changing the output state, we know that (1) in such a case, the inputs to the circuit have changed from a valid binary value to another state, contradicting the definition of a Binary Plus compatible source, or (2) the *pfet* network, *nfet* network, or "centering" circuitry is malfunctioning, contradicting the assumption of the theorem that the gate is "properly operating".

Q.E.D.

Theorem 3.3 *A properly functioning Binary Plus logic stage will be free from races during its Binary Plus evaluation phase; that is, once an output from a Binary Plus logic stage transitions from a ϕ state to a valid binary output state, there will be no further change in that output for the remainder of the evaluation phase.*

Proof: Suppose that there is a properly operating Binary Plus logic stage whose output is observed to transition from a ϕ state to a valid binary value, and then to some other state over the duration of its Binary Plus evaluation phase. Then either:

(1) One or more inputs to the Binary Plus logic stage have changed from a valid binary value to or through a ϕ state, (2) a Binary Plus logic gate receiving inputs from Binary Plus compatible sources or other Binary Plus logic gates is providing an intermediate result that varies in the manner described to later Binary Plus logic gates that themselves generate intermediate results, (3) the final Binary Plus logic gate is directly generating the suspect output from Binary Plus compatible sources or other Binary Plus logic gates, or (4) there is a sequential dependency in the Binary Plus logic stage.

If (1), then the source of the signal is either not a Binary Plus compatible source as defined, or it is not properly functioning. Either or both of these contradict the assumptions of the theorem.

If (2), since outputs from either properly functioning Binary Plus gates or properly functioning Binary Plus compatible sources cannot exhibit the observed behavior, one of these sources is malfunctioning, which contradicts the assumptions of the theorem.

If (3), the argument from (2) applies.

If (4), as a Binary Plus logic stage is defined to be a *combinational* construct, sequential operation contradicts the assumptions of the theorem.

Q.E.D.

3.4 Summary

This chapter has defined, intuitively and formally, Binary Plus logic. We have seen that Binary Plus logic *is* a binary logic, for, by definition, when critical input values are valid, the product is identical to what it would be if processed by Boolean binary logic.

The characteristic that distinguishes Binary Plus logic from classic binary logic is its use of zoned binary, wherein there is a third state between a binary 0 and binary 1. This state is not a new value, but instead represents a *signal* that the value is unknown. Binary Plus logic maintains the integrity of zoned binary through its gates, implying that an output remains in the unknown range, represented by the

zoned binary notation ϕ , until inputs defining a critical set for a valid output have themselves become valid binary zeros or ones.

The design characteristics of Binary Plus logic gates have been defined (and formally proven) to include connection of detector outputs to the *nfet* and *pfet* networks of the gate, while the details of detector and gate design have been left for Chapter 4.

The Binary Plus logic stage has been defined, and formally shown to be immune from races.

Chapter 4

Design and Implementation

In this chapter we shall examine the design considerations and methods employed in the creation of Binary Plus gates. The design of a detector for zoned binary is discussed in detail.

We shall then proceed to briefly discuss some rudimentary applications for the concepts embodied in zoned binary and Binary Plus logic, discussions that will motivate our in-detail look at two applications areas discussed in Chapters 5 and 6.

Finally, introductory information on a fabricated proof-of-concept integrated circuit will be given, to include testing of elementary detection concepts and Binary Plus gates.

4.1 Detector design

The requirements for our detector as described in Chapter 3 allow us to draw an initial block diagram for the required detector (see Figure 4.1).

Clearly, there must be a form of voltage comparison taking place in order to determine in which zone the input exists at any moment.

While we could use a scheme that compares a logic level to two reference voltages, either supplied externally or generated in some way internal to the integrated circuit, it was desired to use a simple method, not using approaches thought of as “analog”. Consequently, a novel method of voltage comparison was devised that, by itself,

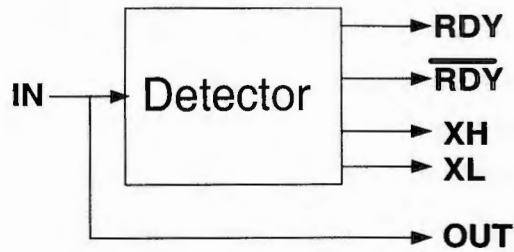


Figure 4.1: Detector: Simple Block Diagram

requires no more than the standard supplies for V_{ss} (ground) and V_{dd} .

No claim is made that the comparison method chosen is the most efficient; designing the fastest or most space-efficient detector was not an aim of this work. It is simply a demonstration that one need not have reference voltage supplies available to implement this concept.

The design approach is suggested by an observation made in Section 2.2.2 of this work. Inverter behavior - specifically the transition voltage - can vary from V_h to a certain degree based on fabrication variability. If it is possible to vary the transition voltage purposely, then one could devise a zone detector as shown in Figure 4.2.

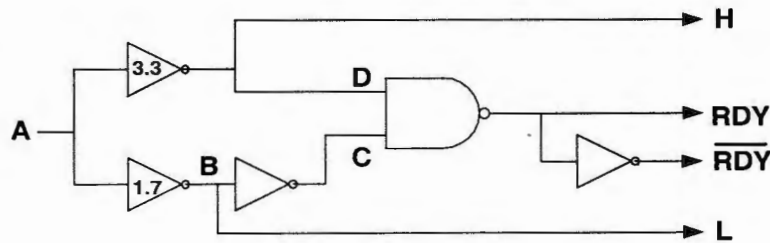


Figure 4.2: Detector: with Varied Transition Voltage Inverters

In Figure 4.2, a 5-volt supply for V_{dd} (relative to V_{ss}) is assumed, and therefore the desired $1/3 V_{dd}$ transition point occurs at approximately 1.7 volts, and the $2/3 V_{dd}$ transition point occurs at approximately 3.3 volts. The same scheme should scale for any supply voltage, provided care is taken to ensure that the desired transition voltage of the inverter does not approach the threshold voltage of either transistor; the values for a 5 volt supply are shown in this and following figures since that is the

supply voltage for the proof of concept circuit discussed later in this work.

The two inverters shown with “3.3” and “1.7” inscribed within their symbols have been designed by some as yet undiscussed means to have transition voltages of 3.3 volts and 1.7 volts, respectively. Note the behavior of the inverters - and the rest of the circuit shown in Figure 4.2 - for the three zones of the $V_{ss} \Rightarrow V_{dd}$ range shown in Table 4.1:

A	A (on 5v scale)	B (XL)	C	D (XH)	RDY
$V_{ss} - 1/3V_{dd}$	0v - 1.6v	5v	0v	5v	5v
$1/3V_{dd} - 2/3V_{dd}$	1.6v - 3.3v	0v	5v	5v	0v
$2/3V_{dd} - V_{dd}$	3.3v - 5v	0v	5v	0v	5v

Table 4.1: Inverter Pair Behavior

If we can design and fabricate inverters to have these transition points, then it becomes practical to decode the input value into zones without the presence of supplied reference voltages. We shall now proceed to derive the design equations for such inverters.

4.1.1 The design equations

A basic inverter consists of one *pfet* transistor and one *nfet* transistor, arranged as shown in Figure 4.3.

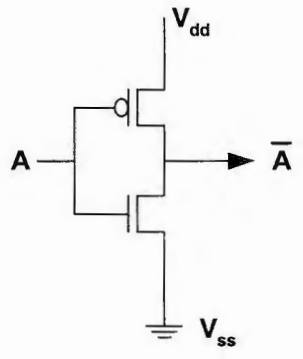


Figure 4.3: Basic Inverter Design

Weste and Eshraghian[14] discuss the electronic characteristics of the inverter in detail, and the reader is referred to that text for an in-depth treatment. They note that at the transition point of the inverter, both the *pfet* and *nfet* transistors are in a state of saturation, and that the saturation currents for the two transistors are given by:

$$I_{ds_p} = \frac{1}{2}\beta_p(V_{in} - V_{dd} - V_{t_p})^2$$

$$I_{ds_n} = \frac{1}{2}\beta_n(V_{in} - V_{t_n})^2$$

where:

$$\beta_n = \frac{\mu_n \varepsilon}{t_{ox}} \left(\frac{W_n}{L_n} \right)$$

$$\beta_p = \frac{\mu_p \varepsilon}{t_{ox}} \left(\frac{W_p}{L_p} \right)$$

and:

V_{in} = input voltage to the inverter

V_{t_n} = threshold voltage of *nfet* transistor

V_{t_p} = threshold voltage of *pfet* transistor

μ_n = mobility of electrons

μ_p = mobility of holes

W_n = channel width of *nfet* transistor

W_p = channel width of *pfet* transistor

L_n = channel length of *nfet* transistor

L_p = channel length of *pfet* transistor.

Weste and Eshraghian[14] then derive an expression for the transition point of the inverter (V_{in}) by noting that, in the inverter,

$$I_{ds_p} = -I_{ds_n}$$

which yields:

$$V_{in} = \frac{V_{dd} + V_{t_p} + V_{t_n} \sqrt{\frac{\beta_n}{\beta_p}}}{1 + \sqrt{\frac{\beta_n}{\beta_p}}} \quad (4.1)$$

Assuming for approximation purposes that $V_{t_n} = -V_{t_p}$, and setting $\beta_n = \beta_p$, they obtain:

$$V_{in} = \frac{V_{dd}}{2}$$

, establishing that, in the ideal case and with the lengths and widths of the *pfet* and *nfet* transistors in an appropriate ratio, the transition point of the inverter will be V_h .

As we wish to derive an expression for the design-modifiable characteristics of the *pfet* and *nfet* transition voltages as a function of the desired transition voltage V_{in} , we rearrange 4.1 appropriately and obtain:

$$\frac{\beta_n}{\beta_p} = \left(\frac{V_{dd} + V_{t_p} - V_{in}}{V_{in} - V_{t_n}} \right)^2 \quad (4.2)$$

as our expression for the *nfet:pfet* ratio of the betas of the transistors.

Our aim now becomes expressions for the size of the *nfet* or *pfet* transistors as functions of the other device's size and the *nfet:pfet* ratio of the betas of the transistors in 4.2 above. For clarity, we define:

$$R = \frac{\beta_n}{\beta_p} \quad (4.3)$$

as our term for the *nfet:pfet* ratio of the betas of the transistors. We recall from [14] that:

$$\beta_n = \frac{\mu_n \epsilon}{t_{ox}} \left(\frac{W_n}{L_n} \right)$$

and

$$\beta_p = \frac{\mu_p \epsilon}{t_{ox}} \left(\frac{W_p}{L_p} \right)$$

, so we will also define ratio terms G_n and G_p such that:

$$G_n = \frac{W_n}{L_n} \quad (4.4)$$

and

$$G_p = \frac{W_p}{L_p} \quad (4.5)$$

Restating 4.3 above:

$$R = \frac{\beta_n}{\beta_p} = \frac{\mu_n \varepsilon G_n}{t_{ox}} \cdot \frac{t_{ox}}{\mu_p \varepsilon G_p} = \frac{\mu_n G_n}{\mu_p G_p} \quad (4.6)$$

Obtaining expressions for G_n and G_p :

$$G_n = \frac{R G_p \mu_p}{\mu_n} \quad (4.7)$$

and

$$G_p = \frac{G_n \mu_n}{R \mu_p} \quad (4.8)$$

We have in 4.7 and 4.8 expressions for the required geometry of the *nfet* and *pfet* transistors, in terms of the required beta ratio, the geometry of the other transistor, and two fabrication parameters. If we further wish to assume equal channel lengths L_n and L_p , and referring to 4.4 and 4.5 we have:

$$W_n = \frac{R W_p \mu_p}{\mu_n} \quad (4.9)$$

and

$$W_p = \frac{W_n \mu_n}{R \mu_p} \quad (4.10)$$

Finally, eliminating our convenience terms R and G completely by remembering from 4.2 and 4.3 that:

$$R = \frac{\beta_n}{\beta_p} = \left(\frac{V_{dd} + V_{t_p} - V_{in}}{V_{in} - V_{t_n}} \right)^2$$

, we can now state complete expressions for the width of the *nfet* and *pfet* transistors:

$$W_n = \frac{\left(\frac{V_{dd}+V_{tp}-V_{in}}{V_{in}-V_{tn}}\right)^2 W_p \mu_p}{\mu_n} \quad (4.11)$$

and

$$W_p = \frac{W_n \mu_n}{\left(\frac{V_{dd}+V_{tp}-V_{in}}{V_{in}-V_{tn}}\right)^2 \mu_p} \quad (4.12)$$

The expressions in 4.11 and 4.12 become the design equations for sizing the active elements of an inverter to achieve a specified transition point. This makes it possible to create the detector circuit shown in Figure 4.2.

4.2 Binary Plus gate design

Binary Plus gate design was described and proven, in the general case, in Chapter 3. Now we will look at design as applied to a specific gate.

The detector design shown in Figure 4.2 provides the needed XH and XL signals for gate design. Consider, however, that we do not need a RDY signal, and can therefore dispense with that circuitry from our original detector design. The inverter pair alone provides us with the needed XH and XL signals.

We now can see why XH and XL were defined in Chapter 3 as inverted versions of the input - they can be easily generated through the use of inverter pairs.

As a first step in making use of this to design a Binary Plus OR gate, we need an expression for OR that will include inverters on the inputs. Beginning with:

$$f = A + B \quad (4.13)$$

we apply DeMorgan's theorem to yield:

$$f = \overline{\overline{A} \cdot \overline{B}} \quad (4.14)$$

Figure 4.4 shows the circuit equivalent to Equation 4.14, while Figure 4.5 shows the same circuit with the NAND expanded to device level, and the *pfet* and *nfet* networks labelled.

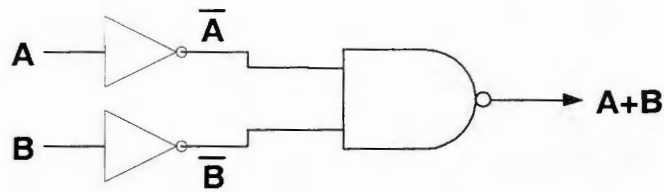


Figure 4.4: OR Created with Inverters and a NAND

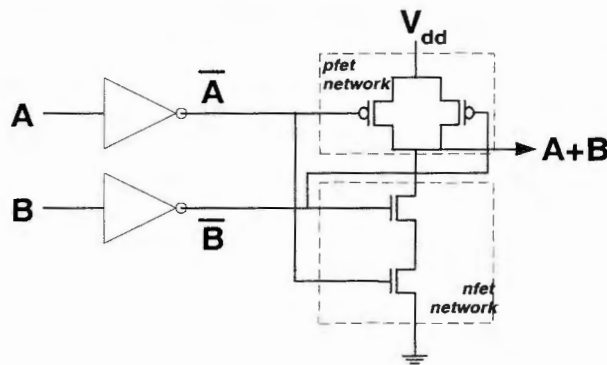


Figure 4.5: OR Created with Inverters and a Device Level NAND

We know from our general development in Chapter 3 that substituting an “high/low” inverter pair for each of the single inverters in Figure 4.5, and connecting the XH outputs to the *pfet* network and the XL outputs to the *nfet* network, should provide the Boolean characteristics of a Binary Plus gate. This arrangement is shown in Figure 4.6.

Inspection of Figure 4.6 will quickly verify that the cases for outputs of 0 and 1 are satisfied. However, the output line *floats* when the conditions for an output of 1 or 0 are not present - resulting in *neither* the *pfet* network nor the *nfet* network conducting. The result of this would be that the gate would tend to display the last valid 0 or 1 output level, at least initially. To ensure this does not occur when an output state of ϕ is appropriate, we follow the development in Chapter 3 by “centering” the output when it would otherwise be floating, creating the circuit shown in Figure 4.7.

The effect of the resistors that “center” the output value in event of a floating

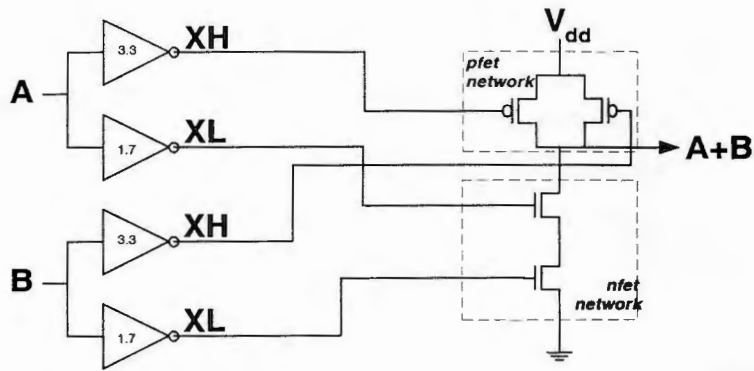


Figure 4.6: Binary Plus OR Gate

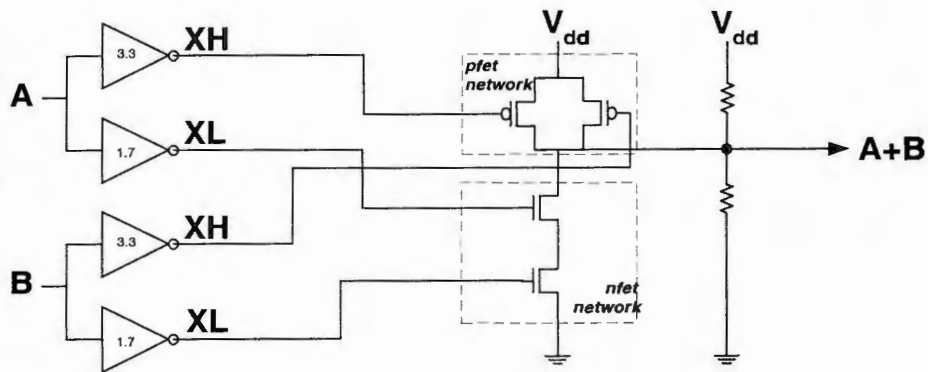


Figure 4.7: Binary Plus OR Gate with Float Centering

condition can be simulated in CMOS circuitry using weak, always-conducting transistors. As we mentioned in Chapter 3, a disadvantage of this approach is that these weak devices are always conducting, resulting in continuous power dissipation, not a desirable condition. We shall see in Chapter 5 how a “dynamic” approach alleviates this problem.

4.2.1 Internal versus external complemented inputs

In Chapter 3 we discussed the internal gate wiring procedure to be used if internally complemented inputs were to be used in a complex gate. The reader will recall that the conclusion was that the complemented XH should be used as input to the *nfet*

network and the complemented XL should be used as input to the *pfet* network, the opposite of their uncomplemented signals.

It should be clear that if we choose to complement outputs *externally* to the Binary Plus gate, then as far as the gate internals are concerned, all inputs are non-complemented - that is, there is no need to connect signals from a complemented “XL” inverter pair output to the *pfet* network nor those from a complemented “XH” inverter pair output to the *nfet* network.

The decision to do this, rather than to complement internally, involves trade-offs that must be considered by the implementer. For example, how many other Binary Plus gates require the same complemented inputs? Such external complementing also increases the number of inverter pairs at the input to the complex gate, as much as doubling them. Additionally, one must bear in mind that any external inverters in such a scheme must be *Binary Plus inverters*, which maintain the integrity of the zoned binary value and signal through the inversion, as shown in Table 4.2, whereas complementing *inside* a Binary Plus gate (“downstream” of the inverter pairs) requires only a pair of standard inverters for each input to be complemented.

A	\overline{A}
0	1
ϕ	ϕ
1	0

Table 4.2: Binary Plus Inverter Truth Table

4.3 Rudimentary applications

Earlier in this chapter we provided a design approach for detecting unknown values and, in combination with material presented in Chapter 3, showed how such detection could be used to implement the Binary Plus logic family.

We shall now consider some additional and rudimentary applications of this knowledge our detection capability enables. It is not suggested that these are demanding or sophisticated uses for this technology, nor that they in any way constitute

an exhaustive list of such uses. They are meant to be illustrative of what can be done with almost trivial applications of the information developed by "decoding" a binary line as a zoned binary source.

Information need not be used to its complete advantage. Sometimes a minor implementation of a concept can lead to "enough" improvement with minimal expenditure in design and space. So it is with the concept of using the fact of uncertain logic levels to solve problems or improve performance. Engineering is, above all, a *practical* process. It is not desirable to implement more of a costly enhancement than is needed to achieve the required level of performance.

In Chapters 5 and 6 we shall study more demanding applications.

4.3.1 Warnings of potential problems

Sometimes it may be adequate to provide warning of circuit inputs that lie in this uncertain zone. Simple indicator lights, readable outputs, or generation of an interrupt to a processor – all are possibly useful features in given circumstances, and could be implemented as desired by the designer. One could even envision a case in which more than one zoning could be performed on the same input as in Figure 4.8.

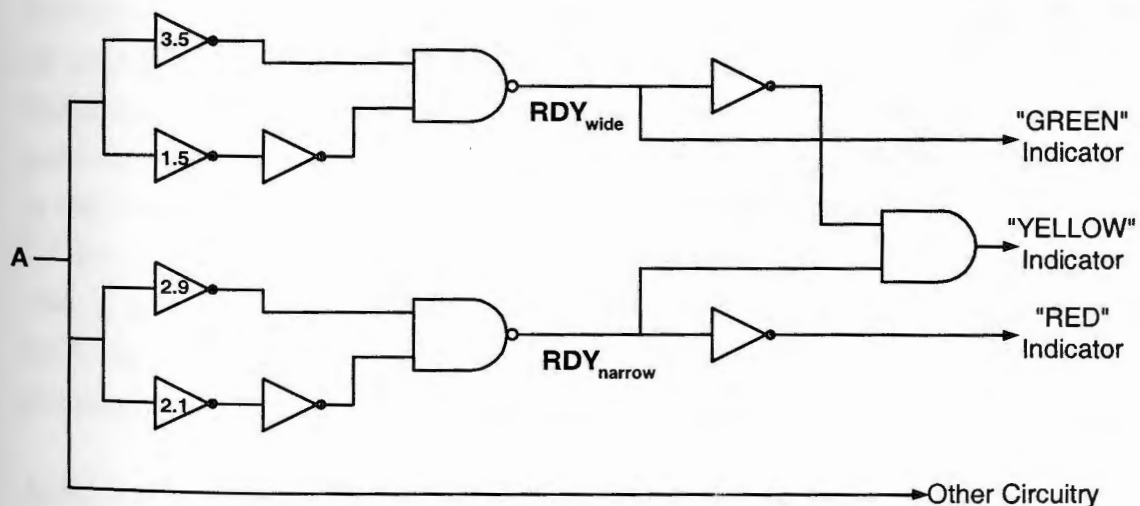


Figure 4.8: Example of Multi-Zoning

4.3.2 The detector revisited as a decoder

When we introduced the detector described in Section 3.1, our motivation was the detection of naturally (or unnaturally) occurring undefined logic levels. Chapter 2 was partially devoted to describing the possible sources of undefined logic levels; our aim in designing a detector was to infer the activity of one or more of these causes.

Section 3.1.1 redefined a line carrying binary data and attached to a detector as a carrier of *zoned binary* data - a line which, it was realized, carried both a value and a signal simultaneously. Table 3.2 specified the binary value and the undefined level signal as separate entities.

Detection of the effect of the normal causes of undefined logic levels does not, however, fully define the domain of uses to which this or equivalent detectors can be put.

Passive encoding

It may be desirable, for example, to determine that a connector has become detached, or that a cable has been cut. Functionally equivalent to an “open”, as discussed in Section 2.2.1, these occurrences would typically result in “floating” inputs, which, we mentioned, might take on a value in the undefined zone, but which might also take on any other value, conceivably even one outside the $V_{ss} \Rightarrow V_{dd}$ range. Therefore this situation, like any open, cannot be reliably detected. However, if we take design action to prevent a floating value, and indeed to force a value in the undefined zone in this circumstance, we then have a reliably detectable condition, as in Figure 4.9.

What we have done here is *explicitly* encoded the ϕ state onto the line, ensuring that, in the even of an open on that line, the condition will be reliably detected. It should be noted that the resistors shown in Figure 4.9 need not even be particularly accurate, depending on the size of the uncertain zone.

Active encoding

Consider another example illustrative of how the encoded nature of zoned binary can be put to work, this one active, in contrast to the passive encoding described above.

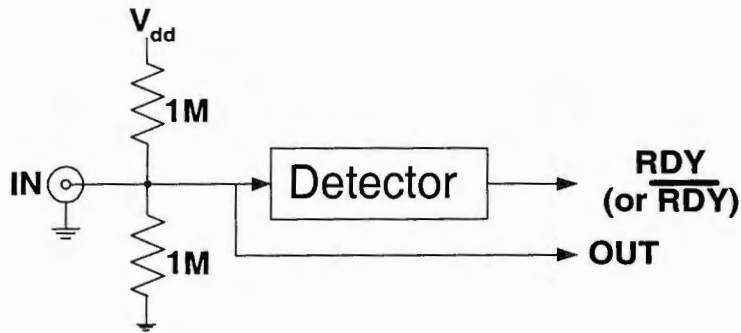


Figure 4.9: Forcing a Zone onto a Floating Line

Figure 4.10 pictures a hypothetical circuit fed by a simple on-off sensor. For example, the sensor might measure the level of gasoline fumes in a confined area and relay a *safe* (1) / *not safe* (0) indication. Part of the sensor circuitry might be devoted to detecting an out-of-range condition in the chemical sensor element. If such a condition existed, neither a *safe* nor a *not safe* indication would be accurate. Of course, a second line could be run for the purpose of indicating this condition, but this would also carry the disadvantage of providing another physical line, providing another point of failure. Instead, the sensor carries tri-state logic on the output, ensuring an electronic disconnect from the line when the measurement is unreliable. This is combined with the passive resistor pair from the previous example to yield a “fail safe” sensor. The design illustrated protects against:

- an out-of-range condition in the sensing element,
- a broken cable,
- a disconnected cable at either end, and
- possibly, a power failure at the sensor.

Other encoding

The two examples given are rudimentary. The concept of using the detector as a zoned binary decoder can be useful in any application in which it is desirable to

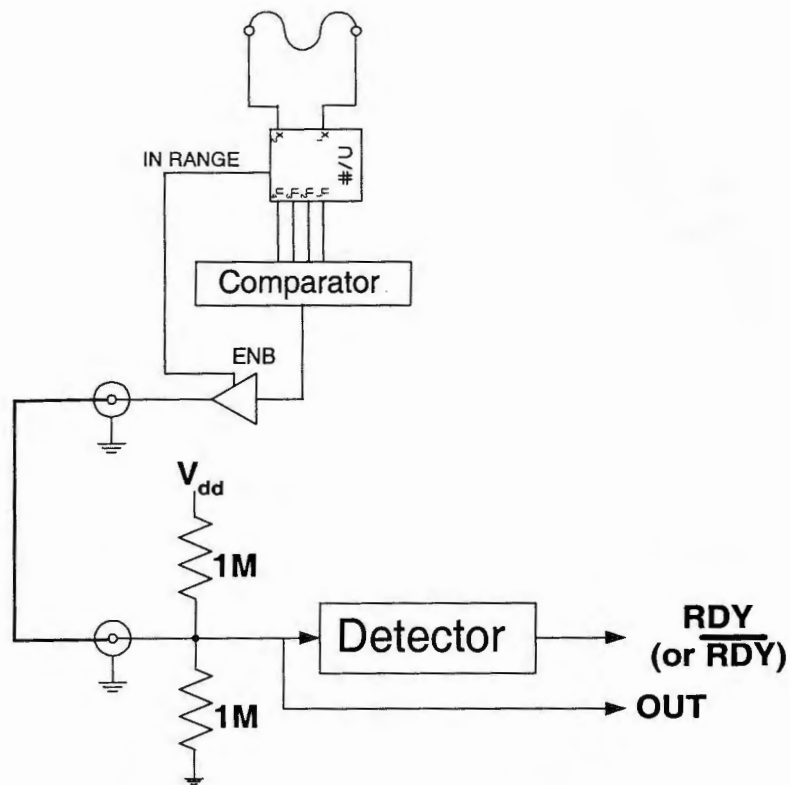


Figure 4.10: Inoperative Sensor Encoding

transmit an indicative signal in lieu of a valid binary value. Further application of this principle is, however, left for other work.

4.4 Introduction to the proof-of-concept circuit

It was desired to test the concepts developed in Chapters 3 and 4, as well as the applications that will be discussed in Chapters 5 and 6, by designing and fabricating a proof of concept circuit addressing some of these areas.

In this chapter we will consider an overall view of this circuit and testing setup, and examine and test in detail elementary zoned binary detection and Binary Plus gates implemented as part of the circuit.

4.4.1 Overall view

It was desired to test as many concepts as possible within the constraints of the space afforded by a 4 mm^2 chip. As there are many different applications of the concepts that are the subjects of this work, it was decided to implement different concepts as independent subsets of circuitry. It was also decided to bypass the testing of trivial applications (such as those discussed in Section 4.3.1 in favor of the more complex areas of asynchronous systems (Chapter 5) and communications applications (Chapter 6).

Experiments implemented

It was decided to implement the following circuits:

- the dual inverters ($1/3 V_{dd}$ and $2/3 V_{dd}$) used to detect the presence of levels in the uncertain zone.
- a small collection of Centered Binary Plus logic elementary gates
- an asynchronous “stage” whose input set sensitivity could be measured
- a circuit illustrating the concept’s use to communications

Dual inverters: This component was included in order to test the proper operation of the inverters at inputs of V_{ss} , V_h and V_{dd} . One input pin and two output pins (“3.3” inverter output and “1.7” inverter output) were required to interface this component to external test circuitry.

Elementary Centered Binary Plus logic gates: It was desirable to test typical Centered Binary Plus logic gates. Four gates were chosen:

- 2-input OR gate
- 3-input OR gate
- 2-input AND gate

- 3-input AND gate

If independently implemented, these gates would have required 10 input pins and four output pins. In the interest of conserving pin availability for other circuitry, it was decided that these gates would partially share inputs. There are three input pins used for the two 3-input gates, and 2 input pins used for the two 2-input gates, for a total of five input pins.

Asynchronous stage: To demonstrate the varying speed of a circuit whose completion time is sensitive to the input pattern, a 4-bit ripple-carry adder, implemented in Centered Binary Plus logic, was chosen. [The concept of *Centered* Binary Plus logic will be covered in Chapter 5.] No effort was made to make this design space-efficient and, instead, standard Centered Binary Plus logic AND and OR gates were used to construct the full adders that make up this design.

The implemented asynchronous stage requires eleven inputs and nine outputs; these will be described in detail in Chapter 5.

Communications application: It was decided to implement a 9-bit simple parity-based checker/corrector, using the concepts developed in Chapter 6. The primary circuitry was developed as a bit-sliced construct containing, in each bit, all circuitry necessary for detection, dual parity checking and output multiplexing.

This circuit requires ten inputs and eleven outputs; these will be described in detail in Chapter 6.

4.4.2 Layout

The circuit was implemented on a 2.3 x 2.3 millimeter MOSIS TinyChip, and fabricated by ORBIT using their SCNA2 (2.0 micron feature size) process under contract to the MOSIS Service, Information Sciences Institute, University of Southern California.

Figure 4.11 shows the relative space and location taken up by the components listed in Section 4.4.1.

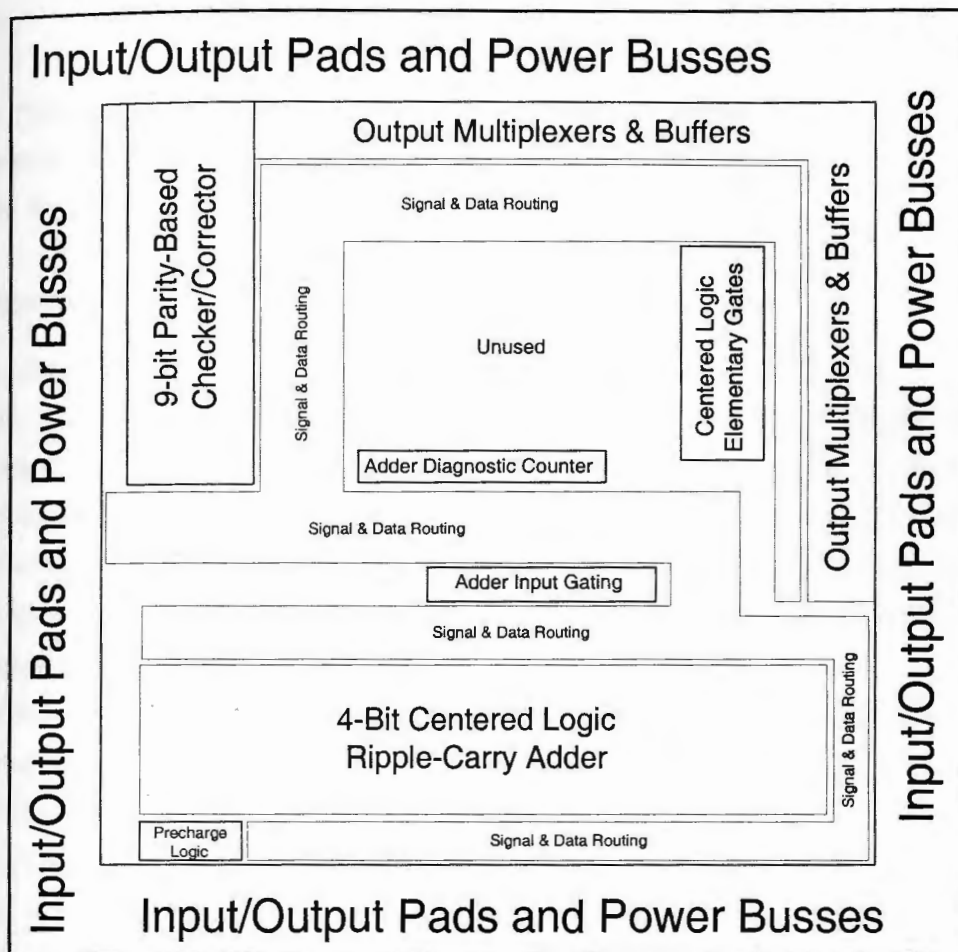


Figure 4.11: Circuit Layout

4.4.3 Pinouts

From the start it was clear that the number of inputs and outputs associated with these circuits would preclude dedicated pins for each. Only 40 pins were available for all power, input and output functions, yet signal and data inputs and outputs listed above in Section 4.4.1 totaled 53 pins, and we have not yet accounted for power requirements, which are:

- V_{ss}
- V_{dd}

- V_h

For reliability, and to ensure an adequate supply of power, at least two pads are customarily allocated for each supply voltage; this would lead to a requirement for 6 power supply pins, for a overall count of 59 pins.

Pin conservation

Two methods were used to reduce the number of required physical pins.

Input sharing: As the 9-bit Parity Checker/Corrector was an entirely separate experiment, there was no need to be able to control its inputs separately from those of the 9-bit Ripple-Carry Adder data. Nine input pins were therefore shared between these two experiments. Additionally, the input to the Binary Plus inverter pair was shared with one of the inputs to the 2-input Centered Binary Plus logic gates. These economies saved 10 pins.

Output pin sharing: Again, as for input pins, the fact that the experiments on this circuit were functionally separate and independent enabled the sharing of output pins. This, of course, required that multiplexers be used to select which of the two possible outputs a pin would relay to the external world. This requirement meant that we would have to allocate a new pin for multiplexer control. But by doing so, it was possible to multiplex eleven outputs from the 9-bit parity checker/corrector with outputs from the adder and the Binary Plus dual inverters.

21 pins were thus made "doubly useful", providing a surplus of two pins in the 40-pin package. One of these was allocated to output multiplexer control, and the other was used as a diagnostic check on the output multiplexing circuit.

Pinout tables and diagram

Table 4.3 shows the *input* pinouts of the circuit as implemented, Table 4.4 the *output* pinouts, and Table 4.5 the *power supply* pinouts.

Figure 4.16, included at the end of this chapter, shows the pinout information in schematic form.

Pin	Input Functions	Pin	Input Functions
14	Parity Odd/Even Set	24	IN8 (Parity Exp.)
16	IN0 (Parity Exp.)		Carry-In (Adder)
	A0 Data (Adder)	27	Precharge Set (Adder)
17	IN1 (Parity Exp.)	28	Precharge Reset (Adder)
	B0 Data (Adder)	29	Input B (2-Input OR)
18	IN2 (Parity Exp.)		Input B (2-Input AND)
	A1 Data (Adder)	30	Input A (2-Input OR)
19	IN3 (Parity Exp.)		Input A (2-Input AND)
	B1 Data (Adder)		Input (Binary+ Dual Inverters)
20	IN4 (Parity Exp.)	31	Input A (3-Input OR)
	A2 Data (Adder)		Input A (3-Input AND)
21	IN5 (Parity Exp.)	32	Input B (3-Input OR)
	B2 Data (Adder)		Input B (3-Input AND)
22	IN6 (Parity Exp.)	33	Input C (3-Input OR)
	A3 Data (Adder)		Input C (3-Input AND)
23	IN7 (Parity Exp.)		
	B3 Data (Adder)	34	MPX (Multiplexer Ctl.)

Table 4.3: Input Pinouts

4.4.4 Test board

A test board was constructed to allow efficient input of allowable values and measurement of outputs. Figure 4.17, also included at the end of this chapter, depicts the schematic of this board.

4.5 Binary Plus component experiments

The purpose of these circuits was to verify the proper operation of the inverter pair that decodes the three-state zoned binary into 0, ϕ and 1, and to check the operation of two and three-input Centered Binary Plus logic AND and OR gates.

Pin	Output for MPX = 0 (Parity Experiment)	Output for MPX = 1 (Other Experiments)
1	Parity Error	Output from "3.3" Inverter (XH)
2	ϕ Detection	Output from "1.7" Inverter (XL)
3	OUT8	8-bit Counter Output
4	OUT7	ALL Ready Signal
7	OUT6	SUM3 Ready Signal
8	OUT5	NONE Ready Signal
9	OUT4	Carry-Out Data
10	OUT3	SUM3 Data
11	OUT2	SUM2 Data
12	OUT1	SUM1 Data
13	OUT0	SUM0 Data
36	0	1
37	\Rightarrow	2-input OR Output
38	\Rightarrow	2-input AND Output
39	\Rightarrow	3-input OR Output
40	\Rightarrow	3-input AND Output

Table 4.4: Output Pinouts

4.5.1 Circuit descriptions

Binary Plus inverter pair

This inverter pair is implemented as shown in Figure 4.12. Outputs XH and XL are routed directly to the appropriate output multiplexers.

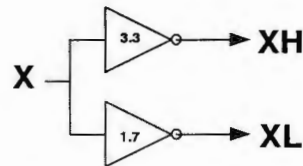


Figure 4.12: Binary Plus Inverter Pair

Pin	Power Supply Voltage
5	V_{dd}
6	V_h
15	V_{ss}
25	V_{dd}
26	V_h
35	V_{ss}

Table 4.5: Power Supply Pinouts

2 and 3-input Binary Plus logic OR gates

These Binary Plus logic gates are implemented as dynamic constructs as will be suggested in Chapter 5, Section 5.3.1. Indeed, the implementation of the 2-input OR is exactly as shown in Figure 5.7.

The implementation of the 3-input OR is shown in Figure 4.13.

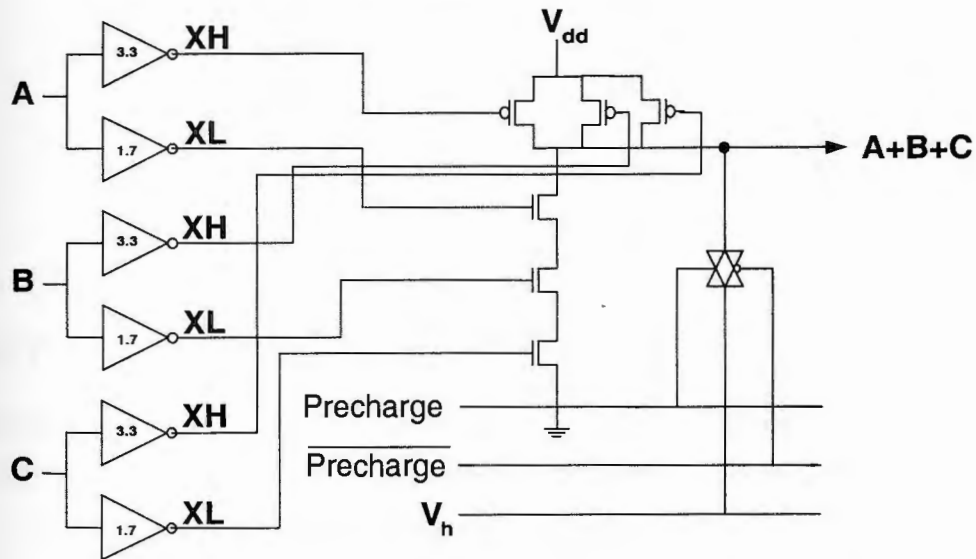


Figure 4.13: Centered Binary Plus logic 3-input OR Gate

The output from these gates was routed to multiplexers for output.

2 and 3-input Centered Binary Plus logic AND gates

The AND gates are implemented in a similar manner to the OR gates discussed in the previous section. The 2 and 3-input versions are shown in Figures 4.14 and 4.15, respectively.

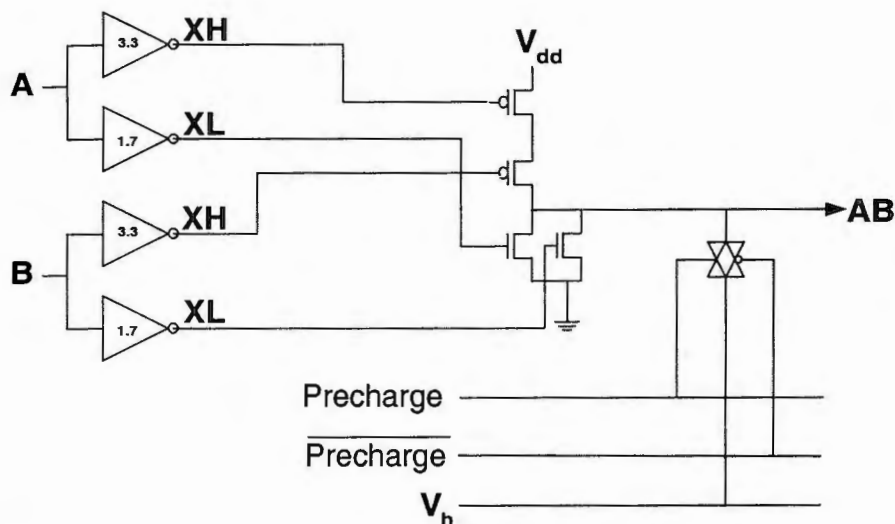


Figure 4.14: Centered Binary Plus logic 2-input AND Gate

4.5.2 Testing results

Binary Plus inverter pair

Testing of the inverter pair was straightforward. Logic level inputs of 0, ϕ and 1 were applied to the input, and the output observed as shown in Table 4.6

Input	XH Output	XL Output
0	1	1
ϕ	1	0
1	0	0

Table 4.6: Test Results: Binary Plus Inverter Pair

Results were as predicted for the inverter pair.

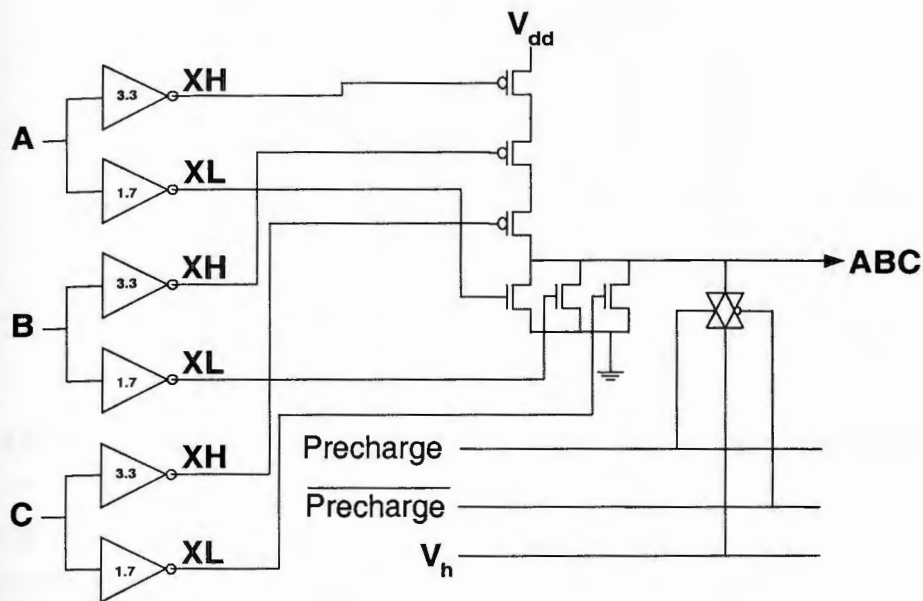


Figure 4.15: Centered Binary Plus logic 3-input AND Gate

2-input gates

All possible input combinations were tested for the 2-input AND and OR gates. Results were as shown in Table 4.7.

The measurements were *not* as predicted. Those entries in Table 4.7 marked with an “*” should have been an output of ϕ . It is likely that this is due to an experimental design oversight on the part of the author.

As designed, the output from each circuit is routed to a multiplexer, the reason for which was discussed earlier in Section 4.4.3, and from there to strong output pad buffers. The multiplexers are constructed from pass switches, and are less likely than other components to alter the transmitted voltage level. The buffers are another matter. In the manner discussed in Section 2.2.2, values in the range of ϕ are highly likely to be transformed to a logic level 0 or logic level 1 by the two powerful, cascaded inverters that make up the buffer.

We can note in advance, however, that the test results for the adder discussed in the next chapter provides evidence that these 2 and 3-input AND and OR gates

A	B	AND Output	OR Output
0	0	0	0
0	ϕ	0	1*
0	1	0	1
ϕ	0	0	1*
ϕ	ϕ	0*	1*
ϕ	1	1*	1
1	0	0	1
1	ϕ	1*	1
1	1	1	1

Table 4.7: Test Results: 2-input Centered Binary Plus logic AND and OR Gates

operate as anticipated, as that adder is constructed from circuits identical to those implemented here, and would not operate as observed unless each gate operated as intended.

3-input gates

All possible input combinations were tested for the 3-input AND and OR gates. The same difficulty with the output buffers converting ϕ outputs to valid 0's and 1's was again noted.

4.6 Summary

In this chapter we developed the design, to include design equations, for the zoned binary detector, as well as illustrating specific designs for Binary Plus gates, the theory for which had already been covered in Chapter 3.

We examined a few rudimentary applications for the concepts involved, and addressed an important point: that once a method of detection of ϕ has been created, originally motivated by the desire to detect a condition created by problems in the circuit or timing inadequacies, it can be used in conjunction with methods that purposely set the logic level on a line as ϕ . Binary Plus concepts can be used in either "mode", although our definition of a Binary Plus logic stage in Chapter 3 was based

around the latter mode.

Finally, we provided an overview of a circuit fabricated to test the concepts in this work, and provided specific details and testing data appropriate to the material covered in this chapter. Circuit details and testing data appropriate to concepts discussed in Chapters 5 and 6 will be covered in those chapters.

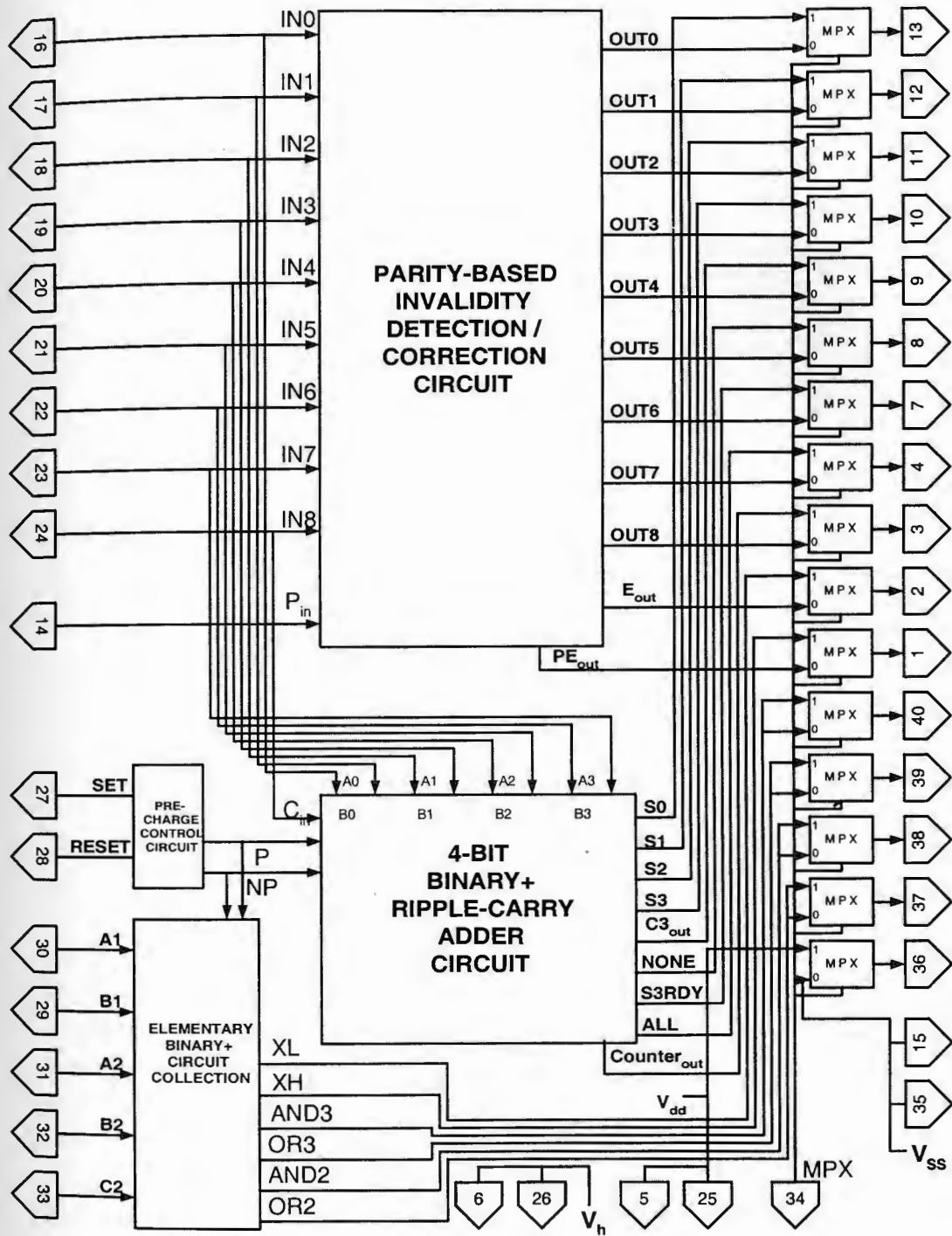


Figure 4.16: Pinout Schematic

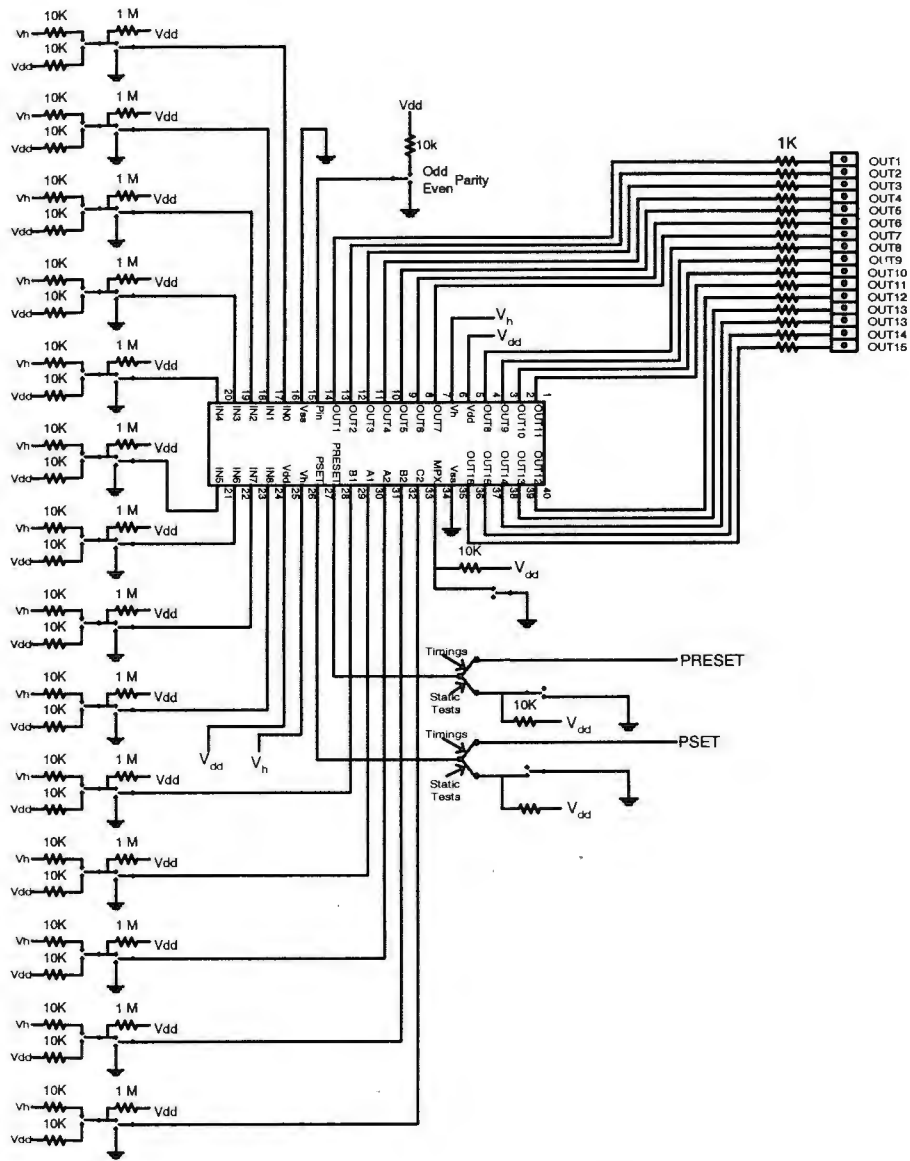


Figure 4.17: Test Board Schematic

Chapter 5

Centered Binary Plus logic

In this chapter we shall further develop the Binary Plus concept to include its dynamic version, Centered Binary Plus logic, and that version's potential for use in asynchronous systems. We will look at gate design for Centered Binary Plus logic, and how gates can be combined into combinational blocks of differing *granularity*.

We shall also examine asynchronous circuitry implemented on the proof-of-concept circuit, and describe the testing procedure and its results.

We begin by very briefly reviewing the operation of "dynamic logic" in VLSI CMOS circuits, and reviewing in more detail the principles behind asynchronous systems.

5.1 Static versus dynamic logic in VLSI design

Static logic designs in CMOS typically use complementary logic, as described in Chapter 3. Complementary *pfet* and *nfet* networks "pull up" or "pull down" the output line. In dynamic logic design, the *pfet* network is replaced by a *precharge* phase, during which a *pfet* device precharges the output to a logic 1 (V_{dd}). Then the *nfet* network is given an opportunity to pull down the output line during an *evaluate* phase. If the *nfet* network does not conduct, the output line remains charged to a logic 1. Figure 5.1 illustrates a NAND gate constructed in this fashion. [It is also possible to use an *nfet* device to precharge to 0 (V_{ss}), and then allow a *pfet* network

the opportunity to pull up the line to a logic 1.]

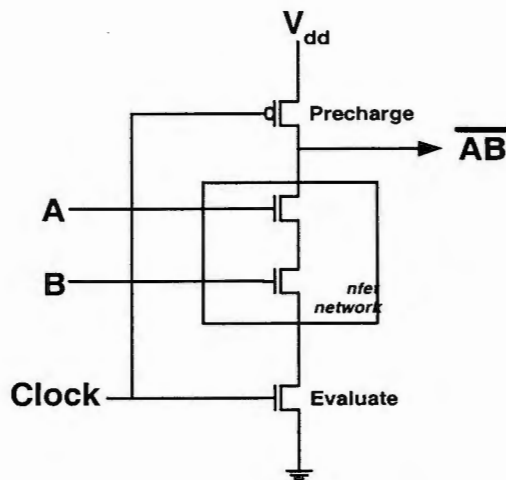


Figure 5.1: Dynamic NAND Gate

The chief advantage of dynamic logic is that it eliminates the need for the *pfet* [or *nfet*] logic network, often saving significantly on space. It does, however, introduce an additional cycle into the operation of the logic, as well as some design complications such as enhanced timing dependencies and charge sharing.[14]

A moment's thought will reveal the sensitivity of dynamic logic to timing - specifically races. If the proper final value of an output is 1, but a race exists in the circuit such that the *nfet* network momentarily conducts, then the output precharge will be dissipated, and the output will take on a value of 0. Even should the race condition then be resolved, and the *nfet* network cease conducting, the damage has been done: there is no mechanism that will "pull up" the output, as there is in a static gate (the *pfet* network). So the consequence of a race to a dynamic circuit can be very serious, and must be guarded against carefully.

Weste and Eshraghian[14] cover dynamic logic design and considerations in some detail, and can be referred to for a fuller understanding, if the reader so desires. Such an understanding is not required for comprehension of this work, as what has been mentioned above should be adequate to our development of *Centered Binary Plus logic* later in this chapter.

5.2 Asynchronous systems - current status and requirements

5.2.1 Overview

Most circuit design today is *synchronous* - data is clocked through sequential circuits (which contain combinational blocks of logic) by a master clock signal. In Section 2.2.1, we discussed the fact that the delay in the slowest block of circuitry was the determining factor in how fast the system, governed by the system clock, could be run. We also made reference in Section 2.2.3 to the criticality of balancing pipeline stage delays so as to allow the master clock governing the pipeline to run at the maximum rate.

A different design philosophy aims to eliminate the need for an all-governing system clock, which in turn can reduce the impact of delays in individual stages on the overall system speed. This approach, called “asynchronous systems”, studies many different forms of systems that do without a global clock signal.

One form, referred to as “wave pipelining”[20], relies on carefully balanced signal transmission paths to enable the sending through of data in *waves*; careful attention to design is needed to ensure that the results from one wave are distinguishable from those in preceding or following waves.

Another approach to asynchronous systems seeks to capture many of the advantages of avoiding a global system clock, while reducing the sensitivity to delay tuning characteristic of wave pipelining circuits. This is referred to as *Globally Asynchronous Locally Synchronous* design, or *GALS*. [5] In a GALS system, each local block runs independently. One set of data is handled by a block at one time, and no further data is admitted to the stage until completion has been detected and the output data latched. A given logic block may complete with one time delay for one set of data, and complete with a different delay for a different set of data. Statistically, the delay attributable to the block is therefore the mean of the delays over a potentially wide range of data input sets, instead of the maximum of those delays over all possible input sets, as would be the case for a globally clocked design.

In Section 2.2.1 we mentioned that increased power consumption is the cost of running a circuit as fast as possible, and explained that power is consumed by transitions from one logic state to another. Self-clocked schemes such as GALS provide one way to reduce power consumption. An independent stage - *not* governed by a global clock - will consume power only when being used. A segment of circuitry not needed will never operate, and will therefore not contribute to power consumption.[5]

Binary Plus logic clearly has the potential to contribute to a completion-signaling scheme. Provided intermediate gate outputs within a combinational block can be initialized to a ϕ state before applying input values to the block, a transition to valid levels at the output of the block can be detected and indicate completion. Centered Binary Plus logic, we shall see, has these necessary characteristics as a byproduct of its design.

5.2.2 Implications for input set sensitivity

In an asynchronous system, a logic block no longer must be given adequate time, every time, to complete its worst case function. The performance can vary with input data; as soon as a function is complete, the output data can be latched and the functional logic block can be given its next set of input.

This latter characteristic has more significant implications for design than might first be thought. For example, the synchronous nature of most systems has resulted in much effort being expended in creation of designs that have good *worst case* performance, versus good or at least adequate *mean* performance.

Consider the “lowly” ripple-carry adder shown in Figure 5.2. This adder is rarely used in synchronous designs because of its very poor worst-case performance.

The worst case gate delay for such an adder, using a typical full adder design, is given by:

$$Delay = 3 + n \cdot 2$$

where n is the operand size in bits. For a 16-bit adder, the worst-case gate delay is 35. This occurs when a carry generated in the low-order bit full adder is propagated

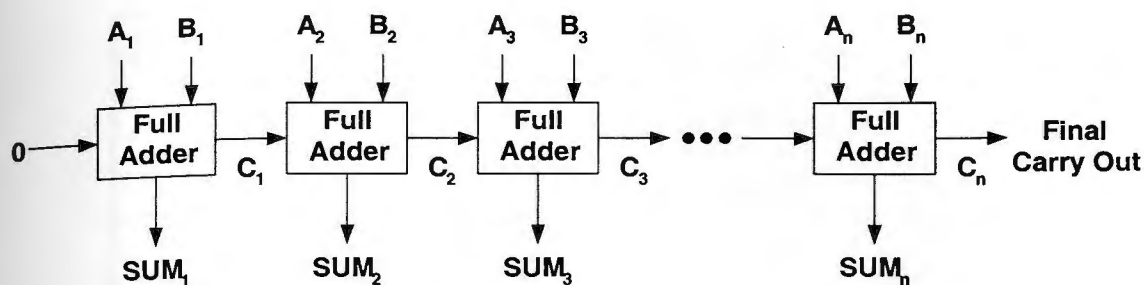


Figure 5.2: Ripple-Carry Adder

through the entire series of full adders. In a synchronous system in which this adder design was used, the synchronizing clock signal would have to allow adequate time for this worst-case carry propagation to occur.

In an asynchronous system, in contrast, the *mean* gate delay is a better measure of an adder design's efficiency. Using a 16-bit adder as an illustrative example, there are 2^{16} possible configurations of input bits for each operand, leading to a total of 2^{32} possible "problems", or input sets, that can be presented to such an adder. For each of these input sets, one can readily see that the total gate delay - the time before all outputs will have "settled" to their final, valid values - can be computed from the above formula, substituting for n the maximum number of consecutive carries (the largest "carry chain") encountered in performing that addition.

Simulating the ripple-carry adder over the 2^{32} possible input sets yields the results shown in Table 5.1.

The mean gate delay can be computed to be approximately 13.27, or roughly 38% of the worst-case delay. There may be situations in which the space advantage of a simple adder design like the ripple-carry, combined with a mean gate delay of 13.25 (and a median gate delay of just over 11), is enough to make its inclusion in a design warranted. If there are additional constraints known to the designer that might further reduce mean delay (for example, knowledge that the Carry-in input is always zero), the simple design may be even more attractive. In any event, this example points to the need to emphasize designs of all kinds with good *mean* performance for use in asynchronous systems, a significant shift in philosophy.

Maximum Carries	Delay (Gates)	No. of Cases	% of Total
0	3	43,046,721	1.0%
1	5	196,197,901	4.6%
2	7	472,945,947	11.0%
3	9	671,448,213	15.6%
4	11	695,429,010	16.2%
5	13	603,021,996	14.0%
6	15	473,355,009	11.0%
7	17	351,502,659	8.2%
8	19	250,962,624	5.8%
9	21	174,890,016	4.1%
10	23	121,247,280	2.8%
11	25	83,613,384	1.9%
12	27	57,395,628	1.3%
13	29	39,326,634	0.9%
14	31	27,103,491	0.6%
15	33	19,131,876	0.4%
16	35	14,348,901	0.3%

Table 5.1: Ripple-Carry Adder Performance Summary

The ripple-carry adder was used as an example for two reasons. Firstly, the significant difference between its *mean* and *worst-case* performances highlights the paradigm shift in design for asynchronous versus synchronous systems. Secondly, a small (4-bit) ripple-carry adder has been implemented on the fabricated proof-of-concept circuit.

5.2.3 Globally asynchronous locally synchronous systems

The term asynchronous systems covers many concepts, grouped together under the common characteristic of not requiring a global clock signal. One such concept, *wavepipelining*, can be described as *locally asynchronous*. Lam and Brayton, in their 1994 book Timed Boolean Functions[20], succinctly describe both the advantage and the complications of wavepipelining:

“... in wavepipelining mode, the circuit... will be clocked at a period less than the maximum topological delay (or true delay) of a stage; thus a data wave is pumped into a stage before the previous wave reaches the registers at the end of the stage. So wavepipelining circuits operate at higher speeds than conventional circuits, sometimes orders of magnitude higher. Since the clock period is shorter than the delay of a circuit, data from neighboring clock cycles co-exist in the circuit simultaneously, and they can interact to cause the circuit to compute incorrectly. For instance, if a long path and a short path converge at a gate and the clock frequency is fast enough, then the present data on the short path can arrive at the gate earlier than the previous data on the long path, resulting in an invalid computation. Hence wavepipelining circuits involve complex signal interactions in the temporal domain and their proper operations require precise timing analysis.”

A type of asynchronous system that removes the need for careful timing control in the combinational logic block, while maintaining the advantages of asynchronous systems on a global scale, comes under the general classification of *Globally Asynchronous Locally Synchronous (GALS)* systems.[5] To develop this type of system from more familiar constructs, let us modify the pipeline shown in Figure 2.16 to explicitly show the interstage “hold and forward” latches that must be a part of any pipeline. You can see in Figure 5.3 that the global clock signal actually controls these latches, each of which receives data from a previous pipeline stage and releases it into the next.

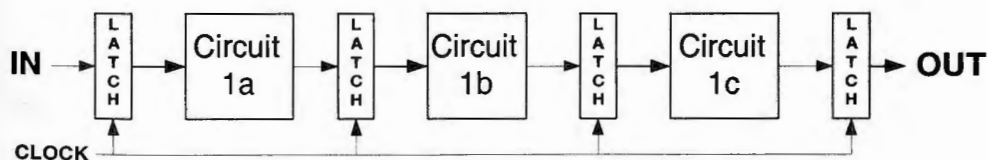


Figure 5.3: Three-Stage Pipeline

To eliminate the global clock in a GALS pipeline, we make each pipeline stage and following latch responsible for recognizing completion of its task, latching the

valid results, and sending back to the previous latch a signal indicating that the next input set can be released into the newly available stage. This modified form of the pipeline is illustrated in Figure 5.4.

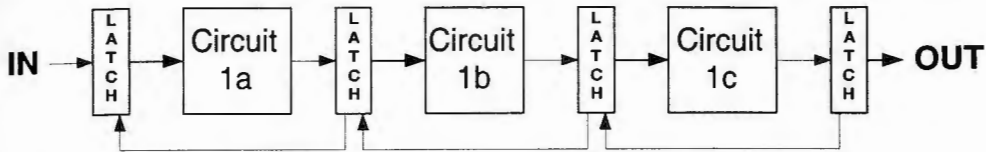


Figure 5.4: GALS Three-Stage Pipeline

Each stage now takes only the amount of time required to accomplish its task with the specific input set presented to it - it need not wait for a global clock signal to cycle.

While one might at first conclude that the overall pipeline speed is still limited by the delay of the slowest stage, we must bear in mind that that delay may be long for some input sets, and short for others. We saw in Table 5.1 that a stage composed of a 16-bit ripple carry adder could vary in delay from three gate delays to thirty-five, depending on the input set. If we wished to make the overall pipeline less sensitive to potentially long data-dependent delays in a pipeline stage, we could provide for storage of multiple results in each latch, which would tend to “average out” the delay of a stage. While this would increase the pipeline latency, it would tend to also increase its throughput in the presence of varying stage delays.

We could further enhance the pipeline by expanding its *width*, as in Figure 5.5.

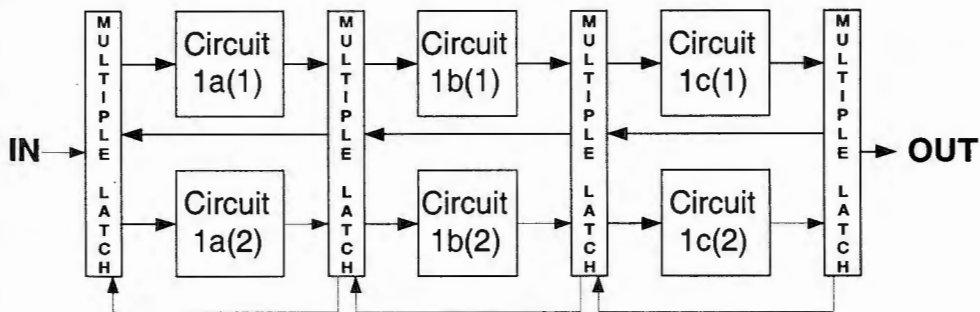


Figure 5.5: Expanded GALS Three-Stage Pipeline

This arrangement, it is seen, would double the capacity of the pipeline. Additionally, since the multiple latches would have the ability to release an available result from, for example, pipeline stage a, into Circuit 1b(1) or 1b(2), depending on which was available first, it would further “smooth” the operation of the pipeline, making it less sensitive to timing “spikes” caused by occasional inputs sets generating large delays.

5.2.4 Currently used methods for completion detection

Self-timed combinational logic blocks must be able to determine when completion has been achieved and results are valid. There are several methods in use for doing this, of which we shall briefly mention a few.

Bounded-delay: *not* detecting completion

The bounded-delay technique, such as described in [37], does not concern itself with detecting completion. Instead, it estimates the maximum (worst-case) delay for a stage, and creates a delay element to provide that much delay before the output data is latched and new data is admitted into the stage. While it might at first seem that this approach gives up the benefit of GALS entirely, such is not really the case. The global clock signal is still eliminated, the prime purpose of GALS constructs. Additionally, although each pipeline stage now has a fixed delay, it need not be the same delay as every other stage. Pipeline latency is reduced (in comparison to an equivalent synchronous pipeline) but throughput will not necessarily be improved unless slow stages are duplicated in a manner similar to that shown in Figure 5.5.

The chief disadvantage of this technique is that it does not take advantage of data dependent delay to improve throughput.[38]

Dual-rail: doing it twice

So-called “dual-rail” techniques, such as proposed in [5], are based on using *two independent nfet networks*; input to these networks are both the normal inputs and inverted inputs, so that one or the other *nfet* networks conducts. The RDY signal

(completion) for a stage goes to logic 1 when either of the two outputs goes to logic 0 (both were precharged to a logic 1 at the start of the cycle).

While these methods take advantage of data dependent delays, they “carry the disadvantages of a very high hardware overhead and slow operation” [38].

Activity-sensing: waiting for steady-state

During the operation of a combinational logic block, the application of new data to the inputs will typically result in various transitions of internal (intermediate result) signals and the output(s). Grass and Jones [38] proposed a method of detecting such transitions; after no transitions had occurred for a specified period of time, completion could be assumed.

Aside from the obvious disadvantage of completion not being signalled until a preset delay period had passed since the last signal transition, the case in which no signal transition takes place also must be addressed; such a circumstance could occur in many ways, but would at least occur when two consecutive input sets were identical. Grass and Jones propose a “minimum delay generator (MDG)” which would signal completion when no transitions at all occurred.[38]

5.2.5 Interstage requirements

In Section 5.2.3 we mentioned the need for “store and forward” latches to receive the results from one stage and, when the following stage becomes available, to apply those results as input to the next stage.

These latches, as has been suggested, can be simple or complex. But at the least, they must be able to:

- Latch the results, possibly on the leading edge of the RDY (completion) signal.
- Initiate any required precharge phase for the combinational logic block from which the results have just been latched.
- Signal the preceding latch when a new input data set may be released into the stage.

- Release the latched data to the next combinational logic block when the following latch signals that it is permissible to do so.

The design of these interstage latches is not a focus of this work. However, it is required that completion-detecting components of the designs to be covered in the next section be able to fulfill the interfacing needs of such latches. These requirements are:

- A completion signal must be supplied to the receiving (*sink*) latch. All outputs from the combinational logic block must be valid and remain valid while this signalling is transitioning from logic 0 to logic 1.
- Any precharge required for completion detection or result determination must be able to be controlled by a signal from the sink latch or as a natural consequence of the results being latched. This process should also reset the completion signal to logic 0.
- Once the precharge has been accomplished, the completion signal must not transition to logic 1 until a new set of data inputs has been presented to the circuit by the input (*source*) latch, and valid results obtained.

5.3 Centered Binary Plus logic

We shall now proceed to adapt the Binary Plus concept to self-timed circuitry. In doing so, we shall combine many concepts covered previously.

In Section 3.3.1, we saw that the output from a Binary Plus gate will take on a valid logic level only when critical inputs have become valid. As, depending on the logic function of the gate, not all inputs are, or remain, critical, Binary Plus logic can be said to take advantage of data dependencies to improve performance. To do this, we must ensure that all inputs and outputs - as well as internal signals (intermediate results) - are given an initial value of V_h .

5.3.1 Precharge is to V_h

To do this, we borrow a technique from dynamic logic, and precharge all results and intermediate values to V_h . It is from this precharging to the center of the $V_{ss} \Rightarrow V_{dd}$ range that we obtain our name for this subset of Binary Plus logic: *Centered Binary Plus logic*.

Two obvious approaches present themselves for this precharging process. One is to provide weak *pfet* and *nfet* transistors to accomplish this precharging. Modifying the Binary Plus OR gate shown in Figure 4.7 yields the circuit shown in Figure 5.6. This approach has some undesirable characteristics, however:

- During the precharge part of the cycle, there is a current path from V_{dd} to V_{ss} , and therefore power will be used.
- To minimize the power use during precharge, the precharge transistors will have to be made very weak. This will slow the precharge process, impacting the speed of the circuit.
- Due to the variance between transistors and fabrication parameters we have discussed in Chapter 2, the strengths of the *pfet* and *nfet* precharge transistors may not be adequately close to equal to assure a precharge value very close to V_h .

In the interest of eliminating the above problems, we introduce a single, additional supply to the circuit, carrying V_h . This modifies the circuit of Figure 5.6 to that shown in Figure 5.7.

Note that a pass switch is necessary, as the output line may have to be either “pulled up” from logic 0 to V_h or “pulled down” from logic 1 to V_h .

The advantages of this circuit over the use of weak precharge transistors are:

- No path is created from V_{dd} to V_{ss} . Those supplies are no longer involved in the precharge process.

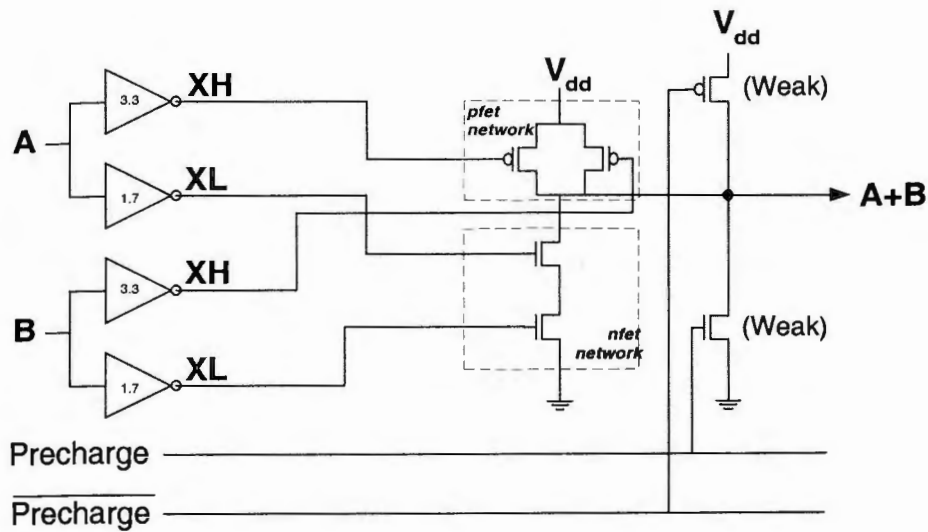


Figure 5.6: Weak Transistor Precharge

- The pass switch need not be made purposely “weak”. Charging of the output directly from a V_h supply should be fast, minimizing the time spent in that part of the cycle.
- Any variance between transistors in the pass switch will not affect the final voltage level held by the output line at the end of the precharge process.

Must have both *pfet* and *nfet* complementary logic

In the dynamic logic discussed in Section 5.1, the *pfet* (or *nfet*) network was eliminated, and a precharge device used in its stead. Due to the fact that Centered Binary Plus logic precharges to V_h , we will still need both a *pfet* network (to pull the output up to logic 1) and an *nfet* network (to pull the output down to logic 0). This additional space requirement will certainly be a consideration in deciding whether to use Centered Binary Plus logic in an asynchronous design, but there are compensations, as we shall now discuss.

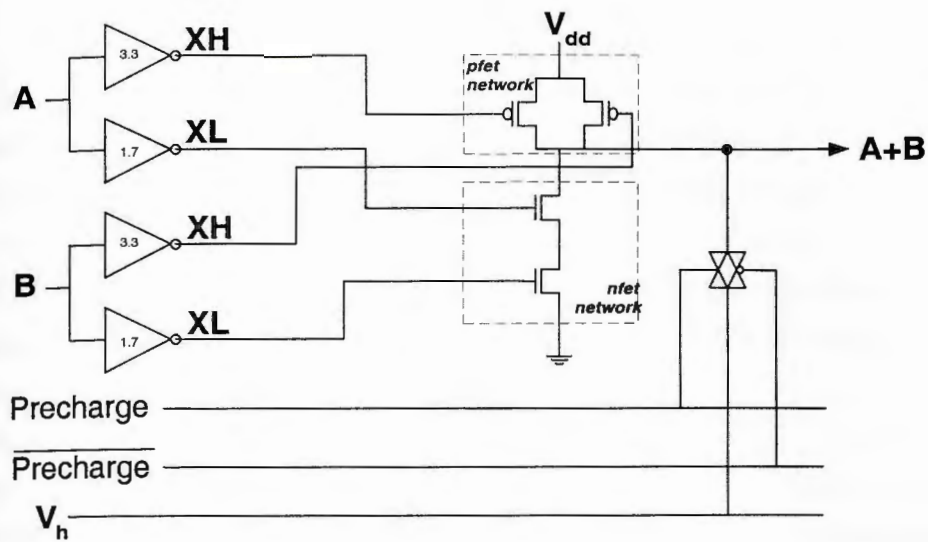


Figure 5.7: Precharge Using V_h Supply

5.3.2 Inherent speed enhancement

In dynamic logic like that illustrated in Figure 5.1, the precharge phase sets result and intermediate lines to one end of the logic range: V_{dd} (or V_{ss}). During the evaluate phase of the cycle, time is required for the *nfet* (or *pfet*) network, if it conducts, to pull the output or intermediate result well past V_h into the other valid logic state, a voltage “distance” of, perhaps, 66% of $V_{dd} - V_{ss}$.

In Centered Binary Plus logic, the precharge is only to V_h . When the “evaluate” phase of the local synchronous cycle starts - when inputs are made available to the stage - as inputs are applied and intermediate results filter through the combinational logic block, the logic level on those intermediate and end result lines have to be pulled up or down only through the boundary between our undefined zone and one of the two valid logic zones, a “distance” of 16.5% of $V_{dd} - V_{ss}$. This can happen much more quickly than the “full-swing” dynamic logic circuit. We can say that Centered Binary Plus logic should enjoy an inherent speed advantage for this reason. Of course, this conclusion can be impacted by the specific implementation of Centered Binary Plus logic, including such considerations as the capacitance of the required inverter pairs, if that specific implementation is taken.

5.3.3 Elimination of races

In Section 5.1, reference was made to the vulnerability of dynamic logic in general to race conditions (hazards). It is in this area that Centered Binary Plus logic shows a significant advantage. As no changes have been made to the basic Binary Plus concept that would invalidate the Theorems in Chapter 3, we can say that Centered Binary Plus logic is immune from races, both within a single gate and within an entire combinational stage. This eliminates the need for careful attention to timing dependencies needed in dynamic logic design.

Intuitively, as no Centered Binary Plus logic gate can display any valid logic level on its output until the inputs have reached a *necessary and sufficient condition* for that output (which implies that the later arrival of a previously unknown input *cannot* change the output), and all such inputs shall be, in turn, zoned binary inputs conditioned by previous Centered Binary Plus logic or Binary Plus compatible input sources, it is clear that races cannot occur in properly functioning Centered Binary Plus logic stages.

5.3.4 Detection of invalid inputs and defects

This chapter has emphasized the use of the characteristics of zoned binary to asynchronous systems, pointing out how those characteristics can provide for a powerful completion-detection capability. But the designer is free to implement additional enhancements taking advantage of the other uses of our detection capability.

For example, self-timed systems could be equipped with an auxiliary timer to detect when an excessive amount of time has elapsed with no completion being detected. Such an "alarm" could signal a hard or soft defect in the circuitry, or, if it were "designed in", that a signal that is in the unknown zone has become critical to the computation being done by the circuit.

Note, however, that Centered Binary Plus logic *is* a dynamic logic, despite the presence of both *pfet* and *nfet* networks. The precharge (to V_h) can dissipate over time, so the detection of non-completing input or circuit conditions must be sensitive to these timing considerations. As the time necessary for inputs to be processed

through a Centered Binary Plus logic stage should be, under normal conditions, far less than the dissipation time, timing determination for this purpose should not be difficult to achieve.

5.3.5 Granularity

Just as a large combinational block in a synchronous system can be broken up into balanced pipeline stages, Centered Binary Plus logic provides the paradigm for a designer's choice for breaking up a circuit into self-timed blocks. A system in which the blocks of combinational logic between latches are small could be referred to as having *fine granularity*, whereas an ALU implemented in one logic block would certainly be said to display *coarse granularity*.

Much the same tradeoffs exist in the coarse to fine granularity decision as in the breakup of circuits into pipeline stages in synchronous systems, with some additional considerations.

- As in synchronous pipelines, making the granularity finer will tend to increase throughput.
- Space overhead, especially in the form of latches, increases as granularity becomes finer, just as in synchronous pipelines.
- For Centered Binary Plus logic (and other GALS constructs), finer granularity allows for easier "widening" of the pipeline for "bottleneck" stages.
- Granularity in Centered Binary Plus logic pipelines can be taken to the single gate extreme, if advisable from a design standpoint. Each gate contains the essential capabilities to be a pipeline stage.

We shall henceforth refer to a self-synchronized Centered Binary Plus logic block as a *granule*.

5.3.6 Control and handshaking

While, as stated earlier, it is not a purpose of this work to look closely at latch and control design, it is desirable to specify methods by which Centered Binary Plus logic granules interface with their source and sink latches.

Completion signaling

We have made clear that Centered Binary Plus logic is inherently capable of detecting a valid output logic signal. It is left for us to briefly define how such detection applied to several outputs might be aggregated into a *granule completion signal (CLS)*.

Let us expand upon the simple ripple carry adder shown in Figure 5.2. We add RDY detectors and combine their outputs with a binary AND gate, yielding the circuit in Figure 5.8.

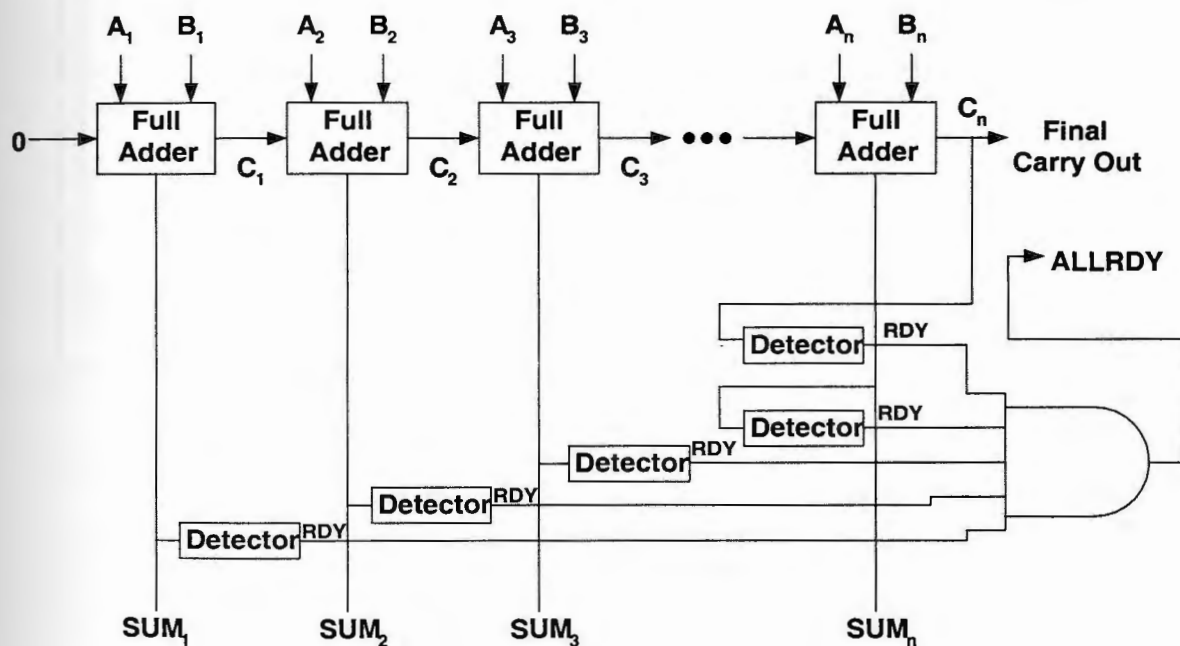


Figure 5.8: Adder with Completion Signal

Signal ALLRDY meets the requirements of a completion signal. Lines SUM_1 through SUM_n and the Final Carry Out would be latched by appropriate circuitry

on the *rising* edge of ALLRDY.[39, 40]

Precharge initiation and completion

As the ALLRDY signal will latch the data as it rises, it is also a signal to the sink latch that the precharge can begin. This would be accomplished through the use of the Precharge SET input, as shown in Figure 5.9.

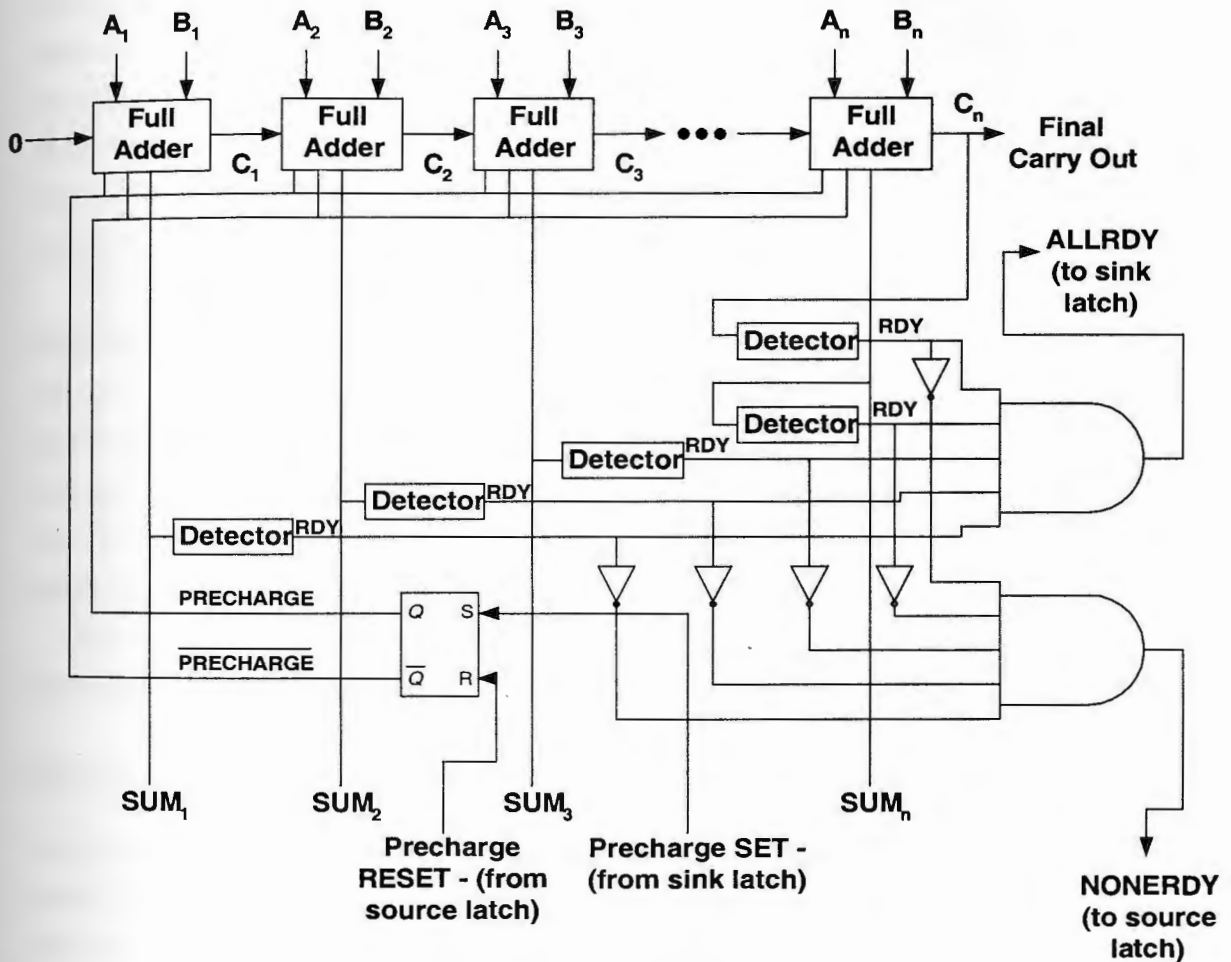


Figure 5.9: Adder Including Precharge Cycle

A second multiple AND is used to aggregate the \overline{RDY} signals to provide an indication to the source latch that precharge is complete and the stage is ready for

another input set. The source latch would then reset the precharge flip-flop and release the inputs into the stage.

A faster method of cycle control

By internally connecting the ALLRDY output to the Precharge flip-flop SET input, we allow the precharge to begin immediately upon completion and latching of the output data. The NONERDY signal can be routed to the Precharge flip-flop RESET input to initiate the evaluate phase as soon as the precharge is complete. However, we require two more features: the ability for the source latch to prevent an evaluate until it has valid input data to present to the stage, and an equivalent ability for the sink latch to prevent an evaluate until there is space in the latch to receive a new output set.

Figure 5.10 illustrates these connections, as well as enhancement of the adder with two enable lines: ENB_I for use by the source, or “input”, latch, and ENB_O for use by the sink, or “output”, latch. Until both enable inputs are high, the precharge phase cannot end, and the new inputs cannot be released into the adder. Although not shown in the figure as drawn, all input lines between the tri-state buffers and the full adders would also have to be precharged, to prevent charge sharing from potentially affecting the results at the very start of the evaluate phase.

Note that power-saving is automatic with this scheme. The circuit is held in precharge phase, using no power, until there is work for it to do.

Satisfaction of requirements

In Section 5.2.5 were listed three requirements for a stage to fulfill the interfacing needs of interstage latches in a GALS pipeline. Let us now review them in light of our preceding development:

- A completion signal must be supplied to the receiving latch. All outputs from the combinational logic block must be valid and remain valid while this signalling is transitioning from logic 0 to logic 1.

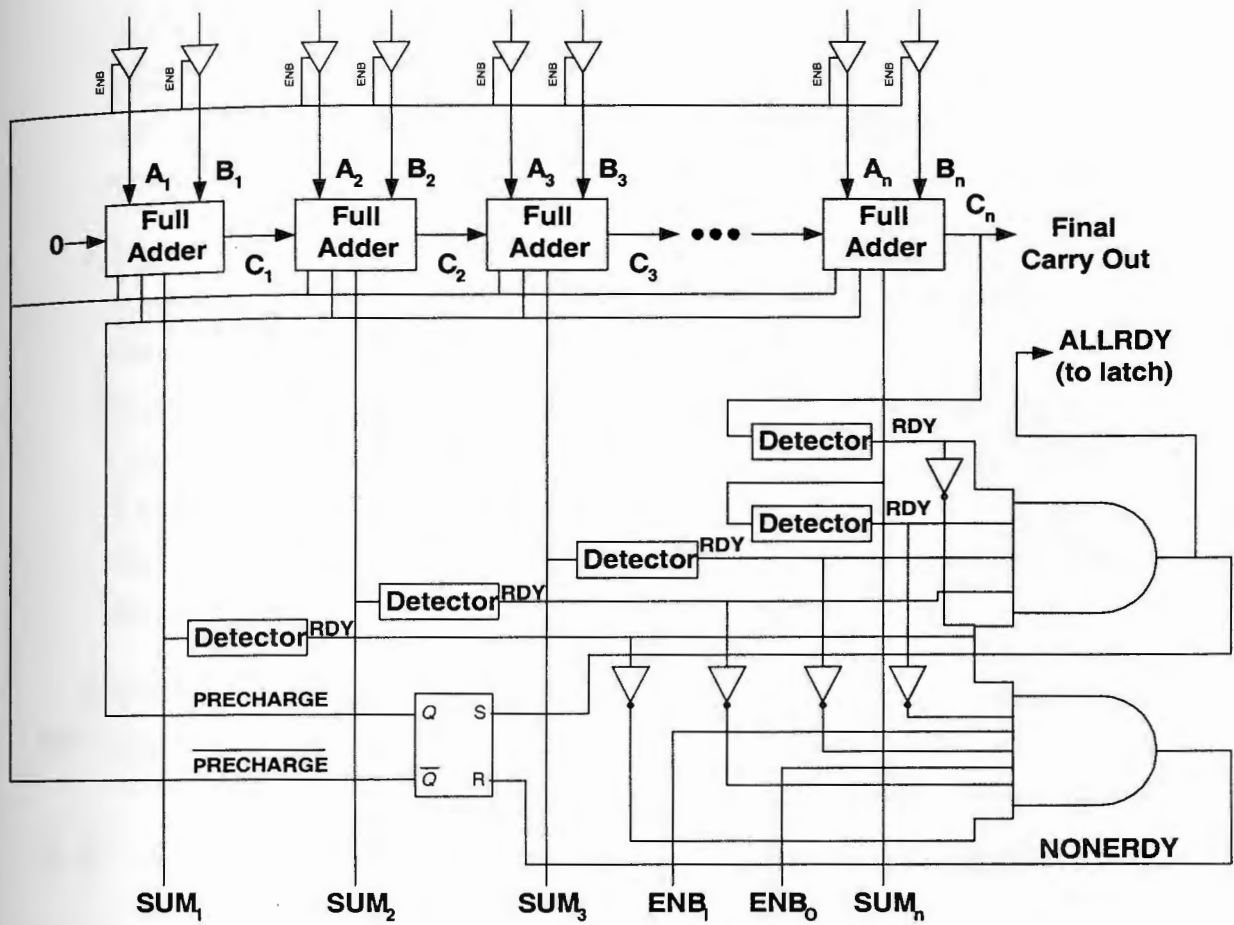


Figure 5.10: Adder Including Enable Controls

Indeed the transitioning of the ALLRDY signal is a clear indication that all outputs are valid and may be latched. As the precharge phase cannot begin to be started until the ALLRDY signal becomes 1, latching (on the leading edge) will be complete before precharge begins.

- Any precharge required for completion detection or result determination must be able to be controlled by a signal from the receiving latch or as a natural consequence of the results being latched. This process should also reset the completion signal to logic 0.

If the Precharge flip-flip SET input is generated by the sink (receiving) latch, this requirement is clearly satisfied. The ALLRDY signal will go to logic 0 as soon as the first of the results moves out of its valid range due to the precharge operation.

- Once the precharge has been accomplished, the completion signal must not transition to logic 1 until a new set of data inputs has been presented to the circuit, and valid results obtained.

The ALLRDY signal cannot again transition to logic 1 until (a) the precharge phase is released by both the source and sink latches (this implies that both a new input set is ready for release into the stage and that there is “room” in the sink latch for the next result set) and (b) the input set propagates through the stage and makes all results valid.

It would seem that the requirements have been satisfied. Design of the latch is left to the implementer.

5.4 Comparison with other GALS self-clocking methods

In Section 5.2.3 were listed other, currently used methods for detecting stage completion in a GALS pipeline stage. We now compare these techniques with the Centered Binary Plus pipeline stage approach just developed:

Bounded-delay: The Centered Binary Plus pipeline approach takes advantage of input pattern dependencies in completion time, whereas the bounded-delay technique[37] is similar to synchronous approaches in that it requires a worst-case delay be built into the pipeline stage timing. The bounded-delay method, of course, requires significantly less hardware overhead than the Centered Binary Logic method or other methods do.

Dual-rail: The dual-rail technique[5], as has been mentioned before, is characterized by high hardware overhead and slow operation. While a speed comparison is inappropriate at this time (as no effort has been made to design a detector optimized for speed), we may fairly say that the Centered Binary Plus technique will have a significant hardware overhead. However, it has been proven not to suffer from the sensitivity to races that dynamic techniques like dual-rail have, so Centered Binary Plus pipeline stages should be more robust.

Activity-sensing The chief advantage of Centered Binary Plus logic over activity sensing[38] is that there must be a delay built into activity-sensing stages, over and above the actual completion time. Minimizing such delays makes it necessary to do detailed timing analyses of such stages to ensure that the delay is not excessive.

No claim is made that Centered Binary Plus logic is the *best* approach to use in all GALS pipelines. However, it *does* possess its own significant advantages with regard to currently used techniques - factors a designer will take into account in determining the best technique to use in a specific implementation.

5.5 Fabricated 4-bit ripple-carry adder experiment

There are typically two primary approaches in designing a complex combinational circuit to perform a given function. One is to use complex gates to implement the function; this method reduces the gate count, but increases design complexity and time and tends to decrease modularity.

The other approach is to use standard circuits for logic functions, even at the expense of additional space. This maximizes regularity, and not only can lead to a reduction in the time to create and simulate a design, but can also lead to being able to judge the design correct by construction.[14]

Although the use of complex gates can lead to significant space savings in Binary

Plus and Centered Binary Plus logic (due to the reduction in the number of dual inverter based “zone decoders”), it was decided to implement the proof-of-concept asynchronous circuit by use of standard Centered Binary Plus logic AND gates, OR gates and inverters.

5.5.1 Ripple-carry adder

The circuit selected to demonstrate the use to asynchronous design of the concepts of Centered Binary Plus logic is the ripple-carry adder. This adder should vary in completion time with differing input data patterns. It was not an aim of this work to produce a fast or space-efficient implementation.

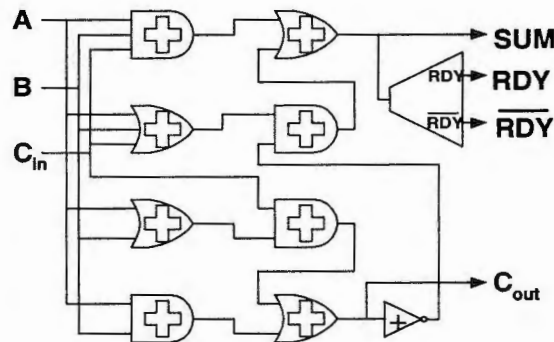


Figure 5.11: Centered Binary Plus logic Full Adder

The gate-level diagram of the full adder circuit used in this design is shown in Figure 5.11. We introduce two conventions at this point.

Centered Binary Plus logic gates are denoted in the above diagram by the use of standard binary logic gate symbols, superimposed by a “+”. This implies:

- the existence of zone decoding dual inverters on all inputs,
- standard Binary Plus gate design - that is, the routing of the “high transition voltage inverter” output to the *pfet* network and the routing of the “low transition voltage inverter” output to the *nfet* network, and
- inclusion of components necessary to precharge the output of the gate to V_h .

A standard symbol is shown to represent a full “Ready detector”, with its output of both “RDY”, indicating that the logic level being measured is in one of the valid binary ranges, and its inverse, \overline{RDY} , indicating that the logic level being measured is in the intermediate, ϕ range. The presence of both outputs is necessary for proper functioning of the precharge/evaluate cycle, as discussed in Section 5.3.6 and as we shall see shortly.

The organization of the adder itself is very similar to that shown in Figure 5.10. As modified to use the Centered Binary Plus adder shown above, its final form appears in Figure 5.12.

Precharge control

The prime method for control of the precharge/evaluate cycle in this proof-of-concept circuit is via the PSET and PRESET inputs:

- A short pulse on the PSET input will set the precharge flip-flop, resulting in the internal PRECHARGE line going high and its complement, the $\overline{PRECHARGE}$ line going low. This turns on the pass switches in the Centered Binary Plus logic gates to charge all intermediate results and gate outputs to V_h . It also isolates the adder inputs from the logic.
- A short pulse on the PRESET input will reset the precharge flip-flop, resulting in the internal PRECHARGE line going low and its complement, the $\overline{PRECHARGE}$ line going high. This turns off the pass switches in the Centered Binary Plus logic gates, isolating the intermediate result lines and gate outputs from the V_h supply. It also has the effect of turning on the pass switches that gate the adder inputs to the logic.

Were this circuit to be used as part of a Centered Binary Plus asynchronous pipeline, the ALL output would be used to latch the data from the adder into the sink latch. The sink latch would then initiate the precharge phase by sending a pulse to the PSET input. Once NONE had gone high, indicating that the precharge was

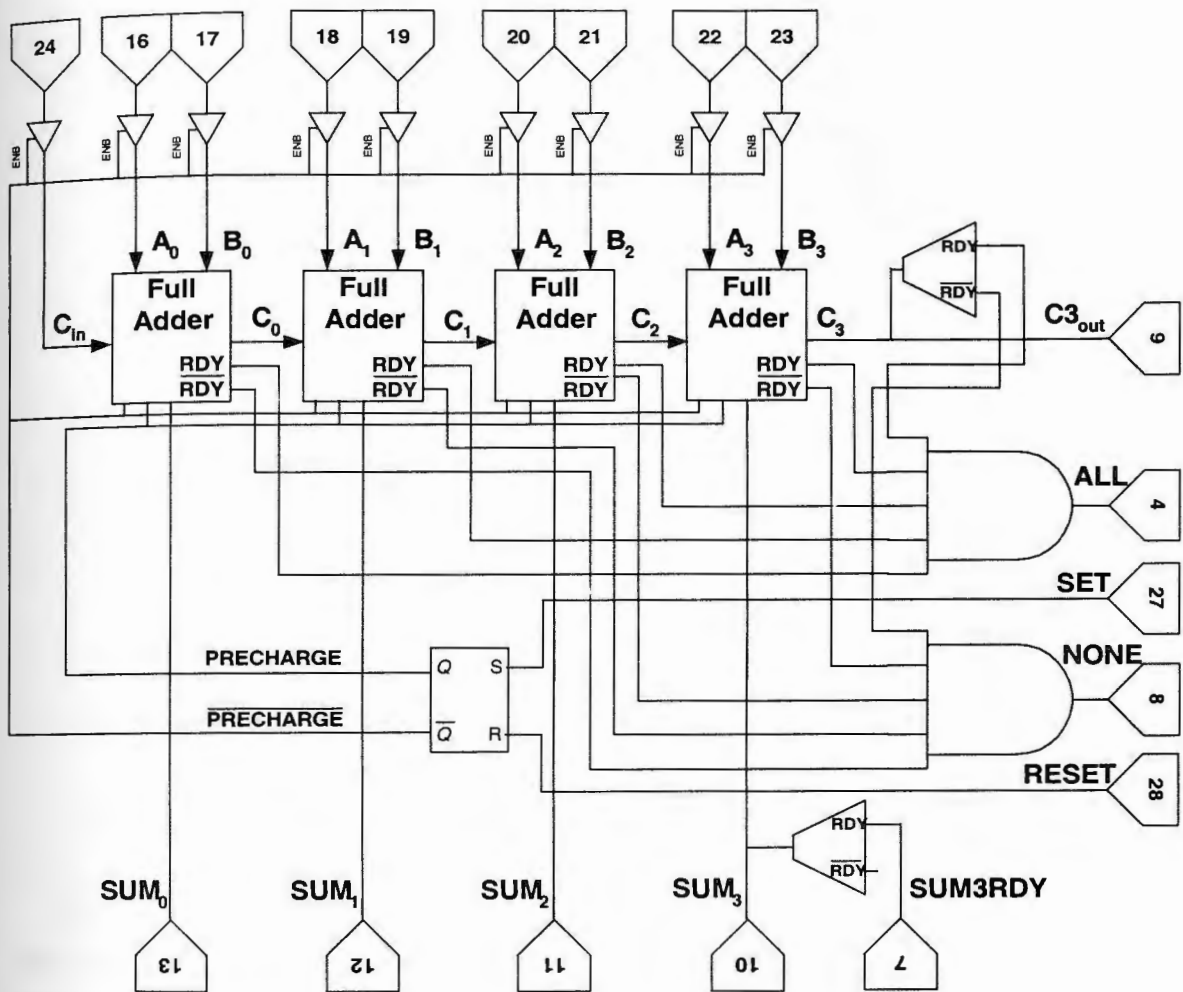


Figure 5.12: Centered Binary Plus 4-Bit Ripple Carry Adder

complete, the source latch, if data was available, would initiate the evaluate cycle by sending a pulse to the PRESET input.

This is an appropriate point to mention that a fully correct implementation would include in the creation of the NONE signal from not only the \overline{RDY} signals for each output, but also from the equivalent for each of the intermediate results within each full adder. To avoid an AND gate of impractical size, this would most likely be implemented on a modular basis: the full adder circuit diagram would be modified as shown in Figure 5.13.

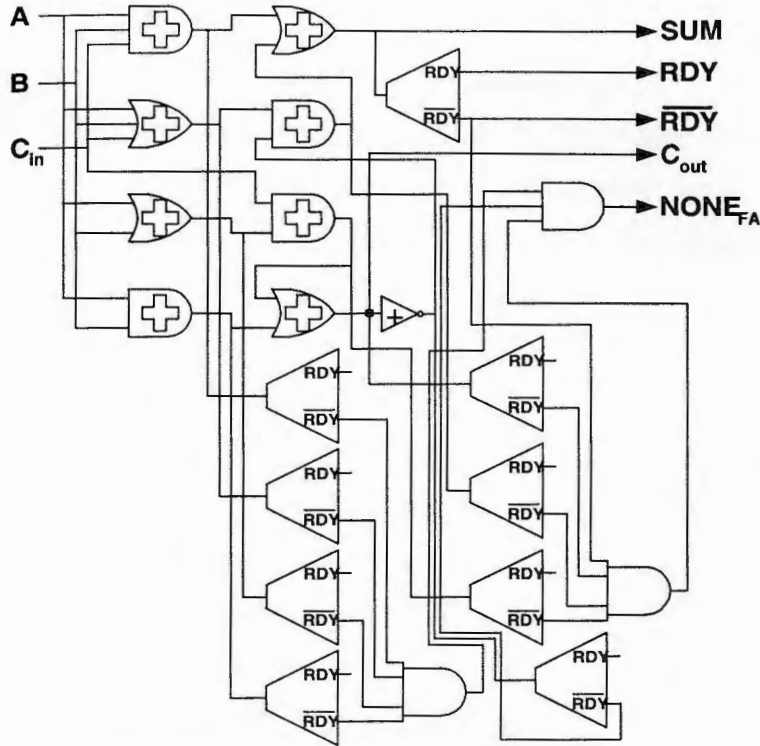


Figure 5.13: Centered Binary Plus logic Full Adder with NONE_{fa}

Note that the need to ensure that all intermediate results have returned to ϕ before the precharge phase can terminate leads to significant expansion of the circuit. This problem could be largely eliminated by the use of complex gates. In reality, however, the designer is likely to find that going to the extreme shown in Figure 5.13 is not necessary in the practical sense, for the following reasons:

- The load and other capacitance on the output lines (SUMs and Carry-Out) will in most cases be greater than that on the intermediate result lines, making it highly likely that intermediate result lines will have reached ϕ during the precharge phase before the outputs do.
- It takes additional time for the NONE signal to be generated once all lines have gone not ready, and more time for the reset on the precharge flip-flop to take effect. This provides a margin of error for intermediate values to become

adequately centered.

- Considering the extra time that will be used by the AND tree in Figure 5.13, the designer could just as easily build a short delay into the initiation of the evaluate cycle without adversely affecting comparative timing, allowing even more time for intermediate values to reach ϕ while reducing greatly the space requirements of the full adder circuits.

5.5.2 Testing strategy

Following the difficulty encountered and discussed in Section 4.5.2 regarding getting predicted results from elementary Centered Binary Plus gates in cases when one or more inputs were ϕ , it was decided to run static tests on the adder, in addition to those planned for dynamic operation.

The prime purpose of this experiment, however, was to demonstrate the varying completion times for the adder over a range of input sets. A short pulse was generated using a function generator; this was used to set the precharge flip flop, and was also used as a trigger to a pulse generator, which generated another short pulse delayed from the first. This second pulse was used to reset the precharge flip-flop. This second pulse was also used to trigger a dual trace oscilloscope, on which the output of the ALL signal was also displayed. In this manner, the delay between the beginning of the evaluate phase (the start of the flip-flop reset signal) and the completion signal (the ALL output) could be measured. The duration of the cycle could thus be measured and recorded. The input set could be modified at any time, and a new duration measured and recorded.

As it was desired to obtain some *a priori* prediction of adder performance relative to input set, in order to compare actual performance with predicted to confirm intended operation, a gate-level simulator was constructed. As it was desired only to get a rough prediction of performance, this software assumed that the delay for each gate-type construct in the circuit was equal. When run on all 512 possible input problems, the following gate delay predictions shown in Table 5.2 were computed.

The mean gate delay predicted is 9.3 gates.

Gate Delay (gates)	Frequency	%
5	1	$\ll 1\%$
6	7	1%
7	56	11%
8	124	24%
9	132	26%
10	72	14%
11	56	11%
12	40	8%
13	24	5%

Table 5.2: Results of Gate-Level Simulation of 4-bit Ripple Carry Adder

5.5.3 Testing results

Static testing

Several input patterns were applied to the adder in a static mode. As was the case with the elementary circuit testing discussed in Section 4.5.2, results were correct when all inputs (or a critical subset of inputs) were Valid; when these conditions were not met, the result came down on “one side or the other”. Again, this is likely due to output buffer conversion of ϕ to 0 or 1, although the time required for static measurements would allow for dissipation of the V_h precharge anyway.

Dynamic testing

Randomly selected input bit patterns were applied to the adder and the completion delay measured as described above. Table 5.3 lists the results, trial by trial.

From Table 5.3 it is difficult to see by inspection any more than a rough relationship between the input set and the completion time. It is clear, however, that the input set does affect the completion time. To determine if the completion times measured were, in fact, related to the input-set related performance of the adder as predicted by the gate-level simulator, a correlation was run between the number of gate delays as determined by the gate-level simulator and the actual measured completion time.

A correlation coefficient of 0.5832 was reported (a reasonably positive correlation). It was reported to be statistically significant at the $p=.000$ level - highly significant. It is therefore highly likely that the variation in completion time is due to the predicted operation of the adder circuit and that, therefore, the adder is operating as intended.

While the variation in completion time (from a tested minimum of 76.1 ns. to a maximum of 106.0 ns., only 39% greater) is not great, it is likely that there are constant-time factors that are having the effect of minimizing the variation. If we assume that the variation in actual completion time (excluding constant factors such as precharge time and output buffer delay) is roughly proportional to the variation in gate delays as predicted by the gate-level simulator, then we can estimate the constant time C as follows. Since t , the total time measured for completion, can be roughly given as:

$$t = C + d_p \cdot d_g$$

, where C is the constant time due to factors not related to the input set pattern, d_p is the number of gate delays as predicted by the gate-level simulator and d_g is the delay in nanoseconds *per gate delay*, then we can use our extreme measurements to set up a simple set of simultaneous equations in two variables:

$$106.0 = C + 12 \cdot d_g$$

$$76.1 = C + 5 \cdot d_g$$

Solving gives us:

$$d_g = 4.271ns.$$

and

$$C = 54.7ns.$$

Based on a predicted gate delay range of from 5 to 13 gate delays, we can estimate that our input set dependent delay - ignoring constant-time causes - will range from approximately 21 to 56 ns., a variation of 166%.

It is likely that running the stage isolated from output buffer influences will significantly lessen the constant time factor.

5.6 Summary

In this chapter we developed the design of Centered Binary Plus logic gates and stages. We saw that Centered Binary Plus logic has several advantages, and is fully capable of interfacing with latches as part of a Globally Asynchronous Locally Synchronous (GALS) pipeline. The technique proposed has significant advantages over each of the examined alternative methods of self-clocking.

We examined a 4-bit ripple-carry adder implemented as part of the proof-of-concept circuit, and presented test results showing input set related variations in completion time which were statistically shown to correlate very significantly with the predicted behavior as shown by a gate level simulator designed for the circuit.

A	B	C_{in}	Out	Time(ns)	A	B	C_{in}	Out	Time(ns)
0000	0000	0	0 0000	88.5	0001	0010	1	0 0100	89.3
0000	0111	0	0 0111	98.5	0001	0011	1	0 0101	86.2
0000	1000	0	0 1000	96.2	0001	1001	1	0 1011	94.7
0001	0010	0	0 0011	87.8	0001	1011	1	0 1101	91.5
0001	1110	0	0 1111	105.6	0001	1110	1	1 0000	100.7
0010	0010	0	0 0100	88.3	0010	0101	1	0 1000	98.8
0010	1110	0	1 0000	100.1	0011	0000	1	0 0100	88.1
0011	1010	0	0 1101	95.5	0011	1100	1	1 0000	101.1
0100	0101	0	0 1001	91.0	0100	1000	1	0 1101	101.4
0101	0001	0	0 0110	94.2	0101	0100	1	0 1010	90.5
0101	1101	0	1 0010	88.1	0101	1111	1	1 0101	86.3
0110	0010	0	0 1000	91.3	0110	1011	1	1 0010	92.6
0110	1001	0	0 1111	106.0	0111	0111	1	0 1111	87.2
0111	0100	0	0 1011	89.7	1000	0011	1	0 1100	94.5
0111	1111	0	1 0110	88.3	1000	1110	1	1 0111	96.7
1000	0000	0	0 1000	96.7	1001	1010	1	1 0100	84.5
1000	1100	0	1 0100	92.3	1011	0101	1	1 0000	101.8
1001	1000	0	1 0001	84.8	1010	0110	1	1 0001	91.9
1010	0011	0	0 1101	95.1	1011	0010	1	0 1110	94.4
1010	1111	0	1 1001	86.6	1011	1101	1	1 1001	88.5
1011	1011	0	1 0110	84.2	1100	1001	1	1 0110	91.1
1100	0111	0	1 0011	88.8	1101	0101	1	1 0011	90.0
1101	0010	0	0 1111	105.4	1110	0001	1	1 0000	100.2
1101	1110	0	1 1011	82.5	1110	0110	1	1 0101	86.3
1110	1010	0	1 1000	86.4	1110	1100	1	1 1011	83.7
1111	0110	0	1 0101	87.9	1110	1110	1	1 1101	78.7
0000	0001	1	0 0010	88.3	1111	0000	1	1 0000	99.4
0000	1101	1	0 1110	101.9	1111	1000	1	1 1000	90.7
0000	1111	1	1 0000	100.4	1111	1111	1	1 1111	76.1

Table 5.3: Timings of Adder Cycle Time Across Input Patterns

Chapter 6

Communications applications

Data communications is an increasingly important part of technology. Rarely is it understood, however, how pervasive the concept really is. For communication takes place over not only large but also very small distances. Data must be communicated from one part of an integrated circuit to another, or between integrated circuits in a Multi-Chip Module (MCM) or on a circuit board (for example, from main memory to and from the CPU). One of the two primary purposes of the backplane in systems and other digital devices is to *communicate* data among the circuit boards in the system.

For our purposes we will consider communication as the moving of digital data (whether by digital or analog communications media) from one location to another, placing no upper or lower bounds on the distance over which it is moved. We shall see that the information that can be derived by use of the detector of Figure 4.2 can be used to good advantage in enhancing the reliability of communications.

Reliability in communications on all scales is generally addressed under the general heading of “error-control coding”. We will not propose an alternative to error-control coding, but will instead show how the use of the information provided by the detection techniques covered in Chapter 4 can be used in conjunction with error-control coding strategies covered in the literature.[7, 6, 8, 9, 10]

6.1 Hardware and error detection/correction

Much attention was paid in Chapter 2 of this work to transient and static problems that can result in undefined logic levels occurring during the transmission of data from one place in a system to another. While static errors would presumably be detected by an adequate post-manufacturing testing process, transient errors can occur at any time. There are also cases in which new static errors can appear; for example, a cable can be broken, a connector detached or aging of a circuit can cause bus line or device failure.

Many schemes address the detection and correction of such errors.[6] The simplest of these schemes remains the single parity bit found in some semiconductor memories and common in communication designs. It is axiomatic that a single parity bit is limited to detecting 1-bit errors. Errors involving an even number of bits cannot, by definition, be detected by such a scheme. Additionally, the scheme is limited to detection only - an error indication implies that an odd number of bits (usually one) are in error, but cannot identify those bits. Schemes involving a larger number of check bits are generally able to detect a larger number of errors than a 1-bit scheme, and may also be able to point at the bit in error. In a binary system, correction requires merely being able to identify the offending bit; with only two possible values, correction is comparatively trivial.

Now consider what effect an undefined logic value might have on a typical circuit based on a 1-bit parity design. As we have discussed in Section 2.2.2, circuitry is going to resolve an undefined logic level into a valid 0 or 1. If the value happens to be the correct one, then no parity error will be detected and the user of the results - human or system - will never be made aware of the possible problem. If, on the other hand, the resolved value is the incorrect one, a parity error will be signaled and the received word will be considered incorrect.

In the above example, we have an excellent illustration of the consequences of discarding information. In one result, the value passed on is presumably correct, but lost was a possible indication that a problem exists with the transmission link. The alternate result indicates the existence of a problem, but the *location* (bit-wise) of

the problem is lost.

6.2 Error-control coding

In their book, Error Control Coding for Computer Systems, T. R. N. Rao and E. Fujiwara begin Chapter 1 thusly:

“In computer systems, large amounts of data move between various subsystems. For instance, the data traffic between the CPU and main memory may be of the order of 100 million bits every second. Even though the systems are designed for very high reliability, there are bound to be a few errors in these communications caused by such things as atmospheric, electrical noise, component or device malfunctions, or sometimes design or program faults. It is important that the system detect these errors as and when they occur. Some remedial action such as *error correction* or *error recovery* must take place before a more serious situation like a system crash arises.” [6]

Rao and Fujiwara’s text provides excellent coverage of the topic of error-control coding, and the reader is referred to that work for an in-depth understanding, including analyses of the probability of various errors in different channel models. We will cover the topic of error-control coding in only enough detail to provide an adequate background for the adaptations proposed in this chapter.

6.2.1 Channel models and errors

When data is transmitted from one site to another, bits may arrive as transmitted or may be received as some other value. Depending on the characteristics of the communications channel, different types of data modification may be possible, with varying probabilities. An examination of some typical models will lead the way to a model most appropriate for the contribution described in this chapter.

Classical (symmetric) error model

A *binary symmetric channel* is one in which errors may be of the $0 \Rightarrow 1$ or $1 \Rightarrow 0$ variety, with equal probability. Additionally, the errors are bitwise independent - an error in one bit neither increases nor decreases the probability that any other bit will be in error. [6]

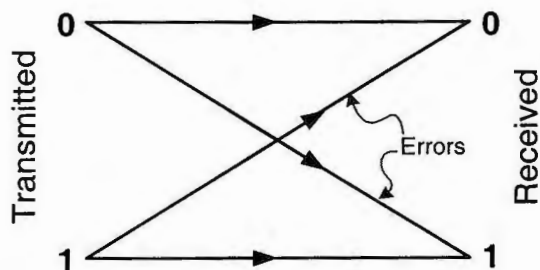


Figure 6.1: Symmetric Error Model

Figure 6.1, adapted from Rao and Fujiwara[6], summarizes the behavior of the binary symmetric channel.

Asymmetric error model

For binary symmetric channels, we mentioned that the probability of a $0 \Rightarrow 1$ error was equal to that of a $1 \Rightarrow 0$ error. This is the constraint that is relaxed to form the *binary asymmetric channel*.

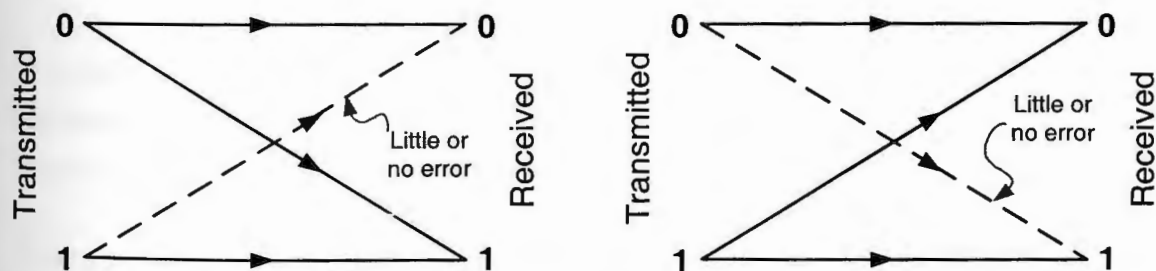


Figure 6.2: Ideal Asymmetric Error Model

In an *ideal* asymmetric channel, as shown in Figure 6.2, the probability of one of the error transitions is virtually zero.

Unidirectional error model

The unidirectional model is a “word-by-word” special case of the asymmetric error model. Rao and Fujiwara define it as follows: “Both 1-errors and 0-errors can occur in the received words, but in any particular received word, all errors shall be of one type; these errors are characterized as *unidirectional errors*.” [6]

Binary erasure error model

Rao and Fujiwara define a *binary erasure model*. In such a channel, $0 \Rightarrow 1$ and $1 \Rightarrow 0$ do not occur, but there may be *erasures* - a change of a 0 or 1 to a non-existent value. This channel is depicted in Figure 6.3.

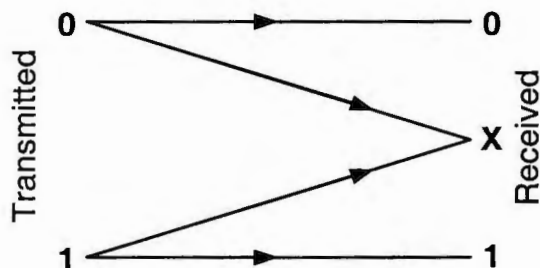


Figure 6.3: Binary Erasure Error Model

This diagram should be of particular interest to us, as it implies the existence of a *third state* - neither 0 nor 1. In actuality, such a non-value state need not be signaled by a value close to V_h ; any other method of determining that a bit is not known (such as a plane-wise parity error in a memory) may be used. [6, 7]

General analog model

Our discussion in Chapter 2 regarding the effects of using what is inevitably analog circuitry to process digital values leads us to a more general error model of the communications channel.

As the state of a practical binary circuit or channel driven by a circuit is not a dichotomy of values, but a continuum, we can depict the change in a transmitted data bit over the communications process in a diagram *similar* to those used in the previous digital channel examples (although the characterization of Figure 6.3 as a truly digital channel is open to question). This is shown in Figure 6.4.

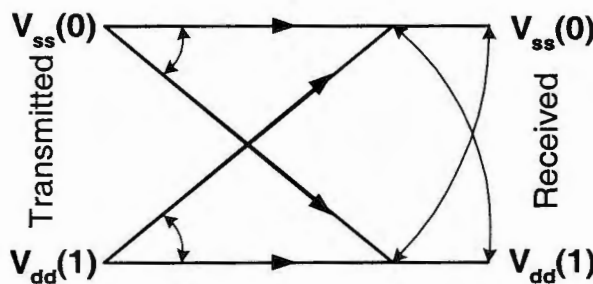


Figure 6.4: General Channel

It can be seen that the general case can be simplified into any of the previously shown channel error models, dependent on the distribution of error frequencies along each of the arcs shown in Figure 6.4.

Symmetric with erasures model

We can take the general model shown in Figure 6.4 and “digitize” it. If we use a typical division point of V_h , then the general model simplifies to that of Figure 6.1.

If, however, we also wish to detect “erasures”, which we will now define as bits that fall within our undefined zone, we have the diagram shown in Figure 6.5.

Now adopting our three-state notation of Chapter 3, we can say that an information bit that is transmitted as a 0 may be received correctly as a 0, or incorrectly as a 1 or a ϕ . Symmetrically, an information bit that is transmitted as a 1 may be received correctly as a 1, or incorrectly as a 0 or a ϕ . The probabilities of any of these outcomes is dependent on the specific characteristics of the communications channel; their determination is outside the scope of this work.

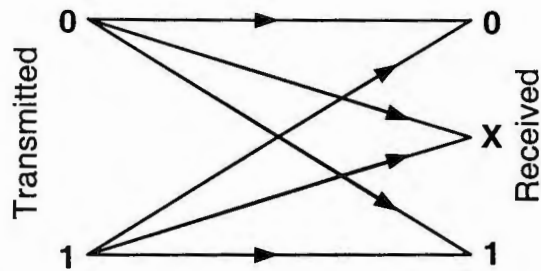


Figure 6.5: Symmetric Channel with Erasures

A caution about transmitting zoned binary

Heretofore we have used a working assumption, first made in Section 2.1, that the boundaries between logic 0 and ϕ and that between ϕ and logic 1 are placed at $1/3 V_{dd}$ and $2/3 V_{dd}$ respectively. The implementer must be cautioned against assuming that this is an always appropriate choice. Let us consider the transmission of a zoned binary bit from one location to another.

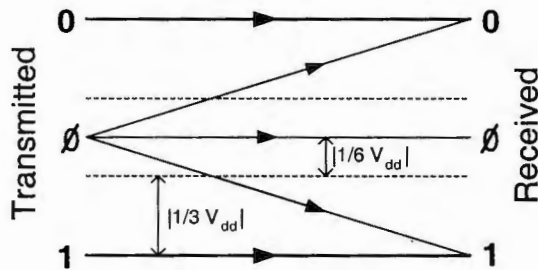


Figure 6.6: Transmission of ϕ

In Figure 6.6, only the “digitized” paths of the ϕ state are shown; the error transitions shown in Figure 6.5 are still present, but have been omitted from the figure for clarity.

We see that we must admit for consistency the possibility of $\phi \Rightarrow 1$ and $\phi \Rightarrow 0$ errors.

Returning to our analog equivalence, we realize that for a $0 \Rightarrow \phi$ or a $1 \Rightarrow \phi$ transition, there must be an absolute change in analog value of $1/3 V_{dd}$, using our boundary divisions as defined in Section 2.1 and shown as dotted lines in Figure 6.6.

But for a $\phi (V_h)$ to 1 or 0 transition, there need be an absolute change in analog value of only $1/6 V_{dd}$. Such errors may be even more dangerous, as they will be, by definition, undetectable except by error-coding techniques.

The designer must consider this problem, especially when contemplating the transmission of encoded zoned binary data over long or noisy communications channels, and consider moving the boundaries for such exceptions to, perhaps, $1/4 V_{dd}$ and $3/4 V_{dd}$, thereby making the analog "distance" between any valid state and the adjoining state(s) equal to $1/4 V_{dd}$.

6.2.2 Distance

All error-control codes are characterized by the fact that not all of the words that can be formed by different combinations of bits are valid. Those that are, are termed *codewords*, while those that are not are indications of error.

The *Hamming distance* between two equal-sized strings of binary bits can be computed by counting the number of bit positions in which the values of those two strings differ. The *distance* (d_{min}) of a code is the minimum Hamming distance between all pairs of codewords.[6]

The *distance* of a code serves as an indicator of the theoretical ability of the code to detect and/or correct errors. Three theorems from Rao and Fujiwara's text are quoted:

"It is necessary and sufficient that the distance (d_{min}) of a code is at least d in order to detect any error pattern of weight $d - 1$ or less."

"A code C can detect and correct all patterns of t or fewer errors if and only if the code has minimum distance $\geq 2t + 1$."

"A code can correct any combination of t errors and detect up to d errors ($d \geq t$) if and only if the d_{min} of the code $\geq t + d + 1$." [6]

A distance-2 code, therefore, can detect one-bit errors and correct none. A distance-3 code can detect up to two-bit errors, or, if error correction was required, could detect and correct one-bit errors. To detect up to two-bit errors while correcting one-bit errors would require a distance-4 code.

6.2.3 Simple parity code

A simple parity code is probably the cheapest and easiest error-control coding scheme in use. It uses one parity bit (or “check” bit) to “protect” any number of data bits.

Intuitively, to generate a parity check bit, we count the number of data bits with a value of 1, and then set the check bit to ensure that the number of ones (including the check bit) is always odd (for “odd parity”) or even (for “even parity”).

It is easy to see why the simple parity code is a distance-2 code. If you take a valid code word (some number of data bits plus an appropriately computed parity check bit), and change one data bit position (from 0 to 1 or from 1 to 0), you must also change the parity bit. Therefore each codeword differs from any other codeword by a minimum Hamming distance of 2 bits.

With a d_{min} of 2, the simple parity code is capable of detecting a single-bit error.

6.2.4 SEC and SEC/DED codes

There are a number of linear codes that provide minimum distances of 3 and 4.

The distance-3 Hamming code can be used as either a DED (double error detecting) or a SEC (single error correcting) code. By adding an overall parity bit to the distance-3 Hamming code, we obtain a distance-4 code, which can be used for DED and SEC purposes simultaneously. Such a code is referred to as a SEC/DED code.[6]

In our discussion later in this chapter, we will not be concerned with the construction of these codes and their implementation with encoders and decoders, for which Rao and Fujiwara can be referred to. We will, however, treat them as functional units that can be used to detect and/or correct errors on the basis of the received code alone.

6.3 Error location with zoned binary detector

It is clear that a bit received and identified as being in the uncertain zone by our detector of Figure 4.1 has at least a strong potential for being in error. So an array

of these detectors - one for each bit of a received word - can provide additional information regarding the location of a possible error that would otherwise be lost.

It is, of course, possible that an error occurs that causes the bit in error to take on a valid value opposite to what was intended. In this event, our detector would not be able to identify it. In this case, we would be no better off than without the detectors, but no worse off either. The error detection circuitry based on error-control coding would at least detect the error, if not correct it.

But if an error-correcting code scheme is in use, why implement the detector scheme in addition? Does the additional location information it might provide gain us anything?

It would seem this is so, according to Rao and Fujiwara:

“Because the positions of the erasures are known, the correction of erasures in a received word will be simpler than the correction of errors. Thus, a given code that is used for error correction can be employed more efficiently to correct erasures.”[6]

It should be clear from earlier in this chapter that a received value of ϕ functionally indicates an “erasure” - that is, it has changed from a 0 or 1 to *neither*.

6.3.1 An easy case: the unidirectional channel

Using the known location of erasures in the unidirectional channel described in Section 6.2.1 provides a clear and easy path toward enhancing communications reliability. We know by definition that errors in a unidirectional channel word are all of the same direction: $0 \Rightarrow 1$ or $1 \Rightarrow 0$. Therefore, the proper binary value of *any* error is known, provided only that we can identify its location. As our detector points to the location(s) of erasures, those locations can simply be set to their proper value. The enhancement in reliability comes from the fact that this strategy effectively moves the boundaries between logic 0 and logic 1 to a point $2/3$ towards the only error transition that can be made. More precisely:

- When the only possible error direction is $0 \Rightarrow 1$, any ϕ should be set to 0, effectively moving the boundary between logic 0 and logic 1 to $2/3 V_{dd}$.
- When the only possible error direction is $1 \Rightarrow 0$, any ϕ should be set to 1, effectively moving the boundary between logic 0 and logic 1 to $1/3 V_{dd}$.

The same strategy could be applied to an ideal asymmetric channel, as described in Section 6.2.1.

6.4 Error correction strategies for ϕ errors

In this section, we shall see how the uncertainty detector can be used to indicate erasures to schemes suggested by Rao and Fujiwara.[6] We shall also extend these approaches into a channel model not considered in that text: the “symmetric with erasures model” in Figure 6.5 that we developed from the general model shown in Figure 6.4. This model requires less *a priori* knowledge about channel characteristics than other discussed models, and so should be more widely usable.

Consider that a simple parity scheme with a single check bit can detect one error in a received word and correct none, as it is a distance-2 code. As this is a theoretical limit of the coding structure itself, we must step “outside” the code decoding circuitry if we wish to enhance the performance of a receiving device using such a simple code.

Likewise, coding schemes developed to have more capability, such as DED and SEC/DED codes, have their theoretical limits. An external approach must be used - that is, the input must be *conditioned* in some way by taking advantage of the additional knowledge of error location.

Provided that we can identify the location of a bit in error by virtue of its being an *erasure*, we know one critical fact about that bit: it was originally transmitted as a 0 or as a 1. This may seem trivial, but it points us toward a correction strategy. The strategy involves the generation of alternative received words, varying only in the values of the bits that were identified as unknown.

6.4.1 Strategy for simple parity codes

Consider a receiver utilizing codewords based on a simple parity check bit. This is a distance-2 code, and so should be capable of detecting a one-bit error and correcting none. Consider, however, the following example:

If a received word is “0 1 0 0 ϕ 1 1 0 1”, then it is likely that the transmitted word was either “0 1 0 0 0 1 1 0 1” or “0 1 0 0 1 1 1 0 1”. We can now use the error detection capability of the simple 1-bit parity method to determine which alternative is *not* in error.

Our strategy for correcting single bit unknowns (“erasures”) is therefore to generate two words from our received word, differing only in the value assigned to the unknown bit. Both are then processed by a parity checker (either in parallel by two identical checking circuits, or sequentially by one) to choose which of the generated words is the valid codeword.

6.4.2 Extension of strategy to DED codes

A DED code is a distance-3 code, which implies that it should be capable of either correcting a one-bit error or detecting two-bit errors and correcting none. The difference between detecting and correcting is really one of determining the location of the error.

Our strategy is similar to that used for a simple parity code, but since we have *two* unknown bits (erasures), there are *four* possibilities for the settings of those two bits. The four words generated by these four possibilities are independently processed by DED checkers; the one that is error-free is selected.

6.4.3 Extension to SEC/DED codes

SEC/DED codes are distance-4 codes, which implies that they can detect 3-bit errors or, alternatively, detect 2-bit errors while correcting one error.

For erasure errors, however, “error location capability allows a distance-4 code (SEC-DED code) to correct up to three errors.” [6]

Suppose we have a received word with a 3-bit erasure. We can generate 2^3 alternative words, and check them all for errors. This is certainly getting to the point where a sequential approach is more practical, as providing eight independent, parallel code-checking circuits can be space-consuming. If, of course, time constraints were extreme enough, the expenditure of space might be warranted.

6.4.4 Extension to the general model

The general channel depicted in Figure 6.4 yielded more possible error transitions (as shown in Figure 6.5) than was the case in either the “classical” symmetric error model (Figure 6.1) or the Binary Erasure Error Model (Figure 6.3). A transmitted 1 may be received in error as a 0 or as a ϕ , while a transmitted 0 may be received in error as a 1 or as a ϕ .

It should be intuitively clear that we can no longer correct three errors. Since we can no longer “point” to all three error locations, it will “cost” us to determine the location of that non-erasure error.

We can still, however, do better than correct a single one-bit error, the theoretical maximum that we could accomplish with the symmetric error model of Figure 6.1.

The strategy described earlier in Section 6.4.1 can be adapted to fit this new model, as follows:

- Generate two alternatives of the received word, based on the two possible values of the erasure error (whose location is known).
- Route these two alternatives to independent SEC/DED checkers.
- Select the output from the checker that reports a single, corrected error.
 - If the only error was an erasure error, *both* checkers will output the correct codeword; one checker will indicate a single, corrected error, while the other will indicate no error.
 - If there is a single, non-erasure error, *both* checkers will output the correct codeword and report a single, corrected error.

- If there is both a single erasure and a single non-erasure error, one checker will output the correct codeword and indicate a single, corrected error, while the other will output an incorrect codeword and indicate a double error.

Note that this method can be adapted to a circumstance in which *two* erasure errors were detected. In this case, a value could be arbitrarily assigned to the second erasure (making it either the correct value or a non-erasure error), and sent to the same circuitry.

It should be pointed out that it is not even necessary for this second erasure to be assigned the *same* arbitrary value in the two generated alternatives. This may simplify the design of the circuitry generating the alternatives.

We have seen how the information from our uncertainty detector can be used to extend the correction capabilities of standard error-control coding schemes to handle a model in which both erasures (transitions to ϕ) and classic $1 \Rightarrow 0$ and $0 \Rightarrow 1$ errors can be received.

6.5 Implementation example: simple parity code

We can now proceed to illustrate the design of a correction system appropriate to the error-control coding strategies of both Sections 6.4.1 and 6.4.4. A very simple 4-bit codeword scheme will be shown.

Figure 6.7 is not a complete circuit diagram. Depending on the specific error-control coding scheme being used, there would be additional desirable outputs. Specifically, one might find various error indicators useful, such as:

- An indicator that at least one of the inputs was an erasure (ϕ).
- An indicator that at least two of the inputs were erasures (ϕ).
- For a SEC/DED code, an indicator that more than two of the inputs were erasures. (ϕ).

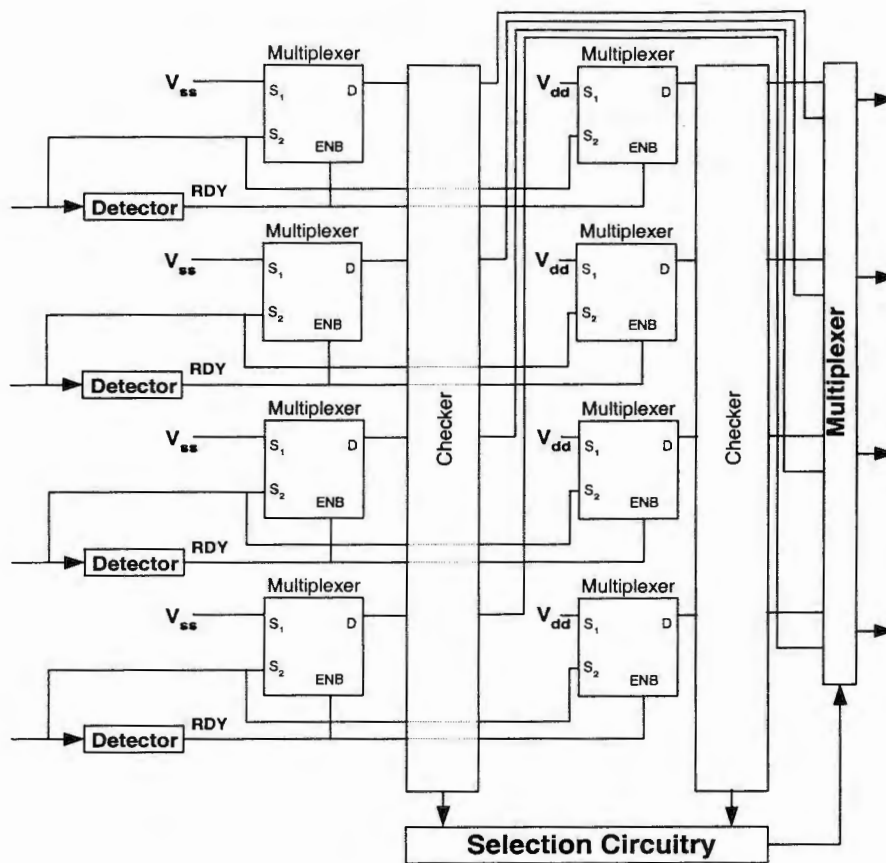


Figure 6.7: Illustrative Correction System

- An indicator that errors are present that could not be corrected.
- An indicator that no errors of any kind were present.

The multiplexers at the inputs to the two checkers are used to either (1) pass the original value of the input bit to the checker, or (2) pass a 0 or 1 (for the left or right checker, respectively) to the checker in place of the original input bit (for erasures).

The two checkers each return a “parity correct/error” signal to the “Selection Circuitry”, which chooses which checker’s output is to be used.

6.6 The detector once again revisited as a decoder

In Section 4.3.2, we mentioned that the undefined range that can be discerned by the detector need not be a natural outcome of circuit conditions we wish to detect - it can be explicitly coded, should there be a valid need.

Early in this chapter, we defined “communications” as “the moving of digital data (whether by digital or analog communications media) from one location to another, placing no upper or lower bounds on the distance over which it is moved.” There are many forms of transport media; certainly not all depend on varying voltage levels to represent a 0 or 1. There may be many transmission modes, and various modulation/demodulation methods appropriate to them.

It is possible that a demodulation subsystem may detect an indeterminate state for one or more bits in a received word of digital data. In such a circumstance, that subsystem could emit as output a zoned binary value, encoding the *uncertain bit(s)* as ϕ . The methods of this chapter could then treat those bits as erasures.

6.7 Partial utilization: some gain at lower cost

Sometimes the tradeoff of space (or time) in order to achieve a given performance gain is not practical. This must be judged on an implementation by implementation basis by the designer. The methods already discussed in this chapter do provide significant performance gain, but at the undeniable cost of either:

- at least two code-checker circuits, implemented in parallel, with associated multiplexers and selection circuitry, or
- a single code checker, with required circuitry to sequentially present the alternatives to it until a successful decoding into a codeword occurs, the impossibility of doing so is recognized, or the list of alternatives is exhausted.

Space is impacted to some degree, and, in the second approach, time is also lengthened, which may not be practical in a time-constrained system.

Is there any other way in which the information provided by our detector can be used to good advantage, while not requiring such a significant expenditure of resources?

6.7.1 Code-independent advantage

Simply by detecting that one or more bits are in the uncertain range provides the receiver with more information than it had. As this condition would indicate some measure of difficulty with the communications media or transmitting device, it could signal an actual or developing problem *before* it was detected by the code checker, if any.

In fact, it is simple to link detectors together in such a way as to provide an indication when more than one "erasure" is detected in the same received word, providing an indication of the possibility of a two-bit error, one that would not be detected by, for example, a simple one-bit parity code checker. While this is obviously not the only kind of two-bit error that can occur, it will certainly detect some of them.

Additionally, we might refer to the simple application illustrated in Figure 4.9. For an external parallel input, for example, ANDing the \overline{RDY} signals obtainable from the detectors for all lines would provide a single signal indicating the probability of a broken or disconnected cable, or a totally malfunctioning communications link.

6.7.2 Simple set to zero with uniform distribution of erasure errors

Consider the simple expedient of setting all ϕ inputs to 0. [One could just as easily set them all to one, or set them to one or zero depending on the bit position - it is truly arbitrary, unless there is *a priori* knowledge about the error distribution (or data distribution) that would bias the decision one way or the other.] We will assume for the moment that the distribution of correct values when ϕ is detected is a dichotomy with a probability of .5 for each.

Simple one-bit parity checker

Use of this approach would gain no operational advantage with a simple one-bit parity code (distance-2) checker, other than those mentioned above in Section 6.7.1. It would have an equal probability of causing a bit that would have been correctly interpreted as a 1 (greater than V_h but less than $2/3 V_{dd}$) to be forced to a zero, causing an error. While this is counter-balanced by the possibility that its proper value was a zero, it is at best a draw.

SEC/DED codes

Consider the possible consequences of setting erasure bits to zero, or some other arbitrary assignment:

- When there is one error, and that error is an erasure: setting the erasure bit to zero and passing the resulting word to the SEC/DED code checker will result in either:
 - if 0 was the correct value, no error will be indicated, and the output will be correct, or
 - if 0 was the incorrect value, the SEC/DED checker will correct the error, a single, corrected error will be indicated, and the output will be correct.
- When there is one error and that error is not an erasure: there is no impact. The error is corrected by the SEC/DED code checker.
- When there are two errors, and both are erasure errors: setting both erasure bits to zero and passing the resulting word to the SEC/DED code checker will result in one of the following:
 - if 0 was the correct value for *both* bits, no error will be indicated, and the output will be correct, or
 - if 0 was the correct value for one of the bits and the incorrect value for the other bit, then the SEC/DED checker will correct the remaining error, a single, corrected error will be indicated, and the output will be correct, or

- if 0 was the incorrect value for *both* bits, then the SEC/DED checker will detect and indicate a double-bit error, and the output will be incorrect (but this will be known because of the double-bit error indication).
- When there are two errors, and one is an erasure and one is not an erasure: setting the erasure bit to zero and passing the resulting word to the SEC/DED code checker will result in either:
 - if 0 was the correct value for the erasure, the SEC/DED checker will correct the remaining, non-erasure error, a single, corrected error will be indicated, and the output will be correct, or
 - if 0 was the incorrect value for the erasure, then the SEC/DED checker will detect and indicate a double-bit error, and the output will be incorrect (but this will be known because of the double-bit error indication).
- When there are two errors, and both are non-erasures: there is no impact. The SEC/DED checker will detect and indicate a double-bit error, and the output will be incorrect (but this will be known because of the double-bit error indication).

We can determine that there will be no gain over a system in which the received value of all bits in the region of V_h are allowed to resolve themselves into a 0 or a 1 by chance.

Consider that being consistent in the assignment of 0 or 1 will have no effect on the outcomes listed above. Assignment as a 1 or a 0 is as likely to be correct as incorrect.

Since the assignment of the value in the above scheme is arbitrary, and consistency confers no advantage, a random assignment (such as might occur by allowing the values around V_h to resolve themselves) works just as well.

But this conclusion does not eliminate the possible use of this simplified approach in situations in which the distribution of values within ϕ is *not* uniform, as we shall see in the next section.

6.7.3 Simple set to most probable value with asymmetric distribution of erasure errors

In Section 6.3.1 we discussed the simple expedient of setting an erasure bit to the “error-susceptible” value for a unidirectional channel or *ideal* asymmetric channel. For both of these types of channels, we possessed *a priori* knowledge that, for any given word, the probability of one of the two possible error transitions is very close to zero. Therefore, knowing that only one of the two transmitted values could be “corrupted” during transmission implied that any “corrupted” value received had to have been transmitted as the “corruptible” value, and so it could be set to that value.

If we have an asymmetric channel, even if not an *ideal* asymmetric channel (characterized by the fact that the probability of one of the two possible error transitions is very close to zero), the negative conclusions of Section 6.7.2 may be mitigated.

If the probabilities of the $1 \Rightarrow 0$ and $0 \Rightarrow 1$ error transitions differ from .5 significantly, the assignment of erasures to 0 or 1 is no longer arbitrary, and so modifying the strategy to set erasures to the most “corruptible” value may yield gains. The designer will have to consider the relative probabilities involved, together with any other characteristics of the communications channel, in deciding whether to implement any partial approach.

6.7.4 Possible enhancements

There are three possible modifications to the approaches discussed in this section, which may be used to some advantage.

Simplified detector

The techniques described in this section do not require a full detection capability. If, for example, it was desired to set all ϕ inputs to 0, which might be desirable in processing received words from an ideal asymmetric channel, one could simply pre-process each input as shown in Figure 6.8.

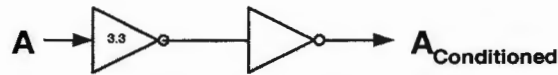


Figure 6.8: Input Bit Pre-Processing ($\phi \Rightarrow 0$)

As the “3.3 inverter” will not transition to an output of zero until the input rises out of the ϕ range into the range of logic level 1, all inputs are conditioned by the pre-processing circuit such that all inputs in the ϕ range will be received as logic level 0.

It should also be pointed out that the designer has the option of varying the transition point of the inverter using the design equations in Chapter 4 so that it will occur at some point other than $2/3 V_{dd}$, in order to best fit the error distribution of the channel.

Post-toggling two incorrect erasures

In one of the cases described under SEC/DED codes in Section 6.7.2, we described the consequences when there were two erasures. For 25% of the cases (in a uniform distribution), *both* erasures will be set *incorrectly* by the simplified scheme discussed in that section, and a double-bit error will be detected and reported; the output will be unusable.

By detecting:

- the double erasure (as opposed to any other double-bit error), and
- the double-bit error returned by the code checker,

we can post-process those two bits using a circuit such as that depicted in Figure 6.9.

In this manner, we can correct those two bits with as much confidence as we could in the double-checker scheme discussed earlier in this chapter. While there is additional space expended on this circuitry, it is not as much as a full dual checker implementation, while it does correct more than other single-checker approaches discussed. As always, the designer must consider the tradeoffs involved, especially

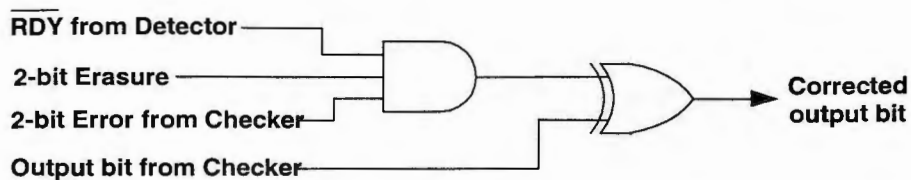


Figure 6.9: Post-Processing for Two Erasures after SEC/DED Checker

including the comparatively unlikely possibility of a word with two erasures plus one full error; as there is no checking done following post-processing, such an error would be neither corrected nor detected.

6.7.5 Special case: Bridge detection and correction for bus communications

It should be again emphasized that the techniques suggested in this chapter are meant to be, above all, *practical* techniques. This implies that, in cases in which special circumstances exist, the designer must as always be alert to the possibility of cost-effective modifications to the underlying concepts. As an example of such an implementation, we consider here the special case of an internal data bus in which temporary bridges are of specific concern.

In Section 2.2.1, we discussed various physical defects that could cause undefined logic levels. Figure 2.2, reproduced here as Figure 6.10, illustrated one of these defects - a bridge between adjacent bus lines.

It is clear that a bridge between two adjacent bus lines can produce a two-bit error. We know from our earlier discussion that we require a distance-3 code to be able to correct two erasures. We also found that it was necessary to generate *four* alternatives, passing them through four parallel distance-3 code checkers (or sequentially through one).

Consideration of the special case of bridges, however, allows us to eliminate two of the alternatives. For if, in Figure 6.10, the driven value of D_1 and D_2 are *both* 0 or *both* 1, then there is no error - in fact, the effects of the bridge will be undetectable. Only when one of the driven values is 0 and the other 1 will there be a potential

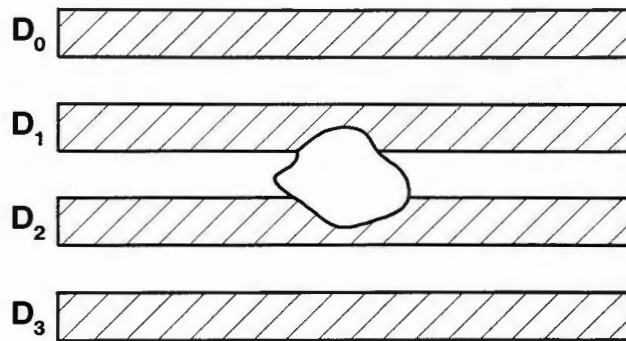


Figure 6.10: Physical Bridge (Short) Between Two Adjacent Bus Lines

problem. Additionally, only when the resistance of the bridge is low enough will the values be pulled “toward” each other enough to become undefined; if not, they retain their proper, driven values. In the former case, under the reasonable assumption (for parallel bus lines) that both lines are driven and loaded equally, the effect of our low-resistance bridge will be to create two adjacent bit values in the undefined zone.

Since we need to check only two alternatives, we need only two parallel distance-3 code checkers, very similar to the arrangement shown in Figure 6.7. That figure need be only slightly modified, as shown in Figure 6.11, by alternating the V_{ss} and V_{dd} multiplexer inputs so that both “01” and “10” patterns will be generated for any pair of adjacent erasures.

Again, the simplicity of this arrangement for correcting a two-bit error depends on an *a priori* understanding of the defects that are likely to occur. While this circuit would also properly correct a single-bit erasure, a two-bit erasure in which the proper values were “00” or “11” would not be corrected - instead, the circuit would indicate an uncorrectable error.

6.8 Comparison with classic method

It might be asked how these methods compare with the use of code-checking circuits alone. To illustrate, we use the example of a 9-bit parity checker/corrector circuit fabricated on our proof-of-concept circuit, as discussed and tested in the following

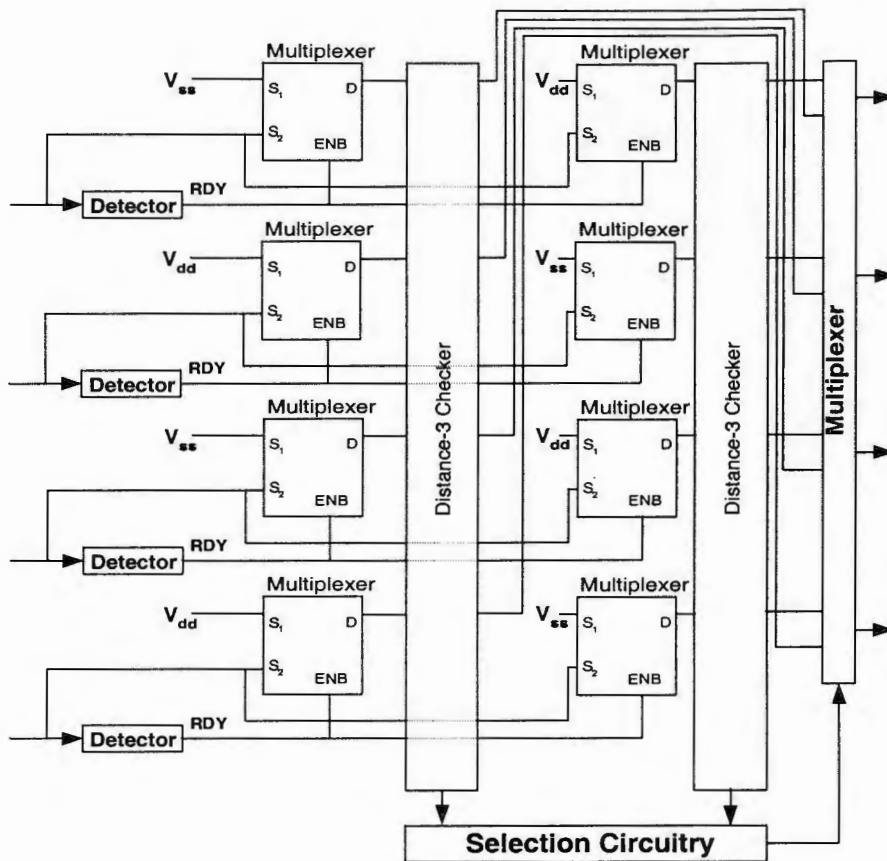


Figure 6.11: Distance-3 Correction System for Adjacent Bus Line Bridges

section, compared with a simple 1-bit parity checker. Table 6.1, limited to those cases in which a maximum of three errors of both types appear in a 9-bit received word, details the differences in capability based on different input conditions, including patterns that can be successfully handled by *neither* checker.

The percentage shown for each condition that can be handled by each checking scheme assumes a uniform distribution across ϕ : that is, an equal number of ϕ inputs would be interpreted as zeros and ones by the classic parity checker.

The experimental circuit displays results superior to the classic simple parity checker when there is a single erasure. The simple parity checker is superior in detecting errors when there are both a single erasure and one or two full errors in the same word. The results are identical or mixed in other cases.

Errors		Parity Only				Corrector Circuit			
Era- sures	Full Errs	Good	False Pos	False Neg	Ind Err	Good	False Pos	False Neg	Ind Err
0	1	0.0	0.0	0.0	100.0	0.0	0.0	0.0	100.0
1	0	50.0	0.0	0.0	50.0	100.0	0.0	0.0	0.0
0	2	0.0	0.0	100.0	0.0	0.0	0.0	100.0	0.0
2	0	25.0	0.0	25.0	50.0	25.0	0.0	25.0	50.0
0	3	0.0	0.0	0.0	100.0	0.0	0.0	0.0	100.0
3	0	12.5	0.0	37.5	50.0	25.0	0.0	75.0	0.0
1	1	0.0	0.0	50.0	50.0	0.0	0.0	100.0	0.0
1	2	0.0	0.0	50.0	50.0	0.0	0.0	100.0	0.0
2	1	0.0	0.0	50.0	50.0	0.0	0.0	50.0	50.0

Table 6.1: Comparison with Classic Parity Checker

6.9 Fabricated 9-bit parity-based corrector experiment

A 9-bit parity-based correction circuit, similar to the 4-bit version shown in Figure 6.7, was implemented, with minor enhancements. We show the implemented version (as a 4-bit example for visibility) in Figure 6.12.

Two enhancements are shown:

- The P_{in} signal is used to set “odd” or “even” parity.
- A signal D_ϕ is generated such that one or more ϕ inputs will set it to 1.

6.9.1 Actual design topology

For reasons of extensibility to any number of bits, the actual design implemented a “bit-slice” approach. A circuit was designed that contained all one-bit components required for the detector, input multiplexers, two parity-based checkers and the output multiplexer, such as shown in Figure 6.13.

Using this approach led to space efficiency as well as to extensibility to greater than 9-bit inputs.

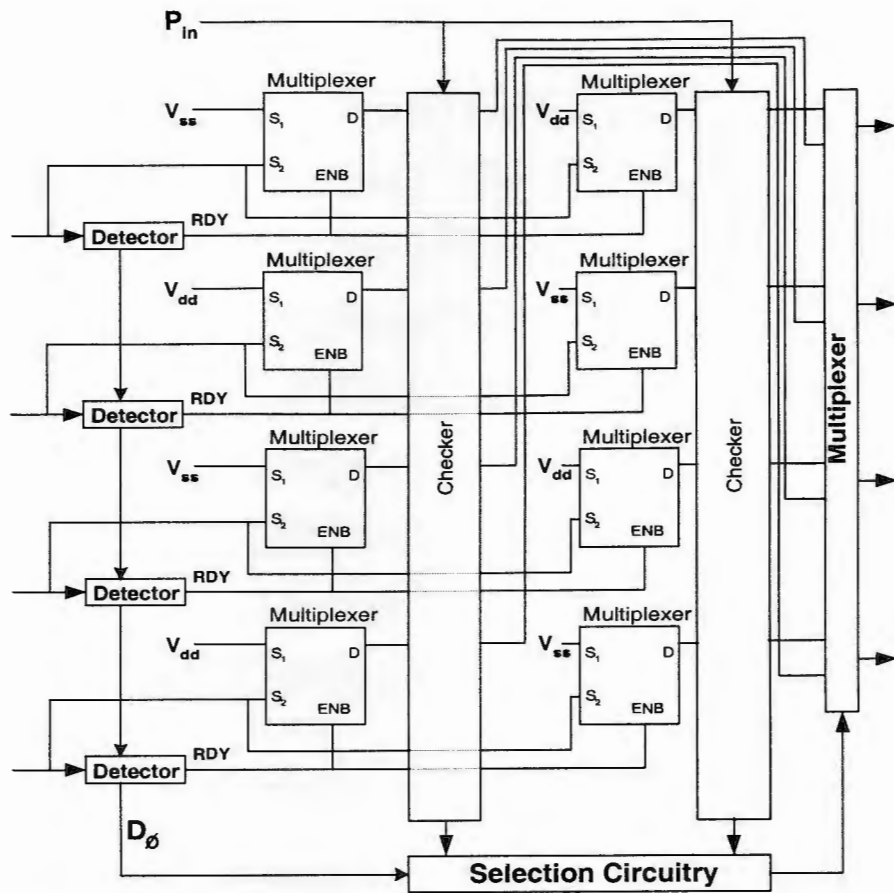


Figure 6.12: 9-bit Implemented Correction System (4 bits shown)

6.9.2 Functional unit topology

For clarity, we present the design of the implemented circuit organized by function.

Detectors and input multiplexers

The design of the detector is straightforward along the lines described fully in Chapter 3. One output, RDY, is used as a selection signal for the two input multiplexers for each input bit.

When RDY is high, both multiplexers pass the original (valid) input bit through to the pair of checkers. When RDY is low, indicating a ϕ input level, one multiplexer

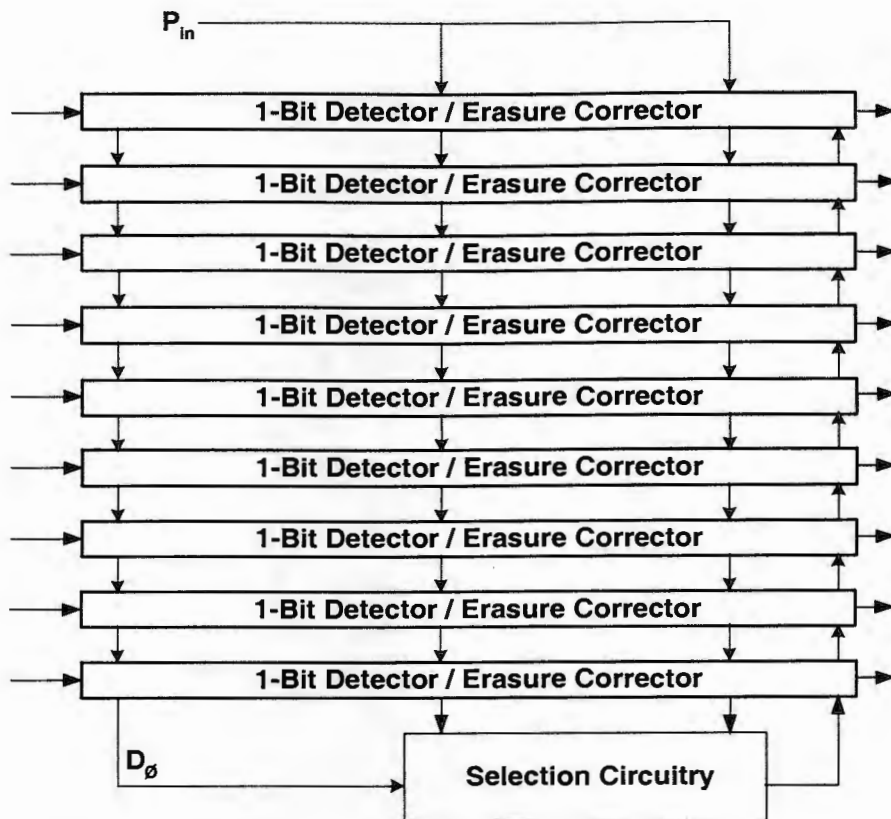


Figure 6.13: Implemented Correction System (Bit-Slice View)

sends a 0 to its checker in place of the original input bit value, and the other sends a 1 to its checker.

Note that the V_{ss} and V_{dd} inputs alternate multiplexers for successive bits, as in Figure 6.11. This is simply because this part of the circuit was designed to be adaptable to the technique covered in Section 6.7.5 with the substitution of distance-3 checkers for the distance-2 checker implemented. As the assignment of bits in the implementation's scheme is arbitrary, it has no effect on the ability of this circuit to correct 1-bit erasures.

Parity checkers

The checkers implemented in this circuit are straightforward, implementing a bit-by-bit exclusive or. The output at the “bottom” of each checker is 0 if a parity error is detected, and 1 if the parity check passes.

Selection circuitry and multiplexer

The selection circuitry is shown in Figure 6.14.

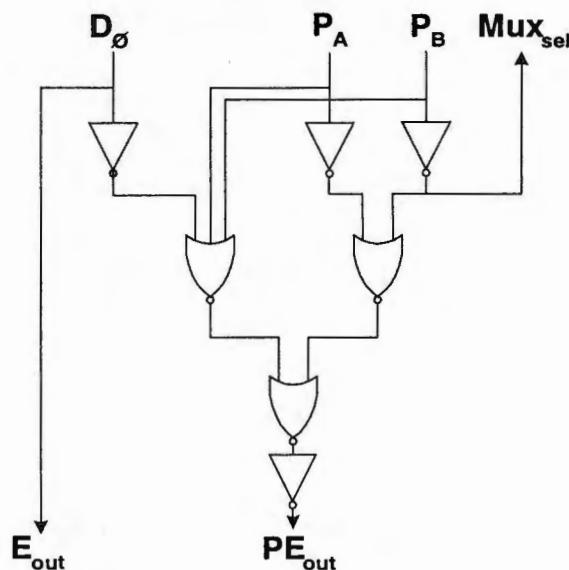


Figure 6.14: Selection Circuit for 9-bit Parity-Based Corrector

Inputs consist of a “parity error” indicator (0 = no error, 1 = error) from each of the two checkers and the D_ϕ line indicating that at least one of the inputs was in the ϕ zone (1 = one or more inputs are ϕ , 0 = no inputs are ϕ). The circuitry generates the select signal for the bit-sliced output multiplexer, as well as a Parity Error (PE_{out}) output.

The truth table for PE_{out} is shown in Table 6.2.

Notes that apply to the entries in Table 6.2 are as follows:

1. This is the normal state when there are no erasures or other one-bit errors. It can also occur when there are an even number of non-erasures errors.

$D_\phi(E_{out})$	P_A	P_B	PE_{out}	See Note
0	0	0	0	1
0	0	1	X	2
0	1	0	X	2
0	1	1	1	3
1	0	0	1	4
1	0	1	0	5
1	1	0	0	5
1	1	1	1	4

Table 6.2: Truth Table for PE_{out}

2. These states cannot occur. If there are no erasures, input sets to the two code checkers are identical, so there cannot be different parity results.
3. This state occurs when there is no erasure, but there is a one-bit error (or any odd number of one-bit errors) on the input.
4. These states occur when an erasure is indicated, but the two checkers return identical results. This can happen only in the presence of more than one erasure - technically, an even number of erasures.
5. These states occur when there is an erasure that has been corrected. It can also occur when there are an odd number of errors, at least one of which is an erasure.

6.9.3 Testing results

Testing results for this circuit are shown in Tables 6.3 through 6.6. Table 6.3 shows results when all inputs are in the valid binary ranges and Parity is set to "Even", Table 6.4 shows results when all inputs are in the valid binary ranges and Parity is set to "Odd", Table 6.5 shows results when one or more inputs is in the ϕ zone and Parity is set to "Even", and Table 6.6 shows results when one or more inputs is in the ϕ zone and Parity is set to "Odd".

The results show that the circuit performs as intended.

Input	PE	ϕ	Output	Input	PE	ϕ	Output
00000000	0	0	00000000	11111110	0	0	11111110
00000001	1	0	00000001	11111100	1	0	11111100
00000011	0	0	00000011	11111000	0	0	11111000
00000111	1	0	00000111	11110000	1	0	11110000
00001111	0	0	00001111	11100000	0	0	11100000
00011111	1	0	00011111	11000000	1	0	11000000
00111111	0	0	00111111	10101010	1	0	10101010
01111111	1	0	01111111	01010101	0	0	01010101
11111111	0	0	11111111	11111111	1	0	11111111

Table 6.3: All Inputs in Valid Ranges and Parity = "Even"

Input	PE	ϕ	Output	Input	PE	ϕ	Output
00000000	1	0	00000000	11111110	1	0	11111110
00000001	0	0	00000001	11111100	0	0	11111100
00000011	1	0	00000011	11111000	1	0	11111000
00000111	0	0	00000111	11110000	0	0	11110000
00001111	1	0	00001111	11100000	1	0	11100000
00011111	0	0	00011111	11000000	0	0	11000000
00111111	1	0	00111111	10101010	0	0	10101010
01111111	0	0	01111111	01010101	1	0	01010101
11111111	1	0	11111111	11111111	0	0	11111111

Table 6.4: All Inputs in Valid Ranges and Parity = "Odd"

6.10 Summary

We have briefly reviewed channel models and their associated errors, as well as some basic theoretical concepts in error-control coding, such as *distance*. We then proceeded to adapt our uncertainty detector to serve the purpose of error location. This allowed us to use strategies described in the literature to boost the correction capabilities of error-control coding schemes.

We also considered the possibilities for partial implementation of these principles, and found them dependent for their efficacy on asymmetry in the error distribution, or on a restricted set of possible error patterns, both of which are realistic possibilities in specific implementations.

We compared the performance of a parity-based correction circuit to classic parity-based error detection. The proposed circuit allowed error location (and therefore correction) in cases where there was one erasure ($0 \Rightarrow \phi$ or $1 \Rightarrow \phi$) and no full errors ($0 \Rightarrow 1$ or $1 \Rightarrow 0$) in the received codeword. Use of the circuit was not without its disadvantages, however; when there were both erasures and full errors in the same codeword, error detection was reduced in some cases. As always, the designer of the specific implementation must take channel error characteristics into account, including the probabilities of various types of single and compound errors, in deciding which scheme to use.

Finally, we depicted the design of a 9-bit, parity-based error correction circuit fabricated on the proof-of-concept circuit. We described the bit-sliced design of this experimental circuit, and presented the testing results showing that the circuit performs as intended.

Input	PE	ϕ	Output	Input	PE	ϕ	Output
00000000 ϕ	0	1	000000000	ϕ 101 ϕ 1010	1	1	010101010
00000000 ϕ 1	0	1	000000011	ϕ 1010 ϕ 010	1	1	010101010
00000001 ϕ 1	0	1	000000101	ϕ 10101 ϕ 10	1	1	010101010
000000 $\phi\phi$ 1	1	1	000000011	ϕ 101010 ϕ 0	1	1	010101010
00000 $\phi\phi\phi$ 1	0	1	000000101	ϕ 1010101 ϕ	1	1	010101010
0000 $\phi\phi\phi\phi$ 1	1	1	000010101	ϕ 01010101	0	1	001010101
000 $\phi\phi\phi\phi\phi$ 1	0	1	000101011	1 ϕ 1010101	0	1	111010101
00 $\phi\phi\phi\phi\phi\phi$ 1	1	1	000101011	10 ϕ 010101	0	1	100010101
0 $\phi\phi\phi\phi\phi\phi\phi$ 1	0	1	001010101	101 ϕ 10101	0	1	101110101
$\phi\phi\phi\phi\phi\phi\phi\phi$ 1	1	1	101010101	1010 ϕ 0101	0	1	101000101
$\phi\phi\phi\phi\phi\phi\phi\phi\phi$	0	1	010101010	10101 ϕ 101	0	1	101011101
0 ϕ 0101010	0	1	010101010	101010 ϕ 01	0	1	101010001
01 ϕ 101010	0	1	010101010	1010101 ϕ 1	0	1	101010111
010 ϕ 01010	0	1	010101010	10101010 ϕ	0	1	101010100
0101 ϕ 1010	0	1	010101010	$\phi\phi$ 1010101	1	1	101010101
01010 ϕ 010	0	1	010101010	1 $\phi\phi$ 010101	1	1	101010101
010101 ϕ 10	0	1	010101010	10 $\phi\phi$ 10101	1	1	101010101
0101010 ϕ 0	0	1	010101010	101 $\phi\phi$ 0101	1	1	101010101
01010101 ϕ	0	1	010101010	1010 $\phi\phi$ 101	1	1	101010101
$\phi\phi$ 0101010	1	1	010101010	10101 $\phi\phi$ 01	1	1	101010101
0 $\phi\phi$ 101010	1	1	010101010	101010 $\phi\phi$ 1	1	1	101010101
01 $\phi\phi$ 01010	1	1	010101010	1010101 $\phi\phi$	1	1	101010101
010 $\phi\phi$ 1010	1	1	010101010	ϕ 0 ϕ 010101	1	1	101010101
0101 $\phi\phi$ 010	1	1	010101010	ϕ 01 ϕ 10101	1	1	101010101
01010 $\phi\phi$ 10	1	1	010101010	ϕ 010 ϕ 0101	1	1	101010101
010101 $\phi\phi$ 0	1	1	010101010	ϕ 0101 ϕ 101	1	1	101010101
0101010 $\phi\phi$	1	1	010101010	ϕ 01010 ϕ 01	1	1	101010101
ϕ 1 ϕ 101010	1	1	010101010	ϕ 010101 ϕ 1	1	1	101010101
ϕ 10 ϕ 01010	1	1	010101010	ϕ 0101010 ϕ	1	1	101010101

Table 6.5: Some Inputs in ϕ Range and Parity = "Even"

Input	PE	ϕ	Output	Input	PE	ϕ	Output
00000000 ϕ	0	1	000000001	ϕ 101 ϕ 1010	1	1	110111010
0000000 ϕ 1	0	1	000000001	ϕ 1010 ϕ 010	1	1	110100010
0000001 ϕ 1	0	1	000000111	ϕ 10101 ϕ 10	1	1	110101110
000000 $\phi\phi$ 1	1	1	000000101	ϕ 101010 ϕ 0	1	1	110101000
00000 $\phi\phi\phi$ 1	0	1	000001011	ϕ 1010101 ϕ	1	1	110101011
0000 $\phi\phi\phi\phi$ 1	1	1	000001011	ϕ 01010101	0	1	101010101
000 $\phi\phi\phi\phi\phi$ 1	0	1	000010101	1 ϕ 1010101	0	1	101010101
00 $\phi\phi\phi\phi\phi\phi$ 1	1	1	001010101	10 ϕ 010101	0	1	101010101
0 $\phi\phi\phi\phi\phi\phi\phi$ 1	0	1	010101011	101 ϕ 10101	0	1	101010101
$\phi\phi\phi\phi\phi\phi\phi\phi$ 1	1	1	010101011	1010 ϕ 0101	0	1	101010101
$\phi\phi\phi\phi\phi\phi\phi\phi\phi$	0	1	101010101	10101 ϕ 101	0	1	101010101
0 ϕ 0101010	0	1	000101010	101010 ϕ 01	0	1	101010101
01 ϕ 101010	0	1	011101010	1010101 ϕ 1	0	1	101010101
010 ϕ 01010	0	1	010001010	10101010 ϕ	0	1	101010101
0101 ϕ 1010	0	1	010111010	$\phi\phi$ 1010101	1	1	011010101
01010 ϕ 010	0	1	010100010	1 $\phi\phi$ 010101	1	1	110010101
010101 ϕ 10	0	1	010101110	10 $\phi\phi$ 10101	1	1	100110101
0101010 ϕ 0	0	1	010101000	101 $\phi\phi$ 0101	1	1	101100101
01010101 ϕ	0	1	010101011	1010 $\phi\phi$ 101	1	1	101001101
$\phi\phi$ 0101010	1	1	100101010	10101 $\phi\phi$ 01	1	1	101011001
0 $\phi\phi$ 101010	1	1	001101010	101010 $\phi\phi$ 1	1	1	101010011
01 $\phi\phi$ 01010	1	1	011001010	1010101 $\phi\phi$	1	1	101010110
010 $\phi\phi$ 1010	1	1	010011010	ϕ 0 ϕ 010101	1	1	000010101
0101 $\phi\phi$ 010	1	1	010110010	ϕ 01 ϕ 10101	1	1	001110101
01010 $\phi\phi$ 10	1	1	010100110	ϕ 010 ϕ 0101	1	1	001000101
010101 $\phi\phi$ 0	1	1	010101100	ϕ 0101 ϕ 101	1	1	001011101
0101010 $\phi\phi$	1	1	010101001	ϕ 01010 ϕ 01	1	1	001010001
ϕ 1 ϕ 101010	1	1	111101010	ϕ 010101 ϕ 1	1	1	001010111
ϕ 10 ϕ 01010	1	1	110001010	ϕ 0101010 ϕ	1	1	001010100

Table 6.6: Some Inputs in ϕ Range and Parity = "Odd"

Chapter 7

Summary and conclusions

The major contribution of this research is the consideration of unknown logic level values as *information*. Much of digital logic design views logic as an abstraction, a dichotomy of zero and one. Although it is well acknowledged in VLSI texts that digital logic circuitry is analog in its ultimate nature, efforts are made to make the reality fit, insofar as possible, the abstraction.

In this work, we developed a design for a detector for unknown logic values that does not depend on the existence of reference voltages. While no implication is made that this is the most efficient detector in any regard, it does provide the required *information* necessary to demonstrate the validity of the concepts covered in this thesis.

Several uses were described for this information, some of them rudimentary but potentially of practical application. We focussed, however, on two specific application areas to illustrate and demonstrate the contribution of this research.

Clock skew, as a result of increasing circuit speeds and concurrently increasing die size, is a serious problem for the future of processor design. Power consumption by advanced processors is also of increasing concern, especially with the proliferation of laptop systems and other portable computing devices. Asynchronous system concepts, especially the GALS (Globally Asynchronous Locally Synchronous) constructs, are well suited to address both of these problems. As logic stages are independently, locally clocked, the need for a global clock is reduced or eliminated. Power usage

can be greatly reduced without impacting performance, since a local stage without work to do undergoes no state transitions, so uses no power.

A logic family, Binary Plus logic, and its dynamic version, Centered Binary Plus logic, was developed to fulfill the completion recognition and self-clocking requirements of GALS systems. The design technique for a Binary Plus gate was developed and proven valid, and Binary Plus gates and combinational multiple-gate logic blocks were shown to be free from race conditions. Binary Plus gates recognize an undefined value on the input, and do not display a valid output until there is a necessary and sufficient condition on the inputs to justify it. This provides clear completion recognition, and also allows the logic stage to take advantage of low-delay input sets. The method has significant advantages over other currently used completion-detection techniques in asynchronous design.

To demonstrate the use of these concepts in asynchronous system design, we designed and fabricated a proof-of-concept circuit containing a 4-bit ripple-carry adder, implemented as a Centered Binary Logic stage. Tests on this circuit showed the anticipated effects of input-dependent variations in completion time; a correlation between measured completion time and the performance predicted by a gate-level simulator constructed for the circuit was positive and showed very high statistical significance.

In communications applications, error-control coding techniques have long been used to guard against transmission errors, some of which may be transitions to undefined values. These transitions are termed *erasures* in the literature. By knowing the *location* of an error, correcting it is greatly simplified, and an error-detecting/correcting code can be used to correct more errors than would be possible without the knowledge of the location of an error. Detecting an undefined logic level on an input can be used as an erasure location technique, enabling us to use erasure correction methods well documented in the literature. Such erasure-correction methods were previously limited to environments in which the error could be localized in other ways, such as a current spike (due to an α -particle strike) or from multi-dimensional parity checks in memory arrays.

A 9-bit, simple parity-based erasure detector/corrector was implemented on the

proof-of-concept circuit. This system showed itself capable of correcting a one-bit erasure, demonstrating that the knowledge that an input is undefined can be used to boost the detection/correction capability of error-control coding.

7.1 Future work

There are many directions in which further research could be taken to explore the concepts introduced in this work.

More *space-efficient or faster versions of the detector* must be developed. The detector as designed in this work is large; this both takes up space and increases capacitance in the driving circuit, limiting its speed. It is possible that techniques using a V_h supply and non-ratioed inverters might create a faster, more space-efficient detector. As a V_h supply is of use in precharging Centered Binary logic stages, this would simply be an additional use for it.

Issues of *noise margins* for this logic family should be examined. It is clear that in some ways the noise margin is decreased from that of standard binary logic, while in other ways it is increased. For example, it would take less noise to cause a change from a valid binary value to another state (ϕ) as the boundary between either valid value and that state is closer than the boundary between 0 and 1 in pure binary logic. On the other hand, it would take *more* noise to cause a change from a valid binary value to the *opposite binary value*, as that boundary has been pushed farther away. In short, the chance of transitioning to a detectable error is greater, while the chance of transitioning to a non-detectable error is less. In the event that it was desired to transmit a ϕ from one location to another - and have it arrive as a ϕ - noise could be a serious consideration, for reasons that were covered in Chapter 6.

For asynchronous designs, *Binary Plus compatible input sources*, such as pipeline stage source latches, should be developed and tested. Techniques for widening the pipeline should be explored, including expansion of the source latch concept into a stage router.

A major application area not addressed in this work is the use of the concepts we have developed in the area of *circuit testing*. Adaptation of Boundary Scan

techniques to the detection of unknown values would be a significant topic by itself.

References

- [1] P. Weiss, "Quantum internet," *Science News*, vol. 155, pp. 220–221, April 3 1999.
- [2] D. Matzke, "Will physical scalability sabotage performance gains?," *Computer*, vol. 30, pp. 37–39, September 1997.
- [3] S. Hamilton, "Semiconductor research corporation: Taking Moore's law into the next century," *Computer*, vol. 32, pp. 43–48, January 1999.
- [4] M. Schlett, "Trends in embedded-microprocessor design," *Computer*, vol. 31, pp. 44–49, August 1998.
- [5] S. J. Jou and I. Y. Chung, "Low-power self-timed circuit-design technique," *Electronics Letters*, vol. 33, pp. 110–111, January 16 1997.
- [6] T. R. N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1989.
- [7] D. C. Bossen and M. Y. Hsiao, "A system solution to the memory soft error problem," *IBM Journal of Research and Development*, vol. 24, pp. 390–397, May 1980.
- [8] O. Keren and S. Litsyn, "A class of array codes correcting multiple column erasures," *IEEE Transactions on Information Theory*, vol. 43, pp. 1843–1851, November 1997.

- [9] T. Calin, F. Vargas, M. Nicolaidis, and R. Velazco, "A low-cost, highly reliable SEU-tolerant SRAM - prototype and test-results," *IEEE Transactions on Nuclear Science*, vol. 42, no. 6, pp. 1592–1598, 1995.
- [10] F. Vargas and M. Nicolaidis, "SEU-tolerant SRAM design based on current monitoring," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pp. 105–115, 1994.
- [11] F. J. Mowle, *A Systematic Approach to Digital Logic Design*. Reading, MA: Addison-Wesley Publishing Company, 1976.
- [12] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill Book Company, 1978.
- [13] E. McCluskey, *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [14] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*. Reading, MA: Addison-Wesley Publishing Company, 1988.
- [15] M. Favalli, P. Olivo, M. Damiani, and B. Ricco, "Novel design for testability schemes for CMOS IC's," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1239–1246, October 1990.
- [16] T. Juhnke and H. Klar, "Calculation of the soft error rate of submicron CMOS logic circuits," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 830–834, July 1995.
- [17] C. Henderson, J. Soden, and C. Hawkins, "The behavior and testing implications of CMOS IC logic gate open circuits," in *Proceedings of the 1991 International Test Conference*, pp. 302–310, 1991.
- [18] H. Hao and E. McCluskey, "'Resistive shorts' within CMOS gates," in *Proceedings of the 1991 International Test Conference*, pp. 292–301, 1991.

- [19] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Incorporated, 1990.
- [20] W. K. C. Lam and R. Brayton, *Timed Boolean Functions - A Unified Formalism for Exact Timing Analysis*. Boston: Kluwer Academic Publishers, 1994.
- [21] H. S. Stone, *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley Publishing Company, 1990.
- [22] E. Suhir, "Flip-chip solder joint interconnections and encapsulants in silicon-on-silicon MCM technology: Thermally induced stresses and mechanical reliability," in *Proceedings of the 1993 IEEE Multi-Chip Module Conference*, pp. 92–99, 1993.
- [23] M. Jacamet and W. Guggenbuhl, "Layout-dependent fault analysis and test synthesis for CMOS circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 888–899, June 1993.
- [24] B. Bennetts and C. Maunder, *Notes for Tutorial 6: Boundary Scan and Other 1149.x Standards*. Test Technology Committee of the IEEE Computer Society, Washington, DC, October 18 1998.
- [25] J. Lien and M. Breuer, "Maximal diagnosis for wiring networks," in *Proceedings of the 1991 International Test Conference*, pp. 96–105, 1991.
- [26] T. Storey, W. Maly, J. Andrews, and M. Miske, "Stuck fault and current testing comparison using CMOS chip test," in *Proceedings of the 1991 International Test Conference*, pp. 311–318, 1991.
- [27] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin, "Two fault injection techniques for test of fault handling mechanisms," in *Proceedings of the 1991 International Test Conference*, pp. 140–149, 1991.
- [28] M. Marzouki, J. Laurent, and B. Courtois, "Coupling electron-beam probing with knowledge-based fault localization," in *Proceedings of the 1991 International Test Conference*, pp. 238–247, 1991.

- [29] J. Salinas, Y. Shen, and F. Lombardi, "A sweeping line approach to interconnect testing," *IEEE Transactions on Computers*, vol. 45, pp. 917–929, August 1996.
- [30] H. Wu, N. Zhuang, and M. Perkowski, "Novel CMOS scan design for VLSI testability," in *Proceedings of the 23rd International Symposium on Multiple-Valued Logic*, pp. 82–86, 1993.
- [31] C. Hwang, M. Ismail, and J. DeGroat, "On-chip 1_{DDQ} testability schemes for detecting multiple faults in CMOS IC's," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 732–739, May 1996.
- [32] M. Soma, "An experimental approach to analog fault models," in *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference*, May 1991.
- [33] C. Metra, M. Favalli, M. Olivo, and B. Ricco, "On-line detection of bridging and delay faults in functional blocks of CMOS self-checking circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 770–776, July 1997.
- [34] Y. Crouzet and C. Landrault, "Design specification of a self-checking detection processor," in *Proceedings of the 10th Symposium on Fault-Tolerant Computing*, pp. 2775–2777, 1980.
- [35] N. Gaitanis, "A totally self-checking error indicator," *IEEE Transactions on Computers*, vol. C-34, pp. 758–761, August 1985.
- [36] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, vol. C-31, pp. 589–595, July 1982.
- [37] S. S. Appleton, S. V. Morton, and M. J. Liebelt, "Technique for high speed asynchronous pipeline control," *Electronics Letters*, vol. 32, pp. 1973–1974, October 10 1996.

- [38] E. Grass and S. Jones, "Activity-monitoring completion-detection (AMCD): A new approach to achieve self-timing," *Electronics Letters*, vol. 32, pp. 86–88, January 18 1996.
- [39] M. Renaudin, B. Elhassan, and A. Guyot, "A new asynchronous pipeline scheme - applications to the design of a self-timed ring divider," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1001–1013, July 1996.
- [40] A. Ishii, C. Leiserson, and M. Papaefthymiou, "Optimizing 2-phase, level-clocked circuitry," *Journal of the ACM*, vol. 44, pp. 148–199, January 1997.

Bibliography

Abramovici, M., Breuer, M., and Friedman, A., *Digital Systems Testing and Testable Design*. New York: IEEE Press, 1990.

Appleton, S. S., Morton, S. V., and Liebelt, M. J., "Technique for high speed asynchronous pipeline control," *Electronics Letters*, vol. 32, pp. 1973–1974, October 10 1996.

Belabbes, N., Guterman, A. J., Savaria, Y., and Dagenais, M., "Ratioed voter circuit for testing and fault tolerance in VLSI processing arrays," *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 43, pp. 143–151, February 1996.

Bennetts, B. and Maunder, C., *Notes for Tutorial 6: Boundary Scan and Other 1149.x Standards*. Test Technology Committee of the IEEE Computer Society, Washington, DC, October 18 1998.

Bossen, D. C. and Hsiao, M. Y., "A system solution to the memory soft error problem," *IBM Journal of Research and Development*, vol. 24, pp. 390–397, May 1980.

Calin, T., Vargas, F., Nicolaidis, M., and Velazco, R., "A low-cost, highly reliable SEU-tolerant SRAM - prototype and test-results," *IEEE Transactions on Nuclear Science*, vol. 42, pp. 1592–1598, 1995.

Crouzet, Y. and Landrault, C., "Design specification of a self-checking detection processor," in *Proceedings of the 10th Symposium on Fault-Tolerant Computing*, pp. 2775–2777, 1980.

Downie, N. M. and Heath, R. W., *Basic Statistical Methods*. New York: Harper and Row, Publishers, 1965.

- Escriba, J. and Carrasco, J. A., "Self-timed Manchester chain carry propagate adder," *Electronics Letters*, vol. 32, pp. 708–710, April 11 1996.
- Favalli, M., Olivo, P., Damiani, M., and Ricco, B., "Novel design for testability schemes for CMOS IC's," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1239–1246, October 1990.
- Foster, C., *Computer Architecture*. New York: Van Nostrand Reinhold Company, 1976.
- Gaitanis, N., "A totally self-checking error indicator," *IEEE Transactions on Computers*, vol. C-34, pp. 758–761, August 1985.
- Goldsmith, A. and Varaiya, P., "Capacity of fading channels with channel side information," *IEEE Transactions on Information Theory*, vol. 43, pp. 1986–1992, November 1997.
- Grass, E. and Jones, S., "Activity-monitoring completion-detection (AMCD): A new approach to achieve self-timing," *Electronics Letters*, vol. 32, pp. 86–88, January 18 1996.
- Hamilton, S., "Semiconductor research corporation: Taking Moore's law into the next century," *Computer*, vol. 32, pp. 43–48, January 1999.
- Hao, H. and McCluskey, E., "'Resistive shorts' within CMOS gates," in *Proceedings of the 1991 International Test Conference*, pp. 292–301, 1991.
- Hashimoto, T. and Taguchi, M., "Performance of explicit error-detection and threshold decision in decoding with erasures," *IEEE Transactions on Information Theory*, vol. 43, pp. 1650–1655, September 1997.
- Henderson, C., Soden, J., and Hawkins, C., "The behavior and testing implications of CMOS IC logic gate open circuits," in *Proceedings of the 1991 International Test Conference*, pp. 302–310, 1991.
- Hennessy, J. and Patterson, D., *Computer Architecture - A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Incorporated, 1990.

Hwang, C., Ismail, M., and DeGroat, J., "On-chip ν_{DDQ} testability schemes for detecting multiple faults in CMOS IC's," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 732–739, May 1996.

Ishii, A., C. Leiserson, and Papaefthymiou, M., "Optimizing 2-phase, level-clocked circuitry," *Journal of the ACM*, vol. 44, pp. 148–199, January 1997.

Itoh, K., Sasaki, K., and Nakagome, Y., "Trends in low-power RAM circuit technologies," *Proceedings of the IEEE*, vol. 83, pp. 524–543, April 1995.

Jacamet, M. and Guggenbuhl, W., "Layout-dependent fault analysis and test synthesis for CMOS circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 888–899, June 1993.

Jou, S. J. and Chung, I. Y., "Low-power self-timed circuit-design technique," *Electronics Letters*, vol. 33, pp. 110–111, January 16 1997.

Juhnke, T. and Klar, H., "Calculation of the soft error rate of submicron CMOS logic circuits," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 830–834, July 1995.

Karlsson, J., Gunneflo, U., Liden, P., and Torin, J., "Two fault injection techniques for test of fault handling mechanisms," in *Proceedings of the 1991 International Test Conference*, pp. 140–149, 1991.

Keren, O. and Litsyn, S., "A class of array codes correcting multiple column erasures," *IEEE Transactions on Information Theory*, vol. 43, pp. 1843–1851, November 1997.

Kohavi, Z., *Switching and Finite Automata Theory*. New York: McGraw-Hill Book Company, 1978.

Lam, W. K. C. and Brayton, R., *Timed Boolean Functions - A Unified Formalism for Exact Timing Analysis*. Boston: Kluwer Academic Publishers, 1994.

Lien, J. and Breuer, M., "Maximal diagnosis for wiring networks," in *Proceedings of the 1991 International Test Conference*, pp. 96–105, 1991.

Mano, M., *Computer System Architecture*. Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1982.

Marzouki, M., Laurent, J., and Courtois, B., "Coupling electron-beam probing with knowledge-based fault localization," in *Proceedings of the 1991 International Test Conference*, pp. 238–247, 1991.

Matsuzawa, K. and Fujiwara, E., "Masking asymmetric line faults using semi-distance codes," *The Transactions of the IEICE*, vol. E73, pp. 1278–1286, August 1990.

Matzke, D., "Will physical scalability sabotage performance gains?," *Computer*, vol. 30, pp. 37–39, September 1997.

McCluskey, E., *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Mendenhall, W. and Scheaffer, R., *Mathematical Statistics with Applications*. North Scituate, MA: Duxbury Press, 1973.

Metra, C., Favalli, M., Olivo, M., and Ricco, B., "On-line detection of bridging and delay faults in functional blocks of CMOS self-checking circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 770–776, July 1997.

Mowle, F. J., *A Systematic Approach to Digital Logic Design*. Reading, MA: Addison-Wesley Publishing Company, 1976.

Patel, J. H. and Fung, L. Y., "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, vol. C-31, pp. 589–595, July 1982.

Rao, T. R. N. and Fujiwara, E., *Error-Control Coding for Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1989.

Renaudin, M., Elhassan, B., and Guyot, A., "A new asynchronous pipeline scheme - applications to the design of a self-timed ring divider," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1001–1013, July 1996.

Salinas, J., Shen, Y., and Lombardi, F., "A sweeping line approach to interconnect testing," *IEEE Transactions on Computers*, vol. 45, pp. 917–929, August 1996.

- Schlett, M., "Trends in embedded-microprocessor design," *Computer*, vol. 31, pp. 44-49, August 1998.
- Sit, V. W. Y., Choy, C. S., and Chan, C. F., "Use of current sensing technique in designing asynchronous static RAM for self-timed systems," *Electronics Letters*, vol. 33, pp. 667-668, April 10 1997.
- Soma, M., "An experimental approach to analog fault models," in *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference*, May 1991.
- Sternheim, E., Singh, R., and Trivedi, Y., *Digital Design with Verilog HDL*. Cupertino, CA: Automata Publishing Company, 1990.
- Stone, H. S., *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley Publishing Company, 1990.
- Storey, T., Maly, W., Andrews, J., and Miske, M., "Stuck fault and current testing comparison using CMOS chip test," in *Proceedings of the 1991 International Test Conference*, pp. 311-318, 1991.
- Suhir, E., "Flip-chip solder joint interconnections and encapsulants in silicon-on-silicon MCM technology: Thermally induced stresses and mechanical reliability," in *Proceedings of the 1993 IEEE Multi-Chip Module Conference*, pp. 92-99, 1993.
- Vargas, F. and Nicolaidis, M., "SEU-tolerant SRAM design based on current monitoring," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pp. 105-115, 1994.
- Weiss, P., "Quantum internet," *Science News*, vol. 155, pp. 220-221, April 3 1999.
- Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design - A Systems Perspective*. Reading, MA: Addison-Wesley Publishing Company, 1988.
- Wu, H., Zhuang, N., and Perkowski, M., "Novel CMOS scan design for VLSI testability," in *Proceedings of the 23rd International Symposium on Multiple-Valued Logic*, pp. 82-86, 1993.